Feature Engineering

### **Mapping numeric values**

Integer and floating-point data don't need a special encoding because they can be multiplied by a numeric weight. As suggested in Figure 2, converting the raw integer value 6 to the feature value 6.0 is trivial:

### **Mapping categorical values**

[Categorical features](https://developers.google.com/machine-learning/glossary#categorical_data) have a discrete set of possible values. For example, there might be a feature called street\_name with options that include:

{'Charleston Road', 'North Shoreline Boulevard', 'Shorebird Way', 'Rengstorff Avenue'}

We can accomplish this by defining a mapping from the feature values, which we'll refer to as the **vocabulary** of possible values, to integers. Since not every street in the world will appear in our dataset, we can group all other streets into a catch-all "other" category, known as an **OOV (out-of-vocabulary) bucket**.

Using this approach, here's how we can map our street names to numbers:

* map Charleston Road to 0
* map North Shoreline Boulevard to 1
* map Shorebird Way to 2
* map Rengstorff Avenue to 3
* map everything else (OOV) to 4

However, if we incorporate these index numbers directly into our model, it will impose some constraints that might be problematic:

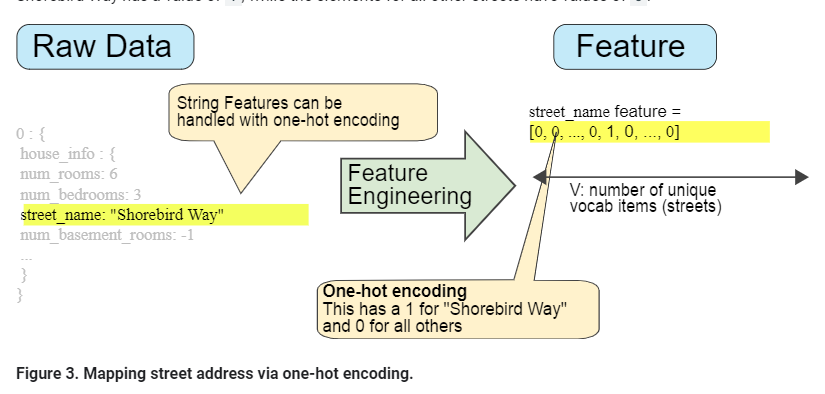
* We'll be learning a single weight that applies to all streets. For example, if we learn a weight of 6 for street\_name, then we will multiply it by 0 for Charleston Road, by 1 for North Shoreline Boulevard, 2 for Shorebird Way and so on. Consider a model that predicts house prices using street\_name as a feature. It is unlikely that there is a linear adjustment of price based on the street name, and furthermore this would assume you have ordered the streets based on their average house price. Our model needs the flexibility of learning different weights for each street that will be added to the price estimated using the other features.
* We aren't accounting for cases where street\_name may take multiple values. For example, many houses are located at the corner of two streets, and there's no way to encode that information in the street\_name value if it contains a single index.

To remove both these constraints, we can instead create a binary vector for each categorical feature in our model that represents values as follows:

* For values that apply to the example, set corresponding vector elements to 1.
* Set all other elements to 0.

The length of this vector is equal to the number of elements in the vocabulary. This representation is called a **one-hot encoding** when a single value is 1, and a **multi-hot encoding** when multiple values are 1.

Figure 3 illustrates a one-hot encoding of a particular street: Shorebird Way. The element in the binary vector for Shorebird Way has a value of 1, while the elements for all other streets have values of 0.



One-hot encoding extends to numeric data that you do not want to directly multiply by a weight, such as a postal code.

### **Sparse Representation**

Suppose that you had 1,000,000 different street names in your data set that you wanted to include as values for street\_name. Explicitly creating a binary vector of 1,000,000 elements where only 1 or 2 elements are true is a very inefficient representation in terms of both storage and computation time when processing these vectors. In this situation, a common approach is to use a [sparse representation](https://developers.google.com/machine-learning/glossary#sparse_representation) in which only nonzero values are stored. In sparse representations, an independent model weight is still learned for each feature value, as described above.

# Qualities of Good Features

### **Avoid rarely used discrete feature values**

Good feature values should appear more than 5 or so times in a data set. Doing so enables a model to learn how this feature value relates to the label. That is, having many examples with the same discrete value gives the model a chance to see the feature in different settings, and in turn, determine when it's a good predictor for the label. For example, a house\_type feature would likely contain many examples in which its value was victorian:

✔house\_type: victorian

Conversely, if a feature's value appears only once or very rarely, the model can't make predictions based on that feature. For example, unique\_house\_id is a bad feature because each value would be used only once, so the model couldn't learn anything from it:

✘unique\_house\_id: 8SK982ZZ1242Z

### **Prefer clear and obvious meanings**

Each feature should have a clear and obvious meaning to anyone on the project. For example, the following good feature is clearly named and the value makes sense with respect to the name:

✔house\_age\_years: 27

### **Don't mix "magic" values with actual data**

Good floating-point features don't contain peculiar out-of-range discontinuities or "magic" values. For example, suppose a feature holds a floating-point value between 0 and 1. So, values like the following are fine:

✔quality\_rating: 0.82

quality\_rating: 0.37

However, if a user didn't enter a quality\_rating, perhaps the data set represented its absence with a magic value like the following:

✘quality\_rating: -1

To explicitly mark magic values, create a Boolean feature that indicates whether or not a quality\_rating was supplied. Give this Boolean feature a name like is\_quality\_rating\_defined.

### **Account for upstream instability**

The definition of a feature shouldn't change over time. For example, the following value is useful because the city name probably won't change. (Note that we'll still need to convert a string like "br/sao\_paulo" to a one-hot vector.)

✔city\_id: "br/sao\_paulo"

But gathering a value inferred by another model carries additional costs. Perhaps the value "219" currently represents Sao Paulo, but that representation could easily change on a future run of the other model:

✘inferred\_city\_cluster: "219"

# Cleaning Data

### **Scaling feature values**

**Scaling** means converting floating-point feature values from their natural range (for example, 100 to 900) into a standard range (for example, 0 to 1 or -1 to +1). If a feature set consists of only a single feature, then scaling provides little to no practical benefit. If, however, a feature set consists of multiple features, then feature scaling provides the following benefits:

* Helps gradient descent converge more quickly.
* Helps avoid the "NaN trap," in which one number in the model becomes a [NaN](https://wikipedia.org/wiki/NaN) (e.g., when a value exceeds the floating-point precision limit during training), and—due to math operations—every other number in the model also eventually becomes a NaN.
* Helps the model learn appropriate weights for each feature. Without feature scaling, the model will pay too much attention to the features having a wider range.

You don't have to give every floating-point feature exactly the same scale. Nothing terrible will happen if Feature A is scaled from -1 to +1 while Feature B is scaled from -3 to +3. However, your model will react poorly if Feature B is scaled from 5000 to 100000.

One obvious way to scale numerical data is to linearly map [min value, max value] to a small scale, such as [-1, +1].

Another popular scaling tactic is to calculate the Z score of each value. The Z score relates the number of standard deviations away from the mean. In other words:

scaledvalue=(value−mean)/stddev.

For example, given:

* mean = 100
* standard deviation = 20
* original value = 130

then:

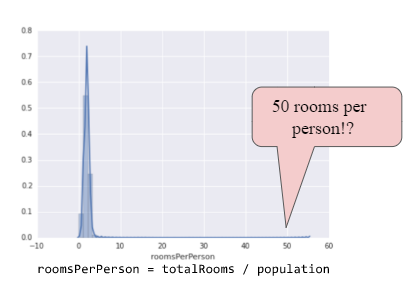
scaled\_value = (130 - 100) / 20

scaled\_value = 1.5

Scaling with Z scores means that most scaled values will be between -3 and +3, but a few values will be a little higher or lower than that range.

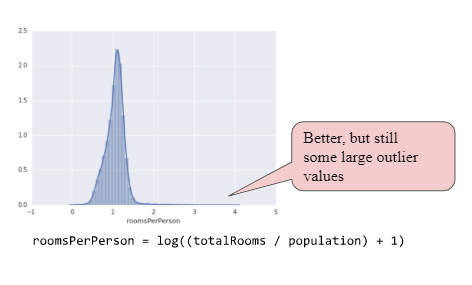
### **Handling extreme outliers**

The following plot represents a feature called roomsPerPerson from the [California Housing data set](https://developers.google.com/machine-learning/crash-course/california-housing-data-description). The value of roomsPerPerson was calculated by dividing the total number of rooms for an area by the population for that area. The plot shows that the vast majority of areas in California have one or two rooms per person. But take a look along the x-axis.



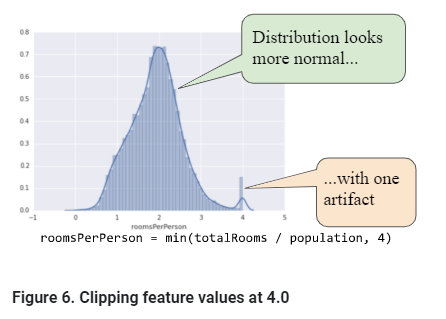
**Figure 4. A verrrrry lonnnnnnng tail.**

How could we minimize the influence of those extreme outliers? Well, one way would be to take the log of every value:



**Figure 5. Logarithmic scaling still leaves a tail.**

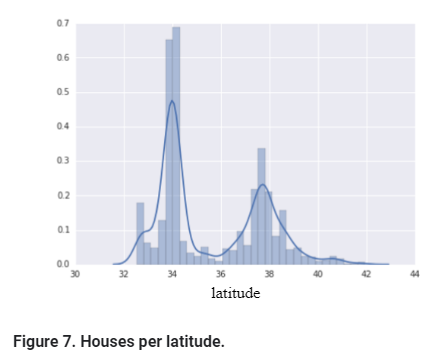
Log scaling does a slightly better job, but there's still a significant tail of outlier values. Let's pick yet another approach. What if we simply "cap" or "clip" the maximum value of roomsPerPerson at an arbitrary value, say 4.0?



Clipping the feature value at 4.0 doesn't mean that we ignore all values greater than 4.0. Rather, it means that all values that were greater than 4.0 now become 4.0. This explains the funny hill at 4.0. Despite that hill, the scaled feature set is now more useful than the original data.

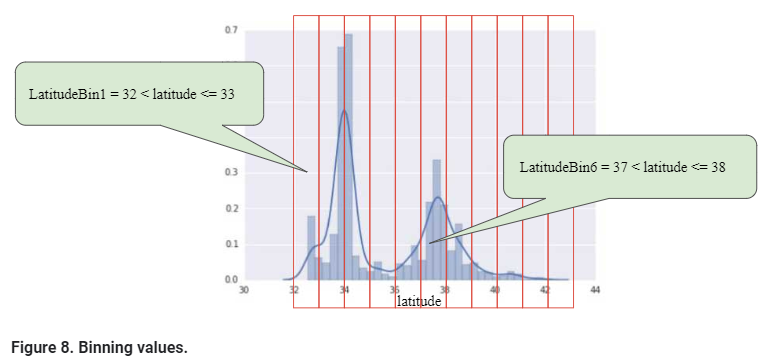
### **Binning**

The following plot shows the relative prevalence of houses at different latitudes in California. Notice the clustering—Los Angeles is about at latitude 34 and San Francisco is roughly at latitude 38.



In the data set, latitude is a floating-point value. However, it doesn't make sense to represent latitude as a floating-point feature in our model. That's because no linear relationship exists between latitude and housing values. For example, houses in latitude 35 are not 3534 more expensive (or less expensive) than houses at latitude 34. And yet, individual latitudes probably are a pretty good predictor of house values.

To make latitude a helpful predictor, let's divide latitudes into "bins" as suggested by the following figure:



Instead of having one floating-point feature, we now have 11 distinct boolean features (LatitudeBin1, LatitudeBin2, ..., LatitudeBin11). Having 11 separate features is somewhat inelegant, so let's unite them into a single 11-element vector. Doing so will enable us to represent latitude 37.4 as follows:

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

Thanks to binning, our model can now learn completely different weights for each latitude.

For simplicity's sake in the latitude example, we used whole numbers as bin boundaries. Had we wanted finer-grain resolution, we could have split bin boundaries at, say, every tenth of a degree. Adding more bins enables the model to learn different behaviors from latitude 37.4 than latitude 37.5, but only if there are sufficient examples at each tenth of a latitude.

Another approach is to bin by [quantile](https://wikipedia.org/wiki/Quantile), which ensures that the number of examples in each bucket is equal. Binning by quantile completely removes the need to worry about outliers.

### **Scrubbing**

Until now, we've assumed that all the data used for training and testing was trustworthy. In real-life, many examples in data sets are unreliable due to one or more of the following:

* **Omitted values.** For instance, a person forgot to enter a value for a house's age.
* **Duplicate examples.** For example, a server mistakenly uploaded the same logs twice.
* **Bad labels.** For instance, a person mislabeled a picture of an oak tree as a maple.
* **Bad feature values.** For example, someone typed in an extra digit, or a thermometer was left out in the sun.

Once detected, you typically "fix" bad examples by removing them from the data set. To detect omitted values or duplicated examples, you can write a simple program. Detecting bad feature values or labels can be far trickier.

In addition to detecting bad individual examples, you must also detect bad data in the aggregate. Histograms are a great mechanism for visualizing your data in the aggregate. In addition, getting statistics like the following can help:

* Maximum and minimum
* Mean and median
* Standard deviation

Consider generating lists of the most common values for discrete features. For example, do the number of examples with country:uk match the number you expect. Should language:jp really be the most common language in your data set?

### **Know your data**

Follow these rules:

* Keep in mind what you think your data should look like.
* Verify that the data meets these expectations (or that you can explain why it doesn’t).
* Double-check that the training data agrees with other sources (for example, dashboards).

Treat your data with all the care that you would treat any mission-critical code. Good ML relies on good data.