Andriy Burkov

MACHINE LEARNING ENGINEERING

“An optimist sees a glass half full. A pessimist sees a glass half empty. An engineer sees a glass that is twice as big as it needs to be.”
— Unknown

“Death and taxes are unsolved engineering problems.”
— Unknown

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 the second stage of machine learning engineering in the machine learning project life cycle:

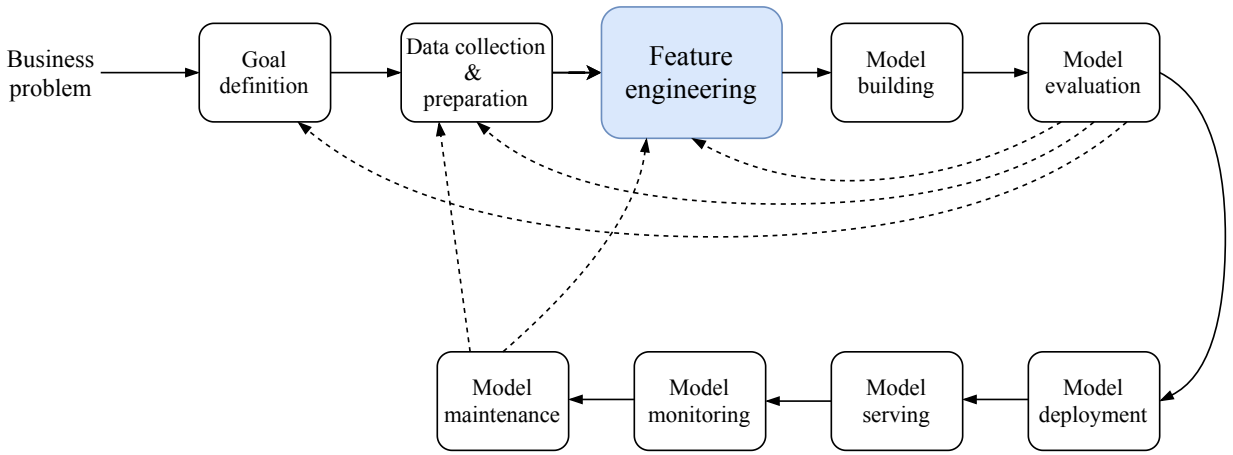


Figure 1: Machine learning project life cycle.

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 the titles of movies in tweets. Imagine that you have a huge collection of movie titles; this is the data you can use indirectly. You also have a collection of tweets that you will use directly to create examples. You can create examples as follows. 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 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.

In the above tweet, our movie title matching index would help us to find the following matches: "avatar", "the terminator", "It", and "her". You can label these four examples and now you have four labeled examples: $\{(\text{avatar}, False), (\text{the terminator}, True), (\text{It}, False), (\text{her}, False)\}$. 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 that precede the match and the five words that follow 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 apply to such data. Machine learning algorithms can only apply to feature vectors. This is where you have to 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 applied 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 vectors of numbers.

4.2.1 Feature Engineering for Text

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

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

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

Now, instead of using, in your imaginary spreadsheet, one column to represent the attribute "Color", you use three columns. The advantage of doing this is that you can now apply a vast range of machine learning algorithms to your dataset: only a handful of learning algorithms support categorical attributes.

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

Imagine that you have a collection of six text documents as shown in Figure 3.

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 classifier of texts by topic. A classification learning algorithm expects that its inputs are labeled feature vectors, so you have to transform your collection of documents into a collection of feature vectors. Bag of words allows you to do just that.

To apply the bag of words technique, the first thing you do is tokenize your texts. **Tokenization** is a procedure of splitting a text into pieces called tokens. Typically tokens are words, but it's not strictly necessary. A tokenizer is a software that takes a string as input and returns a sequence of tokens extracted from the input string. A tokenizer can only extract words, or it can extract words and some punctuation marks. A token can be a punctuation

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

mark, a word, or, in some cases, a combination of words, such as a title of commerce (e.g., McDonald's) or a place (e.g., Red Square). Let's, for our simple collection of texts, use a simple tokenizer that extracts words and ignores everything else. After tokenization we obtain the following collection of tokenized documents:

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 in applying bag of words is to build a vocabulary of tokens. In our case the vocabulary contains 16 tokens:

(I decided to ignore the case of words, but you, as an analyst, might decide to treat two tokens "Love" and "love" as two separate entities in the dictionary.)

The next step is to order your vocabulary in some way and assign each token a unique index. This index will be used to index features in the feature vectors. 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.

Now, as you have a vocabulary and each token in the vocabulary has its unique index, from 1 to 16, you can transform your collection of tokenized documents into a collection of binary feature vectors as shown below:

	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.

You can see in the above figure, that a feature vector representing a document contains a 1 in a certain position if the corresponding token from the dictionary is present in the text of the document. Otherwise, the feature at that position has a value of 0.

Now you can assign labels to each of the six documents and use the corresponding labeled feature vectors as the training data which any classification learning algorithm can work with. For instance, Document 1 "Love, love is a verb" is represented by the following feature vector:

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

There are several "flavors" of bag of words. The one with binary values I presented above works well in most cases, however, binary values are not the only ones you could use. Alternatives include 1) using counts of tokens, 2) frequencies of tokens or 3) TF-IDF. If you use counts of words, then the feature vector for Document 1 "Love, love is a verb" would have the value of 2 that represents the word "love". In the case of frequencies, the value that represents the word "love" would be $2/5 = 0.2$, assuming that the tokenizer returned five tokens. **TF-IDF** normalizes the frequency of the word by dividing it by a positive value which is bigger for words that are more frequent in the entire collection of the documents and lower for rare words; this results in low values of features representing very common words such as "a", "the", "to", and so on. I will not go in more detail on TF-IDF; you can easily find that information 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. Fix $n = 2$ and decide to ignore the punctuation marks, then all two-grams (usually called bigrams) that can be found in the text "Luke, I'm your father." are as follows: ["Luke I", "I'm", "'m your", "your father"]. The three-grams are ["Luke I'm", "I'm your", "'m 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 then deal with the same way as we deal with a bag of words.

N-grams make a feature vector representing a text even more **sparse** compared to when it's solely based on tokens. At the same time, n-grams allow the machine learning algorithm to learn a more nuanced model, capable of, for example, recognizing in the text the presence of multi-word locations like "New York" or personalities like "Steve Jobs" and behaving differently when those elements are present or absent in the text.

4.2.2 Why Bag of Words Works

One property of a feature vector that should never be violated is that a feature at position j in a feature vector represents the same property in all examples in a dataset. That means that if feature at position j represents the height in cm of a certain person in a dataset of people (where each example represents a different person), then in all other examples from this dataset the feature at position j also represents the height in cm and nothing else.

In feature vectors obtained using the bag of words technique, each feature indeed represents the same property of a document: whether a specific token is present or absent in that document.

The second property of a feature vector that should not be violated is that similar entities in the dataset must be represented by similar feature vectors. This property is also respected in the feature vectors built using the bag of words technique. Indeed, two absolutely identical documents will have absolutely identical feature vectors. At the same time, two texts on the same topic will have higher chances to have similar feature vectors because they will share more words than texts on two different topics.

4.2.3 Converting Categorical Features to Numbers

We have already seen one way to convert a categorical feature to several numerical ones by using one-hot encoding. While frequently used in practice, one-hot encoding is not the only way to do that and not always the best way.

Mean encoding, also known as **bin counting** or **feature calibration**, is another effective technique to convert a categorical feature to a number. The technique works as follows. Each value z of the categorical feature is replaced by the sample mean value of the label; the sample mean is taken over all examples in which the feature has value z. The advantage of this technique over one-hot encoding is that you replace one feature by another (the

dimensionality of your data doesn't increase); furthermore, the numerical value of the feature by design 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, in practice analysts often use the log odds ratio:

$$\begin{aligned}\text{log odds ratio}(\text{infected}, \text{spam}) &= [1, 0, 0] \\ &= \log(145/8) - \log(346/2909) \\ &= \log(145) - \log(8) - \log(346) + \log(2909) = 2.2.\end{aligned}$$

Now you can replace the value "infected" in the above categorical feature by the value of 2.2.

x	x_{sin}	x_{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

You can proceed the same way for other values of that categorical feature and convert all of them into log odds ratio values.

If your categorical feature is ordered but not cyclical like school marks (from "a" to "e") or seniority levels (junior, mid-level, senior) instead of representing them using one-hot encoding or another technique we considered above, it's convenient to represent them by meaningful numbers ("b" higher than "c", or "senior" higher than "junior"). It's recommended to pick uniform numbers in the $[0, 1]$ range, like $1/3$ for "junior", $2/3$ for "mid-level" and 1 for "senior". If you know that some values should be farther from one another, you can reflect that knowledge in your choice of numbers. For example, if you think that "senior" should be farther from "mid-level" than "mid-level" from "junior", you might decide to use the following numbers instead: $1/5$, $2/5$, 1 for "junior", "mid-level", and "senior" respectively. This is where domain knowledge has an important role to play.

On the other hand, of your ordered categorical feature is cyclical like days of a week or hours of a day, a better alternative is to convert that cyclical feature into two features using sine and cosine transformation. Let your feature represents the days of the week. Let encode them first as integers from 1 to 7 to represent days from Monday to Sunday. If we leave this integer encoding, the problem is that the difference between Sunday and Saturday is 1 while the difference between Monday and Sunday is -6. We would expect the same difference of 1 in this case because Monday is just one day past Sunday.

The sine and cosine transformation works as follows. Let x denote the integer value of our cyclical feature. Replace the value x of the cyclical feature by the following two values:

$$x_{sin} = \sin(\frac{2 * \pi * x}{\max(x)}), x_{cos} = \cos(\frac{2 * \pi * x}{\max(x)})$$

The table below contains the values of x_{sin} and x_{cos} the seven days of the week.

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

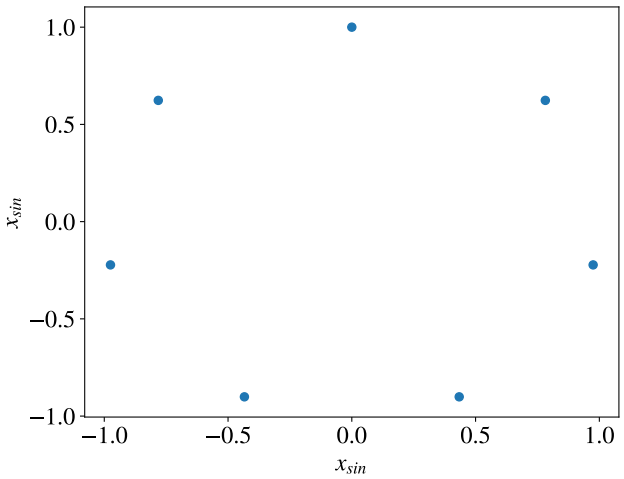


Figure 8: The sine-cosine transformed feature that represents the days of the week.

4.3 Stacking Features

Back to our illustrative problem of movie title classification in tweets. In that problem, each example has three parts:

1) five words[2] 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).

In order to represent such multi-part examples, we can 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.

In our movie title classification problem, we first collect all the left contexts. We can then apply bag of words to transform each left context into a binary feature vector. We then collect all extractions and transform, also using bag of words, each extraction into a feature vector. After that, we collect all the right contexts and apply bag of words once again to transform each right context into a feature vector. Finally, for each example, we concatenate the feature vectors of the left context, the extraction, and the right context, to obtain the final feature vector that represents the entire example, as shown in Figure 9.

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

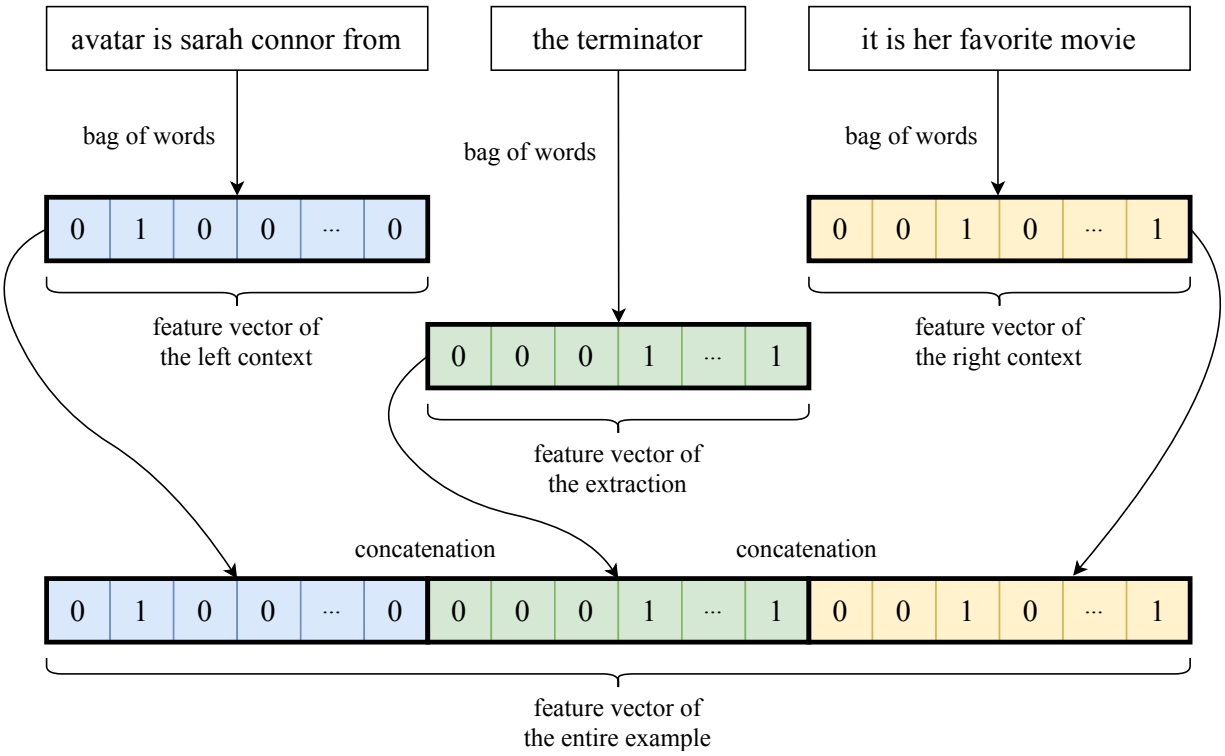


Figure 9: Feature vector stacking.

Note that the three feature vectors (one for 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 dimensionality of the feature vector for each part can also be different.

The order in which you concatenate feature vectors doesn't matter: the features of the left context can be put, in the final feature vector, in the middle or on the right side. However, you have to keep the same order of concatenation for all examples. The latter requirement is important to make sure that each feature represents the same property from one example to another.

Until now, we engineered features in bulk: one-hot encoding and bag of words often generate thousands of binary features. This is a very time-efficient way of engineering features, but, of course, some problems require doing more than that to obtain feature vectors with high enough predictive power.

Imagine you already have a classifier 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

A. In this case, you will engineer one feature which can be described as "whether the topic of the tweet is cinema" and that feature will also be binary: 1 if the topic of the entire tweet is cinema and 0 otherwise. Similarly to how we concatenated three partial feature vectors, this new feature can be concatenated to those three as shown in Figure 10.

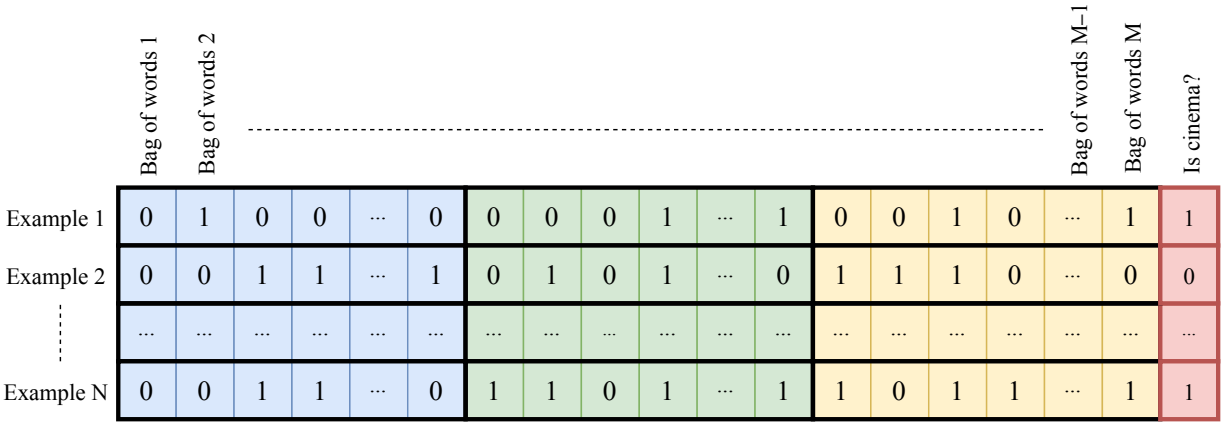


Figure 10: Single feature stacking.

You might come up with many more features that you might think would be useful for the classification of titles in tweets. Examples of such features are:

- the average IMDB score of the movies with that title,
- the total number of votes for the movies with that title on IMDB,
- whether the movie with that title is recent (or the number that represents the release year),
- whether the context text contains other movie titles,
- whether the context text contains 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 important condition, as usual, is that, for all examples, they are concatenated in the same order.

4.4 Properties of Good Features

First of all, a good feature has high **predictive power**. In the previous chapter, you already read about predictive power as a property of data. However, a feature can also have high or low predictive power. The problem of having features with low predictive power can be illustrated like this: let's say you want to predict whether a patient has cancer, and the features you have, among others, are the make of the person's car and whether the person is married. The latter two features are clearly not good predictors for cancer so our machine

learning algorithm will not be able to 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 can have high predictive power if the problem was different.

Another property of a good feature is that it can be computed fast. Let's say you want to predict the topic of a tweet and you use bag of words to represent the 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. The problem of sparse vectors is that the learning algorithm will have a hard time seeing patterns in such vectors because they contain little information compared to their size. The latter property is not the problem per se; the problem is that the information in one sparse vector is rarely contained in the same dimensions as the information in another sparse vector, even if they represent relatively similar concepts.

To reduce sparsity, you might want to augment your sparse feature vectors with additional non-zero values. To do that you might decide to send the text of the tweet to Wikipedia as a search query and then extract additional 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 to your query. For real-time systems, feature extraction must be fast: a less informative feature compensated with more training data is often preferred to a feature with a high predictive power that takes seconds to compute. If your application has to be fast, the features obtained from Wikipedia might not be appropriate for your task.

A good feature has also to be reliable. Again, in our previous Wikipedia example, we cannot have a guarantee that Wikipedia will respond at all: the website can be down, on planned maintenance or the API can currently be temporarily overused and is rejecting some requests. Therefore, we cannot trust that the 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.

It's usually not good if some features are highly correlated. **Correlation** of two features means that their values are related. For example, if the growth of one feature means the growth of the other one and the inverse is also true, then the two features are highly correlated. If you have many correlated features in your feature vector, then tiny changes in feature values might result in significant changes in the predictions. As we will discuss later, once the model is deployed in production, its performance may change because the properties of the input data may change over time. So, when many of your features are highly correlated, a minor change in the properties of the input data may result in major changes in the model's behavior. Feature selection techniques that we consider below help to reduce the number of correlated features.

An important property of a good feature is that the distribution of its values in the training set is similar to the one the model will receive in production. For example, the date of a tweet could be important for some predictions about the tweet. However, if you will apply the model built on the 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 an important error[3].

Finally, features that you design should be unitary, easy to understand and maintain. The property of being unitary means that the feature represents a certain simple to understand and explain quantity. For example, in a problem of classifying a car by type given its characteristics, examples of unitary features are weight, length, width, and color. A feature like "length divided by weight" is not unitary as it's composed of two unitary features. While some learning algorithms can benefit from combining features in some unordinary ways, it's preferable to combine features automatically in a dedicated stage in the model training pipeline. We will consider feature combination and generation of synthetic features from unitary ones later in this chapter.

4.5 Feature Selection

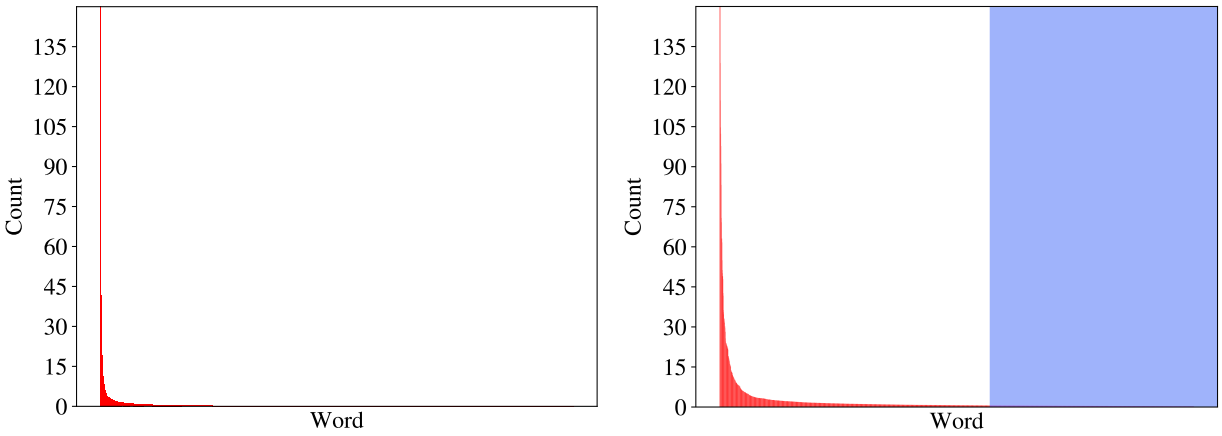
As I already mentioned above, not all features will be equally important for your problem. For instance, in the problem of detecting movies in tweets, the length of the movie might not be a very important feature. At the same time, when you use bag of words, the size of vocabulary can be very big, while 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 one or a couple of training examples, it is very unlikely that the algorithm will learn any useful pattern from that feature. At the same time, if the feature vector is very wide (contains thousands or millions of features) the training time can become very long; furthermore, because of the size of the feature vectors, the overall size of the training data can become too big to fit in the RAM of a conventional server.

If we had the capability to estimate the importance of features, we could keep only the most important ones in our data. That would allow us to save time, fit more examples in memory and even improve the quality of the model.

4.5.1 Cutting the Long Tail

Typically, if a certain feature contains information for a handful of examples, such a feature could be removed from the feature vector. In bag of words, to decide how many times a token has to be seen in the collection of texts to be considered important, you can build a graph with the distribution of token counts and then cut off the so-called long tail as shown in Figure 11.

[3]Date information can still be included in the training data and it's very often relevant for a machine learning model running in production. 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 is very useful.



(a) distribution of word counts in English

(b) the long tail

Figure 11: 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" (the count of 615); the lowest count corresponds to "zambia" (the count of 1).

For a distribution, the **long tail** is 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 account for at least half of all the counts.

The decision on a threshold for defining the long tail is rather subjective. You can, of course, set it as a hyperparameter for your problem. On the other hand, the decision can be made by looking at the distribution of counts, as the one shown in Figure 11. As you can see, I cut off the long tail at a point (Figure 11b) where the distribution of the elements in the tail has become visually flat.

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

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, among others, in Kaggle competitions for estimating the importance of specific features is **Boruta**. Boruta iteratively trains **random forest**

models and runs **statistical tests** to identify features as important or not important. Boruta exists both in the form of an R package and a Python module.

Boruta works as a wrapper around the random forest learning algorithm whence 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**, which consists of making many random samples of the training set and then training 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 generic bagging algorithm is that in the former, the trained statistical models are decision trees and at each split of the decision tree, a random subset of all features is considered.

One useful feature of the random forest is that it has a 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 performs the classification of all training examples from the original training set. Each decision tree in the random forest model contributes its votes only to 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 importance score for a feature, the importance measures for individual trees for that feature 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 scores of the features obtained for individual trees. Then the z-score for that feature is obtained by subtracting the average value from the score and then dividing the result 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 underlining idea of Boruta is simple: we first extend the list of features by adding a randomized copy of each original feature and build a classifier based on this extended dataset. To assess the importance of an original feature we compare it with that of all randomized features. Only features for which the importance is higher than that of the randomized features — and the difference of importance is 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[4] 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 features, both original and replicated.
  - 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. The null hypothesis is that the importance of the feature 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 number of hits. (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 is worth noting though before using Boruta

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

in production. Boruta is a heuristic. There are no theoretical guarantees for its performance. If you want to be sure that Boruta doesn't do harm: run it multiple times and make sure that the feature selection is stable (that is 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 practicians. Below, I describe several other popular methods.

4.5.3 Recursive Feature Elimination

If you use a learning algorithm that, for each feature, computes its weight (such as linear regression or random forest) then you can use a **recursive feature elimination** algorithm to select important features. The algorithm works as follows.

Fix the number d of features you would like to have. First, train a model using the entire set of features. Remove from the set of features of your dataset the feature that received the least weight (to speed up the process, you may decide to remove the k least weighted features). Train a new model using the remaining features and repeat the process until the desired number of features d are remaining.

The optimal number of features to select, d, can be tuned using **grid search** with **cross-validation**, a popular technique of hyperparameter optimization. We will consider it in the next chapter.

4.5.4 Method of Low Sample Variance

A simple method of feature selection is **removing features with low sample variance**. It's very easy to implement. You first compute the sample variance of values of each feature, then set a threshold v. If the sample variance of some feature is below v, you eliminate that feature from your dataset.

To compute the **sample variance** of a certain feature j, where j ranges from 1 to the total number of features D, we will use the entire training data. Let N be the number of training examples and the entire set of examples be $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$. The **sample mean** (or the average) of the feature j, denoted as $\bar{x}^{(j)}$ is given by,

$$\bar{x}^{(j)} \stackrel{\text{def}}{=} \frac{1}{N}\sum_{i=1}^{N} x_i^{(j)} = \frac{1}{N}\left(x_1^{(j)} + x_2^{(j)} + \ldots + x_N^{(j)}\right),$$

where the symbol $\stackrel{\text{def}}{=}$ means "is defined as" and $x_i^{(j)}$ denotes the value of feature j in the training example $\mathbf{x}_i$.

From the sample mean we can now compute the sample variance, denoted as $\sigma^2_{x^{(j)}}$, as follows,

$$\sigma^2_{x^{(j)}} \stackrel{\text{def}}{=} \frac{1}{N}\sum_{i=1}^{N}\left(x_i^{(j)} - \bar{x}^{(j)}\right)^2 = \frac{1}{N}\left(x_1^{(j)} - \bar{x}^{(j)}\right)^2 + \frac{1}{N}\left(x_2^{(j)} - \bar{x}^{(j)}\right)^2 + \ldots + \frac{1}{N}\left(x_N^{(j)} - \bar{x}^{(j)}\right)^2.$$

Consider the dataset from Figure **??** after the imputation, as shown in Figure 12.

Row	Age	Weight	Height	Salary
1	18	70	177	35,000
2	43	65	175	26,900
3	34	87	177	76,500
4	21	66	187	94,800
5	65	60	169	19,000

Figure 12: The dataset from Figure **??** after the imputation.

Let's calculate the sample variance of the feature Height. The sample mean is equal to $\frac{1}{5}(177 + 175 + 177 + 187 + 169) = 177$. The sample variance is then equal to $\frac{1}{5}(177 - 177)^2 + \frac{1}{5}(175 - 177)^2 + \frac{1}{5}(177 - 177)^2 + \frac{1}{5}(187 - 177)^2 + \frac{1}{5}(169 - 177)^2 = 33.6$. If our threshold for v was equal to 50, then we would eliminate the feature Height from the dataset.

Removing features with low sample variance is definitely not the most effective feature selection method[5], but it's simple to implement, so you can use it as a **baseline**: any method of features selection should do at least as well as the method of low sample variance.

4.5.5 Method of Univariate Statistical Test

Another simple method to decide whether a given feature is important is to apply a **univariate statistical test**. As you might remember from statistics classes, there are statistical tests that allow verifying whether two random variables are independent. For example, if one random variable is whether it rains and the second random variable is whether people wear umbrellas, these two random variables are not independent: people often wear umbrellas when it rains and they rarely wear umbrellas when it doesn't rain. On the other hand, if

[5]The method of low sample variance has many major flaws, the biggest one being that, in most cases, different features represent different quantities measured in different units. Thus, their numerical values have very different meanings (e.g. height and salary) and are not directly comparable in terms of mean and variance. In practice, coming with a meaningful threshold for every feature turns out to be a daunting task.

there is a random variable of whether you see a police car, then that random variable is independent of the random variable of seeing umbrellas. By using a statistical test, it can be mathematically verified whether two random variables are independent.

4.5.5.1 Chi-Square Independence Test

One statistical test frequently used in practice is **chi-square independence test**. To understand how it works, let's first define a **random variable**.

A **random variable**, usually written as an italic capital letter, like X, is a variable whose possible values are numerical outcomes of a random phenomenon. Examples of random phenomena with a numerical outcome include a toss of a coin (0 for heads and 1 for tails), a roll of a dice, or the height of the first stranger you meet outside. There are two types of random variables: **discrete** and **continuous**. To understand the functioning of the chi-square independence test, the knowledge of the discrete random variable is sufficient, so I only define this type of random variable below.

A **discrete random variable** takes on only a countable number of distinct values such as *red*, *yellow*, *blue* or 1, 2, 3, ...

The **probability distribution** of a discrete random variable is described by a list of probabilities associated with each of its possible values. This list of probabilities is called a **probability mass function** (pmf). For example: $\Pr(X = red) = 0.3$, $\Pr(X = yellow) = 0.45$, $\Pr(X = blue) = 0.25$. Each probability in a probability mass function is a value greater than or equal to 0. The sum of probabilities equals 1 (Figure 13).

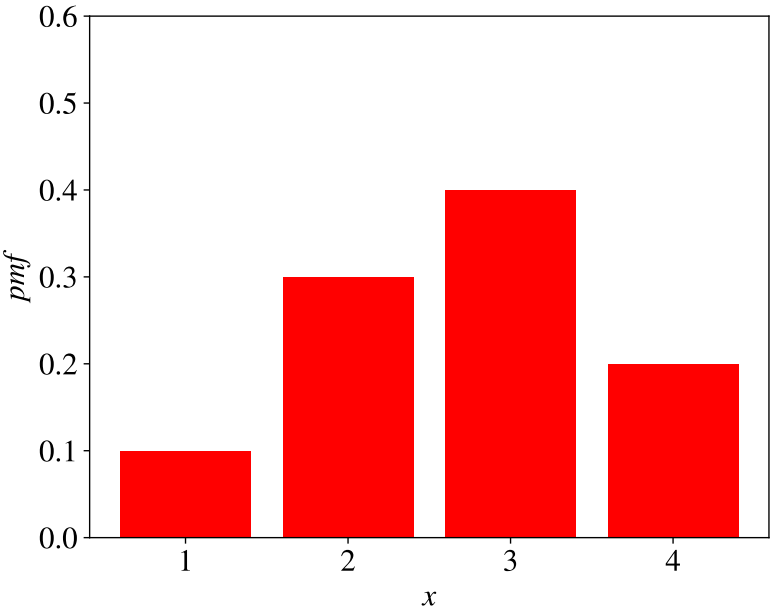


Figure 13: A probability mass function.

A **continuous random variable** takes on infinitely many values such as a distance, a mass, a volume, and so on.

A feature in a dataset can be seen as a random variable. In each feature vector in the dataset, the value of that feature can, therefore, be seen as the observed value of that random variable.

If you see a certain feature in your labeled dataset as a random variable, and the label as another random variable, you can use a statistical test to find out if the two random variables are independent, and, if it's the case, remove the corresponding feature from the dataset as not useful. Indeed, when two random variables are independent, you cannot say anything about the second one by knowing the value of the first one.

The chi-square independence test works as follows. Let's say you have a dataset, a part of which is shown in Figure 14. I have shown only two columns of the entire dataset, one of which, say "Family type," is a feature and the other one, "Car type," is the label we want to learn to predict. By using the chi-square independence test, we want to decide whether we want to remove the feature "Family type" from the dataset or we want to keep it as a useful predictor for the label.

Row	Family type	Car type
1	Single	Sedan
2	Couple	SUV
3	Couple with kids	Minivan
...	...	...
375	Single with kids	SUV

Figure 14: An example of a dataset where the label "Car type" may or may not depend on the feature "Family type."

To compute the value of the chi-square test, we first transform the two random variables into the **table of observed frequencies**, as shown below:

	Coupé	Sedan	SUV	Pickup truck	Minivan	**Total**
Single	23	45	26	11	8	113
Couple	15	44	56	45	26	186
Single with kids	8	11	11	4	3	37
Couple with kids	3	12	11	9	4	39
Total	49	112	104	69	41	375

Family type by car type, N = 375

Figure 15: The table of observed frequencies for two variables.

The value in a cell in the row “Couple” and column “SUV” in the table of observed frequencies is the number of examples in the dataset in which the feature “Family type” and the label “Car type” have the values “Couple” and “SUV” respectively. In our dataset, there are 56 such examples.

From the table of observed frequencies, we can now build the **table of expected frequencies**. This table contains values that we would observe if the two random variables were perfectly independent. This table is shown below:

	Coupé	Sedan	SUV	Pickup truck	Minivan	**Total**
Single	14.8	33.7	31.3	20.8	12.4	113.0
Couple	24.3	55.6	51.6	34.2	20.3	186.0
Single with kids	4.8	11.1	10.3	6.8	4.0	37.0
Couple with kids	5.1	11.6	10.8	7.2	4.3	39.0
Total	49.0	112.0	104.0	69.0	41.0	375.0

Figure 16: The table of expected frequencies built using the table of observed frequencies from Figure 22.

The value e_{ab} in a cell in row a and column b in the table of expected frequencies is given by,

$$e_{ab} = \frac{o_a \cdot o_b}{N},$$

where o_a is the Total value for row a and o_b is the Total value for column b. For example, the value for row "Couple" and column "SUV" in the table of expected frequencies is given by $\frac{186 \cdot 104}{375} = 51.6$.

Once we have both observed and expected frequencies, we can calculate the residuals r_{ab}, that is, differences between observed and expected frequencies. The **table of residuals** for our dataset is shown below,

	Coupé	Sedan	SUV	Pickup truck	Minivan
Single	8.2	11.3	-5.0	-9.8	-4.4
Couple	-9.3	-11.6	4.4	10.8	5.7
Single with kids	3.2	-0.1	0.7	-2.8	-1
Couple with kids	-2.1	0.4	0.2	1.8	-0.3

Figure 17: The table of residuals built using the tables of observed and expected frequencies from Figures 22 and 16.

From the table of residuals and the table of expected frequencies, we can now calculate the

table of chi-square points. Each chi-square point c_{ab} for row a and column b is given by the following equation,

$$c_{ab} \stackrel{\text{def}}{=} \frac{r_{ab}^2}{e_{ab}},$$

where r_{ab} and e_{ab} are, correspondingly, the residual and the expected frequency from the cell in row a and column b. The table of chi-square points for our problem is shown below:

	Coupé	Sedan	SUV	Pickup truck	Minivan
Single	4.5	3.8	0.8	4.6	6.7
Couple	3.6	2.4	0.4	3.4	1.6
Single with kids	2.1	0.0	0.0	1.2	0.3
Couple with kids	0.9	0.0	0.0	0.5	0.0

Figure 18: The table of chi-square points built using the table of residuals and expected frequencies from Figures 22 and 17.

The **chi-square test statistic** χ^2 is then given by the sum of all chi-square points,

$$\chi^2 \stackrel{\text{def}}{=} \sum_a \sum_b c_{ab} = 4.5 + 3.8 + \ldots + 0.5 + 0.0 = 36.8.$$

Now that we have the value of the chi-square test statistic, we can find the probability that the two random variables are independent by looking at the chi-square distribution table shown in Figure 19. To use this table, we need to know the number of **degrees of freedom**. The number of degrees of freedom is a number that determines the exact shape of the chi-square distribution that applies to our test. This number can be obtained from the shape of our table of observed frequencies by using the following equation:

$$\text{d.f.} \stackrel{\text{def}}{=} (\text{number of rows} - 1) \cdot (\text{number of columns} - 1) = 3 \cdot 4 = 12,$$

where d.f. is the number of degrees of freedom of our test.

The probability that our two random variables are independent is the probability from the table in Figure 19 such that d.f. $= 12$ and chi-square value is the rightmost value below 36.8, that is, on our case, 0.01. This means that our two random variables are independent with a

probability below 0.01, which is very low. In this case, we will not remove the feature "Family type" from the dataset because it's very likely to contain information about the label.

d.f.	.995	.99	.975	.95	.9	.1	.05	.025	.01
1	0.00	0.00	0.00	0.00	0.02	2.71	3.84	5.02	6.63
2	0.01	0.02	0.05	0.10	0.21	4.61	5.99	7.38	9.21
3	0.07	0.11	0.22	0.35	0.58	6.25	7.81	9.35	11.34
4	0.21	0.30	0.48	0.71	1.06	7.78	9.49	11.14	13.28
5	0.41	0.55	0.83	1.15	1.61	9.24	11.07	12.83	15.09
6	0.68	0.87	1.24	1.64	2.20	10.64	12.59	14.45	16.81
7	0.99	1.24	1.69	2.17	2.83	12.02	14.07	16.01	18.48
8	1.34	1.65	2.18	2.73	3.49	13.36	15.51	17.53	20.09
9	1.73	2.09	2.70	3.33	4.17	14.68	16.92	19.02	21.67
10	2.16	2.56	3.25	3.94	4.87	15.99	18.31	20.48	23.21
11	2.60	3.05	3.82	4.57	5.58	17.28	19.68	21.92	24.72
12	3.07	3.57	4.40	5.23	6.30	18.55	21.03	23.34	26.22
13	3.57	4.11	5.01	5.89	7.04	19.81	22.36	24.74	27.69
14	4.07	4.66	5.63	6.57	7.79	21.06	23.68	26.12	29.14
15	4.60	5.23	6.26	7.26	8.55	22.31	25.00	27.49	30.58
16	5.14	5.81	6.91	7.96	9.31	23.54	26.30	28.85	32.00
17	5.70	6.41	7.56	8.67	10.09	24.77	27.59	30.19	33.41
18	6.26	7.01	8.23	9.39	10.86	25.99	28.87	31.53	34.81
19	6.84	7.63	8.91	10.12	11.65	27.20	30.14	32.85	36.19
20	7.43	8.26	9.59	10.85	12.44	28.41	31.41	34.17	37.57
22	8.64	9.54	10.98	12.34	14.04	30.81	33.92	36.78	40.29
24	9.89	10.86	12.40	13.85	15.66	33.20	36.42	39.36	42.98
26	11.16	12.20	13.84	15.38	17.29	35.56	38.89	41.92	45.64
28	12.46	13.56	15.31	16.93	18.94	37.92	41.34	44.46	48.28
30	13.79	14.95	16.79	18.49	20.60	40.26	43.77	46.98	50.89
32	15.13	16.36	18.29	20.07	22.27	42.58	46.19	49.48	53.49
34	16.50	17.79	19.81	21.66	23.95	44.90	48.60	51.97	56.06
38	19.29	20.69	22.88	24.88	27.34	49.51	53.38	56.90	61.16
42	22.14	23.65	26.00	28.14	30.77	54.09	58.12	61.78	66.21
46	25.04	26.66	29.16	31.44	34.22	58.64	62.83	66.62	71.20
50	27.99	29.71	32.36	34.76	37.69	63.17	67.50	71.42	76.15
55	31.73	33.57	36.40	38.96	42.06	68.80	73.31	77.38	82.29
60	35.53	37.48	40.48	43.19	46.46	74.40	79.08	83.30	88.38
65	39.38	41.44	44.60	47.45	50.88	79.97	84.82	89.18	94.42
70	43.28	45.44	48.76	51.74	55.33	85.53	90.53	95.02	100.43
75	47.21	49.48	52.94	56.05	59.79	91.06	96.22	100.84	106.39
80	51.17	53.54	57.15	60.39	64.28	96.58	101.88	106.63	112.33
85	55.17	57.63	61.39	64.75	68.78	102.08	107.52	112.39	118.24
90	59.20	61.75	65.65	69.13	73.29	107.57	113.15	118.14	124.12
95	63.25	65.90	69.92	73.52	77.82	113.04	118.75	123.86	129.97
100	67.33	70.06	74.22	77.93	82.36	118.50	124.34	129.56	135.81

Figure 19: Chi-square distribution table.

The chi-square independence test only applies to discrete random variables. If you have a classification problem with real-valued numerical features, such as temperature or weight, you can **discretize** a feature by applying such technique as **binning**. Binning allows transforming a numerical feature into a categorical one by replacing numerical values in a specific range by a constant categorical value. We will consider this technique later in this chapter. Alternatively, for a classification problem with real-valued numerical features, the importance of features can be measured by applying the **one-way ANOVA** technique which we consider below.

4.5.5.2 F-Test: One-Way ANOVA

Consider the following dataset:

Row	Gender	Age	Education
1	M	18	Associate
2	F	25	Undergraduate
3	F	28	Graduate
4	M	31	Graduate
5	F	21	Associate
6	M	26	Undergraduate
7	M	19	Associate
8	F	29	Graduate
9	M	25	Undergraduate
10	F	34	Graduate

Figure 20: A dataset for the problem of the education level prediction.

Let's say we want to predict the education level from age and gender and we want to reduce the dimensionality of this dataset. (It's a toy example.) To do that, we consider the features one by one and estimate their importance for predicting the label. Consider the feature "Age". It's numerical, while the label is categorical. One-way ANOVA allows estimating whether the average age for all examples with label "Associate" is the same as the average age for all examples with the label "Undergraduate" and is the same as the average age for all examples with the label "Graduate". I say "is the same" in the statistically significant sense: the values of the average age for each value of label can be different or same by chance.

One-way ANOVA allows estimating that they are the same with high enough certainty.

Let's see how one-way ANOVA works. First, we calculate the sample mean Age per each label:

$$\begin{aligned}\overline{\text{Age}}_{\text{Associate}} &= (18+21+19)/3 = 19.3,\\ \overline{\text{Age}}_{\text{Undergraduate}} &= (25+26+25)/3 = 25.3,\\ \overline{\text{Age}}_{\text{Graduate}} &= (28+31+29+34)/4 = 30.5.\end{aligned}$$

Then, we calculate the grand mean Age as the weighted average of the sample means,

$$\overline{\text{Age}} = (3 \cdot 19.3 + 3 \cdot 25.3 + 4 \cdot 30.5)/10 = 25.6.$$

Let's denote as $G_{\text{Associate}}$, $G_{\text{Undergraduate}}$, and G_{Graduate}, the groups of values of feature "Age" from examples with labels, repsectively, "Associate," "Undergraduate," and "Graduate." For example, $G_{\text{Associate}} = \{18, 21, 19\}$.

Then, we calculate the **sum-of-squares for between-group variability**, a number that reflects how variable the values of age are between groups, using the following equation,

$$\begin{aligned}\text{SS}_{\text{between}} \stackrel{\text{def}}{=} & N_{\text{Associate}} \cdot (\overline{\text{Age}}_{\text{Associate}} - \overline{\text{Age}})^2\\ & + N_{\text{Undergraduate}} \cdot (\overline{\text{Age}}_{\text{Undergraduate}} - \overline{\text{Age}})^2\\ & + N_{\text{Graduate}} \cdot (\overline{\text{Age}}_{\text{Graduate}} - \overline{\text{Age}})^2,\end{aligned}$$

where $\text{SS}_{\text{between}}$ denotes the sum-of-squares for between-group variability and $N_{\text{Associate}}$, $N_{\text{Undergraduate}}$ and N_{Graduate} are, respectively, the number of examples with labels "Associate," "Undergraduate" and "Graduate" in the dataset. By putting the numbers into the above equation, we obtain,

$$\text{SS}_{\text{between}} = 3 \cdot (19.3 - 25.6)^2 + 3 \cdot (25.3 - 25.6)^2 + 4 \cdot (30.5 - 25.6)^2 = 215.4.$$

From $\text{SS}_{\text{between}}$, we calculate $\text{MS}_{\text{between}}$, the **mean square for between-group variability**,

$$\text{MS}_{\text{between}} \stackrel{\text{def}}{=} \frac{\text{SS}_{\text{between}}}{\text{d.f.between}},$$

where d.f.between is the number of degrees of freedom, which is equal to the number of sample means minus one. Therefore,

$$\text{MS}_{\text{between}} = 215.4/2 = 107.7. \tag{1}$$

$\text{MS}_{\text{between}}$ is a number that measures how three different groups of numbers vary with respect to one another. Now, let's measure a within-group variability, that is how numbers vary within each group. This measure is denoted as $\text{MS}_{\text{within}}$ and is found as follows. First, we find the **sum of squares for within-group variability**, $\text{SS}_{\text{within}}$, defined as,

$$\begin{aligned}\text{SS}_{\text{within}} \stackrel{\text{def}}{=} &\sum_{\text{Age}\in G_{\text{Associate}}} (\text{Age} - \overline{\text{Age}}_{\text{Associate}})^2 + \sum_{\text{Age}\in G_{\text{Undergraduate}}} (\text{Age} - \overline{\text{Age}}_{\text{Undergraduate}})^2 \\ &+ \sum_{\text{Age}\in G_{\text{Graduate}}} (\text{Age} - \overline{\text{Age}}_{\text{Graduate}})^2.\end{aligned}$$

By putting numbers in the above equation, we obtain,

$$\begin{aligned}\text{SS}_{\text{within}} =&(18-19.3)^2 + (21-19.3)^2 + (19-19.3)^2 \\ &+(25-25.3)^2 + (26-25.3)^2 + (25-25.3)^2 \\ &+(28-30.5)^2 + (31-30.5)^2 + (29-30.5)^2 + (34-30.5)^2 = 26.3.\end{aligned}$$

The **mean square for within-group variability**, $\text{MS}_{\text{within}}$ is then given by,

$$\text{MS}_{\text{within}} \stackrel{\text{def}}{=} \frac{\text{SS}_{\text{within}}}{\text{d.f.within}}, \tag{2}$$

where the number of degrees of freedom d.f.within is given by N minus the number of sample means, that is $\text{d.f.within} = 10 - 3 = 7$. The value of $\text{MS}_{\text{within}}$ for our problem is, therefore, equal to,

$$\text{MS}_{\text{within}} = 26.3/7 = 3.8.$$

The F-statistic is then found by dividing the between-group variability by the within-group variability:

$$F_{\text{Age,Education}} \stackrel{\text{def}}{=} \frac{\text{MS}_{\text{between}}}{\text{MS}_{\text{within}}} = 107.7/3.8 = 28.3.$$

Having the F-statistic and the values of the degrees of freedom, we can now find the probability that the three groups of age values come from the same statistical distribution (and thus are useless for classification) or the three groups are statistically significantly different (and thus contain information useful to predict the label). That probability can be found in a so-called F-distribution table or, simply, F-table. You can find more complete F-tables online or in the statistical books. Here, I only show a part of it useful for our problem (Figure 21).

d.f.1	1	2	3	4	5	6	7	8	9
d.f.2									
1	161.4	199.5	215.7	224.6	230.2	234.0	236.8	238.9	240.5
2	18.51	19.00	19.16	19.25	19.3	19.33	19.35	19.37	19.38
3	10.13	9.55	9.28	9.12	9.01	8.94	8.89	8.85	8.81
4	7.71	6.94	6.59	6.39	6.26	6.16	6.09	6.04	6.00
5	6.61	5.79	5.41	5.19	5.05	4.95	4.88	4.82	4.77
6	5.99	5.14	4.76	4.53	4.39	4.28	4.21	4.15	4.10
7	5.59	4.74	4.35	4.12	3.97	3.87	3.79	3.73	3.68
8	5.32	4.46	4.07	3.84	3.69	3.58	3.50	3.44	3.39
9	5.12	4.26	3.86	3.63	3.48	3.37	3.29	3.23	3.18
10	4.96	4.10	3.71	3.48	3.33	3.22	3.14	3.07	3.02
11	4.84	3.98	3.59	3.36	3.20	3.09	3.01	2.95	2.90
12	4.75	3.89	3.49	3.26	3.11	3.00	2.91	2.85	2.80
13	4.67	3.81	3.41	3.18	3.03	2.92	2.83	2.77	2.71
14	4.60	3.74	3.34	3.11	2.96	2.85	2.76	2.70	2.65
15	4.54	3.68	3.29	3.06	2.90	2.79	2.71	2.64	2.59

(a) significance level is 0.05

d.f.1	1	2	3	4	5	6	7	8	9
d.f.2									
1	4052	4999.5	5403	5625	5764	5859	5928	5982	6022
2	98.50	99.00	99.17	99.25	99.30	99.33	99.36	99.37	99.39
3	34.12	30.82	29.46	28.71	28.24	27.91	27.67	27.49	27.35
4	21.20	18.00	16.69	15.98	15.52	15.21	14.98	14.80	14.66
5	16.26	13.27	12.06	11.39	10.97	10.67	10.46	10.29	10.16
6	13.75	10.92	9.78	9.15	8.75	8.47	8.26	8.10	7.98
7	12.25	9.55	8.45	7.85	7.46	7.19	6.99	6.84	6.72
8	11.26	8.65	7.59	7.01	6.63	6.37	6.18	6.03	5.91
9	10.56	8.02	6.99	6.42	6.06	5.80	5.61	5.47	5.35
10	10.04	7.56	6.55	5.99	5.64	5.39	5.2	5.06	4.94
11	9.65	7.21	6.22	5.67	5.32	5.07	4.89	4.74	4.63
12	9.33	6.93	5.95	5.41	5.06	4.82	4.64	4.50	4.39
13	9.07	6.70	5.74	5.21	4.86	4.62	4.44	4.30	4.14
14	8.86	6.51	5.56	5.04	4.69	4.46	4.28	4.14	4.03
15	8.68	6.36	5.42	4.89	4.56	4.32	4.14	4.00	3.89

(b) significance level is 0.01

Figure 21: F-table for significance levels of 0.05 and 0.01.

In the F-table, we have to find the number corresponding to d.f.1 = 2 (the number of degrees of freedom in the denominator of eq. 1) and d.f.2 = 7 (the number of degrees of freedom in

the denominator of eq. 2). For the significance level of 0.01, this value is 9.55. The value of our F-statistic, 28.3, is greater than 9.55, which means that there's less than 1% chance that the values of Age in the three groups come from the same probability distribution. It's a tiny chance and, therefore, we will keep the feature "Age" in our dataset.

4.5.5.3 Univariate Linear Regression Test For a regression problem, where, in a general case, both the target and the features are real-valued, a **univariate linear regression test**-based approach for feature selection is often used. To find the ***F*-statistic** for two continuous random variables, $X^{(j)}$, representing a certain feature j and Y, representing the target in our dataset, we proceed as follows. Compute the correlation $r_{X^{(j)},Y}$ of $X^{(j)}$ with Y,

$$r_{X^{(j)},Y} \stackrel{\text{def}}{=} \frac{\sum_{i=1}^{N}\left(x_i^{(j)} - \bar{X}^{(j)}\right)\left(y_i - \bar{Y}\right)}{\sqrt{\sum_{i=1}^{N}\left(x_i^{(j)} - \bar{X}^{(j)}\right)^2}\sqrt{\sum_{i=1}^{N}\left(y_i - \bar{Y}\right)^2}},$$

where N is the number of examples in the dataset while $x_i^{(j)}$ and y_i are, respectively, the value of feature j and the value of target in the i^{th} example; $\bar{X}^{(j)}$ and $\bar{Y}$ are, respectively, the sample means of feature j and the target; i ranges from 1 to N.

Once we have the correlation between two variables, we can convert it into an F-statistic as follows,

$$F_{X^{(j)},Y} \stackrel{\text{def}}{=} \frac{r^2_{X^{(j)},Y}}{1 - r^2_{X^{(j)},Y}} \cdot (N-2),$$

where $N-2$ is the number of degrees of freedom.

To illustrate the above calculations, consider once again the toy dataset from Figure 12. Let's compute the F-statistic for the feature Age assuming that Salary is the target. First, we compute the sample means:

$$\begin{aligned}\overline{\text{Age}} &= (23+18+44+53+16)/5 = 30.8,\\ \overline{\text{Salary}} &= (35000+26900+76500+94800+19000)/5 = 50440.\end{aligned}$$

$$r_{\text{Age,Salary}} = \frac{(23-30.8)(35000-50440)+\cdots+(16-30.8)(19000-50440)}{\sqrt{(23-30.8)^2+\cdots+(16-30.8)^2}\sqrt{(35000-50440)^2+\cdots+(19000-50440)^2}} = 0.999.$$

$$F_{\text{Age,Salary}} = \frac{(-0.999)^2}{1-(-0.999)^2}\cdot(5-2) = 1712.8.$$

By using the F-table from Figure 21 for significance level of 0.05, we can find the value corresponding to $d.f.1 = N - 2 = 3$ and $d.f.2 = 1$[6], which is equal to 215.7 and is below 1712.8. Once again, there's a probability below 5% that the correlation between the values of "Age" and "Salary" was observed by chance. Because this probability is tiny, we conclude that the feature "Age" is likely important to predict "Salary" and we keep the feature "Age" in our dataset. Please note that for the significance level of 0.01, the value in the F-table is 5403, which is above 1712.8. That means that if we required the statistical significance level of 0.01, we should have concluded that there's not enough evidence that the age and salary are correlated. In practice, however, 0.05 is enough to decide to keep the feature: there's only one chance of 20 that this feature will not be useful.

Alternatively, after you calculated the F-statistic for each feature, you can keep d features with the highest values of the F-statistic. The optimal number of features to select can, once again, be tuned using grid search with cross-validation.

4.5.6 Method of Mutual Information

Mutual information (MI) of two random variables is a measure of the interdependence between the two variables. MI quantifies the "amount of information" that can be obtained about one random variable through observing the other one. If two random variables are absolutely independent, that is you cannot say anything about the value of the second variable by seeing the value of the first one, then the mutual information of such two random variables is zero. Otherwise, MI is a positive number. The more the two variables are interdependent, the higher is the value of MI.

If we have the **pmf** of the two discrete random variables X and Y, then we can calculate $\mathrm{MI}_{X,Y}$ as,

$$\mathrm{MI}_{X,Y} \stackrel{\text{def}}{=} \sum_{x \in X} \sum_{y \in Y} \Pr(X = x, Y = y) \log_2 \frac{\Pr(X = x, Y = y)}{\Pr(X = x) \Pr(Y = y)},$$

where $\sum_{x \in X}$ means that the summation is performed over all possible values x of the random variable X.

It's easy to calculate the mutual information for two categorical features because such values as $\Pr(X = x, Y = y)$, $\Pr(X = x)$, and $\Pr(Y = y)$ can be obtained from the dataset. It will not be exact probabilities, because our dataset is of finite size, but the bigger the dataset, the closer the values obtained from it will be to the real values.

Consider once again, the examples of predicting the car type knowing the family type and other attributes. Let us want to decide whether the feature "Family type" has high mutual information with the label "Car type". For convenience, I repeat the table of observed frequencies below:

[6] As per the implementation of the method in scikit-learn.

	Coupé	Sedan	SUV	Pickup truck	Minivan	**Total**
Single	23	45	26	11	8	113
Couple	15	44	56	45	26	186
Single with kids	8	11	11	4	3	37
Couple with kids	3	12	11	9	4	39
Total	49	112	104	69	41	375

Family type by car type, N = 375

Figure 22: The table of observed frequencies for two variables.

For example, from the above table we can calculate $\Pr(\text{Car type} = \text{Coupé}) = 49/375 = 0.13$, $\Pr(\text{Family type} = \text{Single}) = 113/375 = 0.3$, $\Pr(\text{Car type} = \text{Coupé}, \text{Family type} = \text{Single}) = 23/375 = 0.06$, and so on. We can continue calculations for each unique pair of values of Car type and Family type from left to right, top to bottom until we reach the last one (Couple with kids, Minivan) for which the corresponding probabilities will be $\Pr(\text{Car type} = \text{Minivan}) = 41/375 = 0.11$, $\Pr(\text{Family type} = \text{Couple with kids}) = 39/375 = 0.1$, $\Pr(\text{Car type} = \text{Minivan}, \text{Family type} = \text{Couple with kids}) = 4/375 = 0.01$. We then calculate $\text{MI}_{Familytype,Cartype}$ as,

$$\text{MI}_{Family\ type,Car\ type} = 0.06 \cdot \log_2 \frac{0.06}{0.13 \cdot 0.3} + \cdots + 0.01 \cdot \log_2 \frac{0.01}{0.11 \cdot 0.1} = 0.06.$$

Now, as you calculated the mutual information with the target for all features, you can keep d features with the highest values of MI. The optimal number of features to select can, once again, be tuned using grid search with cross-validation.

Computing mutual information when one of the random variables is continuous is trickier, because, in that case, you cannot build a table of observed frequencies. You could, indeed, discretize a feature by using a technique like binning that we will consider below, but if the label is numerical like the target in regression problems, you wouldn't want to discretize it.

In the case where you have one discrete and one continuous random variable, you can apply the **nearest neighbor method** to estimate the mutual information for these two random variables. The method works as follows[7]. Let X be our discreet random variable and Y be a continuous one. Notice that in each example i in our dataset, the random variable X has a value x_i and the random variable Y has a value y_i. For each example i, we will compute a

[7]Based on the paper "Mutual Information between Discrete and Continuous Data Sets" by Brian Ross published in PLoS ONE 9(2) in 2014.

number I_i based on its nearest-neighbors in the continuous variable Y. To do that, we first find the k^{th}-closest neighbor to the example i among those N_{x_i} data points whose value of the discrete variable X equals x_i. To find the closest values we can use some distance metric of our choice. Let the full dataset be as shown in the top line in Figure 23. The N_{x_i} data points are shown in Figure 23 in the bottom line.

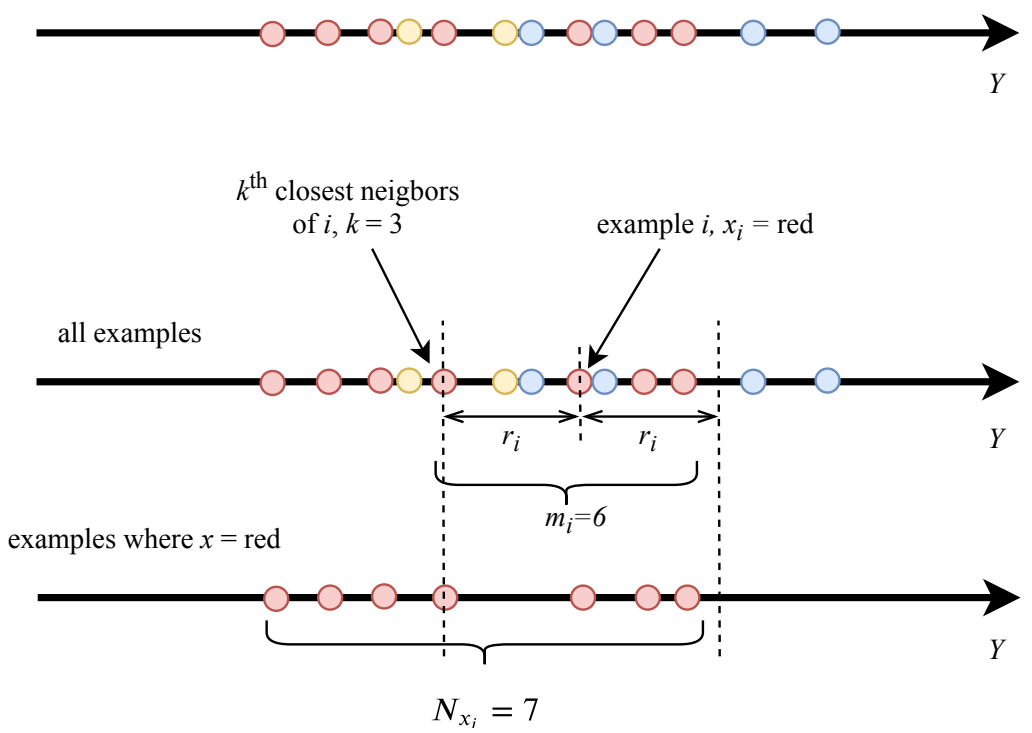


Figure 23: An example of the computation of I_i using the nearest neighbor method.

Define r_i as the distance to this k^{th}-closest neighbor. We then count the number of neighbors m_i in the full data set that lie within distance r_i to point i, including the k^{th}-closest neighbor itself as shown in Figure 23 in the middle line. Based on N_{x_i} and m_i we compute,

$$I_i \leftarrow \psi(N) - \psi(N_{x_i}) + \psi(k) - \psi(m_i),$$

where $\psi(\cdot)$ is the **digamma function** defined for all positive integer numbers x as,

$$\psi(x) \stackrel{\text{def}}{=} -\gamma + \sum_{p=1}^{\infty} \frac{x-1}{p \cdot (x+p-1)},$$

where γ is **Euler's constant**. The digamma function is implemented in most scientific packages and is, in particular, available in Python and R, so you don't have to implement the above infinite summation.

Once we have computed I_i for each example i from 1 to D, we can now find the mutual information $\text{MI}_{X,Y}$ between the variables X and Y as the average among all I_i,

$$\text{MI}_{X,Y} \stackrel{\text{def}}{=} \frac{1}{D}\sum_{i=1}^{D} I_i.$$

We can optimize k and the number d of top-performing features to keep as hyperparameters by using grid search with cross-validation.

4.5.7 L1-Regularization

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

While regularization doesn't let you choose which features to keep in the feature vector, 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 is the same: L1 penalizes the model for being too complex.

To understand the effect of L1 on the model, consider the **linear regression** objective:

$$\min_{\mathbf{w},b} \frac{1}{N}\sum_{i=1}^{N}(f_{\mathbf{w},b}(\mathbf{x}_i) - y_i)^2, \tag{3}$$

where $\mathbf{w}$ and b are, respectively, the vector of parameters and the bias term we want to learn from data and they entirely define our model $f_{\mathbf{w},b}$.

An L1-regularized objective looks like this:

$$\min_{\mathbf{w},b} \left[C|\mathbf{w}| + \frac{1}{N}\sum_{i=1}^{N}(f_{\mathbf{w},b}(\mathbf{x}_i) - y_i)^2 \right], \tag{4}$$

where $|\mathbf{w}| \stackrel{\text{def}}{=} \sum_{j=1}^{D} |w^{(j)}|$, $|\cdot|$ denotes the absolute value, and C is a hyperparameter that controls the importance of regularization. If we set C to zero, the model becomes a standard non-regularized linear regression model. On the other hand, if we set C to a high positive value, the learning algorithm will try to set most $w^{(j)}$ to zero to minimize the objective, and

the model will become very simple which can lead to **underfitting** (the situation when the model performs poorly on the training data). Your role as the data analyst is to find such a value of C that provides the best generalization for the problem at hand. In the next chapter, we will see how to do that.

In practice, L1 regularization produces a **sparse model**, that is a model that has most of its parameters (in case of linear models, most of $w^{(j)}$) equal to zero, provided the hyperparameter C is large enough. Therefore, L1 performs feature selection by deciding which features are essential for prediction and which are not. That can be useful in case you want to increase model explainability.

4.6 Synthesizing Features from Relational Data

Data analysts often work with data that is contained 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**. To build a classifier that will predict whether a given customer will soon leave, we have to represent a customer with a set of features. Let's say the data on the users is contained in three tables: User, Order, and Call as shown in Figure 24.

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 24: Relational data for churn analysis.

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 rows 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 four additional features for users 2 and 4. You can see the values of the additional features for these two users in Figure 25.

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 25: Synthetic features based on sample mean and standard deviation.

Sometimes, a relational database can have a deeper structure. For example, a user can have orders, while each order can have ordered items. In this case, to create a feature 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 generated features based on all possible combinations of statistics, you can select the most useful ones by using one of the feature selection methods discussed above.

4.7 Feature Discretization

Despite the fact that most modern learning algorithms work with numerical features (the learning algorithms implemented in the most popular ML package for Python, **scikit-learn**, only work with numerical features), it can still be useful to convert numerical features into categorical ones. The reasons to discretize a real-valued numerical feature can be numerous. We have already seen above two such reasons: **chi-square independence test** can only apply to discrete random variables, and it's also simpler to calculate the **mutual information** between two discrete random variables. A successful discretization also adds

useful additional 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 to interpret the prediction of a model if the prediction is based on discrete groups of values such as age groups or geographical regions.

Binning, also known as **bucketing**, is a popular technique 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,
- quantile-based binning.

In all three cases, you should decide how many bins you want to have. Consider an illustration in Figure 26. 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 26 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 26 in the middle. In quantile-based binning, all bins have the same number of examples as shown in Figure 26 at the bottom.

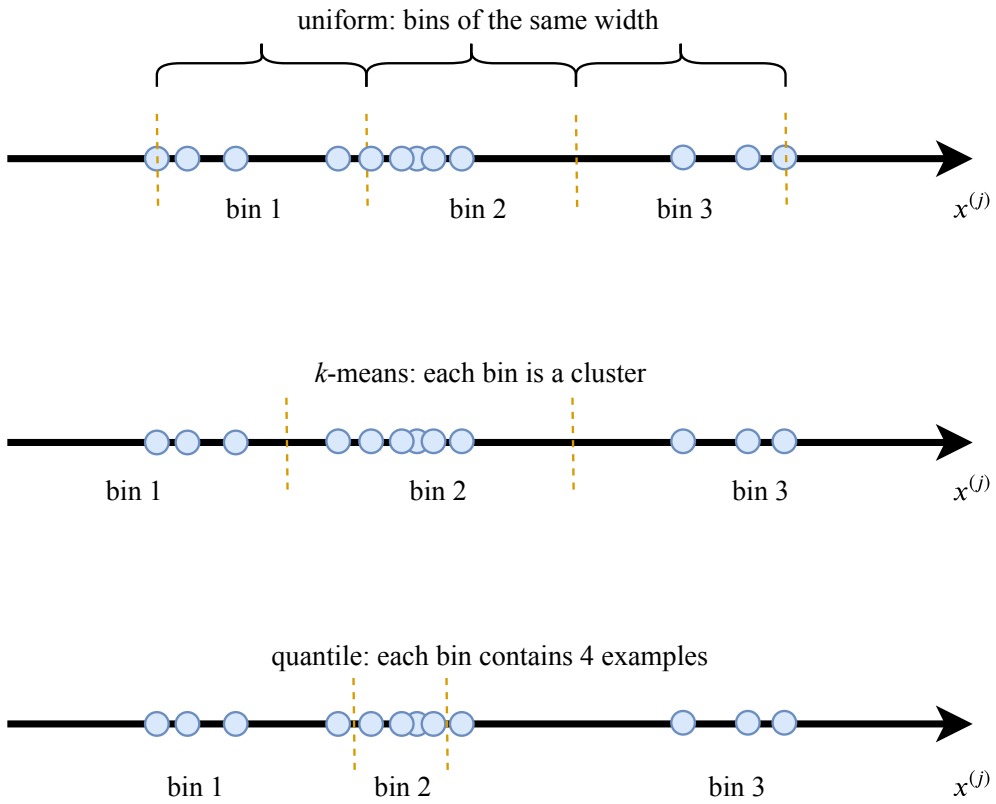


Figure 26: Illustrations of three binning approaches: uniform, k-means-based and quantile-based.

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.

Because, as I already mentioned, most modern machine learning algorithm implementations require numerical features, the bins have to be transformed into binary values by using a technique like one-hot encoding.

4.8 Synthesizing Features from Other Features

Neural networks are notoriously known for their ability to learn complex features by combining simple features in unordinary ways by letting the values of simple features undergo several levels of nested non-linear transformations. If you have data in abundance, you can train a

deep **multilayer perceptron** (MLP) model that will learn to combine in a clever way the basic unitary features it receives as input.

If you don't have an infinite supply of training examples (which is often the case in practice) very deep neural networks lose their appeal. 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 used 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 the values 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 $\div$. 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 transformations) and then, by using one of the feature selection methods discussed above, select those that increase the quality of the model.

4.9 Learning Features from Data

4.9.1 Word Embeddings

In the previous chapter, we have already mentioned word embeddings in the data augmentation context. **Word embeddings** are feature vectors that represent words. They have the property that 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, where a shallow neural network with one hidden layer (called the **embedding layer**) is trained to predict a word give it's surrounding words or to predict the surrounding words, given a 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. The most widely used algorithm, invented at Google with the code available in open source, is **word2vec**. Pretrained word2vec embeddings for many languages are available to download online.

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 with an example of a 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 zeroes 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." Now, consider the same sentence from which we have removed one word, say "book." Our sentence becomes: "I am attentively reading the · on machine learning." Now let's only keep the three words before the · and three words after: "attentively reading the · on machine learning." Looking at this seven-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: because they share similar contexts in multiple texts.

It turns out that it works the other way around too: a word can predict the context that surrounds it. The piece "finished reading the · on machine learning" is called a skip-gram with window size 7 (3 + 1 + 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 like this: $[\mathbf{x}_{-3}, \mathbf{x}_{-2}, \mathbf{x}_{-1}, \mathbf{x}, \mathbf{x}_{+1}, \mathbf{x}_{+2}, \mathbf{x}_{+3}]$. In our sentence, $\mathbf{x}_{-3}$ is the one-hot vector for "finished," $\mathbf{x}_{-2}$ corresponds to "reading," $\mathbf{x}$ is the skipped word $(\cdot)$, $\mathbf{x}_{+1}$ is "on" and so on. A skip-gram with window size 5 will look like this: $[\mathbf{x}_{-2}, \mathbf{x}_{-1}, \mathbf{x}, \mathbf{x}_{+1}, \mathbf{x}_{+2}]$.

The skip-gram model with window size 5 is schematically depicted in Figure 27. It is a fully-connected network, like the multilayer perceptron. The input word is the one denoted as · in the skip-gram. The neural network has to learn to predict the context words of the skip-gram given the central word.

You can see now why the learning of this kind is called **self-supervised**: the labeled examples get extracted from the unlabeled data such as text.

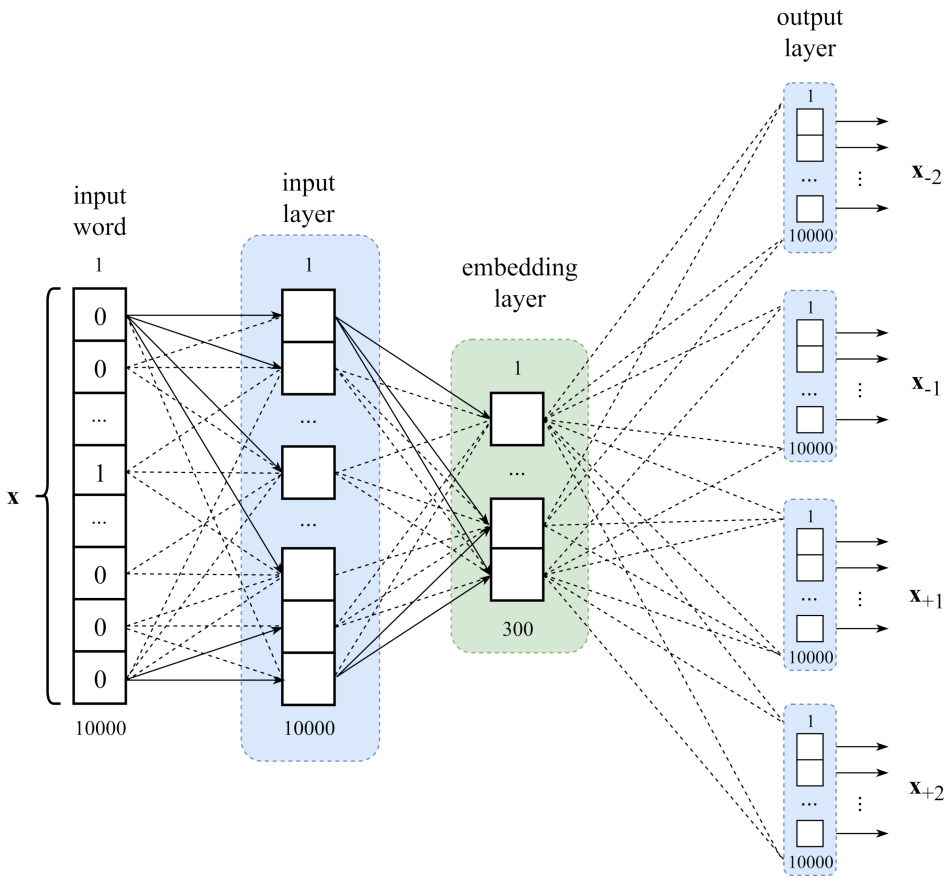


Figure 27: The skip-gram model with window size 5 and the embedding layer of 300 units.

The activation function used in the output layer is softmax. The cost function is the negative log-likelihood. The embedding for a word is obtained as the output of the embedding layer when the one-hot encoding of this 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 it 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, even if that word doesn't belong to the corpus used to build the embeddings. One such architecture, the most frequently 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 like a unitary entity and learns a vector for each word. On the other hand, 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 is an effective way of representing natural language texts for using in such neural network architectures as recurrent neural networks (RNN) and convolutional neural networks (CNN) which are adapted for working with sequences. However, if you want to use them 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 sum or average. The representation of a text document obtained as an average of the words composing that document is not very useful in practice.

4.9.2 Document Embeddings

A popular way of obtaining an embedding for an entire document is to use a **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 en input word is done by first averaging the two embedding vectors (that is the document and the word embedding vectors) and then predicting the surrounding words from that average. In order to be able to average the two vectors, they have to be of the same dimensionality. Interestingly that the latter property 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 have exactly the same properties as the word vectors 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 and gets a new document ID assigned to it. Then the existing neural network model is additionally trained for several epochs with all trained parameters being frozen but the new ones, corresponding to the new document ID. The document ID on the input of the neural network is provided as a one-hot encoding.

4.9.3 Embeddings of Anything

To obtain embedding vectors for any object and not just words or documents, the following technique is commonly used. 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 model, similar to **AlexNet** is often used to train embeddings for images. An illustration of the embedding layers for images is shown in Figure 28. In this illustration, we have a deep confolutional neural network 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 which was 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 one of the fully connected layers is better depends on the task you want to solve with those embeddings and has to be decided experimentally.

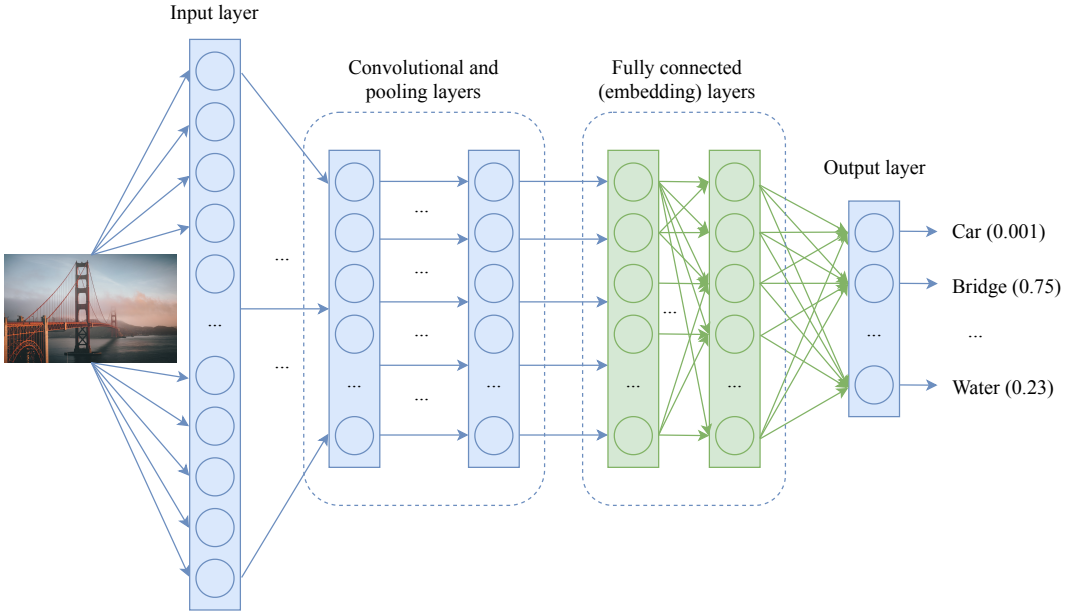


Figure 28: A neural network architecture for training image embeddings. The embedding layers are shown in green.

By following the above approach, embeddings of any type of objects can be trained. The only three things that have to be figured out by the data analyst are:

- which supervised learning model we want to solve (for images, it's object classification),
- how to represent the input for the neural network (for images, it's matrices of pixels, one per channel), and

- what will be the architecture of the neural network before the fully connected layers (for images it's a deep CNN).

4.10 Dimensionality Reduction

Sometimes, it might be necessary to reduce the dimensionality of the examples. It 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 by a new vector, of lower dimensionality, of synthetic features.

Besides obtaining obvious advantages such as the reduced size of the data, increased speed of learning, and sometimes better accuracy, nowadays, a typical reason for dimensionality reduction is the visualization of the dataset: humans can only see data in at most three dimensions.

There are several techniques of dimensionality reduction, each being more popular than others depending on why we want to reduce dimensionality. Those techniques are usually well described in the books on machine learning theory, so, in this section, I will only discuss when a data analyst should use one technique or another.

Principal Component Analysis (PCA) is the oldest among the techniques of dimensionality reduction. It is also by far the faster 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 the model building in your machine learning pipeline, and find the optimal value of the reduced dimensionality experimentally as part of the hyperparameter tuning process.

PCA's biggest disadvantage 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 bathes of the dataset, loading in memory one batch at a time, but Incremental PCA is an order of magnitude slower than PCA. PCA is also less appropriate for visualization purposes.

If visualization is your goal, then you would prefer the **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 they produce very different looking visualizations, so you would prefer one over another based on the properties of the specific dataset you work with. Furthermore, similarly to PCA, UMAP requires all data to be in memory, while autoencoder can read data in batches.

4.11 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 distribution. Multiple experiments demonstrated that a learning algorithm may produce a better model when training on scaled features. While there's no guarantee that scaling will have a positive impact on the quality of your model, it's considered a good practice to do so. Scaling can also increase the speed of training 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 which computers have when working with very small or very big numbers.

4.11.1 Normalization

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

For example, let the natural range of a particular feature be 350 to 1450. By subtracting 350 from every value of the feature, and dividing the result by 1100, one can normalize those values to the range $[0, 1]$.

More generally, the normalization formula looks like this:

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

where $min^{(j)}$ and $max^{(j)}$ are, respectively, the minimum and the maximum value of the feature j in the dataset.

4.11.2 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 mean (the average value of the feature, averaged over all examples in the dataset) and σ is the standard deviation from the mean.

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

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

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.

You may wonder when you should use normalization and when standardization. There's no definitive answer to this question. 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.

If you don't have time to run multiple experiments, as a rule of thumb:

- unsupervised learning algorithms, in practice, more often benefit from standardization than from normalization;
- standardization is also preferred for a feature if the values this feature takes are distributed close to a normal distribution (so-called bell curve);
- again, standardization is preferred for a feature if it can sometimes have extremely high or low values (outliers); this is because normalization will "squeeze" the normal values into a very small range;
- in all other cases, normalization is preferable.

Feature rescaling is usually beneficial to most learning algorithms.

4.12 Data Leakage in Feature Engineering

Data leakage during the feature engineering can happen in several situations. For an arbitrary dataset, it can happen during discretization and scaling. Imagine that you use your entire dataset to calculate the ranges of each bin or the feature scaling factors and 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. This means that the learning algorithm will be exposed to the information it is not supposed to be exposed to. While it seems like not significant leakage if your dataset is small enough that might result in an overly optimistic performance of your model on the holdout data.

A similar form of leakage may happen when you work with text and use bag-of-words to create features. Imagine that you use the entire dataset to build the dictionary and then 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 documents from the holdout sets. Again, the model you will obtain might display better performance on the holdout data than the performance you would observe had you divided your data before feature engineering.

A solution to this form of leakage, as you might already have guessed, is to first split the entire dataset into training and holdout sets and only then do feature engineering based on the training data only. This will also help 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.

4.13 Storing and Documenting Features

Even if you plan to work on the model right after you finished engineering features, it's recommended to design a **schema file** that provides a description of the expected properties of the features. This document has to be machine-readable, versioned and updated each time you make 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 (or to have a non-zero value)
  - The minimum and maximum value
  - Sample mean and variance
  - Does it allow zeroes
  - Does it allow undefined values

En 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
}
```

Large and distributed organizations may use a **feature store** that allows keeping, documenting, reusing and sharing features across multiple data science teams and projects. In such organizations, how features are maintained and served can differ significantly across projects and teams. That difference introduces infrastructure complexity and often results in duplicated work. Below are several challenges, related to features, large distributed organizations face:

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.
Many machine learning models that are designed to apply to real-time data are not using informative but computationally intensive features. Having those features available in a fast store would allow using such features not only in batch mode.

Inconsistency between training and serving
The model is usually trained by using the historical data, while when it's served, it gets 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, the same feature must have exactly the same value for the same input data entity, both on the offline (development) and online (production) mode.

Feature expiration is unknown
. When, in the production environment, a new input example comes in, 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 in a feature store is described by four elements: 1) name, 2) description, 3) metadata, and 4)definition.

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

The description of a feature is an informal textual description of the property of an object that feature represents, for example, "The average length of the session for a user." or "The number of words in the document."

Metadata can contain the following information about the feature: the input type (e.g. numerical, string, image), the output type (e.g., numerical, categorical, numerical vector), whether the value of the feature must be cached by the feature store, and if yes, for how long. A feature can also be marked as available either for both online and offline, or just 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. Of course, a value of some feature for a given input example can only be read from a cache or a value store if the values of that feature for all possible inputs were already precrunched; this is only possible if the input has a finite domain such as user IDs or words of the English language. Examples of features that can be computed in real-time include squaring the input number, determining the shape of the word, or making a search in a search engine.

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

Feature store allows data engineers to insert features, while data analysts and machine learning engineers can use an API to get values of features which they deem relevant for their problem. A feature store provides a way to obtain features for a batch of inputs (for example, when the analyst works on a model offline and wants to convert the training data into a collection of feature vectors) or a single input (when the features have to be obtained for an online input).

If necessary, not just the feature definition but also the values of features can be versioned. If it's the case, then after the value of a given feature for a given input is updated, the previous value is not erased but rather saved with a timestamp indicating when that value was generated. Feature value versioning is useful for reproducibility because it provides the data analyst with a possibility to rebuild the model by using the same values of features as the values that were used to build the previous version of the model.

The place of the feature store in the overall machine learning pipeline is shown in Figure 29.

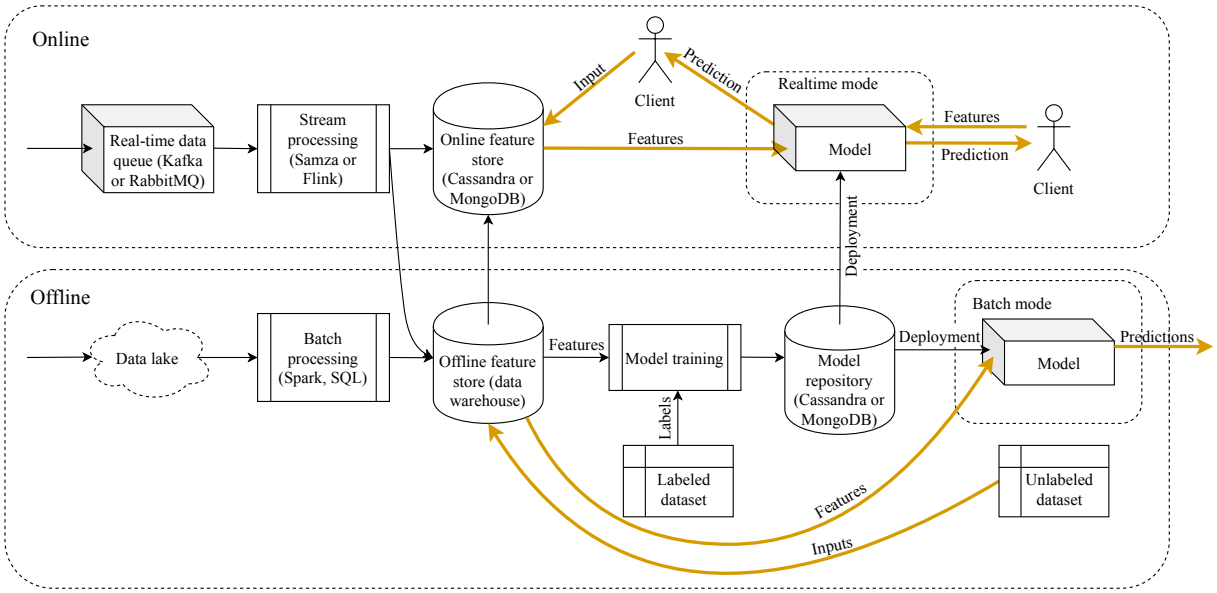


Figure 29: The place of the feature store in the overall machine learning pipeline.

In Figure 29, the architecture, inspired by Uber's Michelangelo machine Learning platform, 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, while the offline feature store is updated in batch-mode by using values of some features computed online, as well as 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 one or several times a day.

4.14 Feature Engineering Best Practices

The feature engineering code must be carefully tested. A unit tests have to exist to cover each feature extractor. One broken feature extractor can result in arbitrarily bad performance of the model, feature extractors is the first place to look for if the problem's behavior is strange.

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

Once deployed in the production environment, and each time the model is loaded, tests of each feature extractor have to be run. If a feature consumes some external resources like a

database or an API, these resources might be unavailable in production.

The version of the feature code has to be in sync with the version of the model and the data used to build the model. The three have to be deployed or rolled back at the same time. Each time the model is loaded in production, a code has to check that the three elements are in sync, that is their versions are the same.

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

Always normalize or standardize features. Only do feature selection when it's strictly necessary. The reason for feature selection could be 1) the need to have an explainable model (so you keep the most significant predictors) or 2) hardware requirements, such as RAM, hard drive space 3) time available to experiment and/or rebuild the model in production. If you decide to do feature selection, start with Boruta.

Use categorical features with many values (more than a dozen) only when you want the model to have different "modes" of behavior, depending on the value of that categorical feature. Typical examples of categorical features with many values are zip code (postal code) or country. You may want to use the categorical feature "Country" if you want the model to behave differently in, say, Russia and the United States for otherwise similar inputs[8].

If you have a feature with many values but you cannot say that you need a specific behavior of the model depending on the value of that feature, then try to reduce the cardinality (number of values) of that feature[9]. There are several ways to do that:

Group similar values

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

Group the long tail

Alternatively, try to group the long tail of infrequent values under the name "other" or merge them with some similar frequent values.

[8]Often, what you want 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, such as the US and Russia, in reality, you might get poor model performance because the distribution of labels in the training data is indeed different for different countries, but you decided to remove the country information from the feature vector.

[9]The reduction of the granularity of a feature has to be done with care. Categorical features often have functional dependencies with other categorical features and their predictive power 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.

Use feature hashing
: A hash function is a deterministic function that maps a potentially unbounded integer to a finite integer range $[1, m]$ of bins. For example, if your categorical feature has a thousand different values, a hashing function, depending on the function, might transform each of these thousand values into one of a dozen bins. This means that two different values of the original categorical feature will be transformed into the same bin in the range $[1, m]$ of bins. This is called a collision. A uniform hash function ensures that roughly the same number of input values are mapped into each of the m bins.

Remove the feature
: If one value clearly dominates all other values of the categorical feature, consider removing the feature entirely.

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

The same caution must be applied when you group feature values based on how frequent those values are in the dataset. Infrequent values today may become frequent with time and more data. In general, the model and the features need to be re-evaluated from time to time.

When possible, serialize (pickle in Python, RDS in R) jointly the model and the feature extractor class 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 exactly the same feature extraction code when you build the model and when you serve it. Even a tiny difference between the code you wrote to build the model and the optimized code the IT-team might have written for the production environment may result in significant prediction error. If you are not the person responsible for writing the code for production, once the production code for feature extraction is ready, use it to rebuild the model. Always completely rebuild the model after any change in the feature extraction code.

Feature extraction code is one the most important parts in a machine learning system, therefore it has to be extensively and systematically tested, which includes:

- unit tests on all feature extractors,
- performance tests to make sure the code is still efficient,
- regular runs of feature extractors on a fixed test data to see that the distribution of feature values remains the same,
- logging of the features extracted in production for a sample of examples; when you will

work on the new version of the model, make sure that the values of the features logged in production are the same as the values you observe when working on the model.

4.15 Contributors

I'm grateful to the following people for their valuable contributions to the quality of this chapter: Mathieu Bouchard, Yacin Bahi, Samir Char, Luis Leopoldo Perez, Sylvain Truong, Fernando Hannaka, **Ana Fotina**, and **Alexander Sack**.