Big Data Meets Small Data

Sentiment analysis in R

Contents

VU Amsterdam

This tutorial

In the first tutorial manual you learned a (quick) validation of the OBI4wan sentiment analysis tool. Your specific findings were different (because you could choose your own topic), but in general people found that the sentiment labels (positive, negative or neutral) were not very accurate. In the reflection on this accuracy you identified various limitations. For instance, it makes sense that a computer would have a hard time dealing with sarcasm, where someone might say something positive, but actually mean it in a negative way. But there were also less obvious limitations. A message like "Corona related hospital admissions have declined" might be labeled negative, even though it's good news. What is happening here is that the individual words tend to be associated with negative messages (corona, hospital, declined). It is only when we interpret the whole sentence that we see that in this case something negative (declining) is happening to something negative (corona related hospital admissions), and that makes it pretty positive. Another issue was the simplification of sentiment into just three categories. Overall, most messages were labeled neutral, but messages are rarely completely neutral, so how negative or positive should a message be before we label it as such?

In this tutorial you will develop a better intuition for the limitations of automatic sentiment analysis by taking a look under the hood of two general approaches for classifying texts into categories: rule-based and supervised machine learning. As you'll see, these techniques can be applied for many more purposes than just sentiment analysis. Supervised machine learning in particular has become a hugely important technique

to understand (or at least know about), that is at the hearth of many revolutionary feats of computers, such as machine translation, image recognition, and even self-driving cars. We won't dive that deep into the method, but you will see what it is, and even apply it yourself.

Rule-based approach vs supervised machine learning

You worked with `unsupervised machine learning` in the previous tutorial, when you trained and interpreted a `topic model`. What set's this approach apart is that it does not require any supervision. We simply gave our computer a bunch of texts, and it came up with useful patterns in the data (in this case topics).

For sentiment analysis this won't do. We can't expect a computer to learn how to measure sentiment by itself. We need to teach a computer how to do this. The difference between `rule-based` and `supervised machine learning` is basically about how we will teach this:

Rule-based approach

One way to teach a computer to perform a certain measurement, is by giving the computer a set of specific rules for how to measure. An example that you already worked with is search queries. A query tells a computer very specifically what to look for. The query `corona AND vaccin*` tells the computer unambiguously: if a text contains the word "corona", and contains a word that starts with "vaccin", measure this as TRUE (i.e. the text did match the query). If not, measure this as FALSE.

The benefit of this approach is that you have complete control over how the computer performs the measurement. The limitation is that for good performance on complicated measurements you would need very many and very specific rules.

Sentiment analysis is a very good example of such a complicated measurement task. A common rule-based approach for measuring sentiment is to create a `dictionary`, that tells us which words are `positive` or `negative`. We can then simply let the computer count how many negative and positive words occur in a text, and use this to calculate a sentiment score. Note that this is very similar to working with search queries. Each term in the dictionary is a simple rule for the computer: for every time the word "decline" occurs, add 1 to the number of negative words in this text.

This approach works to some extent, and has been used in many studies. However, it has some clear limitations, and several recent studies show that the accuracy of rule-based sentiment analysis with dictionaries is often bad (see e.g., Van Atteveldt, Van der Velden & Boukes, 2021). The problem is simply that sentiment in language is so complex, that it's virtually impossible to make a sufficiently complex list of rules. Some dictionaries have thousands of words, include special rules for negations ("not good") and amplifiers ("very good!"), and distinguish between how positive or negative words are ("best" is more positive than "good"). And yet still, it tends to perform poorly for many real-life applications.

Supervised machine-learning

So, a big limitation of rule-based approaches for sentiment analysis is that we just can't come up with a sufficiently complex set of measurement rules. But how can we teach a computer to do something without specifically telling it how to do it? This is where `supervised machine-learning` comes in. In `supervised machine learning` we provide the computer with a `training data set`, and then let the computer figure out the rules by itself!

For example, consider the case of a `spam filter`. We want to teach a computer (otherwise known as 'model' or 'algorithm') to automatically label new emails as spam or not spam. We first need to create our training data set. For this, we collect some old emails, and manually label them as either spam or not spam. So now

we can give the computer a set of emails that are spam, and a set of emails that are not spam, according to our own interpretation. During the learning phase in supervised machine learning, the computer gets a feel for differences between spam and non-spam emails. Stated differently, the computer learns the 'rules'[1] for distinguishing between spam and non-spam emails. Then, when the computer receives new emails, it can draw on its knowledge of the rules to decide whether each email is more likely to be spam or not.

A very simple machine learning algorithm would, for instance, just look at the occurrences of certain words. By looking at the training data set, spam-detection algorithms learn that certain words are much more likely to appear in spam emails. As a result, when spam-detection algorithms identify such words in new emails, they decide that these emails are likely to be spam as well.

The cool thing about supervised machine learning is that this same approach works for many different types of data. The input can be a set of emails that we want to classify into either spam or not spam, but it can also be a set of animal pictures that we want to classify by the name of the animal shown in the them. If you take this much further, it could be many games of chess to learn which next move is most likely to make you win. Generally speaking, supervised machine leaning lets you teach computers to perform complicated tasks without having to tell them how to perform these task. All you need to do is expose the computer to a sufficient training data set so that it learns the rules for performing a task.

Sentiment analysis in R

For this tutorial, we will use data from Amazon product reviews. This is nice data to demonstrate automatic sentiment analysis, because next to the text of the review, we also have data about the actual review score. It is to be expected that reviews that gave a low score are also more negative in their review text, and that reviews with a high score are overall more positive.

So what we'll do is pretend that these review scores are like a gold standard. Based on the review score (from 1 to 5), we can compute sentiment labels (negative, neutral and positive). Our goal will then be to predict these labels based on just the text. We can then compare how well this works with a rules based approach and with supervised machine learning. Using review data is mainly for sake of convenience, because it means that we won't have have to manually code our data for this tutorial[2].

The data for this tutorial

We provided two sets of Amazon reviews. One about pet products, and one about beauty products. For the first part we'll focus on just the pet products. The csv files have already been cleaned, and can be imported with the regular `read_csv` function from the `readr` package.

```
library(readr)
pet_supplies = read_csv(here::here('data/Pet_Supplies_5.csv'))
```

Let's see what we have.

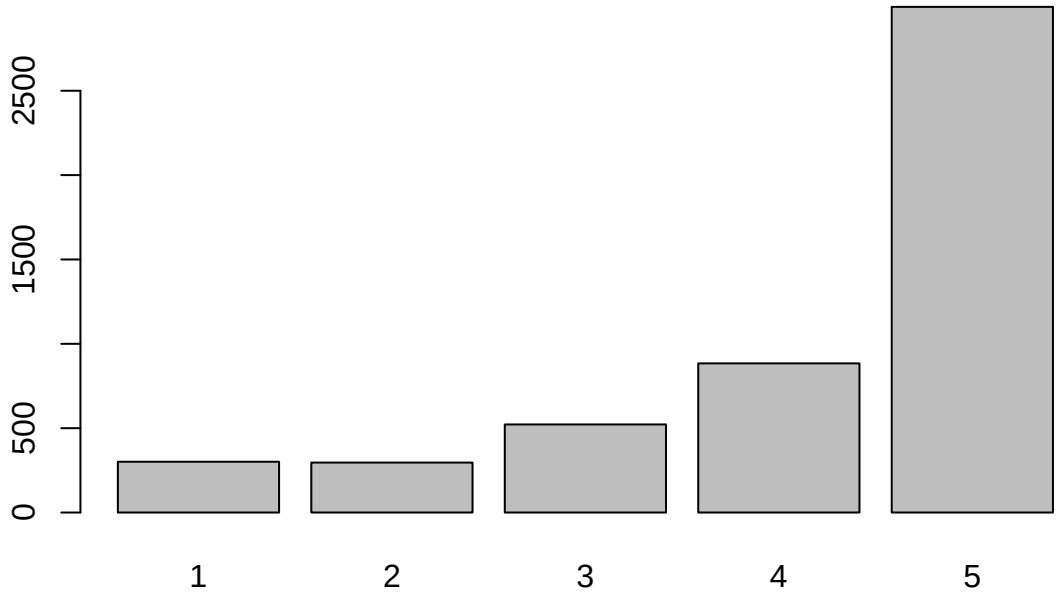
[1] It should be mentioned that at this point the definition of 'rules' is getting a bit fuzzy. Here, rules are being used in the sense of 'patterns' or 'relationships'. In simple machine learning algorithms, the computer will come up with simple rules. The Naive Bayes algorithm that you'll use in this tutorial basically creates something very similar to a dictionary. But there are also much more complex algorithms where 'rules' exist as a complex neural network.

[2] Although we use reviews here just for convenience, they are often used to train machine learning models. By training a model to predict a review score based on a review text, it can learn something useful about how people might express their positive or negative opinion in text. The problem, of course, is that a model trained on review data might not perform very well to measure sentiment on social media texts. Positive and negative opinions about certain types of products are likely to be expressed rather differently from opinions about a brand on Twitter.

```
pet_supplies
```

The columns that we're mainly interested in are the `text`, the `summary`, and the `score`. The score is the review score on a scale from 1 to 5. The following code makes a bar chart of the number of reviews per score, so we get an idea of how the review scores are distributed.

```
score_frequencies = table(pet_supplies$score)
barplot(score_frequencies)
```



Most reviews are very positive (5/5), and relatively few reviews are very negative (1/5).

Rule based sentiment analysis in R

We could do a dictionary analysis with `quanteda`, but we'll now use a different package called `corpustools`. The reason is that this package allows us to visualize how the dictionary matches the texts, which helps us see where the dictionary might make mistakes.

```
install.packages('corpustools')
```

We'll also install the `quanteda.dictionaries` plugin, which has some pre-made dictionaries that we can use. Installing this package is a bit different from other packages (long story short, this package mainly contains dictionary data, so it's larger than 'normal' packages). We first install the `remotes` package, which then allows us to install the package from `GitHub`. You might get a question: `These packages have more`

`recent versions available. It is recommended to update all of them. Which would you like to update?`. In this case, you should choose option 1 or 2 (type 1 or 2 in your console and press enter).

(If you can't install quanteda.dictionaries, you can also continue without it. We'll provide a csv file with the dictionary for this tutorial as a back-up plan)

```
install.packages('remotes')
remotes::install_github('kbenoit/quanteda.dictionaries')
```

Remember that you only need to install packages once, but you'll still need to run `library(packagename)` to use them in the current session.

```
library(corpustools)
library(quanteda.dictionaries)
```

Corpustools is similar to `quanteda` in that we first tokenize our data by breaking texts into words. The main difference is that corpustools preserves the order of words. Here we create the corpus, specifying that we want to use the `text` column for our analysis. (note that we could also have used the texts in the `summary` column, and we might just ask you to do that later on)

```
pet_supplies = tidyr::drop_na(pet_supplies, text)
tc = create_tcorpus(pet_supplies, doc_column = 'id', text_columns = 'text')
tc
```

```
## tCorpus containing 517366 tokens
## grouped by documents (n = 4997)
## contains:
##   - 3 columns in $tokens:    doc_id, token_id, token
##   - 4 columns in $meta:      doc_id, name, summary, score
```

We now have a corpus with about 500,000 words, over 5000 reviews.

Now, let's prepare a dictionary. As mentioned above, a dictionary is just a list of words, where each word is related to some kind of category or score. There are many different available sentiment dictionaries, that have been developed for different purposes (e.g., sentiment in economic news, sentiment on Twitter) Here we'll use the Affective Norms for English Words dictionary, which was developed for microblogs such as Twitter.
We already got this data by opening `quanteda.dictionaries` (so besides functions, packages can also provide data)

> !! If you weren't able to install quanteda.dictionaries, you can also load the dictionary from a csv file

```
## only run this code if you couldn't install quanteda.dictionaries
data_dictionary_AFINN = read_csv(here::here('data/AFINN_dictionary.csv'))
data_dictionary_AFINN$dict_sentiment = data_dictionary_AFINN$code
```

```
data_dictionary_AFINN
```

Here you see a glimpse of the words in this dictionary. There are two categories: positive and negative. For the positive category there are 878 words, and for the negative category there are 1599 words.

Now, we'll use this dictionary to find the positive and negative words in our corpus. We store the results in a column called `dict\_sentiment` (sentiment based on the dictionary), to make a clear distinction from the "real" sentiment based on the review score.

```
tc$code_dictionary(data_dictionary_AFINN, column='dict_sentiment')
```

Note that this function is a little different from what you've seen before. Also, we don't assign the output to a name (normally we'd do something like: `output = function(input)`) We'll skip over the details, but just know that in this case the results of applying the dictionary have directly been stored in our corpus.

Let's actually look under the hood what this looks like, because it will show you that what we've now done is really quite straightforward. In the corpus `tc`, the texts are represented as a data.frame in which each row is a words. This means that the data.frame is pretty huge (500,000+ rows). So let's just look at the head (i.e. the top) of the data.frame, for the first 50 rows.

```
head(tc$tokens, 50)
```

This gave you 50 rows, where each row tells us the id of the document, the id of the token (the position of the words in the document), the token text, and now also columns for `sentiment_id` and `sentiment`. If you scroll to row 35, you see that for the token `likes`, the sentiment column tells us that in the dictionary this word is marked as a `positive` word.

So really, all we did is look up the words of the dictionary in all of our reviews, and add the dictionary labels. The rest of the analysis is simply counting the words, and using the frequencies of positive and negative words to calculate a sentiment score.

We'll do this in the next section, but let's first show why we decided to do this part in `corpustools`. To count single words, we could just as well have used a document term matrix, which would have been more efficient (we wouldn't have had to store the locations of each word). But by keeping the order of words intact, we can now also visualize the dictionary words in the original texts.

```
browse_texts(tc, n = 10, category='dict_sentiment')
```

You now see the texts for the first 10 documents, with colours indicating which words the dictionary considers `positive` (blue) and `negative` (red). This should give you a pretty good idea of how a dictionary analysis works. It really just matches words (or short phrases) to texts.

(note that the colors are random, because for the computer these are just categories. It's just a coincidence that the negative words are red in this case)

Calculating a sentiment score

Now let's calculate a similarity score based on the number of positive and negative words in a review. For this, we'll first create a data.frame in which we count the total number of positive and negative words for every review. Here we also include the `score` (the review score on a scale from 1 to 5), because we can use this later to see whether the sentiment measured by the dictionary is similar to the 'real' sentiment of the reviewer.

```
sent_counts = count_tcorpus(tc, meta_cols = c('doc_id', 'score'),
                            feature='dict_sentiment', count = 'tokens')
sent_counts
```

```
##       doc_id score N   V negative positive
##    1:      1     5 1 278        2       10
##    2:     10     5 1 112        3        4
##    3:    100     5 1   3        0        1
##    4:   1000     2 1 124        2        3
```

```
##    5:   1001     2 1 121        3        1
##   ---
## 4993:    995     5 1  51        0        7
## 4994:    996     5 1  33        0        1
## 4995:    997     3 1 103        0        3
## 4996:    998     1 1  39        2        0
## 4997:    999     5 1  92        0        2
```

Now, we can use the `negative` and `positive` columns to calculate a single sentiment score. This is actually not as simple as it sounds, because there are different possible formulas with different implications. For example, one common formula is.

$$\frac{positive - negative}{positive + negative}$$

This gives us a nice score ranging from -1 (if all words are negative) to 1 (if all words are positive). However, we then do not take into account how many sentiment words there are in total. If a review only contains 1 positive word, and no negative words, this would give us the highest possible sentiment score of 1 (try it by pluggin these numbers into the formula). We could instead divide by the total number of words in the text, in which case -1 would mean that all words in the text are negative, and 1 would mean that all words are positive. However, since most words are neutral, this would mean that all scores are squeezed to very small numbers. Instead, we'll use a nice alternative formula, that balances these advantages and disadvantages.

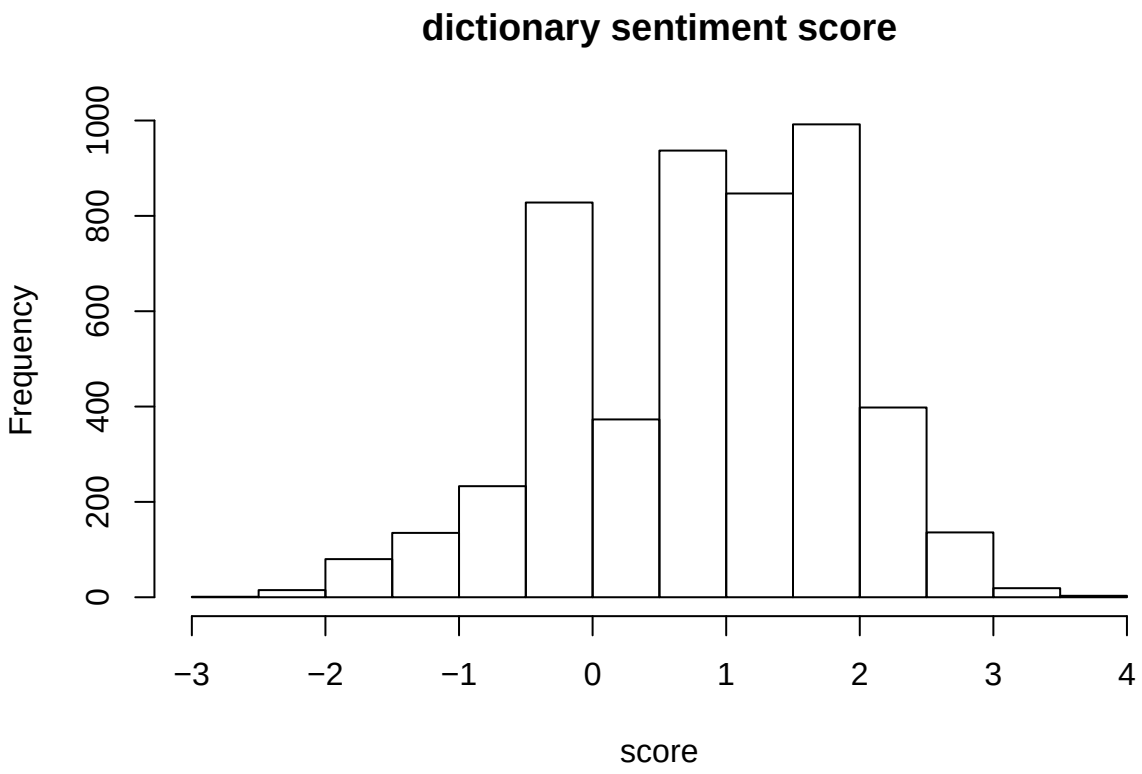
$$\log(positive + 0.5) - log(negative + 0.5)$$

This measure also creates a score where a negative value means negative sentiment, and a positive value means positive sentiment. The scores can be higher than 1 and lower than -1, but the log transformation ensures that you don't get extremely high values (you don't need to understand the math for this course). The following code performs the calculation for every row in our data. We store the results in the column `dict_score` (dictionary score).

```
sent_counts$dict_score = log(sent_counts$positive + 0.5) - log(sent_counts$negative + 0.5)
```

Now we have a sentiment score. Let's first look at how this score is distributed. Here we make a histogram of the scores.

```
hist(sent_counts$dict_score, xlab='score', main='dictionary sentiment score')
```

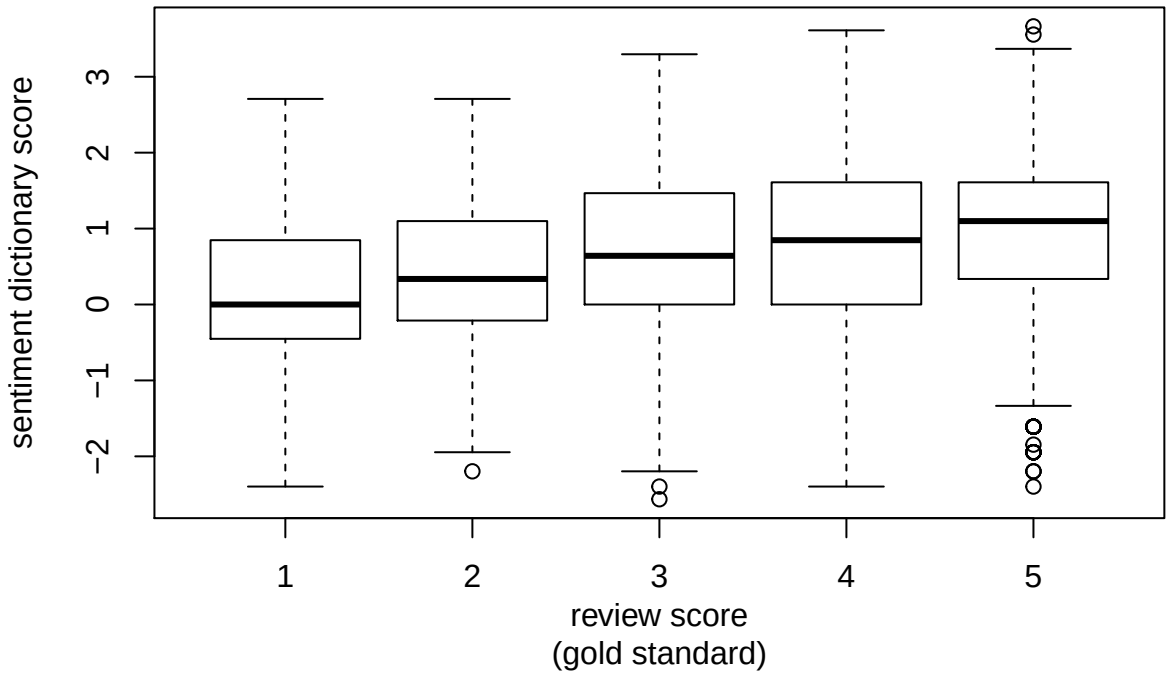


On the x-axis we see the score, and on the y-axis we see how many reviews had (approximately) this score. Most reviews had scores around 1 or 2, which indicates that according to our dictionary analysis most reviews are positive. This is a good sign, because before we saw that most reviews were indeed positive based on the review scores.

Validating the sentiment dictionary

However, it doesn't yet mean that the results are good. It might be that the dictionary is just inclined towards more positive scores. To look a bit closer, let's look at the average dictionary scores for every review scores (1 to 5). The next code produces a boxplot.

```
boxplot(dict_score ~ score, data=sent_counts,
        xlab='review score\n(gold standard)',
        ylab='sentiment dictionary score')
```



Here on the x-axis we see the review scores, where 1 is the most negative and 5 the most positive. On the y-axis we see the average sentiment dictionary scores in boxplots. The black lines tell us the average score, and the grey box shows the interquartile range (that is, 50% of the observations are within this box. For example, we see that for the most negative reviews (1 on the x-axis), the average score from the sentiment dictionary is around 0, and 50% of the scores range from around -0.45 to 0.85.

If you're paying attention, this should surprise you. It means that even for the most negative reviews (1 star out of five), our sentiment dictionary says the reviews are neutral! So that's a clear limitation of using this dictionary to measure how positive these reviews are. On the other hand, when we look at the full picture, we do see that the sentiment dictionary does something right. As the review score increases from 1 to 5, the average sentiment dictionary score also goes up. This indicates that there is a positive correlation between the review score and the sentiment dictionary score. In fact, we can calculate this.

```
cor.test(sent_counts$score, sent_counts$dict_score)
```

The correlation is positive and significant ($p < 0.001$), but also rather weak, $r(4998) = 0.243$. You could have guessed this from the boxplot, where there is quite some overlap between the interquartile range boxes.

But hey, you might be thinking, what if we just look at whether the review is positive, negative or neutral? This way we could calculate the precision and recall, which might be more intuitive, and maybe at this level the performance is also actually pretty ok? Excellent idea, so let's do that in the first assignment.

Assignment 1: The sentiment dictionary approach

For the first assignment you'll do the validation yourself, but this time we'll use the `Beauty products` review data to switch things up a bit. We'll first recap the code that we discussed above for preparing the data.

Although we don't actually need to run the `library()` functions again, we include every step here so that the code is fully self contained (i.e. does not rely on previous steps).

> Make sure to run this code, because otherwise you might accidentally use the pet supplies data for this assignment. Note that you'll need to change the location of the file in `read_csv`.

```
library(readr)
library(corpustools)
library(quanteda.dictionaries)

## read data
## (with n_max = 5000 we only read the first 5000 reviews in this csv)
beauty = read_csv(here::here('data/Beauty_5.csv'), n_max = 5000)
beauty = tidyr::drop_na(beauty, text)

## create corpus and apply dictionary
tc = create_tcorpus(beauty, doc_column = 'id', text_columns = 'text')
tc$code_dictionary(data_dictionary_AFINN, column='dict_sentiment')

## calculating sentiment score
sent_counts = count_tcorpus(tc, meta_cols = c('doc_id', 'score'),
                            feature='dict_sentiment', count = 'tokens')
sent_counts$dict_score = log(sent_counts$positive + 0.5) - log(sent_counts$negative + 0.5)
```

We'll start with a simple, face value validation. That is, you'll have a look at some of the articles to get a first look at how accurate the dictionary is for our data. To do so, we'll again use the `browse_texts` function.

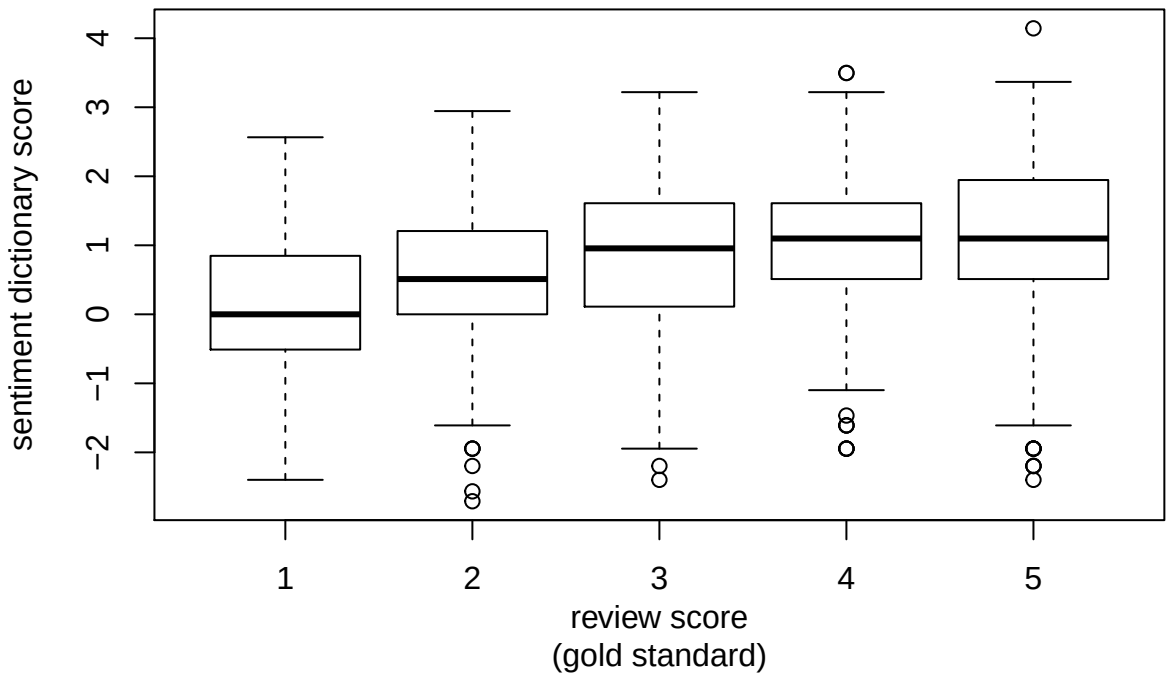
```
browse_texts(tc, n=10, category='dict_sentiment')
```

Have a good look at the first 10 reviews and answer the following questions (read all questions first, so you know what to look for).

> **Question 1.a**. How well does the number of positive and negative words correspond with the real review score (shown at the top of each review)? Does the dictionary seem to do well in measuring the sentiment?
> **Question 1.b**. List five cases where you think the label (positive or negative) that the dictionary assigned to a word is not correct. Explain why they're wrong. **Question 1.c**. List five words that the dictionary did not label (i.e. that have not been coloured) that should have been labeled. Would you be able to improve the dictionary by adding these terms?

Now let's look at the boxplot of the sentiment dictionary scores per review score for this data.

```
boxplot(dict_score ~ score, data=sent_counts,
        xlab='review score\n(gold standard)',
        ylab='sentiment dictionary score')
```



Question 1.d. Describe what the boxplot tells you about the validity of the sentiment dictionary analysis. Mention at least one good sign and one bad sign.

Now, let's calculate the precision and recall. This does mean that we first need to transform our sentiment scores into sentiment labels (positive, negative or neutral). To do so, we can use a cut-off point in both the real sentiment score (the reviewer score) and the dictionary sentiment score. That we, we say that if the score is lower than a certain value, it's negative, and if it's higher than a certain value it's positive.

This is, however, a bit arbitrary. There isn't really a good way to determine what threshold makes most sense, and in this case we're working with two rather different scales. We also don't want to complicate this part of the assignment, so we'll simply pick some thresholds around the center scores. For the reviews on a 5-point scale, we'll say that it was negative if the score was 2 or lower, positive if it was 4 or higher, and neutral if it was 3. For the dictionary score, we'll say that it was negative if the score was below -0.5, positive if it was above 0.5, and neutral if it was in between.

```
sent_counts$real_sentiment = 'neutral'
sent_counts$real_sentiment[sent_counts$score <= 2] = 'negative'
sent_counts$real_sentiment[sent_counts$score >= 4] = 'positive'

sent_counts$dict_sentiment = 'neutral'
sent_counts$dict_sentiment[sent_counts$dict_score < -0.5] = 'negative'
sent_counts$dict_sentiment[sent_counts$dict_score > 0.5] = 'positive'
```

We can look at the old and new columns to see that this worked

```
sent_counts[,c('doc_id', 'score', 'real_sentiment', 'dict_score', 'dict_sentiment')]
```

For every review we now have a `real\_sentiment` score, that's based on the actual score given by the reviews. We also have a `dict\_sentiment` score, which is the sentiment label based on the dictionary analysis. So now we hope you remember the first tutorial, because we have all we need to perform a precision and recall measurement. The `real_sentiment` score is our gold standard. It tells us how positive the reviewer really was. The `dict_sentiment` score is our measurement. We can now make the same table that we used in the first tutorial to calculate precision and recall.

```
table(gold_standard = sent_counts$real_sentiment,
      dictionary = sent_counts$dict_sentiment)
```

```
##              dictionary
## gold_standard negative neutral positive
##      negative      139     205      307
##      neutral        41     130      378
##      positive      191     675     2932
```

> **Question 1.e**. Calculate the precision, recall and F1 score for all three categories (positive, negative and netural). Make sure to also report the calculations. **Question 1.f**. Use these scores to interpret the validity of the sentiment dictionary. You don't have to use all scores, just the ones that are most relevant to determine where the measurement is good (or at least decent) and bad.

Supervised machine learning

Now let's do more or less the same thing, but with supervised machine learning. That is, we're going to `train` a machine learning model for automatically labeling a review as `positive`, `negative`, or `neutral`, based on the review text. There's going to be quite some code here, but we'll re-cap it in a single chunk when we get to the assignment.

Just like above, we will use the actual review scores (from 1 to 5) as a gold standard. However, in this case we'll also need to use these scores to `train` our data first. That is, we'll need to give our computer a set of review texts with the "real" sentiment label (based on the review scores) so that it can `learn` how to measure sentiment.

For this part we'll use quanteda. This time, we'll also need to install another quanteda plugin for textmodels (which include machine learning models).

```
install.packages('quanteda.textmodels')
```

```
library(quanteda)
library(quanteda.textmodels)
```

Preparing the data

Let's first prepare our data. This time we'll immediately use the Beauty products data, so that you can more easily compare it to assignment 1. Also, this time we'll read all the data (20,000 reviews), because we'll need some additional reviews for training our model. We'll also immediately transform the review scores (from 1 to 5) to sentiment labels (negative, neutral, positive), using the same approach as before.

```
library(readr)
beauty = read_csv(here::here('data/Beauty_5.csv'))
beauty$real_sentiment = 'neutral'
beauty$real_sentiment[beauty$score <= 2] = 'negative'
beauty$real_sentiment[beauty$score >= 4] = 'positive'
```

For this tutorial we'll use a simple (but still very effective) machine learning algorithm that only looks at word frequencies (remember, our bag-of-words assumption). So for this approach we're again going to create a Document Term Matrix (DTM), just like in the previous tutorial.

In the following line of code we use a little `syntactic sugar` for creating the DTM with quanteda. It does the same thing we did before, but with a different style that is convenient for chaining many functions together. We wanted to at least show this once, because you'll often see this style in modern R code, but you don't need to understand what's happening (for those interested, see here) In short, we create a `pipeline` of functions. Here we take our `beauty` data and ram it through 6 functions to (1) make a corpus, (2) tokenize the data, (3) structure it as a DTM, (4) remove words that occur less than 10 times, (5) remove stopwords, and finally (6) stem the words.

```
dtm = beauty %>%
  corpus(text_field='text') %>%
  tokens() %>%
  dfm() %>%
  dfm_trim(min_termfreq=10) %>%
  dfm_remove(stopwords('en')) %>%
  dfm_wordstem()
```

This might give a warning that "NA is replaced by empty string", but you can ignore it. In general, `warnings` are often not a problem. In this case, it just let's us know that some reviews in our data has missing (NA) texts.

The next step is that we'll split our data into `test` and `training` data. We'll use the training data to train our model, and the test data to evaluate our model by calculating the precision and recall. It's important to keep these separated, because if we test the model on the same data that we trained it on, it's not really a fair test. It would be like giving students the same questions that they get on the exam to train themselves for the exam.

In this case we'll just split our data so that 75% (15,000 reviews) are in our training data, and we holdout 25% (5000 reviews) for testing. To do this, we first sample 5000 document names for testing. Then we take a subset of the dtm where the document name is in this sample and where it's not in this sample (the `!` means NOT). Note that this is a random sample! So your results might look slightly different from ours.

```
test_docnames = sample(docnames(dtm), size = 5000)
test_dtm = dfm_subset(dtm, docnames(dtm) %in% test_docnames)
train_dtm = dfm_subset(dtm, !docnames(dtm) %in% test_docnames)
```

This is called the `holdout method`, because we holdout a sample of the data for testing. There are actually better approaches. For instance, we could split the data in 5 equal parts, then train on 4 parts and test on 1, and repeat this 5 times while rotating the parts. This way each review will be used for both training and testing. This is called `k-fold cross-validation`, where k is the number of parts (in this example 5). However, here we'll just stick with the holdout method because it's easier to understand and apply.

Training and testing the model

So now we can use the training data to train the model. In this case we're using a Naive Bayes classifier. For this we use the `textmodel_nb` (`nb` for `Naive Bayes`) function from the quanteda.textmodels plugin. As

input we just need to provide the DTM (train_dtm), and the labels that we want to model to be able to predict. In our case these are the `real_sentiment` labels, based on the reviewer scores. We'll call our sentiment classifier `sent_classifier`.

```
training_labels = docvars(train_dtm, 'real_sentiment')
sent_classifier = textmodel_nb(train_dtm, y = training_labels, prior = 'docfreq')
```

So now we've trained ourselves a sentiment classifier! From here on it's smooth sailing. We can now simply use this classifier to `predict` the sentiment labels of our test data. Notice that this is also quite literally what the following code says.

```
predicted_sentiment = predict(sent_classifier, newdata = test_dtm)
```

And we also know the `real_sentiment` labels for the test data.

```
real_sentiment = docvars(test_dtm, 'real_sentiment')
```

So now we have both the `real_sentiment` and the `predicted_sentiment`. These are both vectors with just the positive, negative and neutral labels. So now we can create the confusion matrix just like before.

```
cm = table(gold_standard = real_sentiment,
           predicted = predicted_sentiment)
cm
```

```
##              predicted
## gold_standard negative neutral positive
##      negative      300      81      236
##      neutral        93     139      279
##      positive      173     198     3501
```

This might also be a good time to show you an easier way to calculate the precision and recall. The diagonal of the matrix always holds the true positives for each category. We can divide this by the column sums (number of times the category was measured/predicted) to get the precision, and divide by the row sums (number of times the category occured in gold standard) to get the recall.

```
precision = diag(cm) / colSums(cm)
recall = diag(cm) / rowSums(cm)
F1 = 2 * (precision * recall) / (precision + recall)

data.frame(precision, recall, F1) %>% round(2) ## there's that pipe thing again!
```

```
##          precision recall   F1
## negative      0.53   0.49 0.51
## neutral       0.33   0.27 0.30
## positive      0.87   0.90 0.89
```

These results are already much better than the sentiment dictionary, but they're still not amazing. (note that your results can be a bit different, because the selection of train/test cases was random). In particular, the neutral category turns out to be hard to predict. For the positive and negative categories it works pretty ok.

Could we have done better? Certainly. We now blindly used a rather simple machine learning model approach. In fact, if we look under the hood of our Naive Bayes classifier, we'll find that it bears some resemblance to a dictionary, but where the computer itself learned to relate words to sentiment labels. Here you see a selection of features, with for each feature a score for every label.

```
summary(sent_classifier)
```

This is nice, because it means that we can really understand what's going on under the hood in this particular model. But as argued above, a major benefit of machine learning is that the computer can learn much more complicated patterns, and with this model we didn't really make use of that. We could have used a more powerful model, but that is out of the scope of this assignment, as we suspect that your heads are pretty full by now. For a more rigorous comparison of precision, recall and F1 scores for several dictionaries and machine learning models on a sentiment classification task, you could check out Table 2 of this paper.

Assignment 2: The supervised machine learning approach

For the second assignment you'll interpret the results we just discussed in your own words. Here we provide the same code as above, so you can just copy paste this code to be sure that it works. (though you will need to change the location of the file in `read_csv`) Note that the results that you'll get can be a bit different, because spitting the DTM in to the test and train DTM's is random (due to the `sample` function).

```
library(quanteda)
library(quanteda.textmodels)
library(readr)

## prepare the data
beauty = read_csv('data/Beauty_5.csv')
beauty$real_sentiment = 'neutral'
beauty$real_sentiment[beauty$score <= 2] = 'negative'
beauty$real_sentiment[beauty$score >= 4] = 'positive'

## create the dtm, and split into test and training parts
dtm = beauty %>% corpus(text_field='text') %>% tokens() %>% dfm() %>%
  dfm_trim(min_termfreq=10) %>% dfm_remove(stopwords('en')) %>% dfm_wordstem()

test_docnames = sample(docnames(dtm), size = 5000)
test_dtm = dfm_subset(dtm, docnames(dtm) %in% test_docnames)
train_dtm = dfm_subset(dtm, !docnames(dtm) %in% test_docnames)

## train the model
training_labels = docvars(train_dtm, 'real_sentiment')
sent_classifier = textmodel_nb(train_dtm, y = training_labels, prior='docfreq')

## test the model
real_sentiment = docvars(test_dtm, 'real_sentiment')
predicted_sentiment = predict(sent_classifier, newdata = test_dtm)

cm = table(gold_standard = real_sentiment,
           predicted = predicted_sentiment)
cm

## calculate precision and recall
```

```
precision = diag(cm) / colSums(cm)
recall = diag(cm) / rowSums(cm)
F1 = 2 * (precision * recall) / (precision + recall)
data.frame(precision, recall, F1) %>% round(2)
```

> **Question 2.a**. Compare the validity (precision, recall and F1) of the dictionary analysis (question 1.e) to that of the supervised machine learning analysis. Which approach is better in what ways? **Question 2.b**. On the line where we create dtm (`dtm = ...`), we have a pipeline in which we take several pre-processing steps. One of these is that we remove stopwords. Remove this step from the pipeline and repeat the analysis (i.e. run all the code). Does the precision / recall / F1 change a lot? How would you explain this? (we don't expect a deep technical answer, but you should be able to reflect on how this might affect how the computer can measure sentiment) **Question 2.c**. This time, remove the step from the pipeline where we trim the dfm to remove all words that occur less than 10 times. Does this change the precision / recall / F1 scores, and how would you explain this?

Now first restore the code (make sure that you again include `dfm_trim` and `dfm_remove` in the pipeline).

It turns out that especially the neutral category is pretty hard to predict, and before we also saw that the majority of reviews is positive. So instead of trying to predict all three categories, we might also see whether we can at least predict whether a review was positive or not. This can be quite useful, because a company might for instance just want to identify when people are not (completely) satisfied, and so would only be interested in accurately finding cases where a review was not positive.

> **Question 2.d**. Change the code so that instead of working with three categories (negative, neutral and positive), you'll just work with two categories ('super positive' and 'not super positive'). As a cut-off point, say that all reviews with a score of 5 are 'super positive', and all reviews with a score of 4 or lower are 'not super positive'. Now re-train the model and calculate the precision and recall. Paste the results in your report, and interpret what you see. Is the model any good, and how would you be able to use this?

Assignment 3: reflection

We've now talked a lot about dictionaries and machine learning as two approaches for automatic sentiment analysis. But now we'll let you do the talking. The final assignment is to share what you learned with yet another fictional boss for a fictional company that you fictionally work for.

> **Question 3**. You work for Mike, who sells shoes. Mike figured that to know what his customers do and do not want in the fascinating world of shoes, it could be usefull to automatically find social media messages about shoes where people are either very positive or very negative. Mike doesn't like social media dashboards (Mike only likes shoes), so he wants the company to built its own tool. He heard about dictionaries and supervised machine learning as two possible approaches, but he doesn't quite get how they are different and which one would make sense to use in this case. You are asked to write a short summary about what these methods are, and what the pro's and con's would be for this new tool.