SoC Project Report

Introduction to Machine Learning

Checkpoint - 1

MOODIFY

Bhavesh Khichi

Mentor : Abhishek Pai Angle, Divyanshi Kamra

April 11, 2021

Chapter 1: **ML Concepts**

The project report is a very important part of your project and its preparation and presentation should be of extremely high quality. Remember that a significant portion of the marks for your project are awarded for this report. **The electronic submission of your report must be in PDF format. You can use the menu option *File->Save As* to generate it.**

This document is a style guide for final year project reports in the Department of Computer Science. As such, it constitutes a collection of predefined Microsoft Word formatting styles for the production of your final report.

While this may sound like a rather prescriptive approach to report writing, it is introduced for the following reasons.

1. The style guide allows students to focus on the critical task of producing clear and concise content, instead of being distracted by font settings and paragraph spacing.

2. By providing a comprehensive style guide the Department benefits from a consistent and professional look to its internal project reports.

The remainder of this document briefly outlines the main components and their usage.

A **final project report** is approximately 15,000 words and must include a word count. It is acceptable to have other material in appendixes. Your **interim report** for the December Review meeting, even if it is a collection of reports, should have a total word count of about 5,000 words. This should summarise the work you have done so far, with sections on the theory you have learnt and the code that you have written.

Also remember that any details of report content and submission rules, as well as other deliverables, are defined in the project booklet [1].

1.1 Framing: Key ML Terminology

a. Labels :- A **label** is the thing we're predicting—the y variable in simple linear regression. The label could be the future price of wheat, the kind of animal shown in a picture, the meaning of an audio clip, or just about anything.
b. Features :- A **feature** is an input variable—the x variable in simple linear regression. A simple machine learning project might use a single feature, while a more sophisticated machine learning project could use millions of features, specified as:

$$x_1, x_2, \ldots x_N$$

c. Models :- A model defines the relationship between features and label. For example, a spam detection model might associate certain features strongly with "spam". Let's highlight two phases of a model's life:
    - **Training** means creating or **learning** the model. That is, you show the model labeled examples and enable the model to gradually learn the relationships between features and labels.

- **Inference** means applying the trained model to unlabeled examples. That is, you use the trained model to make useful predictions (`y'`). For example, during inference, you can predict `medianHouseValue` for new unlabeled examples.

d. Regression :- A **regression** model predicts continuous values. For example, regression models make predictions that answer questions like the following:
   - What is the value of a house in California?
   - What is the probability that a user will click on this ad?

e. Classification :- A **classification** model predicts discrete values. For example, classification models make predictions that answer questions like the following:
   - Is a given email message spam or not spam?
   - Is this an image of a dog, a cat, or a hamster?

1.2 Descending into ML

1.2.1 Linear Regression

Linear regression is a linear model, e.g. a model that assumes a linear relationship between the input variables (x) and the single output variable (y). More specifically, that y can be calculated from a linear combination of the input variables (x). When there is a single input variable (x), the method is referred to as simple linear regression. When there are multiple input variables, literature from statistics often refers to the method as multiple linear regression.

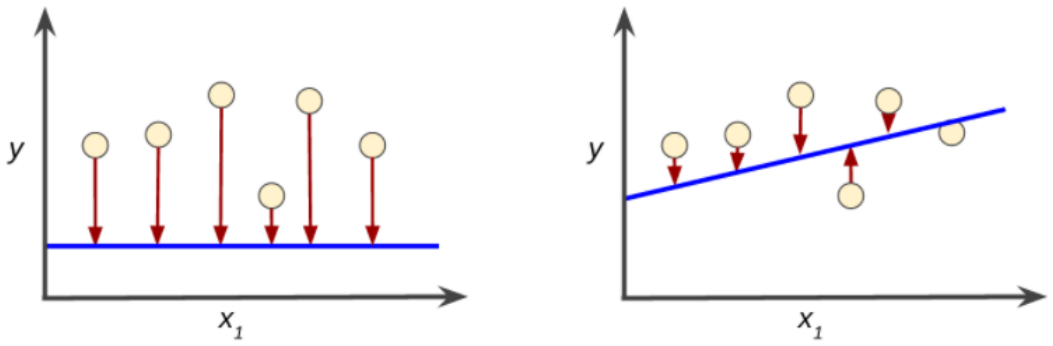
Different techniques can be used to prepare or train the linear regression equation from data, the most common of which is called Ordinary Least Squares. It is common to therefore refer to a model prepared this way as Ordinary Least Squares Linear Regression or just Least Squares Regression.

1.2.2 Training and loss

Training a model simply means learning (determining) good values for all the weights and the bias from labeled examples. In supervised learning, a machine learning algorithm builds a model by examining many examples and attempting to find a model that minimizes loss; this process is called empirical risk minimization.

Loss is the penalty for a bad prediction. That is, loss is a number indicating how bad the model's prediction was on a single example. If the model's prediction is perfect, the loss is zero; otherwise, the loss is greater. The goal of training a model is to find a set of weights and biases that have *low* loss, on average, across all examples. For example, Figure 3 shows a high loss model on the left and a low loss model on the right. Note the following about the figure:

- The arrows represent loss.
- The blue lines represent predictions.



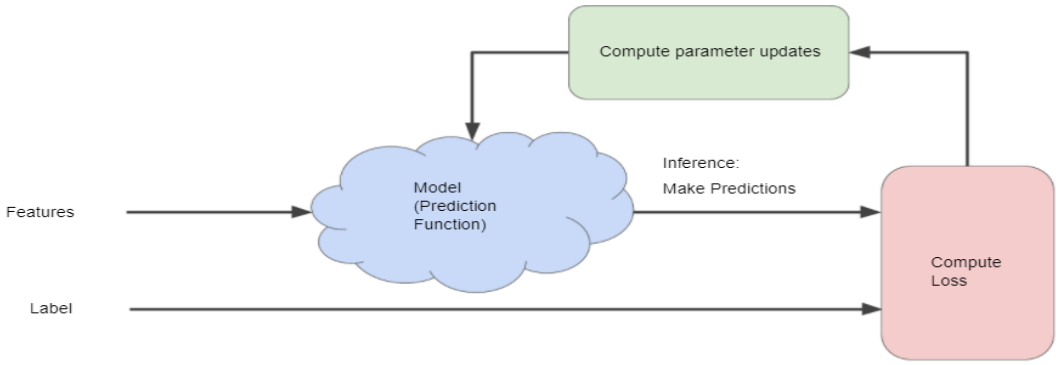
Mean square error (**MSE**) is the average squared loss per example over the whole dataset. To calculate MSE, sum up all the squared losses for individual examples and then divide by the number of examples:

$$MSE = \frac{1}{N} \sum_{(x,y) \in D} (y - prediction(x))^2$$

1.3 Reducing Loss

1.3.1 Interactive Approach

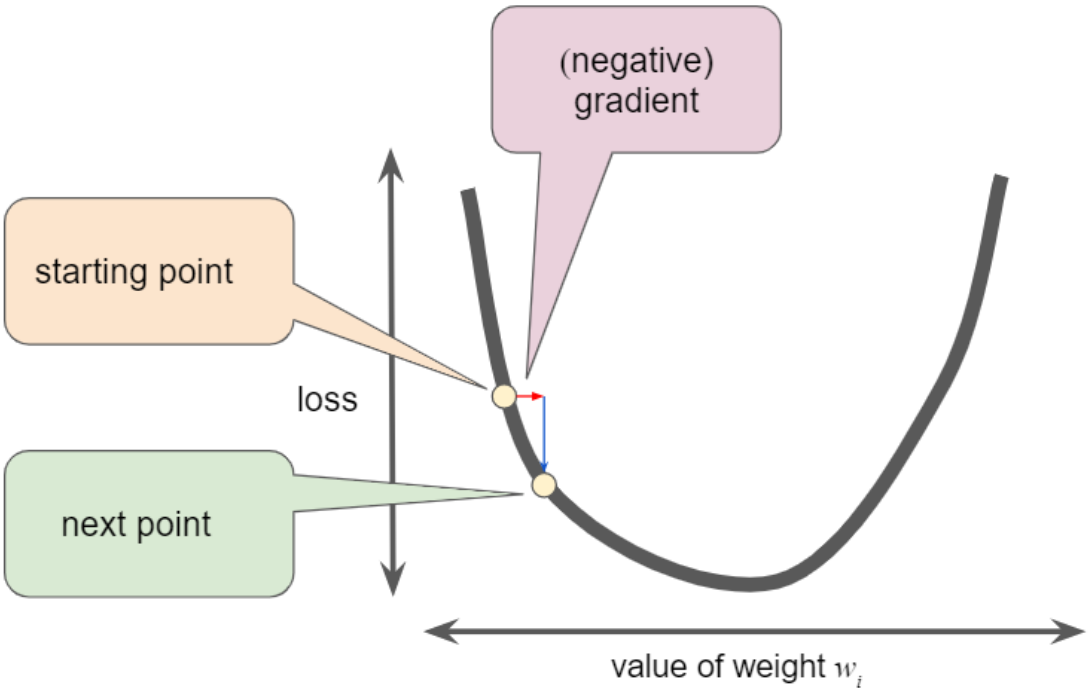
The figure suggests the iterative trial-and-error process that machine learning algorithms use to train a model, The "model" takes one or more features as input and returns one prediction (y') as output.



1.3.2 Gradient Descent

The gradient descent algorithm then calculates the gradient of the loss curve at the starting point. The gradient of the loss is equal to the derivative (slope) of the curve, and tells you which way is "warmer" or "colder." When there are multiple weights, the **gradient** is a vector of partial derivatives with respect to the weights.

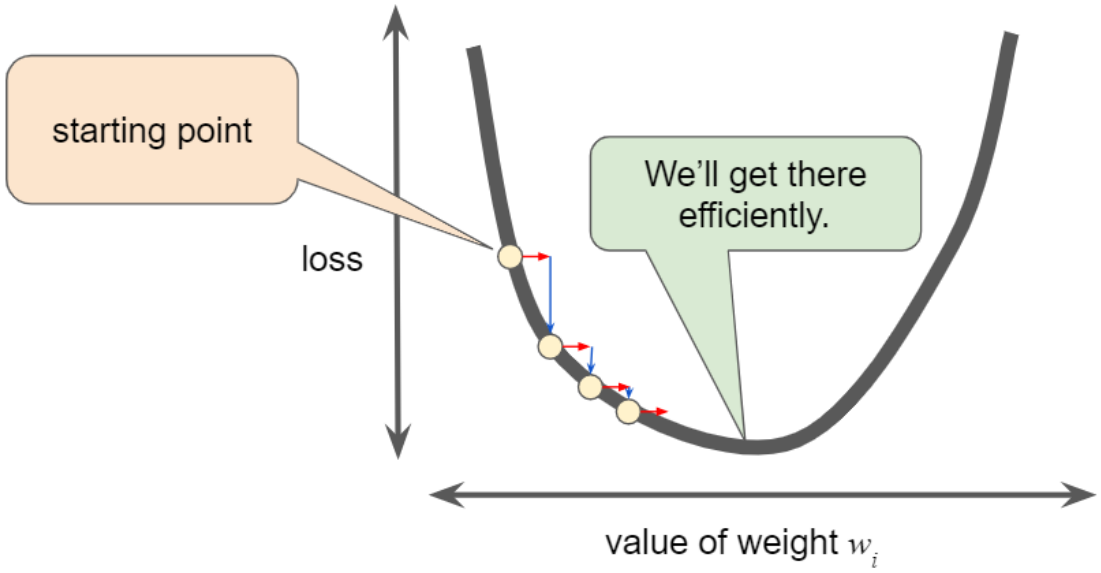
To determine the next point along the loss function curve, the gradient descent algorithm adds some fraction of the gradient's magnitude to the starting point as shown



1.3.3 Learning Rate

Hyperparameters are the knobs that programmers tweak in machine learning algorithms. Most machine learning programmers spend a fair amount of time tuning the learning rate.

There's a Goldilocks learning rate for every regression problem. The Goldilocks value is related to how flat the loss function is. If you know the gradient of the loss function is small then you can safely try a larger learning rate, which compensates for the small gradient and results in a larger step size.



1.3.4 Stochastic Gradient Descent

A large data set with randomly sampled examples probably contains redundant data. In fact, redundancy becomes more likely as the batch size grows. Some redundancy can be useful to smooth out noisy gradients, but enormous batches tend not to carry much more predictive value than large batches.

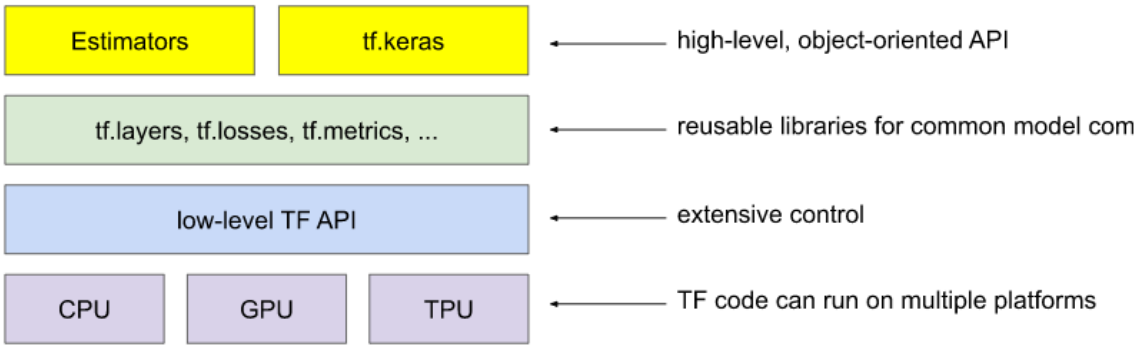
we could estimate (albeit, noisily) a big average from a much smaller one. **SGD** takes this idea to the extreme--it uses only a single example (a batch size of 1) per iteration. Given enough iterations, SGD works but is very noisy. The term "stochastic" indicates that the one example comprising each batch is chosen at random.

1.4 Introduction to TensorFlow

TensorFlow is an end-to-end open source platform for machine learning. TensorFlow is a rich system for managing all aspects of a machine learning

system; however, this class focuses on using a particular TensorFlow API to develop and train machine learning models. See the TensorFlow documentation for complete details on the broader TensorFlow system.

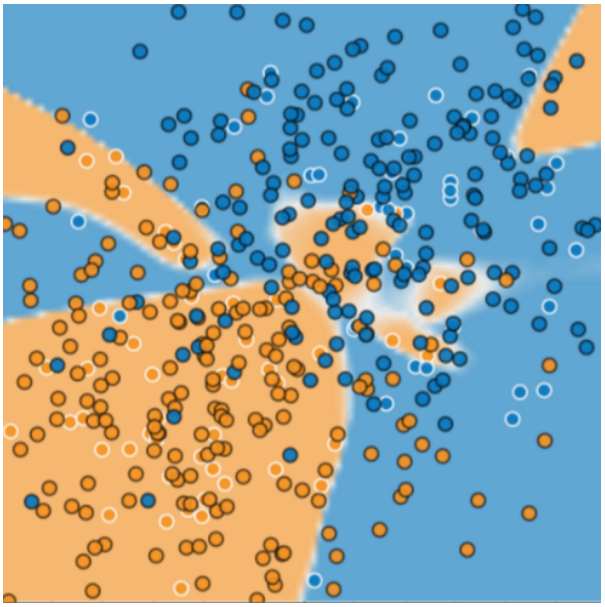
TensorFlow APIs are arranged hierarchically, with the high-level APIs built on the low-level APIs. Machine learning researchers use the low-level APIs to create and explore new machine learning algorithms. In this class, you will use a high-level API named tf.keras to define and train machine learning models and to make predictions. tf.keras is the TensorFlow variant of the open-source Keras API.



1.5 Generalization

1.5.1 Peril of Overfitting

Low loss, models are not always good, The model shown in Figures **overfits** the peculiarities of the data it trained on. An overfit model gets a low loss during training but does a poor job predicting new data. If a model fits the current sample well.



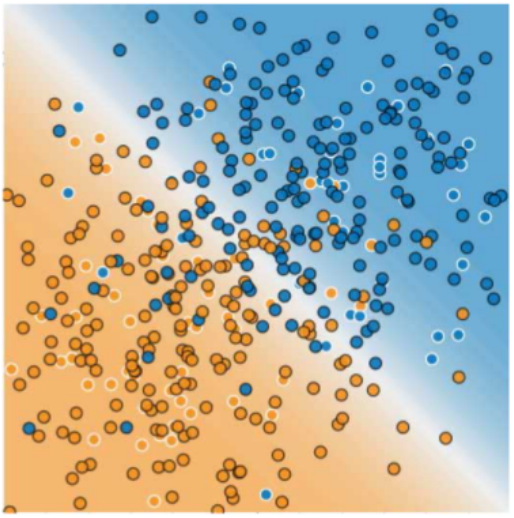
Machine learning's goal is to predict well on new data drawn from a (hidden) true probability distribution. Unfortunately, the model can't see the whole truth; the model can only sample from a training data set. If a model fits the current examples well

William of Ockham, a 14th century friar and philosopher, loved simplicity. He believed that scientists should prefer simpler formulas or theories over more complex ones. To put Ockham's razor in machine learning terms:

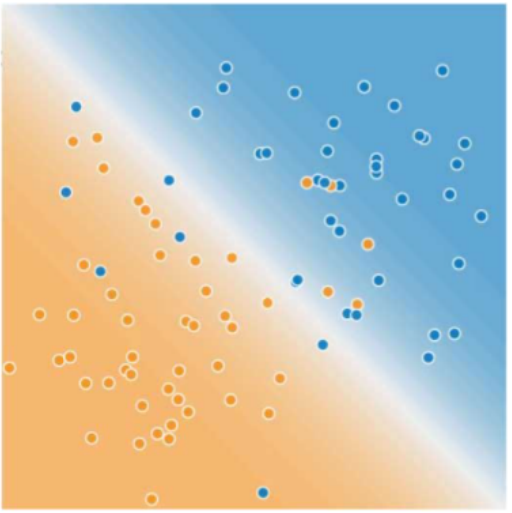
1.6 Training & Test Sets

1.6.1 Splitting Data

- **training set**—a subset to train a model.
- **test set**—a subset to test the trained model.
    a. Is large enough to yield statistically meaningful results.
    b. Is representative of the data set as a whole. In other words, don't pick a test set with different characteristics than the training set.



Training Data



Test Data

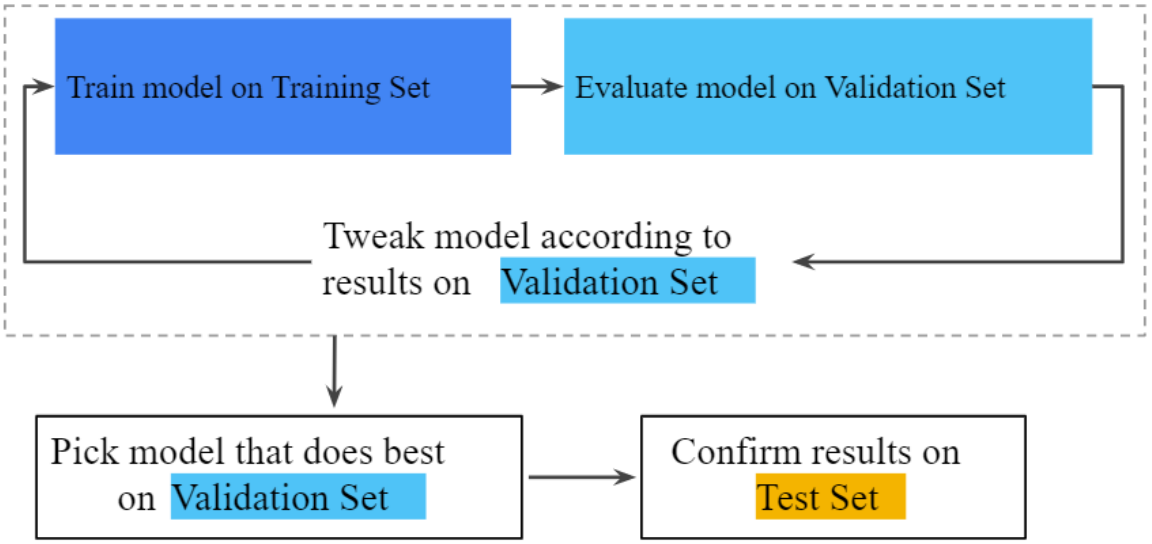
Never train on test data. If you are seeing surprisingly good results on your evaluation metrics, it might be a sign that you are accidentally training on the test set. For example, high accuracy might indicate that test data has leaked into the training set.

1.7 Validation Set

In the figure, "Tweak model" means adjusting anything about the model you can dream up—from changing the learning rate, to adding or removing features, to designing a completely new model from scratch. At the end of this workflow, you pick the model that does best on the *test set*.

Dividing the data set into two sets is a good idea, but not a panacea. You can greatly reduce your chances of overfitting by partitioning the data set into the three subsets shown in the following figure:

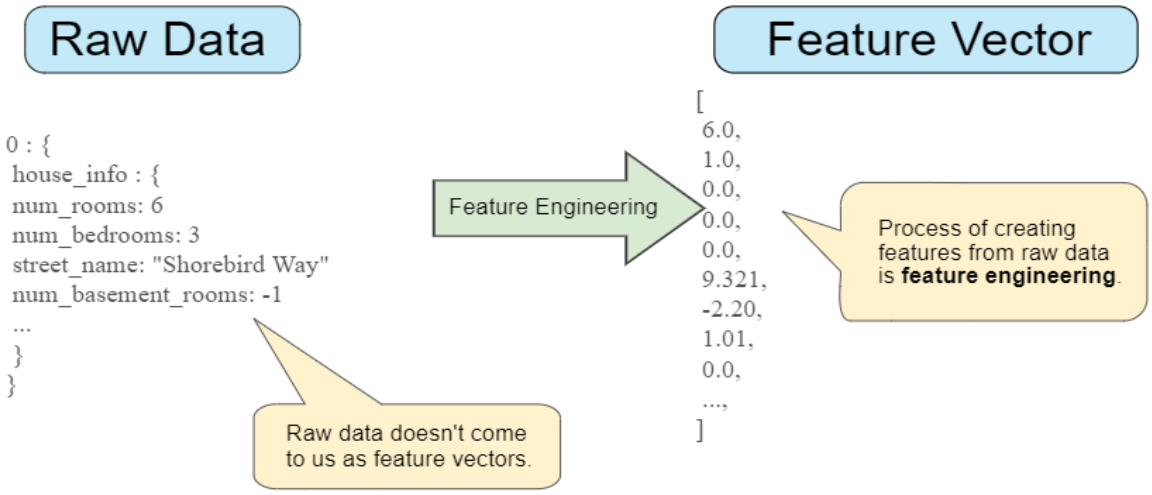
Use the **validation set** to evaluate results from the training set. Then, use the test set to double-check your evaluation *after* the model has "passed" the validation set. The following figure shows this new workflow:



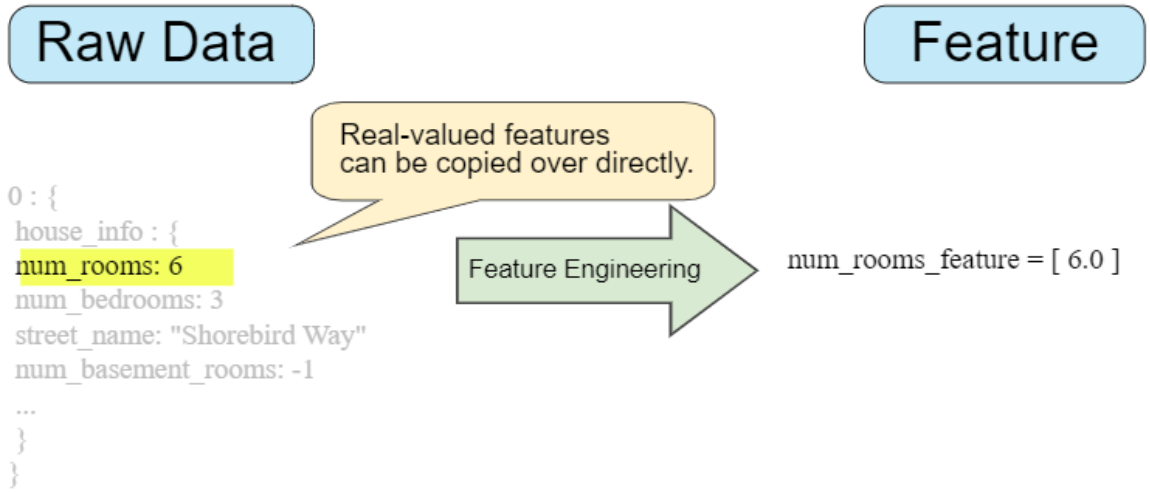
1.8 Representation

1.8.1 Feature Engineering

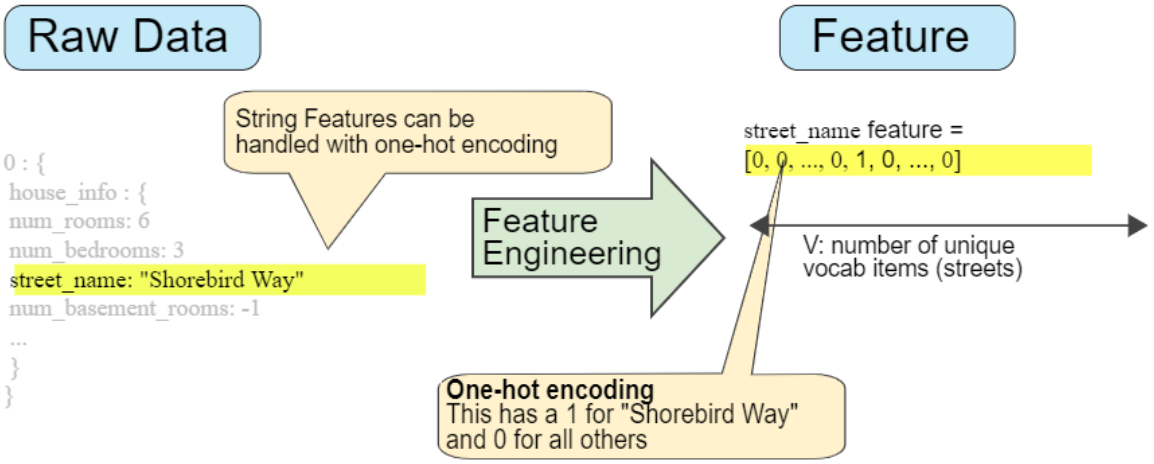
Mapping Raw Data to Features :- The left side of figure illustrates raw data from an input data source; the right side illustrates a **feature vector**, which is the set of floating-point values comprising the examples in your data set. **Feature engineering** means transforming raw data into a feature 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 , converting the raw integer value 6 to the feature value 6.0 is trivial



Mapping categorical value :- 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`.



1.8.2 Qualities of Good Features

We must now explore what kinds of values actually make good features within those feature vectors.

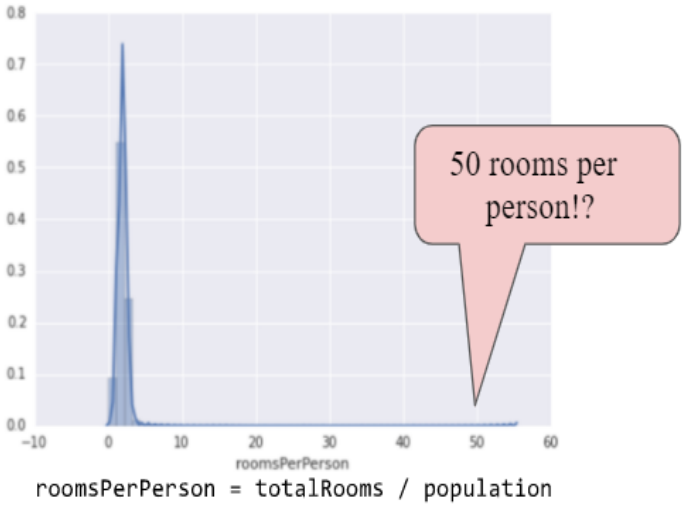
- Avoid rarely used discrete feature value
- Prefer clear and obvious meanings
- Don't mix "magic" values with actual data
- Account for upstream instability

1.8.3 Cleaning Data

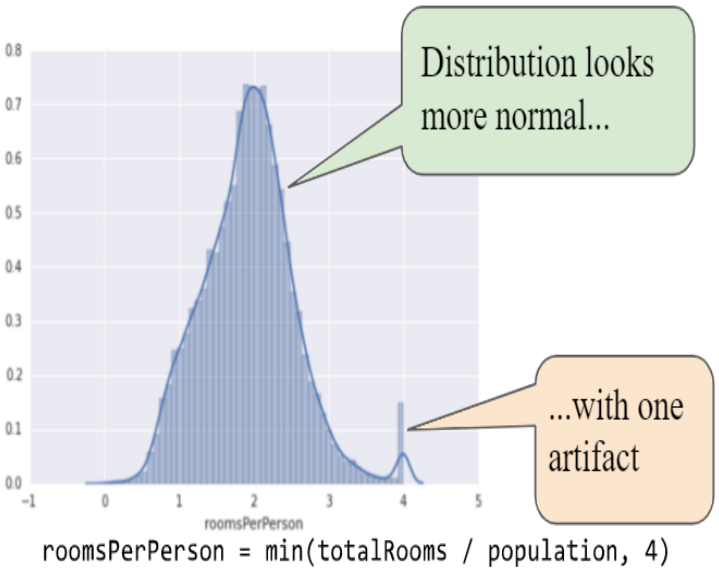
Scaling feature values

- 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.

Handling extreme outliers

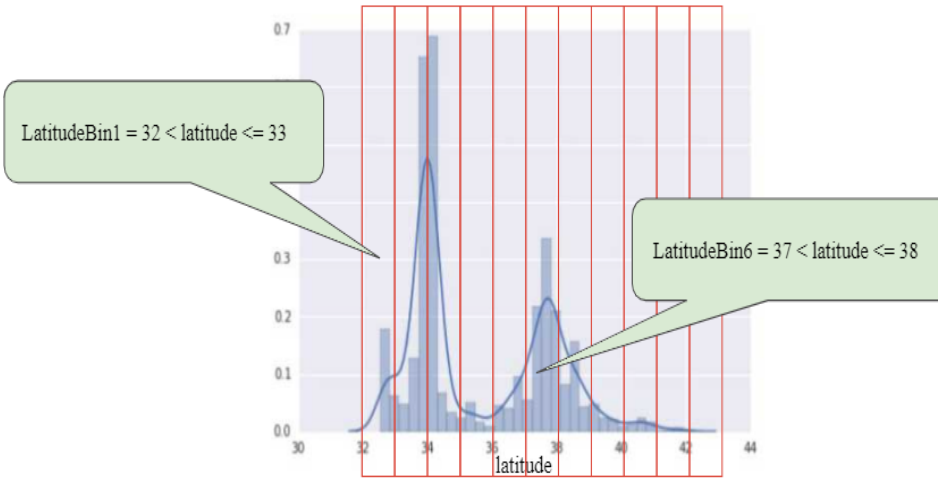


The following plot represents a feature called `roomsPerPerson` from the California Housing data set. The value of `roomsPerPerson` was calculated by dividing the total number of rooms for an area by the population for that area. The plot shows that the vast majority of areas in California have one or two rooms per person. But take a look along the x-axis.



Log scaling does a slightly better job, but there's still a significant tail of outlier values.

Binning



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 35/34 more expensive (or less expensive) than houses at latitude 34. And yet, individual latitudes probably are a pretty good predictor of house values.

Scrubbing

In real-life, many examples in data sets are unreliable due to one or more of the following:

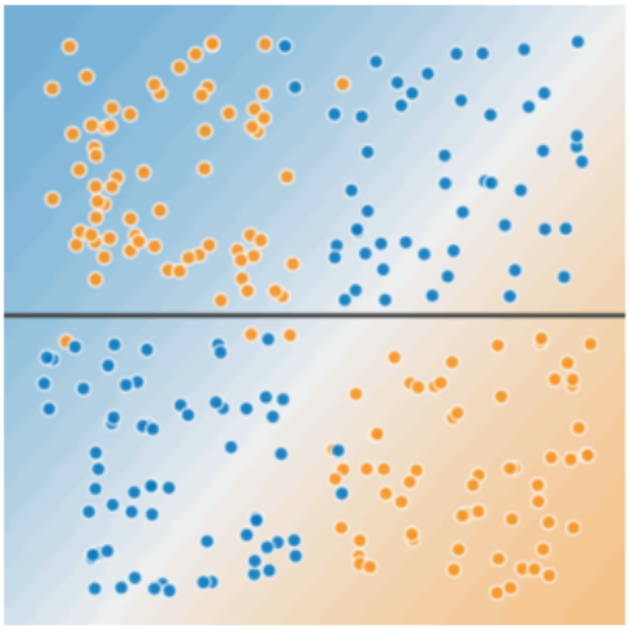
- **Omitted values.** For instance, a person forgot to enter a value for a house's age.
- **Duplicate examples.** For example, a server mistakenly uploaded the same logs twice.
- **Bad labels.** For instance, a person mislabeled a picture of an oak tree as a maple.
- **Bad feature values.** For example,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

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

1.9 Feature Crosses

1.9.1 Encoding Nonlinearity



This is a nonlinear problem. Any line you draw will be a poor predictor of tree health. To solve the nonlinear problem shown in Figure, create a feature cross. 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 x3 by crossing x1 and x2:

$$x_3 = x_1 x_2$$

We treat this newly minted x3 feature cross just like any other feature. The linear formula becomes:

$$y = b + w_1 x_1 + w_2 x_2 + w_3 x_3$$

A linear algorithm can learn a weight for w3 just as it would for w1and w2 In other words, although w3 encodes nonlinear information, you don't need to change how the linear model trains to determine the value of w3

Kinds of feature crosses

We can create many different kinds of feature crosses. For example:

- `[A X B]`: a feature cross formed by multiplying the values of two features.
- `[A x B x C x D x E]`: a feature cross formed by multiplying the values of five features.
- `[A x A]`: a feature cross formed by squaring a single feature.

Thanks to stochastic gradient descent, linear models can be trained efficiently. Consequently, supplementing scaled linear models with feature crosses has traditionally been an efficient way to train on massive-scale data sets.

.

1.9.2 Crossing One-Hot Vectors

In practice, machine learning models seldom cross continuous features. However, machine learning models do frequently cross one-hot feature vectors. Think of feature crosses of one-hot feature vectors as logical conjunctions. For example, suppose we have two features: country and language.

1.10 Regularization for Simplicity

1.10.1 L_2 Regularization

the following **generalization curve**, which shows the loss for both the training set and validation set against the number of training iterations.

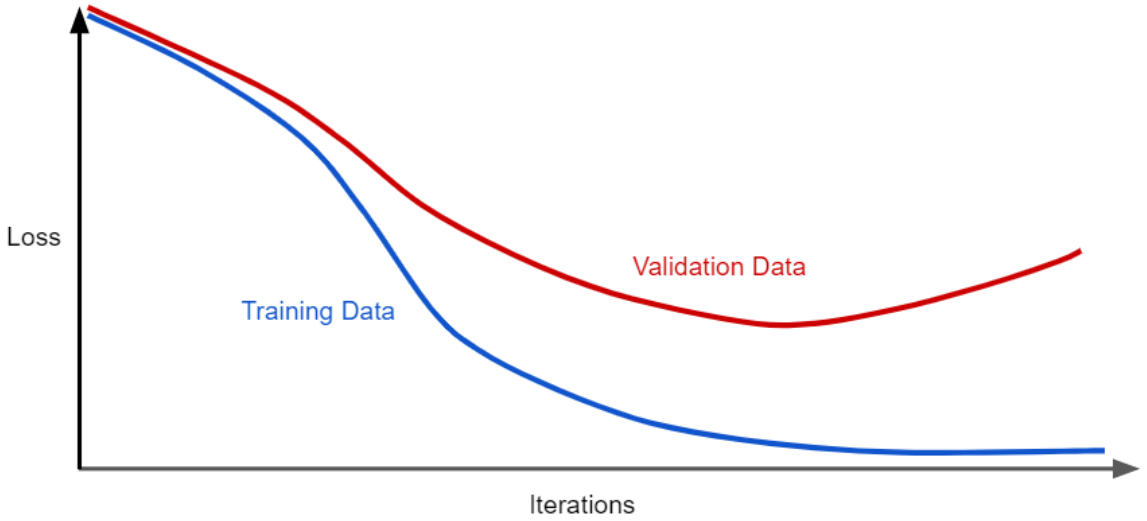


Figure 1 shows a model in which training loss gradually decreases, but validation loss eventually goes up. In other words, this generalization curve shows that the model is overfitting to the data in the training set. Channeling our inner Ockham, perhaps we could prevent overfitting by penalizing complex models, a principle called **regularization**. In other words, instead of simply aiming to minimize loss (empirical risk minimization):

$$\text{minimize}(\text{Loss}(\text{Data}|\text{Model}))$$

we'll now minimize loss+complexity, which is called **structural risk minimization**:

$$\text{minimize}(\text{Loss}(\text{Data}|\text{Model}) + \text{complexity}(\text{Model}))$$

Our training optimization algorithm is now a function of two terms: the **loss term**, which measures how well the model fits the data, and the **regularization term**, which measures model complexity.

Machine Learning Crash Course focuses on two common (and somewhat related) ways to think of model complexity:

- Model complexity as a function of the *weights* of all the features in the model.
- Model complexity as a function of the *total number of features* with nonzero weights. (A later module covers this approach.)

If model complexity is a function of weights, a feature weight with a high absolute value is more complex than a feature weight with a low absolute value.

We can quantify complexity using the ***L_2* regularization** formula, which defines the regularization term as the sum of the squares of all the feature weights:

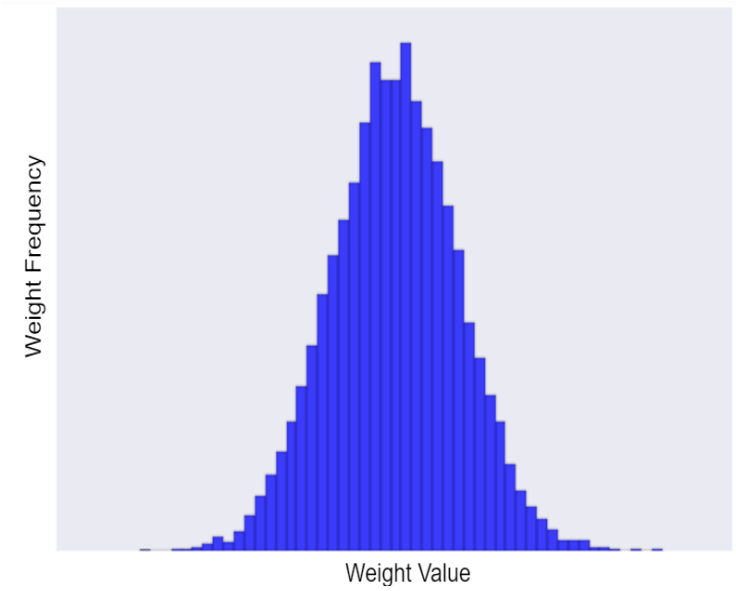
$$L_2 \text{ regularization term} = ||\boldsymbol{w}||_2^2 = w_1^2 + w_2^2 + \ldots + w_n^2$$

In this formula, weights close to zero have little effect on model complexity, while outlier weights can have a huge impact.

1.10.2 Lambda

Model developers tune the overall impact of the regularization term by multiplying its value by a scalar known as **lambda** (also called the **regularization rate**). That is, model developers aim to do the following:

$$\text{minimize}(\text{Loss}(\text{Data}|\text{Model}) + \lambda \, \text{complexity}(\text{Model}))$$



Performing L_2 regularization has the following effect on a model

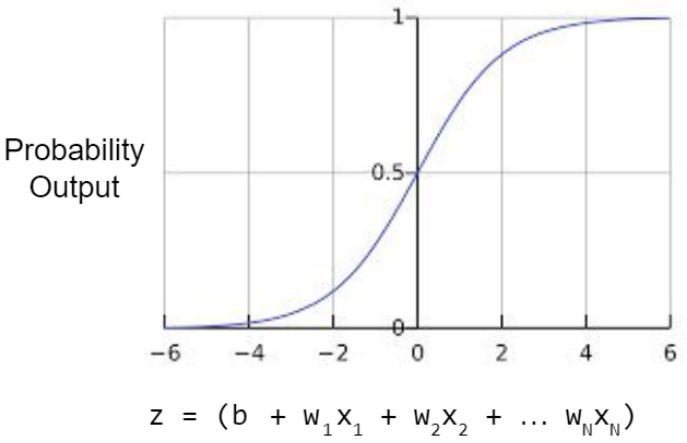
- Encourages weight values toward 0 (but not exactly 0)
- Encourages the mean of the weights toward 0, with a normal (bell-shaped or Gaussian) distribution.

Increasing the lambda value strengthens the regularization effect. For example, the histogram of weights for a high value of lambda might look as shown in Figure

1.11 Logistic Regression

1.11.1 Calculating a Probability

we might use the probability "as is." Suppose we create a logistic regression model to predict the probability that a dog will bark during the middle of the night. We'll call that probability:



- The *w* values are the model's learned weights, and *b* is the bias.
- The *x* values are the feature values for a particular example.

$$z = log(\frac{y}{1-y})$$

1.11.2 Loss and Regularization

The loss function for linear regression is squared loss. The loss function for logistic regression is **Log Loss**, which is defined as follows:

$$\text{Log Loss} = \sum_{(x,y)\in D} -y\log(y') - (1-y)\log(1-y')$$

Regularization is extremely important in logistic regression modeling. Without regularization, the asymptotic nature of logistic regression would keep driving loss towards 0 in high dimensions. Consequently, most logistic regression models use one of the following two strategies to dampen model complexity:

- L_2 regularization.
- Early stopping, that is, limiting the number of training steps or the learning rate.

1.12 Classification

1.12.1 Thresholding

A logistic regression model that returns 0.9995 for a particular email message is predicting that it is very likely to be spam. Conversely, another email message with a prediction score of 0.0003 on that same logistic regression model is very likely not spam.

1.12.2 True vs. False and Positive vs. Negative

We can summarize our "wolf-prediction" model using a 2x2 confusion matrix that depicts all four possible outcomes:

True Positive (TP): • Reality: A wolf threatened. • Shepherd said: "Wolf." • Outcome: Shepherd is a hero.	**False Positive (FP):** • Reality: No wolf threatened. • Shepherd said: "Wolf." • Outcome: Villagers are angry at shepherd for waking them up.
False Negative (FN): • Reality: A wolf threatened. • Shepherd said: "No wolf." • Outcome: The wolf ate all the sheep.	**True Negative (TN):** • Reality: No wolf threatened. • Shepherd said: "No wolf." • Outcome: Everyone is fine.

A **true positive** is an outcome where the model *correctly* predicts the *positive* class. Similarly,

A **true negative** is an outcome where the model *correctly* predicts the *negative* class.

A **false positive** is an outcome where the model *incorrectly* predicts the *positive* class.

A **false negative** is an outcome where the model *incorrectly* predicts the *negative* class.

1.12.3 Accuracy

Accuracy is one metric for evaluating classification models. Informally, accuracy is the fraction of predictions our model got right. Formally, accuracy has the following definition:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

For binary classification, accuracy can also be calculated in terms of positives and negatives as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

1.13 Regularization for Sparsity

1.13.1 L_1 Regularization

Sparse vectors often contain many dimensions. Creating a feature cross results in even more dimensions. Given such high-dimensional feature vectors, model size may become huge and require huge amounts of RAM.

In a high-dimensional sparse vector, it would be nice to encourage weights to drop to exactly `0` where possible. A weight of exactly 0 essentially removes the corresponding feature from the model. Zeroing out features will save RAM and may reduce noise in the model.

L_1 vs. L_2 regularization

L_2 and L_1 penalize weights differently:

- L_2 penalizes $weight^2$.
- L_1 penalizes $|weight|$.

Consequently, L_2 and L_1 have different derivatives:

- The derivative of L_2 is 2 * *weight*.
- The derivative of L_1 is *k* (a constant, whose value is independent of weight).

1.14 Neural Networks

1.14.1 Structure

Artificial neural networks, usually simply called neural networks, are computing systems vaguely inspired by the biological neural networks that constitute animal brains. An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain