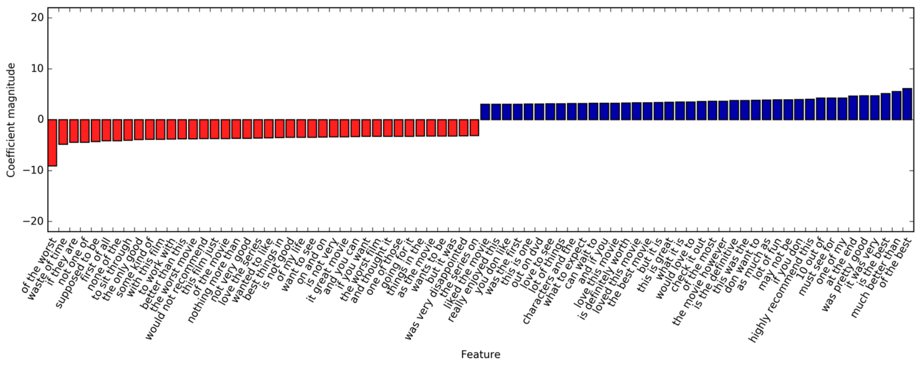

Figure 7-5. Visualization of only the important trigram features of the model

Advanced Tokenization, Stemming, and Lemmatization

As mentioned previously, the feature extraction in the `CountVectorizer` and `Tfidf Vectorizer` is relatively simple, and much more elaborate methods are possible. One particular step that is often improved in more sophisticated text-processing applications is the first step in the bag-of-words model: tokenization. This step defines what constitutes a word for the purpose of feature extraction.

We saw earlier that the vocabulary often contains singular and plural versions of some words, as in `"drawback"` and `"drawbacks"`, `"drawer"` and `"drawers"`, and `"drawing"` and `"drawings"`. For the purposes of a bag-of-words model, the semantics of `"drawback"` and `"drawbacks"` are so close that distinguishing them will only increase overfitting, and not allow the model to fully exploit the training data. Similarly, we found the vocabulary includes words like `"replace"`, `"replaced"`, `"replace ment"`, `"replaces"`, and `"replacing"`, which are different verb forms and a noun relating to the verb "to replace." Similarly to having singular and plural forms of a noun, treating different verb forms and related words as distinct tokens is disadvantageous for building a model that generalizes well.

This problem can be overcome by representing each word using its *word stem*, which involves identifying (or *conflating*) all the words that have the same word stem. If this is done by using a rule-based heuristic, like dropping common suffixes, it is usually referred to as *stemming*. If instead a dictionary of known word forms is used (an explicit and human-verified system), and the role of the word in the sentence is taken into account, the process is referred to as *lemmatization* and the standardized form of the word is referred to as the *lemma*. Both processing methods, lemmatization and stemming, are forms of *normalization* that try to extract some normal form of a word. Another interesting case of normalization is spelling correction, which can be helpful in practice but is outside of the scope of this book.

To get a better understanding of normalization, let's compare a method for stemming —the Porter stemmer, a widely used collection of heuristics (here imported from the nltk package)—to lemmatization as implemented in the spacy package:[8]

In[36]:

```
import spacy
import nltk

# load spacy's English-language models
en_nlp = spacy.load('en')
# instantiate nltk's Porter stemmer
stemmer = nltk.stem.PorterStemmer()

# define function to compare lemmatization in spacy with stemming in nltk
def compare_normalization(doc):
    # tokenize document in spacy
    doc_spacy = en_nlp(doc)
    # print lemmas found by spacy
    print("Lemmatization:")
    print([token.lemma_ for token in doc_spacy])
    # print tokens found by Porter stemmer
    print("Stemming:")
    print([stemmer.stem(token.norm_.lower()) for token in doc_spacy])
```

We will compare lemmatization and the Porter stemmer on a sentence designed to show some of the differences:

In[37]:

```
compare_normalization(u"Our meeting today was worse than yesterday, "
                       "I'm scared of meeting the clients tomorrow.")
```

Out[37]:

```
Lemmatization:
['our', 'meeting', 'today', 'be', 'bad', 'than', 'yesterday', ',', 'i', 'be',
 'scared', 'of', 'meet', 'the', 'client', 'tomorrow', '.']
Stemming:
['our', 'meet', 'today', 'wa', 'wors', 'than', 'yesterday', ',', 'i', "'m",
 'scare', 'of', 'meet', 'the', 'client', 'tomorrow', '.']
```

Stemming is always restricted to trimming the word to a stem, so "was" becomes "wa", while lemmatization can retrieve the correct base verb form, "be". Similarly, lemmatization can normalize "worse" to "bad", while stemming produces "wors". Another major difference is that stemming reduces both occurrences of "meeting" to "meet". Using lemmatization, the first occurrence of "meeting" is recognized as a

8 For details of the interface, consult the nltk and spacy documentation. We are more interested in the general principles here.

noun and left as is, while the second occurrence is recognized as a verb and reduced to "meet". In general, lemmatization is a much more involved process than stemming, but it usually produces better results than stemming when used for normalizing tokens for machine learning.

While scikit-learn implements neither form of normalization, CountVectorizer allows specifying your own tokenizer to convert each document into a list of tokens using the tokenizer parameter. We can use the lemmatization from spacy to create a callable that will take a string and produce a list of lemmas:

In[38]:

```
# Technicality: we want to use the regexp-based tokenizer
# that is used by CountVectorizer and only use the lemmatization
# from spacy. To this end, we replace en_nlp.tokenizer (the spacy tokenizer)
# with the regexp-based tokenization.
import re
# regexp used in CountVectorizer
regexp = re.compile('(?u)\\b\\w\\w+\\b')

# load spacy language model and save old tokenizer
en_nlp = spacy.load('en')
old_tokenizer = en_nlp.tokenizer
# replace the tokenizer with the preceding regexp
en_nlp.tokenizer = lambda string: old_tokenizer.tokens_from_list(
    regexp.findall(string))

# create a custom tokenizer using the spacy document processing pipeline
# (now using our own tokenizer)
def custom_tokenizer(document):
    doc_spacy = en_nlp(document, entity=False, parse=False)
    return [token.lemma_ for token in doc_spacy]

# define a count vectorizer with the custom tokenizer
lemma_vect = CountVectorizer(tokenizer=custom_tokenizer, min_df=5)
```

Let's transform the data and inspect the vocabulary size:

In[39]:

```
# transform text_train using CountVectorizer with lemmatization
X_train_lemma = lemma_vect.fit_transform(text_train)
print("X_train_lemma.shape: {}".format(X_train_lemma.shape))

# standard CountVectorizer for reference
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("X_train.shape: {}".format(X_train.shape))
```

Out[39]:

```
X_train_lemma.shape:  (25000, 21596)
X_train.shape:  (25000, 27271)
```

As you can see from the output, lemmatization reduced the number of features from 27,271 (with the standard `CountVectorizer` processing) to 21,596. Lemmatization can be seen as a kind of regularization, as it conflates certain features. Therefore, we expect lemmatization to improve performance most when the dataset is small. To illustrate how lemmatization can help, we will use `StratifiedShuffleSplit` for cross-validation, using only 1% of the data as training data and the rest as test data:

In[40]:

```
# build a grid search using only 1% of the data as the training set
from sklearn.model_selection import StratifiedShuffleSplit

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
cv = StratifiedShuffleSplit(n_iter=5, test_size=0.99,
                            train_size=0.01, random_state=0)
grid = GridSearchCV(LogisticRegression(), param_grid, cv=cv)
# perform grid search with standard CountVectorizer
grid.fit(X_train, y_train)
print("Best cross-validation score "
      "(standard CountVectorizer): {:.3f}".format(grid.best_score_))
# perform grid search with lemmatization
grid.fit(X_train_lemma, y_train)
print("Best cross-validation score "
      "(lemmatization): {:.3f}".format(grid.best_score_))
```

Out[40]:

```
Best cross-validation score (standard CountVectorizer): 0.721
Best cross-validation score (lemmatization): 0.731
```

In this case, lemmatization provided a modest improvement in performance. As with many of the different feature extraction techniques, the result varies depending on the dataset. Lemmatization and stemming can sometimes help in building better (or at least more compact) models, so we suggest you give these techniques a try when trying to squeeze out the last bit of performance on a particular task.

Topic Modeling and Document Clustering

One particular technique that is often applied to text data is *topic modeling*, which is an umbrella term describing the task of assigning each document to one or multiple *topics*, usually without supervision. A good example for this is news data, which might be categorized into topics like "politics," "sports," "finance," and so on. If each document is assigned a single topic, this is the task of clustering the documents, as discussed in Chapter 3. If each document can have more than one topic, the task