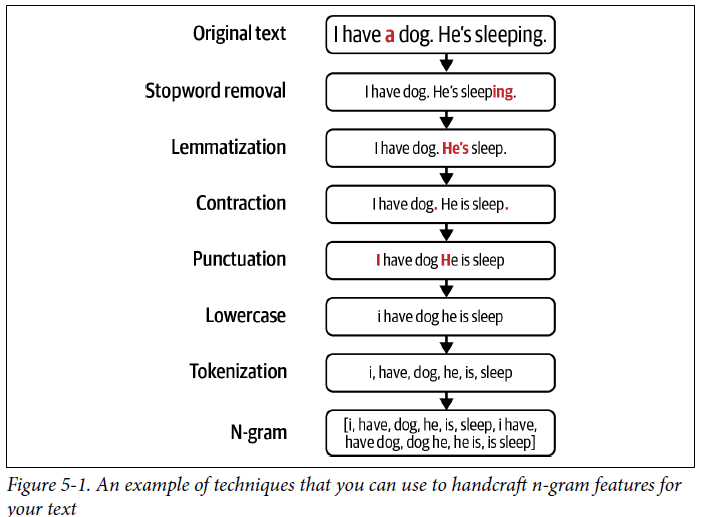
# Designing Machine Learning Systems - Chip Huyen

## Chapter 5 – Feature Engineering

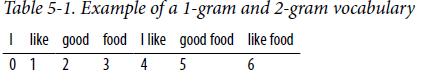
* 2014 Facebook paper “Practical Lessons from Predicting Clicks on Ads at Facebook” claimed that having the *right features* is the *most* important thing in developing their ML models
* Many companies have discovered time and time again that once they have a workable model, **having the right features 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 MLE and data science jobs is to come up with new useful features**
* Be aware of **data leakage = a subtle yet disastrous problem that has derailed many ML systems in production**, and how to detect and avoid it.
* NOTE: Talking about feature engineering, some people might think of **feature stores**
* *Feature stores are actually closer to infrastructure to support multiple ML applications*

### Learned vs. Engineered Features

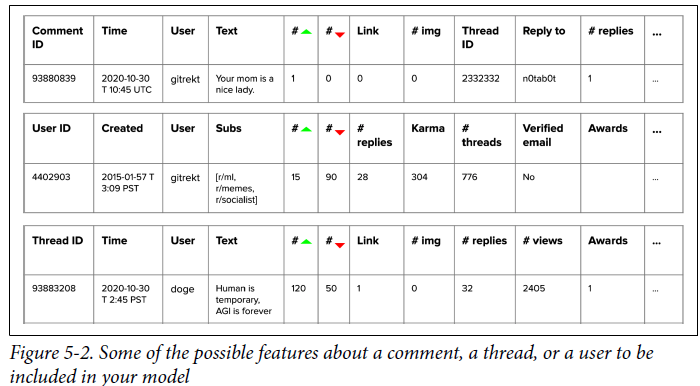
* Why do we have to worry about feature engineering? Doesn’t DL promise that we no longer have to engineer features?
* This are right 🡪 The **promise of DL is that we won’t have to handcraft features**
* For this reason, DL is sometimes called **feature learning** 🡪 Many features can be automatically 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 the **majority of ML applications in production aren’t DL**
* Ex: You want to build a sentiment analysis classifier to classify whether a comment is spam or not
* Before DL, when given a piece of text, you’d have to manually apply classical text processing techniques such as lemmatization, expanding contractions, removing punctuation, and lowercasing everything
* After that, you might want to split your text into n-grams with nvalues of your choice.
* An **n-gram** = **a contiguous sequence of nitems from a given sample of text**
* Items can be phonemes, syllables, letters, or words
* Ex: 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 nto be 1 and 2, is: [“I”, “like”, “food”, “I like”, “like food”]
* See an example of classical text processing techniques you can use to handcraft n-gram features for your text below



* Once you’ve generated n-grams for your training data, you can create a **vocabulary** that **maps each n-gram to an index**
* Then you can **convert each post into a vector based on its n-grams’ indices**
* Ex: If we have a vocabulary of 7 n-grams as shown in the table below, 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 used as an input into an ML model
* **Feature engineering requires knowledge of domain-specific techniques** (Ex: The domain is NLP and the native language of the text)
* It **tends to be an iterative process**, which **can be brittle**.
* However, **much of this pain has been alleviated since the rise of DL**
* Instead of having to worry about lemmatization, punctuation, or stop-word removal, you can just split your raw text into words (i.e., **tokenization**), create a **vocabulary** out of those words, + convert each of your words into one-shot 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 DL models
* **However, an ML system will likely need data beyond just text and images**
* Ex: 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 (How many upvotes/downvotes does it have?), user who posted this comment (When was this account created, how often do they post, and how many upvotes/downvotes do they have?), thread in which the comment was posted (How many views does it have? Popular threads tend to attract more spam), etc.
* **There are many possible features to use in *this* model**. Some of are shown below:



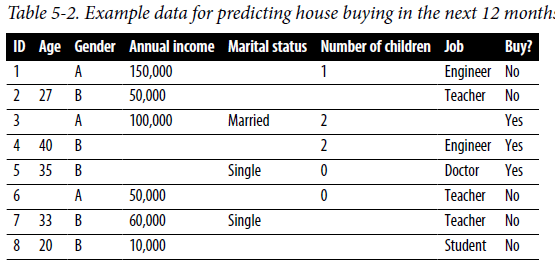
* The **process of choosing what information to use and how to extract this information into a format usable by your ML models** is **feature engineering**.
* **For complex tasks** such as recommending videos for users to watch next on TikTok, the **number of features used can go up to millions**
* **For domain-specific tasks** (predicting whether a transaction is fraudulent), **you might need subject matter expertise** (with banking and frauds) **to be able to come up with useful features**.

### Common Feature Engineering Operations

* **Because of the importance + ubiquity of feature engineering in ML projects, there have been many techniques developed to streamline the process**.
* Several of the most important operations that you might want to consider while engineering features from data include **handling missing values, scaling, discretization, encoding categorical features, and 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.*

#### 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 MLE’s don’t know is that **not all types of missing values are equal**
* Consider the task of predicting whether someone is going to buy a house in the next 12 months with the data below



* 3 types of missing values
* **1) Missing not at random (MNAR)** = **a value is missing is *because of the true value itself***
* Might notice that some respondents didn’t disclose income
* Upon investigation it may turn out that the income of respondents who failed to report tends to be higher than that of those who did disclose
* *The income values are missing for reasons related to the values themselves*
* **2) Missing at random (MAR)** = **a value is missing is NOT due to the value itself, but due to *another, observed* variable**
* 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)** = **there’s no pattern in when the value is missing**
* Might think that the missing values for the column “Job” might be completely random, not because of the job itself and not because of any other 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**
* We **can either fill in the missing values with certain values** (**imputation**) or **remove the missing values** (**deletion**)

##### Deletion

* **Many tend to prefer deletion**, not because it’s a better method, but **because it’s easier to do**
* 1) One way to delete is **column deletion**: **if a variable has too many missing values, just remove that variable**
* Above, over 50% of the values for the “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 people
* 2) Another way to delete is **row deletion**: **if a sample has missing value(s), just remove that sample**
* 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 samples 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)**
* Ex: Don’t want to remove samples of gender B respondents with missing income because *the fact that 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)**
* Ex: If you remove all examples missing age values in the data, you will remove all respondents with gender A from your data, and *your model won’t be able to make good predictions for respondents with gender A*

##### Imputation

* **Deletion is tempting because it’s easy to do, but deleting data can lead to losing important information and introduce biases into your model**
* If you **don’t want to delete missing values**, you will **have to impute them** = fill them with certain values
* **Deciding which “certain values” to use is the hard part**
* 1) One common practice is to **fill in missing values with their defaults**
* Ex: If job is missing, might fill it with an empty string ““
* 2) Another common practice is to **fill in missing values with the mean, median, or mode**
* Ex: If temperature is missing for a data sample 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-pulling bugs**
* Ex: A model spitting out garbage because the app’s frontend 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 want to avoid filling missing values with possible values**
* Ex: Filling "missing “number of children” with 0, since 0 = a possible value for number of children
* **It makes it hard to distinguish between people whose information is missing** 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 injecting your *own* bias into and adding noise to your data, or worse, data leakage**

#### Scaling

* The values of the variable Age in our house buying prediction data range from 20 to 40, whereas the values of the variable Annual Income range from 10,000 to 150,000
* When we input these 2 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**
* Then, **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 generating predictions**
* **Before inputting features into models, it’s important to scale them to be similar ranges** = **feature scaling**
* This is **one of the simplest things you can do that often results in a performance boost for a model**
* **Neglecting to do so can cause a model to make gibberish predictions**, *especially with classical algorithms like gradient-boosted trees and logistic regression.*
* An **intuitive way to scale your features is to get them 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:



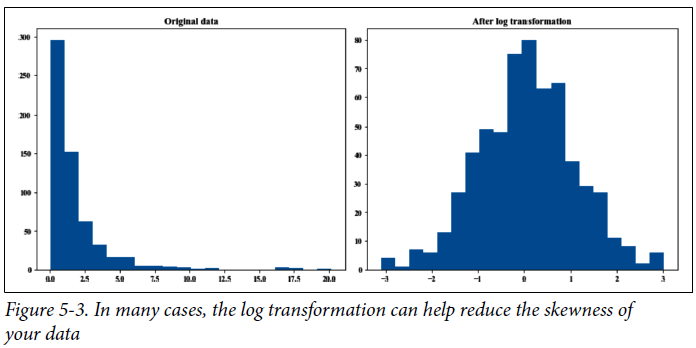
* Can validate that if xis the maximum value, the scaled value x` will be 1
* If xis the minimum value, the scaled value x` will be 0
* If you **want your feature to be in an arbitrary range [a, b]** (empirically, *the range [–1, 1] can work better than the range [0, 1]*), you can use the following formula:



* ***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 = standardization**:



* with x-- = the mean of variable x, and σ = its standard deviation
* **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**



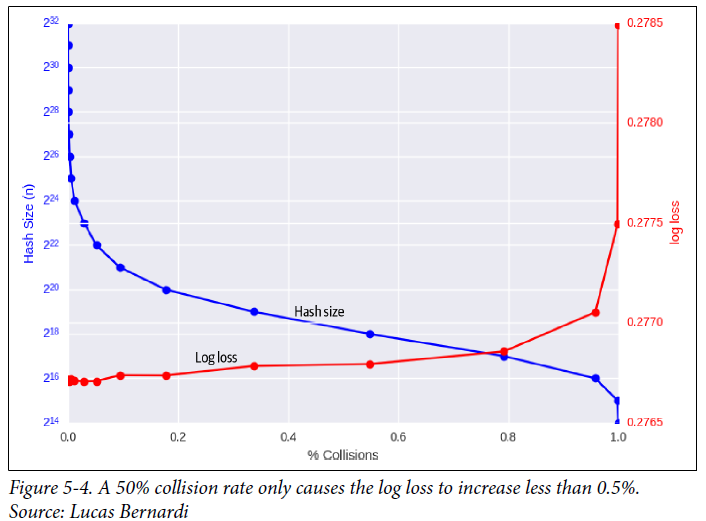
* **While this technique can yield performance gain in many cases, it doesn’t work for all cases, so be wary of analysis performed on log-transformed data instead of original data**
* **2 important things to note about scaling**
* **1) It’s a common source of data leakage**
* **2) It often requires global 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 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**

#### Discretization

* **In practice, it’s rarely found that discretization helps**
* During training of our housing prediction model, our model has seen the annual income values of “150,000,” “50,000,” “100,000,” and so on
* During *inference*, our model encounters an example with an annual income of “9,000.50.”
* *Intuitively*, we know $9,000.50 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 9,000.50 is different from 10,000, and it will treat them differently*
* **Discretization** = **process of turning a continuous feature into a discrete feature** (also known as **quantization** or **binning**) 🡪 **done by creating buckets for the given values**
* For annual income, you might want to group them into 3 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
* **Instead of having to learn an infinite number of possible incomes**, our model can **focus on learning only 3 categories**, which is a **much easier task to learn**
* This technique is ***supposed* to be more helpful with *limited* training data**.
* Even though, by definition, **discretization** is meant for continuous features, it **can be used for discrete features too**
* Age variable is discrete, but it might still be useful to group the values into buckets:
* < 18, Between 18 and 22, Between 22 and 30, Between 30 and 40, Between 40 and 65, > 65
* The **downside is that this categorization introduces discontinuities at the category boundaries**
* **$34,999 is now treated as completely different from $35,000, which is treated the same as $100,000**
* **Choosing the boundaries of categories might not be all that easy**
* Can try to plot the histograms of the values and choose boundaries that make sense
* **In general, common sense, basic quantiles, and sometimes subject matter expertise can help**

#### Encoding Categorical Features

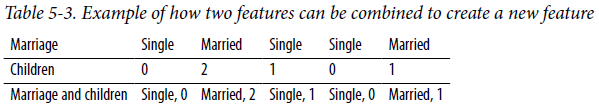
* **People who haven’t worked with data in production tend to assume categories are *static*** (don’t change over time)
* This is **true for many categories**
* Age brackets and income brackets are unlikely to change, and you know exactly how many categories there are in advance
* Handling these categories is straightforward 🡪 just give each category a number, + you’re done
* **However,** **in production, categories change**
* Ex: Building a recommender system to predict what products users might want to buy from Amazon
* One of the features you want to use is product brand
* When looking at Amazon’s historical data, you realize that there are a lot of brands (in 2019 , there were >2M brands)
* You encode each brand as a number, so now you have 2M numbers, from 0 to 1,999,999, corresponding to 2M 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
* To address this, create a category UNKNOWN with the value of 2,000,000 to catch all brands your model hasn’t seen during training.
* Your **model doesn’t crash anymore, but sellers complain that their new brands are not getting any traffic**
* **Because your model didn’t see the category UNKNOWN in the train set, it just doesn’t recommend any product of the UNKNOWN brand**
* 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 one hour, then CTR on product recommendations plummets**
* Over the last hour, 20 new brands joined your site; some are new luxury brands, some are sketchy knockoff brands, some are established brands
* However, **your model treats them all the same way it treats unpopular brands in the training data**
* This isn’t an extreme example that only happens if you work at Amazon, this problem happens quite a lot
* Ex: If you want to predict whether a comment is spam, might want to use the account that posted this comment as a feature, but new accounts are being created all the time
* Same goes for new product types, website domains, restaurants, companies, IP addresses, etc.
* ***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**
* **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 = the **hashing trick**, popularized by the package Vowpal Wabbit developed at Microsoft
* The **gist** of this trick = you **use a hash function to generate a hashed value of each category**
* The **hashed value will become the index of that category**
* Because 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**
* Ex: If you choose a hash space of 18 bits, which corresponds to 218 = 262,144 possible hashed values, all the categories, even the ones your model has never seen before, will be encoded by an index between 0 and 262,143
* **One problem with hashed functions is collision**: **2** **categories being assigned the same index**
* ***However, with many hash functions, the collisions are random***;
* New brands can share an index with any of the existing brands instead of always sharing an index with unpopular brands, which is what happens when we use the preceding UNKNOWN category
* The **impact of colliding hashed features is, fortunately, not that bad**
* Booking.com research: Even for 50% colliding features, performance loss is < 0.5%



* **Can choose a hash space large enough to reduce the collision**, or can also **choose a hash function with properties 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 hashing is 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 of scikit-learn, TensorFlow, and genism
* It **can be especially useful in continual learning settings where your model learns from incoming examples in production**

#### Feature Crossing

* **Feature crossing** = the **technique to combine 2+ features to generate new features**
* This technique is **useful to model the nonlinear relationships between features**
* Ex: 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 nonlinear relationship between marital status and number of children, so you combine them to create a new feature “marriage and children”



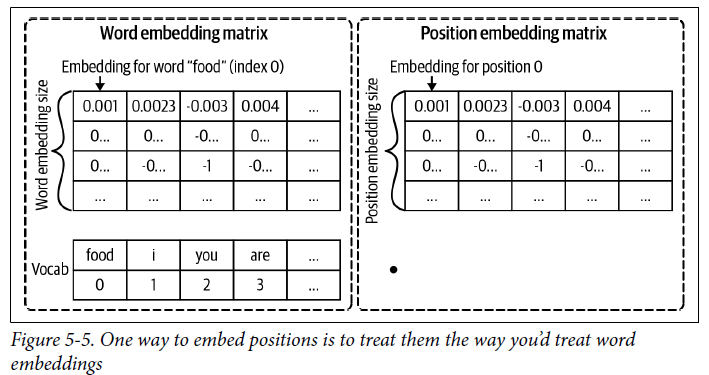
* **Because *feature crossing helps model nonlinear relationships between variables*, it’s essential for models that can’t learn or are bad at learning nonlinear relationships, such as linear regression, logistic regression, and tree-based models**
* It’s **less important in NN’s**, but **can still be useful because explicit feature crossing occasionally helps NN’s learn nonlinear relationships *faster***
* DeepFM and xDeepFM = the family of models that have successfully leveraged explicit feature interactions for recommender systems and CTR prediction
* A **caveat of feature crossing** is that **it can make your feature space blow up**
* Ex: Feature A has 100 possible values and feature B has 100 possible values; crossing these 2 features will result in a feature with 100 × 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**

#### Embeddings

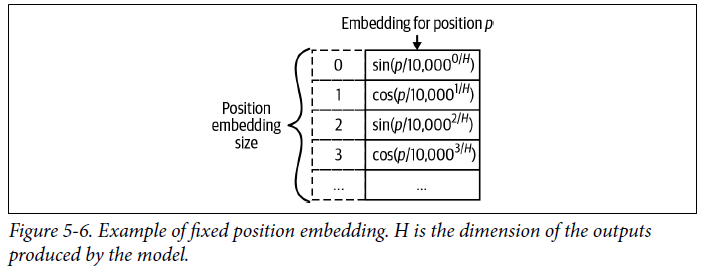
* An **embedding** = **a vector that represents a piece of data**
* Call **the set of all possible embeddings generated by the same algorithm for a type of data** an **embedding space**
* **All embedding vectors in the *same* space are of the *same* size**.
* One of the most common uses of embeddings is **word embeddings**, where you **represent each word with a vector**
* However, *embeddings for other types of data are increasingly popular*
* Ecommerce solutions like Criteo and Coveo have embeddings for products, Pinterest has embeddings for images, graphs, queries, + even users
* **Given that there are so many types of data with embeddings, there has been a lot of interest in creating universal embeddings for multimodal data**

#### Discrete and Continuous Positional Embeddings

* **Positional embedding** has become **a standard data engineering technique for many applications in both computer vision and NLP**
* Ex: Consider the task of language modeling where you want to predict the next **token** (e.g., a word, character, or sub-word) based on the previous sequence of tokens
* In practice, a sequence length can be up to 512, if not larger, but 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 the order of these words** (“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 a model because empirically, **NN’s don’t work well with inputs that aren’t unit-variance** (that’s why we scale our features)
* 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 2 positions will be too small for NN’s 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 w/ the previous sequence size of 8, positions go from 0 to 7:



* **The embedding size for positions is usually the same as the embedding size for words so that they can be summed**
* Ex: 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 Hugging Face’s BERT (August 2021)
* **Because embeddings change as the model weights get updated, we say that the position embeddings are *learned***
* **Position embeddings can also be *fixed***
* The embedding for each position is still a vector with Selements (S = the position embedding size), but each element is predefined using a function, usually sine and cosine
* Original **Transformer paper** = if the element is at an even index, use sine. Else, use cosine



* **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
* Ex: A task involving representations of 3D objects, such as a teapot.
* Each position on the surface of the teapot is represented by a 3D 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
* The following is the generalized format for the embedding vector at coordinate v, also called the **Fourier features of coordinate v**



* **Fourier features have been shown to improve models’ performance for tasks that take in coordinates (or positions) as inputs**

### Data Leakage

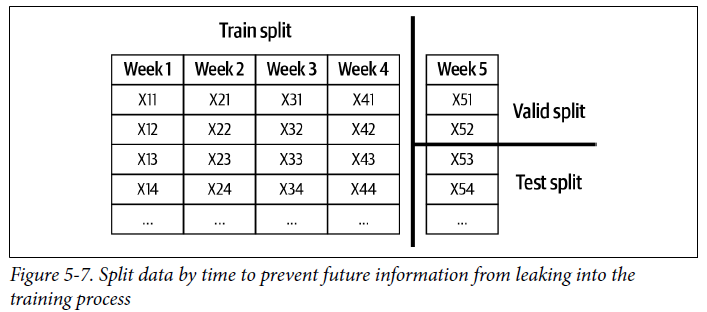
* Ex: Hundreds of AI tools built to catch Covid and none helped
* These models were trained to predict COVID-19 risks from medical scans, and there were multiple examples where ML models that performed well during evaluation failed to be usable in actual production settings
* In one example, researchers trained their model on a mix of scans taken when patients were lying down and standing up
* “Because patients scanned while lying down were more likely to be seriously ill, the model learned to predict serious covid risk from a person’s position”
* In some other cases, models were found to be picking up on the text font that certain hospitals used to label the scans. As a result, fonts from hospitals with more serious caseloads became predictors of covid risk
* Both of these are examples of **data leakage** = the phenomenon **when a form of the label “leaks” into the set of features used for making predictions, and this same information is not available during inference**
* Data leakage is challenging because **often the leakage is nonobvious**
* It’s **dangerous because it can cause your models to fail in an unexpected and spectacular way**, **even after extensive evaluation and testing**
* Ex: Suppose you want to build an ML model to predict if a CT scan of a lung shows signs of cancer
* Obtained the data from hospital A, removed diagnosis from the data, and trained your model
* It did really well on the test data from hospital A, but poorly on the data from hospital B.
* After extensive investigation, you learned that at hospital A, when doctors think that a patient has lung cancer, they send that patient to a more advanced scan machine, which outputs slightly different CT scan images
* Your model learned to rely on the information on the scan machine used to make predictions on whether a scan image shows signs of lung cancer
* Hospital B sends the patients to different CT scan machines at random, so your model has no information to rely on
* We say that **labels are leaked into the features during training.**
* **Data leakage can happen not only with newcomers to the field, but has also happened to several experienced researchers**
* **Despite its prevalence, data leakage is rarely covered in ML curricula**

#### Common Causes of Data Leakage

* Common causes for data leakage and how to avoid them:

##### 1) Splitting time-correlated data randomly instead of by time

* When learning ML, you’re **often taught to *randomly* split data** **into train, validation, and test splits**
* *This is also how data is often reportedly split in ML research papers*
* However, **this is also one common cause for data leakage.**
* **In many cases, data is time-correlated, which means that the time the data is generated affects its label distribution**
* **Sometimes, the correlation is obvious**, as in the case of stock prices
* To oversimplify it, the prices of similar stocks tend to move together 🡪 If 90% of tech stocks go down today, it’s very likely the other 10% of tech stocks go down too
* When building models to predict the future stock prices, you want to **split training data by time, such as training a model on data from the first 6 days + evaluating it on data from the 7th day**
* If you ***randomly* split**, **prices from the 7th day will be included in your train split and leak the condition of the market on that day into your model**
* We say that the **information *from the future* is leaked into the training process**
* **However, in many cases, the correlation is nonobvious**
* Ex: Predicting whether someone will click on a song recommendation
* Whether someone will listen to a song depends not *only* on their music taste but *also on the general music trend that day*
* If an artist passes away one day, people will be much more likely to listen to that artist
* By including samples from a certain day in the train split, information about the music trend that day will be passed into your model, *making it easier for it to make predictions on other samples on that same day*
* ***To prevent future information from leaking into the training process and allowing models to cheat during evaluation, split your data by time, instead of splitting randomly, whenever possible***
* Ex: If you have data from 5 weeks, use the first 4 weeks for the train split, then randomly split week 5 into validation and test splits



##### 2) Scaling before splitting

* It’s important to **scale your features**, which **requires global statistics** (e.g., mean, variance) of your data
* One **common mistake** is to **use the *entire* training data to generate global statistics before splitting it into different splits, leaking the mean and variance of the test samples into the training process, allowing a model to adjust its predictions for the test samples**
* ***This information isn’t available in production, so the model’s performance will likely degrade***
* To **avoid this type of leakage, always split your data first *before* scaling, then use the statistics from the *train* split to scale *all* the splits**
* Some even suggest that we split our data before any EDA and data processing, so that we don’t accidentally gain information about the test split

##### 3) Filling in missing data with statistics from the test split

* One common way to handle the missing values of a feature is to fill (input) them with the mean or median of *all* values present
* **Leakage might occur if the mean or median is calculated using entire data instead of just the train split**
* This type of leakage is **similar to the type of leakage caused by scaling**, and it can be **prevented by using only statistics from the train split to fill in missing values in *all* the splits**

##### 4) Poor handling of data duplication before splitting

* If you have **duplicates or near-duplicates** in your data, **failing to remove them before splitting your data might cause the same samples to appear in both train and validation/test splits**
* **Data duplication is quite common in the industry, and has also been found in popular research datasets** (CIFAR-10 and CIFAR-100, 2 popular datasets used for CV research
* **Data duplication can result from data collection or merging of different data sources**
* Data duplication **can also happen because of data processing**
* Ex: Oversampling might result in duplicating certain examples
* **To avoid this, always check for duplicates *before* splitting *and also after* splitting just to make sure**
* **If you oversample your data, do it *after* splitting**

##### 5) Group leakage

* A **group of examples have strongly correlated labels but are divided into different splits**
* Ex: A patient might have 2 lung CT scans that are a week apart, which likely have the same labels on whether they contain signs of lung cancer, but one of them is in the train split and the second is in the test split
* This **type of leakage is common for objective detection tasks that contain photos of the same object taken milliseconds apart** (some landed in the train split while others landed in the test split)
* **It’s hard avoiding this type of data leakage without understanding how your data was generated**

##### 6) Leakage from data generation process

* **Detecting this type of data leakage requires a deep understanding of the way data is collected**
* The example earlier about how information on whether a CT scan shows signs of lung cancer is leaked via the scan machine is an example of this type of leakage
* It would be very hard to figure out that the model’s poor performance in hospital B is due to its different scan machine procedure if you don’t know about different scan machines or that the procedures at the 2 hospitals are different.
* There’s **no foolproof way to avoid this type of leakage**, but you can **mitigate the risk by keeping track of the sources of your data and understanding how it is collected and processed**
* **Normalize your data so that data from different sources can have the same means and variances**
* Ex: If different CT scan machines output images with different resolutions, normalizing all images to have the same resolution would make it harder for models to know which image is from which scan machine
* ***And don’t forget to incorporate subject matter experts, who might have more contexts on how data is collected and used, into the ML design process!***

#### Detecting Data Leakage

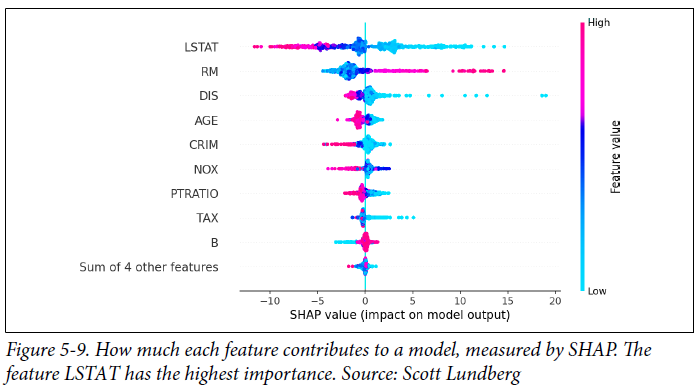
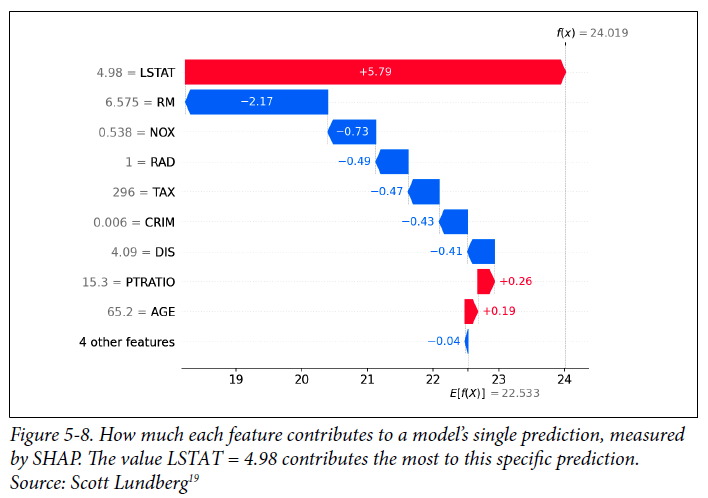
* **Data leakage can happen during many steps, from generating, collecting, sampling, splitting, and processing data to feature engineering**
* It’s **important to monitor for data leakage during the *entire lifecycle* of an ML project**
* **Measure the predictive power of each feature or a set of features with respect to the target variable (label)**
* **If a feature has unusually high correlation, investigate how this feature is generated and whether the correlation makes sense**
* It’s **possible that 2 features independently don’t contain leakage, but 2 features *together* can contain leakage**
* Ex: Building a model to predict how long an employee will stay at a company, start and end date separately don’t tell us much about their tenure, but both together can give us that information
* **Do ablation studies to measure how important a feature or a set of features is to your model**
* **If removing a feature causes the model’s performance to deteriorate significantly, investigate why that feature is so important**
* If you have a massive amount of features, say a thousand features, it **might be infeasible to do ablation studies on every possible combination of them, but it can still be useful to occasionally *do ablation studies with a subset of features that you suspect the most***
* *Another* example of **how subject matter expertise is in handy in feature engineering**
* Ablation studies **can be run offline at your own schedule, so you can leverage your** machines during **downtime for this purpose**
* **Keep an eye out for new features added to your model**
* **If adding a new feature significantly improves model performance, either that feature is really good or that feature just contains leaked information about labels**
* **Be very careful every time you look at the test split**
* **If you use the test split in any way other than to report a model’s final performance, whether to come up with ideas for new features or to tune hyperparameters, you risk leaking information from the future into your training process**

### Engineering Good Features

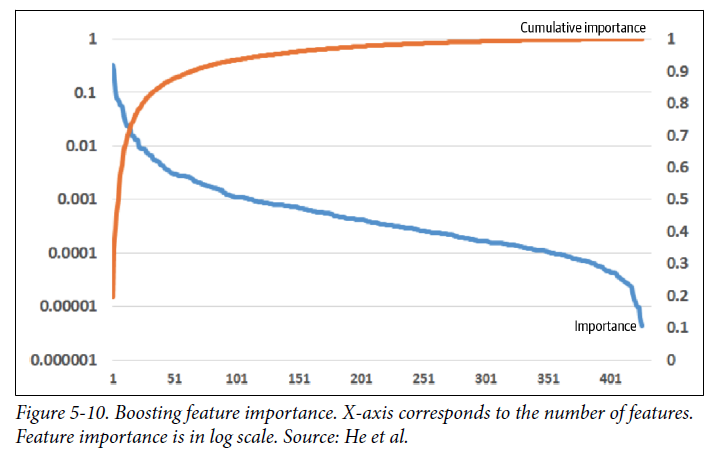
* ***Generally*, adding more features leads to better model performance**
* In my experience, **the list of features used for a model in production only grows over time**
* However, **more features doesn’t *ALWAYS* mean better model performance**
* Having too many features can be bad both during training and serving your model for the following reasons:
* **More opportunities there are for data leakage.**
* **Can cause overfitting.**
* **Can increase memory required to serve a model, which, in turn, might require you to use a more expensive machine/instance to serve your model.**
* **Can increase inference latency when doing online prediction, especially if you need to extract these features from raw data for prediction online**
* **Useless features become technical debts**
* *Whenever your data pipeline changes, all affected features need to be adjusted accordingly*
* Ex: if one day your application decides to no longer take in information about users’ age, all features that use users’ age need to be updated
* In theory, if a feature doesn’t help a model make good predictions, regularization techniques like L1 regularization *should* reduce that feature’s weight to 0
* **However, in practice, it might help models learn faster if features that are no longer useful (and even possibly harmful) are removed, prioritizing good features**
* You **can store removed features to add them back later**
* Can also just store general feature definitions to reuse + share across teams in an organization
* When talking about feature definition management, some people might think of **feature stores** as the solution
* *However, not all feature stores manage feature definitions*
* 2 factors to consider when evaluating whether a feature is good for a model: **importance** **to the** **model** and **generalization to unseen data**.

#### Feature Importance

* There are many different methods for **measuring a feature’s importance**
* If you use a **classical ML algorithm** (ex: boosted gradient trees), the easiest way to measure the importance of your features is to **use built-in feature importance functions**
* For more **model-agnostic methods**, you might want to look into **SHAP (SHapley Additive exPlanations)**
* The **exact algorithm for feature importance measurement is complex,** but intuitively, a **feature’s importance to a model is measured by how much that model’s performance deteriorates if that feature or a set of features containing that feature is removed from the model**
* *SHAP is great because it not only measures a feature’s importance to an entire model, it also measures each feature’s contribution to a model’s specific prediction.*
* Figures below show how SHAP can help you understand the contribution of each feature to a model’s predictions



* **Often, a small number of features accounts for a large portion of your model’s feature importance**
* Ex: When measuring feature importance for a CTR prediction model, the Facebook ads team found out that the top 10 features are responsible for ~50% of the model’s total feature importance, whereas the last 300 features contribute < 1% feature importance



* **Not only good for choosing the right features, feature importance techniques are also great for interpretability as they help you understand how your models work under the hood**

#### Feature Generalization

* Since the **goal of an ML model is to make correct predictions on unseen data**, **features used for the model should generalize to unseen data**
* ***Not all features generalize equally***
* Ex: for the task of predicting whether a comment is spam, the identifier of each comment is not generalizable at all and shouldn’t be used as a feature for the model
* However, the identifier of the user who posts the comment, such as username, might still be useful for a model to make predictions
* **Measuring feature generalization is a lot less scientific than measuring feature importance**, and it **requires both intuition and subject matter expertise on top of statistical knowledge**
* **2 aspects you might want to consider with regards to generalization**: **feature coverage** and **distribution of feature values**.
* **Coverage** is the **percentage of the samples that has values for this feature in the data**
* So, the fewer values that are missing, the higher the coverage.
* *Rough* rule of thumb = if a feature appears in a very small % of your data, it’s not going to be very generalizable.
* Ex: To build a model to predict whether someone will buy a house in the next 12 months, if you think the number of children someone has will be a good feature, but you can only get this information for 1% of your data, this feature might not be very useful
* **This rule of thumb is rough because some features can still be useful even if they are missing in most of your data**
* **Especially true when the missing values are not at random**, which means ***having the feature or not might be a strong indication of its value***
* Ex: If a feature appears only in 1% of your data, but 99% of the examples *with* this feature have POSITIVE labels, this feature is useful and you should use it
* **Coverage of a feature can differ wildly between different slices of data and even in the same slice of data over time**
* **If the coverage of a feature differs a lot between the train and test split** (it appears in 90% of the examples in the train split but only in 20% of the examples in the test split), **this is an indication that your train and test splits don’t come from the same distribution**
* You might want to investigate whether the way you split your data makes sense and whether this feature is a cause for data leakage
* **For the feature values that *are* present**, you might want to **look into their** **distribution**.
* If the **set of values that appears in the seen data** (such as the train split) **has *NO* overlap with the set of values that appears in the unseen data** (such as the test split), **this feature might even hurt your model’s performance**
* Ex: You want to build a model to estimate the time it will take for a given taxi ride
* You retrain this model every week, + want to use the data from the last 6 days to predict the ETAs for today
* One of the features is DAY\_OF\_THE\_WEEK, which you think is useful, because traffic on weekdays is usually worse than on the weekend
* This feature coverage is 100%, because it’s present in every sample
* However, in the train split, the values for this feature are Monday to Saturday, whereas in the test split, the value for this feature is Sunday
* If you include this feature in your model without a clever scheme to encode the days, it won’t generalize to the test split, + might harm your model’s performance
* On the other hand, HOUR\_OF\_THE\_DAY is a great feature, because the time in the day affects the traffic too, and *the range of values for this feature in the train split overlaps with the test split 100%*
* **When considering a feature’s generalization, there’s a trade-off between generalization and specificity**
* Ex: Might realize that traffic during an hour only changes depending on whether that hour is the rush hour
* So, you generate the feature IS\_RUSH\_HOUR and set it to 1 if the hour is between 7 a.m. and 9 a.m. or between 4 p.m. and 6 p.m
* **IS\_RUSH\_HOUR is more generalizable but less specific than HOUR\_OF\_THE\_DAY**
* Using IS\_RUSH\_HOUR *without* HOUR\_OF\_THE\_DAY might cause models to lose important information about the hour

### Summary

* **Because the success of today’s ML systems still depends on their features, it’s important for organizations interested in using ML in production to invest time and effort into feature engineering**
* **How to engineer “good” features is a complex question with no foolproof answers**
* The best way to learn is **through experience**: trying out different features and observing how they affect your models’ performance
* It’s also possible to **learn from experts**
* **Feature engineering often involves subject matter expertise**, and **subject matter experts might not always be engineers**, so **it’s important to design your workflow in a way that allows non-engineers to contribute to the process**.
* Summary of best practices for feature engineering:
* **Split data by time into train/valid/test splits instead of doing it randomly**
* **If you oversample your data, do it after splitting**
* **Scale and normalize your data after splitting to avoid data leakage.**
* **Use statistics from only the train split, instead of the entire data, to scale features and handle missing values.**
* **Understand how your data is generated, collected, and processed**, + **involve domain experts** if possible.
* **Keep track of your data’s lineage.**
* **Understand feature importance to your model.**
* **Use features that generalize well.**
* **Remove no longer useful features from your models.**
* **With a set of good features**, the next part of the workflow = training ML models
* **Moving to modeling doesn’t mean we’re done with handling data or feature engineering**
* We are **NEVER done with data and features**
* **In most real-world ML projects, the process of collecting data and feature engineering goes on as long as your models are in production**