22 NATURAL LANGUAGE PROCESSING

In which we see how to make use of the copious knowledge that is expressed in natural language.

Homo sapiens is set apart from other species by the capacity for language. Somewhere around 100,000 years ago, humans learned how to speak, and about 7,000 years ago learned to write. Although chimpanzees, dolphins, and other animals have shown vocabularies of hundreds of signs, only humans can reliably communicate an unbounded number of qualitatively different messages on any topic using discrete signs.

Of course, there are other attributes that are uniquely human: no other species wears clothes, creates representational art, or watches three hours of television a day. But when Alan Turing proposed his Test (see Section 1.1.1), he based it on language, not art or TV. There are two main reasons why we want our computer agents to be able to process natural languages: first, to communicate with humans, a topic we take up in Chapter 23, and second, to acquire information from written language, the focus of this chapter.

KNOWLEDGE ACQUISITION

There are over a trillion pages of information on the Web, almost all of it in natural language. An agent that wants to do **knowledge acquisition** needs to understand (at least partially) the ambiguous, messy languages that humans use. We examine the problem from the point of view of specific information-seeking tasks: text classification, information retrieval, and information extraction. One common factor in addressing these tasks is the use of

LANGUAGE MODEL

language models: models that predict the probability distribution of language expressions.

22.1 LANGUAGE MODELS

LANGUAGE

Formal languages, such as the programming languages Java or Python, have precisely defined language models. A **language** can be defined as a set of strings; "`print(2 + 2)`" is a legal program in the language Python, whereas "`2)+(2 print`" is not. Since there are an infinite number of legal programs, they cannot be enumerated; instead they are specified by a

GRAMMAR

set of rules called a **grammar**. Formal languages also have rules that define the meaning or

SEMANTICS

semantics of a program; for example, the rules say that the "meaning" of "`2 + 2`" is 4, and the meaning of "`1/0`" is that an error is signaled.

Natural languages, such as English or Spanish, cannot be characterized as a definitive set of sentences. Everyone agrees that "Not to be invited is sad" is a sentence of English, but people disagree on the grammaticality of "To be not invited is sad." Therefore, it is more fruitful to define a natural language model as a probability distribution over sentences rather than a definitive set. That is, rather than asking if a string of *words* is or is not a member of the set defining the language, we instead ask for $P(S = words)$—what is the probability that a random sentence would be *words*.

AMBIGUITY

Natural languages are also **ambiguous**. "He saw her duck" can mean either that he saw a waterfowl belonging to her, or that he saw her move to evade something. Thus, again, we cannot speak of a single meaning for a sentence, but rather of a probability distribution over possible meanings.

Finally, natural languages are difficult to deal with because they are very large, and constantly changing. Thus, our language models are, at best, an approximation. We start with the simplest possible approximations and move up from there.

22.1.1 *N*-gram character models

CHARACTERS

Ultimately, a written text is composed of **characters**—letters, digits, punctuation, and spaces in English (and more exotic characters in some other languages). Thus, one of the simplest language models is a probability distribution over sequences of characters. As in Chapter 15, we write $P(c_{1:N})$ for the probability of a sequence of N characters, c_1 through c_N. In one Web collection, $P(\text{"the"}) = 0.027$ and $P(\text{"zgq"}) = 0.000000002$. A sequence of written symbols of length n is called an n-gram (from the Greek root for writing or letters), with special case "unigram" for 1-gram, "bigram" for 2-gram, and "trigram" for 3-gram. A model of the probability distribution of n-letter sequences is thus called an n**-gram model**. (But be careful: we can have n-gram models over sequences of words, syllables, or other units; not just over characters.)

N-GRAM MODEL

An n-gram model is defined as a **Markov chain** of order $n-1$. Recall from page 568 that in a Markov chain the probability of character c_i depends only on the immediately preceding characters, not on any other characters. So in a trigram model (Markov chain of order 2) we have

$$P(c_i \mid c_{1:i-1}) = P(c_i \mid c_{i-2:i-1}) \ .$$

We can define the probability of a sequence of characters $P(c_{1:N})$ under the trigram model by first factoring with the chain rule and then using the Markov assumption:

$$P(c_{1:N}) = \prod_{i=1}^{N} P(c_i \mid c_{1:i-1}) = \prod_{i=1}^{N} P(c_i \mid c_{i-2:i-1}) \ .$$

For a trigram character model in a language with 100 characters, $\mathbf{P}(C_i|C_{i-2:i-1})$ has a million entries, and can be accurately estimated by counting character sequences in a body of text of 10 million characters or more. We call a body of text a **corpus** (plural *corpora*), from the Latin word for *body*.

CORPUS

LANGUAGE IDENTIFICATION

What can we do with n-gram character models? One task for which they are well suited is **language identification**: given a text, determine what natural language it is written in. This is a relatively easy task; even with short texts such as "Hello, world" or "Wie geht es dir," it is easy to identify the first as English and the second as German. Computer systems identify languages with greater than 99% accuracy; occasionally, closely related languages, such as Swedish and Norwegian, are confused.

One approach to language identification is to first build a trigram character model of each candidate language, $P(c_i \mid c_{i-2:i-1}, \ell)$, where the variable ℓ ranges over languages. For each ℓ the model is built by counting trigrams in a corpus of that language. (About 100,000 characters of each language are needed.) That gives us a model of $\mathbf{P}(\mathit{Text} \mid \mathit{Language})$, but we want to select the most probable language given the text, so we apply Bayes' rule followed by the Markov assumption to get the most probable language:

$$\begin{aligned}\ell^* &= \operatorname*{argmax}_\ell P(\ell \mid c_{1:N}) \\ &= \operatorname*{argmax}_\ell P(\ell) P(c_{1:N} \mid \ell) \\ &= \operatorname*{argmax}_\ell P(\ell) \prod_{i=1}^{N} P(c_i \mid c_{i-2:i-1}, \ell)\end{aligned}$$

The trigram model can be learned from a corpus, but what about the prior probability $P(\ell)$? We may have some estimate of these values; for example, if we are selecting a random Web page we know that English is the most likely language and that the probability of Macedonian will be less than 1%. The exact number we select for these priors is not critical because the trigram model usually selects one language that is several orders of magnitude more probable than any other.

Other tasks for character models include spelling correction, genre classification, and named-entity recognition. Genre classification means deciding if a text is a news story, a legal document, a scientific article, etc. While many features help make this classification, counts of punctuation and other character n-gram features go a long way (Kessler *et al.*, 1997). Named-entity recognition is the task of finding names of things in a document and deciding what class they belong to. For example, in the text "Mr. Sopersteen was prescribed aciphex," we should recognize that "Mr. Sopersteen" is the name of a person and "aciphex" is the name of a drug. Character-level models are good for this task because they can associate the character sequence "ex␣" ("ex" followed by a space) with a drug name and "steen␣" with a person name, and thereby identify words that they have never seen before.

22.1.2 Smoothing *n*-gram models

The major complication of n-gram models is that the training corpus provides only an estimate of the true probability distribution. For common character sequences such as "␣th" any English corpus will give a good estimate: about 1.5% of all trigrams. On the other hand, "␣ht" is very uncommon—no dictionary words start with ht. It is likely that the sequence would have a count of zero in a training corpus of standard English. Does that mean we should assign $P(\text{"␣th"}) = 0$? If we did, then the text "The program issues an http request" would have

an English probability of zero, which seems wrong. We have a problem in generalization: we want our language models to generalize well to texts they haven't seen yet. Just because we have never seen "␣http" before does not mean that our model should claim that it is impossible. Thus, we will adjust our language model so that sequences that have a count of zero in the training corpus will be assigned a small nonzero probability (and the other counts will be adjusted downward slightly so that the probability still sums to 1). The process od adjusting the probability of low-frequency counts is called **smoothing**.

SMOOTHING

The simplest type of smoothing was suggested by Pierre-Simon Laplace in the 18th century: he said that, in the lack of further information, if a random Boolean variable X has been false in all n observations so far then the estimate for $P(X = true)$ should be $1/(n+2)$. That is, he assumes that with two more trials, one might be true and one false. Laplace smoothing (also called add-one smoothing) is a step in the right direction, but performs relatively poorly.

BACKOFF MODEL

A better approach is a **backoff model**, in which we start by estimating n-gram counts, but for any particular sequence that has a low (or zero) count, we back off to $(n-1)$-grams. **Linear interpolation smoothing** is a backoff model that combines trigram, bigram, and unigram models by linear interpolation. It defines the probability estimate as

LINEAR INTERPOLATION SMOOTHING

$$\widehat{P}(c_i|c_{i-2:i-1}) = \lambda_3 P(c_i|c_{i-2:i-1}) + \lambda_2 P(c_i|c_{i-1}) + \lambda_1 P(c_i) \ ,$$

where $\lambda_3 + \lambda_2 + \lambda_1 = 1$. The parameter values λ_i can be fixed, or they can be trained with an expectation–maximization algorithm. It is also possible to have the values of λ_i depend on the counts: if we have a high count of trigrams, then we weigh them relatively more; if only a low count, then we put more weight on the bigram and unigram models. One camp of researchers has developed ever more sophisticated smoothing models, while the other camp suggests gathering a larger corpus so that even simple smoothing models work well. Both are getting at the same goal: reducing the variance in the language model.

One complication: note that the expression $P(c_i \mid c_{i-2:i-1})$ asks for $P(c_1 \mid c_{-1:0})$ when $i = 1$, but there are no characters before c_1. We can introduce artificial characters, for example, defining c_0 to be a space character or a special "begin text" character. Or we can fall back on lower-order Markov models, in effect defining $c_{-1:0}$ to be the empty sequence and thus $P(c_1 \mid c_{-1:0}) = P(c_1)$.

22.1.3 Model evaluation

With so many possible n-gram models—unigram, bigram, trigram, interpolated smoothing with different values of λ, etc.—how do we know what model to choose? We can evaluate a model with cross-validation. Split the corpus into a training corpus and a validation corpus. Determine the parameters of the model from the training data. Then evaluate the model on the validation corpus.

The evaluation can be a task-specific metric, such as measuring accuracy on language identification. Alternatively we can have a task-independent model of language quality: calculate the probability assigned to the validation corpus by the model; the higher the probability the better. This metric is inconvenient because the probability of a large corpus will be a very small number, and floating-point underflow becomes an issue. A different way of describing the probability of a sequence is with a measure called **perplexity**, defined as

PERPLEXITY

$$Perplexity(c_{1:N}) = P(c_{1:N})^{-\frac{1}{N}} .$$

Perplexity can be thought of as the reciprocal of probability, normalized by sequence length. It can also be thought of as the weighted average branching factor of a model. Suppose there are 100 characters in our language, and our model says they are all equally likely. Then for a sequence of any length, the perplexity will be 100. If some characters are more likely than others, and the model reflects that, then the model will have a perplexity less than 100.

22.1.4 *N*-gram word models

VOCABULARY

Now we turn to n-gram models over words rather than characters. All the same mechanism applies equally to word and character models. The main difference is that the **vocabulary**—the set of symbols that make up the corpus and the model—is larger. There are only about 100 characters in most languages, and sometimes we build character models that are even more restrictive, for example by treating "A" and "a" as the same symbol or by treating all punctuation as the same symbol. But with word models we have at least tens of thousands of symbols, and sometimes millions. The wide range is because it is not clear what constitutes a word. In English a sequence of letters surrounded by spaces is a word, but in some languages, like Chinese, words are not separated by spaces, and even in English many decisions must be made to have a clear policy on word boundaries: how many words are in "ne'er-do-well"? Or in "(Tel:1-800-960-5660x123)"?

OUT OF VOCABULARY

Word n-gram models need to deal with **out of vocabulary** words. With character models, we didn't have to worry about someone inventing a new letter of the alphabet.[1] But with word models there is always the chance of a new word that was not seen in the training corpus, so we need to model that explicitly in our language model. This can be done by adding just one new word to the vocabulary: `<UNK>`, standing for the unknown word. We can estimate n-gram counts for `<UNK>` by this trick: go through the training corpus, and the first time any individual word appears it is previously unknown, so replace it with the symbol `<UNK>`. All subsequent appearances of the word remain unchanged. Then compute n-gram counts for the corpus as usual, treating `<UNK>` just like any other word. Then when an unknown word appears in a test set, we look up its probability under `<UNK>`. Sometimes multiple unknown-word symbols are used, for different classes. For example, any string of digits might be replaced with `<NUM>`, or any email address with `<EMAIL>`.

To get a feeling for what word models can do, we built unigram, bigram, and trigram models over the words in this book and then randomly sampled sequences of words from the models. The results are

Unigram: logical are as are confusion a may right tries agent goal the was ...
Bigram: systems are very similar computational approach would be represented ...
Trigram: planning and scheduling are integrated the success of naive bayes model is ...

Even with this small sample, it should be clear that the unigram model is a poor approximation of either English or the content of an AI textbook, and that the bigram and trigram models are

[1] With the possible exception of the groundbreaking work of T. Geisel (1955).

much better. The models agree with this assessment: the perplexity was 891 for the unigram model, 142 for the bigram model and 91 for the trigram model.

With the basics of n-gram models—both character- and word-based—established, we can turn now to some language tasks.

22.2 TEXT CLASSIFICATION

TEXT CLASSIFICATION

We now consider in depth the task of **text classification**, also known as **categorization**: given a text of some kind, decide which of a predefined set of classes it belongs to. Language identification and genre classification are examples of text classification, as is sentiment analysis

SPAM DETECTION

(classifying a movie or product review as positive or negative) and **spam detection** (classifying an email message as spam or not-spam). Since "not-spam" is awkward, researchers have coined the term **ham** for not-spam. We can treat spam detection as a problem in supervised learning. A training set is readily available: the positive (spam) examples are in my spam folder, the negative (ham) examples are in my inbox. Here is an excerpt:

> Spam: Wholesale Fashion Watches -57% today. Designer watches for cheap ...
> Spam: You can buy ViagraFr$1.85 All Medications at unbeatable prices! ...
> Spam: WE CAN TREAT ANYTHING YOU SUFFER FROM JUST TRUST US ...
> Spam: Sta.rt earn*ing the salary yo,u d-eserve by o'btaining the prope,r crede'ntials!
>
> Ham: The practical significance of hypertree width in identifying more ...
> Ham: Abstract: We will motivate the problem of social identity clustering: ...
> Ham: Good to see you my friend. Hey Peter, It was good to hear from you. ...
> Ham: PDS implies convexity of the resulting optimization problem (Kernel Ridge ...

From this excerpt we can start to get an idea of what might be good features to include in the supervised learning model. Word n-grams such as "for cheap" and "You can buy" seem to be indicators of spam (although they would have a nonzero probability in ham as well). Character-level features also seem important: spam is more likely to be all uppercase and to have punctuation embedded in words. Apparently the spammers thought that the word bigram "you deserve" would be too indicative of spam, and thus wrote "yo,u d-eserve" instead. A character model should detect this. We could either create a full character n-gram model of spam and ham, or we could handcraft features such as "number of punctuation marks embedded in words."

Note that we have two complementary ways of talking about classification. In the language-modeling approach, we define one n-gram language model for $\mathbf{P}(Message \mid spam)$ by training on the spam folder, and one model for $\mathbf{P}(Message \mid ham)$ by training on the inbox. Then we can classify a new message with an application of Bayes' rule:

$$\underset{c \in \{spam, ham\}}{\operatorname{argmax}} P(c \mid message) = \underset{c \in \{spam, ham\}}{\operatorname{argmax}} P(message \mid c)\, P(c)\ .$$

where $P(c)$ is estimated just by counting the total number of spam and ham messages. This approach works well for spam detection, just as it did for language identification.

In the machine-learning approach we represent the message as a set of feature/value pairs and apply a classification algorithm h to the feature vector $\mathbf{X}$. We can make the language-modeling and machine-learning approaches compatible by thinking of the n-grams as features. This is easiest to see with a unigram model. The features are the words in the vocabulary: "a," "aardvark," ..., and the values are the number of times each word appears in the message. That makes the feature vector large and sparse. If there are 100,000 words in the language model, then the feature vector has length 100,000, but for a short email message almost all the features will have count zero. This unigram representation has been called the

BAG OF WORDS

bag of words model. You can think of the model as putting the words of the training corpus in a bag and then selecting words one at a time. The notion of order of the words is lost; a unigram model gives the same probability to any permutation of a text. Higher-order n-gram models maintain some local notion of word order.

With bigrams and trigrams the number of features is squared or cubed, and we can add in other, non-n-gram features: the time the message was sent, whether a URL or an image is part of the message, an ID number for the sender of the message, the sender's number of previous spam and ham messages, and so on. The choice of features is the most important part of creating a good spam detector—more important than the choice of algorithm for processing the features. In part this is because there is a lot of training data, so if we can propose a feature, the data can accurately determine if it is good or not. It is necessary to constantly update features, because spam detection is an **adversarial task**; the spammers modify their spam in response to the spam detector's changes.

It can be expensive to run algorithms on a very large feature vector, so often a process

FEATURE SELECTION

of **feature selection** is used to keep only the features that best discriminate between spam and ham. For example, the bigram "of the" is frequent in English, and may be equally frequent in spam and ham, so there is no sense in counting it. Often the top hundred or so features do a good job of discriminating between classes.

Once we have chosen a set of features, we can apply any of the supervised learning techniques we have seen; popular ones for text categorization include k-nearest-neighbors, support vector machines, decision trees, naive Bayes, and logistic regression. All of these have been applied to spam detection, usually with accuracy in the 98%–99% range. With a carefully designed feature set, accuracy can exceed 99.9%.

22.2.1 Classification by data compression

DATA COMPRESSION

Another way to think about classification is as a problem in **data compression**. A lossless compression algorithm takes a sequence of symbols, detects repeated patterns in it, and writes a description of the sequence that is more compact than the original. For example, the text "0.142857142857142857" might be compressed to "0.[142857]*3." Compression algorithms work by building dictionaries of subsequences of the text, and then referring to entries in the dictionary. The example here had only one dictionary entry, "142857."

In effect, compression algorithms are creating a language model. The LZW algorithm in particular directly models a maximum-entropy probability distribution. To do classification by compression, we first lump together all the spam training messages and compress them as