Google Machine Learning Course

Amaan Ahmad

July 2021

Abstract

Machine Learning Crash Course with TensorFlow APIs Google's fast-paced, practical introduction to machine learning. Learn and apply fundamental machine learning concepts with the Crash Course, get real-world experience with the companion Kaggle competition, or visit Learn with Google AI to explore the full library of training resources.
Course can be found here

Contents

1 Framing

1.1 Supervised Machine Learning

The basic framework that will be explored is 'Supervised Machine Learning'. Here, models are creates that combine inputs to produce useful predictions, even on previously unseen data.

1.2 Labels

A label is the target variable that is to be predicted — 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.

In software such as a spam detector, the labels could include 'spam' or 'not spam'.

1.3 Features

Features are the way data is represented. 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 multiples of features, specified as: $x_1, x_2, ..., x_n$

In a spam detector example, the features could include the following:

- email address
- words or phrases in the content of the email
- time of day that email was sent

1.4 Examples

An 'example' is a particular instance of data - such as one email. This could be either:

- a **labelled example** - which has both feature information represented in the email and the label value of 'spam' or 'not spam'. These examples are used to **train** the model.
- an **unlabelled example** - which there is the feature information but no information on whether it is 'spam' or 'not spam'. These examples are used to **test** the model.

1.5 Models

A 'model' defines the relationship between the features and the label and this is what is doing the predicting through a process of **learning** from data.

For example, with the spam detection model, the model may associate some features stronger with 'spam' than others.

There are two phases in a models life:

- **Training** - means creating or **learning** the model. In this phase, you show the model labelled examples to enable the model to gradually learn the relationships between features and label (x).
- **Inference** - means applying the trained model to unlabeled examples. Here, the trained model is used to make useful predictions (y).

1.6 Regression or Classification

- **Regression** models predicts continuous values (i.e. value of a house or probability).
- **Classification** models predict discrete values (i.e. if an email is 'spam' or 'not spam', or is an image a 'dog', a 'cat' or 'horse').

2 Descending into ML

2.1 Linear Regression

For a linear relationship, we can model it using the equation for a line, which is:

$$y = mx + c$$

However, in machine learning we use:

$$y' = w_1 x_1 + b$$

Where:

- y' is the predicted label - the output.
- w_1 is the weight of feature 1.
- x_1 is the feature.
- b is the bias - also known as w_0.

This model only uses one feature however if we wanted to use multiple features, each feature would have a different weight and hence the formula would look like this:

$$y' = b + w_1 x_1 + w_2 x_2 + w_3 x_3 + ... + w_n x_n$$

2.2 Training and Loss

Training a model means determining good values for all the weights and the bias from labelled examples. In supervised learning, a machine learning algorithm determines a suitable 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 of making a bad predictions. It is a number indicating how bad the model's prediction was on a single example. The goal of training a model is to find a set of weights and biases that have a *low loss* on average with the aim of a perfect model having the loss of zero. Figure 1 demonstrates how a model attempted to find an appropriate model. Here the Arrowed lines represent the loss and the blue line represent the predictions.

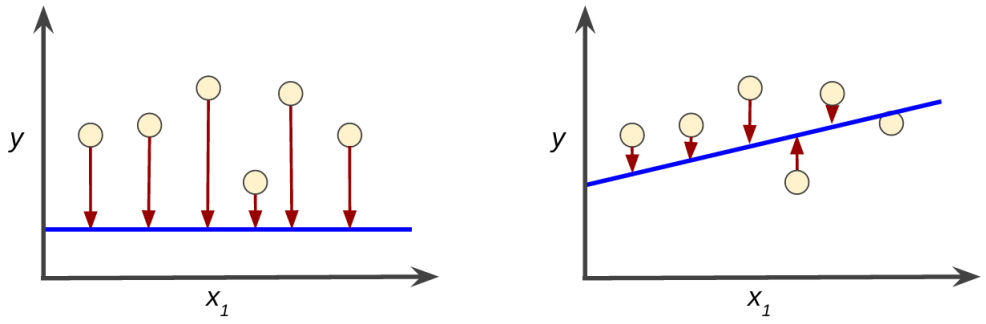


Figure 1: Shows a high loss model (left) and a low loss model (right)

Mathematical functions — called **loss functions** — are used to aggregate the individual losses in a meaningful fashion.

2.2.1 Squared Loss

We use a loss function on linear regression called **Squared Loss** (also known as L_2 **loss**). It is the square of the difference of the label and the prediction as shown by the equation:

$$(observation - prediction(x))^2 \text{ or } (y - y')^2$$

2.2.2 Mean Square Error

MSE is the average squared loss per example over the entire dataset. MSE is calculated but summing all the squared losses and then divide by the number of examples:

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

Where:

- (x, y) is an example in which
  - x is the set of features that the model uses to make predictions.

- y is the example's label.

- $prediction(x)$ is a function of the weights and bias with the set of features x.
- D is the data set which contain numerous labelled examples, which are (x, y) pairs.
- N is the number of examples in D.

Although MSE is commonly-used in machine learning, it is neither the only practical loss function nor the best loss function for all circumstances.

3 Reducing Loss

3.1 Iterative Approach

Similar to the *Hot or Cold game* where we start with a guess then see what the loss is, then try another guess of model parameters, (w_1 and b_1), and see if you are closer to the smallest loss. Figure 2 shows use this process.

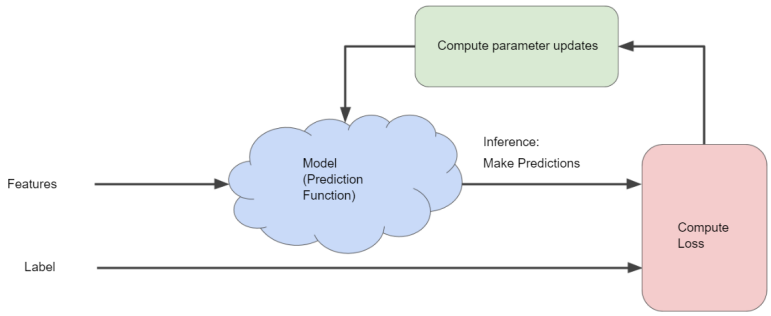


Figure 2: Shows the trial and error approach which machine learning algorithms us to train the model

For example for a linear regression:

$$y' = w_1 x_1 + b$$

We choose random numbers for example; $b = 0$ and $w_1 = 0$ The initial values we choose does not matter. We then get the predict value y'. Using this predicted value we use the loss function (finding the difference between y' and the correct label y corresponding to features x) to determine the loss. The program then iterates over to find the corresponding pair of values to find the lowest loss. Eventually, the loss stops changing (or at least changes extremely slowly) and this means the model **converges**. Iterative strategies are prevalent in machine learning, primarily because they scale so well to large data sets.

3.2 Gradient Decent

For linear regression problems, the resulting graph of loss vs w_1 will always be convex. which looks like this:

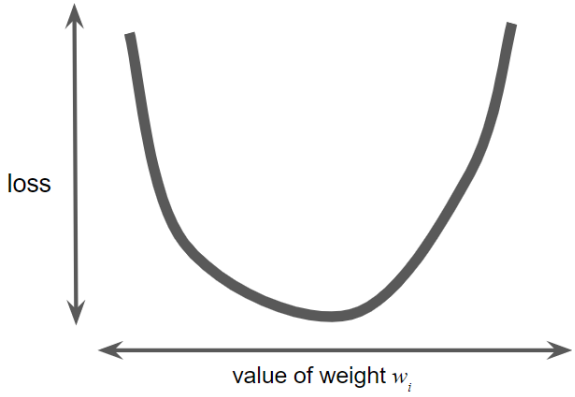


Figure 3: Shows the convex shaped graph

As shown in figure 3, we can see that the convex problem only has one minimum and this is where the loss function converges. To find the convergence point we use a mechanism called **gradient descent**.
The algorithm picks a starting point (doesn't matter much so most algorithms set $w_1 = 0$), finds the gradient of the curve, then the algorithm takes a **step** in the direction of negative gradient to reduce loss. 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. (Note: The gradient is a vector, so it has both direction and magnitude.) The gradient always points in the direction of steepest increase in the loss function. The gradient descent algorithm takes a step in the direction of the negative gradient in order to reduce loss as quickly as possible. To determine the next point along the loss function curve, the gradient descent algorithm adds some fraction (step) of the gradient's magnitude to the starting point. The value of this step is very important, as shown below in figure 4.

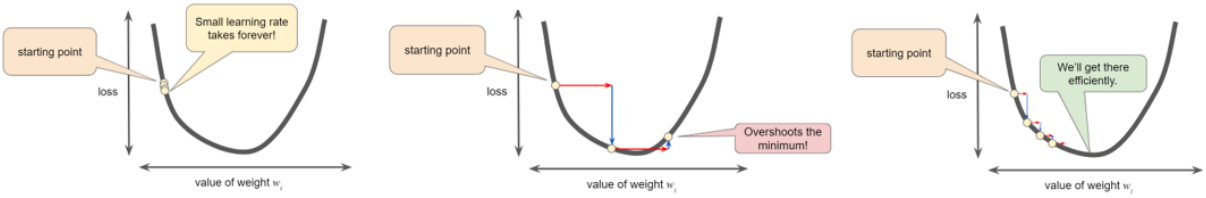


Figure 4: Left graph shows the small steps, Middle graph shows the effect of large steps and the Right graph shows the Goldilocks approach

When performing gradient descent, we generalize the above process to tune all the model parameters simultaneously. For example, to find the optimal values of both w_1 and the bias b, we calculate the gradients with respect to both w_1 and b. Next, we modify the values of w_1 and b based on their respective gradients. Then we repeat these steps until we reach minimum loss.
The step size is also referred as the **learning rate**. Gradient descent algorithms multiply the gradient by this value to determine the next point. **Hyperparameters** are "knobs" that programmers tweak in machine learning algorithms to tune the learning rate. As you can see from figure 4, if the learning rate is too small the learning would take too long. If the learning rate is to big, there is a possibility that you can overshoot the optimal point. For every regression problem there is a Goldilocks learning rate which is related to how flat the loss funciton is. If the gradient of the loss function at the point is small then you can try a larger learning rate and vice-versa.

3.3 Stochastic Gradient Descent - Reducing Loss

In gradient descent, a **batch** is the total number of examples you use to calculate the gradient in a single iteration. When working with large data sets, there are going to be huge numbers of features and as such even a single iteration can take a very long time to compute. A large data set with randomly sampled examples probably contains redundant data. In fact, redundancy becomes more likely as the batch size grows.
Stochastic gradient descent (SGD) 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.
Mini-batch stochastic gradient descent (mini-batch SGD) is a compromise between full-batch iteration and SGD. A mini-batch is typically between 10 and 1,000 examples, chosen at random. Mini-batch SGD reduces the amount of noise in SGD but is still more efficient than full-batch.

4 First Steps with 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. Please note that use of tf.keras requires understanding of **NumPy** and **pandas**. For a brief recap view these **NumPy** and **pandas** tutorials.
To see the code used in this section please refer to `code/FirstStepsWithTF`.

The following figure shows the hierarchy of TensorFlow toolkits:

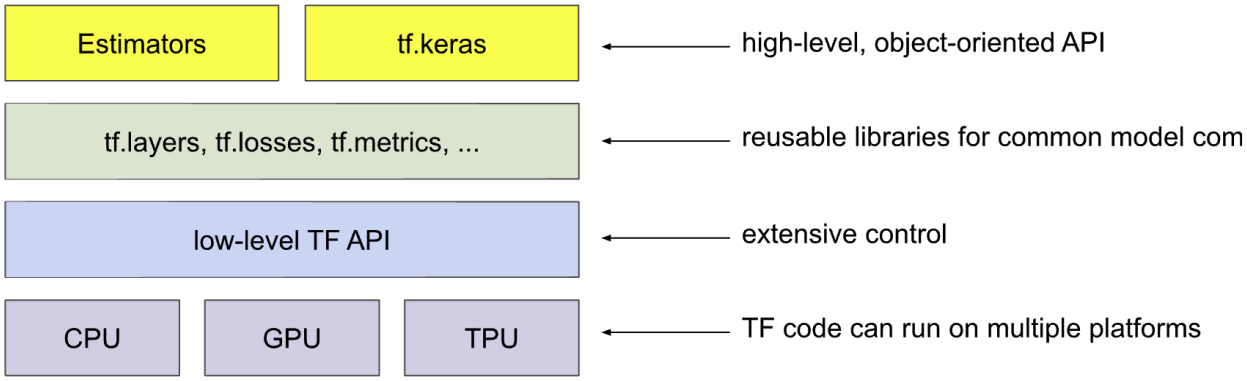


Figure 5: TensorFlow toolkit hierachy

4.1 Summary of hyperparameter tuning

Most machine learning problems require a lot of hyperparameter tuning. Unfortunately, we can't provide concrete tuning rules for every model. Lowering the learning rate can help one model converge efficiently but make another model converge much too slowly. You must experiment to find the best set of hyperparameters for your dataset. That said, here are a few rules of thumb:

- Training loss should steadily decrease, steeply at first, and then more slowly until the slope of the curve reaches or approaches zero.
- If the training loss does not converge, train for more epochs.
- If the training loss decreases too slowly, increase the learning rate. Note that setting the learning rate too high may also prevent training loss from converging.
- If the training loss varies wildly (that is, the training loss jumps around), decrease the learning rate.
- Lowering the learning rate while increasing the number of epochs or the batch size is often a good combination.
- Setting the batch size to a very small batch number can also cause instability. First, try large batch size values. Then, decrease the batch size until you see degradation.
- For real-world datasets consisting of a very large number of examples, the entire dataset might not fit into memory. In such cases, you'll need to reduce the batch size to enable a batch to fit into memory.

Remember: the ideal combination of hyperparameters is data dependent, so you must always experiment and verify.

4.2 Correlation Matrix

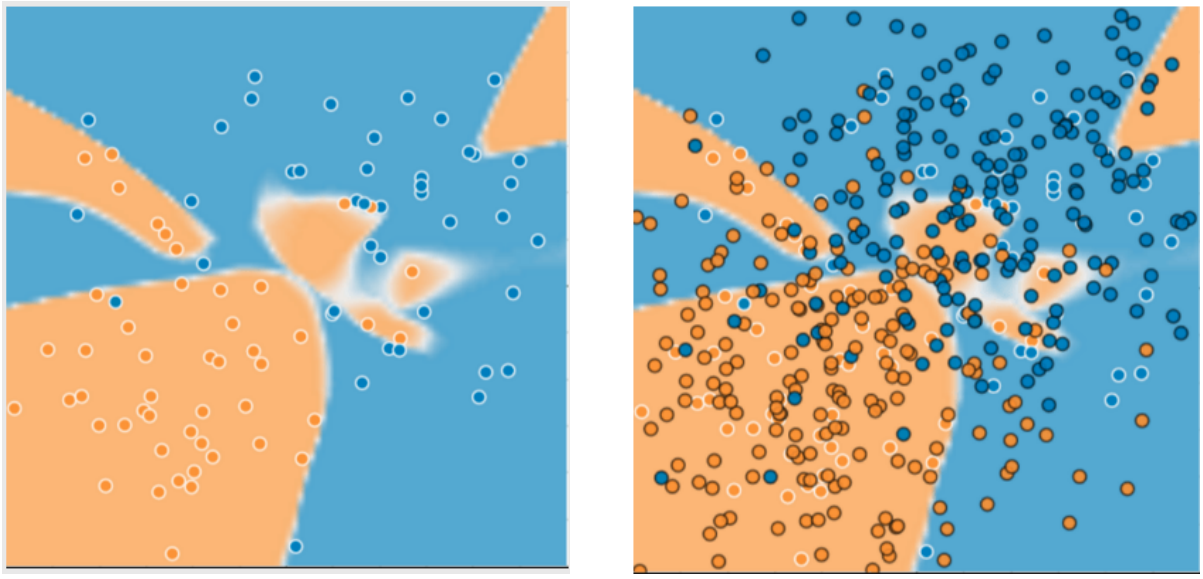
A **correlation matrix** indicates how each attribute's raw values relate to the other attributes' raw values. Correlation values have the following meanings:

- 1.0: perfect positive correlation; that is, when one attribute rises, the other attribute rises.
- -1.0: perfect negative correlation; that is, when one attribute rises, the other attribute falls.
- 0.0: no correlation; the two columns are not linearly related.

In general, the higher the absolute value of a correlation value, the greater its predictive power.

5 Generalisation

The figure below shows the two graphs **overfit** the traits of the data that the model was trained on. The model gets a low loss during training but does a poor job **predicting** new data as shown by the figure on the right.



(a) A complex model for distinguishing sick from healthy trees.

(b) The model did a bad job predicting new data.

Figure 6

Overfitting is caused by making a model more complex than necessary. The fundamental tension of machine learning is between fitting our data well, but also fitting the data as simply as possible.

5.1 William of Ockham

William of Ockham was a 14th century philosopher and loved simplicity. He believed that simpler formulae and theories are better than more complex ones. Thus in machine learning terms:

"The less complex an ML model, the more likely that a good empirical result is not just due to the peculiarities of the sample."

In modern times, we've formalised Ockham's razor into the fields of **statistical learning theory** and **computational learning theory**. These fields have developed **generalisation bounds** - a statistical description of a model's ability to **generalise** to new data based on factors such as:

- the complexity of the model
- the model's performance on training data

While the theoretical analysis provides formal guarantees under idealised assumptions, they can be difficult to apply in practice. Therefore, let's focus instead on empirical evaluation to judge a model's ability to generalise to new data. A machine learning model aims to make good predictions on new, previously unseen data. But if you are building a model from your data set, how would you get the previously unseen data? Well, one way is to divide your data set into two subsets;

- **training set** - a subset to train a model
- **test set** - a subset to test the model

Good performance on the test set is a useful indicator of good performance on the new data in general, assuming that:

- The test set is large enough.
- You don't cheat by using the same test set over and over.

5.2 Machine Learning Fine Print

There are three basic assumptions that aid generalisation:

- Drawing examples **independantly and identically (i.i.d)** at random. This ensures the examples do not influence each other.
- The distribution is **stationary** and so it doesn't change within the data.
- Drawing examples from partitions from the same distribution.

If the key assumptions of supervised ML are not met, then we lose important theoretical guarantees on our ability to predict on new data. However, in practise, we sometimes violate these assumptions.

6 Training and Test Sets

If you have a single data set, it is important to split the data into two subsets; the **training set** and the **test set**.
It is important that the test set is:

- Large enough to get meaningful results
- Is representative of the data in the entire set.

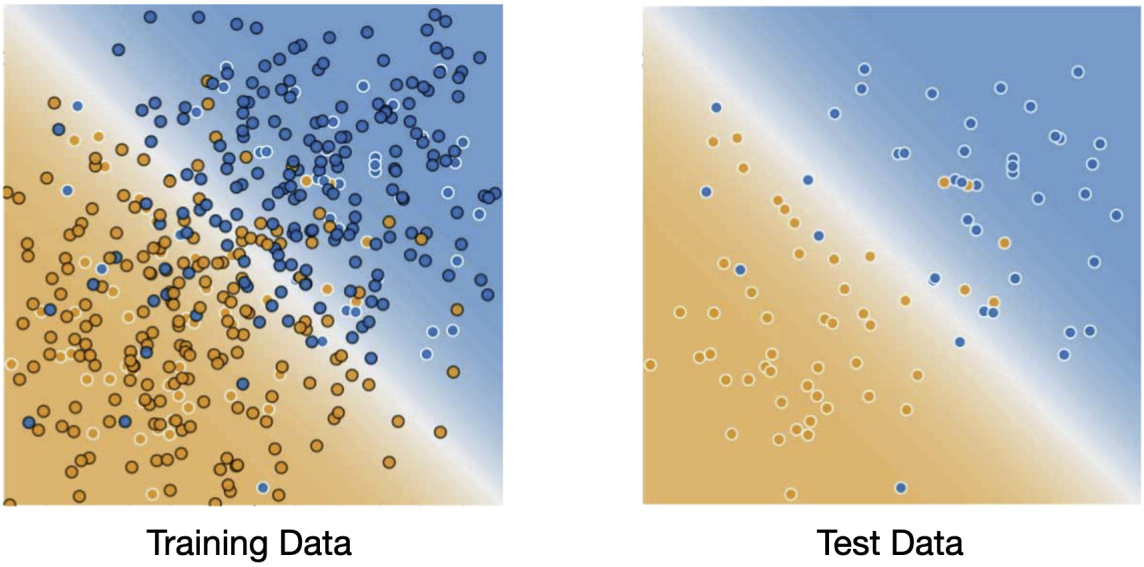
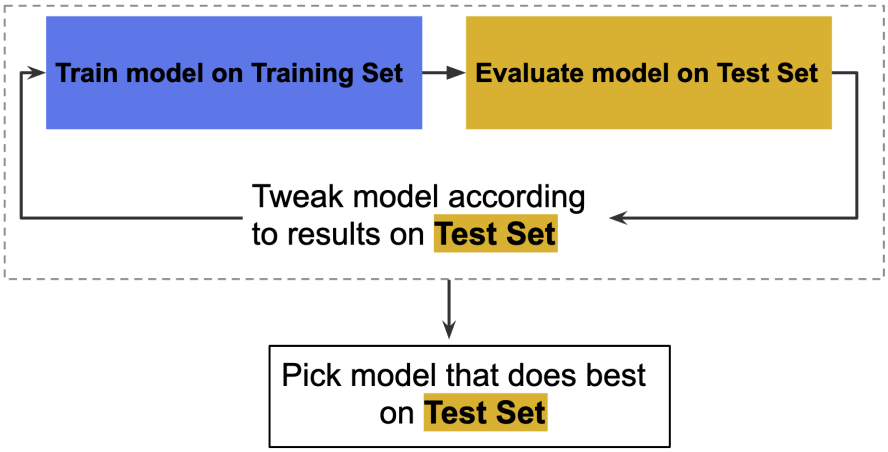


Figure 7: Good example of a training and a testing set

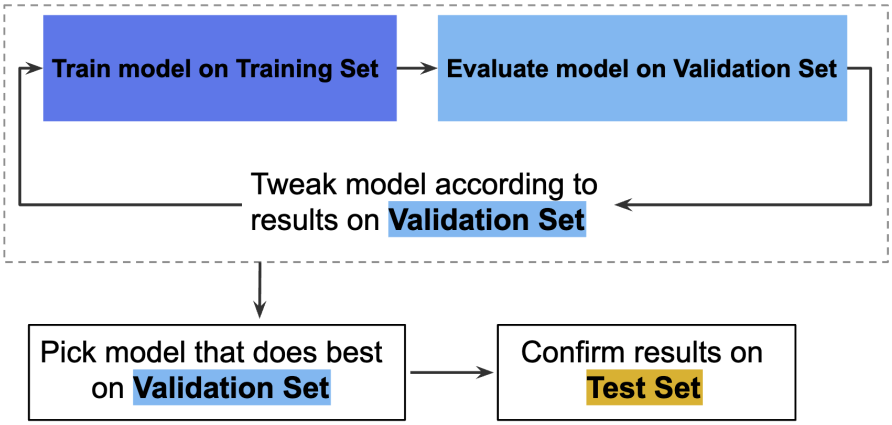
Another important thing is that you must make sure that you never train the model using the test set as this gives you unrealistically strong opinions on how good your model is.

7 Validation

In the model mentioned previously, the dataset is split into 2 subsets as seen in **Figure 8a**. However, this poseses a problem for us as the "Tweak Model", i.e. adjusting anything about the model (e.g. learning rate) to designing a new model, is evaluated against the test set and leads to easily overfitting. Due to this, we divide the set into 3 subsets as seen in **Figure 8b**. This has an improved workflow as it creates fewer exposures to the test set. As such we pick the model that does best on the new **validation set** and double check the model against our test set. This helps as the model is chosen based on the validation set and also that the model is double checked by the test set.



(a) Workflow with two partitions of the dataset



(b) Workflow with three partitions of the dataset

Figure 8

8 Representation

8.1 Feature Engineering

The left side of **Figure 9** demonstrates the raw data from the source and the right shows the **Feature Vector**, which is the set of floating-point values comprising the examples in your data set.

Feature Engineering is the process in which raw data is transformed into a feature vector. Machine Learning Engineers spend a lot of time (about 75%) 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.

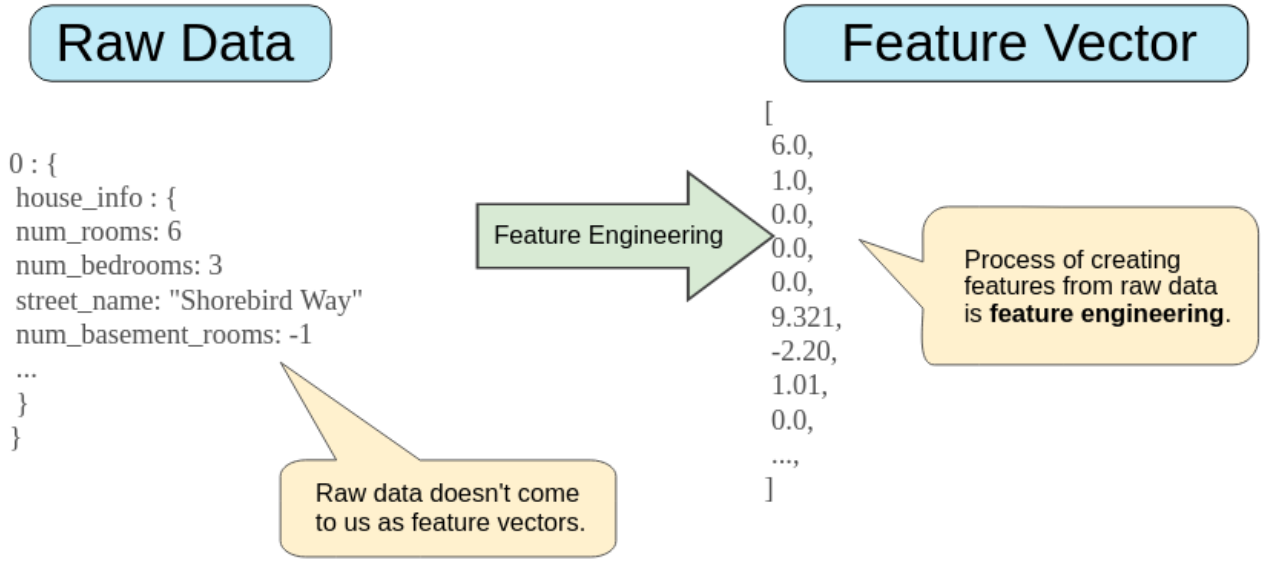


Figure 9: Feature Engineering example

8.1.1 Mapping Numerical Values

Mapping numerical values to floating-point is trivial as they can be multiplied by a numeric weight. The figure below gives an example.

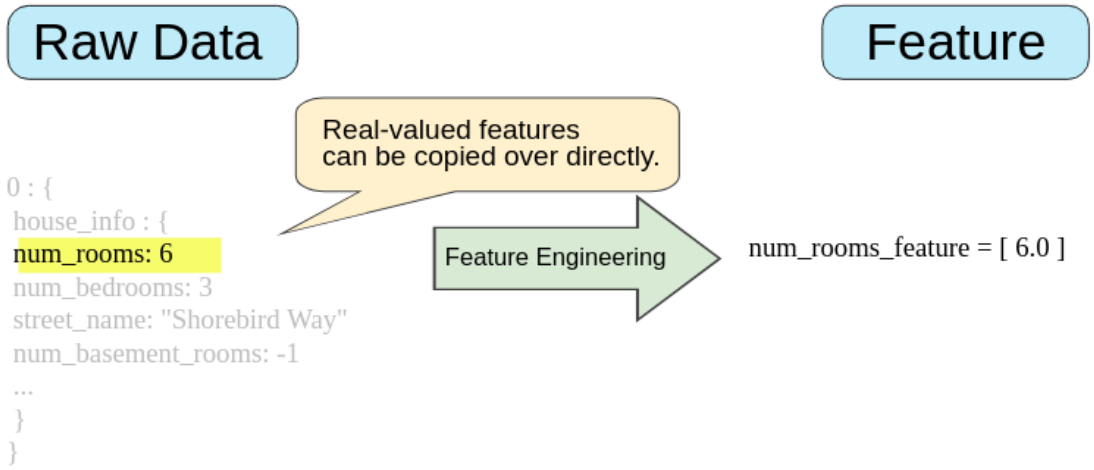


Figure 10: Mapping numerical values to floating point values

8.1.2 Mapping Categorical Values

Categorical features have a discrete set of possible values. For example, with a feature called `street_name`, options might 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, we can map our street names to numbers:

- Charleston Road = 0
- North Shoreline Boulevard = 1
- Shorebird Way = 2
- Rengstorff Avenue = 3
- Everything else (OOV) = 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. This approach effectively creates a Boolean variable for every

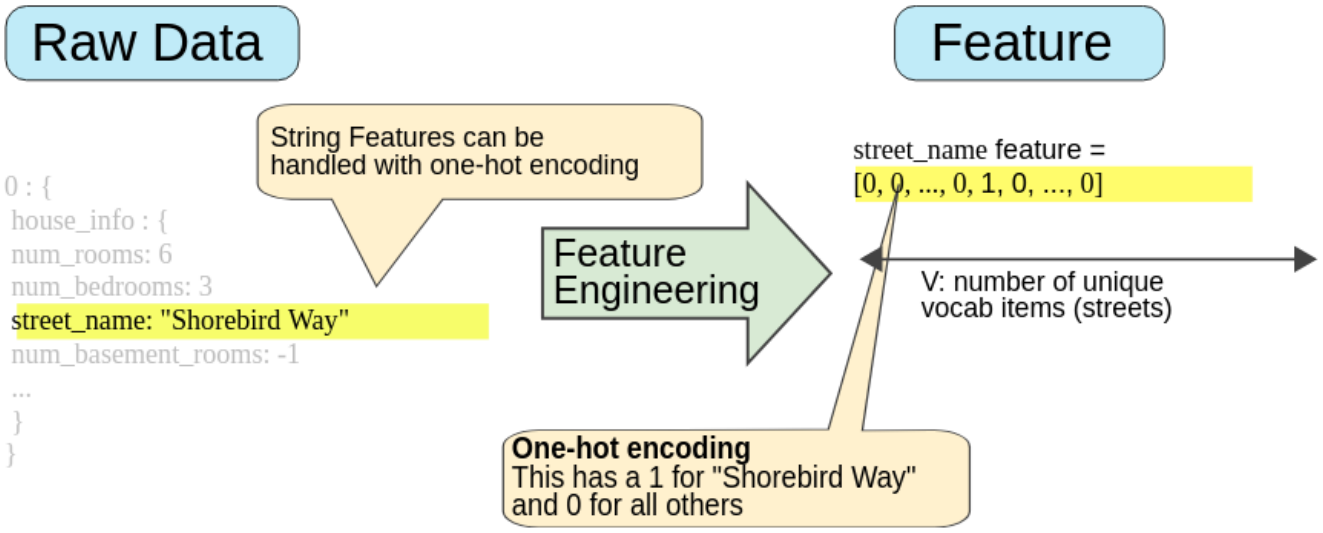


Figure 11: illustrates a one-hot encoding of a particular street: Shorebird Way. The element in the binary vector for Shorebird Way has a value of 1, while the elements for all other streets have values of 0.

feature value (e.g., `street_name`). Here, if a house is on Shorebird Way then the binary value is 1 only for Shorebird Way. Thus, the model uses only the weight for Shorebird Way.

Similarly, if a house is at the corner of two streets, then two binary values are set to 1, and the model uses both their respective weights. Note : **One-hot encoding** extends to numeric data that you do not want to directly multiply by a weight, such as a postal code.

8.1.3 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** in which only nonzero values are stored. In sparse representations, an independent model weight is still learned for each feature value, as described above.

8.2 Qualities of Good Features

8.2.1 Avoid rarely used Discrete Feature Values

Having many **discrete feature values** would help the model see the feature in different situations, and thus would be able to determine when it is a good predictor. For example, having `house_type:` victorian instead of `unique_house_id:` 8SK982ZZ1242Z. Having the latter, would mean the model would not be able to learn anything from it as it would appear

very rarly. A good feature value should appear more than 5 or so times in a data set. Doing so enables a model to learn how this feature values relates to the label.

8.2.2 Provide Obvious Meanings

Each feature should give a clear meaning for example, having `house_age_years:` 27 instead of `house_age:` 851472000. This helps us find noisy data for example `user_age_years:` 277, which is easier to spot that it this cannot be valid.

8.2.3 Don't mix actual data with "Magic Numbers"

Good floating-point features don't contain peculiar out-of-range discontinuities or "magic" values. If there is no value associated, e.g. if the user did not enter a value, then have a Boolean feature (e.g. `quality_rating` = -1) to specify if that feature is not defined.

To mark these "magic' values, in the original feature, replace them 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.

8.2.4 Account for changes in definitions for Feature Values

The meaning of the feature value should not change over time. For example, having `city_lol:` "london" instead of `inferred_city_cluster:` 23. As "london" is less likely to change in comparison to the meaning of 23.

8.3 Cleaning Data

8.3.1 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 multiple features, the 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** (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.

Note: you don't have to give every floating-point feature exactly the same scale as if Feature A is ranged -1 to +1 while Feature B is -3 to +3, nothing bad will happen. However if Feature B is scaled from 5000 to 10000 your model will react poorly.
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:

$$scaledValue = \frac{(value - mean)}{(stdDev)}$$

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.

8.3.2 Handling Extreme Outliers

For **outliers**, there are numberious ways to avoid this:

- Logging the values - e.g. roomsPerPerson = log((totalRooms population) + 1)
- Clipping, capping the values at a certain point. Therefore any data larger than this max point is compressed together as each take up its value.

8.3.3 Binning

Binning is grouping values together as shown below.

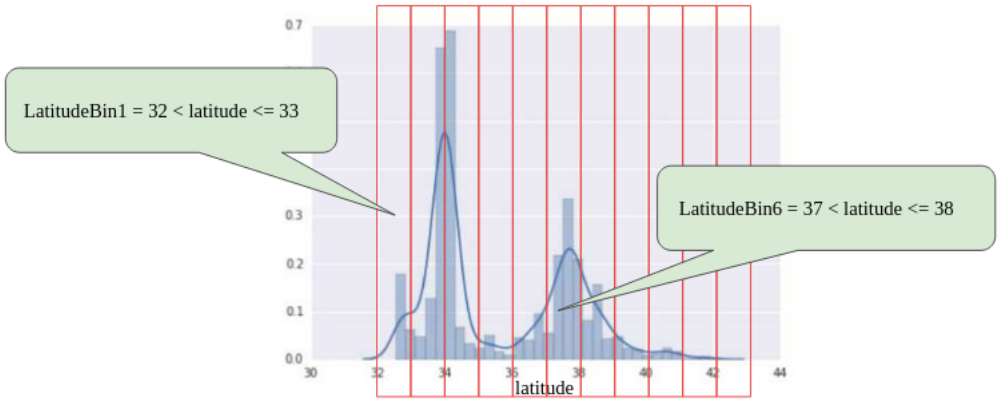


Figure 12: Binning Latitude

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 $\frac{35}{34}$ more expensive (or less expensive) than houses at latitude 34. And yet, individual latitudes probably are a pretty good predictor of house values. To make latitude a helpful

predictor, they are instead divided up into the "bins" that can be seen above. Another approach is to bin by **quantile**, which ensures that the number of examples in each bucket is equal. Binning by quantile completely removes the need to worry about outliers. Instead of having one floating-point feature, we now have 11 distinct boolean features. Having 11 separate features is somewhat inelegant, so let's unite them into a single 11-element vector.

8.3.4 Scrubbing

Many examples in data sets are unreliable, and hence these values need to be removed from the set. Reasons include:

- **Omitted Values** - If the user forgets to enter a value.
- **Duplicates**
- **Bad Labels** - Labelling errors.
- **Bad Feature Values** - Value errors e.g. adding an extra digit or a wrong one.

To find bad examples, a Histogram would be a good way to represent the data, helping you to visualise and detect bad data in aggregate. Other statistics that would help find errors would be; Maximum, Minimum, Mean, Median and Standard Deviation. 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. However, detecting bad feature values or labels can be far trickier.
Ultimately you want to **know your data** and as such:

- 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 (e.g. dashboards).

9 Feature Crosses

If it is difficult to separate the categories in the model as shown in **Figure 13**, as this is no longer a linear problem. Instead, we create a feature cross to solve the nonlinearity of this problem. A **feature cross** is a **synthetic feature** that encodes nonlinearity in the feature space by multiplying two or more input features together. For example, if we had features x_1 and x_2, the feature cross, x_3, would be $x_3 = x_1 * x_2$.
This would mean the linear formula becomes: $y = b + w_1x_1 + w_2x_2 + w_3x_3$. Where the weights w_1, w_2 and w_3 are all learnt by the linear model even thought w_3 encodes nonlinear information. Many different kinds of feature crosses exist such as $[AXB]$ to $[AXBXCXDXE]$ to $[AXA]$.

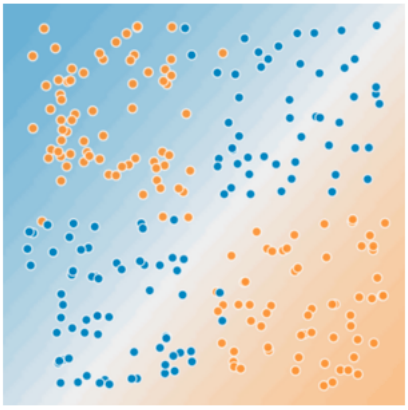


Figure 13: Figure showing a model with nonlinearity

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

10 Regularisation for Simplicity

As you can see in **Figure 14** as you increase the iterations, the training loss decreases however the validation loss eventually begins to increase. This is because the model is **overfitting** to the given data. Up till now we have only been trying to decrease the training loss, now we will explore how we can avoid overfitting using a principle called **regularisation**.

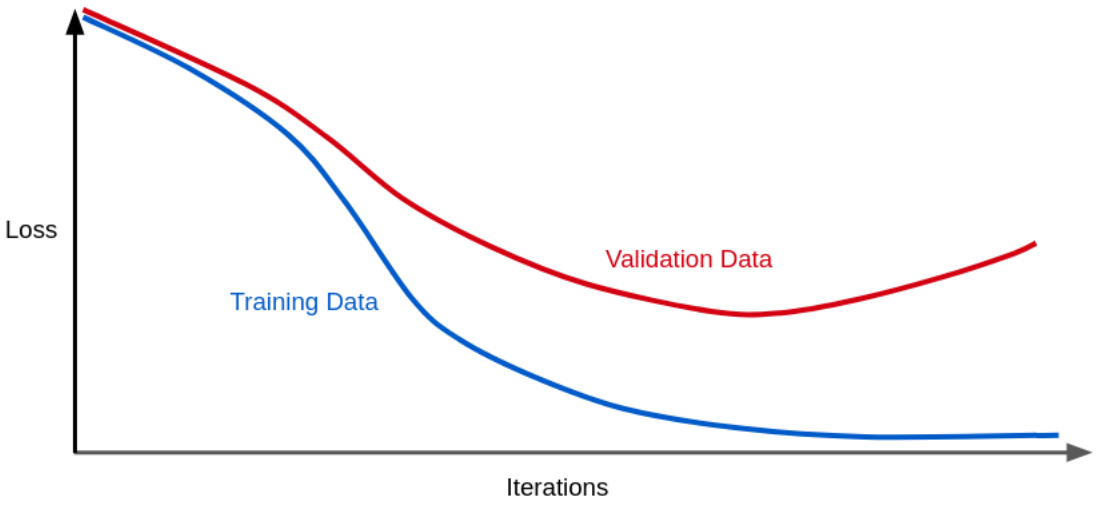


Figure 14: Figure showing the generalization curve

Instead of simply aiming to minimize loss (empirical risk minimization), our aim to is minimize loss+complexity, known as **structural risk minimization**:

$$minimize(Loss(Data|Model)) + complexity(Model)$$

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. As such, by usingL_2 **regularization**, which quantifies the regularization term as the sum of squares of the feature weights, weights close to zero have little effect on model complexity, while outlier weights can have a huge impact.

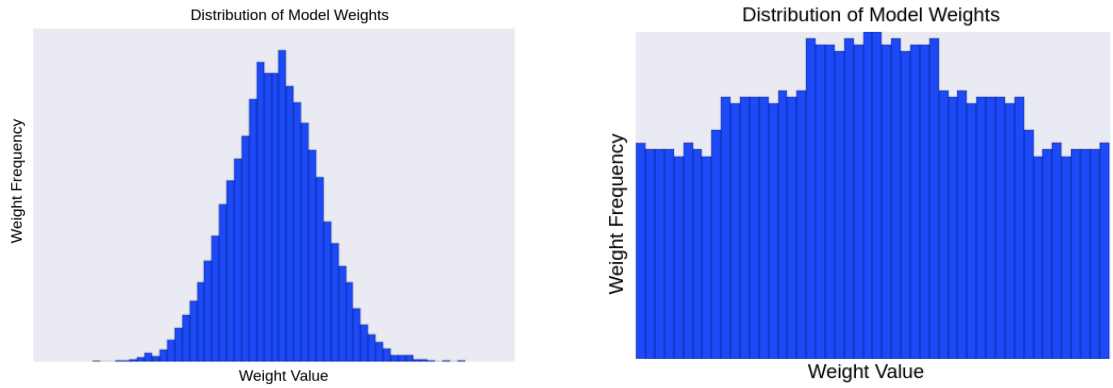
$$L_2 \text{ regularization term} = w_1^2 + w_2^2 + .. + w_n^2$$

10.1 Lambda - The Regularization Rate

To further tune the impact of the regularization term, we multiply the L_2 value by a scalar, λ (the **regularisation rate**). Thus we now aim to:

$$minimize(Loss(Data|Model)) + \lambda complexity(Model)$$

Therefore, the effect of regularization encourages the model's weight values and the mean of these weights towards 0, to form a normal Gaussian Distribution. Increasing the lambda value strengthens the regularization effect as seen below.



(a) Histogram of weights with a high lamda value.

(b) Histogram of weights with a low lamda va'Makefile' has lue.

Figure 15: Lowering the value of lambda tends to yield a flatter histogram

When choosing the right lambda value, the goal is to strike the right balance between simplicity and training-data fit. If the lambda value is:

- too high, the model would be simple however there is a greater risk of underfitting the data. Hence the model model won't learn enough about the training data to make useful predictions.

- too low, the model would be more complex however there is a greater risk of overfitting the data. This means it will learn too much about the particularities of the training data set, and won't be able to generalize to new data.

Note: Setting lambda to zero removes regularization completely. In this case, training focuses exclusively on minimizing loss, which poses the highest possible overfitting risk.

The ideal value of lambda produces a model that generalizes well to new, previously unseen data. Unfortunately, that ideal value of lambda is data-dependent, so you'll need to do some tuning.

10.2 Learning rate and Lambda

There's a close connection between learning rate and lambda. Strong L_2 regularization values tend to drive feature weights closer to 0. Lower learning rates (with **early stopping**) often produce the same effect because the steps away from 0 aren't as large. Consequently, tweaking learning rate and lambda simultaneously may have confounding effects.

Early stopping means ending training before the model fully reaches convergence. In practice, we often end up with some amount of implicit early stopping when training in an online (continuous) fashion. That is, some new trends just haven't had enough data yet to converge. The effects from changes to regularization parameters can be confounded with the effects from changes in learning rate or number of iterations. One useful practice (when training across a fixed batch of data) is to give yourself a high enough number of iterations that early stopping doesn't play into things.

11 Logistic Regression

There a two ways of calculating probability, "As is" and Converted to a binary category. The "As is" approach is self-explanatory. However, in classification cases we use a **sigmoid function**, which produces an output which always falls between 0 to 1. The sigmoid function is as follows:

$$y' = \frac{1}{1+e^{-z}}$$

Where

- y' is the output of the logistic regression model for a particular example
- $z = b + w_1x_1 + w_2x_2 + .. + w_Nx_N$

11.1 Loss function for Logistic Regression

The loss function for linear regression is squared loss. The loss function for logistic regression is Log Loss:

$$LogLoss = \Sigma_{(x,y)\in D} -ylog(y') - (1-y)log(1-y')$$

Where

- $(x, y) \in D$ is the dataset containing many labelled examples.
- y is the label in the label example. y is 0 or 1 as this is a logistic regression.
- y' is the predicted value, which is between 0 and 1.

12 Classification

12.1 Thresholding

Logistic regression returns a probability. We use the probability to predict how likely something is e.g. if an email was spam. We create a classification threshold to do the obvious, if its above a value then it is spam and if below it is not.

12.2 True vs. False, Positve vs. Negative

When predicting scenarios, there are four situations:

- True positive - when the model correctly predicts the positive class.
- True negative - when the model correctly predicts the negative class.
- False positive - when the model incorrectly predicts the positive class.
- False negative - when the model incorrectly predicts the negative class.

12.3 Accuracy, Precision and Recall

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

$$Precision = \frac{Number\ of\ True\ Positives}{Total\ number\ of\ Positive\ Predicitons}$$

$$Recall = \frac{Number\ of\ True\ Positives}{Number\ of\ True\ Positives\ Predicitons\ and\ Number\ of\ False\ Negatives}$$

12.4 ROC and AUC

ROC curve (Receiver Operating Characteristic Curve) is a graph that shows the performance of a classifcation model, it has two parameters: True Positive Rate and False Positive Rate.

$$TPR = \frac{TP}{TP+FN}$$
$$FPR = \frac{FP}{FP+TN}$$

The ROC pots TPR vs FPR at various classification decision thresholds.
AUC is Area Under the ROC Curve. It provides a total performance measure over all classificatoin thresholds.

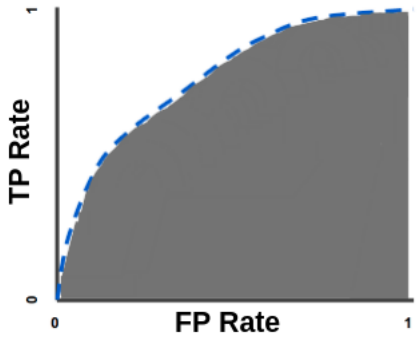


Figure 16: Area Under the ROC Curve

AUC can be interpreted as the probability that the model ranks a random positive example more highly than a random negative example. Therefore, the closer the AUC to 1.0, the better as an AUC value of 1.0 means the predictions are 100% correct.

12.5 Prediction Bias

Logistic regression predictions should be unbiased and this means:

$$average\ of\ predictions \approx average\ of\ observations$$

The prediction bias is a the measure of how far apart the two are:

$$prediction\ bias\ =\ average\ of\ predictions - average\ of\ observations$$

There are various causes of a prediction bias; Incomplete data set, Noisy data set, Buggy pipeline, Biased training sample or Over strong regularization. In order to correct the prediction bias by adding calibration layer. However, this could be problematic as you could be fixing symptoms rather than the cause or the system is brittle and you would need to constantly keep the model up-to-date.

As the logistic regression model predicts a value between 0 and 1 however all labelled exapmles are either 0 or 1 therefore we can use "bucketing" to mitigate this issue. The are two ways to form buckets, linearly breaking up the target predictions or forming quantiles.
Consider the following calibration plot from a particular model. Each dot represents a bucket of 1,000 values.

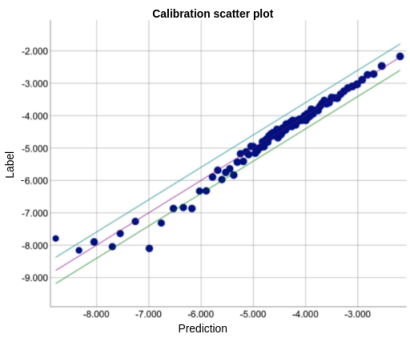


Figure 17: Area Under the ROC Curve

13 Regularization: Sparsity

Sparse vectors often contains many dimensions and creating a feature cross, creates even more dimensions. However, having numerous dimension features means when training the model, it becomes large and requires a lot of RAM.
In order to optimize the model we use regularization, however L_2 would not fix the issue as it does not force the weights to 0.0.
An apporoach we could take is L_0 regularization which is when chosen weights are reduced to 0 in order to be able to get the model to fit the data. However, this would turn our convex optimization problem into a non-convex optimization problem.
An alternate is called L_1 regularization which means to make the noisy uninformative coefficients to 0 in order to create an approxi mate to L_0 which saves a lot of RAM.

14 Neural Networks

A "non-linear" classification problem means that we can't accurately predict a label which has a model in the form of $b + w_1x_1 + w_2x_2$. For example:

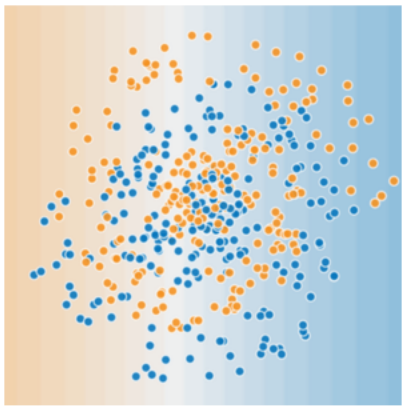


Figure 18: A Complex Nonlinear model

As you can see, we can't predict the model shown in **Figure 16** with a linear model. To begin understanding neural networks, let's visualize a linear model:

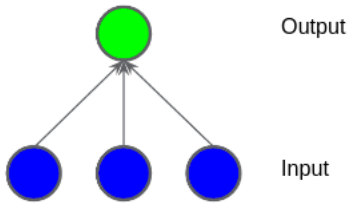


Figure 19: A Linear model as a graph

Each blue node represents an input feature and the green node which represents the weighted sum of the inputs.

14.1 Hidden Layers

We have now added a second layer to the model. The yellow nodes are the hidden layer which compute the weighted sum of the blue nodes and the green node computes the weighted sum of the yellow nodes. This model is still linear as it produces one output.

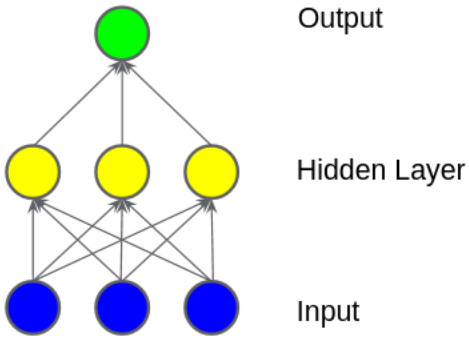


Figure 20: Graph if a two-layer model

14.2 Activation Functions

To model a nonlinear problem, we can introduce nonlinearity. In the model below we see that Hidden Layer 1 is transformed by a nonlinear function before computing the weighted sum for the next layer. This nonlinear function is called the activation function.

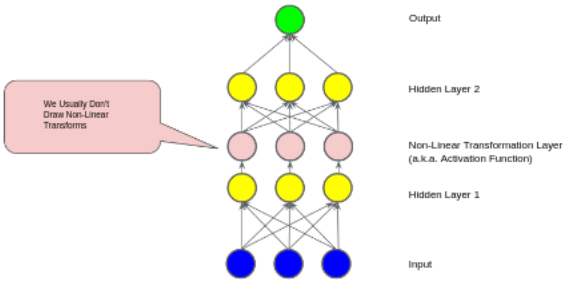


Figure 21: Three-layer model with activation function

In this way we add more impact as stacking nonlinearities lets us model more complicated relationships.

14.3 Common Activation Functions

The sigmoid function converts the weighted sum to a value between 0 and 1.

$$F(x) = \frac{1}{1+e^{-x}}$$

The **rectified linear unit** activation function (ReLU) works slightly better than the sigmoid "smooth" function.

$$F(x) = max(0, x)$$

However, any mathematical function can be an activation function.

$$\text{value of a node in the network} = \sigma(w \cdot x + b)$$

Where σ represents our activation function e.g. sigmoid or ReLU.

15 Best Practices when training neural networks

15.1 Failure Cases

There are numerous ways backpropagation can go wrong.

15.1.1 Vanishing Gradients

Gradients for the layers closer to the input can be very small and when working with deep networks, the gradients could vanish towards 0.
The ReLU function can help mitigate this potential issue.

15.1.2 Exploding Gradients

If the weights are very large, the gradient of the lower layers involves products of many large terms. This overall means that the gradents become to large that they begin to converge.
We can use batch normalizations to prevent exploding gradients.

15.1.3 Dead ReLU Units

When the weighted sum for a ReLU falls below 0, the ReLU unit may get stuck. It then outputs 0 activation and thus the gradients become cut off. To avoid this we can lower the learning rate.

15.2 Dropout Regularization

Another type of regularization is Dropout regularization. It works by randomly dropping unit activations in a network for a single gradient step. The more you drop, the stronger the regularization. 0.0 = No dropout regularization and 1.0 = Drop out everything which means model learns nothing.