Machine learning

Contents

[An introduction to Machine Learning 3](#_Toc523066626)

[What is Machine Learning? Why Machine Learning? 3](#_Toc523066627)

[What is Machine Learning? - Definition 6](#_Toc523066628)

[What’s a statistical model? 6](#_Toc523066629)

[What exactly is being learnt 7](#_Toc523066630)

[Desirable properties of machine learning 8](#_Toc523066631)

[Linear Regression Tutorial with Example 9](#_Toc523066632)

[Linear regression 11](#_Toc523066633)

[Simple linear regression 11](#_Toc523066634)

[Multiple linear regression 12](#_Toc523066635)

[Cost functions 13](#_Toc523066636)

[Example 13](#_Toc523066637)

[Residuals 14](#_Toc523066638)

[Why mean squared error? 15](#_Toc523066639)

[Optimization using Gradient Descent 16](#_Toc523066640)

[Footnotes 17](#_Toc523066641)

[Gradient Descent: An Intuitive Introduction 17](#_Toc523066642)

[Introduction and Overview 17](#_Toc523066643)

[Pseudocode for Gradient Descent 18](#_Toc523066644)

[Intuition for Gradient Descent 18](#_Toc523066645)

[Variants of Gradient Descent 19](#_Toc523066646)

[Choosing the learning rate 20](#_Toc523066647)

[Footnotes 21](#_Toc523066648)

[Types of Machine Learning problems: Supervised, Unsupervised and Reinforcement Learning 21](#_Toc523066649)

[Supervised Learning 21](#_Toc523066650)

[Classification and Regression 22](#_Toc523066651)

[Examples of Supervised Learning 22](#_Toc523066652)

[Clustering and Association Problems 22](#_Toc523066653)

[Semi-Supervised Learning 24](#_Toc523066654)

[Transfer Learning 24](#_Toc523066655)

[Algorithms for Supervised Learning 25](#_Toc523066656)

[Logistic Regression Model (for binary classification) 27](#_Toc523066657)

[Training the model / Optimization 28](#_Toc523066658)

[Making Predictions 29](#_Toc523066659)

[Example with Scikit-learn on predicting Diabetes 29](#_Toc523066660)

[K-nearest neighbors 33](#_Toc523066661)

[Understanding the classification algorithm (illustration) 33](#_Toc523066662)

[KNN as regression algorithm 36](#_Toc523066663)

[Tuning the hyper-parameter K 36](#_Toc523066664)

[Distance metrics 37](#_Toc523066665)

[Scikit-learn implementation 38](#_Toc523066666)

[Parametric and non-parametric models 39](#_Toc523066667)

[Support Vector Machine (SVM) 39](#_Toc523066668)

[Max-margin classifier 40](#_Toc523066669)

[Why maximum-margin? 41](#_Toc523066670)

[Multiclass SVM 50](#_Toc523066671)

[Naive Bayes (and text classification) 53](#_Toc523066672)

[Pseudocode 53](#_Toc523066673)

[Variants 53](#_Toc523066674)

[Applications 55](#_Toc523066675)

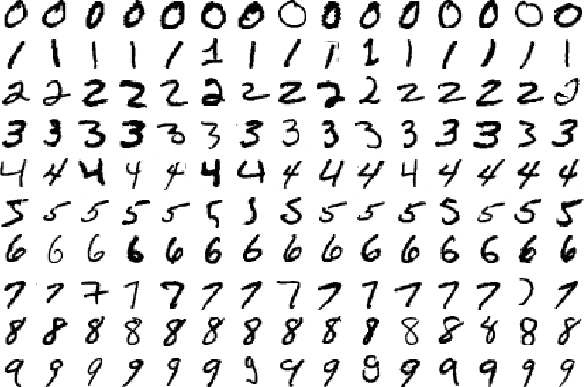
## An introduction to Machine Learning

We'll start by describing what machine learning is, and introduce a simple learning algorithm: linear regression + gradient descent. Using this algorithm, we'll introduce the core concepts in machine learning: model parameters, cost function, optimization method, and overfitting and regularization. This section ends with a visual review of these concepts and a tutorial on the different types of machine learning problems.

## What is Machine Learning? Why Machine Learning?

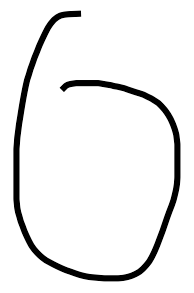
Motivation behind Machine Learning

Sometimes we encounter problems for which it's really hard to write a computer program to solve. For example, let's say we wanted to program a computer to recognize hand-written digits:



Source: MNIST handwritten database

You could imagine trying to devise a set of rules to distinguish each individual digit. Zeros, for instance, are basically one closed loop. But what if the person didn't perfectly close the loop. Or what if the right top of the loop closes below where the left top of the loop starts?



A zero that's difficult to distinguish from a six

In this case, we have difficulty differentiating zeroes from sixes. We could establish some sort of cutoff, but how would you decide the cutoff in the first place? As you can see, it quickly becomes quite complicated to compile a list of heuristics (i.e., rules and guesses) that accurately classifies handwritten digits.

And there are so many more classes of problems that fall into this category. Recognizing objects, understanding concepts, comprehending speech. We don't know what program to write because we still don't know how it's done by our own brains. And even if we did have a good idea about how to do it, the program might be horrendously complicated.

So instead of trying to write a program, we try to develop an algorithm that a computer can use to look at hundreds or thousands of examples (and the correct answers), and then the computer uses that experience to solve the same problem in new situations. Essentially, our goal is to teach the computer to solve by example, very similar to how we might teach a young child to distinguish a cat from a dog.

## What is Machine Learning? - Definition

The field itself: ML is a field of study which harnesses principles of computer science and statistics to create statistical models. These models are generally used to do two things:

Prediction: make predictions about the future based on data about the past

Inference: discover patterns in data

Difference between ML and AI

Machine Learning (ML) and Artificial Intelligence (AI) are highly interconnected fields, and there is no universally agreed upon distinction between the two.

However, in general when people say machine learning, they are referring to making machines (computers) learn certain patterns and then to make predictions using those learnt patterns. When people say artificial intelligence, they are usually referring to computers behaving intelligently. That is, AI makes use of machine learning. Hence, machine learning is a subset / a component of AI.

In addition, historically (1960s-1970s), people often used AI to refer to a technical field which focused on programming computers to make decisions (usually based on hardcoded rules), whereas ML focuses more on making predictions about the future (based on patterns learnt from data).

## What’s a statistical model?

Models: Teaching a computer to make predictions involves feeding data into machine learning models, which are representations of how the world supposedly works. If I tell a statistical model that the world works a certain way (say, for example, that the rent of a house grows with the number of rooms in the house), then this model can then tell me which house will have a higher rent, between one that has 2 rooms, and another which has 3 rooms.

I may believe, based on what I’ve seen, that a given house's rent is, on average, equal to the number of rooms times 1200, plus the number of restrooms times 400. That, is

Rent = Rooms × $1200 + Restrooms × $400

So, if it has 2 rooms and 1 restroom, then I’ll guess that the rent is probably $2800 / month. If is has 3 rooms and 2 restrooms, I think the rent is $4400 / month.

Here’s the main point: Machine learning refers to a set of techniques for estimating functions (like the one involving rent) based on datasets (room count, restroom count and rent for many many houses). These functions, which are called models, can then be used for predictions of future data.

Here, room count and restroom count are the features or variables, and rent is the target. In this problem, we have 2 features, but in general, we may have many (for example, size of rooms, the year the house was constructed, and so on).

## What exactly is being learnt

To explain what is being learnt in machine learning, let's start with an example application, spam classification. One approach to write a computer program to classify spam emails from non-spam emails, is to maintain a list of words that appear more frequently in spam emails. For example, some example of such words might be 'loan', '$', 'credit', 'discount', 'offer', 'password', 'viagra', and so on. When a new email comes in, we split the email into individual words, and if the email has a substantial number of these spammy words, it should be classified as spam.

Although the strategy above might give fairly good results (say detect spam with an accuracy of 80%), the accuracy depends in large part on the list of words we maintain, and on the precise threshold we choose to classify an email as spam.

In machine learning, the strategy is to learn the list of words and the threshold from examples. In fact, in addition to which words are considered spammy, we could also learn how spammy each word is. (This example is quite realistic, and is how many spam classification algorithms work.)

So in this case, the thing being learnt is, a notion of how spammy each word is. Note that this is not the only way to frame the problem. We framed the problem this way because we noticed a pattern that spam emails often contain specific words, and then we came up with a strategy that would analyze every possible word as a possible suspect. This strategy might give inaccurate results for other tasks, or be too inefficient.

## Desirable properties of machine learning

You might notice that using machine learning to learn how bad each word is has many desirable properties over maintaining this list manually.

1. It reduces the amount of manual work involved in creating the list. Think about how long this list could get if you try to do this manually. Also, if you're trying to maintain the list manually, how would you deal with hundreds of languages across the world? This task can easily become infeasible without machine learning.
2. The same strategy works for other similar tasks. Say we wanted to classify whether a movie review is speaking positively or negatively about a movie. If we were creating lists of words manually, then we would have to create a new list of words manually. But if we learn it, the same algorithm would work given that we already have some data (say ratings and reviews left by users on imdb).
3. It updates automatically. Lets say tomorrow the spammers become more advanced and start typing the word 'password' as 'passw0rd'. Or they might try to sell you insurance, something we haven't yet encountered. We can simply set the machine learning algorithm to be trained daily, and it will use the new data available and keep adapting over time to changing behavior.

# Linear Regression Tutorial with Example

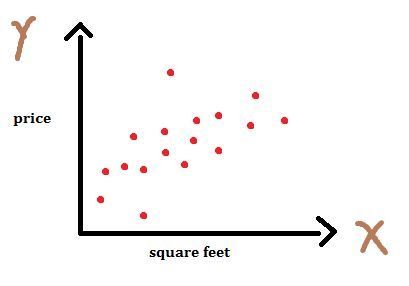
Linear Regression is a simple machine learning model for regression problems, i.e., when the target variable is a real value.

Example

Let's start with an example — suppose we have a dataset with information about the area of a house (in square feet) and its price (in thousands of dollars) and our task is to build a machine learning model which can predict the price given the area. Here is what our dataset looks like



If we plot our data, we might get something similar to the following:



# Linear regression

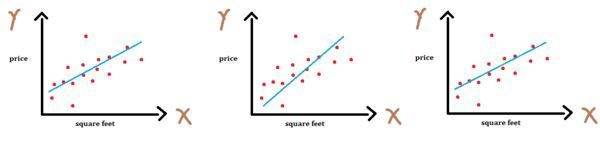
## Simple linear regression

In simple linear regression, we establish a relationship between target variable and input variables by fitting a line, known as the regression line.

In general, a line can be represented by linear equation y = m \* x + b. Where, y is the dependent variable, x is the independent variable, m is the slope, b is the intercept.

In machine learning, we rewrite our equation as y(x) = w0 + w1 \* x where w's are the parameters of the model, x is the input, and y is the target variable. This is the standard notation in machine learning, and makes it easier to add more dimensions. We can simply add variables w2, w3, ... and x2, x3, ... as we add more dimensions. (See footnote [1]).

Different values of w0 and w1 will give us different lines, as shown below



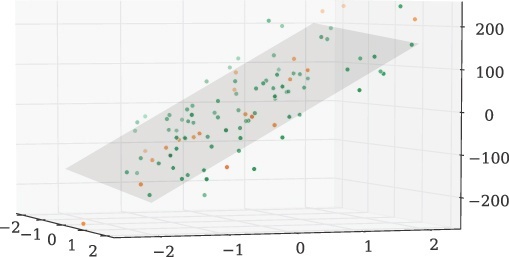
Each of the values of the parameters determine what predictions the model will make. For example, let's consider (w0, w1) = (0.0, 0.2), and the first data point, where x = 3456 and ytrue = 600. The prediction made by the model, y(x) = 0.0 + 0.2\*3456 = 691.2. If instead the weights were (w0, w1) = (80.0, 0.15), then the prediction would be y(x) = 80.0 + 0.15\*3456 = 598.4, which is much closer to the ytrue = 600.

## Multiple linear regression

The above equation can be used when we have one input variable (also called feature). However, in general, we usually deal with datasets which have multiple input variables. For example, in our dataset, we could have additional feature variables such as number of rooms in the house, the year the home was constructed, and so on.

The case when we have more than one feature is known as multiple linear regression, or simply, linear regression. We can generalize our previous equation for simple linear regression to multiple linear regression:

In the case of multiple linear regression, instead of our prediction being a line in 2-dimensional space, it is a hyperplane in n-dimensional space. For example, in 3D, our plot would look as follows



## Cost functions

Different values of the weights (w0, w1, w2, ... wn) gives us different lines (or hyperplanes), and our task is to find weights for which we get best fit.

One question you may have is, how can we determine how well a particular line fits our data? Or, given two lines, how do we determine which one is better? For this, we introduce a cost function which measures, given a particular value for the w's, how close the y's are to corresponding ytrue's. That is, how well do a particular set of weights predict the target value.

For linear regression, the most commonly used cost function is the mean squared error cost function. It is the average over the various data points (xi, yi) of the squared error between the predicted value y(x) and the target value ytrue.

## Example

Continuing the same example as above, the squared error for the first data point x = 3456 and ytrue = 600 when (w0, w1) = (0.0, 0.2) is given by (y(x) - ytrue)2 =

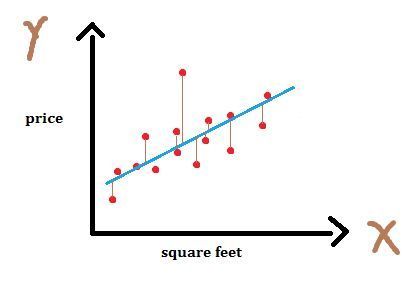
(691.2 - 600.0)2 = 91.22 = 8,317.44. Similarly, we calculate the squared error for each data point, and then average them. The squared error for the other two data points are 519.84 and 2621.44, which makes the mean squared error = J(w) = (8,317.44 + 519.84 + 2621.44)/3 = 3819.57.

Similarly, if we calculate the mean squared error for weights (w0, w1) = (80.0, 0.15) we get mean squared error = J(w) = (2.56 + 2.72 + 3648.16)/3 = 1217.81. Since the error J(w) is lower for (w0, w1) = (80.0, 0.15), we say those weights are better. [2]

The minimum error is achieved by (w0, w1) = (15.0, 0.17), and the corresponding error is J(w) = 395.83. In the next tutorial on gradient descent, we'll see how to find the weights which achieve the minimum error.

## Residuals

The cost function defines a cost based on the distance between true target and predicted target (shown in the graph as lines between sample points and the regression line), also known as the residual. The residuals are visualized below,



If a particular line is far from all the points, the residuals will be higher, and so will the cost function. If a line is close to the points, the residuals will be small, and hence the cost function.

## Why mean squared error?

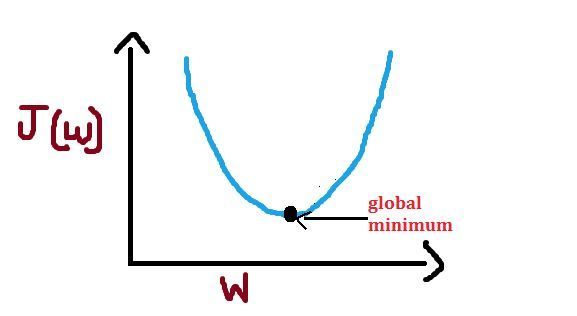
One question you might have is, why do we not use the sum of the residuals as our error function? Why squared? Why mean?

Squaring makes the existence of any "large" residuals negatively impact the cost function more than if a linear weight (not squared) was used. The result is a regression with more uniform residuals and less drastic outliers.

Mean so that the result is independent of the number of data points used. A sum would be proportional to the number of data points, while a mean is not. It makes comparison between data sets easier and the results more meaningful to when performing regressions in different problem spaces.

## Optimization using Gradient Descent

Each value of the weight vector w gives us a corresponding cost J(w). We want to find the value of weights for which cost is minimum. We can visualize this as follows:



Note: Above we have used the word "global" because the shape of the cost-function for linear regression is convex (i.e. like a bowl). It has a single minimum, and it smoothly increases in all directions around it.

Given the linear regression model and the cost function, we can use Gradient Descent (covered in the next article) to find a good set of values for the weight vector. The process of finding the best model out of the many possible models is called optimization.

## Footnotes

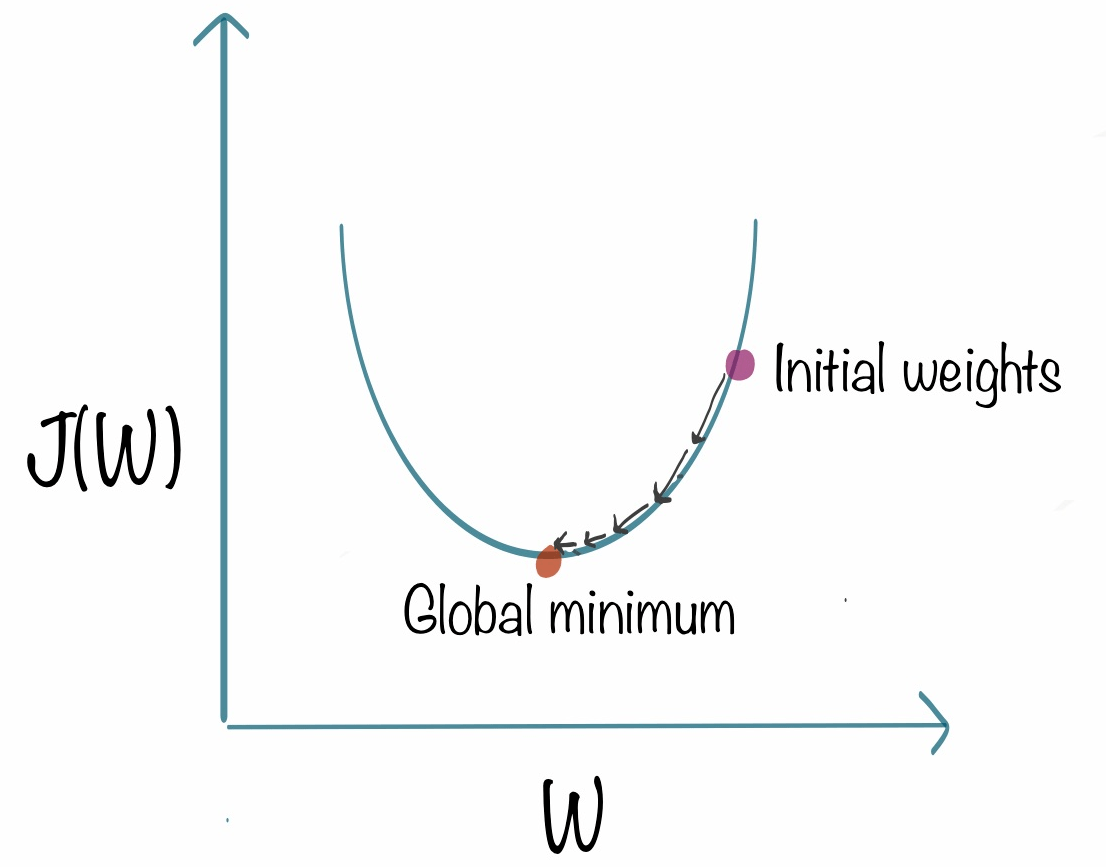
In addition, variables b and w0 are used interchangeably. Often when the w0 notation is used, we add an imaginary dimension x0 to the input, which is always equal to 1. This makes the dot product having a special case for dimension 0. However, there are cases when needs to be treated differently from, such as during regularization (penalizing large weights). In which case, using the

notation can be confusing.Calculations for predictions, residual, squared error and cost function can be found on this Google Spreadsheet.

# Gradient Descent: An Intuitive Introduction

## Introduction and Overview

Gradient Descent is one of the most popular and widely used optimization algorithms. Given a machine learning model with parameters (weights and biases) and a cost function to evaluate how good a particular model is, our learning problem reduces to that of finding a good set of weights for our model which minimizes the cost function.



Gradient descent is an iterative method. We start with some set of values for our model parameters (weights and biases), and improve them slowly. To improve a given set of weights, we try to get a sense of the value of the cost function for weights similar to the current weights (by calculating the gradient) and move in the direction in which the cost function reduces. By repeating this step thousands of times we'll continually minimize our cost function.

## Pseudocode for Gradient Descent

Gradient descent is used to minimize a cost function J(w) parameterized by a model parameters w. The gradient (or derivative) tells us the incline or slope of the cost function. Hence, to minimize the cost function, we move in the direction opposite to the gradient.

## Intuition for Gradient Descent

Imagine you're blind folded in a rough terrain, and your objective is to reach the lowest altitude. One of the simplest strategies you can use, is to feel the ground in every direction, and take a step in the direction where the ground is descending the fastest. If you keep repeating this process, you might end up at the lake, or even better, somewhere in the huge valley.



The rough terrain is analogous to the cost function. Minimizing the cost function is analogous to trying to reach lower altitudes. You are blind folded, since we don't have the luxury of evaluating (seeing) the value of the function for every possible set of parameters. Feeling the slope of the terrain around you is analogous to calculating the gradient, and taking a step is analogous to one iteration of update to the parameters.

## Variants of Gradient Descent

There are multiple variants of gradient descent depending on how much of the data is being used to calculate the gradient. The main reason behind these variations is computational efficiency. A dataset may have millions of data points, and calculating the gradient over the entire dataset can be computationally expensive.

Batch gradient descent computes the gradient of the cost function w.r.t to parameter w for entire training data. Since we need to calculate the gradients for the whole dataset to perform one parameter update, batch gradient descent can be very slow.

Stochastic gradient descent (SGD) computes the gradient for each update using a single training data point xi (chosen at random). The idea is the gradient calculated this way is a stochastic approximation to the gradient calculated using the entire training data. Each update is now much faster than in batch gradient descent, and over many updates, we will head in the same general direction.

In mini-batch gradient descent, we calculate the gradient for each small mini-batch of training data. That is, we first divide the training data into small batches (say M samples / batch). We perform one update per mini-batch. M is usually in the range 30-500, depending on the problem. Usually mini-batch GD is used because computing infrastructure - compilers, CPUs, GPUs - are often optimized for performing vector additions and vector multiplications.

Of these, SGD and mini-batch GD are most popular. In a typical scenario, we do several passes over the training data before the termination criteria is met. Each pass is called an epoch. Also note that since the update step is much more computationally efficient in SGD and mini-batch GD, we typically perform 100s-1000s of updates in between checks for termination criteria being met.

## Choosing the learning rate

Typically, the value of the learning rate is chosen manually. We usually start with a small value such as 0.1, 0.01 or 0.001 and adapt it based on whether the cost function is reducing very slowly (increase learning rate) or is exploding / being erratic (decrease learning rate).

Although manually choosing a learning rate is still the most common practice, several methods such as Adam optimizer, AdaGrad and RMSProp have been proposed to automatically choose a suitable learning rate. Suggested further reading for going deeper into this topic: An overview of gradient descent optimization algorithms.

## Footnotes

The value of the gradient depends on the cost function and the inputs, and you might need to revisit the topic of differentiation if you are calculating them by hand.

It is common for machine learning software libraries to calculate the gradients automatically once the cost function is specified. However, if one is implementing a novel machine learning algorithm and applying the gradient descent optimization technique on it, it is important to have at-least a rough idea of what the gradients look like in-order to make sure that the method works as desired and does not get stuck / behave erratically.

# Types of Machine Learning problems: Supervised, Unsupervised and Reinforcement Learning

## Supervised Learning

Currently, most of the machine learning products use supervised learning. In this, we have a set of features or inputs X (for example, an image) and our model will predict a target or output variable y (for example, caption for the image).

In other words, our model learns a function that maps inputs to desired outputs. Features are independent variables and targets are the dependent variable.

## Classification and Regression

Supervised learning problems can be further grouped into classification and regression problems. When the output variable is a category, such as "spam" or "ham" (non-spam) then the problem is a classification problem. When the output variable is a real value, such as "price of the house", then it is a regression problem.

## Examples of Supervised Learning

Classification

Spam filtering: Is an email spam or not

Image classification: Given an image, output which objects are present in the image (dog, cat, computer, building, so on)

Regression

Given information about a house, predict its price

Netflix: Given a user and a movie, predict the rating the user is going to give to the movie (which can then be used for providing recommendations)

Unsupervised Learning

In unsupervised learning we have input data X but no output variable y. Goal of unsupervised learning is to model the distribution of the data, i.e. identify patterns in the data.

## Clustering and Association Problems

Unsupervised learning problems can be further grouped into clustering and association problems. When we want to discover inherent groupings in the input data it is known as clustering problem. When we want to discover rules that describe portions of the input data it is known as an association problem.

Examples of Unsupervised Learning

Clustering

Given a list of customers and information about them, discover groups of similar users. This knowledge can then be used for targeted marketing.

Anomaly detection: Given measurements from sensors in a manufacturing facility, identify anomalies, i.e. that something is wrong

Association

Discover patterns in data such as whenever it rains, people tend to stay indoors. When it is hot, people buy more ice-cream.

Reinforcement Learning

In the above two problem categories, the input is given to us. In reinforcement learning, the key difference is that the input itself depends on actions we take. For example, in robotics, we might start in a situation where the robot does not know anything about the surrounding it is in. As it does certain actions, it finds out more about the world. But the world it sees depends on whether it chose to move forward, or turn right.

The robot is known as an agent, and is in some environment (surrounding). At each time step, it can take some action and it might receive some reward (say the robot fell in a ditch, or found a lake on Mars).

Example reinforcement learning problems:

Robotics: A robot is in a maze, and it needs to find a way out.

Training an AI for a complex game such as Civilization or Dota

Other problem types

Problems may not necessarily fall into one of the above categories cleanly.

## Semi-Supervised Learning

For example, in the image classification problem, we are given a number of images and the objects present in those images as training data. However, we may have a strategy for using the large amount of images available on the web (for which the objects have not been annotated). This is an example of semi-supervised learning, i.e. we have some data that is labelled, and some that is not labelled.

## Transfer Learning

Another example is, we might want to learn a model that given an image of a handwritten sentence, tells us what the sentence is (this problem is called Optical Character Recognition). However, we have only 100 images of sentences and the corresponding label, but we also have 100,000 images of alphabets and their corresponding labels. Then, we could learn a model on the alphabets, and re-use the model in some way when learning a model on the sentences. This concept is called transfer learning (as in, we are transferring our knowledge from one domain to another).

Note: 'Problem type' is often fluid

We should note that the boundaries between the above problem types are quite fluid. For example, a supervised learning problem could become a semi-supervised or transfer learning problem if we find we have more data that can help our model in some way, even though it doesn't look exactly like the data for the problem we have at hand.

Note: Current state of machine learning

We've made a lot more progress towards solving supervised learning than unsupervised learning. That is, on average, there are a lot more problems for which supervised machine learning algorithms make accurate predictions, than

there are unsupervised machine learning algorithms or reinforcement learning algorithms.

A majority of the successful machine learning products currently fall under the category of supervised learning. Such as, predicting what rating a user might give to a movie (think Netflix), how likely is a person to buy a particular product (think Amazon), or how likely is if for a particular email to be spam.

Unsupervised and reinforcement learning are areas of active research, and we've recently made significant progress in both with algorithms such as Generative Adversarial Networks (for unsupervised learning) and Deep Q-networks (for reinforcement learning).

# Algorithms for Supervised Learning

In this section, we'll see various algorithms for supervised machine learning. Different algorithms perform better for different types of data, where deciding factors would include total number of dimensions in input data, whether the data is text or numerical or a time series, whether or not the data is sparse, size of dataset, and so on. For each algorithm, we'll explain how it works and the applications for which it produces state-of-the-art results.

The algorithms don't depend on each other and require different amounts of mathematical maturity. If you have trouble with some of the mathematically heavy ones (such as SVMs), feel free to skip or take extra time.

Logistic Regression[ Edit ]

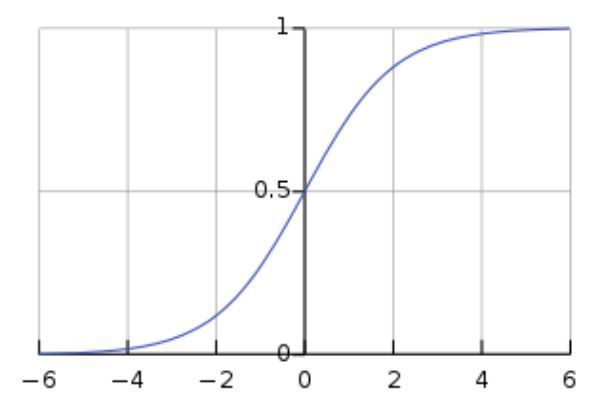
Logistic Regression is a variant of linear regression where dependent or output variable is categorical, i.e. it takes out of a few possible discrete values. Don't be confused by the name logistic regression, it's a classification algorithm.

In particular, we can use it for binary classification (two categories). For example, we might want to predict whether or not a person has diabetes, or whether or not an email is spam.

The logistic (or sigmoid) function

The term logistic in logistic regression comes from the logistic function (also known as sigmoid function), which can be written as:

The following is what we get if we plot f(z):



Plot of logistic (sigmoid) function. As we can see, the sigmoid function squashes the input value between [0, 1]. For large negative values, the output is very close to 0, and the large positive values, the output is very close to 1. Since the range of the output is between 0 and 1, we can interpret the output as a probability.

The logistic function also has the desirable property that it is a differentiable function. Hence, we can train the machine learning model using gradient descent.

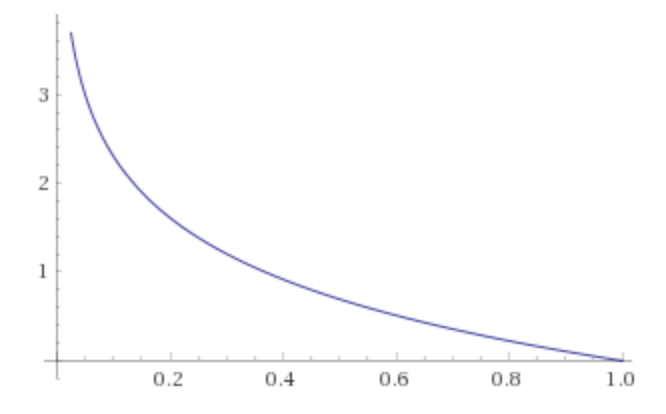
## Logistic Regression Model (for binary classification)

In linear regression we have a linear equation ( y = w0 + w1x1 ) as our hypothesis. Since y can take any arbitrarily large negative or positive value, linear regression is not a good choice when output is a binary categorical variable. For example, y = spam or not spam.

In logistic regression, the output yw(x) is squashed by a sigmoid function, i.e.

Plot

The figure below is a plot of negative log probability. As we can see, this cost is high when the target class is assigned a low probability, and is 0 if the assigned probability is 1.



plot of -log(p) for p in range 0 to 1

## Training the model / Optimization

We use gradient descent to optimize the model. In fact, the cost function above is chosen so that the gradients dL/dw we get are meaningful. In fact, if you do the algebra and derive the mathematical formula for the gradients (we'll skip this since the algebra is quite messy and doesn't add much to the conceptual knowledge), you'll find that the gradients are exactly the same as in linear regression (even though the predictions are not the same, and the cost is not the same). Roughly speaking, the log in the cost function 'undoes' the sigmoid function.

## Making Predictions

After training, we can predict the class by calculating probability of each class. The prediction will be the class with the highest probability, i.e. if yw(x) > 0.5 then the class is 1 otherwise 0.

## Example with Scikit-learn on predicting Diabetes

In this section, we'll see an example for using logistic regression. We'll use the Pima Indians Diabetes Database, where all patients belong to the Pima Indian heritage (subgroup of Native Americans), and are females of ages 21 and above.

import pandas as pd

import numpy as np

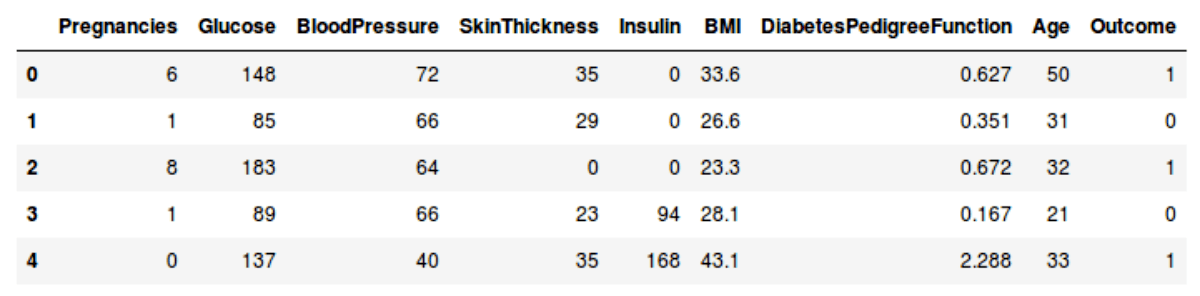
import matplotlib.pyplot as plt

from sklearn.linear\_model import LogisticRegression

## load the data

diabetesDF = pd.read\_csv('diabetes.csv')

print(diabetesDF.head())



First 5 records in the Pima Indians Diabetes Database

Outcome is whether or not the patient is diabetic. 0 denotes non-diabetic, and 1 is diabetic.

## split and normalize the data

# split into train and test

dfTrain = diabetesDF[:650]

dfTest = diabetesDF[650:750]

dfCheck = diabetesDF[750:]

# split features from target variable

trainLabel = np.asarray(dfTrain['Outcome'])

trainData = np.asarray(dfTrain.drop('Outcome',1))

testLabel = np.asarray(dfTest['Outcome'])

testData = np.asarray(dfTest.drop('Outcome',1))

# normalize the data

# makes it easier to interpret the model by looking at its weights

means = np.mean(trainData, axis=0)

stds = np.std(trainData, axis=0)

trainData = (trainData - means)/stds

testData = (testData - means)/stds

## train and evaluate the model

# models target t as sigmoid(w0 + w1\*x1 + w2\*x2 + ... + wd\*xd)

diabetesCheck = LogisticRegression()

diabetesCheck.fit(trainData, trainLabel)

accuracy = diabetesCheck.score(testData, testLabel)

print("accuracy = ", accuracy \* 100, "%")

# prints "accuracy = 78.0%"

## interpreting the model

coeff = list(diabetesCheck.coef\_[0])

labels = list(dfTrain.drop('Outcome',1).columns)

features = pd.DataFrame()

features['Features'] = labels

features['importance'] = coeff

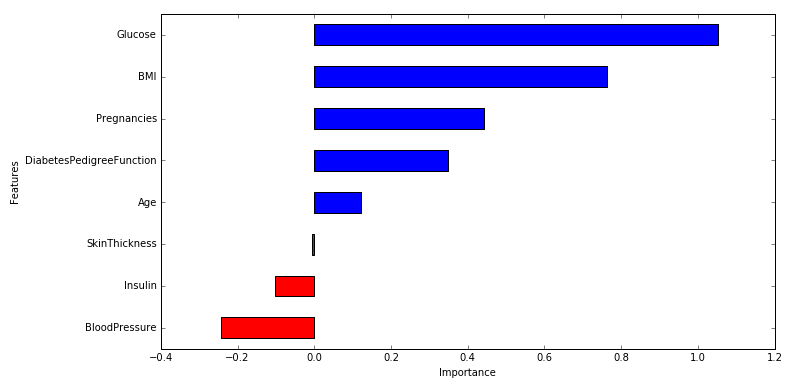
features.sort\_values(by=['importance'], ascending=True, inplace=True)

features['positive'] = features['importance'] > 0

features.set\_index('Features', inplace=True)

features.importance.plot(kind='barh', figsize=(11, 6),color = features.positive.map({True: 'blue', False: 'red'}))

plt.xlabel('Importance')



Visualization of the weights in the Logistic Regression model corresponding to each of the feature variables

Notice how the model assigns largest weights corresponding to features glucose and BMI. It is good to see the machine learning model match what we have been hearing from doctors our entire lives!

## making predictions

sampleData = dfCheck[:1]

# prepare sample

sampleDataFeatures = np.asarray(sampleData.drop('Outcome',1))

sampleDataFeatures = (sampleDataFeatures - means)/stds

# predict

predictionProbability = diabetesCheck.predict\_proba(sampleDataFeatures)

prediction = diabetesCheck.predict(sampleDataFeatures)

print('Probability:', predictionProbability)

print('prediction:', prediction)

The output produced by the above code is

Probability: [[ 0.4385153, 0.5614847]]

prediction: [1]

That is, the model thinks where is a 56.14% chance that the person is diabetic

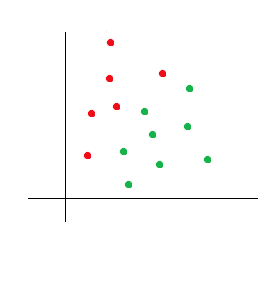
# K-nearest neighbors

K-nearest neighbors (KNN) is one of the simplest Machine Learning algorithms. It is a supervised learning algorithm which can be used for both classification and regression.

## Understanding the classification algorithm (illustration)

Let us understand this algorithm with a classification problem. For simplicity of visualization, we'll assume that our input data is 2 dimensional. We have two possible classes - green and red.

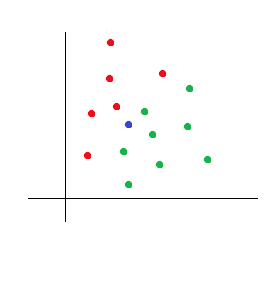
Lets plot out training data in feature space.



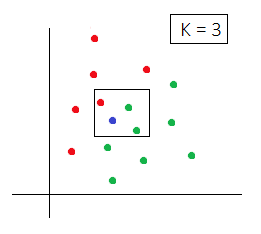
There is no explicit training phase in KNN! In other words, for classifying new data points, we'll directly use our dataset (in some sense, the dataset is the model).

To classify a new data point, we find the k points in the training data closest to it, and make a prediction based on whichever class is most common among these k points (i.e. we simulate a vote). Here closest is defined by a suitable distance metric such as euclidean distance. Other distance metrics are discussed below.

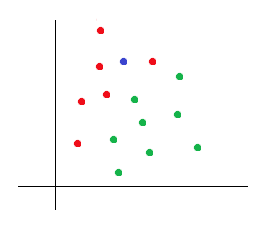
For example, if we want to classify blue point as shown in following figure, we consider k nearest data points and we assign the class which has the majority.



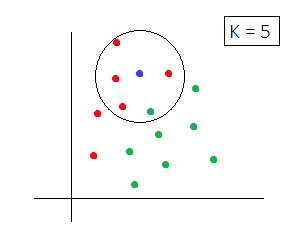
If k = 3, we get two data points with green class and one data point with red class. Hence, we'll predict green class for the new point.



Here's another example, let us change the position of new point (blue point) as shown below.



If we take k = 5 then we get four neighbors with red class and one neighbor with green class. Hence, new point will be classified as red point.



## KNN as regression algorithm

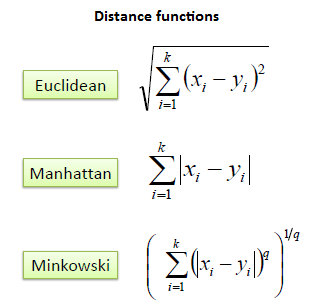
In case of regression (when target variable is a real value), we take the average of the K nearest neighbors.

## Tuning the hyper-parameter K

A small value of k means that noise will have a higher influence on the result and large value make the algorithm computationally expensive. Usually, we perform cross-validation to find out best k value (or to choose the value of k that best suits our accuracy / speed trade-off). If you don't want to try multiple values of k, a rule of thumb is to set k equal to the square root of total number of data points. For more on choosing best value of k, refer this stackoverflow thread.

## Distance metrics

There are various options available for distance metric such as euclidian or manhattan distance. The most commonly used metric is euclidian distance.



Minkowski is the generalization of Euclidian and Manhattan distance.

Note that you'll want to do some pre-processing on the input data (for example, make sure each dimension has 0 mean and unit variance) so that the distance metrics above are meaningful.

## Scikit-learn implementation

## load the dataset

from sklearn.datasets import load\_iris

dataset = load\_iris()

X = dataset.data

y = dataset.target

## precessing

# standardize the data to make sure each feature contributes equally

# to the distance

from sklearn.preprocessing import StandardScaler

ss = StandardScaler()

X\_processed = ss.fit\_transform(X)

## split the dataset into train and test set

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_processed, y, test\_size=0.3, random\_state=42)

## fit n nearest neighbor model

from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier(n\_neighbors = 5, metric="minkowski", p=2)

# p=2 for euclidian distance

model.fit(X\_train, y\_train)

# output:

# KNeighborsClassifier(algorithm='auto', leaf\_size=30, metric='minkowski',

# metric\_params=None, n\_jobs=1, n\_neighbors=5, p=2,

# weights='uniform')

## evaluate

model.score(X\_test, y\_test)

# output: 1.0

## Parametric and non-parametric models

In a parametric model, we continuously update a fixed number of parameters to learn a function which can classify new data point without requiring the training data (for example, logistic regression). In a non-parametric model, the number of parameters grows with the size of training data. This is what happens in KNN.

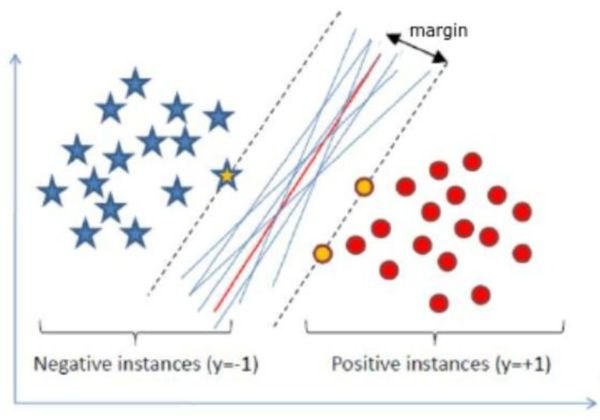
# Support Vector Machine (SVM)

A Support Vector Machine (SVM) is a classification algorithm, typically used for binary classification. An SVM with gaussian kernel has been consistently shown to be one of the best machine learning models, achieving the highest accuracy in a large variety of datasets, specially datasets which do not involve images or audio.

Understanding the various hyper-parameters of an SVM are central to achieving high accuracy. In this tutorial, we'll learn what an SVM is, and what is the purpose of its various hyper-parameters.

## Max-margin classifier

Given a binary classification dataset, an SVM aims to find a separating hyperplane between positive and negative instances. For example, in the figure below, each of the blue lines is a plane which separates the positive and negative data points. All blue points lie on one side, and all red points lie on the other rise. New unseen samples are categorized based on the side of the hyperplane in which they fall.



Each of the blue lines (separating hyperplanes) classify all data correctly, but the red line is the max-margin classifier. Image source: ResearchGate

Although there are many possible hyperplanes that separate all data points correctly, the SVM algorithm seeks the hyperplane with the largest margin. That is, it maximizes the distance between the hyperplane and the nearest data point.

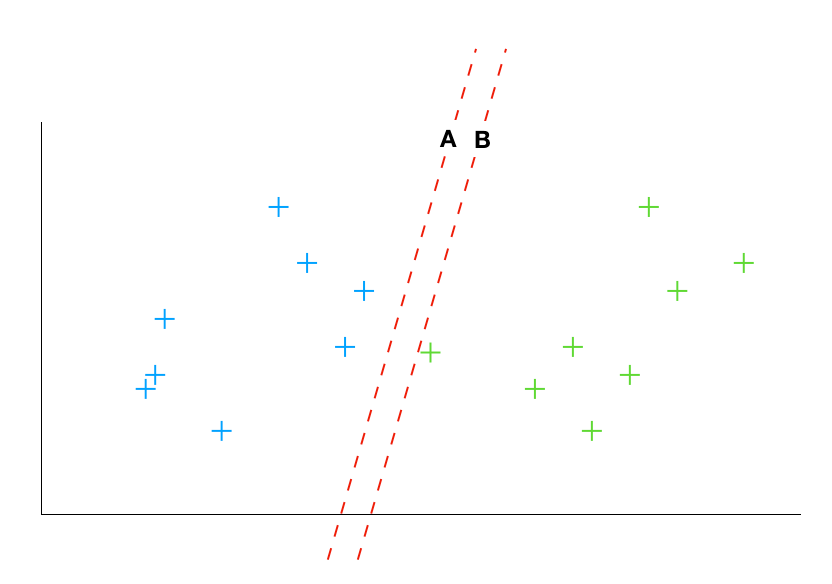
Hence, it is known as a maximum-margin classifier. The closest data points are called support vectors.

In the illustration above, the red line is the maximum-margin classifier, and the data points marked in the yellow are the corresponding support vectors.

## Why maximum-margin?

The main reason for preferring the max-margin classifier over other hyperplanes is that intuitively, it is expected to perform better on test data / unseen data.

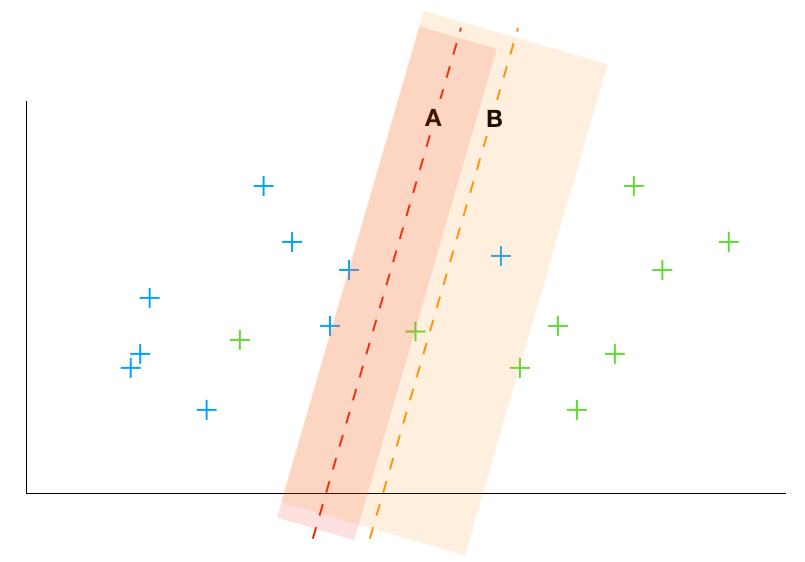
Consider the following example, if new data points were added to this dataset, we would expect classifier A to perform better than classifier B, since classifier B is very close to one of the green data points.



What if data is overlapping?

Although linearly separable datasets are nice, in practice those are difficult to find. For overlapping datasets, the SVM algorithm includes a regularization parameter (usually denoted as C), which controls how important it is to avoid misclassifying data points.

Large values of C imply that the SVM classifier will classify more points correctly, even if that means a small margin. Whereas small values of C imply that more data points might be misclassified, but the resulting classifier will have a larger margin with the rest of the data points.

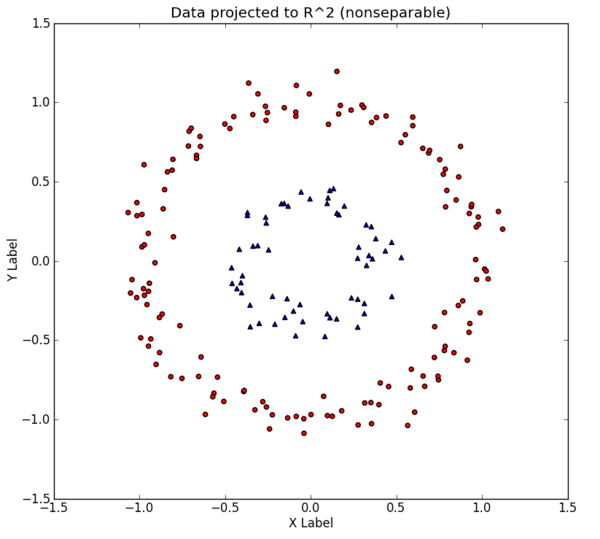


Classifier A misclassifies 2 points and has a small margin. Classifier B misclassifies 3 points, but has a larger margin

In general, if the dataset has more outliers, we want to allow for more data points to be misclassified. In practice, we try many different values for the hyper-parameter C, and choose the one that performs best on the validation dataset.

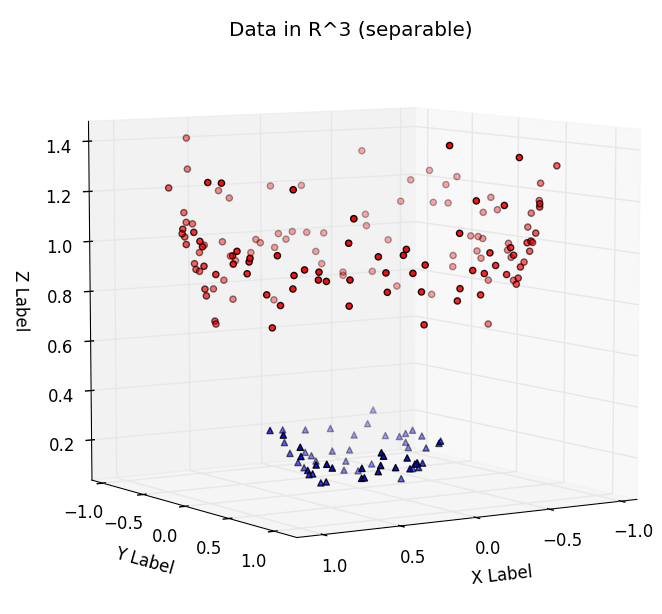
What if optimal classifier is not linear?

One limitation of an SVM (even with soft-margin) is that the classification boundary is linear. However, in real-life datasets, the best classification boundary might not be a straight line. For example, consider the dataset below. Clearly, the best decision boundary in this case is a circle.



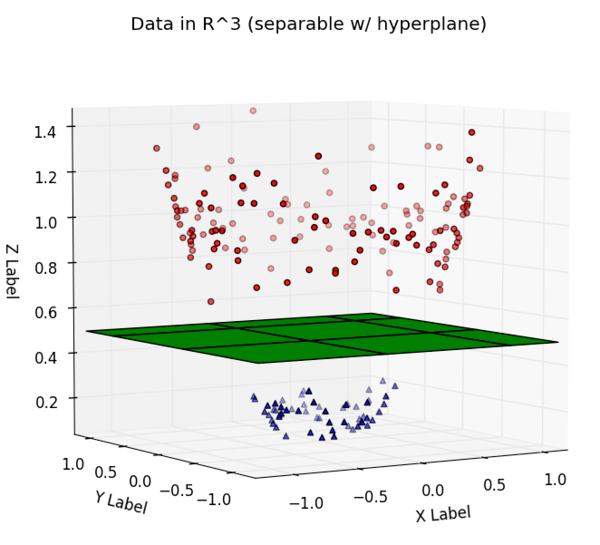
2D data where best decision boundary is a circle (non-linear). Image source: eric-kim.net

Now, when we plot the data, we get the following plot:



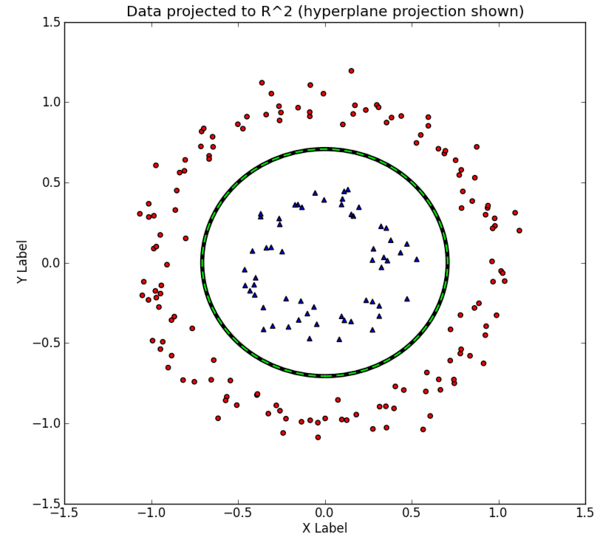
2D data mapped to 3D

Now, it is possible to separate the data with a linear hyperplane.



Data projected to 3D and the linear hyperplane that separates it.

Finally, we can also look at what the decision boundary looks like if the project it back to our original 2D dataset.



Hyperplane projected back on to the 2D dataset

As we can see, the learnt decision boundary is a circle. In summary, projecting the data to a higher dimension allows us to learn classifiers which are non-linear with respect to the original data.

Kernel trick in practice

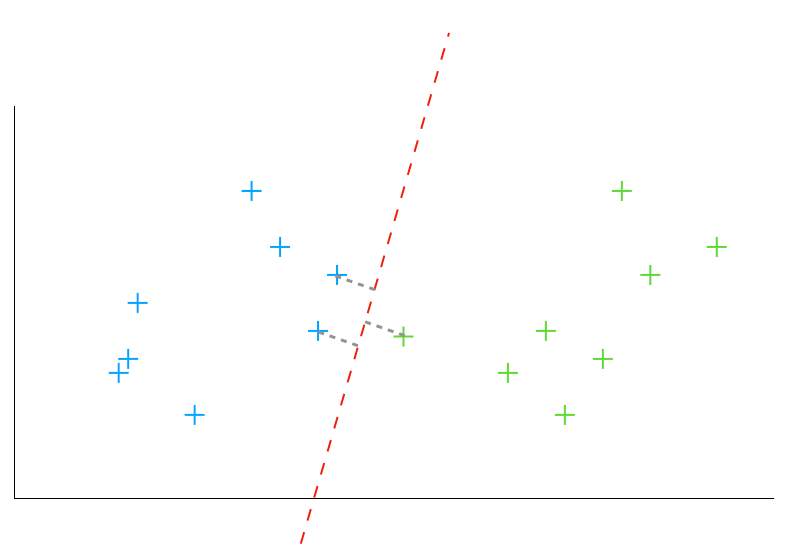
There are two more important things to know about SVM kernels.

SVM kernels provide a way to perform the transformation on the data implicitly. That is, we need not go through every data point and calculate the values for the new dimensions. For a restricted set of transformations, the SVM optimizer is able to implicitly work in such a high-dimensional space by looking at the dot products between pair of samples. This makes things a lot more computationally efficient when the number of higher dimensions is much much larger.

We have figured out a set of kernels which are extremely powerful and work very well on a large number of datasets. In particular, the polynomial kernel and gaussian kernel (also known as radial basis function) are the most popular ones.

Adjusting the number of support vectors - Gamma

Finally, SVM provides us with a third powerful hyper-parameter (apart from regularization parameter C and choice of kernel). This hyper-parameter, typically denoted by



γ

(gamma), controls how far the influence of each data point reaches.

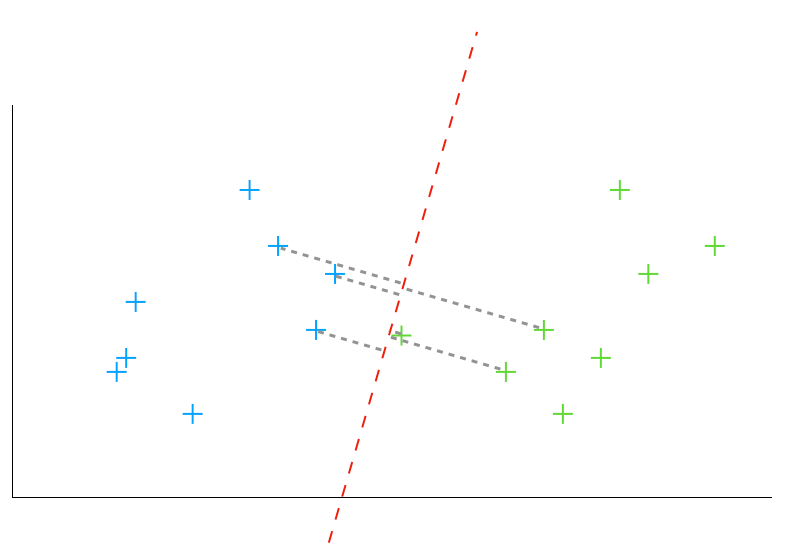
That is, high values of

γ

imply a small influence and hence a small number of support vectors. Whereas low values of

γ

imply a large influence and hence a large number of support vectors.



Classifier obtained with a large gamma value implies a small number of support vectors

Classifier obtained with a small gamma value implies a large number of support vectors

## Multiclass SVM

So far, only the binary classification model was described. SVM can be extended to be used on a multi-class dataset. There are two popular options:

One-versus-rest: train one SVM classifier per class to distinguish one single class from the rest. For predictions, we follow the winner-takes-all strategy. That is, new instances will be categorized based on the highest scoring output.

One-versus-one: train one SVM classifier for each pair of classes. New samples are predicted following the max-wins voting strategy. That is, each classifier assigns the instance to one of the two classes, and the final prediction is given by the class with the most votes.

Putting things together (and Code)

You can see the documentation for SVMs in the scikit-learn machine learning library here: sklearn.svm.SVC — scikit-learn documentation. The following is the list of arguments the function can take:

class sklearn.svm.SVC(C=1.0, kernel=’rbf’, degree=3, gamma=’auto’,

coef0=0.0, shrinking=True, probability=False, tol=0.001,

cache\_size=200, class\_weight=None, verbose=False, max\_iter=-1,

decision\_function\_shape=’ovr’, random\_state=None)

Out of the above arguments, the following are the most important ones (most already discussed in the tutorial).

Model hyper-parameters

C = regularization parameter = controls how important it is to avoid misclassifying data points

kernel = 'rbf' (radial basis function, i.e. Gaussian kernel) and 'poly' (polynomial kernel) are most popular

degree = only applicable for polynomial kernel.

gamma = controls the range of influence of each data point

Stopping criterion for training

tol = tolerance for stopping

max\_iter = hard limit for number of iterations

Multiclass SVM:

decision\_function\_shape = 'ovo' (one-versus-one) or 'ovr' (one-versus-rest)

Sample interaction

Here's a sample interaction for training the SVM

>>> # source: scikit-learn

>>> import numpy as np

>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])

>>> y = np.array([1, 1, 2, 2])

>>> from sklearn.svm import SVC

>>> clf = SVC()

>>> clf.fit(X, y)

SVC(C=1.0, cache\_size=200, class\_weight=None, coef0=0.0,

decision\_function\_shape='ovr', degree=3, gamma='auto', kernel='rbf',

max\_iter=-1, probability=False, random\_state=None, shrinking=True,

tol=0.001, verbose=False)

>>> print(clf.predict([[-0.8, -1]]))

Applications

SVM with Gaussian kernel has been consistently shown to be one of the best machine learning models for a large variety of datasets (specially non-media datasets, where the input isn't an image or an audio).

Theory

If you'd like to read about the theoretical background behind SVMs, check out our tutorial on Support Vector Machine (SVM) Theory.

# Naive Bayes (and text classification)

Naive Bayes is a widely used classification algorithm. It is a supervised learning algorithm based on Bayes’ Theorem. The word naive comes from the assumption of independence among features. That is, if our input vector is (x1, x2,...,xn), then xi's are conditionally independent given y.

## Pseudocode

Training:

Estimate P(y): P(y=t) = number of times class t appears in the dataset / size of dataset

Estimate P(xi|y): P(xi=k|y=t) = number of times xi has value k and y has value t / number of data points of class t

Predicting:

Estimate P(y|x1,...,xn): Use above estimated values of P(y) and P(xi|y) and equation (1). Thereafter, normalize the values.

## Variants

There are several variants of naive bayes which use different distributions for P(xi|y) such as gaussian distribution (gaussian naive bayes), multinomial distribution (multinomial naive bayes) and bernoulli distribution (bernoulli naive bayes).

Scikit-learn implementation

# we will use iris dataset

from sklearn.datasets import load\_iris

from sklearn.naive\_bayes import GaussianNB

import numpy as np

# load the dataset

data = load\_iris()

model = GaussianNB()

model.fit(data.data, data.target)

# evalaute

print(model.score(data.data, data.target))

# output = 0.96

# predict

model.predict([4.2, 3, 0.9, 2.1])

# 0 = setosa,1 = versicolor, and 2 = virginica

## Applications

Naive bayes is one of the simplest yet effective algorithms for

Text classification: For example, we have a number of news articles, and we want to learn to classify if the article is about politics, health, technology, sports or lifestyle.

Spam filtering: We have a number of emails, and we want to learn to classify if the email is spam or not.

Gender classification: Given features such as height, weight, etc, predict whether the person is male or female.