Survival Analysis

Jason Jea

October 4, 2015

This is a guide for fitting a survival model to RMN app user data. Ideally, this will allow multiple people to tackle app lapse from many different angles.

At a high level, survival analysis attempts to estimate $P(T > t|X)$ where T is the time of of death and X is a vector of covariates.

For the case of RMN native app analysis, our T is the time of lapse after join, and our X vector can be any collection of characteristics and actions our users have taken.

One of the most commonly used survival models is the Cox-Proportional Hazards model. It allows us to estimate the multiplicative effects of covariates on the hazard ratio (the probability of death at any unit time).

I've decided to look at visit and outclick activity on the app for the first 28 days of lifecycle, and how that affects the survival curves for our userbase.

First to set up your R session:

```r
library(survival)
library(ggplot2)
library(plyr)
library(dplyr)
library(reshape2)
library(RPostgreSQL)
library(tidyr)
library(lubridate)
library(scales)
```

The **survival** package contains the main analysis toolsets we plan on using. It allows us to easily estimate survival curves and multiplicate coefficients for our covariates. **RPostgreSQL** allows us to connect directly to any Redshift database. These are the only two necessary packages. However, **plyr**, **dplyr**, **reshape2**, **tidyr**, and **lubridate** make the data cleaning and manipulation much easier, and **ggplot2** allows us much greater flexibility when it comes to visualization.

I've outlined how to set up your data connections in this wiki page

I like to have some semblance of structured data already created in my Redshift datamart (let's let databases do what they do best). In this case, my dataset is a table of app users with their first visit date, their last visit date, if they have lapsed or not, and then the page type and event type with the associate counts.

```r
dbGetQuery(redshift, "select * from bi_work.jj_app_lapse_predictions limit 5;")
```

```
##                                    udid lastvisitdate firstvisitdate
## 1 0eae4977-abad-3603-a53f-f5feb0abd6b3    2015-06-23     2015-05-30
## 2 12651e14-b38e-3218-a631-9523e7a82a1a    2015-05-10     2015-05-05
## 3 2ca135a7-c427-31b8-b166-acf99a14ce79    2015-09-05     2015-09-05
## 4 3ff25213-d42a-3414-9a92-555b6b8e2cac    2015-09-04     2015-07-11
## 5 3bcfeecf-fec3-3300-bf0d-e56c4440ae74    2015-08-28     2015-06-13
##   eventtype  pagetype count
## 1  pageView offerview     5
## 2  pageView       jfy     1
## 3  pageView coverflow     1
## 4  outClick  category     2
## 5  pageView  category    24
```

The data in that table has been exploded out so that each unique pagetype and eventype combination will occupy a row for each user. We want to pivot this data so that each of those combinations becomes a column. This will be necessary in order to feed the data into any sort of model. In order words we want:

```
##   udid  pagetype eventtype counts
## 1  abc coverflow  pageview      1
## 2  abc    nearby  pageview      4
## 3  abc      mall  outclick      3
```

to become:

```
##   udid coverflow_pageview mall_outclick nearby_pageview
## 1  abc                  1             3               4
```

To do this, we can use the very useful "Hadley-verse" packages: **dplyr**, **plyr**, **reshape2**, and **tidyr**.

One of my favorite pairs of functions from the **reshape2** package are the melt() and dcast() functions. In order to massage our data to look like the table above, we need to use dcast() function, which will essentially pivot your data.

```
dcast(data = test, udid ~ pagetype + eventtype, value.var = "counts")
```

The main arguments dcast() accepts is a dataframe, a formula that represents how you want to "cast" your data, and the name of the column which stores the values you want to pivot. You can also pass dcast() a aggregation function if the combination of your columns isn't distinct. The columns before the ~ are the columns you want to group by. Every distinct combination of values in the columns after the ~ will become a new column column name, and the cell values will become whatever you specified in the value.var argument.

The SQL equivalent of this would be:

```
select udid
, max(case when pagetype = 'coverflow' and eventtype = 'pageview' then counts else null end) as coverfl
, max(case when pagetype = 'mall' and eventtype = 'outclick' then counts else null end) as mall_outclic
, max(case when pagetype = 'nearby' and eventtype = 'pageview' then counts else null end) as nearby_pag
from test
group by 1
```

If we wanted to collapse our dataframe similar to what it was previously, we can use melt().

```
test <- dcast(data = test, udid ~ pagetype + eventtype, value.var = "counts")
melt(data = test, id.vars = c("udid"))
```

```
##   udid           variable value
## 1  abc coverflow_pageview     1
## 2  abc      mall_outclick     3
## 3  abc    nearby_pageview     4
```

Let's go back to our analysis data and perform the same operations such that each distinct event and page type gets transformed as a column name, with the associate counts as the cell values.

```
data <- dbGetQuery(redshift, "select * from bi_work.jj_app_lapse_predictions limit 10000;")
data <-
  dcast(data = data, udid + lastvisitdate + firstvisitdate ~ pagetype + eventtype, value.var = "count")
colnames(data)
```

```
##  [1] "udid"                            "lastvisitdate"
##  [3] "firstvisitdate"                  "category_coverflowImpression"
##  [5] "category_outClick"               "category_pageView"
##  [7] "coverflow_coverFlow"             "coverflow_coverflowImpression"
##  [9] "coverflow_outClick"              "coverflow_pageView"
## [11] "favorites_coverFlow"             "favorites_outClick"
## [13] "favorites_pageView"              "jfy_coverFlow"
## [15] "jfy_coverflowImpression"         "jfy_outClick"
## [17] "jfy_pageView"                    "mall_coverflowImpression"
## [19] "mall_outClick"                   "mall_pageView"
## [21] "nearby_coverflowImpression"      "nearby_outClick"
## [23] "nearby_pageView"                 "offerview_coverFlow"
## [25] "offerview_coverflowImpression" "offerview_outClick"
## [27] "offerview_pageView"              "products_coverflowImpression"
## [29] "products_pageView"               "search_coverflowImpression"
## [31] "search_pageView"                 "storepage_coverflowImpression"
## [33] "storepage_outClick"              "storepage_pageView"
## [35] "NA_coverFlow"                    "NA_coverflowImpression"
## [37] "NA_outClick"                     "NA_pageView"
## [39] "NA_NA"
```

One thing that dcast() does is if a particular udid doesn't have a pagetype + eventtype combination, it sets the cell value to NA. We want to replace all NA's with 0, which is very easy using the is.na() function. is.na() will take a vector *or* dataframe and return the same vector or dataframe with TRUE or FALSE values.

```
data[is.na(data)] <- 0
```

Our data is getting closer to being suitable for survival analysis. There are a few things left that we want to do. First, the **survival** package requires an event variable, usually specifying if a death happened. It also requires an interval variable, which is either the time until death, or the time until the last observation was taking (in the case of right censored data).
The analogous definitions for RMN app users is pretty straightforward. We want a variable that represents if a user is lapsed or still active, and then another variable that represents his lifespan.

```
data$lapsed = 1
data[data$lastvisitdate >= as.Date("2015-09-02"),]$lapsed <- 0
data <-
  mutate(data, intervalend = ifelse(lapsed == 1, as.numeric(lastvisitdate - firstvisitdate) + 1,
                              as.numeric(as.Date("2015-10-01") - firstvisitdate) + 1))
```

To create the lapsed column, we use traditional R bracket notation. However to create the interval column, We make use of the **dplyr** package here and its mutate() function. Creating new columns off of calculations in R can be cumbersome, but mutate allows us to cleanly assign any number of new variables to a dataframe.

There is one final thing that we can do that will make modeling easier. There are 21 potential covariates that we can use in our model. However, some of these covariates aren't actually real and were only artificially created when we used dcast() on our dataframe. I wrote a function that takes in a dataframe, identifies variables where every single record is 0, and then filters out those columns.

```
cleandata <-
  cbind(filterGoodVariables(data[,c(4:25)]), intervalend = data$intervalend, lapsed = data$lapsed)
colnames(cleandata)
```

```
##  [1] "category_coverflowImpression"  "category_outClick"
##  [3] "category_pageView"             "coverflow_coverFlow"
##  [5] "coverflow_coverflowImpression" "coverflow_outClick"
##  [7] "coverflow_pageView"            "favorites_coverFlow"
##  [9] "favorites_outClick"            "favorites_pageView"
## [11] "jfy_coverFlow"                 "jfy_coverflowImpression"
## [13] "jfy_outClick"                  "jfy_pageView"
## [15] "mall_coverflowImpression"      "mall_outClick"
## [17] "mall_pageView"                 "nearby_coverflowImpression"
## [19] "nearby_outClick"               "nearby_pageView"
## [21] "offerview_coverFlow"           "offerview_coverflowImpression"
## [23] "intervalend"                   "lapsed"
```

Our final dataframe will have all the covariates that we want to use in the model, in addition to the lapsed and intervalend variables.

All of this work so far was into cleaning our dataset in prepartion for the modeling. Next, we can actually estimate our native app survival curves and also the proportional effect of our covariates on the hazard ratio. For those familiar with regression modeling in R, the **Cox-Proportional Hazards** model is specified in a very similar fashion. The only difference is that, whereas in a typical regression model the response variable only takes in one column name as a paramter, the CoxPH model takes in a **Surv** object. A **Surv** object is defined by two parameters: first the interval variable, and then the event variable. The independent variables are specified in the exact same fashion.

```
model1 <- coxph(Surv(cleandata$intervalend, cleandata$lapsed) ~ .,
                      data = cleandata[,1:22])
```

```
## Warning in fitter(X, Y, strats, offset, init, control, weights = weights, :
## Loglik converged before variable 15 ; beta may be infinite.
```

The warning message is most commonly generated if we have certain variables that all have the exact same value. Fortunately, the cleaning we did earlier takes care of that, so we can ignore the warning.

We can take a look at the summary of the model in order to get a better idea of coefficients, significance levels, and confidence intervals. The coefficients are interpretated like any other multiplicative model i.e. logistic regression, and are the relative effects on the **hazard ratio**, or more specifically, the risk of lapsing.

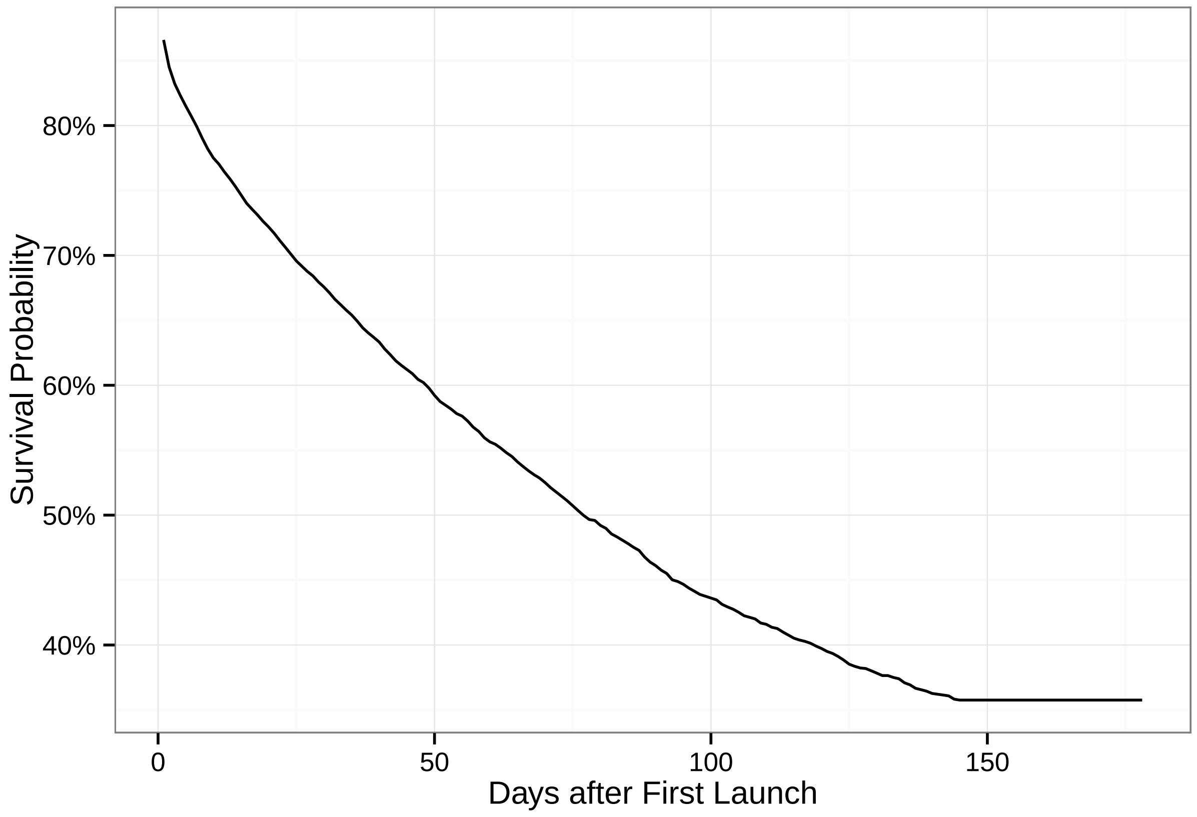
```
summary(model1)
```

```
## Call:
## coxph(formula = Surv(cleandata$intervalend, cleandata$lapsed) ~
##     ., data = cleandata[, 1:22])
##
##   n= 9426, number of events= 4410
##
##                                     coef  exp(coef)   se(coef)      z
## category_coverflowImpression   2.315e-02  1.023e+00  2.000e-01  0.116
## category_outClick              1.161e-01  1.123e+00  8.696e-02  1.335
## category_pageView            -1.624e-02  9.839e-01  9.940e-03 -1.633
## coverflow_coverFlow          -5.146e-02  9.498e-01  1.926e-02 -2.671
## coverflow_coverflowImpression -4.858e-03  9.952e-01  9.100e-04 -5.339
## coverflow_outClick           -2.419e-01  7.851e-01  5.415e-02 -4.467
## coverflow_pageView           -1.389e-02  9.862e-01  3.958e-03 -3.509
## favorites_coverFlow           1.240e+00  3.455e+00  1.000e+00  1.239
## favorites_outClick           -9.699e-01  3.791e-01  7.074e-01 -1.371
## favorites_pageView           -2.684e-02  9.735e-01  1.008e-02 -2.662
## jfy_coverFlow                 9.216e-01  2.513e+00  7.074e-01  1.303
## jfy_coverflowImpression      -2.410e-02  9.762e-01  9.447e-02 -0.255
## jfy_outClick                 -2.542e-01  7.755e-01  2.354e-01 -1.080
## jfy_pageView                  1.435e-04  1.000e+00  1.650e-02  0.009
## mall_coverflowImpression     -1.160e+01  9.163e-06  3.117e+02 -0.037
## mall_outClick                -1.545e-01  8.568e-01  1.851e-01 -0.835
## mall_pageView                -7.896e-02  9.241e-01  2.058e-02 -3.837
## nearby_coverflowImpression    1.845e+00  6.326e+00  7.075e-01  2.607
## nearby_outClick               6.278e-02  1.065e+00  2.334e-01  0.269
## nearby_pageView              -2.445e-02  9.758e-01  1.326e-02 -1.844
## offerview_coverFlow          -1.641e-01  8.487e-01  6.253e-02 -2.625
## offerview_coverflowImpression -2.639e-01  7.681e-01  1.283e-01 -2.057
##                              Pr(>|z|)
## category_coverflowImpression  0.907849
## category_outClick            0.181968
## category_pageView            0.102391
## coverflow_coverFlow          0.007555 **
## coverflow_coverflowImpression 9.35e-08 ***
## coverflow_outClick           7.93e-06 ***
## coverflow_pageView           0.000449 ***
## favorites_coverFlow          0.215209
## favorites_outClick           0.170305
## favorites_pageView           0.007761 **
## jfy_coverFlow                0.192644
## jfy_coverflowImpression      0.798644
## jfy_outClick                 0.280092
## jfy_pageView                 0.993060
## mall_coverflowImpression     0.970311
## mall_outClick                0.403964
## mall_pageView                0.000125 ***
## nearby_coverflowImpression   0.009130 **
## nearby_outClick              0.787914
## nearby_pageView              0.065168 .
## offerview_coverFlow          0.008674 **
## offerview_coverflowImpression 0.039697 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##                              exp(coef) exp(-coef)  lower .95  upper .95
## category_coverflowImpression  1.023e+00  9.771e-01  6.915e-01  1.515e+00
## category_outClick            1.123e+00  8.904e-01  9.471e-01  1.332e+00
## category_pageView            9.839e-01  1.016e+00  9.649e-01  1.003e+00
## coverflow_coverFlow          9.498e-01  1.053e+00  9.147e-01  9.864e-01
## coverflow_coverflowImpression 9.952e-01  1.005e+00  9.934e-01  9.969e-01
## coverflow_outClick           7.851e-01  1.274e+00  7.061e-01  8.731e-01
## coverflow_pageView           9.862e-01  1.014e+00  9.786e-01  9.939e-01
## favorites_coverFlow           3.455e+00  2.895e-01  4.864e-01  2.454e+01
## favorites_outClick            3.791e-01  2.638e+00  9.477e-02  1.517e+00
## favorites_pageView            9.735e-01  1.027e+00  9.545e-01  9.929e-01
## jfy_coverFlow                 2.513e+00  3.979e-01  6.282e-01  1.005e+01
## jfy_coverflowImpression       9.762e-01  1.024e+00  8.112e-01  1.175e+00
## jfy_outClick                  7.755e-01  1.289e+00  4.889e-01  1.230e+00
## jfy_pageView                  1.000e+00  9.999e-01  9.683e-01  1.033e+00
## mall_coverflowImpression      9.163e-06  1.091e+05 4.545e-271 1.848e+260
## mall_outClick                 8.568e-01  1.167e+00  5.961e-01  1.232e+00
## mall_pageView                 9.241e-01  1.082e+00  8.875e-01  9.621e-01
## nearby_coverflowImpression    6.326e+00  1.581e-01  1.581e+00  2.532e+01
## nearby_outClick               1.065e+00  9.391e-01  6.739e-01  1.682e+00
## nearby_pageView               9.758e-01  1.025e+00  9.508e-01  1.002e+00
## offerview_coverFlow           8.487e-01  1.178e+00  7.508e-01  9.593e-01
## offerview_coverflowImpression 7.681e-01  1.302e+00  5.973e-01  9.876e-01
##
## Concordance= 0.528  (se = 0.005 )
## Rsquare= 0.014   (max possible= 1 )
## Likelihood ratio test= 136.9  on 22 df,   p=0
## Wald test            = 108  on 22 df,   p=2.518e-13
## Score (logrank) test = 111.6  on 22 df,   p=5.795e-14
```

We can see for example, that the more mall pageviews a user has in his first 28 days, all other things held constant, the lower the risk is of lapsing.

Of highest interest to everyone, of course, are the actual survival curves. The **survival** library allows us to estimate survival curves. The default curve generated is the base survival curve for the average user, but we'll see later that it's very easy to visualize the effects of certain covariates by fitting newly generated artifical datasets.

```r
survivalfit = survfit(model1)

data.frame(cbind(time = survivalfit[[2]], survivalrate = survivalfit[[6]])) %>%
ggplot(aes(x=time,y=survivalrate))+geom_line() +
  theme_bw() + scale_x_continuous(name = "Days after First Launch") + scale_y_continuous(labels = percen
```



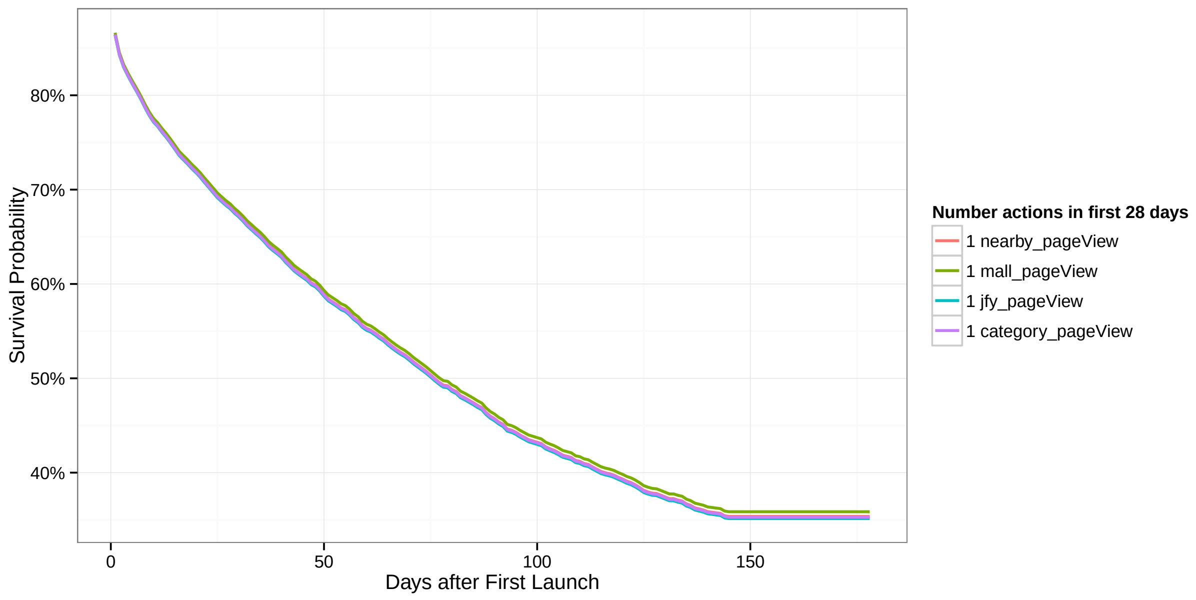
The **survfit** object contains a number of attributes, the most important being the survival probabilites at each unit time. We can extract out that curve and plot it using **ggplot2**.

In order to visualize the effects of covariates on the survival curve, the **survival** package allows us to generate multiple curves based on a dataframe passed as a parameter. The columns of the dataframe need to match the covariates of the model, and each row will generate a different curve. What we can do then, is generate a dataframe that has the average values of each covariate. In order the compare the effect of two covariates, we need to have row where covariate $\mathbf{A} = 0$ and covariate $\mathbf{B} = 1$, and then another row where $A = 1$ and $B = 1$. This can be generalized to compare many different covariates. Additionally, you can also look at the effect of an increase in X units of one covariate in a similar method.

I wrote two functions: **oneVarSurvData()** automatically generates the survival curves of **n** different values of one covariate. **compareVarSurvData()** automatically generates the survival curves of **n** different covariates, given one value to compare between.

```
oneVarSurvData <- function(model, data, metric, values) {
  avgdata <- colwise(mean)(data)
  avgdata <- avgdata[,names(avgdata) != metric]

  finaldata <-
    data.frame(avgdata[rep(seq_len(nrow(avgdata)),length(values)),],
               newcol = values)

  names(finaldata)[names(finaldata) == "newcol"] <- metric

  fit <- survfit(model, newdata = finaldata)
  plot.data <-
    data.frame(cbind(time = fit[[2]], survivalrate = fit[[6]]))

  names(plot.data)[1:length(values) + 1] <- paste(values, metric, sep = " ")

  plot.data <-
    plot.data %>% melt("time") %>% dcast(time~variable)

  return(plot.data)

}

compareVarSurvData <- function(model, data, metrics, value) {
  avgdata <- colwise(mean)(data)
  avgdata <-
    cbind(avgdata[rep(seq_len(nrow(avgdata)),length(metrics)),], seqnum = seq(1:length(metrics))) %>% me

  metrics <- data.frame(cbind(metrics, seqnum = seq(1:length(metrics))))

  avgdata[avgdata$variable %in% metrics$metrics,]$value <- 0

  avgdata <-
    ddply(metrics, 1, function(metrics) {

      avgdata[avgdata$seqnum == as.numeric(metrics$seqnum) & avgdata$variable == as.character(metrics$me
        avgdata[avgdata$seqnum == as.numeric(metrics$seqnum) & avgdata$variable == as.character(metrics$

      return(avgdata)
    })

  avgdata <-
    avgdata[,2:4] %>%
    dcast(., seqnum~variable, fun = mean)

  fit <- survfit(model, newdata = avgdata[2:ncol(avgdata)])

  plot.data <-
    data.frame(cbind(time = fit[[2]], survivalrate = fit[[6]]))

  names(plot.data)[1:nrow(metrics) + 1] <- paste(value, metrics$metrics, sep = " ")

  plot.data <-
    plot.data %>% melt("time") %>% dcast(time~variable)

  return(plot.data)

}
```

```
compareVarSurvData(model1, cleandata[,1:22], c("nearby_pageView", "mall_pageView","jfy_pageView","catego
  ggplot(aes(x = time, y = value, colour = variable)) + geom_line(size = .8) + theme_bw() +
  scale_y_continuous(labels = percent, name = "Survival Probability") + scale_x_continuous(name = "Days
  scale_colour_discrete(name = "Number actions in first 28 days")
```



For the particular variables we chose, we can see that the subsequent effects on the surival curve is very similar. However, we can also see that having one pageview to the mall page has the most positive impact on survivability.