CHAPTER 5
Logistic Regression

The contributor for this chapter is Brian Dalessandro. Brian works at Media6Degrees as a VP of data science, and he's active in the research community. He's also served as cochair of the KDD competition. M6D (also known as Media 6 Degrees) is a startup in New York City in the online advertising space. Figure 5-1 shows Brian's data science profile—his y-axis is scaled from Clown to Rockstar.

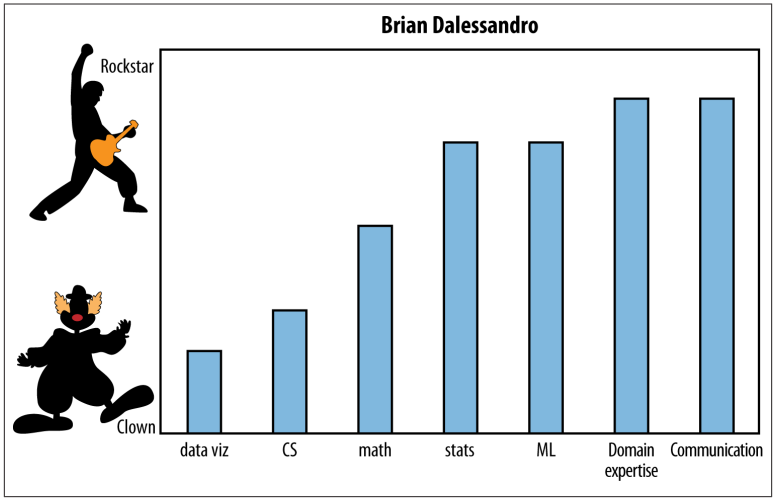

Figure 5-1. Brian's data science profile

Brian came to talk to the class about logistic regression and evaluation, but he started out with two thought experiments.

Thought Experiments

1. How would data science differ if we had a "grand unified theory of everything"? Take this to mean a symbolic explanation of how the world works. This one question raises a bunch of other questions:

   - Would we even need data science if we had such a theory?
   - Is it even theoretically possible to have such a theory? Do such theories lie only in the realm of, say, physics, where we can anticipate the exact return of a comet we see once a century?
   - What's the critical difference between physics and data science that makes such a theory implausible?
   - Is it just accuracy? Or more generally, how much we imagine *can* be explained? Is it because we predict human behavior, which can be affected *by* our predictions, creating a feedback loop?

     It might be useful to think of the sciences as a continuum, where physics is all the way on the right, and as you go left, you get more chaotic—you're adding randomness (and salary). And where is economics on this spectrum? Marketing? Finance?

     If we could model this data science stuff like we already know how to model physics, we'd actually *know* when people will click on what ad, just as we know where the Mars Rover will land. That said, there's general consensus that the real world isn't as well-understood, nor do we expect it to be in the future.

2. In what sense does data science deserve the word "science" in its name?

   Never underestimate the power of creativity—people often have a vision they can describe but no method as to how to get there. As the data scientist, you have to turn that vision into a mathematical model within certain operational constraints. You need to state a well-defined problem, argue for a metric, and optimize for it as well. You also have to make sure you've actually answered the original question.

   There is *art* in data science—it's in translating human problems into the mathematical context of data science and back.

But we always have more than one way of doing this translation—more than one possible model, more than one associated metric, and possibly more than one optimization. So the *science* in data science is—given raw data, constraints, and a problem statement—how to navigate through that maze and make the best choices. Every design choice you make can be formulated as an hypothesis, against which you will use rigorous testing and experimentation to either validate or refute.

This process, whereby one formulates a well-defined hypothesis and then tests it, might rise to the level of a science in certain cases. Specifically, the scientific method is adopted in data science as follows:

- You hold on to your existing best performer.
- Once you have a new idea to prototype, set up an experiment wherein the two best models compete.
- Rinse and repeat (while not overfitting).

Classifiers

This section focuses on the process of choosing a *classifier*. Classification involves mapping your data points into a finite set of labels or the probability of a given label or labels. We've already seen some examples of classification algorithms, such as Naive Bayes and k-nearest neighbors (k-NN), in the previous chapters. Table 5-1 shows a few examples of when you'd want to use classification:

Table 5-1. Classifier example questions and answers

"Will someone click on this ad?"	0 or 1 (no or yes)
"What number is this (image recognition)?"	0, 1, 2, etc.
"What is this news article about?"	"Sports"
"Is this spam?"	0 or 1
"Is this pill good for headaches?"	0 or 1

From now on we'll talk about binary classification only (0 or 1).

In this chapter, we're talking about logistic regression, but there's other classification algorithms available, including decision trees (which we'll cover in Chapter 7), random forests (Chapter 7), and support

vector machines and neural networks (which we aren't covering in this book).

The big picture is that given data, a real-world classification problem, and constraints, you need to determine:

1. Which classifier to use
2. Which optimization method to employ
3. Which loss function to minimize
4. Which features to take from the data
5. Which evaluation metric to use

Let's talk about the first one: how do you know which classifier to choose? One possibility is to try them all, and choose the best performer. This is fine if you have no constraints, or if you ignore constraints. But usually constraints are a big deal—you might have tons of data, or not much time, or both. This is something people don't talk about enough. Let's look at some constraints that are common across most algorithms.

Runtime

Say you need to update 500 models a day. That is the case at M6D, where their models end up being bidding decisions. In that case, they start to care about various speed issues. First, how long it takes to update a model, and second, how long it takes to use a model to actually make a decision if you have it. This second kind of consideration is usually more important and is called *runtime*.

Some algorithms are slow at runtime. For example, consider k-NN: given a new data point in some large-dimensional space, you actually have to find the k closest data points to it. In particular, you need to have all of your data points in memory.

Linear models, by contrast, are very fast, both to update and to use at runtime. As we'll see in Chapter 6, you can keep running estimates of the constituent parts and just update with new data, which is fast and doesn't require holding old data in memory. Once you have a linear model, it's just a matter of storing the coefficient vector in a runtime machine and doing a single dot product against the user's feature vector to get an answer.

You

One underappreciated constraint of being a data scientist is your own understanding of the algorithm. Ask yourself carefully, do you understand it for real? *Really*? It's OK to admit it if you don't.

You don't have to be a master of every algorithm to be a good data scientist. The truth is, getting the best fit of an algorithm often requires intimate knowledge of said algorithm. Sometimes you need to tweak an algorithm to make it fit your data. A common mistake for people not completely familiar with an algorithm is to overfit when they think they're tweaking.

Interpretability

You often need to be able to interpret your model for the sake of the business. Decision trees are very easy to interpret. Random forests, on the other hand, are not, even though they are almost the same thing. They can take exponentially longer to explain in full. If you don't have 15 years to spend understanding a result, you may be willing to give up some accuracy in order to have it be easier to understand.

For example, by law, credit card companies have to be able to explain their denial-of-credit decisions, so decision trees make more sense than random forests. You might not have a law about it where you work, but it still might make good sense for your business to have a simpler way to explain the model's decision.

Scalability

How about scalability? In general, there are three things you have to keep in mind when considering scalability:

1. Learning time: How much time does it take to train the model?
2. Scoring time: How much time does it take to give a new user a score once the model is in production?
3. Model storage: How much memory does the production model use up?

Here's a useful paper to look at when comparing models: An Empirical Comparison of Supervised Learning Algorithms, from which we've learned:

- Simpler models are more interpretable but aren't as good performers.
- The question of which algorithm works best is problem-dependent.
- It's also constraint-dependent.

M6D Logistic Regression Case Study

Brian and his team have three core problems as data scientists at M6D:

1. Feature engineering: Figuring out which features to use and how to use them.
2. User-level conversion prediction: Forecasting when someone will click.
3. Bidding: How much it is worth to show a given ad to a given user?

This case study focuses on the second problem. M6D uses logistic regression for this problem because it's highly scalable and works great for binary outcomes like clicks.

Click Models

At M6D, they need to match clients, which represent advertising companies, to individual users. Generally speaking, the advertising companies want to target ads to users based on a user's likelihood to click. Let's discuss what kind of data they have available first, and then how you'd build a model using that data.

M6D keeps track of the websites users have visited, but the data scientists don't look at the contents of the page. Instead they take the associated URL and hash it into some random string. They thus accumulate information about users, which they stash in a vector. As an example, consider the user "u" during some chosen time period:

```
u = < &ltfxyz, 123, sdqwe, 13ms&gtg >
```

This means "u" visited four sites and the URLs that "u" visited are hashed to those strings. After collecting information like this for all the users, they build a giant matrix whose columns correspond to sites and whose rows correspond to users, and a given entry is "1" if that

user went to that site. Note it's a sparse matrix, because most people don't go to all that many sites.

To make this a classification problem, they need to have *classes* they are trying to predict. Let's say in this case, they want to predict whether a given user will click on a shoe ad or not. So there's two classes: "users who clicked on the shoe ad" and "users who did not click on the shoe ad." In the training dataset, then, in addition to the sparse matrix described, they'll also have a variable, or column, of labels. They label the behavior "clicked on a shoe ad" as "1," say, and "didn't click" as "0." Once they fit the classifier to the dataset, for any new user, the classifier will predict whether he will click or not (the label) based on the predictors (the user's browsing history captured in the URL matrix).

Now it's your turn: your goal is to build and train the model from a training set. Recall that in Chapter 3 you learned about *spam classifiers*, where the features are words. But you didn't particularly care about the meaning of the words. They might as well be strings. Once you've labeled as described earlier, this looks just like spam classification because you have a binary outcome with a large sparse binary matrix capturing the predictors. You can now rely on well-established algorithms developed for spam detection.

You've reduced your current problem to a previously solved problem! In the previous chapter, we showed how to solve this with Naive Bayes, but here we'll focus on using logistic regression as the model.

The output of a logistic regression model is the *probability* of a given click in this context. Likewise, the spam filters really judge the *probability* of a given email being spam. You can use these probabilities directly or you could find a threshold so that if the probability is above that threshhold (say, 0.75), you predict a click (i.e., you show an ad), and below it you decide it's not worth it to show the ad. The point being here that unlike with linear regression—which does its best to predict the actual value—the aim of logistic regression isn't to predict the actual value (0 or 1). Its job is to output a probability.

Although technically it's possible to implement a linear model such as linear regression on such a dataset (i.e., R will let you do it and won't break or tell you that you shouldn't do it), one of the problems with a linear model like linear regression is that it would give predictions below 0 and above 1, so these aren't directly interpretable as probabilities.

The Underlying Math

So far we've seen that the beauty of logistic regression is it outputs values bounded by 0 and 1; hence they can be directly interpreted as probabilities. Let's get into the math behind it a bit. You want a function that takes the data and transforms it into a single value bounded inside the closed interval $[0,1]$. For an example of a function bounded between 0 and 1, consider the inverse-logit function shown in Figure 5-2.

$$P(t) = \text{logit}^{-1}(t) \equiv \frac{1}{\left(1+e^{-t}\right)} = \frac{e^{t}}{1+e^{t}}$$

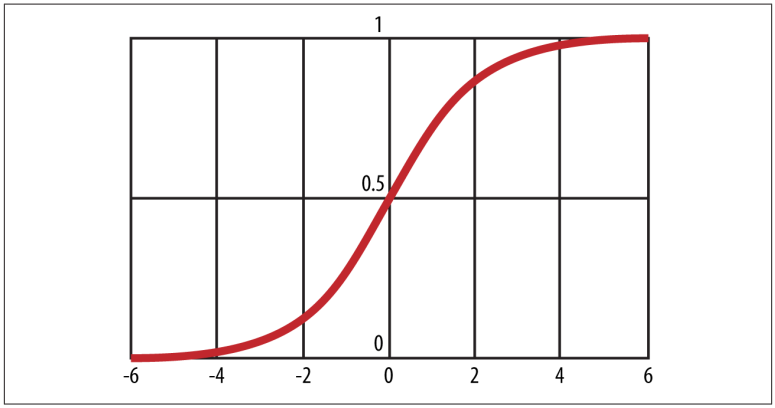

Figure 5-2. The inverse-logit function

Logit Versus Inverse-logit

The logit function takes x values in the range $[0,1]$ and transforms them to y values along the entire real line:

$$logit(p) = log\left(\frac{p}{1-p}\right) = log(p) - log(1-p)$$

The inverse-logit does the reverse, and takes x values along the real line and tranforms them to y values in the range $[0,1]$.

Note when t is large, e^{-t} is tiny so the denominator is close to 1 and the overall value is close to 1. Similarly when t is small, e^{-t} is large so

the denominator is large, which makes the function close to zero. So that's the inverse-logit function, which you'll use to begin deriving a logistic regression model. In order to model the data, you need to work with a slightly more general function that expresses the relationship between the data and a probability of a click. Start by defining:

$$P(c_i|x_i) = \left[\text{logit}^{-1}\left(\alpha + \beta^\tau x_i\right)\right]^{c_i *} \left[1 - \text{logit}^{-1}\left(\alpha + \beta^\tau x_i\right)\right]^{1-c_i}$$

Here c_i is the labels or classes (clicked or not), and x_i is the vector of features for user i. Observe that c_i can only be 1 or 0, which means that if $c_i = 1$, the second term cancels out and you have:

$$P(c_i = 1|x_i) = \frac{1}{1 + e^{-\left(\alpha + \beta^\tau x_i\right)}} = \text{logit}^{-1}\left(\alpha + \beta^\tau x_i\right)$$

And similarly, if $c_i = 0$, the first term cancels out and you have:

$$P(c_i = 0|x_i) = 1 - \text{logit}^{-1}\left(\alpha + \beta^\tau x_i\right)$$

To make this a linear model in the outcomes c_i, take the log of the odds ratio:

$$\log\left(P(c_i = 1|x_i) / \left(1 - P(c_i = 1|x_i)\right)\right) = \alpha + \beta^\tau x_i.$$

Which can also be written as:

$$\text{logit}\left(P(c_i = 1|x_i)\right) = \alpha + \beta^\tau x_i.$$

If it feels to you that we went in a bit of a circle here (this last equation was also implied by earlier equations), it's because we did. The purpose of this was to show you how to go back and forth between the probabilities and the linearity.

So the logit of the probability that user *i* clicks on the shoe ad is being modeled as a linear function of the features, which were the URLs that user *i* visited. This model is called the *logistic regression* model.

The parameter α is what we call the *base rate*, or the unconditional probability of "1" or "click" knowing nothing more about a given user's feature vector x_i. In the case of measuring the likelihood of an average

user clicking on an ad, the base rate would correspond to the click-through rate, i.e., the tendency over all users to click on ads. This is typically on the order of 1%.

If you had no information about your specific situation except the base rate, the average prediction would be given by just α:

$$P(c_i = 1) = \frac{1}{1 + e^{-\alpha}}$$

The variable β defines the slope of the logit function. Note that in general it's a vector that is as long as the number of features you are using for each data point. The vector β determines the extent to which certain features are markers for increased or decreased likelihood to click on an ad.

Estimating α and β

Your immediate modeling goal is to use the training data to find the best choices for α and β. In general you want to solve this with maximum likelihood estimation and use a convex optimization algorithm because the likelihood function is convex; you can't just use derivatives and vector calculus like you did with linear regression because it's a complicated function of your data, and in particular there is no closed-form solution.

Denote by Θ the pair $\{\alpha, \beta\}$. The *likelihood function* L is defined by:

$$L(\Theta | X_1, X_2, \cdots, X_n) = P(X | \Theta) = P(X_1 | \Theta) \cdot \cdots \cdot P(X_n | \Theta)$$

where you are assuming the data points X_i are independent, where $i = 1, \ldots, n$ represent your n users. This independence assumption corresponds to saying that the click behavior of any given user doesn't affect the click behavior of all the other users—in this case, "click behavior" means "probability of clicking." It's a relatively safe assumption at a given point in time, but not forever. (Remember the independence assumption is what allows you to express the likelihood function as the product of the densities for each of the n observations.)

You then search for the parameters that maximize the likelihood, having observed your data:

$$\Theta_{MLE} = argmax_{\Theta} \prod_1^n P(X_i|\Theta).$$

Setting $p_i = 1/\left(1+e^{-(\alpha+\beta^t x_i)}\right)$, the probability of a single observation, $P(X_i|\Theta)$ is:

$$p_i^{c_i} \cdot (1-p_i)^{1-c_i}$$

So putting it all together, you have:

$$\Theta_{MLE} = argmax_{\Theta} \prod_1^n p_i^{c_i} \cdot (1-p_i)^{1-c_i}$$

Now, how do you maximize the likelihood?

Well, when you faced this situation with linear regression, you took the derivative of the likelihood with respect to α and β, set that equal to zero, and solved. But if you try that in this case, it's not possible to get a closed-form solution. The key is that the values that maximize the likelihood will also maximize the log likelihood, which is equivalent to minimizing the *negative* log likelihood. So you transform the problem to find the minimum of the negative log likelihood.

Now you have to decide which optimization method to use. As it turns out, under reasonable conditions, both of the optimization methods we describe below will converge to a *global maximum* when they converge at all. The "reasonable condition" is that the variables are not linearly dependent, and in particular guarantees that the Hessian matrix will be positive definite.

More on Maximum Likelihood Estimation

We realize that we went through that a bit fast, so if you want more details with respect to maximum likelihood estimation, we suggest looking in *Statistical Inference* by Casella and Berger, or if it's linear algebra in general you want more details on, check out Gilbert Strang's *Linear Algebra and Its Applications*.

Newton's Method

You can use numerical techniques to find your global maximum by following the reasoning underlying Newton's method from calculus; namely, that you can pretty well approximate a function with the first few terms of its Taylor Series.

Specifically, given a step size γ, you compute a local gradient $\nabla\Theta$, which corresponds to the first derivative, and a Hessian matrix H, which corresponds to the second derivative. You put them together and each step of the algorithm looks something like this:

$$\Theta_{n+1} = \Theta_n - \gamma H^{-1} \cdot \nabla\Theta.$$

Newton's method uses the curvature of log-likelihood to choose an appropriate step direction. Note that this calculation involves inverting the $k \times k$ Hessian matrix, which is bad when there's lots of features, as in 10,000 or something. Typically you don't have that many features, but it's not impossible.

In practice you'd never actually invert the Hessian—instead you'd solve an equation of the form $Ax = y$, which is much more computationally stable than finding A^{-1}.

Stochastic Gradient Descent

Another possible method to maximize your likelihood (or minimize your negative log likelihood) is called Stochastic Gradient Descent. It approximates a gradient using a single observation at a time. The algorithm updates the current best-fit parameters each time it sees a new data point. The good news is that there's no big matrix inversion, and it works well with both huge data and sparse features; it's a big deal in Mahout and Vowpal Wabbit, two open source projects to enable large-scale machine learning across a variety of algorithms. The bad news is it's not such a great optimizer and it's very dependent on step size.

Implementation

In practice you don't have to code up the iterative reweighted least squares or stochastic gradient optimization method yourself; this is implemented in R or in any package that implements logistic regression. So suppose you had a dataset that had the following first five rows:

```
click url_1  url_2  url_3  url_4  url_5
1      0       0      0      1      0
1      0       1      1      0      1
0      1       0      0      1      0
1      0       0      0      0      0
1      1       0      1      0      1
```

Call this matrix "train," and then the command line in R would be:

```
fit <- glm(click ~ url_1 + url_2 + url_3 + url_4 + url_5,
           data = train, family = binomial(logit))
```

Evaluation

Let's go back to the big picture from earlier in the chapter where we told you that you have many choices you need to make when confronted with a classification problem. One of the choices is how you're going to evaluate your model. We discussed this already in Chapter 3 with respect to linear regression and k-NN, as well as in the previous chapter with respect to Naive Bayes. We generally use different evaluation metrics for different kinds of models, and in different contexts. Even logistic regression can be applied in multiple contexts, and depending on the context, you may want to evaluate it in different ways.

First, consider the context of using logistic regression as a ranking model—meaning you are trying to determine the *order* in which you show ads or items to a user based on the probability they would click. You could use logistic regression to estimate probabilities, and then rank-order the ads or items in decreasing order of likelihood to click based on your model. If you wanted to know how good your model was at discovering relative rank (notice in this case, you could care less about the absolute scores), you'd look to one of:

Area under the receiver operating curve (AUC)
: In signal detection theory, a receiver operating characteristic curve, or ROC curve, is defined as a plot of the true positive rate against the false positive rate for a binary classification problem as you change a threshold. In particular, if you took your training set and ranked the items according to their probabilities and varied the threshold (from ∞ to $-\infty$) that determined whether to classify the item as 1 or 0 , and kept plotting the true positive rate versus the false positive rate, you'd get the ROC curve. The area under that curve, referred to as the AUC, is a way to measure the success of a classifier or to compare two classifiers. Here's a nice paper on it by Tom Fawcett, "Introduction to ROC Analysis".

Area under the cumulative lift curve

An alternative is the area under the cumulative lift curve, which is frequently used in direct marketing and captures how many times it is better to use a model versus not using the model (i.e., just selecting at random).

We'll see more of both of these in Chapter 13.

Warning: Feedback Loop!

If you want to productionize logistic regression to rank ads or items based on clicks and impressions, then let's think about what that means in terms of the data getting generated. So let's say you put an ad for hair gel above an ad for deodorant, and then more people click on the ad for hair gel—is it because you put it on top or because more people want hair gel? How can you feed that data into future iterations of your algorithm given that you potentially caused the clicks yourself and it has nothing to do with the ad quality? One solution is to always be logging the *position* or *rank* that you showed the ads, and then use *that* as one of the predictors in your algorithm. So you would then model the probability of a click as a function of position, vertical, brand, or whatever other features you want. You could then use the parameter estimated for position and use that going forward as a "position normalizer." There's a whole division at Google called Ads Quality devoted to problems such as these, so one little paragraph can't do justice to all the nuances of it.

Second, now suppose instead that you're using logistic regression for the purposes of classification. Remember that although the observed output was a binary outcome (1,0), the output of a logistic model is a probability. In order to use this for classification purposes, for any given unlabeled item, you would get its predicted probability of a click. Then to minimize the misclassification rate, if the predicted probability is > 0.5 that the label is 1 (click), you would label the item a 1 (click), and otherwise 0. You have several options for how you'd then evaluate the quality of the model, some of which we already discussed in Chapters 3 and 4, but we'll tell you about them again here so you see their pervasiveness:

Lift

How much more people are buying or clicking because of a model (once we've introduced it into production).

Accuracy
: How often the correct outcome is being predicted, as discussed in Chapters 3 and 4.

Precision
: This is the (number of true positives)/(number of true positives + number of false positives).

Recall
: This is the (number of true positives)/(number of true positives + number of false negatives).

F-score
: We didn't tell you about this one yet. It essentially combines precision and recall into a single score. It's the harmonic mean of precision and recall, so $(2 \times \text{precision} \times \text{recall})/(\text{precision} + \text{recall})$. There are generalizations of this that are essentially changing how much you weight one or the other.

Finally, for density estimation, where we need to know an actual probability rather than a relative score, we'd look to:

Mean squared error
: We discussed with respect to linear regression. As a reminder, this is the average squared distance between the predicted and actual values.

Root squared error
: The square root of mean squared error.

Mean absolute error
: This is a variation on mean squared error and is simply the average of the absolute value of the difference between the predicted and actual values.

In general, it's hard to compare lift curves, but you can compare AUC (area under the receiver operator curve)—they are "base rate invariant." In other words, if you bring the click-through rate from 1% to 2%, that's 100% lift; but if you bring it from 4% to 7%, that's less lift but more effect. AUC does a better job in such a situation when you want to compare.

Density estimation tests tell you how well are you fitting for conditional probability. In advertising, this may arise if you have a situation where each ad impression costs $\$c$ and for each conversion you receive

$q. You will want to target every conversion that has a positive expected value, i.e., whenever:

$$P(Conversion|X) \cdot \$q > \$c$$

But to do this you need to make sure the probability estimate on the left is *accurate*, which in this case means something like the mean squared error of the estimator is small. Note a model can give you good relative rankings—it gets the order right—but bad estimates of the probability.

Think about it this way: it could say to rank items in order 1, 2, and 3 and estimate the probabilities as .7, .5, and .3. It might be that the *true* probabilities are .03, .02, and .01, so our estimates are totally off, but the ranking was correct.

Using A/B Testing for Evaluation

When we build models and optimize them with respect to some evaluation metric such as accuracy or mean squared error, the estimation method itself is built to optimize parameters *with respect* to these metrics. In some contexts, the metrics we might want to optimize for are something else altogether, such as revenue. So we might try to build an algorithm that optimizes for accuracy, when our real goal is to make money. The model itself may not directly capture this. So a way to capture it is to run A/B tests (or statistical experiments) where we divert some set of users to one version of the algorithm and another set of users to another version of the algorithm, and check the difference in performance of metrics we care about, such as revenue or revenue per user, or something like that. We'll discuss A/B testing more in Chapter 11.

Media 6 Degrees Exercise

Media 6 Degrees kindly provided a dataset that is perfect for exploring logistic regression models, and evaluating how good the models are. Follow along by implementing the following R code. The dataset can be found at *https://github.com/oreillymedia/doing_data_science*.

Sample R Code

```
# Author: Brian Dalessandro
# Read in data, look at the variables and create a training
and test set
file <- "binary_class_dataset.txt"
set <- read.table(file, header = TRUE, sep = "\t",
                  row.names = "client_id")
names(set)

split <- .65
set["rand"] <- runif(nrow(set))
train <- set[(set$rand <= split), ]
test <- set[(set$rand > split), ]
set$Y <- set$Y_BUY


##############################################################
###########             R FUNCTIONS             ##########
##############################################################

library(mgcv)

# GAM Smoothed plot
plotrel <- function(x, y, b, title) {
    # Produce a GAM smoothed representation of the data
    g <- gam(as.formula("y ~ x"), family = "binomial",
             data = set)
    xs <- seq(min(x), max(x), length = 200)
    p <- predict(g, newdata = data.frame(x = xs),
                 type = "response")

    # Now get empirical estimates (and discretize if
    # non discrete)
    if (length(unique(x)) > b) {
        div <- floor(max(x) / b)
        x_b <- floor(x / div) * div
        c <- table(x_b, y)
    }
    else { c <- table(x, y) }
    pact <- c[ , 2]/(c[ , 1]+c[ , 2])
    cnt <- c[ , 1]+c[ , 2]
    xd <- as.integer(rownames(c))
    plot(xs, p, type="l", main=title,
         ylab = "P(Conversion | Ad, X)", xlab="X")
         points(xd, pact, type="p", col="red")
    rug(x+runif(length(x)))
}


library(plyr)
```

```
# wMAE plot and calculation
getmae <- function(p, y, b, title, doplot) {
    # Normalize to interval [0,1]
    max_p <- max(p)
    p_norm <- p / max_p
    # break up to b bins and rescale
    bin <- max_p * floor(p_norm * b) / b
    d <- data.frame(bin, p, y)
    t <- table(bin)
    summ <- ddply(d, .(bin), summarise, mean_p = mean(p),
                  mean_y = mean(y))
    fin <- data.frame(bin = summ$bin, mean_p = summ$mean_p,
                      mean_y = summ$mean_y, t)
    # Get wMAE
    num = 0
    den = 0
    for (i in c(1:nrow(fin))) {
        num <- num + fin$Freq[i] * abs(fin$mean_p[i] -
                                       fin$mean_y[i])
        den <- den + fin$Freq[i]
    }
    wmae <- num / den
    if (doplot == 1) {
        plot(summ$bin, summ$mean_p, type = "p",
             main = paste(title," MAE =", wmae),
             col = "blue", ylab = "P(C | AD, X)",
             xlab = "P(C | AD, X)")
        points(summ$bin, summ$mean_y, type = "p", col = "red")
        rug(p)
    }
    return(wmae)
}


library(ROCR)
get_auc <- function(ind, y) {
    pred <- prediction(ind, y)
    perf <- performance(pred, 'auc', fpr.stop = 1)
    auc <- as.numeric(substr(slot(perf, "y.values"), 1, 8),
                      double)
    return(auc)
}


# Get X-Validated performance metrics for a given feature set

getxval <- function(vars, data, folds, mae_bins) {
    # assign each observation to a fold
    data["fold"] <- floor(runif(nrow(data)) * folds) + 1
    auc <- c()
    wmae <- c()
```

```
    fold <- c()
    # make a formula object
    f = as.formula(paste("Y", "~", paste(vars,
                    collapse = "+")))
    for (i in c(1:folds)) {
        train <- data[(data$fold != i), ]
        test <- data[(data$fold == i), ]
        mod_x <- glm(f, data=train, family = binomial(logit))
        p <- predict(mod_x, newdata = test, type = "response")
        # Get wMAE
        wmae <- c(wmae, getmae(p, test$Y, mae_bins,
                  "dummy", 0))
        fold <- c(fold, i)
        auc <- c(auc, get_auc(p, test$Y))
    }
    return(data.frame(fold, wmae, auc))
}

#####################################################################
##########           MAIN: MODELS AND PLOTS          ##########
#####################################################################
# Now build a model on all variables and look at coefficients
and model fit
vlist      <-      c("AT_BUY_BOOLEAN",      "AT_FREQ_BUY",
"AT_FREQ_LAST24_BUY",
    "AT_FREQ_LAST24_SV", "AT_FREQ_SV", "EXPECTED_TIME_BUY",
    "EXPECTED_TIME_SV", "LAST_BUY", "LAST_SV", "num_checkins")
f = as.formula(paste("Y_BUY", "~" , paste(vlist,
               collapse = "+")))
fit <- glm(f, data = train, family = binomial(logit))
summary(fit)

# Get performance metrics on each variable

vlist      <-      c("AT_BUY_BOOLEAN",      "AT_FREQ_BUY",
"AT_FREQ_LAST24_BUY",
    "AT_FREQ_LAST24_SV", "AT_FREQ_SV", "EXPECTED_TIME_BUY",
    "EXPECTED_TIME_SV", "LAST_BUY", "LAST_SV", "num_checkins")

# Create empty vectors to store the performance/evaluation met
rics
auc_mu <- c()
auc_sig <- c()
mae_mu <- c()
mae_sig <- c()

for (i in c(1:length(vlist))) {
    a <- getxval(c(vlist[i]), set, 10, 100)
    auc_mu <- c(auc_mu, mean(a$auc))
    auc_sig <- c(auc_sig, sd(a$auc))
    mae_mu <- c(mae_mu, mean(a$wmae))
```

```
    mae_sig <- c(mae_sig, sd(a$wmae))
}

univar <- data.frame(vlist, auc_mu, auc_sig, mae_mu, mae_sig)

# Get MAE plot on single variable -
# use holdout group for evaluation
set <- read.table(file, header = TRUE, sep = "\t",
                  row.names="client_id")
names(set)

split<-.65
set["rand"] <- runif(nrow(set))
train <- set[(set$rand <= split), ]
test <- set[(set$rand > split), ]
set$Y <- set$Y_BUY

fit <- glm(Y_BUY ~ num_checkins, data = train,
           family = binomial(logit))
y <- test$Y_BUY
p <- predict(fit, newdata = test, type = "response")

getmae(p,y,50,"num_checkins",1)

# Greedy Forward Selection
rvars <- c("LAST_SV", "AT_FREQ_SV", "AT_FREQ_BUY",
    "AT_BUY_BOOLEAN", "LAST_BUY", "AT_FREQ_LAST24_SV",
    "EXPECTED_TIME_SV", "num_checkins",
    "EXPECTED_TIME_BUY", "AT_FREQ_LAST24_BUY")
# Create empty vectors
auc_mu <- c()
auc_sig <- c()
mae_mu <- c()
mae_sig <- c()

for (i in c(1:length(rvars))) {
    vars <- rvars[1:i]
    vars
    a <- getxval(vars, set, 10, 100)
    auc_mu <- c(auc_mu, mean(a$auc))
    auc_sig <- c(auc_sig, sd(a$auc))
    mae_mu <- c(mae_mu, mean(a$wmae))
    mae_sig <- c(mae_sig, sd(a$wmae))
}
kvar<-data.frame(auc_mu, auc_sig, mae_mu, mae_sig)

# Plot 3 AUC Curves
y <- test$Y_BUY

fit <- glm(Y_BUY~LAST_SV, data=train,
           family = binomial(logit))
```

```
p1 <- predict(fit, newdata=test, type="response")
fit <- glm(Y_BUY~LAST_BUY, data=train,
           family = binomial(logit))
p2 <- predict(fit, newdata=test, type="response")
fit <- glm(Y_BUY~num_checkins, data=train,
           family = binomial(logit))
p3 <- predict(fit, newdata=test,type="response")

pred <- prediction(p1,y)
perf1 <- performance(pred,'tpr','fpr')
pred <- prediction(p2,y)
perf2 <- performance(pred,'tpr','fpr')
pred <- prediction(p3,y)
perf3 <- performance(pred,'tpr','fpr')


plot(perf1, color="blue", main="LAST_SV (blue),
     LAST_BUY (red), num_checkins (green)")
plot(perf2, col="red", add=TRUE)
plot(perf3, col="green", add=TRUE)
```