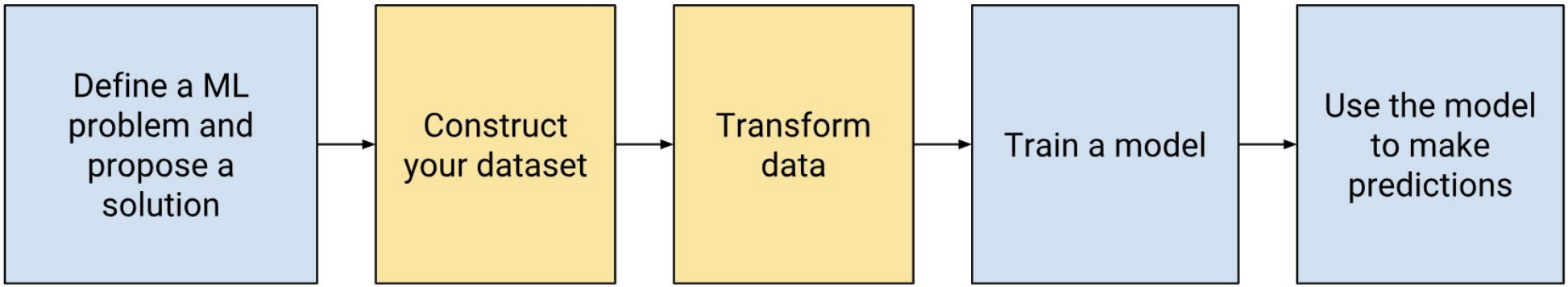
Data types and partitioning



Behind the hood - 1

1. The algorithms take a subset of observations called as the **training data** and tests them on a different subset of data called as the **test data**.

2. **Dev (development set)** - this is used to tune parameters, select features, and make other decisions regarding the learning algorithm. It is also sometimes called the **hold-out cross validation set**

3. **Test set -** this is used to evaluate the performance of the algorithm, but not to make any decisions regarding what learning algorithm or parameters to use.

- Dev set and test set data should come from the same distribution

Behind the hood-II

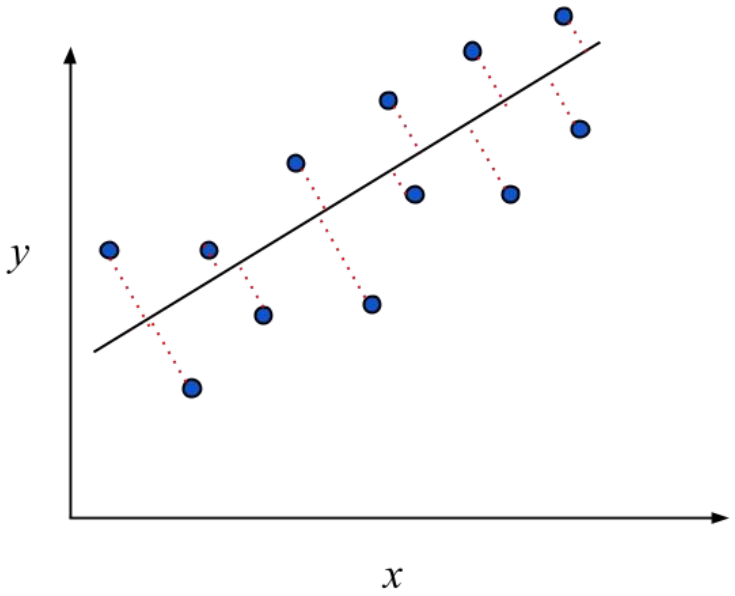
4. The error between the prediction of the outcome variable the actual data is evaluated as **test error**.

5. The **objective function** of the algorithm is to minimise these test errors by tuning the parameters of the hypothesis.

6. Models that successfully capture these desired outcomes are further evaluated for **Bias** and **Variance**(overfitting and underfitting).

For linear models

architecture



hypothesis

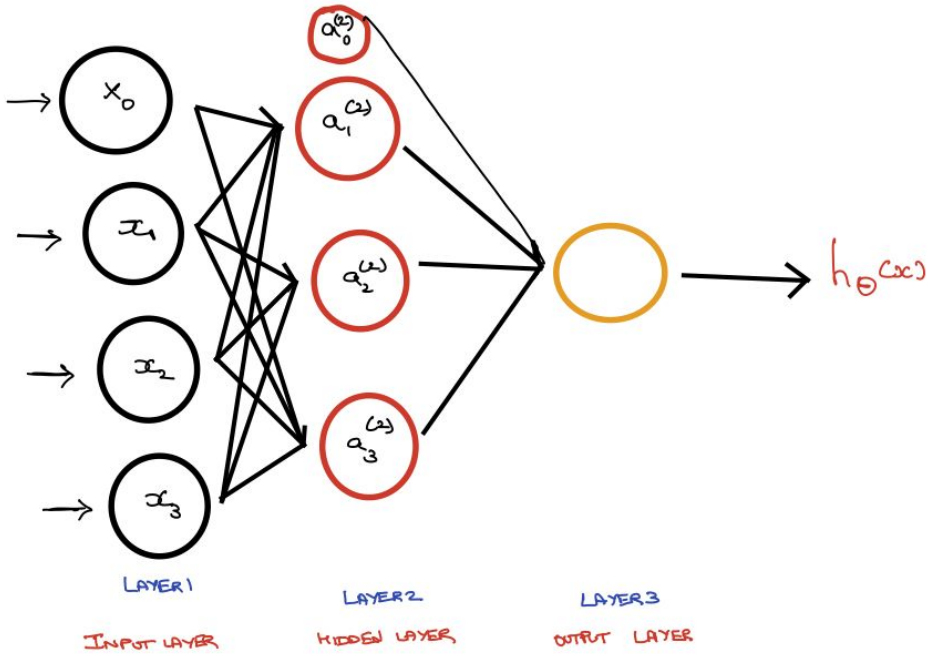
$$h_\theta(x) = \theta_1 x + \theta_0$$

Cost function = $$1/2n \sum_{i=1}^{n} (h_\theta(x^i) - y^i)^2$$

Machine learning algorithm tries to **minimise** this Cost Function

For neural network models

architecture



hypothesis

$$a_1^{(2)} = g(\theta_{10}^{(1)} x_0 + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2 + \theta_{13}^{(1)} x_3)$$
$$a_2^{(2)} = g(\theta_{20}^{(1)} x_0 + \theta_{21}^{(1)} x_1 + \theta_{22}^{(1)} x_2 + \theta_{23}^{(1)} x_3)$$
$$a_3^{(2)} = g(\theta_{30}^{(1)} x_0 + \theta_{31}^{(1)} x_1 + \theta_{32}^{(1)} x_2 + \theta_{33}^{(1)} x_3)$$
$$h_\theta(x) = a_1^{(3)} = g(\theta_{10}^{(2)} a_0^{(2)} + \theta_{11}^{(2)} a_1^{(2)} + \theta_{12}^{(1)} a_2^{(2)} + \theta_{13}^{(2)} a_3^{(2)})$$

Cost function =

$$CF(\Theta) = -1/m \left[\sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) log(1 - (h_\Theta(x^{(i)}))_k) \right]$$
$$+\lambda/2m \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{S_{l+1}} (\Theta_{ji}^{(l)})^2$$

Data transformation

If you had to prioritize improving one of the areas below in your machine learning project, which would have the most impact?

Using the latest optimization algorithm

A more clever loss function

A deeper network

The quality and size of your data

How large Dev set & test set should be?

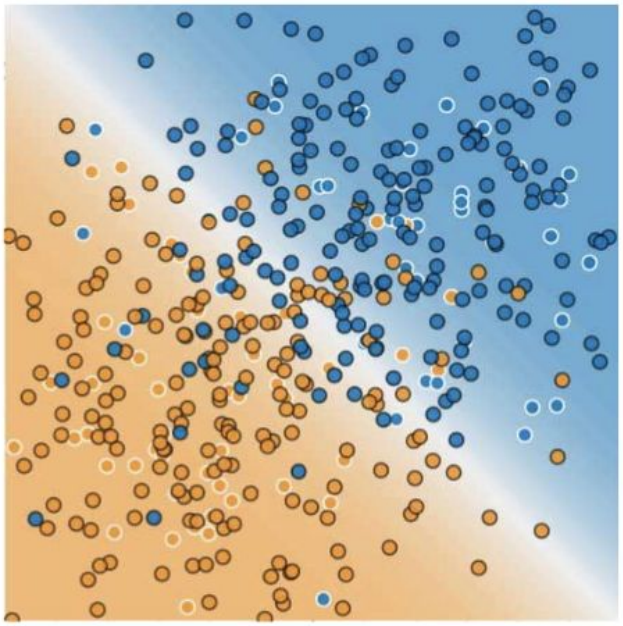
The dev set should be large enough to detect differences between algorithms that you are trying out.

if classifier A has an accuracy of 95.0% and classifier B has an accuracy of 95.1%, then a dev set of 100 examples would not be able to detect this 0.1% difference.

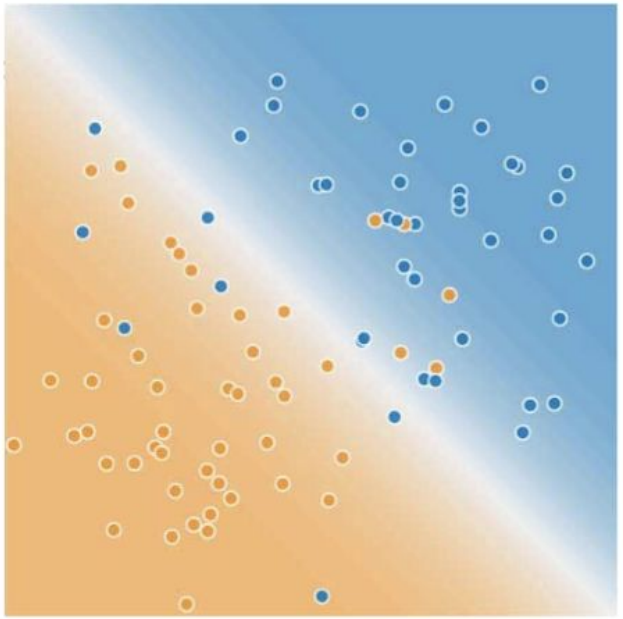
Dev sets with sizes from 1,000 to 10,000 examples are common. With 10,000 examples, there is a good chance of detecting an improvement of 0.1%

There is no need to have excessively large test sets beyond what is needed to evaluate the performance of your algorithms. For example 30% is large in the big data world

How large Dev set & test set should be?



Training Data



Test Data

More training better model

Better confidence intervals

For smaller dataset we do cross validation

Constructing data

Collect the raw data.
Identify feature and label sources.
Select a sampling strategy.
Split the data.

You're on a brand new machine learning project, about to select your first features. How many features should you pick?

Pick 1-3 features that seem to have strong predictive power.
Pick 4-6 features that seem to have strong predictive power.
Pick as many features as you can, so you can start observing which features have the strongest predictive power.

Know your data

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.

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

Data transformation - sanity check

Explore, Clean, and Visualize Your Data

Explore and clean up your data before performing any transformations on it.

Examine several rows of data.

Check basic statistics.

Fix missing numerical entries.

What type of data do you have?

Transactional log - record a specific event

Attribute data contains snapshots of information.

Aggregate statistics create an attribute from multiple transactional logs

Identifying features

Labelled

housingMedianAge (feature)	totalRooms (feature)	totalBedrooms (feature)	medianHouseValue (label)
15	5612	1283	66900
19	7650	1901	80100
17	720	174	85700
14	1501	337	73400
20	1454	326	65500

UnLabelled

housingMedianAge (feature)	totalRooms (feature)	totalBedrooms (feature)
42	1686	361
34	1226	180
33	1077	271

Identifying label and sources

The best label is a **direct label** of what you want to predict

A **derived label** because it does not directly measure what you want to predict

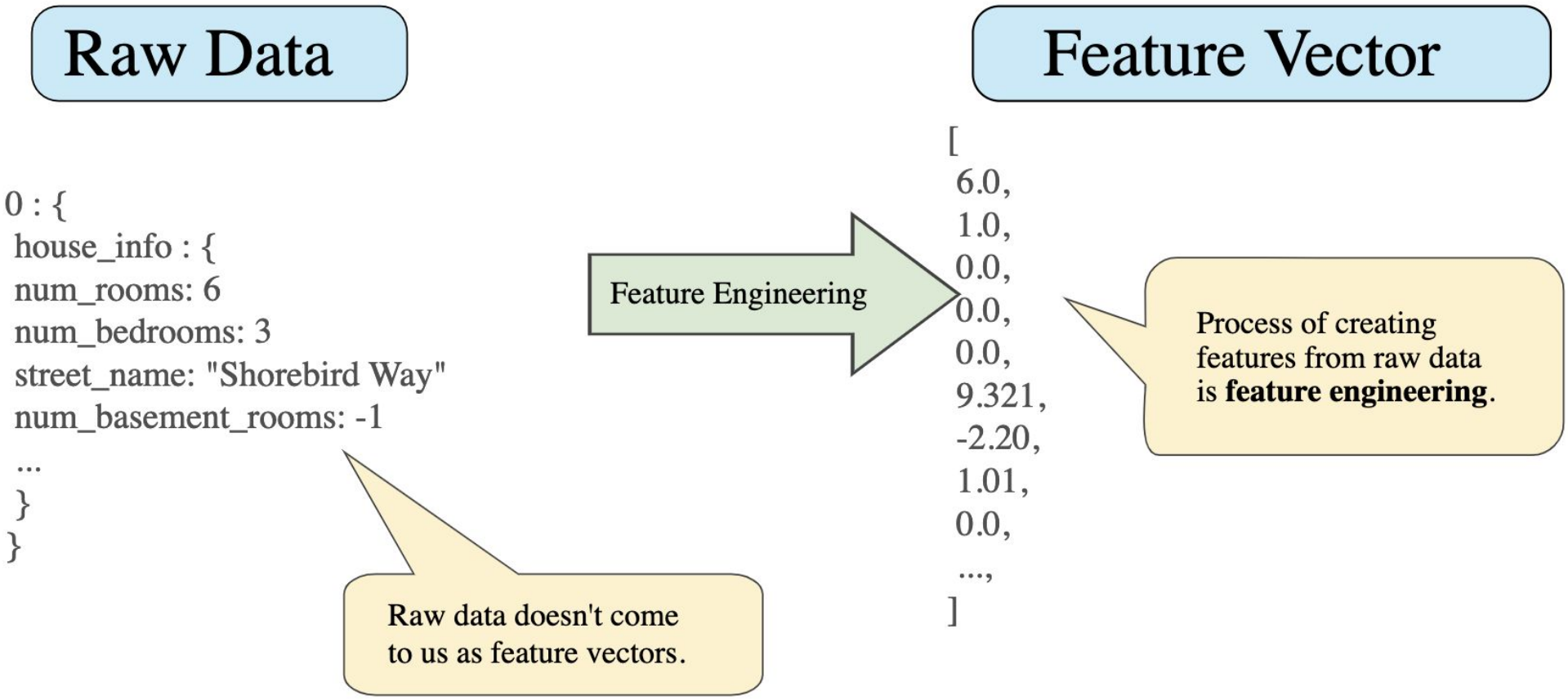
The output of your model could be either an Event or an Attribute. This results in the following two types of labels:

Direct label for Events, such as “Did the user click the top search result?”
Direct label for Attributes, such as “Will the advertiser spend more than $X in the next week?”

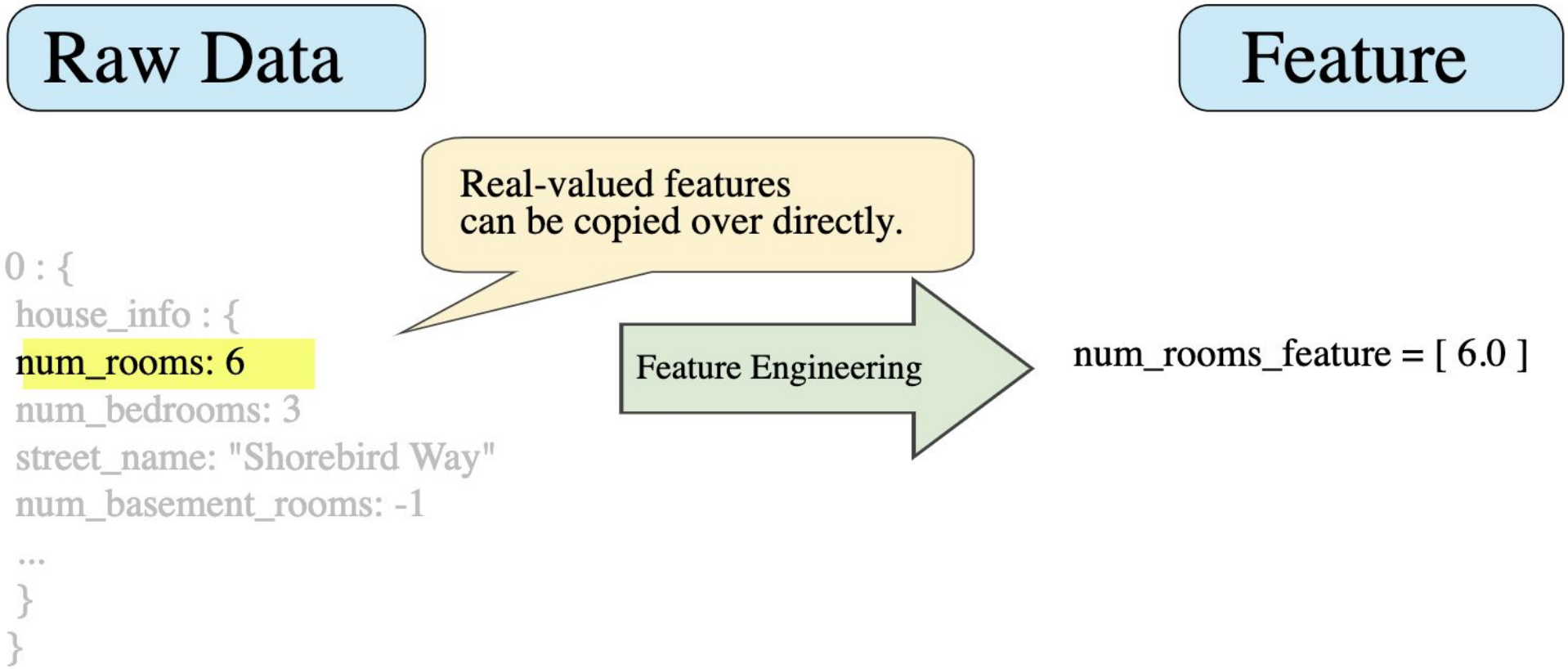
Qualities of good features

1. Avoid rarely used discrete feature values
2. Prefer clear and obvious meanings
3. Don't mix "magic" values with actual data
4. Feature shouldn't change over time

Feature engineering

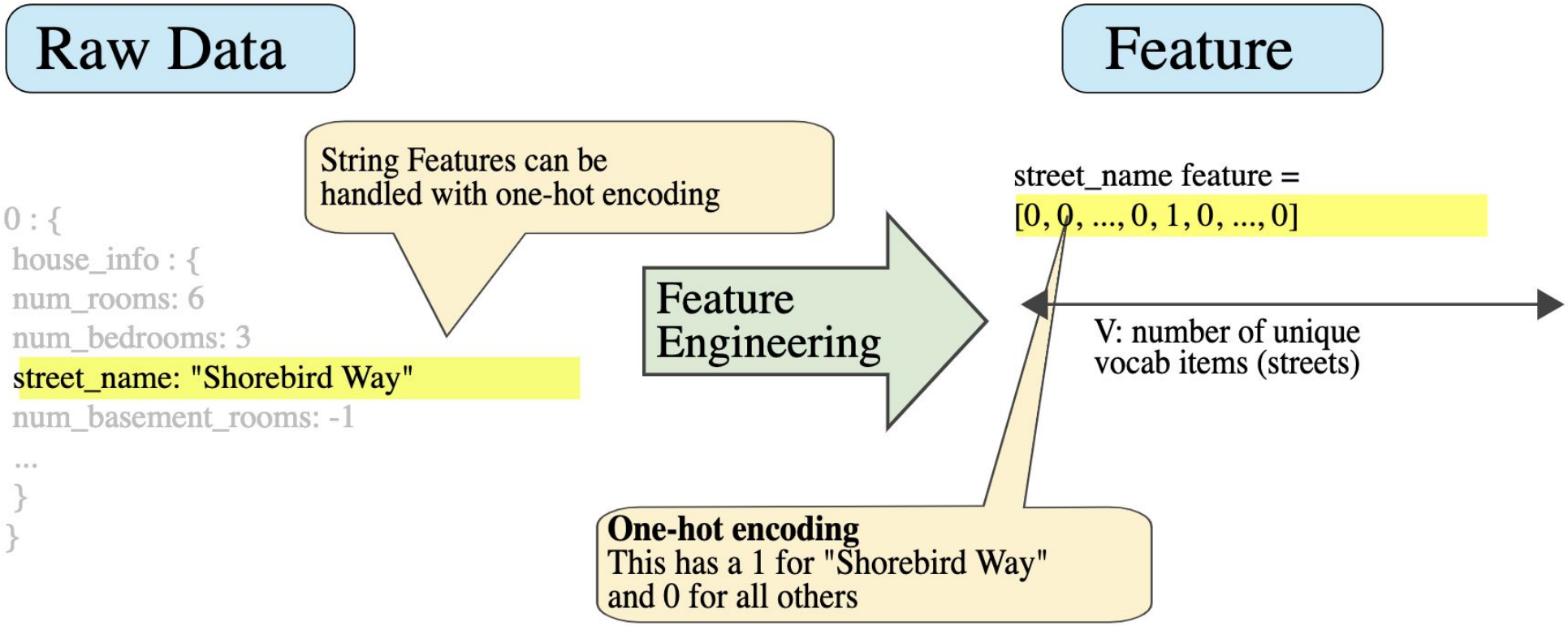


Mapping numeric values



Integer and floating-point data don't need a special encoding because they can be multiplied by a numeric weight.

Mapping categorical values



Features having a discrete set of possible values. For example, consider a categorical feature named house style, which has a discrete set of three possible values: Tudor, ranch, colonial

Numerical Data transformation - scaling

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)

Helps gradient descent converge more quickly.
Helps avoid the "NaN trap," in which one number in the model becomes a 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

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:

$$scaledvalue = (value - mean)/stddev.$$

Numerical Data transformation - normalization

You may need to apply two kinds of transformations to numeric data:

- **Normalizing** - transforming numeric data to the same scale as other numeric data.
- **Bucketing** - transforming numeric (usually continuous) data to categorical data.

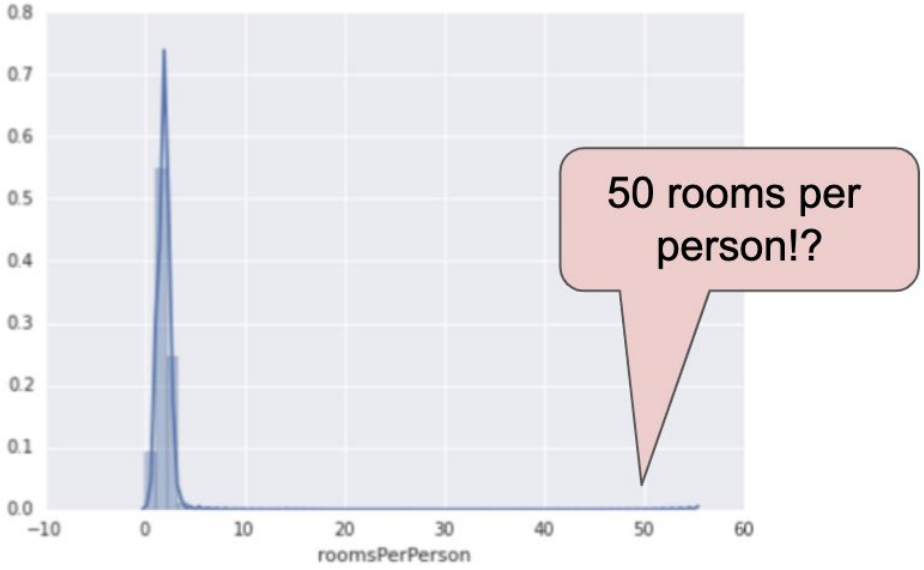
Why Normalize Numeric Features?

Normalization is necessary if you have very different values within the same feature (for example, city population). Without normalization, your training could blow up with NaNs if the gradient update is too large.

You might have two different features with widely different ranges (e.g., age and income), causing the gradient descent to "bounce" and slow down convergence

Data transformation - Normalisation

Handling extreme outliers

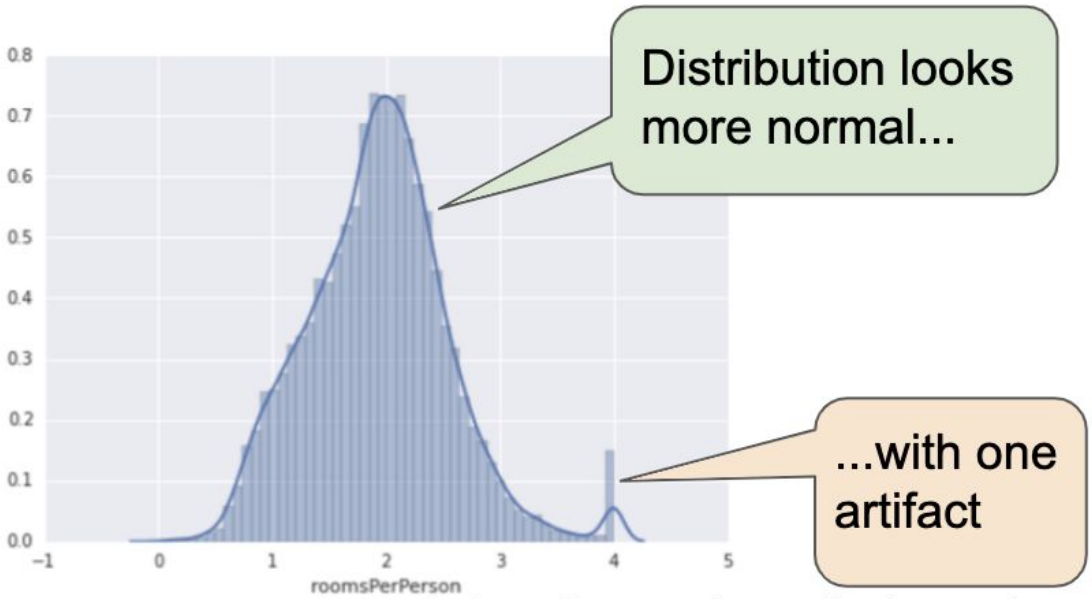



```
roomsPerPerson = totalRooms / population
```


```
roomsPerPerson = log((totalRooms / population) + 1)
```

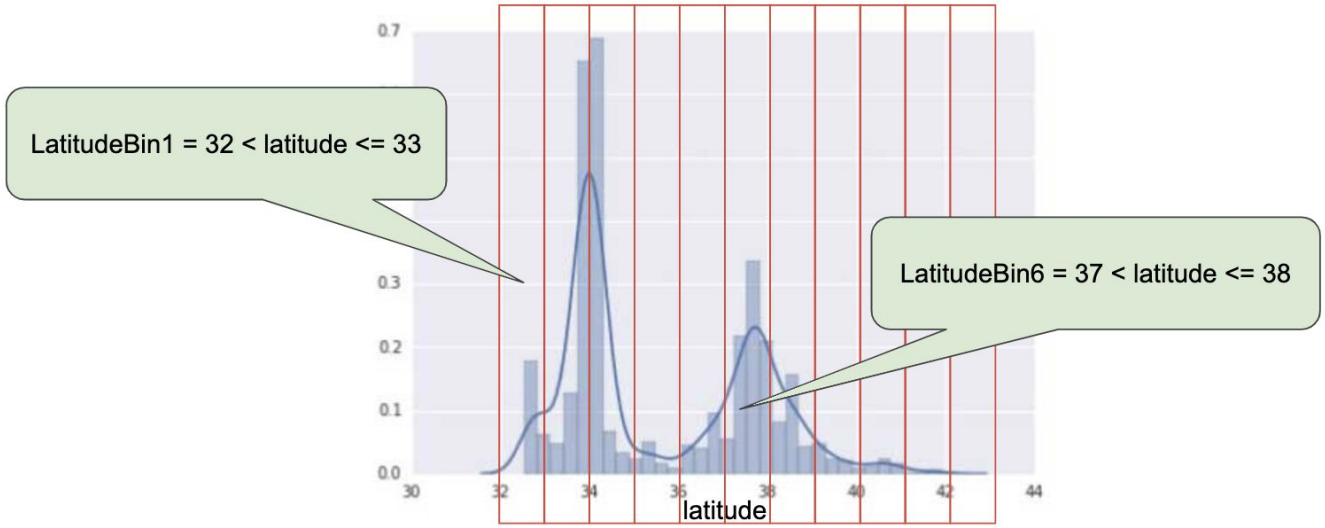
Data transformation - Normalisation

Clipping feature values at 4.0



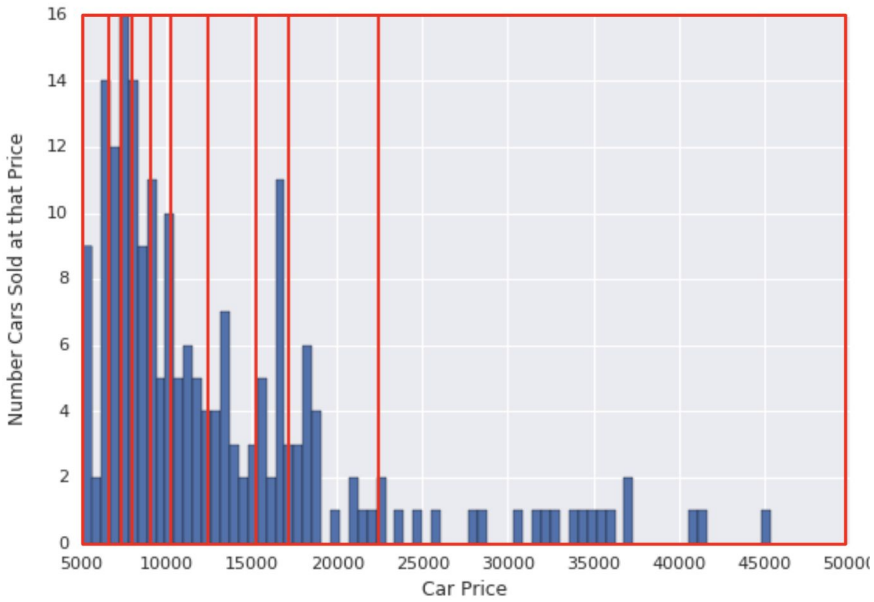

```
roomsPerPerson = min(totalRooms / population, 4)
```

Data transformation - Binning/ Bucketing



Buckets with equal bins

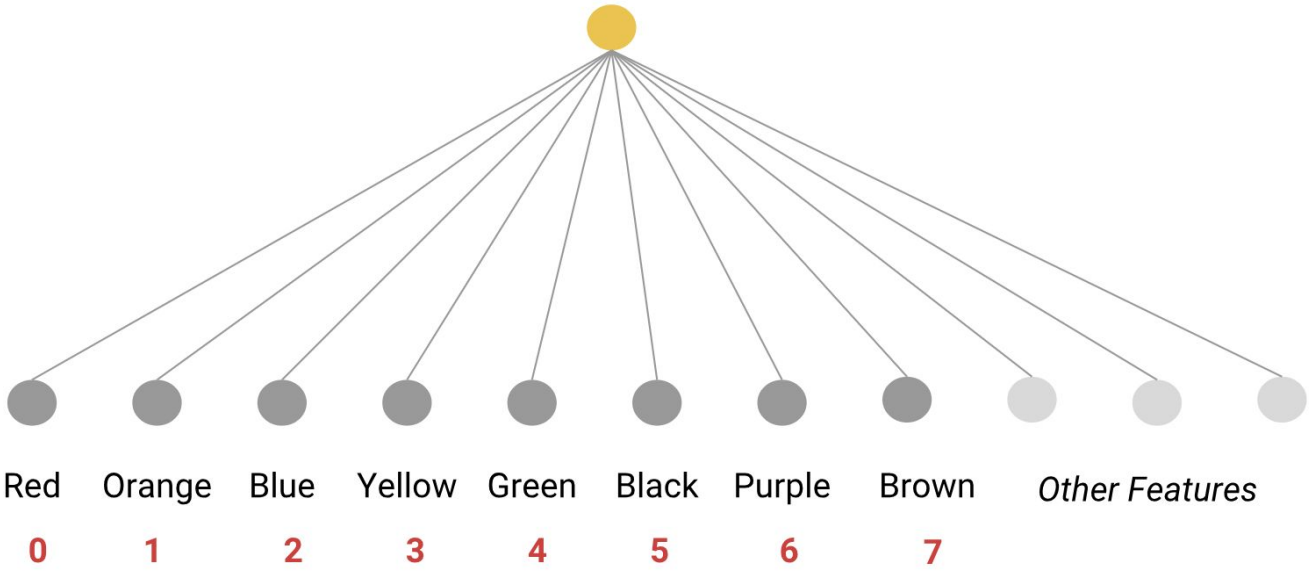
Buckets with quantile bins



Data transformation summary

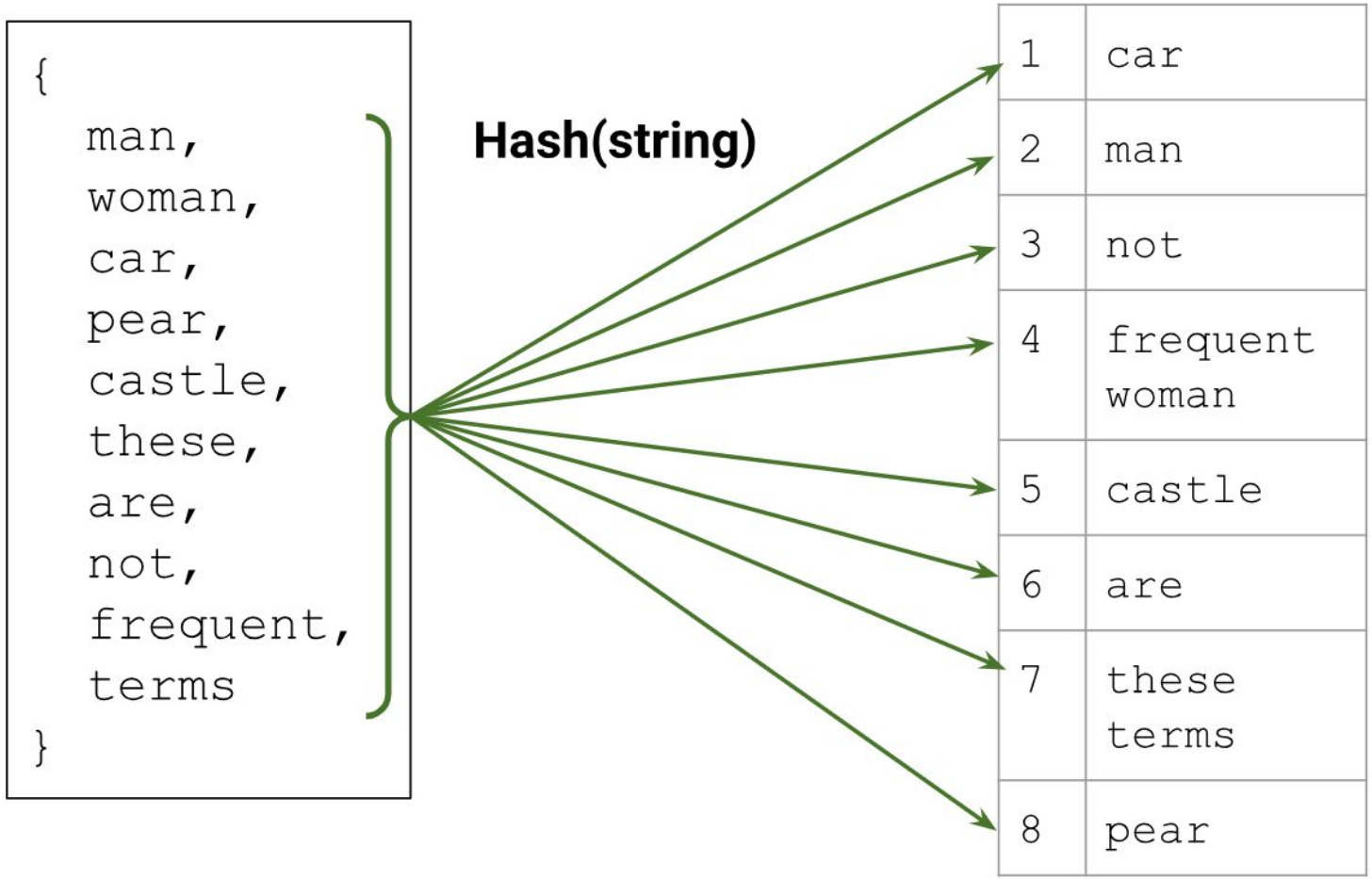
Normalization Technique	Formula	When to Use
Linear Scaling	$x' = (x - x_{min})/(x_{max} - x_{min})$	When the feature is more-or-less uniformly distributed across a fixed range.
Clipping	if x > max, then x' = max. if x < min, then x' = min	When the feature contains some extreme outliers.
Log Scaling	x' = log(x)	When the feature conforms to the power law.
Z-score	x' = (x - μ) / σ	When the feature distribution does not contain extreme outliers.

Categorical Data transformation



This sort of mapping is called a **vocabulary**.

Categorical Data transformation



Hashing

Reasons for Data transformation

Converting non-numeric features into numeric. You can't do matrix multiplication on a string

Resizing inputs to a fixed size. Linear models and feed-forward neural networks have a fixed number of input nodes, so your input data must always have the same size. For example, image models need to reshape the images in their dataset to a fixed size.
Optional quality transformations that may help the model perform better.

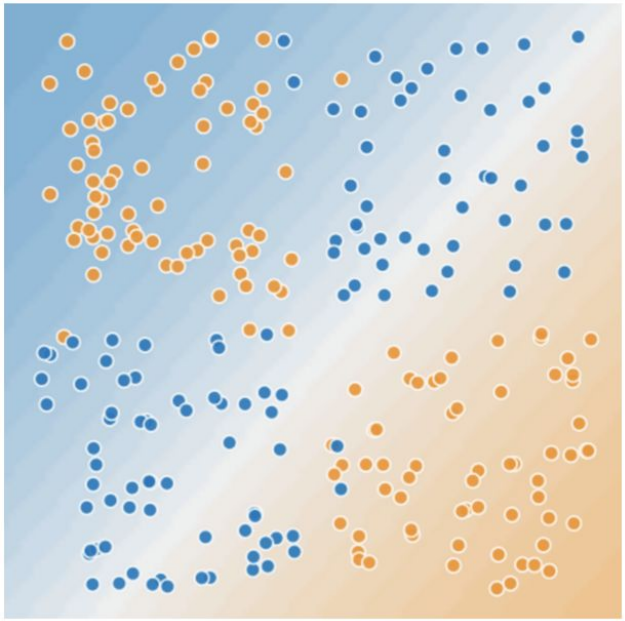
Tokenization or lower-casing of text features.

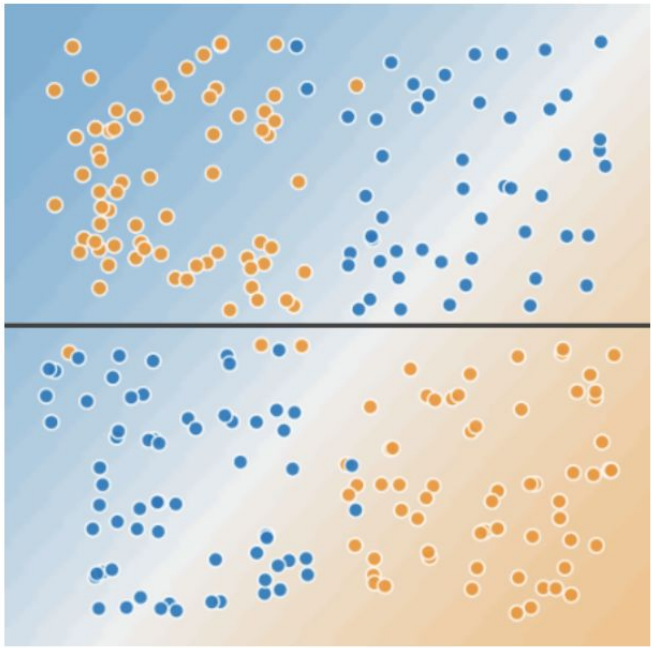
Normalized numeric features (most models perform better afterwards).

Allowing linear models to introduce non-linearities into the feature space.

Strictly speaking, quality transformations are not necessary--your model could still run without them. But using these techniques may enable the model to give better results.

Features crosses encodes nonlinearity





A feature cross is a synthetic feature that encodes nonlinearity in the feature space by multiplying two or more input features together. (The term cross comes from cross product.)

Let's create a feature cross named x_3 by crossing x_1 and : x_2

We treat this newly minted feature cross just like any other feature. The linear formula becomes:

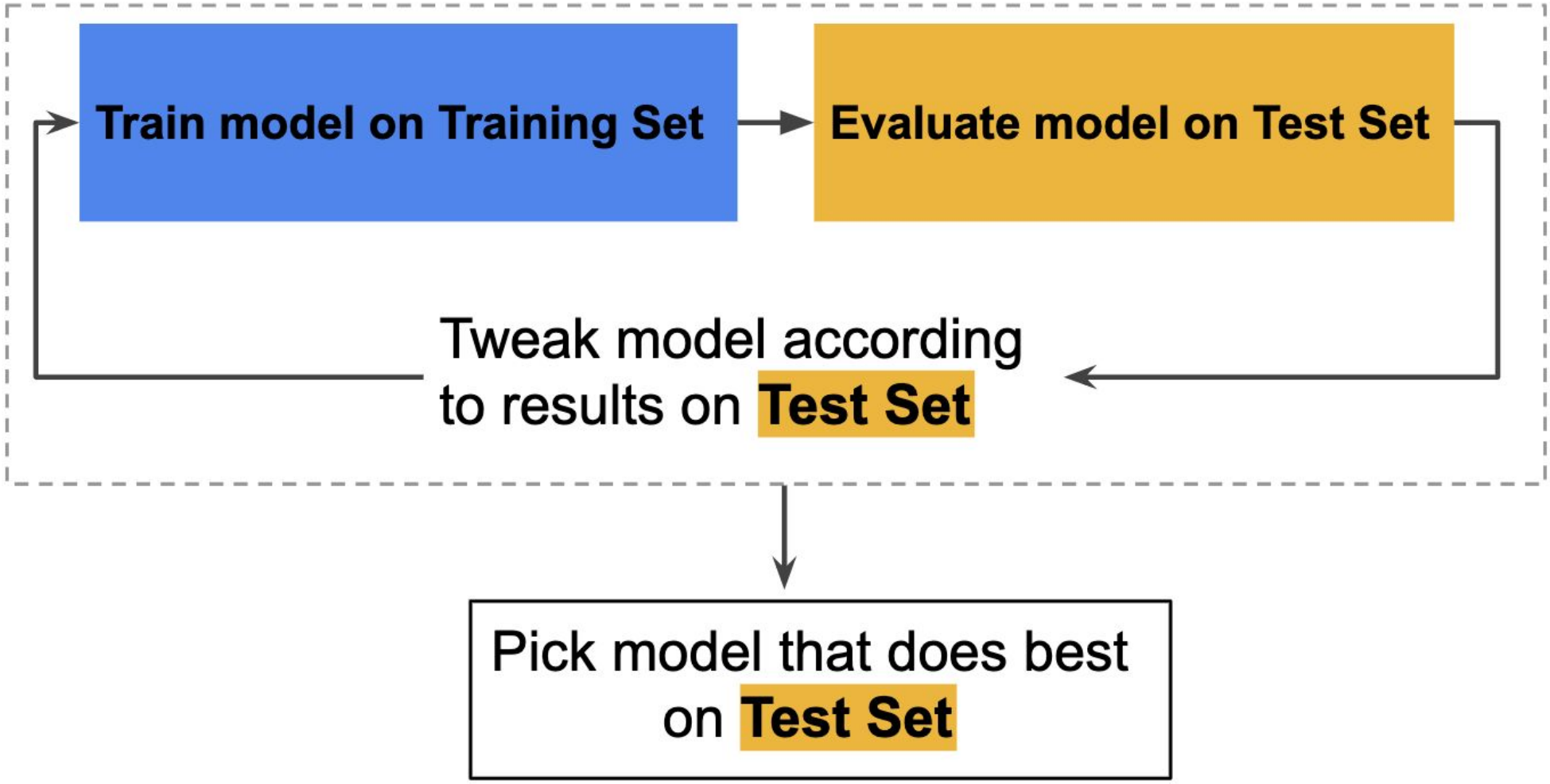
$$y = b + w_1x_1 + w_2x_2 + w_3x_3$$

A linear algorithm can learn a weight for just as it would for and . In other words, although encodes nonlinear information, you don't need to change how the linear model trains to determine the value of .

Partitioning dataset - part I

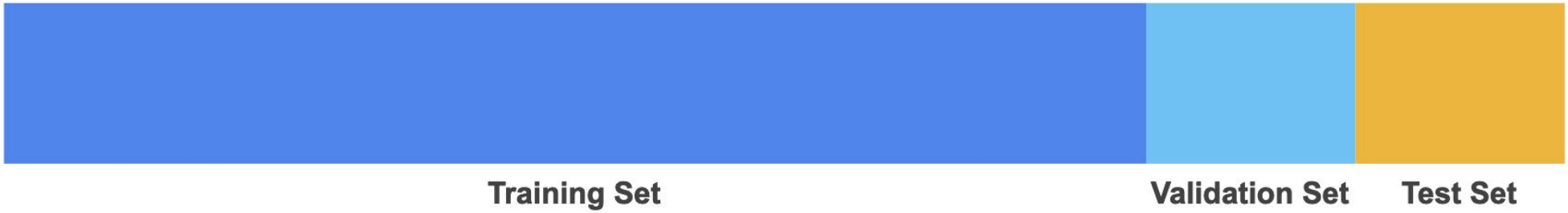
One large data set

Train-test workflow

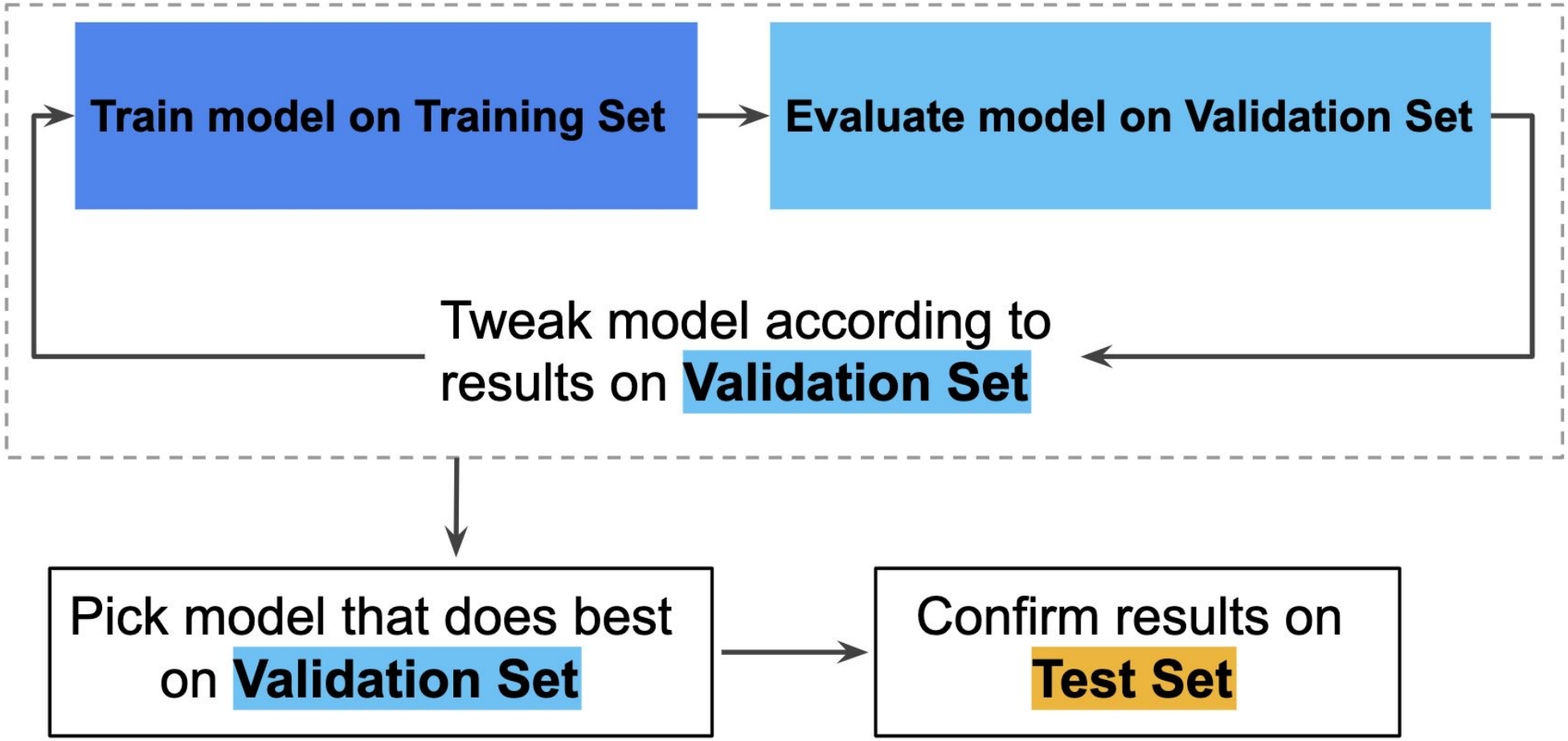


Problem: Overfitting on test set

Partitioning dataset - part II



Train-test workflow



Pick the model that does best on the validation set.
Double-check that model against the test set.

Three basic assumptions of the test/validation dataset

1. We draw examples **independently and identically (i.i.d.)** at random from the distribution

2. The distribution is **stationary**: It doesn't change over time

3. We always pull from the **same distribution**: Including training, validation, and test sets

Introduction to sampling

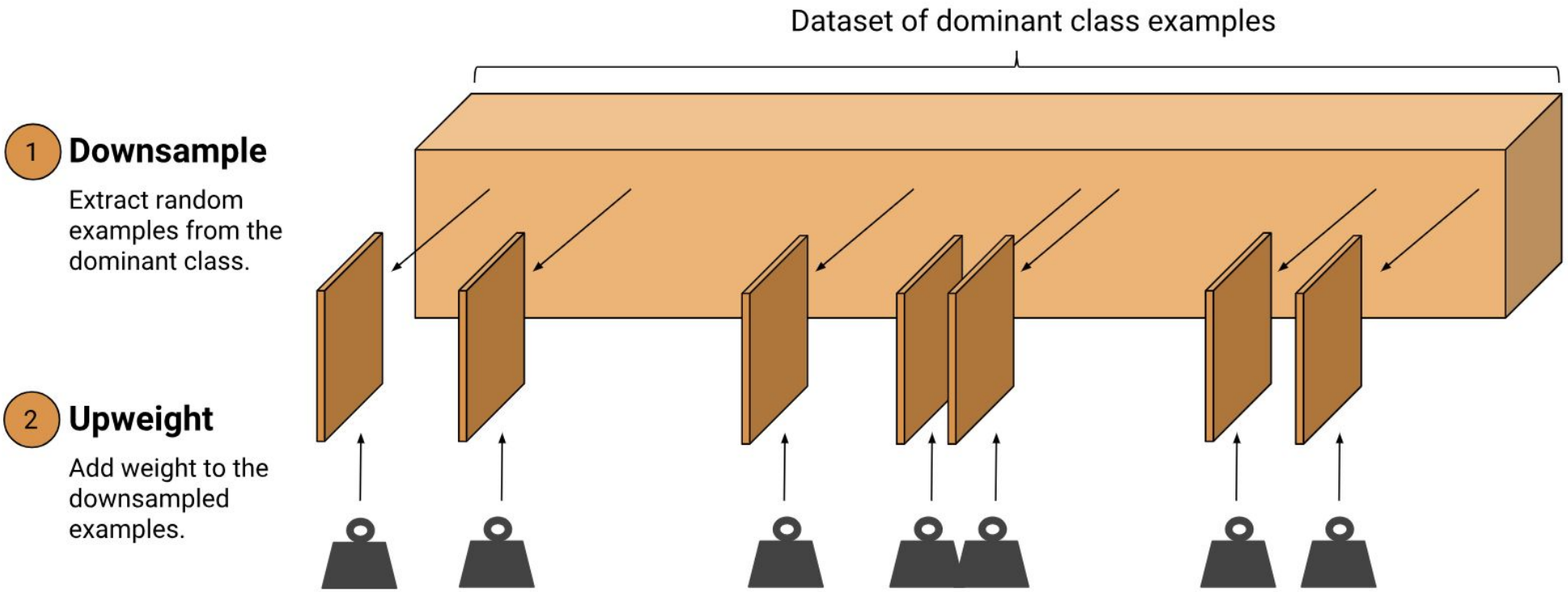
It's often a struggle to gather enough data for a machine learning project. Sometimes, however, there is too much data, and you must select a subset of examples for training.

Imbalanced data
A classification data set with skewed class proportions is called imbalanced. Classes that make up a large proportion of the data set are called majority classes. Those that make up a smaller proportion are minority classes.

Downsampling and Upweighting
An effective way to handle imbalanced data is to downsample and upweight the majority class. Let's start by defining those two new terms:

Downsampling (in this context) means training on a disproportionately low subset of the majority class examples.

Upweighting means adding an example weight to the downsampled class equal to the factor by which you downsampled.



Why Downsample and Upweight?
It may seem odd to add example weights after downsampling. We were trying to make our model improve on the minority class -- why would we upweight the majority? These are the resulting changes:

Faster convergence: During training, we see the minority class more often, which will help the model converge faster.
Disk space: By consolidating the majority class into fewer examples with larger weights, we spend less disk space storing them. This savings allows more disk space for the minority class, so we can collect a greater number and a wider range of examples from that class.
Calibration: Upweighting ensures our model is still calibrated; the outputs can still be interpreted as probabilities.

Issues with random splitting

When Random Splitting isn't the Best Approach
While random splitting is the best approach for many ML problems, it isn't always the right solution. For example, consider data sets in which the examples are naturally clustered into similar examples.

Suppose you want your model to classify the topic from the text of a news article. Why would a random split be problematic?

Practical considerations for random splitting

Make your data generation pipeline reproducible. Say you want to add a feature to see how it affects model quality. For a fair experiment, your datasets should be identical except for this new feature. If your data generation runs are not reproducible, you can't make these datasets.

In that spirit, make sure any randomization in data generation can be made deterministic:

Seed your random number generators (RNGs). Seeding ensures that the RNG outputs the same values in the same order each time you run it, recreating your dataset.

Use **invariant hash keys**. Hashing is a common way to split or sample data. You can hash each example, and use the resulting integer to decide in which split to place the example. The inputs to your hash function shouldn't change each time you run the data generation program. Don't use the current time or a random number in your hash, for example, if you want to recreate your hashes on demand.

The preceding approaches apply both to sampling and splitting your data.

Resampling methods

Resampling methods are an indispensable tool in modern statistics. They involve repeatedly drawing samples from a training set and refitting a model of interest on each sample in order to obtain additional information about the fitted model. For example, in order to estimate the variability of a linear regression fit, we can repeatedly draw different samples from the training data, fit a linear regression to each new sample, and then examine the extent to which the resulting fits differ. Such an approach may allow us to obtain information that would not be available from fitting the model only once using the original training sample.

Cross validation & Bootstrapping

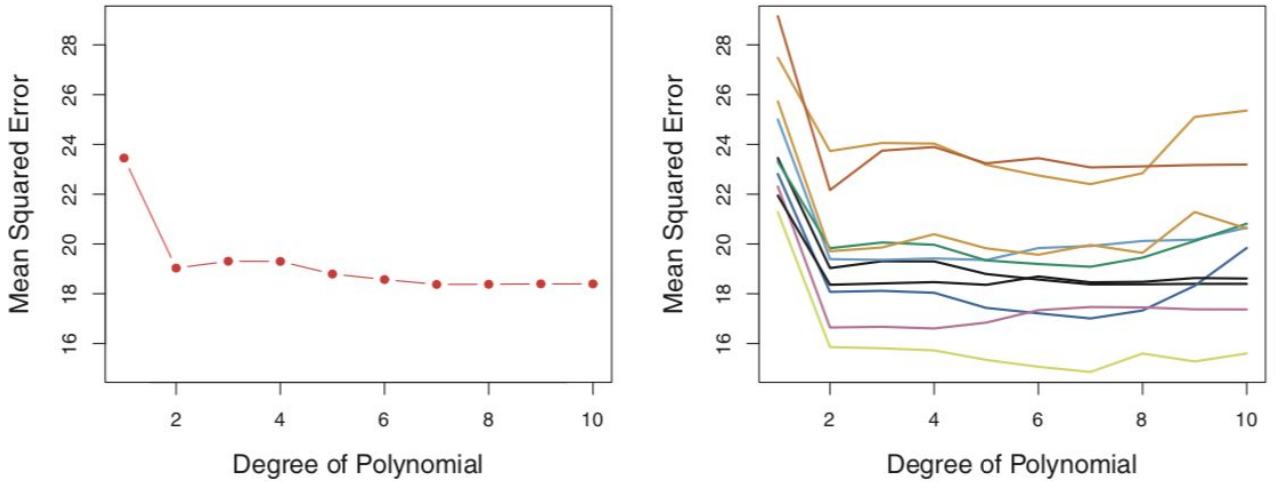
Both methods are important tools in the practical application of many statistical learning procedures.

For example, cross-validation can be used to estimate the test error associated with a given statistical learning method in order to evaluate its performance, or to select the appropriate level of flexibility.

The process of evaluating a model's performance is known as **model assessment**, whereas the process of selecting the proper level of flexibility for a model is known as **model selection**.

The bootstrap is used in several contexts, most commonly to provide a measure of accuracy of a parameter estimate or of a given statistical learning method.

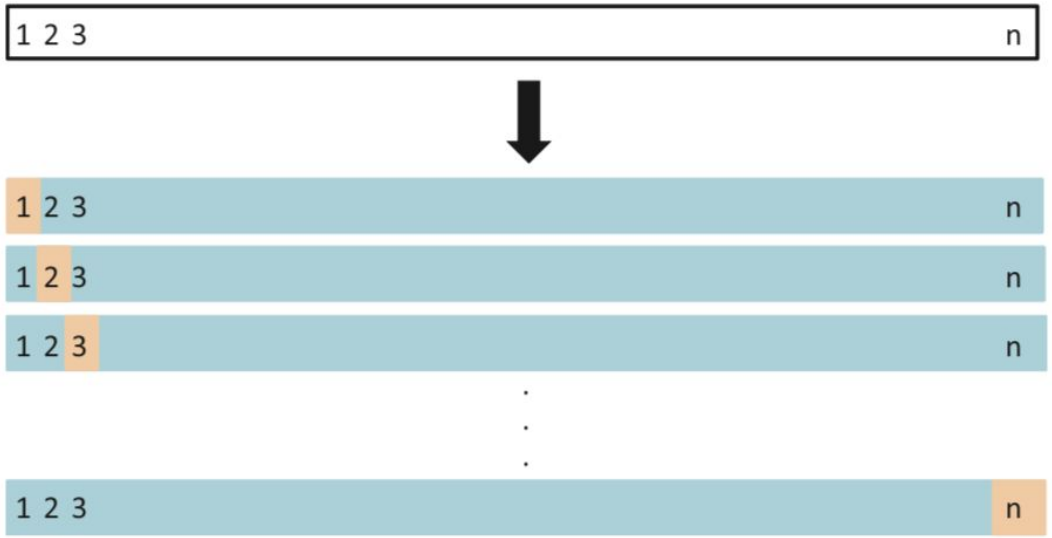
Validation set approach



the training error rate often is quite different from the test error rate, and in particular the former can dramatically underestimate the latter.

Leave-one-cross-validation

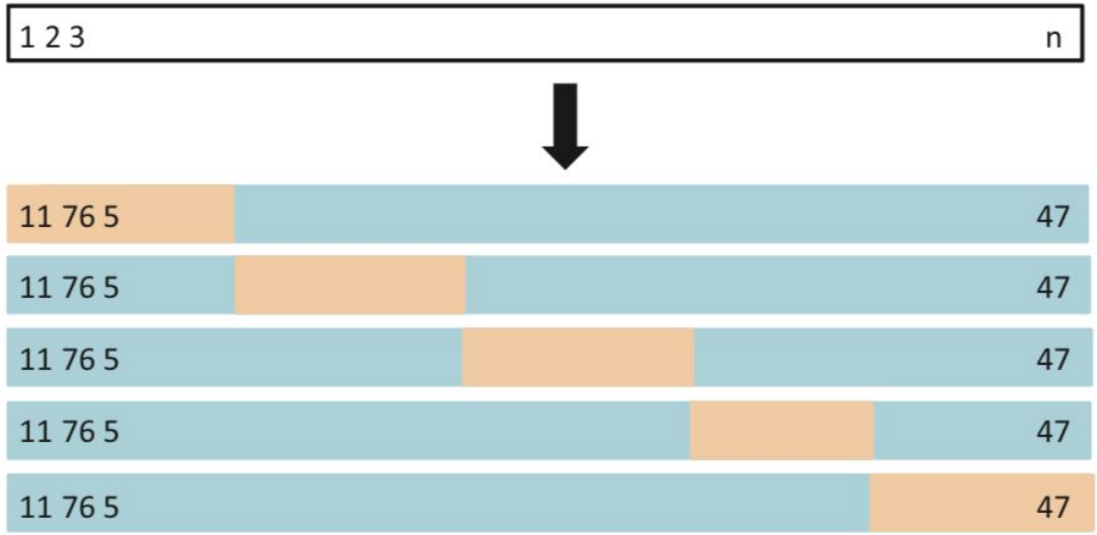
Leave-one-out cross-validation (LOOCV) is closely related to the validation leave-one- set approach, but it attempts to address that method's out drawbacks.



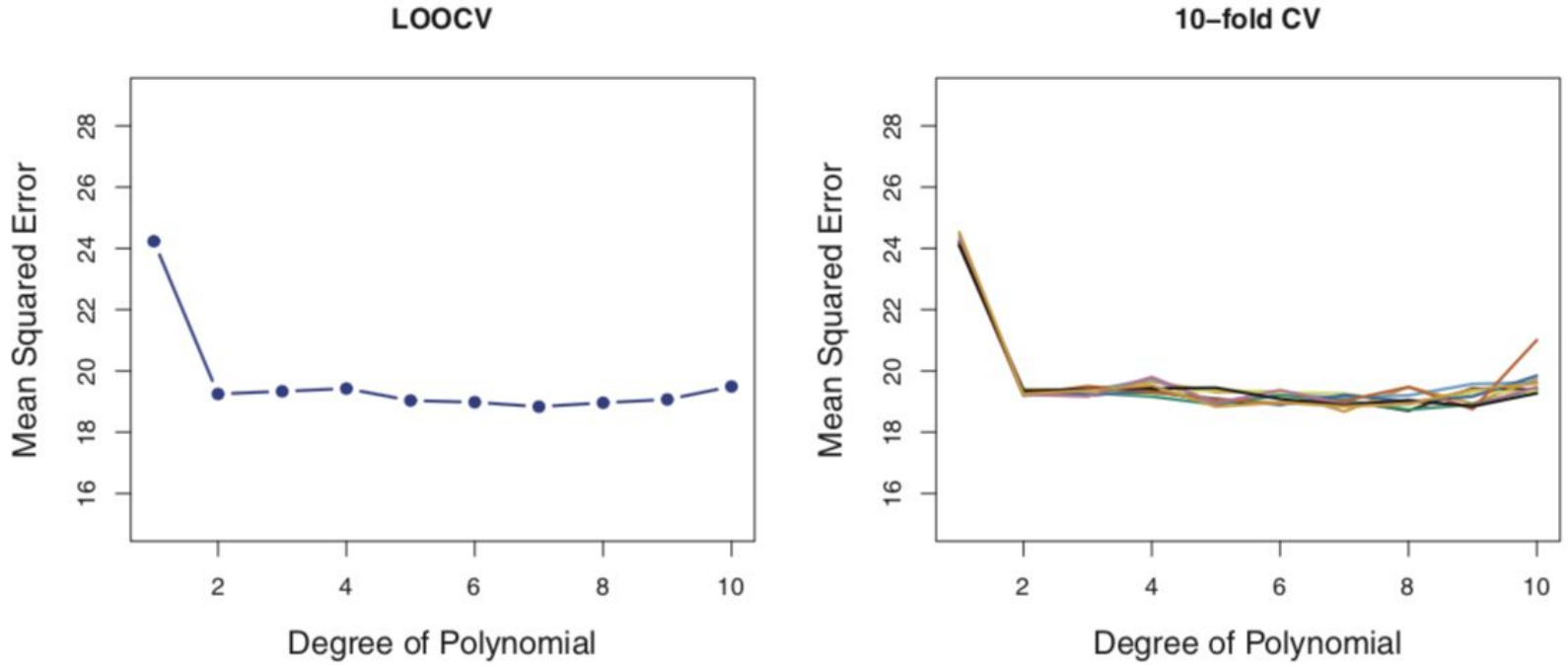
Like the validation set approach, LOOCV involves splitting the set of

observations into two parts. However, instead of creating two subsets of comparable size, a single observation (x_1,y_1) is used for the validation set, and the remaining observations $\{(x_2, y_2), \ldots, (x_n, y_n)\}$ make up the training set. The statistical learning method is fit on the n − 1 training observations, and a prediction $\hat{y}_1$ is made for the excluded observation, using its value x_1. Since (x_1, y_1) was not used in the fitting process, $MSE_1 = (y_1 - \hat{y}_1)_2$ provides an approximately unbiased estimate for the test error. But even though MSE_1 is unbiased for the test error, it is a poor estimate because it is highly variable, since it is based upon a single observation (x_1,y_1).

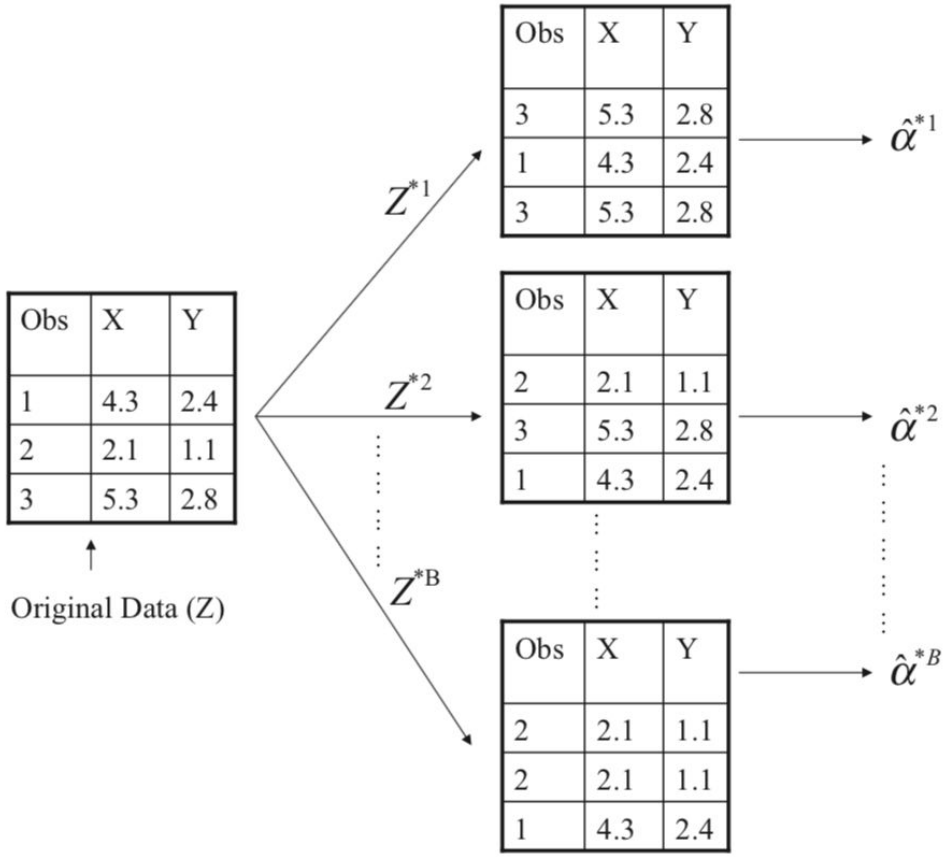
K-fold Cross validation



Bias Variance trade off



Bootstrapping



The bootstrap is a widely applicable and extremely powerful statistical tool that can be used to quantify the uncertainty associated with a given estimator or statistical learning method.

How to evaluate your ML algorithm

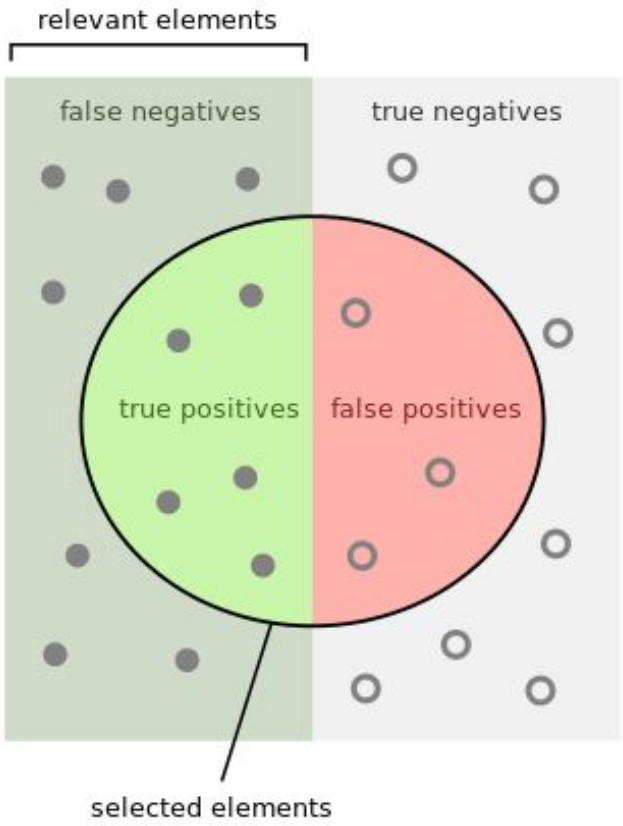
A single number metric is the easiest one!

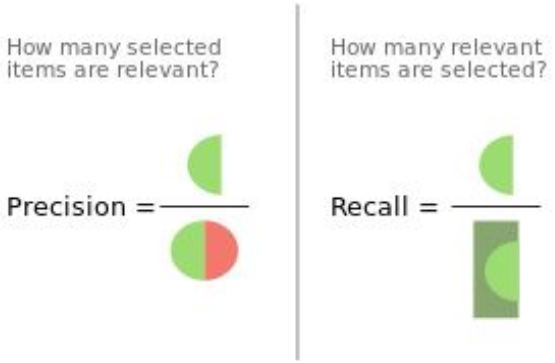
For example Accuracy

If a classifier A gives 97% accuracy and classifier B gives 95% accuracy, then we can conclude that A is better

Often Precision Recalls are used

Precision - Recall





precision is "how useful the search results are", and recall is "how complete the results are".

Classifier A 95% Precision and 90% Recall
Classifier B 98% Precision and 85% Recall

Average or an harmonic average - F1 score - of the precision and recall, where an F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0.

A single number is the best way to evaluate a ML algorithm

Source: Wikipedia

Other metrics

Run time

Size of the application

False positives - False negatives

Error Analysis

Error Analysis examining dev set examples that your algorithm misclassified, so that you can understand the underlying causes of the errors. This can help you prioritize projects.

Write down all the misclassified items and analyse them to efficiently evaluate and fix problems

If the Dev set is large, split it into two and evaluate one manually

Bias and Variance - I

There are two major sources of error in machine learning: bias and variance.

The algorithm's error rate on the training set - **bias**

The algorithm does on the dev (or test) set than the training set - **variance**

- Training error = 1% (bias)
- Dev error = 11% (variance=10%)

Overfitting

- Training error = 15% (bias)
- Dev error = 30% (variance = 15%)

High Bias and High Variance

- Training error = 15% (bias)
- Dev error = 16% (viance =1%)

Underfitting

- Training error = 0.5% (bias)
- Deverror = 1% (variance)

Perfect

Optimal error rate ("unavoidable bias")

Bias and Variance - II

To reduce bias

1. Increase the model size
2. Modifying features based on error analysis
3. Reduce or eliminate regularization
4. Modify model architecture

To reduce variance

1. Add more training data
2. Add regularization
3. Feature selection to decrease the number of inputs or features
4. Decrease the model size
5. Modify input features based on error analysis