O'REILLY®

Early Release
RAW & UNEDITED

Mastering Feature Engineering

PRINCIPLES AND TECHNIQUES FOR DATA SCIENTISTS

Alice Zheng

Mastering Feature Engineering

Principles and Techniques for Data Scientists

Alice X. Zheng

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Mastering Feature Engineering

by Alice Zheng

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/ institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Shannon Cutt
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2017: First Edition

Revision History for the First Edition

2016-06-13: First Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491953242* for release details.

While the publisher and the author(s) have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author(s) disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-95324-2

[FILL IN]

Table of Contents

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at https://github.com/oreillymedia/title_title.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at http://www.oreilly.com/catalog/<catalog page>.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at http://www.oreilly.com.

Find us on Facebook: http://facebook.com/oreilly

Follow us on Twitter: http://twitter.com/oreillymedia

Watch us on YouTube: http://www.youtube.com/oreillymedia

Acknowledgments

CHAPTER 1
Introduction

Feature engineering sits right between “data" and “modeling" in the machine learning pipeline for making sense of data. It is a crucial step, because the right features can make the job of modeling much easier, and therefore the whole process has a higher chance of success. Some people estimate that 80% of their effort in a machine learning application is spent on feature engineering and data cleaning. Despite its importance, the topic is rarely discussed on its own. Perhaps it’s because the right features can only be defined in the context of both the model and the data. Since data and models are so diverse, it’s difficult to generalize the practice of feature engineering across projects.

Nevertheless, feature engineering is not just an ad hoc practice. There are deeper principles at work, and they are best illustrated in situ. Each chapter of this book addresses one data problem: how to represent text data or image data, how to reduce dimensionality of auto-generated features, when and how to normalize, etc. Think of this as a collection of inter-connected short stories, as opposed to a single long novel. Each chapter provides a vignette into in the vast array of existing feature engineering techniques. Together, they illustrate some of the overarching principles.

Mastering a subject is not just about knowing the definitions and being able to derive the formulas. It is not enough to know how the mechanism works and what it can do. It must also involve understanding why it is designed that way, how it relates to other techniques that we already know, and what are the pros and cons of each approach. Mastery is about knowing precisely how something is done, having an intuition for the underlying principles, and integrating it into the knowledge web of what we already know. One does not become a master of something by simply reading a book, though a good book can open new doors. It has to involve practice—putting the ideas to use, which is an iterative process. With every iteration, we know the ideas better and become increasingly more adept and creative at applying them. The goal of this book is to facilitate the application of its ideas.

This is not a normal textbook. Instead of only discussing *how* something is done, we try to teach the *why*. Our goal is to provide the *intuition* behind the ideas, so that the reader may understand how and when to apply them. There are tons of descriptions and pictures for different folks who prefer to think in different ways. Mathematical formulas are presented in order to make the intuitions precise, and also to bridge this book with other existing offerings of knowledge.

Code examples in this book are given in Python, using a variety of free and open-source packages. Pandas provides a powerful dataframe that is the building block of data science in Python. Scikit-learn is a general purpose machine learning package with extensive coverage of models and feature transformers. Both of these libraries are in-memory. For larger datasets, GraphLab Create provides an on-disk dataframe and associated machine learning library.

This book is meant for folks who are just starting out with data science and machine learning, as well as those with more experience who are looking for ways to systematize their feature engineering efforts. It assumes knowledge of basic machine learning concepts, such as “what is a model," and “what is the difference between supervised and unsupervised learning." It does not assume mastery of mathematics or statistics. Experience with linear algebra, probability distributions, and optimization are helpful, but not necessary.

Feature engineering is a vast topic, and more methods are being invented everyday, particularly in the direction of automatic feature learning. In order to limit the scope of the book to a manageable size, we have had to make some cuts. This book does not discuss Fourier analysis for audio data, though it is a beautiful subject that is closely related to eigen analysis in linear algebra (which we touch upon in Chapter 3 and ???) and random features. We provide an introduction to feature learning via deep learning for image data, but do not go in-depth into the numerous deep learning models under active development. Also out of scope are advanced research ideas like feature hashing, random projections, complex text featurization models such as word2vec and Brown clustering, and latent space models like Latent Dirichlet Analysis and matrix factorization. If those words mean nothing to you, then you are in luck. If the frontiers of feature learning is where your interest lies, then this is probably not the book for you.

The Machine Learning Pipeline

Before diving into feature engineering, let us take a moment to take a look at the overall machine learning pipeline. This will help us get situated in the larger picture of the application. To that end, let us start with a little musing on the basic concepts like *data* and *model*.

Data

What we call "*data*" are observations of real world phenomena. For instance, stock market data might involve observations of daily stock prices, announcements of earnings from individual companies, and even opinion articles from pundits. Personal biometric data can include measurements of our minute-by-minute heart rate, blood sugar level, blood pressure, etc. Customer intelligence data include observations such as "Alice bought two books on Sunday," "Bob browsed these pages on the website," and "Charlie clicked on the special offer link from last week." We can come up with endless examples of data across different domains.

Each piece of data provides a small window into one aspect of reality. The collection of all of these observations give us a picture of the whole. But the picture is messy because it is composed of a thousand little pieces, and there's always measurement noise and missing pieces.

Tasks

Why do we collect data? Usually, there are tasks we'd like to accomplish using data. These tasks might be: "Decide which stocks I should invest in," "Understand how to have a healthier lifestyle," or "Understand my customers' changing tastes, so that my business can serve them better."

The path from data to answers is usually a giant ball of mess. This is because the workflow probably has to pass through multiple steps before resulting in a reasonably useful answer. For instance, the stock prices are observed on the trading floors, aggregated by an intermediary like Thompson Reuters, stored in a database, bought by your company, converted into a Hive store on a Hadoop cluster, pulled out of the store by a script, subsampled, massaged and cleaned by another script, dumped to a file on your desktop, converted to a format that you can try out in your favorite modeling library in R, Python or Scala, predictions dumped back out to a csv file, parsed by an evaluator, iterated multiple times, finally rewritten in C++ or Java by your production team, run on all of the data, and final predictions pumped out to another database.

Figure 1-1. The messy path from data to answers.

Disregarding the mess of tools and systems for a moment, the process involves two mathematical entities that are the bread and butter of machine learning: *models* and *features*.

Models

Trying to understand the world through data is like trying to piece together reality using a noisy, incomplete jigsaw puzzle with a bunch of extra pieces. This is where mathematical modeling—in particular statistical modeling—comes in. The language of statistics contains concepts for many frequent characteristics of data: missing, redundant, or wrong. As such, it is good raw material out of which to build models.

A *mathematical model* of data describes the relationship between different aspects of data. For instance, a model that predicts stock prices might be a formula that maps the company's earning history, past stock prices, and industry to the predicted stock price. A model that recommends music might measure the similarity between users, and recommend the same artists for users who have listened to a lot of the same songs.

Mathematical formulas relate numeric quantities to each other. But raw data is often not numeric. (The action "Alice bought the 'Lord of the Rings' trilogy on Wednesday" is not numeric, neither is the review that she subsequently writes about the book.) So there must be a piece that connects the two together. This is where features come in.

Features

A *feature* is a numeric representation of raw data. There are many ways to turn raw data into numeric measurements. So features could end up looking like a lot of things. The choice of features is tightly coupled with the characteristics of raw data and the choice of the model. Naturally, features must derive from the type of data that is available. Perhaps less obvious is the fact that they are also tied to the model; some models are more appropriate for some type of features, and vice versa. *Feature engineering* is the process of formulating the most appropriate features given the data and the model.

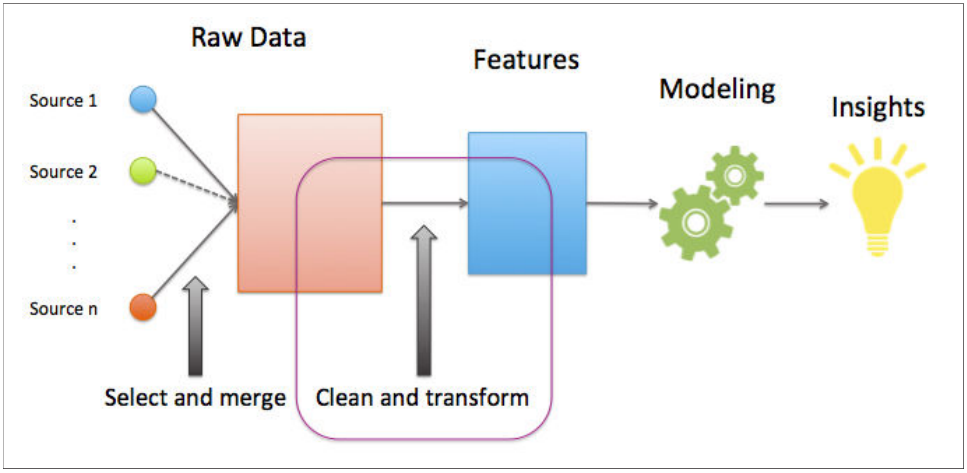


Figure 1-2. The place of feature engineering in the machine learning workflow.

Features and models sit between raw data and the desired insight. In a machine learning workflow, we pick not only the model, but also the features. This is a double-jointed lever, and the choice of one affects the other. Good features make the subsequent modeling step easy and the resulting model more capable of achieving the desired task. Bad features may require a much more complicated model to achieve the same level of performance. In the rest of the this book, we will cover different kinds of features, and discuss their pros and cons for different types of data and models. Without further ado, let's get started!

CHAPTER 2

Basic Feature Engineering for Text Data: Flatten and Filter

Suppose we are trying to analyze the following paragraph.

> Emma knocked on the door. No answer. She knocked again, and just happened to glance at the large maple tree next to the house. There was a giant raven perched on top of it! Under the afternoon sun, the raven gleamed magnificently black. Its beak was hard and pointed, its claws sharp and strong. It looked regal and imposing. It reigned the tree it stood on. The raven was looking straight at Emma with its beady black eyes. Emma was slightly intimidated. She took a step back from the door and tentatively said, "hello?"

The paragraph contains a lot of information. We know that it involves someone named Emma and a raven. There is a house and a tree, and Emma is trying to get into the house but sees the raven instead. The raven is magnificent and noticed Emma, who is a little scared but is making an attempt at communication.

So, which parts of this trove of information are salient features that we should extract? To start with, it seems like a good idea to extract the names of the main characters, Emma and the raven. Next, it might also be good to note the setting of a house, a door, and a tree. What about the descriptions of the raven? What about Emma's actions, knocking on the door, taking a step back, and saying hello?

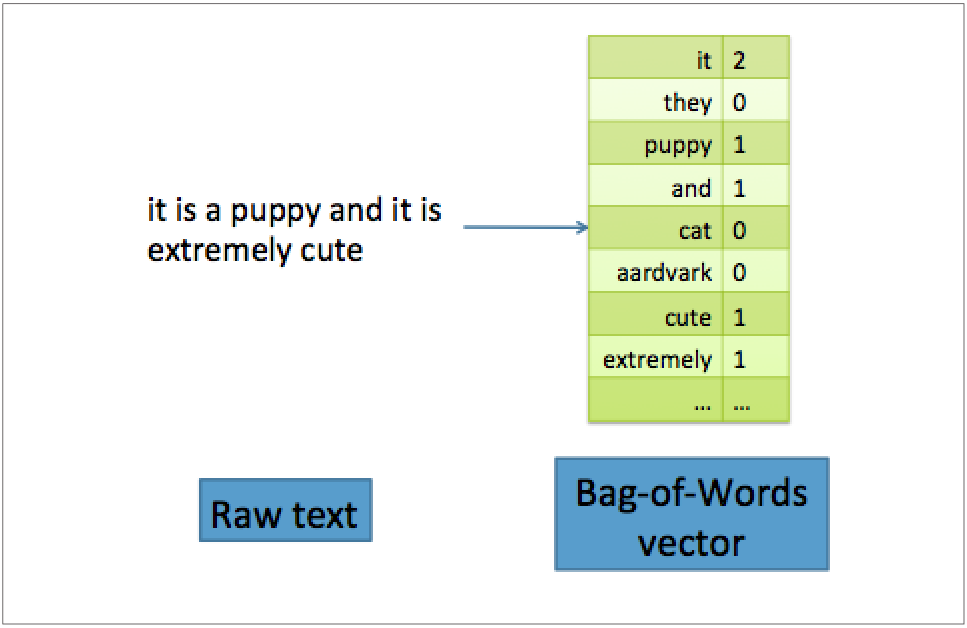
Turning Natural Text into Flat Vectors

Whether it's modeling or feature engineering, simplicity and interpretability are both desirable to have. Simple things are easy to try, and interpretable features and models are easier to debug than complex ones. Simple and interpretable features do not always lead to the most accurate model. But it's a good idea to start simple, and only add complexity when absolutely necessary.

For text data, it turns out that a list of word count statistics called bag-of-words is a great place to start. It's useful for classifying the category or topic of a document. It can also be used in information retrieval, where the goal is to retrieve the set of documents that are relevant to an input text query. Both tasks are well-served by word-level features because the presence or absence of certain words is a great indicator of the topic content of the document.

Bag-of-words

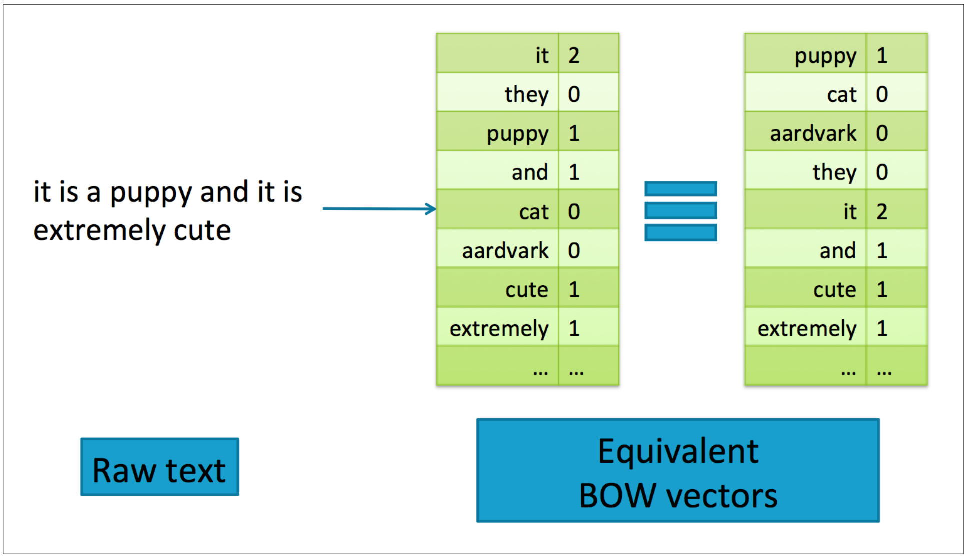
In bag-of-words featurization, a text document is converted into a vector of counts. (A vector is just a collection of n numbers.) The vector contains an entry for every possible word in the vocabulary. If the word, say "aardvark," appears three times in the document, then the feature vector has a count of 3 in the position corresponding to the word. If a word in the vocabulary doesn't appear in the document, then it gets a count of zero. For example, the sentence "it is a puppy and it is extremely cute" has the BOW representation shown in Figure 2-1.



it	2
they	0
puppy	1
and	1
cat	0
aardvark	0
cute	1
extremely	1
...	...

Figure 2-1. Turning raw text into bag-of-words representation

Bag-of-words converts a text document into a flat vector. It is "flat" because it doesn't contain any of the original textual structures. The original text is a sequence of words. But a bag-of-words has no sequence; it just remembers how many times each word appears in the text. Neither does bag-of-words represent any concept of word hierarchy. For example, the concept of "animal" includes "dog," "cat," "raven," etc. But in a bag-of-words representation, these words are all equal elements of the vector.



it	2
they	0
puppy	1
and	1
cat	0
aardvark	0
cute	1
extremely	1
...	...

puppy	1
cat	0
aardvark	0
they	0
it	2
and	1
cute	1
extremely	1
...	...

Figure 2-2. Two equivalent BOW vectors. The ordering of words in the vector is not important, as long as it is consistent for all documents in the dataset.

What is important here is the geometry of data in feature space. In a bag-of-words vector, each word becomes a dimension of the vector. If there are n words in the vocabulary, then a document becomes a point[1] in n-dimensional space. It is difficult to visualize the geometry of anything beyond 2 or 3 dimensions, so we will have to use our imagination. Figure 2-3 shows what our example sentence looks like in the feature space of 2 dimensions corresponding to the words "puppy" and "cute."

[1] Sometimes people call it the document "vector." The vector extends from the original and ends at the specified point. For our purposes, "vector" and "point" are the same thing.

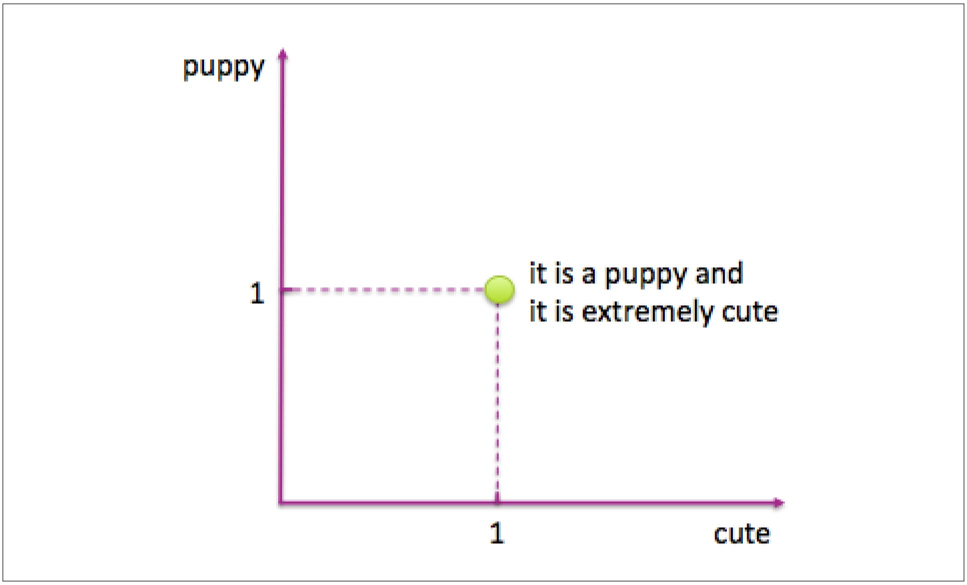


Figure 2-3.
Illustration of a sample text document in feature space

Figure 2-4 shows three sentences in a 3D space corresponding to the words "puppy," "extremely," and "cute."

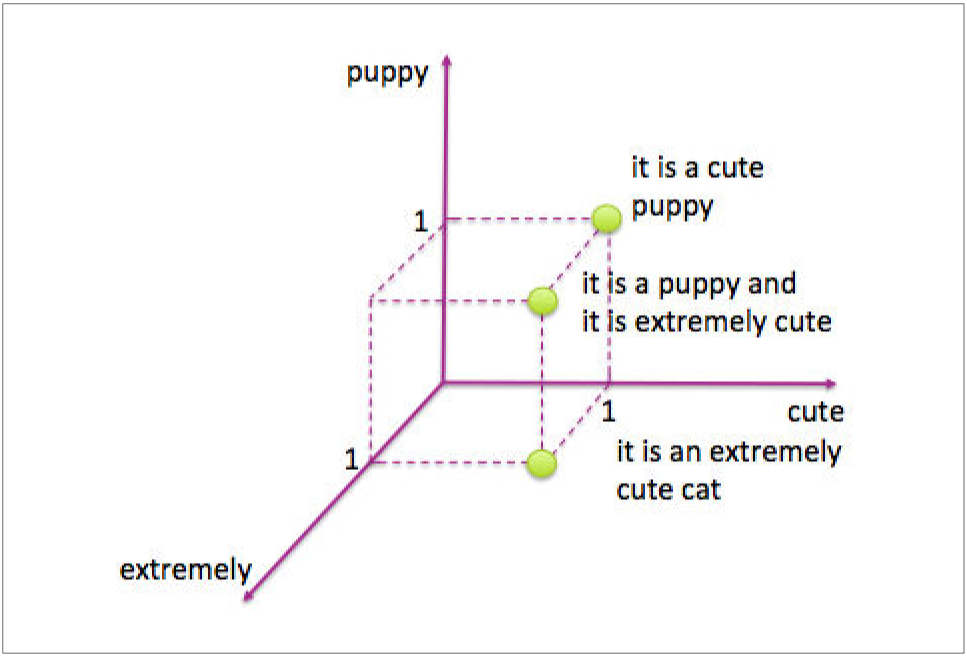

puppy
it is a cute puppy
1
it is a puppy and it is extremely cute
1
cute
1
it is an extremely cute cat
extremely

Figure 2-4. Three sentences in 3D feature space

Figure 2-3 and Figure 2-4 depict data vectors in feature space. The axes denote individual words, which are features under the bag-of-words representation, and the points in space denote data points (text documents). Sometimes it is also informative to look at *feature* vectors in *data* space. A feature vector contains the value of the feature in each data point. The axes denote individual data points, and the points denote feature vectors. Figure 2-5 shows an example. With bag-of-words featurization for text documents, a feature is a word, and a feature vector contains the counts of this word in each document. In this way, a word is represented as a "bag-of-documents." As we shall see in Chapter 3, these bag-of-documents vectors come from the matrix transpose of the bag-of-words vectors.

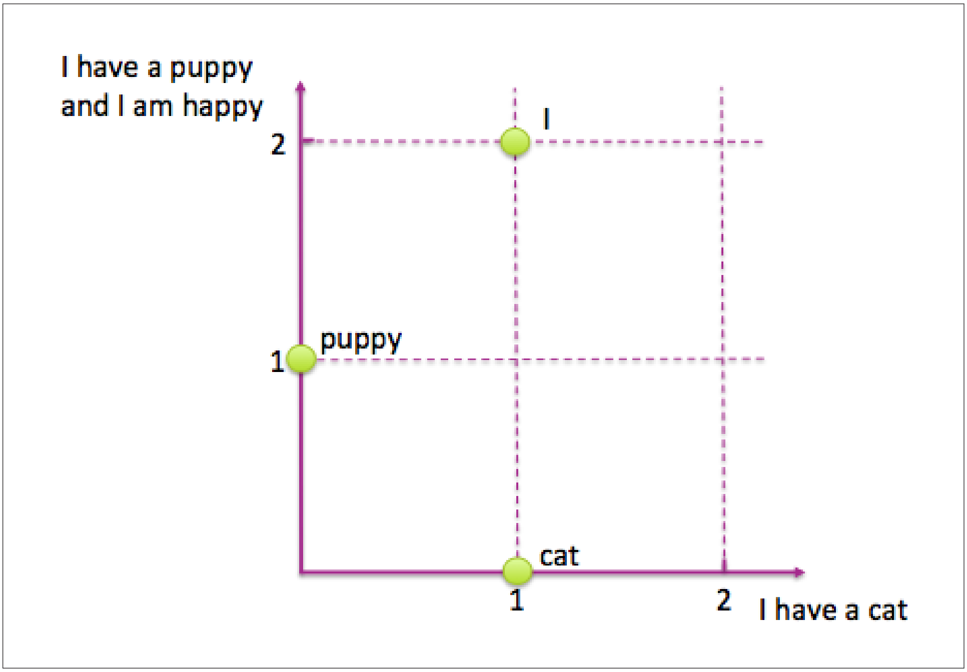


Figure 2-5. Word vectors in document space

Implementing bag-of-words: parsing and tokenization

Now that we understand the concept of bag-of-words, we should talk about its implementation. Most of the time, a text document is represented digitally as a string, which is basically a sequence of characters. In order count the words, the strings need to be first broken up into words. This involves the tasks of *parsing* and *tokenization*, which we discuss next.

Parsing is necessary when the string contains more than plain text. For instance, if the raw data is a webpage, an email, or a log of some sort, then it contains additional structure. One needs to decide how to handle the markups, headers, footers, or the uninteresting sections of the log. If the document is a webpage, then the parser needs to handle URLs. If it is an email, then special fields like From, To, and Subject may require special handling. Otherwise these headers will end up as normal words in the final count, which may not be useful.

After light parsing, the plain text portion of the document can go through tokenization. This turns the string—a sequence of characters—into a sequence of tokens. Each token can then be counted as a word. The tokenizer needs to know what characters indicate that one token has ended and another is beginning. Space characters are usually good separators, as are punctuations. If the text contains tweets, then hashmarks (#) should not be used as separators (also known as *delimiters*).

Sometimes, the analysis needs to operate on sentences instead of entire documents. For instance, n-grams, a generalization of the concept of a word, should not extend beyond sentence boundaries. More complex text featurization methods like word2vec also works with sentences or paragraphs. In these cases, one needs to first parse the document into sentence, then further tokenize each sentence into words.

On a final note, string objects come in various encodings like ASCII or Unicode. Plain English text can be encoded in ASCII. General languages require Unicode. If the document contains non-ASCII characters, then make sure that the tokenizer can handle that particular encoding. Otherwise, the results will be incorrect.

Bag-of-N-Grams

Bag-of-N-Grams, or bag-of-ngrams, is a natural extension of bag-of-words. An n-gram is a sequence of n tokens. A word is essentially a 1-gram, also known as a *unigram*. After tokenization, the counting mechanism can collate individual tokens into word counts, or count overlapping sequences as n-grams. For example, the sentence "Emma knocked on the door" generates the n-grams "Emma knocked," "knocked on," "on the," "the door."

N-grams retain more of the original sequence structure of the text, therefore bag-of-ngrams can be more informative. However, this comes at a cost. Theoretically, with k unique words, there could be k^2 unique 2-grams (also called *bigrams*). In practice, there are not nearly so many, because not every word can follow every other word. Nevertheless, there are usually a lot more distinct n-grams (n > 1) than words. This means that bag-of-ngrams is a much bigger and sparser feature space. It also means that n-grams are more expensive to compute, store, and model. The larger n is, the richer the information, and the more expensive the cost.

To illustrate how the number of n-grams grow with increasing *n*, let's compute n-grams on the *Yelp reviews dataset*. Round 6 of the Yelp dataset challenge contains close to 1.6 million reviews of businesses in six U.S. cities. We compute the n-grams of the first 10,000 reviews using Pandas and the `CountVectorizer` transformer in scikit-learn.

Example 2-1. Example: computing n-grams.

```
>>> import pandas
>>> import json
>>> from sklearn.feature_extraction.text import CountVectorizer

# Load the first 10,000 reviews
>>> f = open('data/yelp/v6/yelp_dataset_challenge_academic_dataset/yelp_academic_dataset_review.json')
>>> js = []
>>> for i in range(10000):
...     js.append(json.loads(f.readline()))
```

```
>>> f.close()
>>> review_df = pd.DataFrame(js)

# Create feature transformers for unigram, bigram, and trigram.
# The default ignores single-character words, which is useful in practice because it trims
# uninformative words. But we explicitly include them in this example for illustration purposes.
>>> bow_converter = CountVectorizer(token_pattern='(?u)\\b\\w+\\b')
>>> bigram_converter = CountVectorizer(ngram_range=(2,2), token_pattern='(?u)\\b\\w+\\b')
>>> trigram_converter = CountVectorizer(ngram_range=(3,3), token_pattern='(?u)\\b\\w+\\b')

# Fit the transformers and look at vocabulary size
>>> bow_converter.fit(review_df['text'])
>>> words = bow_converter.get_feature_names()
>>> bigram_converter.fit(review_df['text'])
>>> bigrams = bigram_converter.get_feature_names()
>>> trigram_converter.fit(review_df['text'])
>>> trigrams = trigram_converter.get_feature_names()
>>> print (len(words), len(bigrams), len(trigrams))
26047 346301 847545

# Sneak a peek at the ngrams themselves
>>> words[:10]
['0', '00', '000', '0002', '00am', '00ish', '00pm', '01', '01am', '02']

>>> bigrams[-10:]
['zucchinis at',
 'zucchinis took',
 'zucchinis we',
 'zuma over',
 'zuppa di',
 'zuppa toscana',
 'zuppe di',
 'zurich and',
 'zz top',
 'à la']

>>> trigrams[:10]
['0 10 definitely',
 '0 2 also',
 '0 25 per',
 '0 3 miles',
 '0 30 a',
 '0 30 everything',
 '0 30 lb',
 '0 35 tip',
 '0 5 curry',
 '0 5 pork']
```

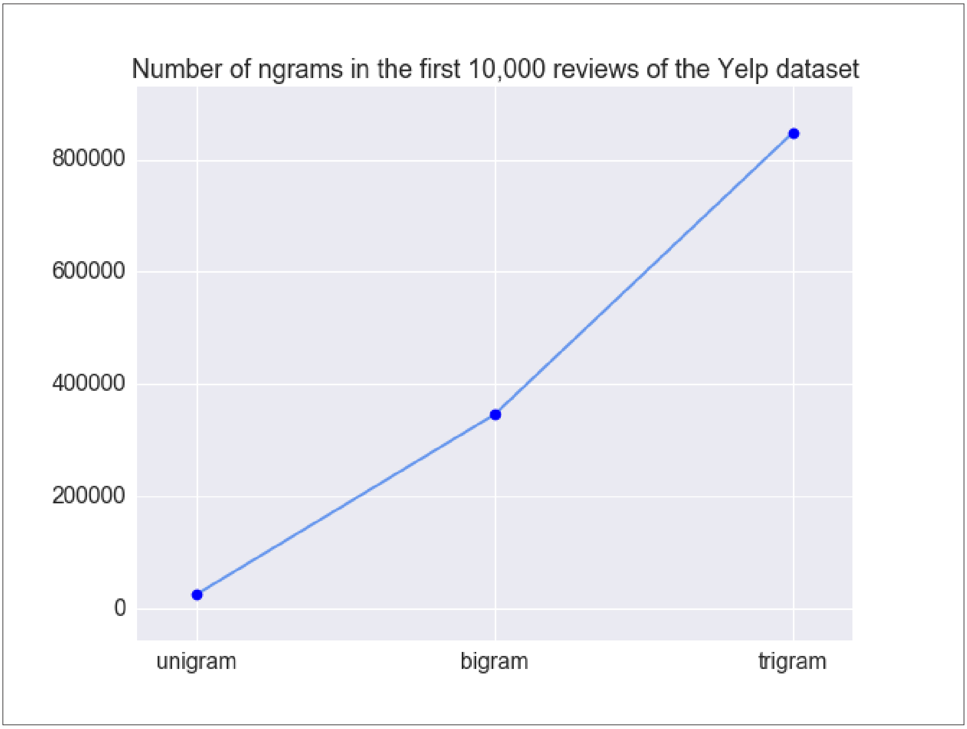


Figure 2-6. Number of unique n-grams in the first 10,000 reviews of the Yelp dataset.

Collocation Extraction for Phrase Detection

The main reason why people use n-grams is to capture useful phrases. In computational Natural Language Processing, the concept of a useful phrase is called *collocation*. In the words of Manning and Schütze (1999: 141): "A COLLOCATION is an expression consisting of two or more words that correspond to some conventional way of saying things."

Collocations are more meaningful than the sum of its parts. For instance, "strong tea" has a different meaning beyond "great physical strength" and "tea," therefore it is considered a collocation. The phrase "cute puppy," on the other hand, means exactly the sum of its parts: "cute" and "puppy." Hence it is not considered a collocation.

Collocations do not have to be consecutive sequences. The sentence "Emma knocked on the door" is considered to contain the collocation "knock door." Hence not every collocation is an n-gram. Conversely, not every n-gram is deemed a meaningful collocation.

Because collocations are more than the sum of its parts, their meaning cannot be adequately captured by individual word counts. Bag-of-words falls short as a representation. Bag-of-ngrams are also problematic because they capture too many meaningless sequences (consider “this is” in the bag-of-ngrams example) and not enough of the meaningful ones.

Collocations are useful as features. But how does one discover and extract them from text? One way is to pre-define them. If we tried really hard, we could probably find comprehensive lists of idioms in various languages, and we can look through the text for any matches. It would be very expensive, but it would work. If the corpus is very domain specific and contains esoteric lingo, then this might be the preferred method. But the list would require a lot of manual curation, and it would need to be constantly updated for evolving corpora. For example, it probably wouldn’t be very realistic for analyzing tweets, or for blogs and articles.

Since the advent of statistical NLP in the last two decades, people have opted more and more for statistical methods for finding phrases. Instead of establishing a fixed list of phrases and idiomatic sayings, statistical collocation extraction methods rely on the ever evolving data to reveal the popular sayings of the day.

Frequency-based methods

A simple hack is to look at the most frequently occurring n-grams. The problem with this approach is that the most frequently occurring ones may not be the most useful ones. Table 2-1 shows the most popular bigrams (n=2) in the entire Yelp reviews dataset. As we can see, the most top 10 frequently occurring bigrams by document count are very generic terms that don’t contain much meaning.

Table 2-1. Most frequently occurring 2-grams in a Yelp reviews dataset

Bigram	Document Count
of the	450849
and the	426346
in the	397821
it was	396713
this place	344800
it s	341090
and i	332415
on the	325044
i was	285012
for the	276946

Hypothesis testing for collocation extraction

Raw popularity count is too crude of a measure. We have to find more clever statistics to be able to pick out meaningful phrases easily. The key idea is to ask whether two words appear together more often than by chance. The statistical machinery for answering this question is called a *hypothesis test*.

Hypothesis testing is a way to boil noisy data down to "yes" or "no" answers. It involves modeling the data as samples drawn from random distributions. The randomness means that one can never be 100% sure about the answer; there's always the chance of an outlier. So the answers are attached to a probability. For example, the outcome of a hypothesis test might be "these two datasets come from the same distribution with 95% probability." For a gentle introduction to hypothesis testing, see the Khan Academy's tutorial on Hypothesis Testing and p-Values.

In the context of collocation extraction, many hypothesis tests have been proposed over the years. One of the most successful methods is based on the likelihood ratio test (Dunning, 1993). It tests whether the probability of seeing the second word is independent of the first word.

Null hypothesis (independent): $P(w_2 \mid w_1) = p = P(w_2 \mid \text{not } w_1)$

Alternate hypothesis (not independent): $P(w_2 \mid w_1) = p_1 \neq p_2 = P(w_2 \mid \text{not } w_1)$

The method estimates the values of p, p_1, and p_2, then computes the probability of the observed count of the word pair under the two hypothesis. The final statistic is the log of the ratio between the two.

$$\log \lambda = \log \frac{L(H_{\text{null}})}{L(H_{\text{alternate}})}$$

Normal hypothesis testing procedure would then test whether the value of the statistic is outside of an allowable range, and decide whether or not to reject the null hypothesis (i.e., call a winner). But in this context, the test statistic (the likelihood ratio score) is often used to simply rank the candidate word pairs. One could then keep the top ranked candidates as features.

There is another statistical approach based on point-wise mutual information. But it is very sensitive to rare words, which are always present in real-world text corpora. Hence it is not commonly used.

Note that all of the statistical methods for collocation extraction, whether using raw frequency, hypothesis testing, or point-wise mutual information, operate by filtering a list of candidate phrases. The easiest and cheapest way to generate such a list is by counting n-grams. It's possible to generate non-consecutive sequences (see chapter on frequent sequence mining) [replace with cross-chapter reference], but they are expensive to compute. In practice, even for consecutive n-grams, people rarely go beyond bi-grams or tri-grams because there are too many of them, even after filtering. (See "Filtering for Cleaner Features" on page 26.) To generate longer phrases, there are other methods such as chunking or combining with part-of-speech tagging.

[to-do: chunking and pos tagging]

Quick summary

Bag-of-words is simple to understand, easy to compute, and useful for classification and search tasks. But sometimes single words are too simplistic to encapsulate some information in the text. To fix this problem, people look to longer sequences. Bag-of-ngrams is a natural generalization of bag-of-words. The concept is still easy to understand, and it's just as easy to compute as bag-of-words.

Bag-of-ngrams generates a lot more distinct ngrams. It increases feature storage cost, as well as the computation cost of the model training and prediction stages. The number of data points remain the same, but the dimension of the feature space is now much larger. Hence the density of data is much more sparse. The higher n is, the higher the storage and computation cost, and the sparser the data. For these reasons, longer n-grams do not always lead to improved model accuracy (or any other performance measure). People usually stop at n=2 or 3. Longer n-grams are rarely used.

One way to combat the increase in sparsity and cost is to filter the n-grams and retain only the most meaningful phrases. This is the goal of collocation extraction. In theory, collocations (or phrases) could form non-consecutive token sequences in the text. In practice, however, looking for non-consecutive phrases has a much higher computation cost for not much gain. So collocation extraction usually starts with a candidate list of bigrams and utilizes statistical methods to filter them.

All of these methods turn a sequence of text tokens into a disconnected set of counts. A set has much less structure compared to a sequence; they lead to flat feature vectors.

Filtering for Cleaner Features

Raw tokenization and counting generates lists of simple words or n-grams, which requires filtering to be more usable. Phrase detection, as discussed, can be seen as a particular bigram filter. Here are a few more ways to perform filtering.

Stopwords

Classification and retrieval do not usually require an in-depth understanding of the text. For instance, in the sentence "Emma knocked on the door," the words "on" and "the" don't contain a lot of information. The pronouns, articles, and prepositions do not add much value most of the time. The popular Python NLP package NLTK contains a linguist-defined stopword list for many languages. (You will need to install NLTK and run ‘nltk.download()' to get all the goodies.) Various stopword lists can also be found on the web. For instance, here are some sample words from the English stopword list:

> **Sample words from the nltk stopword list**
>
> a, about, above, am, an, been, didn’t, couldn’t, i’d, i’ll, itself, let’s, myself, our, they, through, when’s, whom, ...

Note that the list contains apostrophes and the words are un-capitalized. In order to use it as is, the tokenization process must not eat up apostrophes, and the words needs to be converted to lower case.

Frequency-based filtering

Stopword lists are a way of weeding out common words that make for vacuous features. There are other, more statistical ways of getting at the concept of “common words.” In collocation extraction, we see methods that depend on manual definitions, and those that use statistics. The same idea carries to word filtering. We can use frequency statistics here as well.

Frequent words

Frequency statistics are great for filtering out corpus-specific common words as well as general-purpose stopwords. For instance, the phrase “New York Times” and each of the individual words appear frequently in the New York Times articles dataset. The word “house” appears often in the phrase “House of Commons” in the Hansard corpus of Canadian parliament debates, a popular dataset that is used for statistical machine translation, because it contains both an English and a French version of all documents. These words are meaningful in the general language, but not within the corpus. A hand-defined stopword list will catch the general stopwords, but not corpus-specific ones.

Table 2-2 lists the 40 most frequent words in the Yelp reviews dataset. Here, frequency is taken to be the number of documents (reviews) they appear in, not by their count within a document. As we can see, the list covers many stopwords. It also contains some surprises. “s” and “t” are on the list because we used the apostrophe as a tokenization delimiter, and words such as “Mary’s” or “didn’t” got parsed as “Mary s” and “didn t.” The words “good,” “food,” and “great” each appears in around a third of the reviews. But we might want to keep them around because they are very useful for sentiment analysis or business categorization.

Table 2-2. Most frequent words in the Yelp reviews dataset

Rank	Word	Document Frequency	Rank	Word	Document Frequency
1	the	1416058	21	t	684049
2	and	1381324	22	not	649824
3	a	1263126	23	s	626764
4	i	1230214	24	had	620284
5	to	1196238	25	so	608061
6	it	1027835	26	place	601918
7	of	1025638	27	good	598393
8	for	993430	28	at	596317
9	is	988547	29	are	585548
10	in	961518	30	food	562332
11	was	929703	31	be	543588
12	this	844824	32	we	537133
13	but	822313	33	great	520634
14	my	786595	34	were	516685
15	that	777045	35	there	510897
16	with	775044	36	here	481542
17	on	735419	37	all	478490
18	they	720994	38	if	475175
19	you	701015	39	very	460796
20	have	692749	40	out	460452

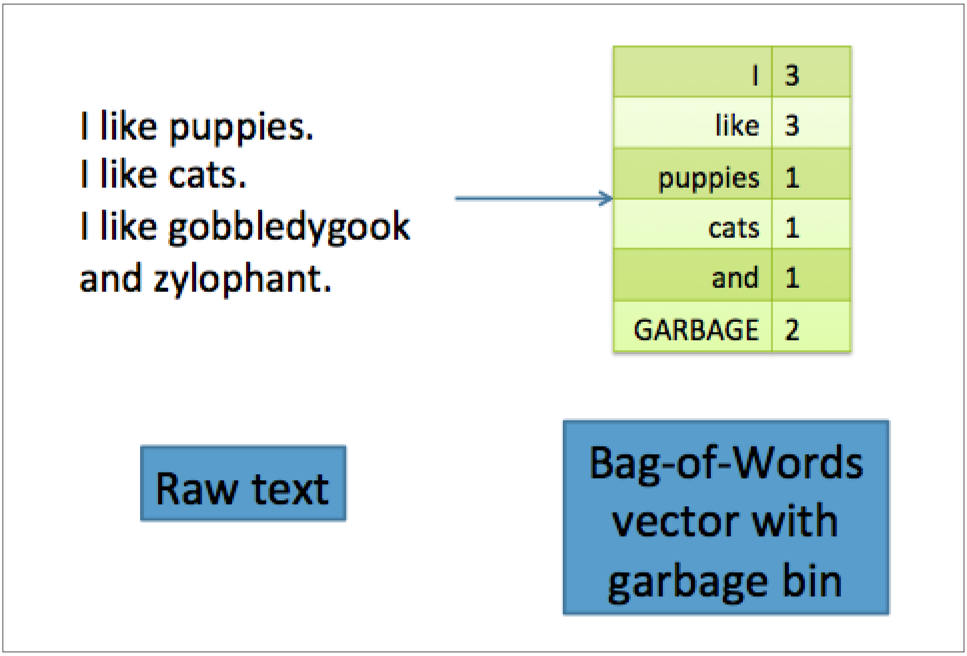
The most frequent words can reveal parsing problems and highlight normally useful words that happens to appear too many times in this corpus. For instance, the most frequent word in the New York Times Corpus is “times." In practice, it helps to combine frequency-based filtering with a stopword list. There is also the tricky question of where to place the cut-off. Unfortunately there is no universal answer. Most of the time the cut-off needs to be determined manually, and may need to be re-examined when the dataset changes.

Rare words

Depending on the task, one might also need to filter out rare words. To a statistical model, a word that appears in only one or two documents is more like noise than useful information. For example, suppose the task is to categorize businesses based on their Yelp reviews, and a single review contains the word “gobbledygook.” How would one tell, based on this one word, whether the business is a restaurant, a beauty salon, or a bar? Even if we knew that the business in this case happened to be a bar, it would probably be a mistake to classify as such for other reviews that contain the word "gobbledygook."

Not only are rare words unreliable as predictors, they also generate computational overhead. The set of 1.6 million Yelp reviews contains 357,481 unique words (tokenized by space and punctuation characters), 189,915 of which appear in only one review, and 41,162 in two reviews. Over 60% of the vocabulary occurs rarely. This is a so-called *heavy-tailed distribution*, and it is very common in real-world data. The training time of many statistical machine learning models scales linearly with the number of features, and some models are quadratic or worse. Rare words incur a large computation and storage cost at not much additional gain.

Rare words can be easily identified and trimmed based on word count statistics. Alternatively, their counts can be aggregated into a special garbage bin, which can serve as an additional feature. Figure 2-7 demonstrates this representation on a short document that contains a bunch of usual words and two rare words "gobbledygook" and "zylophant." The usual words retain their own counts, which can be further filtered by stopword lists or other frequency based methods. The rare words lose their identity and get grouped into a garbage bin feature.



I	3
like	3
puppies	1
cats	1
and	1
GARBAGE	2

Figure 2-7. Bag-of-words feature vector with a garbage bin

Since one wouldn't know which words are rare until the whole corpus has been counted, the garbage bin feature will need to be collected as a post-processing step.

Since this book is about feature engineering, our focus is on features. But the concept of rarity also applies to data points. If a text document is very short, then it likely contains no useful information and should not be used when training a model. One must use caution when applying this rule. The Wikipedia dump contains many pages that are incomplete stubs, which are probably safe to filter out. Tweets, on the other hand, are inherently short, and require other featurization and modeling tricks.

Stemming

One problem with simple parsing is that different variations for the same word get counted as separate words. For instance, "flower" and "flowers" are technically different tokens, and so are "swimmer," "swimming," and "swim," even though they are very close in meaning. It would be nice if all of these different variations get mapped to the same word.

Stemming is an NLP task that tries to chop words down to its basic linguistic word stem form. There are different approaches. Some are based on linguistic rules, others based on observed statistics. A subclass of algorithms known as lemmatization combines part-of-speech tagging and linguistic rules.

Porter stemmer is the most widely used free stemming tool for the English language. The original program is written in ANSI C, but many other packages have since wrapped it to provide access to other languages. Most stemming tools focus on the English language, though efforts are ongoing for other languages.

Here is an example of running the Porter stemmer through the NLTK Python package. As we can see, it handles a large number of cases, including transforming "sixties" and "sixty" to the same root "sixti." But it's not perfect. The word "goes" is mapped to "goe," while "go" is mapped to itself.

```
>>> import nltk
>>> stemmer = nltk.stem.porter.PorterStemmer()
>>> stemmer.stem('flowers')
u'lemon'
>>> stemmer.stem('zeroes')
u'zero'
>>> stemmer.stem('stemmer')
u'stem'
>>> stemmer.stem('sixties')
u'sixti'
>>> stemmer.stem('sixty')
u'sixty'
>>> stemmer.stem('goes')
```

```
u'goe'
>>> stemmer.stem('go')
u'go'
```

Stemming does have a computation cost. Whether the end benefit is greater than the cost is application-dependent.

Summary

In this chapter, we dip our toes into the water with simple text featurization techniques. These techniques turn a piece of natural language text—full of rich semantics structure—into a simple flat vector. We introduce ngrams and collocation extraction as methods that add a little more structure into the flat vector. We also discuss a number of common filtering techniques to clean up the vector entries. The next chapter goes into a lot more detail about another common text featurization trick called *tf-idf*. Subsequent chapters will discuss more methods for adding structure back into a flat vector.

Bibliography

Dunning, Ted. 1993. "Accurate methods for the statistics of surprise and coincidence." *ACM Journal of Computational Linguistics, special issue on using large corpora*, 19:1 (61—74).

"Hypothesis Testing and p-Values." Khan Academy, accessed May 31, 2016, https://www.khanacademy.org/math/probability/statistics-inferential/hypothesis-testing/v/hypothesis-testing-and-p-values.

Manning, Christopher D. and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing*. Cambridge, Massachusettes: MIT Press.

CHAPTER 3

The Effects of Feature Scaling: From Bag-of-Words to Tf-Idf

Bag-of-words is simple to generate but far from perfect. If we count all words equally, then some words end up being emphasized more than we need. Recall our example of Emma and the raven from Chapter 2. We'd like a document representation that emphasizes the two main characters. The words "Emma" and "raven" both appear 3 times, but "the" appears a whopping 8 times, "and" appears 5 times, and "it" and "was" both appear 4 times. The main characters do not stand out by simple frequency count alone. This is problematic.

It would also be nice to pick out words such as "magnificently," "gleamed," "intimidated," "tentatively," and "reigned," because they help to set the overall tone of the paragraph. They indicate sentiment, which can be very valuable information to a data scientist. So, ideally, we'd have a representation that highlights *meaningful* words.

Tf-Idf : A Simple Twist on Bag-of-Words

Tf-idf is a simple twist on top of bag-of-words. It stands for *term frequency—inverse document frequency*. Instead of looking at the raw counts of each word in each document, tf-idf looks at a normalized count where each word count is divided by the number of documents this word appears in.

bow(w, d) = # times word w appears in document d

tf-idf (w, d) = bow(w, d) * N / (# documents in which word w appears)

N is the total number of documents in the dataset. The fraction (N / # documents ...) is what's known as the *inverse document frequency*. If a word appears in many documents, then its inverse document frequency is close to 1. If a word appears in just a few documents, then the inverse document frequency is much higher.

Alternatively, we can take a log transform instead using the raw inverse document frequency. Logarithm turns 1 into 0, and makes large numbers (those much greater than 1) smaller. (More on this later.) If we define tf-idf as

tf-idf (w, d) = bow(w, d) * log (N / # documents in which word w appears),

then a word that appears in every single document will be effectively zeroed out, and a word that appears in very few documents will have an even larger count than before.

Let's look at some pictures to understand what it's all about. Figure 3-1 shows a simple example that contains four sentences: "it is a puppy," "it is a cat," "it is a kitten," and "that is a dog and this is a pen." We plot these sentences in the feature space of three words: "puppy," "cat", and "is."

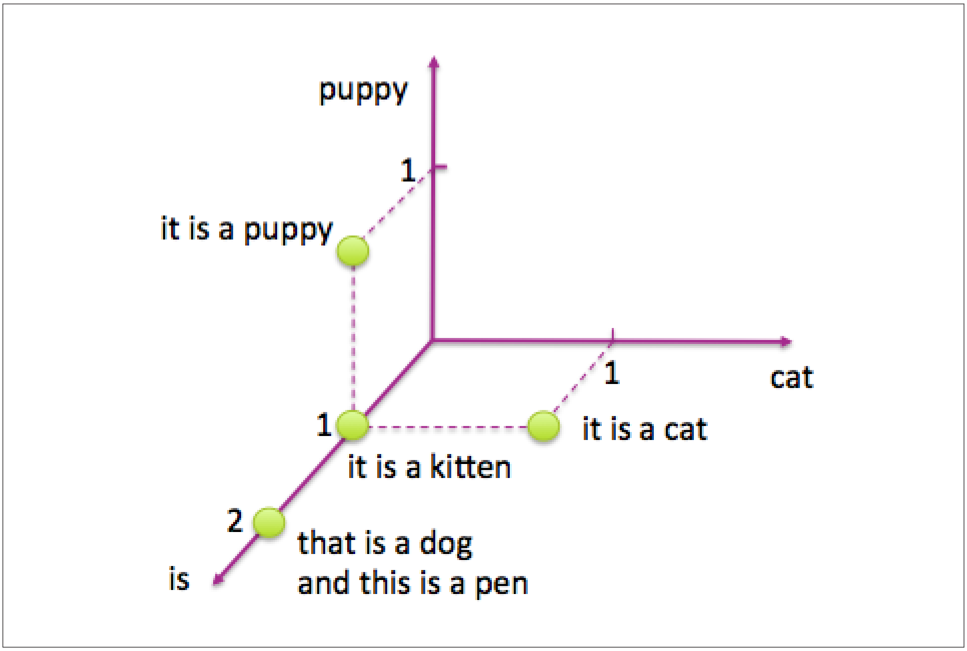


Figure 3-1. Four sentences about dog and cat

Now let's look at the same four sentences in tf-idf representation using the log transform for the inverse document frequency. Figure 3-2 shows the documents in feature space. Notice that the word "is" is effectively eliminated as a feature since it appears in all sentences in this dataset. Also, because they each appear in only one sentence out of the total four, the words "puppy" and "cat" are now counted higher than before (log(4) = 1.38... > 1). Thus **tf-idf makes rare words more prominent and effectively ignores common words**. It is closely related to the frequency-based filtering methods in Chapter 2, but much more mathematically elegant than placing hard cut-off thresholds.

Intuition Behind Tf-Idf

Tf-idf makes rare words more prominent and effectively ignores common words.

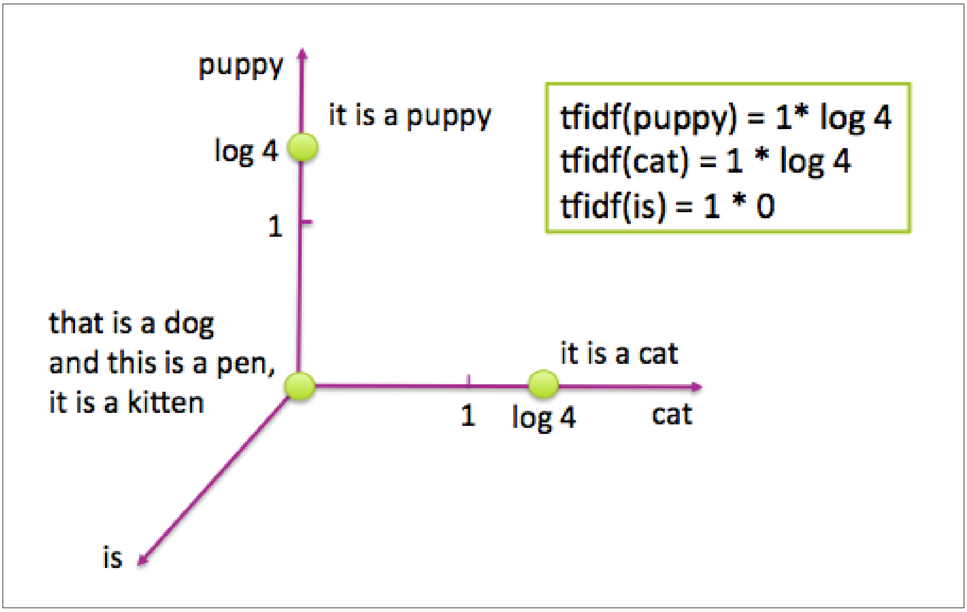


Figure 3-2. Tf-idf representation of the sentences in Figure 3-1

Feature Scaling

Tf-idf is an example of a type of feature engineering known as *feature scaling*. As the name suggests, feature scaling changes the scale of the feature. Sometimes people also call it *feature normalization*. There are several types of common scaling operations, each result in a different distribution of feature values.

Min-max scaling

The formula for min-max scaling is

$$\tilde{x} = \frac{x - \min(x)}{\max(x) - \min(x)}.$$

When we talk about feature scaling, we are usually dealing with just one feature. Let x be an individual feature value (i.e., a value of the feature in some data point), and min(x) and max(x) respectively the minimum and maximum over all values for this feature in this dataset. Min-max scaling squeezes (or stretches) all feature values to be within the range of [0, 1]. Figure 3-3 shows what min-max scaling looks like.

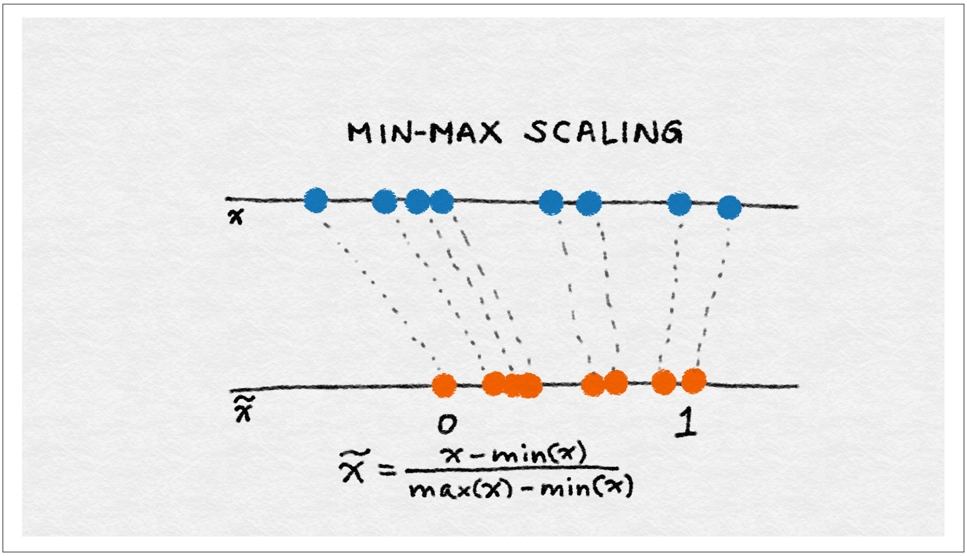


Figure 3-3. Min-max scaling

Standardization (variance scaling)

Here is the formula for standardization:

$$\tilde{x} = \frac{x - \mathrm{mean}(x)}{\mathrm{var}(x)}.$$

It subtracts off the mean of the feature (over all data points) and divides by the variance. Hence it can also be called “variance scaling." The resulting scaled feature is standardized to have a mean of 0 and a variance of 1. If the original feature has a Gaussian distribution, then the scaled feature is a standard Gaussian, a.k.a. standard normal. Figure 3-4 contains an illustration of standardization.

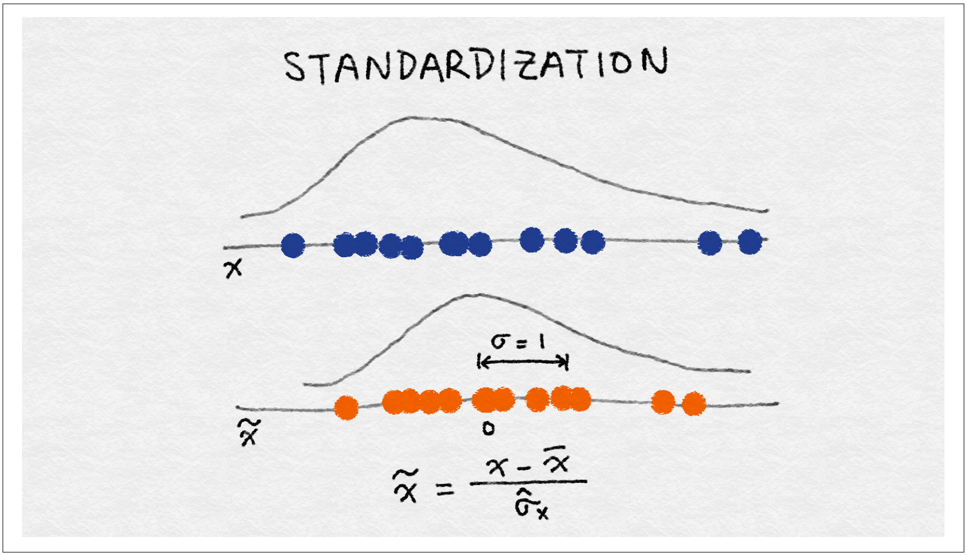


Figure 3-4. Illustration of feature standardization

L^2 normalization

L^2 is another name for the Euclidean norm. Here is the formula:

$$\tilde{x} = \frac{x}{\| x \|_2} .$$

The L^2 norm measures the length of the vector in coordinate space. The definition can be derived from the well-known Pythagorean theorem that gives us the length of the hypotenuse of a right triangle given the lengths of the sides.

$$\| x \|_2 = \sqrt{x_1^2 + x_2^2 + \ . \ . \ . \ + x_m^2}$$

In other words, it sums the square of values of the features across data points, then takes the square root. L^2 normalization can also be called L^2 scaling. (Losely speaking, *scaling* means multiplying by a constant, whereas *normalization* could involve a number of operations.) Figure 3-5 illustrates L^2 normalization.

As a result of L^2 normalization, the feature column now has norm 1. Note that the illustration in Figure 3-5 is in data space, not feature space. One can also do L^2 normalization for the data point instead of the feature, which would result in data vectors that with unit norm (norm of 1). (See the discussion in "Bag-of-words" on page 16 about the complementary nature of data vectors and feature vectors.)

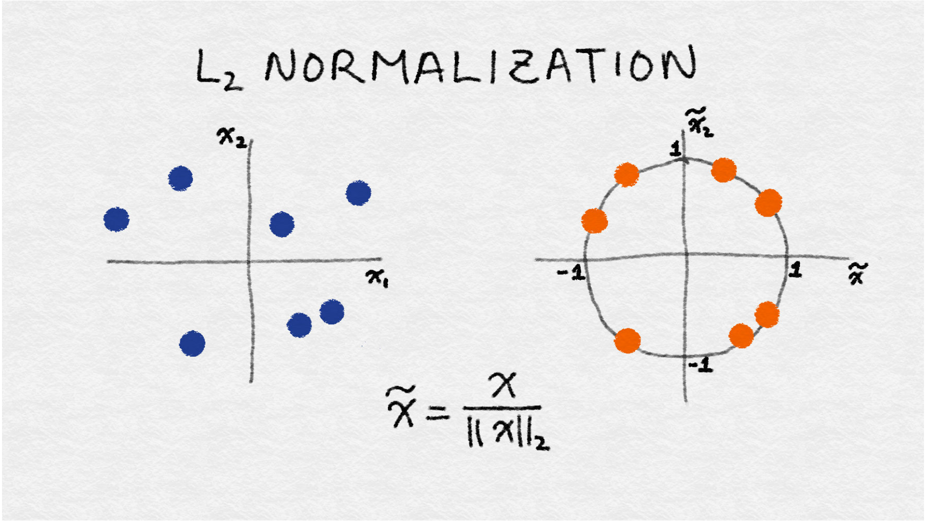


Figure 3-5. Illustration of L^2 feature normalization

Putting it to the Test

How well does feature scaling work in practice? Let's compare the performance of scaled and unscaled features in a simple text classification task. Time for some code!

For this exercise, we take data from round 6 of the *Yelp dataset challenge* and create a much smaller classification dataset. The Yelp dataset contains user reviews of businesses from ten cities across North America and Europe. Each business is labeled with zero or more categories. Here are some relevant statistics about the dataset.

Statistics of Yelp Dataset, Round 6

- There are 782 business categories.
- The full dataset contains 1,569,264 (≈1.6M) reviews and 61,184 (61K) businesses.
- "Restaurants" (990,627 reviews) and "Nightlife" (210,028 reviews) are the most popular categories, review-count-wise.
- No business is categorized as both a restaurant and a nightlife venue. So there is no overlap between the two groups of reviews.

Example 3-1. Loading and cleaning the Yelp reviews dataset in Python

```
import graphlab as gl
# Load the data into an SFrame, one line per JSON blurb
sf = gl.SFrame.read_csv('./yelp/v6/yelp_academic_dataset_review.json',
                        header=False,
                        delimiter='\n')
# Unpack the JSON dictionary into multiple columns
sf = sf.unpack('X1', column_name_prefix='')

# Load and unpack information about the businesses
business_sf = gl.SFrame.read_csv('./yelp/v6/yelp_academic_dataset_business.json',
                                 header=False)
business_sf = business_sf.unpack('X1', column_name_prefix='')

# Join the SFrames so that we have info about the business for each review
combined = sf.join(business_sf, on='business_id')
# The categories are originally in dictionary form, stack them
# so that each category label has its own row
combined_stack = combined.stack('categories', new_column_name='category')

# Pull out the restaurant and nightlight reviews.
restaurant_nightlife = combined_stack[(combined_stack['category'] == 'Restaurants')
                                     | (combined_stack['category'] == 'Nightlife')]
restaurant_nightlife = restaurant_nightlife[['business_id',
                                             'name',
                                             'city',
                                             'state',
                                             'stars',
                                             'text',
                                             'category']]
# Drop those without review text
restaurant_nightlife = restaurant_nightlife[restaurant_nightlife['text'] != '']
```

Creating a classification dataset

Let's see whether we can use the reviews to categorize the business as either a restaurant or a nightlife venue. To save on training time, we can take a subset of the reviews. There is a large difference in review count between the two categories. This is called an *class-imbalanced dataset*. Imbalanced datasets are problematic for modeling because the model would spend most of its effort fitting to the larger class. Since we have plenty of data in both classes, a good way to resolve the problem is to down sample the larger class (Restaurants) to be roughly the same size as the smaller class (Nightlife). Here is an example workflow.

1. Drop all the empty reviews.
2. Take a random sample of 16% of nightlife reviews and 3.4% of restaurant reviews (percentages chosen so the number of examples in each class is roughly equal).
3. Create a 70/30 train-test split of this dataset. In this example, the training set ends up with 46,997 reviews, and the test set 19,920 reviews.

4. The training data contains 81,079 unique words; this is the number of features in the bag-of-words representation. The linear model also include a bias term, making the total number of features to be 81,080.

Example 3-2. Creating a classification data set

```
def create_small_train_test(data):
    # First pull out a roughly equal number of nightlife and restaurant reviews
    # Then further divide into training and testing sets
    subset_nightlife, nightlife_rest =
        data[data['category'] == 'Nightlife'].random_split(0.16)
    subset_restaurant, restaurants_rest =
        data[data['category'] == 'Restaurants'].random_split(0.034)
    nightlife_train, nightlife_test = subset_nightlife.random_split(0.7)
    restaurant_train, restaurant_test = subset_restaurant.random_split(0.7)
    training_data = nightlife_train.append(restaurant_train)
    test_data = nightlife_test.append(restaurant_test)

    # The training algorithm expects a randomized ordering of data. Sort by name
    # so that the nightlife reviews are not all in the first half of the rows.
    # This doesn't matter for testing but could be crucial for certain
    # implementations of the training algorithm.
    training_data.sort('name')

    # Compute bag-of-words representation for both training and testing data
    delims = [' ', '\t', '\r', '\n', '\x0b', '\x0c',
              ',', '.', '!', ':', ';', '"', '-', '+']
    training_data['bow'] =
        gl.text_analytics.count_words(training_data['text'], delimiters=delims)
    test_data['bow'] =
        gl.text_analytics.count_words(test_data['text'], delimiters=delims)

    # Create a tf-idf transformer
    tfidf_transformer = gl.feature_engineering.TFIDF(features='bow',
                                                     output_column_prefix='tfidf')
    # Collect statistics on the training set and
    # add tfidf feature column to the training data
    training_data = tfidf_transformer.fit_transform(training_data)
    # Transform test data using training statistics
    test_data = tfidf_transformer.transform(test_data)

    return training_data, test_data
```

Implementing tf-idf and feature scaling

The goal of this experiment is to compare the effectiveness of bag-of-words, tf-idf, and L^2 normalization for linear classification. Note that doing tf-idf then L^2 normalization is the same as doing L^2 normalization alone. So we only need to test 3 sets of features: bag-of-words, tf-idf, and word-wise L^2 normalization on top of bag-of-words.

We convert the text of each review into a bag-of-words representation using GraphLab Create's `text_analytics.count_words` function. Scikit-learn has a similar function called `CountVectorizer`. All text featurization methods implicitly depend on a tokenizer, which is the module that converts a text string into a list of tokens (words). In this example, we tokenize by space characters and common punctuation marks.

Feature Scaling on the Test Set

Here is a subtle point about feature scaling: it requires knowing feature statistics that we most likely do not know in practice, such as the mean, variance, document frequency, L^2 norm, etc. In order to compute the tf-idf representation, we have to compute the inverse document frequencies based on the *training* data and use these statistics to scale both training and test data.

Next, we want to compute the tf-idf representation of all review text. GraphLab Create and scikit-learn both have feature transformers that can fit to a training set (and remember the statistics) and transform any dataset (training or testing). We fit a TFIDF transformer to combined training and validation data and transform the training, validation, and test datasets.

When we use training statistics to scale test data, the result will look a little fuzzy. Min-max scaling on the test set no longer neatly maps to zero and one. L^2 norms, mean, and variance statistics will all look a little off. This is less problematic than missing data. For instance, the test set may contain words that are not present in the training data, and we would have no document frequency to use for the new words. Different implementations of tf-idf may deal with this differently. There are several simple options: print an error and do nothing (scikit-learn 0.17), or drop the new words in the test set (GraphLab Create 1.7). Dropping the new words may seem irresponsible but is a reasonable approach because the model, which is trained on the training set, would not know what to do with the new word anyway. A slightly less hacky way would be to explicitly learn a "garbage" word and map all rare frequency words to it, even within the training set, as discussed in "Rare words" on page 28.

GraphLab Create can perform automatic L^2 norm feature scaling as part of its logistic regression classifier training process. So we do not need to explicitly generate an L^2-scaled feature column.

Even though we do not experiment with min-max scaling and standardization in this chapter, it is appropriate to also discuss their implementation details here. They both subtract a quantity from the original feature value. For min-max scaling, the shift is the minimum over all values of the current feature; for standardization, it is the mean. If the shift is not zero, then these two transforms can turn a sparse feature vector where most values are zero into a dense one. This in turn could create a huge computational burden for the classifier, depending on how it is implemented. Bag-of-words is a sparse representation, and the classifiers in both GraphLab Create and Vowpal Wabbit (a popular open-source large scale linear classifier) optimize for sparse inputs. It would be horrendous if the representation now includes every word that doesn't appear in a document. One should use extreme caution when performing min-max scaling and standardization on sparse features.

First try: plain logistic regression

We choose logistic regression as the classifier. It's simple, easy to explain, and performs well when given good features. Plain logistic regression, without any bells and whistles, has only one notable parameter—the number of iterations to run the solver. We choose to run 40 iterations when using bag-of-words features, 5 iterations for L^2-normalized BOW, and 10 iterations for tf-idf. The reason for these choices will be made clear in the next section.

Example 3-3. Training simple logistic classifiers with no regularization

```
# On simple bag-of-words
model1 = gl.logistic_classifier.create(training_data,
                                       features=['bow'], target='category',
                                       feature_rescaling=False, max_iterations=40,
                                       l2_penalty=0)
# On L2-normalized bag-of-words
model2 = gl.logistic_classifier.create(training_data,
                                       features=['bow'], target='category',
                                       feature_rescaling=True, max_iterations=40,
                                       l2_penalty=0)
# On tf-idf features
model3 = gl.logistic_classifier.create(training_data,
                                       features=['tfidf.bow'], target='category',
                                       feature_rescaling=False, max_iterations=40,
                                       l2_penalty=0)
# Evaluate on test set
result1 = model1.evaluate(test_data)
result2 = model2.evaluate(test_data)
result3 = model3.evaluate(test_data)

# Plot accuracy and compare
import pandas as pd
import seaborn as sns

result_df = pd.DataFrame({'BOW':   [result1['accuracy']],
                          'L2':    [result2['accuracy']],
                               'TFIDF': [result3['accuracy']]})
sns.pointplot(data=result_df)
```

The model training process automatically splits off 5% of the training data for use as a hold-out validation set. It's optional. If we have very little data, we might not be able to afford it. But since we have plenty of data, this allows us to track the training performance and observe that the validation performance is not dropping precipitously. (Decreasing validation performance is a good sign that the training process is overfitting to the training set; this marks a good stopping point for the solver.)

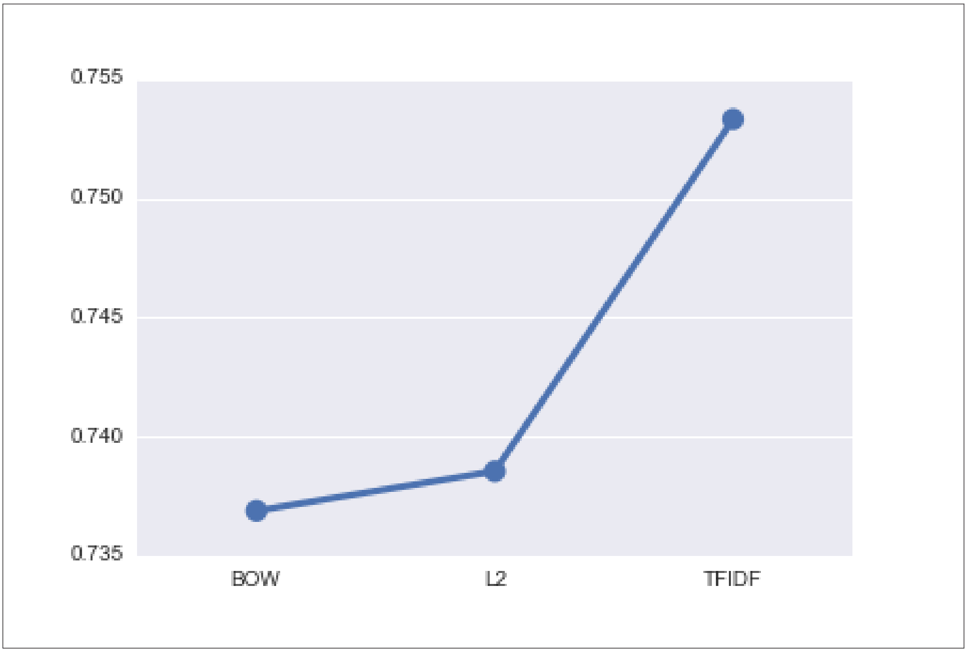


Figure 3-6. Test set accuracy for plain logistic regression classifiers

The results are fascinating. Bag-of-words and L^2 normalization produced models with comparable accuracy, while tf-idf is a tad more accurate than both. But since the differences in accuracy are so small, it's possible that it can be due to statistical randomness in the dataset. In order to find out, we need to train and test on multiple datasets that are ideally independent but all come from the same distribution. Let's do this and expand our experiment.

Second try: logistic regression with regularization

Logistic regression has a few bells and whistles. When the number of features is greater than the number of data points, the problem of finding the best model is said to be *underdetermined*. One way to fix this problem is by placing additional constraints on the training process. This is known as *regularization*, and its technical details are discussed in the next section.

Most implementations of logistic regression allow for regularization. In order to use this functionality, one must specify a regularization parameter. Regularization parameters are *hyperparameters* that are not learned automatically in the model training process. Rather they must be tuned on the problem at hand and given to the training algorithm. This process is known as hyperparameter tuning. (For details on how to evaluate machine learning models, see, e.g., *Evaluating Machine Learning Models*.) One basic method for tuning hyperparameters is called grid search: we specify a grid of hyperparameter values and the tuner programmatically searches for the best hyperparameter setting in the grid. Once the best hyperparameter setting is found, we train a model on the training set using that setting and compare the performance of these best-of-breed models on the test set.

Important: Tune Hyperparameters When Comparing Models

It's essential to tune hyperparameters when comparing models or features. The default settings of a software package will always return a model. But unless the software performs automatic tuning under the hood, it is likely to return a suboptimal model based on suboptimal hyperparameter settings. Some models are more sensitive to hyperparameter settings than others. Logistic regression is relatively robust (or insensitive) to hyperparameter settings. Even so, it is necessary to find and use the right *range* of hyperparameters. Otherwise, the advantages of one model versus another may be solely due to tuning parameters, and do not reflect the actual behavior of the model or features.

Even the best autotuning packages still require specifying the upper and lower limits of search, and finding those limits can take a few manual tries, as it did here.

The optimal hyperparameter setting depends on the scale of the input features. Since tf-idf, bow, and L^2 normalization result in features of different scales, they require separate search grids. We search over the number of solver iterations and ℓ_2 regularization parameter (not to be confused with L^2 normalization of the features). Here, we define the grid manually, after a few tries to narrow down the lower bound and upper bounds for each case. The optimal hyperparameter settings for each feature set is given in Table 3-1. The values for max_iterations are also used in the experiment discussed in the previous section.

Table 3-1. Hyperparameter settings for logistic regression on Yelp reviews

	max_iterations	l2_regularization
BOW_NoReg	40	0
L2_NoReg	5	0
TF-IDF	10	0
BOW	40	0.01
L2	5	50
TF-IDF	8	0.1

We also want to test whether the difference in accuracy between tf-idf and BOW is due to noise. To this end, we extract several resamples of the data. Once we find the optimal hyperparameter settings on each of the feature sets, we train a model on each of the resampled datasets. Figure 3-7 shows a box-and-whiskers plot of the distribution of accuracy measurements for models trained on each of the feature sets, with or without regularization. The middle line in the box marks the median accuracy, the box itself marks the region between the first and third quartiles, and the whiskers extend to the rest of the distribution.

Estimating Variance via Resampling

Modern statistical methods assume that the underlying data comes from a random distribution. The performance measurements of models derived from data is also subject to random noise. In this situation, it is always a good idea to take the measurement not just once, but multiple times, based on datasets of comparable statistics. This gives us a confidence interval for the measurement.

Resampling is a useful technique for generating multiple small samples from the same underlying dataset. Alternatively, we could have divided the data into separate chunks and tested on each. (See *Evaluating Machine Learning Models* for more details on resampling.)

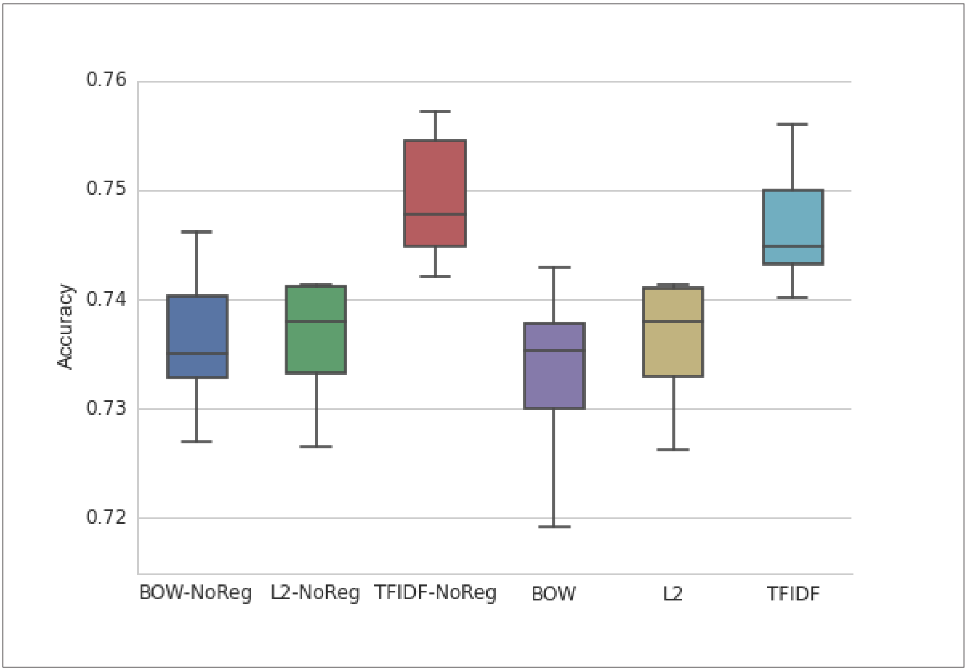


Figure 3-7. Distribution of classifier accuracy under each feature set and regularization setting, measured over resampled datasets

The accuracy results are very similar to what we've already seen in the first experiment. ℓ_2 regularization does not seem to change the accuracy of the learned models on this dataset. The optimal regularization parameters are on different scales for each of the feature sets. But the end result obtained using the best tuned model is the same as before: bag-of-words and L^2 normalized features hover between 0.73 and 0.74, while tf-idf consistently beats them by a small bit. Since the empirical confidence intervals for tf-idf have little overlap with the other two methods, the differences are deemed significant.

The models themselves also look a lot like the ones without regularization, in that they pick out the same set of distinguishing words. Bag-of-words and tf-idf continue to yield useful top words, while L^2 normalization picks out the same gibberish words.

Discussion of results

Here is a summary of our findings:

1. ℓ_2 regularization makes no detectable difference in terms of accuracy or training convergence speed.

2. L^2 feature normalization results in models that are quicker to train (converging in 5 iterations compared to 10 for tf-idf and 40 for BOW) but are no more accurate than BOW.
3. Tf-idf is the overall winner. With tf-idf features, the model is faster to train, has slightly better accuracy, and retains the interpretable results from BOW.

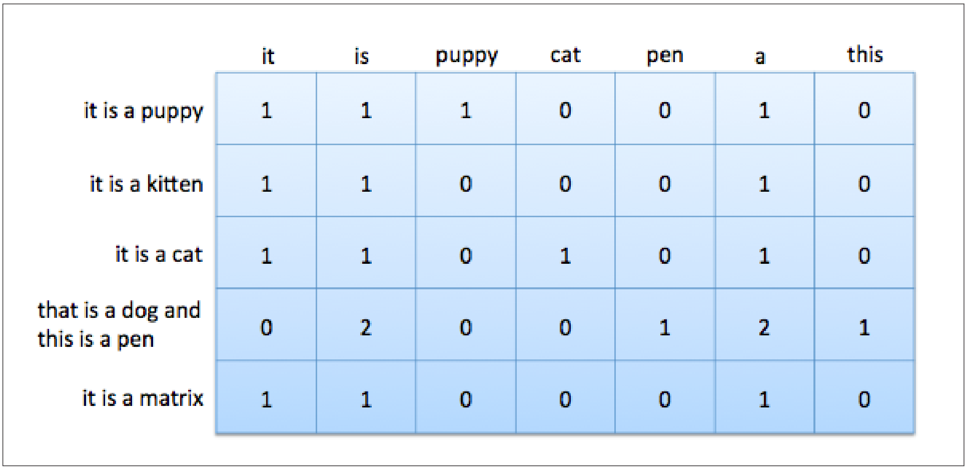
Keep in mind that these findings are limited to this dataset and this task. We certainly do not claim that ℓ_2 regularization and L^2 feature normalization are useless in general.

These results completely mystified me when I first saw them. L^2 normalization and tf-idf are both feature scaling methods. Why does one work better than the other? What is the secret of tf-idf? We will spend the rest of the chapter exploring the answers.

Deep Dive: What is Happening?

In order to understand the "why" behind the results, we have to look at how the features are being used by the model. For linear models like logistic regression, this happens through an intermediary object called the data matrix.

The data matrix contains data points represented as fixed-length flat vector. With bag-of-words vectors, the data matrix is also known as the document-term matrix. Figure 2-1 shows a bag-of-words vector in vector form, and Figure 3-1 illustrates four bag-of-words vectors in feature space. To form a document-term matrix, simply take the document vectors, lay them out flat, and stack them on top of one another. The columns represent all possible words in the vocabulary. Since most documents contain only a small subset of all possible words, most of the entries in this matrix are zero; it is a **sparse** matrix.



	it	is	puppy	cat	pen	a	this
it is a puppy	1	1	1	0	0	1	0
it is a kitten	1	1	0	0	0	1	0
it is a cat	1	1	0	1	0	1	0
that is a dog and this is a pen	0	2	0	0	1	2	1
it is a matrix	1	1	0	0	0	1	0

Figure 3-8. An example document-term matrix of 5 documents and 7 words

Feature scaling methods are essentially column operations on the data matrix. In particular, tf-idf and L^2 normalization both multiply the entire column (an n-gram feature, for example) by a constant.

Tf-idf = Column Scaling

Tf-idf and L^2 normalization are both column operations on the data matrix.

As discussed in Appendix A, training a linear classifier boils down to finding the best linear combination of features, which are column vectors of the data matrix. The solution space is characterized by the column space and the null space of the data matrix. The quality of the trained linear classifier directly depends upon the null space and the column space of the data matrix. A large column space means that there is little linear dependency between the features, which is generally good. The null space contains “novel” data points that cannot be formulated as linear combinations of existing data; a large null space could be problematic. (A perusal of Appendix A is highly recommended for readers who are a little shaky on concepts such as linear decision surface, eigen decomposition, and the fundamental subspaces of a matrix.)

How do column scaling operations affect the column space and null space of the data matrix? The answer is, not very much. But there is a small but crucial difference between tf-idf and L^2 normalization that explains the difference in results.

The null space of the data matrix can be large for a couple of reasons. First, many datasets contain data points that are very similar to one another. Therefore the effective row space is small compared to the number of data points in the dataset.

Second, the number of features can be much larger than the number of data points. Bag-of-words is particularly good at creating giant feature spaces. In our Yelp example, there are 76K features in 33K reviews. Moreover, the number of distinct words usually grows with the number of documents in the dataset. So adding more documents would not necessarily decrease the feature-to-data ratio or reduce the null space.

With bag-of-words, the column space is relatively small compared to the number of features. There could be words that appear roughly the same number of times in the same documents. This would lead to the corresponding column vectors being nearly linearly dependent, which leads to the column space being not as full rank as it could be. This is called a rank deficiency. (Much like how animals can be deficient in vitamins and minerals, matrices can be rank deficient.)

Rank deficient row space and column space lead to the model being overly provisioned for the problem. The linear model outfits a weight parameter for each feature in the dataset. If the row and column spaces were full rank[2], then the model would allow us to generate any target vector in the output space. When they are rank deficient, the model has more degrees of freedom than it needs. This makes it more tricky to pin down a solution.

Can feature scaling solve the rank deficiency problem of the data matrix? Let's take a look.

The column space is defined as the linear combination of all column vectors: $a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \ldots + a_n\mathbf{v}_n$. Feature scaling replaces a column vector with a constant multiple, say $\tilde{\mathbf{v}}_1 = c\mathbf{v}_1$. But we can still generate the original linear combination by just replacing a_1 with $\tilde{a}_1 = a_1/c$. It appears that feature scaling does not change the rank of the column space. Similarly, feature scaling does not affect the rank of the null space, because one can counteract the scaled feature column by reverse scaling the corresponding entry in the weight vector.

However, as usual, there is one catch. If the scalar is 0, then there is no way to recover the original linear combination; $\mathbf{v}_1$ is gone. If that vector is linearly independent to all other columns, then we've effectively shrunk the column space and enlarged the null space.

If that vector is not correlated with the target output, then this is effectively pruning away noisy signals, which is a good thing. This turns out to be the key difference between tf-idf and L^2 normalization. L^2 normalization would never compute a norm of zero, unless the vector contains all zeros. If the vector is close to zero, then its norm is also close to zero. Dividing by the small norm would accentuate the vector and make it longer.

Tf-idf, on the other hand, could generate scaling factors that are close to zero, as shown in Figure 3-2. This happens when the word is present in a large number of documents in the training set. Such a word is likely not strongly correlated with the target vector. Pruning it away allows the solver to focus on the other directions in the column space and find better solutions, as we see in the experiments. The improvement in accuracy is not huge, presumably because there are few noisy direction that are prunable in this way.

Where feature scaling—both L^2 and tf-idf—does have a telling effect is on the convergence speed of the solver. This is a sign that the data matrix now has a much smaller

[2] Strictly speaking, the row space and column space for a rectangular matrix cannot both be full rank. The maximum rank for both subspaces is the smaller of m (the number of rows) and n (the number of columns). This is what we mean by full rank.

condition number. In fact, L^2 normalization makes the condition number nearly uniform. But it's not the case that the better the condition number, the better the solution. As we can see based on the experimental results, L^2 normalization converges much faster than either BOW or tf-idf. But it is also more sensitive to overfitting: it requires much more regularization, and is more sensitive to the number of iterations during optimization. Running the solver past 5 iterations could sometimes decrease the accuracy of the learned model.

Summary

In this chapter, we used tf-idf as an entry point into a detailed analysis of how feature transformations can effect the model (or not). Tf-idf is an example of feature scaling, so we contrasted its performance with another feature scaling method—L^2 normalization.

The results are far from expected. Tf-idf and L^2 normalization are structurally identical, since they both scale the columns of the data matrix. Yet they result in models with different accuracies. The difference is small but persistent. After acquiring some statistical modeling and linear algebra chops, we arrive at an even more mystifying observation conclusion: theoretically, *neither* of these methods should have an effect on the accuracy, since neither of them should change the column space.

After scratching our heads for a moment, we finally realize that one important difference between the two is that tf-idf can "stretch" the word count as well as "compress" it. In other words, it makes some counts bigger, and others close to zero. This latter fact explained the difference in accuracy: tf-idf is eliminating uninformative words, while L^2 normalization makes everything even.

Along the way, we also discovered another effect of feature scaling: it improves the condition number of the data matrix, making linear models much faster to train. Both L^2 normalization and tf-idf had this effect.

To summarize, the lesson is: the *right* feature scaling can be helpful for classification. The right scaling accentuates the informative words and down-weighs the common words. It can also improve the condition number of the data matrix. The right scaling is not necessarily uniform column scaling.

This story is a wonderful illustration of the difficulty of analyzing the effects of feature engineering in the general case. Changing the features affects the training process and the models that ensue. Linear models are the simplest models to understand. Yet it still takes very careful experimentation methodology and a lot of deep mathematical knowledge to tease apart the theoretical and practical impacts. This would be mostly impossible on more complicated models or feature transformations.

Bibliography

Strang, Gilbert. 2006. *Linear Algebra and Its Applications*. Brooks Cole Cengage, fourth edition.

APPENDIX A

Linear Modeling and Linear Algebra Basics

Overview of Linear Classification

When we have a labeled dataset, the feature space is strewn with data points from different classes. It is the job of the classifier to separate the data points from different classes. It can do so by producing an output that is very different for data points from one class versus another. For instance, when there are only two classes, then a good classifier should produce large outputs for one class, and small ones for another. The points right on the cusp of being one class versus another form a *decision surface*.

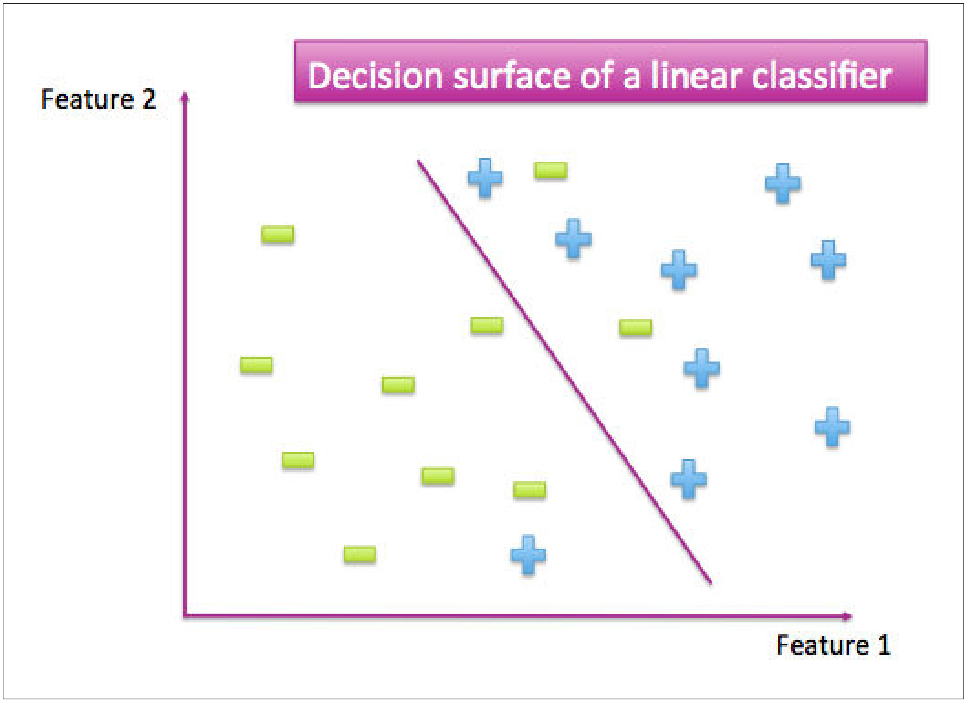


Figure A-1. Simple binary classification finds a surface that separates two classes of data points

Many functions can be made into classifiers. It’s a good idea to look for the *simplest* function that cleanly separates between the classes. First of all, it’s easier to find the best simple separator rather than the best complex separator. Also, simple functions often generalize better to new data, because it’s harder to tailor them too specifically to the training data (a concept known as *overfitting*). A simple model might make mistakes, like in the diagram above where some points are on the wrong side of the divide. But we sacrifice some training accuracy in order to have a simpler decision surface that can achieve better test accuracy. The principle of minimizing complexity and maximizing usefulness is called “Occam’s Razor,” and is widely applicable in science and engineering.

The simplest function is a line. A linear function of one input variable is a familiar sight.

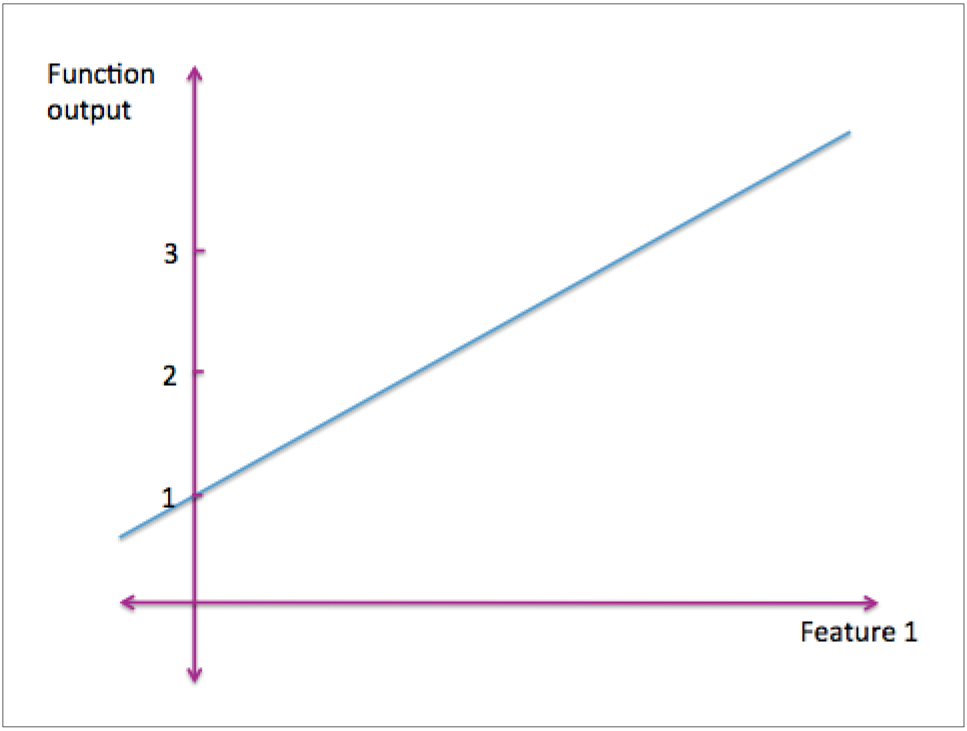


Figure A-2. A linear function of one input variable

A linear function with two input variables can be visualized as either a flat plane in 3D or a contour plot in 2D (shown in Figure A-3). Like a topological geographic map, each line of the contour plot represents points in the input space that have the same output.

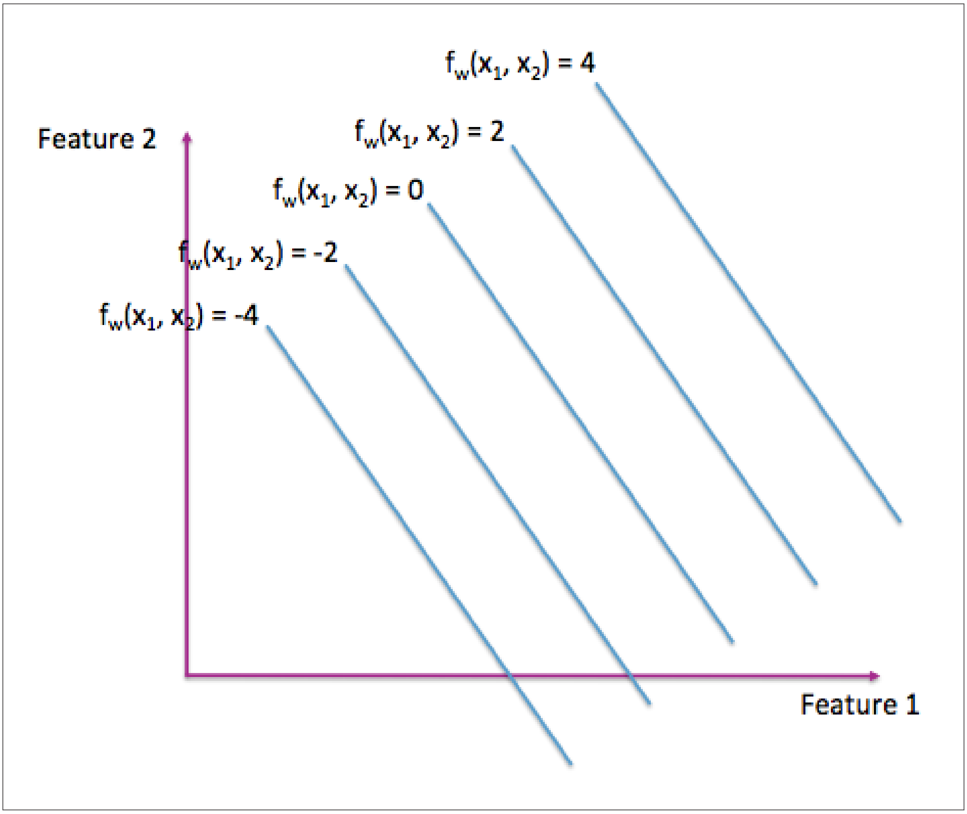


Figure A-3. Contour plot of a linear function in 2D

It's harder to visualize higher dimensional linear functions, which are called hyperplanes. But it's easy enough to write down the algebraic formula. A multi-dimensional linear function has a set of inputs $x_1, x_2, ..., x_\mathrm{n}$ and a set of weight parameters $w_0, w_1, ..., w_\mathrm{n}$:

$f_\mathrm{w}(x_1, x_2, ..., x_n) = w_0 + w_1 * x_1 + w_2 * x_2 + ... + w_n * x_n$.

It can be written more succinctly using vector notation $f_\mathrm{w}(\mathbf{x}) = \mathbf{x}^T\mathbf{w}$. We follow the usual convention for mathematical notations, which uses boldface to indicate a vector and non-boldface to indicate a scalar. The vector **x** is padded with an extra 1 at the beginning, as a placeholder for the intercept term w_0. If all input features are zero, then the output of the function is w_0. So w_0 is also known as the *bias* or *intercept term*.

Training a linear classifier is equivalent to picking out the best separating hyperplane between the classes. This translates into finding the best vector **w** that is oriented exactly right in space. Since each data point has a target label y, we could find a $\mathbf{w}$ that tries to directly emulate the target label[2]

$\mathbf{x}^T\mathbf{w} = \mathbf{y}$.

Since there is usually more than one data point, we want a $\mathbf{w}$ that simultaneously makes all of the predictions close to the target labels:

Linear model equation

$$A\mathbf{w} = \mathbf{y}$$

Here, A is known as the *data matrix* (also known as the design matrix in statistics). It contains the data in a particular form: each row is a data point and each column a feature. (Sometimes people also look at its transpose, where features are on the rows and data points the columns.)

The Anatomy of a Matrix

In order to solve Equation A-1, we need some basic knowledge of linear algebra. For a systematic introduction to the subject, we highly recommend Gilbert Strang's book "Linear Algebra and Its Applications."

Equation A-1 states that when a certain matrix multiplies a certain vector, there is a certain outcome. A matrix is also called a linear operator, a name that makes it more apparent that a matrix is a little machine. This machine takes a vector as input and spits out another vector using a combination of several key operations: rotating a vector's direction, adding or subtracting dimensions, and stretching or compressing its length.

[Illustration of a matrix mapping the 2D plane into a tilted plane in 3D.]

From vectors to subspaces

In order to understand a linear operator, we have to look at how it morphs the input into output. Luckily, we don't have to analyze one input vector at a time. Vectors can be organized into *subspaces*, and **linear operators manipulate vector subspaces.**

[2] Strictly speaking, the formula given here is for linear regression, not linear classification. The difference is that regression allows for real-valued target variables, whereas classification targets are usually integers that represent different classes. A regressor can be turned into a classifier via a non-linear transform. For instance, the logistic regression classifier passes the linear transform of the input through a logistic function. Such models are called generalized linear models and have linear functions at their core. Even though this example is about classification, we use the formula for linear regression as a teaching tool, because it is much easier to analyze. The intuitions readily map to generalized linear classifiers.

A subspace is a set of vectors that satisfies two criteria: 1. if it contains a vector, then it contains the line that passes through the origin and that point, and 2. if it contains two points, then it contains all the linear combinations of those two vectors. Linear combination is a combination of two types of operations: multiplying a vector with a scalar, and adding two vectors together.

One important property of a subspace is its *rank* or dimensionality, which is a measure of the degrees of freedom in this space. A line has rank 1, a 2D plane has rank 2, and so on. If you can imagine a multi-dimensional bird in our multi-dimensional space, then the rank of the subspace tells us in how many "independent" directions the bird could fly. "Independence" here means "linear independence": two vectors are linearly independent if one isn't a constant multiple of another, i.e., they are not pointing in exactly the same or opposite directions.

A subspace can be defined as the span of a set of *basis vectors*. (Span is a technical term that describes the set of all linear combinations of a set of vectors.) The span of a set of vectors is invariant under linear combinations (because it's defined that way). So if we have one set of basis vectors, then we can multiply the vectors by any nonzero constants or add the vectors to get another basis.

It would be nice to have a more unique and identifiable basis to describe a subspace. An *orthonormal basis* contains vectors that have unit length and are orthogonal to each other. Orthogonality is another technical term. (At least 50% of all math and science is made up of technical terms. If you don't believe me, do a bag-of-words count on this book.) Two vectors are *orthogonal* to each other if their inner product is zero. For all intensive purposes, we can think of orthogonal vectors as being at 90 degrees to each other. (This is true in Euclidean space, which closely resembles our physical 3D reality.) Normalizing these vectors to have unit length turns them into a uniform set of measuring sticks.

All in all, a subspace is like a tent, and the orthogonal basis vectors are the number of poles at right angles that are required to prop up the tent. The rank is equal to the total number of orthogonal basis vectors.

In pictures:

[illustrations of inner product, linear combinations, the subspace tent and orthogonal basis vectors.]

For those who think in math, here is some math to make our descriptions precise.

Useful Linear Algebra Definitions

Scalar:

A number c, in contrast to vectors.

Vector:

$\mathbf{x} = (x_1, x_2, \ldots, x_n)$

Linear combination:

$a\mathbf{x} + b\mathbf{y} = (ax_1 + by_1, ax_2 + by_2, \ldots, ax_n + by_n)$

Span of a set of vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$:

The set of vectors $\mathbf{u} = a_1\mathbf{v}_1 + \ldots + a_k\mathbf{v}_k$ for any $a_1, \ldots, a_k$

Linear independence:

x and **y** are independent if $\mathbf{x} \neq c\mathbf{y}$ for any scalar constant c.

Inner product:

$\langle \mathbf{x}, \mathbf{y} \rangle = x_1y_1 + x_2y_2 + \ldots + x_{\mathrm{n}}y_{\mathrm{n}}$

Orthogonal vectors:

Two vectors **x** and **y** are orthogonal if $\langle \mathbf{x}, \mathbf{y} \rangle = 0$

Subspace:

A subset of vectors within a larger containing vector space, satisfying these three criteria:

1. It contains the zero vector.
2. If it contains a vector **v**, then it contains all vectors $c\mathbf{v}$, where c is a scalar.
3. If it contains two vectors **u** and **v**, then it contains the vector $\mathbf{u} + \mathbf{v}$.

Basis:

A set of vectors that span a subspace.

Orthogonal basis:

A basis $\{ \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_d \}$ where $\langle \mathbf{v}_i, \mathbf{v}_j \rangle = 0$ for all i, j.

Rank of subspace:

Minimum number of linearly independent basis vectors that span the subspace.

Singular value decomposition (SVD)

A matrix performs a linear transformation on the input vector. Linear transformations are very simple and constrained. It follows that a matrix can't manipulate a subspace willy-nilly. One of the most fascinating theorems of linear algebra proves that every square matrix, no matter what numbers it contains, must map a certain set of vectors back to themselves with some scaling. In the general case of a rectangular matrix, it maps a set of input vectors into a corresponding set of output vectors, and its *transpose* maps those outputs back to the original inputs. The technical terminology is that square matrices have eigenvectors with eigenvalues, and rectangular matrices have left and right singular vectors with singular values.

Eigenvector and Singular Vector

Let A be an nxn matrix. If there is a vector $\mathbf{v}$ and a scalar λ such that $A\mathbf{v} = \lambda\mathbf{v}$, then $\mathbf{v}$ is an *eigenvector* and λ an *eigenvalue* of A.

Let A be a rectangular matrix. If there are vectors $\mathbf{u}$ and $\mathbf{v}$ and a scalar σ such that $A\mathbf{v} = \sigma\mathbf{u}$ and $A^{\mathrm{T}}\mathbf{u} = \sigma\mathbf{v}$, then $\mathbf{u}$ and $\mathbf{v}$ are called left and right singular vectors and σ is a singular value of A.

Algebraically, the SVD of a matrix looks like this:

$A = U\Sigma V^T$,

where the columns of the matrices U and V form orthonormal bases of the input and output space, respectively. Σ is a diagonal matrix containing the singular values.

Geometrically, a matrix performs the following sequence of transformations:

1. Map the input vector onto the right singular vector basis V;
2. Scale each coordinate by the corresponding singular values;
3. Multiply this score with each of the left singular vectors;
4. Sum up the results.

When A is a real matrix (i.e., all of the elements are real valued), all of the singular values and singular vectors are real-valued. A singular value can be positive, negative, or zero. The ordered set of singular values of a matrix is called its *spectrum*, and it reveals a lot about the matrix. The gap between the singular values effects how stable the solutions are, and the ratio between the maximum and minimum absolute singular values (the condition number) effects how quickly an iterative solver can find the solution. Both of these properties have notable impacts on the quality of the solution one can find.

[Illustration of a matrix as three little machines: rotate right, scale, rotate left.]

The four fundamental subspaces of the data matrix

Another useful way to dissect a matrix is via the four fundamental subspaces: column space, row space, null space, and left null space. These four subspaces completely characterize the solutions to linear systems involving A or A^{T}. Thus they are called the four fundamental subspaces.

For the data matrix, the four fundamental subspaces can be understood in relation to the data and features. Let's look at them in more detail.

Data matrix
A: rows are data points, columns are features

Column space

Mathematical definition:
The set of output vectors $\mathbf{s}$ where $\mathbf{s} = A\mathbf{w}$ as we vary the weight vector $\mathbf{w}$.

Mathematical interpretation:
All possible linear combinations of columns.

Data interpretation:
All outcomes that are linearly predictable based on observed features. The vector $\mathbf{w}$ contains the weight of each feature.

Basis:
The left singular vectors corresponding to non-zero singular values (a subset of the columns of U).

Row space

Mathematical definition:
The set of output vectors $\mathbf{r}$ where $\mathbf{r} = \mathbf{u}^T A$ as we vary the weight vector $\mathbf{u}$.

Mathematical interpretation:
All possible linear combinations of rows.

Data interpretation:
A vector in the row space is something that can be represented as a linear combination of existing data points. Hence this can be interpreted as the space of "non-novel" data. The vector $\mathbf{u}$ contains the weight of each data point in the linear combination.

Basis:
The right singular vectors corresponding to non-zero singular values (a subset of the columns of V).

Null space

Mathematical definition:
The set of input vectors $\mathbf{w}$ where $A\mathbf{w} = 0$.

Mathematical interpretation:
Vectors that are orthogonal to all rows of A. The null space gets squashed to 0 by the matrix. This is the "fluff" that adds volume to the solution space of $A\mathbf{w} = \mathbf{y}$.

Data interpretation:
"Novel" data points that cannot be represented as any linear combination of existing data points.

Basis:

The right singular vectors corresponding to the zero singular values (the rest of the columns of V).

Left null space

Mathematical definition:

The set of input vectors $\mathbf{u}$ where $\mathbf{u}^T A = 0$.

Mathematical interpretation:

Vectors that are orthogonal to all columns of A. The left null space is orthogonal to the column space.

Data interpretation:

"Novel feature vectors" that are not representable by linear combinations of existing features.

Basis:

The left singular vectors corresponding to the zero singular values (the rest of the columns of U).

Column space and row space contain what is already representable based on observed data and features. Those vectors that lie in the column space are non-novel features. Those vectors that lie in the row space are non-novel data points.

For the purposes of modeling and prediction, non-novelty is good. A full column space means that the feature set contains enough information to model any target vector we wish. A full row space means that the different data points contain enough variation to cover all possible corners of the feature space. It's the novel data points and features—respectively contained in the null space and the left null space—that we have to worry about.

In the application of building linear models of data, the null space can also be viewed as the subspace of "novel" data points. Novelty is not a good thing in this context. Novel data points are phantom data that is not linearly representable by the training set. Similarly, the left null space contains novel features that are not representable as linear combinations of existing features.

The null space is orthogonal to the row space. It's easy to see why. The definition of null space states that $\mathbf{w}$ has an inner product of 0 with every row vector in A. Therefore, $\mathbf{w}$ is orthogonal to the space spanned by these row vectors, i.e., the row space. Similarly, the left null space is orthogonal to the column space.

Solving a Linear System

Let's tie all this math back to the problem at hand: training a linear classifier, which is intimately connected to the task of solving a linear system. We look closely at how a matrix operates because we have to reverse engineer it. In order to train a linear model, we have to find the input weight vector $\mathbf{w}$ that maps to the observed output targets $\mathbf{y}$ in the system $A\mathbf{w} = \mathbf{y}$, where A is the data matrix.[2]

Let us try to crank the machine of the linear operator in reverse. If we had the SVD decomposition of A, then we could map $\mathbf{y}$ onto the left singular vectors (columns of U), reverse the scaling factors (multiply by the inverse of the non-zero singular values), and finally map them back to the right singular vectors (columns of V). Ta-da! Simple, right?

This is in fact the process of computing the *pseudo-inverse* of A. It makes use of a key property of an orthonormal basis: the transpose is the inverse. This is why SVD is so powerful. (In practice, real linear system solvers do not use the SVD, because they are rather expensive to compute. There are other, much cheaper ways to decompose a matrix, such as QR or LU or Cholesky decompositions.)

However, we skipped one tiny little detail in our haste. What happens if the singular value is zero? We can't take the inverse of zero because $1/0 = \infty$. This is why it's called the pseudo-inverse. (The real inverse isn't even defined for rectangular matrices. Only square matrices have them (as long as all of the eigenvalues are non-zero).) A singular value of zero squashes whatever input was given; there's no way to retrace its steps and come up with the original input.

Okay, going backwards is stuck on this one little detail. Let's take what we've got and go forward again to see if we can unjam the machine. Suppose we came up with an answer to $A\mathbf{w} = \mathbf{y}$. Let's call it $\mathbf{w}_{\text{particular}}$, because it's particularly suited for $\mathbf{y}$. Let's say that there are also a bunch of input vectors that A squashes to zero. Let's take one of them and call it $\mathbf{w}_{\text{sad-trumpet}}$, because wah wah. Then, what do you think happens when we add $\mathbf{w}_{\text{particular}}$ to $\mathrm{w}_{\text{sad-trumpet}}$?

$A(\mathbf{w}_{\text{particular}} + \mathbf{w}_{\text{sad-trumpet}}) = \mathbf{y}$.

Amazing! So this is a solution too. In fact, any input that gets squashed to zero could be added to a particular solution and give us another solution. The general solution looks like this:

$\mathbf{w}_{\text{general}} = \mathbf{w}_{\text{particular}} + \mathbf{w}_{\text{homogeneous}}$.

[2] Actually, it's a little more complicated than that. y may not be in the column space of A, so there may not be a solution to this equation. Instead of giving up, statistical machine learning looks for an approximate solution. It defines a loss function that quantifies the quality of a solution. If the solution is exact, then the loss is 0. Small errors, small loss; big errors, big loss, and so on. The training process then looks for the best parameters that minimize this loss function. In ordinary linear regression, the loss function is called the squared residual loss, which essentially maps y to the closest point in the column space of A. Logistic regression minimizes the log-loss. In both cases, and linear models in general, the linear system $Aw = y$ often lies at the core. Hence our analysis here is very much relevant.

$\mathbf{w}_{\text{particular}}$ is an exact solution to the equation $A\mathbf{w} = \mathbf{y}$. There may or may not be such a solution. If there isn't, then the system can only be approximately solved. If there is, then $\mathbf{y}$ belongs to what's known as the column space of A. The column space is the set of vectors that A can map *to*, by taking linear combinations of its columns.

$\mathbf{w}_{\text{homogeneous}}$ is a solution to the equation $A\mathbf{w} = 0$. (The grown-up name for $\mathbf{w}_{\text{sad-trumpet}}$ is $\mathbf{w}_{\text{homogeneous}}$.) This should now look familiar. The set of all $\mathbf{w}_{\text{homogeneous}}$ vectors forms the null space of A. This is the span of the right singular vectors with singular value 0.

[Illustration of w_general and null space?]

The name "null space" sounds like the destination of woe for an existential crisis. If the null space contains any vectors other than the all-zero vector, then there are infinitely many solutions to the equation $A\mathbf{w} = \mathbf{y}$. Having too many solutions to choose from is not in itself a bad thing. Sometimes any solution will do. But if there are many possible answers, then there are many sets of features that are useful for the classification task. It becomes difficult to understand which ones are truly important.

One way to fix the problem of a large null space is to *regulate* the model by adding additional constraints:

$A\mathbf{w} = \mathbf{y}$, where $\mathbf{w}$ is such that $\mathbf{w}^T\mathbf{w} = c$

This form of regularization constrains the weight vector to have a certain norm c. The strength of this regularization is controlled by a regularization parameter, which must be tuned, as is done in our experiments.

In general, *feature selection* methods deal with selecting the most useful features to reduce computation burden, decrease the amount of confusion for the model, and make the learned model more unique. This is the focus of [chapter nnn].

Another problem is the "unevenness" of the spectrum of the data matrix. When we train a linear classifier, we care not only that there is a general solution to the linear system, but also that we can find it easily. Typically, the training process employs a solver that works by calculating a gradient of the loss function and walking downhill in small steps. When some singular values are very large and other very close to zero, the solver needs to carefully step around the longer singular vectors (those that correspond to large singular values) and spend a lot of time to dig around the shorter singular vectors to find the true answer. This "unevenness" in the spectrum is measured by the *condition number* of the matrix, which is basically the ratio between the largest and the smallest absolute value of the singular values.

To summarize, in order for there to be a good linear model that is relatively unique, and in order for it to be easy to find, we wish for the following:

1. The label vector can be well approximated by a linear combination of a subset of features (column vectors). Better yet, the set of features should be linearly independent.
2. In order for the null space to be small, the row space must be large. (This is due to the fact that the two subspaces are orthogonal.) The more linearly independent is the set of data points (row vectors), the smaller the null space.
3. In order for the solution to be easy to find, the condition number of the data matrix—the ratio between the maximum and minimum singular values—should be small.

Index