O'REILLY®

Early Release
RAW & UNEDITED

Mastering Feature Engineering

PRINCIPLES AND TECHNIQUES FOR DATA SCIENTISTS

Alice Zheng

Mastering Feature Engineering

by Alice Zheng
Publisher: O’Reilly Media, Inc.
Release Date: March 2017
ISBN: 9781491953242
Topic: Engineering

Book Description

Feature engineering is essential to applied machine learning, but using domain knowledge to strengthen your predictive models can be difficult and expensive. To help fill the information gap on feature engineering, this complete hands-on guide teaches beginning-to-intermediate data scientists how to work with this widely practiced but little discussed topic.

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.

Tip

This element signifies a tip or suggestion.

Note

This element signifies a general note.

Warning

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.

Safari® Books 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. Practitioners agree that the vast majority of time in building 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 Chapter 5) 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.

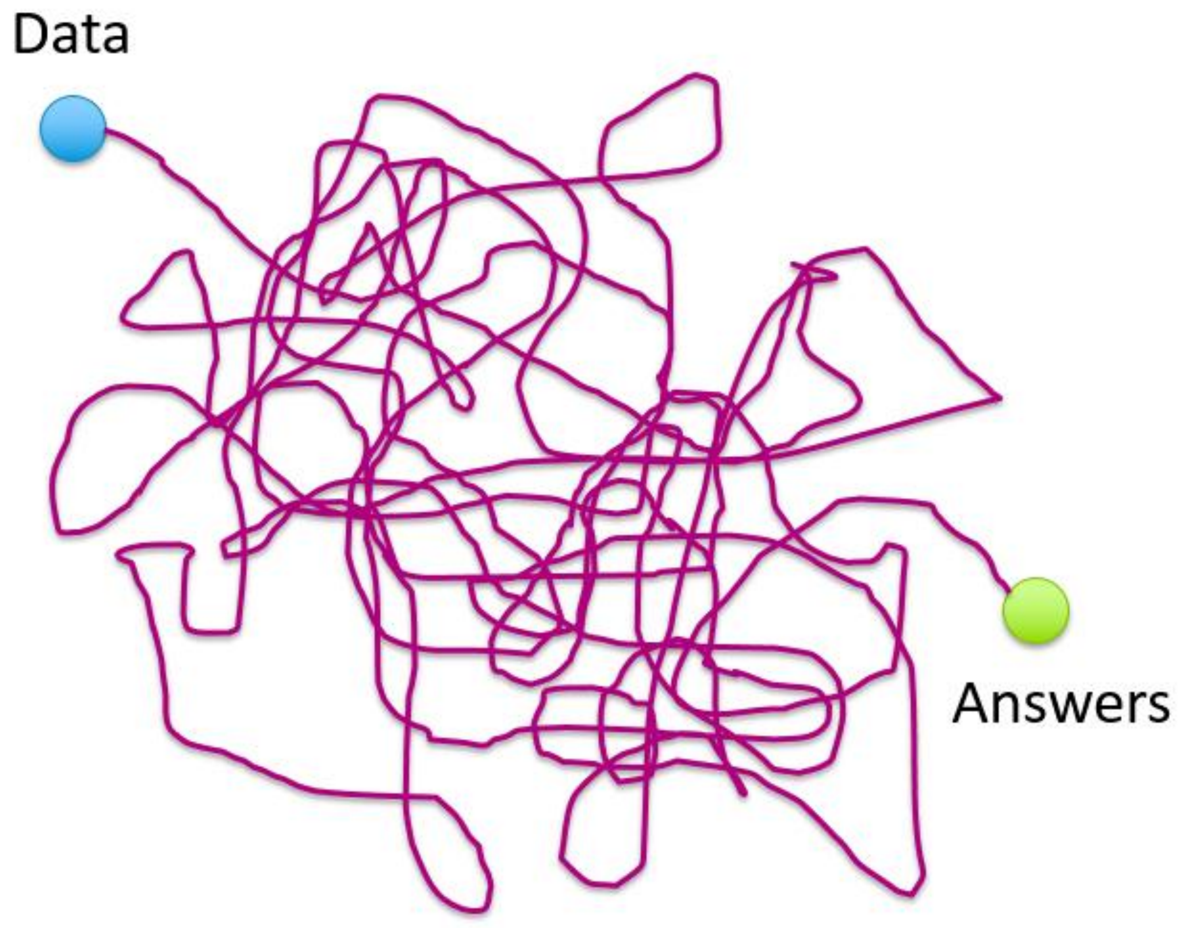


Figure 1-1. The messy path from data to answers.

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. 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. The right features are relevant to the task at hand and should be easy for the model to ingest. *Feature engineering* is the process of formulating the most appropriate features given the data, the model, and the task.

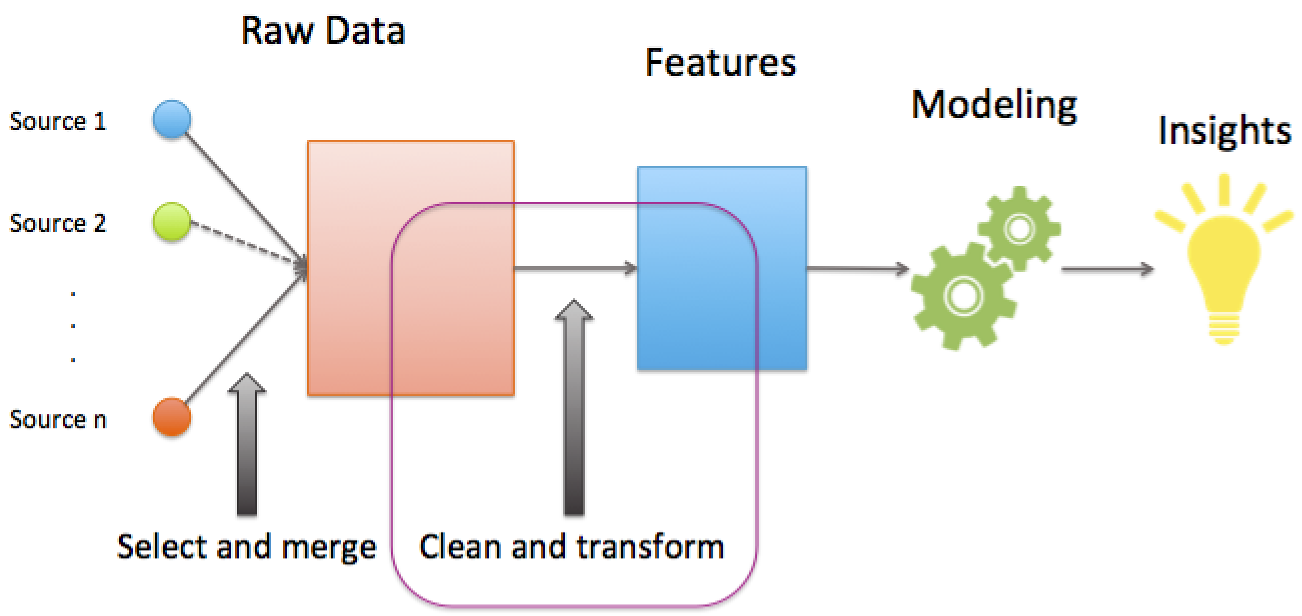


Figure 1-2. The place of feature engineering in the machine learning workflow.

The number of features is also important. If there are not enough informative features, then the model will be unable to fulfill the ultimate task. If there are too many features, or if most of them are irrelevant, then the model will be more expensive and tricky to train. Something could go awry in the training process that impacts the model’ s performance.

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.

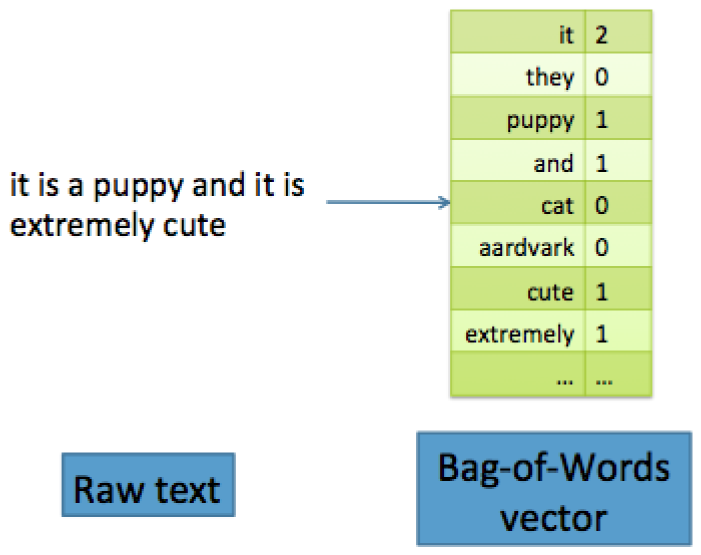


Figure 2-1. Turning raw text into bag-of-words representation

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.

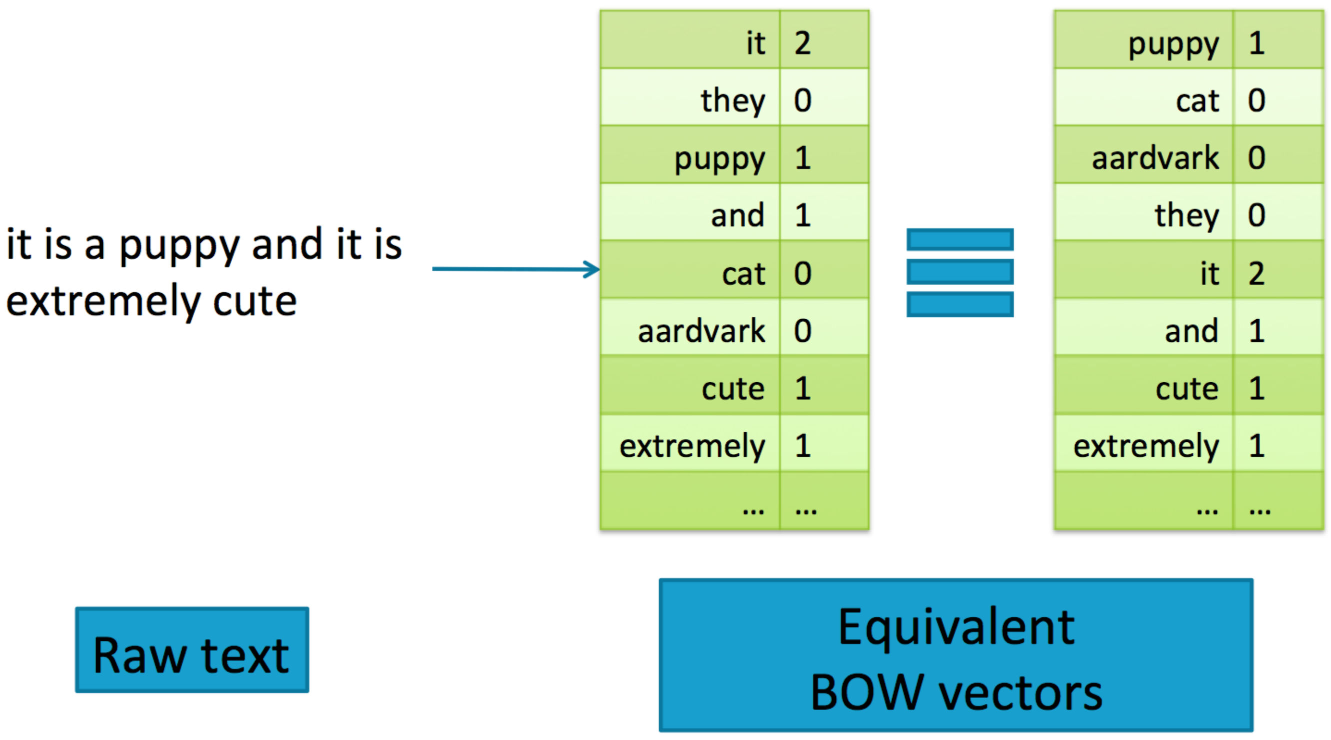


Figure 2-2. Two equivalent BOW vectors. The ordering of words in the vector is not important, as long as it is consistent for all documents in the dataset.

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.”

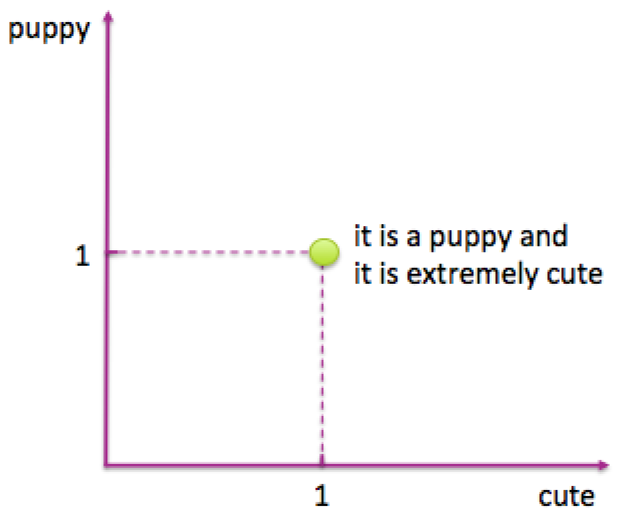


Figure 2-3. Illustration of a sample text document in feature space

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

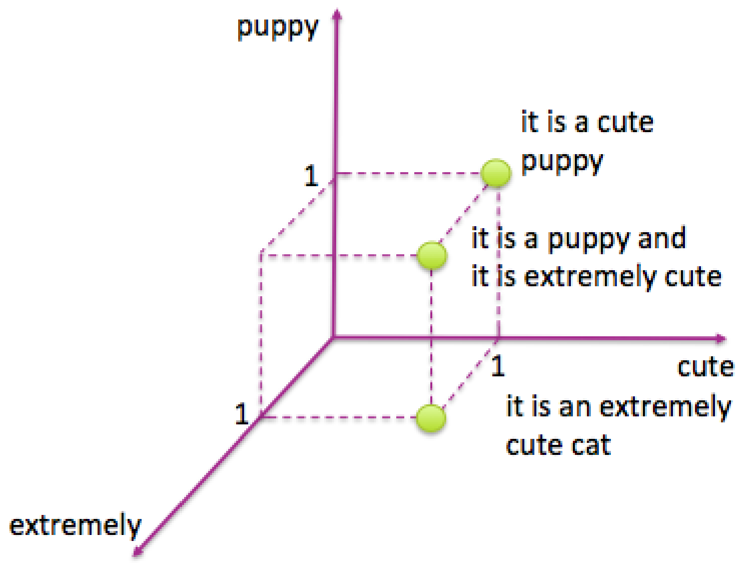

puppy
1
it is a cute puppy
it is a puppy and it is extremely cute
1
cute
1
it is an extremely cute cat
extremely

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.

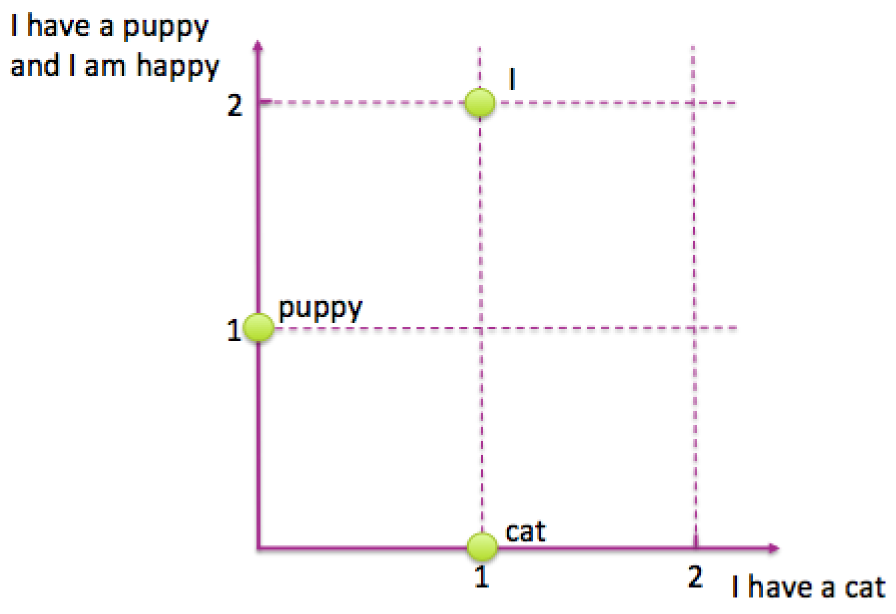


Figure 2-5. Word vectors in document space

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_revie
w.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']
```

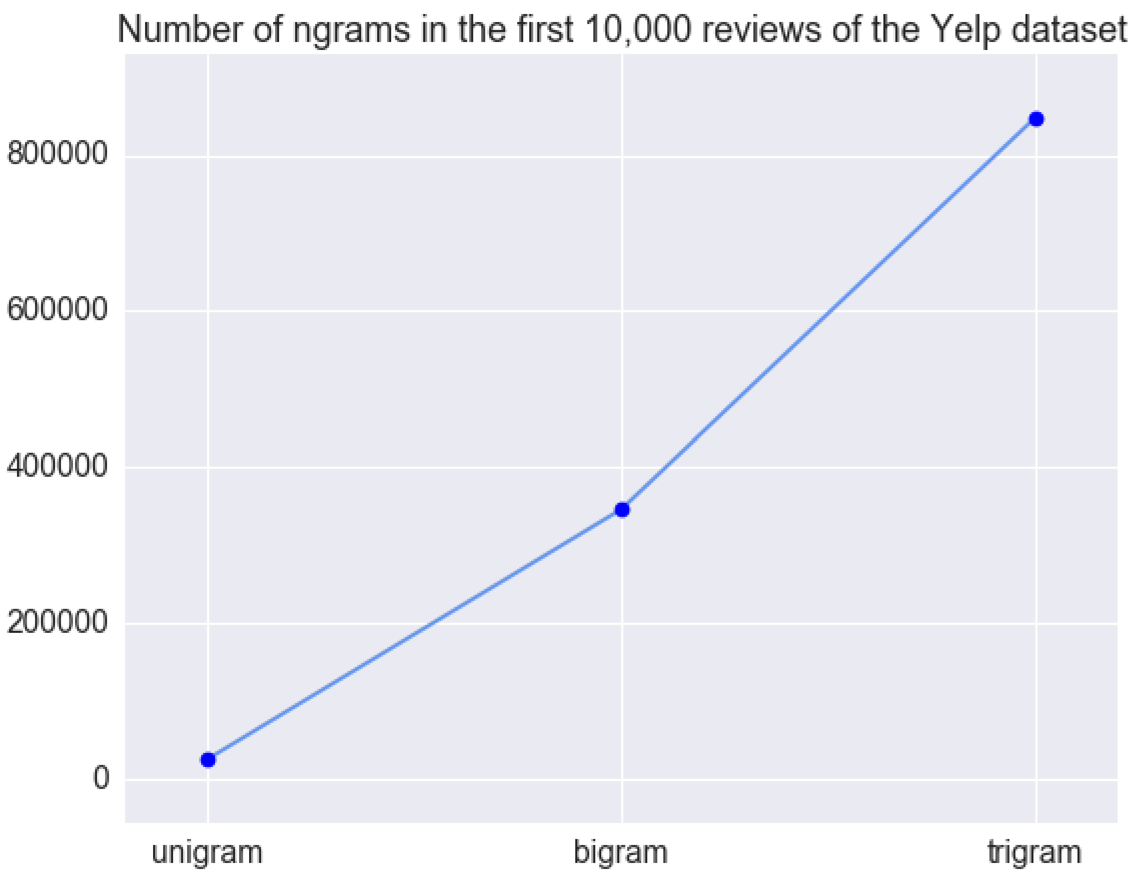


Figure 2-6. Number of unique n-grams in the first 10,000 reviews of the Yelp dataset.

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

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

Bigram	Document Count
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”.) 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.

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.

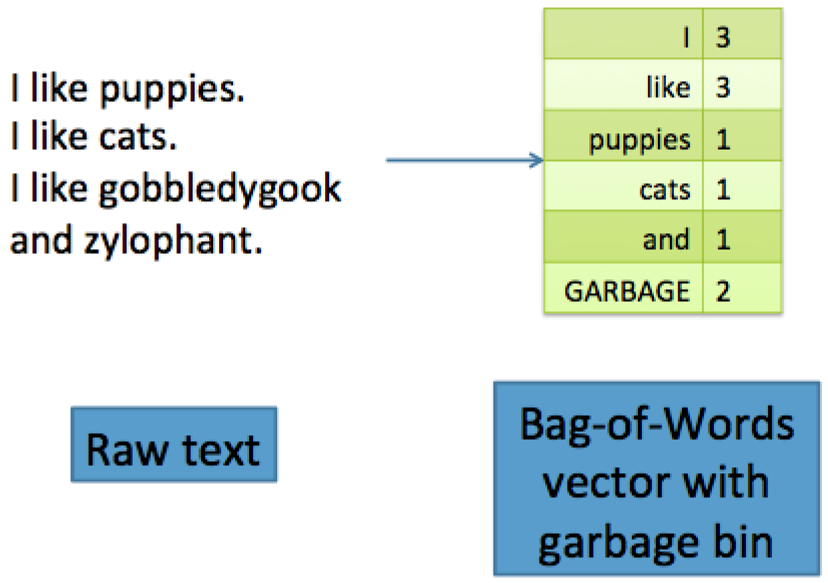


Figure 2-7. Bag-of-words feature vector with a garbage bin

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.

[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.

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.”

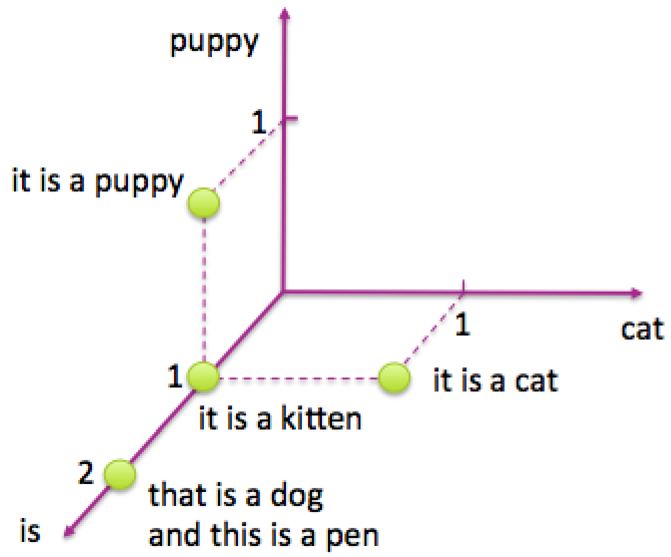


Figure 3-1. Four sentences about dog and cat

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.

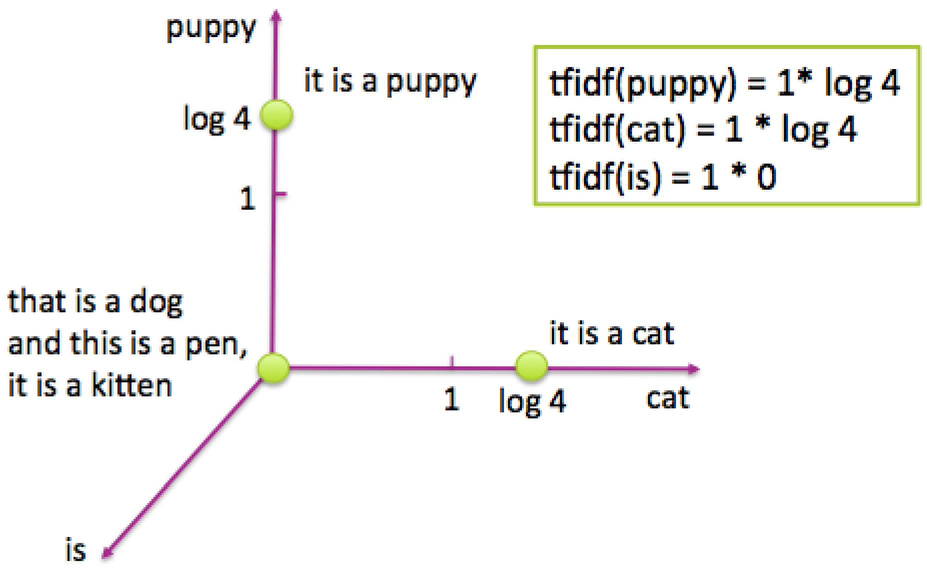


Figure 3-2. Tf-idf representation of the sentences in Figure 3-1

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.

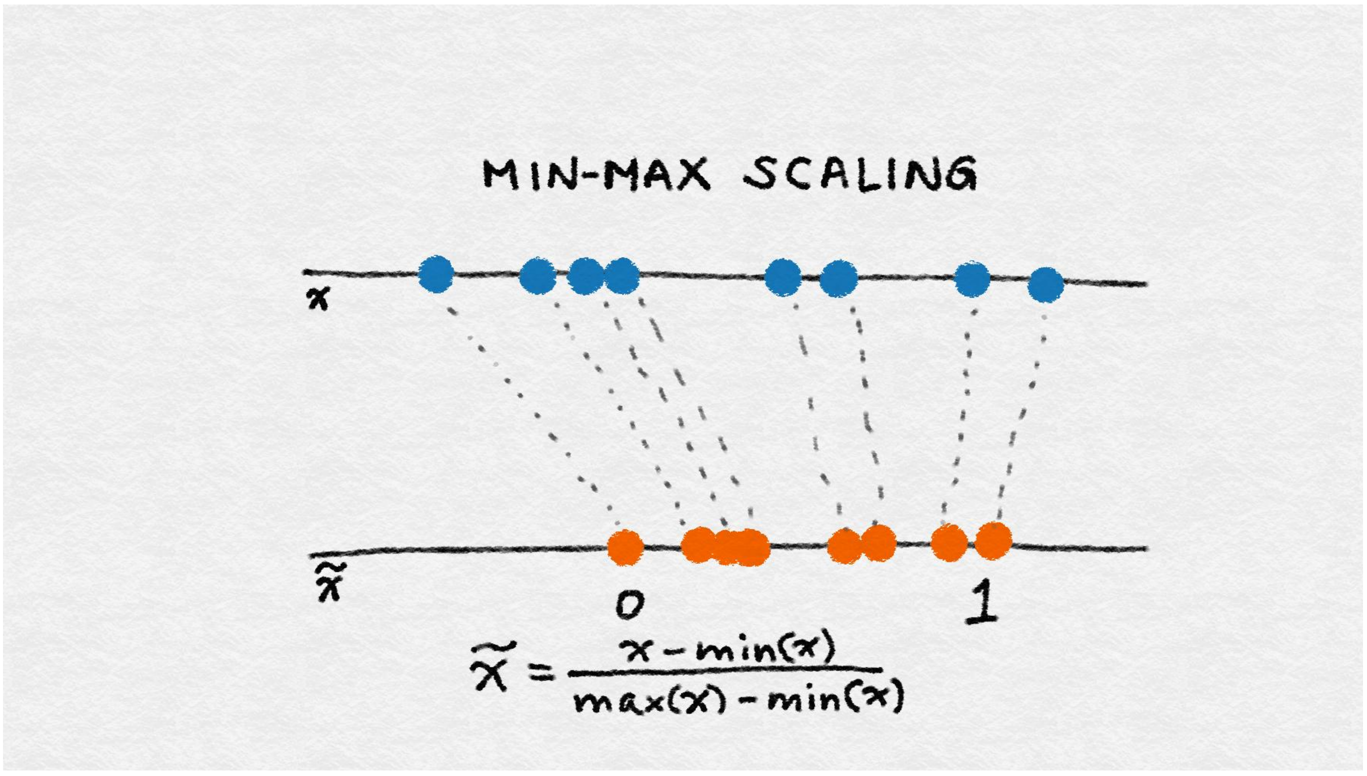


Figure 3-3. Min-max scaling

Standardization (variance scaling)

Here is the formula for standardization:

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

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.

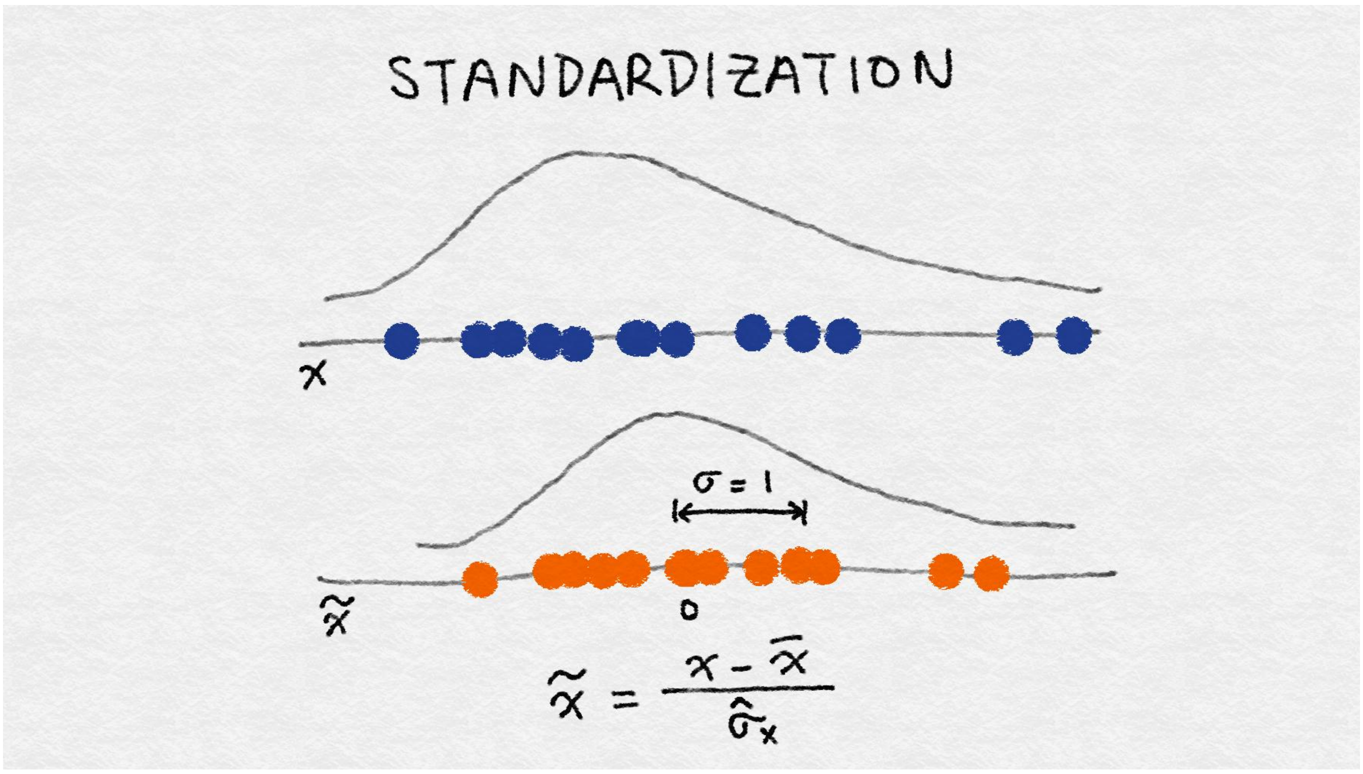


Figure 3-4. Illustration of feature standardization

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.

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

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" about the complementary nature of data vectors and feature vectors.)

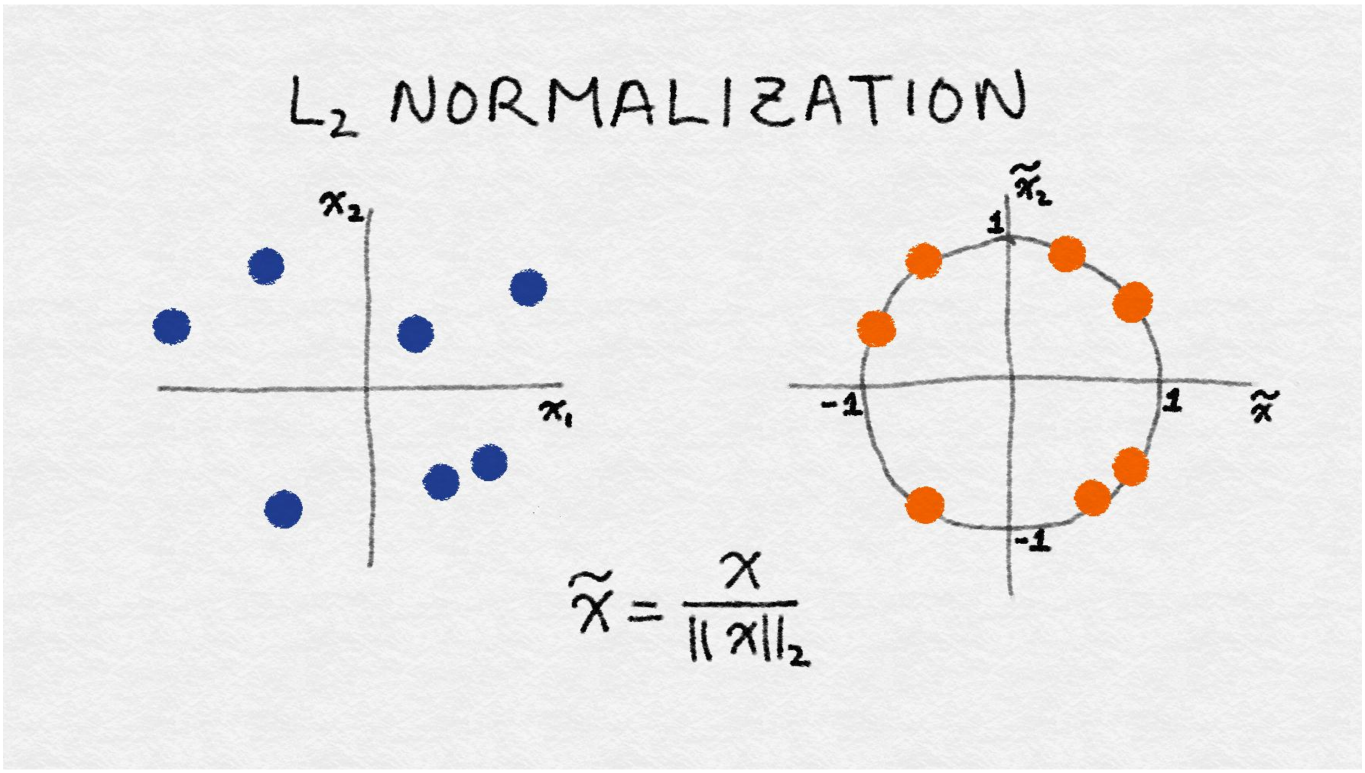


Figure 3-5. Illustration of L^2 feature normalization

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”.

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_penal
ty=0)
# On tf-idf features
model3 = gl.logistic_classifier.create(training_data,
                                                               features=['tfidf.bow'],
target='category',
                                                               feature_rescaling=False,
max_iterations=40,
                                                                                l2_penal
ty=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.)

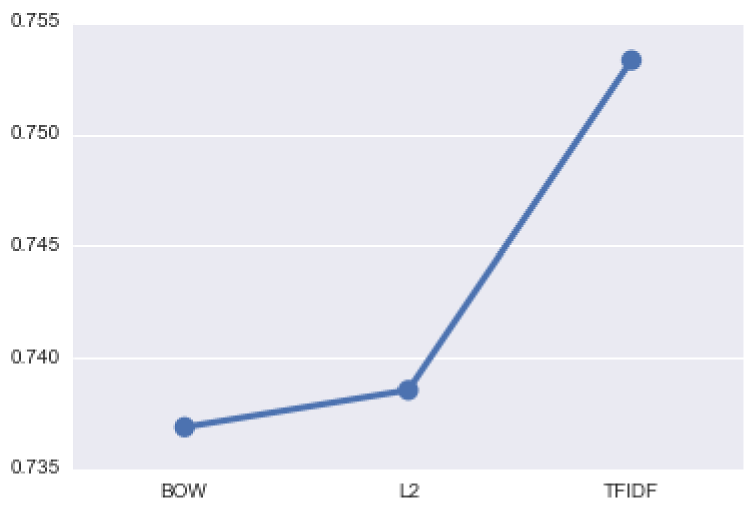


Figure 3-6. Test set accuracy for plain logistic regression classifiers

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.)

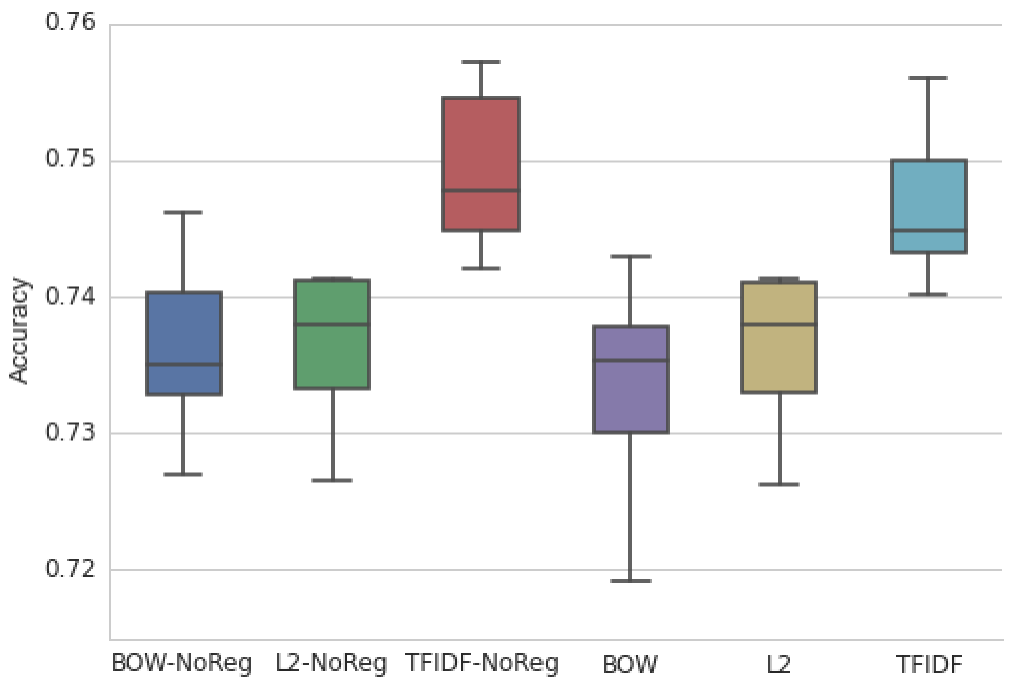


Figure 3-7. Distribution of classifier accuracy under each feature set and regularization setting, measured over resampled datasets

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.

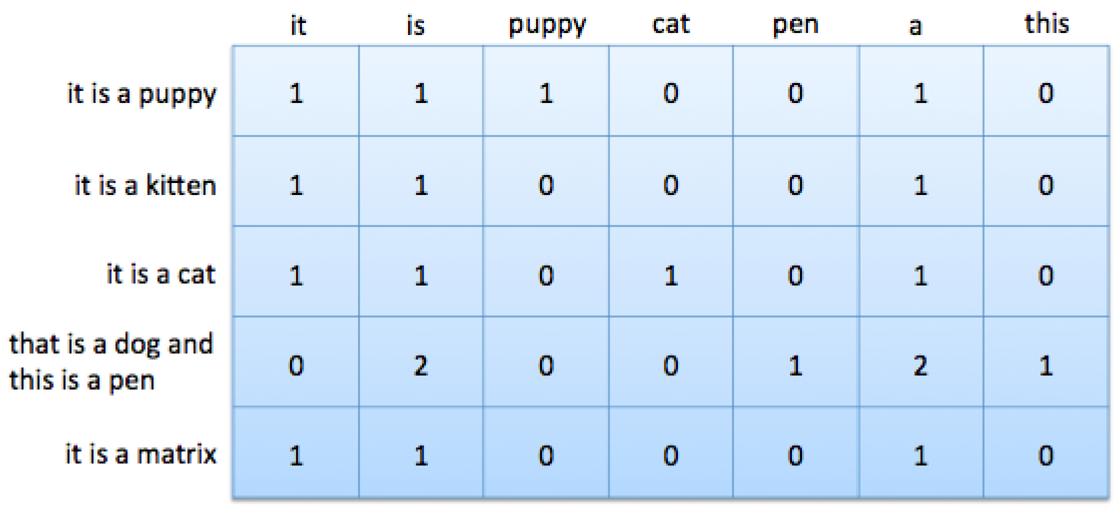
Keep in mind that these findings are limited to this dataset and this task. We certainly do not claim that ℓ_2 regularization and L^2 feature normalization are useless in general.

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[1], 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 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.

[1] 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.

Chapter 4. Counts and Categorical Variables: Counting Eggs in the Age of Robotic Chickens

Categorical variables can take on a (usually) finite number of values or categories. The categories may be represented numerically, but they cannot be ordered with respect to one another. They are *non-ordinal*.

A simple question can serve as litmus test for whether something should be a categorical variable: “Does it matter *how* different two values are, or only that they are different?” A stock price of $500 is five times higher than a price of $100. So stock price should be represented by a continuous numeric variable. The industry of the company (oil, travel, tech, etc.), on the other hand, should probably be categorical. Other examples of things that can be represented as a categorical variable are cities in the world, the four seasons in a year, and eye color of a person.

Counts are also very common in data sets: word occurrences in a document, the number of tweets from a user, the number of reviews on an item, etc. Counts are usually discrete, non-negative integers. They are ordinal, because a count of 3 is interpreted as higher than a count of 2.

Large categorical variables and counts are particularly common in transactional records. For instance, many web services track users using an id, which is a categorical variable with hundreds to hundreds of millions of values, depending on the number of unique users of the service. The IP address of an internet transaction is another example of a large categorical variable. They are categorical variables because, even though user ids and IP addresses are numeric, their magnitude is usually not relevant to the task at hand. For instance, the IP address might be relevant when doing fraud detection of individual transactions. Some IP addresses or subnets may generate more fraudulent transactions than others. But a subnet of 164.203.x.x is not inherently more fraudulent than 164.202.x.x; the numeric value of the subnet does not matter.

Technically speaking, categories and counts are distinct concepts. But they are also very much connected. The vocabulary of a document corpus can be interpreted as a large categorical variable, with the categories being unique words. If a category (e.g., word) can appear multiple times in a data point (document), then we can represent it as a count. Counts and categories also converge in bin-counting, which is a method for handling large categorical variables. We start this discussion with common considerations in dealing with categorical variables and counts. Eventually we will meander our way to a discussion of large categorical variables, which are very common in modern data sets.

Encoding Categorical Variables

The categories of a categorical variable are usually not numeric.[1] For example, eye color can be "black," "blue," "brown," etc. Thus, an encoding method is needed to turn these non-numeric categories into numbers. It is tempting to simply assign an integer, say 1 to *k*, to each of *k* possible categories. But the resulting values would be orderable against each other, which should not be permissible for categories.

One-hot encoding

A better method is to use a group of bits. Each bit represents a possible category. If the variable cannot be multiple categories at once, then only one bit in the group can be on. This is called *one-hot encoding*, and it is implemented in Scikit Learn as sklearn.preprocessing.OneHotEncoder. Each of the bits is a feature. Thus a categorical variable with *k* possible categories is encoded as a feature vector of length *k*.

Table 4-1. One-hot encoding of a category of three cities.

City	e_1	e_2	e_3
San Francisco	1	0	0
New York	0	1	0
Seattle	0	0	1

One-hot encoding is very simple to understand. But it uses one more bit than strictly necessary. If we see that *k-1* of the bits are zero, then the last bit must be one because the variable must take on one of *k* values. Mathematically, one can write down this constraint as "the sum of all bits must be equal to 1."

Equation 4-1. Constraint on one-hot encoding $e_1, \ldots, e_k$

$$e_1 + e_2 + \ldots e_k = 1$$

Thus we have a linear dependency on our hands. Linear dependent features, as we discover in Chapter 3, are slightly annoying because it means that the trained linear models will not be unique. Different linear combinations of the features can make the same predictions, so we would need to jump through extra hoops to understand the effect of the feature on the prediction.

Dummy coding

The problem with one-hot encoding is that it allows for *k* degrees of freedom, where the variable itself needs only *k-1*. *Dummy coding*[2] removes the extra degree of freedom by using only *k-1* features in the representation. One feature is thrown under the bus and represented by the vector of all zeros. This is known as the *reference category*. Dummy coding and one-hot encoding are both implemented in Pandas as pandas.get_dummies.

Table 4-2. Dummy coding of a category of three cities.

City	e_1	e_2
San Francisco	1	0
New York	0	1
Seattle	0	0

The outcome of modeling with dummy coding is more interpretable than one-hot encoding. This is easy to see in a simple linear regression problem. Suppose we have some data about apartment rental prices in three cities: San Francisco, New York, and Seattle. (See Table 4-3.)

Table 4-3. Toy dataset of apartment prices in three cities.

	City	Rent
0	SF	3999
1	SF	4000
2	SF	4001
3	NYC	3499
4	NYC	3500
5	NYC	3501
6	Seattle	2499
7	Seattle	2500
8	Seattle	2501

We can train a linear regressor to predict rental price based solely on the identity of the city.

The linear regression model can be written as

y = w1 x1 + ... + wn xn.

It is customary to fit an extra constant term called the intercept, so that y can be a non-zero value when the x’ s are zero.

y = w1 x1 + ... + wn xn + b.

Example 4-1. Linear regression on a categorical variable using one-hot and dummy codes.

```
>>> import pandas
>>> from sklearn import linear_model

###########
### Define a toy dataset of apartment rental prices in
### New York, San Francisco, and Seattle
>>> df = pd.DataFrame({'City': ['SF', 'SF', 'SF', 'NYC', 'NYC', 'NYC',
...                             'Seattle', 'Seattle', 'Seattle'],
...                              'Rent': [3999, 4000, 4001, 3499, 3500, 3501, 2499, 2500,
2501]})
>>> df['Rent'].mean()
3333.3333333333335

###########
### Convert the categorical variables in the dataframe to one-hot encoding
### and fit a linear regression model
>>> one_hot_df = pd.get_dummies(df, prefix=['city'])
>>> one_hot_df
   Rent  city_NYC  city_SF  city_Seattle
0  3999     0.0       1.0       0.0
1  4000     0.0       1.0       0.0
2  4001     0.0       1.0       0.0
3  3499     1.0       0.0       0.0
4  3500     1.0       0.0       0.0
5  3501     1.0       0.0       0.0
6  2499     0.0       0.0       1.0
7  2500     0.0       0.0       1.0
8  2501     0.0       0.0       1.0

>>> model = linear_regression.LinearRegression()
>>> model.fit(one_hot_df[['city_NYC', 'city_SF', 'city_Seattle']],
...           one_hot_df['Rent'])
>>> model.coef_
array([ 166.66666667,   666.66666667, -833.33333333])
>>> model.intercept_
3333.3333333333335

##########
```

```
### Train a linear regression model on dummy code
### Specify the 'drop_first' flag to get dummy coding
>>> dummy_df = pd.get_dummies(df, prefix=['city'], drop_first=True)
>>> dummy_df
      Rent    city_SF    city_Seattle
0    3999        1.0             0.0
1    4000        1.0             0.0
2    4001        1.0             0.0
3    3499        0.0             0.0
4    3500        0.0             0.0
5    3501        0.0             0.0
6    2499        0.0             1.0
7    2500        0.0             1.0
8    2501        0.0             1.0

>>> model.fit(dummy_df[['city_SF', 'city_Seattle']], dummy_df['Rent'])
>>> model.coef_
array([    500., -1000.])
>>> model.intercept_
3500.0
```

With one-hot encoding, the intercept term represents the global mean of the target variable, Rent, and each of the linear coefficients represent how much that city's average rent differs from the global mean.

With dummy coding, the bias coefficient represents the mean value of the response variable y for the reference category, which in the example is the city NYC. The coefficient for the i-th feature is equal to the difference between the mean response value for the i-th category and the mean of the reference category.

[illustration of rental price in cities]

[illustration of learned coefficients]

[geometry of one-hot vs. dummy encoding?]

Effect coding

Yet another variant of categorical variable encoding is called *effect coding*. Effect coding is very similar to dummy coding, with the difference that the reference category is now represented by the vector of all -1's.

Table 4-4. Effect coding of a categorical variable representing three cities.

City	e_1	e_2
San Francisco	1	0
New York	0	1
Seattle	-1	-1

Effect coding is very similar to dummy coding, but results in linear regression models that are even simpler to interpret. Example 4-2 demonstrates what happens with effect coding as input. The intercept term represents the global mean of the target variable, and the individual coefficients indicate how much the mean of the individual categories differ from the global mean. (This is called the *main effect* of the category or level, hence the name "effect coding.") One-hot encoding actually came up with the same intercept and coefficients, but in that case there are linear coefficients for each city. In effect coding, no single feature represents the reference category. So the effect of the reference category needs to be separately computed as the negative sum of the coefficients of all other categories. (See *What is effect coding?* for more details.)

Example 4-2. Linear regression with effect coding.

```
>>> effect_df = dummy_df.copy()
>>> effect_df.ix[3:5, ['city_SF', 'city_Seattle']] = -1.0
>>> effect_df
     Rent    city_SF    city_Seattle
0    3999        1.0             0.0
1    4000        1.0             0.0
2    4001        1.0             0.0
3    3499       -1.0            -1.0
4    3500       -1.0            -1.0
5    3501       -1.0            -1.0
6    2499        0.0             1.0
7    2500        0.0             1.0
8    2501        0.0             1.0

>>> model.fit(effect_df[['city_SF', 'city_Seattle']], effect_df['Rent'])
>>> model.coef_
array([ 666.66666667, -833.33333333])
>>> model.intercept_
3333.3333333333335
```

Pros and cons of categorical variable encodings

One-hot, dummy, and effect coding are very similar to one another. They each have pros and cons. One-hot encoding is redundant, which allows for multiple valid models for the same problem. The non-uniqueness is sometimes problematic for interpretation. The advantage is that each feature clearly corresponds to a category. Moreover, missing data can be encoded as the all-zeroes vector, and the output should be the overall mean of the target variable.

Dummy coding and effect coding are not redundant. They give rise to unique and interpretable models. The downside of dummy coding is that it cannot easily handle missing data, since the all-zeroes vector is already mapped to the reference category. It also encodes the effect of each category relative to the reference category, which may look strange. Effect coding avoids this problem by using a different code for the reference category. But the vector of all -1’ s is a dense vector, which is expensive for both storage and computation. For this reason, popular ML software packages such as Pandas and Scikit Learn have opted for dummy coding or one-hot encoding instead of effect coding.

All three encoding techniques break down when the number of categories become very large. Different strategies are needed to handle extremely large categorical variables. We will cover this topic in “Dealing with Large Categorical Variables”.

Dealing with Counts

The main things to consider about counts are their scale and distribution. These questions are relevant for some models but not all. But when they are relevant, they can be crucial.

Scale matters to linear models such as logistic regression and support vector machine, but not to scale-independent models such as space-partitioning trees (decision trees, gradient boosted machine, random forest). However, even for trees, counts that grow unbounded could also be problematic.

Distribution matters to models that make explicit or implicit assumptions about the geometry of the input or output variables. For instance, k-means assumes Euclidean geometry in the input space, which is satisfied when the input distribution is close to Gaussian. Ordinary linear regression assumes that the noise in the output is Gaussian. In the case of k-means, one should control the distribution of the input. In the case of linear regression, it is the output that one needs to clean up.

Binarization

The first sanity check is whether the counts should be treated as counts, or as a binary variable to indicate presence.

The user taste profile in the Million Song Dataset contains the full music listening histories of one million users on the Echo Nest. This comes to more than 48 million triplets of user id, music id, and listen count. Suppose the task is to build a recommender to predict how well a user likes a particular song. Should we try to predict the raw listen count? It’ s a reasonable idea, because the more someone likes a song, the more times he or she would play it. However, upon examination of the data, we find that, while 99% of the listen counts are 24 or lower, the maximum listen count is 9,667. There are a few other listen counts in the thousands. (As Figure 4-1 shows, the histogram peaks in the bin closest to 0. But more than 10,000 triplets have greater counts, with a few in the thousands.) These values are anomalously large; if we were to try to predict the actual listen count, the model would be pulled way off course by these large values.

In the internet age, such large counts are far from anomalous. A user might put a song or a movie on infinite playback or use a script to repeatedly check for the availability of tickets for a popular show. These automated mechanisms are like robotic chickens that keep laying eggs. When data can be produced at high volume and velocity, they are very likely to contain a few extreme values. It is a good idea to check for them.

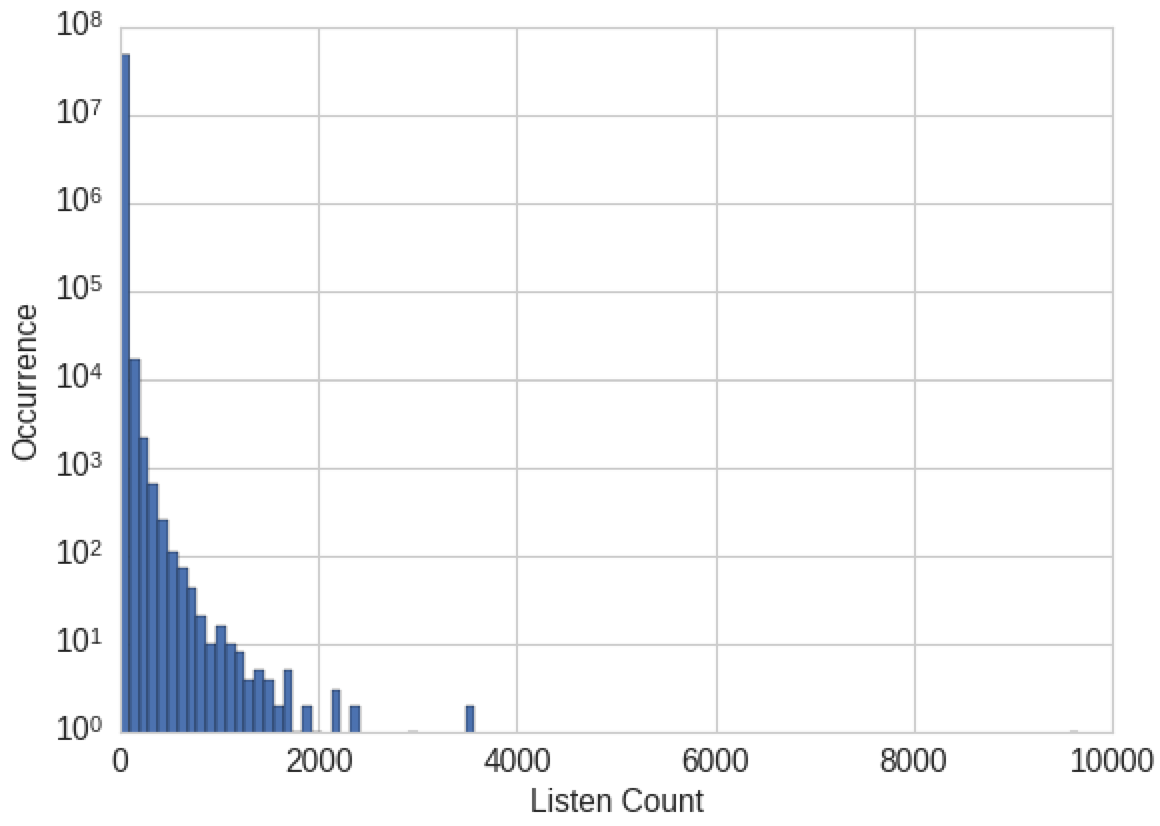


Figure 4-1. Histogram of listen counts in the user taste profile of the Million Song Dataset. Note that the y-axis is on a log scale.

In this case, the raw listen count is not a *robust* measure of user taste. (In statistical lingo, robustness means whether the method works under a wide variety of conditions.) Users have different listening habits. Some people might put their favorite songs on infinite loop, while others might savor them only on special occasions. Without getting into detailed user behavior modeling, it’ s hard to say that someone who listens to a song 20 times must like it twice as much as someone else who listens to it 10 times.

Instead of using the raw listen count as target, we can binarize the count and clip all counts greater than 1 to 1. In other words, if the user listened to a song at least once, then we count it as the user liking the song. This way, the model would not need to spend cycles on predicting the minute differences between the raw counts. The binary target is simpler and possibly better aligned to the underlying goal.

This is an example where we engineer the target variable of the model. Strictly speaking, the target is not a feature because it’ s not the input. But on occasion we do need to modify the target in order to solve the right problem.

Quantization or binning

As another example, let’ s look again at the Yelp reviews dataset. Each business has a review count. Suppose our task is to use collaborative filtering to predict the rating a user might give to a business. The review count might be a useful input feature because there is usually a strong correlation between popularity and good ratings. Now the question is, should we use the raw review count or process it further? Figure 4-2 shows the histogram of all business review counts. We see the same pattern as in music listen counts. Most of the counts are small, but some businesses have reviews in the thousands.

Example 4-3. Visualizing business review counts in the Yelp dataset.

```
>>> import pandas as pd
>>> import json

### Load the data about businesses
>>> biz_file = open('yelp_academic_dataset_business.json')
>>> biz_df = pd.DataFrame([json.loads(x) for x in biz_file.readlines()])
>>> biz_file.close()

>>> import matplotlib.pyplot as plt
>>> import seaborn as sns

### Plot the histogram of the review counts
>>> sns.set_style('whitegrid')
>>> fig, ax = plt.subplots()
>>> biz_df['review_count'].hist(ax=ax, bins=100)
>>> ax.set_yscale('log')
>>> ax.tick_params(labelsize=14)
>>> ax.set_xlabel('Review Count', fontsize=14)
>>> ax.set_ylabel('Occurrence', fontsize=14)
```

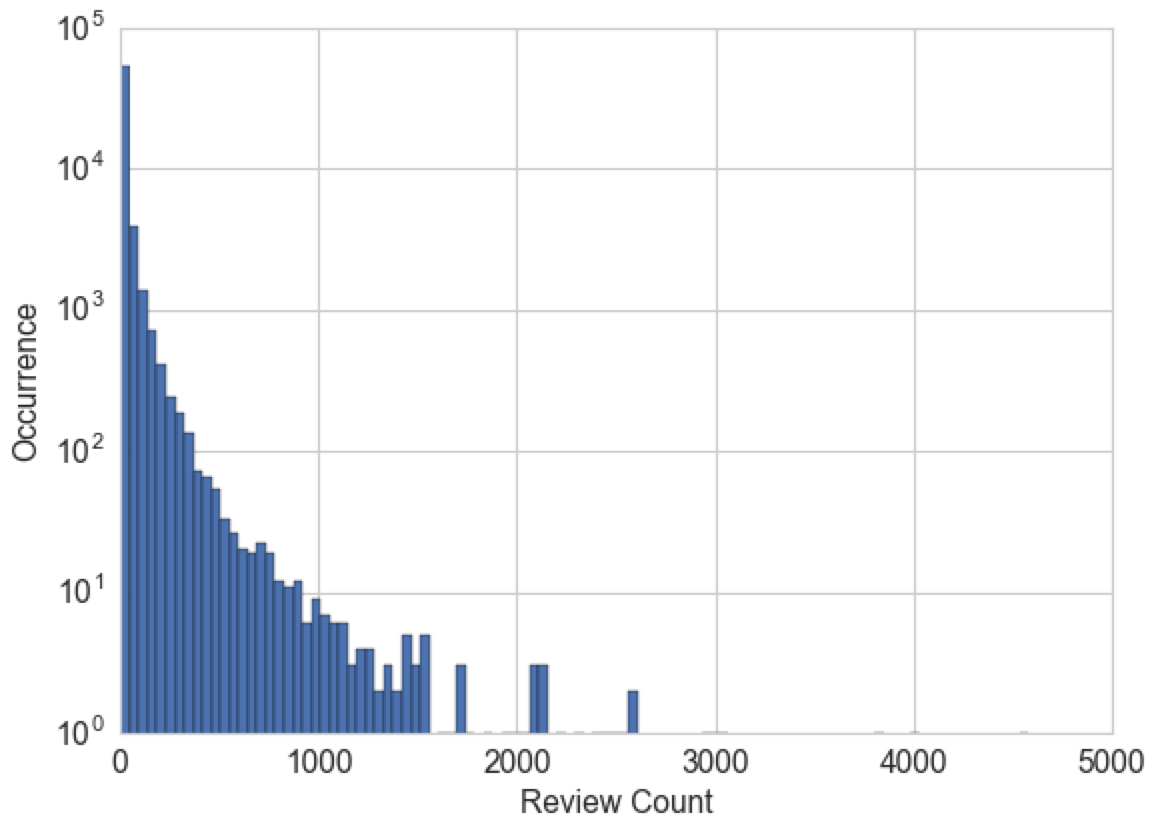


Figure 4-2. Histogram of business review counts in the Yelp reviews dataset. The y-axis is on a log-scale.

Raw counts that span several orders of magnitude are problematic for many models. In a linear model, the same linear coefficient would have to work for all possible values of the count. This includes the case of matrix factorization (a collaborative filtering model), where the count might be included as a linear side feature. It would also wreak havoc on the similarity metrics employed in clustering models such as k-means, where similarity is measured using Euclidean distance.

One solution is to quantize the counts, i.e., assign them into ordered bins to indicate intensity. *Quantization* is the process of mapping a continuous number to a discrete one. We can think of the discretized numbers as an ordered sequence of bins that represent a measure of intensity.

In order to quantize or bin the data, we have to decide how wide each bin should be. The solutions fall under two types: fixed-width or adaptive. We will give an example of each type.

Fixed-width binning

With fixed-width binning, each bin contains a specific numeric range. The ranges can be custom designed or automatically segmented, and they can be linearly scaled or exponentially scaled. For example, we can group a person’ s age into decades: 0-9 years old fall under bin 1, 10-19 years fall under bin 2, etc. To map from the count to the bin, simply divide by the width of the bin and take the integer part.

It’ s also common to see custom designed age ranges that better correspond to stages of life:

- 0-12 years old
- 12-17 years old
- 18-24 years old
- 25-34 years old
- 35-44 years old
- 45-54 years old
- 55-64 years old
- 65-74 years old
- 75 years or older

When the numbers span multiple magnitudes, it may be better to group by powers of 10 (or powers of any constant): 0-9, 10-99, 100-999, 1000-9999, etc. The bin widths grow exponentially, going from O(10), O(100), to O(1000) and beyond. To map from the count to the bin, take the log of the count. This is very much related to the log transform, which we discuss in “Log transformation”.

Example 4-4. Quantizing counts with fixed-width bins.

```
>>> import numpy as np

### Generate 20 random integers uniformly between 0 and 99
>>> small_counts = np.random.randint(0, 100, 20)
>>> small_counts
array([30, 64, 49, 26, 69, 23, 56,    7, 69, 67, 87, 14, 67, 33, 88, 77, 75,
              47, 44, 93])
### Map to evenly spaced bins 0-9 by division
>>> np.floor_divide(small_counts, 10)
array([3, 6, 4, 2, 6, 2, 5, 0, 6, 6, 8, 1, 6, 3, 8, 7, 7, 4, 4, 9], dtype=int32)

### An array of counts that span several magnitudes
>>> large_counts = [296, 8286, 64011, 80, 3, 725, 867, 2215, 7689, 11495, 91897, 44, 28,
7971, 926, 122, 22222]
### Map to exponential-width bins via the log function
```

```
>>> np.floor(np.log10(large_counts))
array([ 2.,    3.,    4.,    1.,    0.,    2.,    2.,    3.,    3.,    4.,    4.,    1
.,    1.,
                3.,    2.,    2.,    4.])
```

Quantile binning

Fixed-width binning is easy to compute. But if there are large gaps in the counts, then there will be many empty bins with no data. This problem can be solved by adaptively positioning the bins based on the distribution of data. This can be done using the quantiles of the distribution.

Quantiles are values that divide the data into equal portions. For example, the median divides the data in halves; half the data are smaller, and half larger than the median. The quartiles divide the data into quarters, the deciles into tenths, etc. Example 4-5 demonstrates how to compute the deciles of the Yelp business review counts, and Figure 4-3 overlays the deciles on the histogram. This gives a much clearer picture of the skew towards smaller counts.

[would the review count contain “leakage”? should we do “leave-one-out” for this count?]

Example 4-5. Computing deciles of Yelp business review counts.

```
>>> deciles = biz_df['review_count'].quantile([.1, .2, .3, .4, .5, .6, .7, .8, .9])
>>> deciles
0.1          3.0
0.2          4.0
0.3          5.0
0.4          6.0
0.5          8.0
0.6         12.0
0.7         17.0
0.8         28.0
0.9         58.0
Name: review_count, dtype: float64

### Visualize the deciles on the histogram
>>> sns.set_style('whitegrid')
>>> fig, ax = plt.subplots()
>>> biz_df['review_count'].hist(ax=ax, bins=100)
>>> for pos in deciles:
...     handle = plt.axvline(pos, color='r')
>>> ax.legend([handle], ['deciles'], fontsize=14)
>>> ax.set_yscale('log')
>>> ax.set_xscale('log')
>>> ax.tick_params(labelsize=14)
>>> ax.set_xlabel('Review Count', fontsize=14)
>>> ax.set_ylabel('Occurrence', fontsize=14)
```

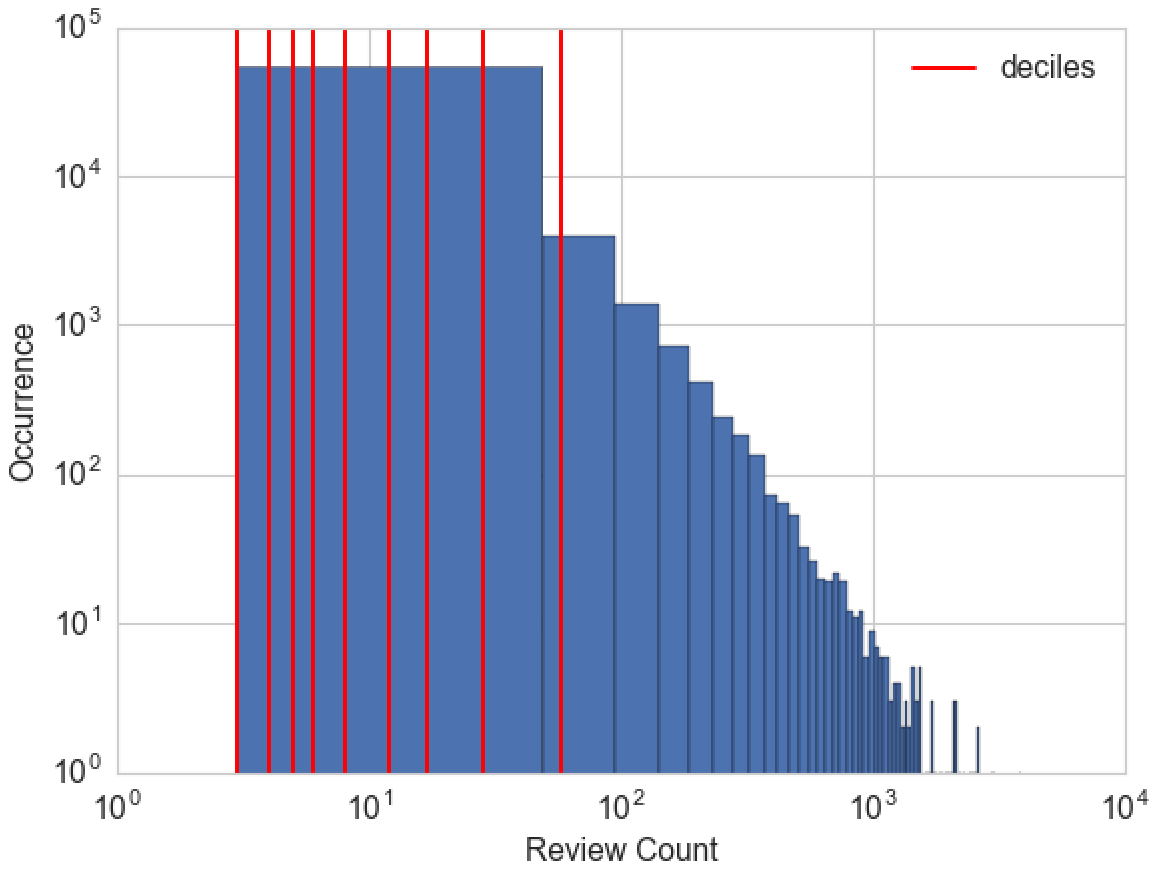


Figure 4-3. Deciles of the review counts in the Yelp reviews dataset. Note that both x- and y-axes are in log scale.

To compute the quantiles and map data into quantile bins, we can use the Pandas library. pandas.DataFrame.quantile and pandas.Series.quantile compute the quantiles. pandas.qcut maps data into a desired number of quantiles.

Example 4-6. Binning counts by quantiles.

```
### Continue example Example 4-4 with large_counts
>>> import pandas as pd

### Map the counts to quartiles
>>> pd.qcut(large_counts, 4, labels=False)
array([1, 2, 3, 0, 0, 1, 1, 2, 2, 3, 3, 0, 0, 2, 1, 0, 3], dtype=int64)
```

```
### Compute the quantiles themselves
>>> large_counts_series = pd.Series(large_counts)
>>> large_counts_series.quantile([0.25, 0.5, 0.75])
0.25          122.0
0.50          926.0
0.75         8286.0
dtype: float64
```

Log transformation

In "Quantization or binning", we briefly introduced the notion of taking the logarithm of the count to map them to exponential sized bins. There is much more to the log transform. So let's take a better look now.

Recall that the log function is defined such that log(1) = 0 and $\log_{10}(10^x) = x$. It maps the small range between (0, 1] to the entire range of negative numbers (−∞, 0], the range of [1, 10] to [0, 1], [10, 100] to [1, 2], and so on. In other words, it compresses the range of large numbers and expands the range of small numbers. The larger x is, the slower log(x) increments. This is easier to digest by looking at a plot of the log function. (See Figure 4-4.)

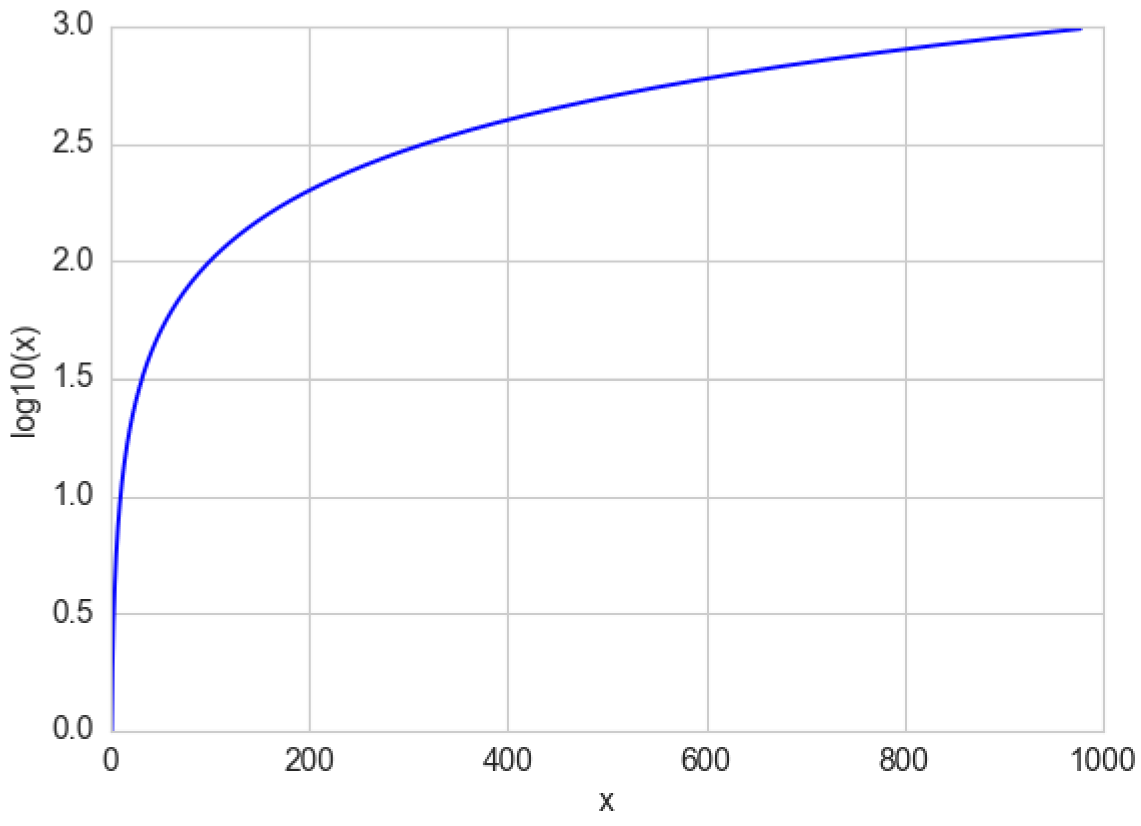


Figure 4-4. A plot of the log function.

The log transform is a powerful tool for dealing with positive numbers with a heavy-tailed distribution. It compresses the long tail in the high end of the distribution into a shorter tail, and expands the low end into a longer head. Figure 4-5 compairs the histograms of Yelp business review counts before and after log transformation. The y-axes are now both on normal (linear) scale. The increased bin spacing in the bottom plot between the range of (0.5, 1] is due to there being only 10 possible integer counts between 1 and 10. Notice that the original review counts are very concentrated in the low count region, with outliers stretching out above 4,000. After log transformation, the occurrence frequencies are more spread out in the low end, and the high end is compressed into a shorter tail.

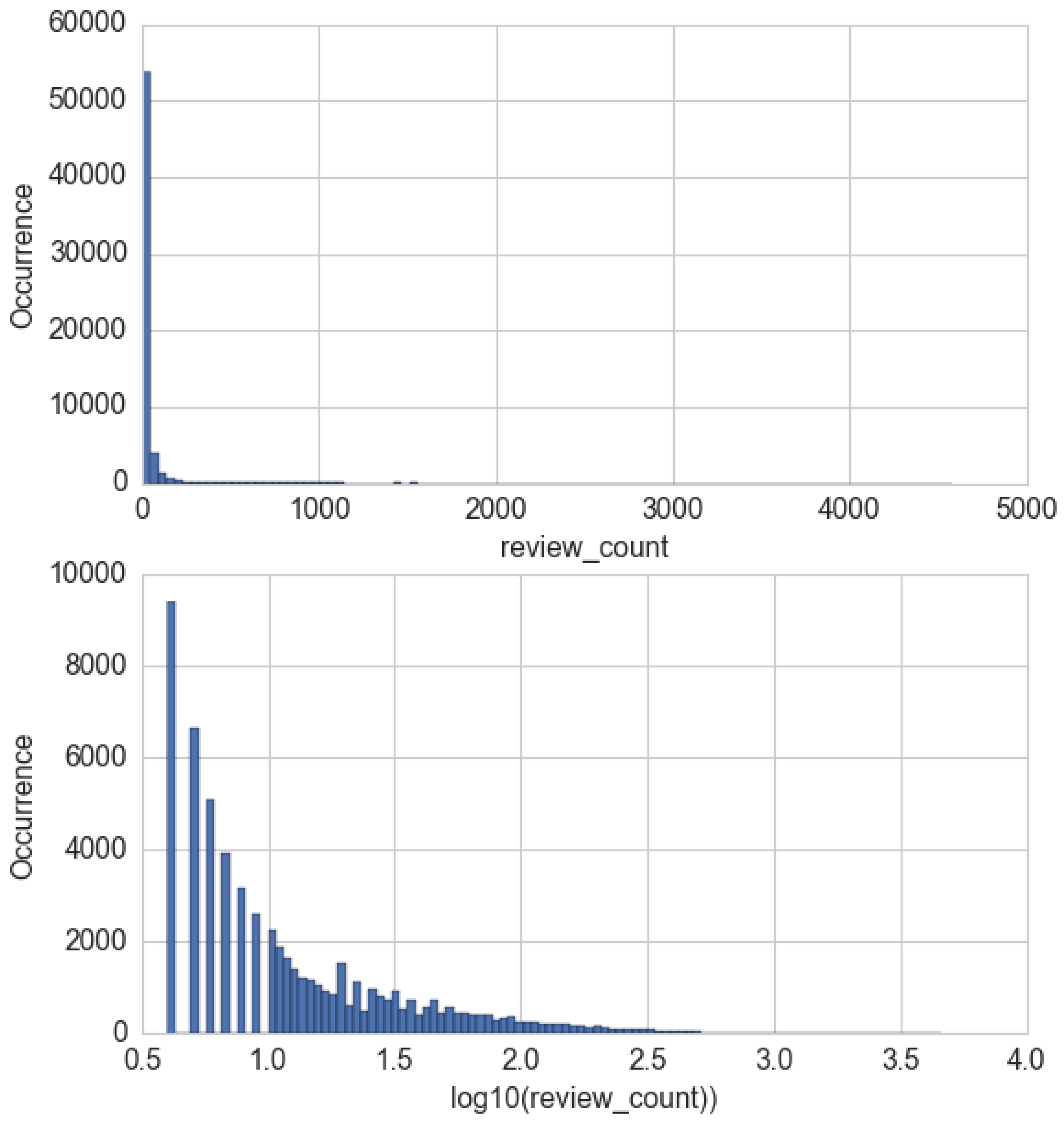


Figure 4-5. Comparison of Yelp business review counts before (top) and after (bottom) log transformation.

The log transform is an example of a *variance stabilization transform*.

[Need example of log transform being useful in a dataset. Use Kaggle bike as a last resort because it works better with the log RMSE but not the normal one. Log of time intervals]

Dealing with Large Categorical Variables

Automated data collection on the internet can generate large categorical variables. This is common in applications such as targeted advertising and fraud detection. In targeted advertising, the task is to match a user with a set of ads, given the user’s search query or the current page. Features include the user id, the website domain for the ad, the search query, the current page, and all possible pairwise conjunctions of those features. (The query is a text string that can be chopped up and turned into the usual text features. However, queries are generally short and are often composed of phrases. So the best course of action in this case is usually to keep them in tact, or passed through a hash function to make storage and comparisons easier. We will discuss hashing in more detail below.) Each of these is a very large categorical variable. The challenge is to find a good feature representation that is memory efficient, yet produces accurate models that are fast to train.

Existing solutions can be categorized (harhar) thus:

1. Do nothing fancy with the encoding. Use a simple model that is cheap to train. Feed one-hot encoding into a linear model (logistic regression or linear support vector machine) on lots of machines.
2. Compress the features. There are two choices.
   a. Feature hashing—popular with linear models.
   b. Bin-counting—popular with linear models as well as trees.

Using the vanilla one-hot encoding is a valid option. For Microsoft’s search advertising engine, Graepel et al. [2010] report using such binary-valued features in a Bayesian probit regression model that can be trained online using simple updates. Meanwhile, other groups argue for the compression approach. Researchers from Yahoo! swear by feature hashing [Weinberger et al. 2009]. Though McMahan et al [2013] experimented with feature hashing on Google’s advertising engine and did not find significant improvements. Yet other folks at Microsoft are taken with the idea of bin-counting [Bilenko, 2015].

As we shall see, all of these ideas have pros and cons. We will first describe the solutions themselves, then discuss their trade-offs.

Feature hashing

A *hash function* is a deterministic function that maps a potentially unbounded integer to a finite integer range [1, *m*]. Since the input domain is potentially larger than the output range, multiple numbers may get mapped to the same output. This is called a *collision*. A *uniform hash function* ensures that roughly the same number of numbers are mapped into the each of the *m* bins. Visually, we can think of a hash function as a machine that intakes numbered balls and routes them to one of *m* bins. Balls with the same number will always get routed to the same bin.

Hash functions can be constructed for any object that can be represented numerically (which is true for any data that can be stored on a computer): numbers, strings, complex structures, etc.

When there are very many features, storing the feature vector could take up a lot of space. Feature hashing compresses the original feature vector into an m-dimensional vector by applying a hash function to the feature id. For instance, if the original features were words in a document, then the hashed version would have a fixed vocabulary size of *m*, no matter how many unique words there are in the input.

Example 4-7. Feature hashing for word features

```
def hash_features(word_list, m):
     output = [0] * m
     for word in word_list:
         index = hash_fcn(word) % m
         output[index] += 1
     return output
```

Another variation of feature hashing adds a sign component, so that counts are either added or subtracted from the hashed bin. This ensures that the inner products between hashed features are equal in expectation to that of the original features.

Example 4-8. Signed feature hashing

```
def hash_features(word_list, m):
        output = [0] * m
        for word in word_list:
                index = hash_fcn(word) % m
         sign_bit = sign_hash(word) % 2
         if (sign_bit == 0):
                    output[index] -= 1
         else:
             output[index] += 1
        return output
```

The value of the inner product after hashing is within $O(\frac{1}{\sqrt{m}})$ of the original inner product. So the size of the hash table *m* can be selected based on acceptable errors. In practice, picking the right *m* could take some trial and error.

Feature hashing can be used for models that involve the inner product of feature vectors and coefficients, e.g., linear models and kernel methods. It has been demonstrated to be successful in the task of spam filtering [Weinberger et al., 2009]. In the case of targeted advertising, McMahan et al. [2013] report not being able to get the prediction errors down to an acceptable level unless *m* were on the order of billions, which does not constitute enough saving in space.

One downside to feature hashing is that the hashed features, being aggregates of original features, are no longer interpretable.

Bin-counting

Bin-counting is one of the perennial rediscoveries in machine learning. It has been reinvented and used in a variety of applications from ad click-through rate prediction to hardware branch prediction [Yeh and Patt, 1991; Lee et al., 1998; Pavlov et al., 2009; Li et al., 2010]. Yet because it is a feature engineering technique and not a modeling or optimization method, there is no research paper on the topic. The most detailed description of the technique can be found in Misha Bilenko's blog post "Big Learning Made Easy with Counts" and the associated slides.

The idea of bin-counting is deviously simple: rather than using the *value* of the categorical variable as the feature, instead use the conditional probability of the target under that value. In other words, instead of encoding the identity of the categorical value, compute the association statistics between that value and the target that we wish to predict. For those familiar with Naïve Bayes classifiers, this statistic should ring a bell, because it is the conditional probability of the class under the assumption that all features are independent. It is best illustrated with an example.

Table 4-5. Example of bin-counting features. (Reproduced from "Big Learning Made Easy with Counts" with permission.)

User	Number of clicks	Number of non-clicks	Probability of click	QueryHash, AdDomain	Number of clicks	Number of non-clicks	Probability of click
Alice	5	120	0.0400	**0x598fd4fe, foo.com**	5,000	30,000	0.167
Bob	20	230	0.0800	**0x50fa3cc0, bar.org**	100	9,00	0.100
...							
Joe	2	3	0.400	**0x437a45e1, qux.net**	6	18	0.250

Bin-counting assumes that historical data is available for computing the statistics. Table 4-5 contains aggregated historical counts for each possible value of the categorical variables. Based on the number of times the user has clicked on any ad and the number of times she has not clicked, we can calculate a probability that the user "Alice" would click on any ad. Similarly, we can compute the probability of click for any query-ad-domain combination. At training time, every time we see "Alice," use her probability of click as the input feature to the model. The same goes for QueryHash-AdDomain pairs like “0x437a45e1, qux.net.”

Suppose there are 10,000 users. One-hot encoding would generate a sparse vector of length 10,000, with a single 1 in the column that corresponds to the value of the current data point. Bin-counting would encode all 10,000 binary columns as a single feature with a real value between 0 and 1.

We can include other features in addition to the historical click-through probability: the raw counts themselves (number of clicks and non-clicks), the log-odds ratio, or any other derivatives of probability. Our example here is for predicting ad click-through rates. But the technique readily applies to general binary classification. It can also be readily extended to multi-class classification using the usual techniques to extend binary classifiers to multi-class, i.e., via one-against-many odds ratios or other multi-class label encodings.

Odds Ratio and Log Odds Ratio for Bin-Counting

The odds ratio is usually defined between two binary variables. It looks at their strength of association by asking the question “how much more likely is it for *Y* to be true when *X* is true.″ For instance, we might ask, “how much more likely is Alice to click on an ad than the general population?” Here, X is the binary variable “is Alice the current user,” and Y the variable “click on ad or not.″ The computation uses what’s called the two-way contingency table (basically, four numbers that correspond to the four possible combinations of *X* and *Y*).

Table 4-6. Contingency table for ad click and user.

	Click	**Non-Click**	**Total**
Alice	5	120	125
Not Alice	995	18,880	19,875
Total	1,000	19,000	20,000

Given an input variable *X* and a target variable *Y*, the odds ratio is defined as

odds ratio = $(P(Y = 1 \mid X = 1) / P(Y = 0 \mid X = 1)) / (P(Y = 1 \mid X = 0)\ P(Y = 0 \mid X = 0))$

In our example, this translates as the ratio between “how much more likely does Alice click on an ad rather than not click” and “how much more likely do other people click rather than not click.” The number, in this case, is

odds ratio (user, ad click) = $((5 / 125) / (120 / 125)) / ((995 / 19{,}875) / (18{,}880 / 19{,}875)) = 0.7906$

More simply, we can just look at the numerator, which examines how much more likely does a single user (Alice) click on an ad vs. not click. This is suitable for large categorical variables with many values, not just two.

odds ratio (Alice, ad click) = (5 / 125) / (120 / 125) = 0.04166

Probabilities ratios could easily become very small or very large. (For instance, there will be users who almost never click on ads, and perhaps users who click on ads much more frequently than not.) The log transform again comes to our rescue. Another useful property of the logarithm is that it turns a division into a subtraction.

log-odds ratio (Alice, ad click) = log (5/125) - log (120/125) = -3.178

In short, bin-counting converts a categorical variable into statistics about the value. It turns a large, sparse, binary representation of the categorical variable into a very small, dense, real-valued numeric representation.

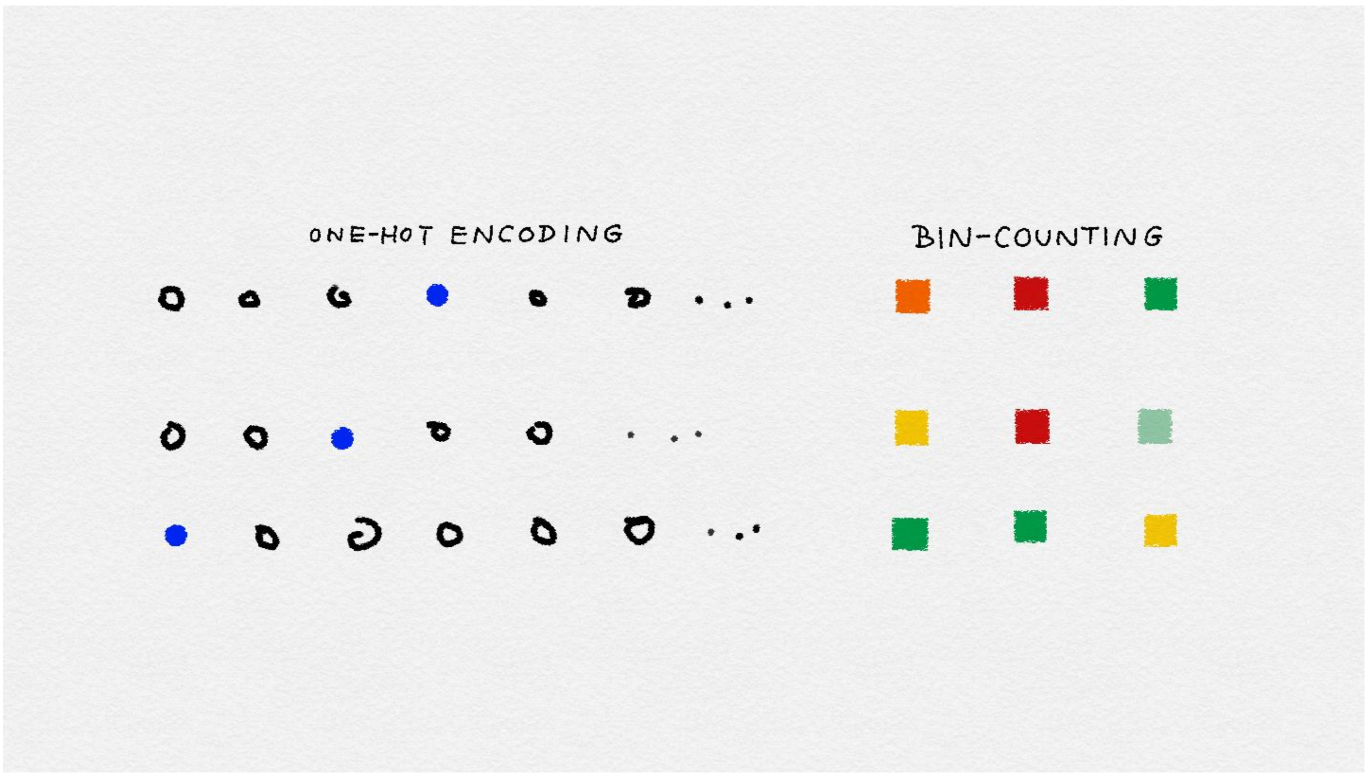


Figure 4-6. An illustration of one-hot encoding vs. bin-counting statistics for categorical variables.

In terms of implementation, bin-counting requires storing a map between each category and its associated counts. (The rest of the statistics can be derived on the fly from the raw counts.) Hence it requires O(k) space, where k is the number of unique values of the categorical variable.

What about rare categories?

Just like rare words, rare categories require special treatment. Think about a user that logs in once a year: there will be very little data to reliably estimate her click-through rate of ads. Moreover, rare categories waste space in the counts table.

One way to deal with this is through *back-off*, a simple technique that accumulates the counts of all rare categories in a special bin. If the count is greater than a certain threshold, then the category gets its own count statistics. Otherwise, use the statistics from the back-off bin. This essentially reverts the statistics for a single rare category to the statistics computed on all rare categories. When using the back-off method, it helps to also add a binary indicator for whether or not the statistics come from the back-off bin.

[back-off illustration]

There is another way to deal with this problem called *count-min sketch* [Cormode and Muthukrishnan, 2005]. In this method, all the categories, rare or frequent alike, are mapped through multiple hash functions with an output range, *m*, much smaller than the number of categories, *k*. When retrieving a statistic, compute all the hashes of the category, and return the smallest statistic. Having multiple hash functions mitigates the probability of collision within a single hash function. The scheme works because the number of hash functions times m, the size of the hash table, can be made smaller than k, the number of categories, and still retain low overall collision probability.

[count-min sketch illustration]

Guarding against data leakage (Bin-counting: days of future past)

Since bin-counting relies on historical data to generate the necessary statistics, it requires waiting through a data collection period, incurring a slight delay in the learning pipeline. It also means that when the data distribution changes, the counts need to be updated. The faster the data changes, the more frequently the counts need to be re-computed. This is particularly important for applications like targeted advertising, where user preferences and popular queries change very quickly, and lack of adaptation to the current distribution could mean huge losses for the advertising platform.

One might ask, why not use the same dataset to compute the relevant statistics and train the model? The idea seems innocent enough. The big problem here is that the statistics involve the target variable, which is what the model tries to predict. Using the output to compute the input features leads to a pernicious problem known as *leakage*. In short, leakage means that information is revealed to the model that gives it an unrealistic advantage to make better predictions. This could happen when test data is leaked into the training set, or when data from the future is leaked to the past. Any time that the model is given information that it shouldn't have access to when it is making predictions in real-time in production, there is leakage. Kaggle's wiki gives more examples of leakage and why it is bad for machine learning applications.

If the bin-counting procedure used the current data point's label to compute part of the input statistic, that constitutes direct leakage. One way to prevent that is by instituting strict separation between count collection (for computing bin-count statistics) and training, i.e., use an earlier batch of data points for counting, use the current data points for training (mapping categorical variables to historical statistics we just collected), and use future data points for testing. This fixes the problem of leakage, but introduces the aforementioned delay (the input statistics and therefore the model will trail behind current data).

[illustration of count/train/test time periods]

It turns out that there is another solution based on differential privacy. A statistic is *approximately leakage proof* if its distribution stays roughly the same with or without any one data point. In practice, adding a small random noise with distribution Laplace(0,1) is sufficient to cover up any potential leakage from a single data point. This idea can be combined with leaving-one-out counting to formulate statistics on current data. Owen Zhang details this trick in his talk on "Winning Data Science Competitions."

Counts without bounds

If the statistics are updated continuously given more and more historical data, the raw counts will grow without bounds. This could be a problem for the model. A trained model "knows" the input data up to the observed scale. A trained decision tree might say "when x is greater than 3, predict 1." A trained linear model might say "multiple x by 0.7 and see if the result is greater than the global average." These might be the correct decisions when x lies between 0 and 5. But what happens beyond that? No one knows.

When the input counts increase, the model will need to be retrained to adapted to the current scale. If the counts accumulate rather slowly, then the effective scale won't change too fast, and the model will not need to be retrained too frequently. But when counts increment very quickly, frequent retraining will be a nuisance.

For this reason, it is often better to use normalized counts that are guaranteed to be bounded in an known interval. For instance, the estimated click-through probability is bounded between [0, 1]. Another method is to take the log transform, which impose a strict bound, but the rate of increase will be very slow when the count is very large.

Neither method will guard against shifting input distributions, e.g., last year's Barbie dolls are now out of style and people will no longer click on those ads. The model will need to be retrained to accommodate these more fundamental changes in input data distribution, or the whole pipeline will need to move to an online learning setting where the model is continuously adapting to the input.

Pros and cons of large categorical variable encodings

Each of the approaches detailed in section has its pros and cons. Here is a run down of the trade-offs therein.

Plain one-hot encoding

Space requirement: O(n) using the sparse vector format, where n is the number of data points

Computation requirement: O(nk) under a linear model, where k is the number of categories

Pros

- Easiest to implement
- Potentially most accurate
- Feasible for online learning

Cons

- Computationally inefficient
- Does not adapt to growing categories
- Not feasible for anything other than linear models
- Requires large-scale distributed optimization with truly large datasets

Feature hashing

Space requirement: O(n) using the sparse matrix format, where n is the number of data points

Computation requirement: O(nm) under a linear or kernel model, where m is the number of hash bins

Pros

- Easy to implement
- Makes model training cheaper
- Easily adaptable to new categories
- Easily handles rare categories
- Feasible for online learning

Cons

- Only suitable for linear or kernelized models

- Hashed features not interpretable
- Mixed reports of accuracy

Bin-counting

Space requirement: O(n + k) for small, dense representation of each data point, plus the count statistics that must be kept for each category

Computation requirement: O(n) for linear models, also usable for non-linear models such as trees

Pros

- Smallest computational burden at training time
- Enables tree-based models
- Relatively easy to adapt to new categories
- Handles rare categories with back-off or count-min sketch
- Interpretable

Cons

- Requires historical data
- Delayed updates required, not completely suitable for online learning
- Higher potential for leakage

As we can see, none of the methods are perfect. The selection of which one to use depends on the desired model. Linear models are cheaper to train and therefore can handle non-compressed representations such as one-hot encoding. Tree-based models, on the other hand, need to repeated search over all features for the right split, and are thus limited to small representations such as bin-counting. Feature hashing sits in between those two extremes, but with mixed reports on the resulting accuracy.

Bibliography

Agarwal, Alekh, Oliveier Chapelle, Miroslav Dudík, and John Langford. 2014. “A Reliable Effective Terascale Linear Learning System.” *Journal of Machine Learning Research (JMLR)*, 15(Mar):1111–1133.

Bertin-Mahieux, Thierry, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. 2011. “The Million Song Dataset.” *Proceedings of the 12th International Society for Music Information Retrieval Conference (ISMIR 2011)*.

Bilenko, Misha. 2015. “Big Learning Made Easy—with Counts!” Cortana Intelligence and Machine Learning Blog, February 17.

https://blogs.technet.microsoft.com/machinelearning/2015/02/17/big-learning-made-easy-with-counts/.

Chen, Ye, Dmitry Pavlov, and John F. Canny. 2009. “Large-scale behavioral targeting.” In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining* (KDD ’09). ACM, New York, NY, USA, 209-218. DOI=http://dx.doi.org/10.1145/1557019.1557048

Cormode, Graham, and S. Muthukrishnan. 2005. “An Improved Data Stream Summary: The Count-Min Sketch and its Applications.” *Algorithms*, 55: 29 - 38.

Fanaee-T, Hadi, and Joao Gama. 2013. “Event labeling combining ensemble detectors and background knowledge.” *Progress in Artificial Intelligence* : 1-15.

Graepel, Thore, Joaquin Quiñonero Candela, Thomas Borchert, and Ralf Herbrich. 2010. “Web-Scale Bayesian Click-Through Rate Prediction for Sponsored Search Advertising in Microsoft’s Bing Search Engine.” *Proceedings of the 27th International Conference on Machine Learning (ICML 2010).*

He, Xinran, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, and Joaquin Quinonero Candela. 2014. “Practical Lessons from Predicting Clicks on Ads at Facebook.” *International Workshop on Data Mining for Online Advertising (ADKDD).*

Introduction to SAS. UCLA: Statistical Consulting Group. from http://www.ats.ucla.edu/stat/sas/notes2/ (accessed November 24, 2007).

Lee, Wenke, Salvatore J. Stolfo, and Kui W. Mok. 1998. “Mining Audit Data to Build Intrusion Detection Models.” In *Proceedings of the 4th ACM SIGKDD international conference on Knowledge discovery and data mining* (KDD 1998), pp. 66-72. 1998.

Li, Wei, Xuerui Wang, Ruofei Zhang, Ying Cui, Jianchang Mao, and Rong Jin. 2010. “Exploitation and exploration in a performance based contextual advertising system.” In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining* (KDD ’10). ACM, New York, NY, USA, 27-36. DOI=http://dx.doi.org/10.1145/1835804.1835811

McMahan, H. Brendan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, Jeremy Kubica. 2013. “Ad Click Prediction: a View from the Trenches.” *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2013).*

Million Song Dataset, official website by Thierry Bertin-Mahieux, available at: http://labrosa.ee.columbia.edu/millionsong/.

Weinberger, Kilian, Anirban Dasgupta, Josh Attenberg, John Langford, Alex Smola. 2009. “Feature Hashing for Large Scale Multitask Learning.” *Proceedings of the 26th International Conference on Machine Learning (ICML 2009)*.

Yeh, Tse-Yu, and Yale N. Patt. 1991. “Two-level adaptive training branch prediction.” In *Proceedings of the 24th annual international symposium on Microarchitecture* (MICRO 24). ACM, New York, NY, USA, 51-61. DOI=http://dx.doi.org/10.1145/123465.123475

[1] In standard statistics literature, the technical term for the categories is *levels*. A categorical variable with two distinct categories has two levels. But there are a number of other things in statistics that are also called levels. So we do not use that terminology here, but instead use the more colloquial and unambiguous term “categories.”

[2] Curious readers might wonder why one is called coding and the other encoding. This is largely convention. My guess is that one-hot encoding first became popular in electrical engineering, where information is encoded and decoded all the time. Dummy coding and effect coding, on the other hand, were invented in the statistics community. Somehow the “en” didn’t make its way over the academic divide.

Chapter 5. Dimensionality Reduction: Squashing the Data Pancake with PCA

With automatic data collection and feature generation techniques, one can quickly obtain a large number of features. But not all of them are useful. In Chapter 2 and Chapter 3, we discussed frequency-based filtering and feature scaling as ways of pruning away uninformative features. Now we will take a close look at the topic of feature dimensionality reduction using Principal Component Analysis (PCA).

This chapter marks an entry into model-based feature engineering techniques. Prior to this point, most of the techniques can be defined without referencing the data. For instance, frequency-based filtering might say, "Get rid of all counts that are smaller than n," a procedure that can be carried out without further input from the data itself. Model-based techniques, on the other hand, require information from the data. For example, PCA is defined around the principal axes of the data. There was never a clear-cut line between data, features, and models. From this point forward, the difference gets increasingly blurry. This is exactly where the excitement lies in current research on feature learning.

Intuition

Dimensionality reduction is about getting rid of "uninformative information" while retaining the crucial bits. There are many ways to define "uninformative." PCA focuses on the notion of linear dependency. In "The Anatomy of a Matrix", we describe the column space of a data matrix as the span of all feature vectors. If the column space is small compared to the total number of features, then most of the features are linear combinations of a few key features. If the next step in the pipeline is a linear model, then linearly depend features are a waste of space and computation power. To avoid this situation, principal component analysis tries to reduce such "fluff" by squashing the data into a much lower dimensional linear subspace.

Picture the set of data points in feature space. Each data point is a dot, and the whole set of data points form a blob. In Figure 5-1(a), the data points spread out evenly across both feature dimensions, and the blob fills the space. In this example, the column space has full rank. However, if some of those features are linear combinations of others, then the blob wouldn't look so plump; it would look more like Figure 5-1(b), a flat blob where feature 1 is a duplicate (or a scalar multiple) of feature 2. In this case, we say that the *intrinsic dimensionality* of the blob is 1, even though it lies in a two-dimensional feature space.

In practice, things are rarely exactly equal to one another. It is more likely that we see features that are very close to being equal, but not quite. In such a case, the data blob might look something like in Figure 5-1(c). It's an emaciated blob. If we want to reduce the number of features to pass to the model, then we could replace feature 1 and feature 2 with a new feature, maybe call it feature 1.5, which lies on the diagonal line between the original two features. The original dataset can then be adequately represented by one number—the position along the direction of feature 1.5—instead of two numbers *f1* and *f2*.

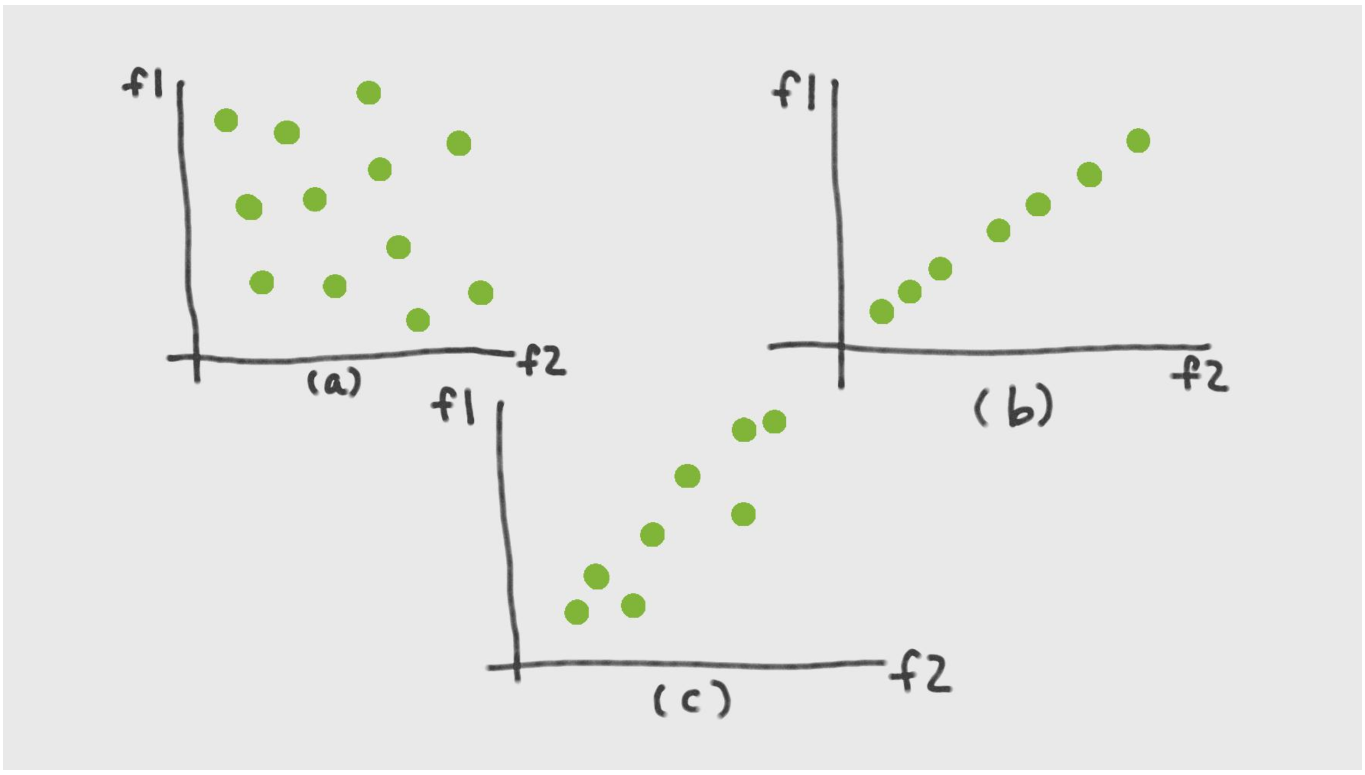


Figure 5-1. Data blobs in feature space. (a) Full-rank data blob. (b) Low dimensional data blob. (c) Approximately low dimensional data blob.

The key idea here is to **replace redundant features with a few new features that adequately summarize information contained in the original feature space.** It's easy to tell what the new feature should be when there are only 2 features. It's much harder when the original feature space has hundreds or thousands of dimensions. We need a way to mathematically describe the new features we are looking for. Then we can use optimization techniques to find them.

One way to mathematically define "adequately summarize information" is to say that the new data blob should retain as much of the original volume as possible. We are squashing the data blob into flat pancakes, but we want the pancake to be as big as possible in the right directions. This means we need a way to measure volume.

Volume has to do with distance. But the notion of distance in a blob of data points is somewhat fuzzy. One could measure the maximum distance between any two pairs of points. But that turns out to be a very difficult function to mathematically optimize. An alternative is to measure the average distance between pairs of points, or equivalently, the average distance between each point and their mean, which is the variance. This turns out to be much easier to optimize. (Life is hard. Statisticians have learned to take convenient short cuts.) Mathematically, this translates into maximizing the variance of the data points in the new feature space.

Derivation

Tips and notations

As before, let X denote the nxd data matrix, where n is the number of data points and d the number of features. Let $\mathbf{x}$ be a column vector containing a single data point. (So $\mathbf{x}$ is the transpose of one of the rows in X.) Let $\mathbf{w}$ denote one of the new feature vectors, or principal components, that we are trying to find.

Singular Value Decomposition (SVD) of a Matrix

Any rectangular matrix can be decomposed into three matrices of particular shapes and characteristics:

$X = U\Sigma V^T$.

Here, U and V are orthogonal matrices (i.e., $U^TU = I$ and $V^TV = I$). Σ is a diagonal matrix containing the singular values of X, which can be positive, zero, or negative. Suppose X has n rows and d columns and $n \geqslant d$ then U has shape nxd, and Σ and V have shape dxd. (See "Singular value decomposition (SVD)" for a full review of SVD and eigen decomposition of a matrix.)

Useful Sum-of-Squares Identity

$$||\mathbf{w}||_2^2 = \mathbf{w}^T\mathbf{w} = \sum_i w_i^2$$

Tips for Navigating Linear Algebra Formulas

To stay oriented in the world of linear algebra, keep track of which quantities are scalars, which are vectors, and which way are the vectors oriented—vertically or horizontally. Know the dimensions of your matrices, because they often tell you whether the vectors of interest are in the rows or columns. Draw the matrices and vectors as rectangles on a page and make sure the shapes match. Just like one can get far in algebra by noting the units of measurement (distance is in miles, speed is in miles per hour), in linear algebra all one needs are the dimensions.

Linear projection

Let' s break down the derivation of PCA step by step. PCA uses linear projection to transform data into the new feature space. Figure 5-2(c) illustrates what a linear projection looks like. When we project **x** onto **w**, the length of the projection is proportional to the inner product between the two, normalized by the norm of **w** (its inner product with itself). Later on, we will constrain **w** to have unit norm. So the only relevant part is the numerator.

Equation 5-1. Projection coordinate

Equation 5-1. $z = \mathbf{x}^T\mathbf{w}$.

Note that z is a scalar, whereas **x** and **v** are column vectors. Since there are a bunch of data points, we can formulate the vector **z** of all of their projection coordinates on the new feature **v**. Here, X is the familiar data matrix where each row is a data point. The resulting **z** is a column vector.

Equation 5-3. Vector of projection coordinates

Equation 5-3. $\mathbf{z} = X\mathbf{w}$.

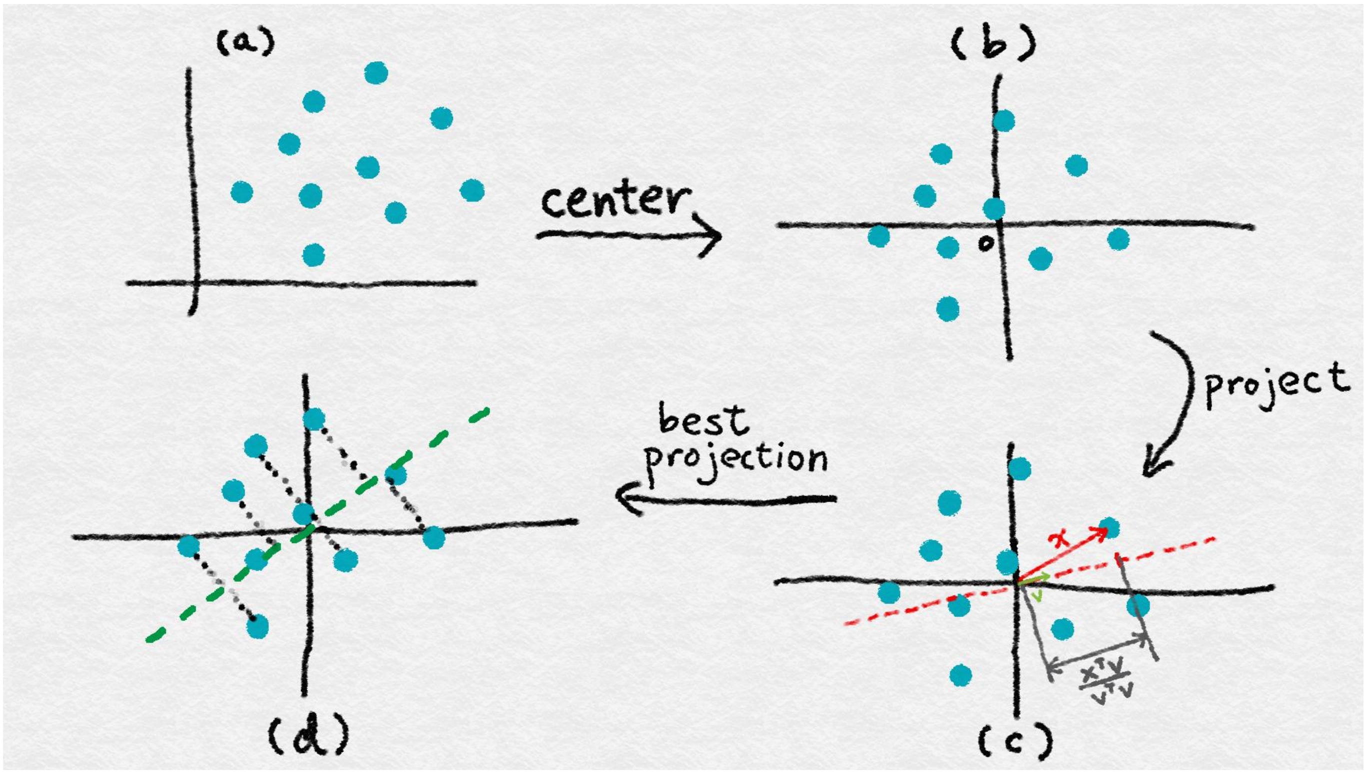


Figure 5-2. Illustration of PCA. (a) Original data in feature space. (b) Centered data. (c) Projecting a data vector x onto another vector v. (d) The direction that maximizes the variance of the projected coordinates is the principal eigenvector of X^TX.

Variance and empirical variance

The next step is to compute the variance of the projections. Variance is defined as the expectation of the squared distance to the mean.

Equation 5-5. Variance of a random variable Z

Equation 5-5. $\text{Var}(Z) = E[Z - E(Z)]^2$.

There is one tiny problem: our formulation of the problem says nothing about the mean, *E(Z)*; it is a free variable. One solution is to remove it from the equation by subtracting the mean from every data point. The resulting data set has mean zero, which means that the variance is simply the expectation of Z^2. Geometrically, subtracting the mean has the effect of centering the data. (See Figure 5-2(a-b).)

A closely related quantity is the covariance between two random variables Z1 and Z2. Think of this as the extension of the idea of variance (of a single random variable) to two random variables.

Equation 5-7. Covariance between two random variables Z_1 and Z_2

Equation 5-7. $\text{Cov}(Z_1, Z_2) = E[(Z_1 - E(Z_1)(Z_2 - E(Z_2)]$.

When the random variables have mean zero, their covariance coincides with their *linear correlation* $E[Z_1Z_2]$. We will hear more about this concept later on.

Statistical quantities like variance and expectation are defined on a data distribution. In practice, we don’t have the true distribution, but only a bunch of observed data points, $z_1, \ldots, z_n$. This is called an *empirical distribution*, and it gives us an empirical estimate of the variance.

Equation 5-9. Empirical variance of *Z* based on observations $z_1, \ldots, z_n$

Equation 5-9. $\mathrm{Var}_{\mathrm{emp}}(Z) = \frac{1}{n-1}\sum_{i=1}^{n} z_i^2.$

Principal components: first formulation

Combined with the definition of z_i in Equation 5-1, we have the following formulation for maximizing the variance of the projected data. (We drop the denominator *n-1* from the definition of empirical variance, because it is a global constant and does not affect where the maximizing value occurs.)

Equation 5-11. Objective function of principal components

Equation 5-11. $\max_{w} \sum_{i=1}^{n} (x_i^T w)^2, \text{ where } w^T w = 1.$

The constraint here forces the inner product of **w** with itself to be 1, which is equivalent to saying that the vector must have unit length. This is because we only care about the direction and not the magnitude of **w**. The magnitude of **w** is an unnecessary degree of freedom, so we get rid of it by setting it to an arbitrary value.

Principal components: matrix-vector formulation

Next comes the tricky step. The sum of squares term in Equation 5-11 is rather cumbersome. It'd be much cleaner in a matrix-vector format. Can we do it? The answer is yes. The key lies in the sum-of-squares identity: the sum of a bunch of squared terms is equal to the squared norm of a vector whose elements are those terms, which is equivalent to the vector's inner product with itself. With this identity in hand, we can rewrite Equation 5-11 in matrix-vector notation.

Equation 5-13. Objective function for principal components, matrix-vector formulation

Equation 5-13. $\max_{w} w^T X^T X w, \text{ where } w^T w = 1.$

This formulation of PCA presents the target more clearly: we look for an input direction that maximizes the norm of the output. Does this sound familiar? The answer lies in the singular value decomposition (SVD) of X. The optimal **w**, as it turns out, is the principal left singular vector of *X*, which is also the principal eigenvector of $X^T X$. The projected data is called a principal component of the original data.

General solution of the principal components

This process can be repeated. Once we find the first principal component, we can rerun Equation 5-13 with the added constraint that the new vector be orthogonal to the previously found vectors.

Equation 5-15. Objective function for k+1st principal components

Equation 5-15. $\max_w w^T X^T X w$, where $w^T w = 1$ and $w^T w_1 = \ldots = w^T w_k = 0$.

The solution is the *k+1*st left singular vectors of X, ordered by descending singular values. Thus, the first k principal components correspond to the first k left singular vectors of X.

Transforming features

Once the principal components are found, we can transform the features using linear projection. Let $X = U\Sigma V^T$ be the SVD of X, and V_k the matrix whose columns contain the first k left singular vectors. X has dimensions nxd, where d is the number of original features, and V_k has dimensions dxk. Instead of a single projection vector as in Equation 5-3, we can simultaneous project onto multiple vectors in a projection matrix.

Equation 5-17. PCA projection matrix

$$W = V_k$$

The matrix of projected coordinates are easy to compute, and can be further simplified using the fact that the singular vectors are orthogonal to each other.

Equation 5-18. Simple PCA transform

$$Z = XW = XV_k = U\Sigma V^T V_k = U_k \Sigma_k$$

The projected values are simply the first k right singular vectors scaled by the first k singular values. Thus, the entire PCA solution, components and projections alike, can be conveniently obtained through the SVD of X.

Implementing PCA

Many derivations of PCA involve first centering the data, then taking the eigen decomposition of the covariance matrix. But the easiest way to implement PCA is by taking the singular value decomposition of the centered data matrix.

PCA Implementation Steps

Equation 5-19. Center the data matrix

$C = X - \mathbf{1}\boldsymbol{\mu}^T$, where $\mathbf{1}$ is a column vector containing all 1's, and $\boldsymbol{\mu}$ is a column vector containing the average of the rows of X.

Equation 5-20. Compute the SVD

$C = U\Sigma V^T$.

Equation 5-21. Principal components

The first k principal components are the first k columns of V, i.e., the right singular vectors corresponding to the k largest singular values.

Equation 5-22. Transformed data

The transformed data is simply the first k columns of U. (If whitening is desired, then scale the vectors by the inverse singular values. This requires that the selected singular values are non-zero. See "Whitening and ZCA".)

PCA in Action

Let's get a better sense for how PCA works by applying it to some image data. The MNIST dataset contains images of handwritten digits from 0 to 9. The original images are 28 x 28 pixels. A lower resolution subset of the images is distributed with scikit-learn, where each image is down sampled into 8 x 8 pixels. The original data in scikit-learn has 64 dimensions. We apply PCA and visualize the dataset using the first three principal components.

Example 5-1. Principal component analysis of the scikit-learn digits dataset (a subset of the MNIST dataset).

```
>>> from sklearn import datasets
>>> from sklearn.decomposition import PCA

# Load the data
>>> digits_data = datasets.load_digits()
>>> n = len(digits_data.images)

# Each image is represented as an 8-by-8 array.
# Flatten this array as input to PCA.
>>> image_data = digits_data.images.reshape((n, -1))
>>> image_data.shape
(1797, 64)

# Groundtruth label of the number appearing in each image
>>> labels = digits_data.target
>>> labels
array([0, 1, 2, ..., 8, 9, 8])

# Fit a PCA transformer to the dataset.
# The number of components is automatically chosen to account for
# at least 80% of the total variance.
>>> pca_transformer = PCA(n_components=0.8)
>>> pca_images = pca_transformer.fit_transform(image_data)
>>> pca_transformer.explained_variance_ratio_
array([ 0.14890594,  0.13618771,  0.11794594,  0.08409979,  0.05782415,
        0.0491691 ,  0.04315987,  0.03661373,  0.03353248,  0.03078806
,
        0.02372341,  0.02272697,  0.01821863])
>>> pca_transformer.explained_variance_ratio_[:3].sum()
0.40303958587675121

# Visualize the results
>>> import matplotlib.pyplot as plt
>>> from mpl_toolkits.mplot3d import Axes3D
>>> %matplotlib notebook
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111, projection='3d')
>>> for i in range(100):
...     ax.scatter(pca_images[i,0], pca_images[i,1], pca_images[i,2],
...                marker=r'${}$'.format(labels[i]), s=64)
```

```
>>> ax.set_xlabel('Principal component 1')
>>> ax.set_ylabel('Principal component 2')
>>> ax.set_zlabel('Principal component 3')
```

A picture of the first 100 projected images are shown in in a 3D plot in Figure 5-3. The markers correspond to the labels. The first three principal components account for roughly 40% of the total variance in the dataset. It is by no means perfect, but it allows for a handy low-dimensional visualization. We see that PCA groups similar numbers close to each other. The numbers 0 and 6 lie in the same region, as do 1 and 7, 3 and 9. The space is roughly divided between 0, 4, and 6 on one side, and the rest of the numbers on the other.

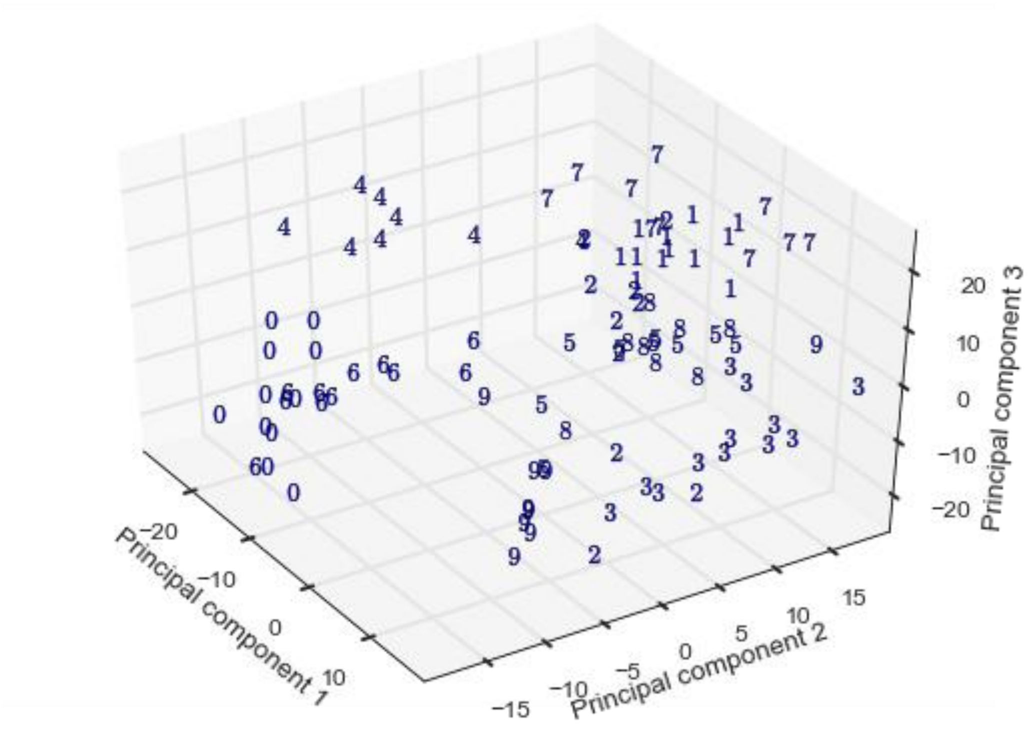


Figure 5-3. PCA projections of subset of MNIST data. Markers correspond to image labels.

Since there is a fair amount of overlap between numbers, it would be difficult to tell them apart using a linear classifier in the projected space. Hence, if the task is to classify the handwritten digits and the chosen model is a linear classifier, then the first three principal components are not sufficient as features. Nevertheless, it is interesting to see how much of a 64-dimensional dataset can be captured in just 3 dimensions.

[Example of PCA or ZCA for image classification or feature extraction]

Whitening and ZCA

Due to the orthogonality constraint in the objective function, PCA transformation produces a nice side effect: the transformed features are no longer correlated. In other words, the inner products between pairs of feature vectors are zero. It’ s easy to prove this using the orthogonality property of the singular vectors: $Z^T Z = \Sigma_k U_k^T U_k \Sigma_k = \Sigma_k^2$. The result is a diagonal matrix containing squares of the singular values representing the correlation of each feature vector with itself, also known as its L^2 norm.

Sometimes, it is useful to also normalize the scale of the features to 1. In signal processing terms, this is known as *whitening*. It results in a set of features that have unit correlation with themselves and zero correlation with each other. Mathematically, whitening can done by multiplying the PCA transformation with the inverse singular values.

Equation 5-23. PCA + whitening

$$W_{white} = V_k \Sigma_k^{-1}$$

$$Z_{white} = X V_k \Sigma_k^{-1} = U \Sigma V^T V_k \Sigma_k^{-1} = U_k$$

Whitening is independent from dimensionality reduction; one can perform one without the other. For example, Zero-phase Component Analysis (ZCA) (Bell and Sejnowski, 1996) is a whitening transformation that is closely related to PCA, but that does not reduce the number of features. ZCA whitening uses the full set of principal components V without reduction, and includes an extra multiplication back onto V^T.

Equation 5-24. ZCA whitening

$$W_{ZCA} = V \Sigma^{-1} V^T$$

$$Z_{zca} = X V \Sigma^{-1} V^T = U \Sigma V^T V \Sigma^{-1} = U$$

Simple PCA projection (Equation 5-18) produces coordinates in the new feature space, where the principal components serve as the basis. These coordinates represent only the length of the projected vector, not the direction. Multiplication with the principal components gives us the length and the orientation. Another valid interpretation is that the extra multiplication rotates the coordinates back into the original feature space. (V is an orthogonal matrix, and orthogonal matrices rotate their input without stretching or compression.) So ZCA produces whitened data that is as close (in Euclidean distance) to the original data as possible.

Considerations and Limitations of PCA

When using PCA for dimensionality reduction, one must address the question of how many principal components (k) to use. Like all hyperparameters, this number can be tuned based on the quality of the resulting model. But there are also heuristics that do not involve expensive computational methods.

One possibility is to pick k to account for a desired proportion of total variance. (This option is available in the scikit-learn package for PCA.) The variance of the projection onto the k-th component is $\| \boldsymbol{X v}_k \|^2 = \| \boldsymbol{u}_k \sigma_k \|^2 = \sigma_k^2$, which is the square of the k-th largest singular value of $\boldsymbol{X}$. The ordered list of singular values of a matrix is called its spectrum. Thus to determine how many components to use, one can perform a simple spectral analysis of the data matrix and pick the threshold that retains enough variance.

Selecting k Based on Accounted Variance

To retain enough components to cover 80% of the total variance in the data, pick k such that $\frac{\Sigma_{i=1}^{k}\sigma_i^2}{\Sigma_{i=1}^{d}\sigma_i^2} \geq .8$.

Another method for picking k involves the intrinsic dimensionality of a data set. This is a more hazy concept, but can also be determined from the spectrum. Basically, if the spectrum contains a few large singular values and a number of tiny ones, then one can probably just harvest the largest singular values and discard the rest. Sometimes the rest of the spectrum is not tiny, but there’s a large gap between the head and the tail values. That would also be a reasonable cutoff. This method is requires visual inspection of the spectrum and hence cannot be performed as part of an automated pipeline.

One key criticism of PCA is that the transformation is fairly complex, and the results are therefore hard to interpret. The principal components and the projected vectors are real valued and could be positive or negative. The principal components are essentially linear combinations of the (centered) rows, and the projection values are linear combinations of the columns. In the stock returns application, for instance, each factor is a linear combination of time slices of stock returns. What does that mean? It is hard to attach a human understandable reason for the learned factors. Therefore, it is hard for analysts to trust the results. If you can’t explain why you are putting billions of other people’s money on particular stocks, you probably won’t choose to use that model.

PCA is computationally expensive. It relies on SVD, which is an expensive procedure. To compute the full SVD of a matrix takes $O(nd^2 + d^3)$ operations [Golub and Van Loan, 2012], assuming $n \geqslant d$, i.e., there are more data points than features. Even if we only want k principal components, computing the truncated SVD (the k largest singular values and vectors) still takes $O((n+d)^2 k) = O(n^2 k)$ operations. This is prohibitive when there are a large number of data points or features.

It is difficult to perform PCA in a streaming fashion, in batch updates, or from a sample of the full data. Streaming computation of the SVD, updating the SVD, and computing the SVD from a subsample are all difficult research problems. Algorithms exist, but at the cost of reduced accuracy. One implication is that one should expect lower representational accuracy when projecting test data onto principal components found on the training set. As the distribution of the data changes, one would have to recompute the principal components on the current dataset.

Lastly, it is best not to apply PCA to raw counts (word counts, music play counts, movie viewing counts, etc.). The reason for this is that such counts often contain large outliers. (The probability is pretty high that there is a fan out there who watched "The Lord of the Rings" 314,582 times, which dwarfs the rest of the counts.) As we know, PCA looks for linear correlations within the features. Correlation and variance statistics are very sensitive to large outliers; a single large number could change the statistics a lot. So, it is a good idea to first trim the data of large values ("Frequency-based filtering"), or apply a scaling transform like tf-idf (Chapter 3), or the log transform ("Log transformation").

Use Cases

PCA reduces feature space dimensionality by looking for linear correlation patterns between features. Since it involves the SVD, PCA is expensive to compute for more than a few thousand features. But for small numbers of real-valued features, it is very much worth trying.

PCA transformation discards information from the data. Thus, downstream model may then be cheaper to train, but possibly less accurate. On the MNIST dataset, some have observed that using reduced-dimensionality data from PCA results in less accurate classification models. In these cases, there is both an upside and a downside to using PCA.

One of the coolest applications of PCA is in anomaly detection of time series. Lakhina, Crovella, and Diot [2004] used PCA to detect and diagnose anomalies in internet traffic. They focused on volume anomalies, i.e., when there is a surge or a dip in the amount of traffic going from one network region to another. These sudden changes may be indicative of a misconfigured network or coordinated Denial of Service attacks. Either way, knowing when and where such changes occur is valuable to internet operators.

Since there is so much total traffic over the internet, isolated surges in small regions are hard to detect. A relatively small set of backbone links handle much of the traffic. Their key insight is that volume anomalies affect multiple links at the same time (because network packets need to hop through multiple nodes to reach their destination). Treat each of the links as a feature, and the amount of traffic at each time step as the measurement.

A data point is a time slice of traffic measurements across all links on the network. The principal components of this matrix indicate the overall traffic trends on the network. The rest of the components represent the residual signal, which contains the anomalies.

PCA is also often used in financial modeling. In those use cases, it works as a type of *factor analysis*, a family of statistical methods that aim to describe observed variability in data using a small number of unobserved *factors*. In factor analysis applications, the goal is to find the explanatory components, not the transformed data.

Financial quantities like stock returns are often correlated with each other. Stocks may move up and down at the same time (positive correlation), or move in opposite directions (negative correlation). In order to balance volatility and reduce risk, an investment portfolio needs a diverse set of stocks that are not correlated with each other. (Don’t put all your eggs in one basket if that basket is going to sink.) Finding strong correlation patterns is helpful for deciding on an investment strategy.

Stock correlation patterns can be industry wide. For example, tech stocks may go up and down together, while airline stocks tend to go down when oil prices are high. But industry may not be the best way to explain the outcome. Analysts also look for unexpected correlations in observed statistics. In particular, s*tatistical factor model* [Connor, 1995] runs PCA on the matrix of time series of individual stock returns to find commonly co-varying stocks. In this use case, the end goal are the principal components themselves, not the transformed data.

ZCA is useful as a pre-processing step when learning from images. In natural images, adjacent pixels often have similar colors. ZCA whitening can remove this correlation, which allows subsequent modeling efforts to focus on more interesting image structures. Alex Krizhevsky’s thesis on "Learning Multiple Layers of Features from Images" contains nice examples that illustrate the effect of ZCA whitening on natural images.

Many deep learning models use PCA or ZCA as a pre-processing step, though it is not always shown to be necessary. In "Factored 3-Way Restricted Boltzmann Machines for Modeling Natural Images," Ranzato et al. remark, “Whitening is not necessary but speeds up the convergence of the algorithm." In "An Analysis of Single-Layer Networks in Unsupervised Feature Learning," Coates et al. find that ZCA whitening is helpful for some models, but not all. (Note the models in this paper are unsupervised feature learning models. So ZCA is used as a feature engineering method for other feature engineering methods. Stacking and chaining of methods is commonplace in machine learning pipelines.)

Summary

This concludes the discussion of PCA. The two main things to remember about PCA are its mechanism (linear projection) and objective (to maximize the variance of projected data). The solution involves the eigen decomposition of the covariance matrix, which is closely related to the SVD of the data matrix. One can also remember PCA with the mental picture of squashing the data into a pancake that is as fluffy as possible.

PCA is an example of model-driven feature engineering. (One should immediately suspect that a model is lurking in the background whenenever an objective function enters the scene.) The modeling assumption here is that variance adequately represents the information contained in the data. Equivalently, the model looks for linear correlations between features. This is used in several applications to reduce the correlation or find common factors in the input.

PCA is a well-known dimensionality reduction method. But it has its limitations, such as high computational cost and uninterpretable outcome. It is useful as a pre-processing step, especially when there are linear correlations between features.

When seen as a method for eliminating linear correlation, PCA is related to the concept of whitening. Its cousin, ZCA, whitens the data in an interpretable way, but does not reduce dimensionality.

Bibliography

Bell, Anthony J. and Terrence J. Sejnowski. 1996. "Edges are the 'Independent Components' of Natural Scenes." *Proceedings of the Conference on Neural Information Processing Systems (NIPS).*

Coates, Adam, Andrew Y. Ng, and Honglak Lee. 2011. "An Analysis of Single-Layer Networks in Unsupervised Feature Learning." *International conference on artificial intelligence and statistics*.

Connor, Gregory. 1995. "The Three Types of Factor Models: A Comparison of Their Explanatory Power." *Financial Analysts Journal* 51, no. 3: 42-46. http://www.jstor.org/stable/4479845.

Golub, Gene H., and Charles F. Van Loan. 2012. *Matrix Computations*. Baltimore and London: Johns Hopkins University Press; fourth edition.

Krizhevsky, Alex. 2009. "Learning Multiple Layers of Features from Tiny Images." MSc thesis, University of Toronto.

Lakhina, Anukool, Mark Crovella, and Christophe Diot. 2004. "Diagnosing network-wide traffic anomalies." *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications* (SIGCOMM '04). DOI=http://dx.doi.org/10.1145/1015467.1015492

Ranzato, Marc'Aurelio, Alex Krizhevsky, and Geoffrey E. Hinton. 2010. "Factored 3-Way Restricted Boltzmann Machines for Modeling Natural Images." *Proceedings of the 13-th International Conference on Artificial Intelligence and Statistics (AISTATS 2010).*

Chapter 6. Non-Linear Featurization and Model Stacking

PCA is very useful when the data lies in a linear subspace like a flat pancake. But what if the data forms a more complicated shape?[1] A flat plane (linear subspace) can be generalized to a *manifold* (non-linear subspace), which can be thought of as a surface that gets stretched and rolled in various ways.[2]

If a linear subspace is a flat sheet of paper, then a rolled up sheet of paper is a simple example of a non-linear manifold. Informally, this is called a Swiss roll. (See Figure 6-1.) Once rolled, a 2-D plane occupies 3-D space. Yet it is essentially still a 2-D object. In other words, it has low intrinsic dimensionality, a concept we' ve already touched upon in "Intuition". If we could somehow unroll the Swiss roll, we' d recover the 2-D plane. This is the goal of *non-linear dimensionality reduction*, which assumes that the manifold is simpler than the full dimension it occupies and attempt to unroll it.

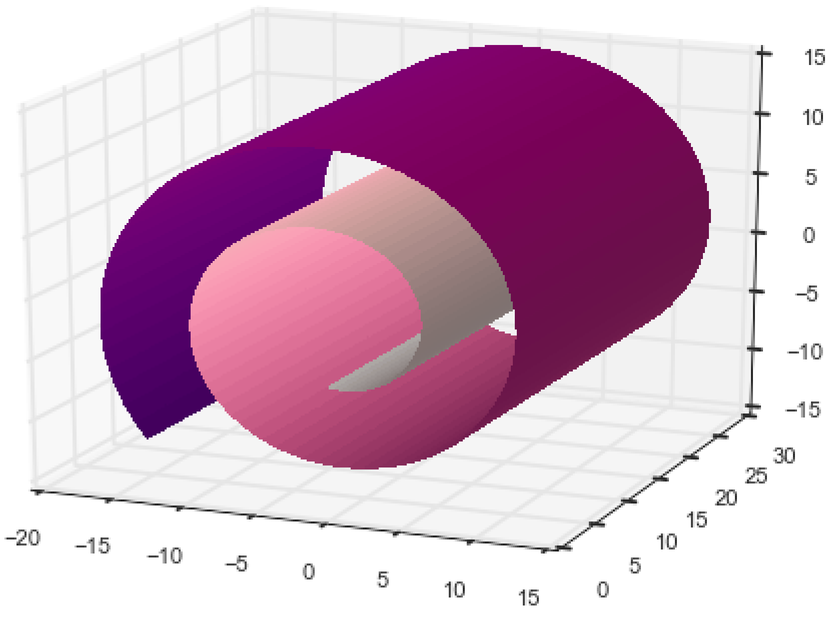


Figure 6-1. The Swiss roll, a non-linear manifold.

The key observation is that even when the big manifold looks complicated, the local neighborhood around each point can often be well approximated with a patch of flat surface.

In other words, they learn to encode global structure using local structure.[3] Non-linear dimensionality reduction is also called *non-linear embedding* or *manifold learning*. Non-linear embeddings are useful for aggressively compressing high-dimensional data into low dimensional data. They is often used for visualization in 2-D or 3-D.

The goal of feature engineering, however, isn't so much to make the feature dimensions as low as possible, but to arrive at the *right* features for the task. In this chapter, the right features are those that represent the spatial characteristics of the data.

Clustering algorithms are usually not presented as techniques for local structure learning. But they in fact do just that. Points that are close to each other (where "closeness" can defined by the data scientist using some metric) belong to the same cluster. Given a clustering, a data point can be represented by its cluster membership vector. If the number of clusters is smaller than the original number of features, then the new representation will have fewer dimensions than the original; the original data is compressed into a lower dimension.

Compared to non-linear embedding techniques, clustering may produce more features. But if the end goal is feature engineering instead of visualization, this is not a problem.

We will illustrate the idea of local structure learning with a common clustering algorithm called k-means. It is simple to understand and implement. Instead of non-linear manifold reduction, it is more apt to say that k-means performs *non-linear manifold feature extraction*. Used correctly, it can be a powerful tool in our feature engineering repertoire.

K-means Clustering

K-means is a clustering algorithm. Clustering algorithms group data depending on how they are laid out in space. They are *unsupervised* in that they do not require any sort of label—it's the algorithm's job to infer cluster labels based solely on the geometry of the data itself.

A clustering algorithm depend on a *metric*—a measurement of closeness between data points. The most popular metric is the Euclidean distance or Euclidean metric. It comes from Euclidean geometry and measures the straightline distance between two points. It should feel very normal to us because this is the distance we see in everyday physical reality.

The Euclidean distance between two vectors x and y is the L^2 norm of $x - y$. (See "L^2 normalization" on the L^2 norm.) In math speak, it is usually written as $\| x - y \|_2$ or just $\| x - y \|$.

K-means establishes a hard clustering, meaning that each data point is assigned to one and only one cluster. The algorithm learns to position the cluster centers such that the total sum of the Euclidean distance between each data point and its cluster center is minimized. For those who like to read math instead of words, here is the objective function:

Equation 6-1. K-means objective

$$\min_{C_1,\ldots,C_k,\mu_1,\ldots,\mu_k} \sum_{i=1}^{k} \sum_{x \in C_i} | | x - \mu_i | |_2$$

Each cluster C_i contains a subset of data points. The center of cluster i is equal to the average of all the data points in the cluster: $\mu_i = \Sigma_{x \in Ci}\, x / n_i$, where n_i denotes the number of data points in cluster i.

Figure 6-2 shows k-means at work on two different randomly generated datasets. The data in (a) is generated from random Gaussian distributions with the same variance but different means. The data in (c) is generated uniformly at random. These toy problems are very simple to solve, and k-means does a good job. (The results could be sensitive to the number of clusters, which must be given to the algorithm.) The code for this example may be found in Example 6-1.

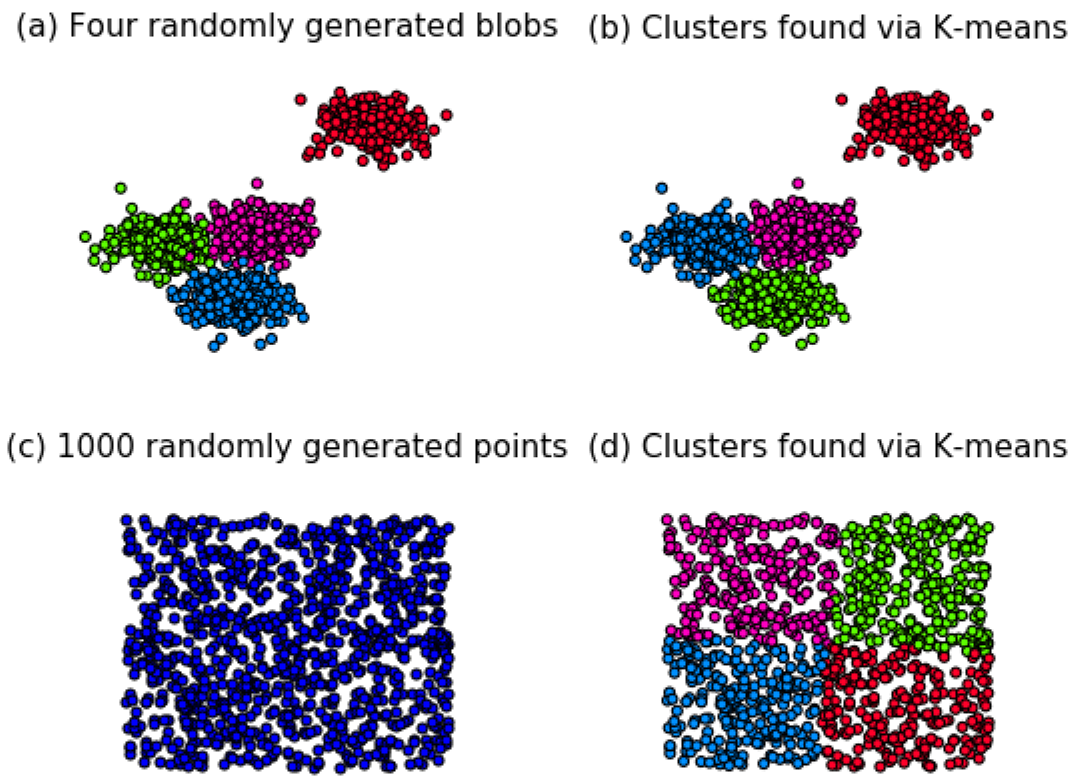


Figure 6-2. K-means examples demonstrating how the clustering algorithm partitions space.

Example 6-1. K-means examples.

```
import numpy as np
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

import matplotlib.pyplot as plt
%matplotlib notebook

n_data = 1000
seed = 1
n_clusters = 4

# Generate random Gaussian blobs and run K-means
blobs, blob_labels = make_blobs(n_samples=n_data, n_features=2,
                                                centers=n_centers, random_state=seed)
clusters_blob = KMeans(n_clusters=n_centers, random_state=seed).fit_predict(blobs)

# Generate data uniformly at random and run K-means
uniform = np.random.rand(n_data, 2)
clusters_uniform = KMeans(n_clusters=n_clusters,
random_state=seed).fit_predict(uniform)

# Matplotlib incantations for visualizing results
figure = plt.figure()
plt.subplot(221)
plt.scatter(blobs[:, 0], blobs[:, 1], c=blob_labels, cmap='gist_rainbow')
plt.title("(a) Four randomly generated blobs", fontsize=14)
plt.axis('off')

plt.subplot(222)
plt.scatter(blobs[:, 0], blobs[:, 1], c=clusters_blob, cmap='gist_rainbow')
plt.title("(b) Clusters found via K-means", fontsize=14)
plt.axis('off')

plt.subplot(223)
plt.scatter(uniform[:, 0], uniform[:, 1])
plt.title("(c) 1000 randomly generated points", fontsize=14)
plt.axis('off')

plt.subplot(224)
plt.scatter(uniform[:, 0], uniform[:, 1], c=clusters_uniform, cmap='gist_rainbow')
plt.title("(d) Clusters found via K-means", fontsize=14)
plt.axis('off')
```

Clustering as surface tiling

Common applications of clustering assume that there are natural clusters to be found, i.e., that there are regions of dense data scattered in an otherwise empty space. In these situations, there is a notion of the correct number of clusters, and people have invented clustering indices that measure the quality of data groupings in order to select for *k*.

However, when data is spread out fairly uniformly like in Figure 6-2(c), there is no longer a correct number of clusters. In this case, the role of a clustering algorithm is *vector quantization*, i.e., partition the data into a finite number of chunks. The number of clusters can be selected based on acceptable approximation error when using quantized vectors instead of the original ones.

Visually, this usage of k-means can be thought of as covering the data surface with patches, like in Figure 6-3. This is indeed what we get if run k-means on a Swiss roll dataset. Example 6-2 uses scikit-learn to generate a noisy dataset on the Swiss roll, cluster it with k-means, and visualize the clustering results using Matplotlib. The data points are colored according to their cluster ids.

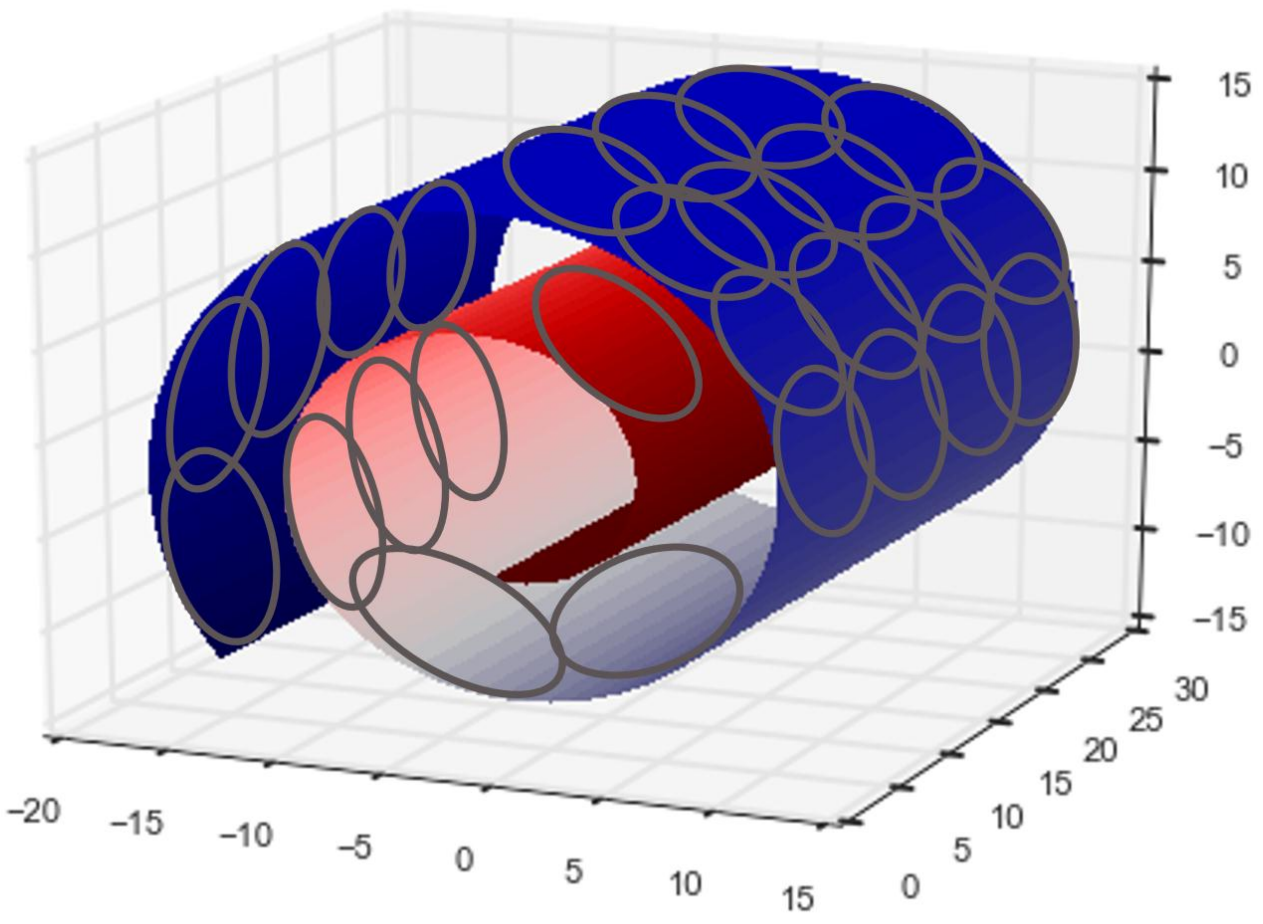


Figure 6-3. Conceptual local patches on the Swiss role from a clustering algorithm.

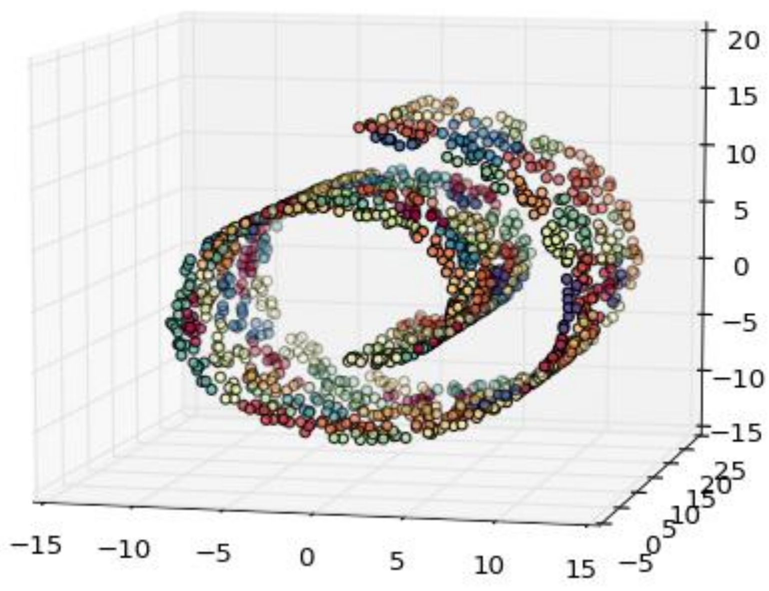


Figure 6-4. Approximating a Swiss roll dataset using k-means with 100 clusters.

Example 6-2. K-means on the Swiss roll.

```
from mpl_toolkits.mplot3d import Axes3D
from sklearn import manifold, datasets

# Generate a noisy Swiss roll dataset
X, color = datasets.samples_generator.make_swiss_roll(n_samples=1500)

# Approximate the data with 100 K-means clusters
clusters_swiss_roll = KMeans(n_clusters=100, random_state=1).fit_predict(X)

# Plot the dataset with K-means cluster id's as the color
fig2 = plt.figure()
ax = fig2.add_subplot(111, projection='3d')
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=clusters_swiss_roll, cmap='Spectral')
```

In this example, we generated 1,500 points at random on the Swiss roll surface, and asked k-means to approximate it with 100 clusters. We pulled the number 100 out of a hat because it seems like a fairly large number to cover fairly small space. The result looks nice; the clusters are indeed very local and different sections of the manifold are mapped to different clusters. Great! Are we done?

The problem is that if we pick a k that is too small, then the results won't be so nice from a manifold learning perspective. Figure 6-5 shows the output of k-means on the Swiss roll with 10 clusters. We can clearly see data from very different sections of the manifold being mapped to the same clusters (e.g., the yellow, purple, green, and magenta clusters).

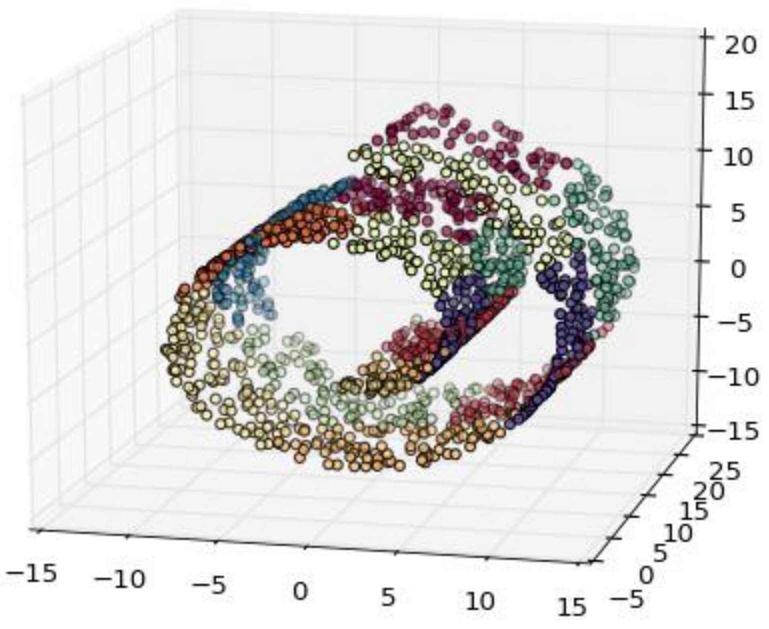


Figure 6-5. K-means on the Swiss roll with 10 clusters.

If the data is distributed uniformly throughout the space, then picking the right *k* boils down to a sphere packing problem. In *d*-dimensions, one could fit roughly $1/r^d$ spheres of radius *r*. Each k-means cluster is a sphere, and the radius is the maximum error of representing points in that sphere with the centroid. So, if we are willing to tolerate a maximum approximation error of *r* per data point, then the number of clusters is $O(1/r^d)$, where *d* is the dimension of the original feature space of the data.

Uniform distribution is the worst-case scenario for k-means. If data density is not uniform, then we will be able to represent more data with fewer clusters. In general, it is difficult to tell how data is distributed in high dimensional space. One can be conservative and pick a larger *k*. But it can't be too large, because *k* will become the number of features for the next modeling step.

K-means featurization for classification

When using k-means as a featurization procedure, a data point can be represented by its cluster membership (a sparse one-hot encoding of the cluster membership categorical variable), which we now illustrate.

If a target variable is also available, then we have the choice of giving that information as a hint to the clustering procedure. One way to incorporate target information is to simply include the target variable as an additional input feature to the k-means algorithm. Since the objective is to minimize the total Euclidean distance over all input dimensions, the clustering procedure will attempt to balance similarity in the target value as well as in the original feature space. The target values can be scaled to get more or less attention from the clustering algorithm. Larger differences in the target will produce clusters that pay more attention to the classification boundary.

K-means Featurization

Clustering algorithms analyze the spatial distribution of data. Therefore, k-means featurization creates a compressed spatial index of the data which can be fed into the model in the next stage. This is an example of *model stacking*.

Example 6-3 shows a simple k-means featurizer. It is defined as a class object that can be fitted to training data and transform any new data. To illustrate the difference between using and not using target information when clustering, we apply the featurizer to a synthetic dataset generated using scikit-learn's make_moons function (Example 6-4). We then plot the Voronoi diagram of the cluster boundaries. Figure 6-6 shows a comparison of the results. The bottom panel shows the clusters trained without target information. Notice that a number of clusters span the empty space between the two classes. The top panel shows that when the clustering algorithm is given target information, the cluster boundaries align much better along class boundaries.

Example 6-3. K-means featurizer.

```
import numpy as np
from sklearn.cluster import KMeans

class KMeansFeaturizer:
```

```
"""Transforms numeric data into k-means cluster memberships.

This transformer runs k-means on the input data and converts each data point
into the id of the closest cluster. If a target variable is present, it is
scaled and included as input to k-means in order to derive clusters that
obey the classification boundary as well as group similar points together.
"""

def __init__(self, k=100, target_scale=5.0, random_state=None):
      self.k = k
      self.target_scale = target_scale
      self.random_state = random_state

def fit(self, X, y=None):
      """Runs k-means on the input data and find centroids.
      """
      if y is None:
            # No target variable, just do plain k-means
            km_model = KMeans(n_clusters=self.k,
                                       n_init=20,
                                       random_state=self.random_state)
            km_model.fit(X)

            self.km_model_ = km_model
            self.cluster_centers_ = km_model.cluster_centers_
            return self

      # There is target information. Apply appropriate scaling and include
      # into input data to k-means
      data_with_target = np.hstack((X, y[:,np.newaxis]*self.target_scale))

      # Build a pre-training k-means model on data and target
      km_model_pretrain = KMeans(n_clusters=self.k,
                                                n_init=20,
                                                random_state=self.random_state)
      km_model_pretrain.fit(data_with_target)

      # Run k-means a second time to get the clusters in the original space
      # without target info. Initialize using centroids found in pre-training.
      # Go through a single iteration of cluster assignment and centroid
      # recomputation.
      km_model = KMeans(n_clusters=self.k,
                                  init=km_model_pretrain.cluster_centers_[:,:2],
                                  n_init=1,
                                  max_iter=1)
      km_model.fit(X)

      self.km_model = km_model
      self.cluster_centers_ = km_model.cluster_centers_
      return self

def transform(self, X, y=None):
      """Outputs the closest cluster id for each input data point.
      """
      clusters = self.km_model.predict(X)
      return clusters[:,np.newaxis]

def fit_transform(self, X, y=None):
      self.fit(X, y)
      return self.transform(X, y)
```

Example 6-4. Kmeans featurization with and without target hints.

```
from scipy.spatial import Voronoi, voronoi_plot_2d
from sklearn.datasets import make_moons

training_data, training_labels = make_moons(n_samples=2000, noise=0.2)
kmf_hint = KMeansFeaturizer(k=100, target_scale=10).fit(training_data, training_labels)
kmf_no_hint = KMeansFeaturizer(k=100, target_scale=0).fit(training_data,
training_labels)

def kmeans_voronoi_plot(X, y, cluster_centers, ax):
    """Plots the Voronoi diagram of the kmeans clusters overlayed with the data"""
    ax.scatter(X[:, 0], X[:, 1], c=y, cmap='Set1', alpha=0.2)
    vor = Voronoi(cluster_centers)
    voronoi_plot_2d(vor, ax=ax, show_vertices=False, alpha=0.5)
```

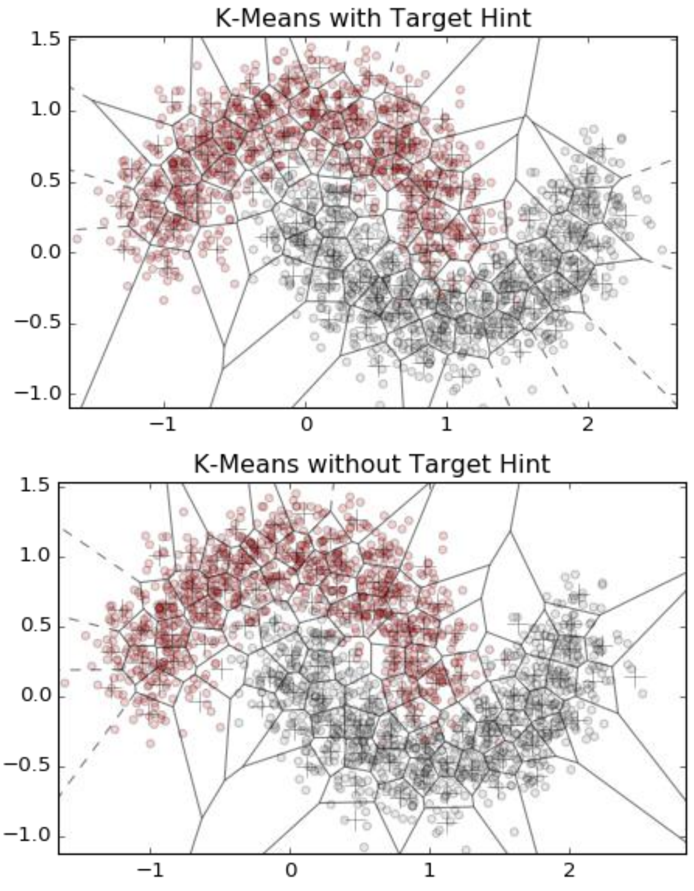


Figure 6-6. K-means clusters with (top panel) and without (bottom panel) using target class information. The two moons of the dataset are colored according to their class labels. Cluster boundaries in the top panel are better aligned with the class boundary. The bottom panels clearly shows a number of clusters bridging the gap between classes.

Let’s test the effectiveness of k-means features for classification. Example 6-5 applies logistic regression on the input data augmented with k-means cluster features. It compares the results against the Support Vector Machine with Radial Basis Function kernel (RBF SVM), k-nearest neighbors (kNN), random forest (RF), and gradient boosted trees (GBT). Random forest and gradient boosted trees are popular non-linear classifiers with state-of-the-art performance. RBF SVM is a reasonable non-linear classifier for Euclidean space. kNN classifies data according to the average of its k nearest neighbors. (See [[Appendix]] for an overview of each of the classifiers.)

The default input data to the classifiers are the 2D coordinates of the data. Logistic regression is also given the cluster membership features (labeled “LR with k-means” in

Figure 6-7). As a baseline, we also try logistic regression on just the 2D coordinates (labeled “LR”).

Example 6-5. Classification with k-means cluster features.

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier

# Generate some test data from the same distribution as training data
test_data, test_labels = make_moons(n_samples=2000, noise=0.3)

# Use the k-means featurizer to generate cluster features
training_cluster_features = kmf_hint.transform(training_data)
test_cluster_features = kmf_hint.transform(test_data)

# Form new input features with cluster features
training_with_cluster = scipy.sparse.hstack((training_data, training_cluster_features))
test_with_cluster = scipy.sparse.hstack((test_data, test_cluster_features))

# Build the classifiers
lr_cluster = LogisticRegression(random_state=seed).fit(training_with_cluster,
training_labels)
classifier_names = ['LR',
                              'kNN',
                              'RBF SVM',
                              'Random Forest',
                              'Boosted Trees']
classifiers = [LogisticRegression(random_state=seed),
                       KNeighborsClassifier(5),
                       SVC(gamma=2, C=1),
                       RandomForestClassifier(max_depth=5, n_estimators=10,
max_features=1),
                       GradientBoostingClassifier(n_estimators=10, learning_rate=1.0,
max_depth=5)]
for model in classifiers:
      model.fit(training_data, training_labels)

# Helper function to evaluate classifier performance using ROC
def test_roc(model, data, labels):
      if hasattr(model, "decision_function"):
            predictions = model.decision_function(data)
      else:
            predictions = model.predict_proba(data)[:,1]
      fpr, tpr, _ = sklearn.metrics.roc_curve(labels, predictions)
      return fpr, tpr

# Plot results
import matplotlib.pyplot as plt
plt.figure()
fpr_cluster, tpr_cluster = test_roc(lr_cluster, test_with_cluster, test_labels)
plt.plot(fpr_cluster, tpr_cluster, 'r-', label='LR with k-means')

for i, model in enumerate(classifiers):
      fpr, tpr = test_roc(model, test_data, test_labels)
      plt.plot(fpr, tpr, label=classifier_names[i])

plt.plot([0, 1], [0, 1], 'k--')
plt.legend()
```

Figure 6-7 shows the Receiver-Operator Characteristic (ROC) curves of each of the classifiers when evaluated on the test set. An ROC curve shows the trade-off between true positives and the false positives as we vary the classification decision boundary. (See [[eval ML book]] for more details.) A good classifier should quickly reach a high true positive rate at a low false positive rate. So curves that rise sharply towards the upper left corner are good.

Our plot shows that logistic regression performs much better with cluster features rather than without. In fact, with cluster features, the linear classifier performs just as well as non-linear classifiers. One minor caveat is that in this toy example, we did not tune the hyperparameters for any of the models. There may be performance differences once the models are fully tuned. But at least it is possible for LR with k-means to be on par with non-linear classifiers. This is a nice result because linear classifiers are much cheaper to train than non-linear classifiers. Lower computation cost allows us to try more models with different features in the same period of time, which increases the chance of ending up with a much better model.

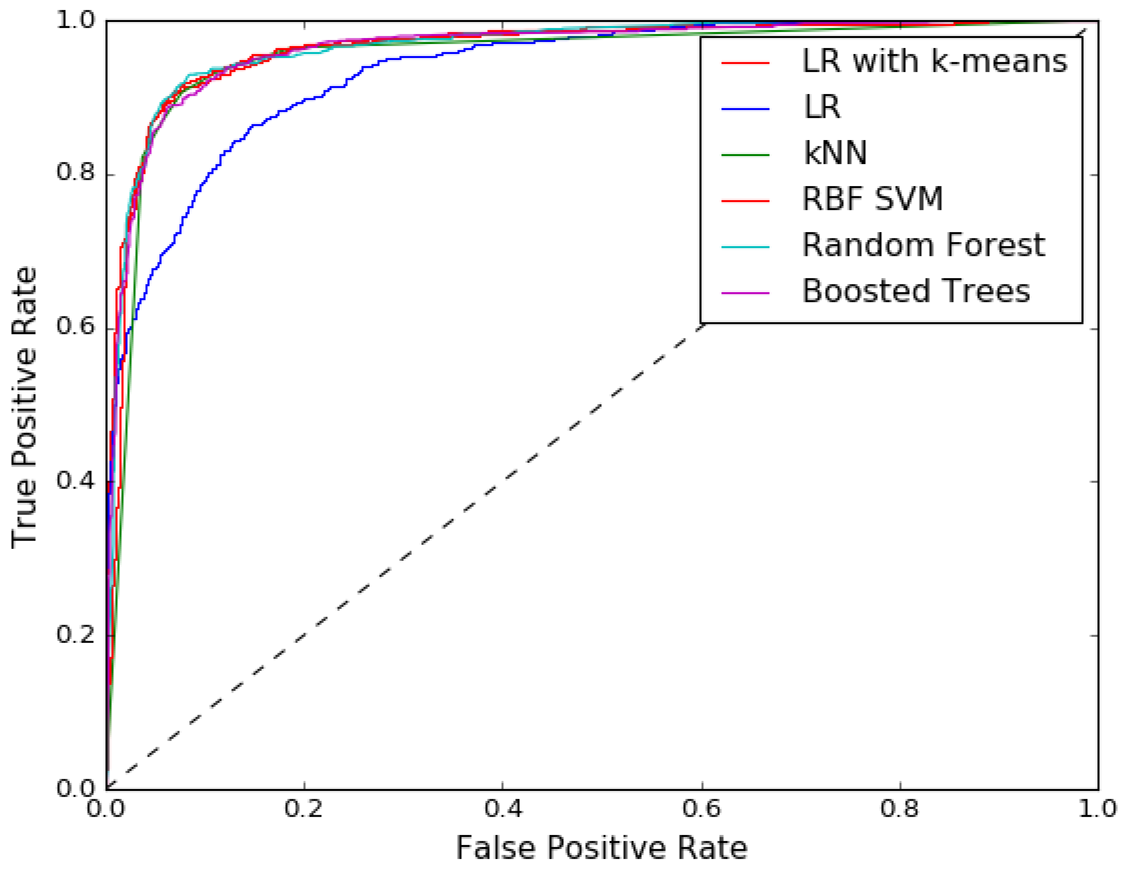


Figure 6-7. ROC of k-means + logistic regression vs. non-linear classifiers and plain logistic regression on the synthetic two-moons dataset.

Alternative dense featurization

Instead of one-hot cluster membership, a data point can also be represented by a dense vector of its inverse distance to each cluster center. This retains more information than simple binary cluster assignment, but the representation is now dense. There is a trade-off here. One-hot cluster membership results in a very lightweight, sparse representation, but one might need a larger k to represent data of complex shapes. Inverse distance representation is dense, which could be more expensive for the modeling step, but one might be able to get away with a smaller k.

A compromise between sparse and dense is to retain inverse distances only for *p* of the closest clusters. But now *p* is an extra hyperparameter to tune. (Can you understand now why feature engineering requires so much fiddling?) There is no free lunch.

Concluding Remarks

Using k-means to turn spatial data into features is an example of *model stacking*—where the input to one model is the output of another. Another example of stacking is to use the output of a decision tree-type model (random forest or gradient boosted trees) as input to a linear classifier. Stacking has become an increasingly popular technique in recent years. Non-linear classifiers are expensive to train and maintain. The key intuition with stacking is to push the non-linearities into the features, and use a very simple, usually linear model as the last layer. The featurizer can be trained offline, which means that one can use expensive models that require more computation power or memory but generate useful features. The simple model at the top level can be quickly adapted to the changing distributions of online data. This is a great trade-off between accuracy and speed, and this is often used in applications that require fast adaptation to changing data distributions like targeted advertising.

Key Intuition for Model Stacking

Sophisticated base layers (often with expensive models) to generate good (often non-linear) features, combined with simple and fast top layer model. This often strikes the right balance between model accuracy and speed.

Compared to using a non-linear classifier, k-means stacked with logistic regression is cheaper to train and store. Table 6-1 is a chart of the training and prediction complexity in both computation and memory for a number of machine learning models. n denotes the number of data points, d the number of (original) features.

For k-means, the training time is $O(nkd)$ because each iteration involves computing the d-dimensional distance between every data point and every centroid (k). We optimistically assume that the number of iterations is not a function of n, though this may not be true in all cases. Prediction requires computing the distance between the new data point and each of the k centroids, which is $O(kd)$. Storage space requirement is $O(kd)$, for the coordinates of the k centroids.

Logistic regression training and prediction is both linear in the number of data points and feature dimensions. RBF SVM training is expensive because it involves computing the kernel matrix for every pair of input data. Prediction is less expensive is linear in the number of support vectors s and the feature dimension d. Boosted trees training and prediction are linear in data size and the size of the model (t trees, each with at most 2^m leaves where m is the maximum depth of the tree). A naive implementation of kNN requires no training time at all because the training data itself is essentially the model. The cost is paid at prediction time where the input must be evaluated against each of the original training points and partially sorted to retrieve the k closest neighbors.

Overall k-means + LR is the only combination that is linear (with respect to the size of training data O(nd) and model size O(kd)) at both training and prediction time. The complexity is most similar to boosted trees, which has costs that are linear in the number of data points, the feature dimension, and the size of the model ($O(2^m t)$). It is hard to say whether k-means + LR or boosted trees will result in a smaller model—it depends on the spatial characteristics of the data.

Table 6-1. Complexity of ML models.

Model	Time	Space
K-means training	$O(nkd)$[a]	$O(kd)$
K-means predict	$O(kd)$	$O(kd)$
LR + cluster features training	$O(n(d+k))$	$O(d+k)$
LR + cluster features predict	$O(d+k)$	$O(d+k)$
RBF SVM training	$O(n^2d)$	$O(n^2)$
RBF SVM predict	$O(sd)$	$O(sd)$
Boosted trees training	$O(nd2^mt)$	$O(nd + 2^mt)$
Boosted trees predict	$O(2^mt)$	$O(2^mt)$
kNN training	$O(1)$	$O(nd)$
kNN predict	$O(nd + k \log n)$	$O(nd)$

[a] Streaming k-means can be done in time $O(nd (\log k + \log \log n))$, which is much faster than $O(nkd)$ for large k.

Potential for Data Leakage

Those who remember our caution regarding data leakage (cf. "Guarding against data leakage (Bin-counting: days of future past)") might ask whether including the target variable in the k-means featurization step would cause such a problem. The answer is "yes," but not as much as the case for bin-counting. If we use the same dataset for learning the clusters and building the classification model, then information about the target will have leaked into the input variables. As a result, accuracy evaluations *on the training data* will probably be overly optimistic, but the bias will go away when evaluating on a hold-out validation set or test set. Furthermore, the leakage will not as bad as in the case of bin-counting statistics (cf. "Bin-counting"), because the lossy compression of the clustering algorithm will have abstracted away some of that information. To be extra careful in preventing leakage, one can always hold out a separate dataset for deriving the clusters, just like in the case of bin-counting.

K-means featurization is useful for real-valued, bounded numeric features that form clumps of dense regions in space. The clumps can be of any shape, because we can just increase the number of clusters to approximate them. (Unlike in the classic clustering setup, we are not concerned with discovering the “true” number of clusters; we only need to cover them.)

K-means cannot handle feature spaces where the Euclidean distance does not make sense, i.e., weirdly distributed numeric variables or categorical variables. If the feature set contains those variables, then there are several ways to handle them.

1. Apply k-means featurization only on the real-valued, bounded numeric features.
2. Define a custom metric (see [[chapter nnn]]) to handle multiple data types and use the k-medoids algorithms. (K-medoids is analogous to k-means but allows for arbitrary distance metrics.)
3. Categorical variables can be converted to binning statistics (see Chapter 4), then featurized using k-means.

Combined with techniques for handling categorical variables and time series, k-means featurization can be adapted to handle the kind of rich data that often appears in customer marketing and sales analytics. The resulting clusters can be thought of as user segments, which are very useful features for the next modeling step.

Deep learning, which we will discuss in the next chapter, takes model stacking to a whole new level by layering neural networks on top of one another. Two recent winners of the ImageNet challenge involved 13 and 22 layers of neural networks. Just like k-means, the lower levels of deep learning models are unsupervised. They take advantage of the availability of lots of unlabeled training images and look for combinations of combinations of pixels that yield good image features.

[1] This chapter is inspired by a conversation with Ted Dunning, active Apache contributor and noted author. The stacking example came directly from Ted, and he provided many helpful comments in the course of writing. If one could have co-authors for individual chapters, Ted would be a co-author for this one.

[2] We use the words “surface” and “manifold” interchangeably in this chapter. The analogy works well for 2-dimensional manifolds embedded in a 3-dimensional space. But it breaks down beyond 3-dimensions. A high-dimensional manifold does not conform to our usual notion of a “surface.” Some of the more outlandish manifolds have holes, and some loop back onto itself in a way that would never appear in the real physical world (e.g., M.C. Escher’s endless waterfall). Most data models assume nice manifolds, not the crazy ones.

[3] This is a tried-and-true idea in mathematics. For instance, the derivative of a function measures the speed of change as each point. Globally, the function may do all sorts of weird things. But locally, it can be approximated by a linear function of the derivative. If we know the derivative at each point, then calculus allows us to more or less recover the entire original function.

Chapter 7. Automating the Featurizer: Image Feature Extraction and Deep Learning

Sight and sound are innate senses for humans. Our brains are hardwired to process visual and auditory signals. Upon birth, most babies are immediately able to process and understand what they see and hear. Language skills, on the other hand, take months to develop and years to master. Most of us take vision and hearing for granted, but we all have to put in a lot of work and intentionally train our brain to understand and use language.

Interestingly, the situation is the reverse for machine learning. We have made much more headway text analysis applications than image or audio. Take the problem of search for example. People have enjoyed years of relative success in information retrieval and text search, whereas image and audio search are still being perfected. The breakthrough in deep learning models in the last five years may finally herald the long-awaited revolution in image and speech analysis.

The difficulty of progress is directly related to the difficulty of extracting meaningful features from the respective types of data. Machine learning models require semantically meaningful features to make semantically meaningful predictions. In text analysis, particularly for languages such as English where a basic unit of semantic meaning (a word) is easily extractable, progress can be made very fast. Images and audio, on the other hand, are recorded as digital pixels or wave forms. A single "atom" in an image is a pixel, and in audio data is a single measurement of waveform intensity. They contains much less semantic information than an atom—a word—of text data. Therefore, the job of feature extraction and engineering is much more challenging on image and audio than on text.

In the last twenty years, computer vision research has focused on manually defined pipelines for extracting good image features. For a while, image feature extractors such as SIFT and HOG were the standard. Recent developments in deep learning research has extended the reach of traditional machine learning models by incorporating automatic feature extraction as base layers. They essentially replaced manually defined feature image extractors with manually defined models that automatically learn and extract features. The manual work is still there, just abstracted further into the belly of the modeling beast.

In this chapter, we will start with popular image feature extraction SIFT and HOG and then dive into the most complicated modeling machinery covered in this book: deep learning for feature learning.

Simplest Image Features (and Why They Don't Work)

What are the *right* features to extract from an image? The answer of course depends on what we are trying to do with those features. Let' s say our task is image retrieval: we are given a picture and asked to similar pictures from a database of images. We need to decide how to represent each image, and how to measure the difference between them. Can we just look at the percentage of different colors in an image? Figure 7-1 shows two pictures having roughly the same color profile but very different meanings; one looks like white cloud in a blue sky, and the other is the flag of Greece. So color information is probably not enough to characterize an image.

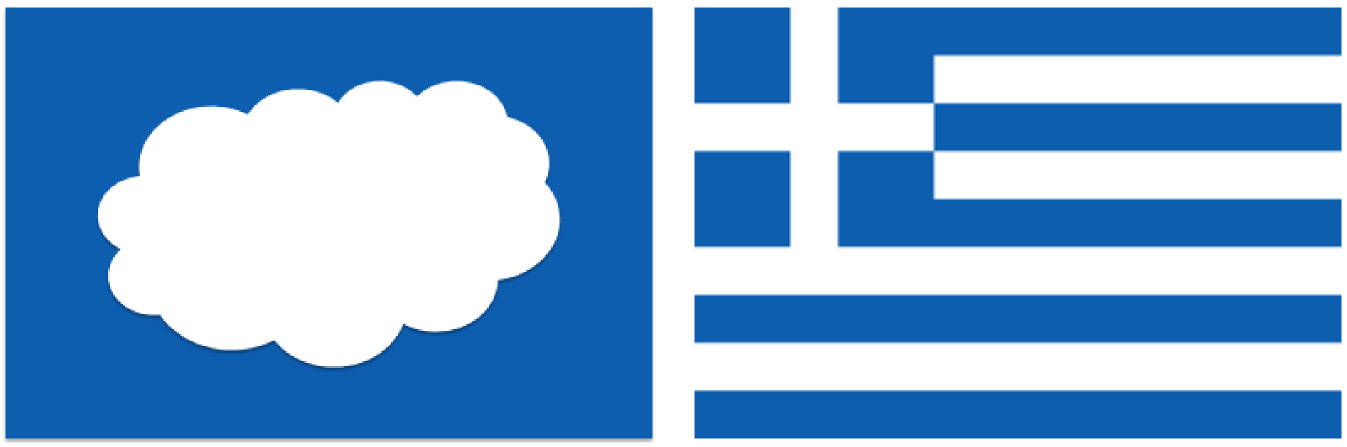


Figure 7-1. Blue and white pictures. Same color profile, very different meaning.

Another simple idea is to measure the pixel value differences between images. First, resize the images to have the same width and height. Each image is represented by a matrix of pixel values. The matrix can be stacked into one long vector, either by row or by column. The color of each pixel (e.g., the RGB encoding of the color) is now a feature of the image. Finally, measure the Euclidean distance between the long pixel vectors. This would definitely tell apart the Greek flag and white clouds. But it is too stringent as a similarity measure. A cloud could take on a thousand different shapes and still be a cloud. It could be shifted to the side of the image, or half of it might lie in shadow. All of these transformations would increase the Euclidean distance, but they shouldn't change the fact that the picture is still a cloud.

[[picture of subtracting two images, and subtracting shifted clouds]]

The problem is that individual pixels do not carry enough semantic information about the image. Therefore they are bad atomic units for analysis. In 1999, computer vision researchers figured out a better way to represent images using statistics of image patches. This was called Scale Invariant Feature Transform (SIFT). [Lowe, 1999]

SIFT was originally developed for the task of object recognition, which involves not only correctly tagging the image as containing an object, but pinpointing its location in the image. The process involves analyzing the image at a pyramid of possible scales, detecting interest points that could indicate the presence of the object, extracting features (commonly called *image descriptors* in computer vision) about the interest points, and determining the pose of the object.

Over the years, the usage of SIFT expanded to extract features not only for interest points but across the entire image. The SIFT feature extraction procedure is very similar to another technique called Histogram of Oriented Gradients (HOG). [Dalal and Triggs, 2005] Both of them essentially compute histograms of gradient orientations. We now describe this in detail.

Manual Feature Extraction: SIFT and HOG

Image gradient

To do better than raw pixel values, we have to somehow “organize” the pixels into more informative units. Differences between neighboring pixels are often very useful. Pixel values usually differ at the boundary of objects, when there is a shadow, within a pattern, or on a textured surface. The difference in value between neighboring pixels is called an *image gradient*.

The simplest way to compute the image gradient is to separately calculate the differences along the horizontal (x-) and vertical (y-) axes of the image, then compose them into a 2D vector. This involves two 1D difference operations that can be handily represented by a vector mask or filter. The mask [1, 0, -1] takes the difference between the left neighbor and the right neighbor or the difference between the up-neighbor and the down-neighbor, depending on which direction we apply the mask. There are 2D gradient filters as well. But for the purpose of this example, the 1D filter suffices.

To apply a filter to an image, we perform a *convolution*. It involves flipping the filter and taking the inner product with a small patch of the image, then moving to the next patch. Convolutions are very common in signal processing. We’ll use * to denote the operation:

[a b c] * [1 2 3] = c*1 + b*2 + a*3 .

The x- and y-gradients at pixel (i,j) are:

\(\begin{eqnarray} g_x(i,j) & = &\begin{bmatrix} 1 & 0 & -1\end{bmatrix} * \begin{bmatrix} I(i-1,j) & I(i,j) & I(i+1,j) \end{bmatrix} \\ & = & -1 * I(i-1,j) + 1 * I(i+1,j) , \\ g_y(i,j) & = & \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} * \begin{bmatrix} I(i,j-1) & I(i,j) & I(i,j+1) \end{bmatrix} \\ & = & -1 * I(i,j-1) + 1 * I(i,j+1) . \end{eqnarray}\)

Together, they form the gradient

$$\nabla I(i,j) = \begin{bmatrix} g_x(i,j) \\ g_y(i,j) \end{bmatrix}.$$

A vector can be completely descried by its direction and magnitude. The magnitude of the gradient is equal to the Euclidean norm of the gradient $\left(\sqrt{g_x^2 + g_y^2}\right)$, which indicates how large the pixel values change around the pixel. The direction or orientation of the gradient depends on the relative size of the change in the horizontal and vertical directions; it can be computed as $\theta = \arctan\left(\frac{g_y}{g_x}\right)$. Figure 7-2 illustrates these mathematical concepts.

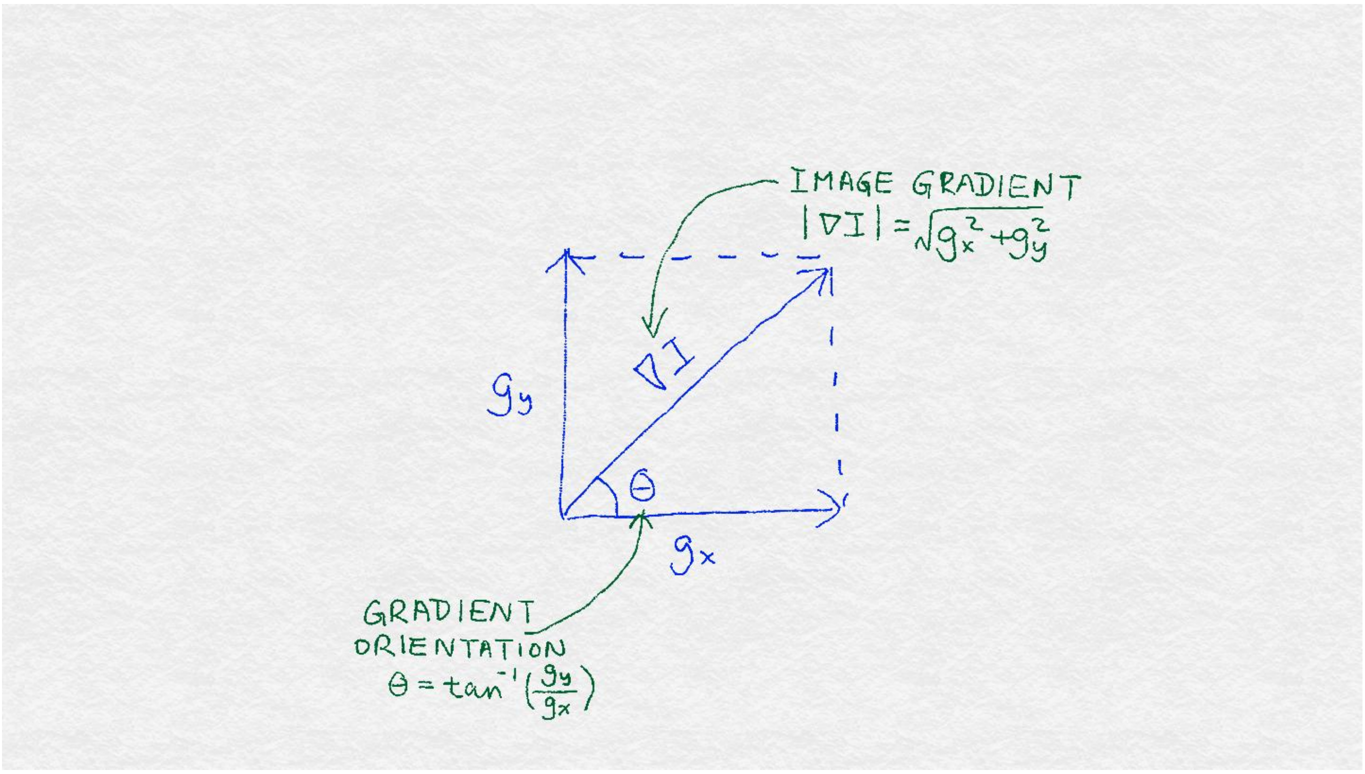


IMAGE GRADIENT
$|\nabla I| = \sqrt{g_x^2 + g_y^2}$
∇I
g_y
θ
g_x
GRADIENT ORIENTATION
$\theta = \tan^{-1}\left(\frac{g_y}{g_x}\right)$

Figure 7-2. Illustration of the definition of an image gradient.

Figure 7-3 illustrates examples of the simple image gradient that is composed of the vertical and horizontal gradients. Each example is an image of 9 pixels. Each pixel is labeled with a grayscale value. (Smaller numbers correspond to a darker color.) The gradient for the center pixel is shown below each image. The image on the left contains horizontal stripes where the color only changes vertically. Therefore the horizontal gradient is zero and the gradient is non-zero vertically. The center image contains vertical stripes; therefore the horizontal gradient is zero. The image on the right contains diagonal stripes and the gradient is also diagonal.

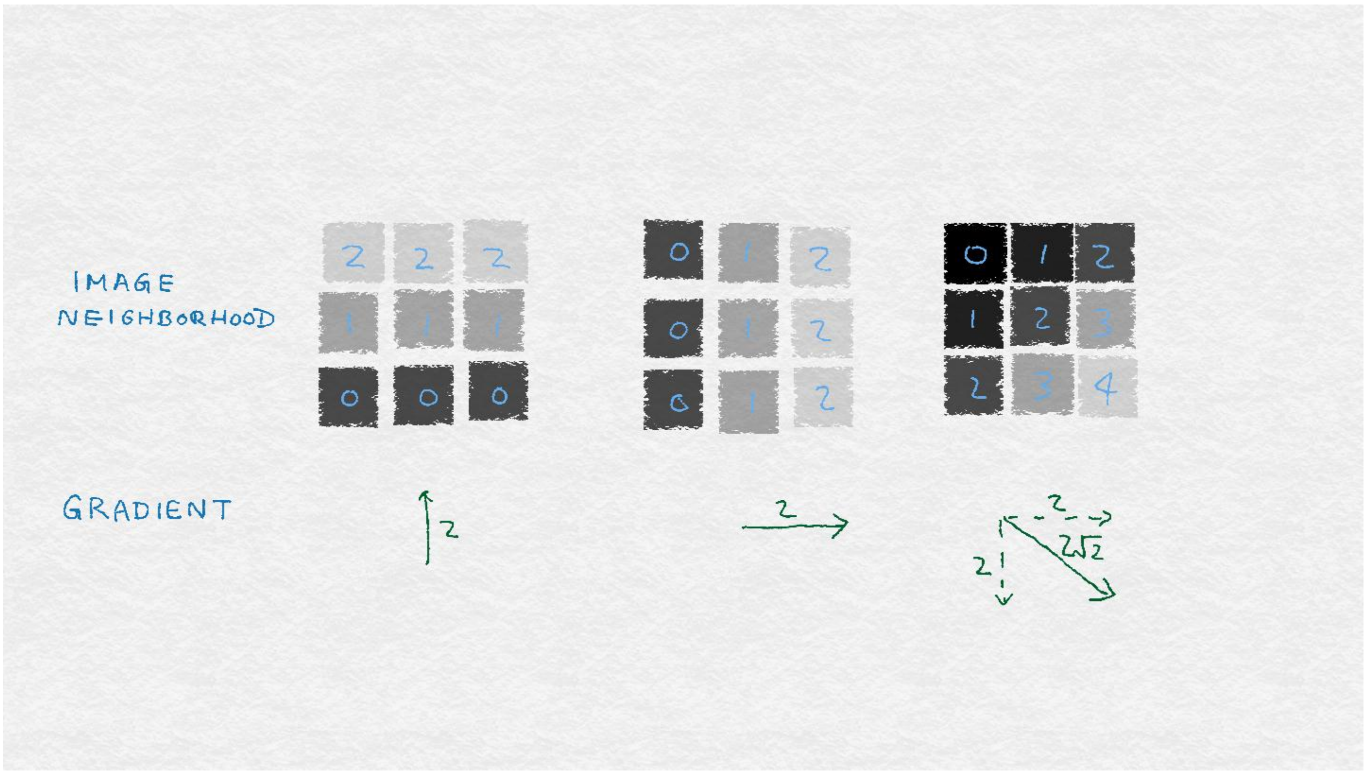


Figure 7-3. Simple examples of the image gradient.

The definition works on synthetic toy examples. Would they work well on a real image? Figure 7-4 shows the horizontal and vertical gradients of a picture of a cat. The code is give in Example 7-1. Since the gradients are computed at every pixel location of the original image, we end up with two new matrices, each of which can be visualized as an image.

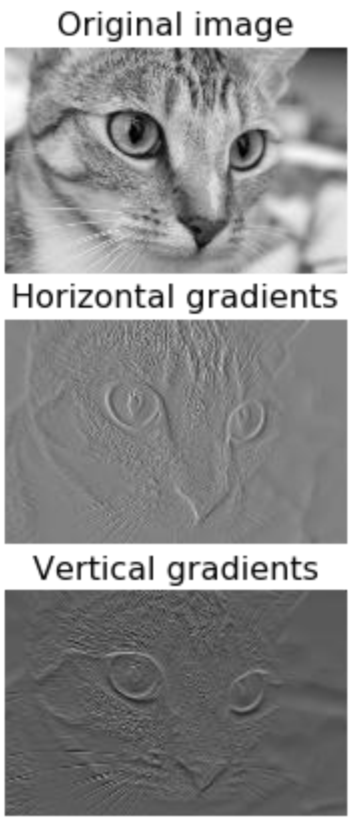


Figure 7-4. Image gradients of an image of a cat from scikit-image.

Example 7-1. Calculating simple image gradients using Python.

```
import matplotlib.pyplot as plt
import numpy as np
from skimage import data, color

# Load the example image and turn it into grayscale
image = color.rgb2gray(data.chelsea())

# Compute the horizontal gradient using the centered 1D filter
# This is equivalent to replacing each non-border pixel with the
# difference between its right and left neighbors. The leftmost
# and rightmost edges have a gradient of 0.
gx = np.empty(image.shape, dtype=np.double)
gx[:, 0] = 0
gx[:, -1] = 0
gx[:, 1:-1] = image[:, :-2] - image[:, 2:]
```

```
# Same deal for the vertical gradient
gy = np.empty(image.shape, dtype=np.double)
gy[0, :] = 0
gy[-1, :] = 0
gy[1:-1, :] = image[:-2, :] - image[2:, :]

# Matplotlib incantations
fig, (ax1, ax2, ax3) = plt.subplots(3, 1,
                                                       figsize=(5, 9),
                                                       sharex=True,
                                                       sharey=True)

ax1.axis('off')
ax1.imshow(image, cmap=plt.cm.gray)
ax1.set_title('Original image')
ax1.set_adjustable('box-forced')

ax2.axis('off')
ax2.imshow(gx, cmap=plt.cm.gray)
ax2.set_title('Horizontal gradients')
ax2.set_adjustable('box-forced')

ax3.axis('off')
ax3.imshow(gy, cmap=plt.cm.gray)
ax3.set_title('Vertical gradients')
ax3.set_adjustable('box-forced')
```

Note that the horizontal gradient picks out strong *vertical* patterns such as the inner edges of the cat' s eyes, while the vertical gradient picks out strong *horizontal* patterns such as the whiskers and the upper and lower lids of the eyes. This might seem a little paradoxical at first, but it makes sense once we think about it a bit more. The horizontal (*x*-) gradient identify changes in the horizontal direction. A strong vertical pattern span multiple *y* pixels at roughly the same *x* position. Hence vertical patterns result in horizontal differences in pixel values. This is what our eyes detect as well.

Gradient orientation histogram

Individual image gradients can pick out minute differences in an image neighborhood. But our eyes see bigger patterns than that. For instance, we see an entire cat' s whisker, not just a small section. The human vision system identifies contiguous patterns in a region. So we still have more work to do to summarize the image gradients in a neighborhood.

How exactly might we summarize vectors? A statistician would answer, "Look at the distribution!" SIFT and HOG both take this path. In particular, they compute (normalized) histograms of the gradient vectors as image features. A histogram divides data into bins and count how many is in each bin; this is an (unnormalized) empirical distribution. Normalization ensures that the counts sum to 1. The mathematical language is that it has unit l1 norm.

An image gradient is a vector, and vectors can be represented by two components: the orientation and magnitude. So we still need to decide how to design the histogram to take both components into account. SIFT and HOG settled on a scheme where the image gradients are binned by their orientation angle θ, weighted by the magnitude of each gradient. Here is the procedure:

1. Divide 0°-360° into equal sized bins.
2. For each pixel in the neighborhood, add a weight *w* to the bin corresponding to its orientation ϑ.
3. *w* is a function of the magnitude of the gradient and other relevant info. For instance, other relevant info could be the inverse distance of the pixel to the center of the image patch. The idea is that the weight should be large if the gradient is large, and pixels near the center of the image neighborhood matter more than pixels that are farther away.
4. Normalize the histogram.

Figure 7-5 provides an illustration of a gradient orientation histogram of 8 bins composed from an image neighborhood of 4x4 pixels.

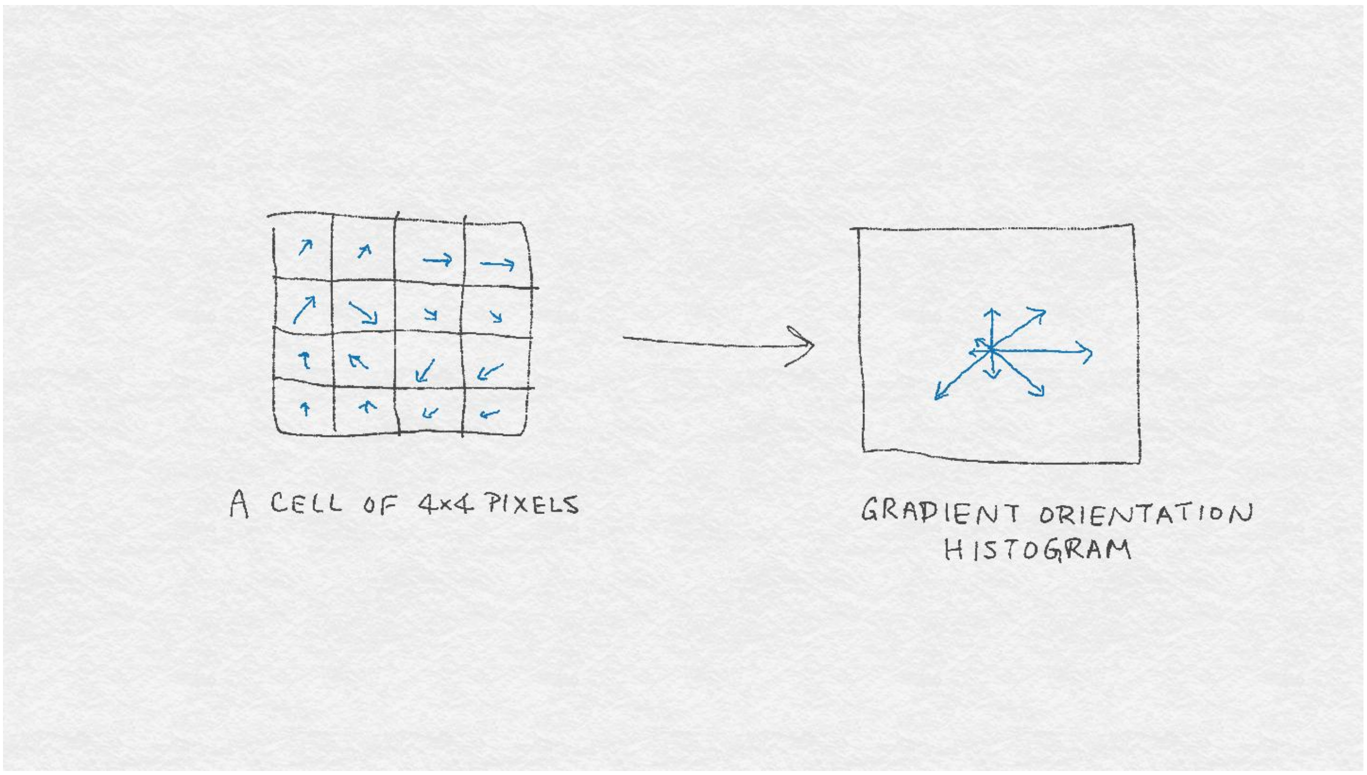


Figure 7-5. Illustration of a gradient orientation histogram of 8 bins based on gradients from a 4x4 square cell of pixels.

There are, of course, a number of knobs to tweak in the basic gradient orientation histogram algorithm, as well as some optional bells and whistles. As usual, the right settings are probably highly dependent on the particular images one wants to analyze.

How many bins? Should they span from 0° -360° (signed gradients) or 0° -180° (unsigned gradients)?

Having more bins leads to finer-grained quantization of gradient orientation, and thus retains more information about the original gradients. But having too many bins is unnecessary and could lead to overfitting to the training data. For example, recognizing a cat in an image probably does not depend on the cat' s whisker being oriented exactly at 3° .

There is also the question of whether the bins should span from 0° -360° , which would retain the sign of the gradient along the y-axis, or from 0° -180° , which would not retain the sign of the vertical gradient. Dalal and Triggs, authors of the original HOG paper, experimentally determined that 9 bins spanning from 0° -180° is best, whereas the SIFT paper [Lowe, 2004] recommended 8 bins spanning from 0° -360° .

What weight functions to use?

The HOG paper compares various gradient magnitude weighting schemes: the magnitude itself, its square, square root, binarized, or clipped at the high or low ends. The plain magnitude, without adornments, performed the best in their experiments.

SIFT also uses the plain magnitude of the gradient. On top of it, it wants to avoid sudden changes in the feature descriptor from small changes in the position of the image window. So it down weighs gradients that come from the edges of the neighborhood using a Gaussian distance function measured from the window center. In other words, the gradient magnitude is multiplied by $\frac{1}{2\pi\sigma^2}e^{-\|p-p_0\|^2/2\sigma^2}$, where p is the location of the pixel that generated the gradient, p_0 the location of the center of the image neighborhood, and σ, the width of the Gaussian, is set to one half the radius of the neighborhood.

SIFT also wants to avoid large changes in the orientation histogram from small changes in the orientation of individual image gradients. So it uses an interpolation trick that spreads the weight from a single gradient into adjacent orientation bins. In particular, the root bin (the bin that the gradient is assigned to) gets a vote of 1 times the weighted magnitude. Each of the adjacent bins get a vote of 1-d, where d is the difference in histogram bin unit from the root bin.

Overall, the vote from a single image gradient for SIFT is

SIFT_weight(gradient of pixel p, bin b) = interpolation weight (b) * Gaussian distance to center (p) * gradient magnitude

How are neighborhoods defined? How should they cover the image?

HOG and SIFT both settled on a two-level representation of image neighborhoods: first, adjacent pixels are organized into cells, neighboring cells are then organized into blocks. A orientation histogram is computed for each cell, and the cell histogram vectors are concatenated to form the final feature descriptor for the whole block.

SIFT uses cells of 16x16 pixels, organized into 8 orientation bins, then grouped by blocks of 4x4 cells, making for 4x4x8=128 features for the image neighborhood.

The HOG paper experimented with rectangular and circular shapes for the cells and blocks. Rectangular cells are called R-HOG. The best R-HOG setting was found to be 8x8 pixels of 9 orientation bins each, grouped into blocks of 2x2 cells. Circular windows are called C-HOG, with variants determined by the radius of the central cell, whether or not the cells are radially divided, the width of the outer cells, etc.

No matter how the neighborhoods are organized, they typically overlap to form the feature vector for the whole image. In other words, cells and blocks shift across the image horizontally and vertically, a few pixels at a time, to cover the entire image.

The main ingredients of neighborhood architecture are multi-level organization and overlapping windows that shift across the image. The same ingredients are utilized in the design of deep learning networks.

What kind of normalization?

Normalization means to “even out” the feature descriptors so that they have comparable magnitude. It is synonymous with scaling, which we discussed in Chapter 3. We found that feature scaling on text features (in the form of tf-idf) did not have a large effect on classification accuracy. The story is quite different for image features, which can be quite sensitive to changes in lighting and contrast that appear in natural images. For instance, consider images of an apple under a strong spotlight versus a soft diffused light coming through a window. The image gradients would have very different magnitudes, even though the object is the same. For this reason, image featurization in computer vision usually starts with global color normalization to remove illumination and contrast variance. For SIFT and HOG, it turns out that such preprocessing is unnecessary so long as we normalize the features.

SIFT follows a normalize-threshold-normalize scheme. First, the block feature vector is normalized to unit length (l2 normalization). Then, the features are clipped to some maximum value in order to deal with extreme lighting effects such as color saturation from the camera. Finally, the clipped features are normalized to unit length again.

The HOG paper experimented with different normalization schemes involving l2 and l1 norms, including the normalize-threshold-normalize scheme used in the SIFT paper. They found that pure l1 normalization to be slightly less reliable than the other methods (which performed comparably).

Preprocessing: smoothing, color/brightness/contrast normalization

To smooth or not to smooth, that is the question. If you zoom into a digital photograph of the sky, you' ll see mottled blues instead of a uniform color. Random spots and variations in color are very common. Often times, the first step in image processing is to smooth the image (by convolving with a smoothing filter). We

SIFT: invariance to orientation of the object

SIFT and HOG dominated the computer vision landscape for nearly a decade. They generate robust and representative features that are useful for object detection and identification. Other prominent featurization methods include spatial pyramids, which takes neighborhood architecture to the next step. Instead of concatenating histogram vectors from cells at the same resolution, a pyramid kernel organizes cells into increasingly larger radius.

One downside of histogram-based features is that they are not interpretable. The information from different pixels are aggregated into the same set of bins, making it difficult to reverse engineer the histogram vector back into a recognizable image pattern. Visual bag of words is a featurization scheme that clusters the orientation histograms using kmeans, then create counts of clusters. Each histogram cluster is like a visual word. (how visual are they?)

Despite huge advances in the area, image featurization is still more of an art than a science. Ten years ago, people handcrafted feature extraction steps using a combination of image gradient, edge detection, orientation, spatial cues, smoothing, and normalization. Nowadays, deep learning architects build models that encapsulate much the same ideas, but the parameters are automatically learned from training images. The magic voodoo is still there, just hidden one abstraction deeper in the model.

Bibliography

Dalal, Navneet, and Bill Triggs. 2005. "Histograms of oriented gradients for human detection," *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR' 05)*, San Diego, CA, USA, 2005, pp. 886-893 vol. 1.
doi: 10.1109/CVPR.2005.177

Lowe, David G. 1999. “Object recognition from local scale-invariant features,″ *International Conference on Computer Vision,* Corfu, Greece (September 1999), pp. 1150-1157.

Lowe, David G.. 2004. “Distinctive image features from scale-invariant keypoints,″ *International Journal of Computer Vision,* 60, 2, pp. 91-110.

Malisiewicz, Tomasz. 2015. “From feature descriptors to deep learning: 20 years of computer vision.” *Tombone’s Computer Vision Blog,* January 20. http://www.computervisionblog.com/2015/01/from-feature-descriptors-to-deep.html

Zeiler, Matthew D., and Rob Fergus. 2014. “Visualizing and Understanding Convolutional Networks,” *Proceedings of the 13th European Conference on Computer Vision*.

Chapter 8 Automating the Featurizer: Image Feature Extraction and Deep **Learning**

Chapter 9 Metrics and Metric Learning

Index

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*.

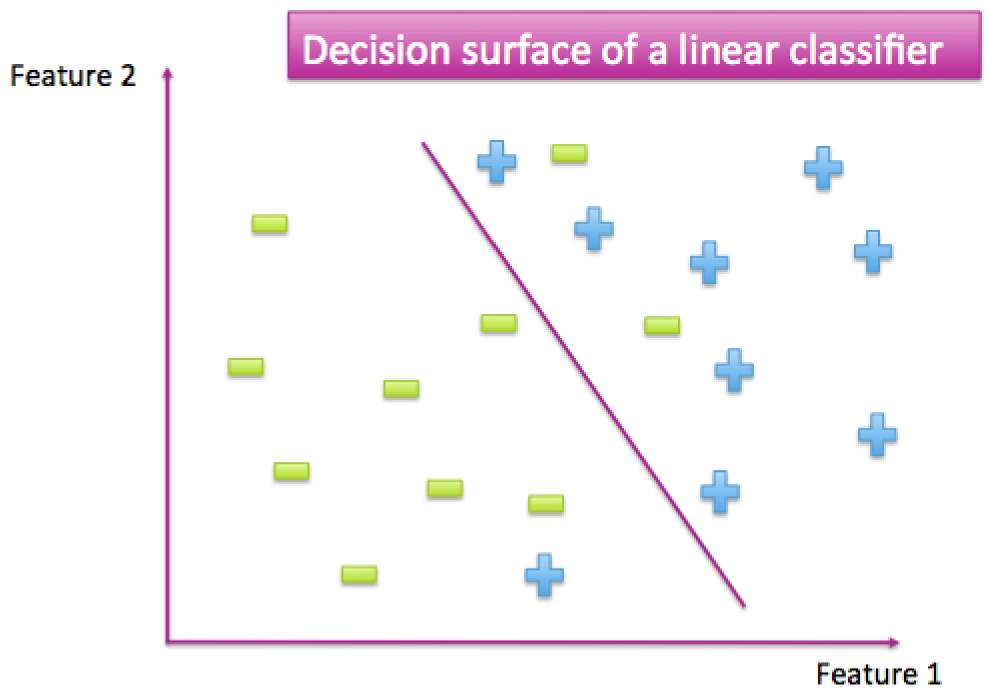


Figure A-1. Simple binary classification finds a surface that separates two classes of data points

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.

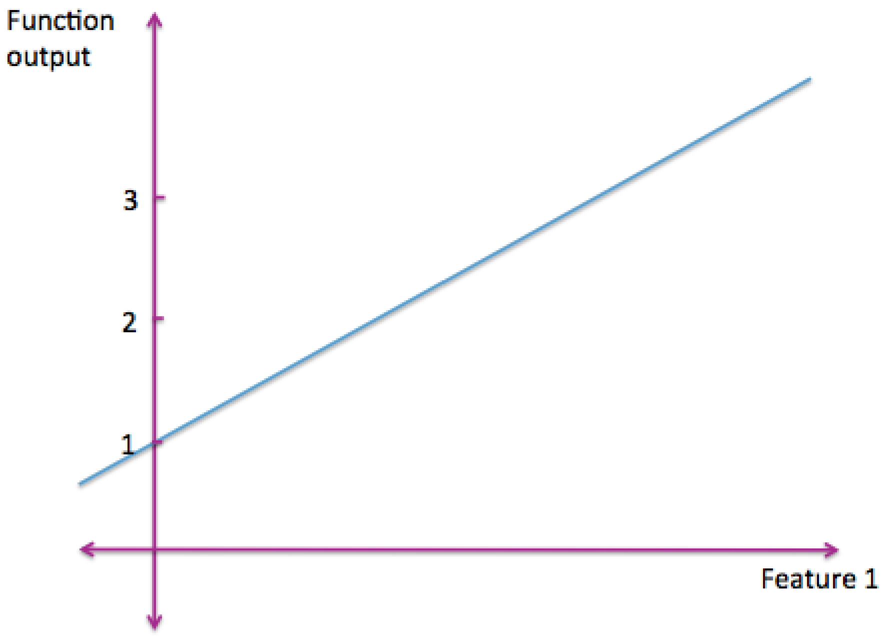


Figure A-2. A linear function of one input variable

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.

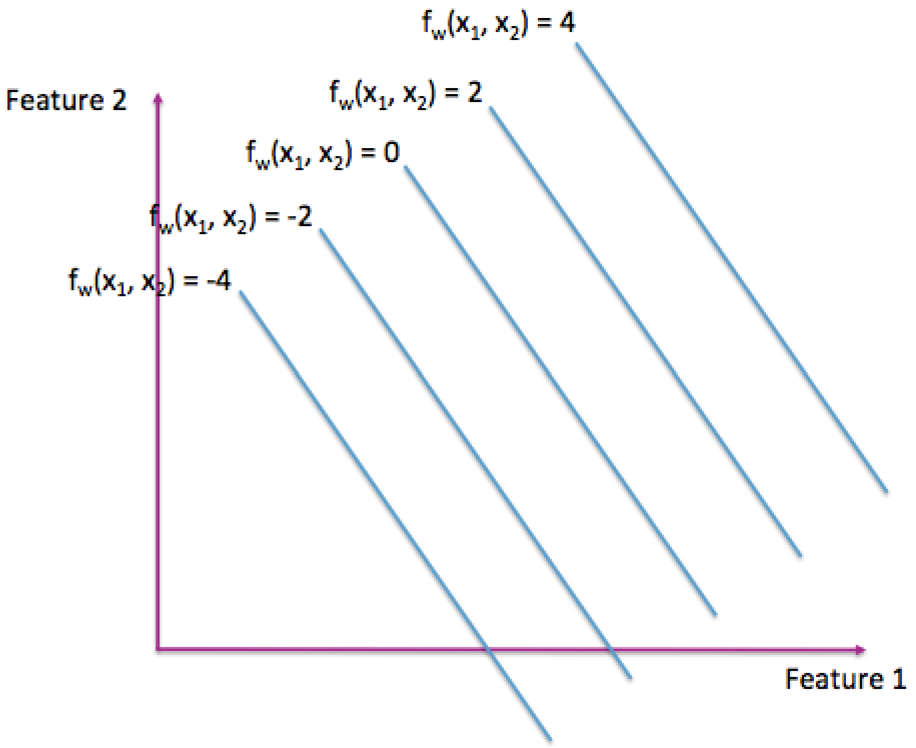


Figure A-3. Contour plot of a linear function in 2D

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 $x_1, x_2, \ldots, x_n$ and a set of weight parameters $w_0, w_1, \ldots, w_n$:

$$f_w(x_1, x_2, \ldots, x_n) = w_0 + w_1 * x_1 + w_2 * x_2 + \ldots + w_n * x_n.$$

It can be written more succinctly using vector notation $f_{\mathbf{w}}(\mathbf{x}) = \mathbf{x}^T\mathbf{w}$. We follow the usual convention for mathematical notations, which uses boldface to indicate a vector and non-boldface to indicate a scalar. The vector $\mathbf{x}$ 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 $\mathbf{w}$ 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[1]

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

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:

Equation A-1. 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.**

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 non-zero 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:

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

Linear combination:

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

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

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

Linear independence:

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

Inner product:

$\langle\mathbf{x}, \mathbf{y}\rangle = x_1y_1 + x_2y_2 + ... + x_ny_n$

Orthogonal vectors:

Two vectors $\mathbf{x}$ and $\mathbf{y}$ 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**v**, where c is a scalar.
3. If it contains two vectors **u** and **v**, then it contains the vector **u** + **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 **v** and a scalar λ such that $A\mathbf{v} = \lambda\mathbf{v}$, then **v** is an *eigenvector* and λ an *eigenvalue* of A.

Let A be a rectangular matrix. If there are vectors **u** and **v** and a scalar σ such that $A\mathbf{v} = \sigma\mathbf{u}$ and $A^T\mathbf{u} = \sigma\mathbf{v}$, then **u** and **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 **s** where **s** = A**w** as we vary the weight vector **w**.

Mathematical interpretation:

All possible linear combinations of columns.

Data interpretation:

All outcomes that are linearly predictable based on observed features. The vector **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 **r** where **r** = $\mathbf{u}^T A$ as we vary the weight vector **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 **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 **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 **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 **w** has an inner product of 0 with every row vector in *A*. Therefore, **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 **w** that maps to the observed output targets **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 **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 **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 $\mathbf{w}_{\text{sad-trumpet}}$?

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

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:

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

$\mathbf{w}_{\text{particular}}$ 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.

[1] 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.

[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.