**Machine Learning:**

The science and art of programming computers so that they learn from data.

**Common Machine Learning Terminology:**

Supervised Learning:

The data includes desired solutions called 'labels' or ground truth.

Attribute:

Is the data type i.e. mileage

Feature:

An attribute with a value i.e. mileage = 15000 mph

Target Value:

Given a set of features (mileage, age, brand) these are predictions (predictors) for our target value (the price of the car). We use a system with many examples (of cars) including predictors (mileage, age, brand of many cars) and labels(price of many cars).

Unsupervised Learning:

The data includes no labels or ground truth. This is usually a much harder problem.

Dimensionality Reduction:

Simplify data without losing info. Merge several correlated features into one

feature #1 and feature #2 feature

i.e. (mileage) and (car age) wear and tear

This is also called feature extraction.

Anomaly Detection:

Train your predictor with normal instances of data and then you can tell if it's normal or an anomaly.

Batch Learning:

Train with all the data available, this is done off-line requiring lots of compute. Since you have trained your model using all of your data you must retrain your model with ALL the data when new data arrives.

Online Learning:

Train a system incrementally by feeding it instances sequentially, either individually or in small groups called mini-batches. Learning step is faster in mini-batches.

Learning Rate:

How fast the online learning system adapts to new data. Higher the rate, the noisier it is and the learning algorithm discards old data faster. Also known as step size in optimization nomenclature.

Feature Engineering:

Is the process to pick good features to train on. Feature selection is picking the most useful existing features to train on. Feature extraction combines existing features.

Overfitting:

Model works well on the training data but doesn't actually generalize well (might be too complex; detects patterns in noise).

Solution to reduce overfitting is

1. Simplify your model, reduce the number of parameters in the model (linear versus polynomial model).
2. Gather more training data.
3. Remove noise in training data (removing outliers, fix data errors)

Regularization:

Constrains an existing model to make it simpler. The hyperparameter is a parameter or parameters that can be tuned in the learning model. It can control the amount of regularization.

Underfitting:

The model is too simple.

Solution to reduce underfitting is

1. Pick a more powerful model.
2. Feed better features to the learning algorithm.
3. Reduce regularization or lower number of hyperparameters in model.

Testing and Validation:

Split the data into 2 sets: training data and testing data.

Train the model on the training set, then test the model on the test set.

Error rate on the test cases is called the generalization error. If the training error is low but the generalization error is high then the model is overfitting the training data.

Don't adapt the model and hyperparameters for the test set! Instead a second hold-out set called the validation set is used.

Given training data you should split it such that 80% is used to train the model and 20% is used for the validation.

**Linear Prediction:**

Generally, we are given data and .

A pair is called a training example or the *i*th instance of data.

The dataset we are trying to learn has *m* training examples where is called the training set.

Therefore our linear predictor has the form

for .

Note that , or the first feature is constant. Then the linear predictor *h* is an affine function of or linear plus a constant term where is called the offset or bias or constant term.

The data matrix is a

where the *i*th row of *X* is the *i*th feature vector transposed. The *j*th column of *X* gives values of the *j*th feature across our all instances of our dataset. We can also say that is the value of the *j*th feature for the *i*th data point.

The regression or least squares problem is to find the parameter that minimizes the following cost function: the mean squared prediction error (also known as MSE loss function or L2 loss).

The root mean squared error is widely used and has the same units/scale as the labels .

Assuming that *X* has full column rank or linearly independent columns (this implies that )

there is a unique optimal **:**

**.**

For large datasets this can be inefficient to compute since we have to compute an inverse. Instead let's try to minimize the using the gradient descent algorithm, which starts with some random that will perform the following update:

where is the gradient of the cost function, and is the learning rate or hyperparameter.

The algorithm will step in the direction of the steepest decrease of until it hopefully converges.

For *m* instances this can be solved as and therefore the gradient descent algorithm is

**.**

(Note the stands for the assignment operator, meaning that the old value of is overwritten by the update step. This efficient process is also called "memoization" )

The above update will be performed over the full set of data or on every example in the entire training set on every step. Also, this update is simultaneously performed for all values of parameters or for .

Let's look at one instance (training example) and so we can neglect the summation in the definition of

For a single training example this gives the update rule

.

After slight rearrangement we see the magnitude of the update is proportional to the error term and so will change more for a larger error.

Stochastic Gradient Descent (SGD):

This method is the more efficient and widely used method. It picks a random instance of the data and computes gradient descent based on that instance. The cost function will "bounce" around. The learning rate is usually set by the learning rate schedule.

Learning Rate Schedule:

Based on varying the learning rate based on a schedule. The epoch is just a period of time.

An example schedule could be

where

Mini-batch:  
Compute gradient descent on a small random sets of instances.

Early Stopping:

Stop learning when validation reaches the minimum, stop training as soon as the validation error stops decreasing.

**Example:** Housing Price Prediction

2000

First the instance of data or *m* = 1:

Let's use linear regression for our model, then our predictor for the first instance is :

.

Plugging in our first instance of data:

where = our weights or parameters to tune.

Our prediction error for the first instance is simply:

Let's perform gradient descent for one training example *m*=1 with *j=3,* . The parameter update will be:

Let's perform batch gradient descent:

def bgd(xi, yi, theta, eta):

# Calculate the gradients

# -2 \* x\_j \* (y - (theta \* x)) = 2 \* x\_j \* ((theta \* x) - y)

gradients = 2\*xi.T.dot(xi.dot(theta) - yi)

theta = theta - eta \* gradients

return theta

Differences between batch gradient descent (BGD), stochastic gradient descent (SGD), and mini-batch gradient descent (mini-BGD):

BGD uses the whole training set at each iteration to compute the gradients. This makes it slow when the training set is large.

SGD picks a random instance of the training set at every iteration and computes the gradients

based solely on that iteration. This makes it faster and less memory intensive. The cost function unfortunately will bounce up and down decreasing only on average. It will reach something close to a global optimum. The randomness helps the algorithm jump out of local minima.

We can resolve the above problem by using a learning rate schedule, whereby we start with a large learning rate (helps jump out of local min) then gradually decrease the learning rate allowing it to settle at a global minimum. The function that determines the learning rate at each iteration is called the learning rate schedule.

Below is code for implementing a learning rate schedule for SGD.

n\_epochs = 50

t0, t1 = 5, 50 # learning schedule hyperparameters

def learning\_schedule(t):

return t0/(t + t1)

theta = np.random.rand(2,1) #random initialization

for epoch in range(n\_epochs):

for i in range(m): #here m is the number of rows in the feature dataset

random\_index = np.random.randint(m)

xi = X\_b[random\_index:random\_index + 1]

yi = y[random\_index:random\_index+1]

gradients = 2\*xi.T.dot(xi.dot(theta) - yi)

# the main factor that decreases the learning rate is the epoch since

t=epoch(0:n\_epochs-1)\*size\_of\_feature\_data+i(0:m)

eta = learning\_schedule(epoch \* m + i)

theta = theta - eta \* gradients

If you want to make sure that the algorithm goes through every instance then you may need to shuffle to make sure it covers all instances of the feature data.

Mini-BGD computes gradients on small random sets of instances called mini-batches. The main advantage is that you get a performance boost from hardware. Here's how we can perform a mini-batch in Tensorflow.

n\_epochs = 10 #number of times you run through the whole thing

batch\_size = 100

# number of batches given data size, m is the number of instances of data

n\_batches = int(np.ceil(m/batch\_size))

def fetch\_batch(epoch, batch\_index, batch\_size):

np.random.seed(epoch \* n\_batches + batch\_index)

indices = np.random.randint(m, size=batch\_size)

X\_batch = scaled\_housing\_data\_plus\_bias[indices]

y\_batch = housing.target.reshape(-1,1)[indices]

return X\_batch, y\_batch

with tf.Session() as sess:

sess.run(init)

for epoch in range(n\_epochs):

for batch\_index in range(n\_batches):

X\_batch, y\_batch = fetch\_batch(epoch, batch\_index, batch\_size)

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch})

best\_theta = theta.eval()

We can plot the progress of each algorithm in parameter space below.



Mini-BGD's progress in parameter space is less erratic than SGD. As a result, mini-BGD will end up walking around a bit closer to the minimum than SGD but may suffer from getting stuck in local-minima. BGD actually ends up at the minimum while SGD and mini-BGD is close enough. BGD will take much longer to find the minimum since each step is much longer and we may reach the minimum with SGD and mini-BGD if we use a good learning rate schedule.

**Classification:**

Logistic Regression:

If the estimated probability, , then the model predicts an instance that belongs to the positive class. If the estimated probability, , then the model predicts an instance that belongs to the negative class. The output of this model is the logistic of this result.

where is known as the sigmoid function. A plot of the sigmoid function

is shown below



Once logistic regression has estimated the probability that the instance belongs to the positive class it can make a positive prediction

therefore, logistic regression predicts

The cost function for one training example:

grows large when for the positive class. Therefore, will be large.

grows small when for the positive class. Therefore, will be small.

grows small when for the negative class. Therefore, will be small.

grows large when for the negative class. Therefore, will be large.

The average cost over the training sets is:

.

There is no closed form that minimizes the above cost function.

The derivative of is

.

Softmax Regression:

Supports classification for multiple classes. Given an instance the softmax regression model computes a score for each class K where the score is

, where each class will have it's own dedicated parameter stored in a parameter matrix .

Once we have computed the score of every class for instance you can estimate the probability that the instance belongs to class *K* by taking the running score through a

softmax, which computes the exponential of every score then normalizes it.

where # of classes, **=**vector containing scores of each class for instance ,**=**estimated probability that instance belongs to class *K* given score of each class for that instance.

Softmax regression classifier predicts with highest probability (or the class with highest score).

where the argmax returns the value of the variable that maximizes the estimated probability. Note the classes must be mutually exclusive (we can use softmax to estimate different types of species of plants).

Cross-entropy:

Suppose we want a model that estimates a high probability for a target class. If we minimize the cross-entropy we get this objective since it penalizes the low-probability model. This can be used to measure how well a set of estimated class probabilities match the target class.

where is equal to 1 if the target class for the *i*th instance is class *k*, otherwise it's equal to zero.

The cross-entropy (between two probability distributions *p* and *q* is:

.

This is also known as the KL-divergence.

**Performance Measures:**

Cross-Validation:

One way to evaluate your model is to split the training set into a smaller training set and a validation set, then train your models against the smaller training set and evaluate them against the validation set.

A great way to perform cross-validation is called *k-fold* cross validation. Here we randomly split the training set into *k* distinct subsets called *folds*, then it trains and evaluates the model *k* times picking a different fold for evaluation every time and training on the other *k-1* folds. This can be easily performed using the Scikit-Learn cross-validation feature.

Confusion Matrix:

Is a better way to evaluate the performance of a classifier. The idea is to count the number of times class A is classified as class B. Specifically, in the case of a dataset like MNIST, which is a set of 70,000 images of digits handwritten by high school students and employees of the US Census Bureau, we may want to know the number of times that the classifier confused images of 5s with 3s. In order to do so we would look at the row 5 and column 3 of the confusion matrix.

Each row in the confusion matrix represents the actual class, while each column represents a predicted class. Note, if the first row of the matrix considers non 5s then the ones that are correctly classified as non 5s are called true negatives and those classified as incorrectly as non-5s are false negatives. Whereas, if the second row considers the images of 5s then those that are correctly identified as 5s are called true positive (5 is the positive class) and those that are identified incorrectly as 5s are false positives. A perfect classifier would only have true positives and true negatives.

The confusion matrix gives a lot information but sometimes we want a more concise metric. One metric for evaluating the accuracy of the positive prediction of the classifier is called the precision, which is defined as

,

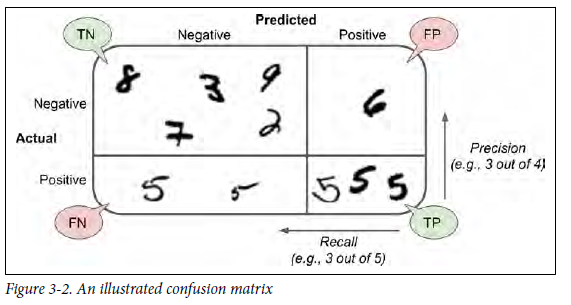
where *TP = the number of true positives, FP = the number of false positives.*

A trivial way to get 100% precision is by one single prediction and ensure that it's correct. This would not be very useful since the classifier would ignore all but one positive instance. Typically, another metric is used along with precision called recall (also called the true-positive rate), which is the ratio of positive instances that are correctly identified by the classifier.

,

where *TP = the number of true positives, FN = the number of false negatives.*

The figure below is an illustration of the confusion matrix using precision and recall.



Precision/Recall Tradeoff:

Let's consider how a classifier (like a SGD classifier) makes its classification decision. It will compute a score based on a decision function, and if that score is greater than a threshold it assigns it to the positive class or else it assigns it to negative class.

Generally, increasing precision reduces recall and vice versa. Many SGD classifiers will set the default threshold to 0 but when we increase the threshold the recall is decreased. We can decide the best threshold to use by obtaining the scores of the instances of the training set using cross-validation but return decision scores. We then compute a precision and recall for all possible thresholds as shown in the figure below (note the precision can also slightly decrease as the threshold increases).



We can also select a good precision/recall tradeoff by plotting each directly against each other as shown below. We can see that precision starts to sharply drop-off around 80% recall. A precision/recall tradeoff should be selected before this drop.



Error Analysis:

Once we have fine-tuned the hyperparameters of our model you can still find ways to improve it. One way to do this is to analyze the types of errors the model makes. First let's look at the confusion matrix for our SGD classifier for the MNIST dataset.

array([[5725, 3, 24, 9, 10, 49, 50, 10, 39, 4],

[ 2, 6493, 43, 25, 7, 40, 5, 10, 109, 8],

[ 51, 41, 5321, 104, 89, 26, 87, 60, 166, 13],

[ 47, 46, 141, 5342, 1, 231, 40, 50, 141, 92],

[ 19, 29, 41, 10, 5366, 9, 56, 37, 86, 189],

[ 73, 45, 36, 193, 64, 4582, 111, 30, 193, 94],

[ 29, 34, 44, 2, 42, 85, 5627, 10, 45, 0],

[ 25, 24, 74, 32, 54, 12, 6, 5787, 15, 236],

[ 52, 161, 73, 156, 10, 163, 61, 25, 5027, 123],

[ 43, 35, 26, 92, 178, 28, 2, 223, 82, 5240]])

The above matrix representation shows many numbers, a more convenient way is to look at an image representation of the confusion matrix



Most of the images are on the main diagonal meaning that they were classified correctly. The 5s are slightly darker than the other digits, which could mean there are fewer 5s in the dataset or the classifier does not perform as well on the 5s.

Let's now focus on the confusion matrix for the error rates (instead of the absolute number of errors as shown above). We can do this by dividing each value in the confusion matrix by the number of images in the corresponding class



Above we can see that the columns for classes 8 and 9 are bright meaning that many images get misclassified as 8s and 9s. Also, the rows for classes 8 and 9 are bright telling you that 8s and 9s are confused with other digits. Row 1 is dark and so most 1s are classified correctly.

The above error rate confusion matrix can be used to improve your classifier. The plot shows that we should focus on improving classification of 8s and 9s as well as fix the 3/5 confusion. We can do this by gathering more training data for these digits. or engineer new features to help the classifier (counting the number of closed loops in a number like 8), or preprocess the images such that some patterns stand out.

Learning Curves:

We can also look at plots of the model's performance on the training set and the validation set

as a function of the training set size (or training iteration). Below is such plot for plain Linear Regression.



From the plot above we can see that when there are just a 2 instances in the training set, the model can fit them perfectly but as new instances are added to the training set, the root-mean-square-error increases because the data is noisy and not linear. The training error increases until it begins to settle out at which point adding new instances doesn't change the average error. When there are very few instances of the validation data, it cannot generalize, which is why the validation error is large. As more examples of validation data are shown the validation error slowly decreases but also slowly settles close to the training error curve. These learning curves are typical of an underfitting model. They are close to each other and settle at a fairly high RMSE.

Below is a similar plot but for a 10th-degree polynomial model on the same data.



As shown, the error is much lower than with Linear Regression. There is a gap between both curves since the error is lower on the training data. This means that the model performs better on the training data, which is typical of overfitting the data. If you use a much larger training set then the two curves would continue to get closer.

The Bias/Variance Tradeoff:

A model's generalization error can be expressed as the sum of 3 terms

1. Bias:

This part of the generalization error is due to wrong assumption such as assuming that the data is linear when in fact it's quadratic. A high bias model is likely to underfit the data.

2. Variance:

This part of the generalization error is due to the model's excessive sensitivity to the small variations in the training data. A model with many degrees of freedom (like a high-degree polynomial) is likely to have high variance and thus overfit the training data.

3. Irreducible error:

This part of the generalization error is due to the noise in the data itself. The only way to reduce this is to clean up the data (fix the data source, remove bad sensors, remove outliers).

Increasing a model's complexity will typically increase its variance and reduce its bias. Conversely, reducing a model's complexity increases the bias and reduces its variance.

**Not Discussed:**

The following Machine Learning topics were not discussed in this article

-Bayesian Networks

-Support Vector Machines

-Decision Trees

-Ensemble Learning

-Random Forests

-Dimensionality Reduction

-Deep Neural Nets

**Machine Learning Software**

**Python Machine Learning Modules:**

NumPy*(https://docs.scipy.org/doc/numpy/user/quickstart.html)*:

This module adds support for large, multi-dimensional [arrays](https://en.wikipedia.org/wiki/Array_data_structure) and [matrices](https://en.wikipedia.org/wiki/Matrix_(math)), along with a large collection of [high-level](https://en.wikipedia.org/wiki/High-level_programming_language) [mathematical](https://en.wikipedia.org/wiki/Mathematics) [functions](https://en.wikipedia.org/wiki/Function_(mathematics)) to operate on these arrays in Python. Using NumPy in Python gives functionality comparable to [MATLAB](https://en.wikipedia.org/wiki/MATLAB) since they are both interpreted, and they both allow the user to write fast programs as long as most operations work on arrays or matrices instead of [scalars](https://en.wikipedia.org/wiki/Scalar_(computing)).

Matplotlib*(https://matplotlib.org/)*:

This module adds support for a plotting package that provides Matlab-like plotting functionality.

Pandas*(https://pandas.pydata.org/)*:

This module adds data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and [time series](https://en.wikipedia.org/wiki/Time_series).

Jupyter Notebook *(http://jupyter.org/):*

This module adds support for a server-client application that allows editing and running [notebook documents](https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html#notebook-document) via a web browser. Notebook documents are documents produced by the, which contain both computer code (e.g. python) and rich text elements (paragraph, equations, figures, links, etc…).

**Python Machine Learning Frameworks:**

Scikit-Learn(*https://scikit-learn.org/):*

This framework easily implements many Machine Learning algorithms efficiently, so it makes for a great entry point to learn Machine Learning but lacks support for very Deep Neural Network models.

Tensorflow *(https://www.tensorflow.org/):*

This framework is for implementing very Deep Neural Network/Deep Learning models. This is a more complex library for distributed numerical computation using numerical dataflow graphs. It makes it possible to train and run very large neural networks efficiently by distributing the computations across potentially thousands of multi-GPU servers. It was created at Google in 2015 and supports many of their large-scale deep learning applications.