Sentiment Analysis of Movie Reviews

by Adit Rada

Suppose, you want to watch a movie a friend suggested in the weekend. You go to IMDb website and look at the review of the movie. The problem is that after a long day, you don't have the patience to read a legnthy review! Would it not be nice to just know if the review is positive or negative without reading the whole thing? ¶

We will try build a model that when we input a movie review, we can get an answer if it is a positive review or a negative review.

We will use a dataset of movie reviews from the IMDb website collected by Andrew Maas. This dataset contains the text of the reviews, together with a label that indicates whether a review is “positive” or “negative.” This is a two-class classification dataset where reviews with a score of 6 or higher are labeled as positive, and the rest as negative.

The dataset is provided as text files in two separate folders, one for the training data and one for the test data. Each of these in turn has two subfolders, one called pos and one called neg. The pos folder contains all the positive reviews, each as a separate text file, and similarly for the neg folder.

We will use the helper function in scikit-learn to load files stored in such a folder structure, called load_files. We apply the load_files function first to the training data:

In [1]:

```
import numpy as np
```

In [2]:

```
from sklearn.datasets import load_files

reviews_train = load_files("aclImdb/train/")
# Load_files returns a bunch, containing training texts and training labels
text_train, y_train = reviews_train.data, reviews_train.target

print("type of text_train: {}".format(type(text_train)))
print("length of text_train: {}".format(len(text_train)))
```

```
type of text_train: <class 'list'>
length of text_train: 25000
```

In [3]:

```
print("text_train[1]:\n{}".format(text_train[1]))
```

```
text_train[1]:
b'Words can\'t describe how bad this movie is. I can\'t explain it by writ
ing only. You have too see it for yourself to get at grip of how horrible
a movie really can be. Not that I recommend you to do that. There are so m
any clich\xc3\xa9s, mistakes (and all other negative things you can imagin
e) here that will just make you cry. To start with the technical first, th
ere are a LOT of mistakes regarding the airplane. I won\'t list them here,
but just mention the coloring of the plane. They didn\'t even manage to sh
ow an airliner in the colors of a fictional airline, but instead used a 74
7 painted in the original Boeing livery. Very bad. The plot is stupid and
has been done many times before, only much, much better. There are so many
ridiculous moments here that i lost count of it really early. Also, I was
on the bad guys\' side all the time in the movie, because the good guys we
re so stupid. "Executive Decision" should without a doubt be you\'re choic
e over this one, even the "Turbulence"-movies are better. In fact, every o
ther movie in the world is better than this one.'
```

This is one movie review as an example.

text_train is a list of length 25,000, where each entry is a string containing a review. We printed the review with index 1.

We can also see that the review contains some HTML line breaks (<b.r /.>); it is better to clean the data and remove this formatting before we proceed:

In [4]:

```
text_train = [doc.replace(b"<br />", b" ") for doc in text_train]
```

The dataset was collected such that the positive class and the negative class balanced, so that there are as many positive as negative strings:

In [5]:

```
print("Samples per class (training): {}".format(np.bincount(y_train)))
```

```
Samples per class (training): [12500 12500]
```

We load the test dataset in the same manner:

In [6]:

```
reviews_test = load_files("aclImdb/test/")
text_test, y_test = reviews_test.data, reviews_test.target

print("Number of documents in test data: {}".format(len(text_test)))
print("Samples per class (test): {}".format(np.bincount(y_test)))

text_test = [doc.replace(b"<br />", b" ") for doc in text_test]
```

```
Number of documents in test data: 25000
Samples per class (test): [12500 12500]
```

GOAL: The task we want to solve is as follows: given a review, we want to assign the label “positive” or “negative” based on the text content of the review.

This is a standard binary classification task. However, the text data is not in a format that a learning algorithim can handle. We need to convert the string representation of the text into a numeric representation that we can apply our learning algorithms to.

Representing Text Data as a Bag of Words

In this representaiton, we only count how often each word appears in each text in the corpus.

There are 3 steps to this proocess:

1) Tokenization. Split each document into the words that appear in it (called tokens)

2) Vocabulary building. Collect a vocabulary of all words that appear in any of the documents, and number them

3) Encoding. For each document, count how often each of the words in the vocabulary appear in this document.

The bag-of-words representation is implemented in CountVectorizer in scikit-learn, which is a transformer.

The bag-of-words representation is stored in a SciPy sparse matrix that only stores the entries that are nonzero

In [7]:

```
from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer()
```

In [9]:

```
vect1 = CountVectorizer().fit(text_train)
X_train1 = vect1.transform(text_train)

print("X_train:\n{}".format(repr(X_train1)))
```

```
X_train:
<25000x74849 sparse matrix of type '<class 'numpy.int64'>'
        with 3431196 stored elements in Compressed Sparse Row format>
```

The shape of X_train, the bag-of-words representation of the training data, is 25,000×74,849, indicating that the vocabulary contains 74,849 entries

Let's look at the vocabulary in a bit more detail. To access the vocabulary, we use the get_feature_name method of the vectorizer, which returns a convenient list where each entry corresponds to one feature:

In [10]:

```
feature_names = vect1.get_feature_names()

print("Number of features: {}".format(len(feature_names)))
print("--"*60)
print("First 20 features:\n{}".format(feature_names[:20]))
print("--"*60)
print("Features 20010 to 20030:\n{}".format(feature_names[20010:20030]))
print("--"*60)
print("Every 2000th feature:\n{}".format(feature_names[::2000]))
```

```
Number of features: 74849
------------------------------------------------------------------------
------------------------------------------
First 20 features:
['00', '000', '0000000000001', '00001', '00015', '000s', '001', '003830',
'006', '007', '0079', '0080', '0083', '0093638', '00am', '00pm', '00s', '0
1', '01pm', '02']
------------------------------------------------------------------------
------------------------------------------
Features 20010 to 20030:
['dratted', 'draub', 'draught', 'draughts', 'draughtswoman', 'draw', 'draw
back', 'drawbacks', 'drawer', 'drawers', 'drawing', 'drawings', 'drawl',
'drawled', 'drawling', 'drawn', 'draws', 'draza', 'dre', 'drea']
------------------------------------------------------------------------
------------------------------------------
Every 2000th feature:
['00', 'aesir', 'aquarian', 'barking', 'blustering', 'bête', 'chicanery',
'condensing', 'cunning', 'detox', 'draper', 'enshrined', 'favorit', 'freez
er', 'goldman', 'hasan', 'huitieme', 'intelligible', 'kantrowitz', 'lawfu
l', 'maars', 'megalunged', 'mostey', 'norrland', 'padilla', 'pincher', 'pr
omisingly', 'receptionist', 'rivals', 'schnaas', 'shunning', 'sparse', 'su
bset', 'temptations', 'treatises', 'unproven', 'walkman', 'xylophonist']
```

Looking in the vocabulary, we find a collection of English words starting with "dra". You might notice that for "draught", "drawback", and "drawer" both the singular and plural forms are contained in the vocabulary as distinct words. These words have very closely related semantic meanings, and counting them as different words, corresponding to different features, might not be ideal.

Before we try to improve our feature extraction, let's obtain a quantitative measure of performance by actually building a classifier. We have the training labels stored in y_train and the bag-of-words representation of the training data in X_train, so we can train a classifier on this data.

For high-dimensional, sparse data like this, linear models like LogisticRegression often work best.

Model 1 - (trial using original data); Test score - 88%

In [11]:

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression

scores = cross_val_score(LogisticRegression(max_iter=10000), X_train1, y_train, cv=5)

print("Mean cross-validation accuracy: {:.2f}".format(np.mean(scores)))
```

```
Mean cross-validation accuracy: 0.88
```

We know that LogisticRegression has a regularization hyperparameter, C, which we can tune via cross-validation:

In [12]:

```
from sklearn.model_selection import GridSearchCV

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}    # Hyperparameter values to search over

grid1 = GridSearchCV(LogisticRegression(max_iter=10000), param_grid, cv=5)
grid1.fit(X_train1, y_train)

print("Best cross-validation score: {:.2f}".format(grid1.best_score_))
print("Best parameters: ", grid1.best_params_)
```

```
Best cross-validation score: 0.89
Best parameters:  {'C': 0.1}
```

We obtain a cross-validation score of 89% using C=0.1. We can now assess the generalization performance of this parameter setting on the test set:

In [13]:

```
X_test1 = vect1.transform(text_test)
print("{:.2f}".format(grid1.score(X_test1, y_test)))
```

```
0.88
```

The generalization score of model 1 is 88%

This trial model actually does quite well! Let us see if we can do any better with feature extraction.

Model 2 - (after feature extraction of setting min_df); Test score - 88%

The CountVectorizer extracts tokens using a regular expression. By default, the regular expression that is used is "\b\w\w+\b". This means it finds all sequences of characters that consist of at least two letters or numbers (\w) and that are separated by word boundaries (\b). It does not find single-letter words, and it splits up contractions like “doesn’t” or “bit.ly”, but it matches “h8ter” as a single word.

The CountVectorizer also converts all words to lowercase characters, so that “soon”, “Soon”, and “sOon” all correspond to the same token

One way toreduce the massive number of features is to only use tokens that appear in at least two documents or any other number. A token that appears only in a single document is unlikely to appear in the test set and is therefore not helpful.

We can set the minimum number of documents a token needs to appear in with the min_df parameter. Let us set it to 5 movie reviews.

In [14]:

```
vect2 = CountVectorizer(min_df=5).fit(text_train)
X_train2 = vect2.transform(text_train)

print("X_train with min_df: {}".format(repr(X_train2)))
```

```
X_train with min_df: <25000x27271 sparse matrix of type '<class 'numpy.int
64'>'
        with 3354014 stored elements in Compressed Sparse Row format>
```

By requiring at least five appearances of each token, we can bring down the number of features to 27,271 (all the way from 74849!) Almost 2/3 of the features are removed. This will make the model drastically faster.

Another advantage of doing this is that any mis-spellings are removed, as it is very unlikely that the same word will be incorrectly spellled by 5 different and independent reviewers.

Let us see if the model perfomance is still similar after removing almost 2/3 of the features.

In [15]:

```
grid2 = GridSearchCV(LogisticRegression(max_iter=10000), param_grid, cv=5)
grid2.fit(X_train2, y_train)

print("Best cross-validation score: {:.2f}".format(grid2.best_score_))
```

```
Best cross-validation score: 0.89
```

The best validation score is still 89% with so many less features and the time taken to train the model was almost twice as quicker.

In [16]:

```
X_test2 = vect2.transform(text_test)
print("{:.2f}".format(grid2.score(X_test2, y_test)))
```

```
0.88
```

The generalization score of model 2 is 88%

Note : If the transform method of CountVectorizer is called on a document that contains words that were not contained in the training data, these words will be ignored as they are not part of the dictionary.

StopWords

Another way that we can get rid of uninformative words is by discarding words that are too frequent to be informative.

There are two main approaches: using a languagespecific list of stopwords, or discarding words that appear too frequently. scikitlearn has a built-in list of English stopwords in the feature_extraction.text module:

In [17]:

```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS

print("Number of stop words: {}".format(len(ENGLISH_STOP_WORDS)))
print("Every 10th stopword:\n{}".format(list(ENGLISH_STOP_WORDS)[::10]))
```

```
Number of stop words: 318
Every 10th stopword:
['take', 'into', 'someone', 'but', 'your', 'ten', 'yourselves', 'interes
t', 'top', 'during', 'you', 'him', 'neither', 'up', 'them', 'around', 'acr
oss', 'amongst', 'part', 'this', 'yourself', 'cry', 'else', 'me', 'afterwa
rds', 'elsewhere', 'move', 'thin', 'please', 'against', 'bill', 'why']
```

Since there are only 318 words, it might not make too big of a difference. Let us try it anyway.

Model 3 - (using Stopwords); Test score - 87%

In [18]:

```
# Specifying stop_words="english" uses the built-in list.
# We could also augment it and pass our own.
vect3 = CountVectorizer(min_df=5, stop_words="english").fit(text_train)
X_train3 = vect3.transform(text_train)

print("X_train with stop words:\n{}".format(repr(X_train3)))
```

```
X_train with stop words:
<25000x26966 sparse matrix of type '<class 'numpy.int64'>'
        with 2149958 stored elements in Compressed Sparse Row format>
```

There are now 305 (27,271–26,966) fewer features in the dataset, which means that most, but not all, of the stopwords appeared. Let's run the grid search again:

In [19]:

```
grid3 = GridSearchCV(LogisticRegression(max_iter=10000), param_grid, cv=5)
grid3.fit(X_train3, y_train)

print("Best cross-validation score: {:.2f}".format(grid3.best_score_))
```

```
Best cross-validation score: 0.88
```

The grid search perfomance did decrease, but only slightly. Since only 305 features were removed, the model perfomance did not increase too. So removing stopwords many not be worth the hassle.

In [20]:

```
X_test3 = vect3.transform(text_test)
print("{:.2f}".format(grid3.score(X_test3, y_test)))
```

```
0.87
```

The generalization score of model 3 is 87%

Rescaling Data with tf–idf

Instead of dropping features that seem to be important, another approach is to give weighting the features.

A common way to do this is using the term frequency–inverse document frequency (tf–idf) method.

The idea of this method is to give high weight to any term that appears often in a particular document, but not in many documents in the corpus. If a word appears often in a particular document, but not in very many documents, it is likely to be very descriptive of the content of that document.

The tf–idf value increases proportionally to the number of times a word appears in the document and is offset by the number of documents in the corpus that contain the word, which helps to adjust for the fact that some words appear more frequently in general.

Let us see what the intuition between the individual terms: Suppose we have a collection of documents and we want rank them based on the query "the brown dog"

Term Frequency

First we collect all documnets where those three words appear by counting the number of times each word appears in the document. This is the term frequency; the higher the count, the more relevant the documnet might be.

Inverse document frequency

Because the term "the" is so common, term frequency will tend to incorrectly emphasize documents which happen to use the word "the" more frequently, without giving enough weight to the more meaningful terms "brown" and "dog".

Hence an inverse document frequency factor is incorporated which diminishes the weight of terms that occur very frequently in the document set as a whole and increases the weight of terms that occur rarely in all the documnets.

So, we can think about it like this: if the word "dog" for example has a high tf-idf for a particulat document, this means that the word appears frequently inside that document and very less frequently in other doucumnets. So this document has to have a high ranking in the search results.

How will this help our classification task?

The simple way to think how this will help us is that if a particular word e.g "Avengers" has a high tf-idf score and say it has a positive review label in the test set. Then, if a similar word with has a similarly high tf-idf score is found in the during prediction, then maybe, that review is positive.

scikit-learn implements the tf–idf method in two classes: TfidfTransformer, which takes in the sparse matrix output produced by CountVectorizer and transforms it, and TfidfVectorizer, which takes in the text data and does both the bag-of-words feature extraction and the tf–idf transformation.

Because tf–idf actually makes use of the statistical properties of the training data, we will use a pipeline:

Model 4 - (using tf-idf); Test score - 89%

In [21]:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline

# Make the pipeline
pipe4 = make_pipeline(TfidfVectorizer(min_df=5, norm=None),LogisticRegression(max_iter=
10000))

# Define the hyperparameter grid to search over
param_grid = {'logisticregression__C': [0.001, 0.01, 0.1, 1, 10]}

# Do the grid search and find the best model
grid4 = GridSearchCV(pipe4, param_grid, cv=5, n_jobs=-1)
grid4.fit(text_train, y_train)

print("Best cross-validation score: {:.2f}".format(grid4.best_score_))
```

```
Best cross-validation score: 0.89
```

We can also inspect which words tf–idf found most important. Keep in mind that the tf–idf scaling is meant to find words that distinguish documents, but it is a purely unsupervised technique. So, “important” here does not necessarily relate to the “positive review” and “negative review” labels we are interested in.

In [22]:

```
# First, we extract the TfidfVectorizer from the pipeline:
vectorizer4 = grid4.best_estimator_.named_steps["tfidfvectorizer"]

# Transform the training dataset
X_train4 = vectorizer4.transform(text_train)

# Find maximum value for each of the features over the dataset
max_value = X_train4.max(axis=0).toarray().ravel()
sorted_by_tfidf = max_value.argsort()

# Get feature names
feature_names = np.array(vectorizer4.get_feature_names())

print("Features with lowest tfidf:\n{}".format(feature_names[sorted_by_tfidf[:20]]))
print("Features with highest tfidf: \n{}".format(feature_names[sorted_by_tfidf[-20:]]))
```

```
Features with lowest tfidf:
['poignant' 'disagree' 'instantly' 'importantly' 'lacked' 'occurred'
 'currently' 'altogether' 'nearby' 'undoubtedly' 'directs' 'fond'
 'stinker' 'avoided' 'emphasis' 'commented' 'disappoint' 'realizing'
 'downhill' 'inane']
Features with highest tfidf:
['coop' 'homer' 'dillinger' 'hackenstein' 'gadget' 'taker' 'macarthur'
 'vargas' 'jesse' 'basket' 'dominick' 'the' 'victor' 'bridget' 'victoria'
 'khouri' 'zizek' 'rob' 'timon' 'titanic']
```

Features with low tf–idf are those that either are very commonly used across documents or are only used sparingly, and only in very long documents. Features with high tf-idf are those that appear frequently in a particular document and less frequently accross the whole set of documents.

We can also find the words that have low inverse document frequency—that is, those that appear frequently and are therefore deemed less important. The inverse document frequency values found on the training set are stored in the idf_ attribute:

In [23]:

```
sorted_by_idf = np.argsort(vectorizer4.idf_)
print("Features with lowest idf:\n{}".format(feature_names[sorted_by_idf[:100]]))
```

```
Features with lowest idf:
['the' 'and' 'of' 'to' 'this' 'is' 'it' 'in' 'that' 'but' 'for' 'with'
 'was' 'as' 'on' 'movie' 'not' 'have' 'one' 'be' 'film' 'are' 'you' 'all'
 'at' 'an' 'by' 'so' 'from' 'like' 'who' 'they' 'there' 'if' 'his' 'out'
 'just' 'about' 'he' 'or' 'has' 'what' 'some' 'good' 'can' 'more' 'when'
 'time' 'up' 'very' 'even' 'only' 'no' 'would' 'my' 'see' 'really' 'story'
 'which' 'well' 'had' 'me' 'than' 'much' 'their' 'get' 'were' 'other'
 'been' 'do' 'most' 'don' 'her' 'also' 'into' 'first' 'made' 'how' 'great'
 'because' 'will' 'people' 'make' 'way' 'could' 'we' 'bad' 'after' 'any'
 'too' 'then' 'them' 'she' 'watch' 'think' 'acting' 'movies' 'seen' 'its'
 'him']
```

As expected, these are mostly English stopwords like "the" and "no". But some are clearly domain-specific to the movie reviews, like "movie", "film", "time", "story", and so on. Interestingly, "good", "great", and "bad" are also among the most frequent and therefore “least relevant” words according to the tf–idf measure, even though we might expect these to be very important for our sentiment analysis task, the learning algorithim does not make use of them.

In [24]:

```
logreg4 = grid4.best_estimator_.named_steps["logisticregression"]

X_test4 = vectorizer4.transform(text_test)
print("{:.2f}".format(logreg4.score(X_test4, y_test)))
```

```
0.89
```

The generalization score of model 4 is 89%

Bag-of-Words with n-Grams

One of the main disadvantages of using a bag-of-words representation is that word order is completely discarded and context is lost.

Therefore, the two strings “it’s bad, not good at all” and “it’s good, not bad at all” have exactly the same meaning.

Fortunately, there is a way of capturing context when using a bag-of-words representation, by not only considering the counts of single tokens, but also the counts of pairs or triplets of tokens that appear next to each other. Pairs of tokens are known as bigrams, triplets of tokens are known as trigrams, and more generally sequences of tokens are known as n-grams

We can change the range of tokens that are considered as features by changing the ngram_range hyperparameter of CountVectorizer or TfidfVectorizer.

ngram_range is a tuple, consisting of the minimum length and the maximum length of the sequences of tokens that are considered.

The default is to consider only single tokens i.e. unigrams so the default is (1,1) If we pass (2,2), then only bigrams are considered. If we pass (1,3), then unigrams, bigrams and trigrams are considered.

Let us try this for our dataset and see if there is any improvement in perfomance. We will use grid search to see if adding bigrams or trigrams makes any difference:

Model 5 - (using bigrams); Test score - 90%

In [25]:

```
pipe5 = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression(max_iter=10000))

param_grid = {"logisticregression__C": [0.01, 0.1, 1, 10, 100],
              "tfidfvectorizer__ngram_range": [(1, 1), (1, 2)]}

grid5 = GridSearchCV(pipe5, param_grid, cv=5, n_jobs=-1)
grid5.fit(text_train, y_train)

print("Best cross-validation score: {:.2f}".format(grid5.best_score_))
print("Best parameters:\n{}".format(grid5.best_params_))
```

```
Best cross-validation score: 0.91
Best parameters:
{'logisticregression__C': 100, 'tfidfvectorizer__ngram_range': (1, 2)}
```

As we can see, we have crossed the coveted 90% mark by adding bigrams, though at the cost of using more computing power and time. There are now almost 155 thousand features.

The improvement in perfomance might have come due to inclusion of words like "definitely worth", "well worth", "so good" and so on. These provide more context and give much more information about the label then if they were considered as unigrams.

Final Model - Conclusion

Comparing all the models we have built with small updates and improvements to each, we can see that using the bigrams model is the best when we get the result from GridSearchCV

Let us see how it generalizes by testing it on the test set.

In [26]:

```
# First, we extract the TfidfVectorizer from the pipeline:
vectorizer5 = grid5.best_estimator_.named_steps["tfidfvectorizer"]

# Transform the test dataset
X_test5 = vectorizer5.transform(text_test, y_test)

# Retirve the logistic regression model from the pipeline
logreg5 = grid5.best_estimator_.named_steps["logisticregression"]

# Perform the test to check generalization
print("{:.2f}".format(logreg5.score(X_test5, y_test)))
```

```
C:\Users\rada_\anaconda3\lib\site-packages\sklearn\feature_extraction\tex
t.py:1874: FutureWarning: elementwise comparison failed; returning scalar
instead, but in the future will perform elementwise comparison
  if copy != "deprecated":
C:\Users\rada_\anaconda3\lib\site-packages\sklearn\feature_extraction\tex
t.py:1878: FutureWarning: 'copy' param is unused and has been deprecated s
ince version 0.22. Backward compatibility for 'copy' will be removed in 0.
24.
  warnings.warn(msg, FutureWarning)

0.90
```

The generalization score of model 5 is 90%. We will use this as the final model for predictions.

Using the model

Let us go back to the motivation behind the model that started this. Remeber the weekend situation? Where you don't want to read the whole long review but just want to see in it is positive or negative.

Let us see how thim model works on an Avatar review (my favourite movie!) taken randomly from the internet.

It is pretty long and all we want to see is if it is positive or negative.

In [27]:

```
avatar_review = [b"I saw this epic last night at the Empire Leicester Sq in London, whi
ch is a superb venue in which to view this film. Huge screen, excellent sound and an ex
traordinary Dolby, 3 dimensional image. The whole effect is mind blowing. This is a Mus
t see movie, innovative, and extraordinary. I think it will be regarded by most cinema
 goers as another milestone in the history of the art. The level of realism achieved is
remarkable, and although the film is relatively long in real time, it retains it's exci
tement and holds the audience's attention to the end. Performances are good, but this i
s not the sort of film that dwells on big star value for the actors, although Sigorney
 Weaver does shine and delivers a very convincing performance, as do the rest of the ca
st. But as there is so much entertainment and action value on screen the human element
 does not dominate in the usual way. As Writer/Director, James Cameron deserves high pr
aise for this creation and in my opinion it will break box office records. I thoroughly
enjoyed this film."]
```

In [28]:

```
# We transform the text review so that the learning algorithim can use it for predictio
n
avatar_test = vectorizer5.transform(avatar_review)

# We now predict the outcome
logreg5.predict(avatar_test)
```

Out[28]:

```
array([1])
```

1 is positive; 0 is negative. This is again a binary classification problem remember.

Horray!!! The model did predicts that this review is positive, we have now saved all that time we would have spent reading a long review.

Citiation:

The dataset can be found at https://ai.stanford.edu/~amaas/data/sentiment/ (https://ai.stanford.edu/~amaas/data/sentiment/)

@InProceedings{maas-EtAl:2011:ACL-HLT2011, author = {Maas, Andrew L. and Daly, Raymond E. and Pham, Peter T. and Huang, Dan and Ng, Andrew Y. and Potts, Christopher}, title = {Learning Word Vectors for Sentiment Analysis}, booktitle = {Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies}, month = {June}, year = {2011}, address = {Portland, Oregon, USA}, publisher = {Association for Computational Linguistics}, pages = {142--150}, url = {http://www.aclweb.org/anthology/P11-1015} (http://www.aclweb.org/anthology/P11-1015}) }