In[26]:

```
mglearn.tools.visualize_coefficients(
    grid.best_estimator_.named_steps["logisticregression"].coef_,
    feature_names, n_top_features=40)
```

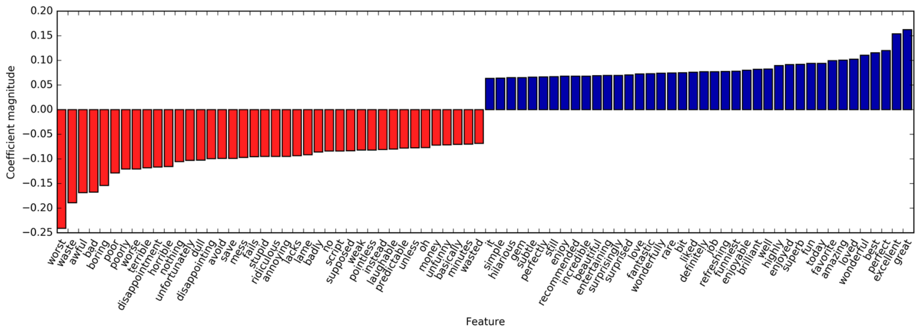

Figure 7-2. Largest and smallest coefficients of logistic regression trained on tf-idf features

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 words 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-words 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 example (if an extreme one) of how context matters. 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` parameter of `CountVectorizer` 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. Here is an example on the toy data we used earlier:

In[27]:

```
print("bards_words:\n{}".format(bards_words))
```

Out[27]:

```
bards_words:
['The fool doth think he is wise,',
 'but the wise man knows himself to be a fool']
```

The default is to create one feature per sequence of tokens that is at least one token long and at most one token long, or in other words exactly one token long (single tokens are also called *unigrams*):

In[28]:

```
cv = CountVectorizer(ngram_range=(1, 1)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

Out[28]:

```
Vocabulary size: 13
Vocabulary:
['be', 'but', 'doth', 'fool', 'he', 'himself', 'is', 'knows', 'man', 'the',
 'think', 'to', 'wise']
```

To look only at bigrams—that is, only at sequences of two tokens following each other—we can set `ngram_range` to `(2, 2)`:

In[29]:

```
cv = CountVectorizer(ngram_range=(2, 2)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

Out[29]:

```
Vocabulary size: 14
Vocabulary:
['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. There is no common bigram between the two phrases in `bard_words`:

In[30]:

```
print("Transformed data (dense):\n{}".format(cv.transform(bards_words).toarray()))
```

Out[30]:

```
Transformed data (dense):
[[0 0 1 1 1 0 1 0 0 1 0 1 0 0]
 [1 1 0 0 0 1 0 1 1 0 1 0 1 1]]
```

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. Adding longer sequences—up to 5-grams—might help too, but this will lead to an explosion of the number of features and might lead to overfitting, as there will be many very specific features. In principle, the number of bigrams could be the number of unigrams squared and the number of trigrams could be the number of unigrams to the power of three, leading to very large feature spaces. In practice, the number of higher *n*-grams that actually appear in the data is much smaller, because of the structure of the (English) language, though it is still large.

Here is what using unigrams, bigrams, and trigrams on `bards_words` looks like:

In[31]:

```
cv = CountVectorizer(ngram_range=(1, 3)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

Out[31]:

```
Vocabulary size: 39
Vocabulary:
['be', 'be fool', 'but', 'but the', 'but the wise', 'doth', 'doth think',
 'doth think he', 'fool', 'fool doth', 'fool doth think', 'he', 'he is',
 'he is wise', 'himself', 'himself to', 'himself to be', 'is', 'is wise',
 'knows', 'knows himself', 'knows himself to', 'man', 'man knows',
 'man knows himself', 'the', 'the fool', 'the fool doth', 'the wise',
 'the wise man', 'think', 'think he', 'think he is', 'to', 'to be',
 'to be fool', 'wise', 'wise man', 'wise man knows']
```

Let's try out the `TfidfVectorizer` on the IMDb movie review data and find the best setting of *n*-gram range using a grid search:

In[32]:

```
pipe = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression())
# running the grid search takes a long time because of the
# relatively large grid and the inclusion of trigrams
param_grid = {"logisticregression__C": [0.001, 0.01, 0.1, 1, 10, 100],
              "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: {:.2f}".format(grid.best_score_))
print("Best parameters:\n{}".format(grid.best_params_))
```

Out[32]:

```
Best cross-validation score: 0.91
Best parameters:
{'tfidfvectorizer__ngram_range': (1, 3), 'logisticregression__C': 100}
```

As you can see from the results, we improved performance by a bit more than a percent by adding bigram and trigram features. We can visualize the cross-validation accuracy as a function of the `ngram_range` and `C` parameter as a heat map, as we did in Chapter 5 (see Figure 7-3):

In[33]:

```
# extract scores from grid_search
scores = grid.cv_results_['mean_test_score'].reshape(-1, 3).T
# visualize heat map
heatmap = mglearn.tools.heatmap(
    scores, xlabel="C", ylabel="ngram_range", cmap="viridis", fmt="%.3f",
    xticklabels=param_grid['logisticregression__C'],
    yticklabels=param_grid['tfidfvectorizer__ngram_range'])
plt.colorbar(heatmap)
```

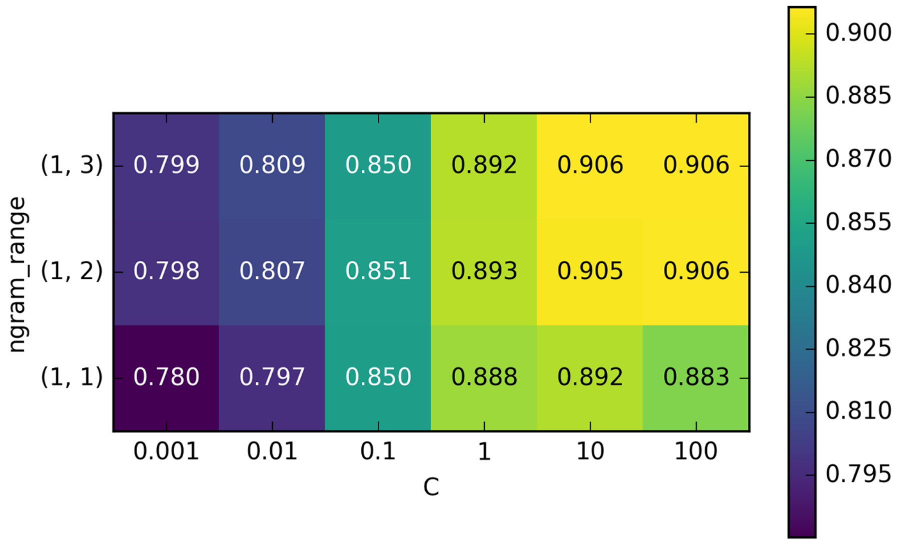

Figure 7-3. Heat map visualization of mean cross-validation accuracy as a function of the parameters ngram_range and C

From the heat map we can see that using bigrams increases performance quite a bit, while adding trigrams only provides a very small benefit in terms of accuracy. To understand better how the model improved, we can visualize the important coefficient for the best model, which includes unigrams, bigrams, and trigrams (see Figure 7-4):

In[34]:

```
# extract feature names and coefficients
vect = grid.best_estimator_.named_steps['tfidfvectorizer']
feature_names = np.array(vect.get_feature_names())
coef = grid.best_estimator_.named_steps['logisticregression'].coef_
mglearn.tools.visualize_coefficients(coef, feature_names, n_top_features=40)
```

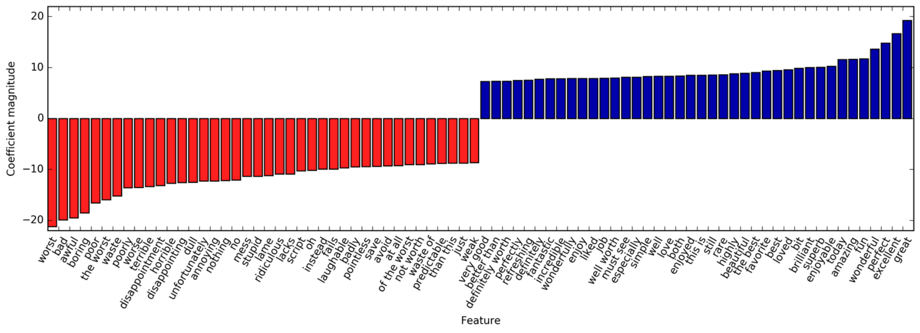

Figure 7-4. Most important features when using unigrams, bigrams, and trigrams with tf-idf rescaling

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 worth" and "well worth" are indicative of a positive review. This is a prime example of context influencing the meaning of the word “worth.”

Next, we’ll visualize only 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 is one", "of the most", and so on. However, the impact of these features is quite limited compared to the importance of the unigram features, as you can see in Figure 7-5:

In[35]:

```
# find 3-gram features
mask = np.array([len(feature.split(" ")) for feature in feature_names]) == 3
# visualize only 3-gram features
mglearn.tools.visualize_coefficients(coef.ravel()[mask],
                                     feature_names[mask], n_top_features=40)
```

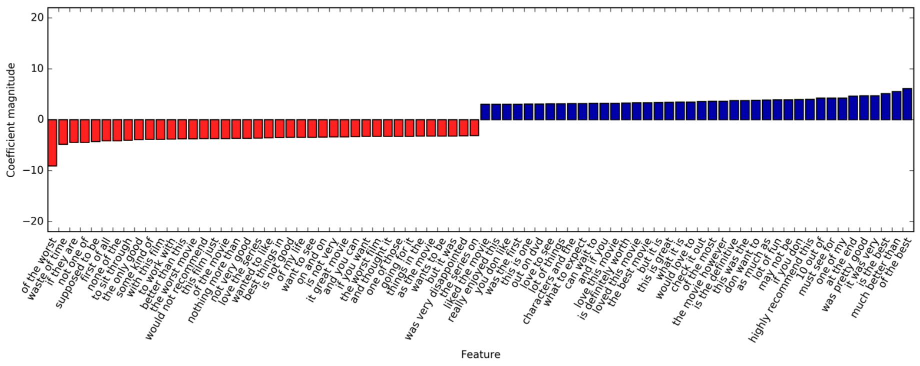

Figure 7-5. Visualization of only the important trigram features of the model

Advanced Tokenization, Stemming, and Lemmatization

As mentioned previously, the feature extraction in the `CountVectorizer` and `TfidfVectorizer` 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"`, `"replacement"`, `"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.