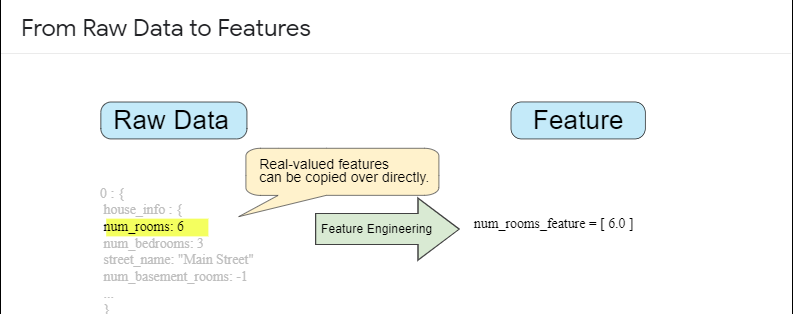
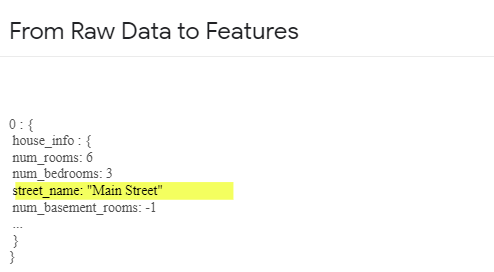
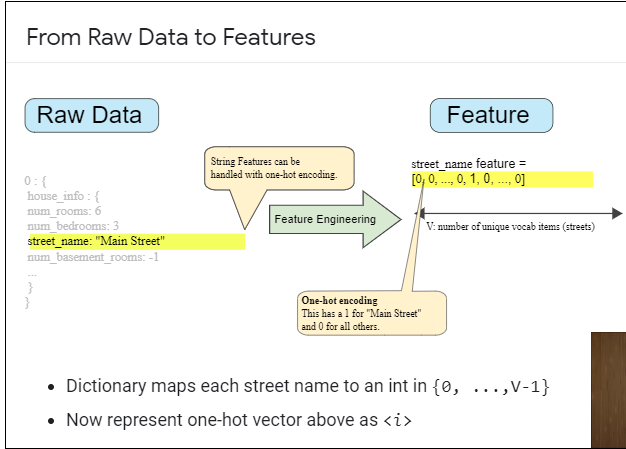
**Properties of the Feature:**

**If feature is a integer:**



**If feature is a string:**





**How to select a feature:**

**Properties of a Good Feature**

Feature values should appear with non-zero value more than a small handful of times in the dataset.

my\_device\_id:8SK982ZZ1242Z

device\_model:galaxy\_s6

## Properties of a Good Feature

Feature values should appear with non-zero value more than a small handful of times in the dataset.

my\_device\_id:8SK982ZZ1242Z

device\_model:galaxy\_s6

Features shouldn't take on "magic" values

(use an additional boolean feature like is\_watch\_time\_defined instead!)

watch\_time: -1.0

watch\_time: 1.023

watch\_time\_is\_defined: 1.0

The definition of a feature shouldn't change over time.

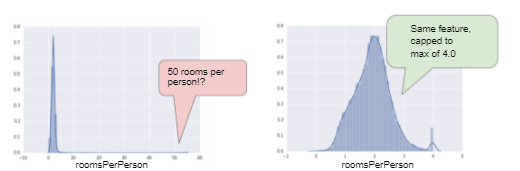
(Beware of depending on other ML systems!)

city\_id:"br/sao\_paulo"

inferred\_city\_cluster\_id:219

Distribution should not have extreme outliers

Ideally all features transformed to a similar range, like (-1, 1) or (0, 5).



## Good Habits

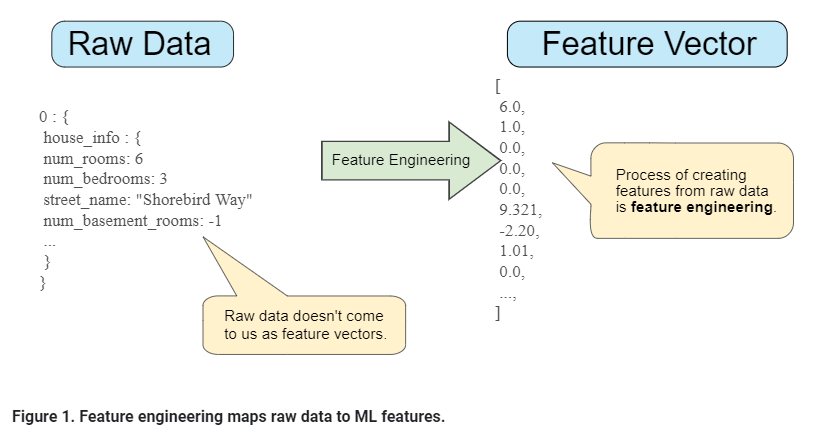
**KNOW YOUR DATA**

* **Visualize**: Plot histograms, rank most to least common.
* **Debug**: Duplicate examples? Missing values? Outliers? Data agrees with dashboards? Training and Validation data similar?
* **Monitor**: Feature quantiles, number of examples over time?

## Mapping Raw Data to Features

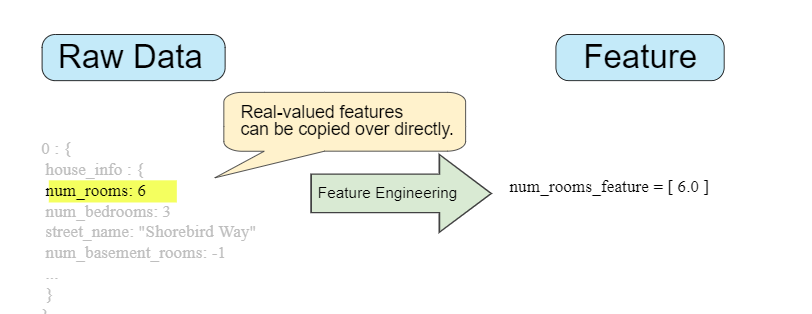
The left side of Figure 1 illustrates raw data from an input data source; the right side illustrates a **feature vector**, which is the set of floating-point values comprising the examples in your data set. **Feature engineering** means transforming raw data into a feature vector. Expect to spend significant time doing feature engineering.

Many machine learning models must represent the features as real-numbered vectors since the feature values must be multiplied by the model weights.



### 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'}

Since models cannot multiply strings by the learned weights, we use feature engineering to convert strings to numeric values.

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:

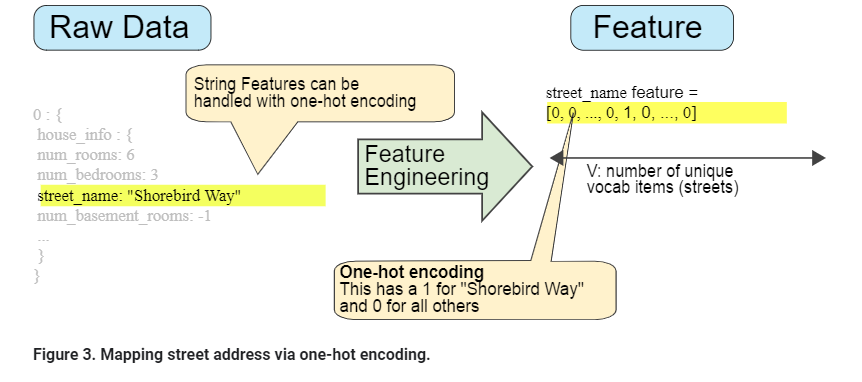
* We'll be learning a single weight that applies to all streets. For example, if we learn a weight of 6 for street\_name, then we will multiply it by 0 for Charleston Road, by 1 for North Shoreline Boulevard, 2 for Shorebird Way and so on. Consider a model that predicts house prices using street\_name 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.



### 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.

# Representation: Qualities of Good Features



**Estimated Time:** 10 minutes

We've explored ways to map raw data into suitable feature vectors, but that's only part of the work. We must now explore what kinds of values actually make good features within those feature vectors.

### 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

Conversely, the meaning of the following feature value is pretty much indecipherable to anyone but the engineer who created it:

✘house\_age: 851472000

In some cases, noisy data (rather than bad engineering choices) causes unclear values. For example, the following user\_age\_years came from a source that didn't check for appropriate values:

✘user\_age\_years: 277

### 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.

In the original feature, replace the magic values as follows:

* For variables that take a finite set of values (discrete variables), add a new value to the set and use it to signify that the feature value is missing.
* For continuous variables, ensure missing values do not affect the model by using the mean value of the feature's data.

### 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"

# Representation: 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.

### 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?