Word Embeddings

Promothesh Chatterjee[*]

[*]Copyright© by Promothesh Chatterjee. All rights reserved. No part of this note may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author.

So far we have ignored the nearby context of a word when representing text numerically in the bag-of-word models (except for a brief illustration of n-grams in week 1, and a conceptual reference to Term Co-occurrence Matrix in Latent Semantic Analysis) . We have ignored what the impact of neighboring words can be its meaning and how those relationships affect the overall meaning of the sentence. In word embedding models (also called word vectors), we will create much smaller bags-of-words from a "neighborhood" of only a few words (around 10 tokens or so).

Word embeddings are numerical vector representations of word that capture literal and implied meaning. These word vectors, due to the way they are represented in the vector space, are able to capture the relationships between words that can help us to identify synonyms, antonyms or words belonging to the same category. You can argue that we did the same with Latent Semantic Analysis, but not only is the accuracy here much better but also the word embedding models capture the hidden meanings and relationships much better than LSA. Remember the term matrix (along with the Singular value matrix and document matrix) that we got as a result of running LSA on text data, you can think of word embeddings as akin to the term matrix. Each word is represented along some pre-decided number of dimensions (like in LSA we truncated our dimensions to 300). Once we have an accurate representation of words as vectors, we can then perform any vector operations on the word vectors as per need.

There are many practical utilities of word embeddings in businesses and research. Here are some examples. If you search Amazon for "byrna" (it is non-lethal self-defense pepper gun not sold on Amazon), the search returns other brands of non-lethal self-defense guns. Obviously, there is no synonym of byrna, it being a proper noun. Amazon uses some version of word embeddings to figure out such complex meanings and relationships. Similarly, Spotify uses it to help provide music recommendation. Stitch Fix uses it to recommend clothing. Google is thought to use word2vec (a kind of word embedding) in RankBrain as part of their search algorithm. Word embeddings are often used for sentiment analysis etc. Some of the popular word embedding algorithms are Word2vec (from Google researcher Mikolov), Stanford's GloVe, and Facebook's fastText (again by Mikolov when he moved to FB). Before we go to the technical details, let's create some embeddings and look at some uses.

R Implementation of Word2vec

Let's look at the legendary word2vec model. Mikolov was interning at Google when he and others came up with the idea behind word2vec. Now of course, there are many advanced versions of the word2vec, but it is still very powerful.

I have downloaded 50,000 reviews of movies from IMDB (the link for downloading the file is below). It is around 62.5 MB in size, so will take some time to train (around an hour at least on a regular computer). https://www.dropbox.com/s/eq9kvbadqx5jhlk/IMDB.txt?dl=0

We will use this file to train our word embeddings. We will need to install library wordVectors from Github first. Windows users may need to install "Rtools" as well.

```r
if (!require(wordVectors)) {
    if (!(require(devtools))) {
        install.packages("devtools")
    }
    devtools::install_github("bmschmidt/wordVectors")
}
```

We will first load the libraries wordVectors and dplyr (for the pipe function), then prep the text file before training the word embeddings.

```
library(wordVectors)
library(dplyr)

prep_word2vec(origin="Data/IMDB.txt",destination="Data/IMDB1.txt",
              lowercase=T,bundle_ngrams=2)
```

Prepping does a couple things:

1. Creates a single text file (IMDB1.txt) with the contents of every file in the original document (in our case original file is single file but we can also specify a folder that contains multiple text files here);
2. Uses the "tokenizers" package to clean and lowercase the original text,
3. If "bundle_ngrams" is greater than 1, joins together common bigrams into a single word. For example, "olive oil" may be joined together into "olive_oil" wherever it occurs.

Now, we are ready to train our word embeddings.

```
model = read.vectors("Data/IMDB_vectors.bin")
```

```
##   |                                                                               |
```

```
#model = train_word2vec("IMDB1.txt","IMDB_vectors.bin",
#vectors=200,threads=4,window=12,iter=5,negative_samples=0)
```

Since, I have already trained the word2vec model on IMDB reviews, I will access it using the function read.vectors and specify the .bin file that was created during training. However, if you are running this for the first time, the initial step is to train the word2vec model on the text using train_word2vec function.

The parameter "vectors" indicates the dimensions. This is similar to specifying k in the truncated LSA. Here we want the word vector to have 200 dimensions. Bigger number means more accuracy but slower operations. Typical number of dimensions range between 100-500. The "threads" parameter refers to the number of processors available for use on the machine. On my machine I have 4 cores and 8 logical processors, so I am using 4 of those. "window=12" implies the context window is 12, we will talk about this in greater detail in when discuss the technical details. The argument "iter" indicated how many times to read through the corpus. With smaller corpus, it can greatly help to increase the number of passes; if you're working with billions of words, it probably matters less. If the number of iterations is too low, that words that aren't closely related will seem to be closer than they are.

Let's check what we can find out about movies.

```
model %>% closest_to("connery")# since the embeddings are in lower case,
```

```
##            word similarity to "connery"
## 1       connery               1.0000000
## 2  sean_connery               0.6572764
## 3          bond               0.6414308
## 4   roger_moore               0.6097384
## 5           007               0.6078718
## 6     connery's               0.5630338
## 7     octopussy               0.5439834
## 8   thunderball               0.5309151
## 9     brandauer               0.5270423
## 10   james_bond               0.5205522
```

```
# our query should be in lower case too
```

Running the function closest_to allows us to find out how top 10 closest representations of the target word. This is an amazing output. If you go back and train an LSA model on the IMDB dataset and then run the query on target word "connery", then you will see the difference in evaluation.

We can do vector math on the word embeddings. In fact, the most famous example of word2vec model trained on Google news corpus is able to infer relationship using vector math. For instance, we can use word vectors to help us answer "King – Man + Woman = ?" question and arrive at the result "Queen"! Which is truly a phenomenal result. Let's test a vector addition. So if we add the word vectors of batman and villain, what we are saying is that we want to find out the words that are closest to the *combination* of "batman" and "villain":

```
model %>% closest_to(~"batman"+"villain")
```

```
##               word similarity to "batman" + "villain"
## 1           batman                          0.8928429
## 2            joker                          0.7519927
## 3          penguin                          0.6980331
## 4        mr_freeze                          0.6749374
## 5          villain                          0.6739089
## 6         catwoman                          0.6738374
## 7   batman_returns                          0.6435991
## 8     kevin_conroy                          0.6331808
## 9           gotham                          0.6312612
## 10   batman_begins                          0.6241993
```

We could also find words that are shaded towards just "batman" but *not* "villain" by using subtraction.

```
model %>% closest_to(~"batman" - "villain")
```

```
##                word similarity to "batman" - "villain"
## 1            batman                          0.8183636
## 2   animated_series                          0.6371671
## 3    batman_returns                          0.5444857
## 4            gotham                          0.5151331
## 5       bruce_wayne                          0.5010245
## 6        tim_burton                          0.5006367
## 7          phantasm                          0.4914324
## 8   justice_league                          0.4903468
## 9   batman_forever                          0.4838172
## 10        batman's                          0.4812850
```

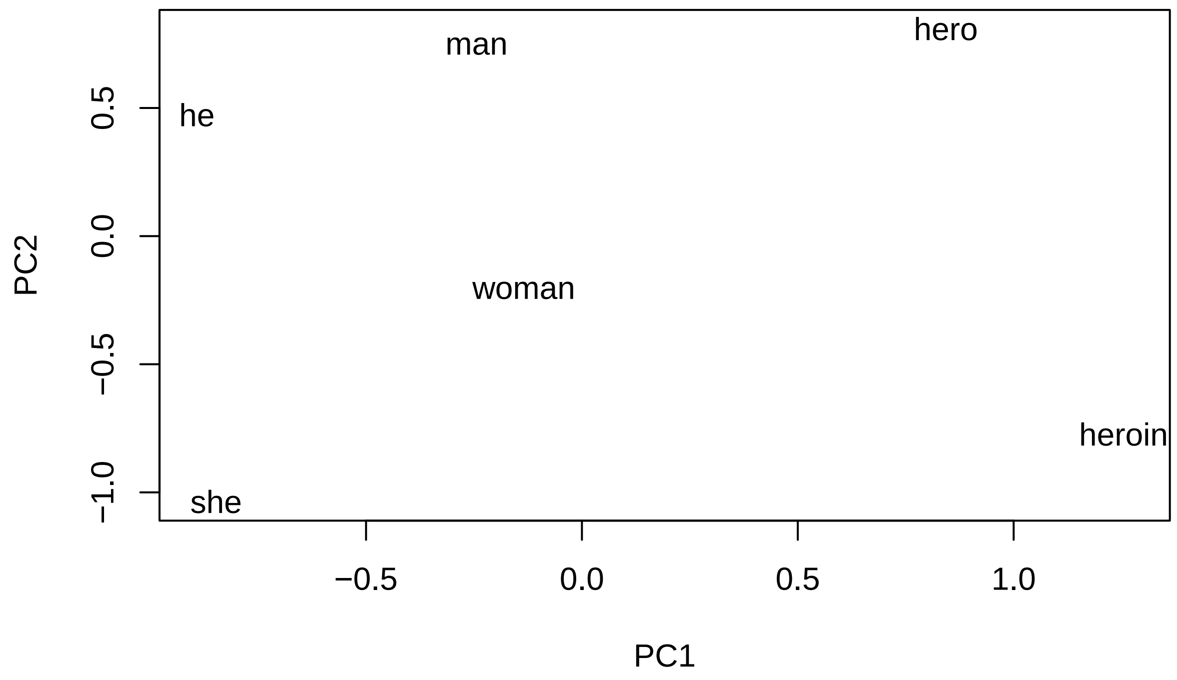
We can explore analogies with word embeddings. Male actors are referred to as ‘heroes’. We can find the female equivalent using word embedding analogies. We can think of this as starting at “hero”, removing its similarity to “he”, and adding a similarity to “she”. This yields the answer: the most similar term to “hero” for a lady is “heroine.”

```
model %>% closest_to(~ "hero" - "he" + "she")
```

```
##             word similarity to "hero" - "he" + "she"
## 1        heroine                           0.7301585
## 2           hero                           0.6262348
## 3        herself                           0.5913090
## 4          she's                           0.5596492
## 5            she                           0.5548443
## 6           girl                           0.5232518
## 7          woman                           0.5112448
## 8        actress                           0.5054336
## 9            her                           0.4884516
## 10 femme_fatale                            0.4798519
```

We can further use Principal Component Analysis (PCA is a data compression technique which is fairly popular) to plot a subset of these vectors to see how they relate. For instance, we can imagine an arrow from “he” to “she”, from “hero” to “heroine”, and from “man” to “woman”; all in the same direction.

```
model[[c("heroine","woman","man","he","she","hero"), average=F]] %>%
  plot(method="pca")
```



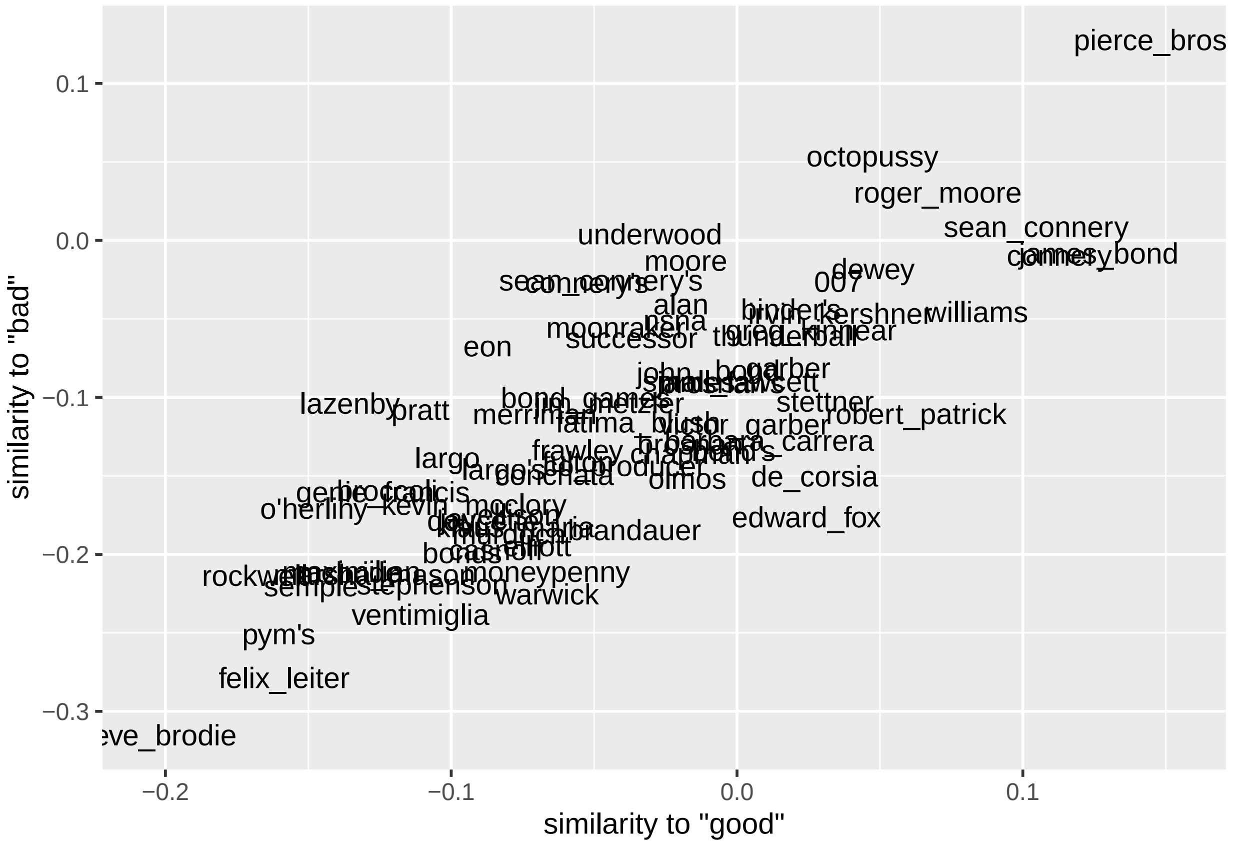
We can combine these two methods to look at positive and negative words used to evaluate characters related to James Bond.

First we build up three data_frames: first, a list of the 50 top evaluative words, and then complete lists of similarity to `"bad"` and `"good"`.

```
top_evaluative_words = model %>%
   closest_to(~ "james"+"bond",n=75)
villany = model %>%
  closest_to(~ "bad",n=Inf) # 'n=Inf' returns the full list
hero1 = model %>%
  closest_to(~ "good", n=Inf)
```

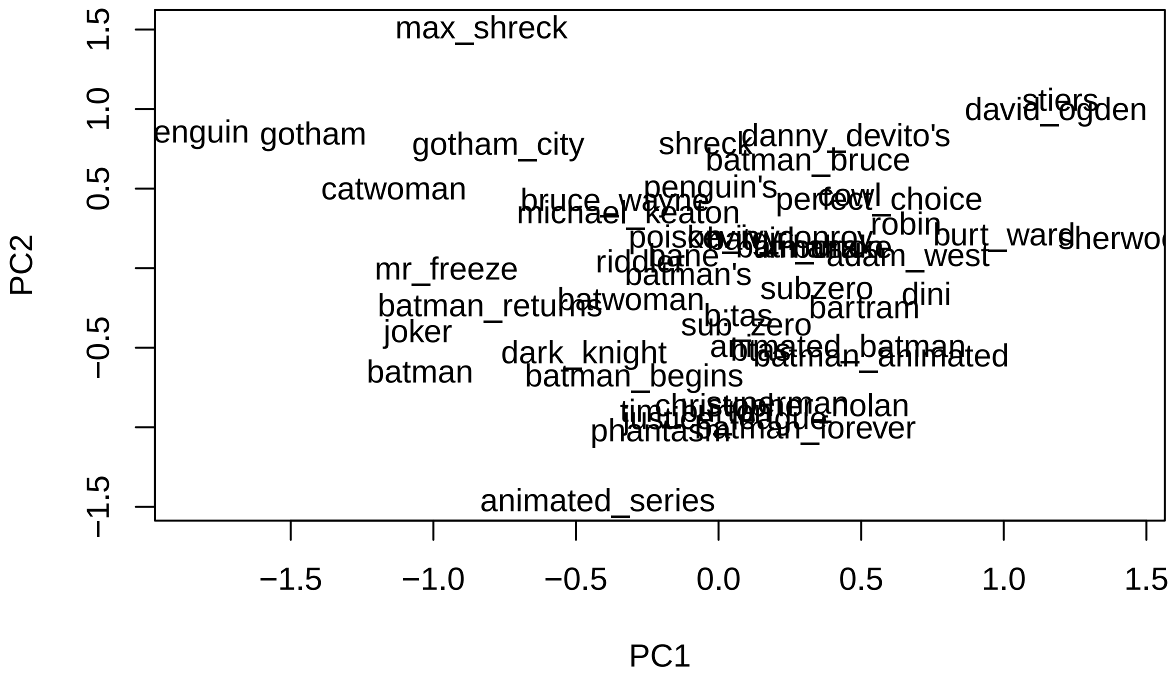
Then we can use tidyverse packages to join and plot these. An `inner_join` restricts us down to just those top 50 words, and ggplot can array the words on axes.

```
library(ggplot2)
library(dplyr)
top_evaluative_words %>%
  inner_join(villany) %>%
  inner_join(hero1) %>%
  ggplot() +
  geom_text(aes(x=`similarity to "good"`,
                y=`similarity to "bad"`,
                label=word))
```



We can also visualize the elements closest to query terms using the function plot and compressing the output using PCA to two dimensions.

```
some_chr = closest_to(model,model[[c("batman","robin")]],50)
charac = model[[some_chr$word,average=F]]
plot(charac,method="pca")
```



We can even cluster the word embeddings based on relationships. Though, we have not talked about kmeans clustering, the idea is fairly straight forward. We indicate how many centroids we want in the dataset. A centroid represents the center of a cluster. Similar to least squares idea, every data point is allocated to each of the clusters by reducing the in-cluster sum of squares. Thus, the K-means algorithm identifies k number of centroids, and then allocates every data point to the nearest cluster, while keeping the centroids as small as possible. You can think of this as a sort of topic model, although unlike more sophisticated topic modeling algorithms like Latent Dirichlet Allocation, each word must be tied to single particular topic. Here are ten random "topics" produced through this method. Each of the columns are the ten most frequent words in one random cluster.

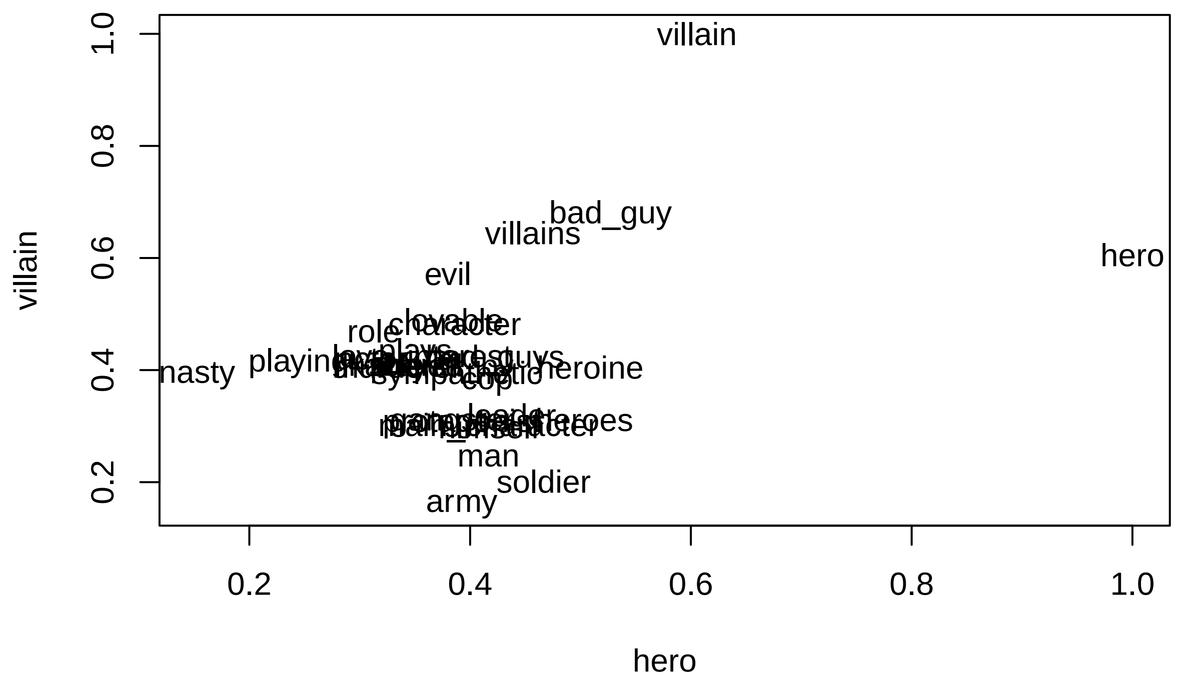
```
set.seed(10)
centers = 150
clustering = kmeans(model,centers=centers,iter.max = 40)

sapply(sample(1:centers,10),function(n) {
    names(clustering$cluster[clustering$cluster==n][1:10])
})
```

```
##       [,1]       [,2]    [,3]       [,4]        [,5]
##  [1,] "wants"    "have"  "long"     "played_by" "portrayed"
##  [2,] "his_wife" "so"    "half"     "douglas"   "gay"
##  [3,] "finds"    "about" "minutes"  "walter"    "male"
##  [4,] "decides"   "what"    "spent"      "barbara"       "innocent"
##  [5,] "sees"      "see"     "hour"       "philip"        "acts"
##  [6,] "dies"      "can"     "minute"     "robinson"      "tough"
##  [7,] "hospital"  "if"      "an_hour"    "butler"        "loving"
##  [8,] "apartment" "get"     "hours"      "dorothy"       "behavior"
##  [9,] "asks"      "do"      "less_than"  "also_starring" "attitude"
## [10,] "returns"   "because" "sit_through" "patricia"     "portrays"
##       [,6]         [,7]     [,8]     [,9]             [,10]
##  [1,] "three"      "blood"  "dog"    "score"          "la"
##  [2,] "early"      "flesh"  "food"   "sharp"          "de"
##  [3,] "including"  "bodies" "eat"    "haunting"       "e"
##  [4,] "along_with" "teeth"  "eating" "musical_score"  "jean"
##  [5,] "famous"     "legs"   "dirty"  "splendid"       "le"
##  [6,] "late"       "arm"    "ball"   "composed"       "carmen"
##  [7,] "include"    "knife"  "fish"   "superbly"       "mario"
##  [8,] "fame"       "guts"   "shop"   "music_score"    "el"
##  [9,] "featured"   "neck"   "dogs"   "composer"       "les"
## [10,] "m"          "corpse" "drink"  "accompanied_by" "da"
```

Visualising Relationship Planes.

We can project the high-dimensional word embeddings into a plane which makes visualizing the relationships easier. For instance, we can take the words "hero" and "villain," find the twenty words most similar to either of them, and plot those in a hero-villain plane.

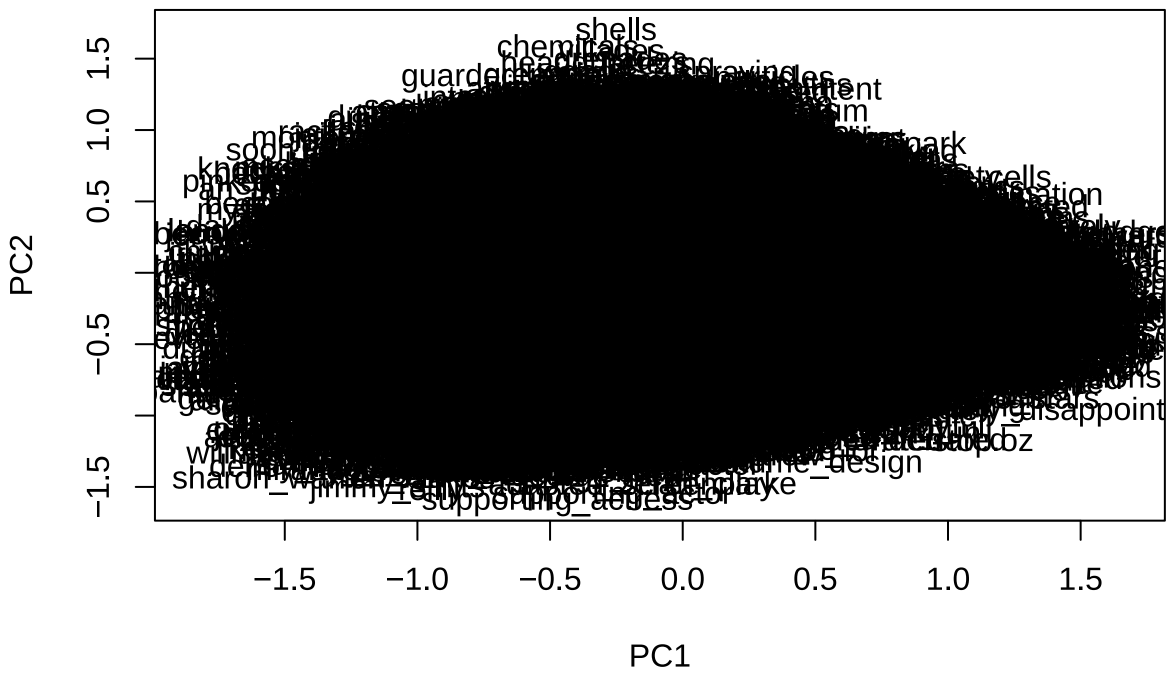
```r
cast = model[[c("hero","villain"),average=F]]
# model[1:3000,] here restricts to the 3000 most common words in the set.
hero_and_villain = model[1:3000,] %>% cosineSimilarity(cast)
# Filter to the top 20 sweet or salty.
hero_and_villain = hero_and_villain[
  rank(-hero_and_villain[,1])<20 |
  rank(-hero_and_villain[,2])<20,
  ]
plot(hero_and_villain,type='n')
text(hero_and_villain,labels=rownames(hero_and_villain))
```



TSNE

In the library wordVectors, there is method built in to visualize the embeddings in two dimensions- TSNE dimensionality reduction (T-distributed Stochastic Neighbor Embedding). In contrast to PCA, T-SNE is a nonlinear dimensionality reduction machine learning algorithm for data visualization. T-SNE aims to keep the relative distance between the points same, yet reduce dimensional space between them. Thus, t-SNE algorithm reduces multidimensional data to two or more dimensions which are easy to visualize while preserving the neighborhood relationships. The function "plot" from the wordVectors library, will display a 2 dimensional word cloud based on t-SNE algorithm if you do not specify the method as PCA. The argument "perplexity" refers to the optimal number of neighbors for each word. By default it's 50; smaller numbers may cause clusters to appear more dramatically at the cost of overall coherence.

```
plot(model,perplexity=50, method="pca")
```



To try and understand the concept of word embeddings, we trained our own embeddings. Actually, you don't need to train the embeddings all the time. Researchers have made a lot of pre-trained embeddings available. For instance, pretrained embeddings on Google News corpus (which was trained on TBs of data) is available, as are embeddings for Wikipedia, Twitter etc. These huge embeddings are very useful for various research and commercial purposes.