# Deciding on Machine Learning

## Start clear and simple

At this point, the statement can be qualitative, but it must capture the real goal of the project, not an indirect goal.

Complete the following:

* “I want my ML model to …”

## The ideal outcome

Don't limit yourself to metrics for which your product has already been optimizing. Instead, try to focus on the larger objective of your product or service. Adding the ML model to your system should produce a desirable outcome.

Complete the following:

* “The ideal outcome of implementing my ML model is …”

## Success and failure metrics

Both success and failure metrics must be considered for understanding the actual impact of the model’s implementation. Success metrics represent the successful consequences of the model’s implementation, while failure metrics help you understand that the model has failed and/or is not good enough. Both success and failure metrics must be written independently of the model’s evaluation metrics (precision, recall, AUC, etc.).

These are some tips for writing the metrics:

* Are the metrics measurable?
* How will they be measured? (live experiments are accepted; many success metrics can't be captured offline)
* When will you be able to measure them?
* How long will it take for you to know whether your new ML system is a success or failure?
* Consider engineering and maintenance costs over the long-term.
* Failure may not only be caused simply by non-achievement of a success metric

Complete the following:

* “My success metrics are …”
* “My key results for the success metrics are …”
* “My ML model is considered a failure if …”

## The output

The model’s output must be quantifiable with a definition that machine can produce.

* Are you able to obtain example outputs to use for training data?
* How and from what source?
* Your output examples may need to be engineered, such as turning watch time into a percentile.

If it is difficult to obtain example outputs to use for training, you may need to revisit your answers in previous steps of this exercise. This may lead you to reformulate your problem and goals into ones that will allow you to train a model on your data.

Complete the following:

* “The output of my ML model will be …”
* “The output is defined as …”

## Using the output

When is the model’s output obtained and how is it used in the product it integrates?

* Will it be presented immediately to the user in a UI?
* Will it be consumed by subsequent business logic?
* What latency requirements do you have?

Furthermore, the model’s requirements can impact what information can be used to make predictions. For example:

* The latency of using data from remote services may make them infeasible to use
* Data logs may be generated only once a day (for instance)
* Certain information is not known until it actually happens (such as conversion events)
* The Oracle Test: assume you always had the correct answer. How would you use that in your product?

Complete the following:

* “My ML model’s output will be made …”
* “The output will be used for …”

## Heuristics

Try writing how you would solve the problem without ML. For example, what heuristics or rules you would use. In other words, try writing a solution using hardcoded logic.

Complete the following:

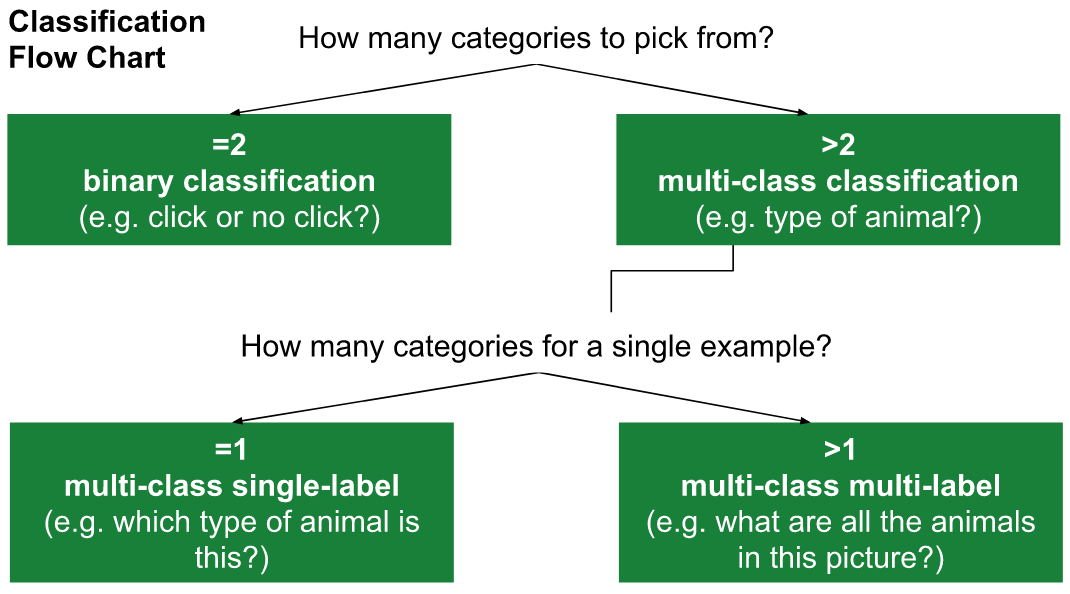
* “If I hardcoded my logic, it would be …”

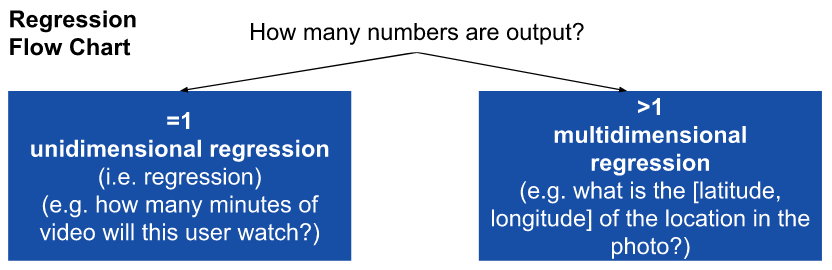
# Formulating a Problem

## Articulate the problem

The first step is to articulate the problem in terms of ML: the type of model and its desired output/prediction.

In other words, choose the type of model you will build:





(among other options)

And the prediction: classify email as spam or not spam, tomorrow’s weather, the price of a house, etc.

## Start simple

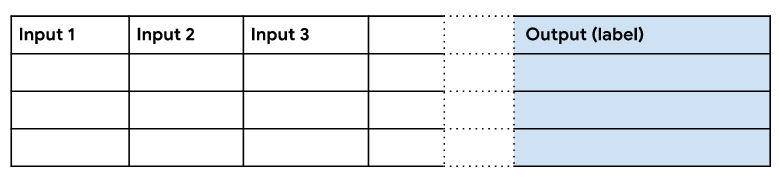
After articulating the problem, try to simplify it in terms of binary classification, unidimensional regression, or both. It may not be possible but that is the point of the exercise: to try to simplify the problem.

## Identify data sources

You won’t be able to build your model without data, so identify where you can get your data from.

## Design the data for the model

Design the data you wish to use to train you model.



For instance, in the case of supervised learning, your data will need to include one or more input features/columns and one column for the output/label/prediction of the model.

Keep in mind that:

* One row constitutes one piece of data for which one prediction is made
* Only include information that is available when the prediction is made
* Each input can be a scalar or 1D list of integer, float, or bytes (including strings)
* If an input has a structure different from a scalar or 1D list, you may wish to consider whether that is the best representation for your data. For example:
  + If a cell represents two or more semantically different things in a 1D list, you may wish to split these into separate inputs
  + If a cell represents a nested protocol buffer, you may wish to flatten out each field of the nested protocol buffer
  + Exceptions: audio, image and video data, where a cell is a blob of bytes

## Determine the cost of obtaining data

At this point you have identified your data sources and designed the data, so now you need to pair each input to the corresponding data source. This way you can assess the difficulty and feasibility of obtaining each input.

## Identify a few valuable and easily obtained inputs

Among the inputs you’ve defined, pick one to three inputs that are easy to obtain and that you believe produce a good enough initial result.

## Evaluate the model’s ability to learn

Think about the ability of your ML model to learn. Your data might suffer from the following problems:

* It doesn’t include enough positive labels
* The training data doesn’t include enough examples
* The labels are too noisy
* The system overfits to the training data

## Consider bias in the data

Many dataset are biased in some way, which can impact the model in negative ways. For instance:

* Biased data may not translate across multiple contexts
* Training data may not be representative of the model’s end-users, hence it may produce inaccurate results

# Data Preparation and Feature Engineering

## Steps to constructing a dataset

To construct a dataset for a machine learning project, and before data transformations, it is needed to:

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

## The quality of a dataset

While collecting data, it is important have a fairly concrete definition of quality. Certain aspects of quality tend to correspond to better-performing models:

* Reliability
* Feature representation
* Minimization of skewness

### Reliability

Reliability refers to the degree of which you can trust the data. In measuring reliability, you must determine:

* How common are label errors?
* Are the features noisy?
* Is the properly filtered for the project?

On the other hand, data can be unreliable due to:

* Omitted values
* Duplicate examples
* Bad labels
* Bad feature values

### Feature representation

Feature representation refers to the mapping of data to useful features. Thus you’ll need to consider:

* How is data shown in the model?
* Should numeric values be normalized?
* How to handle outliers?

### Data skewness

In addition to existing skewness in the original dataset, there can also be skewness between the training data and the live data, that is, between training the model offline and implementing it in a live/online scenario.

Always consider what data is available to your model at prediction time. During training:

* Use only the features you’ll have available online
* Make sure your training set is representative of the live data

## Joining data

When assembling a training set, sometimes you need to join data from multiple sources.

So, you might work with multiple types of logs:

* Transactional logs
* Attribute data
* Aggregate statistics

### Transactional logs

Data logs that record specific events. For example, a transactional log might record the IP address of the user making a query at a specific date and time.

### Attribute data

Attribute data contains snapshots of information, such as user demographics or search history at the time of a query.

Attribute data isn’t specific to an event or moment in time but can still be used for making predictions. For prediction tasks not bound to a specific event, this type of data might be the only appropriate type to use.

With that said, attribute data and transactional logs are related. For example, you can create a type of attribute data by aggregating several logs, creating aggregate statistics. In this case, you can look at many transactional logs to create a single attribute for a user.

### Aggregate statistics

Aggregate statistics create an attribute from multiple transactional logs, such as the frequency of user clicks.

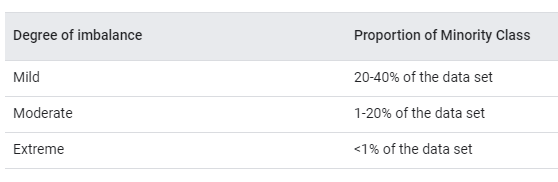
## Direct vs Derived labels

Machine Learning is easier with well-defined labels. For instance, when training a model to predict whether a user is a fan of Dark Souls, a direct label stating “yes” or “no” would be best.

On the other hand, a simple test to find this answer could be whether the person has watched a video about Dark Souls on YouTube. If they have, then they are considered a fan, otherwise they are not. In this case, this is a derived label because it does not directly measure what we want to predict.

## Imbalanced data

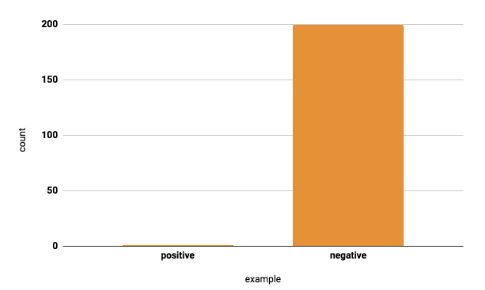
A classification dataset with skewed class proportions is called imbalanced. Classes that make up a large proportion of the dataset are called majority classes. Those that make up a smaller proportion are the minority classes.



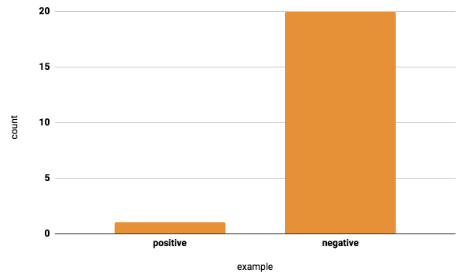
If you have an imbalanced dataset, start by trying to train on the true distribution. If the model works well and generalize, that’s it. If not, try downsampling and upweighting.

* Downsampling: training the model in a disproportionately low subset of the majority class examples
* Upweighting: adding an example weight to the downsampled class equal to the factor by which you downsampled

Consider a data set where you have 200 of examples of the negative class for each example of the positive class.



You can start by downsampling the negative class so that you have only 20 examples of the negative class for each example of the positive class



And then you can upweight, that is, add weight to examples of the downsampled class. This weight should be the same as the factor by which the examples were downsized.

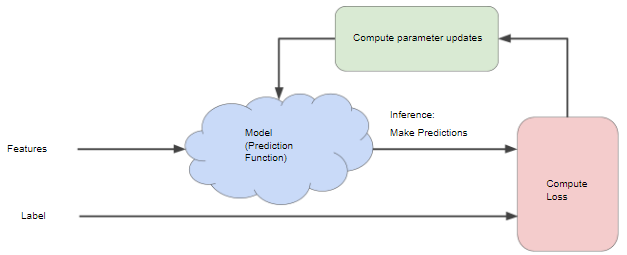
With this example weight, individual examples will be treated as more important during training.

Thanks to the use of both downsampling and upweighting, the following advantages are realized:

* Faster convergence: the minority class is seen more often during training, helping the model to converge faster
* Disk space: because the majority class is consolidated into fewer examples, it requires less disk space to be stored
* Calibration: upweighting ensures the model is still calibrated and its output can still be interpreted as probabilities

# Reducing Loss

## An Iterative Approach



The model receives features as input and returns a prediction (*y'*) as the output, that is, the model makes prediction based on the on the input features. In the example of single-feature models

What initial values should we set for *b* and *w1*? For linear regression problems, it turns out that the starting values aren't important. We can pick random values such as zero for each parameter. If we feed a feature of 10 to the model, then we end up with:

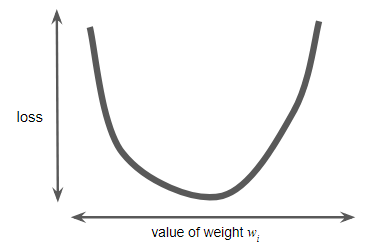
The “Compute Loss” part of the diagram refers to the loss function used by the model. For example, if it uses the squared loss function then it will take two input values to compute the loss: the model’s prediction (*y’*) for features *x* and the correct label (*y*) for the same features.

Advancing to the “Compute Parameters Updates” part, the model then examines the loss values and generates new values for *b* and *w1*. The model iterates this process until overall loss stops changing or at least changes extremely slowly. When it reaches that point, it is said the model has converged.

In short, a Machine Learning model is trained by starting with an initial guess for the weights and bias. The model then iteratively adjusts those values, learning the weights and bias with the lowest possible loss.

## Gradient Descent

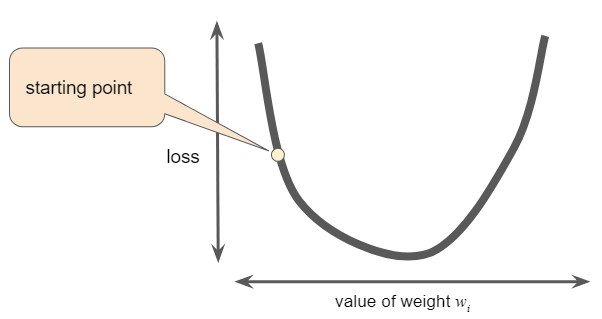
Let’s take another linear regression problem as an example to compute the loss for all possible *w1* (weight) values. If we now plot a loss vs weight graph, we’d end up with a convex figure, like this



Convex problems have only one minimum, that is, only one point where the slope is exactly 0. That minimum is where the loss function converges.

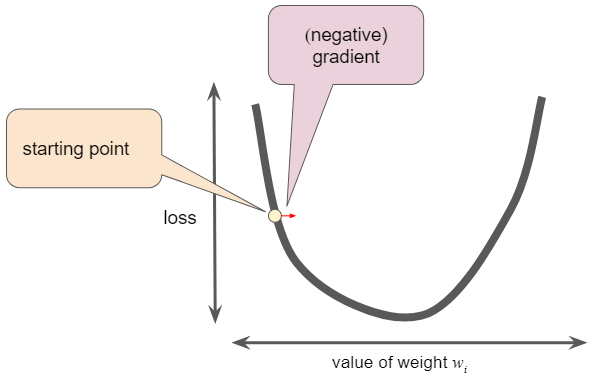
Calculating the loss function for every value of *w1*over the entire data set would be an inefficient way of doing things. Instead, we can use Gradient Descent.

The first stage in gradient descent is to pick a starting value for *w1*. The starting point doesn't matter much, therefore many algorithms simply set *w1* to zero or pick a random value. The following figure shows a starting value/a *w1* greater than zero.



The algorithm then calculates the gradient of the loss curve at the starting point. In the case of the example above with a single weight, the gradient of the loss is equal to the slope or derivative of the curve at each point. However, when there are multiple weights, the gradient is a vector of partial derivatives with respect to the weights. Lastly, don’t forget that a gradient is a vector, so it has both a direction and a magnitude.

At any rate, the objective is to minimize the loss, thus we try to do it by following the negative gradients. In the context of the previous example, this means following points with a negative slope or derivative. Thus, the function takes a step in the direction of the negative gradient to reduce loss.



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. The gradient descent then repeats this process, edging ever closer to the minimum.

## Learning Rate

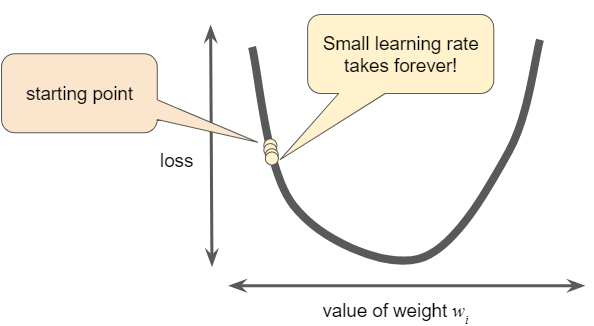
Gradient descent algorithms multiply the gradient by a scalar known as the Learning Rate or Step Size to determine the next point. For example, if the gradient magnitude is 2.5 and the learning rate is 0.01, then the gradient descent algorithm will pick the next point 0.025 away from the previous point.

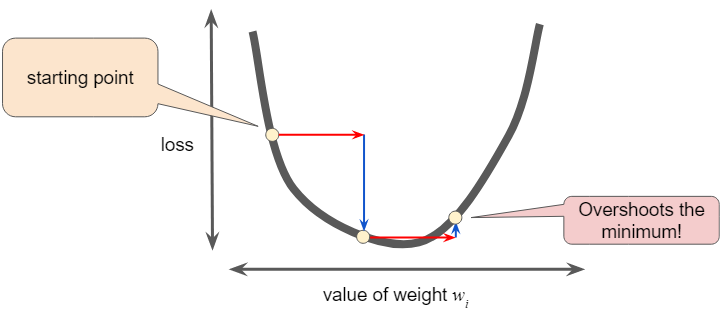
If you pick a learning rate that is too small, learning will take too long. However, if you pick a learning rate too large, the model might pass the loss function’s minimum. If you know the gradient of the loss function is small, then you can safely try a larger learning rate, which compensates for the small gradient and results in a larger step size.

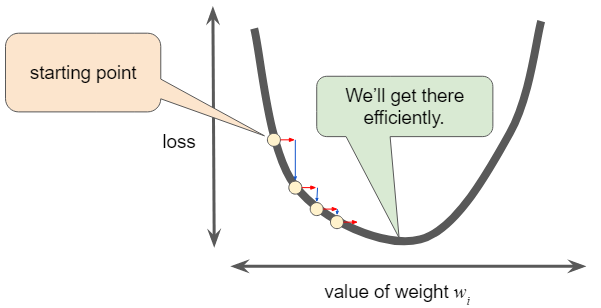
The ideal learning rate in one-dimension is the inverse of the second derivative of *f(x)* at *x*.

The ideal learning rate for two or more dimensions is the inverse of the Hessian (matrix of second partial derivatives).

In practice, finding a "perfect" (or near-perfect) learning rate is not essential for successful model training. The goal is to find a learning rate large enough that gradient descent converges efficiently, but not so large that it never converges.







## Stochastic Gradient Descent

In gradient descent, a batch is the total number of examples you use to calculate the gradient in a single iteration.

A large dataset 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 to not carry much more predictive value than large batches.

Stochastic Gradient Descent (SGD) uses a single (random) example, that is, a batch size of one, per iteration. Given enough iterations, SGD works, but is very noisy.

Mini-batchStochastic 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. This way, mini-batch SGD reduces the amount of noise in SGD but still manages to be more efficient than full-batch.

## Is There a Standard Heuristic for Model Tuning?

The short answer is that the effects of different hyperparameters are data dependent. Thus, there are no hard-and-fast rules, you'll need to test your data.

However, there are still a few rules of thumb that may help:

* Training error should steadily decrease, steeply at first, and should eventually plateau as training converges to zero
* If the training has not converged, train for more epochs
* If the training loss decreases too slowly, increase the learning rate, though a learning rate too high can prevent training loss from converging
* If the training loss varies wildly, decrease the learning rate
* Lowering the learning rate while increasing the number of epochs or the batch size is often a good combination
* Very small batch sizes can also cause instability; start with larger batch sizes and decrease until you see degradation
* The entire dataset might not fit into memory, in which case you'll need to reduce the batch size to enable a batch to fit into memory

Again, never go strictly by these rules of thumb because the effects are data dependent. Always experiment and verify.

# Hyperparameters

Hyperparameters are the elements tweaked during successive runs of training of a Machine Learning model.

Some of the most common hyperparameters include the learning rate, the steps and the batch size.

## Batch Size

Batch Size is the number of examples (chosen at random) for a single step.

For example, the batch size for Stochastic Gradient Descent is 1.

## Learning Rate

A scalar used to train a model via Gradient Descent.

For each iteration, the Gradient Descent algorithm multiplies the learning rate by the Gradient, with the resulting product being called the Gradient Step.

## Steps

Steps are the total number of training iterations.

One step calculates the loss from one batch and uses that value to modify the model's weights once.

# The Risk of Overfitting

Overfitting is caused by making a model more complex than necessary. An overfit model gets a low loss during training but does a poor job predicting new data.

The fundamental tension of Machine Learning is between fitting our data well, but also fitting the data as simply as possible.

However, if a model fits the current sample well, how much can it be trusted to make good predictions on new data? Machine learning's goal is to predict well on new data drawn from a (hidden) true probability distribution. Unfortunately, the model can only sample from a training dataset.

A Machine Learning model aims to make good predictions on new, previously unseen data. But, if you are building a model from your dataset, how would you get the previously unseen data? One way is to divide the dataset 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
* The same test set is not used repeatedly

The following three basic assumptions guide generalization:

* Draw examples independently and identically (i.i.d), at random, from the distribution. In other words, examples don't influence each other
* The distribution is stationary, that is, the distribution doesn't change within the dataset
* Examples are drawn from partitions of the same distribution

In practice, these assumptions are sometimes not followed. For example:

* Consider a model that chooses ads to display. The i.i.d. assumption would be violated if the model bases its choice of ads, in part, on what ads the user has previously seen
* Consider a dataset that contains retail sales information for a year. User's purchases change seasonally, which would violate stationarity

When we know that any of the preceding three basic assumptions are violated, we must pay careful attention to metrics.

# Splitting Data

When training a model in Machine Learning, there’s the necessity of splitting the given dataset into two separate sets: a training set, used to train the model, and a test set, used to test the trained model.

However, the test set must meet the following two conditions:

* It is large enough to yield statistically meaningful results
* It is representative of the dataset, that is, the test set should have identical characteristics to the training set

Assuming the test set meets the preceding two conditions, the goal is to create a model that generalizes well to new data, that is, the test set serves as a proxy for new data.

Keep in mind to never train on test data. The evaluation metrics showing surprisingly good results might be a sign that you are accidentally training on the test set. High accuracy might indicate that test data has leaked into the training set, for example, due to duplicate examples across both sets.

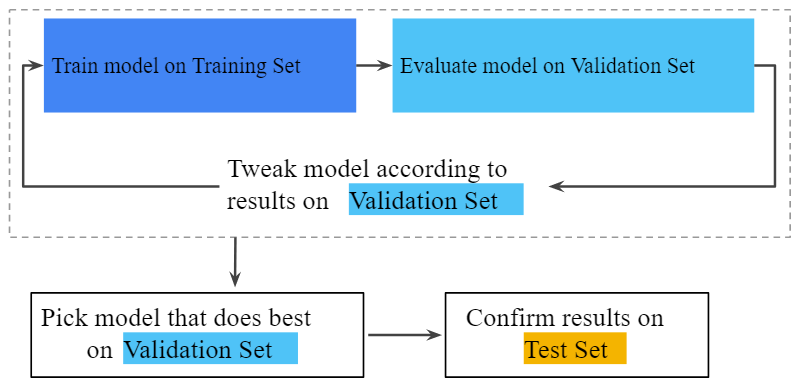
## Another Set

Dividing the dataset into two sets is a good idea, but not a surefire solution. You can greatly reduce the chances of overfitting by partitioning the dataset into three, instead of two, subsets:

* Training set
* Validation set
* Test set

The validation set is used to evaluate results from the training set, namely tweak hyperparameters if needed. Then, the test set can be used to double-check the evaluation after the model has "passed" the validation set.

This way, the model’s performance can be double checked against the test set, after being vetted by the validation set and deemed successful. It also helps in exposing the model less to the test set.

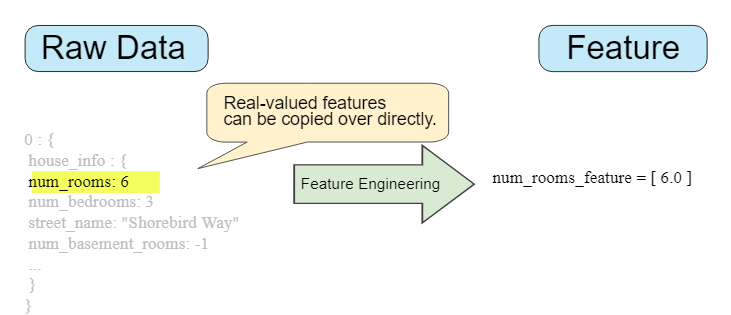


# Data Representation

## Feature Engineering

In a Machine Learning model, raw input data must be feature engineered, that is, transformed into numeric values.

Integer and floating-point data don't need a special encoding because they can be multiplied by a numeric weight. For example, a feature that has a value of *6* can be easily feature engineered as *6.0*.



However, in the case of categorical features, these have a discrete set of possible values. Since models cannot multiply strings by the learned weights, we use feature engineering to convert strings to numeric values. This can be accomplished by defining a mapping from the feature values to integers, that is, the vocabulary of possible values. In the context of the vocabulary, we can group all values that are not defined into a catch-all category, known as an out-of-vocabulary (OOV) bucket.

For instance, if we have a set of street names such as:

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

Then we can easily map the streets to numbers:

* Charleston Road -> 0
* North Shoreline Boulevard -> 1
* Shorebird Way -> 2
* Rengstorff Avenue -> 3
* Everything else (OOV) -> 4

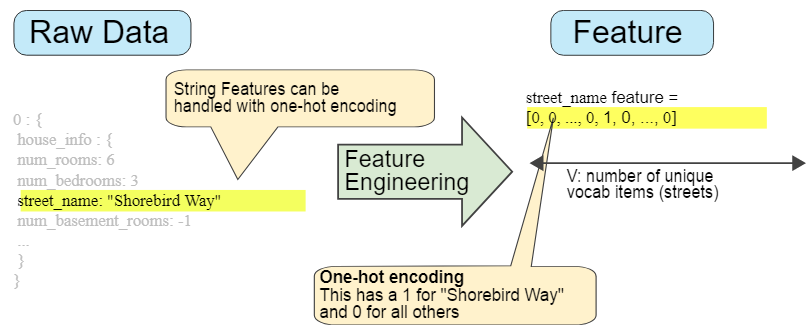
Still, if we try to use these indexing values in the model, it will create two problems:

* The model will learn a single weight for the feature (e.g. for the street names example above, the model will multiply the learned weight by the mapped numeric value of each street)
* This mapping doesn’t take into account situations where a feature may take multiple values (e.g. a house may be located at the corner of two streets)

To correct these problems, we can instead create a binary vector. This is a vector that contains only True/1 or False/0 values for each categorical feature used in the model:

* True/1: vector elements that correspond to values that apply to the example
* False/0: vector elements that do not correspond to values that apply to the example

This vector or tensor will be as big as the vocabulary. The vector is called a one-shot encoding if it contains a single element True/1 value; if it contains multiple True/1 values then it’s called a multi-shot encoding. Regardless of the type of vector, the model will use the weights for the elements with a value of True/1.



In the example of the figure above, the one-hot encoded vector will store a value of *1* in the place of “Shorebird Way” and a value of *0* for all other streets because that’s the name of the street from the raw data. If the house was located in two streets, then the vector would have two elements with a value of *1*, thus it would be multi-shot encoding instead.

If the vector ends up growing large in size and only few elements have a value of True/1, then there’s the alternative sparse representation. Instead of including all the possible values and their corresponding weights saved in vector, this representation will show only the values with positive weights, that is, it will contain just the elements of the vector that have a value of True/1, that is, nonzero values. Due note that an independent model weight is still learned for each feature value in sparse representations.

## Choosing Good Features

Not all feature values make up for good features to train a model on. Instead, we should take into account some general rules to judge whether given feature values will be good features:

* Avoid rarely used discrete feature values
  + Type of house would be a good feature
  + A house id would only have unique values
* Prefer clear and obvious meanings
  + For instance, a column of house age in years that contains values in a reasonable range
* Don’t mix “magic” values with actual data
  + For instance, if a rating is normalized to the 0-1 range, don’t use values such as -1 to indicate a missing data point; instead create a Boolean feature that specified is the user gave a rating or not
* The definition of a feature shouldn’t change over time
  + The name of a city is presumably a static value

## Cleaning Data

More often than not, data won’t be completely clean and ready to be used in your model. Instead, you’ll have to spend some time cleaning your data, namely taking care of outliers. For this purpose, there are some handy mechanisms:

* Scaling: scale the values of a feature into a single range (the most usual ones are *-1* to *+1* and *-3* to *+3*)
  + Z score:
  + Helps gradient descent converge more quickly
  + Helps the model learn appropriate weights for each feature
  + Helps avoid the "NaN trap," in which one number in the model becomes a NaN, and eventually every other number in the model becomes a NaN due to mathematical operations
* Clipping: values that were greater/smaller than a given value now become that value, namely clipping outliers to the upper and/or lower bounds of a range of values
* Binning: bucketing data so that we can represent values with one or multi-shot encoded vectors, allowing the model to learn different weights for each bucket/bin
* Scrubbing: general term for solving some more specific problems, including omitted values, duplicate examples, bad labels and bad feature values
* Knowing your data:
  + Keep in mind what you think the data should look like
  + Verify that the data meets the expectations mentioned
  + Double-check the veracity of the data

# Feature Crosses

To solve a nonlinear problem, we can use feature crosses. These are synthetic features that encode nonlinearity in the feature space by multiplying two or more input features.

For example, given the features *x1* and *x2*, we can create a feature cross *x3* as:

And so the new *x3* feature can be treated as any other:

A linear algorithm can learn a weight for *x3* just as it would for *x1* and *x2*, that is, while *x3* encodes nonlinear information, it’s not needed to change how the model is trained to determine the value of *w3*.

However, a feature cross doesn’t necessarily need to be created by multiplying just two other features. The following are some examples of how to create a feature cross:

* [A x B]: created as the cross product of two features (what was shown previously)
* [A x B x C x D x E]: created using the cross product of five features
* [A x A]: created by squaring a single feature

Supplementing scaled linear models with feature crosses has traditionally been an efficient way to train on massive-scale datasets.

## Crossing One-Hot Vectors

In practice, Machine Learning models rarely cross continuous features. However, they do frequently cross one-hot feature vectors. Think of feature crosses of one-hot feature vectors as logical conjunctions.

For example, suppose you bin latitude and longitude, producing separate one-hot five-element feature vectors. Then, a given latitude and longitude could be represented as:

*binned\_latitude = [0, 0, 0, 1, 0]*

*binned\_longitude = [0, 1, 0, 0, 0]*

If we were to create a feature cross based on these two features, the resulting feature would be a 25-element one-hot vector (24 zeroes and 1 one). The single 1 in the cross identifies a particular conjunction of latitude and longitude. Your model can then learn particular associations about that conjunction.

If we simplify the above example and represent it in another way, such as this:

*binned\_latitude(lat) = [*

*0 < lat <= 10*

*10 < lat <= 20*

*20 < lat <= 30*

*]*

*binned\_longitude(lon) = [*

*0 < lon <= 15*

*15 < lon <= 30*

*]*

Creating a feature cross of the above creates a feature that contains the following values:

*binned\_latitude\_X\_longitude(lat, lon) = [*

*0 < lat <= 10 AND 0 < lon <= 15*

*0 < lat <= 10 AND 15 < lon <= 30*

*10 < lat <= 20 AND 0 < lon <= 15*

*10 < lat <= 20 AND 15 < lon <= 30*

*20 < lat <= 30 AND 0 < lon <= 15*

*20 < lat <= 30 AND 15 < lon <= 30*

*]*

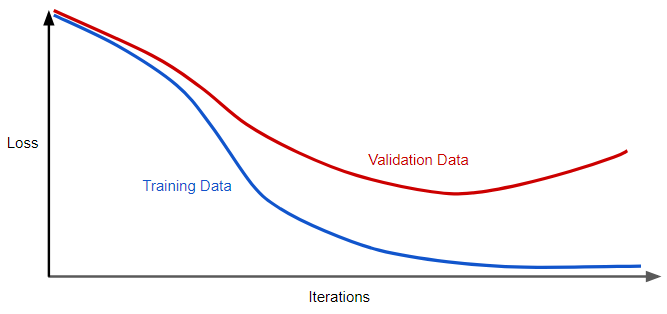
Let us consider another, such as dogs. Here we have a feature for the behavior type and another for time of the day. If we create a feature cross *[behavior type X time of day]*, then the model will end up with much more predictive ability than either feature on its own. If a dog cries (happily) at 5:00 pm when the owner returns from work, it will likely be a great positive predictor of owner satisfaction. Crying (miserably) at 3:00 am when the owner was sleeping soundly will likely be a strong negative predictor of owner satisfaction.

# Regularization

Regularization is the penalty applied on a model's complexity. Regularization helps prevent overfitting.

## L2 Regularization

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



The figure shows a model in which training loss gradually decreases, but validation loss eventually goes up. In other words, this generalization curve shows a model that is overfitting to the data in the training set. This could be prevented by penalizing complex models, a principle called regularization.

Instead of simply aiming to minimize loss (empirical risk minimization):

we'll now minimize loss and complexity, that is, structural risk minimization:

The training optimization algorithm is now a function of two terms:

* The loss term, which measures how well the model fits the data
* The regularization term, which measures model complexity

Model complexity is commonly thought of in two ways:

* 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

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.

Complexity can be quantified using the L2 Regularization formula, which defines the regularization term as the sum of the squares of all the feature weights:

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

For example, a linear model with the following weights:

*{w1=0.2, w2=0.5, w3=5, w4=1, w5=0.25, w6=0.75}*

Has an L2 Regularization term of 26.915:

*w3*, with a squared or complexity value of *25* is responsible for almost all of the complexity of the model.

## Lambda

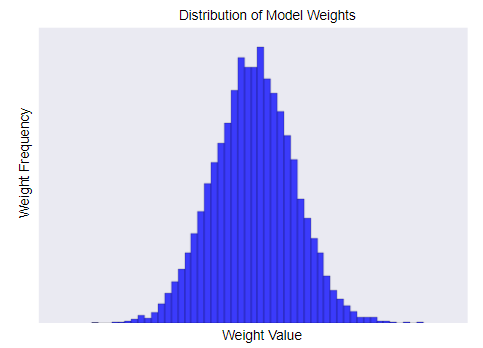
Model developers tune the overall impact of the regularization term by multiplying its value by a scalar known as lambda or regularization rate, that is, model developers aim to do the following:

Performing L2 Regularization has the following effects on a model:

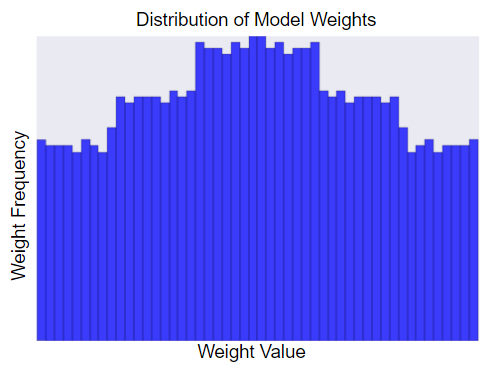
* Encourages weight values toward 0 (but not exactly 0)
* Encourages the mean of the weights toward 0, with a normal (bell-shaped or Gaussian) distribution

Increasing the lambda value strengthens the regularization effect.

The following figure is the histogram of weights for a high value of Lambda:



The following is the histogram of weights for a low value of Lambda:



When choosing a 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 will be simple, but you run the risk of underfitting your data. Your model won't learn enough about the training data to make useful predictions
* If the lambda value is too low, the model will be more complex, and you run the risk of overfitting your data. Your model will learn too much about the particularities of the training data and won't be able to generalize to new data
* 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 it requires fine-tuning.

## L2 Regularization and Learning Rate

There's a close connection between learning rate and lambda. Strong L2 Regularization values tend to drive feature weights closer to zero. Lower learning rates (with early stopping) often produce the same effect because the steps away from zero 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.

As noted, the effects from changes to regularization parameters can be confused 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 so that early stopping doesn't play into things.

# Logistic Regression

## Calculating a Probability

Many problems require a probability estimate as output. Logistic regression is an extremely efficient mechanism for calculating such 3probabilities.

Practically speaking, the returned probability can be used in either of two ways:

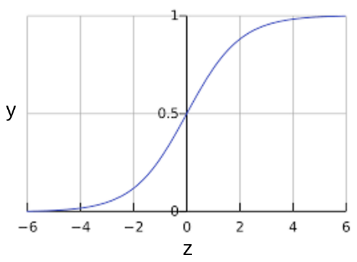
* "As is"
* Converted to a binary category

Let's consider how 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:

If the logistic regression model predicts a probability of 0.05, then, over a year, the dog should bark approximately 18 times:

In many cases, you'll map the logistic regression output into the solution to a binary classification problem, in which the goal is to correctly predict one of two possible labels (e.g., "spam" or "not spam").

A sigmoid function, produces an output between 0 and 1, ensuring the logistic regression model’s output always falls in this range, and yields a plot of this kind:



*z* represents the output of the linear layer of a model trained with logistic regression, that is, *sigmoid(z)* yields a value/probability between 0 and 1. In mathematical terms:

, where:

* *y’* is the output of the Logistic Regression model
* *z* corresponds to
  + The *b* value is the model’s bias
  + The *w* values are the model’s learned weights
  + The *x* values are the model’s feature values

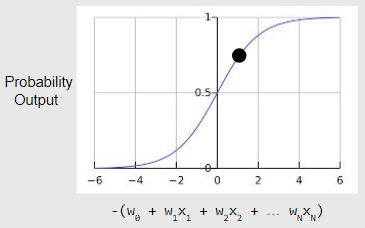
Please note that *z* is also referred to as the *log-odds* because the inverse of the sigmoid states that *z* can be defined as the log of the probability of the "1" label (e.g., "dog barks") divided by the probability of the "0" label (e.g., "dog doesn't bark"):

For example, given a logistic regression model with three features *x1*, *x2* and *x3* and the following learned bias and weights:

* *x1 = 0*
* *x2 = 10*
* *x3 = 2*
* *w1 = 2*
* *w2 = -1*
* *w3 = 5*
* *b = 1*

The model’s log odds or *z* value for this example would be:

Thus, the Logistic Regression prediction for this particular example will be 0.731:



## Log Loss

While the loss function for linear regression is squared loss, logistic regression uses log loss, which is mathematically defined as:

where:

* is the dataset containing labeled *(x, y)* examples
* *y* is the label in the labeled example. Given that this is logistic regression, every value of *y* must be either *0* or *1*
* *y’* is the predicted value in the range of 0 to 1, given the set of features in *x*

## Regularization in Logistic Regression

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

* L2 Regularization
* Early Stopping
* L1 Regularization

# Classification

## Thresholding

Logistic regression returns a probability and then that probability can be used "as is", for example, as the numerical probability that a user will click on a link. The returned probability can also be converted to a binary value, such as in a “spam” or “not spam” example.

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. However, what about an email message with a prediction score of 0.6?

To map a logistic regression value to a binary category, a classification threshold must be defined, also called the decision threshold. A value above that threshold indicates "spam"; a value below indicates "not spam." It is tempting to assume that the classification threshold should always be 0.5, but thresholds are problem-dependent and so must be manually tuned.

Please note that tuning a threshold for Logistic Regression is different from tuning hyperparameters such as the learning rate. Part of choosing a threshold is assessing how much you'll suffer for making a mistake. For example, mistakenly labeling a non-spam message as spam is very bad. However, mistakenly labeling a spam message as non-spam is unpleasant, but hardly the end of your job.

## True vs. False and Positive vs. Negative

Given a model that predicts whether a patient has a tumor or not, let’s use the following definitions:

* “Tumor” is a positive class
* “Non-tumor” is a negative class

The tumor prediction can be summarized using a confusion matrix, depicting all four possible scenarios, including some sample predictions:

|  |  |  |
| --- | --- | --- |
|  | Tumor (predicted) | Non-tumor (predicted) |
| Tumor (positive class) | 18 | 1 |
| Non-tumor (negative class) | 6 | 452 |

The confusion matrix for this example shows that, of the 19 (18 + 1) samples that actually had tumors, the model correctly classified 18 as having tumors (18 true positives), and incorrectly classified 1 as not having a tumor (1 false negative). Similarly, of 458 (6 + 452) samples that did not have tumors, 452 were correctly classified (452 true negatives) and 6 were incorrectly classified (6 false positives).

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. Lastly, a false negative is an outcome where the model incorrectly predicts the negative class.

## Accuracy

Accuracy is a metric for evaluating classification models. Accuracy is the fraction of predictions the model got right, defined as:

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

Where:

* TP: True Positives
* TN: True Negatives
* FP: False Positives
* FN: False Negatives

For example, recovering the sample classification matrix for the classification model that predicted whether a patient had a tumor:

|  |  |  |
| --- | --- | --- |
|  | Tumor (predicted) | Non-tumor (predicted) |
| Tumor (positive class) | 18 | 1 |
| Non-tumor (negative class) | 6 | 452 |

We can calculate its accuracy as:

Thus, the model had an accuracy of 0.985 or 98.5% for the given sample.

However, accuracy alone doesn't tell the full story when working with a class-imbalanced dataset, where there is a significant disparity between the number of positive and negative labels.

## Precision and Recall

Precision measures the percentage of examples correctly predicted with the positive class (e.g., percentage of emails classified as spam that were actually spam). On the other hand, recall measures the percentage of positive examples that were correctly identified by the model (e.g., actual spam emails that were correctly identified).

If we recover again the sample about a model that predicted whether a person had a tumor or not:

|  |  |  |
| --- | --- | --- |
|  | Tumor (predicted) | Non-tumor (predicted) |
| Tumor (positive class) | 18 | 1 |
| Non-tumor (negative class) | 6 | 452 |

, the sample has the following positive and negative values:

* True Positive (TP): 18
* False Positive (FP): 1
* True Negative (TN): 452
* False Negative (FN): 6

And so we can calculate the precision of the model, that is, the proportion of positive detections that are actually positive, using its formula:

And the recall, that is, the proportion of actual positives detected correctly, using its formula as well:

To fully evaluate the effectiveness of a model, you must examine both precision and recall. Unfortunately, these two are often in tension, that is, improving precision typically reduces recall and vice versa. In general, raising the classification threshold reduces false positives, thus raising precision. On the other hand, raising the classification causes the number of true positives to decrease or stay the same and will cause the number of false negatives to increase or stay the same, thus, making recall stay at the same value or decrease.

## ROC and AUC

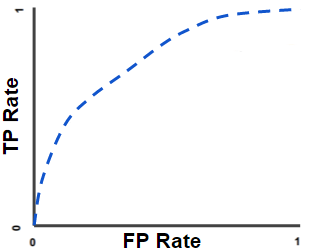
A ROC curve (Receiver Operating Characteristic curve) is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters:

* True positive rate
* False positive rate

The true positive rate (TPR) is mathematically defined as:

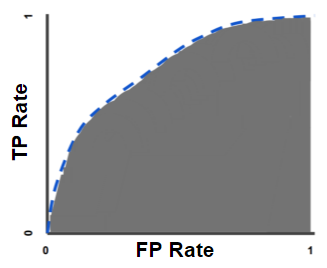
The false positive rate (FPR) is mathematically defined as:

Lowering the classification threshold classifies more items as positive, thus increasing both false positives and true positives. The following figure shows a typical ROC curve:



To compute the points in a ROC curve, we could evaluate a logistic regression model many times with different classification thresholds, but this would be inefficient. Fortunately, there's an efficient, sorting-based algorithm that can provide this information for us, AUC.

AUC (Area Under the ROC Curve) measures the entire two-dimensional area underneath the entire ROC curve from (0,0) to (1,1):



AUC provides an aggregate measure of performance across all possible classification thresholds. One way of interpreting AUC is to interpret it as the probability that the model ranks a random positive example more highly than a random negative example.

AUC ranges in value from 0 to 1. A model whose predictions are 100% wrong has an AUC of 0.0; one whose predictions are 100% correct has an AUC of 1.0.

AUC is desirable for the following two reasons:

* AUC is scale-invariant, that is, it measures how well predictions are ranked, rather than their absolute values
* AUC is classification-threshold-invariant, that is, it measures the quality of the model's predictions irrespective of what classification threshold is chosen

However, both these reasons come with caveats, which may limit the usefulness of AUC in certain use cases:

* Scale invariance is not always desirable. For example, sometimes we really do need well calibrated probability outputs, and AUC won’t tell us about that
* Classification-threshold invariance is not always desirable. In cases where there are wide disparities in the cost of false negatives vs. false positives, it may be critical to minimize one type of classification error. For example, when doing email spam detection, you likely want to prioritize minimizing false positives (even if that results in a significant increase of false negatives). AUC isn't a useful metric for this type of optimization

## Prediction Bias

Logistic regression predictions should be unbiased, that is, the average value of predictions should be approximate to the average value of observations.

Prediction bias is a quantity that measures how far apart those two averages are, which in mathematical terms translates to:

A significant nonzero prediction bias tells you there is a bug somewhere in your model, as it indicates that the model is wrong about how frequently positive labels occur.

For example, let's say 1% of all emails are spam. If we don't know anything at all about a given email, we should predict that it's 1% likely to be spam. Similarly, a good spam classification model should predict, on average, that emails are 1% likely to be spam. If, instead, the model's average prediction of an email being spam is 20%, we can conclude that it exhibits prediction bias.

Possible root causes of prediction bias are:

* Incomplete feature set
* Noisy dataset
* Buggy pipeline
* Biased training sample
* Overly strong regularization

You might be tempted to correct prediction bias by post-processing the learned model with an additional calibration layerthat adjusts the model’s output to reduce the prediction bias. For example, if your model has +3% bias, you could add a calibration layer that lowers the mean prediction by 3%. However, adding a calibration layer is a bad idea for the following reasons:

* You're fixing the symptom rather than the cause
* You've built a more brittle system that you must now keep up to date

If possible, calibration layers should be avoided.

## Bucketing and Prediction Bias

Logistic regression predicts a value between 0 and 1. However, all labeled examples are either exactly 0 (meaning, for example, "not spam") or exactly 1 (meaning, for example, "spam"). Therefore, prediction bias cannot be accurately determined based on a single example, instead, you should examine the prediction bias on a bucket of examples.

Prediction bias for logistic regression only makes sense when grouping enough examples together to be able to compare a predicted value (for example, 0.392) to observed values (for example, 0.394).

Buckets can be formed in the following ways:

* Linearly breaking up the target predictions
* Forming quantiles

## L1 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 extraordinary 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.

For example, consider a housing dataset that covers not just California but the entire globe. Bucketing global latitude at the minute level (60 minutes per degree) gives about 10,000 dimensions in a sparse encoding. Bucketing global longitude at the minute level gives about 20,000 dimensions. A feature cross of these two features would result in roughly 200,000,000 dimensions. Many of those represent areas of such limited residence (for example, the middle of the ocean) that it would be difficult to use that data to generalize effectively. Therefore, it would be nice to encourage the weights for the meaningless dimensions to drop to exactly 0, which would allow us to avoid paying for the storage cost of these model coefficients at inference time. This idea can be incorporated into the optimization problem done at training time by adding an appropriately chosen regularization term.

Would L2 Regularization accomplish this task? Unfortunately, not. L2 Regularization encourages weights to be small but doesn't force them to exactly 0.

There is a regularization term called L1 Regularization that has the advantage of being convex and thus efficient to compute. So, we can use L1 Regularization to encourage many of the uninformative coefficients in our model to be exactly 0.

## L1 vs L2 Regularization

L2 and L1 penalize weights differently:

* L2 penalizes *weight2* (weight squared)
* L1 penalizes *|weight|* (absolute weight)

Consequently, L2 and L1 have different derivatives:

* The derivative of L2 is *2 \* weight*
* The derivative of L1 is *k* (a constant, whose value is independent of weight)

You can think of the derivative of L2 as a force that removes a percentage of the weight every time. Unfortunately, L2 does not normally drive weights to zero.

You can think of the derivative of L1 as a force that subtracts some constant from the weight every time. However, thanks to absolute values, L1 has a discontinuity at 0, which causes subtraction results that cross 0 to become zeroed out. For example, if subtraction forces a weight from +0.1 to -0.2, L1 will set the weight to exactly 0. Thus, L1 zeroes out the weight.

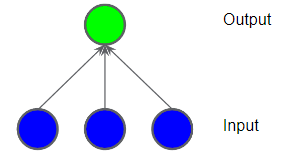
Note that this description is true for a one-dimensional model.

# Neural Networks

## Introduction

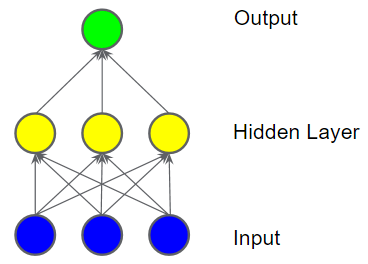
A neural network is a model that is composed of layers (at least one of which is hidden), consisting of simple connected units or neurons followed by nonlinearities.

For example, a linear model of the form could be represented as a neural network in its simplest form in the following way:



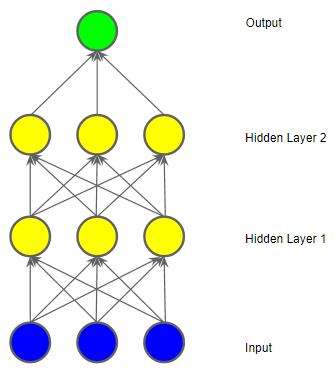
Each blue circle represents an input feature (the input layer), and the green circle represents the weighted sum of the inputs (the output layer).

However, we should add a hidden layer of intermediary values:



Each yellow node in the hidden layer is a weighted sum of the blue input node values. The output is a weighted sum of the yellow nodes.

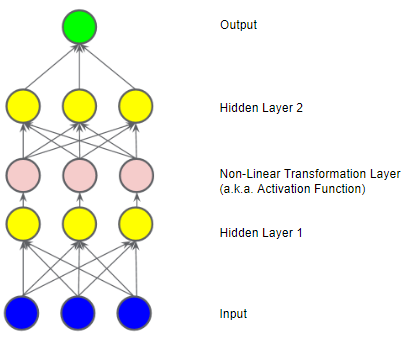
Is this model linear? Yes, since its output is still a linear combination of its inputs. Even if we add a second hidden layer, the model will remain linear because when you express the output as a function of the input and simplify it, you get just another weighted sum of the inputs.



## Activation Functions

To model a nonlinear problem, we can introduce a nonlinearity. We can pipe each hidden layer node through a nonlinear function.

In the following model representation, the value of each node in hidden layer 1 is transformed by a nonlinear function before being passed on to the weighted sums of the next layer. This nonlinear function is called the activation function.



Now that we've added an activation function, adding layers has more impact. Stacking nonlinearities on nonlinearities lets us model complicated relationships between the inputs and the predicted outputs. In other words, each layer is effectively learning a more complex, higher-level function over the raw inputs.

In summary, a neural network is composed by:

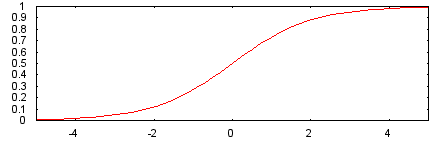
* A set of nodes, analogous to neurons, organized in layers
* A set of weights representing the connections between each layer and the layer beneath it
* A set of biases, one per node
* An activation function that transforms the output of each node in the layer

## Common Activation Functions

There are two main activation functions: Sigmoid and ReLU.

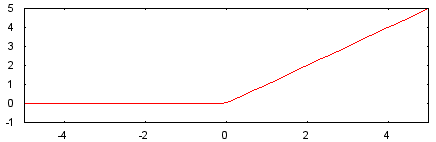
A Sigmoid activation function converts the weighted sum to a value between 0 and 1.

Mathematically, it’s defined as:



A ReLU (Rectified Linear Unit) activation function often works a little better than a smooth function like the Sigmoid, while also being significantly easier to compute. The superiority of ReLU is based on empirical findings, probably driven by ReLU having a more useful range of responsiveness. A Sigmoid's responsiveness falls off relatively quickly on both sides.

Mathematically, it’s defined as:



Actually, any mathematical function can serve as an activation function. Using the (sigma) symbol to represent a given activation function, the value of a node in the network is given by the following formula:

## Training Neural Networks: Failure Cases

When training a neural network, a couple of common problems may arise, such as vanishing gradients, exploding gradients or dead ReLU units.

### Vanishing Gradients

The gradients for the lower layers (closer to the input) can become very small and, as a consequence, these layers train very slowly or not at all.

In deep networks, computing these gradients can involve taking the product of many small terms.

The ReLU activation function can help prevent vanishing gradients.

### Exploding Gradients

If the weights in a network are very large, then the gradients for the lower layers involve products of many large terms. In this case you can have exploding gradients: gradients that get too large to converge.

Batch normalization can help prevent exploding gradients, as can lowering the learning rate.

### Dead ReLU Units

Once the weighted sum for a ReLU unit falls below 0, the ReLU unit can get stuck. It outputs 0 activation, contributing nothing to the network's output, and gradients can no longer flow through it during backpropagation.

With a source of gradients cut off, the input to the ReLU may never change enough to bring the weighted sum back above 0.

Lowering the learning rate can help keep ReLU units from dying.

## Dropout Regularization

Dropout Regularization works by randomly "dropping out" unit activations in a network for a single gradient step. The more you drop out, the stronger the regularization:

* 0.0 -> no dropout regularization
* 1.0 -> drop out everything; the model learns nothing
* Values between 0.0 and 1.0 -> more useful

## Multi-Class Neural Networks

A multi-class classification model is a model that distinguishes between more than two classes (e.g., a model that predicts a dog’s breed). Then, a multi-class neural network is a neural network that works with multi-class classification.

### One vs. All

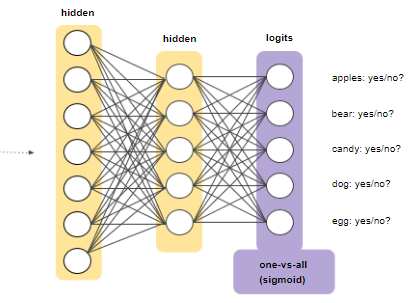
One vs. All provides a way to leverage binary classification. Given a classification problem with N possible solutions, a One vs. All solution consists of N separate binary classifiers: one binary classifier for each possible outcome. During training, the model runs through a sequence of binary classifiers, training each to answer a separate classification question.

For example, given a picture of a dog five different recognizers might be trained, each trained for a different binary classification problem. When presented with the picture of a dog, we’d end up with the following result (in a simplified way):

1. Is this image an apple? No
2. Is this image a bear? No
3. Is this image candy? No
4. Is this image a dog? Yes
5. Is this image an egg? No

This approach is fairly reasonable when the total number of classes is small but becomes increasingly inefficient as the number of classes rises.

We can create a significantly more efficient One vs. All model with a Deep Neural Network in which each output node represents a different class, such as the one in the following example:



### Softmax

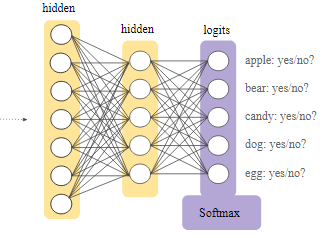
Logistic regression produces a decimal between 0 and 1.0. For example, a logistic regression output of 0.8 from an email classifier suggests an 80% chance of an email being spam and a 20% chance of it being not spam. In other words, the sum of the probabilities adds up to 1.0.

Softmax extends this idea into multi-class problems. Softmax assigns decimal probabilities to each class in a multi-class problem, with its total sum being 1.0. This additional constraint helps training converge more quickly than it otherwise would.

For example, given an image analysis model that classifies an image as an “apple”, as a “bear”, as “candy”, as a “dog” or as an “egg”, given the image of a dog to classify, this could be a possible output:

* apple: 0.001
* bear: 0.04
* candy: 0.008
* dog: 0.95
* egg: 0.001

Softmax is implemented through a neural network layer just before the output layer. This layer must have the same number of nodes as the output layer.



Softmax also has two variants: Full Softmax and Candidate Sampling:

* Full softmax calculates a probability for every possible class (what has been discussed so far)
* Candidate sampling calculates a probability for all the positive labels but only for a random sample of negative labels. For example, if we are interested in determining whether an input image is a beagle or a bloodhound, we don't have to provide probabilities for every example that is not of a dog

Full softmax is fairly cheap when the number of classes is small but becomes prohibitively expensive at the other hand of the spectrum. Candidate sampling can improve efficiency in problems with many classes.

Softmax assumes that each example is a member of exactly one class. Thus, softmax can’t be used in cases where examples belong to multiple classes. In these cases, you must rely on multiple logistic regressions.

# Embeddings

## Introduction

An embedding is a relatively low-dimensional space into which you can translate high-dimensional vectors. Embeddings make it easier to do Machine Learning on large inputs like sparse vectors representing words.

Ideally, an embedding captures some of the semantics of the input by placing semantically similar inputs close together in the embedding space. An embedding can be learned and reused across models.

## Collaborative Filtering

Collaborative filtering is the task of making predictions about the interests of a user based on interests of many other users.

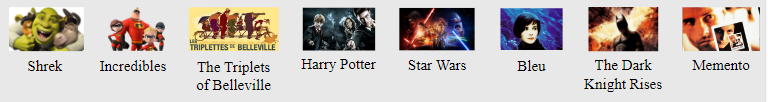
For example, let's look at the task of movie recommendation. Suppose we have one million users and a list of the movies each user has watched, from a catalog of five hundred thousand movies. Our goal is to recommend movies to users.

To solve this problem, we need a method to determine which movies are similar to which. We can achieve this goal by embedding the movies into a low-dimensional space created so that similar movies are nearby.

Suppose we have the following list of movies:

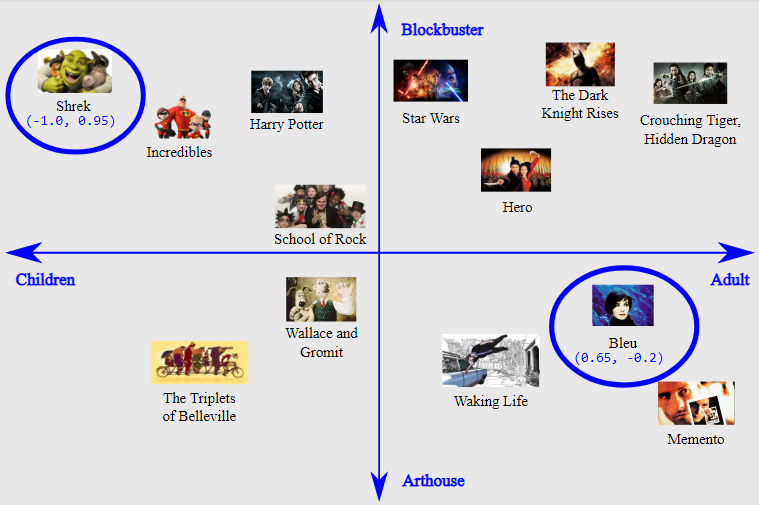
* Bleu
* The Dark Knight Rises
* Harry Potter and the Sorcerer’s Stone
* The Incredibles
* Shrek
* Star Wars
* The Triplets of Belleville
* Memento

The following is one possible way of arranging these movies in a one-dimensional vector so that the movies closest to each other are the most closely related:



This embedding helps capture how much each movie is geared towards children versus adults. However, there are many more aspects of a movie that one would want to capture when making recommendations.

We can add a second dimension to include a Blockbuster vs. Arthouse aspect to the problem:



With this two-dimensional embedding, we define a distance between movies such that movies are nearby if they are both alike in the context of being geared towards children vs. adults, as well as in the context of blockbuster movies vs. arthouse movies. These, of course, are just two of many characteristics of movies that might be important.

More generally, this two-dimensional approach maps these movies into an embedding space, where each word is described by a two-dimensional set of coordinates. For example, in this space, Shrek maps to (-1.0, 0.95) and Bleu maps to (0.65, -0.2).

In general, when learning a *d*-dimensional embedding, each movie is represented by *d* real-valued numbers, each one giving the coordinate in one dimension.

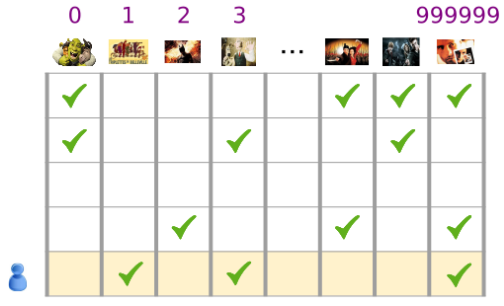
In this example, we have given a name to each dimension. When learning embeddings, the individual dimensions are not learned with names. Sometimes, we can look at the embeddings and assign semantic meanings to the dimensions, and other times we cannot. Often, each such dimension is called a latent dimension, as it represents a feature that is not explicit in the data but rather inferred from it.

Ultimately, it is the distances between movies in the embedding space that are meaningful, rather than a single movie's values along any given dimension.

## Categorical Input Data

Categorical data refers to input features that represent one or more discrete items from a finite set of choices. For example, it can be the set of movies a user has watched, the set of words in a document, or the occupation of a person.

This type of data is most efficiently represented via sparse tensors, tensors with very few non-zero elements. For example, if we're building a movie recommendation model, we can assign a unique ID to each possible movie, and then represent each user by a sparse tensor of the movies they have watched. Take the following matrix as a sample:



Each row of the matrix is an example capturing a user's movie-viewing history and is represented as a sparse tensor because each user only watches a small fraction of all possible movies. The last row corresponds to the sparse tensor [1, 3, 999999], using the vocabulary indices shown above the movie icons.

In order to use such representations within a Machine Learning system, we need a way to represent each sparse vector as a vector of numbers so that semantically similar items (movies or words) have similar distances in the vector space. But how can a word be represented as a vector of numbers?

The simplest way is to define a giant input layer with a node for every word in your vocabulary, or at least a node for every word that appears in your data. If 500,000 unique words appear in your data, you could represent the data with a length 500,000 vector and assign each word to a slot in the vector.

If you assign “horse” to index 1247, then feeding “horse” into your Network might copy a 1 into the 1247th input node and 0s into the rest (effectively, a one-hot encoded vector). This sort of representation is called a one-hot encoding, because only one index has a non-zero value.

More typically, your vector might contain counts of the words in a larger chunk of text. This is known as a bag of words representation. In a bag of words vector, several of the 500,000 nodes would have non-zero value.

But no matter how you determine the non-zero values, one-node-per-word gives you very sparse input vectors, that is, very large vectors with relatively few non-zero values. Sparse representations have a couple of problems that can make it hard for a model to learn effectively.

Huge input vectors mean a super-huge number of weights for a neural network. If there are M words in your vocabulary and N nodes in the first layer of the Network above the input, you have MxN weights to train for that layer. A large number of weights causes further problems:

* Amount of data: the more weights in your model, the more data you need to train effectively
* Amount of computation: the more weights, the more computation (and hardware) is required to train and use the model

If you feed the pixel values of RGB channels into an image classifier, it makes sense to talk about "close" values. Reddish blue is close to pure blue, both semantically and in terms of the geometric distance between vectors. But a vector with a 1 at index 1247 for “horse” is not any closer to a vector with a 1 at index 50,430 for “antelope” than it is to a vector with a 1 at index 238 for ”television”.

The solution to these problems is to use embeddings, which translate large sparse vectors into a lower-dimensional space that preserves semantic relationships.

## Translating to a Lower-Dimensional Space

While we want enough dimensions to encode rich semantic relations, we also want an embedding space that is small enough to allow us to train the model (relatively) quickly.

A useful embedding may be on the order of hundreds of dimensions. However, this is likely several orders of magnitude smaller than the size of the vocabulary for a natural language task.

An embedding is a matrix in which each column is the vector that corresponds to an item in your vocabulary. Thus, to get the dense vector for a single vocabulary item, you retrieve the column corresponding to that item.

But how would you translate a sparse bag of words vector? To get the dense vector for a sparse vector representing multiple vocabulary items (e.g., all the words in a sentence), you could retrieve the embedding for each individual item and then add them together. If the sparse vector contains counts of the vocabulary items, you could multiply each embedding by the count of its corresponding item before adding it to the sum.

## Obtaining Embeddings

### Standard Dimensionality Reduction Techniques

There are many existing mathematical techniques for capturing the important structure of a high-dimensional space in a low dimensional space. In theory, any of these techniques could be used to create an embedding for a machine learning system.

For example, principal component analysis (PCA) has been used to create word embeddings. Given a set of instances like bag of words vectors, PCA tries to find highly correlated dimensions that can be collapsed into a single dimension.

### Word2vec

Word2vec is an algorithm invented at Google for training word embeddings. Word2vec relies on the distributional hypothesis to map semantically similar words to geometrically close embedding vectors.

The distributional hypothesis states that words which often have the same neighboring words tend to be semantically similar. Both "dog" and "cat" frequently appear close to the word "vet", and this fact reflects their semantic similarity. As the linguist John Firth put it in 1957, "You shall know a word by the company it keeps".

Word2Vec exploits contextual information like this by training a neural net to distinguish actually co-occurring groups of words from randomly grouped words. The input layer takes a sparse representation of a target word together with one or more context words. This input connects to a single, smaller hidden layer.

In one version of the algorithm, the system makes a negative example by substituting a random noise word for the target word. Given the positive example "the plane flies", the system might swap in "jogging" to create the contrasting negative example "the jogging flies".

The other version of the algorithm creates negative examples by pairing the true target word with randomly chosen context words. So it might take the positive examples (the, plane), (flies, plane) and the negative examples (compiled, plane), (who, plane) and learn to identify which pairs actually appeared together in text.

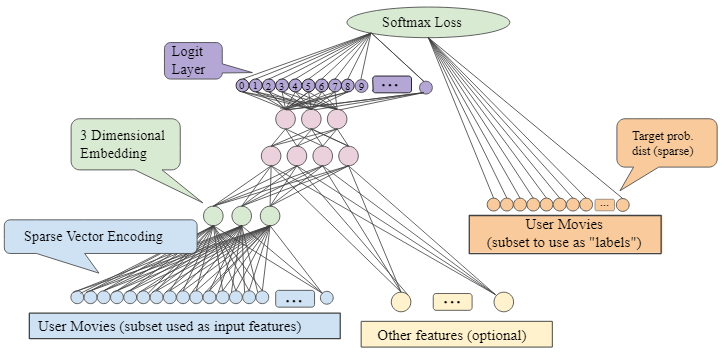
However, the classifier is not the real goal for either version of the system. After the model has been trained, you have an embedding. You can use the weights connecting the input layer with the hidden layer to map sparse representations of words to smaller vectors. This embedding can be reused in other classifiers.

### Training an Embedding as part of a larger model

You can also learn an embedding as part of the neural network for your target task. This approach gets you an embedding well customized for your particular system, but it may take longer than training the embedding separately.

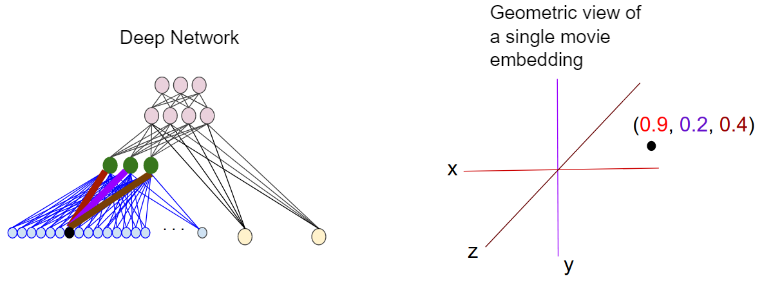
In general, when you have sparse data (or dense data that you'd like to embed), you can create an embedding unit that is just a special type of hidden unit of size d. This embedding layer can be combined with any other features and hidden layers. As in any DNN, the final layer will be the loss that is being optimized.

For example, let's say we're performing collaborative filtering, where the goal is to predict a user's interests from the interests of other users. We can model this as a supervised learning problem by randomly setting aside (or holding out) a small number of the movies that the user has watched as the positive labels, and then optimize a softmax loss.



As another example, if you want to create an embedding layer for the words in a real-estate ad as part of a DNN to predict housing prices, then you'd optimize an L2 Loss using the known sale price of homes in your training data as the label.

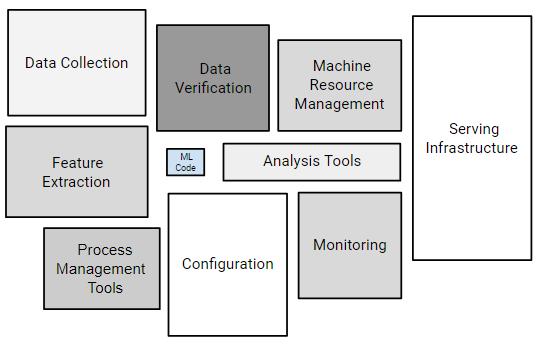
When learning a d-dimensional embedding each item is mapped to a point in a d-dimensional space so that the similar items are nearby in this space. The following figure helps to illustrate the relationship between the weights learned in the embedding layer and the geometric view. The edge weights between an input node and the nodes in the d-dimensional embedding layer correspond to the coordinate values for each of the d axes.



# Machine Learning Engineering

## Production Machine Learning Systems

Real-world production Machine Learning systems are large ecosystems of which the model is just a small part:



The Machine Learning code is at the heart of a real-world Machine Learning production system, but that box often represents only 5% or less of the overall code of that total Machine Learning production system. Notice that a Machine Learning production system devotes considerable resources to input data: collecting it, verifying it, and extracting features from it. Furthermore, notice that a serving infrastructure must be in place to put the ML model's predictions into practical use in the real world. Fortunately, many of these components are reusable.

## Static vs. Dynamic Training

Broadly speaking, there are two ways to train a model:

* A static model is trained offline, that is, we train the model exactly once and then use that trained model for a while
* A dynamic model is trained online, that is, data is continually entering the system and we're incorporating that data into the model through continuous updates

Generally, the following points dominate the static vs. dynamic training decision:

* Static models are easier to build and test
* Dynamic models adapt to changing data

If your dataset truly isn't changing over time, choose static training because it is cheaper to create and maintain than dynamic training. However, many information sources really do change over time, even those with features that you think are as constant. Even with static training, you must still monitor your input data for change.

## Static vs. Dynamic Inference

You can choose either of the following inference strategies:

* Offline inference: you make all possible predictions in a batch, using a MapReduce or something similar. You then write the predictions to an SSTable or Bigtable, and then feed these to a cache/lookup table
* Online inference: you predict on demand, using a server

Here are the pros and cons of offline inference:

* Don’t need to worry much about cost of inference ✔
* Can likely use batch quota or some giant MapReduce ✔
* Can do post-verification of predictions before pushing ✔
* Can only predict things we know about (bad for long tail) ❌
* Update latency is likely measured in hours or days ❌

Here are the pros and cons of online inference:

* Pro: Can make a prediction on any new item as it comes in (great for long tail) ✔
* Compute intensive, latency sensitive (may limit model complexity) ❌
* Monitoring needs are more intensive ❌

## Data Dependencies

The behavior of a Machine Learning system is dependent on the behavior and qualities of its input features. As the input data for those features changes, so will the model. Sometimes that change is desirable, but sometimes it is not.

In traditional software development, you focus more on code than on data. In Machine Learning development, although coding is still part of the job, your focus must widen to include data. For example, on traditional software development projects, it is a best practice to write unit tests to validate your code. On Machine Learning projects, you must continuously test, verify, and monitor your input data.

For example, you should continuously monitor your model to remove unused (or little used) features. Imagine a certain feature that has been contributing little or nothing to the model. If the input data for that feature abruptly changes, your model's behavior might also abruptly change in undesirable ways.

Thus, you should keep in mind five core factors about the data you’re using: reliability, versioning, necessity, correlations and feedback loops.

### Reliability

Some questions to ask about the reliability of your input data:

* Is the signal always going to be available or is it coming from an unreliable source?
* Is the signal coming from a server that crashes under heavy load?
* Is the signal coming from humans that go on vacation every August?

### Versioning

Some questions to ask about versioning:

* Does the system that computes this data ever change?
* How often?
* How will you know when that system changes?

Sometimes, data comes from an upstream process. If that process changes abruptly, your model can suffer.

Consider creating your own copy of the data you receive from the upstream process. Then, only advance to the next version of the upstream data when you are certain that it is safe to do so.

### Necessity

* Does the usefulness of the feature justify the cost of including it?

It is always tempting to add more features to the model. For example, suppose you find a new feature whose addition makes your model slightly more accurate. More accuracy certainly sounds better than less accuracy. However, you've just added to your maintenance burden. That additional feature could degrade unexpectedly, so you've got to monitor it. Think carefully before adding features that lead to minor short-term wins.

### Correlations

Some features correlate (positively or negatively) with other features:

* Are any features so tied together that you need additional strategies to separate them?

### Feedback Loops

Sometimes a model can affect its own training data. For example, the results from some models, in turn, are directly or indirectly input features to that same model or even a separate model. For example, consider two models for predicting stock prices:

* Model A, which is a bad predictive model
* Model B

Since Model A is buggy, it mistakenly decides to buy stock in Stock X. Those purchases drive up the price of Stock X. Model B uses the price of Stock X as an input feature, so Model B can easily come to some false conclusions about the value of Stock X stock. Model B could, therefore, buy or sell shares of Stock X based on the buggy behavior of Model A. Model B's behavior, in turn, can affect Model A, possibly triggering a tulip mania or a slide in Company X's stock.

## Real-World Guidelines

Here are some effective Machine Learning guidelines:

* Keep your first model simple
* Focus on ensuring data pipeline correctness
* Use a simple, observable metric for training and evaluation
* Own and monitor your input features
* Treat your model configuration as code: review it and check it
* Write down the results of all experiments, especially "failures"

# Fairness

Machine learning models are not inherently objective. Engineers train models by feeding them a data set of training examples, and human involvement in the provision and curation of this data can make a model's predictions susceptible to bias.

When building models, it's important to be aware of common human biases that can manifest in your data, so you can take proactive steps to mitigate their effects.

## Some Types of Bias

### Reporting Bias

Reporting bias occurs when the frequency of events, properties, and/or outcomes captured in a dataset does not accurately reflect their real-world frequency. This bias can arise because people tend to focus on documenting circumstances that are unusual or especially memorable, assuming that the ordinary can "go without saying."

### Automation Bias

Automation bias is a tendency to favor results generated by automated systems over those generated by non-automated systems, irrespective of the error rates of each.

### Selection Bias

Selection bias occurs if a dataset's examples are chosen in a way that is not reflective of their real-world distribution. Selection bias can take many different forms:

* Coverage bias: Data is not selected in a representative fashion
* Non-response bias (or participation bias): Data ends up being unrepresentative due to participation gaps in the data-collection process
* Sampling bias: Proper randomization is not used during data collection

### Group Attribution Bias

Group attribution bias is a tendency to generalize what is true of individuals to an entire group to which they belong. Two key manifestations of this bias are:

* In-group bias: A preference for members of a group to which you also belong, or for characteristics that you also share
* Out-group homogeneity bias: A tendency to stereotype individual members of a group to which you do not belong, or to see their characteristics as more uniform

### Implicit Bias

Implicit bias occurs when assumptions are made based on one's own mental models and personal experiences that do not necessarily apply more generally.

A common form of implicit bias is confirmation bias, where model builders unconsciously process data in ways that affirm preexisting beliefs and hypotheses. In some cases, a model builder may keep training a model until it produces a result that aligns with their original hypothesis; this is called experimenter's bias.

## Identifying Bias

As you explore your data to determine how best to represent it in your model, it's important to also keep issues of fairness in mind and proactively audit for potential sources of bias. Where might bias lurk? Three red flags to look out for are:

* Missing feature values
* Unexpected feature values
* Data skew

### Missing Feature Values

If your data set has one or more features that have missing values for many examples, that could be an indicator that certain key characteristics of your data set are under-represented.

### Unexpected Feature Values

When exploring data, you should also look for examples that contain feature values that stand out as especially uncharacteristic or unusual. These unexpected feature values could indicate problems that occurred during data collection or other inaccuracies that could introduce bias.

### Data Skew

Any sort of skew in your data, where certain groups or characteristics may be under- or over-represented relative to their real-world prevalence, can introduce bias into your model.

## Evaluating for Bias

When evaluating a model, metrics calculated against an entire test or validation set don't always give an accurate picture of how fair the model is.

For example, consider a model that predicts whether a patient has a disease or not. In addition to calculating metrics such as precision and recall for the entire dataset, it could be valuable to calculate these metrics for the men and women subgroups. Comparing these results might lead to uncovering bias that was not known before.

# Convolutional Neural Networks (CNN)

## Introduction

A Convolutional Neural Network (CNN) is a specific type of neural network in which at least one layer is a convolutional layer. A typical CNN consists of some combination of the following layers:

* Convolutional layers
* Pooling layers
* Dense layers

CNN have had great success in certain kinds of problems, such as image recognition.

## CNNs for Image Classification

CNN could be used to progressively extract higher and higher-level representations of the image content. Instead of preprocessing the data to derive features like textures and shapes, a CNN takes only the image's raw pixel data as input and learns how to extract these features so that it can infer what object they constitute.

To start, the CNN receives an input feature map: a three-dimensional matrix where the size of the first two dimensions corresponds to the length and width of the images in pixels. The size of the third dimension is three, corresponding to the three channels of a color image: red, green, and blue.

The CNN comprises a stack of modules, each of which performs three operations: convolution, application of the ReLU activation function and pooling.

### Convolution

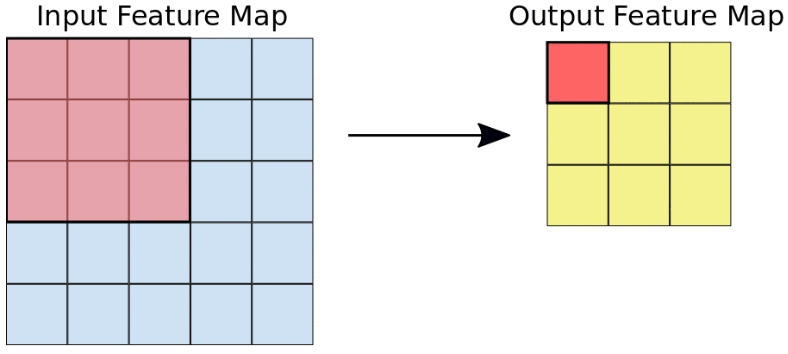
Convolution extracts tiles of the input feature map and applies filters to the tiles to compute new features, producing an output feature map, also known as convolved feature (which may have a different size and depth than the input feature map).

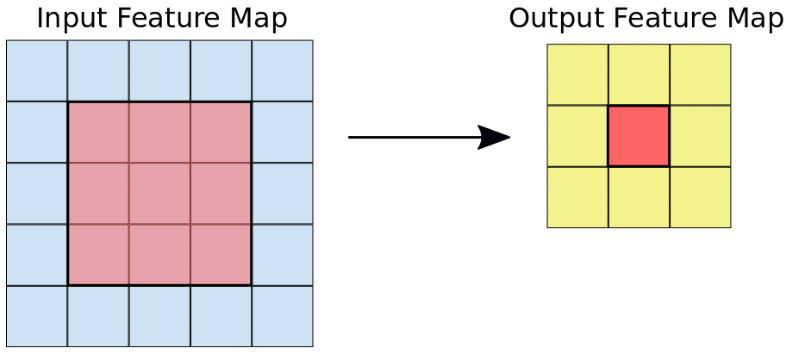
Convolutions are defined by two parameters:

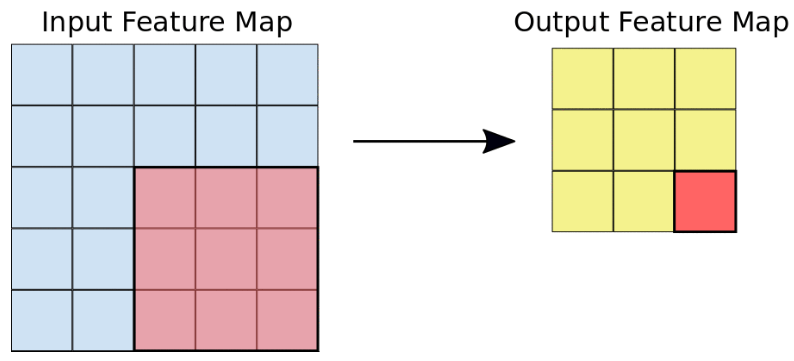
* Size of the tiles that are extracted (typically 3x3 or 5x5 pixels)
* The depth of the output feature map, which corresponds to the number of filters that are applied

During a convolution, the filters (matrices of the same size as the tile size) slide over the input feature map's grid horizontally and vertically, one pixel at a time, extracting each corresponding tile.

The following images show the process of a 3x3 convolution of depth 1 performed over a 5x5 input feature map, also of depth 1. There are nine possible 3x3 locations to extract tiles from the 5x5 feature map, so this convolution produces a 3x3 output feature map.

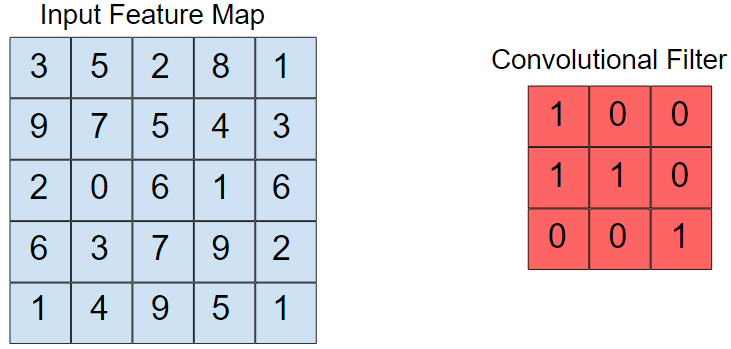




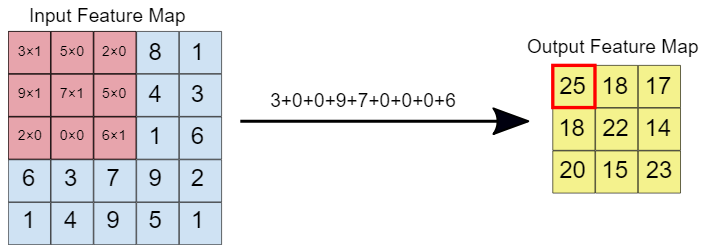


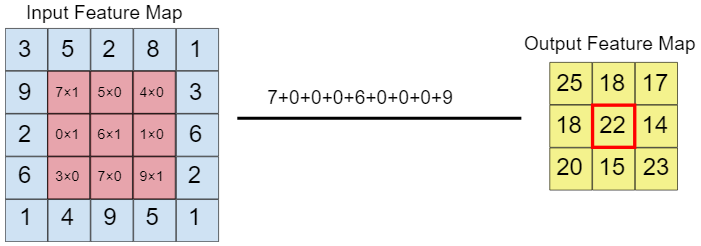
For each filter-tile pair, the CNN performs element-wise multiplication of the filter matrix and the tile matrix, and then sums all the elements of the resulting matrix to get a single value. Each of these results for every filter-tile pair is then the output in the convolved feature matrix.

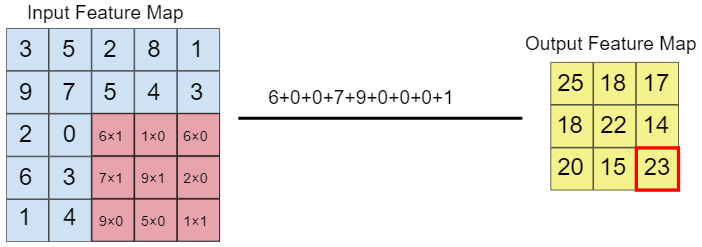
For example, using the following input feature map and convolutional filter:



The output matrix is calculated by repeating the following process over the entire input map:







During training, the Convolutional Neural Network learns the optimal values for the filter matrices, enabling it to extract meaningful features (textures, edges, shapes) from the input feature map.

As the number of filters (the output feature map depth) applied to the input increases, so does the number of features the CNN can extract. However, the tradeoff is that filters compose most resources expended by the network, so training time also increases as more filters are added. Additionally, each filter added provides less and less value than the previous one, so engineers aim to construct networks that use the minimum number of filters needed to extract the features necessary for accurate image classification.

### ReLU

Following each convolution operation, the CNN applies a ReLU (Rectified Linear Unit) transformation to the convolved feature, in order to introduce nonlinearity into the model.

The ReLU function,

returns *x* for all values of *x > 0* and returns *0* for all values of *x ≤ 0*.

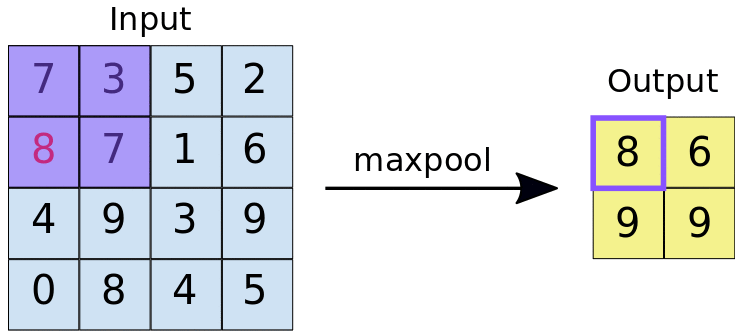
### Pooling

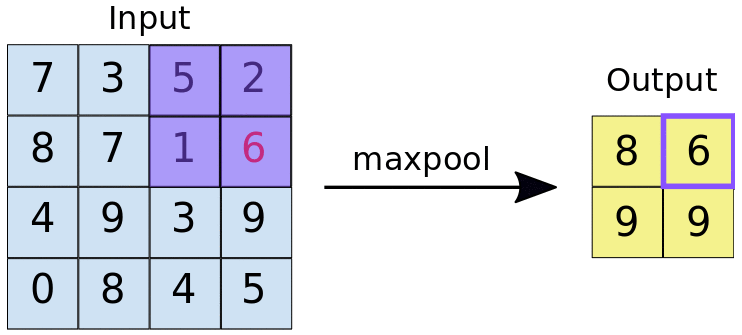
After ReLU comes a pooling step, in which the CNN down samples the convoluted feature (to save on processing time), reducing the number of dimensions of the feature map while still preserving the most critical feature information. A common algorithm used for this process is called max pooling.

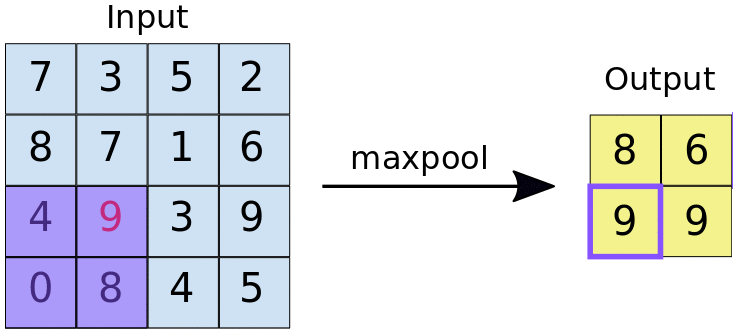
Max pooling operates in a similar fashion to convolution. We slide over the feature map and extract tiles of a specified size. For each tile covered by the filter, the maximum value is output to a new feature map, and all other values are discarded. This operations takes two parameters:

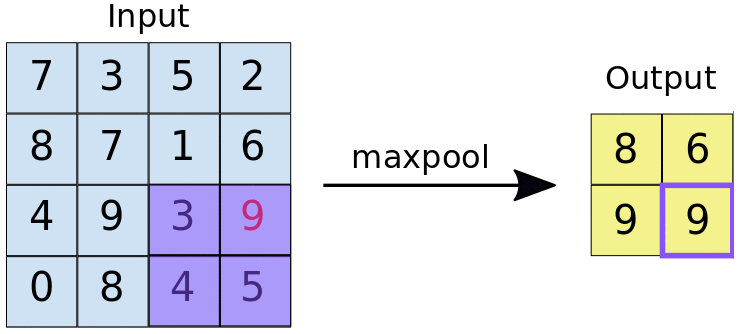
* The size of the max-pooling filter (typically 2x2 pixels)
* The stride: the distance, in pixels, separating each extracted tile. Unlike with convolution, where filters slide over the feature map pixel by pixel, in max pooling, the stride determines the locations where each tile is extracted. For a 2x2 filter, a stride of 2 specifies that the max pooling operation will extract all nonoverlapping 2x2 tiles from the feature map

The following is an example where we have a 4x4 input feature map, for which we use a 2x2 filter with a stride value of 2:









## Fully Connected Layers

At the end of a CNN are one or more fully connected layers. We call a pair of layers “fully connected layers” when every node in the first layer is connected to every node in the second layer.

The job of these layers is to perform classification based on the features extracted by the convolutions. Typically, the final fully connected layer contains a Softmax activation function, which outputs a probability value from 0 to 1 for each of the classification labels the model is trying to predict.

## Preventing Overfitting for CNNs

There are two major techniques to prevent overfitting when building a CNN:

* Data augmentation: the process of artificially increasing the number of training examples by performing random transformations to existing images to create a new set of variants
* Dropout regularization: randomly remove units from the neural network during a training gradient step

Due note that, for machine learning in general, overfitting is more of a problem when the available dataset is small. For bigger datasets (such as ones with millions of images), applying dropout is unnecessary, as well as for data augmentation which sees its merits diminished.

It’s also worth nothing these two techniques should be used together since applying data augmentation alone to a dataset might not be enough to get rid of overfitting (while the model isn’t trained twice using the same input, training examples will still be heavily correlated).

# Sources

Main bulk of the document- <https://developers.google.com/machine-learning/crash-course/>

“Is There a Standard Heuristic for Model Tuning?”-

[https://colab.research.google.com/notebooks/mlcc/first\_steps\_with\_tensor\_flow.ipynb](https://colab.research.google.com/notebooks/mlcc/first_steps_with_tensor_flow.ipynb?utm_source=mlcc&utm_campaign=colab-external&utm_medium=referral&utm_content=firststeps-colab&hl=en#scrollTo=QU5sLyYTqzqL)

Tensor concept- <https://hackernoon.com/learning-ai-if-you-suck-at-math-p4-tensors-illustrated-with-cats-27f0002c9b32>

F1 score concept- <https://en.wikipedia.org/wiki/F1_score>

Linear vs Logistic regression- <https://stackoverflow.com/a/39227073>

Min-Max Normalization and Z-score concepts:

<https://www.codecademy.com/courses/machine-learning/articles/normalization?action=resume_content_item>

K-Nearest Neighbors definition:

<https://www.codecademy.com/courses/machine-learning/lessons/knn/exercises/knn?action=resume_content_item>

Clustering definition:

<https://www.codecademy.com/courses/machine-learning/lessons/machine-learning-clustering/exercises/introduction?action=resume_content_item>

K-Means definition:

<https://www.codecademy.com/courses/machine-learning/lessons/machine-learning-clustering/exercises/k-means?action=resume_content_item>

Convolutional Neural Networks

<https://developers.google.com/machine-learning/practica/image-classification/convolutional-neural-networks>

Formulate ML problems: <https://developers.google.com/machine-learning/problem-framing/formulate>

Deciding on ML: <https://developers.google.com/machine-learning/problem-framing/framing>

Framing a ML project: <https://developers.google.com/machine-learning/problem-framing/try-it/framing-exercise>

Data preparation and feature engineering: <https://developers.google.com/machine-learning/data-prep>