School of Computer Science and Software Engineering

CITS4009 Introduction to Data Science

SEMESTER 2, 2017: PART 2 MODELLING METHODS

CHAPTER 6 MEMORIZATION METHODS

Chapter Objectives

- Building single-variable models
- Cross-validated variable selection
- Building basic multivariable models.
- Starting with decision trees, nearest neighbor, and naive Bayes models.

Introduction

- *memorization methods* are methods that generate answers by returning a majority category (in the case of classification) or average value (in the case of scoring) of a subset of the original training data.
- These methods can vary from models depending on a single variable (similar to the analyst's pivot table), to decision trees (similar to what are called *business rules*), to nearest neighbour and Naive Bayes methods.
- In this chapter, you'll learn how to use these memorization methods to solve classification problems (though the same techniques also work for scoring problems).

KDD and KDD Cup 2009

- Every year KDD hosts a data mining cup, where teams analyze a dataset and then are ranked against each other. The KDD Cup is a huge deal and the inspiration for the famous Netflix Prize and even Kaggle competitions.
- The KDD Cup 2009 provided a dataset about customer relationship management.
- The contest supplied 230 facts about 50,000 credit card accounts.
- From these features, the goal was to predict account cancellation (called *churn*), the innate tendency to use new products and services (called *appetency*), and willingness to respond favorably to marketing pitches (called *upselling*)
- We have the advantage that the data is already in a ready-to-model format (all input variables and the results arranged in single rows). But we don't know the meaning of any variable (so we can't merge in outside data sources), and we can't use any method that treats time and repetition of events carefully (such as time series methods or survival analysis).
- To simulate the data science processes, we'll assume that we can use any column we're given to make predictions (that all of these columns are known prior to needing a prediction[3]), the

The worst possible modelling outcome

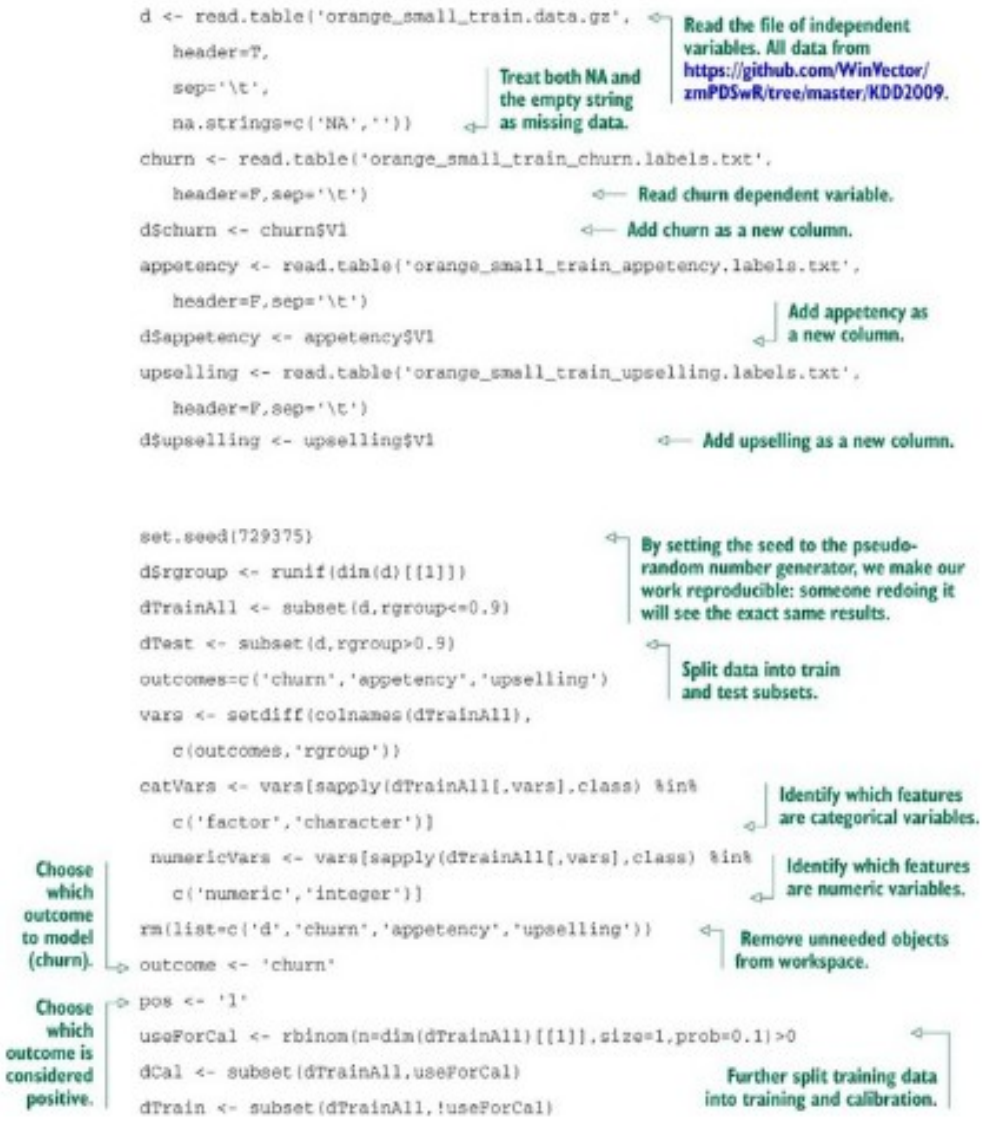
- The worst possible modelling outcome is not failing to find a good model.
- The worst possible modelling outcome is thinking you have a good model when you don’t.
- One of the easiest ways to accidentally build such a deficient model is to have an instrumental or independent variable that is in fact a subtle function of the outcome.
- The point is this: such variables won’t actually be available in a real deployment and often are in training data packaged up by others.

Getting started with KDD Cup 2009 data

- Try to predict churn in the KDD dataset.
- The KDD contest was judged in terms of AUC (*area under the curve*, a measure of prediction quality discussed in chapter 5), so we'll also use AUC as our measure of performance.
- The winning team achieved an AUC of 0.76 on churn, so we'll treat that as our **upper bound** on possible performance. Our **lower bound** on performance is an AUC of 0.5, as this is the performance of a useless model.
- This problem has a large number of variables, many of which have a large number of possible levels.
- We're also using the AUC measure, which isn't particularly resistant to overfitting (not having built-in model complexity or chance corrections).
- Because of this concern, we'll split our data into three sets**: training, calibration, and test**.
- The intent of the three-way split is this:
  - we'll use the training set for most of our work,
  - and we'll never look at the test set (we'll reserve it for our final report of model performance).

Getting started- C

- Steps like calibration and cross-validation estimates be performed by repeatedly splitting the training portion of the data (allowing for more efficient estimation than a single split, and keeping the test data completely out of the modelling effort).
- For simplicity in this example, we'll split the training portion of our data into training and calibration only a single time.
- It's often an excellent idea to first work on a small subset of your training data, so that it takes seconds to debug your code instead of minutes.



Preparing the KDD data for analysis

Building single-variable models

- Single-variable models are simply models built using only one variable at a time.
- Single-variable models can be powerful tools, so it's worth learning how to work well with them before jumping into general modelling (which almost always means multiple variable models).
- You can build single-variable models from both categorical and numeric variables.

Using categorical features

- A single-variable model based on categorical features is easiest to describe as a table.
- a *pivot table* (which promotes values or levels of a feature to be families of new columns) and statisticians use what's called a *contingency table* (where each possibility is given a column name).
- **Churn rates grouped by variable 218 codes:**

```
>
print(table218[,2]/(table218[,1]+table218

      cJvF       UYBR       <NA>
0.05994389 0.08223821 0.26523297
```

```
table218 <- table(
   Var218=dTrain[,'Var218'],
   churn=dTrain[,outcome],
   useNA='ifany')
print(table218)
```

Tabulate levels of Var218.

Tabulate levels of churn outcome.

Include NA values in tabulation.

```
        churn
Var218     -1     1
  cJvF  19101  1218
  UYBR  17599  1577
  <NA>    410   148
```

Plotting churn grouped by variable 218 levels

- This summary tells us that when variable 218 takes on a value of cJvF, around 6% of the customers churn; when it's UYBR, 8% of the customers churn; and when it's not recorded (NA), 27% of the customers churn. The utility of any variable level is a combination of how often the level occurs (rare levels aren't very useful) and how extreme the distribution of the outcome is for records matching a given level.

- Variable 218 seems like a feature that's easy to use and helpful with prediction. In real work, we'd want to research with our business partners why it has missing values and what's the best thing to do when values are missing (this will depend on how the data was prepared).
- We also need to design a strategy for what to do if a new level not seen during training were to occur during model use.
- Since this is a contest problem with no available project partners, we'll build a function that converts NA to a level (as it seems to be pretty informative) and also treats novel values as uninformative. Our function to convert a categorical variable into a single model prediction is shown here:

```
mkPredC <- function(outCol,varCol,appCol) {
  pPos <- sum(outCol==pos)/length(outCol)
  naTab <- table(as.factor(outCol[is.na(varCol)]))
  pPosNna <- (naTab/sum(naTab))[pos]
  vTab <- table(as.factor(outCol),varCol)
  pPosWv <- (vTab[pos,]+1.0e-3*pPos)/(colSums(vTab)+1.0e-3)
  pred <- pPosWv[appCol]
  pred[is.na(appCol)] <- pPosNna
  pred[is.na(pred)] <- pPos
  pred
}
```

Given a vector of training outcomes (outCol), a categorical training variable (varCol), and a prediction variable (appCol), use outCol and varCol to build a single-variable model and then apply the model to appCol to get new predictions.

Get stats on how often outcome is positive during training.

Get stats on how often outcome is positive for NA values of variable during training.

Get stats on how often outcome is positive, conditioned on levels of training variable.

Make predictions by looking up levels of appCol.

Add in predictions for NA levels of appCol.

Add in predictions for levels of appCol that weren't known during training.

Return vector of predictions.

- Placing all of the steps in a function lets us apply the technique to many variables quickly.
- The dataset we're working with has 38 categorical variables, many of which are almost always NA, and many of which have over 10,000 distinct levels.
- So we definitely want to automate working with these variables as we have.
- Our first automated step is to adjoin a prediction or forecast (in this case, the predicted probability of churning) for each categorical variable, as shown here:

```
for(v in catVars) {
pi <- paste('pred',v,sep='')
 dTrain[,pi] <-mkPredC(dTrain[,outcome],dTrain[,v],dTrain[,v])
dCal[,pi] <- mkPredC(dTrain[,outcome],dTrain[,v],dCal[,v])
dTest[,pi] <- mkPredC(dTrain[,outcome],dTrain[,v],dTest[,v])
 }
```

- Note that in all cases we train with the training data frame and then apply to all three data frames dTrain, dCal, and dTest.
- We're using an extra calibration data frame (dCal) because we have so many categorical variables that have a very large number of levels and are subject to overfitting.
- Once we have the predictions, we can find the categorical variables that have a good AUC both on the training data and on the calibration data not used during training.

Scoring categorical variables by AUC:

```
library('ROCR')
> calcAUC <- function(predcol,outcol) {
   perf <- performance(prediction(predcol,outcol==pos),'auc') as.numeric(perf@y.values)
   }
 > for(v in catVars) {
    pi <- paste('pred',v,sep=` ')
   aucTrain <- calcAUC(dTrain[,pi],dTrain[,outcome]) if(aucTrain>=0.8) {
       aucCal <- calcAUC(dCal[,pi],dCal[,outcome])
     print(sprintf("%s, trainAUC: %4.3f calibrationAUC: %4.3f", pi,aucTrain,aucCal))
      }
    }
[1] "predVar200, trainAUC: 0.828 calibrationAUC: 0.527“
[1] "predVar202, trainAUC: 0.829 calibrationAUC: 0.522"
[1] "predVar214, trainAUC: 0.828 calibrationAUC: 0.527"
[1] "predVar217, trainAUC: 0.898 calibrationAUC: 0.553"
```

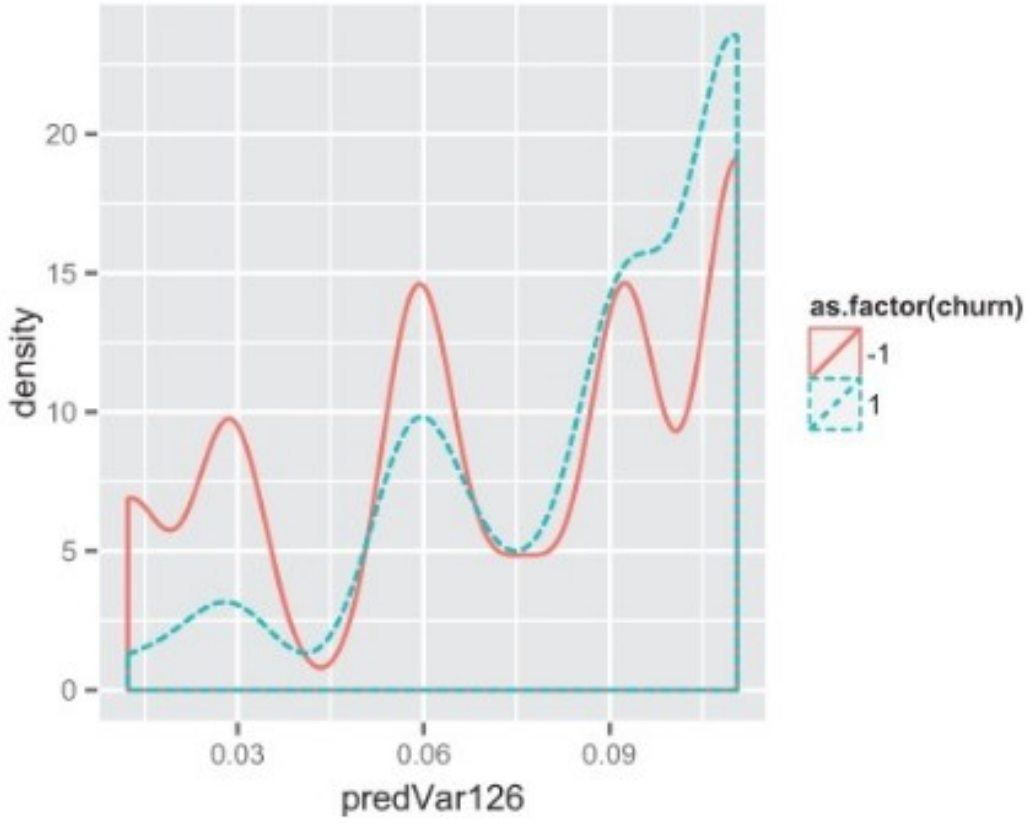
Using numeric features

- There are a number of ways to use a numeric feature to make predictions.
- A common method is to bin the numeric feature into a number of ranges and then use the range labels as a new categorical variable.
- R can do this quickly with its **`quantile()`** and **`cut()`** commands.

```
>mkPredN <- function(outCol,varCol,appCol) {
    cuts <- unique(as.numeric(quantile(varCol, probs=seq(0, 1, 0.1),na.rm=T)))
    varC <- cut(varCol,cuts)
    appC <- cut(appCol,cuts) mkPredC(outCol,varC,appC)
 }
>for(v in numericVars) {
>pi <- paste('pred',v,sep='')
dTrain[,pi] <- mkPredN(dTrain[,outcome],dTrain[,v],dTrain[,v])
dTest[,pi] <- mkPredN(dTrain[,outcome],dTrain[,v],dTest[,v])
dCal[,pi] <- mkPredN(dTrain[,outcome],dTrain[,v],dCal[,v])
aucTrain <- calcAUC(dTrain[,pi],dTrain[,outcome])
    if(aucTrain>=0.55) {
         aucCal <- calcAUC(dCal[,pi],dCal[,outcome])
            print(sprintf("%s, trainAUC: %4.3f calibrationAUC: %4.3f", pi,aucTrain,aucCal))
            }
         }
[1] "predVar6, trainAUC: 0.557 calibrationAUC: 0.554"
[1] "predVar7, trainAUC: 0.555 calibrationAUC: 0.565"
[1] "predVar13, trainAUC: 0.568 calibrationAUC: 0.553"
[1] "predVar73, trainAUC: 0.608 calibrationAUC: 0.616"
[1] "predVar74, trainAUC: 0.574 calibrationAUC: 0.566"
[1] "predVar81, trainAUC: 0.558 calibrationAUC: 0.542"
[1] "predVar113, trainAUC: 0.557 calibrationAUC: 0.567"
[1] "predVar126, trainAUC: 0.635 calibrationAUC: 0.629"
[1] "predVar140, trainAUC: 0.561 calibrationAUC: 0.560"
[1] "predVar189, trainAUC: 0.574 calibrationAUC: 0.599"
```

- Notice in this case the numeric variables behave similarly on the training and calibration data.
- This is because our prediction method converts numeric variables into categorical variables with around 10 well-distributed levels, so our training estimate tends to be good and not overfit.
- We could improve our numeric estimate by interpolating between quantiles.
- Other methods we could've used are kernel-based density estimation and parametric fitting.
- Both of these methods are usually available in the variable treatment steps of Naive Bayes classifiers.

- Good way to visualize the predictive power of a numeric variable is the double density plot, where we plot on the same graph the variable score distribution for positive examples and variable score distribution of negative examples as two groups.
- The figure here shows the performance of the single-variable model built from the numeric feature **`Var126`**.
- It is showing the conditional distribution of **`predVar126`** for churning accounts (the dashed-line density plot) and the distribution of **`predVar126`** for nor churning accounts (the solid-line density plot).
- We can deduce that low values of **`predVar126`** are rare for churning accounts and not as rare for non-churning accounts (the graph is read by comparing areas under the curves).
- This (by Bayes law) lets us in turn say that a low value of predVar126 is good evidence that an account will not churn.

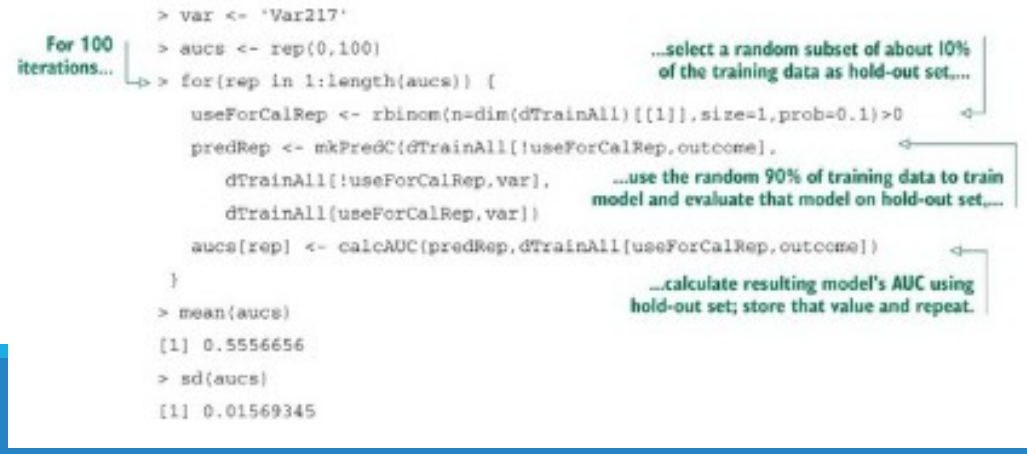


Dealing with missing values in numeric variables

- First, for each numeric variable, introduce a new advisory variable that is 1 when the original variable had a missing value and 0 otherwise.
- Second, replace all missing values of the original variable with 0.
- You now have removed all of the missing values and have recorded enough details so that missing values aren't confused with actual zero values.

Using cross-validation to estimate effects of overfitting

- *cross-validation* used to estimate the degree of overfit we have hidden in our models.
- Cross-validation applies in *all* modelling situations.
- In repeated cross-validation, a subset of the training data is used to build a model, and a complementary subset of the training data is used to score the model



Running a repeated cross-validation experiment

Cross-validation

- In many other circumstances, estimations from a single calibration set are good enough.
- And in extreme cases (such as fitting models with very many variables or level values), you're well advised to use replicated cross-validation estimates of variable utilities and model fits.
- Automatic cross-validation is *extremely* useful in all modelling situations, so it's critical you automate your modelling steps so you can perform cross-validation studies.

Aside: cross-validation in functional notation

- `for(){}` loops are considered an undesirable crutch in R.
- We used a for loop in our cross-validation example, as this is the style of programming that is likely to be most familiar to nonspecialists.
- The point is that for loops over-specify computation (they describe both what you want and the exact order of steps to achieve it).
- For loops tend to be less reusable and less composable than other computational methods.
- When you become proficient in R, you look to eliminate for loops from your code and use either vectorized or functional methods where appropriate.
- Function arguments in R are *not* evaluated prior to being passed in to a function, but instead are evaluated inside the function.
- This is called *promise-based* argument evaluation and is powerful (it allows user-defined macros, lazy evaluation, placement of variable names on plots, user-defined control structures, and user-defined exceptions).
- This can also be complicated, so it's best to think of R as having mostly *call-by-value semantics*, where arguments are passed to functions as values evaluated prior to entering the function and alterations of these values aren't seen outside of the function.

Building models using many variables

- Models that combine the effects of many variables tend to be much more powerful than models that use only a single variable.
- The most fundamental multiple-variable models:
  1. Decision trees,
  2. Nearest neighbour,
  3. Naive Bayes.

Variable selection

- A key part of building many variable models is selecting what variables[6] to use and how the variables are to be transformed or treated.
- *When* variables are available has a huge impact on model utility.
- For instance, a variable that's coincident with (available near or even after) the time that the outcome occurs may make a very accurate model with little utility (as it can't be used for long-range prediction).
- The analyst has to watch out for variables that are functions of or "contaminated by" the value to be predicted
- Sometimes you may want to improve model utility (at a possible cost of accuracy) by removing variables from the project design.
- An acceptable prediction one day before an event can be much more useful than a more accurate prediction one hour before the event.

Using decision trees

- Decision trees are a simple model type: they make a prediction that is piecewise constant.
- This is interesting because the null hypothesis that we're trying to outperform is often a single constant for the whole dataset, so we can view a decision tree as a procedure to split the training data into pieces and use a simple memorized constant on each piece.
- Decision trees (especially a type called *classification and regression trees*, or *CART*) can be used to quickly predict either categorical or numeric outcomes.
- The best way to grasp the concept of decision trees is to think of them as machine-generated business rules.

Fitting a decision tree model

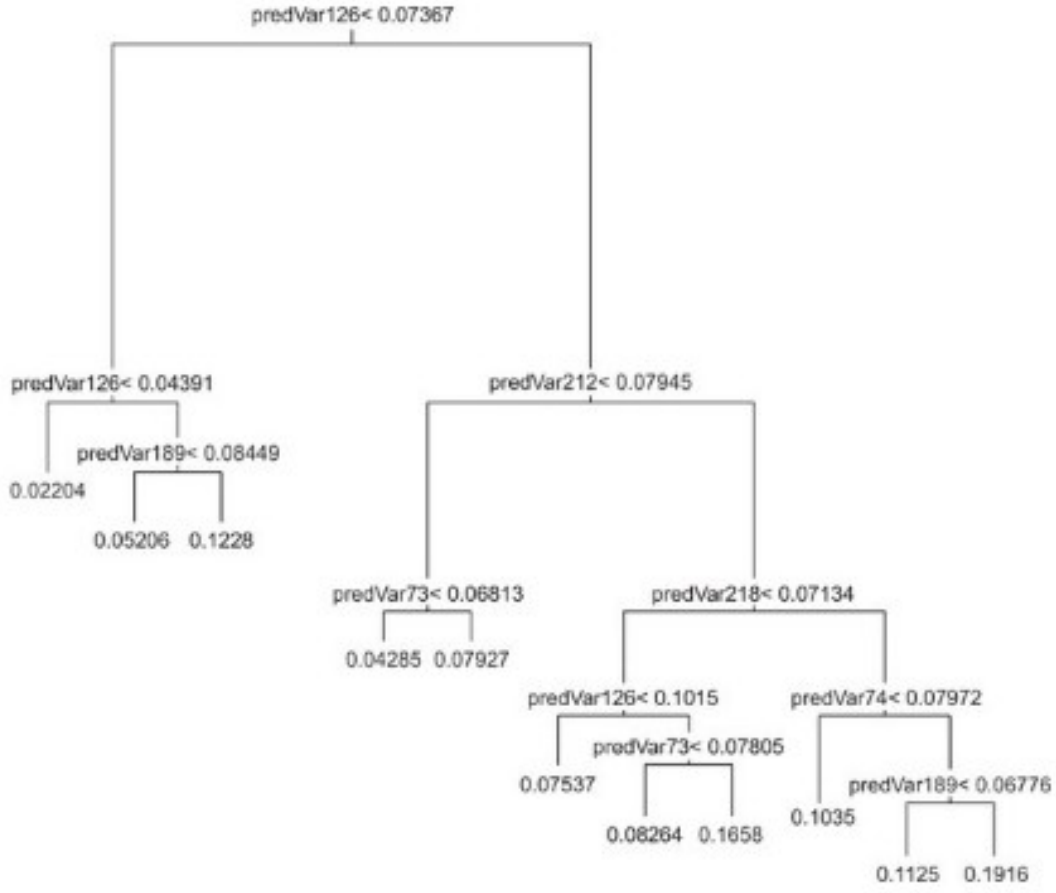
- Building a decision tree involves proposing many possible *data cuts* and then choosing best cuts based on simultaneous competing criteria of predictive power, cross-validation strength, and interaction with other chosen cuts.
- One of the advantages of using a canned package for decision tree work is not having to worry about tree construction details.
- **To build a decision tree:**
- Call `rpart()` is to just give it a list of variables and see what happens (`rpart()`, unlike many R modelling techniques, has built-in code for dealing with missing values).
- There are chances of building bad decision tree when we have a complicated model, categorical variables with very many levels, and have a lot more NAs/missing data.
- To improve the tree try to `control rpart()` model complexity, we need to monkey a bit with the controls. We pass in an extra argument, `rpart.control()`

How decision tree models work

- Node 1 called *Root*
- Each node other than the root node has a parent,
- The parent of node *k* is node `floor(k/2)`.
- A node with no children, which is called a *leaf node* (marked with stars)
- The remaining three numbers reported for each node are the number of training items that navigated to the node, the deviance of the set of training items that navigated to the node (a measure of how much uncertainty remains at a given decision tree node), and the fraction of items that were in the positive class at the node (which is the prediction for leaf nodes).

Decision tree

```
>print(tmodel)
n= 40518
 node), split, n, deviance, yval
    * denotes terminal node
1) root 40518 2769.3550 0.07379436
2) predVar126< 0.07366888 18188
726.4097 0.04167583
4) predVar126< 0.04391312 8804
189.7251 0.02203544 *
```



Using nearest neighbour methods

- A *k-nearest neighbor* (*KNN*) method scores an example by finding the *k* training examples nearest to the example and then taking the average of their outcomes as the score.
- The notion of nearness is basic Euclidean distance, so it can be useful to select nonduplicate variables, rescale variables, and orthogonalize variables.
- One problem with KNN is the nature of its concept space.
- For example, if we were to run a 3-nearest neighbour analysis on our data, we have to understand that with three neighbours from the training data, we'll always see either zero, one, two, or three examples of churn.
- For events with unbalanced outcomes (that is, probabilities not near 50%), we suggest using a large *k* so KNN can express a useful range of probabilities.
- For a good *k*, we suggest trying something such that you have a good chance of seeing 10 positive examples in each neighborhood (allowing your model to express rates smaller than your baseline rate to some precision).
- KNN is expensive both in time and space. Sometimes we can get similar results with more efficient methods such as logistic regression

Using Naive Bayes

- Naive Bayes is an interesting method that memorizes how each training variable is related to outcome, and then makes predictions by multiplying together the effects of each variable.
- demonstrate this, let's use a scenario in which we're trying to predict whether somebody is employed based on their level of education, their geographic region, and other variables.
- Naive Bayes begins by reversing that logic and asking this question: Given that you are employed, what is the probability that you have a high school education? From that data, we can then make our prediction regarding employment.
- Suppose we define our evidence **`(ev_1)`** as the predicate `education=="High School"`, which is true when the variable **`x_1`** (education) takes on the value **`X_1`** ("High School").
- Let's call the outcome **`y`** (taking on values **`T`** or `True` if the person is employed and `F` otherwise).
- Then the fraction of all positive examples where `ev_1` is true is an approximation to the *conditional probability* of **`ev_1`**, given **`y==T`**. This is usually written as **`P(ev1|y==T)`**.
- But what we want to estimate is the conditional probability of a subject being employed, given that they have a high school education: **`P(y==T|ev1)`**.
- How do we get from **`P(ev1|y==T)`** (the quantities we know from our training data) to an estimate of **`P(y==T|ev1 ... evN)`** (what we want to predict)?

Bayes' law tells us we can expand P(y==T|ev1) and P(y==F|ev1) like this:

$$P(y==T \mid ev_1) = \frac{P(y==T) \times P(ev_1 \mid y==T)}{P(ev_1)}$$

$$P(y==F \mid ev_1) = \frac{P(y==F) \times (P(ev_1 \mid y==F)}{P(ev_1)}$$

The left-hand side is what you want; the right-hand side is all quantities that can be estimated from the statistics of the training data. For a single feature ev1, this buys us little as we could derive P(y==T|ev1) as easily from our training data as from P(ev1|y==T). For multiple features (ev1 … evN) this sort of expansion is useful. The *Naive Bayes assumption* lets us assume that all the evidence is conditionally independent of each other for a given outcome:

$P(ev_1 \& \ldots ev_N \mid y==T) \approx P(ev_1 \mid y==T) \times P(ev_2 \mid y==T) \times \ldots P(ev_N \mid y==T)$

$P(ev_1 \& \ldots ev_N \mid y==F) \approx P(ev_1 \mid y==F) \times P(ev_2 \mid y==F) \times \ldots P(ev_N \mid y==F)$

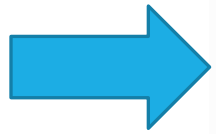
This gives us the following:

$$P(y==T \mid ev_1 \& \cdots ev_N) \approx \frac{P(y==T) \times (P(ev_1 \mid y==T) \times \cdots P(ev_N \mid y==T))}{P(ev_1 \& \cdots ev_N)}$$

$$P(y==F \mid ev_1 \& \cdots ev_N) \approx \frac{P(y==F) \times (P(ev_1 \mid y==F) \times \cdots P(ev_N \mid y==F))}{P(ev_1 \& \cdots ev_N)}$$

The numerator terms of the right sides of the final expressions can be calculated efficiently from the training data, while the left sides can't. We don't have a direct scheme for estimating the denominators in the Naive Bayes expression (these are called the *joint probability of the evidence*).

Building, applying, and evaluating a Naive Bayes model



```
pPos <- sum(dTrain[,outcome]==pos)/length(dTrain[,outcome])

nBayes <- function(pPos,pf) {
   pNeg <- 1 - pPos
   smoothingEpsilon <- 1.0e-5
   scorePos <- log(pPos + smoothingEpsilon) +
      rowSums(log(pf/pPos + smoothingEpsilon))
   scoreNeg <- log(pNeg + smoothingEpsilon) +
      rowSums(log((1-pf)/(1-pPos) + smoothingEpsilon))
    m <- pmax(scorePos,scoreNeg)
   expScorePos <- exp(scorePos-m)
   expScoreNeg <- exp(scoreNeg-m)
   expScorePos/(expScorePos+expScoreNeg)
}
pVars <- paste('pred',c(numericVars,catVars),sep='')
dTrain$nbpred1 <- nBayes(pPos,dTrain[,pVars])
dCal$nbpred1 <- nBayes(pPos,dCal[,pVars])
dTest$nbpred1 <- nBayes(pPos,dTest[,pVars])
 print(calcAUC(dTrain$nbpred1,dTrain[,outcome]))
## [1] 0.9757348
print(calcAUC(dCal$nbpred1,dCal[,outcome]))
## [1] 0.5995206
print(calcAUC(dTest$nbpred1,dTest[,outcome]))
## [1] 0.5956515
```

Define a function that performs the Naive Bayes prediction.

For each row, compute (with a smoothing term) the sum of log(P[positive & evidence_i]/P[positive]) across all columns. This is equivalent to the log of the product of P[evidence_i | positive] up to terms that don't depend on the positive/negative outcome.

Exponentiate to turn sums back into products, but make sure we don't cause a floating point overflow in doing so.

For each row, compute (with a smoothing term) the sum of log(P[negative & evidence_i]/P[negative]) across all columns. This is equivalent to the log of the product of P[evidence_i | negative] up to terms that don't depend on the positive/negative outcome.

Use the fact that the predicted positive probability plus the predicted negative probability should sum to 1.0 to find and eliminate Z. Return the correctly scaled predicted odds of being positive as our forecast.

Apply the function to make the predictions.

Calculate the AUCs. Notice the overfit—fantastic performance on the training set that isn't repeated on the calibration or test sets.

Smoothing

- Most important design parameter in Naive Bayes is how *smoothing* is handled.
- The idea of smoothing is an attempt to obey *Cromwell's rule* that no probability estimate of `0` should ever be used in probabilistic reasoning.
- This is because if you're combining probabilities by multiplication (the most common method of combining probability estimates), then once some term is 0, the entire estimate will be 0 *no matter what the values of the other terms are*.
- The most common form of smoothing is called *Laplace smoothing*, which counts k successes out of n trials as a success ratio of `(k+1)/(n+1)` and not as a ratio of k/n (defending against the `k=0` case).
- Frequentist statisticians think of smoothing as a form of regularization and Bayesian statisticians think of smoothing in terms of priors.

Document classification and Naive Bayes

- Naive Bayes is the workhorse method when classifying text documents.
- This is because the standard model for text documents (usually called *bag-of-words* or *bag-of-k-grams*) can have an extreme number of possible features.
- In the bag-of-k-grams model, we pick a small k (typically 2) and each possible consecutive sequence of k words is a possible feature.
- Each document is represented as a *bag*, which is a sparse ector indicating which k-grams are in the document.
- The number of possible features runs into the millions, but each document only has a non-zero value on a number of features proportional to k times the size of the document.

Key Takeaways

- Always try single-variable models before trying more complicated techniques.
- Single-variable modelling techniques give you a useful start on variable selection.
- Always compare your model performance to the performance of your best single-variable model.
- Consider decision trees, nearest neighbour, and naive Bayes models as basic data memorization techniques and, if appropriate, try them early in your projects.