In [3]:
```
cd /home/nv/Downloads
```

```
/home/nv/Downloads
```

In [4]:
```
from __future__ import print_function
import time
import numpy as np
np.random.seed(1337)  # for reproducibility

from keras.preprocessing import sequence
from keras.models import Model, Sequential
from keras.layers import Dense, Dropout, Embedding, LSTM, Input, merge, BatchNormalization
from keras.datasets import imdb

import os
from keras.preprocessing.text import Tokenizer
```

In [5]:
```
max_features = 10000
max_len = 200  # cut texts after this number of words (among top max_features most common words)
```

In [6]:
```
# read in the train data
X_train = []
y_train = []

path = './aclImdb/train/pos/'
X_train.extend([open(path + f).read() for f in os.listdir(path) if f.endswith('.txt')])
y_train.extend([1 for _ in range(12500)])

path = './aclImdb/train/neg/'
X_train.extend([open(path + f).read() for f in os.listdir(path) if f.endswith('.txt')])
y_train.extend([0 for _ in range(12500)])

print('x:')
print(X_train[:1])
print(X_train[-1:])
print(len(X_train))
print('y:')
print(y_train[:1])
print(y_train[-1:])
print(len(y_train))
```

```
x:
["Here's a gritty, get-the-bad guys revenge story starring a relentless and rough Denzel Washington.
 He's three personalities here: a down-and-out-low-key-now drunk- former mercenary, then a loving fa
ther-type person to a little girl and then a brutal maniac on the loose seeking answers and revenge.
<br /><br />The story is about Washington hired to be a bodyguard for a little American girl living
in Mexico, where kidnappings of children occur regularly (at least according to the movie.) He becom
es attached to the kid, played winningly by THE child actress of our day, Dakota Fanning. When Fanni
ng is kidnapped in front of him, Washington goes after the men responsible and spares no one. Beware
: this film is not for the squeamish.<br /><br />This is stylish film-making, which is good and bad.
 I liked it, but a number of people found it too frenetic for their tastes as the camera-work is one
 that could give you a headache. I thought it fit the tense storyline and was fascinating to view, b
ut it's (the shaky camera) not for all tastes.<br /><br />Besides the two stars, there is the always
-interesting Christopher Walken, in an uncharacteristically low-key role, and a number of other fine
 actors.<br /><br />The film panders to the base emotions in all of us, but it works."]
['Colleges, High Schools, Fraternities and Sororities have been the most popular stalking grounds fo
r maniacal madmen since the slasher cycle first became a popular cinema culture throughout the late
seventies. Even backwoods cabins and campsites have rode shotgun to the amount of massacres that hav
e taken place on campuses since Halloween categorised the genre as a cult horror category. From earl
y entries like To all a Good Night right up until the big budgeted schlock of titles like Urban Lege
nd or Schools Out, there\'s usually always been a campus slasher lurking somewhere in the pipeline.
```

Despite being picked up by Troma - the titans of B movie badness \xc2\x96 Splatter University was he avily panned upon release and never really found an audience. Even notorious hack and slash websites like HYSTERIA-LIVES have written off Richard Haines\' splatter yarn as one of the worst of the earl y eighties boom. I always approach criticised movies optimistically because there\'s often the chanc e than a few bad reviews can be unfairly contagious like a dose of the flu, which crowds the judgeme nt of certain authors.

It begins in traditional fashion at the place where any maniac wor th his salts emerges. Yep you guessed it \xc2\x96 an insane asylum! It seems that one of the inmates has decided that he\'s unhappy with the level of service at the institution and therefore he\'s loo king to take his business elsewhere. The unseen nut-job makes his break after stabbing an unfortunat e orderly where the sun certainly doesn\'t shine. He obviously favours the dress sense of the murder ed worker, so he takes the liberty of borrowing his uniform, blood stained trousers and all!
<b r />Three years later, we transfer to St Trinians College, an educational establishment that is cont rolled by catholic priests. A teacher is busy after hours marking her students work when all of a su dden there\'s a knock at the door. Before she has a chance to find out what the unseen visitor wants , he stabs her in the chest with a kitchen knife and she falls to the floor in a bloody heap. This o f course means that there\'s a vacancy at the university and so we\'re introduced to Julie Parker (F rancine Forbes), the lovable replacement for the recently departed lecturer. It seems that her arriv al has inadvertently given the resident maniac all the motivation that he needs to go on a no holds barred slaughter-thon. Before long students and teachers alike are dropping like flies to the camera shy menace as he stalks the corridors and local areas armed with an exceptionally large blade. Susp icious suspects abound, but can professor Parker solve the mystery of the campus murderer before she becomes just another statistic?

I\'m not precisely sure how many versions of this movie are available. The UK altered video was released under the alias of Campus Killings, but the US cop y that I own states that it\'s the complete unedited edition, which could mean that there is a censo red print floating about somewhere? I\'d be fairly surprised if that was the case as Splatter Univer sity certainly isn\'t as gore-delicious as the hyperbole packaging would lead you to believe. One or two litres of corn syrup certainly don\'t stand up to gore hound\'s scrutiny when compared to the l ikes of Blood Rage or Pieces, so in this instance the movie is somewhat over hyped. One thing that m any critics have failed to mention is the charming lead performance from Francine Forbes, who ends u p carrying the entire picture on her shoulders throughout the 79-minute running time. Despite amateu rish direction from Richard Haines she still unveils some magnificent potential that should have lea d to the chance of another stab at serious acting under a more accomplished helmer. Unfortunately th at possibility never came, and bottom of the barrel bombs like Death Ring and Splitz certainly didn\ 't help to nurture a talent that could have improved dramatically under the right scholarship.

The rest of the cast members were par for the course of movie obscurity, especially the wooden plank teenagers who for some strange reason acted like they were auditioning for a remake of Grease or The Wanderers. The bog standard point and shoot direction couldn\'t have helped to build much co nfidence in the project and the fact that the few signs of potential were undermined by the clumsy h andling of the script writer left the feature effectively unredeemable. Perhaps the only claim of or iginality to be found in Haines\' slasher is the brave attempt for the contrasting conclusion. Let\' s just say that it\'s not a final that I was expecting to witness in a movie that was so typical of the cycle.

At one point in the runtime, one of the teens says, "Man that Parker bores me

```
to tears\xc2\x85" Well the same can be said for Splatter University, which never lifts the pace abov
e slow motion. With that said though, Francine Forbes made for a delectable scream queen and undoubt
edly one that I would have paid to watch again in a similar role. So that pretty much sums up this u
n-troma-tising ride. Slow paced, shoddy but still strangely alluring; you\'d have to be especially f
orgiving to give it a chance\xc2\x85']
25000
y:
[1]
[0]
25000
```

In [7]:

```
# read in the test data

X_test = []
y_test = []

path = './aclImdb/test/pos/'
X_test.extend([open(path + f).read() for f in os.listdir(path) if f.endswith('.txt')])
y_test.extend([1 for _ in range(12500)])

path = './aclImdb/test/neg/'
X_test.extend([open(path + f).read() for f in os.listdir(path) if f.endswith('.txt')])
y_test.extend([0 for _ in range(12500)])

print('x:')
print(X_test[:1])
print(X_test[-1:])
print(len(X_test))
print('y:')
print(y_test[:1])
print(y_test[-1:])
print(len(y_test))
```

```
x:
['A light, uplifting and engaging movie. Watching Irene Dunne is a delight! As you watch her, she ce
ases to be Irene Dunne and becomes in every way Paula Wharton.<br /><br />I have enjoyed Irene Dunne
 in every movie that I have seen and that would be nearly all of them. What a shame that most of her
 movies need restoration so badly. I do hope Irene Dunne movie are restored before it is too late th
ey are such treasures Thank goodness this is not the case with Over 21.<br /><br />It is a must see
if you like superb acting and witty comedy with serious overtones. I agree with a previous comment o
n the speech "The World and Apple Pie" it was one of the many highlights of the movie. I read somewh
ere that Irene Dunne helped in writing that speech along with Director Vidor (Irene Dunne was a very
 good and charitable person in private life) and it certainly seems to show through in her movies!']
["I have not figured out what the chosen title has to do with the movie. This is another gathering o
f monsters just like the HOUSE OF FRANKENSTEIN. Not exactly a masterful plot, but Universal needed t
o capitalize again.<br /><br />Dr. Edelman (Onslow Stevens) is either very ambitious or over the top
 in the ego department. He is working on the cure to keep Larry Talbot from turning into the Wolf Ma
n. Somehow Count Dracula happens to drop by to get a fix on his vampirism. And rounding out the good
 doctor's experiments is the restoring of the Frankenstein monster's energy. Along the way, the kind
 hearted doctor's blood is tainted with that of Dracula.<br /><br />John Carradine plays Dracula aga
in. This time he is more convincing. Lon Chaney Jr. as usual is the soulful Wolf Man. Glenn Strange
is the Frankenstein monster, who has very little to do this outing. Also with mentionable roles are
Lionel Atwill and Martha O'Driscoll."]
25000
y:
[1]
[0]
25000
```

In [8]:

```
#tokenize works to list of integers where each integer is a key to a word
reviewTokenizer = Tokenizer(nb_words=max_features)

reviewTokenizer.fit_on_texts(X_train)
```

In [9]:
```
#print top 10 words
#note zero is reserved for non frequent words
for word, value in reviewTokenizer.word_index.items():
    if value < 10:
        print(value, word)
```

```
9 it
6 is
8 in
4 of
2 and
3 a
7 br
1 the
5 to
```

In [10]:
```
#create int to word dictionary
wordDic = {}
for word, value in reviewTokenizer.word_index.items():
    wordDic[value] = word

#add a symbol for null placeholder
wordDic[0] = "!!!NA!!!"

print(wordDic[1])
print(wordDic[2])
print(wordDic[32])
```

```
the
and
an
```

In [11]:
```
#convert word strings to integer sequence lists
print(X_train[0])
print(reviewTokenizer.texts_to_sequences(X_train[:1]))
for value in reviewTokenizer.texts_to_sequences(X_train[:1])[0]:
    print(wordDic[value])

X_train = reviewTokenizer.texts_to_sequences(X_train)
X_test = reviewTokenizer.texts_to_sequences(X_test)
```

Here's a gritty, get-the-bad guys revenge story starring a relentless and rough Denzel Washington. He's three personalities here: a down-and-out-low-key-now drunk- former mercenary, then a loving father-type person to a little girl and then a brutal maniac on the loose seeking answers and revenge.

The story is about Washington hired to be a bodyguard for a little American girl living in Mexico, where kidnappings of children occur regularly (at least according to the movie.) He becomes attached to the kid, played winningly by THE child actress of our day, Dakota Fanning. When Fanning is kidnapped in front of him, Washington goes after the men responsible and spares no one. Beware: this film is not for the squeamish.

This is stylish film-making, which is good and bad. I liked it, but a number of people found it too frenetic for their tastes as the camera-work is one that could give you a headache. I thought it fit the tense storyline and was fascinating to view, but it's (the shaky camera) not for all tastes.

Besides the two stars, there is the always-interesting Christopher Walken, in an uncharacteristically low-key role, and a number of other fine actors.

The film panders to the base emotions in all of us, but it works.

[[1974, 3, 2539, 76, 1, 75, 490, 1057, 62, 1181, 3, 6162, 2, 2680, 3384, 2076, 237, 286, 3048, 130, 3, 177, 2, 43, 361, 1314, 147, 1816, 1135, 92, 3, 1711, 333, 549, 412, 5, 3, 114, 247, 2, 92, 3, 1767, 5046, 20, 1, 1885, 2985, 2754, 2, 1057, 7, 7, 1, 62, 6, 41, 2076, 2630, 5, 27, 3, 8240, 15, 3, 114, 295, 247, 578, 8, 2710, 118, 4, 473, 3914, 6945, 30, 219, 1790, 5, 1, 17, 26, 458, 3461, 5, 1, 551, 253, 31, 1, 503, 520, 4, 260, 248, 8820, 51, 8820, 6, 3528, 8, 1008, 4, 87, 2076, 268, 100, 1, 346, 1890, 2, 54, 28, 5515, 11, 19, 6, 21, 15, 1, 7, 7, 11, 6, 3003, 19, 228, 60, 6, 49, 2, 75, 10, 420, 9, 18, 3, 609, 4, 81, 255, 9, 96, 9831, 15, 65, 5178, 14, 1, 367, 154, 6, 28, 12, 97, 199, 22, 3, 6254, 10, 194, 9, 1180, 1, 3091, 766, 2, 13, 1426, 5, 647, 18, 42, 1, 5085, 367, 21, 15, 29, 5178, 7, 7, 1367, 1, 104, 378, 47, 6, 1, 207, 218, 1366, 3586, 8, 32, 361, 1314, 214, 2, 3, 609, 4, 82, 475, 153, 7, 7, 1, 19, 5, 1, 2807, 1435, 8, 29, 4, 175, 18, 9, 492]]

here's
a
gritty
get
the
bad
guys
revenge
story
starring
a
relentless
and
rough
denzel
washington
he's
three
personalities
here

a
down
and
out
low
key
now
drunk
former
then
a
loving
father
type
person
to
a
little
girl
and
then
a
brutal
maniac
on
the
loose
seeking
answers
and
revenge
br
br
the
story
is
about
washington
hired
to
be
a

bodyguard
for
a
little
american
girl
living
in
mexico
where
of
children
occur
regularly
at
least
according
to
the
movie
he
becomes
attached
to
the
kid
played
by
the
child
actress
of
our
day
fanning
when
fanning
is
kidnapped
in
front
of

```
him
washington
goes
after
the
men
responsible
and
no
one
beware
this
film
is
not
for
the
br
br
this
is
stylish
film
making
which
is
good
and
bad
i
liked
it
but
a
number
of
people
found
it
too
frenetic
for
```

```
their
tastes
as
the
camera
work
is
one
that
could
give
you
a
headache
i
thought
it
fit
the
tense
storyline
and
was
fascinating
to
view
but
it's
the
shaky
camera
not
for
all
tastes
br
br
besides
the
two
stars
there
```

```
is
the
always
interesting
christopher
walken
in
an
low
key
role
and
a
number
of
other
fine
actors
br
br
the
film
to
the
base
emotions
in
all
of
us
but
it
works
```

In [12]:
```
# Censor the data by having a max review length (in number of words)

#use this function to load data from keras pickle instead of munging as shown above
#(X_train, y_train), (X_test, y_test) = imdb.load_data(nb_words=max_features,
#                                                      test_split=0.2)

print(len(X_train), 'train sequences')
print(len(X_test), 'test sequences')

print("Pad sequences (samples x time)")
X_train = sequence.pad_sequences(X_train, maxlen=max_len)
X_test = sequence.pad_sequences(X_test, maxlen=max_len)
print('X_train shape:', X_train.shape)
print('X_test shape:', X_test.shape)
y_train = np.array(y_train)
y_test = np.array(y_test)
```

```
25000 train sequences
25000 test sequences
Pad sequences (samples x time)
X_train shape: (25000, 200)
X_test shape: (25000, 200)
```

In [13]:
```
#example of a sentence sequence, note that lower integers are words that occur more commonly
print("x:", X_train[0]) #per observation vector of 20000 words
print("y:", y_train[0]) #positive or negative review encoding
```

```
x: [ 177    2   43  361 1314  147 1816 1135   92    3 1711  333  549  412    5
    3  114  247    2   92    3 1767 5046   20    1 1885 2985 2754    2 1057
    7    7    1   62    6   41 2076 2630    5   27    3 8240   15    3  114
  295  247  578    8 2710  118    4  473 3914 6945   30  219 1790    5    1
   17   26  458 3461    5    1  551  253   31    1  503  520    4  260  248
 8820   51 8820    6 3528    8 1008    4   87 2076  268  100    1  346 1890
    2   54   28 5515   11   19    6   21   15    1    7    7   11    6 3003
   19  228   60    6   49    2   75   10  420    9   18    3  609    4   81
  255    9   96 9831   15   65 5178   14    1  367  154    6   28   12   97
  199   22    3 6254   10  194    9 1180    1 3091  766    2   13 1426    5
  647   18   42    1 5085  367   21   15   29 5178    7    7 1367    1  104
  378   47    6    1  207  218 1366 3586    8   32  361 1314  214    2    3
  609    4   82  475  153    7    7    1   19    5    1 2807 1435    8   29
    4  175   18    9  492]
y: 1
```

In [14]:
```
# double check that word sequences behave/final dimensions are as expected
print("y distribution:", np.unique(y_train, return_counts=True))
print("max x word:", np.max(X_train), "; min x word", np.min(X_train))
print("y distribution test:", np.unique(y_test, return_counts=True))
print("max x word test:", np.max(X_test), "; min x word", np.min(X_test))
```

```
y distribution: (array([0, 1]), array([12500, 12500]))
max x word: 9999 ; min x word 0
y distribution test: (array([0, 1]), array([12500, 12500]))
max x word test: 9999 ; min x word 0
```

In [15]:
```
print("most and least popular words: ")
print(np.unique(X_train, return_counts=True))
# as expected zero is the highly used word for words not in index
```

```
most and least popular words:
(array([   0,    1,    2, ..., 9997, 9998, 9999], dtype=int32), array([1084314,  232315,  115675, ..
.,      18,      21,      29]))
```

In [16]:
```
#set model hyper parameters
epochs = 6
embedding_neurons = 128
lstm_neurons = 64
batch_size = 32
```

In [17]:
```
# Forward Pass LSTM Network

# this is the placeholder tensor for the input sequences
sequence = Input(shape=(max_len,), dtype='int32')
# this embedding layer will transform the sequences of integers
# into vectors of size embedding
# embedding layer converts dense int input to one-hot in real time to save memory
embedded = Embedding(max_features, embedding_neurons, input_length=max_len)(sequence)
# normalize embeddings by input/word in sentence
bnorm = BatchNormalization()(embedded)

# apply forwards LSTM layer size lstm_neurons
forwards = LSTM(lstm_neurons, dropout_W=0.2, dropout_U=0.2)(bnorm)

# dropout
after_dp = Dropout(0.5)(forwards)
output = Dense(1, activation='sigmoid')(after_dp)

model_fdir_atom = Model(input=sequence, output=output)
# review model structure
print(model_fdir_atom.summary())
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================
input_1 (InputLayer)         (None, 200)               0
_________________________________________________________________
embedding_1 (Embedding)      (None, 200, 128)          1280000     input_1[0][0]
_________________________________________________________________
batchnormalization_1 (BatchNormal(None, 200, 128)      256         embedding_1[0][0]
_________________________________________________________________
lstm_1 (LSTM)                (None, 64)                49408       batchnormalization_1[0][0]
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0           lstm_1[0][0]
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 65          dropout_1[0][0]
=================================================================
Total params: 1329729
_________________________________________________________________
None
```

In [18]:
```
# Forward pass LSTM network

# try using different optimizers and different optimizer configs
model_fdir_atom.compile('adam', 'binary_crossentropy', metrics=['accuracy'])

print('Train...')
start_time = time.time()

history_fdir_atom = model_fdir_atom.fit(X_train, y_train,
                    batch_size=batch_size,
                    nb_epoch=epochs,
                    validation_data=[X_test, y_test],
                    verbose=2)

end_time = time.time()
average_time_per_epoch = (end_time - start_time) / epochs
print("avg sec per epoch:", average_time_per_epoch)
```

```
Train...
Train on 25000 samples, validate on 25000 samples
Epoch 1/6
387s - loss: 0.5194 - acc: 0.7319 - val_loss: 0.3669 - val_acc: 0.8514
Epoch 2/6
405s - loss: 0.2996 - acc: 0.8808 - val_loss: 0.3170 - val_acc: 0.8678
Epoch 3/6
343s - loss: 0.2100 - acc: 0.9184 - val_loss: 0.3833 - val_acc: 0.8556
Epoch 4/6
349s - loss: 0.1695 - acc: 0.9362 - val_loss: 0.3743 - val_acc: 0.8681
Epoch 5/6
320s - loss: 0.1246 - acc: 0.9556 - val_loss: 0.4199 - val_acc: 0.8654
Epoch 6/6
350s - loss: 0.0975 - acc: 0.9648 - val_loss: 0.4894 - val_acc: 0.8600
avg sec per epoch: 368.793916345
```

In [20]:
```
# Bi-directional Atom

# based on keras tutorial: https://github.com/fchollet/keras/blob/master/examples/imdb_bidirectional_lstm.py

# this is the placeholder tensor for the input sequences
sequence = Input(shape=(max_len,), dtype='int32')
# this embedding layer will transform the sequences of integers
# into vectors of size embedding
# embedding layer converts dense int input to one-hot in real time to save memory
embedded = Embedding(max_features, embedding_neurons, input_length=max_len)(sequence)
# normalize embeddings by input/word in sentence
bnorm = BatchNormalization()(embedded)

# apply forwards LSTM layer size lstm_neurons
forwards = LSTM(lstm_neurons, dropout_W=0.4, dropout_U=0.4)(bnorm)
# apply backwards LSTM
backwards = LSTM(lstm_neurons, dropout_W=0.4, dropout_U=0.4, go_backwards=True)(bnorm)

# concatenate the outputs of the 2 LSTMs
merged = merge([forwards, backwards], mode='concat', concat_axis=-1)
after_dp = Dropout(0.5)(merged)
output = Dense(1, activation='sigmoid')(after_dp)

model_bidir_atom = Model(input=sequence, output=output)
# review model structure
print(model_bidir_atom.summary())
```

Layer (type)	Output Shape	Param #	Connected to
input_2 (InputLayer)	(None, 200)	0	
embedding_2 (Embedding)	(None, 200, 128)	1280000	input_2[0][0]
batchnormalization_2 (BatchNormal	(None, 200, 128)	256	embedding_2[0][0]
lstm_2 (LSTM)	(None, 64)	49408	batchnormalization_2[0][0]
lstm_3 (LSTM)	(None, 64)	49408	batchnormalization_2[0][0]
merge_1 (Merge)	(None, 128)	0	lstm_2[0][0] lstm_3[0][0]
dropout_2 (Dropout)	(None, 128)	0	merge_1[0][0]
dense_2 (Dense)	(None, 1)	129	dropout_2[0][0]

Total params: 1379201

None

In [21]:
```
# Bi-directional Atom

# try using different optimizers and different optimizer configs
model_bidir_atom.compile('adam', 'binary_crossentropy', metrics=['accuracy'])

print('Train...')
start_time = time.time()

history_bidir_atom = model_bidir_atom.fit(X_train, y_train,
                    batch_size=batch_size,
                    nb_epoch=epochs,
                    validation_data=[X_test, y_test],
                    verbose=2)

end_time = time.time()
average_time_per_epoch = (end_time - start_time) / epochs
print("avg sec per epoch:", average_time_per_epoch)
```

```
Train...
Train on 25000 samples, validate on 25000 samples
Epoch 1/6
635s - loss: 0.5909 - acc: 0.6729 - val_loss: 0.4145 - val_acc: 0.8260
Epoch 2/6
660s - loss: 0.3806 - acc: 0.8371 - val_loss: 0.4096 - val_acc: 0.8499
Epoch 3/6
656s - loss: 0.2828 - acc: 0.8862 - val_loss: 0.3524 - val_acc: 0.8632
Epoch 4/6
636s - loss: 0.2249 - acc: 0.9148 - val_loss: 0.3772 - val_acc: 0.8645
Epoch 5/6
606s - loss: 0.1830 - acc: 0.9317 - val_loss: 0.4205 - val_acc: 0.8641
Epoch 6/6
615s - loss: 0.1514 - acc: 0.9428 - val_loss: 0.4554 - val_acc: 0.8640
avg sec per epoch: 644.418573181
```

In [22]:
```
# run simple linear regression to compare performance

#based on grid search done by:
#https://github.com/rasbt/python-machine-learning-book/blob/master/code/ch08/ch08.ipynb

#the tfidf vectors capture co-occurance statistics, think of each number representing how many times
#a word occured in a text and scaled by word frequency

tfidfTokenizer = Tokenizer(nb_words=max_features)
tfidfTokenizer.fit_on_sequences(X_train.tolist())
X_train_tfidf = np.asarray(tfidfTokenizer.sequences_to_matrix(X_train.tolist(), mode="tfidf"))
X_test_tfidf = np.asarray(tfidfTokenizer.sequences_to_matrix(X_test.tolist(), mode="tfidf"))
```

In [23]:
```
#check tfidf matrix
print(X_train_tfidf)
print(X_train_tfidf.shape, X_test_tfidf.shape)
```

```
[[ 0.          2.53756219  2.09414849 ...,  0.          0.          0.        ]
 [ 5.53114288  2.14733479  1.49183293 ...,  0.          0.          0.        ]
 [ 0.          2.48588573  2.09414849 ...,  0.          0.          0.        ]
 ...,
 [ 5.20620903  1.94673033  1.69633644 ...,  0.          0.          0.        ]
 [ 0.          2.22946644  1.85496169 ...,  0.          0.          0.        ]
 [ 0.          2.78627629  1.49183293 ...,  0.          0.          0.        ]]
(25000, 10000) (25000, 10000)
```

In [24]:
```
from sklearn.linear_model import LogisticRegression

model_tfidf_reg = LogisticRegression(random_state=0, C=0.001, penalty='l2', verbose=1)
model_tfidf_reg.fit(X_train_tfidf, y_train)
```

```
[LibLinear]
```

Out[24]:
```
LogisticRegression(C=0.001, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=0, solver='liblinear', tol=0.0001,
          verbose=1, warm_start=False)
```

In [25]:
```
from sklearn.metrics import accuracy_score
#calculate test and train accuracy
print("train acc:", accuracy_score(y_test, model_tfidf_reg.predict(X_train_tfidf)))
print("test acc:", accuracy_score(y_test, model_tfidf_reg.predict(X_test_tfidf)))
```

```
train acc: 0.9352
test acc: 0.88188
```

In [26]:
```
#get weights from embedding layer and visualize

print(model_bidir_atom.layers[1].get_config())
embmatrix = model_bidir_atom.layers[1].get_weights()[0]
print(embmatrix.shape)
```

```
{'W_constraint': None, 'activity_regularizer': None, 'name': 'embedding_2', 'output_dim': 128, 'trai
nable': True, 'init': 'uniform', 'input_dtype': 'int32', 'mask_zero': False, 'batch_input_shape': (N
one, 200), 'W_regularizer': None, 'dropout': 0.0, 'input_dim': 10000, 'input_length': 200}
(10000, 128)
```

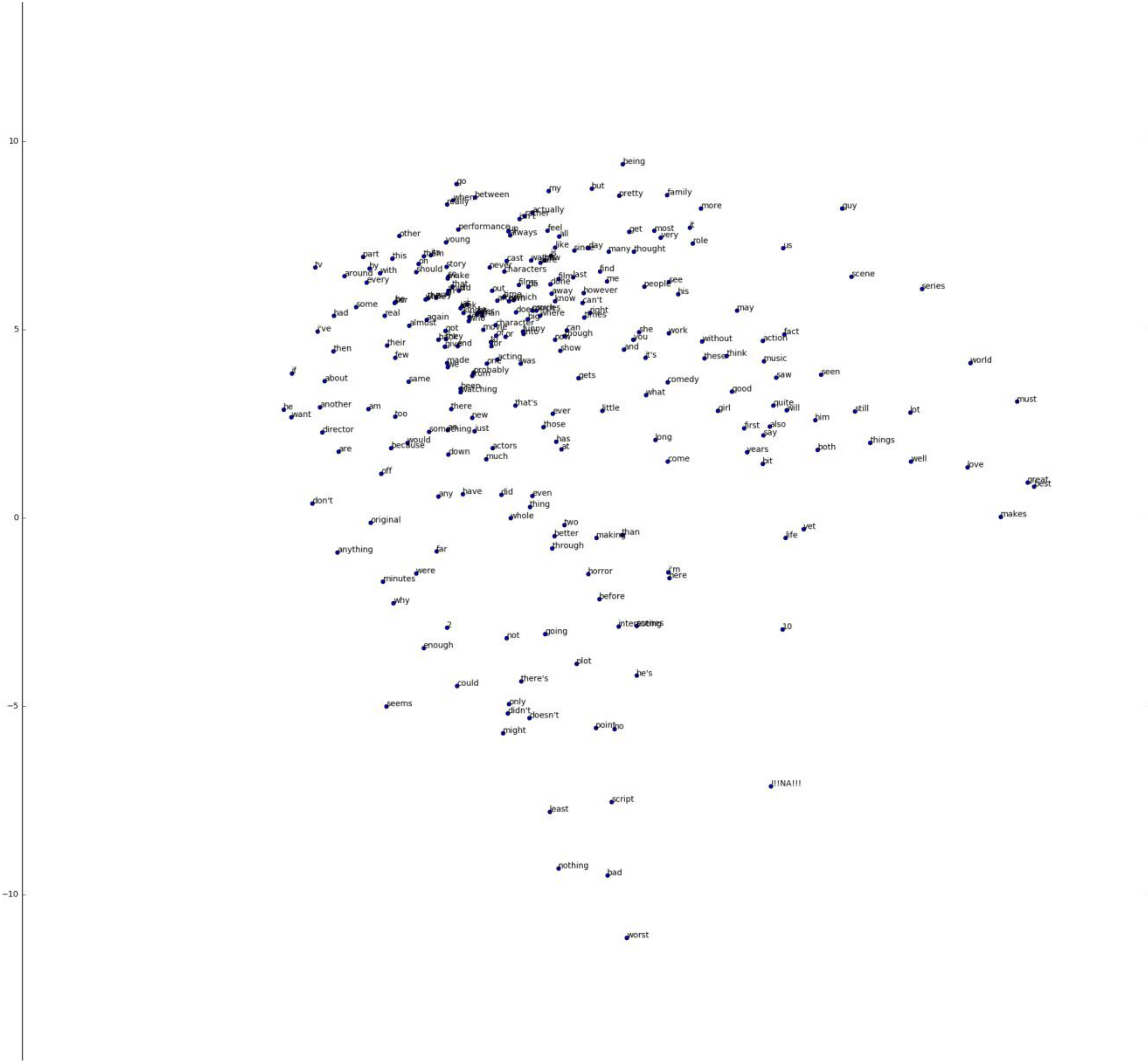
In [27]:
```
from sklearn.manifold import TSNE
topnwords = 5000
toptsne = TSNE(n_components=2, random_state=0)
tsneXY = toptsne.fit_transform(embmatrix[:topnwords, :])
tsneXY.shape
```

Out[27]:
```
(5000, 2)
```

In [28]:
```
%matplotlib inline
displaytopnwords = 250
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.scatter(tsneXY[:displaytopnwords, 0], tsneXY[:displaytopnwords, 1])

for i in range(displaytopnwords):
    ax.annotate(wordDic[i], (tsneXY[i, 0], tsneXY[i, 1]))

fig.set_size_inches(25, 25)
plt.show()
```


10
5
0
−5
−10
being
go
my
but
pretty
family
between
more
guy
actually
performance
always
feel
get
most
very
role
us
other
young
like
day
many
thought
part
this
cast
tv
around
with
should
story
never
characters
find
me
scene
every
make
films
last
people
see
series
out
which
away
however
his
some
know
can't
had
real
again
almost
may
i've
got
character
can
she
work
without
action
fact
their
you
then
and
it's
these
think
music
few
made
acting
was
show
world
probably
gets
if
about
same
watching
saw
seen
comedy
good
what
must
he
another
am
there
that's
little
quite
will
girl
still
lot
want
too
new
ever
him
first
also
director
just
those
say
would
long
things
has
because
actors
are
down
much
both
years
come
well
off
bit
love
great
best
any
have
did
even
don't
thing
whole
makes
original
two
better
making
than
yet
life
anything
far
through
were
horror
minutes
before
why
2
interesting
10
not
going
enough
plot
he's
there's
could
seems
only
didn't
doesn't
point
no
might
!!NA!!!
script
least
nothing
bad
worst

In [29]:
```
# Bi-directional rmsprop

# this example illistrate's that choice of optimizer is an important hyper-parameter for RNNs
# rmsprop gives substancially better results than atom
# in the literature these two optimizers commonly do well on RNNs

# this is the placeholder tensor for the input sequences
sequence = Input(shape=(max_len,), dtype='int32')
# this embedding layer will transform the sequences of integers
# into vectors of size embedding
# embedding layer converts dense int input to one-hot in real time to save memory
embedded = Embedding(max_features, embedding_neurons, input_length=max_len)(sequence)
# normalize embeddings by input/word in sentence
bnorm = BatchNormalization()(embedded)

# apply forwards LSTM layer size lstm_neurons
forwards = LSTM(lstm_neurons, dropout_W=0.4, dropout_U=0.4)(bnorm)
# apply backwards LSTM
backwards = LSTM(lstm_neurons, dropout_W=0.4, dropout_U=0.4, go_backwards=True)(bnorm)

# concatenate the outputs of the 2 LSTMs
merged = merge([forwards, backwards], mode='concat', concat_axis=-1)
after_dp = Dropout(0.5)(merged)
output = Dense(1, activation='sigmoid')(after_dp)

model_bidir_rmsprop = Model(input=sequence, output=output)
# review model structure
print(model_bidir_rmsprop.summary())
```

Layer (type)	Output Shape	Param #	Connected to
input_3 (InputLayer)	(None, 200)	0	
embedding_3 (Embedding)	(None, 200, 128)	1280000	input_3[0][0]
batchnormalization_3 (BatchNormal	(None, 200, 128)	256	embedding_3[0][0]
lstm_4 (LSTM)	(None, 64)	49408	batchnormalization_3[0][0]
lstm_5 (LSTM)	(None, 64)	49408	batchnormalization_3[0][0]
merge_2 (Merge)	(None, 128)	0	lstm_4[0][0] lstm_5[0][0]
dropout_3 (Dropout)	(None, 128)	0	merge_2[0][0]
dense_3 (Dense)	(None, 1)	129	dropout_3[0][0]

Total params: 1379201

None

In [31]:
```
# Bi-directional rmsprop

model_bidir_rmsprop.compile('rmsprop', 'binary_crossentropy', metrics=['accuracy'])

print('Train...')
start_time = time.time()

history_bidir_rmsprop = model_bidir_rmsprop.fit(X_train, y_train,
                    batch_size=batch_size,
                    nb_epoch=epochs,
                    validation_data=[X_test, y_test],
                    verbose=2)

end_time = time.time()
average_time_per_epoch = (end_time - start_time) / epochs
print("avg sec per epoch:", average_time_per_epoch)
```

```
Train...

INFO (theano.gof.compilelock): Refreshing lock /home/nv/.theano/compiledir_Linux-4.4--generic-x86_64
-with-debian-jessie-sid-x86_64-2.7.12-64/lock_dir/lock

Train on 25000 samples, validate on 25000 samples
Epoch 1/6
721s - loss: 0.5745 - acc: 0.6842 - val_loss: 0.5299 - val_acc: 0.7867
Epoch 2/6
605s - loss: 0.3705 - acc: 0.8439 - val_loss: 0.3479 - val_acc: 0.8684
Epoch 3/6
573s - loss: 0.2817 - acc: 0.8889 - val_loss: 0.3336 - val_acc: 0.8798
Epoch 4/6
610s - loss: 0.2387 - acc: 0.9077 - val_loss: 0.3566 - val_acc: 0.8770
Epoch 5/6
584s - loss: 0.2112 - acc: 0.9218 - val_loss: 0.3495 - val_acc: 0.8809
Epoch 6/6
592s - loss: 0.1857 - acc: 0.9309 - val_loss: 0.3884 - val_acc: 0.8770
avg sec per epoch: 620.017516812
```

In [32]:
```
#get weights from embedding layer and visualize

print(model_bidir_rmsprop.layers[1].get_config())
embmatrix = model_bidir_rmsprop.layers[1].get_weights()[0]
print(embmatrix.shape)
```

```
{'W_constraint': None, 'activity_regularizer': None, 'name': 'embedding_3', 'output_dim': 128, 'trai
nable': True, 'init': 'uniform', 'input_dtype': 'int32', 'mask_zero': False, 'batch_input_shape': (N
one, 200), 'W_regularizer': None, 'dropout': 0.0, 'input_dim': 10000, 'input_length': 200}
(10000, 128)
```

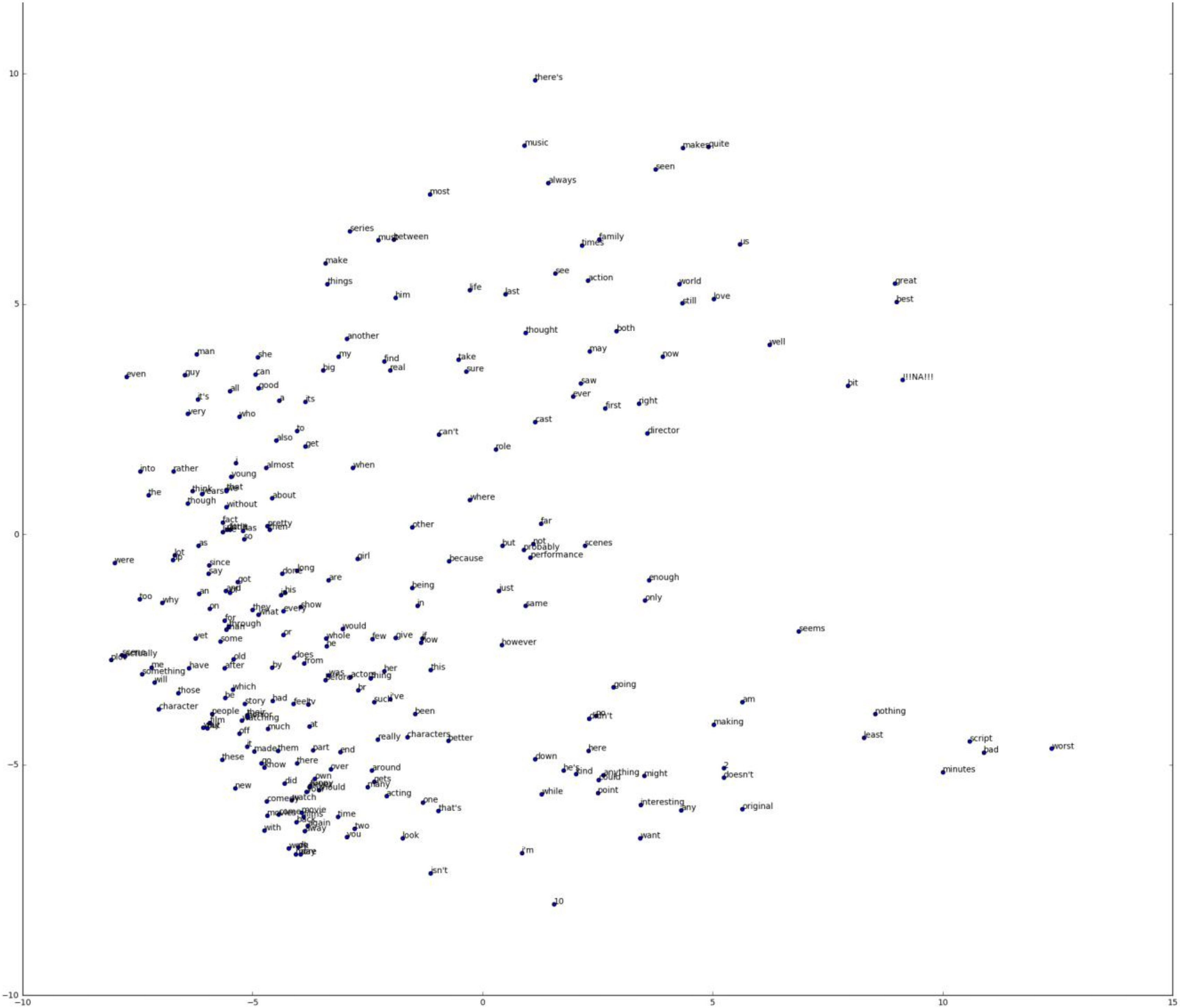
In [33]:
```
from sklearn.manifold import TSNE
topnwords = 5000
toptsne = TSNE(n_components=2, random_state=0)
tsneXY = toptsne.fit_transform(embmatrix[:topnwords, :])
tsneXY.shape
```

Out[33]:
```
(5000, 2)
```

In [35]:
```
%matplotlib inline
displaytopnwords = 250
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.scatter(tsneXY[:displaytopnwords, 0], tsneXY[:displaytopnwords, 1])

for i in range(displaytopnwords):
    ax.annotate(wordDic[i], (tsneXY[i, 0], tsneXY[i, 1]))

fig.set_size_inches(25, 25)
plt.show()
# notice that great, most, well are clustered
# bad don't even are clustered
# We've learned structure in our sentiment embedding
# neural networks give us this and other useful features for free
```


10
5
0
−5
−10
−10
−5
0
5
10
15
there's
music
makes
quite
seen
always
most
series
between
family
times
us
make
see
action
things
life
last
him
world
great
love
still
best
thought
both
another
well
man
she
my
find
take
may
now
even
guy
can
big
real
sure
saw
bit
!!!NA!!!
all
good
it's
a
its
ever
first
right
very
who
cast
to
also
can't
director
get
role
into
rather
almost
when
young
the
think
that
about
where
though
without
far
fact
pretty
other
so
as
but
probably
scenes
performance
were
since
girl
because
say
long
enough
got
are
being
just
too
why
an
his
only
on
they
what
every
show
in
same
through
would
seems
yet
some
whole
few
give
how
however
old
does
after
from
have
by
something
me
this
will
her
going
those
which
am
character
be
story
had
such
i've
nothing
people
been
making
much
at
script
off
really
characters
better
least
worst
bad
these
made
them
part
end
here
down
there
over
around
minutes
did
own
gets
many
acting
find
anything
might
doesn't
new
point
while
watch
one
that's
interesting
any
original
movie
time
again
two
with
away
you
look
want
i'm
isn't
10

In [37]:
```
#guide to chart above

for i in range(displaytopnwords):
    print((tsneXY[i, 0], tsneXY[i, 1], wordDic[i]))
```

```
(9.1211692360485888, 3.3552482852105676, '!!!NA!!!')
(-7.2644603398758392, 0.85623484414007378, 'the')
(-5.582567206511027, -1.2177951558293023, 'and')
(-4.4273495051914242, 2.911185050237155, 'a')
(-5.4973410617344447, -1.2591324915722879, 'of')
(-4.0366726879824721, 2.2548322902151607, 'to')
(-4.383855036639976, -1.3105004141292329, 'is')
(-2.7085226486866909, -3.3754355804585705, 'br')
(-1.4218316898014791, -1.5408734940116677, 'in')
(-5.1257651545397094, -4.5989591637595684, 'it')
(-5.3667350166442764, 1.549921539891653, 'i')
(-1.1325155048811215, -2.9399067570314772, 'this')
(-5.5702449759336261, 0.97744084617167037, 'that')
(-3.3528798064274206, -3.0532338222347368, 'was')
(-6.177514726109627, -0.23510272792882372, 'as')
(-5.6075017891167018, -1.8646128990918505, 'for')
(-4.7523456620900024, -6.4140369557827013, 'with')
(-3.9327567671782688, -6.0279587395764072, 'movie')
(0.42985409846709755, -0.24456759763656638, 'but')
(-5.9316355864741155, -4.0784487974359163, 'film')
(-5.9401240896718388, -1.6111132516576014, 'on')
(1.0978680301529811, -0.19535939056273438, 'not')
(-2.9597483343308082, -6.5670411623399287, 'you')
(-3.3489778995774606, -0.98391478999090964, 'are')
(-4.2985294176054403, -1.2561910573031525, 'his')
(-6.3828381553612079, -2.8941622133997256, 'have')
(-3.3922874353146044, -2.4186367689334038, 'he')
(-5.5995818142060605, -3.5445398676732629, 'be')
(-1.2995917172778786, -5.8146318802879389, 'one')
(-5.4968946150387543, 3.1160919419479121, 'all')
(-3.7670615621141192, -4.168124839649586, 'at')
(-4.5737578683029447, -2.8827239829689479, 'by')
(-6.165832337120591, -1.2858819008329687, 'an')
(-4.9991630985846633, -1.633479446104388, 'they')
(-5.2910832175745961, 2.5576130599734714, 'who')
(-5.1896827984658316, -0.10149043189996521, 'so')
(-3.8775655265796547, -2.7975413658651895, 'from')
```

```
(-5.6512721870895231, 0.056665573743656932, 'like')
(-2.140469090396564, -2.9578315202812684, 'her')
(-4.3336074784285952, -2.1798507558528466, 'or')
(0.34936695400468742, -1.2154786979602663, 'just')
(-4.5753405700554453, 0.7981214125705608, 'about')
(-6.1901876255409611, 2.9389625709183984, "it's")
(-5.9926287412894741, -4.1981097081870793, 'out')
(-5.212806709851165, 0.084253389410884616, 'has')
(-1.3197057199094018, -2.257103754451363, 'if')
(-5.6987612632758768, -2.3175619856948524, 'some')
(-4.0378414722868197, -4.9639128559266004, 'there')
(-4.8756283090330115, -1.7337855849213466, 'what')
(-4.8724662792115332, 3.177907227986644, 'good')
(-4.0649842065828867, -6.9376496639421097, 'more')
(-2.8235063384650414, 1.4464507254200525, 'when')
(-6.409700640281998, 2.618851021514975, 'very')
(-6.7348152826270136, -0.54971379206277193, 'up')
(2.4603302285663577, -3.928537233209731, 'no')
(-3.1527978359637143, -6.1259315412605879, 'time')
(-4.8864000133772576, 3.8455361472691219, 'she')
(-7.7456089106931216, 3.4308395110929282, 'even')
(-3.1308834251779243, 3.8624303629656271, 'my')
(-3.0458786751521303, -2.0442747205250003, 'would')
(-5.4328618122870083, -3.3671722377437741, 'which')
(3.5235221099567915, -1.4264510819230523, 'only')
(-5.1748761846678217, -3.6771769811832247, 'story')
(-2.2794664712668968, -4.4463793130550631, 'really')
(1.5725183301851537, 5.6717218500589341, 'see')
(-5.1249050926290538, -3.9127390275785423, 'their')
(-4.5730267534057933, -3.6009169173387066, 'had')
(-4.9419399782692262, 3.4764629961204228, 'can')
(-8.0036878520404393, -0.61389605231728805, 'were')
(-7.1972744067584093, -2.8819658107662622, 'me')
(6.22953658541255, 4.1230897619441809, 'well')
(-5.5691936401932782, -2.0629803318210538, 'than')
(-5.5739257874264414, 0.95264113385131721, 'we')
(-4.6720559668058881, -4.2193822973882025, 'much')
(-1.4729082979351917, -3.8919967491490413, 'been')
(10.889828677170922, -4.7271413502038735, 'bad')
(-3.8540033718313733, 1.9198727464419594, 'get')
(-7.1361473612640056, -3.205664757450188, 'will')
(-4.013939175206386, -6.7871793940589153, 'do')
```

```
(-4.4859141438087926, 2.0425488717949598, 'also')
(-7.4438580744366831, 1.3737926218368226, 'into')
(-5.8804038338635154, -3.8923009290140258, 'people')
(-1.5286438025894569, 0.1637821863098925, 'other')
(2.662523615330771, 2.7394599551157732, 'first')
(8.9596249243806412, 5.4540713725281842, 'great')
(-0.73116770307265944, -0.57903966882446622, 'because')
(-1.3376631252945936, -2.3433752740017249, 'how')
(-1.8955108346203497, 5.1375374228756092, 'him')
(-1.1488599666039039, 7.3889416037151765, 'most')
(-5.5605601849711972, 0.10789339710138118, "don't")
(-4.9656668361124474, -4.6983194795934864, 'made')
(-3.8513725737171054, 2.8844207149498957, 'its')
(-4.632764514234621, 0.10689506195314502, 'then')
(-6.0717888214630928, -4.1923680558363596, 'way')
(-3.4244077696003838, 5.8934982022325562, 'make')
(-4.4568699659171411, -4.6950828475790862, 'them')
(-7.4587826703497084, -1.3957904522793616, 'too')
(2.5230436936454765, -5.3247958508001441, 'could')
(4.3154393693549107, -5.9844159805699242, 'any')
(-4.6835669256141639, -6.0951513639335531, 'movies')
(-5.6110289356944465, -2.9021707984891707, 'after')
(-6.3136098656374235, 0.94166543138357217, 'think')
(-1.6441687592065848, -4.3929171700112182, 'characters')
(-4.1484851285175566, -5.7622918755466861, 'watch')
(-2.7834285030140848, -6.3860872262371808, 'two')
(-3.8894680460636502, -6.1196847567326609, 'films')
(-7.0389270174969436, -3.7829352369948364, 'character')
(3.7568513855725421, 7.9242586044545957, 'seen')
(-2.5059997833083281, -5.480415929547676, 'many')
(-1.5361017161470376, -1.1491189745891888, 'being')
(-0.28748360096605136, 5.311986561672982, 'life')
(-8.0795392354782951, -2.7195747647947037, 'plot')
(-3.7684559655568752, -5.479805309254612, 'never')
(-2.0952003559216026, -5.6715291346067058, 'acting')
(-5.4954135780751399, 0.10922726390067503, 'little')
(8.9901454957530689, 5.0465323374722875, 'best')
(5.0186793896500301, 5.1120566515812573, 'love')
(-3.308617029198889, -5.0955795841933247, 'over')
(-0.28207796473115082, 0.75590724603676696, 'where')
(-4.3116860560572112, -5.4015285621945965, 'did')
(-3.9590857594333997, -1.5742624291358314, 'show')
```

```
(-4.743193721459888, -5.0526805288607033, 'know')
(-5.2929823533313103, -4.3224085147524942, 'off')
(1.9586053516468693, 3.0054965656457422, 'ever')
(-4.1046466975241023, -2.6610863642815619, 'does')
(-0.74981332233081255, -4.4729430355267485, 'better')
(-3.8250366383086729, -5.5831753738353962, 'your')
(-3.0993096730588299, -4.7227461574867888, 'end')
(4.3413088804461246, 5.0193491409785604, 'still')
(-6.2125086954334297, 3.9197218726625347, 'man')
(2.2977889355483501, -4.6905809811864252, 'here')
(-5.6680250728478221, -4.8796543651460951, 'these')
(-5.9547049482878149, -0.85056875348720729, 'say')
(-7.8434218108218303, -2.6183472281285702, 'scene')
(1.2774740272050886, -5.6302893170315063, 'while')
(-6.96072255113988, -1.4823472750556488, 'why')
(2.2156276072625865, -0.24128267411125223, 'scenes')
(-4.8076325424627093, -4.9627110919499353, 'go')
(-2.3633424820747191, -3.6274287649699284, 'such')
(-7.405378486125338, -3.0259128078693855, 'something')
(-5.5155670175667888, -1.989246452659293, 'through')
(-3.5631552734248015, -5.5452553303168894, 'should')
(-4.045300285190284, -6.2353992392190101, 'back')
(0.85286682870630803, -6.9096358600457375, "i'm")
(-2.0126485730114285, 3.5661922689690169, 'real')
(-6.6238421867383321, -3.4361982851590711, 'those')
(-5.2324839372975482, -4.0301155426938706, 'watching')
(3.9058523985958185, 3.8619391636137581, 'now')
(-6.4113670288181988, 0.67559008571048751, 'though')
(5.2336205339284536, -5.2669508320066667, "doesn't")
(-6.1035559073106782, 0.88951710179300136, 'years')
(-5.4183979988908249, -2.7019618296505672, 'old')
(-2.435311246336938, -3.11791409021858, 'thing')
(-2.8721690135579037, -3.0908751635093861, 'actors')
(-4.2160194904645838, -6.8058525304073756, 'work')
(1.5461288449104116, -8.0163432612061811, '10')
(-3.4188562785316949, -3.1594718764492842, 'before')
(-2.9515663450980636, 4.2546675421278684, 'another')
(2.3090666541873524, -3.9960201562036946, "didn't")
(-5.3830332166411097, -5.5030005190123514, 'new')
(-3.7512930093262731, -5.4526573821574553, 'funny')
(8.5329959905828208, -3.8869844121163872, 'nothing')
(-7.7759185332888032, -2.6392637983879221, 'actually')
```

```
(4.3520393763842105, 8.3920662709006475, 'makes')
(3.5807126476109583, 2.1957339231566815, 'director')
(-1.7391776606925933, -6.5931228514952531, 'look')
(-2.1396790218550858, 3.7635001142664879, 'find')
(2.837448162518097, -3.3070383993584729, 'going')
(-2.4040326741832187, -2.270162741129238, 'few')
(0.93557963316607384, -1.5484431829540963, 'same')
(-3.6956781016877231, -4.6795491372568367, 'part')
(-3.8040673700016607, -6.3147595988731657, 'again')
(-4.3368658060992447, -1.655968049144759, 'every')
(-6.6999131049807685, -0.44314330605111185, 'lot')
(1.1393714646020969, 2.4429289130712353, 'cast')
(5.593465323222178, 6.2999076670423886, 'us')
(4.9043936710013938, 8.4169014128376531, 'quite')
(1.1364111598001807, -4.8667956602777327, 'down')
(3.4227609413521911, -6.586581743597149, 'want')
(4.2702481076056129, 5.4365629561745292, 'world')
(-3.3740010202624857, 5.4412566946005363, 'things')
(-4.6794984669898527, 0.18612598371116024, 'pretty')
(-5.4662260003454257, 1.2602019231067867, 'young')
(6.8634310651981476, -2.0977153968067443, 'seems')
(-2.4147878637376490, -5.1126209992661167, 'around')
(-5.3262411808001859, -1.0288415463767113, 'got')
(-5.1095604084376687, -3.9679177320410761, 'horror')
(0.41111159030379152, -2.39955939409472, 'however')
(-0.96031115552927726, 2.1709736905035859, "can't")
(-5.6570341438196534, 0.26022471831302696, 'fact')
(-0.53419597467402802, 3.7999290148883786, 'take')
(-3.4699300588914284, 3.570825675985045, 'big')
(3.6104893057778202, -0.98264056249305576, 'enough')
(-4.0330895798237298, -0.77682628836111955, 'long')
(0.93520264596561564, 4.3760806834112076, 'thought')
(-0.96273309144084362, -5.9960319157748962, "that's")
(2.9085382874605448, 4.4232447609782186, 'both')
(-1.9351104602005782, 6.4031161809264683, 'between')
(-2.8884826596778659, 6.5850673172686607, 'series')
(-1.8941399412343751, -2.2408169145169934, 'give')
(2.3213184801466498, 3.9849978694906598, 'may')
(5.639164724970783, -5.9589769215087305, 'original')
(-3.6463307283905837, -5.3026923881206658, 'own')
(2.2806086088140116, 5.5121709707985396, 'action')
(-2.0131476843997991, -3.5691395255937364, "i've")
```

```
(3.394657674885333, 2.8410694140865829, 'right')
(-5.5685124940563755, 0.59997828560251043, 'without')
(1.4230778239253532, 7.6340369096398861, 'always')
(2.1537009784396708, 6.2732022430436629, 'times')
(-4.6987731599183125, -5.7836993479011483, 'comedy')
(2.5057217244204741, -5.6012523147246753, 'point')
(-2.3642808208145083, -5.3618949649748657, 'gets')
(-2.2717834377342552, 6.3972336502737877, 'must')
(-4.4329220180535351, -6.0701017888406694, 'come')
(0.28795657606449809, 1.8507178043504213, 'role')
(-1.1290606514922099, -7.3427851263633288, "isn't")
(2.1277622737805468, 3.2813165718747599, 'saw')
(-4.702763472321414, 1.4475031380199257, 'almost')
(3.4316338860751365, -5.8626851784439271, 'interesting')
(8.2856976951788397, -4.4033893794445262, 'least')
(2.5297770856202044, 6.4043968839291914, 'family')
(-4.3553029630465927, -0.84039445903613041, 'done')
(1.1333749907291035, 9.8678307797847911, "there's")
(-3.4001933521082179, -2.2538737154434565, 'whole')
(7.9320800009727588, 3.2344072027758113, 'bit')
(0.90870127754317964, 8.4451381831528067, 'music')
(10.585960472294396, -4.4840818417515207, 'script')
(1.2641586165238741, 0.23932654927084043, 'far')
(5.0151467879044924, -4.1217096922167702, 'making')
(-6.4708221896849789, 3.4611588886766769, 'guy')
(2.6156840935642132, -5.2247738122973617, 'anything')
(9.9949610929164798, -5.1555968327342638, 'minutes')
(-4.1109658539633953, -3.677854973675291, 'feel')
(0.4922479377772111, 5.2239754118795787, 'last')
(-5.9423302075918931, -0.66063985507641065, 'since')
(3.5128229749654403, -5.2394013056675668, 'might')
(1.037846957778908, -0.49810999474719914, 'performance')
(1.750001099747452, -5.1114983538700907, "he's")
(5.2356262704039871, -5.0675006035045023, '2')
(0.88602569733389291, -0.33540699033719951, 'probably')
(2.0277933257551957, -5.1970044390235479, 'kind')
(5.6446363044972445, -3.6299381301745459, 'am')
(-3.8660450948509619, -6.4266239469634927, 'away')
(-6.2494089412853171, -2.2465761909600777, 'yet')
(-6.722481451423965, 1.370741254754009, 'rather')
(-3.798850934995829, -3.6825212352934269, 'tv')
(12.368484158687448, -4.6458487801049113, 'worst')
```

```
(-2.7249769352184527, -0.52673922168423604, 'girl')
(-3.9562759581817795, -6.9314353360247436, 'day')
(-0.36221171884599057, 3.5391120752446636, 'sure')
```

In [38]:
```
# Lets see what the embedding learned,
# provoking is close to great in cosine space, that's cool and definettly movie specific

from scipy.spatial.distance import euclidean

for value in np.argsort(np.apply_along_axis(lambda x: euclidean(x, embmatrix[reviewTokenizer.word_index
['great'],:]),
                                            1, embmatrix))[:20]:
    print((wordDic[value], euclidean(embmatrix[value,:], embmatrix[reviewTokenizer.word_index['great'],
:])))
```

```
('great', 0.0)
('wonderful', 0.5050570964813232)
('courage', 0.5211796164512634)
('sweet', 0.525455892086029)
('touched', 0.5273863077163696)
('favourite', 0.5292160511016846)
('keeper', 0.53465735912323)
('tremendous', 0.5355396270751953)
('cried', 0.538032054901123)
('stone', 0.5390382409095764)
('underrated', 0.5393837690353394)
('prince', 0.5457939505577087)
('steals', 0.5471921563148499)
('unique', 0.5480167269706726)
('wonderfully', 0.5497465133666992)
('absorbing', 0.5497508645057678)
('impact', 0.5505578517913818)
('terrific', 0.5513530373573303)
('unforgettable', 0.5517123937606812)
('chavez', 0.5537125468254089)
```

In [39]:
```
from scipy.spatial.distance import cosine

for value in np.argsort(np.apply_along_axis(lambda x: cosine(x, embmatrix[reviewTokenizer.word_index['great'],:]),
                                            1, embmatrix))[:20]:
    print((wordDic[value], cosine(embmatrix[value,:], embmatrix[reviewTokenizer.word_index['great'],:])))
```

```
('great', -4.968141431582751e-08)
('wonderful', 0.22880693842383337)
('gem', 0.24017793216085392)
('excellent', 0.27010345380361955)
('favourite', 0.27908294232026432)
('terrific', 0.28615594423807988)
('courage', 0.2861641785088691)
('favorite', 0.29017409763723823)
('wonderfully', 0.29305467770506521)
('unforgettable', 0.29309888019949715)
('touched', 0.29871607709852777)
('perfect', 0.299027911919717)
('underrated', 0.2993400818958184)
('cried', 0.29983298338034214)
('delightful', 0.30433546962950253)
('amazing', 0.30735002923320831)
('sweet', 0.30837306478078952)
('today', 0.31421591438039453)
('crafted', 0.31456208191961121)
('prince', 0.31530052864802272)
```

In [40]:
```
reviewTokenizer.word_index['great']
```

Out[40]: 84

In [ ]:
```

```