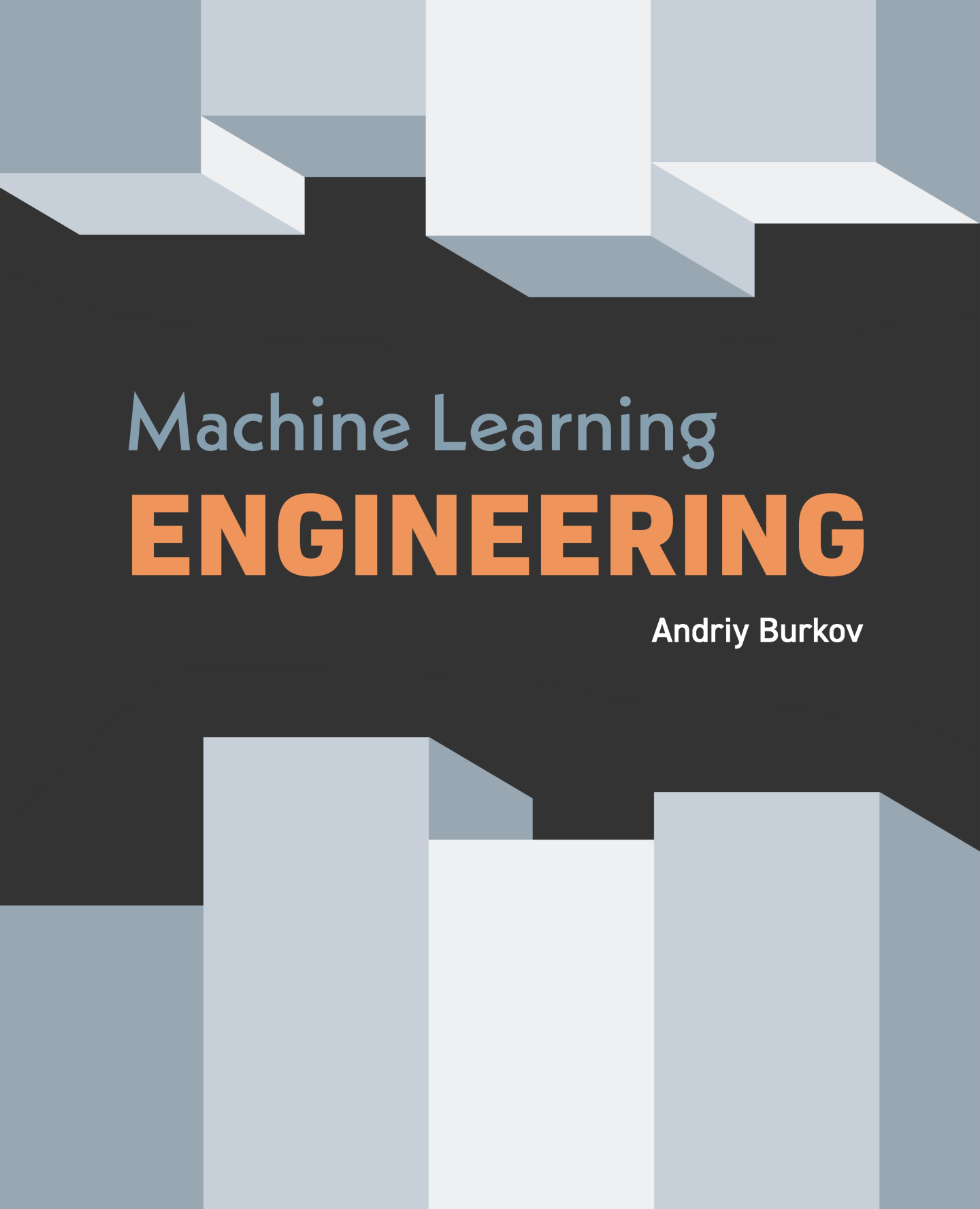

Machine Learning
ENGINEERING
Andriy Burkov

“In theory, there is no difference between theory and practice. But in practice, there is.”

— Benjamin Brewster

“The perfect project plan is possible if one first documents a list of all the unknowns.”

— Bill Langley

“When you’re fundraising, it’s AI. When you’re hiring, it’s ML. When you’re implementing, it’s linear regression. When you’re debugging, it’s printf().”

— Baron Schwartz

The book is distributed on the “read first, buy later” principle.

4 Feature Engineering

After data collection and preparation, feature engineering is the second most important activity in machine learning. It's also the third stage in the machine learning project life cycle:

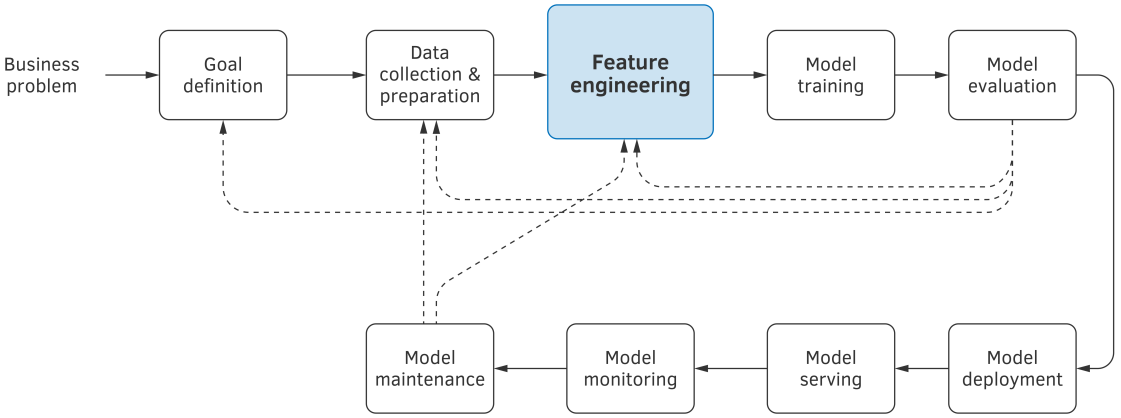


Figure 1: Machine learning project life cycle.

Feature engineering is a process of first conceptually and then programmatically transforming a raw example into a feature vector. It consists of conceptualizing a feature and then writing the programming code that would transform the entire raw example, with potentially the help of some indirect data, into a feature.

4.1 Why Engineer Features

To be more specific, consider the problem of recognizing movie titles in tweets. Say you have a vast collection of movie titles; this is data to use **indirectly**. You also have a collection of tweets; this data will be used **directly** to create examples. First, build an index of movie titles for fast string matching.[1] Then find all movie title matches in your tweets. Now stipulate that your examples are matches, and your machine learning problem is that of binary classification: whether a match is a movie, or is not a movie.

Consider the following tweet:

[1] To build an index for fast string matching, you can, for example, use the **Aho–Corasick algorithm**.

Figure 2: A tweet from Kyle.

Our movie title matching index would help us find the following matches: "avatar," "the terminator," "It," and "her". That gives us four unlabeled examples. You can label those four examples: $\{(\text{avatar}, \text{False}), (\text{the terminator}, \text{True}), (\text{It}, \text{False}), (\text{her}, \text{False})\}$. However, a machine learning algorithm cannot learn anything from the movie title alone (neither can a human): it needs a context. You might decide that the five words preceding the match and the five words following it are a sufficiently informative context. In machine learning jargon, we call such a context a "ten-word window" around a match. You can tune the width of the window as a hyperparameter.

Now, your examples are labeled matches in their context. However, a learning algorithm cannot be applied to such data. Machine learning algorithms can only apply to feature vectors. This is why you resort to feature engineering.

4.2 How to Engineer Features

Feature engineering is a creative process where the analyst applies their imagination, intuition, and domain expertise. In our illustrative problem of movie title recognition in tweets, we used our intuition to fix the width of the window around the match to ten. Now, we need to be even more creative to transform string sequences into numerical vectors.

4.2.1 Feature Engineering for Text

When it comes to text, scientists and engineers often use simple feature engineering tricks. Two such tricks are one-hot encoding and bag-of-words.

Generally speaking, **one-hot encoding** transforms a categorical attribute into several binary ones. Let's say your dataset has an attribute "Color" with possible values "red," "yellow," and "green." We transform each value into a three-dimensional binary vector, as shown below:

$$\begin{aligned} \text{red} &= [1, 0, 0] \\ \text{yellow} &= [0, 1, 0] \\ \text{green} &= [0, 0, 1]. \end{aligned}$$

In a spreadsheet, instead of one column headed with the attribute "Color," you will use three synthetic columns, with the values 1 or 0. The advantage is you now have a vast range of machine learning algorithms at your disposal, for only a handful of learning algorithms support categorical attributes.

Bag-of-words is a generalization of applying the one-hot encoding technique to text data. Instead of representing one attribute as a binary vector, you use this technique to represent an entire text document as a binary vector. Let's see how it works.

Imagine that you have a collection of six text documents, as shown below:

Document 1	Love, love is a verb
Document 2	Love is a doing word
Document 3	Feathers on my breath
Document 4	Gentle impulsion
Document 5	Shakes me, makes me lighter
Document 6	Feathers on my breath

Figure 3: A collection of six documents.

Let your problem be to build a text classifier by topic. A classification learning algorithm expects inputs to be labeled feature vectors, so you have to transform the text document collection into a feature vector collection. Bag-of-words allows you to do just that.

First, tokenize the texts. **Tokenization** is a procedure of splitting a text into pieces called "tokens." A **tokenizer** is software that takes a string as input, and returns a sequence of tokens extracted from that string. Typically, tokens are words, but it's not strictly necessary. It can be a punctuation mark, a word, or, in some cases, a combination of words, such as a company (e.g., McDonald's) or a place (e.g., Red Square). Let's use a simple tokenizer that extracts words and ignores everything else. We obtain the following collection:

Document 1	[Love, love, is a verb]
Document 2	[Love, is, a, doing, word]
Document 3	[Feathers, on, my, breath]
Document 4	[Gentle, impulsion]
Document 5	[Shakes, me, makes, me lighter]
Document 6	[Feathers, on, my, breath]

Figure 4: The collection of tokenized documents.

The next step is to build a vocabulary. It contains 16 tokens:[2]

a	breath	doing	feathers
gentle	impulsion	is	lighter
love	makes	me	my
on	shakes	verb	word

Now order your vocabulary in some way and assign a unique index to each token. I ordered the tokens alphabetically:

a	breath	doing	feathers	gentle	impulsion	is	lighter	love	makes	me	my	on	shakes	verb	word
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 5: Ordered and indexed tokens.

Each token in the vocabulary has a unique index, from 1 to 16. We transform our collection into a collection of binary feature vectors, as shown below:

[2]I decided to ignore capitalization, but you, as an analyst, might choose to treat the two tokens "Love" and "love" as two separate vocabulary entities.

	a						...								word
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Document 1	1	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0
Document 2	1	0	1	0	0	0	1	0	1	0	0	0	0	0	0	1
Document 3	0	1	0	1	0	0	0	0	0	0	0	1	1	0	0	0
Document 4	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
Document 5	0	0	0	0	0	0	0	1	0	1	1	0	0	1	0	0
Document 6	0	1	0	1	0	0	0	0	0	0	0	1	1	0	0	0

Figure 6: Feature vectors.

The 1 is in a specific position if the corresponding token is present in the text. Otherwise, the feature at that position has a 0.

For instance, document 1 "Love, love is a verb" is represented by the following feature vector:

$$[1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0]$$

Use the corresponding labeled feature vectors as the training data, which any classification learning algorithm can work with.

There are several bag-of-words "flavors." The above binary-value model often works well. Alternatives to binary values include 1) counts of tokens, 2) frequencies of tokens, or 3) **TF-IDF** (term frequency-inverse document frequency). If you use the counts of words, then the feature value for "love" in Document 1 "Love, love is a verb" would be 2, representing the number of times the word "love" appears in the document. If applying frequencies of tokens, the value for "love" would be $2/5 = 0.4$, assuming that the tokenizer extracted two "love" tokens, and five total tokens from Document 1. The TF–IDF value increases proportionally to the frequency of a word in the document and is offset by the number of documents in the corpus that contain that word. This adjusts for some words, such as prepositions and pronouns, appearing more frequently in general. I will not go into further detail on TF-IDF, but would recommend the interested reader to learn more about it online.

A straightforward extension of the bag-of-words technique is **bag-of-n-grams**. An **n-gram** is a sequence of n words taken from the corpus. If $n = 2$, and you ignore the punctuation, then all two-grams (usually called **bigrams**) that can be found in the text "No, I am your father." are as follows: ["No I," "I am," "am your," "your father"]. The three-grams are ["No

I am," "I am your," "am your father"]. By mixing all n-grams, up to a certain n, with tokens in one dictionary, we obtain a bag of n-grams that we can tokenize the same way as we deal with a bag-of-words model.

Because sequences of words are often less common than individual words, using n-grams creates a more **sparse** feature vector. At the same time, n-grams allow the machine learning algorithm to learn a more nuanced model. For example, the expressions "this movie was not good and boring" and "this movie was good and not boring" have opposite meaning, but would result in the same bag-of-words vectors, based solely on words. If we consider bigrams of words, then bag-of-words vectors of bigrams for those two expressions would be different.

4.2.2 Why Bag-of-Words Works

Feature vectors only work when certain rules are followed. One rule is that a feature at position j in a feature vector must represent the same property in all examples in the dataset. If that feature represents the height in cm of a certain person in a dataset, where each example represents a different person, then that must hold true in all other examples. The feature at position j must always represent the height in cm, and nothing else.

The bag-of-words technique works the same way. Each feature represents the same property of a document: whether a specific token is present or absent in a document.

Another rule is that similar feature vectors must represent similar entities in the dataset. This property is also respected when using the bag-of-words technique. Two identical documents will have identical feature vectors. Likewise, two texts regarding the same topic will have higher chances to have similar feature vectors, because they will share more words than those of two different topics.

4.2.3 Converting Categorical Features to Numbers

One-hot encoding is not the only way to convert categorical features to numbers, and it's not always the best way.

Mean encoding, also known as **bin counting** or **feature calibration**, is another technique. First, the **sample mean** of the label is calculated using all examples where the feature has value z. Each value z of the categorical feature is then replaced by that sample mean value. The advantage of this technique is that the data dimensionality doesn't increase, and by design, the numerical value contains some information about the label.

If you work on a binary classification problem, in addition to sample mean, you can use other useful quantities: the raw counts of the positive class for a given value of z, the **odds ratio**, and the **log-odds ratio**. The odds ratio (OR) is usually defined between two random variables. In a general sense, OR is a statistic that quantifies the strength of the association between two events A and B. Two events are considered independent if the OR equals 1, that is, the odds of one event are the same in either the presence or absence of the other event.

In application to quantifying a categorical feature, we can calculate the odds ratio between the value z of a categorical feature (event A) and the positive label (event B). Let's illustrate that with an example. Let our problem be to predict whether an email message is spam or not spam. Let's assume that we have a labeled dataset of email messages, and we engineered a feature that contains the most frequent word in each email message. Let us find the numerical value that would replace the categorical value "infected" of this feature. We first build the **contingency table** for "infected" and "spam":

	Spam	Not Spam	Total
contains "infected"	145	8	153
doesn't contain "infected"	346	2909	3255
Total	491	2917	3408

Figure 7: Contingency table for "infected" and "spam."

The odds-ratio of "infected" and "spam" is given by:

$$\text{odds ratio}(\text{infected}, \text{spam}) = \frac{145/8}{346/2909} = 152.4.$$

As you can see, the odds ratio, depending on the values in the contingency table, can be extremely low (near zero) or extremely high (an arbitrarily high positive value). To avoid numerical overflow issues, analysts often use the log-odds ratio:

$$\begin{aligned}\text{log odds ratio}(\text{infected}, \text{spam}) &= \log(145/8) - \log(346/2909) \\ &= \log(145) - \log(8) - \log(346) + \log(2909) = 2.2.\end{aligned}$$

Now you can replace the value "infected" in the above categorical feature with the value of 2.2. You can proceed the same way for other values of that categorical feature and convert all of them into log-odds ratio values.

Sometimes, categorical features are ordered, but not cyclical. Examples include school marks (from "A" to "E") and seniority levels ("junior," "mid-level," "senior"). Instead of using one-hot encoding, it's convenient to represent them with meaningful numbers. Use uniform numbers in the $[0, 1]$ range, like $1/3$ for "junior", $2/3$ for "mid-level" and 1 for "senior." If some values should be farther apart, you can reflect that with different ratios. If "senior" should be farther from "mid-level" than "mid-level" from "junior," you might use $1/5$, $2/5$, 1 for "junior," "mid-level," and "senior," respectively. This is why domain knowledge is important.

When categorical features are cyclical, integer encoding does not work well. For example, try converting Monday through Sunday to the integers 1 through 7. The difference between Sunday and Saturday is 1, while the difference between Monday and Sunday is -6. However, our reasoning suggests the same difference of 1, because Monday is just one day past Sunday.

Instead, use the **sine-cosine transformation**. It converts a cyclical feature into two synthetic features. Let p denote the integer value of our cyclical feature. Replace the value p of the cyclical feature with the following two values:

$$p_{sin} = \sin\left(\frac{2 \times \pi \times p}{\max(p)}\right), p_{cos} = \cos\left(\frac{2 \times \pi \times p}{\max(p)}\right).$$

The table below contains the values of p_{sin} and p_{cos} for the seven days of the week:

p	p_{sin}	p_{cos}
1	0.78	0.62
2	0.97	-0.22
3	0.43	-0.9
4	-0.43	-0.9
5	-0.97	-0.22
6	-0.78	0.62
7	0	1

Figure 8 contains the scatter plot built using the above table. You can see the cyclical nature of the two new features.

Now, in your tidy data, replace "Monday" with two values $[0.78, 0.62]$, "Tuesday" with $[0.97, -0.22]$, and so on. The dataset has added another dimension, but the model's predictive quality is significantly better, compared to integer encoding.

4.2.4 Feature Hashing

Feature hashing, or **hashing trick**, converts text data, or categorical attributes with many values, into a feature vector of arbitrary dimensionality. One-hot encoding and bag-of-words have a drawback: many unique values will create high-dimensional feature vectors. For example, if there are one million unique tokens in a collection of text documents, bag-of-words will produce feature vectors that each have a dimensionality of one million. Working with such high-dimensional data might be very computationally expensive.

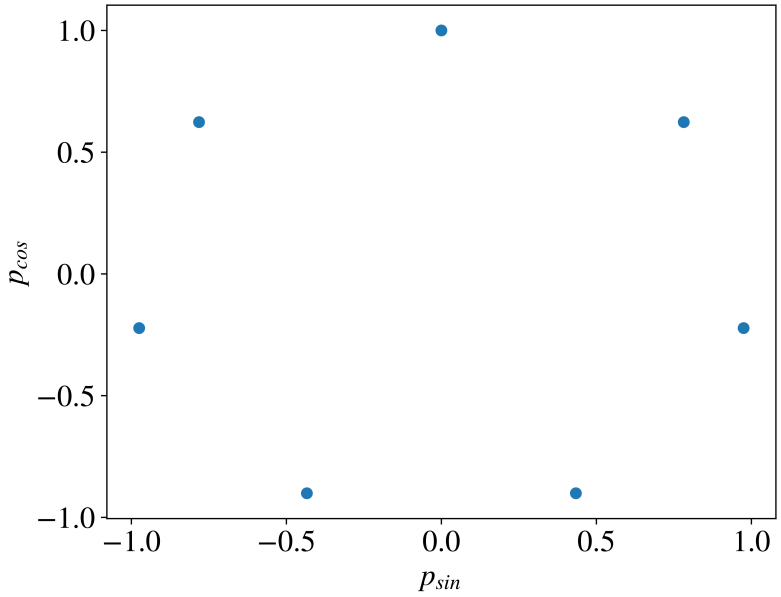


Figure 8: The sine-cosine transformed feature that represents the days of the week.

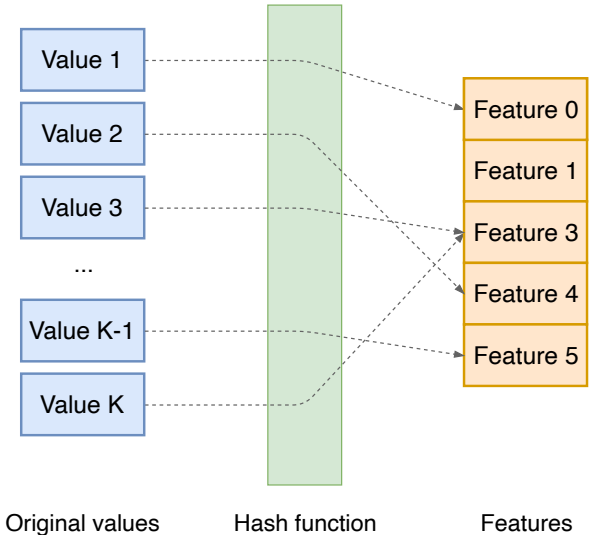


Figure 9: An illustration of the hashing trick for the desired dimensionality of 5 for the original cardinality K of values of an attribute.

To keep your data manageable, you can use the hashing trick that works as follows. First you decide on the desired dimensionality of your feature vectors. Then, using a **hash function**, you first convert all values of your categorical attribute (or all tokens in your collection of documents) into a number, and then you convert this number into an index of your feature vector. The process is illustrated in Figure 9.

Let's illustrate how it would work for converting a text "Love is a doing word" into a feature vector. Let us have a hash function h that takes a string as input and outputs a non-negative integer, and let the desired dimensionality be 5. By applying the hash function to each word and applying the modulo of 5 to obtain the index of the word, we get:

$$\begin{aligned} \text{h(love) mod } 5 &= 0 \\ \text{h(is) mod } 5 &= 3 \\ \text{h(a) mod } 5 &= 1 \\ \text{h(doing) mod } 5 &= 3 \\ \text{h(word) mod } 5 &= 4. \end{aligned}$$

Then we build the feature vector as,

$$[1, 1, 0, 2, 1].$$

Indeed, h(love) mod $5 = 0$ means that we have one word in dimension 0 of the feature vector; h(is) mod $5 = 3$ and h(doing) mod $5 = 3$ means that we have two words in dimension 3 of the feature vector, and so on. As you can see, there is a **collision** between words "is" and "doing": they both are represented by dimension 3. The lower the desired dimensionality, the higher are the chances of collision. This is the trade-off between speed and quality of learning.

Commonly used hash functions are **MurmurHash3**, **Jenkins**, **CityHash**, and **MD5**.

4.2.5 Topic Modeling

Topic modeling is a family of techniques that uses unlabeled data, typically in the form of natural language text documents. The model learns to represent a document as a vector of topics. For example, in a collection of news articles, the five major topics could be "sports," "politics," "entertainment," "finance," and "technology". Then, each document could be represented as a five-dimensional feature vector, one dimension per topic:

$$[0.04, 0.5, 0.1, 0.3, 0.06]$$

The above feature vector represents a document that mixes two major topics: politics (with a weight of 0.5) and finance (with a weight of 0.3). Topic modeling algorithms, such as **Latent**

Semantic Analysis (LSA) and **Latent Dirichlet Allocation** (LDA), learn by analyzing the unlabeled documents. These two algorithms produce similar outputs, but are based on different mathematical models. LSA uses **singular value decomposition** (SVD) of the word-to-document matrix (constructed using a binary **bag-of-words** or **TF-IDF**). LDA uses a hierarchical **Bayesian model**, in which each document is a **mixture** of several topics, and each word's presence is attributable to one of the topics.

Let us illustrate how it works in Python and R. Below is a Python code for LSA:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD

class LSA():
    def __init__(self, docs):
        # Convert documents to TF-IDF vectors
        self.TF_IDF = TfidfVectorizer()
        self.TF_IDF.fit(docs)
        vectors = self.TF_IDF.transform(docs)

        # Build the LSA topic model
        self.LSA_model = TruncatedSVD(n_components=50)
        self.LSA_model.fit(vectors)
        return

    def get_features(self, new_docs):
        # Get topic-based features for new documents
        new_vectors = self.TF_IDF.transform(new_docs)
        return self.LSA_model.transform(new_vectors)

# Later, in production, instantiate LSA model
docs = ["This is a text.", "This another one."]
LSA_featurizer = LSA(docs)

# Get topic-based features for new_docs
new_docs = ["This is a third text.", "This is a fourth one."]
LSA_features = LSA_featurizer.get_features(new_docs)
```

The corresponding code[3] in R is shown below:

```
library(tm)
library(lsa)

get_features <- function(LSA_model, new_docs){
    # new_docs can be passed as a tm::Corpus object or as a vector
```

[3]The R code for LSA and LDA is courtesy of Julian Amon.

```
    # holding character strings representing documents:
    if(!inherits(new_docs, "Corpus")) new_docs <- VCorpus(VectorSource(new_docs))
    tdm_test <- TermDocumentMatrix(
        new_docs,
        control = list(
            dictionary = rownames(LSA_model$tk),
            weighting = weightTfIdf
        )
    )
    txt_mat <- as.textmatrix(as.matrix(tdm_test))
    crossprod(t(crossprod(txt_mat, LSA_model$tk)), diag(1/LSA_model$sk))
}

# Train LSA model using docs
docs <- c("This is a text.", "This another one.")
corpus <- VCorpus(VectorSource(docs))
tdm_train <- TermDocumentMatrix(
corpus, control = list(weighting = weightTfIdf))
txt_mat <- as.textmatrix(as.matrix(tdm_train))
LSA_fit <- lsa(txt_mat, dims = 2)

# Later, in production, get topic-based features for new_docs
new_docs <- c("This is a third text.", "This is a fourth one.")
LSA_features <- get_features(LSA_fit, new_docs)
```

Below is a Python code for LDA:

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import LatentDirichletAllocation

class LDA():
    def __init__(self, docs):
        # Convert documents to TF-IDF vectors
        self.TF = CountVectorizer()
        self.TF.fit(docs)
        vectors = self.TF.transform(docs)
        # Build the LDA topic model
        self.LDA_model = LatentDirichletAllocation(n_components=50)
        self.LDA_model.fit(vectors)
        return
    def get_features(self, new_docs):
        # Get topic-based features for new documents
        new_vectors = self.TF.transform(new_docs)
        return self.LDA_model.transform(new_vectors)

```

```
# Later, in production, instantiate LDA model
docs = ["This is a text.", "This another one."]
LDA_featurizer = LDA(docs)

# Get topic-based features for new_docs
new_docs = ["This is a third text.", "This is a fourth one."]
LDA_features = LDA_featurizer.get_features(new_docs)
```

And here is the corresponding code in R:

```
library(tm)
library(topicmodels)

# Generate feature for new_docs by using LDA_model
get_features <- function(LDA_mode, new_docs){
  # new_docs can be passed as tm::Corpus object or as a vector
  # holding character strings representing documents:
  if(!inherits(new_docs, "Corpus")) new_docs <- VCorpus(VectorSource(new_docs))
  new_dtm <- DocumentTermMatrix(new_docs, control = list(weighting = weightTf))
  posterior(LDA_mode, newdata = new_dtm)$topics
}

# train LDA model using docs
docs <- c("This is a text.", "This another one.")
corpus <- VCorpus(VectorSource(docs))
dtm <- DocumentTermMatrix(corpus, control = list(weighting = weightTf))
LDA_fit <- LDA(dtm, k = 5)

# later, in production, get topic-based features for new_docs
new_docs <- c("This is a third text.", "This is a fourth one.")
LDA_features <- get_features(LDA_fit, new_docs)
```

In the above listings, docs is a collection of text documents. It can, for example, be a list of strings, where each string is a document.

4.2.6 Features for Time-Series

Time-series data is different from the traditional supervised learning data, which has a form of unordered collections of independent observations. A time series is an ordered sequence of observations, and each is marked with a time-related attribute, such as timestamp, date, month-year, year, and so on. An example of a time-series data is given in Figure 10.

Date	Stock Price	S&P 500	Dow Jones
2020-01-11	...	...	...
2020-01-12	14.5	3,345	28,583
2020-01-12	14.7	3,352	28,611
2020-01-12	15.9	3,347	29,001
2020-01-13	17.9	3,298	28,312
2016-01-13	16.8	3,521	28,127
2020-01-14	17.9	3,687	28,564
2016-01-15	16.8	3,540	27,998
2016-01-16	...	...	...

Figure 10: An example of time-series data in the form of an event stream.

Date	Stock Price	S&P 500	Dow Jones
2020-01-11	...	...	...
2020-01-12	15.0	3,348	28,732
2020-01-13	17.4	3,410	28,220
2020-01-14	17.9	3,687	28,564
2016-01-15	16.8	3,540	27,998
2016-01-16	...	...	...

Figure 11: Classical time series obtained by aggregating the event stream from Figure 10.

In Figure 10, each row corresponds to the cost of a certain stock at a moment in time, as well as the values of two indices: S&P 500 and Dow Jones. The observations were made irregularly: on 2020-01-12, three observations were made. On 2020-01-13, there were two observations. In the **classical time-series data**, observations are evenly spaced over time,

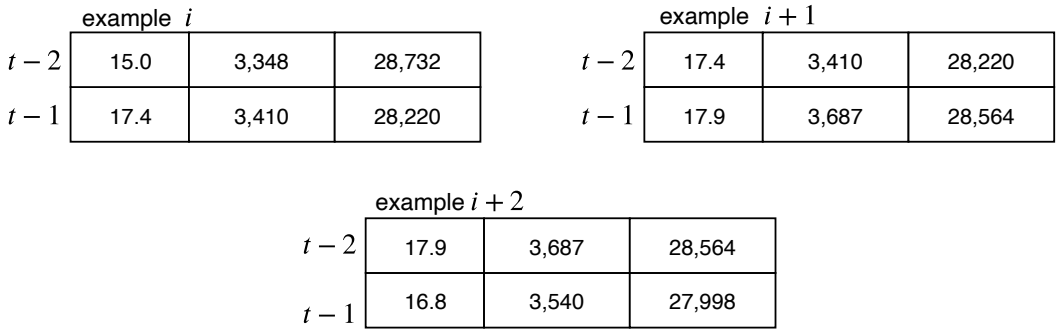


Figure 12: Time-series chunked into segments of length $w = 2$.

such as one observation per second, per minute, per day, and so on. If observations are irregular, such time-series data is called a **point process** or an **event stream**.

It's usually possible to convert an event stream into the classical time-series data by aggregating observations. Examples of aggregation operators are COUNT and AVERAGE. By applying the AVERAGE operator to the event stream data in Figure 10, we obtain the classical time-series data shown in Figure 11.

While it's possible to directly work with event streams, bringing time series to the classical form makes it simpler to apply further aggregations and generate features for machine learning.

Analysts typically use time-series data to solve two kinds of prediction problems. Given a sequence of recent observations:

- predict something about the next observation (for example, given the stock price and the value of stock indices for the last seven days, predict the stock price for tomorrow), or
- predict something about the phenomenon that generated that sequence (for example, given a user's connection log to a software system, predict whether they are likely to cancel their subscription during the current quarter).

Before neural networks reached their modern learning capacity, analysts worked with time-series data using the **shallow machine learning** toolkit. To transform a time-series into training data in the form of feature vectors, two decisions must be made:

- how many of the consecutive observations are needed to make an accurate prediction (so-called prediction window), and
- how to convert a sequence of observations into a fixed-dimensionality feature vector.

There's no simple way to answer either question. Usually decisions are made based on the subject-matter expert's knowledge, or by using a **hyperparameter tuning** technique. However, some recipes work for many time-series data. Below is one such recipe:

1) chunk the entire time series into segments of length w,

2) create a training example e from each segment s,
3) for each e, calculate various statistics on the observations in s.

We take Figure 11's data and chunk it into segments of length $w = 2$, where w the length of the prediction window. Figure 12 shows that each segment is now a separate example.

In practice, w is usually larger than 2. Let's say our prediction window has a length of seven. The statistics calculated at step (3) of the above recipe could be:

- average (e.g., the **mean** or **median** of the stock price during the last seven days),
- spread (e.g., **standard deviation**, **median absolute deviation**, or **interquartile range** of the values of the S&P 500 index during the last seven days),
- outliers (e.g., the fraction of observations, in which the values of the Dow Jones index was atypically low; for example, more than two standard deviations from the mean),
- growth (e.g., whether the values of the S&P 500 index have grown between the day $t - 6$ and t, days $t - 3$ and t, and between $t - 1$ and t.
- visual (e.g., how different the curve of the stock price values is from a known visual image, such as a hat, or head and shoulders).

Now you see why converting a time series into a classical form is recommended: the above statistics are only meaningful when calculated on the comparable values.

It should be noted that in the modern neural-network era, analysts most often prefer to train deep neural networks. **Long short-term memory** (LSTM), **convolutional neural network** (CNN), and **Transformer** are popular choices of architecture for a time-series model. These can read arbitrary length time-series as input, and generate a prediction based on the entire sequence. Similarly, neural networks are often applied to texts by reading them word-by-word, or character-by-character. Words and characters are usually represented as **embedding vectors**; the latter are learned from large corpora of text documents. We will talk about embeddings in Section 4.7.1.

4.2.7 Use Your Creativity

As I mentioned at the beginning of this section, feature engineering is a creative process. As an analyst, you are in the best position to determine what are good features for your prediction model. Put yourself "in the shoes" of a learning algorithm and imagine what you would look at in your data to decide which label to assign.

Say you are classifying emails as important or unimportant. You might notice that a significant number of important messages come from the government revenue agency on the first Monday of each month. Create a feature "government first monday." Let it equal 1 when the email came from the government revenue agency on the first Monday of a month, and 0 otherwise. Alternatively, you might notice that an email with more than one smiley is rarely important. Create a feature "contains smileys." Let it equal 1 when an email contains more than one smiley, and 0 otherwise.

4.3 Stacking Features

Back to our problem of movie title classification in tweets. Each example has three parts:

1) five words[4] that precede the extracted potential movie title (the left context),
2) the extracted potential movie title (the extraction),
3) five words that follow the extracted movie title (the right context).

To represent such multi-part examples, we first transform each part into a feature vector, and then stack the three feature vectors next to one another to obtain the feature vector for the entire example.

4.3.1 Stacking Feature Vectors

In our movie title classification problem, we first collect all the left contexts. We then apply bag-of-words to transform each left context into a binary feature vector. Next, collect all extractions and, using bag-of-words, transform each extraction into a binary feature vector. Then we collect all the right contexts and apply bag-of-words to transform each right context into a binary feature vector. Finally, we concatenate each example, joining the feature vectors of the left context, the extraction, and the right context. We obtain the final feature vector that represents the entire example, as shown in Figure 13.

Note that the three feature vectors (one from each part of the example) are created independently of one another. This means that the vocabulary of tokens is different for each part and, therefore, the feature vector dimensionality of each part may also be different.

The order in which you concatenate feature vectors doesn't matter. The left context features can be placed in the middle or right side of the final feature vector. However, you must keep the same concatenation order in all examples. This ensures each feature represents the same property from one example to another.

4.3.2 Stacking Individual Features

Until now, we engineered features in bulk. One-hot encoding and bag-of-words often generate thousands of features. This is a very time-efficient way of engineering features, but some problems require more to obtain feature vectors with high enough **predictive power**. We consider the predictive power of a feature in the next section.

Imagine that you already have a classifier m_A that takes an entire tweet as input and predicts its topic. Let one of the topics be cinema. You might want to enrich the feature vectors in your movie title classification problem with this additional information available from the classifier m_A. In this case, you will engineer one feature that can be described as "whether

[4] In practice, the context to the left or the right of the potential movie title can for some examples be shorter than five words, because it's either the beginning or the end of the tweet.

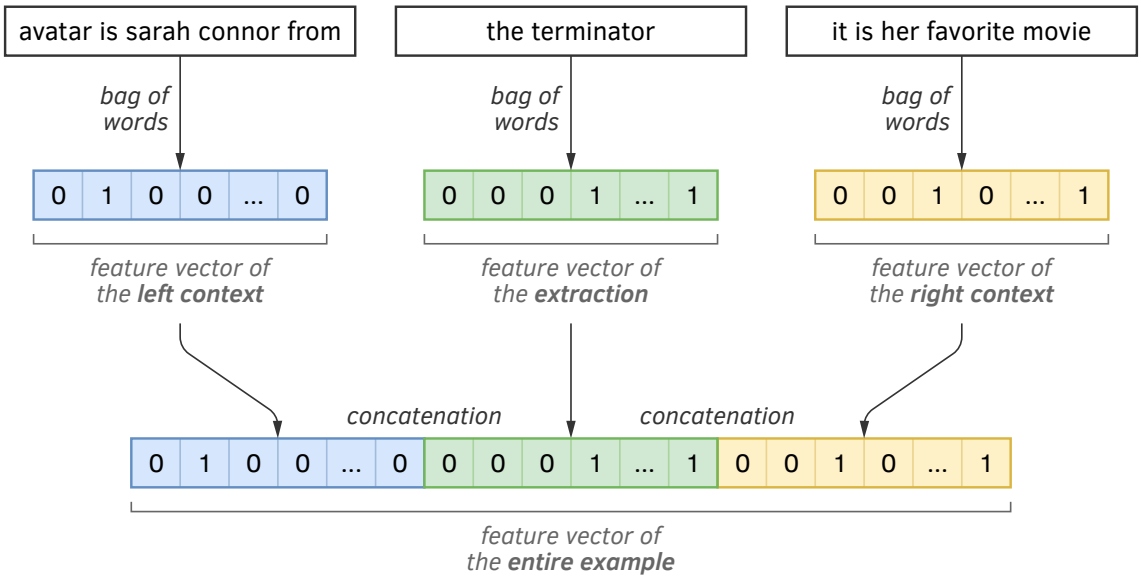


Figure 13: Creating and stacking feature vectors.

the topic of the tweet is cinema" and that feature will also be binary: 1 if the topic predicted by m_A for the entire tweet is cinema, and 0 otherwise. Again, we concatenate the three partial feature vectors, as shown in Figure 14.

You might come up with many more useful features for title classification in tweets. Examples of such features are:

- the average IMDB score of the movie,
- the number of votes for the movie on IMDB,
- the Rotten Tomato score of the movie,
- whether the movie is recent (or the number that represents the release year),
- whether the tweet text contains other movie titles, and
- whether the tweet text includes the names of actors or directors.

All these additional features, as long as they are numerical, can be concatenated to the feature vector. The only condition is that they are concatenated in the same order in all examples.

4.4 Properties of Good Features

Not all features are created equal. In this section, we consider the properties of a good feature.

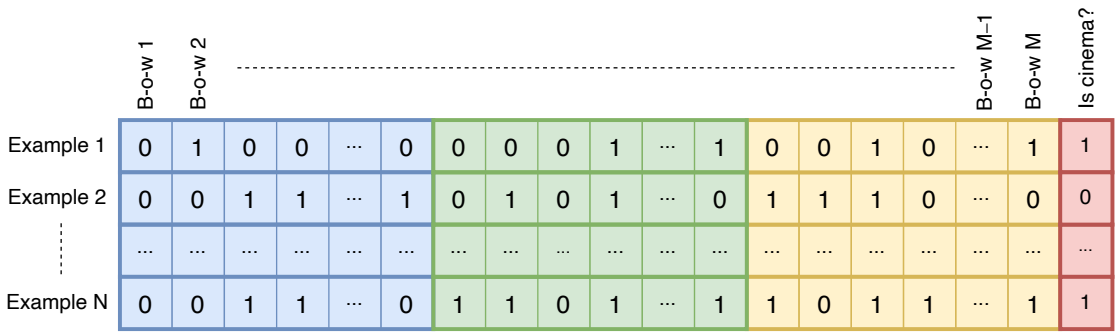


Figure 14: Single feature stacking.

4.4.1 High Predictive Power

First of all, a good feature has high **predictive power**. In Chapter 3, you read about predictive power as a property of data. However, a feature can also have high or low predictive power. Let's say you want to predict whether a patient has cancer. Among other features, you know the make of the person's car and whether the person is married. These two features are not good predictors for cancer, so our machine learning algorithm will not learn a meaningful relationship between these features and the label. Predictive power is a property of the feature with respect to the problem. The make of the person's car and whether the person is married could have high predictive power if the problem were different.

4.4.2 Fast Computability

Good features can be computed fast. Let's say you want to predict the topic of a tweet. A tweet is short, and a bag-of-words-based feature vector will be sparse. A **sparse vector** is a vector whose values in most dimensions are zero. If your dataset is small and the texts are short, the learning algorithm will have a hard time seeing patterns in sparse vectors because they contain little information compared to their size. The information in one sparse vector is rarely contained in the same dimensions as the information in another sparse vector, even if they represent similar concepts.

To reduce sparsity, you might want to augment your sparse feature vectors with additional non-zero values. To do that, you might send the tweet text to Wikipedia as a search query, and then extract other words from the search results. Wikipedia's API doesn't give any guarantee for the speed of response, so it could take several seconds to get a response. For real-time systems, feature extraction must be fast: a less informative feature computed in a fraction of a millisecond is often preferred to a feature with a high predictive power that takes seconds to compute. If your application must be fast, the features obtained from Wikipedia might not be appropriate for your task.

4.4.3 Reliability

A good feature must also be reliable. Again, in our Wikipedia example, we cannot have a guarantee that the website will respond at all: it can be down, on planned maintenance, or the API may be temporarily overused and is rejecting requests. Therefore, we cannot trust that Wikipedia-based features will always be available and complete. Thus, we cannot call such features reliable. One unreliable feature can reduce the quality of predictions made by your model. Furthermore, some predictions can become entirely wrong if the value of an important feature is missing.

4.4.4 Uncorrelatedness

Correlation of two features means their values are related. If the growth of one feature implies the growth of the other, and the inverse is also true, then the two features are correlated.

Once the model is in production, its performance may change because the input data's properties may change over time. When many of your features are highly correlated, even a minor change in the input data's properties may result in significant changes in the model's behavior.

Sometimes the model was built under strict time constraints, so the developer used all possible sources of features. With time, maintaining those sources can become costly. It's generally recommended to eliminate redundant or highly correlated features. Feature selection techniques help reduce such features.

4.4.5 Other Properties

An essential property of a good feature is that the distribution of its values in the training set is similar to the distribution it will receive in production. For example, a tweet's date might be necessary for some predictions about it. However, if you apply the model built on historical tweets to predict something about current tweets, the date of your production examples will always be out of the training distribution, which can result in a significant error.[5]

Finally, features that you design should be unitary, easy to understand, and maintain. Unitary means the feature represents a certain simple-to-understand and -explain quantity. For example, when classifying a car's type given its characteristics, you may use such unitary features as weight, length, width, and color. A feature like "length divided by weight" is not unitary, as it's composed of two unitary features.

[5]The date information often is relevant for machine learning and can still be included in the training data. For instance, you could consider engineering **cyclical features** like "hour of the day," "day of the week," "month of the year." For the prediction problems in which time seasonality has predictive power, having such features can be useful.

Some learning algorithms may benefit from combining features. However, it's preferable to do this in a dedicated stage in the model training pipeline. We will consider feature combination and generation of synthetic features later in this chapter.

4.5 Feature Selection

Not all features will be equally important for your problem. For instance, in the problem of detecting movies in tweets, the length of the movie might not be a very important feature. At the same time, when you use bag-of-words, the vocabulary can be very large, while most tokens will appear in the collection of texts only once. If the learning algorithm "sees" that some feature has a non-zero value only in a couple of training examples, it is doubtful the algorithm will learn any useful pattern from that feature. However, if the feature vector is very wide (contains thousands or millions of features), the training time can become prohibitively long. Furthermore, the overall size of the training data can become too large to fit in the RAM of a conventional server.

If we could estimate the importance of features, we would keep only the most important ones. That would allow us to save time, fit more examples in memory, and improve the model's quality. Below, we consider some feature selection techniques.

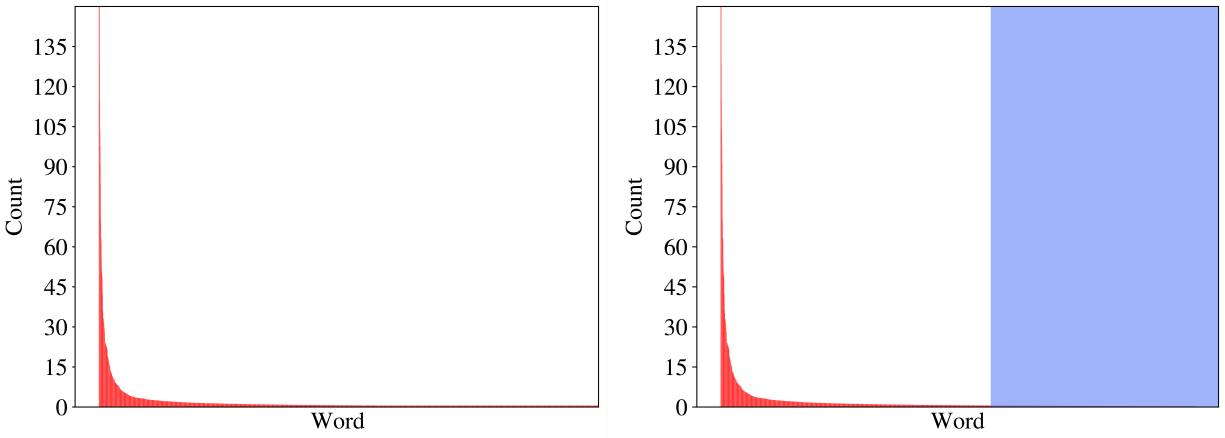
4.5.1 Cutting the Long Tail

Typically, if a feature contains information (e.g., a non-zero value) only for a handful of examples, such a feature could be removed from the feature vector. In **bag-of-words**, you can build a graph with the distribution of token counts, and then cut off the so-called long tail, as shown in Figure 15.

A **long tail** of a distribution is such a part of that distribution that contains elements with substantially lower counts compared to a smaller group of elements with the highest counts. This smaller group is called the head of the distribution, and their aggregated counts make for at least half of all the counts.

The decision on a threshold for defining the long tail is somewhat subjective. You can set it as a hyperparameter for your problem and discover the optimal value experimentally. On the other hand, the decision can be made by looking at the distribution of counts, as shown in Figure 15a. As you can see, I cut off the long tail at a point where the distribution of the elements in the tail has become visually flat (Figure 15b).

Whether to cut the long tail, and where to do it, is debatable. In classification problems with many classes, the difference between some classes can be very subtle. Even features whose values are rarely non-zero may become important. However, removing long-tail features often results in faster learning and a better model.



(a) distribution of word counts in English

(b) the long tail

Figure 15: The distribution of word counts in a collection of texts in English (a) and the long tail (b, zone in blue). The highest count corresponds to "the" (a count of 615); the lowest count corresponds to "zambia" (a count of 1).

4.5.2 Boruta

Cutting the long tail is not the only way to select important features and remove less important ones. One popular tool used in **Kaggle** competitions is **Boruta**. Boruta iteratively trains **random forest** models and runs **statistical tests** to identify features as important and unimportant. The tool exists both in the form of an R package and a Python module.

Boruta works as a wrapper around the random forest learning algorithm, hence its name — Boruta is a spirit of the forests in Slavic mythology. To understand the Boruta algorithm, let's first recall how the random forest learning algorithm works.

Random forest is based on the idea of **bagging**. It makes many random samples of the training set and then trains a different statistical model on each sample. The prediction is then made by taking the majority vote (for classification) or an average (for regression) of all models. The only substantial difference of random forest from the vanilla bagging algorithm is that in the former, the trained statistical models are decision trees. At each split of the decision tree, a random subset of all features is considered.

One useful feature of the random forest is its built-in capability to estimate the importance of each feature. Below, I will explain how this estimation works for the case of classification.

The algorithm works in two stages. First, it classifies all training examples from the original training set. Each decision tree in the random forest model votes only on the classification of examples that weren't used to build that tree. After a tree is tested, the number of correct predictions is recorded for that tree.

At the second stage, the values of a certain feature are randomly permuted across examples, and the tests are repeated. The number of correct predictions is once again recorded for each tree. The importance of the feature for a single tree is then computed as the difference between the number of correct classifications between the original and permuted setting, divided by the number of examples. To obtain the feature importance score, the feature importance measures for individual trees are averaged. While not strictly necessary, it's convenient to use **z-scores** instead of the raw importance scores.

To obtain a z-score for a feature, we first find the average value and the standard deviation of individual feature scores for individual trees. The feature's z-score is obtained by subtracting the average value from the score, and then dividing it by the standard deviation.

You might stop here and use the z-scores of each feature as the criterion to keep it (the higher, the better). However, in practice, the importance score alone often doesn't reflect meaningful correlations between features and the target. Therefore, we need a different tool to distinguish the truly important features from the non-important ones, and, as you could guess, Boruta provides that tool.

The underlying idea of Boruta is simple: we first extend the list of features by adding a randomized copy of each original feature, and then build a classifier based on this extended dataset. To assess the importance of an original feature, we compare it to all randomized features. Only features for which the importance is higher than that of the randomized features — and statistically significant — are considered truly important.

Below, I outline the main steps of the Boruta algorithm in the way it was described by its authors[6] with adaptations for consistency and clarity:

The Boruta Algorithm

- Build extended training feature vectors, where each original feature is replicated. Randomly permute the values of the replicated features across the training examples to remove any correlation between the replicated variables and the target.
- Perform several random forest learning runs. The replicated features are randomized before each run by applying the same random feature value permutation process as in the previous step.
- For each run, compute the importance (z-score) of all original and replicated features.
  - A feature is deemed important for a single run if its importance is higher than the maximal importance among all replicated features.
- Perform a **statistical test** for all original features.

[6]Miron B. Kursa, Aleksander Jankowski, Witold R. Rudnicki, "Boruta - A System for Feature Selection," published in Fundamenta Informaticae 101 in 2010, pages 271–285.

  - The **null hypothesis** is that the feature's importance is equal to the maximal importance of the replicated features (MIRA).
  - The statistical test is a **two-sided equality test** - the hypothesis may be rejected either when the importance of the feature is significantly higher or significantly lower than MIRA.
  - For each original feature, we count and record the number of hits.
  - The number of hits for a feature is the number of runs in which the importance of that feature was higher than MIRA.
  - The expected number of hits for R runs is $E(R) = 0.5R$ with standard deviation $S = \sqrt{0.25R}$ (**binomial distribution** with $p = q = 0.5$).
  - An original feature is deemed important (accepted) when the number of hits is significantly higher than the expected number of hits and is deemed unimportant (rejected) when the number of hits is significantly lower than the expected. (It is possible to compute limits for accepting and rejecting feature for any number of runs for the desired confidence level.)
- Remove the features which are deemed unimportant from the feature vectors (both original and replicated).
- Perform the same procedure for a predefined number of iterations, or until all features are either rejected or conclusively deemed important, whichever comes first.

Boruta worked well for many Kaggle competitions; therefore, you can consider it a universally applicable tool for feature selection. One thing worth noting, though, before using Boruta in production: Boruta is a heuristic. There are no theoretical guarantees for its performance. If you want to be sure that Boruta doesn't harm, run it multiple times and make sure that the feature selection is stable (i.e., consistent across multiple Boruta applications to your data). If the feature selection is not stable, make sure that the number of trees in the random forest is large enough to generate stable results.

Though Boruta is an effective method of feature selection, it's not the only one used by practitioners. You will find the description of several other methods in the book's companion wiki in an extended version of this chapter.

4.5.3 L1-Regularization

Regularization is an umbrella term for a range of techniques that improve the **generalization** of the model. Generalization, in turn, is the model's ability to correctly predict the label for unseen examples.

While regularization doesn't let you identify important features, some regularization techniques, such as L1, allow the machine learning algorithm to learn to ignore some features.

Depending on the kind of model you train, L1 may apply differently, but the main principle remains the same: L1 penalizes the model for being too complex.

In practice, L1 regularization produces a **sparse model**, which is a model that has most of its parameters equal to zero. Therefore, L1 implicitly performs feature selection by deciding which features are essential for prediction, and which ones are not. We will talk about regularization in more detail in the next chapter.

4.5.4 Task-Specific Feature Selection

Feature selection can also be task-specific. For example, we can remove some features from bag-of-words vectors representing natural language texts by excluding the dimensions corresponding to **stop words**. Stop words are the words that are too generic or common for the problem we are trying to solve. Frequent examples of stop words are articles, prepositions, and pronouns. Dictionaries of stop words for most languages are available online.

To further reduce the feature vector dimensionality obtained from the text data, sometimes it's practical to preprocess the text by replacing infrequent words (e.g., those whose count in the corpus is below three) with the same synthetic token, for example RARE_WORD.

4.6 Synthesizing Features

The learning algorithms implemented in the most popular machine learning package for Python, **scikit-learn**, only work with numerical features. But it can still be useful to convert numerical features into categorical ones.

4.6.1 Feature Discretization

The reasons to discretize a real-valued numerical feature can be numerous. For example, some feature selection techniques only apply to categorical features. A successful discretization adds useful information to the learning algorithm when the training dataset is relatively small. Numerous studies show that discretization can lead to improved predictive accuracy. It is also simpler for a human to interpret a model's prediction if it is based on discrete groups of values, such as age groups or salary ranges.

Binning, also known as **bucketing**, is a popular technique that allows transforming a numerical feature into a categorical one by replacing numerical values in a specific range by a constant categorical value.

There are three typical approaches to binning:

- uniform binning,
- k-means-based binning, and
- quantile-based binning.

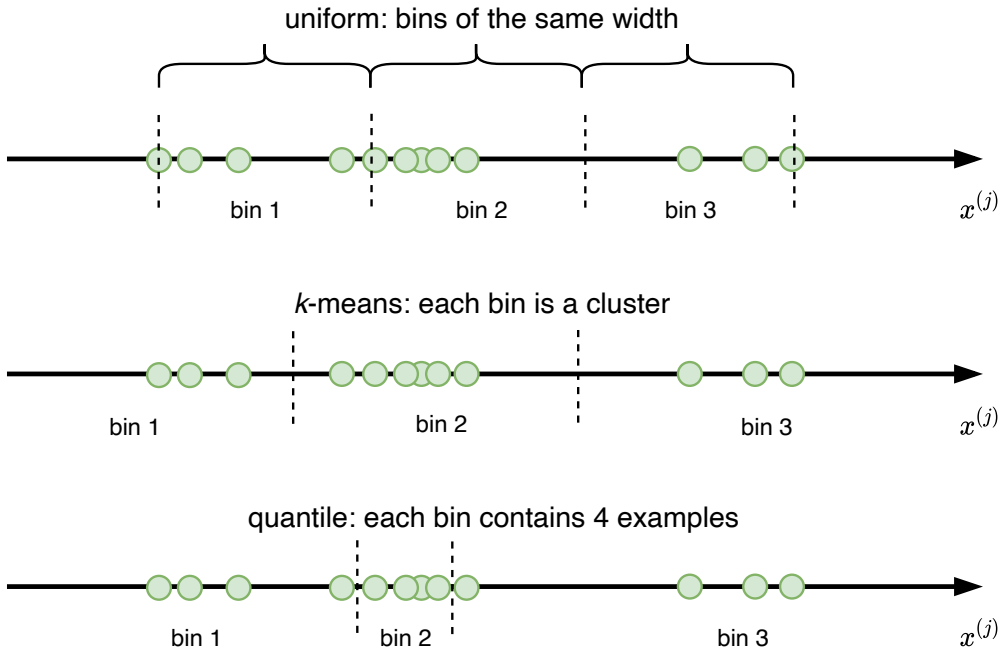


Figure 16: Three binning approaches: uniform, k-means-based, and quantile-based.

In all three cases, you should decide how many bins you want to have. Consider an illustration in Figure 16. Here, we have a numerical feature j and 12 values of this feature, one for each of the 12 examples in our dataset. Let's say we decided to have three bins. In uniform binning, all bins for a feature have identical widths, as illustrated in Figure 16 on the top.

In k-means-based binning, values in each bin belong to the nearest one-dimensional k-means cluster, as shown in Figure 16 in the middle.

In quantile-based binning, all bins have the same number of examples, as shown in Figure 16 at the bottom.

In uniform binning, once the model is deployed in production, if the value of the feature in the input feature vector is below or above the range of any bin, then the closest bin is assigned, which is either the leftmost or the rightmost bin.

Remember, most modern machine learning algorithm implementations require numerical features. The bins must be transformed back to numerical values by using a technique like **one-hot encoding**.

User

User ID	Gender	Age	...	Date Subscribed
1	M	18	...	2016-01-12
2	F	25	...	2017-08-23
3	F	28	...	2019-12-19
4	M	19	...	2019-12-18
5	F	21	...	2016-11-30

Order

Order ID	User ID	Amount	...	Order Date
1	2	23	...	2017-09-13
2	4	18	...	2018-11-23
3	2	7.5	...	2019-12-19
4	2	8.3	...	2016-11-30

Call

Call ID	User ID	Call Duration	...	Call Date
1	4	55	...	2016-01-12
2	2	235	...	2016-01-13
3	3	476	...	2016-12-17
4	4	334	...	2019-12-19
5	4	14	...	2016-11-30

Figure 17: Relational data for churn analysis.

User features

User ID	Gender	Age	Mean Order Amount	Std Dev Order Amount	Mean Call Duration	Std Dev Call Duration
2	F	25	12.9	7.1	235	0
4	M	19	18	0	134.3	142.7

Figure 18: Synthetic features based on sample mean and standard deviation.

4.6.2 Synthesizing Features from Relational Data

Data analysts often work with data in a **relational database**. For example, a mobile phone operator wants to know whether a customer will soon abandon the subscription. This problem is known as **churn analysis**. We have to represent each customer as a vector of features.

Let's say the data on the users is contained in three relational tables: User, Order, and Call, as shown in Figure 17.

The table User already contains two potentially useful features: Gender and Age. We can also create synthetic features using the data from tables Order and Call. As you can see, user 2 has three rows in table Order, while user 4 has one row in table Order, but three rows in table Calls. In order to create a feature that represents one user, we have to reduce those several records into one value. A typical approach is to compute various statistics from the data coming from multiple rows and use the value of each statistic as a feature. The most commonly used statistics are **sample mean** and **standard deviation**. (Standard deviation is the square root of **sample variance**.)

To give a concrete example, I have calculated the values of four features for users 2 and 4. You can find them in Figure 18.

Sometimes, a relational database can have a deeper structure. For example, a user can have orders, while each order can have ordered items. In such a case, we can compute a statistic of a statistic. For example, one feature can be created by first calculating the standard deviation of item prices in each order, and then by taking the average of those standard deviations for a specific user. You can combine the statistics in arbitrary ways: the mean of the mean, the standard deviation of the mean, the standard deviation of the standard deviation, and so on. The same principle applies to the database whose table structure is deeper than two levels.

Once you have generated features based on all possible combinations of statistics, you can select the most useful ones by using one of the feature selection methods.

If you want to increase the predictive power of your feature vectors, or when your training set is rather small, you can synthesize additional features that would help in predictions. There are two typical ways to synthesize additional features: from the data, or from other features.

4.6.3 Synthesizing Features from the Data

One technique commonly used to synthesize one or more additional features is **clustering**. Let us use the **k-means clustering**. Choose a value for k. If your ultimate goal is to build a classification model, a common way to assign a value for k is to use the number C of classes. In regression, use your intuition or apply any technique allowing to determine the right value of clusters in your data, such as **prediction strength** or the **elbow method**. Apply k-means clustering to the feature vectors in your training data. Then add k additional features to your feature vectors. The additional feature $D + j$, where $j = 1, \ldots, k$, will be binary and equal to 1 if the corresponding feature vector belongs to cluster j.

You can synthesize even more features by applying different clustering algorithms, or by restarting k-means multiple times from randomly chosen starting points.

4.6.4 Synthesizing Features from Other Features

Neural networks are notorious for their ability to learn complex features by combining simple features in unordinary ways. They combine simple features by letting their values undergo several levels of nested nonlinear transformations. If you have data in abundance, you can train a deep **multilayer perceptron** model that will learn to cleverly combine the basic unitary features it receives as input.

If you don't have an infinite supply of training examples (often the case in practice), very deep neural networks lose their appeal.[7] In the case of smaller to moderately large datasets (the number of training examples varies between a thousand and a hundred thousand), you might prefer to use a shallow learning algorithm and "help" your learning algorithm learn by providing a richer set of features.

In practice, the most common way to obtain new features from the existing features is to apply a simple transformation to one or a pair of existing features. Three typical simple transformations that apply to a numerical feature j in example i are 1) **discretization** of the feature, 2) squaring the feature, and 3) computing the sample mean and the standard deviation of feature j from k-nearest neighbors of the example i found by using some metric like Euclidean distance or cosine similarity.

Transformations that apply to a pair of numerical features are simple arithmetic operators: $+$, $-$, $\times$, and $\div$ (a technique also known as **feature-crossing**). For example, you can obtain the value of a new feature q in example i, where $q > D$, by combining the values of features 2 and 6 in the following way: $x_i^{(q)} \stackrel{\text{def}}{=} x_i^{(2)} \div x_i^{(6)}$. I selected features 2 and 6, as well as the transformation $\div$ arbitrarily. If the number D of original features is not too large, you can generate all possible transformations (by considering all pairs of features and all arithmetic operators). Then, by using one of the feature selection methods, select those that increase the quality of the model.

[7] Unless you use deep pre-trained models in **transfer learning**, as we will discuss in the next chapter.

4.7 Learning Features from Data

Sometimes, useful features can be learned from data. Learning features from data is especially effective when we can get access to large collections of relevant labeled or unlabeled data, such as text corpora or collections of images from the Web.

4.7.1 Word Embeddings

In Chapter 3, we used word embeddings for data augmentation. **Word embeddings** are feature vectors that represent words. Similar words have similar feature vectors, where similarity is given by a certain measure, such as **cosine similarity**. Word embeddings are learned from large corpora of text documents. A **shallow neural network** with one hidden layer (called the **embedding layer**) is trained to predict a word, given its surrounding words, or to predict the surrounding words, given the word in the middle. Once the neural network is trained, the parameters of the embedding layer are used as word embeddings. There are many algorithms to learn word embeddings from data. The most widely used algorithm, invented at Google, with the code available in open source, is **word2vec**. Pre-trained word2vec embeddings for many languages are available for download.

Once you have a collection of word embeddings for some language, you can use them to represent individual words in sentences or documents written in that language, instead of using **one-hot encoding**.

Let's see how word embeddings are trained by one version of the word2vec algorithm called **skip-gram**. In word embedding learning, our goal is to build a model that we can use to convert a one-hot encoding of a word into a word embedding. Let our dictionary contain 10,000 words. The one-hot vector for each word is a 10,000-dimensional vector of all zeros, except for one dimension that contains a 1. Different words have a 1 in different dimensions.

Consider a sentence: "I am attentively reading the book on machine learning engineering." Now, take the same sentence, but remove one word, say "book." Our sentence becomes: "I am attentively reading the · on machine learning engineering." Now let's only keep the three words before the · and the three words after it: "attentively reading the · on machine learning." Looking at this six-word window around the ·, if I ask you to guess what · stands for, you would probably say: "book," "article," or "paper." That's how the context words let you predict the word they surround. It's also how the machine can learn that words "book," "paper," and "article" have a similar meaning. They share similar contexts in multiple texts.

It turns out that it works the other way around too: a word can predict the context surrounding it. The piece "attentively reading the · on machine learning" is called a skip-gram, with window size 6 (3 + 3). By using the documents available on the Web, we can easily create hundreds of millions of skip-grams.

Let's denote a skip-gram in the following way: $[\mathbf{x}_{-3}, \mathbf{x}_{-2}, \mathbf{x}_{-1}, \mathbf{x}, \mathbf{x}_{+1}, \mathbf{x}_{+2}, \mathbf{x}_{+3}]$. In our sentence, $\mathbf{x}_{-3}$ is the one-hot vector for "attentively," $\mathbf{x}_{-2}$ corresponds to "reading," $\mathbf{x}$ is the

skipped word ($\cdot$), $\mathbf{x}_{+1}$ is "on," and so on.

A skip-gram with window size 4 will look like this: $[\mathbf{x}_{-2}, \mathbf{x}_{-1}, \mathbf{x}, \mathbf{x}_{+1}, \mathbf{x}_{+2}]$. It can also be schematically depicted, as shown in Figure 19. It is a **fully-connected network**, like the **multilayer perceptron**. The input word is denoted as $\cdot$ in the skip-gram. The neural network has to learn to predict the context words of the skip-gram, given the input's central word.

The **activation function** used in the output layer is **softmax**. The **cost function** is the **negative log-likelihood**. The embedding for a word is given by the parameters of the embedding layer that apply when a one-hot encoded word is given as the input to the model.

One problem with word embeddings trained using word2vec is that the set of word embeddings is fixed, and you cannot use the model for out-of-vocabulary words, that is, the words that weren't present in the corpus used to train word embeddings. There are other architectures of neural networks that allow obtaining embeddings for any word, including out-of-vocabulary words. One such architecture, often used in practice, is **fastText**. It was invented at Facebook, and the code is available in open source.

The key difference between word2vec and fastText is that word2vec treats each word in the corpus as a unitary entity, and learns a vector for each word. Alternatively, fastText treats each word as an average of embedding vectors representing character n-grams that word is composed of. For example, the embedding for the word "mouse" is an average of the embedding vectors of the n-grams "<mo," "mou," "<mou," "mous," "<mous," "mouse," "<mouse," "mouse>," "ous," "ouse," "ouse>," "use," "use>," "se>" (assuming that the sizes of the smallest and the largest n-gram are, respectively, 3 and 6).

Word embeddings are an effective way of representing natural language texts for use in such neural network architectures as **recurrent neural networks** (RNN) and convolutional neural networks (CNN) adapted for working with sequences. However, if you want to use word embeddings for representing variable-length texts for **shallow learning** algorithms (which require the input feature vectors of fixed dimensions), you would have to apply some aggregation operation to word vectors, such as weighted sum or average. The representation of a text document obtained as an average of the words composing that document turns out to be not very useful in practice.

4.7.2 Document Embeddings

A popular way of obtaining an embedding for a sentence or an entire document is to use the **doc2vec** neural network architecture, also invented at Google and available in open source. The architecture of doc2vec is very similar to word2vec. The only major difference is that now there are two embedding vectors, one for the document ID and one for the word. The prediction of the surrounding words for an input word is made by, first, averaging the two embedding vectors (the document embedding vector and the word embedding vector), and then predicting the surrounding words from that average. To average the two vectors, they

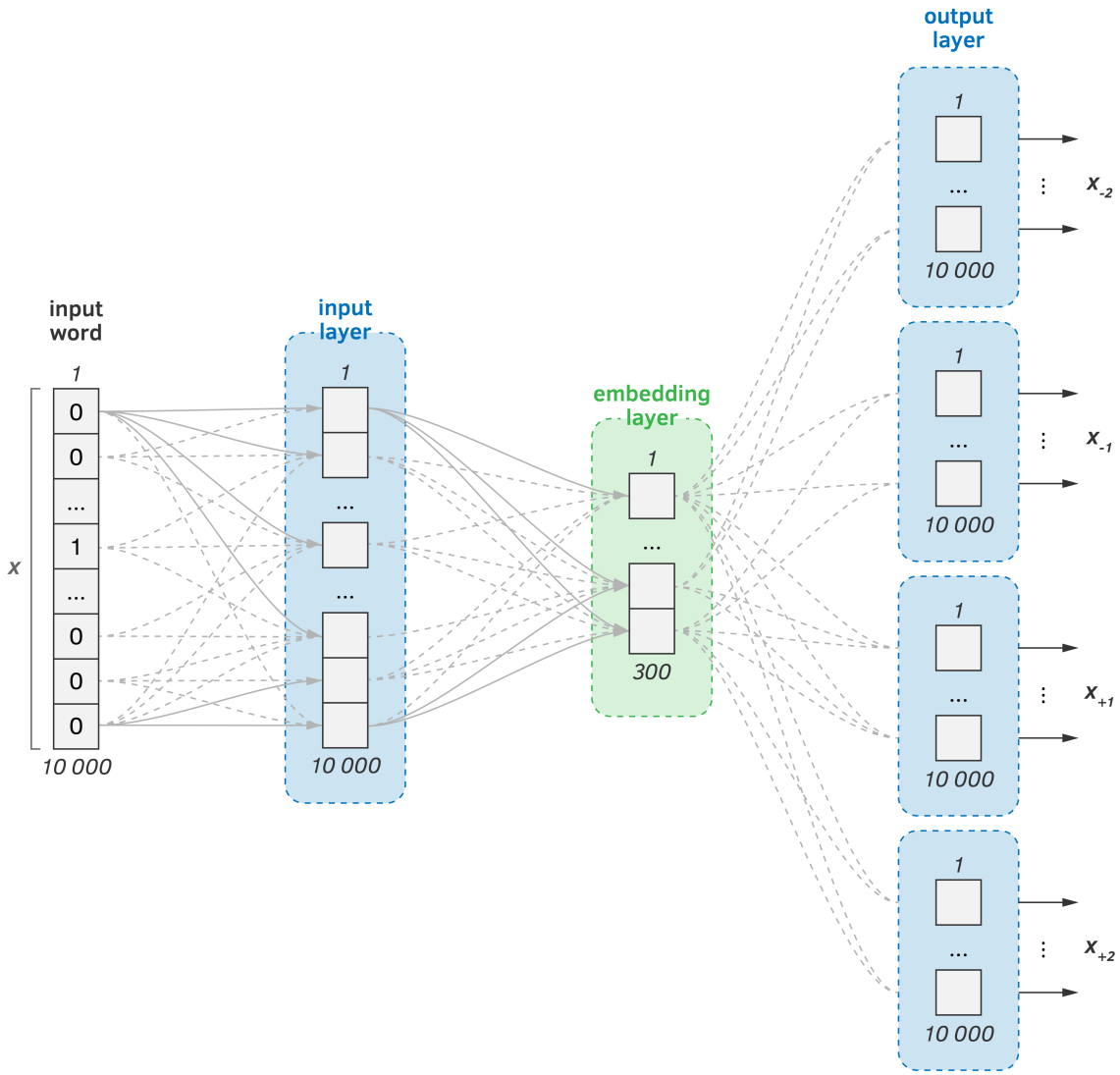


Figure 19: The skip-gram model with window size 4 and the embedding layer of 300 units.

must be of the same dimensionality. Interestingly, this makes it possible to compare not just document vectors (by finding the cosine similarity), but also a document and a word vector. The word vectors trained that way are very similar to those trained using word2vec.

To obtain an embedding for a new document, not belonging to the corpus of documents used to train document embeddings, this new document is first added to the corpus. It gets a new document ID assigned to it. Then the existing model is additionally trained for several epochs with all trained parameters being frozen but the new ones, corresponding to the new document ID. The input document ID is provided as a one-hot encoding.

4.7.3 Embeddings of Anything

The following technique is commonly used to obtain embedding vectors for any object (and not just words or documents). First, we formulate a supervised learning problem that takes our objects as input and outputs a prediction. Then we build a labeled dataset and train a neural network model that solves our supervised learning problem. Then we use the outputs of one of the fully connected layers near the output layer of the neural network model (before non-linearity) as embeddings of the input object.

For example, the ImageNet labeled dataset of images and a deep **convolutional neural network** (CNN) architecture, similar to **AlexNet**, is often used to train embeddings for images. An illustration of the embedding layers for images is shown in Figure 20. In this illustration, we have a deep CNN with two **fully connected layers** near the output. The neural network was trained to predict the object depicted in the image. To obtain an embedding of an image not used for training the model, we send that image (usually represented as three matrices of pixels, one per channel R, G, and B) to the input of the neural network, and then use the output of one of the fully connected layers before non-linearity. Which of the fully connected layers is better depends on the task you want to solve, and must be decided experimentally.

By following the above approach, we can train embeddings of any type. The data analyst only needs to figure out three things:

- what supervised learning problem to solve (for images, usually object classification),
- how to represent the input for the neural network (for images, matrices of pixels, one per channel), and
- what will be the architecture of the neural network before the fully connected layers (for images, usually a deep CNN).

4.7.4 Choosing Embedding Dimensionality

The embedding dimensionality is usually determined experimentally or from experience. For example, Google, in its TensorFlow documentation, recommends the following rule of thumb:

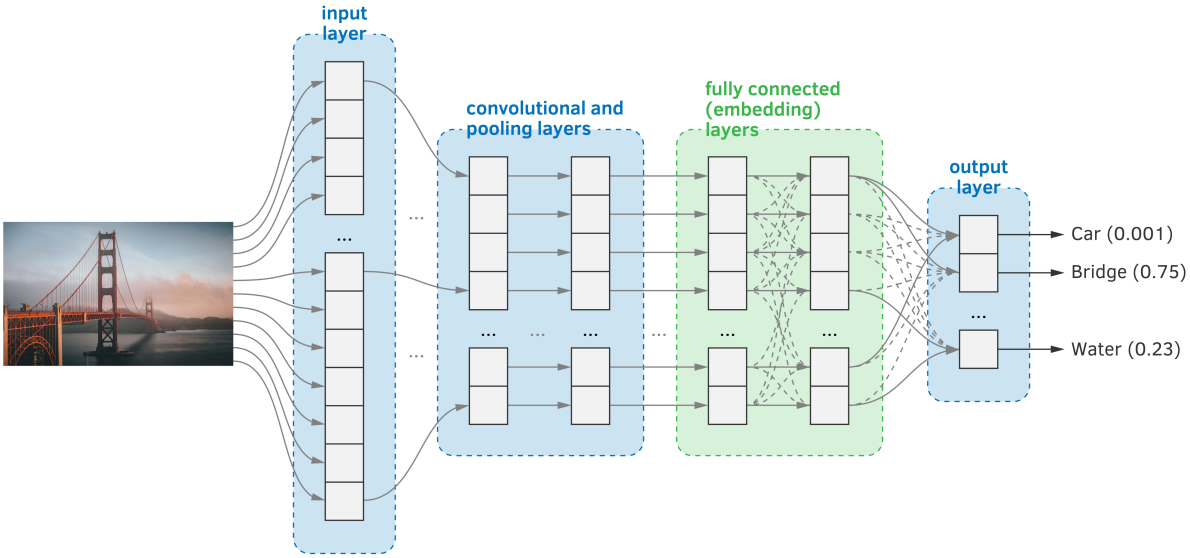


Figure 20: A neural network architecture for training image embeddings. The embedding layers are shown in green.

$$d = \sqrt[4]{D},$$

where d is the embedding dimensionality and D is the "number of categories." The number of categories for word embeddings is the number of unique words in the corpus. For arbitrary embeddings, it's the dimensionality of the original input. For example, if the number of the unique words in the corpus is $D = 5000000$ then the embedding dimensionality $d = \sqrt[4]{5000000} = 47$. In practice, values between 50 and 600 are often used.

A more principled approach to choose the embedding dimensionality is to treat it as a hyperparameter tuned on a downstream task. For example, if you have a labeled corpus of documents, then you can optimize the embedding dimensionality by minimizing the number of prediction errors made by the classifier trained on that labeled data, where the words in the documents are represented by the embeddings.

4.8 Dimensionality Reduction

Sometimes, it might be necessary to reduce the dimensionality of examples. This is different from the problem of feature selection. In the latter, we analyze the properties of all existing features and remove those that, in our opinion, do not contribute much to the quality of the

model. When we apply a **dimensionality reduction** technique to a dataset, we replace all features in the original feature vector with a new vector, of lower dimensionality, and of synthetic features.

Dimensionality reduction often results in increased learning speed and better generalization. In addition, it improves visualization of datasets: humans can only see in three dimensions.

There are several ways to reduce dimensionality. And depending on why we want to do that, some are more popular than others. The dimensionality reduction techniques are well described in the machine learning theory books, so I will only discuss when a data analyst should prefer one technique over the others.

4.8.1 Fast Dimensionality Reduction with PCA

Principal Component Analysis (PCA) is the oldest of the techniques. It is also, by far, the fastest option. Performance comparison tests show a very weak dependence of the speed of the PCA algorithm on the size of the dataset. Therefore, you can effectively use PCA as a step preceding your model training, and find the optimal value of the reduced dimensionality experimentally as part of the hyperparameter tuning process.

PCA's most significant drawback is that the data must fit in memory entirely for the algorithm to work. There's an out-of-core version of PCA, called **Incremental PCA** that allows running the algorithm on batches of the dataset, loading in memory one batch at a time. Still, Incremental PCA is an order of magnitude slower than PCA. PCA is also less practical for visualization purposes as compared to the other two techniques considered below.

4.8.2 Dimensionality Reduction for Visualization

If visualization is your goal, then you would prefer Uniform Manifold Approximation and Projection (**UMAP**) algorithm, or an **autoencoder**. Both can be specifically programmed to produce 2D or 3D feature vectors, while in PCA, the algorithm produces D so-called **principal components** (where D is the dimensionality of your data), and the analyst must pick the first two or three principal components as features for visualization. UMAP is generally much faster than autoencoder, but the two techniques produce very different looking visualizations, so you would prefer one over the other based on the properties of the specific dataset. Furthermore, like PCA, UMAP requires all data to be in memory, while autoencoder can be trained in batches.

Dimensionality reduction can also be task-specific. For example, we can reduce the dimensionality of pictures by using an image editor. Similarly, we can reduce the bit rate and the number of channels of the sound sequences.

4.9 Scaling Features

Once all your features are numerical, you are almost ready to start working on your model. The only remaining step that might be helpful is scaling your features.

Feature scaling is bringing all your features to the same, or very similar, ranges of values or distributions. Multiple experiments demonstrated that a learning algorithm applied to scaled features might produce a better model. While there's no guarantee that scaling will have a positive impact on the quality of your model, it's considered a best practice. Scaling can also increase the training speed of deep neural networks. It also assures that no individual feature dominates, especially in the initial iterations of gradient descent or other iterative optimization algorithms. Finally, scaling reduces the risk of **numerical overflow**, the problem that computers have when working with very small or very big numbers.

4.9.1 Normalization

Normalization is the process of converting an actual range of values, which a numerical feature can take, into a predefined and artificial range of values, typically in the interval $[-1, 1]$ or $[0, 1]$.

For example, let the natural range of a feature be 350 to 1450. By subtracting 350 from every value of the feature, and dividing the result by 1100, we normalize those values to the range $[0, 1]$. More generally, the normalization formula looks like this:

$$\bar{x}^{(j)} \leftarrow \frac{x^{(j)} - \min^{(j)}}{\max^{(j)} - \min^{(j)}},$$

where $x^{(j)}$ is an original value of feature j in some example; $\min^{(j)}$ and $\max^{(j)}$ are, respectively, the minimum and the maximum value of the feature j in the training data.

If you prefer the range of $[-1, 1]$ then the normalization formula would look like this:

$$\bar{x}^{(j)} \leftarrow \frac{2 \times x^{(j)} - \max^{(j)} - \min^{(j)}}{\max^{(j)} - \min^{(j)}},$$

A drawback of normalization is that the values $\max^{(j)}$ and $\min^{(j)}$ are usually outliers, so normalization will "squeeze" the normal feature values into a very small range. One solution to this problem is to apply **clipping**, that is to pick "reasonable" values for $\max^{(j)}$ and $\min^{(j)}$ instead of using extreme values from the training data. Let a reasonable range for a feature be estimated as $[a, b]$. Before calculating the scaled value by using one of the above two formulas, the value $x^{(j)}$ of the feature is set ("clipped") to a if $x^{(j)}$ is below a, or to b if it's above b. A frequent way to estimate the values for a and b is **winsorization**. The technique is named after the engineer and biostatistician Charles Winsor (1895—1951). Winsorization consists of setting all outliers to a specified percentile of the data; for example,

a 90% winsorization would see all data below the 5th percentile set to the 5th percentile, and data above the 95th percentile set to the 95th percentile. In Python, winsorization could be applied to a list of numbers as follows:

```
from scipy.stats.mstats import winsorize
winsorize(list_of_numbers, limits=[0.05, 0.05])
```

The output of the `winsorize` function will be a list of numbers of the same length as the input, with the values of outliers "clipped." A corresponding code in R is shown below:

```
library(DescTools)
DescTools::Winsorize(vector_of_numbers, probs = c(0.05, 0.95))
```

Sometimes, the **mean normalization** is used:

$$\bar{x}^{(j)} \leftarrow \frac{x^{(j)} - \mu^{(j)}}{\max^{(j)} - \min^{(j)}},$$

where $\mu^{(j)}$ is the sample mean of the values of feature j.

4.9.2 Standardization

Standardization (or **z-score normalization**) is the procedure during which the feature values are rescaled so that they have the properties of a **standard normal distribution**, with $\mu = 0$ and $\sigma = 1$, where μ is the **sample mean** (the average value of the feature, averaged over all examples in the training data) and σ is the standard deviation from the sample mean.

Standard scores (or **z-scores**) of features are calculated as follows:

$$\hat{x}^{(j)} \leftarrow \frac{x^{(j)} - \mu^{(j)}}{\sigma^{(j)}},$$

where $\mu^{(j)}$ is the sample mean of the values of feature j, and $\sigma^{(j)}$ is the **standard deviation** of the values of feature j from the sample mean.

In addition, sometimes it's helpful to apply simple mathematical transformations to the feature values prior to applying the scaling techniques described above. Such transformations include taking the logarithm of the feature, squaring it, or extracting the square root of the feature. The idea is to obtain a distribution as close to a normal distribution as possible.

You may wonder when you should use normalization, or when to use standardization. There's no definitive answer to this question. In theory, normalization would work better for uniformly distributed data, while standardization tends to work best for normally distributed data. However, in practice, data is rarely distributed following a perfect curve. Usually, if your

dataset is not too big and you have time, you can try both and see which one performs better for your task. Feature scaling is usually beneficial to most learning algorithms.

4.10 Data Leakage in Feature Engineering

Data leakage during feature engineering can happen in several situations, including feature discretization and scaling.

4.10.1 Possible Problems

Imagine that you use your entire dataset to calculate the ranges of each bin or the feature scaling factors. Then you split the dataset into training, validation, and test sets. If you proceed like that, the values of features in the training data will, in part, be obtained by using the examples that belong to the holdout sets. When your dataset is small enough, it might result in an overly optimistic performance of your model on the holdout data.

Now imagine you are working with text, and that you use **bag-of-words** to create features with the entire dataset. After building the vocabulary, you split your data into the three sets. In this situation, the learning algorithm will be exposed to features based on tokens only present in the holdout sets. Again, the model will display artificially better performance than had you divided your data before feature engineering.

4.10.2 Solution

A solution, as you might have guessed, is first to split the entire dataset into training and holdout sets, and only do feature engineering on the training data. This also applies when you use **mean encoding** to transform a categorical feature to a number: split the data first and then compute the sample mean of the label, based on the training data only.

4.11 Storing and Documenting Features

Even if you plan to train the model right after you finish engineering features, it's advised to design a **schema file** that provides a description of the features' expected properties.

4.11.1 Schema File

A schema file is a document that describes features. This file is machine-readable, versioned, and updated each time someone makes significant updates to features. Here are several examples of the properties that can be encoded in the schema:

- names of features;

- for each feature:
  - its type (categorical, numerical),
  - the fraction of examples that are expected to have that feature present,
  - minimum and maximum values,
  - sample mean and variance,
  - whether it allows zeros,
  - whether it allows undefined values.

An example of a schema file for a four-dimensional dataset is shown below:

```
feature {
    name : "height"
    type : float
    min : 50.0
    max : 300.0
    mean : 160.0
    variance : 17.0
    zeroes : false
    undefined : false
    popularity : 1.0
}

feature {
    name : "color_red"
    type : binary
    zeroes : true
    undefined : false
    popularity : 0.76
}

feature {
    name : "color_green"
    type : binary
    zeroes : true
    undefined : false
    popularity : 0.65
}

feature {
    name : "color_blue"
    type : binary
    zeroes : true
    undefined : false
    popularity : 0.81
```

```
}
```

4.11.2 Feature Store

Large and distributed organizations may use a **feature store** that allows keeping, documenting, reusing, and sharing features across multiple data science teams and projects. The ways features are maintained and served can differ significantly across projects and teams. This introduces infrastructure complexity and often results in duplicated work. Large distributed organizations face some of these challenges:

Features not being reused
: Features representing the same attribute of an entity are being implemented several times by different engineers and teams, when existing work from other teams and existing machine learning pipelines could have been reused.

Feature definitions vary
: Different teams define features differently, and it's not always possible to access the documentation of a feature.

Computationally intensive features
: Some real-time machine learning models aren't based on informative but computationally intensive features. Having those features available in a fast store would allow using such features in real-time, and not only in batch mode.

Inconsistency between training and serving
: The model is usually trained using the historical data, but when served, is exposed to the real-time online data. The values of some features might depend on the entire historical dataset unavailable at the service time. For the model to work correctly, each feature must have the same value for the same input data entity, both in the offline (development) and online (production) mode.

Feature expiration is unknown
: When a new input example comes into the production environment, there is no way to know exactly which features need to be recomputed; rather the entire pipeline needs to be run to compute the values of all features needed for prediction.

A feature store is a central vault for storing documented, curated, and access-controlled features within an organization. Each feature is described by four elements: 1) name, 2) description, 3) metadata, and 4) definition.

The feature name is a string that uniquely identifies the feature, for example: "average_session_length" or "document_length."

The feature description is a natural language textual description of the object's property it represents, for example, "The average length of the session for a user." or "The number of words in the document."

In addition to those attributes in the schema file, the feature metadata may supply: why the feature was added to the model, how it contributes to generalization, the person's name in the organization responsible for maintaining the feature's data source,[8] the input type (e.g., numerical, string, image), the output type (e.g., numerical scalar, categorical, numerical vector), whether the feature store must cache the value of the feature, and if yes, for how long. A feature can also be marked as available online and offline, or just for offline processing. Features available for online processing must be implemented in such a way that their value can be either: 1) read fast from a cache or a value store or 2) computed in real-time. Features that can be computed in real-time include, for example, squaring the input number, determining the shape of the word, or doing a search in the organization's intranet.

The definition of the feature is the versioned code, such as Python or Java. It will be executed in a runtime environment and applied to the input to compute the feature value.

A feature store allows data engineers to insert features. In turn, data analysts and machine learning engineers use an API to get feature values which they deem relevant. A feature store can provide features for a single online input. Or, the analyst working on a model offline may want to convert the training data into a collection of feature vectors, and will send to the feature store a batch of inputs.

For **reproducibility**, feature values in a feature store are versioned. With feature value versioning, the data analyst is able to rebuild the model with the same feature values as those used to train the previous model version. After the feature value for a given input is updated, the previous value is not erased. Rather, it is saved with a timestamp indicating when that value was generated. Furthermore, a feature j used by model m_B can itself be the output of some model m_A. Once model m_A changes, it is important to keep its older versions: model m_B still might expect as input the outputs generated by an older version of m_A.

The feature store is located in the overall machine learning pipeline as shown in Figure 21. The architecture was inspired by Uber's Michelangelo machine learning platform. It contains two feature stores, online and offline, whose data is in sync. At Uber, the online feature store is updated frequently, in near real-time, by using the real-time data. In contrast, the offline feature store is updated in batch-mode by using values of some features computed online, as well as with historical data from logs and offline databases. An example of a feature computed online is "restaurant's average meal preparation time over the last hour." An example of a feature computed offline is "restaurant's average meal preparation time over the last seven days." At Uber, the features from the offline store are synced to the online store once or several times a day.

4.12 Feature Engineering Best Practices

Throughout the years, analysts and engineers have invented, experimented with, and validated various best practices. Today, they are recommended for nearly every machine learning

[8] If the person responsible for the feature leaves the company, the product owner must be alerted automatically.

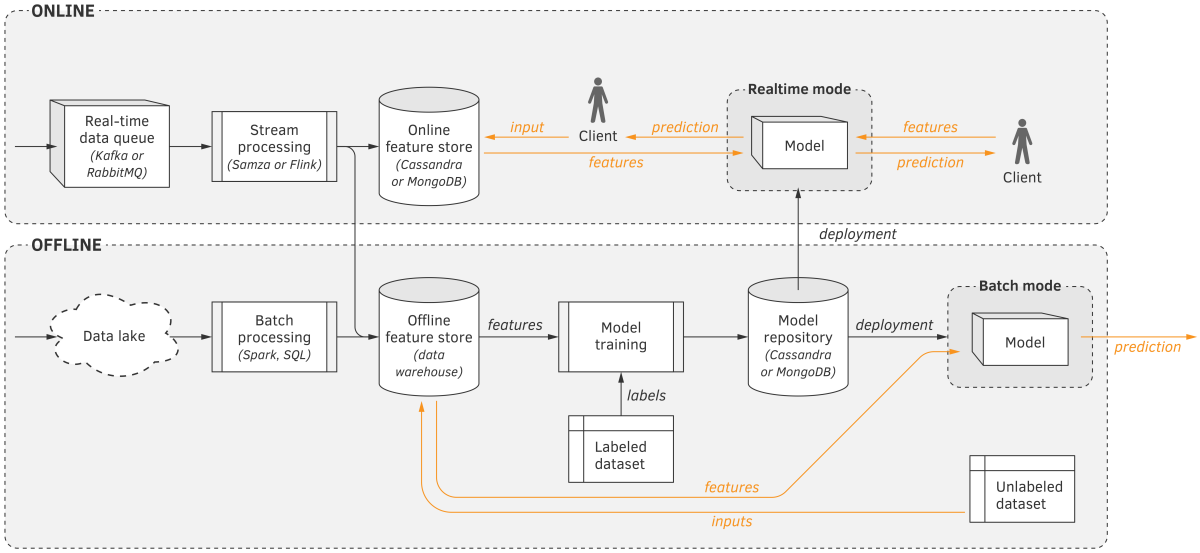


Figure 21: The place of the feature store in the overall machine learning pipeline.

project. Using those best practices might not significantly improve each project, but they will definitely not hurt. One best practice already considered in this chapter is to normalize or standardize the features.

4.12.1 Generate Many Simple Features

At the beginning of the modeling process, try to engineer as many "simple" features as possible. A feature is simple when it doesn't take significant time to code. For example, the bag-of-words approach in document classification generates thousands of features with just a couple lines of code. As long as your hardware has the capacity, use anything measurable as a feature. You cannot know in advance whether some quantity, in combination with other quantities, will be useful for prediction.

4.12.2 Reuse Legacy Systems

When replacing an old, non-machine-learning-based algorithm with a statistical model, use the output of the old algorithm as a feature for the new model. Make sure that the old algorithm doesn't change anymore; otherwise, your model's performance might be negatively affected over time. If the old algorithm is too slow to be a feature, use the old algorithm's inputs as features for the new model.

Use an external system as a feature source only if you control the external system's behavior. Otherwise, there's a chance that the external system evolves with time, unbeknownst to you. Furthermore, an external system's owner might decide to use the output of your model as the input for their model. This creates the **hidden feedback loop**, a situation where you influence the phenomenon from which you learn.

4.12.3 Use IDs as Features when Needed...

Use IDs as features when needed. This might seem counter-intuitive because unique IDs don't contribute to generalization. However, the use of IDs allows creating one model that has one behavior in a general case, and different behaviors in other cases.

For example, you want to make predictions about some location (city or village), and you have some properties of the location as features. By using location ID as a feature, you can add the training examples for one general location, and train the model to behave differently in other specific locations.

However, avoid using example ID as a feature.

4.12.4 ...But Reduce the Cardinality When Possible

Use categorical features with many values (more than a dozen) only when you want the model to have different "modes" of behavior that depend on that categorical feature. Typical examples of this are zip code (postal code) or country. You might consider using the categorical feature "Country" if you want the model to behave differently in Russia versus the United States, for otherwise similar inputs.[9]

If you have a categorical feature with many values, but you do not need a model that has several modes depending on that feature, try to reduce the cardinality (i.e., the number of distinct values) of that feature. There are several ways to do that. We already considered one of them, **feature hashing**, in Section 4.2.4. Other techniques are briefly discussed below:

Group similar values
: Try to group some values into the same category. For example, if you think it's unlikely that, within one region, different locations might need different predictions, then group all postal codes from the same state into one state code. Group states into regions.

Group the long tail
: Likewise, try to group the long tail of infrequent values under the name "Other," or merge them with similar frequent values.

[9] Often, what you want your model to do and what the data dictates are two very different things. Even if you think that the model must make similar predictions independently of the country, in reality, you might get poor model performance because the distribution of labels in the training data is different for different countries.

Remove the feature

If all, or almost all, values of a categorical feature are unique, or one value dominates all other values, consider removing the feature entirely.

The reduction of a feature's granularity should be made with care. Categorical features often have functional dependencies with other categorical features, and their predictive power often comes from their combinations. Take state and city as an example. If we decide to group or remove some values in the state feature, we might inadvertently destroy the information that would allow the model to distinguish one "Springfield" from another.

4.12.5 Use Counts with Caution

Use features based on counts with caution. Some counts remain roughly in the same bounds over time. For example, in bag-of-words, if you use the count of each token instead of the binary value, then it's not a problem, as long as the input document length doesn't grow or shrink with time. But, if you have a feature like "Number of calls since subscription" for a customer of a growing mobile phone provider, some oldtimers can have a very high number of calls, compared to the newer customer base. On the other hand, the training data could have been built when the company was still young and didn't have any oldtimers.

The same caution must be applied when you group feature values in bins based on how common those values are in the dataset. Infrequent values today may become more frequent over time, as more data is added. It is considered a best practice to re-evaluate the model and the features from time to time.

4.12.6 Make Feature Selection When Necessary

Make feature selection when it's necessary. The reasons could be:

- the need to have an explainable model (so you keep the most significant predictors),
- strict hardware requirements, such as RAM, hard drive space, or
- short time available to experiment and/or rebuild the model in production,
- you expect a significant **distribution shift** between two model trainings.

If you decide to do feature selection, start with Boruta.

4.12.7 Test the Code Carefully

The feature engineering code must be carefully tested. Unit tests should cover each feature extractor. Check that each feature is generated correctly using as many inputs as possible. For each boolean feature, check that it is true when it should be true and is false when it should be false. Check numerical features for a reasonable value range. Check for NaNs (Not-a-Number values), nulls, zeros, and empty values. A broken extractor for one feature

can result in arbitrarily poor performance of the model. Feature extractors are the first place to look for a problem if the model's behavior is strange.

Each feature has to be tested for speed, memory consumption, and compatibility with the production environment. What works reasonably well in your local environment may perform poorly when deployed in production.

Once the model is deployed in the production environment, and each time it is loaded, you must rerun feature extractor tests. If a feature consumes some external resources like a database or an API, these resources might be unavailable on a specific production runtime instance. The feature extractor has to throw an exception and die if any resource during feature extraction is unavailable. Avoid silent failures that may remain unnoticed for a long time with model performance degrading or becoming completely wrong.

It is also recommended to perform regular runs of feature extractors on a fixed test data to make sure that the feature value distribution remains the same.

4.12.8 Keep Code, Model, and Data in Sync

The version of the feature extraction code must be in sync with the model's version and the data used to build it. The three have to be deployed or rolled back at the same time. Each time the model is loaded in production, it's useful to check that the three elements are in sync (that is, their versions are the same).

4.12.9 Isolate Feature Extraction Code

The feature extraction code must be independent of the remaining code that supports the model. It should be possible to update the code responsible for each feature without affecting other features, the data processing pipeline, or the way the model is called. The only exception is when many features are generated in bulk, like in one-hot encoding and bag-of-words.

4.12.10 Serialize Together Model and Feature Extractor

When possible, jointly serialize (pickle in Python, RDS in R) the model and the feature extractor object that was used when the model was built. In the production environment, deserialize both and use them. When possible, avoid having several versions of the feature extraction code.

If your production environment doesn't let you deserialize both the model and the feature extraction code, use the same feature extraction code when you train the model and serve it. Even a tiny difference between the code a data scientist used to train the model, and the optimized code the IT team might have written for the production environment, may result in significant prediction error.

Once the production code for feature extraction is ready, use it to retrain the model. Always completely retrain the model after any change in the feature extraction code.

4.12.11 Log the Values of Features

Log the feature values extracted in production for a random sample of online examples. When you work on a new version of the model, these values will be useful to control the quality of the training data. They will allow you to compare and ensure that the feature values logged in the production environment are the same as those you observed in the training data.

4.13 Summary

Features are values extracted from the data entities your model is designed to work with. Each feature represents a specific property of a data entity. Features are organized in feature vectors, and the model learns to perform mathematical operations on those feature vectors to generate the desired output.

For text, features can be generated in bulk by using techniques like bag-of-words. Numbers in the bag-of-words feature vectors mean the presence or absence of specific vocabulary words in the text document. Those numbers can be binary or contain more information, such as the frequency of each word in the document, or a TF-IDF value.

Most machine learning algorithms and libraries require that all features are numerical. To convert categorical features to numbers, techniques such as one-hot encoding and mean encoding are used. If the categorical feature's values are cyclical, like days of the week or hours in a day, a better alternative is to convert that cyclical feature into two features, using the sine-cosine transformation.

Feature hashing is a way to convert text data, or categorical attributes with many values, into a feature vector of an arbitrary dimensionality. That can be useful when one-hot encoding or bag-of-words generate feature vectors with impractical dimensions.

Topic modeling is a family of algorithmic techniques, such as LDA and LSA, that allow us to learn a model that converts any document into a vector of topics of a required dimensionality.

A time series is an ordered sequence of observations. Each observation is marked with a time-related attribute, such as timestamp, date, year, and so on. Before neural networks reached their modern capacity to learn, analysts worked with time-series data using the shallow machine learning toolkit. The time-series had to be converted into "flat" feature vectors. Nowadays, analysts use neural network architectures adapted to work with sequences, such as LSTM, CNN, and Transformer.

Good features have high predictive power, can be computed fast, are reliable and uncorrelated.

It is important that the distribution of feature values in the training set is similar to the distribution of values the production model will receive. Furthermore, good features are

unitary, easy to understand and maintain. The property of being unitary means that the feature represents a simple-to-understand and -explain quantity.

To increase the predictive power of the data, additional features can be synthesized by discretizing an existing numerical feature, clustering training examples, or applying simple transformations to existing features or combining pairs of features.

For text, features can be learned from unlabeled data in the form of word and document embeddings. More generally, embeddings can be trained for any type of data if we manage to formulate an appropriate prediction problem and train a deep model. Embedding vectors are then extracted from several rightmost (i.e., closest to the output) fully-connected layers.

Wise use of feature selection techniques remove features that don't contribute to a model's quality. Two common techniques are cutting the long tail and Boruta. L1 regularization also works as a features selection technique.

Dimensionality reduction can improve visualization of high-dimensional datasets. It can also improve the model's predictive quality. Presently, such techniques as PCA, UMAP, and autoencoders are used for dimensionality reduction. PCA is very fast, but UMAP and autoencoders produce better visualizations.

It is considered best practices to scale features before training the model, store and document features in schema files or feature stores, and keep code, model, and training data in sync.

Feature extraction code is one of the most important parts of a machine learning system. It must be extensively and systematically tested.