A Sentimental Journey

Sentiment Analysis of Movie Reviews

Why sentiment analysis?

Textual data doesn't always come categorized / labeled
- tweets
- blog posts
- mails
- support tickets

Movie reviews - www.imdb.com

The data
- 25,000 labeled training reviews plus 25,000 test reviews

Load preprocessed data
In [1]:

```python
import io
import pandas as pd
import numpy as np

with io.open('data/aclImdb/train-pos.txt', encoding='utf-8') as f:
    train_pos = pd.DataFrame({'review': list(f)})
with io.open('data/aclImdb/train-neg.txt', encoding='utf-8') as f:
    train_neg = pd.DataFrame({'review': list(f)})
train_reviews = pd.concat([train_neg, train_pos], ignore_index=True)

with io.open('data/aclImdb/test-pos.txt', encoding='utf-8') as f:
    test_pos = pd.DataFrame({'review': list(f)})
with io.open('data/aclImdb/test-neg.txt', encoding='utf-8') as f:
    test_neg = pd.DataFrame({'review': list(f)})
test_reviews = pd.concat([test_neg, test_pos], ignore_index=True)

X_train = train_reviews['review']
X_test = test_reviews['review']
y_train = np.append(np.zeros(12500), np.ones(12500))
y_test = np.append(np.zeros(12500), np.ones(12500))
```

---

**First review**

In [2]:

```python
X_train[0]
```

Out[2]:

```
"a reasonable effort is summary for this film . a good sixties film but lacking a
ny sense of achievement . maggie smith gave a decent performance which was believa
ble enough but not as good as she could have given , other actors were just dreadf
ul ! a terrible portrayal . it wasn't very funny and so it didn't really achieve
its genres as it wasn't particularly funny and it wasn't dramatic . the only genre
achieved to a satisfactory level was romance . target audiences were not hit and t
he movie sent out confusing messages . a very basic plot and a very basic storylin
e were not pulled off or performed at all well and people were left confused as to
why the film wasn't as good and who the target audiences were etc . however maggie
was quite good and the storyline was alright with moments of capability . 4 . 
```

**Good or bad?**

What the annotators thought

In [3]:

```python
y_train[0]
```

Out[3]:

```
0.0
```

**A naive approach: Word counts**
Word count in a nutshell
- sum positive words (weighted)
- sum negative words (weighted)
- highest score wins

No-one's gonna sit there and categorize all the words.

Need magic?
Not yet.

We have a training set where reviews have been labeled as good or bad:

<table>
<thead>
<tr>
<th>word</th>
<th>beautiful</th>
<th>bad</th>
<th>awful</th>
<th>decent</th>
<th>horrible</th>
<th>ok</th>
<th>awesome</th>
</tr>
</thead>
<tbody>
<tr>
<td>review 1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>review 2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>review 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Classification
From this, we can algorithmically determine the words' polarities and weights.

<table>
<thead>
<tr>
<th>word</th>
<th>beautiful</th>
<th>bad</th>
<th>awful</th>
<th>decent</th>
<th>horrible</th>
<th>ok</th>
<th>awesome</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td>3.4</td>
<td>-2.9</td>
<td>-5.6</td>
<td>-0.2</td>
<td>-4.9</td>
<td>-0.1</td>
<td>5.2</td>
</tr>
</tbody>
</table>

Right.
But...

There is an additional difficulty.
From our example review above:

- performance which was believable enough but not as good as she could have given
- lacking any sense of achievement
- it wasn’t very funny
- the only genre achieved to a satisfactory level was romance
Context matters

Funny => 😊

Very funny => 😊😊

Wasn't unbelievably funny => 👎

... what if it were

- "wasn't utterly unbelievably funny"
- "however, I wouldn't say that it wasn't utterly unbelievably funny"

Unigrams, bigrams, trigrams - what should we look at?

Instead of guessing let's check what works best on our dataset.

Most frequent unigrams

In [4]:

```python
word_count_1gram = pd.read_csv('word_counts_sorted_ngram_1_stopwords_removed.csv',
                             usecols=['word', 'count'])
word_count_1gram.head(10)
```

Out[4]:

<table>
<thead>
<tr>
<th>count</th>
<th>word</th>
</tr>
</thead>
<tbody>
<tr>
<td>44047</td>
<td>movie</td>
</tr>
<tr>
<td>42623</td>
<td>but</td>
</tr>
<tr>
<td>40159</td>
<td>film</td>
</tr>
<tr>
<td>30632</td>
<td>not</td>
</tr>
<tr>
<td>26795</td>
<td>one</td>
</tr>
<tr>
<td>20281</td>
<td>like</td>
</tr>
<tr>
<td>15147</td>
<td>good</td>
</tr>
<tr>
<td>14067</td>
<td>very</td>
</tr>
<tr>
<td>12727</td>
<td>time</td>
</tr>
<tr>
<td>12716</td>
<td>no</td>
</tr>
</tbody>
</table>

Most frequent bigrams
In [5]:

```python
word_count_2grams = pd.read_csv('word_counts_sorted_ngram_2_stopwords_removed.csv',
                               usecols=['word', 'count'])
word_count_2grams.head(10)
```

Out[5]:

<table>
<thead>
<tr>
<th>count</th>
<th>word</th>
</tr>
</thead>
<tbody>
<tr>
<td>1925</td>
<td>but not</td>
</tr>
<tr>
<td>1321</td>
<td>ever seen</td>
</tr>
<tr>
<td>1284</td>
<td>not only</td>
</tr>
<tr>
<td>1200</td>
<td>very good</td>
</tr>
<tr>
<td>1113</td>
<td>special effects</td>
</tr>
<tr>
<td>1043</td>
<td>even though</td>
</tr>
<tr>
<td>1032</td>
<td>movie but</td>
</tr>
<tr>
<td>1024</td>
<td>don know</td>
</tr>
<tr>
<td>1007</td>
<td>movie not</td>
</tr>
<tr>
<td>888</td>
<td>one best</td>
</tr>
</tbody>
</table>

**Most frequent trigrams**

In [6]:

```python
word_count_3grams = pd.read_csv('word_counts_sorted_ngram_3_stopwords_removed.csv',
                                 usecols=['word', 'count'])
word_count_3grams.head(10)
```

Out[6]:

<table>
<thead>
<tr>
<th>count</th>
<th>word</th>
</tr>
</thead>
<tbody>
<tr>
<td>262</td>
<td>movie ever seen</td>
</tr>
<tr>
<td>243</td>
<td>worst movie ever</td>
</tr>
<tr>
<td>205</td>
<td>don waste time</td>
</tr>
<tr>
<td>177</td>
<td>movies ever seen</td>
</tr>
<tr>
<td>164</td>
<td>new york city</td>
</tr>
<tr>
<td>162</td>
<td>don get wrong</td>
</tr>
<tr>
<td>160</td>
<td>one worst movies</td>
</tr>
<tr>
<td>141</td>
<td>worst movies ever</td>
</tr>
<tr>
<td>120</td>
<td>film ever seen</td>
</tr>
<tr>
<td>114</td>
<td>movie ever made</td>
</tr>
</tbody>
</table>

**In search for the right combination (grid search)**
Which classifier works best?

In combination with which input?
- Unigrams? Bigrams? Trigrams?

With which parameter settings?
- e.g., regularization, number of iterations...

FEW
HOURS
LATER

And the winner is ...

Best accuracy per classifier (test set)

<table>
<thead>
<tr>
<th></th>
<th>1-grams with stopword filtering</th>
<th>1-2-grams with stopword filtering</th>
<th>1-3-grams no stopword filtering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logistic Regression</td>
<td>0.89</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Support Vector Machine</td>
<td></td>
<td>0.84</td>
<td></td>
</tr>
<tr>
<td>Random Forest</td>
<td>0.84</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Exploring the Logistic Regression best fit
```python
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression

stopwords_nltk = set(stopwords.words("english"))
relevant_words = set(["not", "nor", "no", "wasn", "ain", "aren", "very", "only", "but", "don", "isn", "weren"])
stopwords_filtered = list(stopwords_nltk.difference(relevant_words))
vectorizer = CountVectorizer(stop_words=stopwords_filtered, max_features=10000, ngram_range=(1,2))
X_train_features = vectorizer.fit_transform(X_train)
X_test_features = vectorizer.transform(X_test)

logistic_model = LogisticRegression(C=0.03)
logistic_model.fit(X_train_features, y_train)
```

Out[7]:
```
LogisticRegression(C=0.03, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                   penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
                   verbose=0, warm_start=False)
```

**Which words make it positive?**

In [8]:

```python
vocabulary = vectorizer.get_feature_names()
coefs = logistic_model.coef_
word_importances = pd.DataFrame({"word": vocabulary, "coef": coefs.tolist()[0]})
word_importances_sorted = word_importances.sort_values(by='coef', ascending=False)
word_importances_sorted[:10]
```

Out[8]:
```
<table>
<thead>
<tr>
<th>word</th>
<th>coef</th>
</tr>
</thead>
<tbody>
<tr>
<td>excellent</td>
<td>0.672635</td>
</tr>
<tr>
<td>perfect</td>
<td>0.563958</td>
</tr>
<tr>
<td>wonderful</td>
<td>0.521026</td>
</tr>
<tr>
<td>superb</td>
<td>0.520818</td>
</tr>
<tr>
<td>favorite</td>
<td>0.505146</td>
</tr>
<tr>
<td>amazing</td>
<td>0.502118</td>
</tr>
<tr>
<td>must see</td>
<td>0.481505</td>
</tr>
<tr>
<td>loved</td>
<td>0.461807</td>
</tr>
<tr>
<td>funniest</td>
<td>0.458645</td>
</tr>
<tr>
<td>enjoyable</td>
<td>0.453481</td>
</tr>
</tbody>
</table>
```

**Which words make it negative?**
Which 2-grams make it positive?

In [10]:
word_importances_bigrams = word_importances_sorted[word_importances_sorted.word.apply(lambda c: len(c.split())) >= 2]
word_importances_bigrams[:10]

Out[10]:

<table>
<thead>
<tr>
<th>coef</th>
<th>word</th>
</tr>
</thead>
<tbody>
<tr>
<td>5923</td>
<td>must see</td>
</tr>
<tr>
<td>3</td>
<td>10 10</td>
</tr>
<tr>
<td>6350</td>
<td>one best</td>
</tr>
<tr>
<td>9701</td>
<td>well worth</td>
</tr>
<tr>
<td>5452</td>
<td>may not</td>
</tr>
<tr>
<td>6139</td>
<td>not bad</td>
</tr>
<tr>
<td>6970</td>
<td>pretty good</td>
</tr>
<tr>
<td>2259</td>
<td>definitely worth</td>
</tr>
<tr>
<td>5208</td>
<td>love movie</td>
</tr>
<tr>
<td>9432</td>
<td>very good</td>
</tr>
</tbody>
</table>

Which 2-grams make it negative?
In [11]:

word_importances_bigrams[-11:-1]

Out[11]:

<table>
<thead>
<tr>
<th>coef</th>
<th>word</th>
</tr>
</thead>
<tbody>
<tr>
<td>6431</td>
<td>-0.247169 only good</td>
</tr>
<tr>
<td>3151</td>
<td>-0.250090 fast forward</td>
</tr>
<tr>
<td>9861</td>
<td>-0.264564 worst movie</td>
</tr>
<tr>
<td>6201</td>
<td>-0.324169 not recommend</td>
</tr>
<tr>
<td>6153</td>
<td>-0.332796 not even</td>
</tr>
<tr>
<td>6164</td>
<td>-0.333147 not funny</td>
</tr>
<tr>
<td>6217</td>
<td>-0.357056 not very</td>
</tr>
<tr>
<td>6169</td>
<td>-0.368976 not good</td>
</tr>
<tr>
<td>6421</td>
<td>-0.437750 one worst</td>
</tr>
<tr>
<td>9609</td>
<td>-0.451138 waste time</td>
</tr>
</tbody>
</table>

0.89 accuracy is pretty good

With a different approach, can it get any better?

Beyond word counts:

Word embeddings

Bag-of-words (or bag-of-ngrams) basically uses one-hot encoding:
In [16]:

# Tidy datasets are all alike but every messy dataset is messy in its own way.
words = pd.DataFrame({'tidy': [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                      'dataset': [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                      'is': [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                      'all': [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                      'alike': [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                      'every': [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                      'in': [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                      'its': [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                      'own': [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                      'way': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                      '0': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]})

Out[16]:

<table>
<thead>
<tr>
<th></th>
<th>alike</th>
<th>all</th>
<th>but</th>
<th>dataset</th>
<th>every</th>
<th>in</th>
<th>is</th>
<th>its</th>
<th>messy</th>
<th>own</th>
<th>tidy</th>
<th>way</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

In this model, all words are equally distant from each other.

How about similarities between words - semantic dimensions?

To uncover similarities between words

- build word co-occurrence matrix
- perform dimensionality reduction
Co-occurrence matrix

"Tidy datasets are all alike but every messy dataset is messy in its own way." (Hadley Wickham)

“Happy families are all alike; every unhappy family is unhappy in its own way." (Lev Tolstoj)

<table>
<thead>
<tr>
<th></th>
<th>tidy</th>
<th>dataset</th>
<th>is</th>
<th>all</th>
<th>alike</th>
<th>but</th>
<th>every</th>
<th>messy</th>
<th>in</th>
<th>its</th>
<th>own</th>
<th>way</th>
<th>happy</th>
<th>family</th>
<th>unhappy</th>
</tr>
</thead>
<tbody>
<tr>
<td>tidy</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>dataset</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>is</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

[and so on]

In reality, this approach often is not practical. Enter ...

**Distributed Representations - the Neural Network Approach**

Infer the meaning of a word from the contexts it appears in:

- predict word probability depending on surrounding words
- improve prediction at every iteration (backpropagation)

**Distributed Representation of Words**

- Every word is represented not by a single “hot“ bit, but by a vector of continuously-scaled values
- This allows us to find semantic similarities

**word2vec**


- Continuous Bag of Words (CBOW)
- Skip-Gram
Continuous Bag of Words

from: Mikolov et al. 2013

Skip-gram

from: Mikolov et al. 2013
Table 1: Examples of five types of semantic and nine types of syntactic questions in the Semantic-Syntactic Word Relationship test set.

<table>
<thead>
<tr>
<th>Type of relationship</th>
<th>Word Pair 1</th>
<th>Word Pair 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common capital city</td>
<td>Athens</td>
<td>Oslo</td>
</tr>
<tr>
<td>All capital cities</td>
<td>Astana</td>
<td>Harare</td>
</tr>
<tr>
<td>Currency</td>
<td>Angola</td>
<td>Iran</td>
</tr>
<tr>
<td>City-in-state</td>
<td>Chicago</td>
<td>Stockton</td>
</tr>
<tr>
<td>Man-Woman</td>
<td>brother</td>
<td>grandson</td>
</tr>
<tr>
<td>Adjective to adverb</td>
<td>apparent</td>
<td>rapid</td>
</tr>
<tr>
<td>Opposite</td>
<td>possibly</td>
<td>unethical</td>
</tr>
<tr>
<td>Comparative</td>
<td>great</td>
<td>tough</td>
</tr>
<tr>
<td>Superlative</td>
<td>easy</td>
<td>lucky</td>
</tr>
<tr>
<td>Present Participle</td>
<td>think</td>
<td>read</td>
</tr>
<tr>
<td>Nationality adjective</td>
<td>Switzerland</td>
<td>Cambodia</td>
</tr>
<tr>
<td>Past tense</td>
<td>walking</td>
<td>swam</td>
</tr>
<tr>
<td>Plural nouns</td>
<td>mouse</td>
<td>dollar</td>
</tr>
<tr>
<td>Plural verbs</td>
<td>work</td>
<td>speak</td>
</tr>
</tbody>
</table>

from: Mikolov et al. 2013

"Athens" - "Greece" + "Norway" = ?

"walking" - "walked" + "swam" = ?

Word embeddings for the IMDB dataset

word2vec in Python

- provided by gensim library: https://radimrehurek.com/gensim/models/word2vec.html

Load the pre-trained model
In [17]:

```python
from gensim.models import word2vec
# load the trained model from disk
model = word2vec.Word2Vec.load('models/word2vec_100features')
print(model.syn0.shape)
print(model['movie'])
```

```
(20166, 100)
[-0.02515472  0.16707493 -0.05629794 -0.12409752 -0.01091802 -0.13798206
 -0.09231102 -0.09140468 -0.05452388 -0.03556577 -0.08269091 -0.00567267
 -0.09523809 -0.06195637  0.05440474  0.06227686  0.12369317 -0.01537143
 -0.0089783  -0.00528997 -0.04277094  0.07739993 -0.01932896  0.081738
 -0.22357117 -0.14976217  0.05551976  0.13742755 -0.15443996 -0.05471482
 -0.0009601  0.08932991 -0.05292547  0.16765165 -0.05905993 -0.05231098
 -0.08250861 -0.0341751  0.14372236  0.03478728 -0.01529499 -0.0296018
 0.01079863 -0.06377127  0.04163288 -0.07192093  0.25450262 -0.07382536
 -0.07778623  0.07499653 -0.12951691  0.01970425  0.13499822  0.01038768
 0.06625408  0.11575779  0.10367264  0.03894637 -0.07102726  0.00343542
 0.24314043  0.15759529 -0.09808595  0.04601007 -0.01187227 -0.16023833
 -0.17658544 -0.12622575 -0.04592994  0.08045016 -0.11856512  0.04920706
 0.20129348  0.08923753 -0.06545419 -0.05853761 -0.08146987 -0.06782326
 0.17082241  0.02575272  0.058911  0.13305175 -0.1224633 -0.01143302
 0.01318115  0.07662909 -0.09469278 -0.05230315 -0.0121863  0.12192696
 0.19957212 -0.075518  0.16371782 -0.07655586 -0.09539564  0.11822125
 0.04177237  0.11499111 -0.09205962  0.09952193]
```

**Which words are similar to awesome?**

In [18]:

```python
model.most_similar('awesome', topn=10)
```

```
[(u'amazing', 0.7929322123527527),
 (u'incredible', 0.7127916812896729),
 (u'awful', 0.7072071433067322),
 (u'excellent', 0.6961393356323242),
 (u'fantastic', 0.6925109624862671),
 (u'astounding', 0.613292932510376),
 (u'terrific', 0.6013768911361694)]
```

... and to awful?
In [19]:

model.most_similar('awful', topn=10)

Out[19]:

[(u'terrible', 0.8212785124778748),
 (u'horrible', 0.7955455183982849),
 (u'atrocious', 0.7824822664260864),
 (u'dreadful', 0.772217273121582),
 (u'appalling', 0.724443893432617),
 (u'horrendous', 0.7235419154167175),
 (u'abyssmal', 0.720653235912323),
 (u'amazing', 0.708114743232727),
 (u'awesome', 0.7072070837020874),
 (u'bad', 0.6963905096054077)]

Can we "subtract out" awful?

In [20]:

model.most_similar(positive=['awesome'], negative=['awful'])

Out[20]:

[(u'jolly', 0.3947059214115143),
 (u'midget', 0.38988131284713745),
 (u'knight', 0.3789686858654022),
 (u'spooky', 0.36937469244003296),
 (u'nice', 0.3680706322193146),
 (u'looney', 0.3676275610923767),
 (u'ho', 0.3594890832901001),
 (u'gotham', 0.35877227783203125),
 (u'lookalike', 0.3579031229019165),
 (u'devilish', 0.3555443882942197)]

Let's try this again with good - bad: Good ...

In [21]:

model.most_similar('good', topn=10)

Out[21]:

[(u'bad', 0.769078254699707),
 (u'decent', 0.7574324607849121),
 (u'great', 0.7527369260787964),
 (u'nice', 0.6981208324432373),
 (u'cool', 0.653165340423584),
 (u'fine', 0.6289849877357483),
 (u'terrific', 0.6136247515678406),
 (u'terrible', 0.6056008338928223),
 (u'fantastic', 0.596002995967865),
 (u'solid', 0.5957943201065063)]

... and bad:
In [22]:
model.most_similar('bad', topn=10)

Out[22]:
[(u'good', 0.769078254699707),
 (u'terrible', 0.7315745949745178),
 (u'horrible', 0.7259382009506226),
 (u'awful', 0.6963905096054077),
 (u'lame', 0.6728411912918091),
 (u'stupid', 0.6556650996208191),
 (u'dumb', 0.628576934337616),
 (u'ousy', 0.6129568815231323),
 (u'cheesy', 0.6102402210235596),
 (u'poor', 0.5851123929023743)]

So good minus bad is ...

In [23]:
model.most_similar(positive=['good'], negative=['bad'])

Out[23]:
[(u'nice', 0.4700997471809387),
 (u'fine', 0.46652451157569885),
 (u'solid', 0.43668174743652344),
 (u'wonderful', 0.4121875464916229),
 (u'pleasant', 0.4049694538116455),
 (u'decent', 0.3975681960582733),
 (u'commendable', 0.39051422476768494),
 (u'splendid', 0.38586685061454773),
 (u'promising', 0.38155609369277954),
 (u'delightful', 0.3809554278850554)]

Which word doesn't match?

In [24]:
model.doesnt_match("good bad awful terrible".split())

Out[24]:
'good'

In [25]:
model.doesnt_match("awesome bad awful terrible".split())

Out[25]:
'awesome'

In [26]:
model.doesnt_match("nice pleasant fine excellent".split())

Out[26]:
'excellent'
How about our classification task then?
- we have one vector per word
- we need one vector per review
- one way to get there: averaging vectors
- but this way information will be lost!

Classification accuracies with word vectors (word2vec)

<table>
<thead>
<tr>
<th>Best accuracies per classifier</th>
<th>Bag of words</th>
<th>word2vec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logistic Regression</td>
<td>0.89</td>
<td>0.83</td>
</tr>
<tr>
<td>Support Vector Machine</td>
<td>0.84</td>
<td>0.70</td>
</tr>
<tr>
<td>Random Forest</td>
<td>0.84</td>
<td>0.80</td>
</tr>
</tbody>
</table>

In the word2vec model, we lose information
- need to average over vectors in order to arrive at a synthetic "paragraph vector"
- paragraph context is lost (by design)

How about having real paragraph vectors?

Paragraph vectors: doc2vec

- Distributed Memory Model of Paragraph Vectors (PV-DM)
  - paragraph vector gets averaged together with word vectors
  - paragraph vectors can be directly input to machine learning classifiers

- Distributed Bag of Words (PV-DBOW)
  - context words are ignored
Distributed Memory Model of Paragraph Vectors (PV-DM)


**doc2vec in Python:**

- also provided by gensim [https://radimrehurek.com/gensim/models/doc2vec.html](https://radimrehurek.com/gensim/models/doc2vec.html)
- see gensim doc2vec tutorial [https://github.com/RaRe-Technologies/gensim/blob/develop/docs/notebooks/doc2vec-IMDB.ipynb](https://github.com/RaRe-Technologies/gensim/blob/develop/docs/notebooks/doc2vec-IMDB.ipynb) for example usage and configuration

Load pre-trained models
from gensim.models import Doc2Vec
models_dir = 'models'
filenames = ['dmc', 'cbow', 'dmm']
files = map(lambda f: '/'.join([models_dir, f]), filenames)
models = [Doc2Vec.load(fname) for fname in files]

In [28]:
[str(model) for model in models]

Out[28]:
['Doc2Vec(dm/c,d100,n5,w5,mc2,t4)',
 'Doc2Vec(dbow,d100,n5,mc2,t4)',
 'Doc2Vec(dm/m,d100,n5,w10,mc2,t4)']

**Logistic Regression accuracy**

<table>
<thead>
<tr>
<th>Model Type</th>
<th>Test Vectors Inferred</th>
<th>Test Vectors from Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed memory, vectors averaged (dm/m)</td>
<td>0.81</td>
<td>0.87</td>
</tr>
<tr>
<td>Distributed memory, vectors concatenated (dm/c)</td>
<td>0.80</td>
<td>0.82</td>
</tr>
<tr>
<td>Distributed bag of words (dbow)</td>
<td>0.90</td>
<td>0.90</td>
</tr>
</tbody>
</table>

**Most similar to awesome - what does our best performing model say?**

In [29]:
dbow = models[1]
dbow.most_similar('awesome', topn=10)

Out[29]:
[(u'juon', 0.3789939880371094),
 (u'a-pix', 0.3781469762325287),
 (u"rosemary's", 0.37472963333129883),
 (u'schnook', 0.3683214783668518),
 (u"luise's", 0.366854190826416),
 (u'chrysalis', 0.3642809689049524),
 (u'f*', 0.362865686416626),
 (u'decadent', 0.3604990839958191),
 (u'surrogacy', 0.35499149560928345),
 (u"second", 0.35283005237579346)]

Distributed bag of words doesn’t train word vectors ;-)
In [30]:

dm_m = models[2]
dm_m.most_similar('awesome', topn=10)

Out[30]:
[(u'amazing', 0.9163687229156494),
 (u'incredible', 0.901116027832031),
 (u'excellent', 0.886062644424438),
 (u'outstanding', 0.8797732591629028),
 (u'exceptional', 0.8539372682571411),
 (u'awful', 0.8104138970375061),
 (u'astounding', 0.7750493884086609),
 (u'alright', 0.7587056159973145),
 (u'astonishing', 0.756235790252686),
 (u'extraordinary', 0.743841290473938)]

Most similar to awful - distributed memory model (dm/m)

In [31]:

dm_m.most_similar('awful', topn=10)

Out[31]:
[(u'abysmal', 0.8371909856796265),
 (u'appalling', 0.8327066898345947),
 (u'atrocious', 0.8309577703475952),
 (u'horrible', 0.8192445039749146),
 (u'terrible', 0.8124841451644897),
 (u'awful', 0.8104138970375061),
 (u'dreadful', 0.8072893023490906),
 (u'horrendous', 0.7981990575790405),
 (u'amazing', 0.7926105260848999),
 (u'incredible', 0.7852109670639038)]

In [32]:

dm_m.most_similar(positive=['awesome'], negative=['awful'])

Out[32]:
[(u'super', 0.46073806285858154),
 (u"tartakovsky's", 0.3861837387084961),
 (u'nail-bitingly', 0.3633382320404053),
 (u'actionpacked', 0.36290568113327026),
 (u'cassella', 0.3589825034145405),
 (u'outsmarts', 0.3545451760292053),
 (u'nos', 0.35315001010894775),
 (u'takeuchi', 0.3525207042694092),
 (u'keaton/burton', 0.3479143083095505),
 (u'sarinana', 0.3473117535087585)]

Could go on forever with exploration but ...

Conclusion?
Stay tuned!

New developments in this area at least every year:

- 2013: word2vec (Google)
- 2014: doc2vec (Google)
- August 2016: fastText (Facebook)

2017: ???

to be continued ... thank you!!!