Module 2

Sentiment analysis and data stream mining

2.1 Sentiment analysis

2.1.1 Introduction

World of opinions

People are social beings. We want to place ourselves within a group, and part of such placing behaviour is to adjust our needs, tastes, preferences and opinions to others. We are conformists. Moreover, we are living in a constantly growing stream of subjective information.

According to two surveys from 2007 and 2008 conducted on American online users, more than 80% of Internet users did online research on a product at least once. 20% do so on a typical day. Online reviews have strong impact on our choices – it is significant for example in the domain of restaurants, hotels and other services.

Applications

Why are we interested in mining opinion? Let's discuss a couple of examples of applications for sentiment analysis:

- customer feedback, brand analysis – typical online product review, opinion mining on airlines[1];
- opinion retrieval – presidential job approval for Obama [7];
- financial predictions – stock prices following public mood [1];
- following important events – sentiment visualization.

Customer feedback and brand analysis are quite simple – checking product or service reviews. It is also possible to check the common opinion about some events or persons like politics, celebrities and others. As an example look at the work of Brendan O'Connor and his colleagues on mining public opinion on President Obama's work [7]. Another application is the analysis of the financial data and its relation to social media [1]. One may also want to scan public opinion to pick important social, cultural, financial or political events.

[1] See: `http://jeffreybreen.wordpress.com/2011/07/04/twitter-text-mining-r-slides/`

Importance of opinion mining

We have already presented some examples of applications for opinion mining. The question is whether opinion mining is important and interesting at all? Possible reasons:

- popularity – plenty of applications, academic, commercial;
- commercially important;
- interesting text classification task – covering many aspects of NLP.

Opinion mining is interesting because it is popular and potentially rewarding. Many people, research teams and companies are interested in opinion mining. Sentiment analysis has straightforward commercial application. Finally, it is an open area of science which covers many vital aspects of Natural Language Processing like feature extraction, tokenization, part-of-speech tagging, lexicons, classifier, stemming and many other.

Sentiment analysis

Sentiment analysis is *"a computational study of opinions, attitudes, emotions, subjectivity, appraisal, affects, evaluations (...) expressed in texts"* [6]. Sentiment could be defined, using the table of affective states made by Scherer, as a kind of attitude, the affective beliefs or dispositions towards something or somebody.

Opinion decomposition

Important problem is to define what is the opinion and check if the given text is subjective. We define opinion using Liu's quintuple definition [6]:

- source/holder – who is the source of the opinion?;
- target entity;
- aspect of target – what is the subject of sentiment;
- sentiment;
- time.

For example, in a typical product review, target and its aspects, like prices or service quality, are simple to be found. Sentiment may be expressed in a typical way using polar adjectives, "good" or "poor", or domain specific vocabulary. Time and source are given.

Tasks

This is a list of sentiment related tasks, ordered by their difficulty:

1. basic polarity detection – "thumbs up/down";
2. polarity strength;
3. detection of subjectivity;
4. opinion extraction – determine structured opinion from text.

The easiest one it the basic polarity detection. Next is determining the strength of the sentiment, as in star rating systems. One of the hardest tasks is to detect subjectivity. Opinion may not be expressed in an easy way – because of the great complexity of natural language. Extracting fully structured opinion is also difficult. Sentiment analysis is domain-sensitive. For example, sentiment could be expressed using different vocabulary in commenting some service and in opinions about political events. Thus one must be aware of the in-domain and out-of-domain subtleties.

In our brief introduction to opinion mining we will restrict ourselves to the simplest task of polarity detection.

Data sources and their ease of use

Some data sources are easier to analyse than other:

- tweets – short, laconic messages;
- reviews – longer texts, orientated on its target, little noise, openly expressed entity aspects;
- blogs, discussions, comments – noisy, sarcastic, compound, multiple entities and comparisons.

The easiest texts to analyse are short, well-focused messages like tweets. Reviews are also quite good because they are subject-oriented and do not contain too much noise. Moreover, entity aspects are openly expressed. When texts are not so strictly oriented on one target and becoming larger, any sentiment analysis is much harder. Blogs, comments and discussions are compound texts, with multiple comparisons between multiple entities. They may also be full of noise and sarcastic.

Selected publicly available sentiment datasets

Here we present two examples of a publicly available data for sentiment analysis:

- Cornell Movie Review Data[2]
  - sentiment polarity dataset 2.0 – 2002, 700 positive and 700 negative reviews;
  - sentiment scale data – 2005, labels comes from a rating scale;
- Stanford Twitter Corpus[3].

The first one is Cornell Movie Review, which has many subsets. One of them is sentiment polarity dataset 2.0 containing 700 positive and 700 negative reviews, the other is sentiment scale data set where labels come from rating scale. Second example is the Stanford Twitter Corpus available on the linked site.

2.1.2 Feature extraction

How to change a given text into a valid set features?

Successful application of a classifier to text data requires building a valid and adequate input set of features. We will present a brief survey of common methods of feature extraction from text. Such a task is especially important in sentiment analysis because of its complexity and requires application of many NLP techniques:

- bag of words – term presence, term frequency;
- term positions, n-grams, context sensitivity;
- stemming;
- POS – part of speech information;
- Negation – "I didn't like that movie".

We will start with the basics by describing the process of tokenization. Next, so-called bag of words model of text will be presented. The following sections will describe how to deal with a more complex aspects of feature extraction from text: extending bag of words model above unigrams, context related information, analysis of stemming and how to utilize grammatical tagging. Finally we will present negations handling.

[2]See: `http://www.cs.cornell.edu/people/pabo/movie-review-data/`
[3]See: `http://cs.stanford.edu/people/alecmgo/trainingandtestdata.zip`

Tokenization

The simplest way to produce terms is to split a string by whitespaces. Christopher Potts, in his Sentiment Symposium Tutorial [9], made a comparison of a basic whitespace tokenizers: a Treebank-style one and his own, sentiment-aware tokenizer. His approach to a sentiment-friendly text splitting concentrates on extraction of as many as possible opinion-related information from the string:

- named entities, dates – "May 25";
- emoticons;
- mark-ups, hashtags;
- m*!*ed c**!es – masked curses;
- idioms – "when pigs fly", "have a whale of time";
- capital letters and punctuation.

Emoticons may carry strong sentiment as well as masked curses, capitalization and additional punctuation. He created parsing rules for those cases. To reduce sparsity it is also important to find named entities, dates, phone numbers and multi-word expressions. Furthermore we may improve results by extracting Twitter tags and usernames or finding informative HTML tags.

Comparison of tokenizers

Figure 2.1 presents a comparison of three different tokenizers made by Christopher Potts. The main conclusion is that the result mostly depends on the number of input data, shown on the x-axis. Performance of sentiment-aware tokenizer is slightly better for a large data set. It is promising in our case of dealing with potentially massive streams of data, but the effect is not significant.

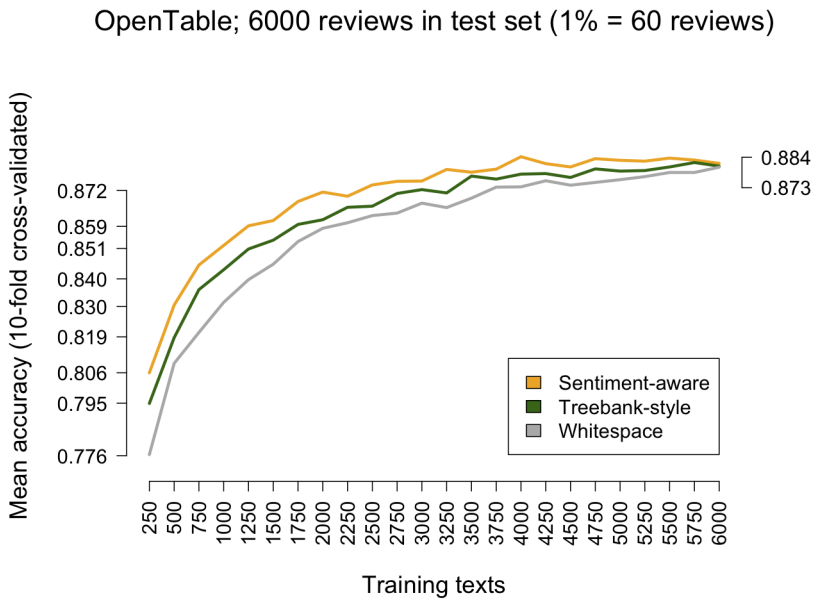

Figure 2.1: Comparison of tokenizers [9].

Bag of words

Bag of words model is the simplest model of text. Text is split into tokens named words or terms. Each word is independent of others. All context, order of words or positional information is lost. Text is transformed into a dictionary of pairs (key, value). Keys are built of unique set of words from the text. Values are counts for corresponding keys. Finally, text is presented as a vector of counts.

Another variant of bag of words is to mark only the presence of a given word in a vector representation. Binary version may be especially useful in our case of sentiment analysis – imagine that the third "amazing" word in a short message does not make it much more positive. Bag of words may result in a very sparse representation of short texts when a vocabulary is huge.

Bigrams, trigrams, n-grams

It is possible to introduce more compound features built of two or more terms. The intuition is that larger attributes contain context data which is lost in the basic bag of words representation. Such extended features or phrases are more specific. However, Pang et al. [8] showed that using unigrams might produce better results than in case of bigrams.

Storing order of words for a given text is expensive, but provides additional context data which may be utilized to built more complex, context-dependent features. We will later see a rule-based example for semantic lexicon generation proposed by Turney [11].

Part-of-speech tagging

Part of speech tagging is a useful technique for opinion mining. Turney [11] used part-of-speech tagging in his rule-based selection of phrases for building sentiment lexicons. For example, the presence of adjectives is strongly correlated with the level of subjectivity. However, one must not forget about using other parts of speech in sentiment analysis. Pang et al. [8] found, while classifying movie reviews, that using feature set restricted only to adjectives leads to much worse results then using the most frequent terms.

Stemming

Stemming could be seen as a method of reducing sparsity. It is a way to aggregate distinct, inflected words into their stem. Potts [9] made a comparison of common stemming algorithms, Lancaster Stemmer, Porter Stemmer, and WordNed Lemmatizer, and with the exception of a WordNed one, did not found stemming worth of its cost in sentiment analysis. Reported accuracy did not rise after stemming. However his analysis was done only for English language and may be invalid for other languages. For example stemming is commonly used, because of language characteristics, in sentiment analysis of Polish texts.

Negation

Negation in the sentence reverses sentiment. Das and Chen [2] and Pang et al. [8] proposed a technique to extract sentiment-aware negation features. The idea is first to find negation in the sentence, using a predefined regular expression, which covers as many valid forms of negations as possible.

Next, all words between negation and the closest clause-level punctuation are converted by adding $\_NEG$ suffixes. In that way we enlarge the feature space by separating terms which were found after negation.

2.1.3 Naive Bayes

General framework for classification

We will now present a basic approach to sentiment analysis using supervised learning. The basic algorithm is as follows. First we have to create a valid feature space. That includes – of course – tokenization and all previously mentioned aspects of feature extraction: POS tagging, negation handling and so on. Now, one must choose a classifier. In case of labelled data we could pick almost any classifier we want, but we will restrict ourselves to the basic one, the naive Bayes classifier.

We will examine in detail how to implement naive Bayes for text classification and analyse its pros and cons. We will present naive Bayes classifier because it is the most simple classifier, treated often as a baseline classifier. Although it is not sophisticated, it is very reliable and works pretty well especially in the area of text classification. How does the naive Bayes classifier work?

First we have to define what are we looking for. Given a document, we are interested in the conditional probability of the class. Using Bayes rule such conditional probability could be rewritten into the probability of a document given a class times the prior probability of the class all divided by the probability of the document itself:

$$P(c|d) = \frac{P(d|c)P(c)}{P(d)}$$

We look for the most probable class given the data. It is the maximum a posteriori class, the one which maximizes the probability of the class conditioned on data. We may forget about the denominator of the equation. Because, if we are looking for the best class given the document d, the denominator will be the same for all classes – so it does not influence the final result. Finally, the equation reduces to the likelihood times prior probability of the class:

$$c_{map} \propto \arg\max_{c \in C} P(d|c)P(c)$$

Naive assumption

Feature space for a given text is a set of attributes from x_1 one to x_n. In the simplest case one attribute corresponds to one word. The problem is that calculating the required joint probability would be infeasible or in case of long texts impossible.

To overcome that, one has to apply naive assumption. Assume that all attributes are conditionally independent. That simplifies the equation. The joint probability transforms into product of conditional probabilities of features given a class.

The naive assumption is very useful, but it is of course wrong. This is especially true for natural language where words are connected on different layers. If we want to use the naive assumption we first have to treat text as a bag of words – which means to remember only presence of words in text and forget all other context- and position-relevant data.

Parameter estimation

The prior probability of the given class c is just the frequency of that particular class – the number of documents annotated by class c divided by total number of documents.

- prior – $\hat{P}(c_j) = \frac{|\{d \in D : d \in c_j\}|}{|\{d \in D\}|}$
- likelihood – $\hat{P}(w_i|c_j) = \frac{|\{w_i \in c_j\}|}{\sum_{w \in V} |\{w \in c_j\}|}$

To estimate likelihood of the word w_i conditioned on class c_j, one must count the frequency of that word amongst all documents labelled by class c_j. The numerator is the sum of counts of w_i in all documents in the class c_j. The denominator is the total number of words in documents of class c_j.

Table 2.1: Example training set.

class	words
l	append, remove, reverse
l	index, append, remove, fromkeys
d	copy, update
d	popitem, fromkeys, update
d	sort, reverse, keys

Table 2.2: Example test set.

class	words
???	popitem, popitem, update, fromkeys

Laplace correction

If we encounter a new word, not present in the vocabulary, or one which could not be found amongst all documents of class c, the final likelihood will be zero. That happens because it is a product of those conditional probabilities of words given particular class.

The solution is to apply Laplace correction or Laplace smoothing. We simply add 1 to the number of counts in the numerator and enlarge denominator by the size of vocabulary. This is equivalent to using prior knowledge that if we have a database of size equal to the size of the vocabulary. Each word must be present in exactly one record.

Another practical hint is how to deal with tiny probabilities, which result from probability multiplication. For a large number of words, even in double precision arithmetic one will soon arrive at the product of probabilities equal to zero. The solution is to use logarithms. It does not affect the result – one will get the same most likely class given the data. However it changes the product into sum of logarithms of small values. The sum is much more reliable than product allowing one not to end with zeros.

Example

Let us study a simple example. We have two classes, l and d, which represent Python's `list` and `dict` types. Our words represent Python's methods. Training and test sets for our example are presented in Tables 2.1 and 2.2, respectively. Vocabulary consists of 10 words (methods), 7 of them are in class l, 8 in class d. Conditional probabilities for each word and class are calculated in Equations 2.1, while prior probabilities are calculated in Equations 2.2.

$$\begin{aligned} P(popitem|l) &= (0+1)/(7+10) = 1/17 \\ P(popitem|d) &= (1+1)/(8+10) = 2/18 \\ P(update|l) &= (0+1)/(7+10) = 1/17 \\ P(update|d) &= (2+1)/(8+10) = 3/18 \\ P(fromkeys|l) &= (1+1)/(7+10) = 2/17 \\ P(fromkeys|d) &= (1+1)/(8+10) = 2/18 \end{aligned} \tag{2.1}$$

$$\begin{aligned} P(l) &= 2/5 \\ P(d) &= 3/5 \end{aligned} \tag{2.2}$$

$$\begin{aligned} P(d|testobj) &\propto P(d) \times P(popitem|d) \times P(popitem|d) \times P(update|d) \times P(fromkeys|d) \\ P(l|testobj) &\propto P(l) \times P(popitem|l) \times P(popitem|l) \times P(update|l) \times P(fromkeys|l) \\ P(d|testobj) &\propto \frac{3}{5} \times \frac{2}{18} \times \frac{2}{18} \times \frac{3}{18} \times \frac{2}{18} \approx 1.4 \times 10^{-4} \\ P(l|testobj) &\propto \frac{2}{5} \times \frac{1}{17} \times \frac{1}{17} \times \frac{1}{17} \times \frac{2}{17} \approx 9.58 \times 10^{-6} \end{aligned} \tag{2.3}$$

In order to classify the test document, we follow the calculations as in Equations 2.3. The result is $P(d|testobj) > P(l|testobj)$, therefore, the test object is classified as d (`dict`).

Summary

To sum up, although naive Bayes is a very simple model it often works quite well. It is very useful in some domains, as in the case of text classification. It is a very stable model, which has high bias and low variance. Naive Bayes may not give the best predictions but is reliable. Naive Bayes has logarithmic sample complexity with the number of attributes, thus in case of little data it may outperform other models. Also the "naive" assumption is not as bad as it looks. It requires conditional independence but not independence of attributes. When there is plenty of data, naive Bayes often gives worse results than other models, like for example SVM or MaxEntropy models.

2.1.4 Sentiment lexicons

We will present an example of an unsupervised approach to opinion mining – the concept of using sentiment lexicons. The starting point is to define what a sentiment lexicon is and what is the main reason for its creation.

Next, some publicly available resources will be presented. Finally, some early work on lexicon induction, by Hatzivassiloglou and McKeown [4] and Turney [11], will be shown, as well as WordNet propagation.

Sentiment lexicon is a set of words labelled by sentiment polarity level induced using unsupervised, or semi-supervised, methods. Once built, it is used to determine the sentiment polarity of a given text or sentence. The final sentiment orientation is a function of polarities of those words from the text which are also elements of the lexicon.

What is the reason for building lexicons? It is possible to use them for domain specific application. Words in different domains may differ in their polarity – depending on the specificity of inner domain language. Using an in-domain lexicon may lead to better classification results than using general or out-of-domain tools. Lexicon induction is also an iterative procedure that could end with very big and rich sets of polarity-labelled data. Finally, using lexicons as classifiers is, once they are built, very fast. Table 2.3 presents a fragment of a lexicon (Harvard General Inquirer), with only 4 column shown – word itself, source and whether is it positive or negative.

Table 2.3: A random sample from the Harvard General Inquirer lexicon.

Entry	Source	Positive	Negative
CHERISH#2	H4Lvd	Positive	
DEMOLISH	H4Lvd		Negative
EXPENSE#1	H4Lvd		Negative
HEAD#5	H4Lvd		
IMPRESS#2	H4Lvd		
INSECT	H4Lvd		
LEADERSHIP	H4Lvd		
MALICE	H4Lvd		Negative
MAY#1	H4Lvd		
POPULAR	H4Lvd	Positive	
SOLACE	H4	Positive	
YEA	H4		

Several sentiment lexicons are in use. The most popular ones are:

- The General Inquirer – 1915 positive, 2291 negative[4];

[4] See: `http://www.wjh.harvard.edu/~inquirer/`

- MPQA – 6885 words annotated for intensity[5];
- LIWC – 2300 words in 70 classes[6];
- Opinion Lexicon – Bing Liu, 6786 words[7];
- SentiWodNet – WordNet synsets, degree of objectiveness, positivity and negativity[8].

Table 2.4: Disagreement levels of sentiment lexicons [9].

	MPQA	Opinion Lexicon	Inquirer	SentiWordNet	LIWC
MPQA	–	33/5402 (0.6%)	49/2867 (2%)	1127/4214 (27%)	12/363 (3%)
Opinion Lexicon		–	32/2411 (1%)	1004/3994 (25%)	9/403 (2%)
Inquirer			–	520/2306 (23%)	1/204 (0.5%)
SentiWordNet				–	174/694 (25%)
LIWC					–

Potts [9] made a comparison between existing lexicons (cf. Table 2.4). He counted the differences between the polarity of words between lexicons. With the exception of the SentiWordNet, the lexicons differ only slightly, sharing almost the same annotation of subjectivity for common words.

Lexicon induction

Sometimes there is a need for creation of new sentiment lexicon instead of using some of the general ones. For example, we want our lexicon to be specific for a chosen domain, thus we will get an advantage over out-of-domain lexicons. Size also matters – one is able to control the size of labelled sets of positive and negative words.

In general, lexicons are induced in unsupervised, or semi-supervised way. The first step is to start with a small polarity annotated seed of words. Next, seed set is enlarged iteratively. We will present three selected algorithms:

1. The first, derived by Hatzivassiloglou and McKeown [4], applied rule-based pattern. Intuition is such that words' polarity depends on the type of conjunction between them.
2. Turney's algorithm utilizes context information – that words of the same polarity are more likely to occur in the same context.
3. The last example is the algorithm of propagation of initial seed set throughout links between WordNet synsets.

Hatzivassiloglou and McKeown's algorithm

We will now look at the method proposed by Hatzivassiloglou and McKeown in 1997 [4]. The main concept is to predict polarity on the basis of the conjunction types between adjectives. It is more probable that terms conjoined by "and" have the same polarity – as for example "great and amazing" are both positive oriented words. Term of opposite polarity are much likely to be conjoined by "but" instead of "and" – like in "nice but slow" example. The algorithm has 4 steps.

First it starts from a polarity annotated seed set. Then, **orientation is propagated using the "and and but" rule** within the Wall Street Journal corpus of 21 million words.

- "and" – same polarity

[5] See: `http://mpqa.cs.pitt.edu/lexicons/subj_lexicon/`
[6] See: `http://www.liwc.net`
[7] See: `http://www2.cs.uic.edu/~liub/FBS/opinion-lexicon-English.rar`
[8] See: `http://sentiwordnet.isti.cnr.it`

  - "great and amazing"
  - "dirty and ugly"
  - "corrupt and brutal"
- "but" – opposite polarity
  - "beautiful but stinky"
  - "nice but slow"
  - "warm but disgusting"

The second step of the algorithm is the process of **graph building** (cf. Table 2.5). All conjoined pairs of adjectives within the corpus are counted. The graph is built on the basis of those pairs and counts. Next a classifier is applied to check if two words have the same polarity (originally logistic regression was used).

Table 2.5: Illustration of graph building phase in the Hatzivassiloglou and McKeown [4] lexicon induction method.

nice	←	and	→	calm	8452
nice	←	and	→	ugly	123
fair	←	and	→	legitimate	4600
fair	←	and	→	brutal	76
fair	←	and	→	brutal	300
...	←	and	→	...	...

Clustering is the third step. It is possible to use any clustering algorithm for graph splitting into disjoint sets. **Finally one must assign labels** to those sets. The average frequencies are compared and positive label is put onto the set with higher frequencies.

Turney's algorithm

Turney [11] proposed his method for lexicon induction in 2002. It is an unsupervised algorithm using phrases as features. It first applies a POS tagger to enable rule-based feature extraction. It looks for phrases containing adjectives and adverbs. Next, it learns semantic orientation of features by measuring its similarity to either a poor word, a negative reference, or an excellent word, an obvious positive case. Finally, a text is annotated by averaging polarities of its components.

To **find a feature** in text, Turney used a set of 5 rules for selecting sensitive relevant phrases. First, all words were assigned their part of speech. The rules work on three consecutive words – but only two first word were selected as a feature. So if a first word was an adjective and the second is a noun or a plural noun, the phrase was taken into account. It also checks for pairs of adverb and verb, triples built of adverb, adjective the third word must not be a noun or plural noun and so on (cf. Table 2.6).

Table 2.6: Feature extraction in Turney's algorithm [11].

First Word	Second Word	Third Word (not extracted)
JJ	NN or NNS	anything
RB, RBR or RBS	JJ	neither NN nor NNS
JJ	JJ	neither NN nor NNS
NN or NNs	JJ	neither NN nor NNS
RB, RBR or RBS	VB, VBD, VBN or VBG	anything

The idea for **polarity assignment** was to use Pointwise Mutual Information as a measure of semantic associations between two words. Pointwise Mutual Information is defined as logarithm

Figure 2.2: Python-like pseudocode for WordNet propagation.

```
def OrientationPrediction(adjective_list, seed_list):
                while True:
                        size1 = len(seed_list) # of words in seed_list
                        OrientationSearch(adjective_list, seed_list)
                        size2 = len(seed_list) # of words in seed_list
                        if size1 == size2: break
def OrientationSearch(adjective_list, seed_list):
        for each adjective wi in adjective_list
                if (wi has synonym s in seed_list):
                        wi_s orientation= s_s orientation
                        add wi with orientation to seed_list
                elif (wi has antonym a in seed_list):
                        wi_s orient. = opposite orientation of a_s orient.
                        add wi with orientation to seed_list
```

of the ratio of the joined probability of the co-occurrence of two words to the product of their partial probability.

$$PMI(w_1, w_2) = log_2 \frac{P(w_1 \wedge w_2)}{P(w_1)P(w_2)}$$

The probability of a word w is simply its frequency within a corpus, i.e., the number of occurrences of the word w divided by the overall number of words. The joined probability is the frequency of finding two given words nearby – number of occurrences of the pair divided by the total number of words squared. Thus, PMI for words may be rewritten using only the number of occurrences – all the normalizers cancel off.

The final step of the Turney's algorithm is to set the outcome **orientation of a phrase** and the whole text. It is done by counting the difference between semantic associations of the phrase with obvious positive and negative words. Thus, semantic orientation of a phrase is a difference between PMI of the phrase itself and the word "excellent", and PMI of the phrase and the word "poor". "Excellent" and "poor" come from the review rating system (1 star denotes a poor review, 5 stars – an excellent one). Polarity of the whole text was simply calculated as a sum of semantic orientations. If the outcome is higher than 0, the text is taken as positive, otherwise as negative.

WordNet propagation algorithm

The last example of lexicon induction is how to use WordNet to predict polarity. It is an algorithm from the work of Hu and Liu [5] from 2004. Algorithm 2.2 presents the pseudocode. The algorithm scans WordNet and a seed list for every given adjective to assign its orientation. First, it looks for synonyms in WordNet, and if one is found the adjective polarity is set with the polarity of its synonym and the adjective itself is put into a seed list. Analogously if no synonyms are found, the procedure looks for antonyms.

2.2 Data stream mining

2.2.1 Why data streams?

Usually, we have random access to the entire data set on which we're working. In this part, we will analyse a different scenario: data on which we will work will not be available to us via a random-access device like a disk drive, or a set of files on a remote server, or an API which lets us issue queries on a database.

Instead, let's assume that we have a possibility to read data as they appear is a stream. For readers familiar with UNIX or GNU/Linux, think of data stream as UNIX pipes. You can read the current value from standard input, but you cannot request a past item or fast-forward to an item upstream.

A natural thing to say at this stage would be: let's store all the contents of a stream to a file system, or to a database, and let's analyse these as we usually do. Whenever we can afford storing all the contents of a data stream and working on that – we should most probably do it.

However, sometimes we encounter data streams of such a great volume that it is simply not feasible to store everything. Instead, we have to devise algorithms that will be able to process data as they arrive. We do have (usually) an auxiliary storage, like a cache, a lookup table, etc. Sometimes we may have a lot of space for such a storage. However, we cannot store the full stream, so we have to choose carefully what kind of information we want to store.

2.2.2 Examples of data streams

In order to appreciate the need to devise algorithms for processing data streams, let us take a look at three motivational examples. Each of the three data streams presented below generates terabytes of data per day.

Twitter

First example is Twitter. As we know from the first module, Twitter offers a streaming API which allows us to process tweets in real-time, as they appear. The full stream of tweets is called "Firehose" and it is really massive. In the second half of 2013, there were approximately 500 million tweets sent each day, so on average between 5,000-6,000 tweets per second[9]. But of course, the rate is not constant and the highest peaks were at over 100,000 tweets per second. Given rich metadata of each tweet, which can be in kilobytes, Twitter generates roughly (order of magnitude) 1 TB of data per day.

It is technically possible, at least for larger companies, to store all that data, but for many applications that simply isn't necessary. What we usually want instead are algorithms that are crafted to work on data streams, for example online machine learning algorithms.

Sensors

This example, and a lot of further material, is taken from an excellent book "Mining Massive Datasets" by Anand Rajaraman, Jure Leskovec and Jeffrey Ullman [10]. The latest copy of the book can be downloaded free of charge from Ullman's website[10].

Image a sensor on a buoy, let's say it is a GPS unit that measures surface height every $\frac{1}{10}$th of a second. Such a measurement can conveniently be stored in a 4-byte floating point number, so it generates about 3.5 MB of data per day. However, imagine not one, but a million of such buoys spread across all the world's oceans. Now, we are processing 3.5 TB of new data daily!

[9] See: `https://blog.twitter.com/2013/new-tweets-per-second-record-and-how`
[10] See: `http://infolab.stanford.edu/~ullman/mmds.html`

Market

The third example that I want to show you is market data. In 2013, between 50-100 billion USD were traded daily on US stock markets alone[11]. Roughly 2-6 billion orders were submitted daily, 1% of which was actually executed. Because of this, market data analytics systems have to be able to process 1 TB of data daily. Not only the volume is challenging: these data streams have to be processed extremely fast. The fastest systems make decisions in 15 microseconds so they are using the fastest hardware available, processing is done in the CPU cache, or FPGAs and GPUs.

So, we have seen three examples of data sources, each generating about a TB of data per day.
http://www.sigmod.org/2013/keynote1_slides.pdf

2.2.3 Sampling

The first technique that you may want to employ to manage data volume is sampling. Let's say we want a fraction of results, for example 1% of the stream. We can have a counter and pick every 100th item in a stream. Or use a pseudorandom number generator, draw an integer number at random from the range 0-99 before processing each item. If we get a 0, we process the item, otherwise we skip it and move to the next one. Algorithm 1 presents a pseudocode for such processing. What are the problems with it? What could possibly go wrong?

Algorithm 1 Naïve sampling

```
1: for item ∈ stream do
2:   if random(100) = 0 then
3:     process(item)
4:   end if
5: end for
```

Let's say we want to estimate the frequency of duplicates in our data stream. For simplicity, let's assume that no item in the stream occurs more than twice. The items could be, for example, e-mail addresses. Our local storage allows as to store information about 1/100th of all the items, so let's use naïve sampling and randomly choose items with probability 1 in a 100. Actually, it would be computationally less expensive to pick every 100th element, but that is not relevant in our example. Algorithm 2 attempts (and fails!) to estimate the fraction of duplicates.

Algorithm 2 Estimating duplicates using naïve sampling (yields incorrect results!)

```
1: (S, dup, all) ← (∅, 0, 0)
2: for item ∈ stream do
3:   if random(100) = 0 then
4:     if item ∈ S then
5:       dup ← dup + 1
6:     end if
7:     all ← all + 1
8:     S ← S ∪ {item}
9:   end if
10: end for
11: return dup/all
```

As you can see, the algorithm maintains a set S of seen items. Whenever an item is randomly chosen for inspection, it is checked if S already contains the item, and if so, the *dup* counter is increased. In any case, the *all* counter is increased.

What is the problem with the algorithm? The fraction returned by the algorithm will be almost certainly very far from the true one! The problem is that the downsizing is likely to break

[11] See: `http://www.sigmod.org/2013/keynote_1.shtml`

duplicates—"break" in the sense that at most one element of the pair will survive downsampling. Let's estimate: a pair will survive downsampling with probability in the order of 1 in 10,000. On the other hand, a singleton will survive downsampling with probability 1 in 100. Even worse, most of the broken duplicates will now be counted as singletons. To sum up, Algorithm 2 will seriously underestimate the fraction of duplicates. This, of course, is only an informal reasoning. One of the exercises for this module asks for a more rigorous analysis.

To solve the problem, we can, instead of using plain pseudorandom number generator, calculate a hash function of each element. Next, we can calculate the value of hash modulo 100, and if that value is 0, process the item. Algorithm 3 presents a pseudocode.

Algorithm 3 Hash-based sampling

```
for item ∈ stream do
    if hash(item) mod 100 = 0 then
        process(item)
    end if
end for
```

The difference is that now it is not possible that a duplicate pair will be broken: either both items are selected, or none is. It is still a fairly pseudorandom decision whether a given item is selected for further processing, but the same decision will be made for its counterpart (if there is one).

2.2.4 Filtering

Let's now investigate another interesting use of hash functions. Let's say we have a large set of items S, that we want to pick from an even larger, potentially infinite, data stream.

We are willing to sacrifice quality in order to gain speed. So we are looking for a solution that can occasionally let through items not in S, so-called false positives, but we absolutely cannot miss anything that is in S. We're assuming that we will have a chance to catch the false positives later on, using more time-consuming techniques.

We will take a look at an interesting concept called Bloom filter, invented in the 1970s, which is very good for such pre-filtering. It has found applications in database systems and in data stream processing.

Algorithm 4 Initialization of a Bloom filter

```
for i ∈ {1, ..., n} do
    B[i] ← 0
end for
for s ∈ S do
    for i ∈ {1, ..., k} do
        B[h_i(s)] ← 1
    end for
end for
```

Bloom filter has a large bit array (which can be in millions or billions of cells), and a number of independent hash functions. Each hash functions maps a value to exactly one of the bits in the array. The bit array is initially zeroed. For each value in the set S, we we calculate all the hash functions on it and set the corresponding bits to one. Algorithm 4 presents the initialization of the Bloom filter.

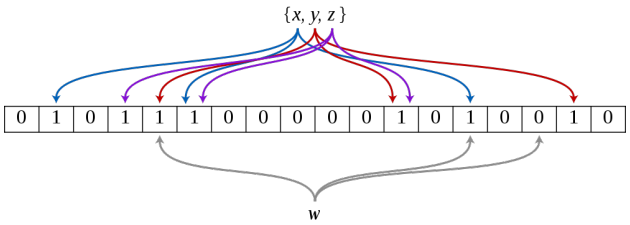

Figure 2.3: Example of a Bloom filter (image in the public domain).

Let's take a look at an example in Figure 2.3 illustrating the process of initializing and querying a Bloom filter. The bit array in the figure is of size 18, and there are 3 hash functions. We also have three values which we want to filter: x, y, and z. For each of the values, and for each of the hash functions, we set the resulting bits to 1. Note that due to hash collisions, only 7 (instead of 9) bits are set, but collisions are natural and are to be expected.

Algorithm 5 Querying a Bloom filter

```
1: for item ∈ stream do
2:   if ∀_i B[h_i(item)] = 1 then
3:     process(item)
4:   end if
5: end for
```

Let's see how we can use this array to filter values. Continuing the example on Figure 2.3, let's say that an item w appears in the stream. If w were in the set S, all of the hash functions would point to cells in the array containing 1s. It is not the case, thus, we are certain that w is not in S. A pseudocode for querying a Bloom filter is presented in Algorithm 5.

The converse is not true, however: if the item were not in the set S, it could still happen, although with low probability, that all the hash functions point to ones, and we would get a false positive result. Probability p of passing through a false positive is approximately equal to: $p = (1 - e^{-km/n})^k$. Table 2.7 presents values of p for various values of the number of items m, bits in the array n, and number of hash functions k. The numbers suggest that for $\frac{m}{n} = \frac{1}{8}$, the most satisfying probability is achieved for $k = 6$ hash functions. Indeed, this is the case. In general, the optimum number of hash functions is: $\frac{n}{m} \log 2$.

So, for the best results, choose the largest bit array that you can afford. and choose the number of hash functions according to the formula on the bottom of the slide!

Table 2.7: Probability p of false positive answer from a Bloom filter as a function of the number of items m, bits in the array n, and number of hash functions k.

m	n	k	p
1,000,000	8,000,000	1	0.117503
1,000,000	8,000,000	2	0.048929
1,000,000	8,000,000	3	0.030579
1,000,000	8,000,000	4	0.023969
1,000,000	8,000,000	5	0.021679
1,000,000	8,000,000	6	0.021577
1,000,000	8,000,000	7	0.022930

2.2.5 Online machine learning

One of the important tasks in machine learning is so-called binary classification problem. We have two classes, let's denote them as 0 and 1. We want to construct a so-called binary classifier, which

will take a vector on input (so-called feature vector), and decide to which class the vector belongs. Our goal is, therefore, to construct a function (called binary classifier) $f : \mathbb{R}^n \to \{0, 1\}$, which takes a feature vector (or observation) $\mathbf{x} \in \mathbb{R}^n$ and assigns it to a **class** (0 or 1). We are given a training set of m observations ($\mathbf{X} \in \mathbb{R}^{m \times n}$) and corresponding classes to which they belong ($\mathbf{y} \in \{0, 1\}^m$).

E-mail spam detection is an example of a binary classification problem. Anti-spam software takes an e-mail, looks at various features of the e-mail (how long is it, how many suspicious words does it have, to whom is it addressed, does it have an attachment, etc.) and based on these features it classifies the e-mail as either spam, or not spam. For an e-mail spam classifier, a training set consists of e-mails (more precisely, features of the e-mails) and information whether each e-mail is or is not spam.

For most machine learning algorithms, the training phase is long, computationally intensive, and what's crucial in this module, requires random access to the training set. However, there are two popular algorithms, one called perceptron and the other called winnow, which require only one pass over a sequence of observations, and therefore are suitable for training on a data stream. Let us see how perceptron can be trained and used on a data stream.

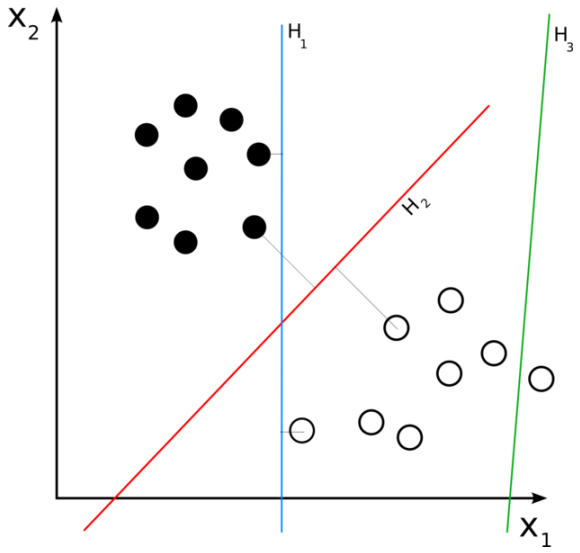

Figure 2.4: Linear classifiers, like perceptron, attempt to find a hyperplane (here, in 2D, hyperplanes are lines) that separates the feature space into two classes (image in the public domain). Black circles are items belonging to one class, white circles are items belonging to the other class. Hyperplanes H_1 and H_2 are both correctly separating the sets of observations, while H_3 is not.

Rosenblatt's perceptron [3], proposed in 1950s, is an algorithm finding a hyperplane separating two classes of observations. If a feature vector is composed from n real numbers, the space in which we are operating is n-dimensional. For simplicity, in Figure 2.4 we have a 2-dimensional space. Each point is one observation, and color encodes the class. Our goal is to find a line separating the two classes. Once we know the line, we can classify new observations: we will say that anything on one side of the line belongs to one class, and anything on the other side belongs to the other class.

The perceptron works as follows (see Algorithm 6): for each new observation it first tries to classify it using current model, parametrized by β_0 and β. At all times, the classifier is[12]: $f(\mathbf{x}) = [\beta_0 + \beta \cdot \mathbf{x} \geq 0]$. Next, the classification result is compared with the true class. If we made a mistake, that is, if we misclassified an observation, we're adjusting the betas a little, so we are moving the hyperplane a bit. The amount by which we move the hyperplane is controlled by the

[12] Here and in Algorithm 6 we are using so-called Iverson bracket, defined as: $[true] = 1$, and $[false] = 0$.

learning rate ρ.

Algorithm 6 Online machine learning using Rosenblatt's perceptron

```
1: β_0 ← 0
2: β ← [0, 0, ..., 0]
3: for (x, y) ∈ stream do
4:    y' ← [β_0 + β · x ≥ 0] {Classification of observation x (note: Iverson bracket)}
5:    if y' ≠ y then
6:       β_0 ← β_0 + ρ(y − y')
7:       β ← β + ρ(y − y')x
8:    end if
9: end for
```

The nice property of this algorithm is that it can work on data streams. At any given time, the beta parameters represent the current hyperplane, which can be used for classification. Any misclassifications on our part give us an opportunity to improve our hyperplane.

2.2.6 Concept drift

Concept drift refers to changes in statistical properties of analysed data over time. For example, changes in socio-economic environment may impact real estate valuation, changes in customer preferences may impact usefulness of movie recommendations, while novel techniques of fast-adapting attackers may render intrusion detection systems useless.

For example, let's say we are building a regression model that is supposed to estimate the value of a real estate, taking into account all the information you can get about the property itself and its neighbourhood, such as, for example, crime rate or public transport services. If we trained the model on data from, say, ten years ago, it is likely to be outdated and give poor results, as the socio-economic environment has probably changed in the meantime, and some preferences, like aversion to crime, or public transport usage patterns may have drifted.

Therefore, to sum up, we should keep in mind that over longer time periods, data streams may yield data of different characteristics.

2.3 Exercises

2.1) You are analysing a stream which contains s singletons (items occurring once) and d pairs of items, in some random order. You are sampling the stream using Algorithm 2, which uses a pseudorandom number generator and takes each item for further processing with probability $\frac{1}{100}$. What is the expected value of the fraction returned by the algorithm, as a function of s and d? Hint: follow the analysis in Section 4.2.1 of [10].

2.2) Implement a naive Bayes classifier:

- download from any source a small collection of documents annotated by several classes/topics;
- split input set into training and test subsets;
- implement a naïve Bayes classifier – do not use existing implementations;
- check accuracy of predictions on a test set;
- report results in the form of working, roughly documented code with the input set;
- you are free to choose your favourite programming language.

Bibliography

[1] Johan Bollen, Huina Mao, and Xiao-Jun Zeng. Twitter mood predicts the stock market. *CoRR*, abs/1010.3003, 2010.

[2] Sanjiv Das and Mike Chen. Yahoo! for Amazon: Sentiment parsing from small talk on the web. In *EFA 2001 Barcelona Meetings*, 2001.

[3] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., 2001.

[4] Vasileios Hatzivassiloglou and Kathleen R. McKeown. Predicting the semantic orientation of adjectives. In *Proceedings of the 8th Conference on European Chapter of the Association for Computational Linguistics*, pages 174–181. Association for Computational Linguistics, 1997.

[5] Minqing Hu and Bing Liu. Mining and summarizing customer reviews. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 168–177. ACM, 2004.

[6] Bing Liu. *Sentiment Analysis and Opinion Mining*. Morgan and Claypool, 2012.

[7] Brendan O'Connor, Ramnath Balasubramanyan, Bryan R. Routledge, and Noah A. Smith. From tweets to polls: Linking text sentiment to public opinion time series. In William W. Cohen and Samuel Gosling, editors, *Proceedings of the International AAAI Conference on Weblogs and Social Media*. The AAAI Press, 2010.

[8] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. Thumbs up?: Sentiment classification using machine learning techniques. In *Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing – Volume 10*, pages 79–86. Association for Computational Linguistics, 2002.

[9] Christopher Potts. Sentiment symposium tutorial. `http://sentiment.christopherpotts.net/`, 2011.

[10] Anand Rajaraman, Jure Leskovec, and Jeffrey David Ullman. *Mining of Massive Datasets*. 2013.

[11] Peter D. Turney. Thumbs up or thumbs down?: Semantic orientation applied to unsupervised classification of reviews. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 417–424. Association for Computational Linguistics, 2002.