Text Data

Zhongjian Lin
University of Georgia

November 8, 2024

Text Data

We have talked about two kinds of features that can represent properties of the data: continuous features that describe a quantity, and categorical features that are items from a fixed list. There is a third kind of feature that can be found in many application, which is text.

- For example, if we want to classify in email into whether it is a legitimate email or spam, the content of the email will certainly contain important information for this classification task.
- Or maybe we want to learn about the opinion of a politician on the topic of immigration. Here, their speeches or tweets might provide useful information.
- In customer services, we often want to find out if a message is a complaint or an inquiry. And depending on the kind of complaint, we might be able to provide automatic advice or forward it to a specific department. These decisions can all be supported by the content of the message that was sent to customer service.

Text data is usually represented as strings, made up of characters. In any of the examples above, the length of the text of each text will be different. This feature is clearly very different from the numeric features that we discussed so far, and we need to process the text data before we can apply our machine learning algorithms to the text data.

Types of data

Text is usually just a string in your dataset, but not each string feature should be treated as text. There are four kinds of string data you might see

- Categorical data: comes from a fixed list. Say you collect data via a survey where you ask people their favorite color, with a drop-down menu that allows them to select from “red”, “green”, “blue”, “yellow”, “black”, “white”, “purple” and “pink”.
- Free strings that can be semantically mapped to categories: responses you can obtain from a text field. It will probably be best to encode this data as a categorical variable, where you can select the categories either using the most common entries, or by defining categories that will capture responses in a way that makes sense for your application
- Structured string data: Often, manually entered values do not correspond to fixed categories, but still have some underlying structure, like addresses, names of places or people, dates, telephone numbers or other identifiers.
- Text data: free form text that consists of phrases or sentences.

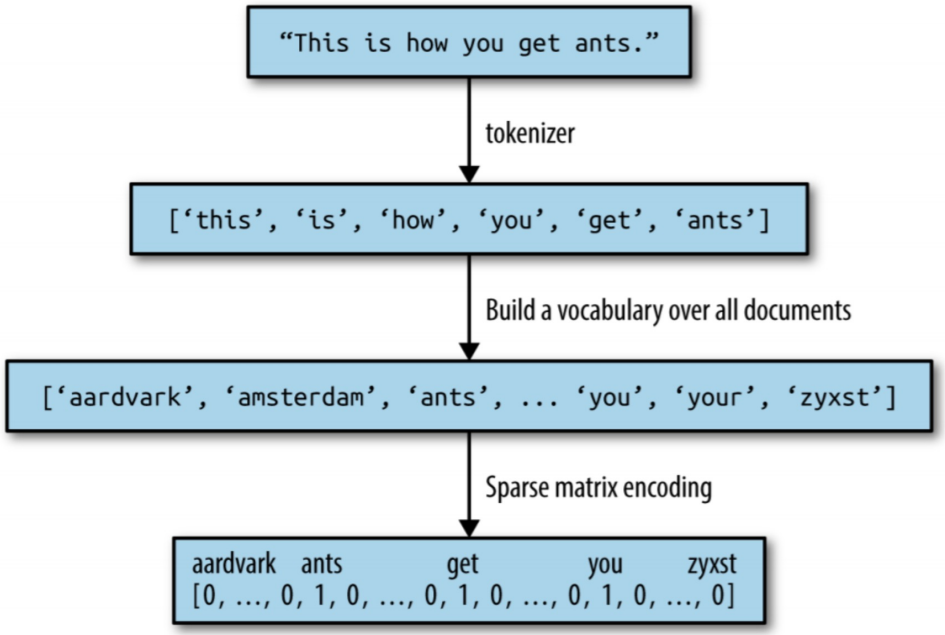
Sentiment analysis of movie reviews

The task we want to solve is given a review, we want to assign the labels "positive" and "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 machine learning model can handle. We need to convert the string representation of the text into a numeric representation that we can apply our machine learning algorithms to.
One of the most simple, but effective and commonly used ways to represent text for machine learning is using the *bag-of-words* representation.

Bag-of-Words

Computing the bag-of-word representation for a corpus of documents consists of the following three steps:

- Tokenization: Split each document into the words that appear in it (called tokens), for example by splitting them by whitespace and punctuation.
- Vocabulary building: Collect a vocabulary of all words that appear in any of the documents, and number them (say in alphabetical order).
- Encoding: For each document, count how often each of the words in the vocabulary appear in this document.



Bag-of-Words

```
bards_words =["The fool doth think he is wise,",
"but the wise man knows himself to be a fool"]
from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer()
vect.fit(bards_words)
print("Vocabulary size: {}".format(len(vect.vocabulary_)))
print("vocabulary content:\n {}".format(vect.vocabulary_))
Vocabulary size: 13
vocabulary content:
 {'the': 9, 'fool': 3, 'doth': 2, 'think': 10, 'he': 4, 'is': 6, 'wise': 12, 'but': 1,
 'man': 8, 'knows': 7, 'himself': 5, 'to': 11, 'be': 0}
 bag_of_words = vect.transform(bards_words)
print("bag_of_words: {}".format(repr(bag_of_words)))
bag_of_words: <2x13 sparse matrix of type '<class 'numpy.int64'>'
        with 16 stored elements in Compressed Sparse Row format>
print("Dense representation of bag_of_words:\n{}".format(bag_of_words.toarray()))
Dense representation of bag_of_words:
[[0 0 1 1 1 0 1 0 0 1 1 0 1]
 [1 1 0 1 0 1 0 1 1 1 0 1 1]]
```

Sentiment analysis of movie reviews

We have training and test data from the IMDb reviews into lists of strings (25,000 in each data). For both train and test dataset, we have equal observation with positive and negative reviews, i.e., 12,500 positive and 12,500 negative. Each observation has statement as well. Using Bag-of-Words, we obtain 74,849 features.

```
vect = CountVectorizer().fit(text_train)
X_train = vect.transform(text_train)
feature_names = vect.get_feature_names_out()
print("Number of features: {}".format(len(feature_names)))
# print first fifty features
print("First 20 features:\n{}".format(feature_names[:20]))
# print feature 20010 to 20030
print("Features 20010 to 20030\n{}".format(feature_names[20010:20030]))
# get every 2000th word to get an overview
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' '01' '01pm' '02']
Features 20010 to 20030
['dratted' 'draub' 'draught' 'draughts' 'draughtswoman' 'draw' 'drawback'
 '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' 'freezer'
 'goldman' 'hasan' 'huitieme' 'intelligible' 'kantrowitz' 'lawful' 'maars'
 'megalunged' 'mostey' 'norrland' 'padilla' 'pincher' 'promisingly'
 'receptionist' 'rivals' 'schnaas' 'shunning' 'sparse' 'subset'
 'temptations' 'treatises' 'unproven' 'walkman' 'xylophonist']
```

Classification with huge number of features

Before we try to improve our feature extraction, let us obtain a quantitative measure of performance by actually building a classifier. We have the training labels stored in y_train and the bag-of-word 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. Let's start by evaluating LogisticRegresssion using cross-validation

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
scores = cross_val_score(LogisticRegression(), X_train, y_train, cv=5)
np.mean(scores)
0.8819199999999998
from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: ", grid.best_score_)
print("Best parameters: ", grid.best_params_)
Best cross-validation score:  0.8890800000000001
Best parameters:  {'C': 0.1}
X_test = vect.transform(text_test)
grid.score(X_test, y_test)
0.87884
```

Sentiment analysis of movie reviews

Now, let's see if we can improve the extraction of words. The way the CountVector izer extracts tokens is using a regular expression. This simple mechanism works quite well in practice, but as we saw above, we get many uninformative features like the numbers. One way to cut back on these is to only use tokens that appear in at least 2 documents (or at least 5 documents etc). 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.

```
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print(repr(X_train))
<25000x27271 sparse matrix of type '<class 'numpy.int64'>'
        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 27,272 (see the output above), only about a third of the original features. Furthermore, there are clearly much fewer numbers, and some of the more obscure words or misspellings seem to have vanished.

Let's see how well our model performs by doing a grid-search again: *min_df* parameter.

```
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: ", grid.best_score_)
0.88764
```

The best validation accuracy of the grid-search is still 88.8%, slightly different from before. We didn't improve our model, but having less features to deal with speeds up processing and throwing away useless features might make the model more interpretable.

Stop-words

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 stop words, or discarding words that appear too frequently. Scikit-learn had a built-in list of English stop-words in the *feature_extraction.text* module.

```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
# print number of stop words
print(len(ENGLISH_STOP_WORDS))
# print some of the stop words
print(list(ENGLISH_STOP_WORDS)[::10])
318
['before', 'least', 'noone', 'is', 'he', 'between', 'seemed', 'because', 'cry', 'that', 'else', 'co', 'less',
'full', 'one', 'become', 'very', 'sixty', 'during', 'to', 'thin', 'inc', 'yourselves', 'whatever', 'go', 'me'
'anyway', 'my', 'onto', 'latterly', 'what', 'meanwhile']
```

Clearly, removing the stop-words in the list can only decrease the number of features by the length of the list, here 318, but it might lead to an improvement/difference in performance.

Rescaling the data with TFIDF

Instead of dropping features that are deemed unimportant, another approach is to rescale features by how informative we expect them to be. One of the most common ways to do this is using the term frequency–inverse document frequency (tf-idf) method. The tf-idf score for word w in document d is given by:

$$\text{tfidf}(w,d) = \text{tf} \times \log\left(\frac{N+1}{N_w+1}\right) + 1$$

where N is the number of documents in the training set, N_w is the number of documents in the training set that the word w appears in, and tf, the term frequency, is the number of times that the word w appears in the query document d. The intuition of this method is to give high weight to a 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.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(TfidfVectorizer(min_df=5, norm=None), LogisticRegression())
param_grid = {'logisticregression__C': [0.001, 0.01, 0.1, 1, 10]}
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Best cross-validation score: ", grid.best_score_)
0.8938400000000002
```

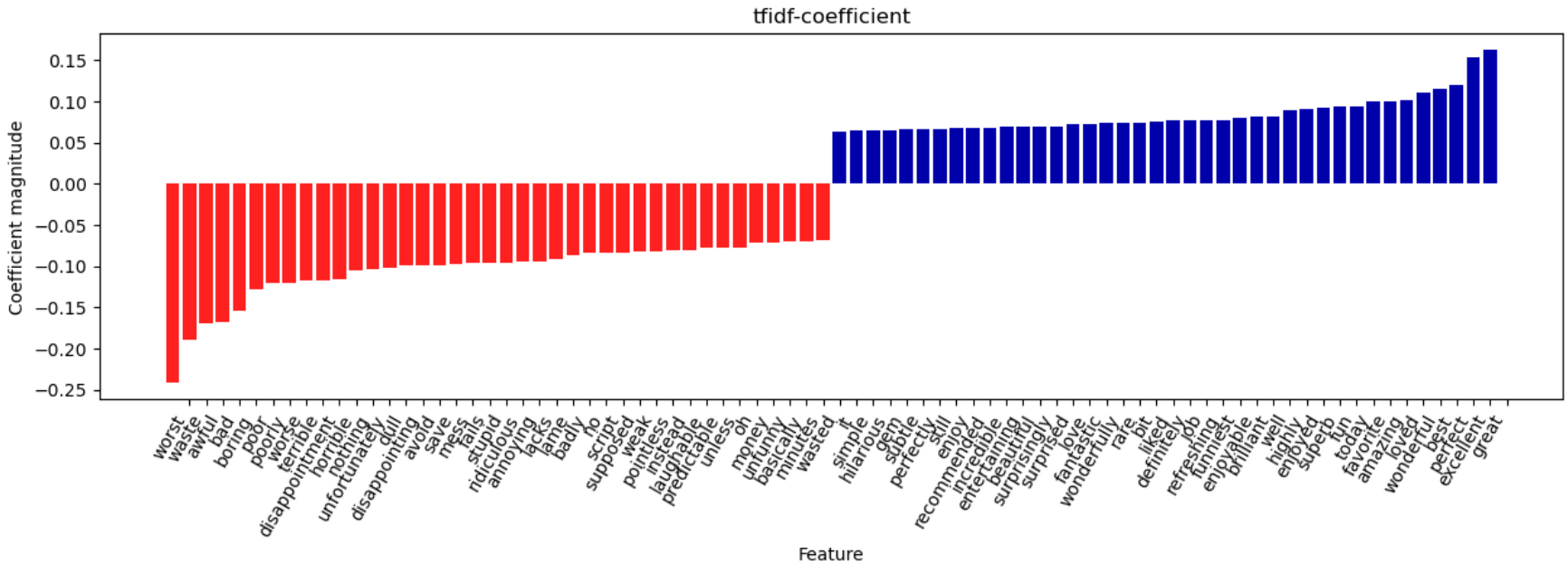
Most important words

We can 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 related to the "positive review" and "negative review" labels we are interested in.

- Features with low tf-idf are those that are either very commonly used across documents, or are only used sparingly, and only in very long documents. Interestingly, many of the high tf-idf features actually identify certain shows or movies. These terms only appear in reviews for this particular show or franchise, but tend to appear very often in these particular reviews.
- We can also find the words that have low inverse document frequency, that is those that appear frequently and are therefore deemed less important. As expected, these are mostly English stop words 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, even though we might expect these to be very important for our sentiment analysis task.

Investigating model coeicients

Finally, let us look into a bit more detail into what our logistic regression model actually learned from the data. Because there are so many features, 27.272 after removing the infrequent ones, we can clearly not look at all of the coefficients at the same time. However, we can look at the largest coefficients, and see which words these correspond to.



Largest Coefficients

The bar-chart in Figure tfidf-coefficient shows the 25 largest and 25 smallest coefficients of the logistic regression model, with the bar showing the size of each coefficient. The negative coefficients on the left belong to words that according to the model are indicative of negative reviews, while the positive coefficients on the right belong to word that according to the model indicate positive reviews. Most of the terms are quite intuitive, like “worst”, “waste”, “disappointment” and “laughable” indicating bad movie reviews, while “excellent”, “wonderful”, “enjoyable” and “refreshing” indicate positive movie reviews. Some words are slightly less clear, like “bit”, “job” and “today”, but these might be part of phrases like “good job” or “best today”.

Bag of words with more than one word (n-grams)

One of the main disadvantages of using a bag-of-word representation is that word order is completely discarded. Therefore the two strings "it's bad, not good at all" and "it's good, not bad at all" have exactly the same representation, even though the meanings are inverted. Putting "not" in front of a word is only one (if extreme) example of how context matters.
There is a way of capturing context when using a bag-of-word representation, by not only considering the counts of single tokens, but also the counts of pairs or triples 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* parameter of the *CountVecorizer* or *TfidfVectorizer*.
The *ngram_range* parameter is a tuple, consisting of the minimum length and the maximum length of the sequences of tokens that are considered.

n-grams

```
print(bards_words)
['The fool doth think he is wise,', 'but the wise man knows himself to be a fool']
cv = CountVectorizer(ngram_range=(1, 1)).fit(bards_words)
print(len(cv.vocabulary_))
print(cv.get_feature_names_out())
13
['be' 'but' 'doth' 'fool' 'he' 'himself' 'is' 'knows' 'man' 'the' 'think'
 'to' 'wise']
 cv = CountVectorizer(ngram_range=(2, 2)).fit(bards_words)
print(len(cv.vocabulary_))
print(cv.get_feature_names_out())
14
['be fool' 'but the' 'doth think' 'fool doth' 'he is' 'himself to'
 'is wise' 'knows himself' 'man knows' 'the fool' 'the wise' 'think he'
 'to be' 'wise man']
```

Using longer sequences of tokens usually results in many more features, and in more specific features. For most applications, the minimum number of tokens should be one, as single words often capture a lot of meaning. Adding bigrams helps in most cases, and adding longer sequences, up to 5-grams, might help, but will lead to an explosion of the number of features, and might lead to overfitting, as there are many very specific features.

Logistic Regression with more features

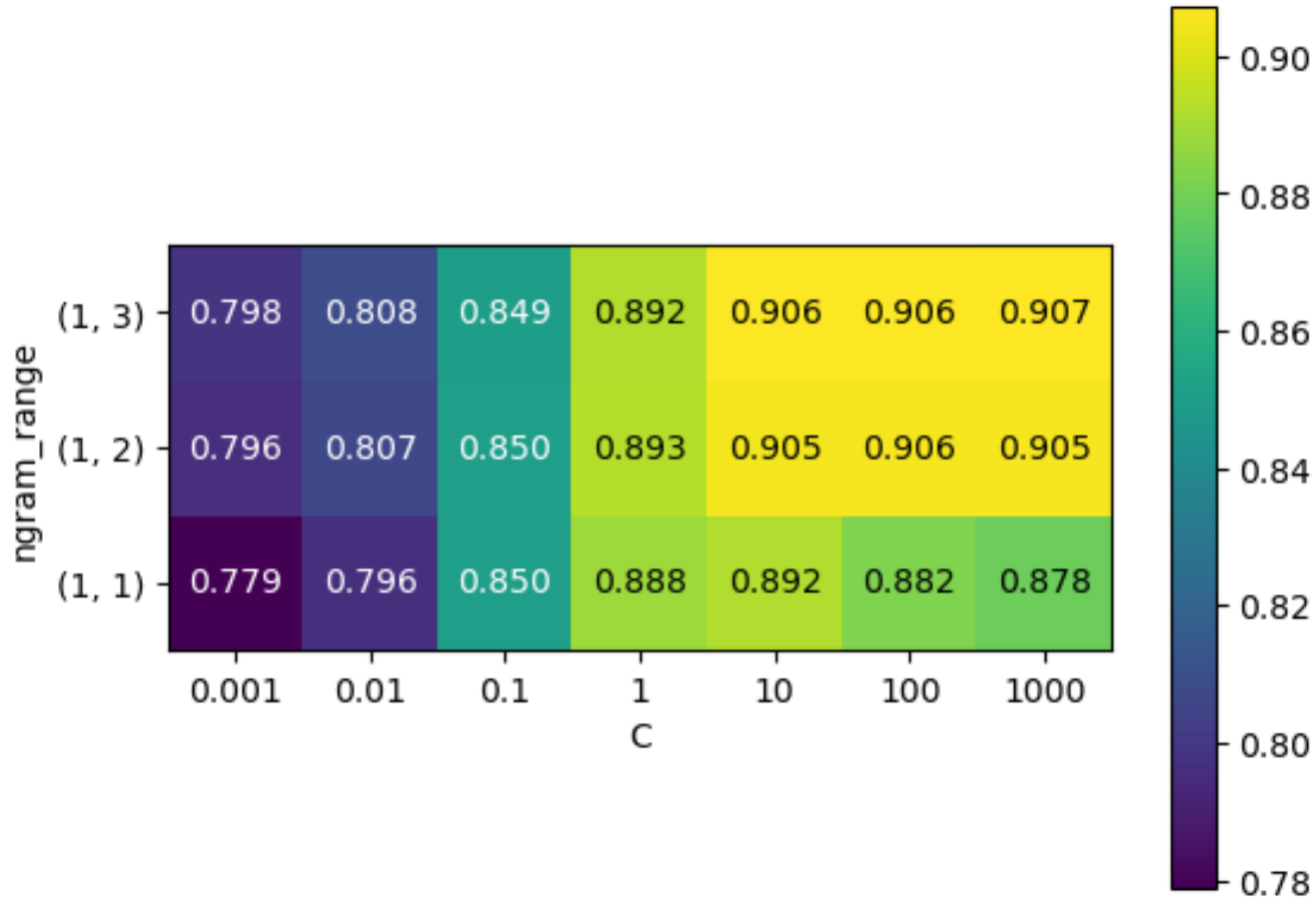
Let's use the TfidfVectorizer on the IMDb movie review data and find the best setting of n-gram range using grid-search.

```
pipe = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression())
param_grid = {'logisticregression__C': [0.001, 0.01, 0.1, 1, 10, 100, 1000],
 "tfidfvectorizer__ngram_range": [(1, 1), (1, 2), (1, 3)]}
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Best cross-validation score: ", grid.best_score_)
grid.best_params_
Best cross-validation score:  0.90708
{'logisticregression__C': 1000, 'tfidfvectorizer__ngram_range': (1, 3)}
```

As you can see from the results, we improved performance a bit more than a percent by adding bigram and trigram features.

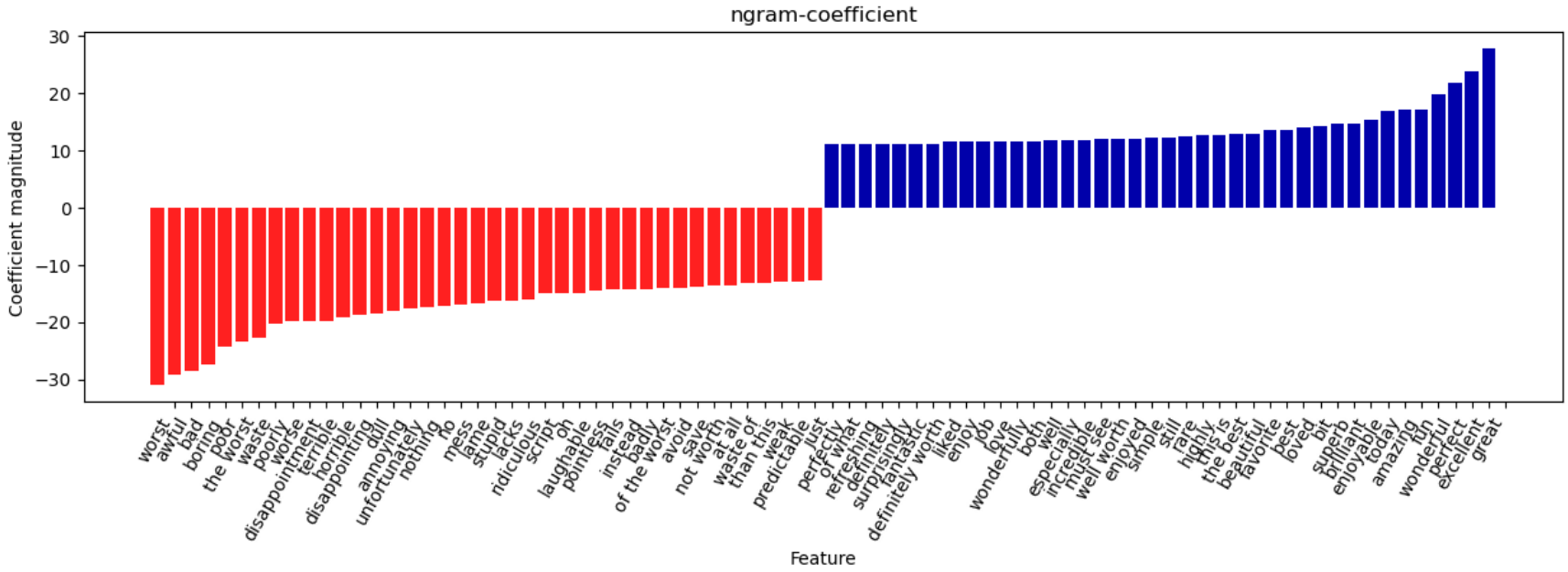
Visualization of Accuracy

We can visualize the cross-validation accuracy as a function of the ngram_range and C parameter as a heat map.



Important Features

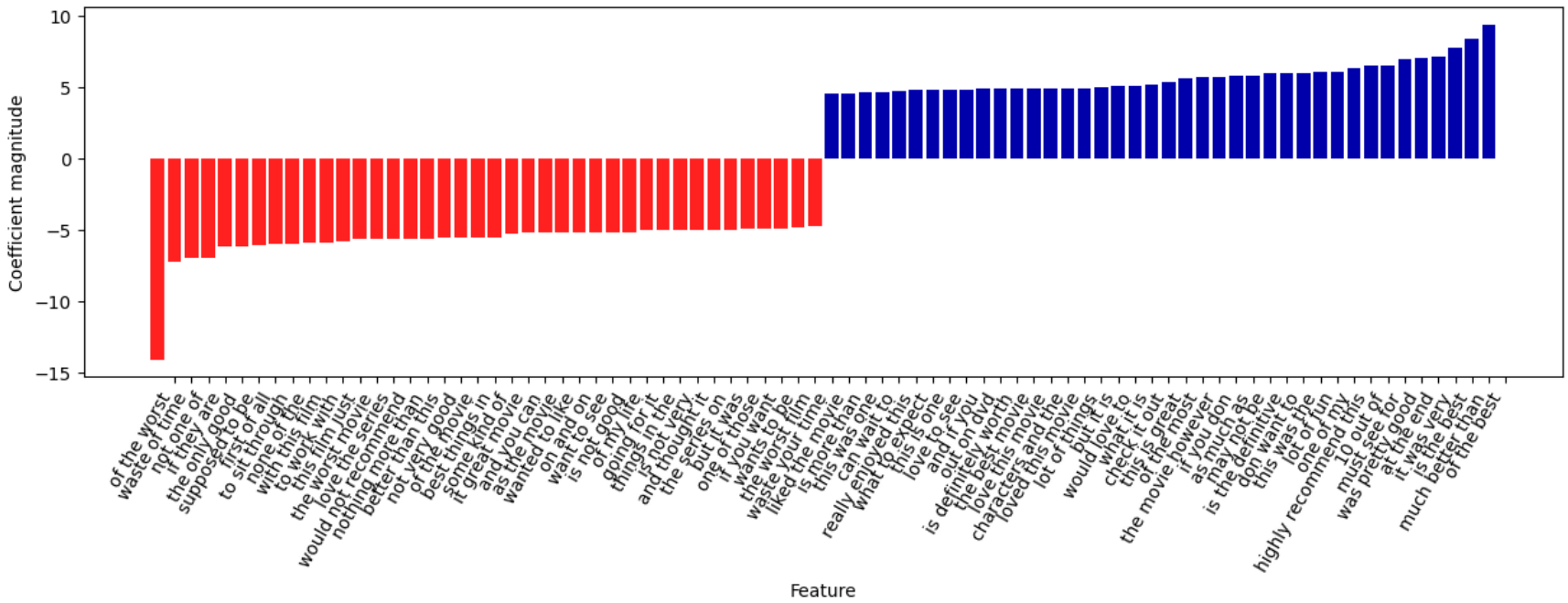
From the heat map we can see that using bigrams increases performance quite a bit, while adding three-grams only provides a very small benefit in terms of accuracy. To understand better how the model improved, we visualize the important coefficient for the best model.



There are particularly interesting features containing the word “worth” that were not present in the unigram model: “not worth” is indicative of a negative review, while “definitely wroth” and “well worth” are indicative of a positive review. This is a prime example of context influencing the meaning of the word “worth”.

Important Features

Below, we visualize only bigrams and trigrams, to provide further insight into why these features are helpful. Many of the useful bigrams and trigrams consist of common words that would not be informative on their own, as in the phrases “none of the”, “the only good”, “on and on”, “this was one of ”, “of the most” and so on. However, the impact of these features is quite limited compared to the importance of the unigram features.



Advanced tokenization, stemming and lemmatization

We saw above that the vocabulary often contains singular and plural version of words as in 'drawback', 'drawbacks', 'drawer', 'drawers', 'drawing', 'drawings'. For the purpose 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', 'replacement', 'replaces', 'replacing', which are different verb forms and a nouns relating to the verb "to replace". Similarly to having singular and plural 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, identifying all the words that have the same word stem. If this is done by using a rule-based heuristic, like dropping common suffixes, this is usually referred to as stemming. If instead a dictionary of known word forms is used (that is using an explicit and human-verified system), and the role of the word in the sentence taken into account, the process is referred to as lemmatization and the standardized form of the word is referred to as lemma.