

Lecture 4. Feature Engineering

Note: This note is a work-in-progress, created for the course [CS 329S: Machine Learning Systems Design](#) (Stanford, 2022). For the fully developed text, see the book [Designing Machine Learning Systems](#) (Chip Huyen, O'Reilly 2022).

Errata, questions, and feedback -- please send to chip@huyenchip.com. Thank you!

Table of contents

Data Augmentation	2
Simple Label-Preserving Transformations	2
Perturbation	3
Data Synthesis	5
Learned Features vs. Engineered Features	7
Common Feature Engineering Operations	10
Handling Missing Values	10
Deletion	11
Imputation	12
Scaling	12
Discretization	14
Encoding Categorical Features	15
Feature Crossing	18
Discrete and Continuous Positional Embeddings	19

In 2014, the paper [*Practical Lessons from Predicting Clicks on Ads at Facebook*](#) claimed that having the right features is the most important thing in developing their ML models. Since then, many of the companies that I've worked with have discovered time and time again that once they have a workable model, having the right features tends to give them the biggest performance boost compared to clever algorithmic techniques such as hyperparameter tuning. State-of-the-art model architectures can still perform poorly if they don't use a good set of features.

Due to its importance, a large part of many ML engineering and data science jobs is to come up with new useful features. In this chapter, we will go over common techniques and important considerations with respect to feature engineering. We will dedicate a section to go into detail about a subtle yet disastrous problem that has derailed many ML systems in production: data leakage and how to detect and avoid it.

We will end the chapter discussing how to engineer good features, taking into account both the feature importance and feature generalization.

Data Augmentation

Data augmentation is a family of techniques that are used to increase the amount of training data. Traditionally, these techniques are used for tasks that have limited training data, such as in medical imaging projects. However, in the last few years, they have shown to be useful even when we have a lot of data because augmented data can make our models more robust to noise and even adversarial attacks.

Data augmentation has become a standard step in many computer vision tasks and is finding its way into natural language processing (NLP) tasks. The techniques depend heavily on the data format, as image manipulation is different from text manipulation. In this section, we will cover three main types of data augmentation: simple label-preserving transformations, perturbation, which is a term for “adding noises”, and data synthesis. In each type, we'll go over examples for both computer vision and NLP.

Simple Label-Preserving Transformations

In computer vision, the simplest data augmentation technique is to randomly modify an image while preserving its label. You can modify the image by cropping, flipping, rotating, inverting (horizontally or vertically), erasing part of the image, and more. This makes sense because a rotated image of a dog is still a dog. Common ML frameworks like PyTorch and Keras both have support for image augmentation. According to Krizhevsky et al., in their legendary AlexNet paper, “*the transformed images are generated in Python code on the CPU while the GPU is*

training on the previous batch of images. So these data augmentation schemes are, in effect, computationally free.¹

In NLP, you can randomly replace a word with a similar word, assuming that this replacement wouldn't change the meaning or the sentiment of the sentence. Similar words can be found either with a dictionary of synonymous words, or by finding words whose embeddings are close to each other in a word embedding space.

Original sentences	I'm so happy to see you.
Generated sentences	I'm so glad to see you. I'm so happy to see y'all . I'm very happy to see you.

Table 3-8: Three sentences generated from an original sentence by replacing a word with another word with similar meaning.

This type of data augmentation is a quick way to double, even triple your training data.

Perturbation

Perturbation is also a label-preserving operation, but because sometimes, it's used to trick models into making wrong predictions, I thought it deserves its own section.

Neural networks, in general, are sensitive to noise. In the case of computer vision, this means that by adding a small amount of noise to an image can cause a neural network to misclassify it. Su et al. showed that 67.97% of the natural images in Kaggle CIFAR-10 test dataset and 16.04% of the ImageNet test images can be misclassified by changing just one pixel² (See Figure 3-11).

Using deceptive data to trick a neural network into making wrong predictions is called adversarial attacks. Adding noise to samples to create adversarial samples is a common technique for adversarial attacks. The success of adversarial attacks is especially exaggerated as the resolution of images increases.

Adding noisy samples to our training data can help our models recognize the weak spots in their learned decision boundary and improve their performance^{3,4}. Noisy samples can be created by either adding random noise or by a search strategy. Moosavi-Dezfooli et al. proposed an algorithm, called DeepFool, that finds the minimum possible noise injection needed to cause a

¹ [ImageNet Classification with Deep Convolutional Neural Networks](#) (Krizhevsky et al., 2012)

² [One pixel attack for fooling deep neural networks](#) (Su et al., 2017)

³ [Explaining and Harnessing Adversarial Examples](#) (Goodfellow et al., 2015)

⁴ [Maxout Networks](#) (Goodfellow et al., 2013)

misclassification with high confidence⁵. This type of augmentation is called adversarial augmentation⁶.

Adversarial augmentation is less common in NLP (an image of a bear with randomly added pixels still looks like a bear, but adding random characters to a random sentence will render it gibberish), but perturbation has been used to make models more robust. One of the most notable examples is BERT, where the model chooses 15% of all tokens in each sequence at random, and chooses to replace 10% of the chosen tokens with random words. For example, given the sentence “my dog is hairy” and the model randomly replaces “hairy” with “apple”, the sentence becomes “my dog is apple”. So 1.5% of all tokens might result in nonsensical meaning. Their ablation studies show that a small fraction of random replacement gives their model a small performance boost⁷.

In chapter 5, we’ll go over how to use perturbation not just as a way to improve your model’s performance, but also a way to evaluate its performance.

⁵ [DeepFool: a simple and accurate method to fool deep neural networks](#) (Moosavi-Dezfooli et al., 2016)

⁶ [Virtual Adversarial Training: A Regularization Method for Supervised and Semi-Supervised Learning](#) (Miyato et al., 2017)

⁷ [BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding](#) (Devlin et al., 2018)

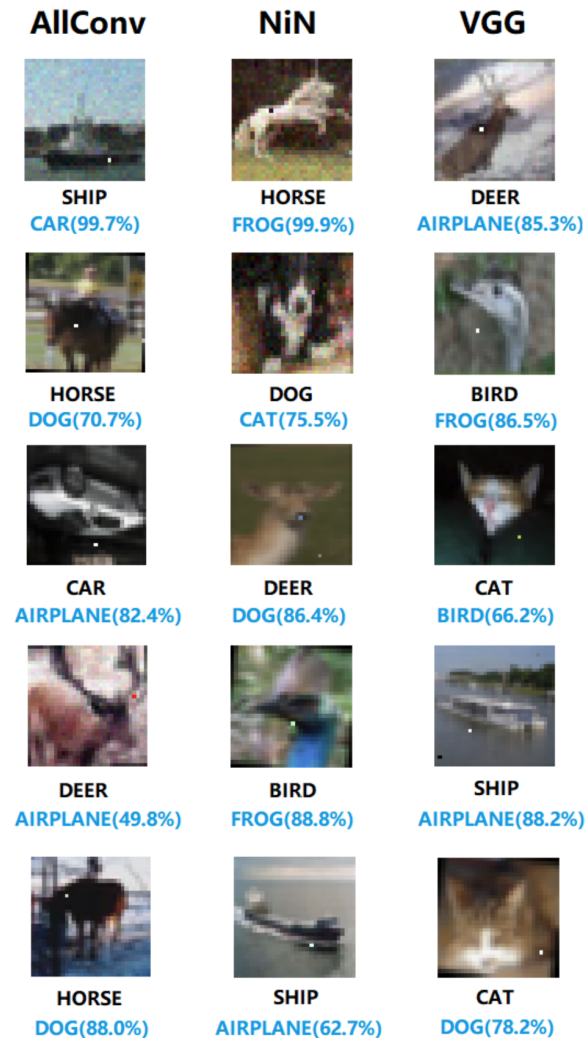


Figure 3-11: Changing one pixel can cause a neural network to make wrong predictions.

Three models used are AllConv, NiN, and VGG.

The original labels made by those models are in black, and the labels made after one pixel was changed are below. Image by [Su et al.](#)

[PERMISSION GRANTED]

Data Synthesis

Since collecting data is expensive and slow with many potential privacy concerns, it'd be a dream if we could sidestep it altogether and train our models with synthesized data. Even though we're still far from being able to synthesize all training data, it's possible to synthesize some training data to boost a model's performance.

In NLP, templates can be a cheap way to bootstrap your model. One of the teams I worked with used templates to bootstrap training data for their conversational AI (chatbot). A template might

look like: “Find me a [CUISINE] restaurant within [NUMBER] miles of [LOCATION].” With lists of all possible cuisines, reasonable numbers (you would probably never want to search for restaurants beyond 1000 miles), and locations (home, office, landmarks, exact addresses) for each city, you can generate thousands of training queries from a template.

Template	Find me a [CUISINE] restaurant within [NUMBER] miles of [LOCATION].
Generated queries	<ul style="list-style-type: none"> • Find me a Vietnamese restaurant within 2 miles of my office. • Find me a Thai restaurant within 5 miles of my home. • Find me a Mexican restaurant within 3 miles of Google headquarters.

Table 3-9: Three sentences generated from a template

In computer vision, a straightforward way to synthesize new data is to combine existing examples with discrete labels to generate continuous labels. Consider a task of classifying images with two possible labels: DOG (encoded as 0) and CAT (encoded as 1). From example x_1 of label DOG and example x_2 of label CAT, you can generate x' such as:

$$x' = \gamma x_1 + (1 - \gamma)x_2$$

The label of x' is a combination of the labels of x_1 and x_2 : $\gamma * 0 + (1 - \gamma) * 1$. This method is called mixup. The authors showed that mixup improves models’ generalization, reduces their memorization of corrupt labels, increases their robustness to adversarial examples, and stabilizes the training of generative adversarial networks.⁸

Using neural networks to synthesize training data is an exciting approach that is actively being researched but not yet popular in production. Sandfort et al. showed that by adding images generated using a CycleGAN to their original training data, they were able to improve their model’s performance significantly on CT segmentation tasks.⁹

If you’re interested in learning more about data augmentation for computer vision, [A survey on Image Data Augmentation for Deep Learning](#) (Connor Shorten & Taghi M. Khoshgoftaar, 2019) is a comprehensive review.

⁸ [mixup: BEYOND EMPIRICAL RISK MINIMIZATION](#) (Zhang et al., 2017)

⁹ [Data augmentation using generative adversarial networks \(CycleGAN\) to improve generalizability in CT segmentation tasks | Scientific Reports](#) (Sandfort et al., Nature 2019)

Learned Features vs. Engineered Features

When I cover this topic in class, my students frequently ask: “Why do we have to worry about feature engineering? Doesn’t deep learning promise us that we no longer have to engineer features?”

They are right. The promise of deep learning is that we won’t have to handcraft features. For this reason, deep learning is sometimes called feature learning¹⁰¹¹. Many features can be automatically learned and extracted by algorithms. However, we’re still far from the point where all features can be automated. This is not to mention that, as of writing, the majority of ML applications in production aren’t deep learning. Let’s go over an example to understand what features can be automatically extracted and what features still need to be handcrafted.

Imagine that you want to build a sentiment analysis classifier to classify whether a comment is spam or not. Before deep learning, when given a piece of text, you would have to manually apply classical text processing techniques such as lemmatization, expanding contraction, removing punctuation, and lowercasing everything. After that, you might want to split your text into n-grams with n values of your choice.

As a refresher, an n-gram is a contiguous sequence of n items from a given sample of text. The items can be phonemes, syllables, letters, or words. For example, given the post “I like food”, its word-level 1-grams are [“I”, “like”, “food”] and its word-level 2-grams are [“I like”, “like food”]. This sentence’s set of n-gram features, if we want n to be 1 and 2, is: [“I”, “like”, “food”, “I like”, “like food”].

Figure 4-1 shows an example of classical text processing techniques you can use to handcraft n-gram features for your text.

¹⁰ [Handcrafted vs. non-handcrafted features for computer vision classification](#) (Nanni et al., 2017)

¹¹ [Feature learning](#) (Wikipedia)



Figure 4-1: An example of techniques that you can use to handcraft n-gram features for your text

Once you've generated n-grams for your training data, you can create a vocabulary that matches each n-gram to an index. Then you can convert each post into a vector based on its n-grams' indices. For example, if we have a vocabulary of 7 n-grams as shown in Table 4-1, each post can be a vector of 7 elements. Each element corresponds to the number of times the n-gram at that index appears in the post. "I like food" will be encoded as the vector [1, 1, 0, 1, 1, 0, 1]. This vector can then be inputted into an ML model such as logistic regression.

I	like	good	food	I like	good food	like food
0	1	2	3	4	5	6



Table 4-1: Example of an 1-gram and 2-gram vocabulary



Feature engineering requires knowledge of domain-specific techniques — in this case, the domain is natural language processing (NLP). It tends to be an iterative process which can be brittle. When I followed this method for one of my early NLP projects, I kept having to restart my process either because I had forgotten to apply one technique or because one technique I used turned out to be working poorly and I had to undo it.

However, much of this pain has been alleviated since the rise of deep learning. Instead of having to worry about lemmatization, punctuation, or stopword removal, you can just split your raw text into words, create a vocabulary out of those words, and convert each of your words into one-hot vectors using this vocabulary. Your model will hopefully learn to extract useful features from this. In this new method, much of feature engineering for text has been automated. Similar progress has been made for images too. Instead of having to manually extract features from raw images and input those features into your ML models, you can just input raw images directly into your deep learning models.

However, an ML system will likely need data beyond just text and images. For example, when detecting whether a comment is spam or not, on top of the text in the comment itself, you might want to use other information about:

- the comment: such as who posted this comment, how many upvotes/downvotes it has.
- the user who posted this comment: such as when this account was created, how often they post, how many upvotes/downvotes they have.
- the thread in which the comment was posted: such as how many views it has, because popular threads tend to attract more spam.

Comment ID	Time	User	Text	# 	# 	Link	# img	Thread ID	Reply to	# replies	...
93880839	2020-10-30 T 10:45 UTC	gitrekt	Your mom is a nice lady.	1	0	0	0	2332332	n0tab0t	1	...

User ID	Created	User	Subs	# 	# 	# replies	Karma	# threads	Verified email	Awards	...
4402903	2015-01-57 T 3:09 PST	gitrekt	[r/ml, r/memes, r/socialist]	15	90	28	304	776	No		...



Thread ID	Time	User	Text	# 	# 	Link	# img	# replies	# views	Awards	...
93883208	2020-10-30 T 2:45 PST	doge	Human is temporary, AGI is forever	120	50	1	0	32	2405	1	...

Figure 4-2: Some of the possible features about a comment, a thread, or a user to be included in your model

There are so many possible features to use in your model, some of them are shown in Figure 4-2. The process of choosing what to use and extracting the information you want to use is feature engineering. For important tasks such as recommending videos for users to watch next on Tiktok, the number of features used can go up to millions. For domain-specific tasks such as

predicting whether a transaction is fraudulent, you might need subject matter expertise with banking and frauds to be able to extract useful features.

Common Feature Engineering Operations

Because of the importance and the ubiquity of feature engineering in ML projects, there have been many techniques developed to streamline the process. In this section, we will discuss several of the most important operations that you should consider, if you haven't already, while engineering features from your data. They include handling missing values, scaling, discretization, encoding categorical features, generating the old-school but still very effective cross features as well as the newer and exciting positional features. This list is nowhere near being comprehensive, but it does comprise some of the most common and useful operations to give you a good starting point. Let's dive in!

Handling Missing Values

One of the first things you might notice when dealing with data in production is that some values are missing. However, one thing that many ML engineers I've interviewed don't know is that not all types of missing values are equal¹². To illustrate this point, consider the task of predicting whether someone is going to buy a house in the next 12 months. A portion of the data we have is in Table 4-2.

ID	Age	Gender	Annual income	Marital status	Number of children	Job	Buy?
1		A	150,000		1	Engineer	No
2	27	B	50,000			Teacher	No
3		A	100,000	Married	2		Yes
4	40	B			2	Engineer	Yes
5	35	B		Single	0	Doctor	Yes
6		A	50,000		0	Teacher	No
7	33	B	60,000	Single		Teacher	No
8	20	B	10,000			Student	No

Table 4-2: A portion of the data we have for the task of predicting whether someone will buy a house in the next 12 months

¹² In my experience, how well a person handles missing values for a given dataset during interviews strongly correlates with how well they will do in their day to day jobs.

There are three types of missing values. The official names for these types are a little bit confusing so we'll go into detailed examples to mitigate the confusion.

1. **Missing not at random** (MNAR): when the reason a value is missing is because of the value itself. In this example, we might notice that respondents of gender "B" with higher income tend not to disclose their income. The income values are missing for reasons related to the values themselves.
2. **Missing at random** (MAR): when the reason a value is missing is not due to the value itself, but due to another observed variable. In this example, we might notice that age values are often missing for respondents of the gender "A", which might be because the people of gender A in this survey don't like disclosing their age.
3. **Missing completely at random** (MCAR): when there's no pattern in when the value is missing. In this example, we might think that the missing values for the column "Job" might be completely random, not because of the job itself and not because of another variable. People just forget to fill in that value sometimes for no particular reason. However, this type of missing is very rare. There are usually reasons why certain values are missing, and you should investigate.

When encountering missing values, you can either fill in the missing values with certain values, (imputation), or remove the missing values (deletion). We'll go over both.

Deletion

When I ask candidates about how to handle missing values during interviews, many tend to prefer deletion, not because it's a better method, but because it's easier to do.

One way to delete is **column deletion**: if a variable has too many missing values, just remove that variable. For example, in the example above, over 50% of the values for the variable "Marital status" are missing, so you might be tempted to remove this variable from your model. The drawback of this approach is that you might remove important information and reduce the accuracy of your model. Marital status might be highly correlated to buying houses, as married couples are much more likely to be homeowners than single people¹³.

Another way to delete is **row deletion**: if an example has missing value(s), just remove that example from the data. This method can work when the missing values are completely at random (MCAR) and the number of examples with missing values is small, such as less than 0.1%. You don't want to do row deletion if that means 10% of your data examples are removed.

However, removing rows of data can also remove important information that your model needs to make predictions, especially if the missing values are not at random (MNAR). For example, you don't want to remove examples of gender B respondents with missing income because

¹³ [3 FACTS ABOUT MARRIAGE AND HOMEOWNERSHIP](#) (Rachel Bogardus Drew, Joint Center for Housing Studies at Harvard University 2014)

whether income is missing is information itself (missing income might mean higher income, and thus, more correlated to buying a house) and can be used to make predictions.

On top of that, removing rows of data can create biases in your model, especially if the missing values are at random (MAR). For example, if you remove all examples missing age values in the data in Table 4-2, you will remove all respondents with gender A from your data, and your model won't be able to make predictions for respondents with gender A.

Imputation

Even though deletion is tempting because it's easy to do, deleting data can lead to losing important information or cause your model to be biased. If you don't want to delete missing values, you will have to impute them, which means "fill them with certain values." Deciding which "certain values" to use is the hard part.

One common practice is to fill in missing values with their defaults. For example, if the job is missing, you might fill it with an empty string "". Another common practice is to fill in missing values with the mean, median, or mode (the most common value). For example, if the temperature value is missing for a data example whose month value is July, it's not a bad idea to fill it with the median temperature of July.

Both practices work well in many cases, but sometimes, they can cause hair-splitting bugs. One time, in one of the projects I was helping with, we discovered that the model was spitting out garbage because the app's front-end no longer asked users to enter their age, so age values were missing, and the model filled them with 0. But the model never saw the age value of 0 during training, so it couldn't make reasonable predictions.

In general, you might not want to fill missing values with possible values, such as filling the missing number of children with 0 — 0 is a possible value for the number of children. It makes it hard to distinguish between people for whom you don't have children information and people who don't have children.

Multiple techniques might be used at the same time or in sequence to handle missing values for a particular set of data. Regardless of what techniques you use, one thing is certain: there is no perfect way to handle missing values. With deletion, you risk losing important information or accentuating biases. With imputation, you risk adding noise to your data, or worse, data leakage. If you don't know what data leakage is, don't panic, we'll cover it in the **Data Leakage** section of this chapter.

Scaling

Consider the task of predicting whether someone will buy a house in the next 12 months, and the data is shown in Table 4-2. The values of the variable Age in our data go between 20 and 40,

whereas the values of the variable Annual Income go between 10,000 and 150,000. When we input these two variables into an ML model, it won't understand that 150,000 and 40 represent different things. It will just see them both as numbers, and because the number 150,000 is much bigger than 40, it might give it more importance, regardless of which variable is actually more useful for the predicting task.

During data processing, it's important to scale your features so that they're in similar ranges. This process is called feature scaling. This is one of the simplest things you can do that often result in a performance boost for your model. Neglecting to do so can cause your model to make gibberish predictions, especially with classical algorithms like gradient-boosted trees and logistic regression¹⁴.

An intuitive way to scale your features is to get each feature to be in the range [0, 1]. Given a variable x , its values can be rescaled to be in this range using the following formula.

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

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

You can validate that if x is the maximum value, the scaled value x' will be 1. If x is the minimum value, the scaled value x' will be 0.

If you want your feature to be in an arbitrary range $[a, b]$ — empirically, I find the range $[-1, 1]$ to work better than the range $[0, 1]$ — you can use the following formula.

$$x' = a + \frac{(x - \min(x))(b - a)}{\max(x) - \min(x)}$$

$$x' = a + \frac{(x - \min(x))(b - a)}{\max(x) - \min(x)}$$

Scaling to an arbitrary range works well when you don't want to make any assumptions about your variables. If you think that your variables might follow a normal distribution, it might be helpful to normalize them so that they have zero-mean and unit variance. This process is called standardization.

$$x' = \frac{x - \bar{x}}{\sigma}$$

with \bar{x} being the mean of variable x , and σ being its standard deviation.

$$x' = \frac{x - \bar{x}}{\sigma}$$

with \bar{x} being the mean of variable x , and σ being its standard deviation.

¹⁴ Feature scaling once boosted my model's performance by almost 10%.

In practice, ML models tend to struggle with features that follow a skewed distribution. To help mitigate the skewness, a technique commonly used is [log transformation](#): apply the log function to your feature. An example of how the log transformation can make your data less skewed is shown in Figure 4-3. While this technique can yield performance gain in many cases, it doesn't work for all cases and you should be wary of the analysis performed on log-transformed data instead of the original data¹⁵.

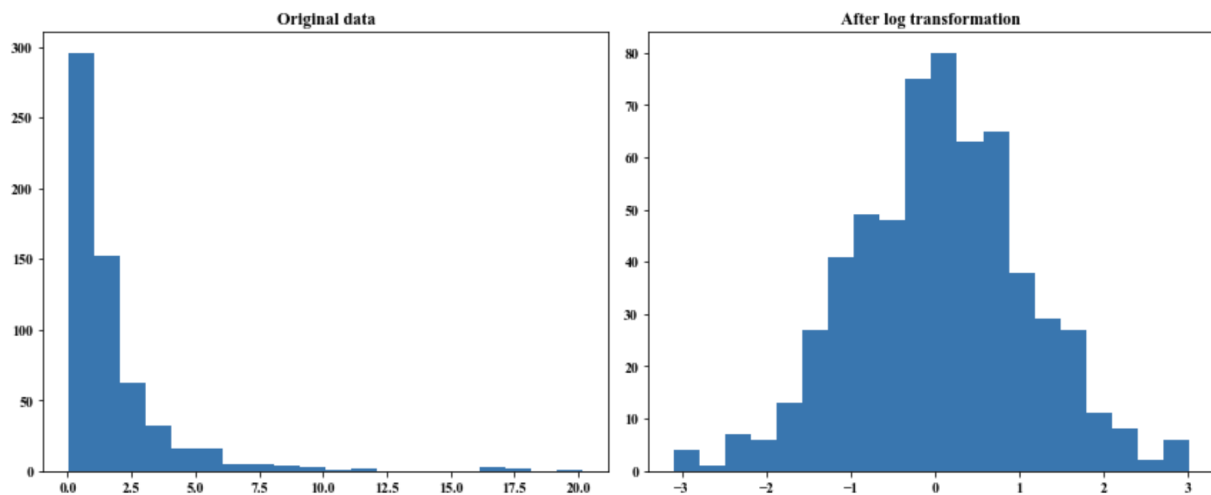


Figure 4-3: In many cases, the log transformation can help reduce the skewness of your data

There are two important things to note about scaling. One is that it's a common source of data leakage, (this will be covered in greater detail in the Data Leakage section). Another is that it often requires global statistics — you have to look at the entire or a subset of training data to calculate its min, max, or mean. During inference, you reuse the statistics you had obtained during training to scale new data. If the new data has changed significantly compared to the training, these statistics won't be very useful. Therefore, it's important to retrain your model often to account for these changes. We'll discuss more on how to handle changing data in production in the section on continual learning in Chapter 8.

Discretization

Imagine that we've built a model with the data in Table 4-2. During training, our model has seen the annual income values of 150000, 50000, 100000, 50000, 60000, and 10000. During inference, our model encounters an example with an annual income of 9000.50.

¹⁵ [Log-transformation and its implications for data analysis](#) (Feng et al., 2014)

Intuitively, we know that \$9000.50 a year isn't much different from \$10,000/year, and we want our model to treat both of them the same way. But the model doesn't know that. Our model only knows that 9000.50 is different from 10000, and will treat them differently.

Discretization is the process of turning a continuous feature into a discrete feature. This process is also known as quantization. This is done by creating buckets for the given values. For annual income, you might want to group them into three buckets as follows.

- Lower income: less than \$35,000/year
- Middle income: between \$35,000 and \$100,000/year
- Upper income: more than \$100,000/year

Now, instead of having to learn an infinite number of possible incomes, our model can focus on learning only three categories, which is a much easier task to learn.

Even though by definition, it's used for continuous features, the same technique can be used for discrete features too. The age variable is discrete, it might still be useful to group them into buckets such as follows.

- Less than 18
- Between 18 and 22
- Between 22 and 30
- Between 30 - 40
- Between 40 - 65
- Over 65

A question with this technique is how to best choose the boundaries of categories. You can try to plot the histograms of the values and choose the boundaries that make sense. In general, common sense, basic quantiles, and sometimes subject matter expertise can get you a long way.

Encoding Categorical Features

We've talked about how to turn continuous features into categorical features. In this section, we'll discuss how to best handle categorical features.

People who haven't worked with data in production tend to assume that categories are *static*, which means the categories don't change over time. This is true for many categories. For example, age brackets and income brackets are unlikely to change and you know exactly how many categories there are in advance. Handling these categories is straightforward. You can just give each category a number and you're done.

However, in production, categories change. Imagine you're building a recommendation system to predict what products users might want to buy for Amazon. One of the features you want to use is the product brand. When looking at Amazon's historical data, you realize that there are a lot of brands. Even back in 2019, there were already over 2 million brands on Amazon¹⁶!

The number of brands is overwhelming but you think: "I can still handle this." You encode each brand a number, so now you have 2 million numbers from 0 to 1,999,999 corresponding to 2 million brands. Your model does spectacularly on the historical test set, and you get approval to test it on 1% of today's traffic.

In production, your model crashes because it encounters a brand it hasn't seen before and therefore can't encode. New brands join Amazon all the time. You create a category "UNKNOWN" with the value of 2,000,000 to catch all the brands your model hasn't seen during training.

Your model doesn't crash anymore but your sellers complain that their new brands are not getting any traffic. It's because your model didn't see the category UNKNOWN in the train set, so it just doesn't recommend any product of the UNKNOWN brand. Then you fix this by encoding only the top 99% most popular brands and encode the bottom 1% brand as UNKNOWN. This way, at least your model knows how to deal with UNKNOWN brands.

Your model seems to work fine for about 1 hour, then the click rate on recommended products plummets. Over the last hour, 20 new brands joined your site, some of them are new luxury brands, some of them are sketchy knockoff brands, some of them are established brands. However, your model treats them all the same way it treats unpopular brands in the training set.

This isn't an extreme example that only happens if you work at Amazon on this task. This problem happens quite a lot. For example, if you want to predict whether a comment is spam, you might want to use the account that posted this comment as a feature, and new accounts are being created all the time. The same goes for new product types, new website domains, new restaurants, new companies, new IP addresses, and so on. If you work with any of them, you'll have to deal with this problem.

Finding a way to solve this problem turns out to be surprisingly difficult. You don't want to put them into a set of buckets because it can be really hard—how would you even go about putting new user accounts into different groups?

One solution to this problem is **the hashing trick**, popularized by the package Vowpal Wabbit developed at Microsoft¹⁷. The gist of this trick is that you use a hash function to generate a hashed value of each category. The hashed value will become the index of that category. Because

¹⁶ [Jun 11, 2019 Two Million Brands on Amazon](#) (Marketplace Pulse)

¹⁷ [Feature hashing](#) (Wikipedia)

you can specify the hash space, you can fix the number of encoded values for a feature in advance, without having to know how many categories there will be. For example, if you choose a hash space of 18 bits, which corresponds to $2^{18} = 262,144$ possible hashed values, all the categories, even the ones that your model has never seen before, will be encoded by an index between 0 and 262,143.

One problem with hashed functions is collision: two categories being assigned the same index. However, with many hash functions, the collisions are random, new brands can share index with any of the old brands instead of always sharing index with unpopular brands, which is what happens when we use the UNKNOWN category above. The impact of colliding hashed features is, fortunately, not that bad. In a research done by Booking.com, even for 50% colliding features, the performance loss is less than half a percent, as shown in Figure 4-4.

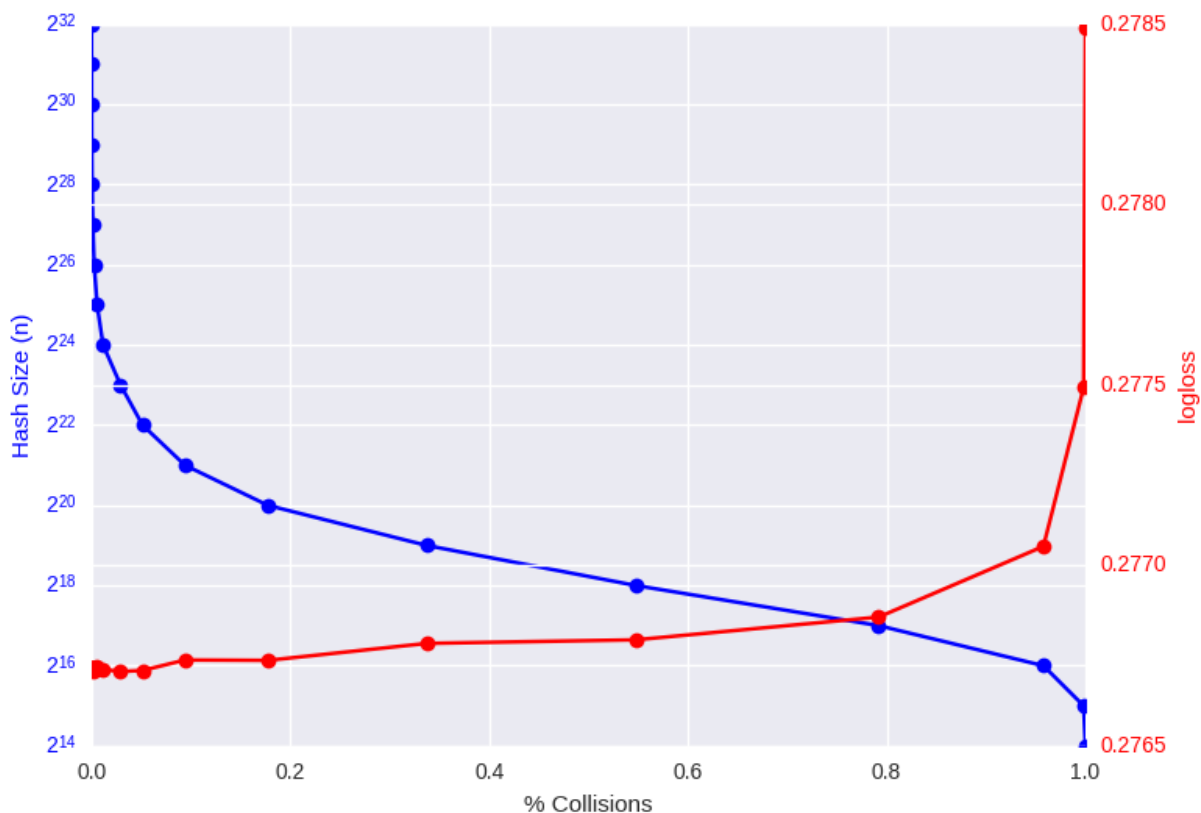


Figure 4-4: 50% collision rate only causes the log loss to increase less than half a percent. Image by [Lucas Bernardi](#).
[PERMISSION GRANTED]

You can choose a hash space large enough to reduce the collision. You can also choose a hash function with properties that you want, such as a locality-sensitive hashing function where

similar categories (such as websites with similar names) are hashed into values close to each other.

Because it's a trick, it's often considered hacky by academics and excluded from ML curricula. But its wide adoption in the industry is a testimonial to how effective the trick is. It's essential to Vowpal Wabbit and it's part of the frameworks scikit-learn, TensorFlow, and gensim. It can be especially useful in continual learning settings where your model learns from incoming examples in production. We'll cover continual learning in Chapter 8.

Feature Crossing

Feature crossing is the technique to combine two or more features to generate new features. This technique is useful to model the non-linear relationships between features. For example, for the task of predicting whether someone will want to buy a house in the next 12 months, you suspect that there might be a non-linear relationship between marital status and number of children, so you combine them to create a new feature “marriage and children” as in Table 4-3.

Marriage	Single	Married	Single	Single	Married
Children	0	2	1	0	1
Marriage & children	Single, 0	Married, 2	Single, 1	Single, 0	Married, 1

Table 4-3: Example of how two features can be combined to create a new feature

Because feature crossing helps model non-linear relationships between variables, it's essential for models that can't learn or are bad at learning non-linear relationships, such as linear regression, logistic regression, and tree-based models. It's less important in neural networks, but can still be useful because explicit feature crossing occasionally helps neural networks learn non-linear relationships faster. [DeepFM](#) and [xDeepFM](#) are the family of models that have successfully leverage explicit feature interactions for recommendation systems and click-through-rate prediction tasks.

A caveat of feature crossing is that it can make your feature space blow up. Imagine feature A has 100 possible values and feature B has 100 possible features, crossing these two features will result in a feature with $100 \times 100 = 10,000$ possible values. You will need a lot more data for models to learn all these possible values. Another caveat is that because feature crossing increases the number of features models use, it can make models overfit to the training data.

Discrete and Continuous Positional Embeddings

First introduced to the deep learning community in the paper [Attention Is All You Need](#) (Vaswani et al., 2017), positional embedding has become a standard data engineering technique for many applications in both computer vision and natural language processing. We'll walk through an example to show why positional embedding is necessary and how to do it.

Consider the task of language modeling where you want to predict the next token based on the previous sequence of tokens. In practice, a token can be a word, a character, or a subword, and a sequence length can be up to 512 if not larger. However, for simplicity, let's use words as our tokens and use the sequence length of 8. Given an arbitrary sequence of 8 words, such as, *"Sometimes all I really want to do is"*, we want to predict the next word.

If we use a recurrent neural network, it will process words in sequential order, which means the order of words is implicitly inputted. However, if we use a model like a transformer, words are processed in parallel, so words' positions need to be explicitly inputted so that our model knows which word follows which word ("a dog bites a child" is very different from "a child bites a dog"). We don't want to input the absolute positions: 0, 1, 2, ..., 7 into our model because empirically, neural networks don't work well with inputs that aren't unit-variance (that's why we scale our features, as discussed previously in the section Scaling).

If we rescale the positions to between 0 and 1, so 0, 1, 2, ..., 7 become 0, 0.143, 0.286, ..., 1, the differences between the two positions will be too small for neural networks to learn to differentiate.

A way to handle position embeddings is to treat it the way we'd treat word embedding. With word embedding, we use an embedding matrix with the vocabulary size as its number of columns, and each column is the embedding for the word at the index of that column. With position embedding, the number of columns is the number of positions. In our case, since we only work with the previous sequence size of 8, the positions go from 0 to 7 (see Figure 4-5).

The embedding size for positions is usually the same as the embedding size for words so that they can be summed. For example, the embedding for the word "food" at position 0 is the sum of the embedding vector for the word "food" and the embedding vector for position 0. This is the way position embeddings are implemented in HuggingFace's BERT as of August 2021. Because the embeddings change as the model weights get updated, we say that the position embeddings are learned.

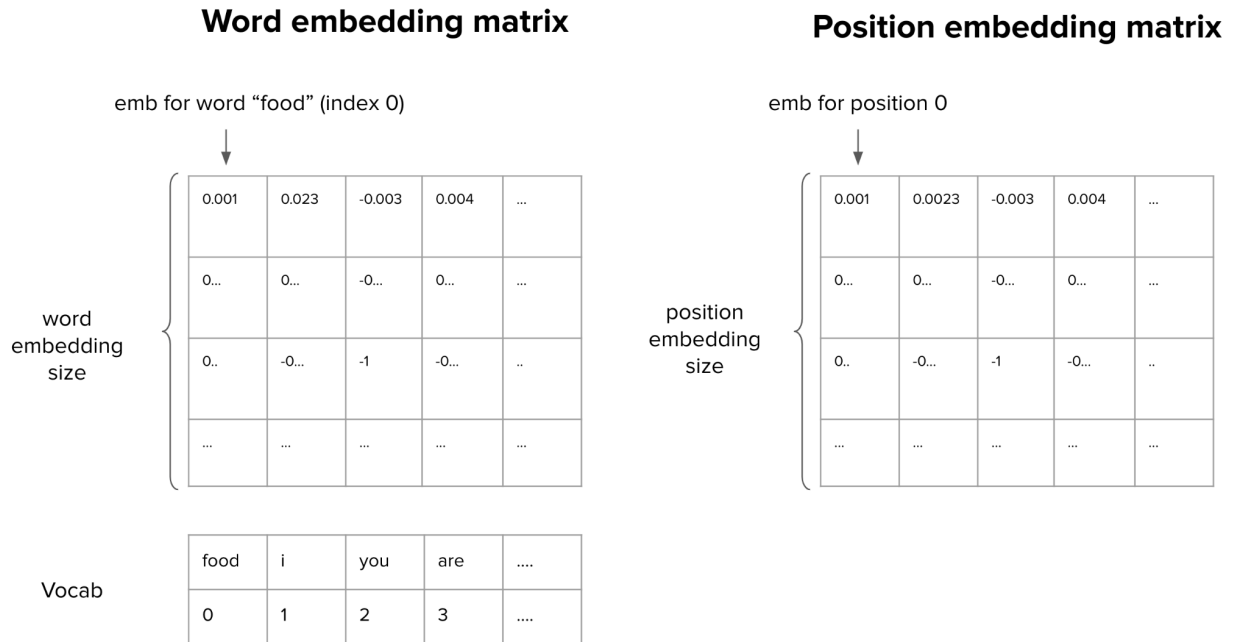


Figure 4-5: One way to embed positions is to treat them the way you'd treat word embeddings

Position embeddings can also be fixed. The embedding for each position is still a vector with S elements (S is the position embedding size), but each element is predefined using a function, usually sine and cosine. In [the original Transformer paper](#), if the element is at an even index, use sine. Else, use cosine. See Figure 4-6.

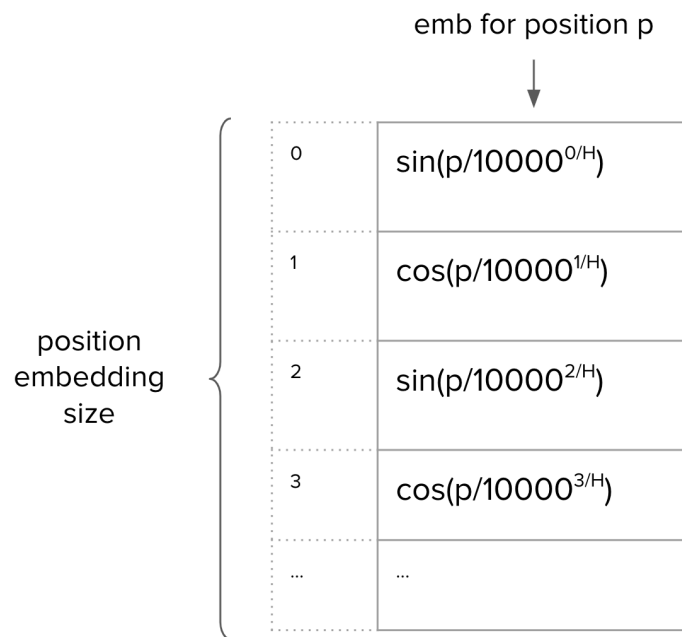


Figure 4-6: Example of fixed position embedding.
 H is the dimension of the outputs produced by the model.

Fixed positional embedding is a special case of what is known as Fourier features. If positions in positional embeddings are discrete, Fourier features can also be continuous. Consider the task involving representations of 3D objects, such as a teapot. Each position on the surface of the teapot is represented by a 3-dimensional coordinate, which is continuous. When positions are continuous, it'd be very hard to build an embedding matrix with continuous column indices, but fixed position embeddings using sine and cosine functions still work.

This is the generalized format for the embedding vector at coordinate \mathbf{v} , also called the Fourier features of coordinate \mathbf{v} . Fourier features have been shown to improve models' performance for tasks that take in coordinates (or positions) as inputs.¹⁸

$$\gamma(\mathbf{v}) = [a_1 \cos(2\pi \mathbf{b}_1^T \mathbf{v}), a_1 \sin(2\pi \mathbf{b}_1^T \mathbf{v}), \dots, a_m \cos(2\pi \mathbf{b}_m^T \mathbf{v}), a_m \sin(2\pi \mathbf{b}_m^T \mathbf{v})]^T$$

$$\gamma(\mathbf{v}) = [a_1 \cos(2\pi \mathbf{b}_1^T \mathbf{v}), a_1 \sin(2\pi \mathbf{b}_1^T \mathbf{v}), \dots, a_m \cos(2\pi \mathbf{b}_m^T \mathbf{v}), a_m \sin(2\pi \mathbf{b}_m^T \mathbf{v})]^T$$

¹⁸ [Fourier features let networks learn high frequency functions in low dimensional domains](#) (Tancik et al., 2020)