Market Basket Analysis R

Manmeet Bains

#Loading Libraries

library(recommenderlab)

## Loading required package: Matrix

## Loading required package: arules

## Warning: package 'arules' was built under R version 3.4.4

##   
## Attaching package: 'arules'

## The following objects are masked from 'package:base':  
##   
## abbreviate, write

## Loading required package: proxy

## Warning: package 'proxy' was built under R version 3.4.4

##   
## Attaching package: 'proxy'

## The following object is masked from 'package:Matrix':  
##   
## as.matrix

## The following objects are masked from 'package:stats':  
##   
## as.dist, dist

## The following object is masked from 'package:base':  
##   
## as.matrix

## Loading required package: registry

## Warning: package 'registry' was built under R version 3.4.4

library(tidyverse)

## Warning: package 'tidyverse' was built under R version 3.4.4

## -- Attaching packages --------------------------------------------------------------------------------------------------- tidyverse 1.2.1 --

## v ggplot2 3.0.0 v purrr 0.2.5  
## v tibble 2.0.1 v dplyr 0.7.6  
## v tidyr 0.8.1 v stringr 1.3.1  
## v readr 1.1.1 v forcats 0.3.0

## Warning: package 'ggplot2' was built under R version 3.4.4

## Warning: package 'tibble' was built under R version 3.4.4

## Warning: package 'tidyr' was built under R version 3.4.4

## Warning: package 'readr' was built under R version 3.4.4

## Warning: package 'purrr' was built under R version 3.4.4

## Warning: package 'dplyr' was built under R version 3.4.4

## Warning: package 'stringr' was built under R version 3.4.4

## Warning: package 'forcats' was built under R version 3.4.4

## -- Conflicts ------------------------------------------------------------------------------------------------------ tidyverse\_conflicts() --  
## x tidyr::expand() masks Matrix::expand()  
## x dplyr::filter() masks stats::filter()  
## x dplyr::lag() masks stats::lag()  
## x dplyr::recode() masks arules::recode()

library(tidyquant)

## Warning: package 'tidyquant' was built under R version 3.4.4

## Loading required package: lubridate

## Warning: package 'lubridate' was built under R version 3.4.4

##   
## Attaching package: 'lubridate'

## The following object is masked from 'package:base':  
##   
## date

## Loading required package: PerformanceAnalytics

## Warning: package 'PerformanceAnalytics' was built under R version 3.4.4

## Loading required package: xts

## Warning: package 'xts' was built under R version 3.4.4

## Loading required package: zoo

## Warning: package 'zoo' was built under R version 3.4.4

##   
## Attaching package: 'zoo'

## The following objects are masked from 'package:base':  
##   
## as.Date, as.Date.numeric

##   
## Attaching package: 'xts'

## The following objects are masked from 'package:dplyr':  
##   
## first, last

##   
## Attaching package: 'PerformanceAnalytics'

## The following object is masked from 'package:graphics':  
##   
## legend

## Loading required package: quantmod

## Warning: package 'quantmod' was built under R version 3.4.4

## Loading required package: TTR

## Warning: package 'TTR' was built under R version 3.4.4

## Version 0.4-0 included new data defaults. See ?getSymbols.

library(fs)

## Warning: package 'fs' was built under R version 3.4.4

library(knitr)

## Warning: package 'knitr' was built under R version 3.4.4

library(glue)

## Warning: package 'glue' was built under R version 3.4.4

##   
## Attaching package: 'glue'

## The following object is masked from 'package:dplyr':  
##   
## collapse

library(cowplot)

## Warning: package 'cowplot' was built under R version 3.4.4

##   
## Attaching package: 'cowplot'

## The following object is masked from 'package:ggplot2':  
##   
## ggsave

#Loading Data—-

#Building Read Directory Function

read\_directory\_to\_list <- function(path, .f = read\_csv, ...) {  
   
 names\_vec <- dir\_ls(path, ...) %>%  
 str\_split("/", simplify = T) %>%  
 .[,ncol(.)] %>%  
 path\_ext\_remove()  
   
 ret\_list <- dir\_ls(path) %>%  
 map(.f) %>%  
 set\_names(names\_vec)  
   
 return(ret\_list)  
   
}

instacart\_raw\_list <- read\_directory\_to\_list("../00\_Data")

## Parsed with column specification:  
## cols(  
## aisle\_id = col\_integer(),  
## aisle = col\_character()  
## )

## Parsed with column specification:  
## cols(  
## department\_id = col\_integer(),  
## department = col\_character()  
## )

## Parsed with column specification:  
## cols(  
## `[.ShellClassInfo]` = col\_character()  
## )

## Parsed with column specification:  
## cols(  
## order\_id = col\_integer(),  
## user\_id = col\_integer(),  
## eval\_set = col\_character(),  
## order\_number = col\_integer(),  
## order\_dow = col\_integer(),  
## order\_hour\_of\_day = col\_character(),  
## days\_since\_prior\_order = col\_double()  
## )

## Parsed with column specification:  
## cols(  
## order\_id = col\_integer(),  
## product\_id = col\_integer(),  
## add\_to\_cart\_order = col\_integer(),  
## reordered = col\_integer()  
## )  
## Parsed with column specification:  
## cols(  
## order\_id = col\_integer(),  
## product\_id = col\_integer(),  
## add\_to\_cart\_order = col\_integer(),  
## reordered = col\_integer()  
## )

## Parsed with column specification:  
## cols(  
## product\_id = col\_integer(),  
## product\_name = col\_character(),  
## aisle\_id = col\_integer(),  
## department\_id = col\_integer()  
## )

The element names for the data (tibbles) stored in instacart\_raw\_list.

instacart\_raw\_list %>% names()

## [1] "aisles" "departments" "desktop"   
## [4] "orders" "order\_products\_\_prior" "order\_products\_\_train"  
## [7] "products"

We’ll create a market basket from the orders by combining "order\_products\_\_prior" and the “products” data. We can see that it’s *32.4M rows*, which is a lot of data to deal with. Creating a recommender with this size data set would be a lot to handle.

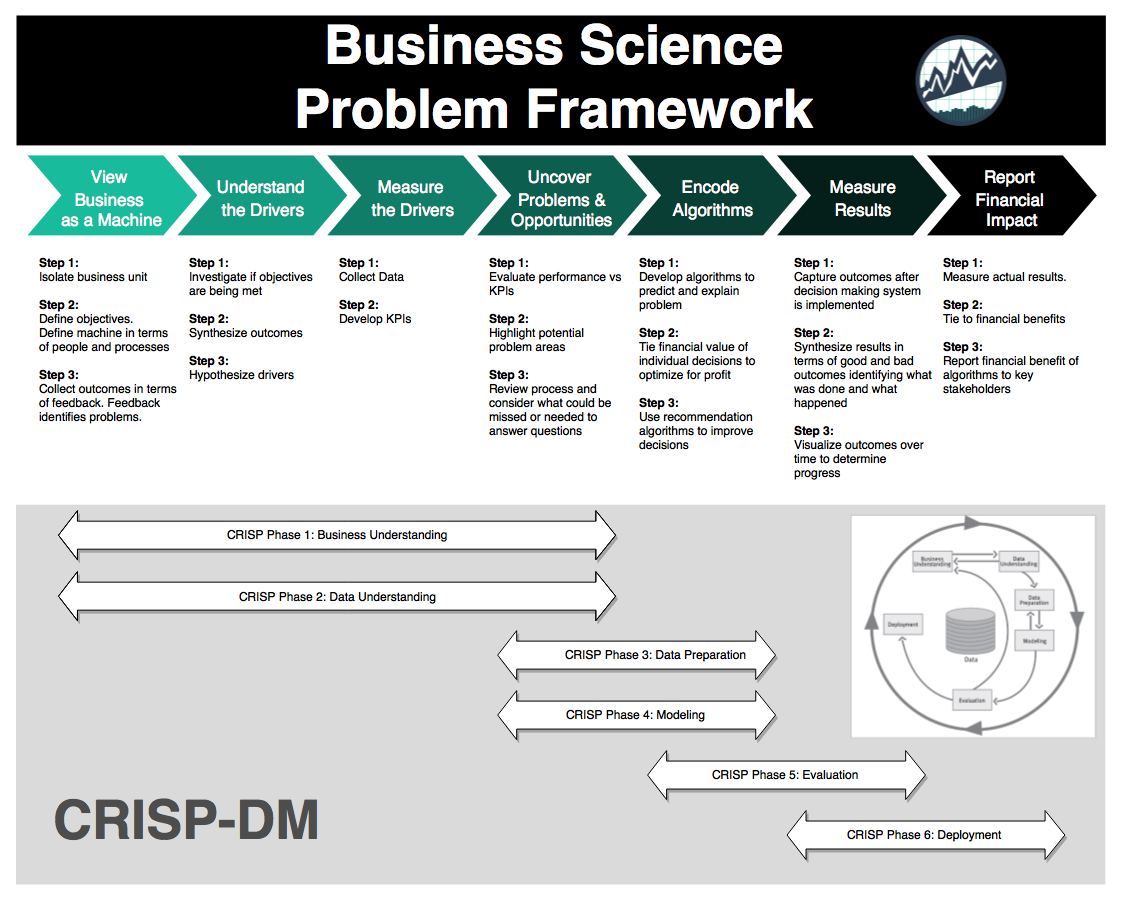
market\_basket\_tbl <- instacart\_raw\_list$order\_products\_\_prior %>%  
 inner\_join(instacart\_raw\_list$products, by = "product\_id")  
  
market\_basket\_tbl %>%  
 select(order\_id, product\_name)

## # A tibble: 32,434,489 x 2  
## order\_id product\_name   
## <int> <chr>   
## 1 2 Organic Egg Whites   
## 2 2 Michigan Organic Kale   
## 3 2 Garlic Powder   
## 4 2 Coconut Butter   
## 5 2 Natural Sweetener   
## 6 2 Carrots   
## 7 2 Original Unflavored Gelatine Mix   
## 8 2 All Natural No Stir Creamy Almond Butter   
## 9 2 Classic Blend Cole Slaw   
## 10 3 Total 2% with Strawberry Lowfat Greek Strained Yogurt  
## # ... with 32,434,479 more rows

## CRISP-DM / BSPF Data Science Process

Following CRISP-DM Framework

include\_graphics("../00\_Images/Business Science Problem Framework.png")



### 1. Business Understanding

The business problem is that we can potentially derive more sales from our customers by recommending products they are likely to want. Often the customer goes to the store or visits the website for a specific reason, but for the organization this is a prime opportunity to increase sales by recommending products the customer may not be thinking about

### 2. Data Understanding

One of the easiest ways to understand a *market basket* is by looking at how frequently items are purchased. We can use the count() function along with some calculations to understand which items are popular:

* Percentage of total “pct”
* Cumulative percentage of total “cumulative\_pct”
* Popular product which we define somewhat arbitrarily as less than or equal to 50% cumulative percent

item\_frequency\_tbl <- market\_basket\_tbl %>%  
 count(product\_name) %>%  
 arrange(desc(n)) %>%  
 mutate(  
 pct = n / sum(n),  
 cumulative\_pct = cumsum(pct),  
 popular\_product = ifelse(cumulative\_pct <= 0.5, "Yes", "No")  
 )

## Warning: package 'bindrcpp' was built under R version 3.4.4

item\_frequency\_tbl

## # A tibble: 49,677 x 5  
## product\_name n pct cumulative\_pct popular\_product  
## <chr> <int> <dbl> <dbl> <chr>   
## 1 Banana 472565 0.0146 0.0146 Yes   
## 2 Bag of Organic Bananas 379450 0.0117 0.0263 Yes   
## 3 Organic Strawberries 264683 0.00816 0.0344 Yes   
## 4 Organic Baby Spinach 241921 0.00746 0.0419 Yes   
## 5 Organic Hass Avocado 213584 0.00659 0.0485 Yes   
## 6 Organic Avocado 176815 0.00545 0.0539 Yes   
## 7 Large Lemon 152657 0.00471 0.0586 Yes   
## 8 Strawberries 142951 0.00441 0.0630 Yes   
## 9 Limes 140627 0.00434 0.0674 Yes   
## 10 Organic Whole Milk 137905 0.00425 0.0716 Yes   
## # ... with 49,667 more rows

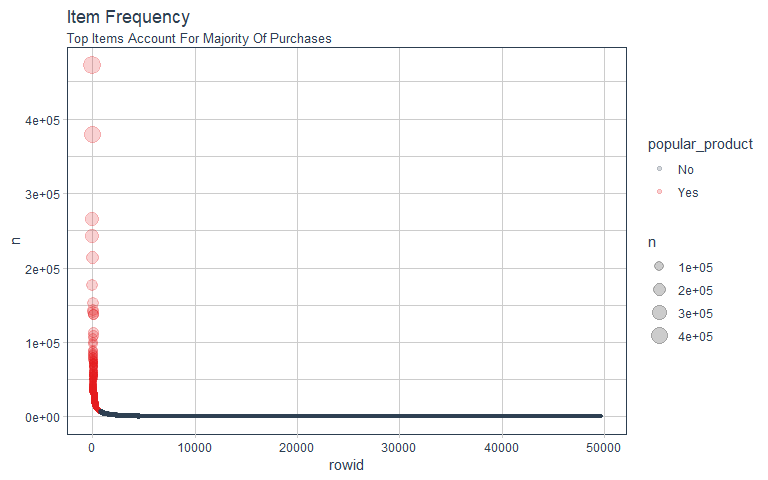
We can see a few things from this table:

1. The top item (1.45% of purchases) is Bananas
2. There are almost 50K items, which is a lot to handle via a ratings matrix (discussed in data preparation)

Let’s visualize to see what we are dealing with. We’ll use ggplot2 to help some of the interesting aspects of the market basket stand out.

* We’ll set the size = n, which increases the size of the point based on how frequently it is purchased.
* We’ll set the color = popular\_product, which separates items in the top 50% of purchase frequency from items in the bottom 50%.

item\_frequency\_tbl %>%  
 rowid\_to\_column() %>%  
 ggplot(aes(rowid, n)) +  
 geom\_point(aes(size = n, color = popular\_product), alpha = 0.2) +  
 theme\_tq() +  
 scale\_color\_tq() +  
 theme(legend.direction = "vertical",   
 legend.position = "right") +  
 labs(title = "Item Frequency",   
 subtitle = "Top Items Account For Majority Of Purchases")



From the visualization, we immediately see that the data is highly skewed. The top 50% of purchases, sum(n), are derived from **only 786** of the almost 50,000 items. This is less than 1.6% of the total products (items).

item\_frequency\_tbl %>%  
 count(popular\_product) %>%  
 mutate(pct = nn / sum(nn))

## # A tibble: 2 x 3  
## popular\_product nn pct  
## <chr> <int> <dbl>  
## 1 No 48891 0.984   
## 2 Yes 786 0.0158

### 3. Data Preparation

We’ll need to create a **ratings matrix**, which has the purchase history formatted in a 2x2 matrix with rows being orders and columns being products. This format is often called a **user-item matrix** because users (e.g. customers or orders) tend to be on the rows and items (e.g. products) in the columns.

The ratings matrix can be extraordinarily sparse given we have 32M+ data points with 50K products. Further, many of these data points are not *meaningful*. We saw that the data is highly skewed, which indicates that the lowest frequency items can likely be discarded because these are by definition “unpopular”. We can plan our data preparation by:

* Taking advantage of item popularity. A small proportion of the products are driving a large proportion of the purchase frequencies. By limiting to the top items, we can reduce the width of the ratings matrix making it much easier to handle without losing much.
* Further reduce the height of the matrix through sampling. We can sample 20,000 orders to make the ratings matrix more manageable. Further, we can limit the market baskets to those with at least 3 popular items, which ensures similarities between multiple items.

First, let’s filter to only the products that are popular, meaning the top products that drive 50% of the purchases.

# Get names of top products  
top\_products\_vec <- item\_frequency\_tbl %>%  
 filter(popular\_product == "Yes") %>%  
 pull(product\_name)  
  
# Use names to filter   
top\_products\_basket\_tbl <- market\_basket\_tbl %>%  
 filter(product\_name %in% top\_products\_vec) %>%  
 select(order\_id, product\_name)  
  
top\_products\_basket\_tbl

## # A tibble: 16,213,256 x 2  
## order\_id product\_name   
## <int> <chr>   
## 1 2 Organic Egg Whites   
## 2 2 Michigan Organic Kale   
## 3 2 Carrots   
## 4 3 Total 2% with Strawberry Lowfat Greek Strained Yogurt  
## 5 3 Unsweetened Almondmilk   
## 6 3 Organic Baby Spinach   
## 7 3 Organic Ginger Root   
## 8 3 Air Chilled Organic Boneless Skinless Chicken Breasts  
## 9 4 Plain Pre-Sliced Bagels   
## 10 4 Goldfish Cheddar Baked Snack Crackers   
## # ... with 16,213,246 more rows

Next, let’s sample 20,000 orders and then filter to those with baskets of at least 3 popular items. Note that it could take a while to filter first due to the aggregation step. The downside is that we end up with less than 20,000 total samples. If desired, we could increase the sample size further.

n\_sample <- 20000  
min\_items <- 3  
  
set.seed(100)  
sample\_order\_ids <- sample(unique(top\_products\_basket\_tbl$order\_id), size = n\_sample)  
  
top\_products\_sample\_tbl <- top\_products\_basket\_tbl %>%  
 # Sample orders  
 filter(order\_id %in% sample\_order\_ids) %>%  
 # Filter using min\_items  
 group\_by(order\_id) %>%  
 filter(n() >= min\_items) %>%  
 ungroup()  
  
top\_products\_sample\_tbl

## # A tibble: 102,599 x 2  
## order\_id product\_name   
## <int> <chr>   
## 1 660 Banana   
## 2 660 Unsweetened Vanilla Almond Milk   
## 3 660 Original Pure Creamy Almond Milk  
## 4 660 Green Seedless Grapes   
## 5 660 Organic Half & Half   
## 6 724 Red Plums   
## 7 724 Organic Raspberries   
## 8 724 Organic Strawberries   
## 9 724 Organic Hass Avocado   
## 10 803 Banana   
## # ... with 102,589 more rows

Last, convert the sampled market baskets to a ratings matrix in the format that recommenderlab uses. The type of ratings matrix is a **“binary rating matrix”**, which consists of 0’s and 1’s indicating whether or not a product was purchased.

ratings\_matrix\_rlab <- top\_products\_sample\_tbl %>%  
 # Spread into user-item format  
 mutate(value = 1) %>%  
 spread(product\_name, value, fill = 0) %>%  
 # Convert to matrix  
 select(-order\_id) %>%  
 as.matrix() %>%  
 # Convert to binaryRatingsMatrix class used by recommenderlab  
 as("binaryRatingMatrix")  
  
ratings\_matrix\_rlab

## 14151 x 786 rating matrix of class 'binaryRatingMatrix' with 102599 ratings.

A second type of ratings matrix, a **“real rating matrix”** consisting of actual user ratings (e.g. Netflix movie ratings on a 1 to 5 scale), is permitted. This format must be normalized, which can be done using the normalize() function. Because we are working with binary data, no normalization is necessary.

### 4. Modeling

The recommenderlab package makes it easy to test multiple algorithms to quickly determine which are promising. We’ll test out several that can be used with binary 0-1 data. We can review the available recommenderlab algorithms for class “binaryRatingMatrix” using the recommenderRegistry$get\_entries() function. By supplying the argument, datatype = "binaryRatingMatrix", we can get only those that pertain to a 1-0 ratings problem. These include:

* ALS: Alternating Least Squares (Not discussed as part of this analysis due to issues with the recommenderlab implementation of the algorithm)
* AR: Association Rules
* IBCF: Item-Based Collaborative Filtering
* Popular: Popularity-Based Recommendations
* Random: A useful baseline case to determine effectiveness
* UBCF: User-Based Collaborative Filtering

recommenderRegistry$get\_entries(dataType = "binaryRatingMatrix")

## $ALS\_implicit\_binaryRatingMatrix  
## Recommender method: ALS\_implicit for binaryRatingMatrix  
## Description: Recommender for implicit data based on latent factors, calculated by alternating least squares algorithm.  
## Reference: Yifan Hu, Yehuda Koren, Chris Volinsky (2008). Collaborative Filtering for Implicit Feedback Datasets, ICDM '08 Proceedings of the 2008 Eighth IEEE International Conference on Data Mining, pages 263-272.  
## Parameters:  
## lambda alpha n\_factors n\_iterations min\_item\_nr seed  
## 1 0.1 10 10 10 1 NULL  
##   
## $AR\_binaryRatingMatrix  
## Recommender method: AR for binaryRatingMatrix  
## Description: Recommender based on association rules.  
## Reference: NA  
## Parameters:  
## support confidence maxlen sort\_measure sort\_decreasing apriori\_control  
## 1 0.1 0.8 3 "confidence" TRUE list()  
## verbose  
## 1 FALSE  
##   
## $IBCF\_binaryRatingMatrix  
## Recommender method: IBCF for binaryRatingMatrix  
## Description: Recommender based on item-based collaborative filtering (binary rating data).  
## Reference: NA  
## Parameters:  
## k method normalize\_sim\_matrix alpha  
## 1 30 "Jaccard" FALSE 0.5  
##   
## $POPULAR\_binaryRatingMatrix  
## Recommender method: POPULAR for binaryRatingMatrix  
## Description: Recommender based on item popularity.  
## Reference: NA  
## Parameters: None  
##   
## $RANDOM\_binaryRatingMatrix  
## Recommender method: RANDOM for binaryRatingMatrix  
## Description: Produce random recommendations (binary ratings).  
## Reference: NA  
## Parameters: None  
##   
## $RERECOMMEND\_binaryRatingMatrix  
## Recommender method: RERECOMMEND for binaryRatingMatrix  
## Description: Re-recommends items (binary ratings).  
## Reference: NA  
## Parameters:  
## data frame with 0 columns and 0 rows  
##   
## $UBCF\_binaryRatingMatrix  
## Recommender method: UBCF for binaryRatingMatrix  
## Description: Recommender based on user-based collaborative filtering.  
## Reference: NA  
## Parameters:  
## method nn weighted sample  
## 1 "jaccard" 25 TRUE FALSE

#### 4.1 Training/Test Split

The data should be separated into training and testing sets if we wish to determine the effectiveness. We can split into training and test sets using the evaluationScheme() function. We can also setup 5-fold cross validation using k = 5. However, to keep processing time low, we will select k = NULL. Setting given = -1 means that all but 1 item will be used for learning and the remaining item will be used for evaluation.

eval\_scheme <- ratings\_matrix\_rlab %>%   
 evaluationScheme(method = "split", train = 0.9, k = NULL, given = -1)  
  
eval\_scheme

## Evaluation scheme using all-but-1 items  
## Method: 'split' with 1 run(s).  
## Training set proportion: 0.900  
## Good ratings: NA  
## Data set: 14151 x 786 rating matrix of class 'binaryRatingMatrix' with 102599 ratings.

#### 4.2 Algorithms

To implement multiple modeling algorithms, we can setup a list of algorithms in a format that can be used by the evaluate() function from the recommenderlab package. Note that we exclude Alternating Least Squares (ALS) because it fail due to a “matrix subsetting issue”.

algorithms\_list <- list(  
 "random items" = list(name = "RANDOM",   
 param = NULL),  
 "popular items" = list(name = "POPULAR",   
 param = NULL),  
 "user-based CF" = list(name = "UBCF",   
 param = list(method = "Cosine", nn = 500)),  
 "item-based CF" = list(name = "IBCF",   
 param = list(k = 5)),  
 "association rules" = list(name = "AR",   
 param = list(supp = 0.01, conf = 0.01))  
)

#### 4.3 Evaluate Scheme

Next, we can process the algorithms using the evaluate() function. This will take a minute to run. We specify the type = "topNList" to evaluate a Top N List recommendation of products rather than a ratings-based evaluation. We specify n = 1:10 to evaluate the accuracy of 1 through 10 recommendations.

# Warning: This will take a minute or so to run  
results\_rlab <- recommenderlab::evaluate(  
 eval\_scheme,   
 algorithms\_list,   
 type = "topNList",   
 n = 1:10)

## RANDOM run fold/sample [model time/prediction time]  
## 1 [0.01sec/2.19sec]   
## POPULAR run fold/sample [model time/prediction time]  
## 1 [0sec/4.43sec]   
## UBCF run fold/sample [model time/prediction time]  
## 1 [0sec/31.55sec]   
## IBCF run fold/sample [model time/prediction time]  
## 1 [10.74sec/0.52sec]   
## AR run fold/sample [model time/prediction time]  
## 1 [0.15sec/30.84sec]

#### 4.4 Model Performance

The result is a list containing the 5 evaluations.

results\_rlab

## List of evaluation results for 5 recommenders:  
## Evaluation results for 1 folds/samples using method 'RANDOM'.  
## Evaluation results for 1 folds/samples using method 'POPULAR'.  
## Evaluation results for 1 folds/samples using method 'UBCF'.  
## Evaluation results for 1 folds/samples using method 'IBCF'.  
## Evaluation results for 1 folds/samples using method 'AR'.

We can investigate a single model by using the getConfusionMatrix() function, which returns a list containing a matrix. Here is the output for the “random items” model.

results\_rlab$`random items` %>%  
 getConfusionMatrix()

## [[1]]  
## TP FP FN TN precision recall  
## 1 0.001412429 0.9985876 0.9985876 785.0014 0.001412429 0.001412429  
## 2 0.004237288 1.9957627 0.9957627 784.0042 0.002118644 0.004237288  
## 3 0.004943503 2.9950565 0.9950565 783.0049 0.001647834 0.004943503  
## 4 0.005649718 3.9943503 0.9943503 782.0056 0.001412429 0.005649718  
## 5 0.007768362 4.9922316 0.9922316 781.0078 0.001553672 0.007768362  
## 6 0.008474576 5.9915254 0.9915254 780.0085 0.001412429 0.008474576  
## 7 0.009180791 6.9908192 0.9908192 779.0092 0.001311542 0.009180791  
## 8 0.009180791 7.9908192 0.9908192 778.0092 0.001147599 0.009180791  
## 9 0.009887006 8.9901130 0.9901130 777.0099 0.001098556 0.009887006  
## 10 0.010593220 9.9894068 0.9894068 776.0106 0.001059322 0.010593220  
## TPR FPR  
## 1 0.001412429 0.001270468  
## 2 0.004237288 0.002539138  
## 3 0.004943503 0.003810504  
## 4 0.005649718 0.005081871  
## 5 0.007768362 0.006351440  
## 6 0.008474576 0.007622806  
## 7 0.009180791 0.008894172  
## 8 0.009180791 0.010166437  
## 9 0.009887006 0.011437803  
## 10 0.010593220 0.012709169

#### 4.5 Tidying The Performance Output

We will tidy up the output by performing the following operations:

* We use pluck() with an integer position of 1 to retrieve the first element (the only element) of the confusion matrix output.
* We convert to a tibble with as.tibble()
* We use rownames\_to\_column() to because the rownames are the number of recommendations evaluated

results\_rlab$`random items` %>%  
 getConfusionMatrix() %>%  
 pluck(1) %>%  
 as.tibble() %>%  
 rownames\_to\_column(var = "n")

## Warning: `as.tibble()` is deprecated, use `as\_tibble()` (but mind the new semantics).  
## This warning is displayed once per session.

## # A tibble: 10 x 9  
## n TP FP FN TN precision recall TPR FPR  
## <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 1 0.00141 0.999 0.999 785. 0.00141 0.00141 0.00141 0.00127  
## 2 2 0.00424 2.00 0.996 784. 0.00212 0.00424 0.00424 0.00254  
## 3 3 0.00494 3.00 0.995 783. 0.00165 0.00494 0.00494 0.00381  
## 4 4 0.00565 3.99 0.994 782. 0.00141 0.00565 0.00565 0.00508  
## 5 5 0.00777 4.99 0.992 781. 0.00155 0.00777 0.00777 0.00635  
## 6 6 0.00847 5.99 0.992 780. 0.00141 0.00847 0.00847 0.00762  
## 7 7 0.00918 6.99 0.991 779. 0.00131 0.00918 0.00918 0.00889  
## 8 8 0.00918 7.99 0.991 778. 0.00115 0.00918 0.00918 0.0102   
## 9 9 0.00989 8.99 0.990 777. 0.00110 0.00989 0.00989 0.0114   
## 10 10 0.0106 9.99 0.989 776. 0.00106 0.0106 0.0106 0.0127

Next, let’s turn this into a function that can be mapped to each element of the list.

tidy\_confusion\_matrix <- function(rlab\_result) {  
 rlab\_result %>%  
 getConfusionMatrix() %>%  
 pluck(1) %>%  
 as.tibble() %>%  
 rownames\_to\_column(var = "n")  
}

We can now map() this function to obtain all of the results in a tidy format. However, this still returns a list of 5 tibbles, and we need a single tibble so we can compare all the models against each other. We’ll use a new function called enframe() to turn the list into a nested tibble with the names in a single column and the values in a nested column. We tack on an unnest() to get the results in a single level, unnested tibble.

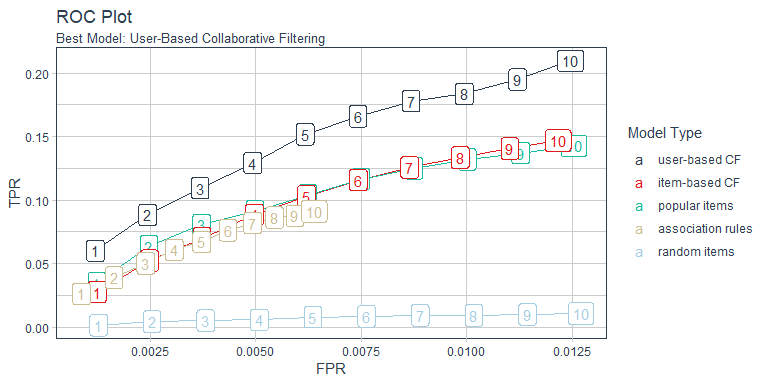
results\_tbl <- results\_rlab %>%  
 map(tidy\_confusion\_matrix) %>%  
 enframe() %>%  
 unnest()  
  
results\_tbl

## # A tibble: 50 x 10  
## name n TP FP FN TN precision recall TPR FPR  
## <chr> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 rando~ 1 0.00141 0.999 0.999 785. 0.00141 0.00141 0.00141 0.00127  
## 2 rando~ 2 0.00424 2.00 0.996 784. 0.00212 0.00424 0.00424 0.00254  
## 3 rando~ 3 0.00494 3.00 0.995 783. 0.00165 0.00494 0.00494 0.00381  
## 4 rando~ 4 0.00565 3.99 0.994 782. 0.00141 0.00565 0.00565 0.00508  
## 5 rando~ 5 0.00777 4.99 0.992 781. 0.00155 0.00777 0.00777 0.00635  
## 6 rando~ 6 0.00847 5.99 0.992 780. 0.00141 0.00847 0.00847 0.00762  
## 7 rando~ 7 0.00918 6.99 0.991 779. 0.00131 0.00918 0.00918 0.00889  
## 8 rando~ 8 0.00918 7.99 0.991 778. 0.00115 0.00918 0.00918 0.0102   
## 9 rando~ 9 0.00989 8.99 0.990 777. 0.00110 0.00989 0.00989 0.0114   
## 10 rando~ 10 0.0106 9.99 0.989 776. 0.00106 0.0106 0.0106 0.0127   
## # ... with 40 more rows

#### 4.6 Visualizing The Performance Results

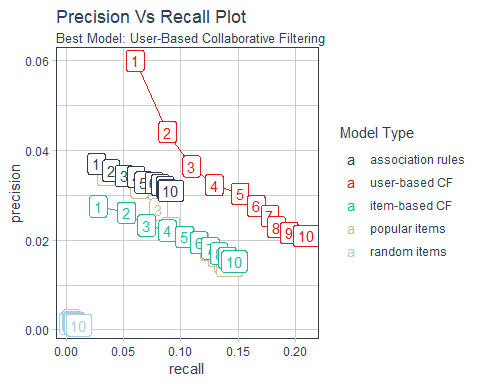
We’ll visualize the performance results to determine which modeling techniques stand out. First, we’ll plot the ROC Curve, which pits the false positive rate against the true positive rate. Note that fct\_reorder2() is useful in this type of plot to order the “name” (model type) by the best final FPR and TPR value, making the plot legend more readable.

results\_tbl %>%  
 ggplot(aes(FPR, TPR,   
 color = fct\_reorder2(as.factor(name), FPR, TPR))) +  
 geom\_line() +  
 geom\_label(aes(label = n)) +  
 theme\_tq() +  
 scale\_color\_tq() +  
 theme(legend.position = "right",  
 legend.direction = "vertical") +  
 labs(  
 title = "ROC Plot",  
 subtitle = "Best Model: User-Based Collaborative Filtering",  
 color = "Model Type"  
 )



Next, we can plot the Precision Vs Recall curves. The Recall goes along the X-axis and the Precision goes along the Y-axis. We again see that the user-based collaborative filtering is the best model.

results\_tbl %>%  
   
 ggplot(aes(recall, precision,   
 color = fct\_reorder2(as.factor(name), recall, precision))) +  
 geom\_line() +  
 geom\_label(aes(label = n)) +  
 theme\_tq() +  
 scale\_color\_tq() +  
 theme(legend.position = "right",  
 legend.direction = "vertical") +  
 labs(  
 title = "Precision Vs Recall Plot",  
 subtitle = "Best Model: User-Based Collaborative Filtering",  
 color = "Model Type"  
 )



#### 4.7 Grid Search (Advanced)

Grid search is a great way to test for an optimal parameter set for the UBCF model. We begin by creating a modeling function, model\_ubcf(), that returns the confusion matrix for a single UBCF model. The function takes three parameters:

* method: One of “Jaccard” or “Cosine”, the two methods employed by UBCF to determine the similarity
* nn: The number of nearest neighbors to use in determining the similarity
* eval\_scheme: The recommenderlab evaluation scheme set up using the evaluationScheme() function

model\_ubcf <- function(method, nn, eval\_scheme) {  
   
 # Define a single algorithm  
 algorithms\_list <- list(  
 "user-based CF" = list(name = "UBCF",   
 param = list(method = method, nn = nn))  
 )  
   
 # Calculate the results on the evaluation scheme  
 eval\_results\_rlab <- recommenderlab::evaluate(  
 eval\_scheme,   
 algorithms\_list,   
 type = "topNList",   
 n = 1:10)  
   
 # Return the confusion matrix using the getConfusionMatrix() function  
 ret <- getConfusionMatrix(eval\_results\_rlab[[1]])[[1]] %>%  
 as.tibble() %>%  
 rownames\_to\_column(var = "n") %>%  
 mutate(n = as.numeric(n))  
   
 return(ret)  
  
}

Next, test out the model\_ubcf() function with a single set of parameters to verify the confusion matrix is returned for each of the evaluation scheme predictions, n = 1:10.

method <- "jaccard"  
nn <- 25  
eval\_scheme <- eval\_scheme # Set up previously using 0.9/0.1 split  
  
model\_ubcf(method, nn, eval\_scheme)

## UBCF run fold/sample [model time/prediction time]  
## 1 [0.04sec/31.48sec]

## # A tibble: 10 x 9  
## n TP FP FN TN precision recall TPR FPR  
## <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 1 0.0494 0.951 0.951 785. 0.0494 0.0494 0.0494 0.00121  
## 2 2 0.0805 1.92 0.919 784. 0.0403 0.0805 0.0805 0.00244  
## 3 3 0.101 2.90 0.899 783. 0.0337 0.101 0.101 0.00369  
## 4 4 0.108 3.89 0.892 782. 0.0270 0.108 0.108 0.00495  
## 5 5 0.121 4.88 0.879 781. 0.0243 0.121 0.121 0.00621  
## 6 6 0.135 5.87 0.865 780. 0.0225 0.135 0.135 0.00746  
## 7 7 0.144 6.86 0.856 779. 0.0206 0.144 0.144 0.00872  
## 8 8 0.150 7.85 0.850 778. 0.0187 0.150 0.150 0.00999  
## 9 9 0.158 8.84 0.842 777. 0.0176 0.158 0.158 0.0112   
## 10 10 0.165 9.83 0.835 776. 0.0165 0.165 0.165 0.0125

Next, we can scale this to a grid of values to test which hyperparameter combinations provide the best confusion matrix results. We can use the cross\_df() function from purrr to create an expanded grid of hyperparameter combinations. We then use mutate() and map2() to run the model\_ubcf() function on each hyper parameter combination, storing the resulting confusion matrix in the “conf\_matrix” column. **Note: This will take several minutes to run.**

# Note: Long running script that will take 3-4 minutes to run  
grid\_search\_tbl <- list(method = c("cosine", "jaccard"),  
 nn = c(25, 150, 500, 1000)) %>%  
 cross\_df() %>%  
 mutate(conf\_matrix = map2(  
 .x = method,   
 .y = nn,   
 .f = model\_ubcf, eval\_scheme))

## UBCF run fold/sample [model time/prediction time]  
## 1 [0.02sec/30.43sec]   
## UBCF run fold/sample [model time/prediction time]  
## 1 [0sec/31.56sec]   
## UBCF run fold/sample [model time/prediction time]  
## 1 [0sec/32.91sec]   
## UBCF run fold/sample [model time/prediction time]  
## 1 [0sec/33.05sec]   
## UBCF run fold/sample [model time/prediction time]  
## 1 [0sec/31.26sec]   
## UBCF run fold/sample [model time/prediction time]  
## 1 [0sec/29.05sec]   
## UBCF run fold/sample [model time/prediction time]  
## 1 [0sec/30.59sec]   
## UBCF run fold/sample [model time/prediction time]  
## 1 [0sec/29.74sec]

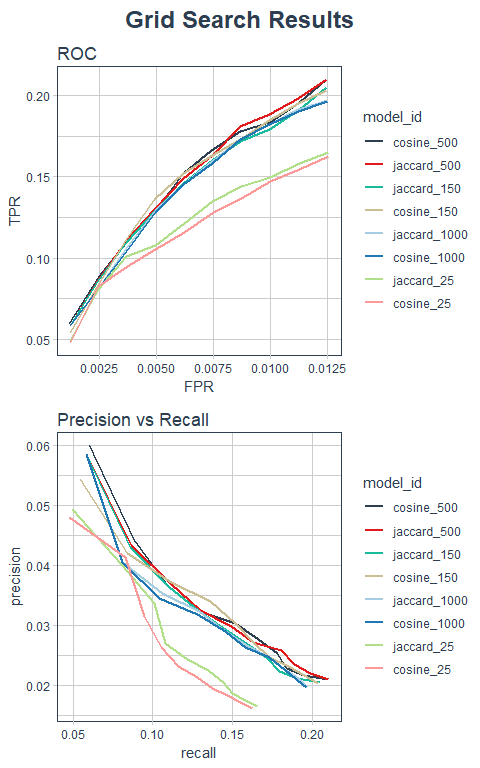
We can then transform the data for performance visualization.

data\_transformed <- grid\_search\_tbl %>%  
 mutate(model\_id = glue("{method}\_{nn}")) %>%  
 select(model\_id, method, nn, conf\_matrix) %>%  
 unnest() %>%  
 mutate(model\_id = as\_factor(model\_id) %>% fct\_reorder2(FPR, TPR))  
  
data\_transformed

## # A tibble: 80 x 12  
## model\_id method nn n TP FP FN TN precision recall  
## <fct> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 cosine\_~ cosine 25 1 0.0480 0.952 0.952 785. 0.0480 0.0480  
## 2 cosine\_~ cosine 25 2 0.0826 1.92 0.917 784. 0.0413 0.0826  
## 3 cosine\_~ cosine 25 3 0.0946 2.91 0.905 783. 0.0315 0.0946  
## 4 cosine\_~ cosine 25 4 0.105 3.89 0.895 782. 0.0263 0.105   
## 5 cosine\_~ cosine 25 5 0.116 4.88 0.884 781. 0.0232 0.116   
## 6 cosine\_~ cosine 25 6 0.128 5.87 0.872 780. 0.0213 0.128   
## 7 cosine\_~ cosine 25 7 0.137 6.86 0.863 779. 0.0196 0.137   
## 8 cosine\_~ cosine 25 8 0.147 7.85 0.853 778. 0.0184 0.147   
## 9 cosine\_~ cosine 25 9 0.155 8.85 0.845 777. 0.0172 0.155   
## 10 cosine\_~ cosine 25 10 0.162 9.84 0.838 776. 0.0162 0.162   
## # ... with 70 more rows, and 2 more variables: TPR <dbl>, FPR <dbl>

Finally, we can use cowplot to generate a consolidated visualization of the ROC and Precision vs Recall Plots.

p1 <- data\_transformed %>%  
 ggplot(aes(FPR, TPR, color = model\_id)) +  
 geom\_line(size = 1) +  
 theme\_tq() +  
 scale\_color\_tq() +  
 labs(title = "ROC") +  
 theme(legend.position = "right")  
  
p2 <- data\_transformed %>%  
 ggplot(aes(recall, precision, color = model\_id)) +  
 geom\_line(size = 1) +  
 theme\_tq() +  
 scale\_color\_tq() +  
 labs(title = "Precision vs Recall") +  
 theme(legend.position = "right")  
  
p\_title <- ggdraw() +  
 draw\_label("Grid Search Results", size = 18,   
 fontface = "bold", colour = palette\_light()[[1]])  
  
plot\_grid(p\_title, p1, p2, ncol = 1, rel\_heights = c(0.1, 1, 1))



We can see that choosing nearest neighbors (nn) of 25 yields poor results, but from 150 and beyond the results are all fairly close. Further, there is no real difference between Jaccard and Cosine performance. As a results, we can feel safe in selecting a combination of method = "Jaccard" and nn = 500. Note that the Jaccard method tends to be less computationally expensive but close in accuracy to the Cosine method.

#### 4.8 Generate Predictions For New Users

This part is really neat. Now that we know what recommendation scheme works best, we’ll create a recommender trained using those settings using the Recommender() function. We’ll use the following settings:

* Model Type: method = "UBCF"
* Model Parameters:
  + method = "Jaccard"
  + nn = 500 for 500 nearest neighbors

The result is a Recommender object of type “UBCF”. Think of this like the output of the lm() function.

train\_recLab <- getData(eval\_scheme, "train")  
  
fit\_ubcf\_recLab <- recommenderlab::Recommender(  
 train\_recLab,   
 method = "UBCF",   
 param = list(method = "Jaccard",   
 nn = 500))  
  
fit\_ubcf\_recLab

## Recommender of type 'UBCF' for 'binaryRatingMatrix'   
## learned using 12735 users.

Next, we’ll create a hypothetical new user basket that contains Bananas and Organic Whole Milk.

new\_user\_basket <- c("Banana", "Organic Whole Milk")

Before we make any predictions, we need to convert this basket to the format that recommenderlab expects: A one hot encoded ratings matrices (wide data) with column names matching the training data and 1’s and 0’s matching whether or not the item exists in the basket.

To begin, we can get the column names from the training data using train\_recLab@data and piping this to the colnames() function.

top\_items\_names <- train\_recLab@data %>% colnames()

Now that we have the column names stored as a character vector, we can create a tibble in the wide format with items in the columns and 1’s and 0’s as values for the users.

new\_user\_basket\_one\_hot\_tbl <- tibble(  
 item = top\_items\_names  
) %>%  
 mutate(value = as.numeric(item %in% new\_user\_basket)) %>%  
 spread(key = item, value = value)  
  
new\_user\_basket\_one\_hot\_tbl

## # A tibble: 1 x 786  
## `0% Greek Strai~ `1% Low Fat Mil~ `1% Lowfat Milk` `100 Calorie P~  
## <dbl> <dbl> <dbl> <dbl>  
## 1 0 0 0 0  
## # ... with 782 more variables: `100% Lactose Free Fat Free Milk` <dbl>,  
## # `100% Natural Spring Water` <dbl>, `100% Pure Pumpkin` <dbl>, `100%  
## # Raw Coconut Water` <dbl>, `100% Recycled Bathroom Tissue` <dbl>, `100%  
## # Recycled Paper Towels` <dbl>, `100% Whole Wheat Bread` <dbl>, `2%  
## # Reduced Fat DHA Omega-3 Reduced Fat Milk` <dbl>, `2% Reduced Fat  
## # Milk` <dbl>, `2% Reduced Fat Organic Milk` <dbl>, `85% Lean Ground  
## # Beef` <dbl>, `90% Lean Ground Beef` <dbl>, `93% Ground Beef` <dbl>,  
## # `Aged White Cheddar Baked Rice & Corn Puffs Gluten Free Lunch  
## # Packs` <dbl>, `Air Chilled Breaded Chicken Breast Nuggets` <dbl>, `Air  
## # Chilled Organic Boneless Skinless Chicken Breasts` <dbl>, `All Natural  
## # Marinara Sauce` <dbl>, `Almond Breeze Original Almond Milk` <dbl>,  
## # `Almond Nut & Rice Cracker Snacks` <dbl>, `ALMONDBREEZE  
## # UNSWEETENED` <dbl>, `Almonds & Sea Salt in Dark Chocolate` <dbl>,  
## # `Alpine Spring Water` <dbl>, `Aluminum Foil` <dbl>, `Apple Cider  
## # Vinegar` <dbl>, `Apple Cinnamon GoGo Squeez` <dbl>, `Apple Honeycrisp  
## # Organic` <dbl>, `Apple Juice` <dbl>, `Apple Pie Fruit & Nut Food  
## # Bar` <dbl>, Apples <dbl>, `Applewood Smoked Bacon` <dbl>, `Arancita  
## # Rossa` <dbl>, Asparagus <dbl>, `Asparation/Broccolini/Baby  
## # Broccoli` <dbl>, `Authentic French Brioche` <dbl>, `Authentic French  
## # Brioche Hamburger Buns` <dbl>, `Baby Arugula` <dbl>, `Baby  
## # Cucumbers` <dbl>, `Baby Food Stage 2 Blueberry Pear & Purple  
## # Carrot` <dbl>, `Baby Seedless Cucumbers` <dbl>, `Baby Spinach` <dbl>,  
## # `Backyard Barbeque Potato Chips` <dbl>, `Bag of Organic  
## # Bananas` <dbl>, `Baked Aged White Cheddar Rice and Corn Puffs` <dbl>,  
## # Banana <dbl>, `Bartlett Pears` <dbl>, Basil <dbl>, `Basil  
## # Pesto` <dbl>, `Berry Medley` <dbl>, `Bicolor Sweet Corn` <dbl>, `Bing  
## # Cherries` <dbl>, `Black Beans` <dbl>, `Black Plum` <dbl>,  
## # Blackberries <dbl>, `Blackberry Cucumber Sparkling Water` <dbl>,  
## # `Blood Oranges` <dbl>, Blueberries <dbl>, `Blueberry Yoghurt` <dbl>,  
## # `Boneless Skinless Chicken Breast` <dbl>, `Boneless Skinless Chicken  
## # Breasts` <dbl>, `Boneless Skinless Chicken Thighs` <dbl>,  
## # `Boomchickapop Sea Salt Popcorn` <dbl>, `Broccoli & Apple Stage 2 Baby  
## # Food` <dbl>, `Broccoli & Cheddar Bake Meal Bowl` <dbl>, `Broccoli  
## # Crown` <dbl>, `Broccoli Florettes` <dbl>, `Brussels Sprouts` <dbl>,  
## # `Bunched Cilantro` <dbl>, `Bunny-Luv Fresh Organic Carrots` <dbl>,  
## # `Bunny Pasta with Yummy Cheese Macaroni & Cheese` <dbl>, Butter <dbl>,  
## # `Butternut Squash` <dbl>, `Cage Free Brown Eggs-Large, Grade A` <dbl>,  
## # `Cage Free Large White Eggs` <dbl>, `California Sourdough  
## # Bread` <dbl>, `Cane Sugar` <dbl>, `Canola Oil` <dbl>,  
## # Cantaloupe <dbl>, Carrots <dbl>, `Cauliflower Florets` <dbl>, `Celery  
## # Hearts` <dbl>, `Celery Sticks` <dbl>, Cereal <dbl>, `Cheddar Bunnies  
## # Snack Crackers` <dbl>, `Cheerios Cereal` <dbl>, `Cheese Pizza` <dbl>,  
## # `Cheese Pizza Snacks` <dbl>, `Cherrios Honey Nut` <dbl>, `Cherry  
## # Pomegranate Greek Yogurt` <dbl>, `Cherubs Heavenly Salad  
## # Tomatoes` <dbl>, `Chicken & Maple Breakfast Sausage` <dbl>, `Chicken  
## # Breast Tenders Breaded` <dbl>, `Chocolate Chip Cookie Dough Ice  
## # Cream` <dbl>, `Chocolate Chip Cookies` <dbl>, `Chocolate Ice  
## # Cream` <dbl>, `Chopped Spinach` <dbl>, `Chopped Walnuts` <dbl>,  
## # `Cinnamon Rolls with Icing` <dbl>, `Classic Hummus` <dbl>, `Classic  
## # White Bread` <dbl>, Clementines <dbl>, ...

One final formatting step is to convert to matrix and then to the “binaryRatingMatrix” class.

new\_user\_basket\_rlab <- new\_user\_basket\_one\_hot\_tbl %>%  
 as.matrix() %>%  
 as("binaryRatingMatrix")  
  
new\_user\_basket\_rlab

## 1 x 786 rating matrix of class 'binaryRatingMatrix' with 2 ratings.

Now we are ready to predict(). We supply the Recommender model, the new data and number of predictions we want to make. We’ll select 5 predictions for the top 5 items similar customers bought.

pred\_ubcf\_recLab <- predict(  
 fit\_ubcf\_recLab,   
 newdata = new\_user\_basket\_rlab,   
 n = 5)  
  
pred\_ubcf\_recLab

## Recommendations as 'topNList' with n = 5 for 1 users.

Now that we have the predictions stored in an object, we can extract the prediction labels, which are stored withing predictions@itemLabels. We’ll create a handy extractor function that takes the predictions object and a user number (which row to select if more than one prediction was made), and returns the product names.

extract\_predictions <- function(predictions, user) {  
 predictions@itemLabels[predictions@items[[user]]]  
}

And voila, we get the predictions when we use the extract\_predictions() functions for the first (and only) new user’s basket.

pred\_ubcf\_recLab %>%  
 extract\_predictions(1)

## [1] "Organic Strawberries" "Strawberries" "Organic Blackberries"  
## [4] "Organic Avocado" "Honeycrisp Apple"

Just imagine an app on your phone as you go through and select products. Now we can easily return similar products that other customers purchased, which would likely increase sales!

## 5. Evaluation

Business case would need to be reviewed for potential for increased sales. The opportunity could be massive given the minimal setup cost and potential for customers to be given better service (options tuned to their needs) while boosting sales.

# References

1. Recommender lab - <https://cran.r-project.org/package=recommenderlab>
2. Recommenderlab Vignette - <https://cran.r-project.org/web/packages/recommenderlab/vignettes/recommenderlab.pdf>