Intro to tidymodels

Author: Steve Pittard [wsp@emory.edu](mailto:wsp@emory.edu)

Last update: 14 Oct 2021

Note you may use this in your work as long as it’s not part of a commercial project. Any use should cite this notebook and author accordingly.

https://cimentadaj.github.io/blog/2020-02-06-the-simplest-tidy-machine-learning-workflow/the-simplest-tidy-machine-learning-workflow/

# Introduction

This notebook motivates the use of the tidy models as a way to simplify the processes associated with building models and evaluating them. While it is entirely possible to use Base R methods to organize data and build models thereon, the tidyverse and tidymodels provide a uniform interface for interacting with data and model assembly.

What I hope you get out of this includes the following:

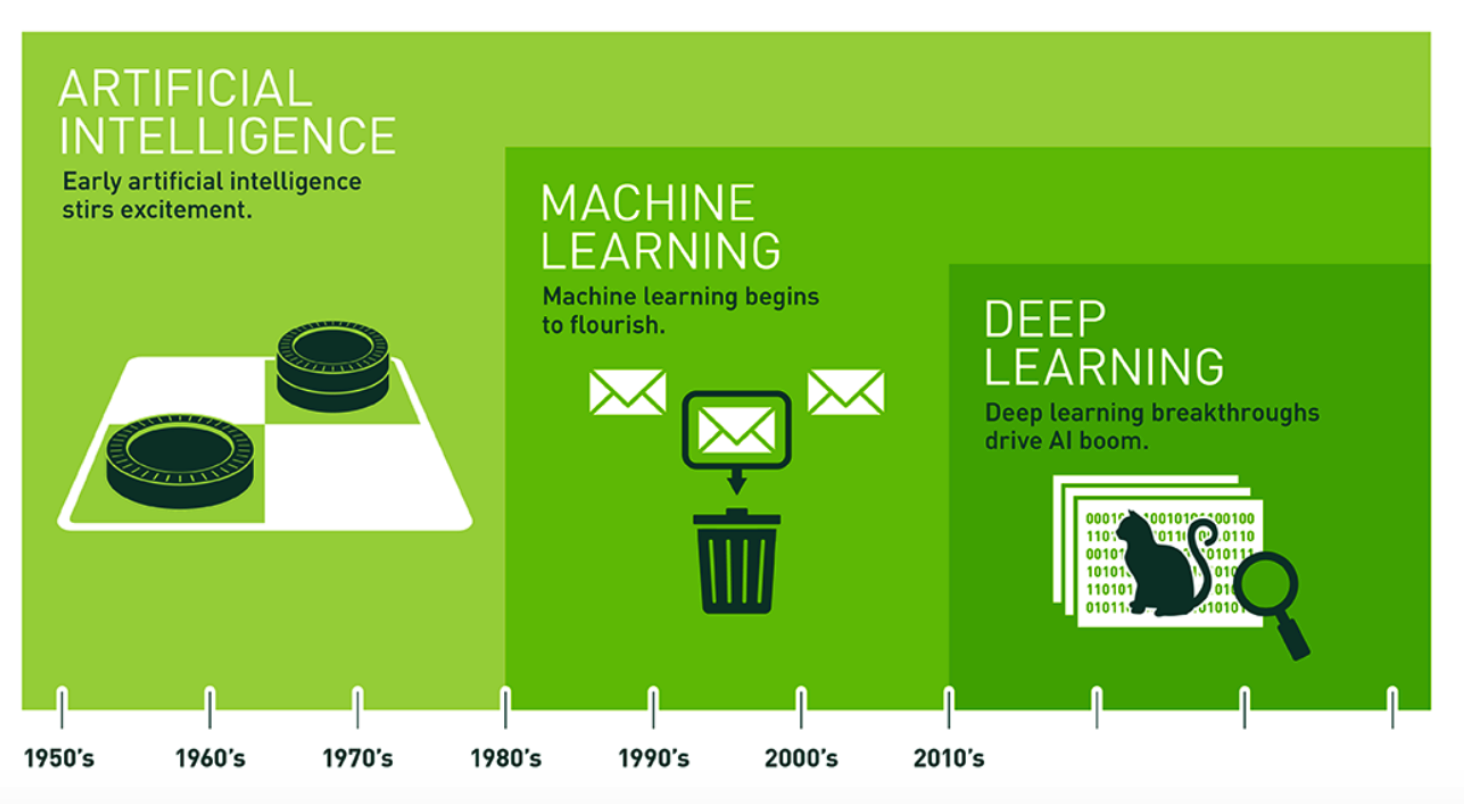
1. An idea of what is involved into basic ML and building models
2. You can use Base R to do modeling and it’s fine
3. But the tidyverse offers a well thought out philosophy for managing data
4. The tidymodels package builds on top of the tidyverse to carry the philosophy into ML

## Predictive Modeling

Predictive Modeling is a type of Machine Learning which itself is a sub branch of Artificial Intelligence. The following graphic provides us with some history of these domains. This is helpful if you are trying to orient yourself in the world of analytics and machine learning. Note that AI has been around for quite some time. The Wikipedia definition of AI is:

The study of "intelligent agents": any device that perceives its environment and takes actions that maximize its chance of successfully achieving its goals

I found the following image on Intel web page but can’t remember where but I didn’t generate it. Someone at Intel did.



Machine Learning relies upon “patterns and inference” to “perform a specific task without using explicit instructions”. It is a form of Applied AI that attempts to automatically learn from experience without being explicitly programmed. Think of Predictive Modeling as a subset of this which falls into two categories:

**Supervised**

Algorithms that build a model on a set of data containing both the inputs and the desired outputs (“labels” or known numeric values). When you want to map input to known output labels. Build a model that, when applied to “new” data, will hopefully predict the correct label.

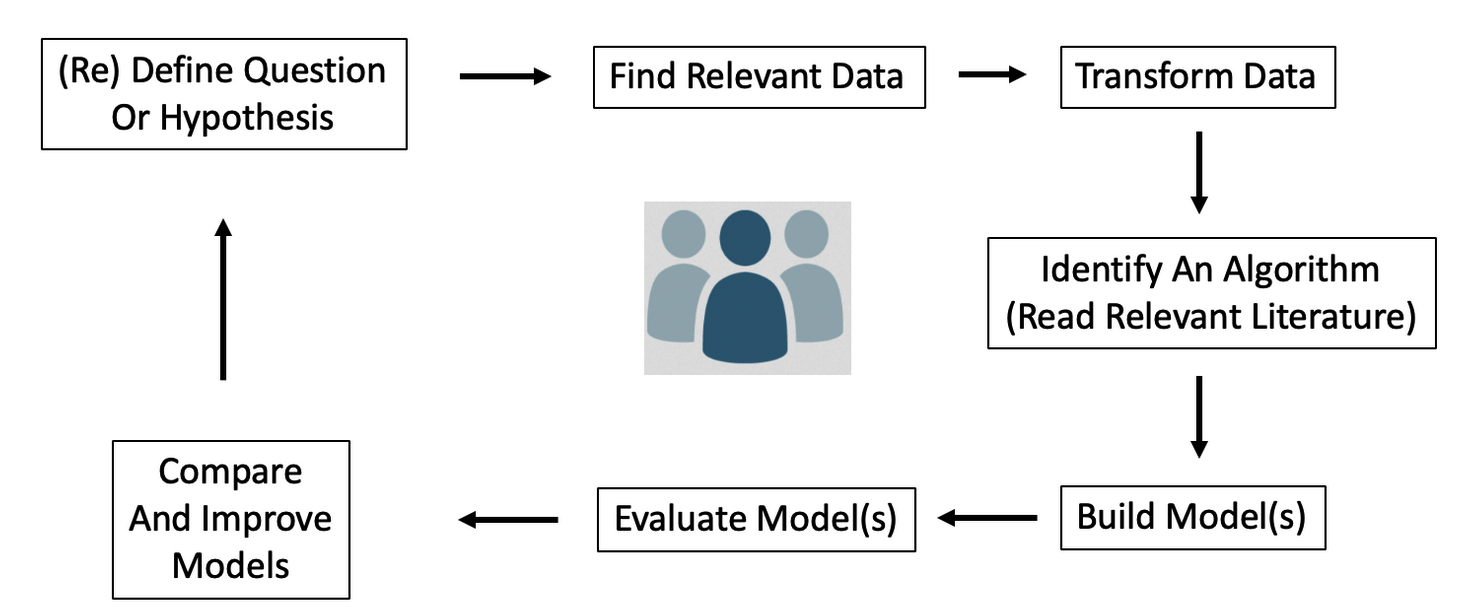
**Unsupervised**

Algorithms that take a set of data that contains only inputs, and find structure in the data (e.g. clustering of data points).

This lecture is concerned primarily with Predictive Modeling. Some examples of Predictive Modeling include:

Predict current CD4 cell count of an HIV-positive patient using genome sequences  
  
Predict Success of Grant Applications  
  
Use attributes of chemical compounds to predict likelihood of hepatic injury  
  
How many copies of a new book will sell ?  
  
Will a customer change Internet Service Providers ?

A typical workflow in support of predictive modeling might look this:

 ### Types of Models

I’m grossly simplifying things here by jumping straight to predictive modeling as if that is the only type of model that matters when it is just one approach to understanding data. We could spend lots of time doing exploratory data analysis which is frequently done as part of, or before predictive modeling, since we might want to first understand possible relationships in our data.

Or we could treat it as an inferential problem wherein we make assumptions about the data and treat it like a statistical hypothesis wherein we implement statistical tests to determine effectiveness say relative to coefficients in a regression. We could have a very statistically rigorous model whose predictive power on new data might not be “good”. We could also have a model whose predictive power on new data is “good” but not so statistically. This can be a trade off. I’m not going to spend a lot of time here comparing the two approaches but they each have their own considerations.

<https://raw.githubusercontent.com/dataprofessor/infographic/master/01-Building-the-Machine-Learning-Model.JPG>

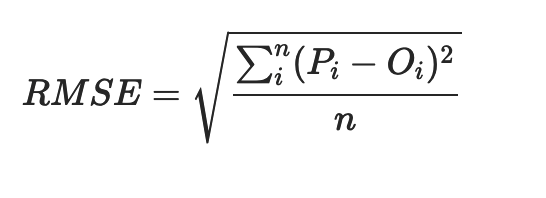
**In-Sample vs Out-Of-Sample Error**

The goal of predictive model is to generate models that can generalize to new data. It would be good if any model we generate could provide a good estimate of out of sample error. It’s easy to generate a model on an entire data set (in sample data) and then turn around and use that data for prediction. But how will it perform on new data ? Haven’t we just over trained our model ?

**Performance Metrics**

For either case (regression vs classification) we need some type of metric or measure to let us know how well a given model will work on new or unseen data - also known as “out of sample” data. for Classification problems we look at things like “sensitivity”, “specificity”, “accuracy”, and “Area Under Curve”.

For Quantitative outcomes, we look at things like Root Mean Square Error (RMSE) or Mean Absolute Error (MAE). Here is the formula for Root Mean Square Error (RMSE). P represents a vector of predictions and O represents a vector of the observed (true) values.



## The “Old Way” - Using Base R and addons

While **old** sounds somewhat negative, there is absolutely nothing wrong using an established approach to model data using Base R. This involves applying specific packages to build, for example, a logistic regression model, a decision tree, or a support vector machine.

The job involves identifying the appropriate package(s) and then dividing the data up in to training and testing pairs after which a function name is called to do the work.

At a minimum, one must specify 1) the training data set and 2) a formula which indicates what the target and predictor variables will be. Then you look at the result

1. Identify appropriate packages(s) - glm, RandomForest, ranger, svm, etc
2. Chop up the data into a training and test pair (possibly use cross fold validation)
3. Use the right function name and pass arguments - data and prediction formula
4. Examine result using a pre-defined performance diagnostic (e.g. RMSE, Accuracy, Sensitivity, AUC)

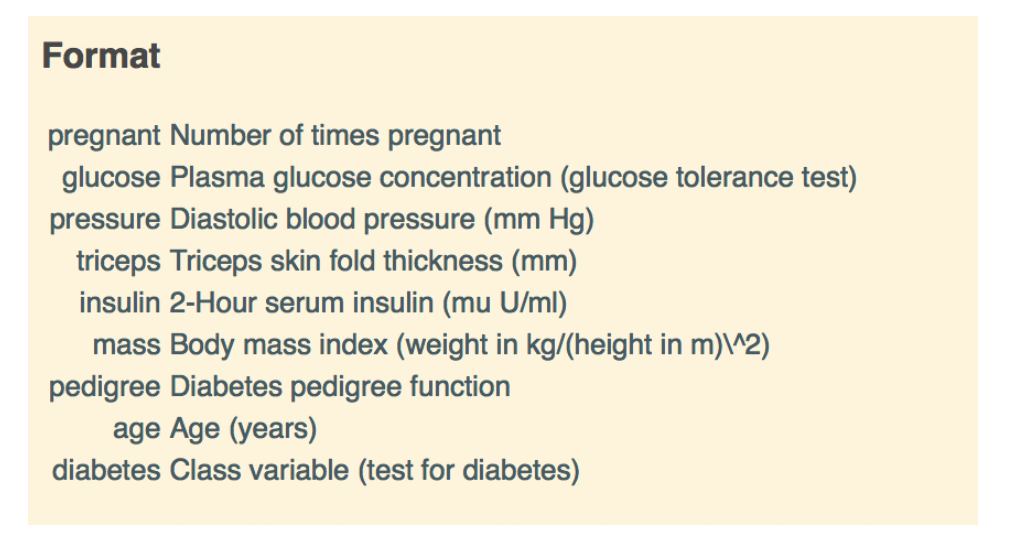
### An Example

Let’s run through an example using the infamous **Pima Indians** data set to predict whether someone is diabetic or not. First, let’s chop up the data into a training and test pair. Read in a copy.

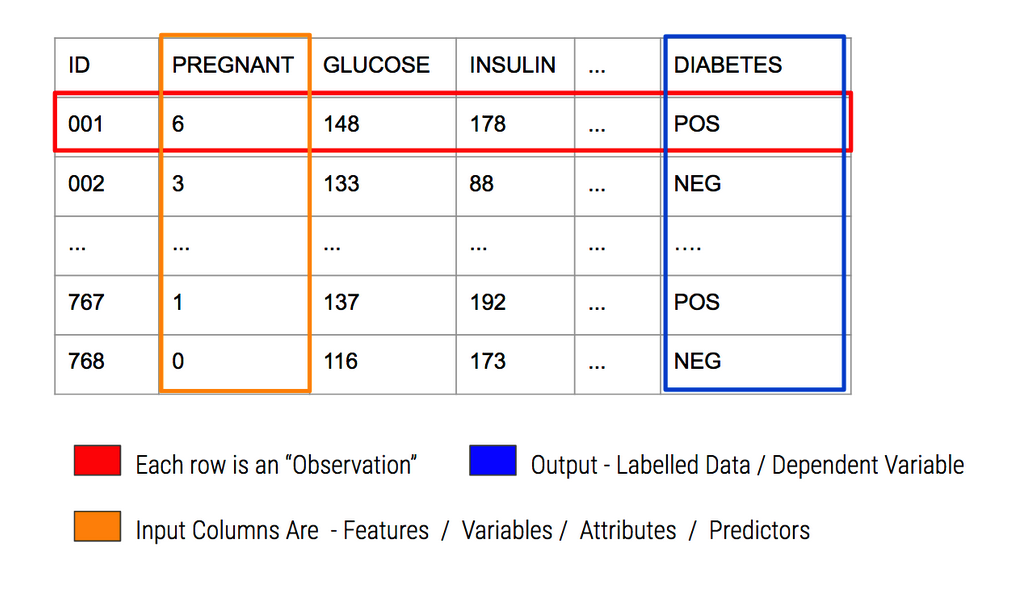
url <- "https://raw.githubusercontent.com/steviep42/bios534\_spring\_2020/master/data/pima.csv"  
pm <- read.csv(url)  
  
head(pm)

## pregnant glucose pressure triceps insulin mass pedigree age diabetes  
## 1 6 148 72 35 0 33.6 0.627 50 pos  
## 2 1 85 66 29 0 26.6 0.351 31 neg  
## 3 8 183 64 0 0 23.3 0.672 32 pos  
## 4 1 89 66 23 94 28.1 0.167 21 neg  
## 5 0 137 40 35 168 43.1 2.288 33 pos  
## 6 5 116 74 0 0 25.6 0.201 30 neg

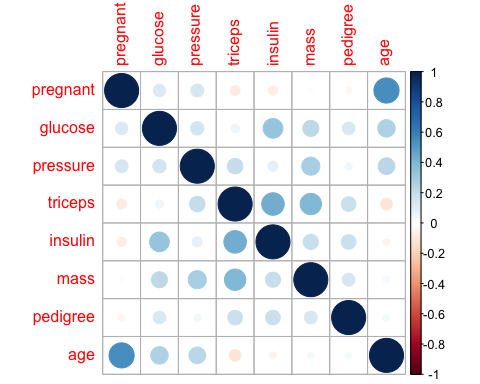
The description of the data set is as follows:



In predictive modeling we have some important terminology:

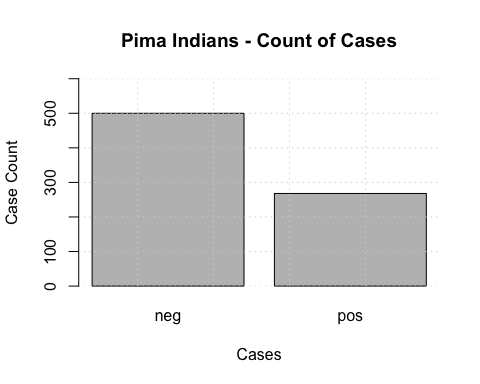


corrplot::corrplot(cor(pm[,-9]))



Look at some differences between the positive vs negative cases:

myt <- table(pm$diabetes)  
barplot(myt,  
 ylim=c(0,600),  
 main="Pima Indians - Count of Cases",  
 names.arg=names(myt),  
 ylab="Case Count",  
 xlab="Cases")  
grid()

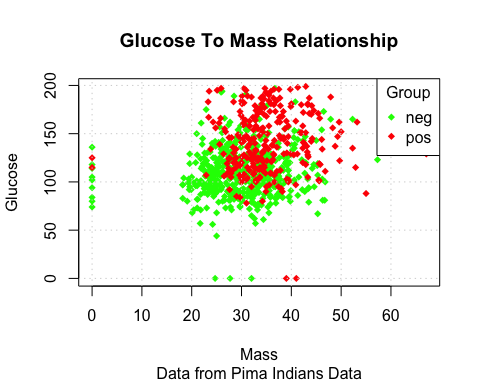


boxplot(glucose~diabetes,  
 data=pm,  
 horizontal = TRUE,  
 col="aquamarine",  
 main="Distribution of Glucose per Group")  
grid()



So let’s plot Glucose vs Mass to see if any interesting hypotheses come to mind (or not):

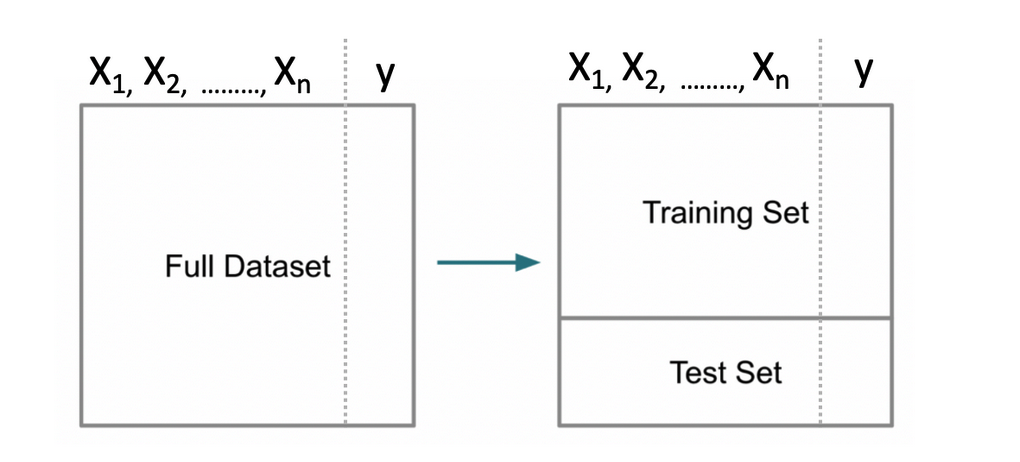
# Split the data on the target variable which is diabetes  
mysplits <- split(pm,pm$diabetes)  
  
# Create a plot window with appropriate dimensions  
plot(pm$mass,pm$glucose,type="n",  
 main="Glucose To Mass Relationship",  
 xlab="Mass",  
 ylab="Glucose",  
 sub="Data from Pima Indians Data")  
  
# Now put up the mass vs glucose for each group (pos and neg)  
cols <- c("green","red")  
for (ii in 1:length(mysplits)) {  
 points(mysplits[[ii]]$mass,  
 mysplits[[ii]]$glucose,  
 col=cols[ii],pch=18)   
}  
# Draw a grid  
grid()  
  
# Put up a legend  
legend("topright",  
 legend=names(mysplits),  
 pch=18,  
 col=cols,title="Group")



## Splitting Data For Model Building

At this point we want to build a model to predict whether a given observation will be positive for diabetes or negative.

What we will do is create a training and testing data set using the **sample** function. Here we will carve out roughly 80% of the data for training and 20% for testing.



set.seed(123) # Makes this example reproducible  
nrows <- nrow(pm)  
propo <- 0.70  
(idx <- sample(1:nrows,propo\*nrows))

## [1] 415 463 179 526 195 118 299 229 244 14 374 665 602 603 709 91 348 649  
## [19] 355 26 519 426 751 211 590 593 555 373 143 544 490 621 23 309 135 224  
## [37] 166 217 290 581 72 588 575 141 722 153 294 277 767 41 431 90 316 223  
## [55] 528 116 606 456 598 39 159 209 758 34 516 13 69 409 308 278 89 537  
## [73] 291 424 286 671 121 110 158 64 483 477 480 67 663 85 165 648 51 74  
## [91] 178 362 236 610 330 127 212 310 243 113 619 687 151 614 160 391 155 747  
## [109] 5 326 280 567 238 339 754 137 455 560 589 83 654 196 694 712 500 344  
## [127] 693 459 20 764 164 52 534 177 554 84 523 392 302 597 668 650 430 428  
## [145] 250 429 398 714 381 545 40 522 473 200 125 265 186 573 252 458 152 54  
## [163] 538 235 289 185 413 617 735 607 205 697 564 684 701 346 664 468 509 57  
## [181] 457 357 279 270 347 129 218 337 749 539 748 553 390 498 222 421 627 163  
## [199] 656 704 674 225 389 117 629 55 731 557 768 134 447 104 591 210 349 401  
## [217] 258 635 386 725 24 466 130 682 377 170 445 234 422 508 689 80 688 475  
## [235] 696 343 323 479 450 111 317 574 287 292 226 297 681 237 628 33 746 396  
## [253] 721 707 76 94 636 30 562 434 175 706 532 115 739 338 96 465 358 543  
## [271] 695 661 502 580 397 404 230 148 667 556 350 425 631 423 202 81 558 503  
## [289] 232 670 106 375 11 669 364 550 403 461 549 31 414 505 699 513 484 16  
## [307] 197 420 678 417 412 551 12 437 609 66 50 204 579 435 741 559 384 122  
## [325] 399 472 315 259 353 248 604 48 331 100 108 301 10 499 658 752 402 515  
## [343] 442 653 600 395 744 8 114 261 29 306 659 679 282 73 267 738 262 733  
## [361] 451 745 219 184 352 662 119 643 685 691 482 36 563 240 379 120 488 304  
## [379] 432 723 449 105 281 180 547 448 241 548 167 47 191 37 174 599 303 207  
## [397] 19 615 708 504 103 760 652 188 139 762 690 189 311 361 572 38 633 319  
## [415] 376 334 416 546 393 371 436 21 199 87 728 497 464 520 536 517 622 367  
## [433] 6 128 156 288 49 227 239 193 462 507 491 190 112 378 625 467 576 321  
## [451] 59 305 61 540 525 736 750 88 132 612 251 203 246 641 460 700 366 131  
## [469] 578 162 645 168 276 78 566 743 95 221 405 161 533 242 181 620 761 594  
## [487] 273 187 171 313 582 136 79 638 521 400 446 427 345 646 138 719 583 686  
## [505] 272 673 454 755 201 637 657 489 632 2 65 260 247 734 710 418 640 206  
## [523] 124 596 7 228 45 514 25 753 470 672 268 501 263 169 711

Now, we create the train/test pair:

pm$diabetes <- factor(pm$diabetes)  
pm\_training <- pm[idx,]  
pm\_testing <- pm[-idx,]  
  
print(paste("Training data has",nrow(pm\_training),"rows"))

## [1] "Training data has 537 rows"

print(paste("Testing data has",nrow(pm\_testing),"rows"))

## [1] "Testing data has 231 rows"

## A First Model

Let’s build a model using the Generalized Linear Models function. Note that there are other types of functions we could use but most people have at least heard of Logistic Regression so let’s start with that as it also provides us the opportunity to understand some basic ML concepts.

The **glm** function is part of Base R so you don’t need to load any additional packages to use it. We need to specify a formula, some training data, and a “family” argument.

myglm <- glm(diabetes ~ .,  
 data = pm\_training,  
 family = "binomial")  
  
summary(myglm)

##   
## Call:  
## glm(formula = diabetes ~ ., family = "binomial", data = pm\_training)  
##   
## Deviance Residuals:   
## Min 1Q Median 3Q Max   
## -2.2424 -0.7256 -0.4283 0.7341 2.9311   
##   
## Coefficients:  
## Estimate Std. Error z value Pr(>|z|)   
## (Intercept) -8.405409 0.841872 -9.984 < 2e-16 \*\*\*  
## pregnant 0.103471 0.037973 2.725 0.00643 \*\*   
## glucose 0.035730 0.004563 7.830 4.89e-15 \*\*\*  
## pressure -0.012707 0.006057 -2.098 0.03590 \*   
## triceps 0.003563 0.008088 0.440 0.65959   
## insulin -0.001710 0.001060 -1.613 0.10671   
## mass 0.088735 0.017954 4.942 7.72e-07 \*\*\*  
## pedigree 0.696250 0.334761 2.080 0.03754 \*   
## age 0.017015 0.011066 1.538 0.12415   
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
##   
## (Dispersion parameter for binomial family taken to be 1)  
##   
## Null deviance: 694.17 on 536 degrees of freedom  
## Residual deviance: 509.76 on 528 degrees of freedom  
## AIC: 527.76  
##   
## Number of Fisher Scoring iterations: 5

It’s helpful to know what this object contains. Many people do not bother to look but there is a lot of information contained therein:

str(myglm)

## List of 30  
## $ coefficients : Named num [1:9] -8.40541 0.10347 0.03573 -0.01271 0.00356 ...  
## ..- attr(\*, "names")= chr [1:9] "(Intercept)" "pregnant" "glucose" "pressure" ...  
## $ residuals : Named num [1:537] 2.82 -1.23 -4.17 -1.05 -1.11 ...  
## ..- attr(\*, "names")= chr [1:537] "415" "463" "179" "526" ...  
## $ fitted.values : Named num [1:537] 0.3547 0.1858 0.7605 0.0438 0.1003 ...  
## ..- attr(\*, "names")= chr [1:537] "415" "463" "179" "526" ...  
## $ effects : Named num [1:537] 4.67 -3.171 -8.547 -0.638 -1.41 ...  
## ..- attr(\*, "names")= chr [1:537] "(Intercept)" "pregnant" "glucose" "pressure" ...  
## $ R : num [1:9, 1:9] -9.11 0 0 0 0 ...  
## ..- attr(\*, "dimnames")=List of 2  
## .. ..$ : chr [1:9] "(Intercept)" "pregnant" "glucose" "pressure" ...  
## .. ..$ : chr [1:9] "(Intercept)" "pregnant" "glucose" "pressure" ...  
## $ rank : int 9  
## $ qr :List of 5  
## ..$ qr : num [1:537, 1:9] -9.1125 0.0427 0.0468 0.0225 0.033 ...  
## .. ..- attr(\*, "dimnames")=List of 2  
## .. .. ..$ : chr [1:537] "415" "463" "179" "526" ...  
## .. .. ..$ : chr [1:9] "(Intercept)" "pregnant" "glucose" "pressure" ...  
## ..$ rank : int 9  
## ..$ qraux: num [1:9] 1.05 1.05 1.03 1 1 ...  
## ..$ pivot: int [1:9] 1 2 3 4 5 6 7 8 9  
## ..$ tol : num 1e-11  
## ..- attr(\*, "class")= chr "qr"  
## $ family :List of 12  
## ..$ family : chr "binomial"  
## ..$ link : chr "logit"  
## ..$ linkfun :function (mu)   
## ..$ linkinv :function (eta)   
## ..$ variance :function (mu)   
## ..$ dev.resids:function (y, mu, wt)   
## ..$ aic :function (y, n, mu, wt, dev)   
## ..$ mu.eta :function (eta)   
## ..$ initialize: language { if (NCOL(y) == 1) { ...  
## ..$ validmu :function (mu)   
## ..$ valideta :function (eta)   
## ..$ simulate :function (object, nsim)   
## ..- attr(\*, "class")= chr "family"  
## $ linear.predictors: Named num [1:537] -0.599 -1.478 1.155 -3.084 -2.194 ...  
## ..- attr(\*, "names")= chr [1:537] "415" "463" "179" "526" ...  
## $ deviance : num 510  
## $ aic : num 528  
## $ null.deviance : num 694  
## $ iter : int 5  
## $ weights : Named num [1:537] 0.2289 0.1513 0.1822 0.0419 0.0903 ...  
## ..- attr(\*, "names")= chr [1:537] "415" "463" "179" "526" ...  
## $ prior.weights : Named num [1:537] 1 1 1 1 1 1 1 1 1 1 ...  
## ..- attr(\*, "names")= chr [1:537] "415" "463" "179" "526" ...  
## $ df.residual : int 528  
## $ df.null : int 536  
## $ y : Named num [1:537] 1 0 0 0 0 0 1 0 1 1 ...  
## ..- attr(\*, "names")= chr [1:537] "415" "463" "179" "526" ...  
## $ converged : logi TRUE  
## $ boundary : logi FALSE  
## $ model :'data.frame': 537 obs. of 9 variables:  
## ..$ diabetes: Factor w/ 2 levels "neg","pos": 2 1 1 1 1 1 2 1 2 2 ...  
## ..$ pregnant: int [1:537] 0 8 5 3 8 5 14 4 6 1 ...  
## ..$ glucose : int [1:537] 138 74 143 87 85 78 100 197 119 189 ...  
## ..$ pressure: int [1:537] 60 70 78 60 55 48 78 70 50 60 ...  
## ..$ triceps : int [1:537] 35 40 0 18 20 0 25 39 22 23 ...  
## ..$ insulin : int [1:537] 167 49 0 0 0 0 184 744 176 846 ...  
## ..$ mass : num [1:537] 34.6 35.3 45 21.8 24.4 33.7 36.6 36.7 27.1 30.1 ...  
## ..$ pedigree: num [1:537] 0.534 0.705 0.19 0.444 0.136 ...  
## ..$ age : int [1:537] 21 39 47 21 42 25 46 31 33 59 ...  
## ..- attr(\*, "terms")=Classes 'terms', 'formula' language diabetes ~ pregnant + glucose + pressure + triceps + insulin + mass + pedigree + age  
## .. .. ..- attr(\*, "variables")= language list(diabetes, pregnant, glucose, pressure, triceps, insulin, mass, pedigree, age)  
## .. .. ..- attr(\*, "factors")= int [1:9, 1:8] 0 1 0 0 0 0 0 0 0 0 ...  
## .. .. .. ..- attr(\*, "dimnames")=List of 2  
## .. .. .. .. ..$ : chr [1:9] "diabetes" "pregnant" "glucose" "pressure" ...  
## .. .. .. .. ..$ : chr [1:8] "pregnant" "glucose" "pressure" "triceps" ...  
## .. .. ..- attr(\*, "term.labels")= chr [1:8] "pregnant" "glucose" "pressure" "triceps" ...  
## .. .. ..- attr(\*, "order")= int [1:8] 1 1 1 1 1 1 1 1  
## .. .. ..- attr(\*, "intercept")= int 1  
## .. .. ..- attr(\*, "response")= int 1  
## .. .. ..- attr(\*, ".Environment")=<environment: R\_GlobalEnv>   
## .. .. ..- attr(\*, "predvars")= language list(diabetes, pregnant, glucose, pressure, triceps, insulin, mass, pedigree, age)  
## .. .. ..- attr(\*, "dataClasses")= Named chr [1:9] "factor" "numeric" "numeric" "numeric" ...  
## .. .. .. ..- attr(\*, "names")= chr [1:9] "diabetes" "pregnant" "glucose" "pressure" ...  
## $ call : language glm(formula = diabetes ~ ., family = "binomial", data = pm\_training)  
## $ formula :Class 'formula' language diabetes ~ .  
## .. ..- attr(\*, ".Environment")=<environment: R\_GlobalEnv>   
## $ terms :Classes 'terms', 'formula' language diabetes ~ pregnant + glucose + pressure + triceps + insulin + mass + pedigree + age  
## .. ..- attr(\*, "variables")= language list(diabetes, pregnant, glucose, pressure, triceps, insulin, mass, pedigree, age)  
## .. ..- attr(\*, "factors")= int [1:9, 1:8] 0 1 0 0 0 0 0 0 0 0 ...  
## .. .. ..- attr(\*, "dimnames")=List of 2  
## .. .. .. ..$ : chr [1:9] "diabetes" "pregnant" "glucose" "pressure" ...  
## .. .. .. ..$ : chr [1:8] "pregnant" "glucose" "pressure" "triceps" ...  
## .. ..- attr(\*, "term.labels")= chr [1:8] "pregnant" "glucose" "pressure" "triceps" ...  
## .. ..- attr(\*, "order")= int [1:8] 1 1 1 1 1 1 1 1  
## .. ..- attr(\*, "intercept")= int 1  
## .. ..- attr(\*, "response")= int 1  
## .. ..- attr(\*, ".Environment")=<environment: R\_GlobalEnv>   
## .. ..- attr(\*, "predvars")= language list(diabetes, pregnant, glucose, pressure, triceps, insulin, mass, pedigree, age)  
## .. ..- attr(\*, "dataClasses")= Named chr [1:9] "factor" "numeric" "numeric" "numeric" ...  
## .. .. ..- attr(\*, "names")= chr [1:9] "diabetes" "pregnant" "glucose" "pressure" ...  
## $ data :'data.frame': 537 obs. of 9 variables:  
## ..$ pregnant: int [1:537] 0 8 5 3 8 5 14 4 6 1 ...  
## ..$ glucose : int [1:537] 138 74 143 87 85 78 100 197 119 189 ...  
## ..$ pressure: int [1:537] 60 70 78 60 55 48 78 70 50 60 ...  
## ..$ triceps : int [1:537] 35 40 0 18 20 0 25 39 22 23 ...  
## ..$ insulin : int [1:537] 167 49 0 0 0 0 184 744 176 846 ...  
## ..$ mass : num [1:537] 34.6 35.3 45 21.8 24.4 33.7 36.6 36.7 27.1 30.1 ...  
## ..$ pedigree: num [1:537] 0.534 0.705 0.19 0.444 0.136 ...  
## ..$ age : int [1:537] 21 39 47 21 42 25 46 31 33 59 ...  
## ..$ diabetes: Factor w/ 2 levels "neg","pos": 2 1 1 1 1 1 2 1 2 2 ...  
## $ offset : NULL  
## $ control :List of 3  
## ..$ epsilon: num 1e-08  
## ..$ maxit : num 25  
## ..$ trace : logi FALSE  
## $ method : chr "glm.fit"  
## $ contrasts : NULL  
## $ xlevels : Named list()  
## - attr(\*, "class")= chr [1:2] "glm" "lm"

For now just observe that some of the features (e.g. age, insulin, triceps) do not appear to be important or significant though we’ll ignore that for now - just like a lot of professional machine learning specialists do everyday (to their own peril).

We could do some “feature filtering” or “engineering” to first transform or remove features to make the model diagnostics better but that’s outside of the scope of this lecture.

Let’s pretend that all is well and now use this new model to predict outcomes using the test data set. Remember that we are attempting to predict a binary outcome - in this case whether the person is positive for diabetes or negative.

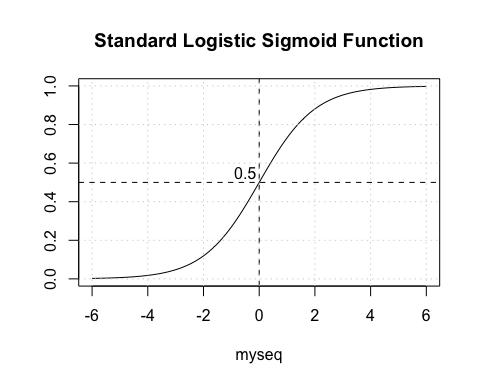
What we get back from the prediction object are probabilities for which we have to determine a threshold above which we would say the observation is “positive” for diabetes and, below the threshold, “negative”.

probs <- predict(myglm,  
 newdata = pm\_testing,  
 type = "response")  
  
probs[1:10]

## 1 3 4 9 15 17 18   
## 0.72751772 0.77340088 0.04425949 0.71110216 0.61359933 0.37579888 0.18733076   
## 22 27 28   
## 0.30034815 0.73345470 0.04357433

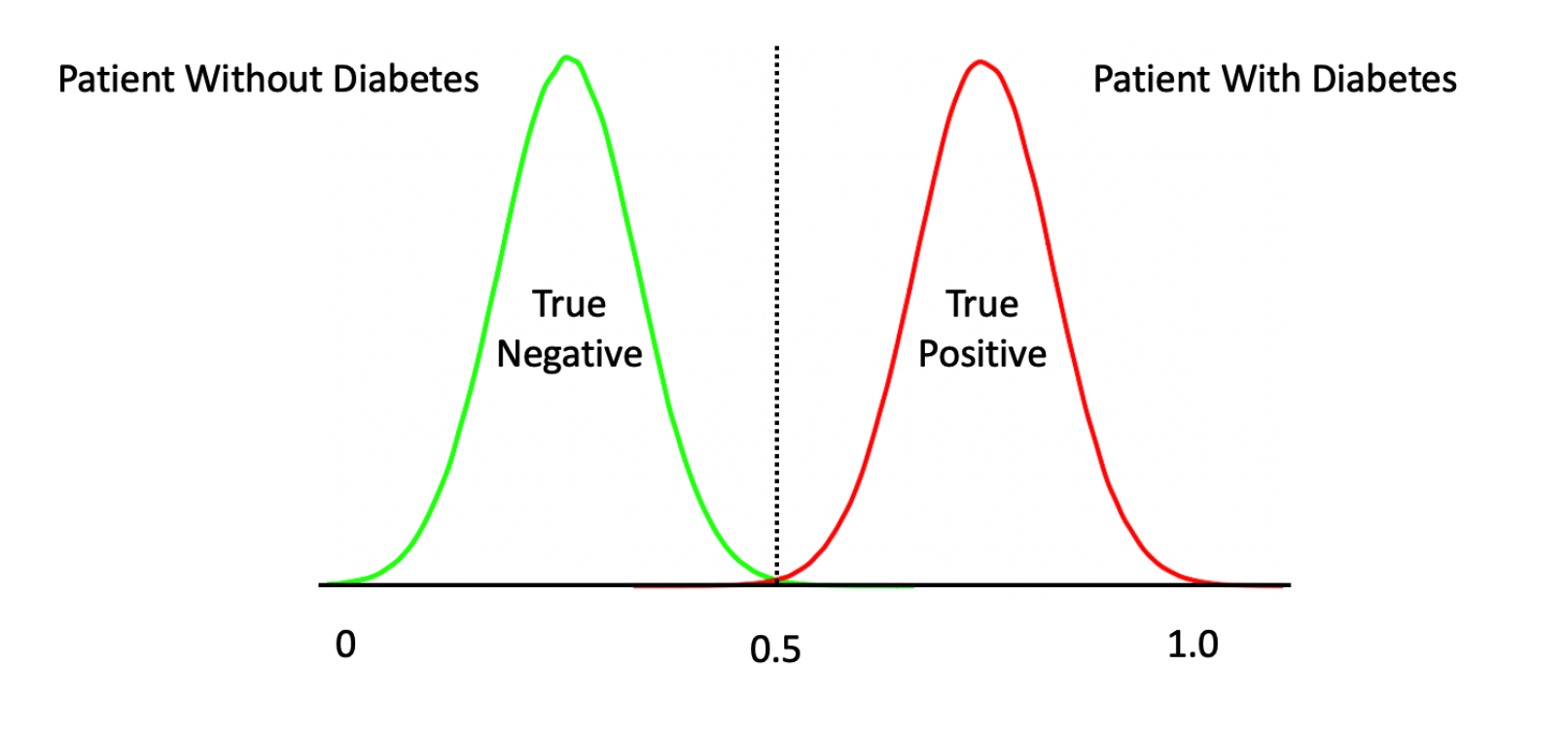
With logistic regression we are dealing with a curve like the one below which is a sigmoid function. The idea is to take our probabilities, which range between 0 and 1, and then pick a threshold (aka “alpha”) over which we would classify that observation / person as being positive for diabetes.

myseq <- seq(-6,6,.1)  
myfunc <- function(x) {1/(1+exp(-x))}  
plot(myseq,myfunc(myseq),  
 type = "l",  
 main = "Standard Logistic Sigmoid Function",  
 ylab = "")  
abline(h=0.5,lty=2)  
abline(v=0,lty=2)  
text(-0.5,0.55,"0.5")  
grid()

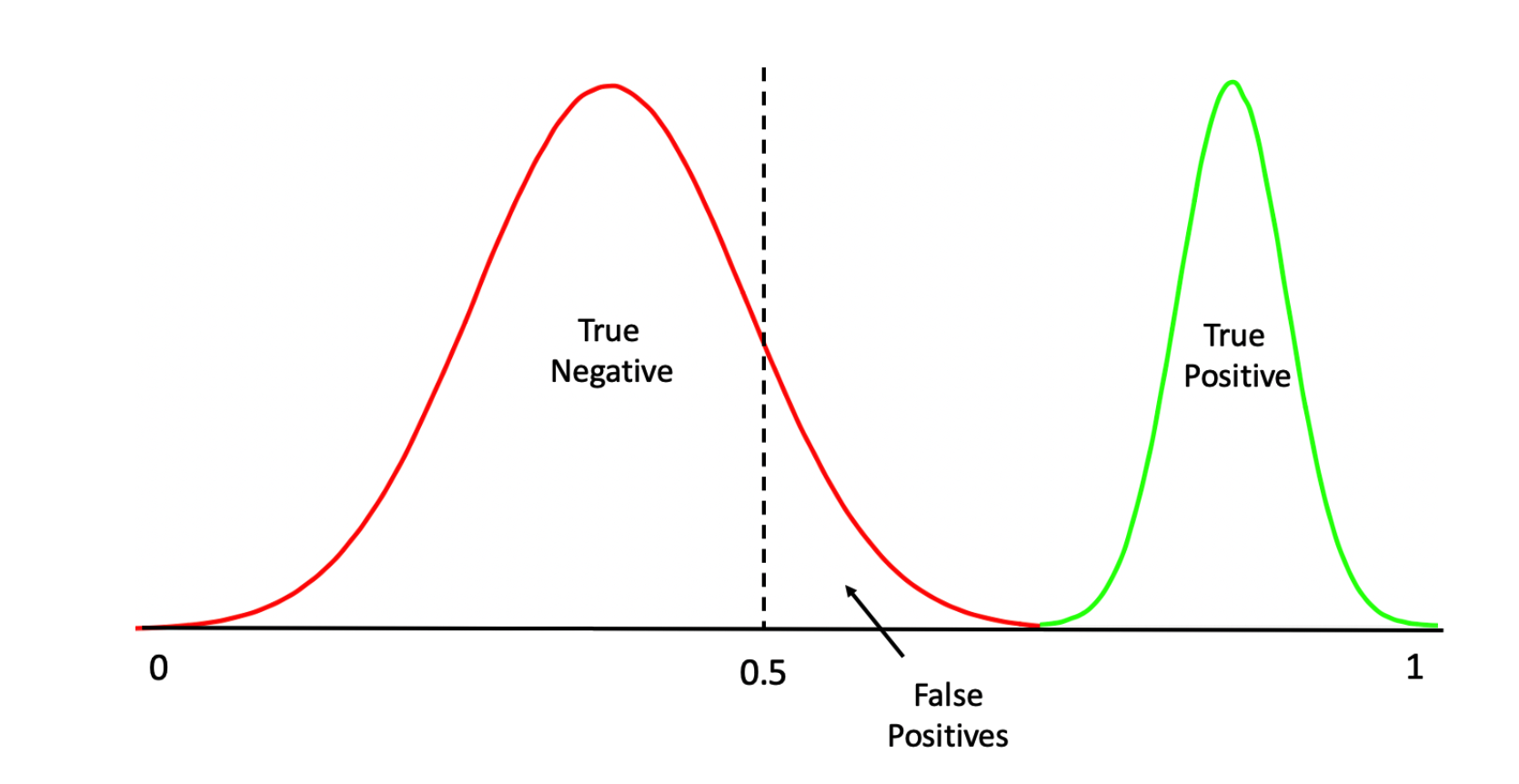


### Selecting The Correct Alpha

The temptation is to select 0.5 as the threshold such that if a returned probability exceeds 0.5 then we classify the associated subject as being “positive” for the disease. But then this assumes that the probabilities are distributed accordingly. This is frequently not the case though it doesn’t stop people from using 0.5. Another way to view this is as follows. This graphic shows a “perfect” classifier which more or less matches the logit function above:

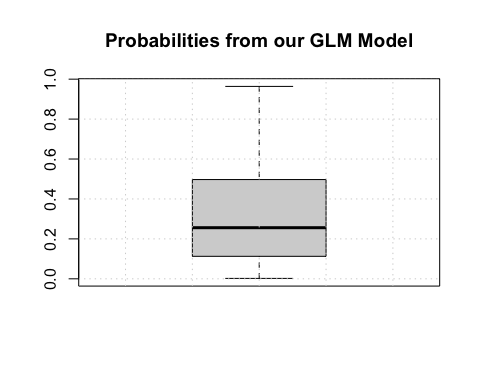


Note that this is a rare situation wherein there exists a clean separation between negative and positive cases. We could more likely have something like this:



One thing we should do here is to look at the distribution of the returned probabilities before making a decision about where to set the threshold. We can see clearly that selecting 0.5 in this case would not be appropriate.

boxplot(probs,   
 main="Probabilities from our GLM Model")  
grid()



The median appears to be somewhere around .25 so we could use that for now although we are just guessing.

mypreds <- ifelse(probs > 0.25,"pos","neg")  
mypreds <- factor(mypreds, levels = levels(pm\_testing[["diabetes"]]))  
mypreds[1:10]

## 1 3 4 9 15 17 18 22 27 28   
## pos pos neg pos pos pos neg pos pos neg   
## Levels: neg pos

### Confusion Matrices

Next, we would compare our predictions against the known outcomes which are stored in the test data frame:

# How does this compare to the truth ?  
table(predicted = mypreds,  
 actual = pm\_testing$diabetes)

## actual  
## predicted neg pos  
## neg 99 16  
## pos 51 65

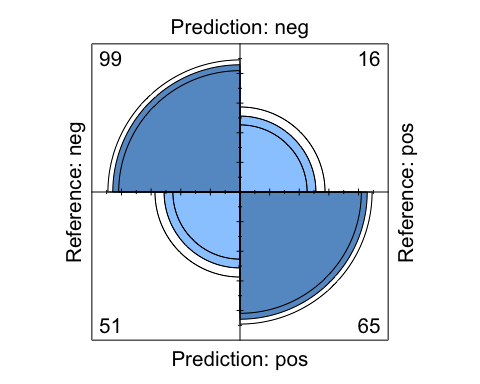
What we are doing is building a “Confusion Matrix” which can help us determine how effective our model is. From such a matrix table we can compute a number of “performance measures”, such as accuracy, precision, sensitivity, specificity and others, to help assess the quality of the model. In predictive modeling we are always interested in how well any given model will perform on “new” data.

There are some functions that can help us compute a confusion matrix. Because the variable we are trying to predict, (diabetes), is a two level factor, (“neg” or “pos”) we’ll need to turn our predictions into a comparable factor. Right now, it’s just a character string.

caret::confusionMatrix(mypreds,pm\_testing$diabetes)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction neg pos  
## neg 99 16  
## pos 51 65  
##   
## Accuracy : 0.71   
## 95% CI : (0.6468, 0.7676)  
## No Information Rate : 0.6494   
## P-Value [Acc > NIR] : 0.02999   
##   
## Kappa : 0.4207   
##   
## Mcnemar's Test P-Value : 3.271e-05   
##   
## Sensitivity : 0.6600   
## Specificity : 0.8025   
## Pos Pred Value : 0.8609   
## Neg Pred Value : 0.5603   
## Prevalence : 0.6494   
## Detection Rate : 0.4286   
## Detection Prevalence : 0.4978   
## Balanced Accuracy : 0.7312   
##   
## 'Positive' Class : neg   
##

fourfoldplot(caret::confusionMatrix(mypreds,pm\_testing$diabetes)$table)



newpreds <- ifelse(probs > 0.55,"pos","neg")  
newpreds <- factor(newpreds, levels = levels(pm\_testing[["diabetes"]]))  
newpreds[1:10]

## 1 3 4 9 15 17 18 22 27 28   
## pos pos neg pos pos neg neg neg pos neg   
## Levels: neg pos

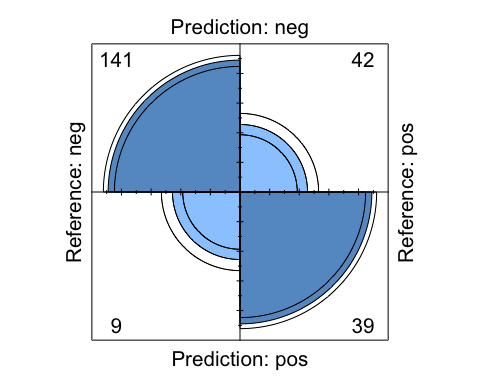
table(predicted = newpreds,  
 actual = pm\_testing$diabetes)

## actual  
## predicted neg pos  
## neg 141 42  
## pos 9 39

caret::confusionMatrix(newpreds,pm\_testing$diabetes)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction neg pos  
## neg 141 42  
## pos 9 39  
##   
## Accuracy : 0.7792   
## 95% CI : (0.7201, 0.831)  
## No Information Rate : 0.6494   
## P-Value [Acc > NIR] : 1.269e-05   
##   
## Kappa : 0.4651   
##   
## Mcnemar's Test P-Value : 7.433e-06   
##   
## Sensitivity : 0.9400   
## Specificity : 0.4815   
## Pos Pred Value : 0.7705   
## Neg Pred Value : 0.8125   
## Prevalence : 0.6494   
## Detection Rate : 0.6104   
## Detection Prevalence : 0.7922   
## Balanced Accuracy : 0.7107   
##   
## 'Positive' Class : neg   
##

fourfoldplot(caret::confusionMatrix(newpreds,pm\_testing$diabetes)$table)



### Performance Measures Revisited

This is helpful stuff although there are a number of measures to select as a primary performance metric. Ideally, we would already know which performance metric we would select to effectively “judge” the quality of our model.

In medical tests, “sensitivity” and “specificity” are commonly used. Some applications use “Accuracy” (which isn’t good when there is large group imbalance).

The problem here is that all we have done is looked at the confusion matrix corresponding to one specific (and arbitrary) threshold value when what we need is to look at a number of confusion matrices corresponding to many different thresholds.

For example, we might get a better sensitivity level had we selected the mean of the returned probabilities. This process could go on and on and on.

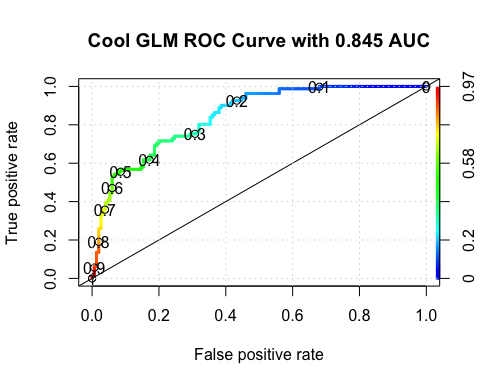
### The ROC Curve

One way to do this is to use something known as the ROC curve. Luckily, R has functions to do this. This isn’t surprising as it is a standard tool that has been in use for decades long before the hype of AI and ML was around. The ROC curve gives us a “one stop shop” for estimating a good value of alpha.

We want to maximize the area under a given ROC curve as it winds up being an effective way to judge the differences between one method and another. So, if we wanted to compare the glm model against a Random Forest model, we could use the respective AUC (Area Under Curve) metric to help us. This isn’t the only way to do this but it’s reasonable for now.

Compute the TPR and FPR for a lot of different alpha values. This is what the ROC curve is all about:

pred <- ROCR::prediction(predictions = probs,  
 labels = pm\_testing$diabetes)  
  
perf <- ROCR::performance(pred,  
 "tpr",  
 "fpr")  
  
myroc <- ROCR::performance(pred,measure="auc")  
  
plot(perf,colorize=T,  
 print.cutoffs.at=seq(0,1,by=0.1),  
 lwd=3,las=1,main=paste("Cool GLM ROC Curve with",round(myroc@y.values[[1]],3),"AUC"))  
abline(a = 0, b = 1)  
  
grid()



So what value of alpha corresponds to the statedAUC of .85 ? We’ll have to dig into the performance object to get that but it looks to be between 0.30 and 0.40. Note that this is somewhat academic since knowing the max AUC alone helps us decide if our model is any “good”. For completeness we could use another R function to nail this down:

proc <- pROC::roc(pm\_testing$diabetes,probs)

## Setting levels: control = neg, case = pos

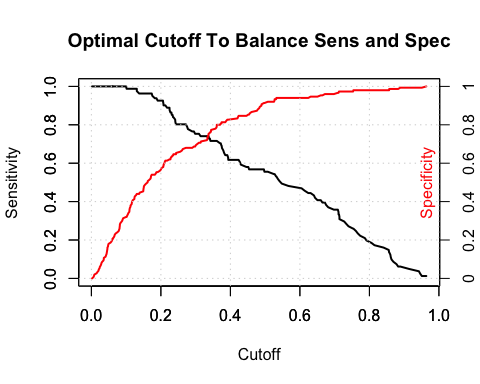
## Setting direction: controls < cases

round(pROC::coords(proc, "b", ret="t", transpose = FALSE),2)

## threshold  
## 1 0.36

Before we leave this example we could have also used the ROCR::performance function to help us answer this question (See <https://www.thetopsites.net/article/50636333.shtml>)

plot(unlist(ROCR::performance(pred, "sens")@x.values),   
 unlist(ROCR::performance(pred, "sens")@y.values),   
 type="l", lwd=2, ylab="Sensitivity", xlab="Cutoff",  
 main="Optimal Cutoff To Balance Sens and Spec")  
  
par(new=TRUE)  
  
plot(unlist(ROCR::performance(pred, "spec")@x.values),   
 unlist(ROCR::performance(pred, "spec")@y.values),   
 type="l", lwd=2, col='red', ylab="", xlab="")  
axis(4, at=seq(0,1,0.2),labels=as.character(seq(0,1,0.2)))  
mtext("Specificity",side=4, padj=-2, col='red')  
grid()



We haven’t accomplished very much here because we need to look at multiple versions of the data in case we sampled a number of outliers in the creation of our training data. Or, maybe we have excluded a large number of outliers in the training set so they wound up in the test data set which means that the predictive power of our model isn’t as robust as it should be.

Our next steps should involve creating multiple versions of the training and test pairs (say 3 times), compute the optimal AUC, and then look at how those values vary for each of those individual versions. If the AUCs vary widely then maybe our model is over training. If it’s not varying widely, it could be that that the model has high bias.

## What Did We Do?

1. So we chopped up some data into training and testing info
2. Built a model on the training data using logistic regression
3. Made a prediction on the test data using some arbitrary alpha value
4. Created a comparison between known reality and our predictions (confusion matrix)
5. We experimented with a couple of different alphas
6. Observed that creating a ROC curve gave us a good way to select an alpha
7. ROC curves can help us decide between one model or another.

## Cross Fold Validation

What if, when splitting the data into a training and testing pair, the test set wound up containing most of the outliers in the data (assuming there were any). This means that our trained model would probably under perform on the test data since the model did not “see” most of the outliers.

Or, consider the situation wherein the training data had most of the outliers and the resulting model was influenced by this. So, when applied to the testing data, the predictive results are not that great.

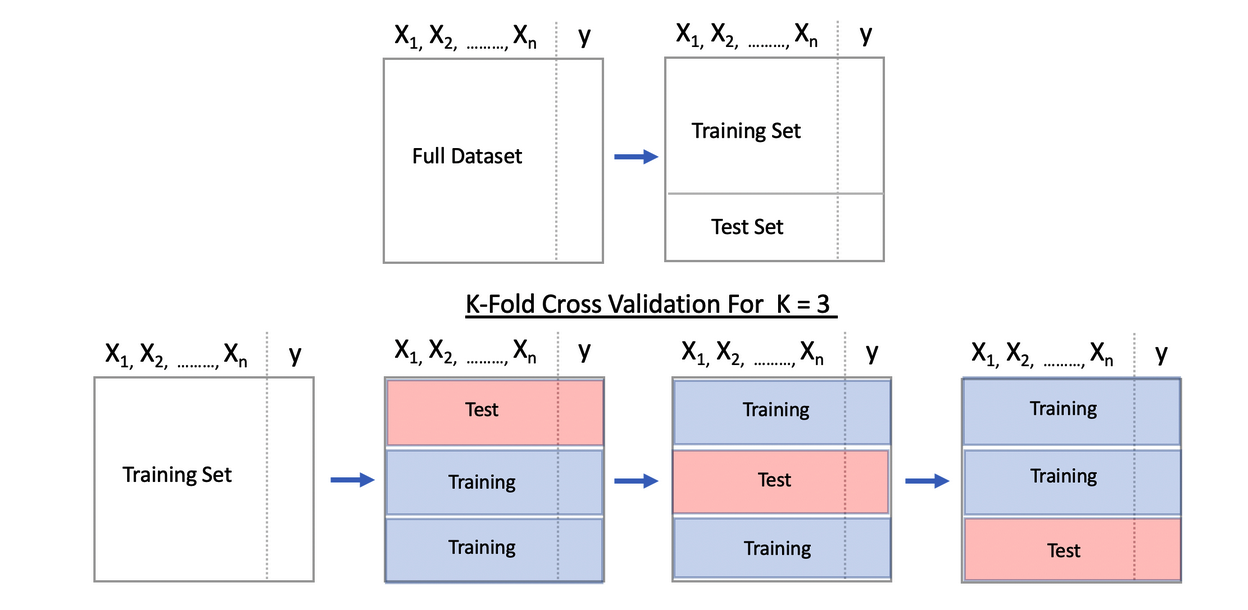
A way to combat this is to split the data into K number of groups aka “folds” - say 3. Then set up a loop that uses K-1 folds to train a model while using the “hold out” fold as the test data set. This means that each fold is used as a test set once.

It also means that each fold also gets used as part of a trained model thus we can offset the impact of any outliers found in the data set. Note, we are not trying to perfectly model the data, we just want to get a realistic idea as to how it might perform on unseen data.

The method works by giving us multiple estimates of out-of-sample error, rather than a single estimate. Let’s assume K is equal to 3 so we will partition our data into 3 individual “folds” which are basically equal in size. Then we’ll create a loop that does the following:

Combines 2 of the folds into a training data set  
Builds a model on the combined 2-folds data  
Applies the model to holdout fold  
Computes the AUC value and stores it

Each fold is simply a portion of the data. We’ll generate a list called “folds” that contains 3 elements each of which are 256 index elements corresponding to rows in pm. The way we did the sample insures that each row shows up only in one fold.



We could write our own function to not only create some number of folds but to also build a model, make predictions and store the accuracy for a given set of folds. Remember, each fold gets to be a “test” data set as part of the process.

cross\_fold <- function(numofolds = 4,df=pm) {  
   
 # Function to Do Cross fold validation  
   
 # Split the data into K folds (numofolds)  
   
 folds <- split(sample(1:nrow(df)),1:numofolds)   
   
 # We setup some blank lists to stash results  
 folddf <- list() # Contains folds  
 modl <- list() # Hold each of the K models  
 predl <- list() # Hold rach of the K predictions  
 auc <- list() # Hold the auc for a given model  
   
 # Create a formula that can be used across multiple  
 # iterations through the loop.   
   
 myform <- "diabetes ~ ."  
   
 for (ii in 1:length(folds)) {  
   
 # This list holds the actual model we create for each of the   
 # 10 folds  
   
 modl[[ii]] <- glm(formula = myform,   
 data = df[-folds[[ii]],],  
 family = "binomial")  
  
   
 # This list will contain / hold the models build on the fold  
   
 predl[[ii]] <- predict(modl[[ii]],  
 newdata=df[folds[[ii]],],  
 type="response")  
   
 # This list will hold the results of the AUC per iteration  
   
 pred <- ROCR::prediction(predl[[ii]],  
 df[folds[[ii]],]$diabetes)  
   
 roc <- ROCR::performance(pred,measure="auc")  
 auc[[ii]] <- roc@y.values[[1]]  
 }  
 return(unlist(auc))  
}

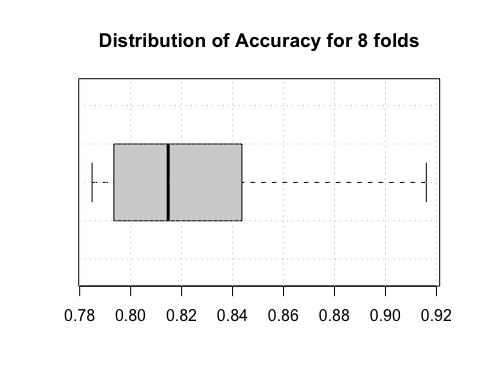
Running this is now quite simple. By default, this function will loop some number of times corresponding to the number of folds. During each iteration it will:

use glm to build a model on the training folds  
create a prediction object using the training fold  
compute the underlying AUC associated with the prediction  
store the AUC in a vector

At the end of the function, the vector containing the computed AUCs will be returned.

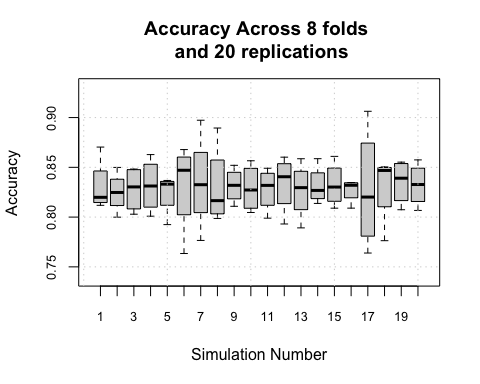
An advantage of this approach is that we can now look at a range of accuracy values (K of them) instead of just one. This gives us a better idea about how the model might perform on future data. Here we will look at accuracy across 8 folds:

numofolds <- 8  
boxplot(cross\_fold(numofolds = numofolds),  
 main=paste("Distribution of Accuracy for",numofolds,"folds"),  
 horizontal = TRUE)  
grid()



We can even replicate this to approximate something known as repeated cross validation:

numofolds <- 8  
numofrepl <- 20  
  
main <- paste("Accuracy Across",numofolds,"folds","\n and",numofrepl,"replications")  
reps <- replicate(numofrepl,cross\_fold())  
  
boxplot(reps,  
 main=main,cex.axis=.8,  
 xlab="Simulation Number",  
 ylim=c(min(reps)-0.025,max(reps)+0.025),  
 ylab="Accuracy")  
grid()



## Summary

So what have we done here. Well let’s recap:

1. Select some data and a method
2. Split the Data into training and testing data sets
3. Build a model on it and make some predictions on the test
4. Evaluate the model on the basis of a selected performance measure (e.g. accuracy, sensitivity, AUC)
5. Look at plots such as ROC

Then we might extend this by doing K-Fold cross validation at Step 2 to get a better idea as to how the model we built might apply to the test data. If we wanted to use another method, say random forests, we could but that would require us to understand the calling sequence for that method.

## A Different Approach

We could do all of the above in a more general manner by using something called **tidymodels**. There is also a package called **caret** which does similar things using base R. But since the author of **caret**, Max Kuhn, has now joined the **tidymodels** team it seems that **caret** is in maintenance mode only. In any case, to understand **tidymodels** one must first understand the **tidyverse**.

The **tidyverse** represents an extension on top of the base functions found in the default installation of R. Using **tidy** commands is an entirely optional activity and there is no need to use them except in pursuit of the concepts the tidyverse seeks to implement.

That sounds like circular logic but the idea is to pursue an approach that generalizes across multiple steps in data manipulation and modeling. Being able to easily reuse R objects (data, results, figures) is a big deal because it simplifies the construction and comparison of models which in turn facilitates reproducibility.

One would also like to be able to transfer knowledge acquired in using one package to other packages in the R environment. That said, in my view (and it’s only that), having a good knowledge of base R is important since you are likely to encounter “legacy” code which does not use things like the **tidyverse** or **tidymodels**.

Some introductory courses dive straight into the **tidyverse** and that is fine to an extent. It’s just that if you enter a work environment that has lots of older R code you will need to work to understand it. As this presentation assumes previous experience with R we will not be going over R basics.

## The tidyverse

The **tidymodels** package comprises a number of supporting packages and installing it is as simple as doing:

install.packages("tidymodels")

The **tidymodels** package relies on the **tidyverse** package which provides verbs that correspond to common data manipulation activities. The **tidyverse** package also provides the **ggplot2** package which is a premier visualization tool. Back to the verbs. Here are some basic examples using the **mtcars** data frame which is frequently used in R education due its simplicity and small size.

Let’s filter the data based on values assumed by one or more columns.

filter(pm, pregnant > 10 & age < 40)

## pregnant glucose pressure triceps insulin mass pedigree age diabetes  
## 1 11 138 76 0 0 33.2 0.420 35 neg  
## 2 12 151 70 40 271 41.8 0.742 38 pos  
## 3 14 175 62 30 0 33.6 0.212 38 pos  
## 4 11 85 74 0 0 30.1 0.300 35 neg  
## 5 13 104 72 0 0 31.2 0.465 38 pos  
## 6 13 153 88 37 140 40.6 1.174 39 neg

# Get all rows where diabetes is "pos" and mass is < 20  
filter(pm, diabetes == "pos" & mass < 20)

## pregnant glucose pressure triceps insulin mass pedigree age diabetes  
## 1 8 125 96 0 0 0 0.232 54 pos  
## 2 10 115 0 0 0 0 0.261 30 pos

We can even do aggregation using tidyverse functions. What is the average glucose per diabetes group?

summarize(group\_by(pm,diabetes),avg=mean(glucose))

## # A tibble: 2 × 2  
## diabetes avg  
## <fct> <dbl>  
## 1 neg 110.  
## 2 pos 141.

### Piping

One of the coolest functions provided by tidyverse is the pipe operator which allows us to feed the output of one command to the input of another in a left-to-right fashion. This concept is now new and was originally developed for use in the UNIX / LINUX operating system.

Base R uses composite functions in accordance of its heritage as a mathematical / statistical language. So structures like f(g(x)) or f(g(h(x))) are easily accommodated but prone to errors in typing due to the need to match parentheses.

# The composite function approach so Favored by mathematicians   
x <- 3.14  
log(cos(sin(tan(x))),base=10)

## [1] -5.508045e-07

Using the pipe structure this would look like the following which allows you to build the line a command at a time which makes it less error prone. You “read” the command from left to right where the output of one command becomes the input to another and so on.

x %>% tan() %>% sin() %>% cos() %>% log(base=10)

## [1] -5.508045e-07

# or even  
  
3.14 %>% tan() %>% sin() %>% cos() %>% log(base=10)

## [1] -5.508045e-07

# The pipe operator is %>%  
pm %>% filter(diabetes == "pos" & mass < 20)

## pregnant glucose pressure triceps insulin mass pedigree age diabetes  
## 1 8 125 96 0 0 0 0.232 54 pos  
## 2 10 115 0 0 0 0 0.261 30 pos

We can conveniently sort data. In this case we filter out cars with MPG greater than 15 and a weight less than 2 tons and then arrange the result in order of highest MPG to lowest.

pm %>%   
 filter(diabetes == "pos" & mass < 25) %>%   
 arrange(desc(glucose))

## pregnant glucose pressure triceps insulin mass pedigree age diabetes  
## 1 6 194 78 0 0 23.5 0.129 59 pos  
## 2 8 183 64 0 0 23.3 0.672 32 pos  
## 3 1 167 74 17 144 23.4 0.447 33 pos  
## 4 6 162 62 0 0 24.3 0.178 50 pos  
## 5 9 156 86 0 0 24.8 0.230 53 pos  
## 6 4 134 72 0 0 23.8 0.277 60 pos  
## 7 8 125 96 0 0 0.0 0.232 54 pos  
## 8 10 115 0 0 0 0.0 0.261 30 pos  
## 9 3 107 62 13 48 22.9 0.678 23 pos

As applied to our above grouping example we could de-couple the somewhat verbose Base R command line:

pm %>% group\_by(diabetes) %>% summarize(average\_glucose=mean(glucose))

## # A tibble: 2 × 2  
## diabetes average\_glucose  
## <fct> <dbl>  
## 1 neg 110.  
## 2 pos 141.

Note that the “%>%” operator allows us to **route** the output of a command to the input of another command. Here is a more simple example:

pm %>% filter(mass > 50)

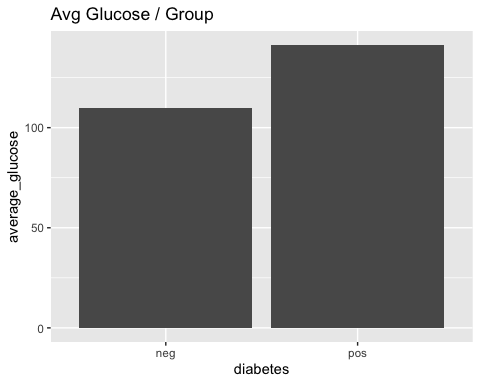
## pregnant glucose pressure triceps insulin mass pedigree age diabetes  
## 1 0 162 76 56 100 53.2 0.759 25 pos  
## 2 1 88 30 42 99 55.0 0.496 26 pos  
## 3 0 129 110 46 130 67.1 0.319 26 pos  
## 4 11 135 0 0 0 52.3 0.578 40 pos  
## 5 0 165 90 33 680 52.3 0.427 23 neg  
## 6 5 115 98 0 0 52.9 0.209 28 pos  
## 7 0 180 78 63 14 59.4 2.420 25 pos  
## 8 3 123 100 35 240 57.3 0.880 22 neg

# same as  
filter(pm, mass > 50)

## pregnant glucose pressure triceps insulin mass pedigree age diabetes  
## 1 0 162 76 56 100 53.2 0.759 25 pos  
## 2 1 88 30 42 99 55.0 0.496 26 pos  
## 3 0 129 110 46 130 67.1 0.319 26 pos  
## 4 11 135 0 0 0 52.3 0.578 40 pos  
## 5 0 165 90 33 680 52.3 0.427 23 neg  
## 6 5 115 98 0 0 52.9 0.209 28 pos  
## 7 0 180 78 63 14 59.4 2.420 25 pos  
## 8 3 123 100 35 240 57.3 0.880 22 neg

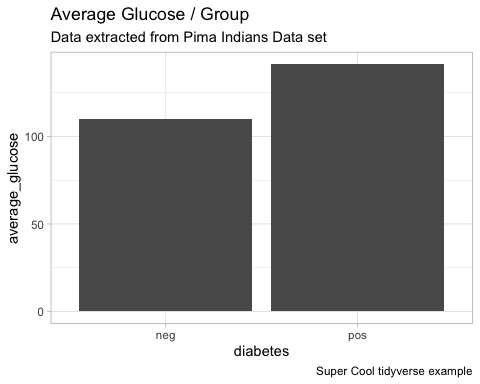
Back to our aggregation pipeline, we can “pipe” the result into the ggplot function to get a visualization. ggplot requires us to provide it tables or data frames. Actually it prefers “tibbles” which is a type of data frame but that’s getting off track a little bit.

pm %>%   
 group\_by(diabetes) %>%   
 summarize(average\_glucose=mean(glucose)) %>%  
 ggplot(aes(x=diabetes,y=average\_glucose)) +   
 geom\_bar(stat="identity") +   
 ggtitle("Avg Glucose / Group")



If we wanted to improve upon this we could, perhaps by making the transmission type into a factor with more intuitive labels. Here we will introduce another **tidyverse** function to **mutate** the **am** variable into a factor.

pm %>%   
 group\_by(diabetes) %>%   
 summarize(average\_glucose=mean(glucose)) %>%  
 ggplot(aes(x=diabetes,y=average\_glucose)) +   
 geom\_bar(stat="identity") +   
 theme\_light() +   
 labs(title = "Average Glucose / Group",   
 subtitle = "Data extracted from Pima Indians Data set",   
 caption = "Super Cool tidyverse example")



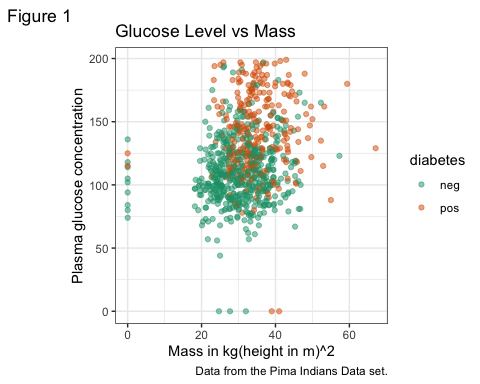
A distinct advantage of this approach is that we did not change in anyway the original data frame. Check it out:

head(pm)

## pregnant glucose pressure triceps insulin mass pedigree age diabetes  
## 1 6 148 72 35 0 33.6 0.627 50 pos  
## 2 1 85 66 29 0 26.6 0.351 31 neg  
## 3 8 183 64 0 0 23.3 0.672 32 pos  
## 4 1 89 66 23 94 28.1 0.167 21 neg  
## 5 0 137 40 35 168 43.1 2.288 33 pos  
## 6 5 116 74 0 0 25.6 0.201 30 neg

The piping allows for free-form experimentation without the need for having to save temporary and/pr incremental versions of data frames which is frequently the case when using Base R commands. Here is another example:

pm %>%  
 ggplot(aes(x=mass,y=glucose,color=diabetes)) +   
 geom\_point(alpha=0.5) +  
 labs(  
 title = "Glucose Level vs Mass",  
 caption = "Data from the Pima Indians Data set.",  
 tag = "Figure 1",  
 x = "Mass in kg(height in m)^2",  
 y = "Plasma glucose concentration",  
 colour = "diabetes"  
 ) + theme\_bw() + scale\_color\_brewer(palette="Dark2")



### Broom - sweeping things up

The tidyverse has a way to help clean up the results offered by Base R packages. Consider the following:

fit <- lm(glucose ~ mass + age,pm\_training)

Look at the output which is not directly in a convenient format.

fit

##   
## Call:  
## lm(formula = glucose ~ mass + age, data = pm\_training)  
##   
## Coefficients:  
## (Intercept) mass age   
## 68.6641 0.9534 0.6775

summary(fit)

##   
## Call:  
## lm(formula = glucose ~ mass + age, data = pm\_training)  
##   
## Residuals:  
## Min 1Q Median 3Q Max   
## -133.625 -19.214 -1.093 18.559 83.383   
##   
## Coefficients:  
## Estimate Std. Error t value Pr(>|t|)   
## (Intercept) 68.6641 6.6155 10.379 < 2e-16 \*\*\*  
## mass 0.9534 0.1663 5.732 1.66e-08 \*\*\*  
## age 0.6775 0.1097 6.173 1.32e-09 \*\*\*  
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
##   
## Residual standard error: 30.07 on 534 degrees of freedom  
## Multiple R-squared: 0.1161, Adjusted R-squared: 0.1128   
## F-statistic: 35.07 on 2 and 534 DF, p-value: 4.893e-15

Access coefficients only:

fit$coefficients

## (Intercept) mass age   
## 68.6640503 0.9534197 0.6774946

But with a tidy approach you get things in a data frame which is inarguably the most flexible data type in the R language. The tidy function “produces a tibble() where each row contains information about an important component of the model. For regression models, this often corresponds to regression coefficients”

tidy(fit)

## # A tibble: 3 × 5  
## term estimate std.error statistic p.value  
## <chr> <dbl> <dbl> <dbl> <dbl>  
## 1 (Intercept) 68.7 6.62 10.4 4.09e-23  
## 2 mass 0.953 0.166 5.73 1.66e- 8  
## 3 age 0.677 0.110 6.17 1.32e- 9

The glance function “returns a tibble with exactly one row of goodness of fitness measures and related statistics. This is useful to check for model misspecification and to compare many models”

glance(fit)

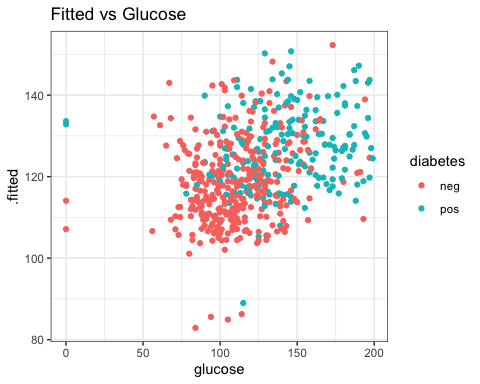
## # A tibble: 1 × 12  
## r.squared adj.r.squared sigma statistic p.value df logLik AIC BIC  
## <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 0.116 0.113 30.1 35.1 4.89e-15 2 -2588. 5184. 5201.  
## # … with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>

The **augment** function “adds columns to a dataset, containing information such as fitted values, residuals or cluster assignments. All columns added to a dataset have . prefix to prevent existing columns from being overwritten.”

augment(fit, pm\_training)[1:10,]

## # A tibble: 10 × 16  
## .rownames pregnant glucose pressure triceps insulin mass pedigree age  
## <chr> <int> <int> <int> <int> <int> <dbl> <dbl> <int>  
## 1 415 0 138 60 35 167 34.6 0.534 21  
## 2 463 8 74 70 40 49 35.3 0.705 39  
## 3 179 5 143 78 0 0 45 0.19 47  
## 4 526 3 87 60 18 0 21.8 0.444 21  
## 5 195 8 85 55 20 0 24.4 0.136 42  
## 6 118 5 78 48 0 0 33.7 0.654 25  
## 7 299 14 100 78 25 184 36.6 0.412 46  
## 8 229 4 197 70 39 744 36.7 2.33 31  
## 9 244 6 119 50 22 176 27.1 1.32 33  
## 10 14 1 189 60 23 846 30.1 0.398 59  
## # … with 7 more variables: diabetes <fct>, .fitted <dbl>, .resid <dbl>,  
## # .hat <dbl>, .sigma <dbl>, .cooksd <dbl>, .std.resid <dbl>

augment(fit, pm\_training) %>%   
 ggplot(aes(x=glucose,y=.fitted,color=diabetes)) +   
 geom\_point() + labs(title="Fitted vs Glucose") + theme\_bw()



## Tidymodels

The **tidymodels** package can be useful even if you are just experimenting since it provides a standard and uniform approach by which to work. In Base R, each modelling function, lm, glm, rpart, svm, will all have arguments specific to the respective command, which contributes to confusion when switching between models for purposes of comparison. This is where **tidymodels** can be very helpful.

One of the benefits of the package is that it provides front ends for the many methods available in R which includes Decision trees and extensions there upon. An advantage of Base R when building models is that many methods will convert the target variable to a label / nominal feature without you asking it to. This is a convenience though the tidyverse generally wants you do be explicit about this.

### Read in the data

url <- "https://raw.githubusercontent.com/steviep42/bios534\_spring\_2020/master/data/pima.csv"  
pm <- read.csv(url)  
  
pm <- pm %>% mutate(diabetes=factor(diabetes))  
skimr::skim(pm)

Data summary

|  |  |
| --- | --- |
| Name | pm |
| Number of rows | 768 |
| Number of columns | 9 |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
| Column type frequency: |  |
| factor | 1 |
| numeric | 8 |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
| Group variables | None |

**Variable type: factor**

| skim\_variable | n\_missing | complete\_rate | ordered | n\_unique | top\_counts |
| --- | --- | --- | --- | --- | --- |
| diabetes | 0 | 1 | FALSE | 2 | neg: 500, pos: 268 |

**Variable type: numeric**

| skim\_variable | n\_missing | complete\_rate | mean | sd | p0 | p25 | p50 | p75 | p100 | hist |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| pregnant | 0 | 1 | 3.85 | 3.37 | 0.00 | 1.00 | 3.00 | 6.00 | 17.00 | ▇▃▂▁▁ |
| glucose | 0 | 1 | 120.89 | 31.97 | 0.00 | 99.00 | 117.00 | 140.25 | 199.00 | ▁▁▇▆▂ |
| pressure | 0 | 1 | 69.11 | 19.36 | 0.00 | 62.00 | 72.00 | 80.00 | 122.00 | ▁▁▇▇▁ |
| triceps | 0 | 1 | 20.54 | 15.95 | 0.00 | 0.00 | 23.00 | 32.00 | 99.00 | ▇▇▂▁▁ |
| insulin | 0 | 1 | 79.80 | 115.24 | 0.00 | 0.00 | 30.50 | 127.25 | 846.00 | ▇▁▁▁▁ |
| mass | 0 | 1 | 31.99 | 7.88 | 0.00 | 27.30 | 32.00 | 36.60 | 67.10 | ▁▃▇▂▁ |
| pedigree | 0 | 1 | 0.47 | 0.33 | 0.08 | 0.24 | 0.37 | 0.63 | 2.42 | ▇▃▁▁▁ |
| age | 0 | 1 | 33.24 | 11.76 | 21.00 | 24.00 | 29.00 | 41.00 | 81.00 | ▇▃▁▁▁ |

### A First tidymodel

We’ll repeat our **glm** example though this time using tidymodels. It is important to understand that the **tidymodels** package does not attempt to replace or rewrite common and popular R packages for model assembly.

The package provides an standard and uniform interface that “sits on top” of various package functions. The advantage to the user is that you do not necessarily need to know all of the individual arguments to a specific command in order to use it though if you do then you can still leverage that information.

pm\_tidy\_glm <- logistic\_reg(mode = "classification") %>%  
 set\_engine("glm")

We can print out the contents of this object just by typing it although there isn’t much happening right now.

pm\_tidy\_glm

## Logistic Regression Model Specification (classification)  
##   
## Computational engine: glm

Tp get more info:

pm\_tidy\_glm %>% translate()

## Logistic Regression Model Specification (classification)  
##   
## Computational engine: glm   
##   
## Model fit template:  
## stats::glm(formula = missing\_arg(), data = missing\_arg(), weights = missing\_arg(),   
## family = stats::binomial)

### Fitting the Model

So now we can fit the model as part of the pipeline. Notice how tidymodels is designed to be totally congruent with the tidyverse.

pm\_tidy\_glm\_fit <- logistic\_reg(mode = "classification") %>%  
 set\_engine("glm") %>%  
 fit(diabetes ~ ., data = pm\_training)

So we get back a **parsnip** object which, using the copy from the **parsnip** package documentation is:

“The goal of **parsnip** is to provide a tidy, unified interface to models that can be used to try a range of models without getting bogged down in the syntactical minutiae of the underlying packages”.

pm\_tidy\_glm\_fit

## parsnip model object  
##   
##   
## Call: stats::glm(formula = diabetes ~ ., family = stats::binomial,   
## data = data)  
##   
## Coefficients:  
## (Intercept) pregnant glucose pressure triceps insulin   
## -8.405409 0.103471 0.035730 -0.012707 0.003563 -0.001710   
## mass pedigree age   
## 0.088735 0.696250 0.017015   
##   
## Degrees of Freedom: 536 Total (i.e. Null); 528 Residual  
## Null Deviance: 694.2   
## Residual Deviance: 509.8 AIC: 527.8

So this is where we could drop down into an inferential way of thinking and figure out if the coefficients in the model are statistically meaningful. But we’ll skip over that for now.

In terms of making a prediction, the **predict** function is what we call a generic function in R which understands how to do its work based on the type of object and data that it is being given. Remember that we built a model which returned an object. We can pass that to the **predict** function:

glance(pm\_tidy\_glm\_fit)

## # A tibble: 1 × 8  
## null.deviance df.null logLik AIC BIC deviance df.residual nobs  
## <dbl> <int> <dbl> <dbl> <dbl> <dbl> <int> <int>  
## 1 694. 536 -255. 528. 566. 510. 528 537

Make the actual prediction on the testing data. We also pass in the **type == “class”** argument since are dealing with classification and, in this case, wish to get labels back.

# Get predicted labels back  
test\_preds <- predict(pm\_tidy\_glm\_fit,pm\_testing,type="class")  
test\_preds[1:10,]

## # A tibble: 10 × 1  
## .pred\_class  
## <fct>   
## 1 pos   
## 2 pos   
## 3 neg   
## 4 pos   
## 5 pos   
## 6 neg   
## 7 neg   
## 8 neg   
## 9 pos   
## 10 neg

### Managing Predictions

By default we get back some predictions as to whether a subject is positive or negative for diabetes which we could then compare to the actual values. This would allow us to create a confusion matrix that would then permit the computation of various performance measures such as accuracy, sensitivity, specificity, etc.

In the tidymodels world, there are functions that will give this information to us without the need for the confusion matrix unless we wanted to print it.

The following shows us the predictions as part of the data frame. The tidyverse very much likes for things to be in a data frame format - actually something called a **tibble** which is a data frame optimized for use with the tidyverse.

pm\_tidy\_glm\_fit %>%  
 predict(pm\_testing) %>%  
 bind\_cols(pm\_testing) %>%  
 glimpse()

## Rows: 231  
## Columns: 10  
## $ .pred\_class <fct> pos, pos, neg, pos, pos, neg, neg, neg, pos, neg, pos, neg…  
## $ pregnant <int> 6, 8, 1, 2, 5, 0, 7, 8, 7, 1, 3, 10, 7, 7, 9, 0, 5, 1, 0, …  
## $ glucose <int> 148, 183, 89, 197, 166, 118, 107, 99, 147, 97, 158, 122, 1…  
## $ pressure <int> 72, 64, 66, 70, 72, 84, 74, 84, 76, 66, 76, 78, 84, 92, 11…  
## $ triceps <int> 35, 0, 23, 45, 19, 47, 0, 0, 0, 15, 36, 31, 0, 18, 24, 39,…  
## $ insulin <int> 0, 0, 94, 543, 175, 230, 0, 0, 0, 140, 245, 0, 0, 0, 240, …  
## $ mass <dbl> 33.6, 23.3, 28.1, 30.5, 25.8, 45.8, 29.6, 35.4, 39.4, 23.2…  
## $ pedigree <dbl> 0.627, 0.672, 0.167, 0.158, 0.587, 0.551, 0.254, 0.388, 0.…  
## $ age <int> 50, 32, 21, 53, 51, 31, 31, 50, 43, 22, 28, 45, 37, 48, 54…  
## $ diabetes <fct> pos, pos, neg, pos, pos, pos, pos, neg, pos, neg, pos, neg…

This sets the stage for evaluating the accuracy of the predictions resulting from the model:

my\_metrics <- metric\_set(accuracy,sensitivity,specificity, precision)  
pm\_tidy\_glm\_fit %>%  
 predict(pm\_testing) %>%  
 bind\_cols(pm\_testing) %>%  
 my\_metrics(truth=diabetes, estimate=.pred\_class,event\_level = "second")

## # A tibble: 4 × 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 accuracy binary 0.792  
## 2 sensitivity binary 0.556  
## 3 specificity binary 0.92   
## 4 precision binary 0.789

We could also derive all the metrics directly from the confusion matrix - if we wanted to

pm\_tidy\_glm\_fit %>%  
 predict(pm\_testing) %>%  
 bind\_cols(pm\_testing) %>%  
 conf\_mat(truth=diabetes, estimate=.pred\_class)

## Truth  
## Prediction neg pos  
## neg 138 36  
## pos 12 45

But we don’t need to do this as tidymodels has its own way of getting multiple metrics all at once or a subset thereof:

pm\_tidy\_glm\_fit %>%  
 predict(pm\_testing) %>%  
 bind\_cols(pm\_testing) %>%  
 conf\_mat(truth=diabetes, estimate=.pred\_class) %>%   
 summary(event\_level="second")

## # A tibble: 13 × 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 accuracy binary 0.792  
## 2 kap binary 0.510  
## 3 sens binary 0.556  
## 4 spec binary 0.92   
## 5 ppv binary 0.789  
## 6 npv binary 0.793  
## 7 mcc binary 0.526  
## 8 j\_index binary 0.476  
## 9 bal\_accuracy binary 0.738  
## 10 detection\_prevalence binary 0.247  
## 11 precision binary 0.789  
## 12 recall binary 0.556  
## 13 f\_meas binary 0.652

Or you can specify just the ones you want without having to even generate the confusion matrix

# Note that sensitivity is also known as recall  
my\_metrics <- metric\_set(accuracy, sensitivity, specificity, precision)  
pm\_tidy\_glm\_fit %>%  
 predict(pm\_testing) %>%  
 bind\_cols(pm\_testing) %>%  
 my\_metrics(truth=diabetes, estimate=.pred\_class, event\_level = "second")

## # A tibble: 4 × 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 accuracy binary 0.792  
## 2 sensitivity binary 0.556  
## 3 specificity binary 0.92   
## 4 precision binary 0.789

The flexibility of tidymodels can be confusing although if you first determine what your metric(s) for success will be BEFORE building a model then the job becomes easier.

### ROC Curves

And what about drawing ROC curves? The tidymodels package has that covered. We just have to remember to get the probabilities associated with the prediction instead of the labels. Getting the probabilities is as easy as providing the keyword “probabilities: to the”type” argument.

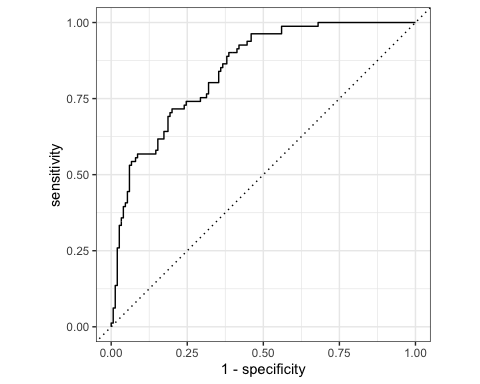
pm\_tidy\_glm\_fit %>%  
 predict(pm\_testing,type="prob")

## # A tibble: 231 × 2  
## .pred\_neg .pred\_pos  
## <dbl> <dbl>  
## 1 0.272 0.728   
## 2 0.227 0.773   
## 3 0.956 0.0443  
## 4 0.289 0.711   
## 5 0.386 0.614   
## 6 0.624 0.376   
## 7 0.813 0.187   
## 8 0.700 0.300   
## 9 0.267 0.733   
## 10 0.956 0.0436  
## # … with 221 more rows

Using this information we can now:

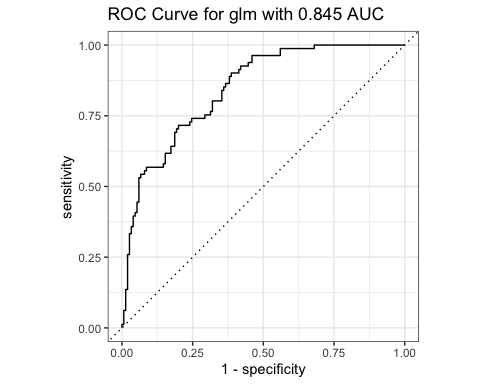
1. Predict using probabilities using the test data
2. Binding that info to the test data
3. Passing #2 into the **roc\_curve** function (specify the correct “event level”)
4. Pass #3 into the **autoplot** function which figures out the appropriate graph

pm\_tidy\_glm\_fit %>%  
 predict(pm\_testing,type="prob") %>% # set type="prob"  
 bind\_cols(pm\_testing) %>%  
 roc\_curve(truth=diabetes,estimate=.pred\_pos,event\_level="second") %>%  
 autoplot()



We can improve upon the default plot. First, we’ll get the auc value to use in our labels. Then we’ll use this information along with various ggplot functions.

# Get the AUC by using the \*\*roc\_auc\*\* function  
pm\_tidy\_glm\_roc\_auc <- pm\_tidy\_glm\_fit %>%  
 predict(pm\_testing,type="prob") %>%  
 bind\_cols(pm\_testing) %>%  
 roc\_auc(truth=diabetes,estimate=.pred\_pos,event\_level="second")  
  
# Put up an annotated plot using ggplot directly for more flexibility  
# in labeling  
  
pm\_tidy\_glm\_fit %>%  
 predict(pm\_testing,type="prob") %>%  
 bind\_cols(pm\_testing) %>%  
 roc\_curve(truth=diabetes,estimate=.pred\_pos,event\_level="second") %>%  
 ggplot(aes(x=1 - specificity, y = sensitivity)) +   
 geom\_path() +   
 geom\_abline(lty=3) +  
 coord\_equal() +   
 labs(title=paste("ROC Curve for glm with",round(pm\_tidy\_glm\_roc\_auc$.estimate,3),"AUC")) +  
 theme\_bw()



### Engine Selection

Let’s take a step back into the definition of the model to learn a little more about what is happening at a basic level. So remember that we wanted to do logistic regression which can be done as follows:

logistic\_reg() %>% translate()

## Logistic Regression Model Specification (classification)  
##   
## Computational engine: glm   
##   
## Model fit template:  
## stats::glm(formula = missing\_arg(), data = missing\_arg(), weights = missing\_arg(),   
## family = stats::binomial)

By default, the above function will select a computational engine of “glm” although we can choose from a number of engines which correspond to different underlying regression methods including the following:

show\_engines("logistic\_reg")

## # A tibble: 7 × 2  
## engine mode   
## <chr> <chr>   
## 1 glm classification  
## 2 glmnet classification  
## 3 LiblineaR classification  
## 4 spark classification  
## 5 keras classification  
## 6 stan classification  
## 7 brulee classification

We can choose a different underlying method if we so desire:

logistic\_reg(mode = "classification",penalty=1) %>%   
 set\_engine("glmnet") %>%   
 translate()

## Logistic Regression Model Specification (classification)  
##   
## Main Arguments:  
## penalty = 1  
##   
## Computational engine: glmnet   
##   
## Model fit template:  
## glmnet::glmnet(x = missing\_arg(), y = missing\_arg(), weights = missing\_arg(),   
## family = "binomial")

This could be economized to:

logistic\_reg(mode = "classification",engine="glmnet",penalty=1) %>%   
 translate()

## Logistic Regression Model Specification (classification)  
##   
## Main Arguments:  
## penalty = 1  
##   
## Computational engine: glmnet   
##   
## Model fit template:  
## glmnet::glmnet(x = missing\_arg(), y = missing\_arg(), weights = missing\_arg(),   
## family = "binomial")

## Using Another Method

Jumping over to another method is easy. Let’s build a random forest using the **randomForest package**. While we provide some arguments specific to the randomForest function we still have a lot of similarity in our previous approach. Note, we do not have to separate the creation of the model from the prediction phase (unless we want to).

# We can get multiple metrics at once.  
my\_metrics <- metric\_set(accuracy, sensitivity, specificity, precision)  
  
# Let's get  
pm\_tidy\_preds\_rf <- rand\_forest(trees = 100, mode = "classification") %>%  
 set\_engine("randomForest") %>%  
 fit(diabetes ~ ., data = pm\_training) %>%  
 predict(pm\_testing) %>%  
 bind\_cols(pm\_testing) %>%   
 my\_metrics(truth=diabetes, estimate=.pred\_class, event\_level = "second")   
  
pm\_tidy\_preds\_rf

## # A tibble: 4 × 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 accuracy binary 0.766  
## 2 sensitivity binary 0.617  
## 3 specificity binary 0.847  
## 4 precision binary 0.685

## Data Management

So another nice thing about the tidymodels package is that it provides a way to create and manage the training and test pair. We implemented our own code to do this in the earlier example. While that is okay to do, the package provides ways to do this via the **initial\_split** function.

### Partitioning The Data

set.seed(123)  
pm\_split <- initial\_split(pm,prop=0.7)  
pm\_split

## <Training/Testing/Total>  
## <537/231/768>

This gives us a self-contained object with all the information necessary to get the training and test data on demand. The following will access the training data

training(pm\_split)[1:10,]

## pregnant glucose pressure triceps insulin mass pedigree age diabetes  
## 415 0 138 60 35 167 34.6 0.534 21 pos  
## 463 8 74 70 40 49 35.3 0.705 39 neg  
## 179 5 143 78 0 0 45.0 0.190 47 neg  
## 526 3 87 60 18 0 21.8 0.444 21 neg  
## 195 8 85 55 20 0 24.4 0.136 42 neg  
## 118 5 78 48 0 0 33.7 0.654 25 neg  
## 299 14 100 78 25 184 36.6 0.412 46 pos  
## 229 4 197 70 39 744 36.7 2.329 31 neg  
## 244 6 119 50 22 176 27.1 1.318 33 pos  
## 14 1 189 60 23 846 30.1 0.398 59 pos

The following will get the test data

testing(pm\_split)[1:10,]

## pregnant glucose pressure triceps insulin mass pedigree age diabetes  
## 1 6 148 72 35 0 33.6 0.627 50 pos  
## 3 8 183 64 0 0 23.3 0.672 32 pos  
## 4 1 89 66 23 94 28.1 0.167 21 neg  
## 9 2 197 70 45 543 30.5 0.158 53 pos  
## 15 5 166 72 19 175 25.8 0.587 51 pos  
## 17 0 118 84 47 230 45.8 0.551 31 pos  
## 18 7 107 74 0 0 29.6 0.254 31 pos  
## 22 8 99 84 0 0 35.4 0.388 50 neg  
## 27 7 147 76 0 0 39.4 0.257 43 pos  
## 28 1 97 66 15 140 23.2 0.487 22 neg

### Using the Split Data

Let’s repeat our glm model on the new training data. We don’t expect much difference. This is just illustrating how we can take advantage of **tidymodels** data splitting functions to help us.

# Get the Training data  
pm\_training <- training(pm\_split)  
  
# Define a model  
pm\_tidy\_glm\_fit <- logistic\_reg(mode = "classification") %>%  
 set\_engine("glm") %>%  
 fit(diabetes ~ ., data = pm\_training)

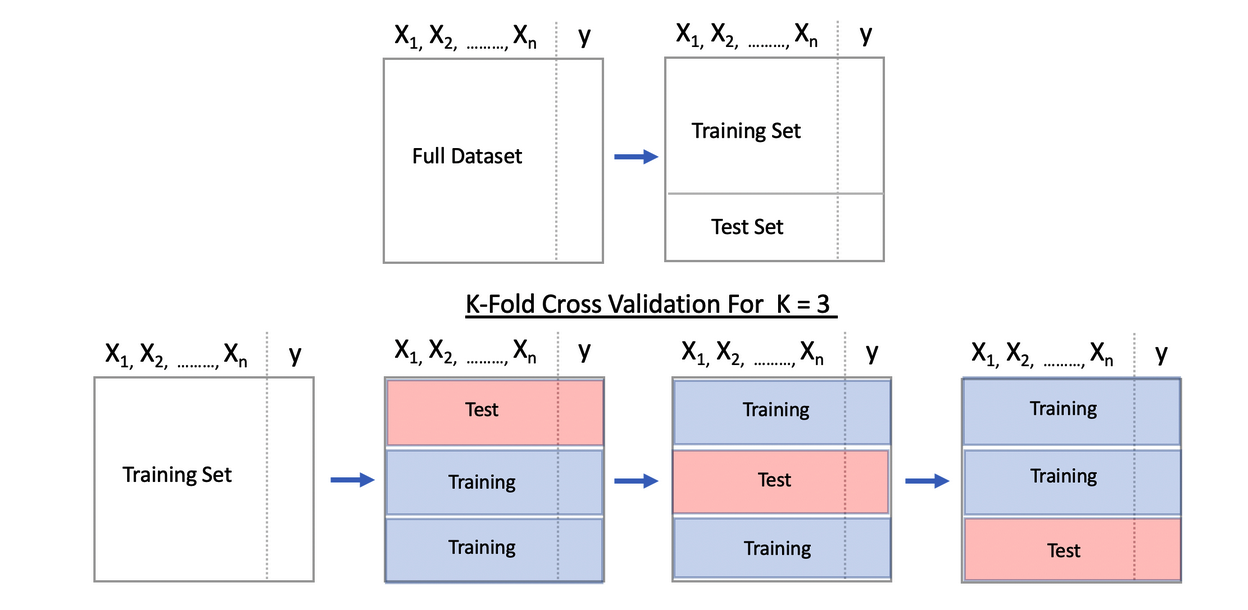
Now let’s do some predictions using the test data

# Get the Testing data  
pm\_testing <- testing(pm\_split)  
  
# Do predictions on the testing data  
pm\_tidy\_glm\_fit %>%  
 predict(pm\_testing) %>%  
 bind\_cols(pm\_testing) %>%  
 accuracy(truth = diabetes, estimate = .pred\_class)

## # A tibble: 1 × 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 accuracy binary 0.792

## Cross Fold Validation

Just as with Base R, we can setup our data to produce a set of K folds to help us better estimate model performance on unseen data.



We could write our own code to do this but we do not need to as tidymodels has the **resamples** package to help us. Here we can get 6 folds from the training data:

pm\_folds <- vfold\_cv(pm\_training, v = 6)

So what does this structure look like?

glimpse(pm\_folds)

## Rows: 6  
## Columns: 2  
## $ splits <list> [<vfold\_split[447 x 90 x 537 x 9]>], [<vfold\_split[447 x 90 x …  
## $ id <chr> "Fold1", "Fold2", "Fold3", "Fold4", "Fold5", "Fold6"

pm\_folds$splits[[1]]

## <Analysis/Assess/Total>  
## <447/90/537>

pm\_folds$splits[[1]][["data"]][1:5,]

## pregnant glucose pressure triceps insulin mass pedigree age diabetes  
## 415 0 138 60 35 167 34.6 0.534 21 pos  
## 463 8 74 70 40 49 35.3 0.705 39 neg  
## 179 5 143 78 0 0 45.0 0.190 47 neg  
## 526 3 87 60 18 0 21.8 0.444 21 neg  
## 195 8 85 55 20 0 24.4 0.136 42 neg

Let’s repeat our exercise with fitting a model using the folds. While we are at it, we will collect the mean accuracy and roc\_auc for this experiment. Understand that we are computing an average accuracy and AUC across the folds. Remember that in cross fold validation we are chopping up the training data into some number of folds - in this case 6. So each time we wish to build and evaluate a model, one of the six folds is held out as testing data while the remaining 5 are used to train the data. It’s like a for loop although we don’t have to program that ourselves like we did previously.

logistic\_reg(mode = "classification") %>%  
 set\_engine("glm") %>%  
 fit\_resamples(diabetes ~ .,pm\_folds) %>%   
 collect\_metrics() %>%   
 select(.metric,mean,n)

## # A tibble: 2 × 3  
## .metric mean n  
## <chr> <dbl> <int>  
## 1 accuracy 0.767 6  
## 2 roc\_auc 0.823 6

## Workflows

It is possible to organize a number of steps into a workflow as a form of abstraction. From the manual:

A workflow is an object that can bundle together your pre-processing, modeling, and post-processing requests. For example, if you have a recipe and parsnip model, these can be combined into a workflow. The advantages are:

You don’t have to keep track of separate objects in your workspace.  
  
The recipe prepping and model fitting can be executed using a single call to fit().

So lets define a model as before. You’ve seen this before so this should not be new to you.

pm\_tidy\_glm <- logistic\_reg(mode = "classification") %>%  
 set\_engine("glm")

Next we will create a workflow to contain it. We can also add in other things than just models but let’s keep it simple for now. Here we will add a formula and a model into a workflow.

pm\_workflow <- workflow() %>%  
 add\_formula(diabetes~.) %>%   
 add\_model(pm\_tidy\_glm)

Check it out:

pm\_workflow

## ══ Workflow ════════════════════════════════════════════════════════════════════  
## Preprocessor: Formula  
## Model: logistic\_reg()  
##   
## ── Preprocessor ────────────────────────────────────────────────────────────────  
## diabetes ~ .  
##   
## ── Model ───────────────────────────────────────────────────────────────────────  
## Logistic Regression Model Specification (classification)  
##   
## Computational engine: glm

Before we do anything with it, I wanted to point out that you could update specific components of an existing workflow. This is helpful if our workflow definition has lots of individual components to it. In this case we do not have that many but this is a good time to point out the advantage of being able to perform an update. So let’s change the existing formula to something new.

update\_formula(pm\_workflow,diabetes~mass)

## ══ Workflow ════════════════════════════════════════════════════════════════════  
## Preprocessor: Formula  
## Model: logistic\_reg()  
##   
## ── Preprocessor ────────────────────────────────────────────────────────────────  
## diabetes ~ mass  
##   
## ── Model ───────────────────────────────────────────────────────────────────────  
## Logistic Regression Model Specification (classification)  
##   
## Computational engine: glm

So we can execute the workflow by “fitting” it.

pm\_workflow\_fit <- fit(pm\_workflow,data=pm\_training)

And then predictions can be made:

predict(pm\_workflow\_fit,pm\_testing) %>%  
 bind\_cols(new\_data=pm\_testing) %>%  
 my\_metrics(truth=diabetes, estimate=.pred\_class,event\_level="second")

## # A tibble: 4 × 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 accuracy binary 0.792  
## 2 sensitivity binary 0.556  
## 3 specificity binary 0.92   
## 4 precision binary 0.789

This could be done in one go if desired. Those facile with tidy commands might prefer this although you are encouraged to keep things simple and readable. You can break up the pipeline into segments wherein you save information along the way if you want others to be able to easily follow your steps. This is up to you but sometimes having long pipelines can be hard to follow.

workflow() %>%  
 add\_formula(diabetes~.) %>%   
 add\_model(pm\_tidy\_glm) %>%   
 fit(data=pm\_training) %>%  
 predict(new\_data=pm\_testing) %>%  
 bind\_cols(pm\_testing) %>%  
 my\_metrics(truth=diabetes, estimate=.pred\_class,event\_level="second")

## # A tibble: 4 × 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 accuracy binary 0.792  
## 2 sensitivity binary 0.556  
## 3 specificity binary 0.92   
## 4 precision binary 0.789

What else can we do with workflows ? What about cross validation ? Yes! Let’s define everything from scratch which should serve as a review

# Define the model  
pm\_tidy\_glm <- logistic\_reg(mode = "classification") %>%  
 set\_engine("glm")   
  
# Define the workflow  
pm\_workflow <- workflow() %>%  
 add\_formula(diabetes~.) %>%   
 add\_model(pm\_tidy\_glm)

So if we wanted to train the model on a set of folds how would we do that? Where would we specify it in the workflow? First, we can create the folds using the following. You’ve seen this before.

# Create 6 folds  
pm\_folds <- vfold\_cv(training(pm\_split), v = 6)  
glimpse(pm\_folds)

## Rows: 6  
## Columns: 2  
## $ splits <list> [<vfold\_split[447 x 90 x 537 x 9]>], [<vfold\_split[447 x 90 x …  
## $ id <chr> "Fold1", "Fold2", "Fold3", "Fold4", "Fold5", "Fold6"

Back to the question of how to get these folds to be used in the training / fitting process?

# We can use the fit\_resamples function to do a fit that uses the folds  
log\_fit <- fit\_resamples(pm\_workflow,   
 pm\_folds,   
 metrics = my\_metrics,  
 control = control\_resamples(event\_level = "second")) %>%  
 collect\_metrics()

What do we get for our trouble ?

log\_fit

## # A tibble: 4 × 6  
## .metric .estimator mean n std\_err .config   
## <chr> <chr> <dbl> <int> <dbl> <chr>   
## 1 accuracy binary 0.763 6 0.0199 Preprocessor1\_Model1  
## 2 precision binary 0.707 6 0.0261 Preprocessor1\_Model1  
## 3 sensitivity binary 0.559 6 0.0319 Preprocessor1\_Model1  
## 4 specificity binary 0.877 6 0.0114 Preprocessor1\_Model1

We see the average metric values across all folds which in this case are 6 in number. Keep in mind that we are simply trying to get an idea about how a trained model might perform on unseen data so if the reported values aren’t impressive that does not mean the model lacks value.

## Cooking Up A Recipe

What else can a workflow contain ? What about including some steps that transform data such as removing highly correlated variables? Or, maybe scaling input data since some ML methods expect (or require) require that prior to use. We might also want to do one-hot encoding on categories data to better represent the different levels of a factor.

The way this works is that we:

1. Create a recipe
2. Prep the recipe
3. Bake the prepped recipe using some data

There are two different types of operations that can be sequentially added to a recipe. Steps can include common operations like scaling a variable, creating dummy variables or interactions and so on. More computationally complex actions such as dimension reduction or imputation can also be specified.

Checks are operations that conduct specific tests of the data. When the test is satisfied, the data are returned without issue or modification. Otherwise, any error is thrown.

Here we define a recipe that indicates what it is we are predicting. We could more granularly define the types of features we have which might mean designating factors.

recipe(diabetes~.,data=training(pm\_split)) %>% summary()

## # A tibble: 9 × 4  
## variable type role source   
## <chr> <chr> <chr> <chr>   
## 1 pregnant numeric predictor original  
## 2 glucose numeric predictor original  
## 3 pressure numeric predictor original  
## 4 triceps numeric predictor original  
## 5 insulin numeric predictor original  
## 6 mass numeric predictor original  
## 7 pedigree numeric predictor original  
## 8 age numeric predictor original  
## 9 diabetes nominal outcome original

So now, let’s create a recipe that centers and scales all variables except the outcome variable. There are different ways to do this but here is once way. Note that we “prep” the recipe by calling the **prep** function. Check the reference for **recipe** at <https://recipes.tidymodels.org/reference/index.html> There are many processing options which are possible.

pm\_recipe\_prepped <- recipe(diabetes ~ .,data=training(pm\_split)) %>%  
 step\_center(all\_numeric(), -all\_outcomes()) %>%  
 step\_scale(all\_numeric(), -all\_outcomes()) %>%  
 prep()  
  
pm\_recipe\_prepped

## Recipe  
##   
## Inputs:  
##   
## role #variables  
## outcome 1  
## predictor 8  
##   
## Training data contained 537 data points and no missing data.  
##   
## Operations:  
##   
## Centering for pregnant, glucose, pressure, triceps, insulin, ... [trained]  
## Scaling for pregnant, glucose, pressure, triceps, insulin, ... [trained]

But the recipe itself is just that - a document of intention for something to be prepped (which we just did) and later “bake” into usable data.

pm\_training\_baked <- pm\_recipe\_prepped %>%  
 bake(training(pm\_split))   
  
glimpse(pm\_training\_baked)

## Rows: 537  
## Columns: 9  
## $ pregnant <dbl> -1.12949397, 1.23517762, 0.34842577, -0.24274213, 1.23517762,…  
## $ glucose <dbl> 0.51182436, -1.49323429, 0.66846956, -1.08595675, -1.14861483…  
## $ pressure <dbl> -0.44669796, 0.07161838, 0.48627146, -0.44669796, -0.70585613…  
## $ triceps <dbl> 0.87650581, 1.18735221, -1.29941900, -0.18037195, -0.05603339…  
## $ insulin <dbl> 0.7418514, -0.2693811, -0.6892998, -0.6892998, -0.6892998, -0…  
## $ mass <dbl> 0.32844867, 0.41809712, 1.66036859, -1.31083739, -0.97785741,…  
## $ pedigree <dbl> 0.1571759, 0.6528145, -0.8398981, -0.1036865, -0.9964156, 0.5…  
## $ age <dbl> -1.02629240, 0.49473525, 1.17074754, -1.02629240, 0.74823986,…  
## $ diabetes <fct> pos, neg, neg, neg, neg, neg, pos, neg, pos, pos, neg, pos, n…

Similarly, if we want the testing data to have the same pre-processing we need to bake it using the prepped recipe. Note that the scaling is determined by the training to avoid data leakage.

pm\_testing\_baked <- pm\_recipe\_prepped %>%  
 bake(testing(pm\_split))   
  
pm\_testing\_baked

## # A tibble: 231 × 9  
## pregnant glucose pressure triceps insulin mass pedigree age diabetes  
## <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fct>   
## 1 0.644 0.825 0.175 0.877 -0.689 0.200 0.427 1.42 pos   
## 2 1.24 1.92 -0.239 -1.30 -0.689 -1.12 0.557 -0.0968 pos   
## 3 -0.834 -1.02 -0.136 0.130 0.116 -0.504 -0.907 -1.03 neg   
## 4 -0.538 2.36 0.0716 1.50 3.96 -0.197 -0.933 1.68 pos   
## 5 0.348 1.39 0.175 -0.118 0.810 -0.799 0.311 1.51 pos   
## 6 -1.13 -0.115 0.797 1.62 1.28 1.76 0.206 -0.181 pos   
## 7 0.940 -0.459 0.279 -1.30 -0.689 -0.312 -0.654 -0.181 pos   
## 8 1.24 -0.710 0.797 -1.30 -0.689 0.431 -0.266 1.42 neg   
## 9 0.940 0.794 0.383 -1.30 -0.689 0.943 -0.646 0.833 pos   
## 10 -0.834 -0.773 -0.136 -0.367 0.510 -1.13 0.0209 -0.942 neg   
## # … with 221 more rows

We can consolidate the creation of a model along with the recipe used to pre-process data into a **workflow**. To review, here is our recipe minus the prep and a formula definition.

First, here is our recipe. We are just scratching the surface here. We could also do things like removing highly correlated variables and/or reduce the dimensionality of the data via PCA using **step** functions. For purposes of explanation, we will keep it simple and just do the centering and scaling.

pm\_recipe <- training(pm\_split) %>%  
 recipe(diabetes ~ .) %>%  
 step\_center(all\_numeric(), -all\_outcomes()) %>%  
 step\_scale(all\_numeric(), -all\_outcomes()) %>%  
 prep()

Here is the model definition

pm\_tidy\_glm <- logistic\_reg(mode = "classification") %>%  
 set\_engine("glm")

Next we will create workflow to contain it. We can also add in other things than just models but let’s keep it simple for now.

pm\_workflow <- workflow() %>%  
 add\_recipe(pm\_recipe) %>%  
 add\_model(pm\_tidy\_glm)

pm\_workflow

## ══ Workflow ════════════════════════════════════════════════════════════════════  
## Preprocessor: Recipe  
## Model: logistic\_reg()  
##   
## ── Preprocessor ────────────────────────────────────────────────────────────────  
## 2 Recipe Steps  
##   
## • step\_center()  
## • step\_scale()  
##   
## ── Model ───────────────────────────────────────────────────────────────────────  
## Logistic Regression Model Specification (classification)  
##   
## Computational engine: glm

Then we fit the workflow which brings it all together

pm\_workflow\_fit <- fit(pm\_workflow,data=pm\_training)

Make predictions:

predict(pm\_workflow\_fit,pm\_testing) %>%  
 bind\_cols(pm\_testing) %>%  
 my\_metrics(truth=diabetes, estimate=.pred\_class,event\_level = "second")

## # A tibble: 4 × 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 accuracy binary 0.792  
## 2 sensitivity binary 0.556  
## 3 specificity binary 0.92   
## 4 precision binary 0.789

We could actually substitute a new model into the existing workflow. Let’s define a new model which in this case uses a **randomForest** method.

pm\_tidy\_rf <- rand\_forest(mtry = 5, trees = 200) %>%   
 set\_mode("classification") %>%   
 set\_engine("randomForest")

Now we can update the workflow with the new model and complete the fit and predict in one go. This example is a little small but imagine if our workflow had a lengthy recipe and model definitions. We don’t have to redefine it, we just update an existing one.

pm\_workflow %>%  
 update\_model(pm\_tidy\_rf) %>%  
 fit(data=pm\_training) %>%  
 predict(new\_data=pm\_testing) %>%  
 bind\_cols(pm\_testing) %>%  
 my\_metrics(truth=diabetes, estimate=.pred\_class,event\_level = "second")

## # A tibble: 4 × 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 accuracy binary 0.766  
## 2 sensitivity binary 0.593  
## 3 specificity binary 0.86   
## 4 precision binary 0.696

## Tuning and Optimizing

Many times the models we wish to create have a number of parameters that can be set to influence the model. In general, the fewer of these we have to consider the easier it is to build the model although if we have a good understanding of a given algorithm well then we can achieve better results. Consider, for example, that random forest methods involve building a number of trees to offset the over training which is possible from any single tree.

This means that the number of desired trees can be set *before* the method is implemented. In fact, the model can be repeated using different values of the number of desired trees, to see if an optimal result can be achieved.

We call this (hyper)parameter tuning which is something that tidymodels will do for us. In effect we are optimizing a designated performance metric (e.g. accuracy, precision, recall) using parameters. Random Forests have other hyperparameters which could also be set such as the minimum number of cases that must reside in a node. Hyperparameters can be tuned together or separately.

In the example below we will create a model specification that indicates our desired method (random forest) along with the hyperparameters we wish to tune as part of our fit process.

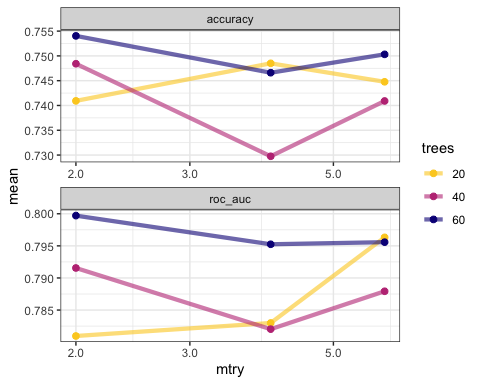
tune\_spec <-   
 rand\_forest(mode = "classification",  
 mtry = tune(),  
 trees = tune()) %>%  
 set\_engine("randomForest")

So we update the workflow to use the tuning specification

man\_tune <- pm\_workflow %>%   
 update\_model(tune\_spec) %>%  
 tune\_grid(resamples = pm\_folds,  
 grid = expand.grid(  
 mtry = c(2,4,6),  
 trees = c(20,40,60)  
 ))  
man\_tune %>% collect\_metrics()

## # A tibble: 18 × 8  
## mtry trees .metric .estimator mean n std\_err .config   
## <dbl> <dbl> <chr> <chr> <dbl> <int> <dbl> <chr>   
## 1 2 20 accuracy binary 0.741 6 0.0242 Preprocessor1\_Model1  
## 2 2 20 roc\_auc binary 0.781 6 0.0242 Preprocessor1\_Model1  
## 3 4 20 accuracy binary 0.749 6 0.00995 Preprocessor1\_Model2  
## 4 4 20 roc\_auc binary 0.783 6 0.0225 Preprocessor1\_Model2  
## 5 6 20 accuracy binary 0.745 6 0.00940 Preprocessor1\_Model3  
## 6 6 20 roc\_auc binary 0.796 6 0.0187 Preprocessor1\_Model3  
## 7 2 40 accuracy binary 0.748 6 0.0250 Preprocessor1\_Model4  
## 8 2 40 roc\_auc binary 0.792 6 0.0225 Preprocessor1\_Model4  
## 9 4 40 accuracy binary 0.730 6 0.0222 Preprocessor1\_Model5  
## 10 4 40 roc\_auc binary 0.782 6 0.0271 Preprocessor1\_Model5  
## 11 6 40 accuracy binary 0.741 6 0.0217 Preprocessor1\_Model6  
## 12 6 40 roc\_auc binary 0.788 6 0.0230 Preprocessor1\_Model6  
## 13 2 60 accuracy binary 0.754 6 0.0158 Preprocessor1\_Model7  
## 14 2 60 roc\_auc binary 0.800 6 0.0234 Preprocessor1\_Model7  
## 15 4 60 accuracy binary 0.747 6 0.0133 Preprocessor1\_Model8  
## 16 4 60 roc\_auc binary 0.795 6 0.0197 Preprocessor1\_Model8  
## 17 6 60 accuracy binary 0.750 6 0.0177 Preprocessor1\_Model9  
## 18 6 60 roc\_auc binary 0.796 6 0.0219 Preprocessor1\_Model9

man\_tune %>%  
 collect\_metrics() %>%  
 mutate(trees = factor(trees)) %>%  
 ggplot(aes(mtry, mean, color = trees)) +  
 geom\_line(size = 1.5, alpha = 0.6) +  
 geom\_point(size = 2) +  
 facet\_wrap(~ .metric, scales = "free", nrow = 2) +  
 scale\_x\_log10(labels = scales::label\_number()) +  
 scale\_color\_viridis\_d(option = "plasma", begin = .9, end = 0) +  
 theme\_bw()



man\_tune %>% show\_best()

## Warning: No value of `metric` was given; metric 'roc\_auc' will be used.

## # A tibble: 5 × 8  
## mtry trees .metric .estimator mean n std\_err .config   
## <dbl> <dbl> <chr> <chr> <dbl> <int> <dbl> <chr>   
## 1 2 60 roc\_auc binary 0.800 6 0.0234 Preprocessor1\_Model7  
## 2 6 20 roc\_auc binary 0.796 6 0.0187 Preprocessor1\_Model3  
## 3 6 60 roc\_auc binary 0.796 6 0.0219 Preprocessor1\_Model9  
## 4 4 60 roc\_auc binary 0.795 6 0.0197 Preprocessor1\_Model8  
## 5 2 40 roc\_auc binary 0.792 6 0.0225 Preprocessor1\_Model4

man\_final <- finalize\_workflow(pm\_workflow,  
 select\_best(man\_tune,metric="roc\_auc")) %>%   
 fit(pm\_training\_baked)  
  
man\_final

## ══ Workflow [trained] ══════════════════════════════════════════════════════════  
## Preprocessor: Recipe  
## Model: logistic\_reg()  
##   
## ── Preprocessor ────────────────────────────────────────────────────────────────  
## 2 Recipe Steps  
##   
## • step\_center()  
## • step\_scale()  
##   
## ── Model ───────────────────────────────────────────────────────────────────────  
##   
## Call: stats::glm(formula = ..y ~ ., family = stats::binomial, data = data)  
##   
## Coefficients:  
## (Intercept) pregnant glucose pressure triceps insulin   
## -0.85736 0.35006 1.14048 -0.24517 0.05731 -0.19957   
## mass pedigree age   
## 0.69287 0.24021 0.20136   
##   
## Degrees of Freedom: 536 Total (i.e. Null); 528 Residual  
## Null Deviance: 694.2   
## Residual Deviance: 509.8 AIC: 527.8

We could set up a more extensive grid that automatically generates values according to a regular pattern. This will consume more CPU time but it could investigate a larger parameter space.

tune\_spec <-   
 rand\_forest(mode = "classification") %>%  
 set\_engine("randomForest", tree\_depth = tune(), cost\_complexity = tune())  
  
# Setup a tuning grid which contains candidate values for the above hyperparameters  
tree\_grid <- grid\_regular(cost\_complexity(),tree\_depth(),  
 levels = 5)  
  
tree\_grid

## # A tibble: 25 × 2  
## cost\_complexity tree\_depth  
## <dbl> <int>  
## 1 0.0000000001 1  
## 2 0.0000000178 1  
## 3 0.00000316 1  
## 4 0.000562 1  
## 5 0.1 1  
## 6 0.0000000001 4  
## 7 0.0000000178 4  
## 8 0.00000316 4  
## 9 0.000562 4  
## 10 0.1 4  
## # … with 15 more rows

Now, we fit the model over a range of parameters after we update the workflow to use the new tuning specification:

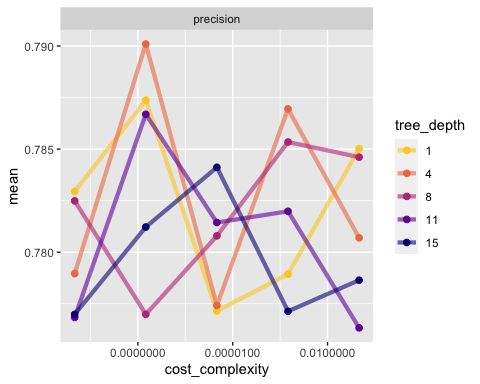
res <- pm\_workflow %>%   
 update\_model(tune\_spec) %>%  
 tune\_grid(resamples = pm\_folds,  
 grid = tree\_grid,  
 metrics = yardstick::metric\_set(yardstick::precision))  
res %>% collect\_metrics()

## # A tibble: 25 × 8  
## tree\_depth cost\_complexity .metric .estimator mean n std\_err .config   
## <int> <dbl> <chr> <chr> <dbl> <int> <dbl> <chr>   
## 1 1 0.0000000001 precision binary 0.783 6 0.0287 Preproce…  
## 2 1 0.0000000178 precision binary 0.787 6 0.0270 Preproce…  
## 3 1 0.00000316 precision binary 0.777 6 0.0257 Preproce…  
## 4 1 0.000562 precision binary 0.779 6 0.0275 Preproce…  
## 5 1 0.1 precision binary 0.785 6 0.0258 Preproce…  
## 6 4 0.0000000001 precision binary 0.779 6 0.0287 Preproce…  
## 7 4 0.0000000178 precision binary 0.790 6 0.0257 Preproce…  
## 8 4 0.00000316 precision binary 0.777 6 0.0258 Preproce…  
## 9 4 0.000562 precision binary 0.787 6 0.0243 Preproce…  
## 10 4 0.1 precision binary 0.781 6 0.0256 Preproce…  
## # … with 15 more rows

best\_params <-  
 res %>%  
 tune::select\_best(metric = "precision")  
  
best\_params

## # A tibble: 1 × 3  
## tree\_depth cost\_complexity .config   
## <int> <dbl> <chr>   
## 1 4 0.0000000178 Preprocessor1\_Model07

res %>%  
 collect\_metrics() %>%  
 mutate(tree\_depth = factor(tree\_depth)) %>%  
 ggplot(aes(cost\_complexity, mean, color = tree\_depth)) +  
 geom\_line(size = 1.5, alpha = 0.6) +  
 geom\_point(size = 2) +  
 facet\_wrap(~ .metric, scales = "free", nrow = 2) +  
 scale\_x\_log10(labels = scales::label\_number()) +  
 scale\_color\_viridis\_d(option = "plasma", begin = .9, end = 0)



## Workflow of Workflows

We can create a set of workflows which in the tidymodels world are referred to as **workflowsets**

Let’s continue our work with the Pima Indians dataset and compare models resulting from the implementation of three different methods. This is commonly done these days. First, we will create a single training and testing pair.

set.seed(1)  
split <- initial\_split(pm)  
  
train\_set <- training(split)  
test\_set <- testing(split)

Let’s pick methods other than logistic regression, Let’s do a Decision Tree and a random forests

# This uses the rpart package  
cart\_spec <-   
 decision\_tree() %>%   
 set\_engine("rpart") %>%   
 set\_mode("classification")  
  
rf\_spec <- rand\_forest(trees = 100, mode = "classification") %>%  
 set\_engine("ranger")

set.seed(2)  
train\_resamples <- bootstraps(train\_set)

all\_workflows <-   
 workflow\_set(  
 preproc = list("formula" = diabetes ~ .),  
 models = list(cart = cart\_spec, rf = rf\_spec)  
 )  
all\_workflows

## # A workflow set/tibble: 2 × 4  
## wflow\_id info option result   
## <chr> <list> <list> <list>   
## 1 formula\_cart <tibble [1 × 4]> <opts[0]> <list [0]>  
## 2 formula\_rf <tibble [1 × 4]> <opts[0]> <list [0]>

all\_workflows <-   
 all\_workflows %>%   
 workflow\_map(resamples = train\_resamples, verbose = TRUE)

## i No tuning parameters. `fit\_resamples()` will be attempted

## i 1 of 2 resampling: formula\_cart

## ✔ 1 of 2 resampling: formula\_cart (6.2s)

## i No tuning parameters. `fit\_resamples()` will be attempted

## i 2 of 2 resampling: formula\_rf

## ✔ 2 of 2 resampling: formula\_rf (6.6s)