Homework, The Last

Aaron Politsky

December 3, 2015

Part 1

After we load up the emails and create a Corpus, let's preprocess it and create the document term matrix.

```
corpus = tm_map(corpus, content_transformer(removeNumbers))
corpus = tm_map(corpus, content_transformer(removePunctuation))
corpus = tm_map(corpus, content_transformer(tolower))
corpus = tm_map(corpus, content_transformer(removeWords), stopwords("english"))
corpus = tm_map(corpus, content_transformer(stripWhitespace))

dtm = DocumentTermMatrix(corpus)
```

We'll use a sparsity value of .99 at first

```
# next we remove infrequent words
# we keep columns that are less than 0.99 percent sparse
# this is the parameter that you will need to tune in the homework
sparse_dtm = removeSparseTerms(x=dtm, sparse = 0.99)
sparse_dtm
```

```
## <<DocumentTermMatrix (documents: 5728, terms: 1636)>>
## Non-/sparse entries: 331057/9039951
## Sparsity           : 96%
## Maximal term length: 16
## Weighting          : term frequency (tf)
```

```
# convert all elements to binary
# The occurrence of the word fantastic tells us a lot
# The fact that it occurs 5 times may not tell us much more
sparse_dtm = weightBin(sparse_dtm)

# split into train and test using sampling_vector
df = as.data.frame(as.matrix(sparse_dtm))
df_train = df[sampling_vector,]
df_test = df[-sampling_vector,]
spam_train = emails$spam[sampling_vector]
spam_test = emails$spam[-sampling_vector]

library(e1071)
```

Naive Bayes

Let's train our classification model:

```
### your code for classification goes below
nb_model = naiveBayes(df_train, spam_train)

if (file.exists("nb_train_predictions.RData")) {
  load("nb_train_predictions.RData")
} else {
  nb_train_predictions = predict(nb_model, df_train)
  save(nb_train_predictions, file = "nb_train_predictions.RData")
}
mean(nb_train_predictions == spam_train)
```

```
## [1] 0.9024443
```

```
table(actual = spam_train, predictions = nb_train_predictions)
```

```
##       predictions
## actual  ham spam
##   ham  3071  431
##   spam   16 1064
```

```
# compute test error
if (file.exists("nb_test_predictions.RData")) {
  load("nb_test_predictions.RData")
} else {
  nb_test_predictions = predict(nb_model, df_test)
  save(nb_test_predictions, file = "nb_test_predictions.RData")
}
```

Our test accuracy is:

```
mean(nb_test_predictions == spam_test)
```

```
## [1] 0.8970332
```

```
table(actual = spam_test, predictions = nb_test_predictions)
```

```
##       predictions
## actual ham spam
##   ham  747  111
##   spam   7  281
```

Stemming

Let's try Stemming

```
# also try stemming as a preprocessing step
stemmed = tm_map(corpus, stemDocument, language = "english")
stemmed.dtm = DocumentTermMatrix(stemmed)
stemmed.dtm = removeSparseTerms(x=stemmed.dtm, sparse = 0.99)
```

```
stemmed.dtm = weightBin(stemmed.dtm)
stemmed_df = as.data.frame(as.matrix(stemmed.dtm))
stemmed_df_train = stemmed_df[sampling_vector,]
stemmed_df_test = stemmed_df[-sampling_vector,]

# train model on stemmed corpora
model_stem = naiveBayes(stemmed_df_train, spam_train)
if (file.exists("nb_test_predictions_stem.RData")) {
  load("nb_test_predictions_stem.RData")
} else {
  nb_test_predictions_stem = predict(model_stem, stemmed_df_test)
  save(nb_test_predictions_stem, file = "nb_test_predictions_stem.RData")
}
```

Our test accuracy using stemming is:

```
mean(nb_test_predictions_stem == spam_test)
```

```
## [1] 0.9144852
```

```
table(actual = spam_test, predictions = nb_test_predictions_stem)
```

```
##       predictions
## actual ham spam
##   ham  769   89
##   spam   9  279
```

Different values of Sparsity

Let's try a range of sparsity values:

```
# try different values of sparsity
stemmed.dtm = DocumentTermMatrix(stemmed)
sparsity <- seq(0.9, 0.99, by = .02)
dtms.by.sparsity <- lapply(sparsity, function(sp) {
  as.data.frame(as.matrix(weightBin(removeSparseTerms(x=stemmed.dtm, sparse = sp))))
})

if (file.exists("nb.test.predictions.by.sparsity.Rda")) {
  load("nb.test.predictions.by.sparsity.Rda")
} else {
  nb.test.predictions.by.sparsity <-
    lapply(dtms.by.sparsity, function(dtm) {
      df.train = dtm[sampling_vector,]
      df.test  = dtm[-sampling_vector,]
      nb_model = naiveBayes(df.train, spam_train)
      nb_test_predictions = predict(nb_model, df.test)
    })
  save(nb.test.predictions.by.sparsity, file = "nb.test.predictions.by.sparsity.Rda")
}
names(nb.test.predictions.by.sparsity) <- sparsity
```

Our test accuracy as a function of sparsity is:

```
sapply(nb.test.predictions.by.sparsity, function(nb_test_predictions)
  mean(nb_test_predictions == spam_test)
)
```

```
##       0.9      0.92      0.94      0.96      0.98
## 0.8952880 0.8926702 0.8952880 0.9162304 0.9354276
```

Part 2

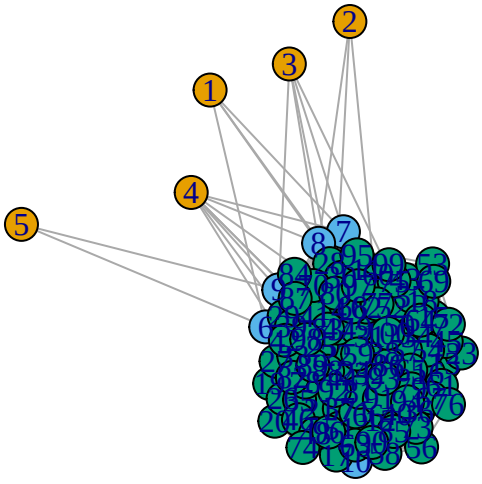
Specify a stochastic block matrix:

```
# block matrix -- assortative
# our graph will have three communities
eps <- .021 # epsilon
f.e <- .5 # fraud-accomplice
M = matrix(c( eps,  f.e,   eps,
              f.e, 2*eps,  1-f.e - 2*eps,
              eps, 1-f.e -2*eps, 1-(1-f.e - 2*eps)),
           nrow = 3)

# sample a random graph
# 100 nodes grouped into 3 communities
num.fraudsters <- num.accomplices <- 5
num.honest <- 90
rg =
  sample_sbm(num.fraudsters + num.accomplices + num.honest, # number of nodes in a random graph
             pref.matrix = M,      # stochastic block matrix M that tells us probability of forming a l
             block.sizes = c(num.fraudsters,
                             num.accomplices,
                             num.honest),  # how many nodes belong to each community
             loops = F,            # no loops (vertex that connects to itself)
             directed = F          # we want an undirected graph
  )

# membership vector used to color vertices
membership_vector = c(rep(1, num.fraudsters),
                      rep(2, num.accomplices),
                      rep(3, num.honest))

plot_layout = layout.fruchterman.reingold(rg)
plot(rg,
     vertex.color=membership_vector,
     layout = plot_layout)
```




```
# given a graph, we would like to uncover parameters of the model
# that is likely to have generated the random graph
# in this example, we know that the graph was generated according to the stochastic block model
# we can compare the estimated parameters to the true parameters
#
# however, in practice, we only see a graph (that is, its adjacency matrix)
# to evaluate our model, we use domain knowledge. for example, whether community
# memberships of nodes make sense

## estimate parameters using the lda package
library(lda)

result =
  mmsb.collapsed.gibbs.sampler(get.adjacency(rg),  # first parameter is the adjacency matrix
                               K = 3,              # we are fitting the graph using a stochastic block
                               num.iterations=10000,    # this and the following parameters specify par
                               alpha = 0.1,
                               burnin = 500L,
                               beta.prior = list(1, 1))

# this matrix tells us for each vertex what is the probability that it belongs to
# one of the K communities
memberships = with(result, t(document_sums) / colSums(document_sums))
head(memberships,20)
```

```
##             [,1]       [,2]       [,3]
##  [1,] 0.00000000 1.00000000 0.00000000
##  [2,] 0.00000000 0.94444444 0.05555556
##  [3,] 0.00000000 1.00000000 0.00000000
##  [4,] 0.06565657 0.93434343 0.00000000
##  [5,] 0.00000000 1.00000000 0.00000000
##  [6,] 0.00000000 0.21212121 0.78787879
##  [7,] 0.00000000 0.37373737 0.62626263
##  [8,] 0.70707071 0.29292929 0.00000000
##  [9,] 0.77777778 0.22222222 0.00000000
## [10,] 0.85858586 0.14141414 0.00000000
## [11,] 0.96969697 0.00000000 0.03030303
## [12,] 1.00000000 0.00000000 0.00000000
```

```
## [13,] 0.00000000 0.00000000 1.00000000
## [14,] 1.00000000 0.00000000 0.00000000
## [15,] 0.08585859 0.00000000 0.91414141
## [16,] 0.88383838 0.00000000 0.11616162
## [17,] 0.00000000 0.04545455 0.95454545
## [18,] 0.49494949 0.00000000 0.50505051
## [19,] 0.25252525 0.05050505 0.69696970
## [20,] 0.23737374 0.21717172 0.54545455
```

Many of these are clearly probably in one community versus the others.

```
# the estimate of the stochastic block matrix M
ratio = with(result, blocks.pos / (blocks.pos + blocks.neg))
ratio
```

```
##             [,1]        [,2]      [,3]
## [1,] 0.514988470 0.004819277 0.6799431
## [2,] 0.009756098 0.428571429 0.0221519
## [3,] 0.685050798 0.000000000 0.3423707
```

```
# actual M
M
```

```
##       [,1]  [,2]  [,3]
## [1,] 0.021 0.500 0.021
## [2,] 0.500 0.042 0.458
## [3,] 0.021 0.458 0.542
```

The ratio doesn't seem to approximate M very well, though.