Part III
LANGUAGE

•

The phonetician whispered
expressions of affection
honeyed words
into her ear:

“ i n c o n s e q u e n t i a l i t h o g r a p h e m e ”

– FLOWEREWOLF, Language (edited)

6 Computational linguistics

Natural language is not unlike playing with toy blocks. In Lego for example, there is a finite set of different blocks to combine. In language, there is a finite alphabet of characters. Characters (or phonemes in spoken language) can be combined into morphemes and morphemes make up words, the basic units that carry meaning. Words make up sentences that express a statement, a question, a command or a suggestion. Just as one toy block may fit on some blocks but not on others, words in a sentence can be described by a set of rules called a grammar (Chomsky, 1957). Sentences make up a text used to communicate an idea. Arguably, language is one of our most creative spontaneous inventions. It is recombinable, flexible and adaptive. Numerous variations have evolved and still are evolving: English, German and Dutch (Germanic languages), Spanish, Portuguese and French (Romance languages), Arabic, Chinese, Hebrew, Hindi, Russian, Swahili, and so on. Some languages gradually die out as speakers shift to another language, or as the language splits into daughter languages (e.g., Latin → Romance languages). Many derivatives have been developed in the form of formal languages, programming languages and constructed languages (e.g., Esperanto).

In the human brain, language is processed in different areas, most notably Wernicke's area and Broca's area (Geschwind, 1970) in the cerebral cortex. Language is a window to the mind (Pinker, 2007). Since each individual mind is different, language (metaphor in particular) underpins our ability to communicate and empathize with others, or to deceive them. A brain imaging study by Bedny, Pascual-Leone & Saxe (2009) on blind adults shows that the way in which humans reason about other humans does not rely on visual observation. Instead, research by Pyers & Senghas (2009) shows that the capacity to understand the beliefs or feelings of others relies on language over and above social experience.

Computational linguistics involves the theoretical study of language as well as the development of computer algorithms to process language, and ultimately to extract meaning and understanding. Figure 25 shows SCANNING–PARSING–UNDERSTANDING (Lechat & De Smedt, 2010), a visual metaphor that represents some chosen aspects of computational linguistics. Parts of it were created in NODEBOX.

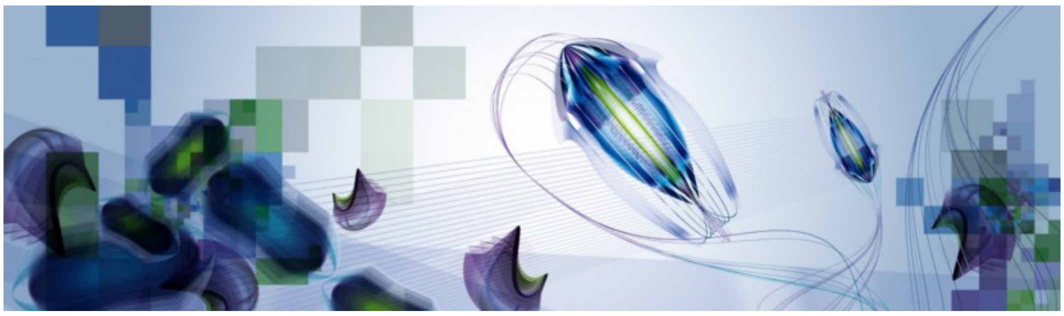


Figure 25.1: SCANNING: raw data is represented as a jumble of cell-like elements. The flowers in the center represent an algorithm that is scanning the data, identifying the words and passing them on one by one.

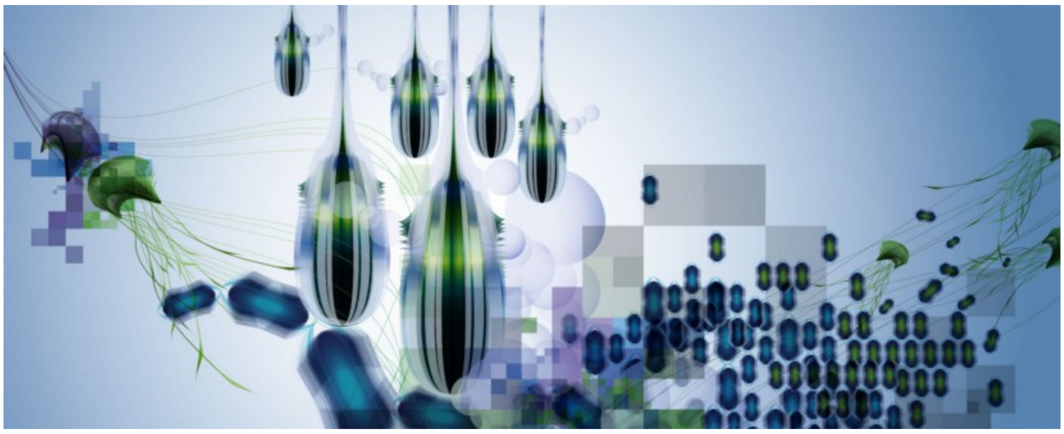


Figure 25.2: PARSING: the central cluster of shapes represents a parser. It appears to grind down on the cells, after which they are organized in a loose grid and assigned different colors, representing different word types.

Figure 25.3: UNDERSTANDING: the cells become more complex in form as meaning is mined from them.

AMBIGUITY

Words can express multiple concepts depending on the context in which they are used. For example, “dash” can mean “crash”, “crush”, “rush”, “panache” or “–” depending on how it is used. This kind of ambiguity does not seem very efficient but in fact it is: it makes a language more compact and allows us to communicate faster (Piantadosi, Tily & Gibson, 2012). On the other hand, different words can also express almost the exact same concept. For example, the verbs “rush”, “hurry”, “scurry”, “run” and “gallop” are all synonymous. This allows us to communicate more subtly, accurately or expressively.

PARTS OF SPEECH

Words fall into different categories. Nouns correspond to persons, objects or concepts. They can function as the subject of a sentence or the object of a verb or a preposition. Verbs convey actions and events. In many languages, verbs are inflected to encode tense or gender (e.g., dash → dashed). Adjectives qualify nouns. They correspond to properties such as size, shape, color, age, quantity and quality (e.g., dashing). Adverbs qualify adjectives and verbs (e.g., very dashing). Pronouns substitute previously mentioned or implicit nouns (e.g., it, he, his). Prepositions indicate a relation with the preceding or successive words (e.g., of, to, in). Conjunctions join two words, phrases or sentences (e.g., and, or). Nouns, verbs, adjectives, adverbs, pronouns, prepositions and conjunctions are the basic word classes or parts of speech in English, together with determiners (e.g., the) and interjections (e.g., uh). Word classes such as nouns, verbs and adjectives are open classes. They are constantly updated with new words borrowed from other languages or coined to describe a new idea (e.g., googling). Word classes such as pronouns, prepositions, conjunctions and determiners are closed. They are also known as function words and commonly used to establish grammatical structure in a sentence. In “the slithy toves gyred and gimbled in the wabe” it is not difficult to derive some meaning even if the nouns, adjectives and verbs are nonsensical. Word suffixes offer valuable clues: `-y` for adjectives, `-ed` for verbs. But nonsensical function words make the sentence incomprehensible: “thove mighty dove geared gard trembled tine thove waves” carries no meaning.

PARTS OF SPEECH TAGS

In natural language processing, the parts of speech are usually denoted with a short tag[20]: **NN** for nouns, **VB** for verbs, **JJ** for adjectives, **RB** for adverbs, **PR** for pronouns, **IN** for prepositions, **CC** for conjunctions and **DT** for determiners (Marcus, Santorini & Marcinkiewicz, 1993). For example:

The	stuffed	bear	sported	a	dashing	cravat	with	polka	dots
DT	**JJ**	**NN**	**VB**	**DT**	**JJ**	**NN**	**IN**	**NN**	**NN**

Table 9. Common part of speech tags in an example sentence.

[20] http://www.clips.ua.ac.be/pages/penn-treebank-tagset

6.1 Natural language processing

Natural language processing or NLP (Jurafsky & Martin, 2000) is a field of AI and linguistics that overlaps with related fields such as text mining (Feldman & Sanger, 2007), text categorization (Sebastiani, 2002) and information retrieval (Manning, Raghavan & Schütze, 2009). Natural language processing is concerned with the structure of text and algorithms that extract meaningful information from text. Meaningful information includes objective facts, subjective opinions and writing style.

Objective facts

"Who did what to whom where and when?" An important NLP task is part-of-speech tagging: identifying the parts of speech of words in a sentence. The task is complicated by ambiguous words that fall into multiple categories depending on their role in the sentence, e.g., "dashing" as verb or adjective. Consecutive words can be chunked into phrases based on their part of speech. For example, "the stuffed bear" (who?) is a noun phrase: **DT** + **JJ** + **NN** = **NP**. The verbs "is napping" (did what?) constitute a verb phrase (**VP**). Phrases convey a single idea. They can consist of other phrases. "The stuffed bear in the box" breaks down into a noun phrase "the stuffed bear" and a prepositional phrase (**PP**) "in the box" (where?), which contains a noun phrase "the box". Identifying the part-of-speech tags is the first step in determining relations between the different parts of a sentence and extracting objective facts (who, what, where).

Subjective opinions

Modal verbs (e.g., can, may, must, will) convey mood and modality. Together with adverbs and negation words (e.g., not, never) they indicate uncertainty (Morante & Sporleder, 2012). For example, the sentence "I <u>wish</u> the cat <u>would</u> stop pawing the box" uses the subjunctive mood. It expresses an opinion or a wish rather than a fact. The sentence "<u>Stop</u> pawing the box<u>!</u>" uses the imperative mood. It is a command. "The cat <u>didn't</u>" is a negation, and a fact. For an example NLP system used to predict modality, see Morante, Van Asch & Daelemans (2010). Adverbs and adjectives are often used to convey positive or negative sentiment (e.g., nice, irritating). Nouns and verbs do not convey sentiment but they can be associated with a positive or negative tone (e.g., party, punishment). This is discussed further in chapter 7.

Writing style

Writing style pertains to how individuals write. It is studied as a branch of computational linguistics called stylometry. For example, research by Pennebaker (2011) shows how function words capture the author's personality and gender. Men use more determiners (a, the, that) whereas women use more pronouns (I, you, she). A person with a higher status uses less first-person singular pronouns (I, me, my). A person who is lying uses more plural pronouns (we), more generalizations (every, all) and more negative sentiment words (Newman, Pennebaker, Berry & Richards, 2003). For an example of how stylometry can be used to attribute authorship to anonymous documents, see Luyckx & Daelemans (2008).

6.2 MBSP for Python

MBSP FOR PYTHON (De Smedt, Van Asch & Daelemans, 2010) is a memory-based shallow parser for Python, based on the MBT and TIMBL memory-based learning toolkits (Daelemans, Zavrel, van der Sloot & van den Bosch, 2004). It provides functionality for tokenization, sentence splitting, part-of-speech tagging, chunking, relation finding, prepositional phrase attachment and lemmatization for English.

A parser transforms sentences into a representation called a parse tree. A parse tree describes how words are grouped into phrases, together with their relations (e.g. subject and object). Different approaches exist to building a parser. One option is to construct a grammar, a set of structural rules that describes the syntax expected to occur in the sentences of a language. For example, simple declarative sentences often have a subject-verb-object structure: "the cat inspects the box". Constituent-based grammars focus on the hierarchical phrase structure of a sentence. Dependency-based grammars focus on grammatical relations between words. A grammar can be constructed by hand by a linguist or it can be induced automatically from a treebank. A treebank is a large collection of texts where each sentence has been manually annotated with its syntax structure. Treebank annotation is time-consuming but the effort pays off: the induced grammar is based on actual language use instead of the linguist's intuition. Moreover, the induced grammar is probabilistic. It contains statistics such as how many times a sentence structure occurs in the treebank or how many times a word occurs in **NP** or **VP** phrases for example.

Shallow parser

MBSP FOR PYTHON is a shallow parser. Shallow parsing (Abney, 1991) describes a set of tasks used to retrieve some syntactic-semantic information from text in an efficient, robust way – for example parsing the parts of speech + phrases in a sentence – at the expense of ignoring detailed configurational syntactic information. Our parser uses a supervised machine learning approach that handles tokenization, sentence splitting, part-of-speech tagging, chunking, relation finding, prepositional phrase attachment and lemmatization. This is illustrated in figure 26. These steps produce a syntactic analysis detailed enough to drive practical applications such as information extraction, information retrieval, question answering and summarization, where large volumes of text (often with errors) have to be analyzed in an efficient way.

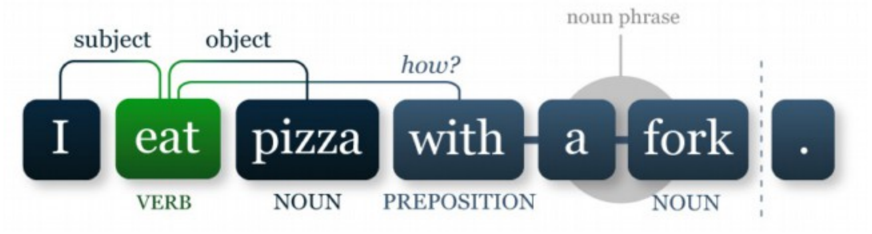


Figure 26. Shallow parsing. The verb "eat" relates sentence subject "I" to object "pizza". The verb is also related to the prepositional phrase "with a fork".

The different steps in the process are described as follows:

TOKENIZATION

First, sentence periods are determined and punctuation marks (e.g., ( )”$%) are split from words, which are separated by spaces. Regular expressions (see Friedl, 2002) are used since the tokenizer has no idea what a word (token) is at this point. Our tokenizer uses a non-trivial approach that deals with word abbreviations, unit suffixes, biomedical expressions, contractions, citations, hyphenation, parenthesis and common errors involving missing spaces and periods.

PART-OF-SPEECH TAGGING + CHUNKING

After tokenization and sentence splitting, each word in each sentence is assigned a part-of-speech tag. At the same time constituents like noun phrases (**NP**), verb phrases (**VP**) and prepositional noun phrases (**PNP**) are detected. The tagger/chunker is based on TIMBL (Daelemans et al., 2004). TIMBL implements a supervised, memory-based learning approach (Daelemans, Buchholz & Veenstra, 1999) using *k*-NN + information gain and the fast IGTREE algorithm (Daelemans, van den Bosch, Weijters, 1997). We will discuss *k*-NN and information gain in more detail shortly. For the Wall Street Journal corpus (WSJ), accuracy (F-score) is around 96.5% for part-of-speech tagging, 92.5% for **NP** chunking and 92% for **VP** chunking as reported in Daelemans & van den Bosch (2005). Across different experiments and data sets, accuracy is in the high nineties for part-of-speech tagging and in the low nineties for chunking.

RELATION FINDING

Grammatical relations (e.g. subject, object) are then predicted between the constituents. For example, in “<u>the cat</u> attacked <u>the stuffed bear</u>”, the subject of the sentence (who attacked what?) is “the cat”. The object (who or what was attacked?) is “the stuffed bear”. Accuracy (F-score) is around 77% for **SBJ** detection and 79% for **OBJ** detection as reported in Buchholz (2002).

PREPOSITIONAL PHRASE ATTACHMENT

Prepositional phrases are related to the constituent to which they belong. For example, in “the cat <u>attacked</u> the bear <u>with fervor</u>”, the “with fervor” **PNP** is related to the “attacked” **VP** and not to the “the bear” **NP**, since the expression “to attack with fervor” is statistically more probable. Accuracy (F-score) is around 83% as reported in Van Asch & Daelemans (2009).

LEMMATIZATION

Lemmatization computes the base form of each word. For example, the base form of “attacked” is “attack”. See van den Bosch & Daelemans (1999) for more information.

MBSP FOR PYTHON is a so-called lazy learner. It keeps the initial training data available, including exceptions which may sometimes be productive. This technique has been shown to achieve higher accuracy than eager (or greedy) methods for many language processing tasks (Daelemans, Buchholz & Veenstra, 1999). However, the machine learning approach is only as good as the treebank that is used for training. This means that a shallow parser trained on a collection of texts containing descriptions of toys will not perform very well on text about physics, and vice versa. Our parser is bundled with two sets of training data: general newspaper language (WSJ) and biomedical language.

Python example of use

MBSP FOR PYTHON uses a client-server architecture. This way, the training data is loaded once when the servers start. By default this happens automatically when the module is imported in Python. Otherwise, the `start()` function explicitly starts the four servers (`CHUNK`, `LEMMA`, `RELATION` and `PREPOSITION`) The module includes precompiled Mac OS X binaries for `MBT` and `TIMBL`. Instructions on how to compile binaries from source are included in the appendix. Once the servers have started, tagging jobs can be executed using the module's `parse()` function.

The `parse()` function takes a string and a number of optional parameters:

```
MBSP.parse(string,
     tokenize = True, # tokenize input string?
         tags = True, # parse part-of-speech tags?
       chunks = True, # parse chunk tags?
    relations = True, # parse chunk relations?
      anchors = True, # parse PP-attachments?
      lemmata = True, # parse word lemmata?
     encoding = 'utf-8')
```

The following example parses a given string and prints the output. The output is slash-formatted. Slashes that are part of words are encoded as `&slash;`.

```
from MBSP import parse

s = 'The cat studied the toy bear with disinterest.'
s = parse(s)
print s
```

See: http://www.clips.ua.ac.be/pages/MBSP#parser

To examine the output more closely, `pprint()` can be used. Its output is shown in table 10.

```
from MBSP import parse, pprint

s = 'The cat studied the toy bear with disinterest.'
s = parse(s)
pprint(s) # pretty-print
```

WORD	TAG	CHUNK	ROLE	ID	PNP	ANCHOR	LEMMA
The	DT	NP	SBJ	1	-	-	the
cat	NN	NP	SBJ	1	-	-	cat
studied	VBD	VP	-	1	-	A1	study
the	DT	NP	OBJ	1	-	-	the
toy	JJ	NP	OBJ	1	-	-	toy
bear	NN	NP	OBJ	1	-	-	bear
with	IN	PP	-	-	PNP	P1	with
disinterest	NN	NP	-	-	PNP	P1	disinterest
.	.	-	-	-	-	-	.

Table 10. Example output of `parse()` and `pprint()` in MBSP.

The `parsetree()` function takes the same arguments as `parse()`. It returns the parse tree as a `Text` object that contains a list of sentences. Each `Sentence` in this list contains a list of words. A `Word` object has properties `string`, `type`, `chunk` and `lemma`:

```
from MBSP import parsetree

s = 'The cat studied the toy bear with disinterest.'
t = parsetree(s)

for sentence in t.sentences:
    for word in sentence.words:
        print word.string   # u'The'
        print word.type     # u'DT'
        print word.chunk    # Chunk('The cat/NP-SBJ-1')
        print word.lemma    # u'the'
        print
```

See: http://www.clips.ua.ac.be/pages/MBSP#tree

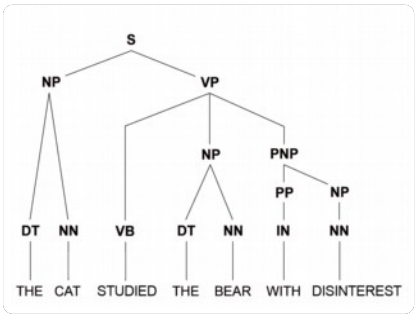


Figure 27. Parse tree.

A sentence contains a list of phrases (chunks). A `Chunk` object has properties `string`, `type`, `role`, `words`, `lemmata` and `related`:

```
for sentence in t.sentences:
    for chunk in sentence.chunks:
        print chunk.string  # u'The cat'
        print chunk.type    # u'NP'
        print chunk.role    # u'SBJ'
        print chunk.words   # [Word(u'The/DT'), Word(u'cat/NN')]
        print chunk.lemmata # [u'the', u'cat']
        print chunk.related # [Chunk('studied/VP-1'), Chunk('the toy bear/NP-OBJ-1')]
        print
```

A sentence contains a list of prepositional noun phrases. A `PNPChunk` object has properties `string`, `anchor` and `chunks`:

```
for sentence in t.sentences:
    for pnp in sentence.pnp:
        print pnp.string    # u'with disinterest'
        print pnp.anchor    # Chunk('studied/VP-1')
        print pnp.chunks    # [Chunk('with/PP'), Chunk('disinterest/NP')]
        print
```

The parsed `Text` can be exported as an XML-formatted string:

```
from MBSP import Text

f = open('output.xml', 'w')
f.write(t.xml)
f.close()
s = open('output.xml').read()
t = Text.from_xml(s)
```

On a final note, because language is ambiguous and since a probabilistic approach is used, the system is not always 100% correct. We can argue that if it was a 100% correct it would actually outperform humans.

6.3 Machine learning

Machine learning (ML) is a branch of AI concerned with the development of algorithms that learn from experience, and by similarity. Popular applications include junk email detection, the Google search engine (Brin & Page, 1998), Google's translation service, Amazon.com product recommendations and iTunes Genius music playlists. A widely quoted definition of machine learning is offered by Mitchell (1997):

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Machine learning algorithms can be supervised, unsupervised or based on reinforcement:

SUPERVISED LEARNING

Supervised learning or classification algorithms are used to recognize patterns or make predictions by inferring from training examples. By analogy, this can be compared to pointing out the names of animals in a picture book to a child. The child can infer from the pictures when it encounters a real animal.

UNSUPERVISED LEARNING

Unsupervised learning or clustering algorithms are used to recognize patterns without training examples. By analogy, this can be compared to sorting out a heap of toy blocks based on shared commonalities, for example block color or theme (e.g., separating space Lego from castle Lego).

REINFORCEMENT LEARNING

Reinforcement learning algorithms are based on reward. See Sutton & Barto (1998).

Distance in n-dimensional space

Teddy bears, rag dolls and sock monkeys are similar. They are all stuffed toys. A stuffed bear and a pair of scissors on the other hand are clearly quite different: one is a toy, the other a tool. This is clear because our mind has a deep understanding of teddy bears and scissors ever since childhood. A machine does not. A teddy bear is a meaningless string of characters: t-e-d-d-y b-e-a-r. But we can train machines, for example using a heuristic approach like PERCEPTION. This approach is interesting but also ambiguous and difficult to evaluate. To clarify this, in chapter 5 we have seen how PERCEPTION attributes similarity between Brussels and a stag or a toad. This is ambiguous, since a stag can be seen as pretty while a toad can be seen as ugly, two ends of a continuum. This leads to interpretation bias. In the artistic sense this is not problematic. Both are viable artistic solutions; PERCEPTION's audience is allowed to fill in the blanks with their own interpretation. But suppose we have a program that needs to assess whether PERCEPTION's output is pretty or ugly. It responds with "both!" or "whatever!" In the context of creativity assessment this is not a very useful program. To *evaluate* the output, we need a program that yields a well-founded, unambiguous answer.

VECTOR SPACE MODEL

Modern machine learning algorithms favor statistical approaches. A well-known and effective technique is the vector space model that represents documents (i.e., descriptions of concepts) as a matrix of n × m dimensions (Salton, Wong & Yang, 1975). A distance metric can then be used as a function of the matrix to compute similarity between documents as a value between `0.0`–`1.0`. The vector space model is fundamental to many tasks in natural language processing and machine learning, from search queries to classification and clustering (Manning et al., 2009). Consider the following documents, describing teddy bears, rag dolls and scissors:

BEAR	A teddy bear is a stuffed toy bear used to entertain children.
RAGDOLL	A rag doll is a stuffed child's toy made from cloth.
SCISSORS	Scissors have sharp steel blades to cut paper or cloth.

Figure 28. Visual representation of each document. Which is the odd one out?
Photography © Andy Norbo, Xavi Temporal. Used with permission.

Notice how **BEAR** and **RAGDOLL** both have the words "stuffed" and "toy", while **SCISSORS** does not. We can use this to our advantage. The idea is to count the different words in each document. We can also normalize each document by collapsing variant word forms (e.g., children → child) and by eliminating words that carry little meaning (e.g., a, or, to). This makes the words that matter stand out.

The normalized **BEAR** document then becomes: "~~a~~ teddy bear ~~be~~ ~~a~~ stuffed toy bear ~~use~~ ~~to~~ entertain child". The document vector is the set of distinct words (called features) and their relative frequency shown in table 11:

bear (2x)	child	entertain	stuffed	teddy	toy
`0.28`	`0.14`	`0.14`	`0.14`	`0.14`	`0.14`

Table 11. Document vector for "A teddy bear is a stuffed toy bear used to entertain children."

Relative frequencies are used so that shorter documents are not at a disadvantage. Notice how the word order is discarded. This is a simplified representation called bag-of-words. For our purposes it is relevant that **BEAR** and **RAGDOLL** both have the word "toy", not so much where the word occurs in each document.

The vector space is the n × m matrix with *n* features in *m* documents:

FEATURES	DOCUMENTS		
	BEAR	**RAGDOLL**	**SCISSORS**
bear	0.28	0	0
blade	0	0	0.14
child	0.14	0.17	0
cloth	0	0.17	0.14
cut	0	0	0.14
doll	0	0.17	0
entertain	0.14	0	0
paper	0	0	0.14
rag	0	0.17	0
scissors	0	0	0.14
sharp	0	0	0.14
steel	0	0	0.14
stuffed	0.14	0.17	0
teddy	0.14	0	0
toy	0.14	0.17	0

Table 12. Vector space with features as rows and the feature weights for each document as columns.

We can use the matrix to calculate which documents are nearer to each other. As a useful metaphor, imagine the documents as dots on a piece of paper. In other words, as a number of points with (*x*, *y*) coordinates (horizontal and vertical position) in a 2D space. Some points will be further apart than others. We can calculate the distance between each two points using Pythagoras' theorem (Euclidean distance):

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{1}$$

The formula can easily be scaled to points with (*x*, *y*, *z*) coordinates in a 3D space (and so on):

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{2}$$

COSINE SIMILARITY

Each of our document vectors has *n* features: (*x*, *y*, *z*, . . . *n*). In other words, they are points in an *n*-D space. We can calculate the Euclidean distance in *n*-D space. Documents that are nearer to each other are more similar. Another distance metric that is often used for text data is called cosine similarity. It is the Euclidean dot product divided by the product of the Euclidean norm of each vector. The cosine similarity for **BEAR–RAGDOLL** is (0.28 × 0) + (0 × 0) + . . . + (0.14 × 0.17) = 0.07 / (0.42 × 0.42) = 0.4. The cosine similarity for **BEAR–SCISSORS** is 0.0; there is no overlap for features in **BEAR** and **SCISSORS**. In this way we have learned that teddy bears and rag dolls are more similar than teddy bears and scissors.

In general, for two vectors a and b with n features in the vector space:

$$\cos(\theta) = \frac{\sum_{i=1}^{n} a_i \times b_i}{\sqrt{\sum_{i=1}^{n} (a_i)^2} \times \sqrt{\sum_{i=1}^{n} (b_i)^2}} \tag{3}$$

In Python code:

```python
class Vector(dict):
    # Feature => weight map.
    pass

def norm(v):
    return sum(w*w for w in v.values()) ** 0.5

def cos(v1, v2):
    s = sum(v1.get(f, 0) * w for f, w in v2.items())
    s = s / (norm(v1) * norm(v2) or 1)
    return s # 0.0-1.0
```

For an overview of different distance metrics, consult Van Asch (2012).

Word frequency and relevance

Buying your toddler a dozen identical teddy bears serves no purpose. Which one to hug? It is more engaging if one bear stands out with a red coat and another stands out with a dashing cravat. In a vector space, we also want each document to stand out so we can more easily compare them. Imagine we have a vector space with many documents (e.g., a thousand different toys) and many features (e.g., the Wikipedia article for each toy). Some words will recur often. For the Wikipedia "teddy bear" article, frequent words include "bear" (100×), "teddy" (80×), "toy" (10×) and "stuffed" (5×). Frequent words have a higher impact on the distance metric. But it is likely that the word "toy" is prevalent in *every* document. It is true of course that all documents in this vector space are somewhat similar, since they are all about toys. But we are interested in what sets them apart; the words that are frequent in one document and rare in others.

TF × IDF

The tf × idf weight or term frequency–inverse document frequency (Salton & Buckley, 1988) is a statistic that reflects the relevance of a word in *one* document. It is the term frequency, that is, the number of times a word appears in the document, divided by the document frequency, the number of times a word appears in all documents. The tf × idf weight is often used instead of the simple relative word count, which we used earlier to compare teddy bears and scissors.

For a word w and a collection of documents D:

$$\mathrm{idf}(w, D) = \log \frac{|D|}{1 + |\{d \in D : w \in d\}|} \tag{4}$$

$$\mathrm{tf} * \mathrm{idf}(w, d, D) = \mathrm{tf}(w, d) \times \mathrm{idf}(w, D) \tag{5}$$

In Python code:

```
from math import log

def tfidf(vectors=[]):
    df = {}
    for v in vectors:
        for f in v:
            df[f] = df.get(f, 0.0) + 1
    for v in vectors:
        for f in v: # Modified in-place.
            v[f] *= log(len(vectors) / df[f]) or 1
```

INFORMATION GAIN

Another, more elaborate measure for feature relevance is information gain (Kullback & Leibler, 1951). It is based on the Shannon entropy (Shannon, 1948). In information theory, high entropy means that a feature is more evenly distributed. Low entropy means that it is unevenly distributed: it stands out. For a given feature, information gain is the entropy of document probabilities (e.g., how many documents labeled **TOY** or **TOOL**), minus the probability of the feature across all documents multiplied by the *conditional* entropy of the feature's probabilities (e.g., how many times it occurs in a **TOY** document, in a **TOOL** document, etc.) In other words, information gain is a measure of the predictive probability of the given feature for certain document labels in the collection. For a detailed overview, consult Mitchell (1997).

For word w in documents D, with entropy H and d_w = weight of word w (`0` or `1`) in document d:

$$IG(D,w) = H(D) - \sum_{v \in values(w)} \frac{|\{d \in D | d_w = v\}|}{|D|} \cdot H(\{d \in D | d_w = v\}) \tag{5}$$

In Python code, for a given list of probabilities `P` (sum to one):

```
def entropy(P=[]):
    s = float(sum(P)) or 1
    return -sum(p / s * log(p / s, len(P)) for p in P if p != 0)
```

Classification

Classification (supervised machine learning) can be used to predict the label of an unlabeled document based on manually labeled training examples. Well-known algorithms include Naive Bayes, *k*-NN and SVM. Naive Bayes is a classic example based on Bayes' theorem of posterior probability (Lewis, 1998). SVM or Support Vector Machine uses hyperplanes to separate a high-dimensional vector space (Cortes & Vapnik, 1995). Some methods such as Random Forests use an ensemble of multiple classifiers (Breiman, 2001).

KNN

The *k*-nearest neighbor algorithm (Fix & Hodges, 1951) is a lazy learning algorithm that classifies unlabeled documents based on the nearest training examples in the vector space. It computes the cosine similarity for a given unlabeled document to all of the labeled documents in the vector space. It then yields the label from the *k* nearest documents using a majority vote.

Consider the following training documents:

TOY	A teddy bear is a stuffed toy bear used to entertain children.
TOY	A rag doll is a stuffed child's toy made from cloth.
TOOL	A bread knife has a jagged steel blade for cutting bread.
TOOL	Scissors have sharp steel blades to cut paper or cloth.

Suppose we now want to learn the label of the unlabeled document: "A needle is steel pin for sewing cloth." Is it a toy or a tool? The *k*-NN algorithm (*k*=`3`) will classify it as a tool. It yields the following cosine similarity values on the normalized training examples: **TOY** `0.0`, **TOY** `0.18`, **TOOL** `0.15`, **TOOL** `0.34`. This means that the nearest neighbor is the last document about scissors (highest cosine similarity). The document also contains the words "steel" and "cloth". The *k*=`3` nearest neighbors include two tools and one toy, so by majority vote the unlabeled document is predicted as a tool.

In Python code:

```python
class KNN:

    def __init__(self, train=[]):
        self.model = train # List of (label, vector)-tuples.

    def project(self, v1):
        return [(cos(v1, v2), label) for label, v2 in self.model]

    def classify(self, v, k=3):
        nn = {}
        for x, label in sorted(self.project(v))[-k:]:
            nn[label] = nn.get(label, 0) + x
        nn = [(x, label) for label, x in nn.items() if x > 0]
        if len(nn) > 0:
            return max(nn)[1]
```

K-FOLD CROSS VALIDATION

How accurate is our **TOY–TOOL** classifier? One straightforward way to know for sure is to test it with a set of hand-labeled documents not used for training (called a gold standard) and evaluate to what extent the predictions correspond to the annotated labels. The gold standard is comprised of documents of which we are certain that they are labeled correctly. This implies annotating or reviewing them by hand, which can be time-consuming but the technique is vacuous without a reliable performance test. For example, if we have a set of a 1,000 correctly labeled documents (so far we have 4), we could use 750 for training and 250 for testing and see how many of these 250 are correctly predicted. We could also run 10 separate tests, each using a different 90%–10% train and test set, where each document is used one time and one time only as test data, and average the results. This is called 10-fold cross validation.

PRECISION & RECALL

How accurate is our **TOY–TOOL** classifier, *really*? Suppose we test it with 90 **TOY** documents and 10 **TOOL** documents. It predicts 100× **TOY** so accuracy is 90% = 90/100 correct predictions. But the estimate can be misleading because the classifier might *always* predict **TOY**, in which case its actual recognition rate for tools is 0%. Your toddler is in the kitchen happily playing with scissors. A better way is to calculate precision and recall. For two labels, called binary classification, there will be a number of true positives (e.g., toys classified as toys), true negatives (tools classified as tools), false positives (tools classified as toys) and false negatives (toys classified as tools). The distinction between *tp*, *tn*, *fp* and *fn* is called the confusion matrix.

Precision, recall and F-measure are then defined as:

$$P = \frac{tp}{tp + fp} \qquad R = \frac{tp}{tp + fn} \qquad F = \frac{2PR}{P + R} \tag{6}$$

Recall is the percentage of toys identified by the classifier. Precision is the percentage of documents classified as **TOY** that really are toys. F-measure is the harmonic mean of precision and recall.

FEATURE SELECTION + EXTRACTION

In general, the performance of a classifier relies on the amount of training documents and the relevance of document features. For example, a vector space with too many features can degrade in performance because assumptions are made based on noisy or irrelevant features. This is called overfitting (see Schaffer, 1993). In this case we want to select a subset of the most predictive features. Techniques for automatic feature selection include information gain and genetic algorithms (see Guyon & Elisseeff, 2003).

Another technique used to reduce the vector space is feature extraction. A well-known example is latent semantic analysis (Landauer, McNamara, Dennis & Kintsch, 2007). LSA assumes that closely related words occur in similar documents. It is based on a matrix factorization called singular value decomposition (Golub & Van Loan, 1996), used to group related features into concepts. The result is a new, reduced vector space with these concepts as features, illustrated in figure 29.

Feature selection and feature extraction can also make the classifier faster, since less time is spent computing distance functions in the vector space.

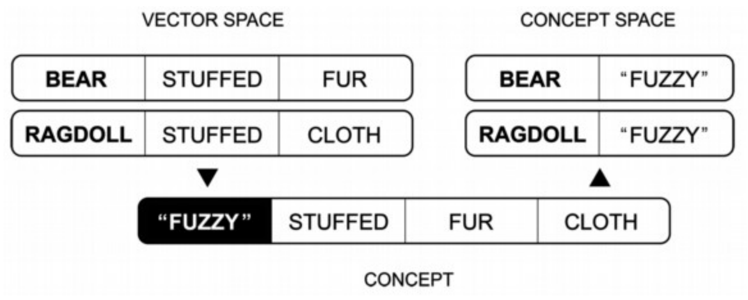


Figure 29. Illustration of latent semantic analysis.
Features are grouped into concepts, resulting in a reduced concept space.

Clustering

Clustering (unsupervised machine learning) is used to create groups of unlabeled documents that are similar to each other. For example, in the context of creativity, clustering can be employed for novelty detection by examining the smallest groups (Markou & Singh, 2003). Well-known clustering algorithms include *k*-means and hierarchical clustering.

k-MEANS

The *k*-means algorithm (Lloyd, 1982) uses a fast heuristic approach to partition a vector space into *k* clusters, so that each document belongs to the nearest cluster. The nearest cluster for a given document has the shortest distance between the document and the centroid of the cluster. The centroid is the mean vector for all documents in a cluster. Once each document is assigned to a cluster, clusters are then iteratively refined by updating their centroids and by re-assigning documents to nearer clusters. Since the initial *k* centroids are chosen randomly there is no guarantee that the algorithm will converge to a global optimum.

HIERARCHICAL CLUSTERING

The hierarchical clustering algorithm builds a hierarchy of clusters. Each document is assigned to its own cluster. Pairs of nearest clusters are then iteratively merged using a chosen distance metric. The result is a top-level cluster consisting of recursively nested clusters called a dendrogram. Distance between two clusters is measured as the distance between either the two nearest documents in the two clusters (Sibson, 1973) or the two most distant (Defays, 1977). The latter approach avoids chaining, where documents in a cluster are connected by a chain of other documents but very distant from each other.

With a basic understanding of NLP and ML, we can now turn our attention to PATTERN, a Python package containing tools for natural language processing (similar to MBSP) and the machine learning techniques discussed above: the cosine similarity distance metric, tf × idf, information gain, *k*-NN, *k*-means and hierarchical clustering.

6.4 Pattern for Python

The World Wide Web is an immense collection of linguistic information that has in the last decade gathered attention as a valuable resource for tasks such as information extraction, machine translation, opinion mining and trend detection, that is, Web as Corpus (Kilgarriff & Grefenstette, 2003). This use of the WWW poses a challenge, since the Web is interspersed with code (HTML markup) and lacks metadata (language identification, part-of-speech tags, semantic labels).

PATTERN (De Smedt & Daelemans, 2012) is a Python package for web data mining, natural language processing, machine learning and network analysis, with a focus on ease-of-use. It offers a mash-up of tools often used when harnessing the Web as a corpus. This usually requires several independent toolkits chained together. Several such toolkits with a user interface exist in the scientific community, for example ORANGE (Demšar, Zupan, Leban & Curk, 2004) for machine learning and GEPHI (Bastian, Heymann & Jacomy, 2009) for graph visualization. By contrast, PATTERN is more related to toolkits such as SCRAPY[21], NLTK (Bird, Klein & Loper, 2009), PYBRAIN (Schaul, Bayer, Wierstra, Sun, Felder et al., 2010) and NETWORKX (Hagberg, Schult & Swart, 2008) in that it is geared towards integration in the user's own programs.

PATTERN aims to be useful to both a scientific and a non-scientific audience. The syntax is straightforward. Function names and parameters were chosen to make the commands self-explanatory. The documentation[22] assumes no prior knowledge. The package is bundled with in-depth examples and unit tests, including a set of corpora for testing accuracy such as POLARITY DATASET 2.0 (Pang & Lee, 2004). The source code is hosted online on GitHub[23] and released under a BSD license. It can be incorporated into proprietary products or used in combination with other open source packages. We believe that the package is valuable as a learning environment for students, as a rapid development framework for web developers, and in research projects with a short development cycle.

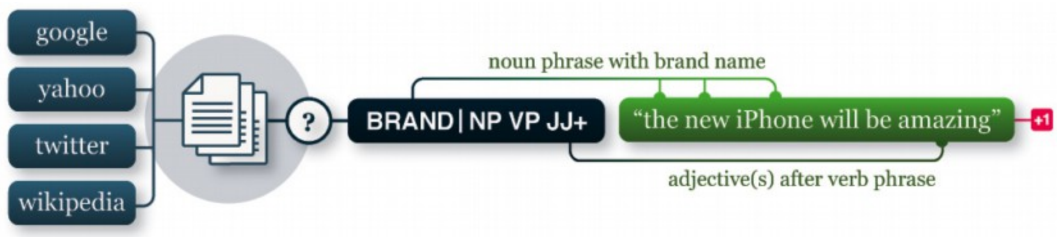

google
yahoo
twitter
wikipedia
?
noun phrase with brand name
BRAND | NP VP JJ+
"the new iPhone will be amazing"
+1
adjective(s) after verb phrase

Figure 30. Example workflow. Text is mined from the web and searched by syntax and semantics. Sentiment analysis (positive/negative) is performed on matching phrases.

[21] http://scrapy.org/
[22] http://www.clips.ua.ac.be/pages/pattern
[23] https://github.com/clips/pattern

PATTERN is organized in separate modules that can be combined. For example, text mined from Wikipedia (module pattern.web) can be parsed for part-of-speech tags (pattern.en) and queried for specific patterns (pattern.search) that can be used as training examples for a classifier (pattern.vector). This is illustrated in figure 30. Following is a discussion of the functionality in the different modules.

pattern.web

The pattern.web module contains tools for web mining, using a download mechanism that supports caching, proxy servers, asynchronous requests and redirection. It has a `SearchEngine` class that provides a uniform API to multiple web services such as Google, Bing, Yahoo!, Twitter, Facebook, Wikipedia, Flickr, and news feeds using FEEDPARSER[24]. The module includes a HTML parser based on BEAUTIFULSOUP[25], a PDF parser based on PDFMINER[26], a web crawler and a webmail interface. The following example executes a Google search query. The `Google` class is a subclass of `SearchEngine`. The `SearchEngine.search()` method returns a list of `Result` objects for the given query. Note the `license` parameter in the constructor. Some web services (i.e., Google, Bing, Yahoo) use a paid model requiring a license key.

```
from pattern.web import Google, plaintext

q = 'wind-up mouse'
G = Google(license=None)
for page in range(1, 3):
    for result in G.search(q, start=page, cached=False):
        print result.url
        print result.title
        print plaintext(result.description)
        print
```

See: http://www.clips.ua.ac.be/pages/pattern-web#services

The `plaintext()` function decodes entities (&eacute; → é), collapses superfluous whitespace and removes HTML tags from the given string. The approach is non-trivial using an SGML parser.

The following example retrieves an article from Wikipedia. Here the `search()` method returns a single `WikipediaArticle` instead of a list of `Result` objects. With the `cached=True` parameter the results will be cached locally so they can be retrieved faster next time.

```
q = 'rubber chicken'
W = Wikipedia(language='en')
article = W.search(q, cached=True)
for section in article.sections:
    print section.title.upper()
    print section.content
    print

print article.links
```

See: http://www.clips.ua.ac.be/pages/pattern-web#wikipedia

24 http://packages.python.org/feedparser/

25 http://www.crummy.com/software/BeautifulSoup/

26 http://unixuser.org/~euske/python/pdfminer/

The module includes an HTML parser that represents HTML source code as a parse tree. The `DOM` class (Document Object Model) offers an efficient and robust way to traverse the parse tree and retrieve HTML elements by tag name, id and class. Suppose that there is a web page that displays a product with customer reviews:

```
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Wind-up mouse toy</title>
</head>
<body>
    [...]
    <div class="review">
        <h3>Pet loves this toy!</h3>
        <p>She loves the eeking sound it makes.</p>
        <span class="rating">****</span>
    </div>
    [...]
</body>
</html>
```

We want to retrieve reviews together with their star rating. In the code below, we download the HTML source of the web page, feed it to the `DOM` parser and then search the parse tree for `<div class="review">` elements. This is accomplished with the `DOM.by_class()` method, which returns a list of `Element` objects whose class attribute is “review”.

```
from pattern.web import URL, DOM, plaintext

url = URL('http://www.store.com/pets/toys/3')
src = url.download(unicode=True)

for e in DOM(src).by_class('review'):
    title, description, rating = (
        plaintext(e.by_tag('h3')[0].content),
        plaintext(e.by_tag('p')[0].content),
        plaintext(e.by_class('rating')[0].content).count('*')
    )
    print title, rating # u'Pet loves this toy', 4
```

See: http://www.clips.ua.ac.be/pages/pattern-web#DOM

We can use the reviews to construct a training corpus for sentiment analysis, for example.

pattern.db

The pattern.db module facilitates file storage of corpora. A corpus is a collection of texts or other data used for statistical analysis and hypothesis testing. A text corpus can be annotated with part-of-speech tags. For example, the Penn Treebank (Marcus et al., 1993) has over 4.5 million tagged words in English text. Some corpora can contain spam email messages, others customer product reviews. Some corpora are multilingual. Since there is no standard format for storing a corpus, many different formats are used: XML, CSV, plain text, and so on. Typical problems pertain to reading the file format of a particular corpus, dealing with Unicode special characters (e.g., ø or ñ) and storing a corpus, i.e., if a comma is used to separate different items, how do we store a comma as part of an item?

The pattern.db module has wrappers for Unicode CSV files and SQLITE and MYSQL databases. For example, the `Datasheet` class is an n × m table that imports and exports CSV files. It has functionality for selecting and sorting columns or grouping rows comparable to a MS Excel sheet.

pattern.en

The pattern.en module implements a fast, regular expressions-based tagger/chunker for English based on Brill's finite state part-of-speech tagger (Brill, 1992). Brill's algorithm automatically acquires a lexicon of known words and a set of rules for tagging unknown words from a training corpus. The tagger is extended with a tokenizer, lemmatizer and chunker. The `parse()` and `parsetree()` functions are identical to those in MBSP FOR PYTHON, so it is trivial to switch to the MBSP parser in a PATTERN workflow. The module contains functions for noun singularization and pluralization (Conway, 1998), verb conjugation, linguistic mood and modality, and sentiment analysis (De Smedt & Daelemans, 2012). It comes bundled with WORDNET and PYWORDNET[27].

Accuracy for Brill's tagger has been shown to be 95%. Although statistical taggers such as MBSP perform better, Brill's tagger is faster and smaller in storage size. We have no accuracy scores for **SBJ** and **OBJ** detection but performance is poorer than that of MBSP. Accuracy (F-score) is 96% for noun singularization and pluralization tested on CELEX (Baayen, Piepenbrock & van Rijn, 1993), 95% for verb conjugation on CELEX, 67% for mood and modality on the CONLL2010 SHARED TASK 1 Wikipedia corpus (Farkas, Vincze, Móra, Csirik & Szarvas, 2010), and 72% for sentiment analysis (discussed in chapter 7).

The following example demonstrates the `parse()` function. The output is shown in table 13.

```
from pattern.en import parse, pprint

s = 'The black cat sat on the mat.'
s = parse(s, relations=True, lemmata=True)
pprint(s)
```

See: http://www.clips.ua.ac.be/pages/pattern-en#parser

WORD	TAG	CHUNK	ROLE	ID	PNP	LEMMA
The	**DT**	**NP**	**SBJ**	1	-	the
black	**JJ**	**NP**	**SBJ**	1	-	black
cat	**NN**	**NP**	**SBJ**	1	-	cat
sat	**VBD**	**VP**	-	1	-	sit
on	**IN**	**PP**	-	-	**PNP**	on
the	**DT**	**NP**	-	-	**PNP**	the
mat	**NN**	**NP**	-	-	**PNP**	mat
.	**.**	-	-	-	-	.

Table 13. Example output of `parse()` and `pprint()` in PATTERN.

[27] http://osteele.com/projects/pywordnet

The `parsetree()` function returns a `Text` object consisting of `Sentence` objects. These provide an easy way to iterate over chunks and words in the parsed text:

```
from pattern.en import parsetree

s = 'The black cat sat on the mat.'
t = parsetree(s, relations=True, lemmata=True)
for sentence in t.sentences:
    for chunk in sentence.chunks:
        for word in chunk.words:
            print chunk.type, word.string, word.type, word.lemma
```

See: http://www.clips.ua.ac.be/pages/pattern-en#tree

When `parse()` and `parsetree()` are used with `lemmata=True`, the lemmatizer will compute the singular form of plural nouns (cats → cat) and the infinitive of conjugated verbs (sat → sit). Lemmatization can also be invoked with separate functions. This is useful to bring different word forms together when training a classifier, for example.

```
from pattern.en import singularize, pluralize
from pattern.en import conjugate

print singularize('cats')     # 'cat'
print pluralize('cat')        # 'cats'
print conjugate('sat', 'inf') # 'sit'
print conjugate('be', '1sg')  # 'am'
```

See: http://www.clips.ua.ac.be/pages/pattern-en#pluralization

The `Text` and `Sentence` objects can serve as input for other functions, such as the `search()` function in the pattern.search module, or `modality()` and `sentiment()`.

The `modality()` function returns a value between `-1.0` and `+1.0`, expressing the degree of certainty based on modal verbs and adverbs in the sentence. For example, "I wish it would stop raining" scores `-0.75` while "It will surely stop raining soon" scores `+0.75`. In Wikipedia terms, modality is sometimes referred to as *weaseling* when the impression is raised that something important is said, but what is really vague and misleading (Farkas et al., 2010). For example: "some people claim that" or "common sense dictates that".

```
from pattern.en import parsetree
from pattern.en import modality

print modality(parsetree('some people claim that')) # 0.125
```

The `sentiment()` function returns a `(polarity, subjectivity)`-tuple, where `polarity` is a value between `-1.0` and `+1.0` expressing negative vs. positive sentiment based on adjectives (e.g., great, horrible) and adverbs (e.g., really, very) in the sentence. For example, "The weather is very nice" scores `+0.64`.

```
from pattern.en import parsetree
from pattern.en import sentiment

print sentiment(parsetree('What a wonderful day!')) # (1.0, 1.0)
```

The pattern.en.wordnet submodule is an interface to the WORDNET lexical database. It is organized in `Synset` objects (synonym sets) with relations to other synsets. Example relations are **hyponymy** (bird → hen), **holonymy** (bird → flock) and **meronymy** (bird → feather). Following are two functions taken from the FLOWEREWOLF program discussed in chapter 5. The `shift()` function returns a random hyponym for a given noun. The `alliterate()` function yields a list of alliterative adjectives for a given noun.

```
from pattern.en import wordnet
from random import choice

def shift(noun):
    s = wordnet.synsets(noun)
    s = s[0]
    h = choice(s.hyponyms(recursive=True) or [s])
    return (h.synonyms[0], h.gloss)

def alliterate(noun, head=2, tail=1):
    for a in wordnet.ADJECTIVES.keys():
        if noun[:head] == a[:head] and noun[-tail:] == a[-tail:]:
            yield a

print shift('soul')
print list(alliterate('specter', head=3))

> ('poltergeist', 'a ghost that announces its presence with rapping')
> ['spectacular', 'specular']
```

See: http://www.clips.ua.ac.be/pages/pattern-en#wordnet

pattern.de

The pattern.de module contains a tagger/chunker for German (Schneider & Volk, 1998) and functions for German noun singularization and pluralization, predicative and attributive adjectives (e.g., neugierig → die neugierige Katze) and verb conjugation. Schneider & Volk report the accuracy of the tagger around 95% for 15% unknown words. Accuracy (F-score) is 84% for noun singularization, 72% for pluralization, 98% for predicative adjectives, 75% for attributive adjectives, and 87% for verb conjugation of unknown verbs. The results were tested on CELEX.

pattern.nl

The pattern.nl module contains a tagger/chunker for Dutch (Geertzen, 2010) and functions for Dutch noun singularization and pluralization, predicative and attributive adjectives (e.g., nieuwsgierig → de nieuwsgierige kat), verb conjugation and sentiment analysis. Geertzen reports the accuracy of the tagger around 92%. Accuracy (F-score) is 91% for noun singularization, 80% for pluralization, 99% for predicative and attributive adjectives, 80% for verb conjugation of unknown verbs. The results were tested on CELEX. Accuracy for sentiment analysis is 81%.

pattern.search

The pattern.search module is useful for information extraction. Information extraction (IE) is the task of obtaining structured information from unstructured data, for example to recognize the names of a person or organization in text (named entity recognition), or to identify relations such as **PERSON works-for ORGANIZATION**. Interesting examples include TEXTRUNNER (Banko, Cafarella, Soderland, Broadhead & Etzioni, 2007), which mines relations from Wikipedia, and CONCEPTNET (Liu & Singh, 2004), an automatically acquired semantic network of common sense.

The pattern.search module implements a pattern matching algorithm for *n*-grams (*n* consecutive words) using a recursive backtracking approach similar to regular expressions. Search queries can include a mixture of words, phrases, part-of-speech-tags, taxonomy terms (e.g., **PET** = dog + cat + goldfish) and control characters such as wildcards (`*`), operators (`|`), quantifiers (`?` and `+`) and groups (`{}`) to extract information from a string or a parsed `Text`.

For example, a simple "**DT NN** is an animal" pattern can be used to learn animal names. It means: a determiner (a, an, the) followed by any noun, followed by the words "is an animal".

```python
from pattern.en import parsetree
from pattern.search import search

s = 'A crocodile is an animal called a reptile.'
t = parsetree(s)
for match in search('DT {NN} is an animal', t):
    print match.group(1)
```

See: http://www.clips.ua.ac.be/pages/pattern-search

This yields a list of words matching the `{NN}` group in the pattern: `[Word('crocodile/NN')]`. Now that we know that a crocodile is an animal we can learn more about it, for example using a "**JJ+** crocodile" pattern. It means: one or more adjectives followed by the word "crocodile". It matches phrases such as "prehistoric crocodile" and "green crocodile".

A scalable approach is to group known animals in a taxonomy:

```python
from pattern.en import parsetree
from pattern.search import taxonomy, search

for animal in ('crocodile', 'snail', 'toad'):
    taxonomy.append(animal, type='animal')

s = 'Picture of a green crocodile covered in pond lily.'
t = parsetree(s)
for match in search('{JJ+} {ANIMAL}', t):
    print match.group(1), match.group(2)
```

See: http://www.clips.ua.ac.be/pages/pattern-search#taxonomy

Since `search()` is case-insensitive, any uppercase word in the pattern is taken to be a taxon. In this case, "ANIMAL" represents the **ANIMAL** category in our taxonomy, with instances "crocodile", "snail" and "toad". The pattern matches phrases such as "green crocodile" but also "squishable snail".

To deploy the learner on the Web we can combine the search patterns with a custom `Spider` from the pattern.web module. In the code below, the `Spider.visit()` method is implemented in a subclass. It is called for each web page the spider visits, by following the links in each page.

```
from pattern.web import Spider, plaintext
from pattern.en import parsetree
from pattern.search import search

class Learner(Spider):
    def visit(self, link, source=None):
        t = parsetree(plaintext(source))
        for match in search('{NN} is an animal', t):
            print match.group(1)

learner = Learner(links=['http://en.wikipedia.org/wiki/Animal'])
while True: # Endless learning.
    learner.crawl(throttle=10)
```

See: http://www.clips.ua.ac.be/pages/pattern-web#spider

pattern.vector

The pattern.vector module implements a vector space model using a `Document` and a `Corpus` class. Documents are lemmatized bag-of-words that can be grouped in a sparse corpus to compute tf × idf, distance metrics (cosine, Euclidean, Manhattan, Hamming) and dimension reduction (latent semantic analysis). The module includes a hierarchical and a *k*-means clustering algorithm, optimized with the *k*-means++ initialization algorithm (Arthur and Vassilvitskii, 2007) and the triangle inequality exploit (Elkan, 2003). A Naive Bayes, a *k*-NN and a SVM classifier using LIBSVM (Chang and Lin, 2011) are included, along with tools for feature selection (information gain) and K-fold cross validation.

The following example replicates our previous **BEAR–RAGDOLL–SCISSORS** example. Each `Document` takes a string. With `stemmer=LEMMA` lemmatization is enabled. With `stopwords=False` (default) words that carry little meaning (e.g., a, or, to) are eliminated. The `type` parameter defines the document label. The documents are bundled in a `Corpus` that computes tf (default is tf × idf) for each `Document.vector`. The `Corpus.similarity()` method returns the cosine similarity for two given documents.

```
from pattern.vector import Document, Corpus, LEMMA, TF, TFIDF

d1 = Document('A teddy bear is a stuffed toy bear used to entertain children.',
      stemmer=LEMMA, stopwords=False, type='BEAR')

d2 = Document('A rag doll is a stuffed child\'s toy made from cloth.',
      stemmer=LEMMA, stopwords=False, type='RAGDOLL')

d3 = Document('Scissors have sharp steel blades to cut paper or cloth.',
      stemmer=LEMMA, stopwords=False, type='SCISSORS')

corpus = Corpus([d1, d2, d3], weight=TF)

print corpus.similarity(d1, d2) # 0.27 bear-doll
print corpus.similarity(d1, d3) # 0.00 bear-scissors
print corpus.similarity(d2, d3) # 0.15 doll-scissors
```

See: http://www.clips.ua.ac.be/pages/pattern-vector#document

Documents can also be passed to a classifier. The `Classifier` base class has three subclasses: `Bayes`, `kNN` and `SVM`. The following example replicates our previous **TOY–TOOL** *k*-NN classifier:

```
from pattern.vector import Document, Corpus
from pattern.vector import LEMMA, TFIDF
from pattern.vector import kNN

train = (
    ('TOY',  'A teddy bear is a stuffed toy bear used to entertain children.'),
    ('TOY',  'A rag doll is a stuffed child\'s toy made from cloth.'),
    ('TOOL', 'Scissors have sharp steel blades to cut paper or cloth.'),
    ('TOOL', 'A bread knife has a jagged steel blade for cutting bread.')
)

test = 'A bread knife has a jagged steel blade for cutting bread.'

corpus = Corpus(weight=TFIDF)
for label, example in train:
    d = Document(example, stemmer=LEMMA, type=label)
    corpus.append(d)

knn = kNN()
for document in corpus:
    knn.train(document, type=document.type)

print knn.classify(test) # 'TOOL'
```

See: http://www.clips.ua.ac.be/pages/pattern-vector#classification

Finally, we chain together four PATTERN modules to train a *k*-NN classifier on adjectives mined from Twitter. We mine a 1,000 tweets with the hashtag #win or #fail (our labels), for example: "$20 tip off a sweet little old lady today #win". We parse the part-of-speech tags for each tweet, keeping adjectives. We group adjective vectors in a corpus and use it to train the classifier. It predicts "sweet" as `WIN` and "stupid" as `FAIL`. The results may vary depending on what is currently buzzing on Twitter. A real-world classifier will also need more training data and more rigorous feature selection.

```
from pattern.web    import Twitter
from pattern.en     import parsetree
from pattern.search import search
from pattern.vector import Document, Corpus, kNN

corpus = Corpus()

for i in range(1, 10):
    for tweet in Twitter().search('#win OR #fail', start=i, count=100):
        p = '#win' in tweet.description.lower() and 'WIN' or 'FAIL'
        s = tweet.description.lower()
        t = parsetree(s)
        m = search('JJ', t)
        m = [match[0].string for match in m]
        if len(m) > 0:
            corpus.append(Document(m, type=p))

classifier = kNN()
for document in corpus:
    classifier.train(document)

print classifier.classify('sweet')  # 'WIN'
print classifier.classify('stupid') # 'FAIL'
```

To apply LSA dimension reduction to a corpus before training a classifier we can do:

```
corpus = corpus.reduce(dimensions=k) # reduce k dimensions
```

To apply feature selection to a corpus before training a classifier we can do:

```
subset = corpus.feature_selection(top=100, method='infogain')
corpus = corpus.filter(features=subset)
```

pattern.graph

The pattern.graph module implements a graph data structure using `Node`, `Edge` and `Graph` classes. It contains algorithms for shortest path finding, subgraph partitioning, eigenvector centrality and betweenness centrality (Brandes, 2001). Centrality algorithms were ported from NETWORKX. For an example of use, see chapter 5. The module has a force-based layout algorithm that positions nodes in 2D space. Visualizations can be exported to HTML and manipulated in a web browser using the `canvas.js` helper module.

pattern.metrics

The pattern.metrics module contains statistical functions for data analysis, including sample mean, variance and standard deviation, histogram, moments, quantiles, boxplot, Fisher's exact test, Pearson's chi-squared test and log-likelihood ratio. Other evaluation metrics include a Python code profiler, functions for accuracy, precision, recall and F-measure, confusion matrix, inter-rater agreement (Fleiss' kappa), string similarity (Levenshtein, Dice) and string readability (Flesch).

canvas.js

The canvas.js module is a simple and robust JavaScript API for the HTML5 `<canvas>` element, which can be used to generate interactive 2D graphics in a web browser, using lines, shapes, paths, images, image filters and text. The functionality in the module closely resembles NODEBOX FOR OPENGL. It is useful to support quantitative data analysis with a graphical representation of the data (e.g., line chart, histogram, scatter plot, box plot). More recent representation techniques include network visualization and custom, interactive and/or online visualizations, in effect any representation that reduces complexity while capturing important information (Fayyad, Wierse & Grinstein, 2002). This is well within the capabilities of canvas.js.

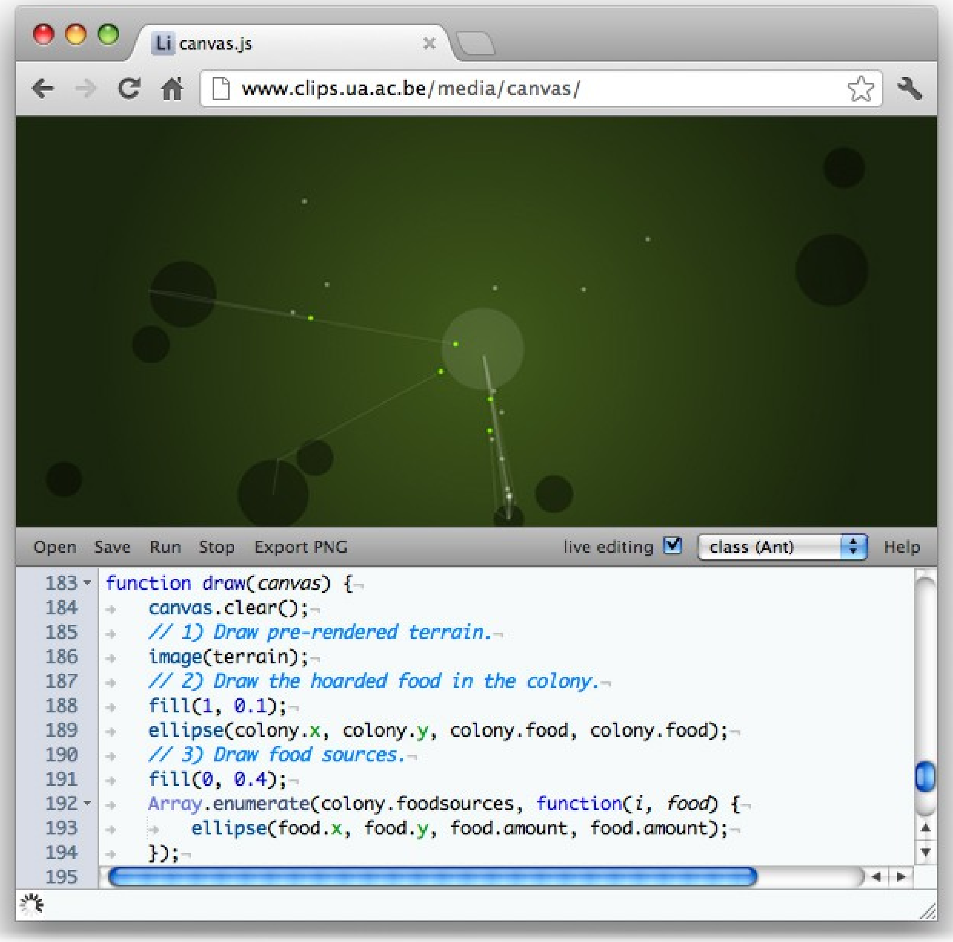
Below is a basic example that draws a rotating red square in the web browser. The source code imports the canvas.js module. Note the `<script type="text/canvas">` that defines the animation. It has a `setup()` function that will be executed once when the animation starts, and a `draw()` function that will be executed each animation frame.

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src="canvas.js"></script>
</head>
<body>
```

```
    <script type="text/canvas">
      function setup(canvas) {
          canvas.size(500, 500);
      }
      function draw(canvas) {
          canvas.clear();
          translate(250, 250);
          rotate(canvas.frame);
          rect(-150, -150, 300, 300, {fill: color(1,0,0,1)});
      }
    </script>
</body>
</html>
```

See: http://www.clips.ua.ac.be/pages/pattern-canvas

Figure 31 shows a screenshot of the online editor[28]. In live-editing mode, any modifications to the source code are instantly reflected in the animation.



```
function draw(canvas) {
    canvas.clear();
    // 1) Draw pre-rendered terrain.
    image(terrain);
    // 2) Draw the hoarded food in the colony.
    fill(1, 0.1);
    ellipse(colony.x, colony.y, colony.food, colony.food);
    // 3) Draw food sources.
    fill(0, 0.4);
    Array.enumerate(colony.foodsources, function(i, food) {
        ellipse(food.x, food.y, food.amount, food.amount);
    });
```

Figure 31. canvas.js editor. With live editing enabled, changes to the JavaScript code are instantly reflected in the animation that is currently running.

[28] http://www.clips.ua.ac.be/media/canvas

6.5 A Mad Tea-Party (creative thinking test revisited)

To conclude this chapter, we return to Guilford's Divergent Thinking test (DT) discussed in chapter 4. In a DT test, subjects are asked to come up with as many answers as possible to an open-ended problem solving task. In chapter 4 (section 4.6) we explained how we subjected 187 participants to a DT test ("How do you open a locked door if you don't have the key?"). Since the scoring process is essentially a language evaluation task, we should be able to replicate it using NLP and machine learning techniques. In other words, can we implement a system that is able to evaluate linguistic creativity? And subsequently, can we implement a system that generates answers that beat all answers given by the participants?

Creativity in the DT test is assessed as follows:

ORIGINALITY	Unusual answers across all answers score `1`, unique answers score `2`.
FLUENCY	The number of answers.
FLEXIBILITY	The number of different categories of answers.
ELABORATION	The amount of detail. For example, "break door" = `0` and "break door with a crowbar" = `1`.

In our setup from chapter 4, we took each answer with a total score greater than average + standard deviation as creative, and smaller as not creative. In the same way, we could implement a `creativity()` function that returns a score for each answer, then partition the scores into creative or not creative, and compare this outcome to the original outcome using precision and recall evaluation metrics. We obtain the best results if we slightly relax the threshold for creative answers = average + standard deviation × `0.95`.

Figure 32 shows the correlation between manual and automatic scores, where the jagged line shows the automatic scores. Following is a discussion of the automatic scoring implementation.

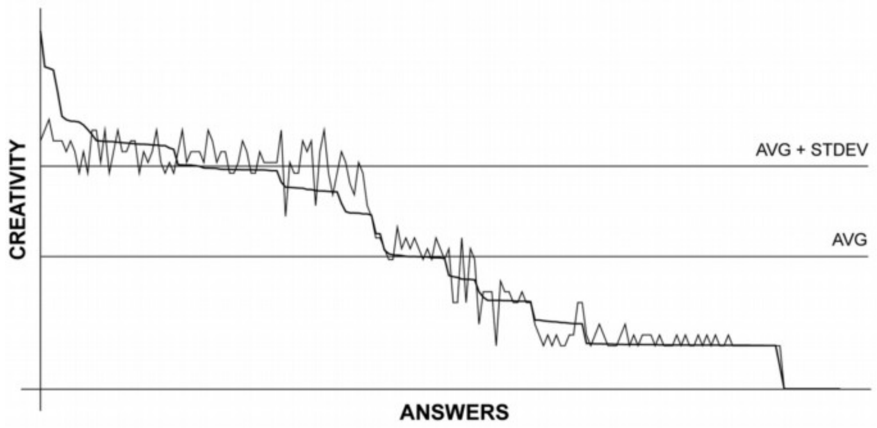


Figure 32. Correlation between the manual and automatic scores for each answer.
The jagged line shows the scores for the machine learner.

The **FLUENCY** score is simply the number of answers for each participant. The **ELABORATION** score corresponds to the number of prepositional phrases in each answer (e.g., "with a crowbar"). We can measure this by constructing a parse tree with MBSP or PATTERN:

```
from pattern.en import parsetree

def fluency(answers):
    return len(answers)

def elaboration(answers):
    return sum(min(len(parsetree(a)[0].pnp), 2) for a in answers)
```

If we only use these two scores to predict creative vs. not creative, we obtain a precision of `0.74` and a recall of `0.67` (combined F1-score of `0.70`). This is our performance baseline.

The **FLEXIBILITY** and **ORIGINALITY** scores rely on a categorization of the answer set. In chapter 4, the categorization was performed manually by two annotators. We can model this process using an unsupervised clustering analysis, where more closely related answers are assigned to the same cluster based on the words in their description. We use the hierarchical clustering algorithm in PATTERN. Assuming we have all the answers as a Python list of strings, we can transform the set into a sparse vector space where each vector is the word count of an answer:

```
from pattern.vector import Vector, count, words

answers = open('locked-door.txt').readlines()
vectors = []
for a in answers:
    v = count(words(a), stemmer='lemma')
    v = Vector(v)
    vectors.append(v)
```

Using this vector space we can then compute clusters of vectors, i.e., categories of similar answers. Since the number of clusters *k* is arbitrary, we need to perform several iterations to see what value of *k* yields the best results. In our setup, higher *k* yields better results. This is because more and smaller clusters have a smaller chance of covering multiple topics, perhaps including an original answer that should really belong to no cluster. In the same way, we are interested in "tight" clusters in which the vectors are near to each other. Loose clusters might be covering outliers. We define a "loose" cluster as a cluster with a high average distance of vectors to the center of the cluster. As it turns out, on our set of 533 answers the best results are obtained with *k*=`250` combined with a post-processing step that removes the top 50 loose clusters.

```
from pattern.vector import hierarchical, distance, centroid
from pattern.metrics import avg

def variance(cluster):
    return avg([distance(centroid(cluster), v) for v in cluster])

clusters = hierarchical(vectors, k=250, distance='cosine')
clusters = [isinstance(v, Vector) and [v] or v.flatten() for v in clusters]
clusters = sorted(clusters, key=variance)[:-50]

categories = {}
for i, cluster in enumerate(clusters):
    for v in cluster:
        categories[answers[vectors.index(v)]] = i
```

The **FLEXIBILITY** score is then defined as:

```
def flexibility(answers):
    return len(set(categories.get(a, a) for a in answers))
```

The **ORIGINALITY** score discerns between common, unusual and unique answers. We define common answers as being part of a larger cluster, whereas unusual answers (top 5%) and unique answers (top 1%) correspond to smaller (or no) clusters. Thus we need to know the relative probability of each cluster:

```
p = {}
for c in categories.values():
    p.setdefault(c, 0.0)
    p[c] += 1
s = sum(p.values())
for c in p:
    p[c] /= s
```

The **ORIGINALITY** score is then defined as:

```
def originality(answers):
    originality = 0
    for a in answers:
        if p.get(categories.get(a, a), 0) < 0.01:
            originality += 1
        if p.get(categories.get(a, a), 0) < 0.05:
            originality += 1
    return originality / (float(fluency(answers)) or 1)
```

Using the total score of `originality()`, `fluency()`, `flexibility()` and `elaboration()` for prediction, we obtain a precision of `0.72` and a recall of `0.83` (combined F1-score of `0.77`). Overall, the learning step has improved the accuracy by 7% over the baseline.

Since we can now automatically (approximately) predict creative answers, we can also generate new answers and evaluate how well they do. We can adopt many strategies to try and beat the answers given by the participants. For example, we could blend known answers into new answers, employ Web search queries, WORDNET, and so on. But the simplest approach is to exploit a weakness in the automatic scoring implementation. This weakness has two angles. \

First, as suggested by the high baseline, the scores emphasize the number of answers per participant and sentence length (i.e., the number of prepositional phrases) over originality. One can argue that the originality score is diminished by dividing it by the number of answers, but this is actually done to correct a known contamination problem in the DT test (Mouchiroud & Lubart, 2001). Second, the scores emphasize originality over appropriateness, since our system has no way to evaluate appropriateness (see chapter 4) as a factor. This implies that longer answers are more creative even if they are nonsensical.

We can therefore beat the automatic scoring process by generating long, nonsensical sentences. We will use random nouns, verbs and adjectives linked to the *door* concept retrieved from the PERCEPTION semantic network. This is done to promote an air of meaningfulness: the retrieved should in some way be semantically related to a door. However, as it turns out things quickly spiral out of control.

```
from pattern.graph.commonsense import Commonsense
from pattern.en import tag, wordnet, pluralize

from random import choice

concept = 'door'
g = Commonsense()
g = g[concept].flatten(4)

tagged = [tag(node.id)[0] for node in g]

JJ = [w for w, pos in tagged if pos == 'JJ']
NN = [w for w, pos in tagged if pos == 'NN']
VB = [w for w, pos in tagged if w.encode('utf-8') in wordnet.VERBS]

for i in range(5):
   a = "%sly %s the %s with a %s %s of %s %s." % (
        choice(JJ).capitalize(),
        choice(VB),
        concept,
        choice(JJ),
        choice(NN),
        choice(JJ),
        pluralize(choice(NN)))
   print a
```

We used some additional post-processing to polish the output: the first word is capitalized, and the last noun is pluralized using the `pluralize()` function in PATTERN. This yields the following set of nonsensical answers for example:

Question: How do you open a locked door?

“Stubbornly club the door with a graceful albatross of pungent swamps.”
“Unpleasantly court the door with a massive exhibition of busy colors.”
“Passionately pet the door with a weak obstacle of skeptical physicians.”
“Rawly storm the door with a dry business lunch of primitive crustaceans.”
“Sensationally duck the door with a fast locomotion of large grandmothers.”

Excellent, all we need now is an albatross. When we include the generator in the cluster analysis on our data and calculate its score, it beats all other scores by at least 15% since it has the maximum number of answers, each with two prepositions, and flavored with adjectives for the reader’s enjoyment. As discussed above, the high score is the result of the system's inability to discern between useful and nonsensical answers. This also suggests that the DT test may benefit from a more explicit directive for scoring appropriateness.

6.6 Discussion

In this chapter we have discussed a statistical approach used in natural language processing and machine learning called the vector space model. In this approach natural language is transformed into a matrix of numeric values. Statistical functions can then be calculated on the matrix to predict the similarity of vectors in the matrix, which represent sentences, paragraphs or text (or pixels, genes, brain activity, and so on). We have presented two Python software packages developed in the course of our work that implement this approach: MBSP FOR PYTHON and PATTERN.

MBSP FOR PYTHON is a Python implementation of the MBSP memory-based shallow parser for English, which detects word types (part-of-speech tagging), word chunks and relations between chunks in text. In terms of modeling creativity, part-of-speech tags can offer valuable linguistic information. For example, to evaluate what adjectives are commonly associated with what nouns or what words constitute x and y in "x is a new y". Such information can be used (for example) to populate a semantic network of common sense or to acquire a model for sentiment analysis.

PATTERN is a Python package for web mining, natural language processing, machine learning and graph analysis. With a combination of PATTERN and MBSP it would not be difficult to construct an endless learning PERCEPTION (chapter 5) with information extracted from actual language use instead of relying on the intuition an annotator. Using PATTERN's classification tools it is possible to replace PERCEPTION entirely with a statistical approach. Using the clustering tools it is possible to predict novelty, i.e., the instances left unclustered after the analysis. In recent work, Shamir & Tarakhovsky (2012) have used unsupervised learning to detect the artistic style of approximately 1,000 paintings, in agreement with the opinion of art historians and outperforming humans not trained in fine art assessment.

Does statistical AI model how the human mind works? It is hard to say for sure. But the state-of-the-art offers a robust approach to many practical tasks in AI and creativity. Whenever such applications yield results, the threshold of what contributes to human intelligence and creativity is raised. For example, when a chess computer beats a human we are quick to remark that it plays a boring chess or that it can't write poetry or translate Chinese. When a fine art learner reaches the level of an art historian, we respond with: "I'll wager I can think of a painting it can't handle... See! It doesn't work."

Many AI problems such as natural language understanding and machine translation are considered to be AI-complete. They may require that all other AI problems are solved as well. Many AI researchers hope that bundling the individual applications will lead to emergent strong AI one day. Some futurists like Kurzweil (2005) argue that this would require artificial consciousness. Kurzweil believes that once machines reach this stage they will be powerful enough to solve (among other) all known medical problems – but such conjectures have also been criticized.

7 Sentiment analysis

Textual information can be broadly categorized into three types: objective facts and subjective opinions (Liu, 2010) and writing style. Opinions carry people's sentiments, appraisals and feelings toward the world. Before the World Wide Web, opinions were acquired by asking friends and families, or by polls and surveys. Since then, the online availability of opinionated text has grown substantially. Sentiment analysis (or opinion mining) is a field that in its more mature work focuses on two main approaches. The first approach is based on subjectivity lexicons (Taboada, Brooks, Tofiloski, Voll & Stede, 2011), dictionaries of words associated with a positive or negative sentiment score (polarity). Such lexicons can be used to classify sentences or phrases as subjective or objective, positive or negative. The second approach is by using supervised machine learning methods (Pang & Vaithyanathan, 2002).

Resources for sentiment analysis are interesting for marketing or sociological research, for example to study customer product reviews (Pang & Vaithyanathan, 2002), public mood (Mishne & de Rijke, 2006), electronic word-of-mouth (Jansen, Zhang, Sobel & Chowdhury, 2009) and informal political discourse (Tumasjan, Sprenger, Sandner & Welpe, 2010).

7.1 A subjectivity lexicon for Dutch adjectives

In De Smedt & Daelemans (2012) we describe a subjectivity lexicon for Dutch adjectives integrated with PATTERN and compatible with CORNETTO, an extension of the Dutch WordNet (Vossen, Hofmann, de Rijke, Tjong Kim Sang & Deschacht, 2007). Esuli & Sebastiani (2006) note that adverbs and adjectives are classified more frequently as subjective (40% and 36%) than verbs (11%). In our approach we focus on adjectives. We used a crawler to extract adjectives from online Dutch book reviews and manually annotated them for polarity, subjectivity and intensity strength. We then experimented with two machine learning methods for expanding the initial lexicon, one semi-supervised and one supervised. Each of the book reviews has an accompanying, user-given "star rating" (1–5), which we used to evaluate the lexicon.

Manual annotation

Since adjectives with high subjectivity will occur more frequently in text that expresses an opinion, we collected 14,000 online Dutch book reviews in which approximately 4,200 Dutch adjective forms occurred. The texts were mined with PATTERN and part-of-speech tagged with FROG (van den Bosch, Busser, Canisius & Daelemans, 2007). We did not apply lemmatization at this stage and therefore some adjectives occur both in citation and inflected form, e.g., "goed<u>e</u>" vs. "goed" (good). The adjective frequency approximates an exponential Zipf distribution, with "goed" being the most frequent (6000+ occurrences), followed by "echt" (real, 4500+) and "heel" (very, 3500+). The top 10% constitutes roughly 90% of all occurrences. We took the top 1,100 most frequent adjectives, or all adjectives that occurred more than four times.

Seven human annotators were asked to classify each adjective in terms of positive–negative polarity and subjectivity. In Esuli & Sebastiani (2006) adjectives are not only classified in terms of polarity and subjectivity but also per word sense, since different senses of the same term may have different opinion-related properties. For example, crazy ≈ insane (negative) vs. crazy ≈ enamored (positive). We adopt a similar approach where annotators assessed word senses using a triangle representation (figure 33). The triangle representation implies that more positive or more negative adjectives are also more subjective. But not all subjective adjectives are necessarily positive or negative, e.g., “blijkbaar” (apparently). We used CORNETTO to retrieve the different senses of each adjective. This is to our knowledge the first subjectivity lexicon for Dutch that applies sense discrimination.

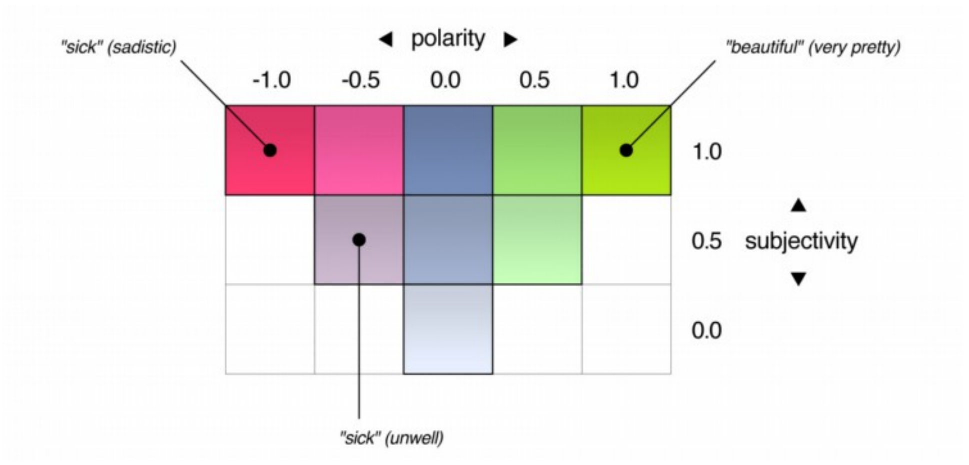


Figure 33. Triangle representation with polarity and subjectivity axes.

Dutch adjectives can be used as adverbs, where in English the ending `-ly` is usually required. For example: “ongelooflijk goed” is understood as “incredibly good” and not as “unbelievable” + “good". Annotators were asked to provide an additional intensity value, which can be used as a multiplier for the successive adjective’s polarity.

AGREEMENT

We removed a number of inflected adjectives, spelling errors and adverbs, bringing the final GOLD1000 lexicon to about 1,000 adjectives (1,500 word senses) with the average scores of the annotators. The lexicon contains 48% positive, 36% negative and 16% neutral assessments. Figure 34 shows a breakdown of the distribution. Table 14 shows the inter-annotator agreement (Fleiss’ kappa). We attain the best agreement for positive–neutral–negative polarity without considering polarity strength ($\varkappa$=`0.63`). Assessment of subjectivity strength is shown to be a much harder task ($\varkappa$=`0.34`), perhaps because it is more vague than classifying positive versus negative.

The DUOMAN subjectivity lexicon for Dutch (Jijkoun & Hofmann, 2009) contains 5,000+ words with polarity assessments by two annotators. About 50% of the adjectives in GOLD1000 also occur in DUOMAN. We compared the positive–neutral–negative polarity (without strength) of the adjectives in GOLD1000, using the average of word senses, to those that also occur in DUOMAN.

Agreement is κ=`0.82`. Some adjectives are positive in GOLD1000 but negative in DUOMAN, or vice-versa. One explanation is the different semantics between Dutch in the Netherlands and Dutch in Flanders for some adjectives (e.g., "maf"). Agreement increases to κ=`0.93` when these aberrant 27 adjectives are omitted from the measurement.

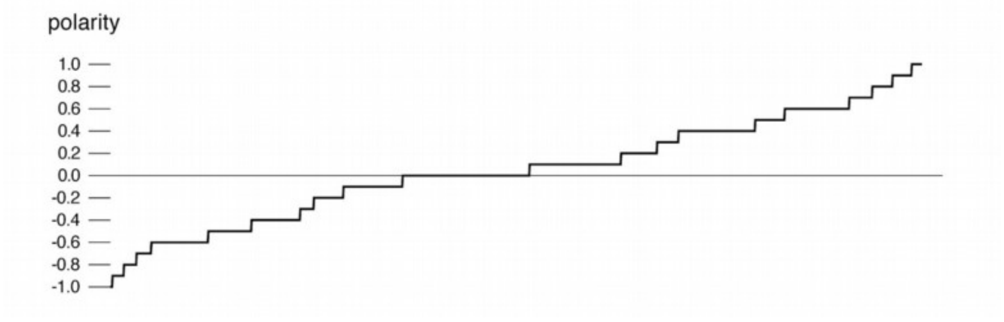


Figure 34. Distribution of positive-negative polarity strength in GOLD1000.

RATING	K
polarity (`-1` or `0` or `+1`)	`+0.63`
polarity (`-1.0` to `+1.0`)	`+0.47`
polarity + subjectivity	`+0.30`
subjectivity	`+0.34`
intensity	`+0.32`

Table 14. Agreement for seven annotators.

Automatic expansion

DISTRIBUTIONAL EXTRACTION

There is a well-known approach in computational linguistics in which semantic relatedness between words is extracted from distributional information (Schütze & Pedersen, 1997). For an example for Dutch, see Van de Cruys (2010). Van de Cruys uses a vector space with adjectives as document labels and nouns as vector features. The value for each feature represents the frequency an adjective precedes a noun. Classification or dimension reduction is then used to create groups of semantically related words. We applied the distributional approach to annotate new adjectives. From the Dutch TWNC newspaper corpus (Ordelman et al., 2002), we analyzed 3,000,000 words and selected the top 2,500 most frequent nouns. For each adjective in TWNC that is also in CORNETTO, we counted the number of times it directly precedes one or more of the top nouns. This results in 5,500+ adjective vectors with 2,500 vector features. For each GOLD1000 adjective we then applied k-NN using cosine similarity to retrieve the top 20 most similar nearest neighbors from the set. For example, for "fantastisch" (fantastic) the top five nearest neighbors are: "geweldig" (great, 70%), "prachtig" (beautiful, 51%), "uitstekend" (excellent, 50%), "prima" (fine, 50%), "mooi" (nice, 49%) and "goed" (good, 47%). The best nearest neighbors were handpicked by 2 annotators. These inherit polarity, subjectivity and intensity from their GOLD1000 anchor. The AUTO3000 lexicon then contains about 3,200 adjectives (3,700 word senses).

SPREADIING ACTIVATION

We then iteratively extend the lexicon by traversing relations between CORNETTO synsets. In CORNETTO or in WORDNET, word senses with a similar meaning are grouped in synsets. Different synsets are related to each other by synonymy (**is-same-as**), antonymy (**is-opposite-of**), hyponymy (**is-a**), and so on. For each adjective word sense we retrieve the CORNETTO synset and inherit polarity, subjectivity and intensity to related word senses. In three iterations we can spread out to over 2,200 new adjectives (3,000 word senses). The AUTO5500 lexicon contains about 5,400 adjectives (6,675 word senses).

Evaluation

For evaluation we tested with a set of 2,000 Dutch book reviews, which were not used in the GOLD1000 lexicon and evenly distributed over negative opinion (star rating 1 and 2) and positive opinion (4 and 5). For each review, we then scored polarity for adjectives that occur in the lexicon and compared the average strength to the original star rating. We took polarity `>= +0.1` as a positive observation and polarity `< +0.1` as a negative observation. This is a form of binary classification in which there will be a number of correctly classified words (true positives and negatives) and incorrectly classified words (false positives and negatives) by which we can calculate precision and recall . The polarity threshold can be lowered or raised, but `+0.1` yields the best results. Overall we attain a precision of 72% and a recall of 78%.

We then used the intensity strength. Instead of scoring “echt teleurgesteld” (truly disappointed) as “echt” (true) + “teleurgesteld” (disappointed) = `0.2` + `-0.4` = `-0.2`, we used “echt” (intensity `1.6`) × teleurgesteld = `1.6` × `-0.4` = `-0.64`. For book reviews, this increases recall to 82% without affecting precision. To provide a cross-domain measurement we tested with 2,000 Dutch music reviews evenly distributed by star rating. We attain a precision of 70% and a recall of 77%.

positive >= 0.1	BOOKS2000				
	# adjectives	A	P	R	F1
GOLD1000	794	0.75	0.72	0.82	0.77
AUTO3000	1,085	0.75	**0.72**	**0.82**	0.77
AUTO5500	1,286	0.74	0.72	0.79	0.75

positive >= 0.1	MUSIC2000				
	# adjectives	A	P	R	F1
GOLD1000	794	0.75	0.72	0.82	0.77
AUTO3000	1,085	0.75	**0.72**	**0.82**	0.77

Table 15. Number of unique adjectives rated, accuracy, precision, recall and F1-scores for opinion prediction.

The results for the GOLD1000 and AUTO3000 lexicons are nearly identical. The reason that precision and recall do not increase by adding more adjectives is that 94.8% of top frequent adjectives is already covered in GOLD1000. As noted, the distribution of adjectives over book reviews is exponential: the most frequent “goed” (good) occurs 6,380 times whereas for example “aapachtig” (apish) occurs only one time. Adding words such as "aapachtig" has a minimal coverage effect. Nevertheless, distributional extraction is a useful expansion technique with no drawback as indicated by the stable results for AUTO3000. For AUTO5500, F-score is less (-2%). The reason is that the adjectives "een" (united), "in" (hip) and "op" (exhausted) are part of the expanded lexicon. In Dutch, these words can also function as a common determiner (een = a/an) and common prepositions (in = in, op = on). Without them the scores for AUTO5500 come close to the results for GOLD1000 and AUTO3000. This suggests that the CORNETTO expansion technique can benefit from manual correction and part-of-speech tagging. Combined with part-of-speech tagging, this technique could be eligible for on-the-fly classification of unknown adjectives, since it can be implemented as a fast PATTERN tagger + CORNETTO traversal algorithm.

Using regular expressions for lemmatization, negation (reverse score for "niet", "nooit" and "geen") and exclamation marks, precision and recall for books increases to 77% and 84% respectively.

Analysis

In overall, positive predictions (57%) were more frequent than negative predictions (43%). By examining the test data, we see three reasons why it may be harder to identify negative opinions:

COMPARISON Some negative opinions make their point by referring to other instances, for example: "dat boek was grappig, origineel, pakkend, maar dit boek vond ik op al die punten tegenvallen" (that book was funny, inspiring, moving, but this book fails on all those points). The adjectives rate as positive but the review is negative.

FEATURAL OPINION In "de eerste tien pagina's zijn sterk, maar dan zakt het als een pudding in elkaar" (the first ten pages are quite good, but it falls apart from thereon) the positive opinion accounts for a specific feature of the book (first ten pages), while the general negative opinion is carried by a figure of speech (falling apart).

IRONY It is not possible to detect irony using a subjectivity lexicon. For example, "zou niet weten wat ik met mijn leven moest als ik dit geweldige boek gemist had" (wouldn't know what to do with my life if I had missed this awesome book).

English translation

Many word senses in CORNETTO have inter-language relations to WORDNET. We took advantage of this to map the polarity and subjectivity scores in the Dutch lexicon to an English lexicon. We then tested our English lexicon against POLARITY DATASET 2.0 (Pang & Lee, 2004) containing a 1,000 positive and a 1,000 negative IMDb movie reviews (imdb.com). Initial test results were poor: 66% precision and 54% recall. If we look at the 1,000 top frequent adjectives in 3,500 random English IMDb movie reviews, only 32% overlaps with the Dutch most frequent adjectives. We proceeded to manually annotate about 500 frequent English adjectives (1,500 word senses) to expand the English lexicon. This was done by a single annotator, but the effect is apparent: precision increases to 72% and recall to 71%.

In the next section, we present a case study using the Dutch lexicon on online political discourse.

7.2 Sentiment analysis for political discourse

In 2011, Belgium experienced the longest government formation period known to modern-day democratic systems (BBC Europe, 2011). At the basis lies the dual federalist structure in Belgium, the two major regions being the Dutch-speaking Flemish Community and the French-speaking Walloon Community. In 2010, the Dutch-speaking right-wing N-VA (Nieuw-Vlaamse Alliantie) emerged both as newcomer and largest party of the Belgian federal elections. The second largest party was the French-speaking left-wing PS (Parti Socialiste). While the N-VA ultimately seeks secession of Flanders from Belgium, the PS is inclined towards state interventionism. Over the following year they were unable to form a government coalition. It eventually took a record-breaking 541 days of negotiations to form a government. The media has played an important and sometimes controversial role in the course taken by the political parties, and passionate assertions have been expressed towards the media and its favoritism (Niven, 2003). However, the question whether there truly exists a media bias or not should be answered by systematic, statistical analysis.

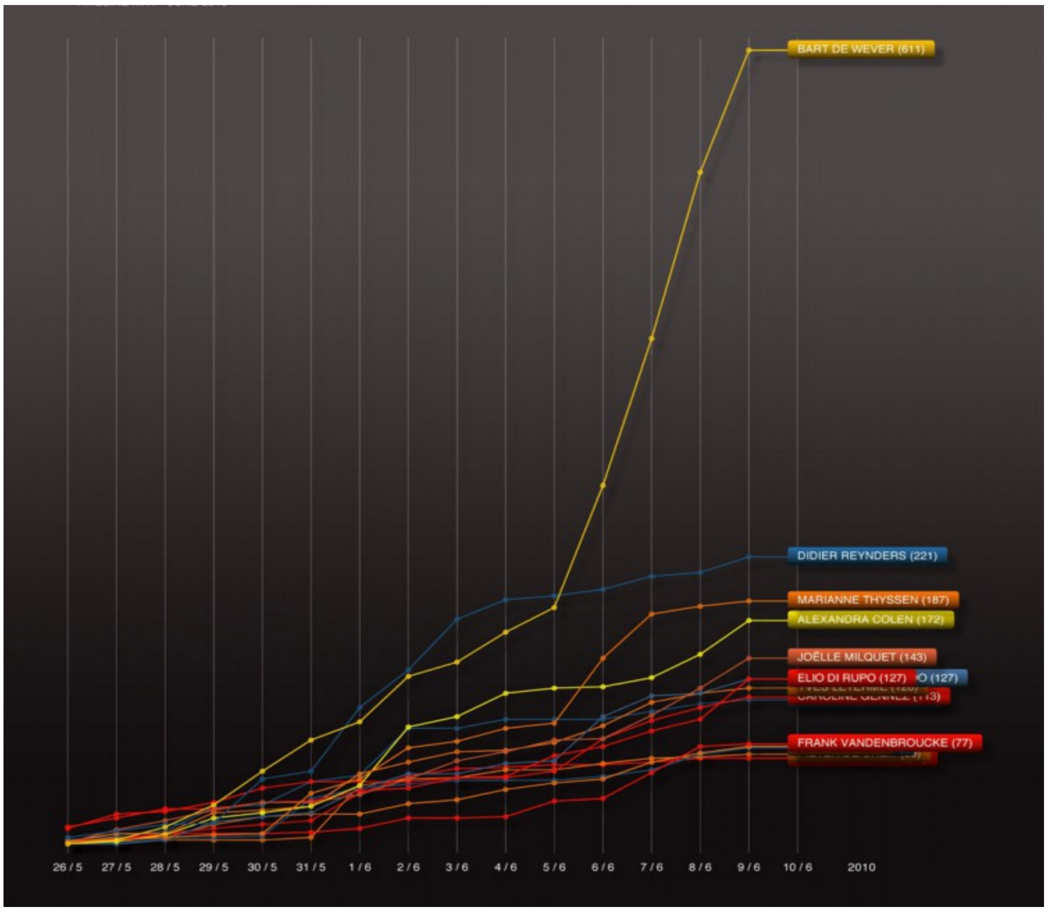


Figure 35. Timeline of Belgian political tweets, federal elections, May 26 to June 9 2010.
Highest ranking is "Bart De Wever" of the N-VA.

Nowadays, newspapers and other news providers are updating their online news feeds in near real-time, allowing those interested to follow the news in near real-time. The Web has thereby become an important broadcast medium for politicians. In Junqué de Fortuny, De Smedt, Martens & Daelemans (2012) we describe a text mining system that acquires online news feeds for Belgian political articles. We have used this system together with NODEBOX to visualize how the tone of reporting evolved throughout 2011, on party, politician and newspaper level.

In a first test[29] we were able to predict key figures in the government formation (e.g., Bart De Wever and Elio Di Rupo) by mining 7,500 Twitter messages containing the name of Belgian politicians prior to the 2010 elections. This is shown in figure 35.

Media bias

A study by Benewick, Birch, Blumler & Ewbank (1969) shows that exposure to a political party's broadcasts is positively related to a more favorable attitude towards the party for those with medium or weak motivation to follow the campaign. Knowing this, McCombs & Shaw (1972) raise the question whether mass media sources actually reproduce the political world perfectly. In an imperfect setting, biases of media could propagate to the public opinion and influence favoritism towards one or another party or politician, thus shaping political reality. Most of this bias is introduced by the selection and editing process. McCombs & Shaw conclude that mass media may well set the agenda of political campaigns. D'Alessio & Allen (2000) found three main bias metrics used to study partisan media bias:

GATEKEEPING

Selection, editing and rejection of news articles to be published introduces a bias. This so-called gatekeeping causes some topics to never surface in the media landscape. It is therefore a sampling bias introduced by editors and journalists (Dexter & White, 1964). However, measuring the gatekeeping bias turns out to be infeasible, since information about rejection is not available.

COVERAGE

Media coverage is the amount of attention that each side of the issue receives. We can measure coverage as the amount of online articles for each entity (i.e., a party or politician). We argue that fair coverage is determined by an *a priori* distribution. The *a priori* distribution represents all entities in the population by their relative importance, as measured by electoral votes in the latest elections. Deviations from the fair distribution show bias towards one or another entity.

STATEMENT

Media coverage can be favorable or unfavorable. We can use the subjectivity lexicon in PATTERN to measure statement bias in terms of positive–negative coverage. This is an associative measure: a party is associated with a negative image when most of its coverage contains negative content. However, when an online article is classified as negative, this does not necessarily imply favoritism from a news source or a journalist. The entity may be purposely interjecting criticism or the article may contain citations from rival entities.

[29] http://www.clips.ua.ac.be/pages/pattern-examples-elections

Web crawler + sentiment analysis

We used the 2010 Chamber election results[30] as a gold standard against which we measured media coverage bias. The rationale is that the media is aware of these results and therefore knows the distribution of political parties beforehand. Similarly, we used the 2010 Senate election results to compare politicians. Politicians are voted for directly in the Senate elections whereas Chamber elections concern party votes. We then implemented a crawler to obtain a corpus of over 68,000 unique online articles from all major Flemish newspapers, spanning a ten-month period (January 1, 2011 to October 31, 2011). We can see how this is not hard to accomplish by following the steps in the pattern.web examples in chapter 6.4. Each article in the corpus contains the name of at least one Flemish political party or leading figure. The criterion for being a political party of interest is based on the votes for that party in the 2010 Chamber elections, that is, we only include parties voted into the Chamber. A leading figure of interest is a politician with a top ten ranking amount of preference votes in the 2010 Senate elections.

We analyzed the occurrences of each entity in each newspaper article. The sentiment associated with an entity in an article is the polarity of adjectives, in a window of two sentences before and two sentences after. The window is important since an article can mention several entities or switch tone. It ensures a more reliable correlation between the entity and the polarity score, contrary to using all adjectives across the article. A similar approach with a 10-word window is used in Balahur et al. (2010). They report improved accuracy compared to measuring all words. We also excluded adjectives that score between `-0.1` and `+0.1` to reduce noise. This results in a set of about 365,000 assessments, where one assessment corresponds to one adjective linked to a party or politician.

For example:

> Bart De Wever (N-VA) verwijt de Franstalige krant Le Soir dat ze aanzet tot haat, omdat zij een opiniestuk publiceerde over de Vlaamse wooncode met daarbij een foto van een Nigeriaans massagraf. De hoofdredactrice legt uit waarom ze De Wever in zijn segregatiezucht hard zal blijven bestrijden. "Verontwaardigd? Ja, we zijn eerst en boven alles verontwaardigd."
>
> Bart De Wever (N-VA) accuses the French newspaper Le Soir of inciting hate after they published an opinion piece on the Flemish housing regulations together with a picture of a Nigerian mass grave. The editor explains why they will continue to fight De Wever's love of segregation firmly. "Outraged? Yes, we are first and above all outraged."

The adjective "hard" (firm) scores `-0.03` and is excluded. The adjective "verontwaardigd" (outraged) scores `-0.4`. In overall the item on the political party N-VA is assessed as negative.

[30] Federal Public Services Home Affairs: http://polling2010.belgium.be/
All percentages in this study were normalized over the Flemish parties and the selection.

Evaluation

The *coverage* of an entity by a newspaper source is the number of articles published by the source on that entity, divided by the total amount of articles by that source in the corpus. The *popularity* of an entity is the relative number of votes (Chamber/Senate). This is the fair distribution. The *coverage bias* of a newspaper is the difference between the real distribution and the fair distribution. Table 16 shows the newspaper sources used for the analysis with their respective number of articles, number of readers and the coverage bias:

SOURCE	REGIONAL	# ARTICLES	# READERS	BIAS
De Morgen	NO	11,303	256,800	21.53%
De Redactie	NO	5,767	146,250	16.16%
De Standaard	NO	9,154	314,000	23.32%
De Tijd	NO	10,061	123,300	22.71%
GVA	YES	9,154	395,700	**27.30**%
HBVL	YES	3,511	423,700	**37.43**%
HLN	NO	11,380	1,125,600	21.62%
Het Nieuwsblad	NO	7,320	1,002,200	20.24%

Table 16. Newspaper source, number of articles, number of readers and coverage bias.

The highest coverage bias is found for the regional newspapers Gazet van Antwerpen (GVA) and Het Belang van Limburg (HBVL). Both newspaper are owned by the media group Concentra. De Morgen, De Tijd and Het Laatste Nieuws (HLN) are owned by De Persgroep. De Standaard and Het Nieuwsblad are owned by Corelio. De Redactie is an online news channel owned by VRT.

Figure 36 shows how a coverage bias is found for some parties, with a maximal positive bias towards CD&V and a maximal negative bias towards the far-right Vlaams Belang (VB). The first result can be justified by the fact that CD&V was running the interim government while the new government formations took place. The latter result gives supportive evidence that the party is being quarantined by the media (Yperman, 2004). These tendencies propagate to the local level, for example Vlaams Belang is under-represented in all newspapers. Some local differences exist as well: the regional newspaper HBVL has a large negative bias towards N-VA. Regional newspapers in general (GVA, HBVL) have a larger coverage bias.

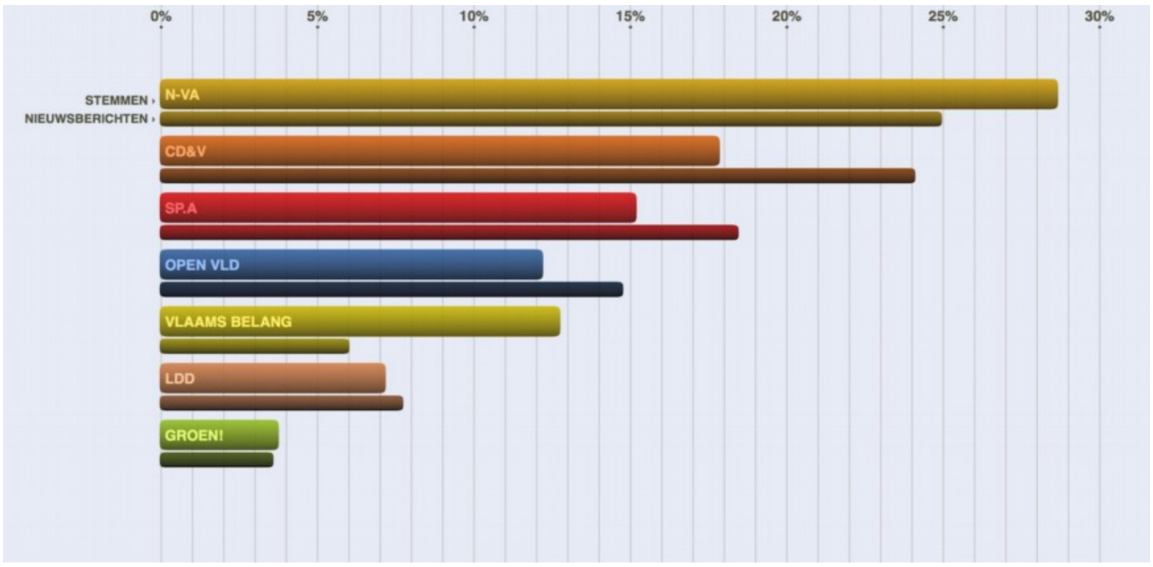


Figure 36. Discrepancy between media coverage and popularity for political parties.
The wide bar represents popularity, the narrow bar media coverage.

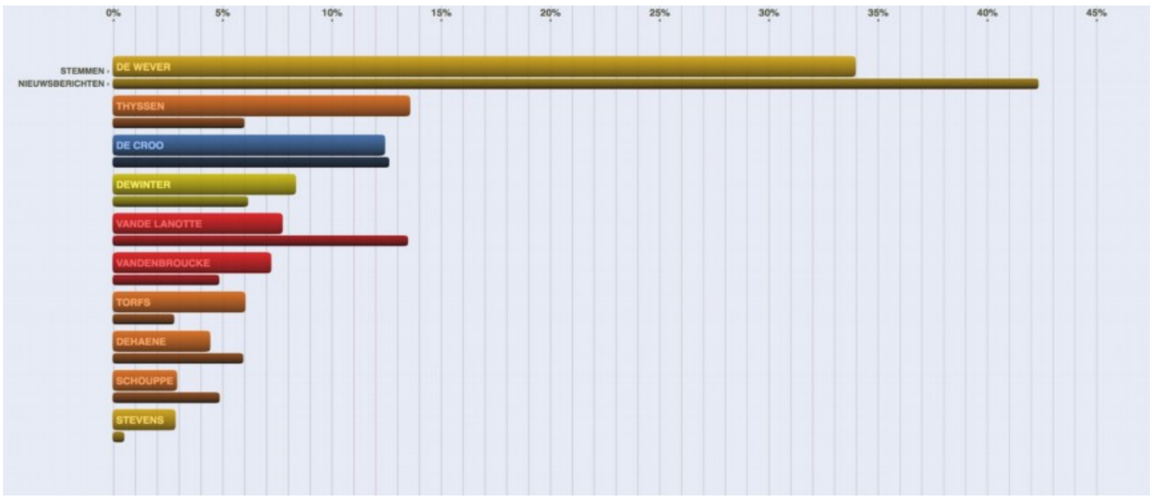


Figure 37. Discrepancy between media coverage and popularity for politicians.
The wide bar represents popularity, the narrow bar media coverage.

Figure 37 shows that the coverage bias for politicians varies irrespective of their party. For example positive bias for Bart De Wever is not reflected in negative bias for his party (N-VA).

Figure 38 shows the distribution of positive–negative sentiment for each political party. Overall, 20–30% of newspaper coverage is assessed as negative. Some statement bias is present under the assumption of uniformity of the sentiment distribution. Highest negative scores are measured for the far-right Vlaams Belang (–30.3%) and for the N-VA (–28.7%).

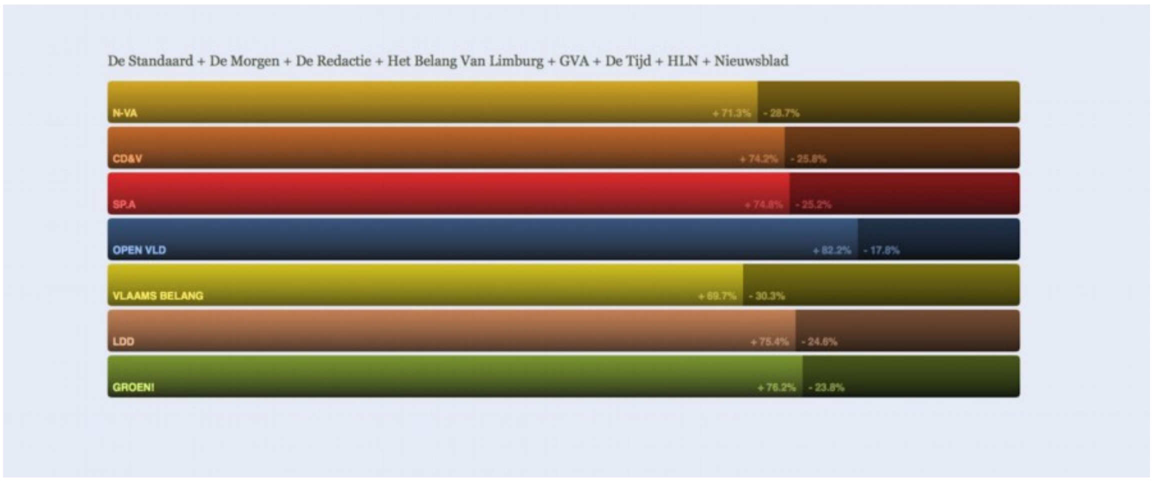


Figure 38. Sentiment for each political party, with the percentage of positive coverage on the left and negative on the right.

For each given political party, we then grouped sentiment assessments in subsets of one week and constructed a timeline of the consecutive weeks. We calculated a simple moving average (SMA) across all weeks to smoothen fluctuation in individual parties and emphasize the differences across parties.

Figure 39 shows the SMA for each political party across all newspapers. It is interesting to note the peak with all parties (except Vlaams Belang) in July-August. At this time the negotiating parties involved in the government formation were on a three-week leave. Once the negotiations resumed around August 15th, the peak drops. In the Belgian political crisis, it seems, no news equals good news.

Figure 40 shows the SMA for each newspaper across all political parties. The curves with the highest fluctuation are those for Het Belang Van Limburg and De Redactie. Het Belang Van Limburg also has the highest average sentiment: `+0.15` against [`0.13`, `0.14`] for all other newspapers. De Standaard appears to deliver the most neutral political articles.

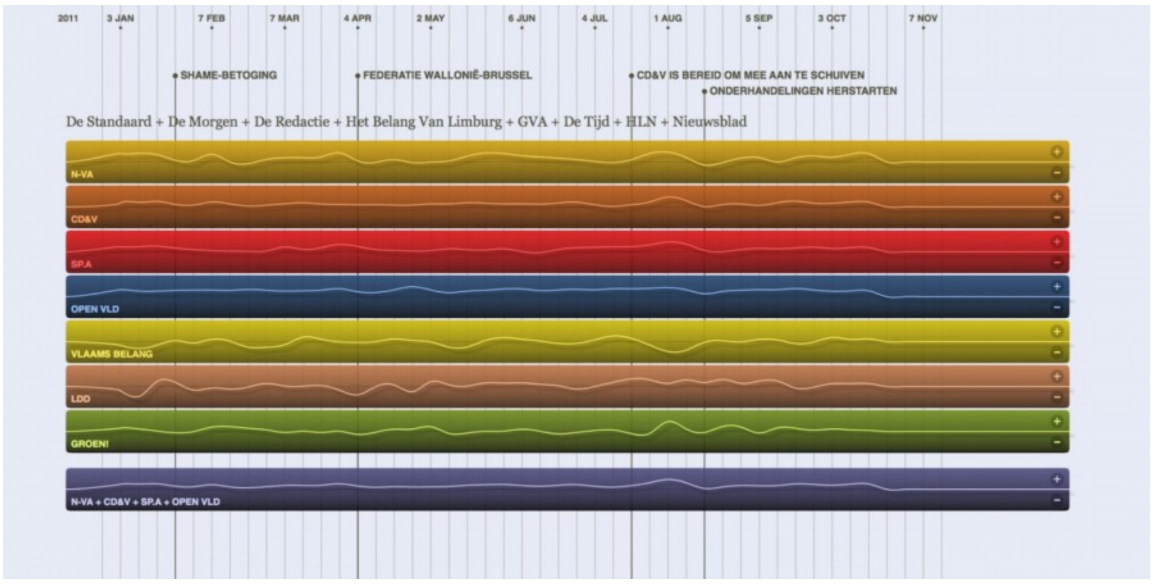


Figure 39. Sentiment of news items in 2011 for each party.

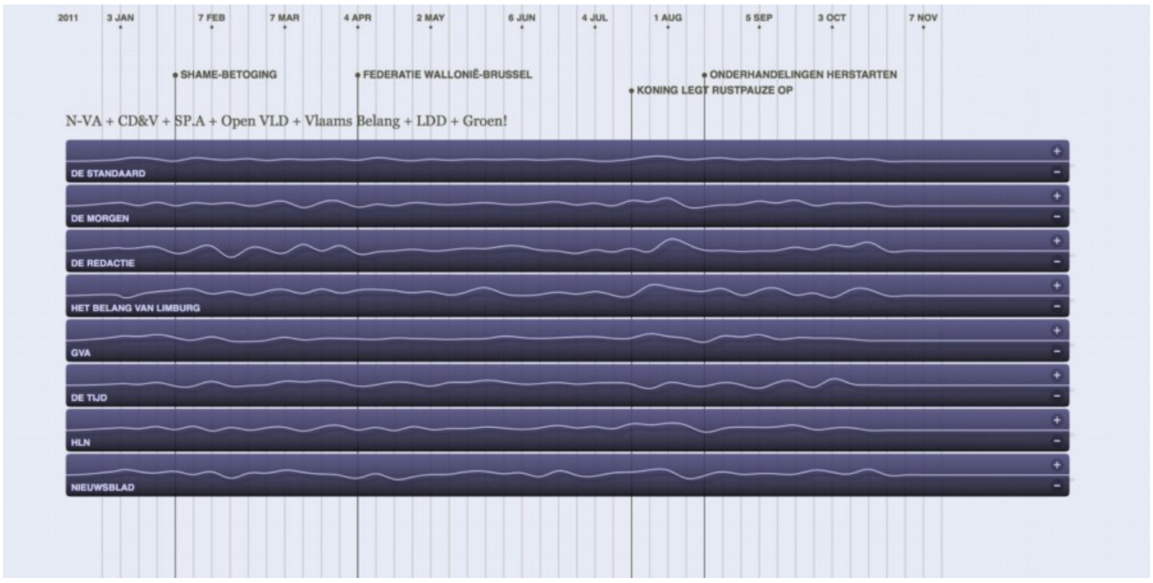


Figure 40. Sentiment of news items in 2011 for each newspaper.

In summary, quantitative analysis shows that a media coverage bias does indeed exist, most notably in regional newspapers. The figures provide support for the claim that some statement bias exists throughout the mass media. Statement bias is most notably found in the online news channel of De Redactie and the regional newspaper Het Belang van Limburg. Methodologically, we have shown that sentiment expert systems are a viable strategy for research in political text corpora.

7.3 Discussion

In this chapter we demonstrated how the PATTERN package discussed in chapter 6 can be used to build a subjectivity lexicon for sentiment analysis. In summary, we mined the Web for product reviews and annotated the most frequent adjectives with a positive–negative score. We applied two techniques (distributional extraction and spreading activation) to expand the lexicon automatically and tested its accuracy using a different set of product reviews + star rating. We then discussed a case study of sentiment analysis for online political discourse. Adjectives retain their polarity regardless of what words they qualify: "good book", "good dog" and "good speech" are all positive. It follows that the approach should be scalable to other domains besides product reviews, such as political discourse. In support of this, the U.S. Colorado Department of Transportation use the English subjectivity lexicon in PATTERN to enhance their asset condition forecasting abilities. Most of the asset condition records are expressed in natural language. An evaluation (figure 41, Cole, R., personal communication, 2012) indeed suggests a correlation between the polarity of the inspector's comment and the age of the asset (e.g., steel girder).

Arguably, the approach can also be used for fine art assessment (e.g., "vibrant colors", "inspiring work", "dull exhibition"). This ties in with Csikszentmihalyi's systems view of creativity (chapter 4), which attributes creativity to the recognition and adoption by peers. We can use sentiment analysis as a means of evaluating the performance of an artistic model of creativity. For example as proposed in chapter 2, using a setup of PERCOLATOR that publishes its works to a web page where visitors can comment on it. A practical case study is left up to future work.

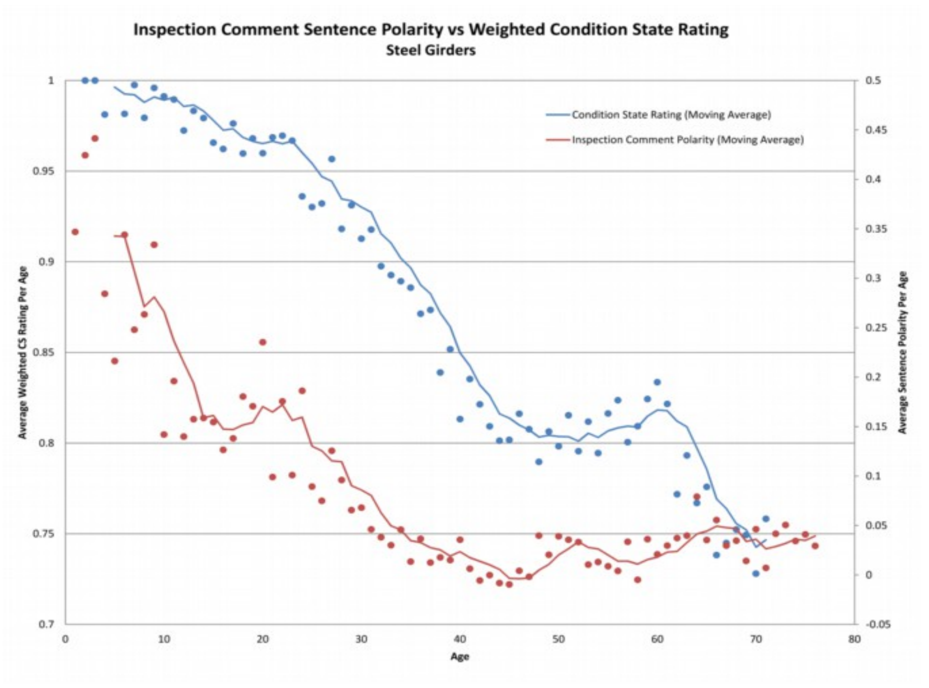


Figure 41. Inspection comment polarity vs. weighted condition state rating.
© U.S. Colorado Department of Transportation. Used with permission.