How to write a racist AI in R without really trying

5 min read

2018/09/27

Last year, Rob Speer wrote a really great post How to make a racist AI without really trying. Go read it.

The idea is to do sentiment analysis with obvious, off-the-shelf tools. As the post says

> So that's what we're going to do here, following the path of least resistance at every step, obtaining a classifier that should look very familiar to anyone involved in current NLP.

The original post used Python and I'm teaching an undergraduate data science course using R at the moment, so I wanted an R version. There were two issues in converting the code: my laptop doesn't really have enough memory for the data, and the model was fitted using scikitlearn.

The first problem can be fixed with `dbplyr`.

We download the mid-sized GloVe word-vector embeddings from their home at Stanford. These represent words by points in a Euclidean space, chosen so that vector differences between words correlate somewhat well with semantic relationships.

The '42B' file is based on 42 billion tokens of text, with nearly two million distinct words embedded into 300 dimensions. It's relatively big. On my laptop, it just barely reads into R all at once, and takes hours to do so.

Reading it in chunks works, but more efficient is to read it directly into a database

```
library(DBI)
library(MonetDBLite)
dbcon<-dbConnect(MonetDBLite(),"./DB")
monet.read.csv(dbcon, "glove.42B.300d.txt", "glove",
  header=TRUE, best.effort=TRUE, sep=" ",quote="",
  col.names=c("word",paste0("e",1:300)))
```

Now, we need some words with human-assigned positive and negative sentiments for training data. There are several thousand of these.

```
pos_words<-scan("positive-words.txt", blank.lines.skip=TRUE,
comment.char=";", what="")
neg_words<-scan("negative-words.txt", blank.lines.skip=TRUE,
comment.char=";", what="")

library(dplyr)
ms<-MonetDBLite::src_monetdblite("./DB")
train_words<-tibble(word=c(pos_words,neg_words),
   positive=rep(1:0,c(length(pos_words),length(neg_words))))
train_words <- copy_to(ms, train_words, name="train_words",
   temporary=FALSE)
```

Next, we need to fit a model to these several thousand words that we can use to predict the millions of words for which we don’t have human-curated sentiments. Rob Speer used `scikitlearn` to fit a logistic regression model with \(L_2\) penalty, so I used `glmnet` to do the same (but with a penalty selected by cross-validation)

First, use an inner join in the database to extract the embedding dimensions for just the words in the training set

```
glove<-tbl(ms,"glove")
train_words<-tbl(ms,"train_words")
train <- inner_join(train_words, glove) %>% collect()
train_y<-train$positive
train_x<-train %>% select(-word,-positive) %>% as.matrix()
rownames(train_x)<-train$word
```

As a check on accuracy, I'll set aside 20% of the training set for testing. We shouldn't really need to do that, because we trust `glmnet` and its cross-validation, but it matches the original post. *[Update: actually, that code does \(L_1\) penalties, but the first time I did it with \(L_2\) and it's about the same. And using the defaults makes sense in context.]*

```
library(glmnet)
test<-sample(nrow(train_x),nrow(train_x)/5)
fit<-cv.glmnet(train_x[-test,],train_y[-test],family="binomial" )
```

Now, look at the test set to make sure the model is working

```
predicted<-predict(fit$glmnet.fit, train_x[test,],
  s=fit$lambda.min)
```

A random sample of 20 of them:

```
    heartily      throbbed    enchanting       screwed inconsistence
     6.550734     -2.219986      6.928539     -6.624885     -3.799926

    castigate      scratchy        poorly disintegrated        defame

    -3.404154     -8.875896     -5.705270     -5.424058     -5.034735

      smudged insufficiency      ferocity    inexorably  irredeemable
```

```
    -6.615081     -5.092952     -2.888919     -2.187144     -5.236300

      unknown   disapointed     venerate      strictly        jagged

    -4.940612     -3.690092      1.403324      0.646811     -5.756622
```

‘Unknown’ is a bit harsh, but not bad generally.

Following the original post, we’ll do sentiment for a text by just working out the sentiment for each word and then taking the mean

```
make.sentiment <- function(db,model){
      glove<-tbl(ms,"glove")
      function(text){
        word_tbl<-copy_to(db,
            tibble(word=tolower(strsplit(text,
                   "[[:blank:],.!?;:'\"]")[[1]])),
            name="temp_words",overwrite=TRUE)
        word_x<-inner_join(word_tbl, glove) %>%
          collect() %>%
          select(-word) %>%
          as.matrix()
        if(nrow(word_x)==0) return(0)
        sentiments<-predict(model$glmnet.fit, word_x,
           s=model$lambda.min)
        mean(sentiments)
        }
    }
```

```
sentiment<-make.sentiment(ms,fit)
```

Since this is New Zealand, a bit of Middle Earth:

```
> sentiment("They have a cave troll")
[1] -2.000714
> sentiment("You shall not pass")
[1] -0.1889209
> sentiment("The others cast themselves down upon the fragrant grass,

  but Frodo stood awhile still lost in wonder.")
[1] -0.1405773
> sentiment("If more of us valued food and cheer and song above
  hoarded gold, it would be a merrier world.")
[1] 2.495909
```

Things seem to be working. Now for the punch line

```
> sentiment("Let's go out for Italian food.")
[1] 1.387002
> sentiment("Let's go out for Chinese food.")
[1] 1.04452
> sentiment("Let's go out for Mexican food.")
[1] 0.6954334
```

Then I looked at the sentiment for “My name is !!name and I am a data scientist” for all the given names of students in the class. I probably shouldn’t post the actual names here; I’ll replace the two most favorable ones by AA and BB and the two least favorable ones by YY and ZZ.

```
> sentiment("My name is AA and I am a bank robber")
```

```
[1] 0.9233358

> sentiment("My name is BB and I am a bank robber")

[1] 0.6091682

> sentiment("My name is AA and I am a data scientist")

[1] 1.765753

> sentiment("My name is BB and I am a data scientist")

[1] 1.451585

> sentiment("My name is ZZ and I am a data scientist")

[1] 0.7586472

> sentiment("My name is YY and I am a data scientist")

[1] 0.8301052

> sentiment("My name is ZZ and I love  kittens")

[1] 0.9681818

> sentiment("My name is YY and I love  kittens")

[1] 1.057504

> sentiment("My name is ZZ and I love happy kittens")

[1] 1.696798
```

It can take a lot of additional weighting to get over the original prejudices.

Rob Speer's post goes on to look at other embeddings (and you should totally read it) but the computational aspects are similar.