Eight to Late

Sensemaking and Analytics for Organizations

A gentle introduction to logistic regression and lasso regularisation using R

with 11 comments

In this day and age of artificial intelligence and deep learning, it is easy to forget that simple algorithms can work well for a surprisingly large range of practical business problems. And the simplest place to start is with the granddaddy of data science algorithms: linear regression and its close cousin, logistic regression. Indeed, in his acclaimed MOOC and accompanying textbook, Yaser Abu-Mostafa spends a good portion of his time talking about linear methods, and with good reason too: linear methods are not only a good way to learn the key principles of machine learning, they can also be remarkably helpful in zeroing in on the most important predictors.

My main aim in this post is to provide a beginner level introduction to logistic regression using R and also introduce LASSO (Least Absolute Shrinkage and Selection Operator), a powerful feature selection technique that is very useful for regression problems. Lasso is essentially a regularization method. If you're unfamiliar with the term, think of it as a way to reduce overfitting using less complicated functions (and if that means nothing to you, check out my prelude to machine learning). One way to do this is to toss out less important variables, after checking that they aren't important. As we'll discuss later, this can be done manually by examining p-values of coefficients and discarding those variables whose coefficients are not significant. However, this can become tedious for classification problems with many independent variables. In such situations, lasso offers a neat way to model the dependent variable while *automagically selecting significant variables by shrinking the coefficients of unimportant predictors to zero. A*ll this without having to mess around with p-values or obscure information criteria. How good is that?

Why not linear regression?

In linear regression one attempts to model a dependent variable (i.e. the one being predicted) using the best straight line fit to a set of predictor variables. The best fit is usually taken to be one that minimises the root mean square error, which is the sum of square of the differences between the actual and predicted values of the dependent variable. One can think of logistic regression as the equivalent of linear regression for a classification problem. In what follows we'll look at binary classification – i.e. a situation where the dependent variable takes on one of two possible values (Yes/No, True/False, 0/1 etc.).

First up, you might be wondering why one can't use linear regression for such problems. The main reason is that classification problems are about determining *class membership* rather than predicting *variable values*, and linear regression is more naturally suited to the latter than the former. One could, in principle, use linear regression for situations where there is a natural ordering of categories like *High*, *Medium* and *Low* for example. However, one then has to map sub-ranges of the predicted values to categories. Moreover, since predicted values are potentially unbounded (in data as yet unseen) there remains a degree of arbitrariness associated with such a mapping.

Logistic regression sidesteps the aforementioned issues by modelling class probabilities instead. Any input to the model yields a number lying between 0 and 1, representing the probability of class membership. One is still left with the problem of determining the threshold probability, i.e. the probability at which the category flips from one to the other. By default this is set to p=0.5, but in reality it should be settled based on how the model will be used. For example, for a marketing model that identifies potentially responsive customers, the threshold for a positive event might be set low (much less than 0.5) because the client does not really care about mailouts going to a non-responsive customer (the negative event). Indeed they may be more than OK with it as there's always a chance – however small – that a non-responsive customer will actually respond. As an opposing example, the cost of a false positive would be high in a machine learning application that grants access to sensitive information. In this case, one might want to set the threshold probability to a value closer to 1, say 0.9 or even higher. The point is, the setting an appropriate threshold probability is a business issue, not a technical one.

Logistic regression in brief

So how does logistic regression work?

For the discussion let's assume that the outcome (predicted variable) and predictors are denoted by Y and X respectively and the two classes of interest are denoted by + and – respectively. We wish to model the conditional probability that the outcome Y is +, given that the input variables (predictors) are X. The conditional probability is denoted by p(Y=+|X) which we'll abbreviate as p(X) since we know we are referring to the positive outcome Y=+.

As mentioned earlier, we are after the *probability of class membership* so we must ensure that the hypothesis function (a fancy word for the model) always lies between 0 and 1. The function assumed in logistic regression is:

$$p(X) = \frac{\exp^{\beta_0+\beta_1 X}}{1+\exp^{\beta_0+\beta_1 X}} .....(1)$$

You can verify that $p(X)$ does indeed lie between 0 and 1 as X varies from $-\infty$ to ∞. Typically, however, the values of X that make sense are bounded as shown in the example (stolen from Wikipedia) shown in Figure 1. The figure also illustrates the typical S-shaped curve characteristic of logistic regression.

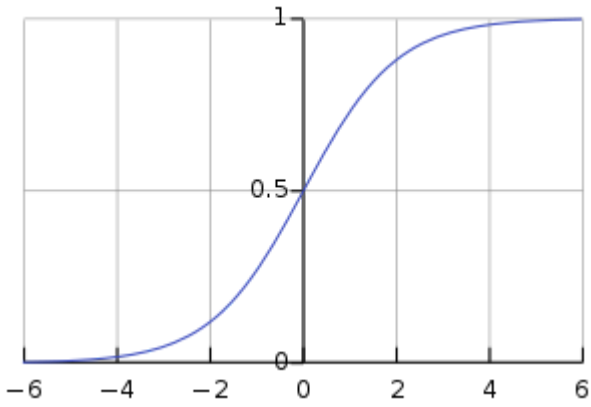

Figure 1: Logistic function

As an aside, you might be wondering where the name *logistic* comes from. An equivalent way of expressing the above equation is:

$$\log(\frac{p(X)}{1-p(X)}) = \beta_0 + \beta_1 X.....(2)$$

The quantity on the left is the logarithm of the odds. So, the model is a linear regression of the *log-odds*, sometimes called *logit*, and hence the name *logistic*.

The problem is to find the values of β_0 and β_1 that results in a $p(X)$ that most accurately classifies all the observed data points – that is, those that belong to the positive class have a probability as close as possible to 1 and those that belong to the negative class have a probability as close as possible to 0. One way to frame this problem is to say that we wish to maximise the product of these probabilities, often referred to as the likelihood:

$$\log(\prod_{i:Y_i=+} p(X_i) \prod_{j:Y_j=-} (1-p(X_j)))$$

Where $\prod$ represents the products over i and j, which run over the +ve and –ve classed points respectively. This approach, called maximum likelihood estimation, is quite common in many machine learning settings, especially those involving probabilities.

It should be noted that in practice one works with the *log* likelihood because it is easier to work with mathematically. Moreover, one *minimises* the *negative* log likelihood which, of course, is the same as maximising the log likelihood. The quantity one minimises is thus:

$$L = -\log(\prod_{i:Y_i=+} p(X_i) \prod_{j:Y_j=-} (1-p(X_j))).....(3)$$

However, these are technical details that I mention only for completeness. As you will see next, they have little bearing on the practical use of logistic regression.

Logistic regression in R – an example

In this example, we'll use the logistic regression option implemented within the *glm* function that comes with the base R installation. This function fits a class of models collectively known as *generalized linear models*. We'll apply the function to the *Pima Indian Diabetes* dataset that comes with

the mlbench package. The code is quite straightforward – particularly if you've read earlier articles in my "gentle introduction" series – so I'll just list the code below noting that the logistic regression option is invoked by setting *family="binomial"* in the *glm* function call.

Here we go:

```
#set working directory if needed (modify path as needed)
#setwd("C:/Users/Kailash/Documents/logistic")
#load required library
library(mlbench)
#load Pima Indian Diabetes dataset
data("PimaIndiansDiabetes")
#set seed to ensure reproducible results
set.seed(42)
#split into training and test sets
PimaIndiansDiabetes["train"] <- ifelse(runif(nrow(PimaIndiansDiabetes))<0.8,1,0)
#separate training and test sets
trainset <- PimaIndiansDiabetes[PimaIndiansDiabetes$train==1,]
testset <- PimaIndiansDiabetes[PimaIndiansDiabetes$train==0,]
#get column index of train flag
trainColNum <- grep("train",names(trainset))
#remove train flag column from train and test sets
trainset <- trainset[,-trainColNum]
testset <- testset[,-trainColNum]
#get column index of predicted variable in dataset
typeColNum <- grep("diabetes",names(PimaIndiansDiabetes))
#build model
glm_model <- glm(diabetes~.,data = trainset, family = binomial)
summary(glm_model)
Call:
glm(formula = diabetes ~ ., family = binomial, data = trainset)
<<output edited>>
Coefficients:
             Estimate  Std. Error z value Pr(>|z|)
(Intercept)-8.1485021 0.7835869 -10.399  < 2e-16 ***
pregnant    0.1200493 0.0355617   3.376  0.000736 ***
glucose     0.0348440 0.0040744   8.552  < 2e-16 ***
pressure   -0.0118977 0.0057685  -2.063  0.039158 *
triceps     0.0053380 0.0076523   0.698  0.485449
insulin    -0.0010892 0.0009789  -1.113  0.265872
mass        0.0775352 0.0161255   4.808  1.52e-06 ***
pedigree    1.2143139 0.3368454   3.605  0.000312 ***
age         0.0117270 0.0103418   1.134  0.256816
—
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#predict probabilities on testset
#type="response" gives probabilities, type="class" gives class
glm_prob <- predict.glm(glm_model,testset[,-typeColNum],type="response")
#which classes do these probabilities refer to? What are 1 and 0?
contrasts(PimaIndiansDiabetes$diabetes)
    pos
```

```
neg 0
pos 1
#make predictions
##…first create vector to hold predictions (we know 0 refers to neg now)
glm_predict <- rep("neg",nrow(testset))
glm_predict[glm_prob>.5] <- "pos"
#confusion matrix
table(pred=glm_predict,true=testset$diabetes)
glm_predict  neg pos
      neg    90 22
      pos     8 33
#accuracy
mean(glm_predict==testset$diabetes)
[1] 0.8039216
```

Although this seems pretty good, we aren't quite done because there is an issue that is lurking under the hood. To see this, let's examine the information output from the model summary, in particular the coefficient estimates (i.e. estimates for β) and their significance. Here's a summary of the information contained in the table:

- Column 2 in the table lists coefficient estimates.
- Column 3 list s the standard error of the estimates (the larger the standard error, the less confident we are about the estimate)
- Column 4 the z statistic (which is the coefficient estimate (column 2) divided by the standard error of the estimate (column 3)) and
- The last column (Pr(>|z|) lists the p-value, which is the probability of getting the listed estimate assuming the predictor has no effect. In essence, the *smaller* the p-value, the *more significant* the estimate is likely to be.

From the table we can conclude that only 4 predictors are significant – *pregnant*, *glucose*, *mass* and *pedigree* (and possibly a fifth – *pressure*). The other variables have little predictive power and worse, may contribute to overfitting. They should, therefore, be eliminated and we'll do that in a minute. However, there's an important point to note before we do so…

In this case we have only 9 variables, so are able to identify the significant ones by a manual inspection of p-values. As you can well imagine, such a process will quickly become tedious as the number of predictors increases. Wouldn't it be be nice if there were an algorithm that could somehow automatically shrink the coefficients of these variables or (better!) set them to zero altogether? It turns out that this is precisely what lasso and its close cousin, ridge regression, do.

Ridge and Lasso

Recall that the values of the logistic regression coefficients β_0 and β_1 are found by minimising the negative log likelihood described in equation (3). Ridge and lasso regularization work by adding a *penalty term* to the log likelihood function. In the case of ridge regression, the penalty term is β_1^2 and in the case of lasso, it is $|\beta_1|$ (Remember, β_1 is a vector, with as many components as there are predictors). The quantity to be minimised in the two cases is thus:

$L + \lambda \sum \beta_1^2 .....(4)$ – for ridge regression,

and

$L + \lambda \sum |\beta_1| .....(5)$ – for lasso regression.

Where λ is a free parameter which is usually selected in such a way that the resulting model minimises the out of sample error. Typically, the optimal value of λ is found using grid search with cross-validation, a process akin to the one described in my discussion on cost-complexity parameter estimation in decision trees. Most canned algorithms provide methods to do this; the one we'll use in the next section is no exception.

In the case of ridge regression, the effect of the penalty term is to shrink the coefficients that contribute most to the error. Put another way, it reduces the magnitude of the coefficients that contribute to *increasing* L. In contrast, in the case of lasso regression, the effect of the penalty term is to set the these coefficients *exactly to zero!* This is cool because what it mean that lasso regression works like a feature selector that picks out the most important coefficients, i.e. those that are most predictive (and have the lowest p-values).

Let's illustrate this through an example. We'll use the glmnet package which implements a combined version of ridge and lasso (called elastic net). Instead of minimising (4) or (5) above, glmnet minimises:

$L + \lambda[(1-\alpha) \sum [\beta_1^2 + \alpha \sum |\beta_1|] ....(6)$

where α controls the "mix" of ridge and lasso regularisation, with $\alpha = 0$ being "pure" ridge and $\alpha = 1$ being "pure" lasso.

Lasso regularisation using glmnet

Let's reanalyse the Pima Indian Diabetes dataset using glmnet with $\alpha = 1$ (pure lasso). Before diving into code, it is worth noting that glmnet:

- does not have a formula interface, so one has to input the predictors as a matrix and the class labels as a vector.
- does not accept categorical predictors, so one has to convert these to numeric values before passing them to glmnet.

The glmnet function model.matrix creates the matrix and also converts categorical predictors to appropriate dummy variables.

Another important point to note is that we'll use the function cv.glmnet, which automatically performs a grid search to find the optimal value of λ.

OK, enough said, here we go:

```
#load required library
library(glmnet)
#convert training data to matrix format
x <- model.matrix(diabetes~.,trainset)
#convert class to numerical variable
```

```
y <- ifelse(trainset$diabetes==”pos”,1,0)
#perform grid search to find optimal value of lambda
#family= binomial => logistic regression, alpha=1 => lasso
# check docs to explore other type.measure options
cv.out <- cv.glmnet(x,y,alpha=1,family=”binomial”,type.measure = “mse” )
#plot result
plot(cv.out)
```

The plot is shown in Figure 2 below:

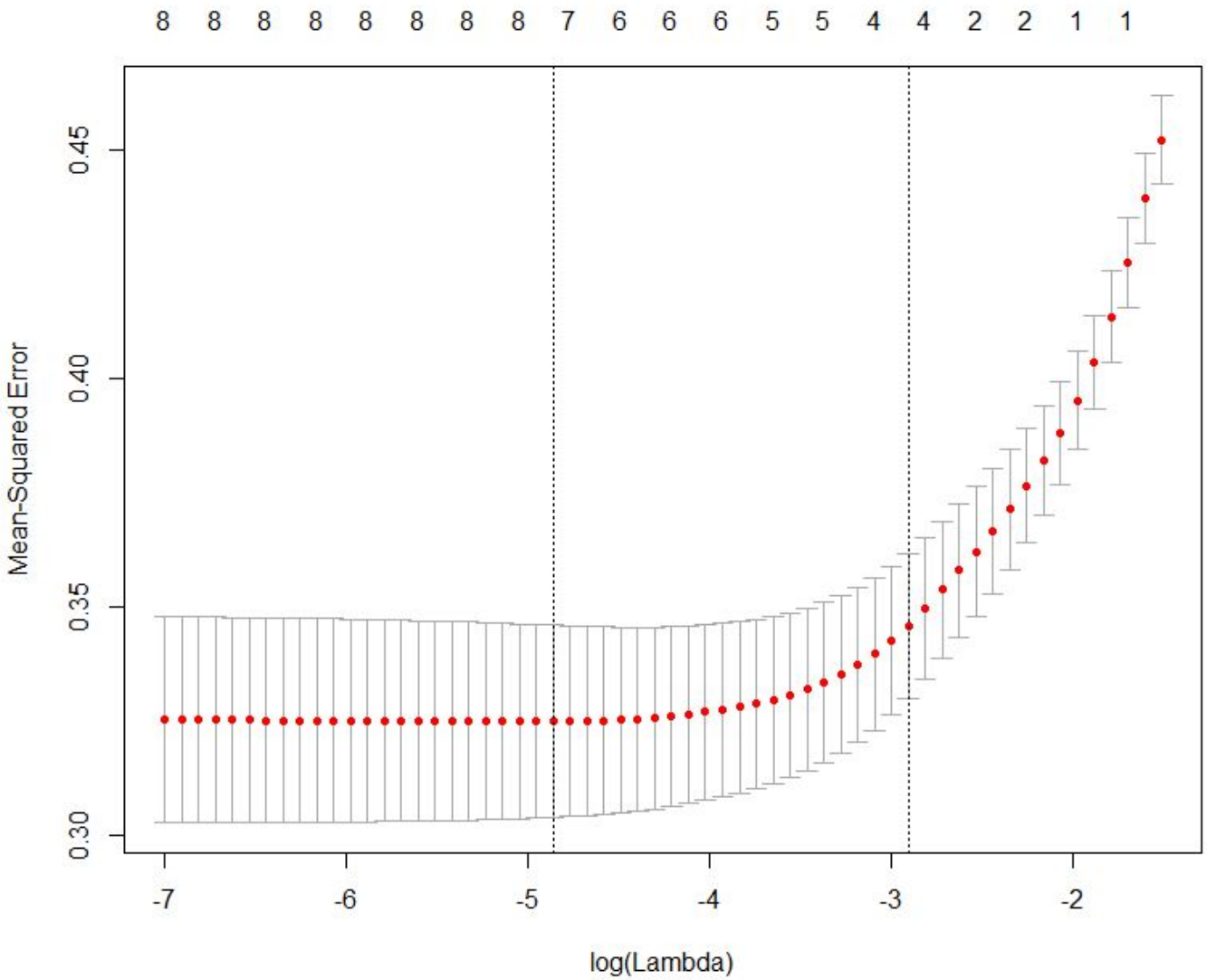

Figure 2: Error as a function of lambda (select lambda that minimises error)

The plot shows that the *log* of the optimal value of lambda (i.e. the one that minimises the root mean square error) is approximately -5. The exact value can be viewed by examining the variable *lambda_min* in the code below. In general though, the objective of regularisation is to balance accuracy *and* simplicity. In the present context, this means a model with the *smallest number of coefficients that also gives a good accuracy*. To this end, the cv.glmnet function finds the value of lambda that gives the simplest model but also lies within one standard error of the optimal value of lambda. This value of lambda (*lambda.1se*) is what we'll use in the rest of the computation. Interested readers should have a look at this article for more on *lambda.1se* vs *lambda.min*.

```
#min value of lambda
lambda_min <- cv.out$lambda.min
#best value of lambda
lambda_1se <- cv.out$lambda.1se
#regression coefficients
coef(cv.out,s=lambda_1se)
10 x 1 sparse Matrix of class "dgCMatrix"
                    1
(Intercept) -4.61706681
(Intercept)  .
pregnant     0.03077434
glucose      0.02314107
pressure     .
triceps      .
insulin      .
mass         0.02779252
pedigree     0.20999511
age          .
```

The output shows that *only* those variables that we had determined to be significant on the basis of p-values have non-zero coefficients. The coefficients of all other variables have been set to zero by the algorithm! Lasso has reduced the complexity of the fitting function massively…and you are no doubt wondering what effect this has on accuracy. Let's see by running the model against our test data:

```
#get test data
x_test <- model.matrix(diabetes~.,testset)
#predict class, type="class"
lasso_prob <- predict(cv.out,newx = x_test,s=lambda_1se,type="response")
#translate probabilities to predictions
lasso_predict <- rep("neg",nrow(testset))
lasso_predict[lasso_prob>.5] <- "pos"
#confusion matrix
table(pred=lasso_predict,true=testset$diabetes)
pred  neg pos
  neg 94 28
  pos  4 27
#accuracy
mean(lasso_predict==testset$diabetes)
[1] 0.7908497
```

Which is a bit less than what we got with the more complex model. So, we get a similar out-of-sample accuracy as we did before, and we do so using a way simpler function (4 non-zero coefficients) than the original one (9 nonzero coefficients). What this means is that the simpler function does at least as good a job fitting the signal in the data as the more complicated one. The bias-variance tradeoff tells us that the simpler function should be preferred because it is less likely to overfit the training data.

Paraphrasing William of Ockham: *all other things being equal, a simple hypothesis should be preferred over a complex one*.

Wrapping up

In this post I have tried to provide a detailed introduction to logistic regression, one of the simplest (and oldest) classification techniques in the machine learning practitioners arsenal. Despite it's simplicity (or I should say, because of it!) logistic regression works well for many business applications which often have a simple decision boundary. Moreover, because of its simplicity it is less prone to overfitting than flexible methods such as decision trees. Further, as we have shown, variables that contribute to overfitting can be eliminated using lasso (or ridge) regularisation, without compromising out-of-sample accuracy. Given these advantages and its inherent simplicity, it isn't surprising that logistic regression remains a workhorse for data scientists.

Written by K

July 11, 2017 at 10:00 pm

Posted in Data Analytics, Data Science, Predictive Analytics, Probability, R, Statistics

11 Responses

Subscribe to comments with RSS.

> […] In this day and age of artificial intelligence and deep learning, it is easy to forget that simple algorithms can work well for a surprisingly large range of practical business problems. And the simplest place to start is with the granddaddy of data science algorithms: linear regression and its close… Read more: A gentle introduction to logistic regression and lasso regularisation using R […]
>
> **A gentle introduction to logistic regression and lasso regularisation using R – Best Project Management Aggregators**
>
> July 12, 2017 at 11:35 am
>
> Reply
> Hi, Kailash
>
> Thanks for informative post.
>
> One remark concerning Occam's Razor and machine learning – it is not so unambiguous – have a look at the work of Geoff Webb – http://i.giwebb.com/index.php/research/occams-razor-in-machine-learning/
>
> Kind Regards
>
> Serhiy

Serhiy Yevtushenko

July 15, 2017 at 2:20 pm

Reply

good implementation, but I found one serious mistake. the second accuracy you obtained from lasso regression is lambda with 1se, not minimum. you should be getting accuracy of 70% by right.

youngseok jeon

September 11, 2017 at 6:13 pm

Reply

sorry I wrote wrong. the accuracy is for lambda with minumum. but the coeffs are for lambda with 1se.

youngseok jeon

September 11, 2017 at 6:17 pm

Reply

Hi Youngseok,

Many thanks for pointing this out, you're absolutely right. I have fixed this in the code and have added a bit of discussion around lambda.1se vs lambda.min.

Regards,

Kailash.

K

September 12, 2017 at 3:04 pm

Reply

I tried to repeat the code, but i get:

"Error in !se.fit : invalid argument type" – error, when i try to run:

lasso_prob <- predict.glm(glm_model,testset[,-typeColNum],s="lambda_1se",type="response")

And what is the purpose of line:

x_test <- model.matrix(diabetes~.,testset)

because variable x_test isn't used anywhere. (Or at least i can't find anywhere)

Raimo Haikari

September 16, 2017 at 5:06 am

Reply

Hi Raimo,

Thanks so much for pointing this out. I had used the incorrect predict function (the one for glm instead of glmnet). This has now been fixed. I've run the code from start to end in a clean environment and seems all good now. Please do let me know if you encounter any issues.

Thanks again, I truly appreciate it.

Regards,

Kailash.

K

September 16, 2017 at 9:13 am

Reply

Any advice on extended this for non-binary classification? I'm trying to use Logistic regression for a 3 class data set and am having very little success

Anna Ahern

October 6, 2017 at 1:51 am

Reply

Hi Anna,

This should be possible (though I have not tried it myself). See the section on *Multinomial Models* on page 23 of the glmnet vignette by Trevor Hastie:
https://web.stanford.edu/~hastie/Papers/Glmnet_Vignette.pdf

Hope this helps.

Regards,

Kailash.

K

October 6, 2017 at 11:35 am

Reply

Great post! I was just wondering if you use type ="response" as your logistic regression loss function measurement, why not use something similar in LASSO implementation. e.g use type ="link" or type = "response", given that this post is about logistic regression and MSE is really not an ideal loss function for logistic regression.

Jenny

October 26, 2017 at 12:58 am

Reply

Agree with Jenny, "deviance" should be a better loss in this case.

LazyBoo

January 11, 2018 at 9:20 am

Reply

This site uses Akismet to reduce spam. Learn how your comment data is processed.

Blog at WordPress.com.