

Introduction to Machine Learning Algorithms

Siddharth Jain



PREFACE

Purpose/Goals

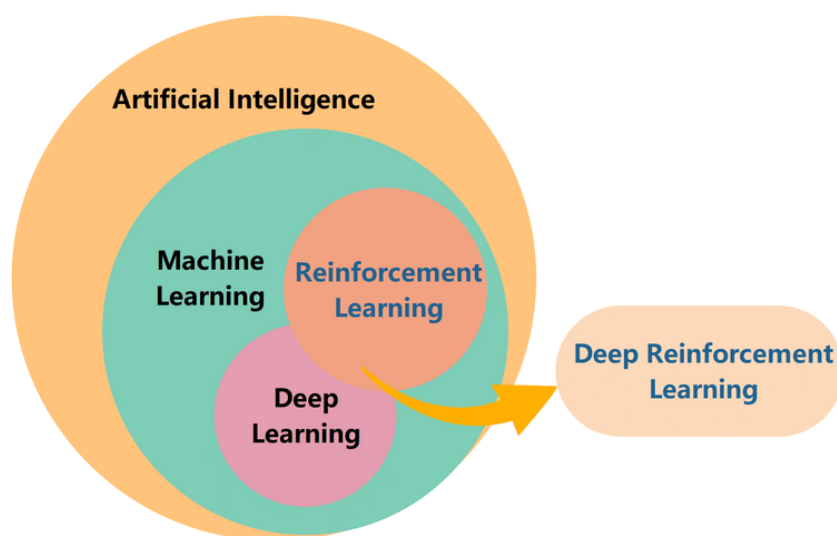
Algorithms are a big part in the field of machine learning. You need to understand what algorithms are out there, and how to use them effectively.

This book describes the frequently used algorithms used in Machine Learning and Data Science that can be directly or indirectly linked to basic sorting, graph theory, DP and number theory algorithms.

Overview

Machine learning is a buzzword for today's technology, and it is growing very rapidly day by day. We are using machine learning in our daily life even without knowing it such as Google Maps, Google assistant, Alexa, etc.

We will start by discussing the difference between some popular terms in data sciences like DL,ML,AI etc and then focus on how fundamental algorithms and concepts are widely used in areas of ML algorithms and its various subsets like RL.



Contents

1. Introduction

- i. Difference between AI, ML, DL
- ii. What is Data Science
- iii. What is Artificial Intelligence
- iv. What is Machine Learning
- v. What is Deep Learning

2. Graph Theory

- i. Connected Components
- ii. Shortest Path
- iii. MST
- iv. Pagerank
- v. Centrality Measures
- vi. A* Algorithm

3. Dynamic Programming

- i. What is Dynamic Programming
- ii. Dynamic programming in Reinforcement Learning
- iii. Reinforcement Learning and Markov Decision Process
- iv. DP Algorithms
- v. Q-Learning: A Paradigm In Dynamic Programming

4. Sorting Algorithms

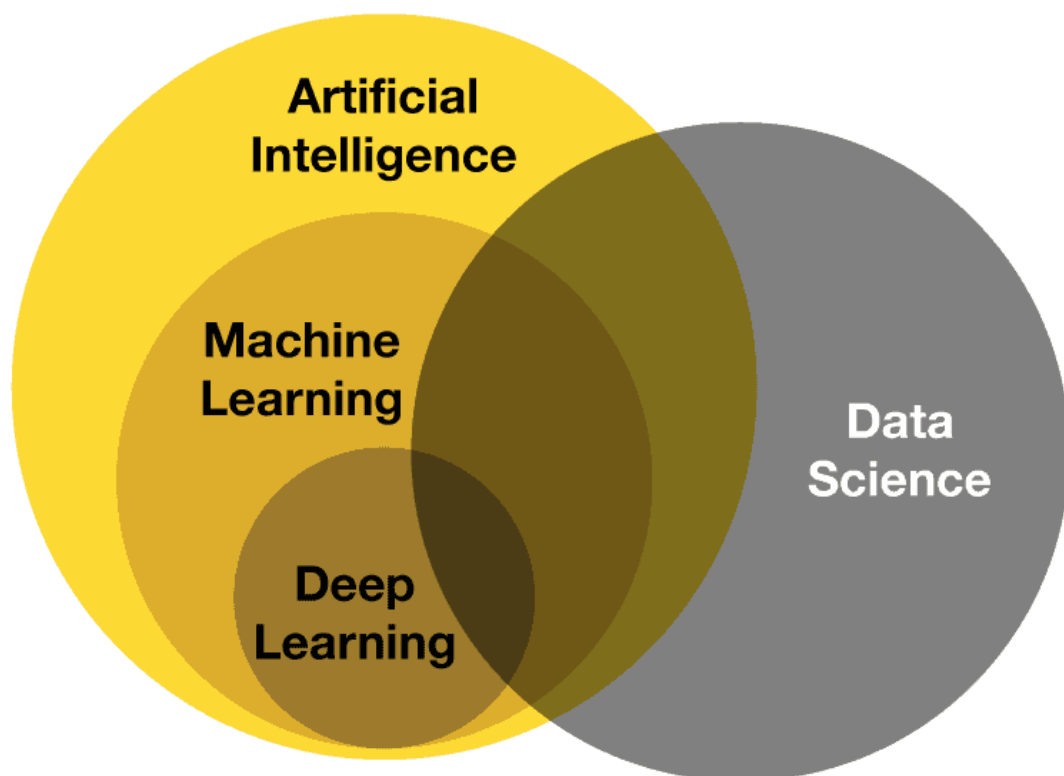
- i. Support vector Machine
- ii. Convex Hull
- iii. How to determine Convex Hull: Graham's Scan

5. References

Introduction

Difference between AI, ML, DL: How to not get them mixed!

In this day, Artificial Intelligence, Machine Learning, Deep Learning, and Data Science are all buzzwords. It's more important than ever to understand what it is and how it differs from others. Although these concepts are similar, there are variations between them; see the graphic below for a visual representation.



Let's see a brief and simple introduction about all the above jargons and explain what they really are.

What is Data Science?

Data science is a vast branch of research that focuses on data systems and processes with the goal of preserving data collections and extracting information from them.

To keep up with the ever-growing data collection, data science focuses on data modelling and data warehousing. Data science

applications collect information that is used to influence business operations and achieve organizational goals.

For instance, Data science algorithms are used to determine virtually anything, from **display banners** on various websites to **digital billboards** at airports.

What is Artificial Intelligence?

Simply put, artificial intelligence aims at enabling machines to execute reasoning by replicating human intelligence i.e., to think in the same way humans do. So, it's not just about programming a computer to drive a car but also display emotion and behaviors of road rage as humans.

Because the major goal of AI processes is to educate machines through experience, it's critical to provide the relevant information and allow for self-correction. Deep learning and natural language processing are used by AI professionals to assist robots in identifying patterns and conclusions.

The most popular use of A.I. comes in the form of digital smart assistants, such as **Siri, Alexa** and **Google Assistant**.

What is Machine Learning?

Machine Learning algorithms enable the computers to learn from data, and even improve themselves, without being explicitly programmed.

It is a subset of Artificial Intelligence that uses statistical learning algorithms to build systems that have the ability to automatically learn and improve from experiences without being explicitly programmed. We will discuss about the classification of ML in detail later.

We use machine learning in our day-to-day life when we use services like **recommendation systems** on Netflix, YouTube, Spotify; **search engines** like google and yahoo; **voice assistants** like google home and amazon Alexa.

What is Deep Learning?

Deep learning is a machine learning technique that is inspired by the way a human brain filters information, it is basically learning from examples.

Deep Learning is basically mimicking the human brain, it can also be defined as a multi neural network architecture containing a large number of parameters and layers. The three fundamental network architectures are (which we will not get into in depth)

- Convolutional Neural Networks
- Recurrent Neural Networks
- Recursive Neural Networks

Since deep learning processes information in a similar manner as a human brain does, it is mostly used in applications that people generally do.

It is the key technology behind **driver-less cars**, that enables them to recognize a stop sign and to distinguish between a pedestrian and lamp post.

Fundamental algorithms and concepts

Graph Theory

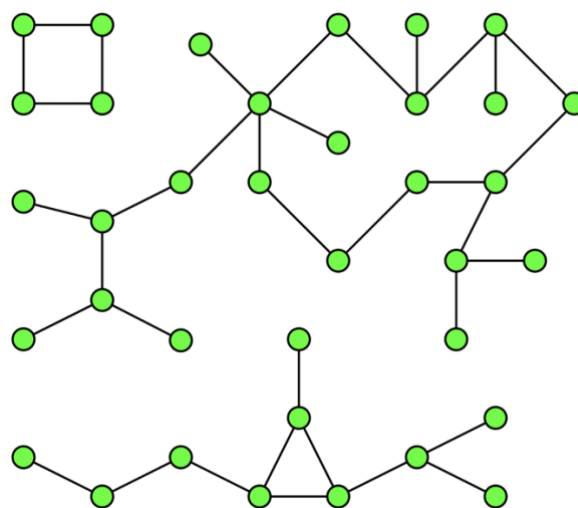
Graph theory is very useful if you work on developing new methods for learning and inference in probabilistic graphic models. The work on using graph-cuts to do exact and approximate inference on graphical models with particular applications in computer vision is extensive.

Here, I talked about some of the most influential graph algorithms that have changed the way we live. With the advent of so much social data, network analysis could help a lot in improving our models and generating value.

Let's talk about some of the most important graph algorithms you should know and how to implement as a data scientist.

Connected Components

You can think of Connected Components as a sort of a hard clustering algorithm which finds clusters/islands in related/connected data.



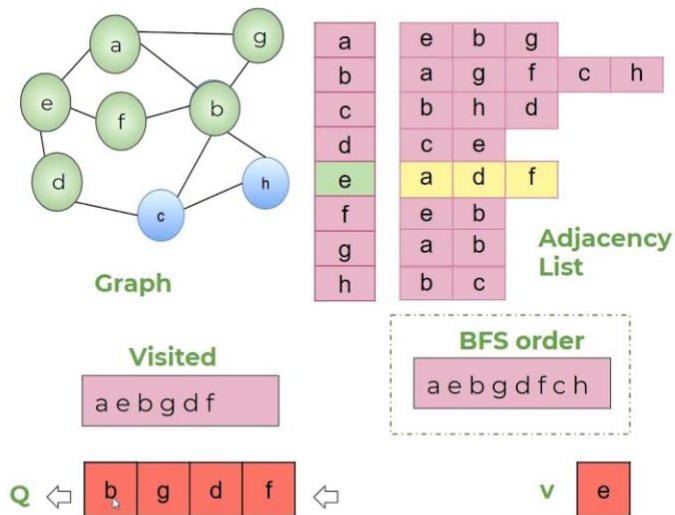
Say you have data about roads joining any two cities in the world. And you need to find out all the continents in the world and which city they contain.

How will you achieve that?

The connected components algorithm that we use to do this is based on a special case of **BFS/DFS**.

Let's look at the pseudo code for BFS and DFS quickly.

Breadth First Search

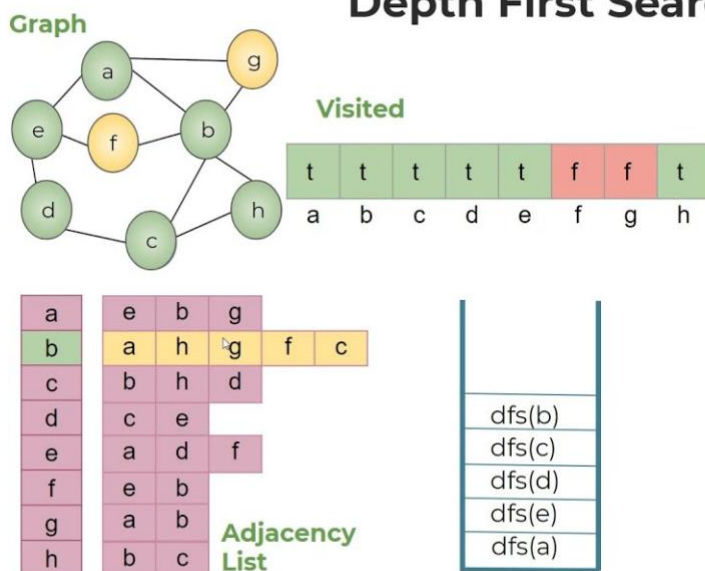


BFS

add start vertex to Q
mark start as visited

while Q not empty
 v ← dequeue from Q
 for each adj vertex av of v
 //... process vertex av...
 if av is not visited
 add av to Q
 mark av as visited

Depth First Search



DFS(v)

mark v as visited
For each adj vertex av of v
 If av is not visited
 DFS(av)

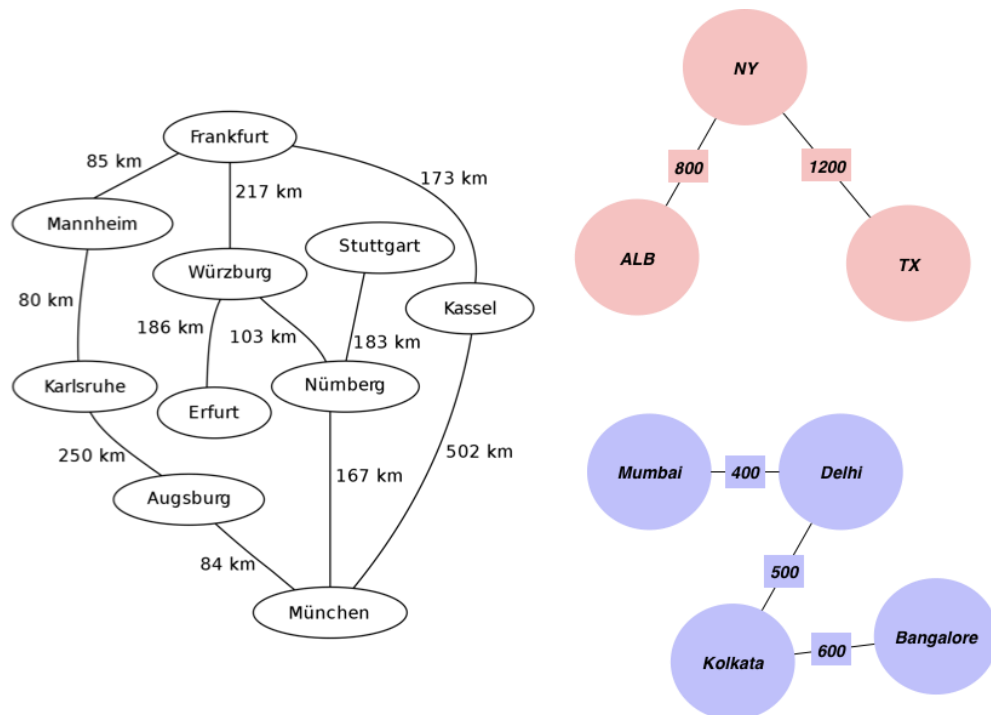
Now let's see our pseudo code for connected components algorithm and understand with an example graph which we are using for our purpose.

```

CONNECTED-COMPONENTS( $G$ )
1  for each vertex  $v \in G.V$ 
2      MAKE-SET( $v$ )
3  for each edge  $(u, v) \in G.E$ 
4      if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
5          UNION( $u, v$ )

```

Let's start by having a graph with random distances. We will be using the **Networkx** module in Python for creating and analysing our graph as it is easier and smaller this way.



We first start by creating a list of edges along with the distances which we will add as the weight of the edge like given below:

```
edgelist = [['Mannheim', 'Frankfurt', 85]]
```

Let us create a graph using Networkx(a python module):

```

g = nx.Graph()
for edge in edgelist:

```

```
g.add_edge(edge[0],edge[1], weight = edge[2])
```

Now we want to find out distinct continents and their cities from this graph. We can now do this using the connected components algorithm as

```
for i, x in enumerate(nx.connected_components(g)):
    print("cc"+str(i)+":",x)
```

The output of the code will be as follows with 3 connected components:

```
cc0: {'Frankfurt', 'Kassel', 'München', 'Nurnberg', 'Erfurt', 'Stuttgart',
'Karlsruhe', 'Wurzburg', 'Mannheim', 'Augsburg'}
cc1: {'Kolkata', 'Bangalore', 'Mumbai', 'Delhi'}
cc2: {'ALB', 'NY', 'TX'}
```

As you can see we are able to find distinct components in our data. Just by using Edges and Vertices. This algorithm could be run on different data to satisfy any use case with applications in Retail Perspective, Finance Perspective or any other perspective. The possibilities are only limited by your own imagination.

Shortest Path

Given a graph and a source vertex in the graph, find the shortest paths from the source to all vertices in the given graph.

For instance in the previous case, you want to find out how to go from Frankfurt (The starting node) to München by covering the shortest distance.

The algorithm that we use for this problem is **Dijkstra Algorithm**.

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem. The pseudo code is as follows:

```
DIJKSTRA( $G, w, s$ )
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2   $S = \emptyset$ 
3   $Q = G.V$ 
4  while  $Q \neq \emptyset$ 
5       $u = \text{EXTRACT-MIN}(Q)$ 
6       $S = S \cup \{u\}$ 
7      for each vertex  $v \in G.Adj[u]$ 
8          RELAX( $u, v, w$ )
```

Applications: Finding the shortest path is a crucial problem in data science and is used in various aspects of our life. Some examples are as follows:

1. Variations of the Dijkstra algorithm is used extensively in **Google Maps** to find the shortest routes. Consider India as a graph and represent a city/place with a vertex and the route between two cities/places as an edge, then by using this algorithm, the shortest routes between any two cities/places or from one city/place to another city/place can be calculated.
2. Social Networking Applications: You have seen how **LinkedIn** shows up 1st-degree connections, 2nd-degree connections. What goes on behind the scenes? The standard Dijkstra algorithm can be applied using the shortest path between users measured through handshakes or connections among them.
3. Robotic Path: **Drones** and **robots** have become commonplace in recent years, some of which are manual and others automated. This algorithm module is loaded into automated drones/robots that are used to deliver packages to a specific location or perform a task, so that when the source and

destination are known, the robot/drone moves in the ordered direction by following the shortest path to keep delivering the package in the shortest amount of time.

Again by using the **Networkx** module in Python we can easily find the shortest distance by:

```
print(nx.shortest_path(g,'Stuttgart','Frankfurt',  
weight='weight'))  
print(nx.shortest_path_length(g,'Stuttgart','Frankfurt',weight='weight'))
```

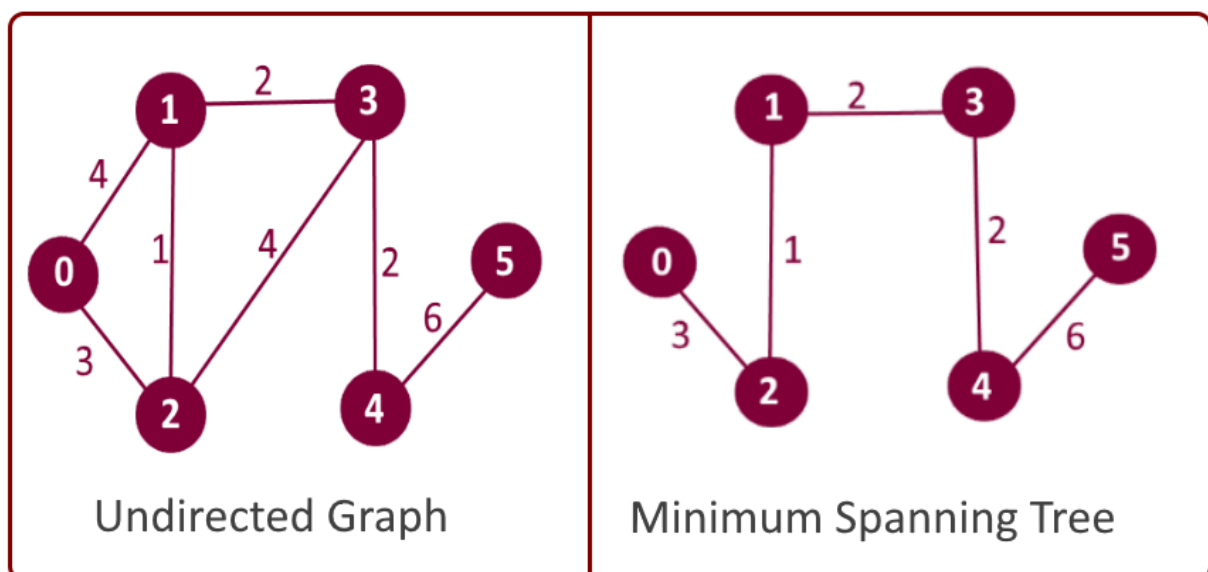
which will give the output for the same graph used before as

```
['Stuttgart', 'Nurnberg', 'Wurzburg', 'Frankfurt']
```

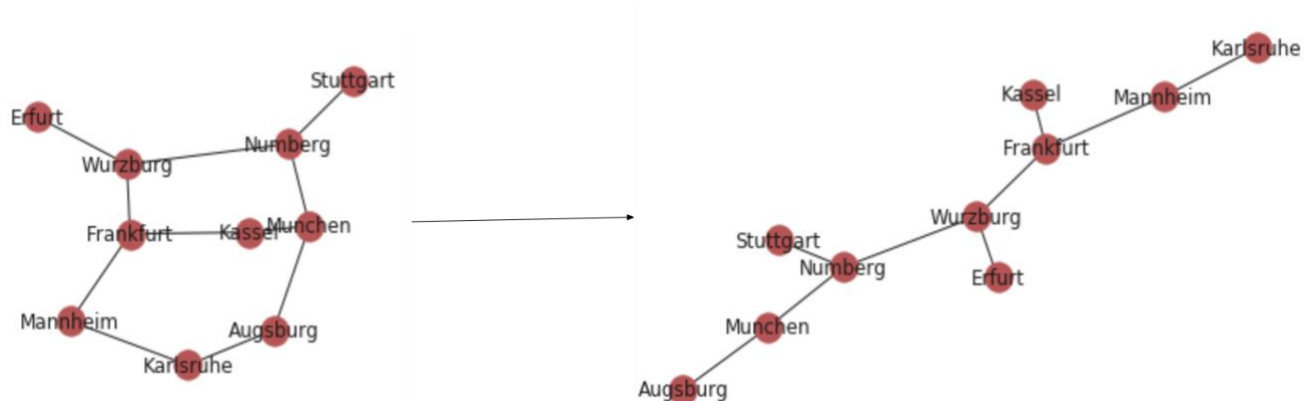
```
503
```

Minimum Spanning Tree

A minimum spanning tree (MST) or minimum weight spanning tree is a spanning tree with a weight less than or equal to the weight of every other spanning tree. It forms a tree that includes every vertex And has the minimum sum of weights among all the trees that can be formed from the graph.



The MST for our problem taken above looks like below



Application

MST has various uses in Data science like:

1. Minimum spanning trees have direct applications in the design of networks, including computer networks, telecommunications networks, transportation networks, water supply networks, and electrical grids (which they were first invented for)
2. MST is used for approximating the **traveling salesman problem**.
3. **Clustering** — First construct MST and then determine a threshold value for breaking some edges in the MST using Intercluster distances and Intracluster distances.
4. **Image Segmentation** — It was used for Image segmentation where we first construct an MST on a graph where pixels are nodes and distances between pixels are based on some similarity measure(colour, intensity, etc.)

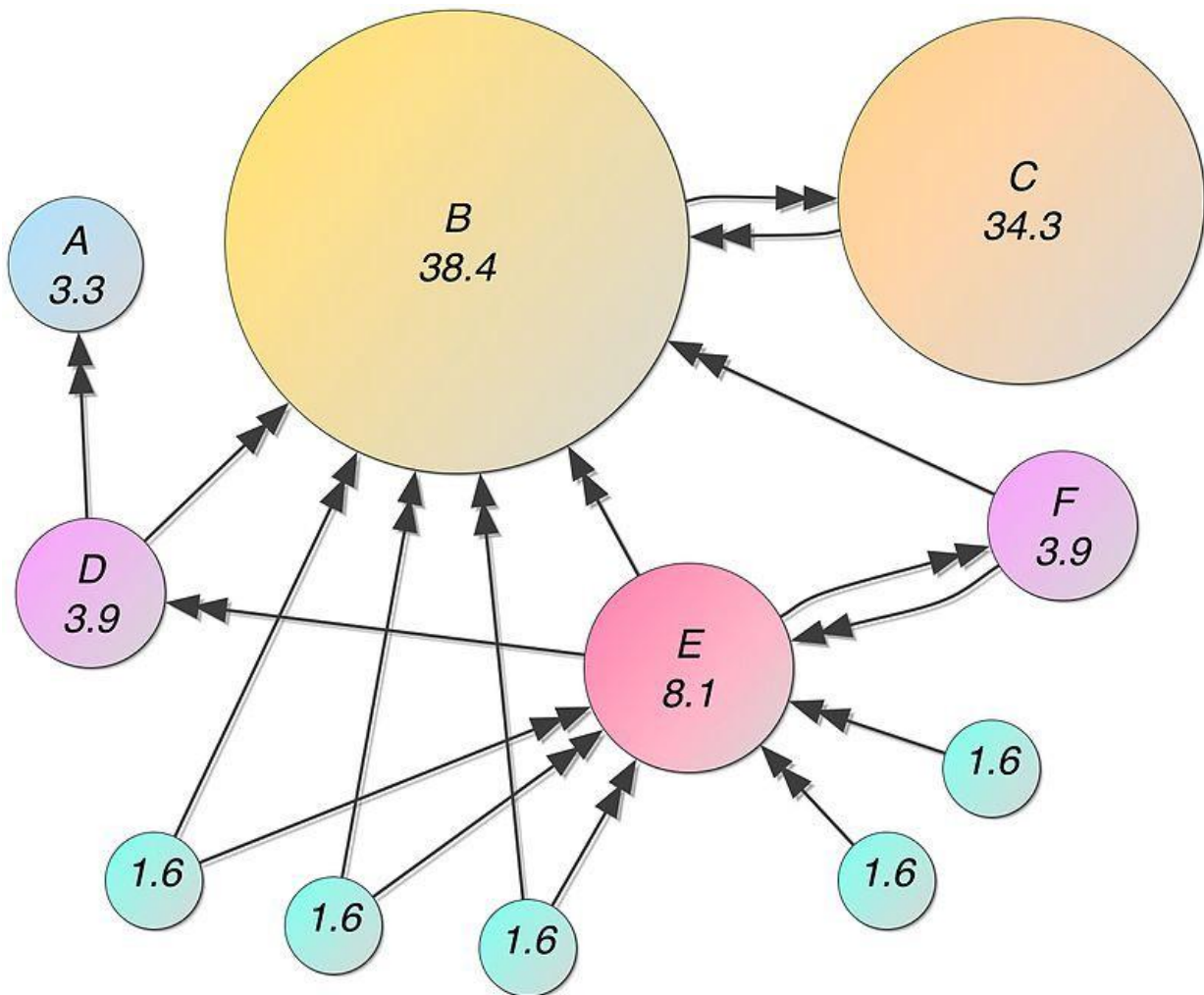
The code can be written as:

```
# nx.minimum_spanning_tree(g) returns an instance of type graph
```

```
nx.draw_networkx(nx.minimum_spanning_tree(g))
```

Pagerank

PageRank (PR) is an algorithm used by Google Search to rank websites in their search engine results.



PageRank is a way of measuring the importance of website pages. This sorting algorithm has powered google for a long time. It assigns scores to pages based on the number and quality of incoming and outgoing links.

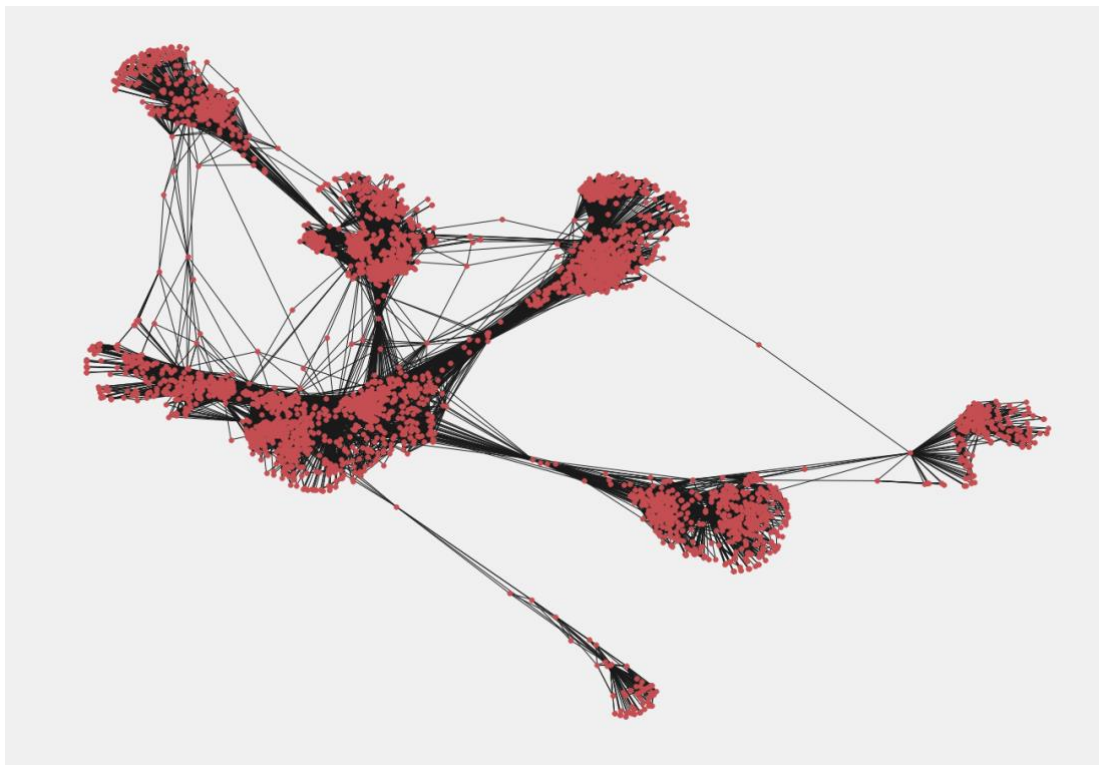
PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is.

The underlying assumption is that more important websites are likely to receive more links from other websites.

Applications

1. Pagerank can be used anywhere where we want to estimate node importance in any network.
2. It has been used for finding the most influential papers using citations.
3. Has been used by **Google** to rank pages
4. It can be used to **rank tweets**- User and Tweets as nodes.
Create Link between user if user A follows user B and Link between user and Tweets if user tweets/retweets a tweet.
5. Recommendation engines

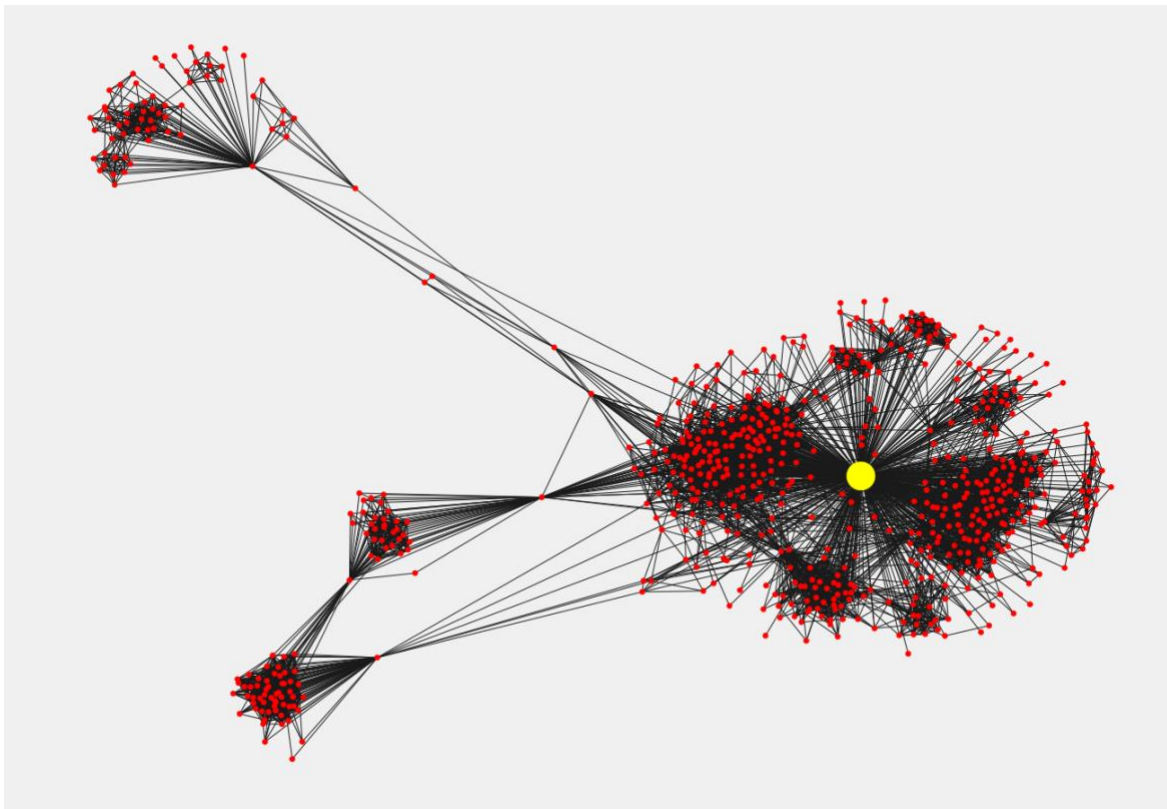
Let's take an example of Facebook data for this exercise. We have a file of edges/links between Facebook users. We first create the FB user graph which looks like:



Now we want to find the users having high influence capability. Intuitively, the Pagerank algorithm will give a higher score to a user who has a lot of friends who in turn have a lot of FB Friends.

We can get the sorted PageRank or most influential users using the `sorted_pagerank` operator and get the IDs for the most influential users.

We can see the subgraph for the most influential user by taking the first and second degree connected nodes:



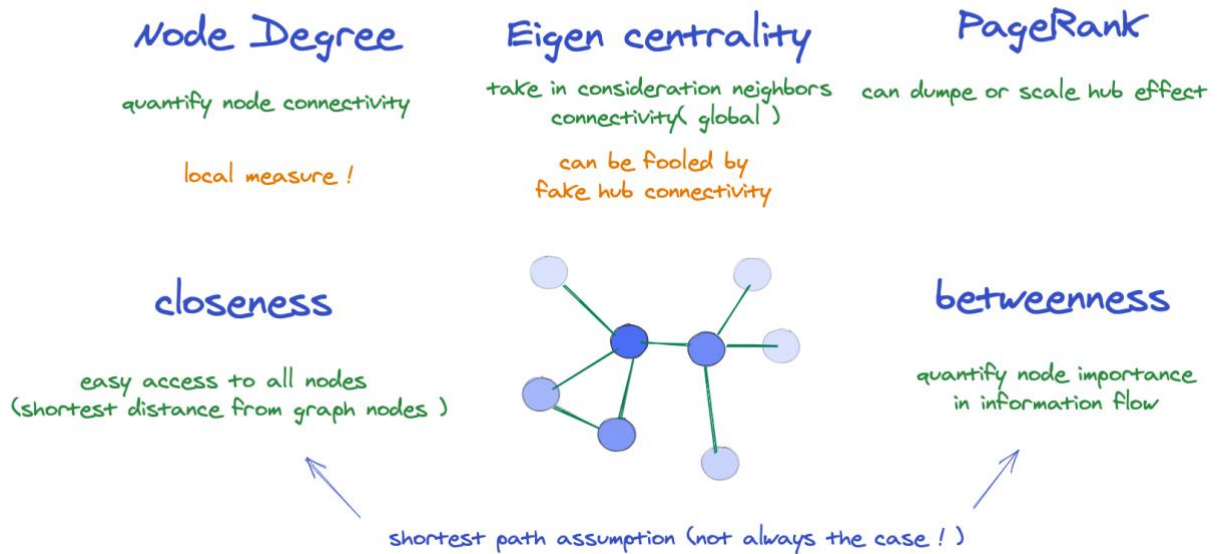
The most influential user(yellow) can be found out by using the PageRank algorithm.

Centrality Measures

There are a lot of centrality measures which you can use as features to your machine learning models.

We will talk about two of them that are very crucial and can see the other in brief below.

Graph Centrality measurements

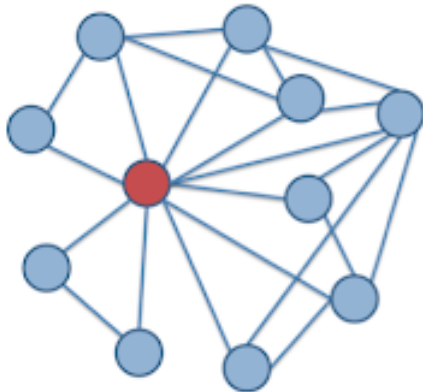


Betweenness Centrality: It is not only the users who have the most friends that are important, the users who connect one geography to another are also important as that lets users see content from diverse geographies.

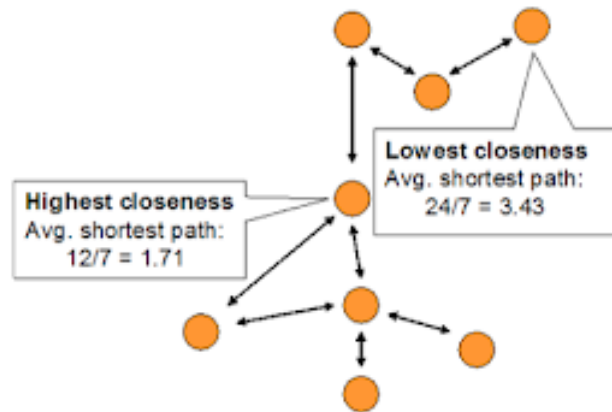
Betweenness centrality quantifies how many times a particular node comes in the shortest chosen path between two other nodes.

Degree Centrality: It is simply the number of connections for a node.
Applications

Degree centrality:
highest number of edges



Closeness centrality:
lowest average shortest
distance to all other nodes



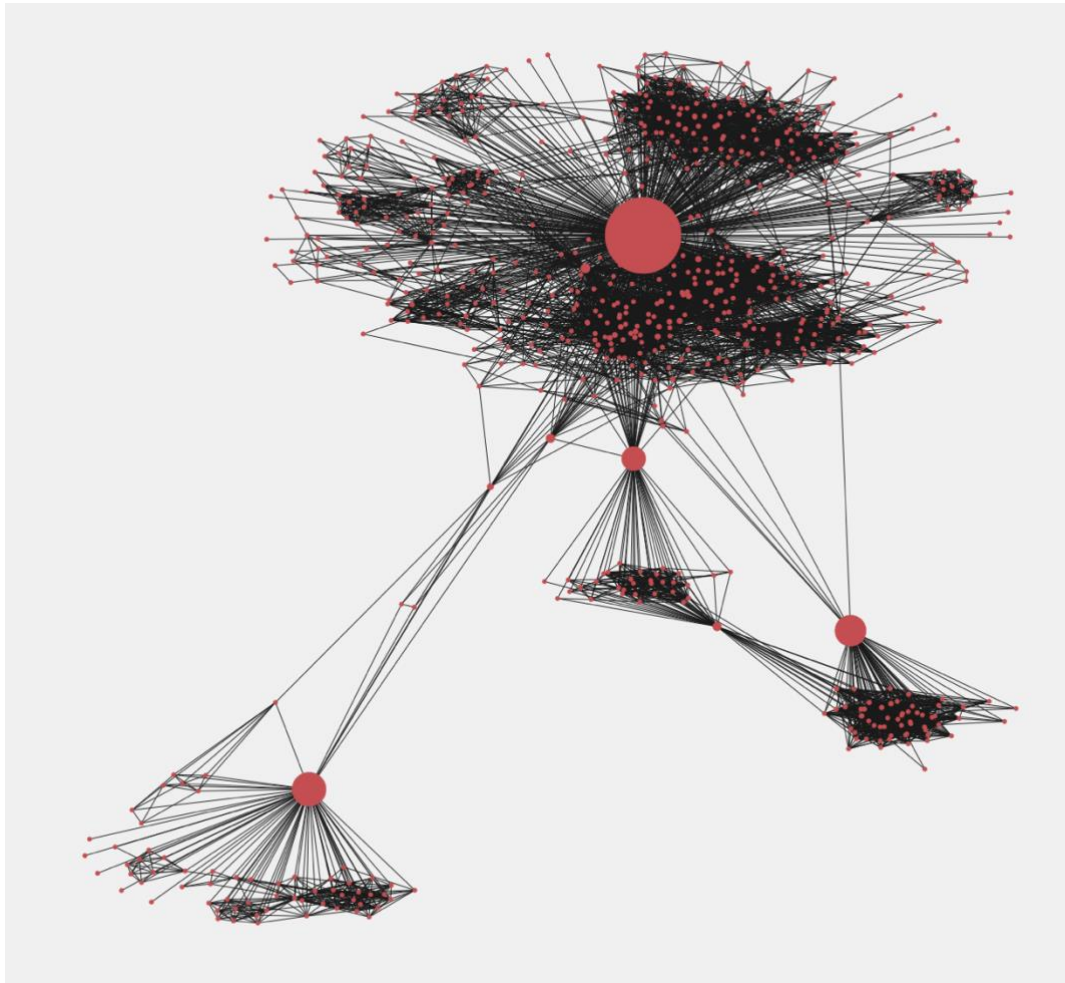
Application: Centrality measures can be used as a feature in any machine learning model.

Let's look at the code for finding the Betweenness centrality for the subgraph to understand a little more clearly.

We will again be using the **Networkx** module in Python.

```
pos = nx.spring_layout(subgraph_3437)
betweennessCentrality =
nx.betweenness_centrality(subgraph_3437,normalized=True,
endpoints=True)

node_size = [v * 10000 for v in betweennessCentrality.values()]
plt.figure(figsize=(20,20))
nx.draw_networkx(subgraph_3437, pos=pos, with_labels=False,
node_size=node_size )
plt.axis('off')
```



You can see the nodes sized by their betweenness centrality values here. They can be thought of as information passers. Breaking any of the nodes with a high betweenness Centrality will break the graph into many parts.

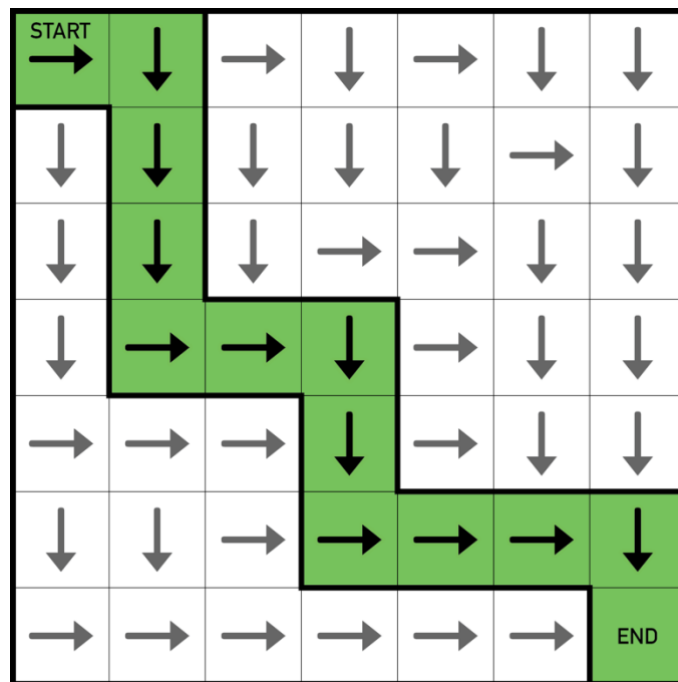
A* Algorithm

In the shortest path section we learned and saw the pseudo code for Dijkstra's algorithm. Now let's add intuition to our code in the graph search algorithm. This manifests in the **A* algorithm**.

A* algorithm can be thought of the decisions a highly talented CEO in his domain makes. He is smart choosing only the best option, that he feels can lead him to his goal, rapidly abandoning paths that won't lead him to his goal, but also he has **good intuition**. Some paths that don't look good in the start, he is sure that they are actually good and he proceeds to go through them and succeed. This A* is also called a

heuristic search. It is proven that if the heuristic (which is basically domain specific knowledge) fulfils certain conditions, then A* is guaranteed to give the correct solution.

Informally speaking, A* Search algorithms, unlike other traversal techniques, it has “*brains*”. A simple example of A* in action is in grid world



Here we can add the domain specific knowledge that moving in a direction that reduces distance towards the goal is almost always better. And so each block can be given a weight according to its distance from END, and A* searches for the lowest sum total.

What A* Search Algorithm does is that at each step it picks the node/cell having the lowest ‘f’(which is sum of ‘g’ and ‘h’), and process that node/cell.

We define ‘g’ and ‘h’ as simply as possible below:

- g = the **movement cost** to move from the starting point to a given square on the grid, following the path generated to get there.
- h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the **heuristic**, which is nothing but a kind of smart guess.

We know how to compute g, but how do we determine h?

We have two options.

- A. Either find the precise value of h, or (which is certainly time consuming). This can be done through pre computation.
- B. Use some heuristics to estimate the value of h. (less time consuming). There are generally three approximation heuristics to calculate h:
 - i. Manhattan distance
 - ii. Diagonal distance
 - iii. Euclidean distance

The pseudo code for the algorithm is given below.

The implementations are similar to Dijkstra's algorithm. Here we first create 2 lists: open and closed just like in Dijkstra's algorithm

A* Search Algorithm

```
1. Initialize the open list
2. Initialize the closed list
   put the starting node on the open
   list (you can leave its f at zero)

3. while the open list is not empty
   a) find the node with the least f on
      the open list, call it "q"

   b) pop q off the open list

   c) generate q's 8 successors and set their
      parents to q

   d) for each successor
      i) if successor is the goal, stop search
         successor.g = q.g + distance between
         successor and q
```

```

    successor.h = distance from goal to
    successor (This can be done using many
    ways, we will discuss three heuristics-
    Manhattan, Diagonal and Euclidean
    Heuristics)

    successor.f = successor.g + successor.h

    ii) if a node with the same position as
        successor is in the OPEN list which has a
        lower f than successor, skip this successor

    iii) if a node with the same position as
        successor is in the CLOSED list which has
        a lower f than successor, skip this successor
        otherwise, add the node to the open list
    end (for loop)

    e) push q on the closed list
end (while loop)

```

Reinforcement Learning can be thought of as trying to learn the most efficient heuristic for a given environment. In a way, the RL algorithm can essentially model most arbitrary heuristic functions present in A* algorithms.

And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently.

As seen in this section, we discussed how Graph theory is a crucial part of our life and many real world relationships are best modelled using graph structures. In many applications, treating the underlying data as a graph can achieve greater efficiency.

While machine learning is not tied to any particular representation of data, most machine learning algorithms today operate over real number **vectors**. In order to feed graph data into a **machine algorithm pipeline**, so-called **embedding frameworks** are

commonly used. They basically perform a mapping between each node or edge of a graph to a vector. A large number of frameworks has been designed so far that intend to encode graph information into low-dimensional real number vectors of fixed length. One embedding framework that gained a lot on popularity since its inception is **node2vec**, a method that learns features for networks by exploring nodes neighbourhoods through **random walks**.

Graph embeddings are just one of the heavily researched concepts when it comes to the field of graph-based machine learning. The research in that field has exploded in the past few years. One technique gaining a lot of attention recently is **graph neural network**.

Graph Neural Network is a type of Neural Network which directly operates on the Graph structure. A typical application of GNN is node classification. Essentially, every node in the graph is associated with a label, and we want to predict the label of the nodes without ground-truth.

We won't be discussing this in detail but some jargons to know here are **GNN**, **DeepWalk**, and **GraphSage** which helps in modelling complex graph structures.

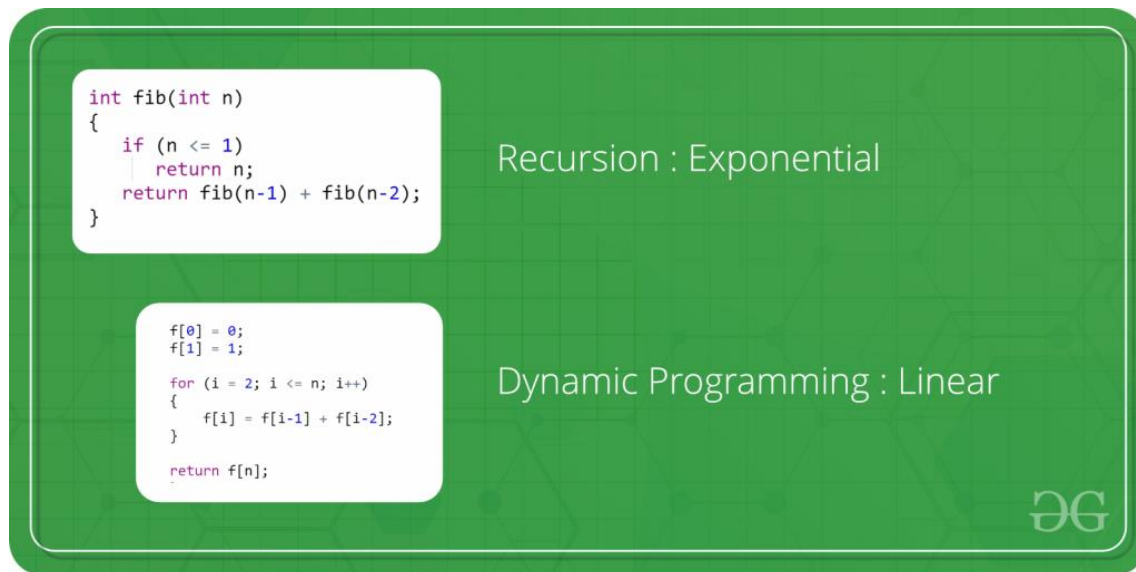
Dynamic programming

Machine learning (ML) models use a lot of data and have a lot of analysis in their methods, thus it's ideal to set up an optimal solution environment in terms of efficacy.

This is where dynamic programming comes into the picture. It is specifically used in the context of reinforcement learning (RL) applications in ML. It is also suitable for applications where decision processes are critical in a highly uncertain environment.

What is Dynamic programming?

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later.



```
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

Recursion : Exponential

```
f[0] = 0;
f[1] = 1;

for (i = 2; i <= n; i++)
{
    f[i] = f[i-1] + f[i-2];
}

return f[n];
```

Dynamic Programming : Linear

GG

Dynamic Programming (DP) is one of the techniques available to solve **self-learning problems**. It is widely used in areas such as operations research, economics and automatic control systems, among others. **Artificial intelligence** is the core application of DP since it mostly deals with learning information from a highly uncertain environment.

Dynamic programming in Reinforcement Learning

Richard Sutton and Andrew Barto gave a concise definition of the concept in their book Reinforcement Learning: An Introduction:

*“The term dynamic programming refers to a collection of algorithms which can be used to compute **optimal policies** given a perfect model of the environment as a **Markov decision process**.”*

Okay so understand this definition let's first get a basic about what is Markov decision process(MDP). We won't be going in full detail as it is beyond the scope of this book and try to understand the gist.

Reinforcement Learning and Markov Decision Process

Reinforcement Learning is a type of Machine Learning. It allows machines and software agents to automatically determine the **ideal behaviour** within a specific context, in order to maximize its performance. Simple **reward feedback** is required for the agent to learn its behaviour; this is known as the **reinforcement signal**.

There are many different algorithms that tackle this issue. As a matter of fact, Reinforcement Learning is defined by a specific type of problem, and all its solutions are classed as Reinforcement Learning algorithms. In the problem, an agent is supposed to decide the best action to select based on his current state. When this step is repeated, the problem is known as a **Markov Decision Process**.

A Markov Decision Process (MDP) model contains:

- A set of possible world states S .
- A set of Models.
- A set of possible actions A .
- A real-valued reward function $R(s,a)$.
- A policy the solution of Markov Decision Process.

States:	S
Model:	$T(S, a, S') \sim P(S' \mid S, a)$
Actions:	$A(S), A$
Reward:	$R(S), R(S, a), R(S, a, S')$
<hr/>	
Policy:	$\Pi(S) \rightarrow a$ Π^*
<i>Markov Decision Process</i>	

As we discussed we won't be covering the whole of MDP in detail. The above terminology provides us with the final results as:

- Small reward each step (can be negative which can also be termed as punishment).
- Big rewards come at the end (good or bad).
- The goal is to Maximize the sum of rewards.

Now that we have a better understanding of MDP let's go back to the definition.

Drawing from the definition, DP assumes the input model to be ideal (with all the probability parameters and reward functions known beforehand) to construct it as a Markov decision process (MDP). Based on this, solutions are obtained for problems wherein a set of action spaces and model states are presented, which are again continuous in nature. Altogether, DP finds the optimal and good policies in a right model structure.

Ultimately, in any DP problem, maximising returns for reward functions over time forms the core idea. In addition, another aspect

DP aims to resolve is at furnishing a learning model environment for reinforcement learning (RL), where a full-blown learning model is generally absent.

DP Algorithms

DP algorithms can be classified into three subclasses.

1. Value iteration algorithms- Learn the values for all states, then we can act according to the gradient. For more details give it a read [here](#).
2. Policy iteration algorithms- Policy learning incrementally looks at the current values and extracts a policy. For more details give it a read [here](#).
3. Policy search algorithms- For more details give it a read [here](#).

These are the fundamental iterative methods of Reinforcement learning. All these algorithms have their respective advantages in various learning applications.

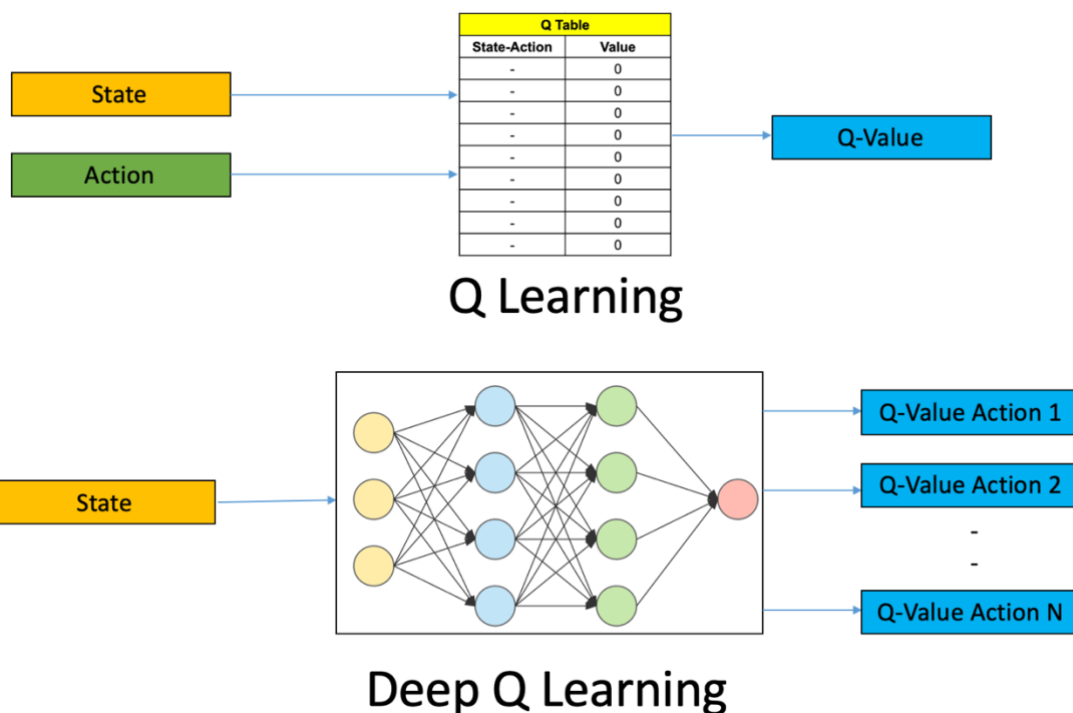
As mentioned earlier, DP is also very useful for large and continuous-space problems in real time. It provides insights from complex policy representations through a technique called 'approximation'. This means providing an estimate of the huge number of possible values achievable through value or policy iterations in DP. Sampling large data can have a great effect on the learning aspect in any of the algorithms.

In this section we saw briefly how we used Dynamic Programming to estimate these rewards using MDP and how these algorithms are a crucial part of Reinforcement Learning.

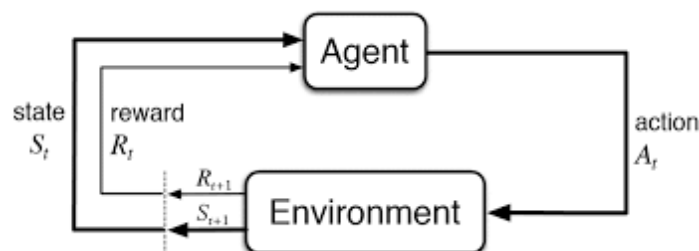
Now let's look at one more important instance where Dynamic Programming plays a vital role.

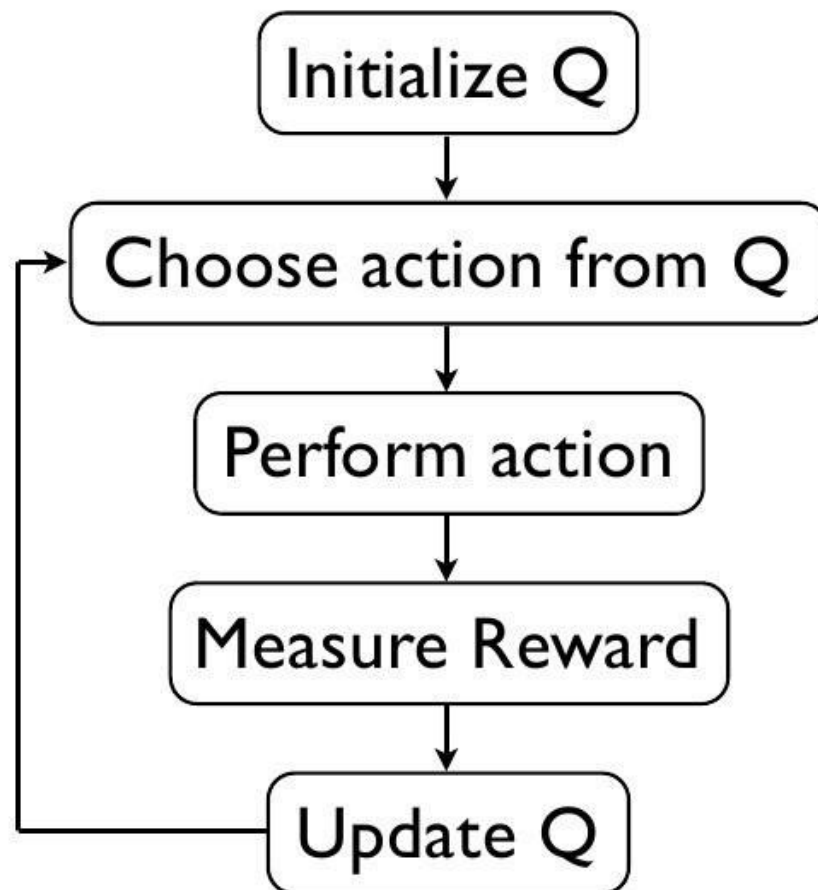
Q-Learning: A Paradigm In Dynamic Programming

In DP, there are a host of algorithms to obtain MDPs. The popular amongst them that finds more implementation in applications is **Q-learning**. Q Learning is one of the most popular reinforcement learning algorithms. This method relies on “Q-values” which are based on actions and states in RL.



Basically, Q-learning considers three crucial parameters in RL, namely transition function (T), reward function (R) and value function (V), by keeping learning rate α and discount factors in mind. It is best suited for applications where the parameters T and R are unknown (a typical scenario in MDP).





In order to get an optimum solution from a set of actions and states after value iteration, Q-learning is preferred since the maximum values cannot be determined for the unknown parameters. In other words, 'q-values' are computed in place of maximum value iterations. A standard formula for is given below:

$$Q_{k+1}(s,a) = \gamma T(s,a,s')[R(s,a,s') + \max_{a'} Q(s', a')]$$

where, $Q_{k+1}(s,a)$ is the iterated function with states 's' and action 'a' by considering the summation values of parameters T and R along with discount factor γ . The apostrophes denote an update function is incorporated all along the process.

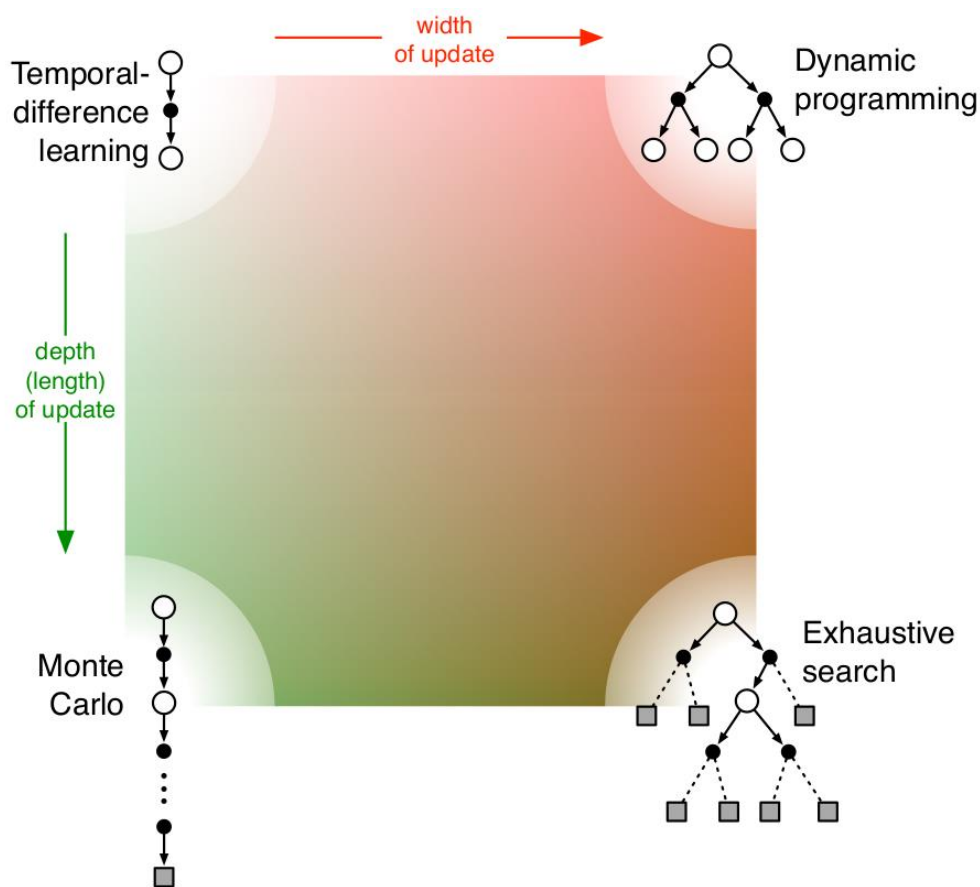
This equation serves as the basis for Q-learning. The update function in the parameters also incorporates an optimal policy for the RL results. The equation can be modified to even RL concepts such as

exploration and exploitation for problems with a very uncertain environment.

The core of the algorithm is a Bellman equation as a simple value iteration update, using the weighted average of the old value and the new information:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \underbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)}_{\text{temporal difference}} \underbrace{\quad}_{\text{new value (temporal difference target)}}$$

We have seen the use of Dynamic programming and Graph search(previous section) in Reinforcement Learning and how it forms the base of RL. Although we have not looked in depth about all the RL algorithms we can use this diagram to sum up the **classical RL**.



Sutton and Barto, Reinforcement Learning: An introduction

Sorting

In this section, we will show that how a simple sorting algorithm is at the heart of solving an important problem in computational geometry and how that relates to a widely used machine learning technique. Although there are many discrete optimization based algorithms to solve the **SVM** problem, this approach demonstrates the importance of **using fundamentally efficient algorithms at the core to build complex learning model for AI.**

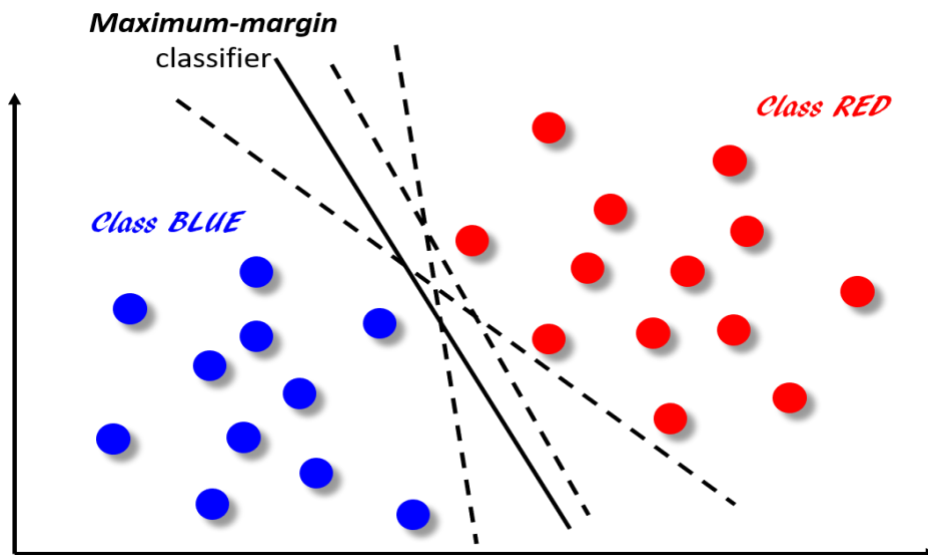
Let's start with understanding what SVM is?

Support Vector Machine

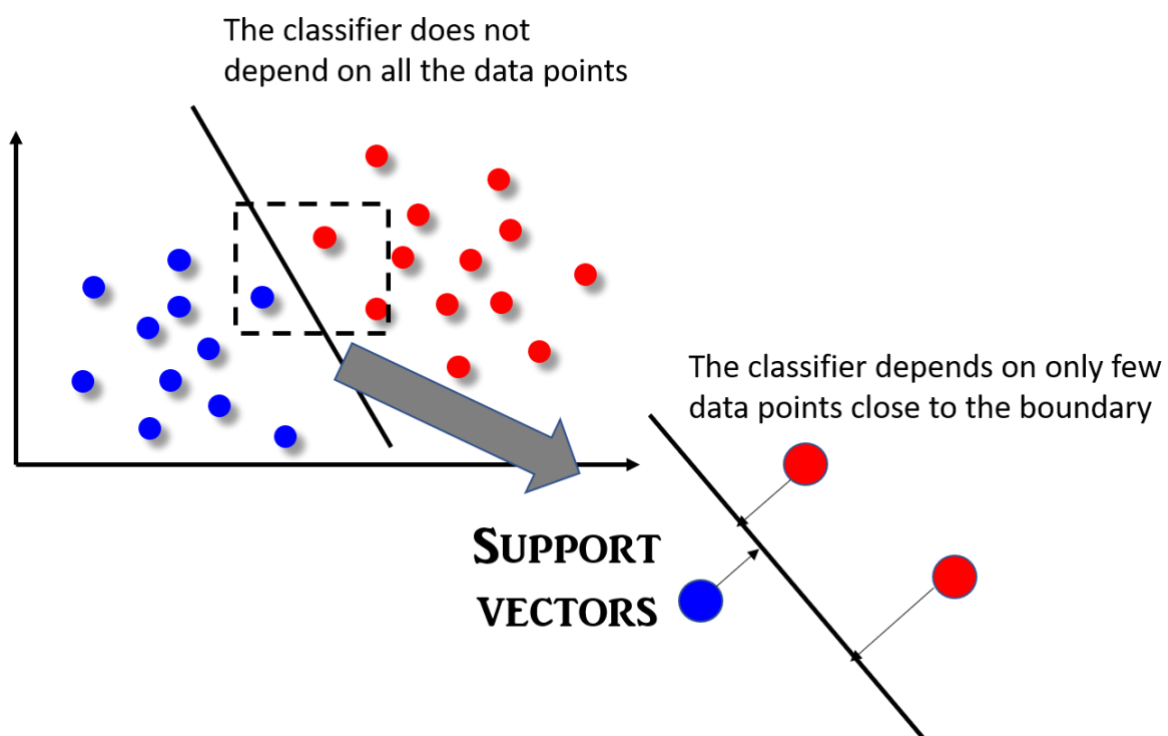
Support vector machine or SVM in short, is one of the most important machine learning techniques, developed in past few decades. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other. It is widely used in industrial systems, text classifications, pattern recognition, biological ML applications, etc.

The goal of the SVM algorithm is to create the **best line** or decision boundary that can **segregate** n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a **hyperplane**.

The unique classifier (with the maximum distance) is shown by the solid line whereas the other classifiers are shown as dotted lines. The utility of this margin maximization is that larger the distance between two classes, lower the generalization error will be for the classification of a new point.



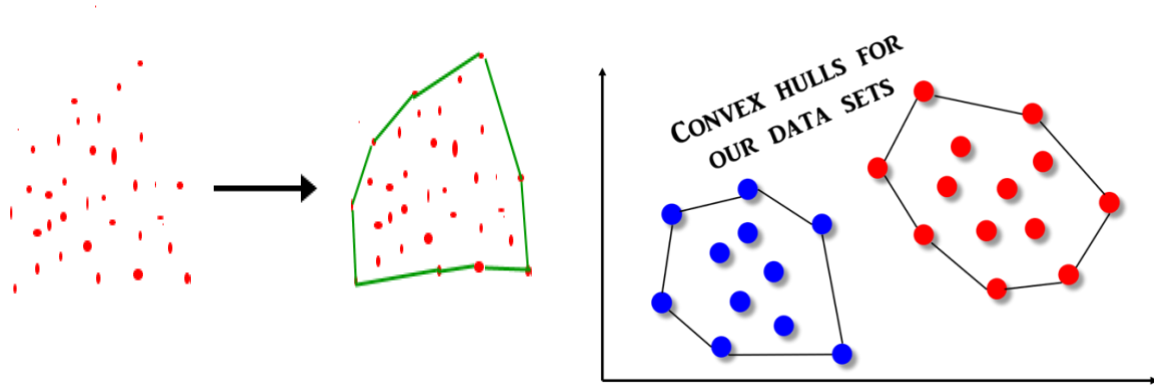
The main differentiating feature of SVM algorithm is that the classifier does not depend on all the data points. In fact, SVM classifier depend on a very small subset of data points, those which lie **closest to the boundary** and whose position in the hyperplane can impact the classifier boundary line. The vectors formed by these points uniquely define the classifier function and they 'support' the classifier, hence the name 'support vector machine'. This concept is shown in the following figure.



Convex Hull

The formal mathematics behind the SVM algorithm is fairly complex but intuitively it can be understood by considering a special geometric construct called convex hull.

What is Convex Hull? Formally a **convex hull** or **convex envelope** of a set X of points in the Euclidean plane or in a Euclidean space is the smallest convex set that contains X . In simple words, given a set of points in the plane, the convex hull of the set is the smallest convex polygon that contains all the points of it.

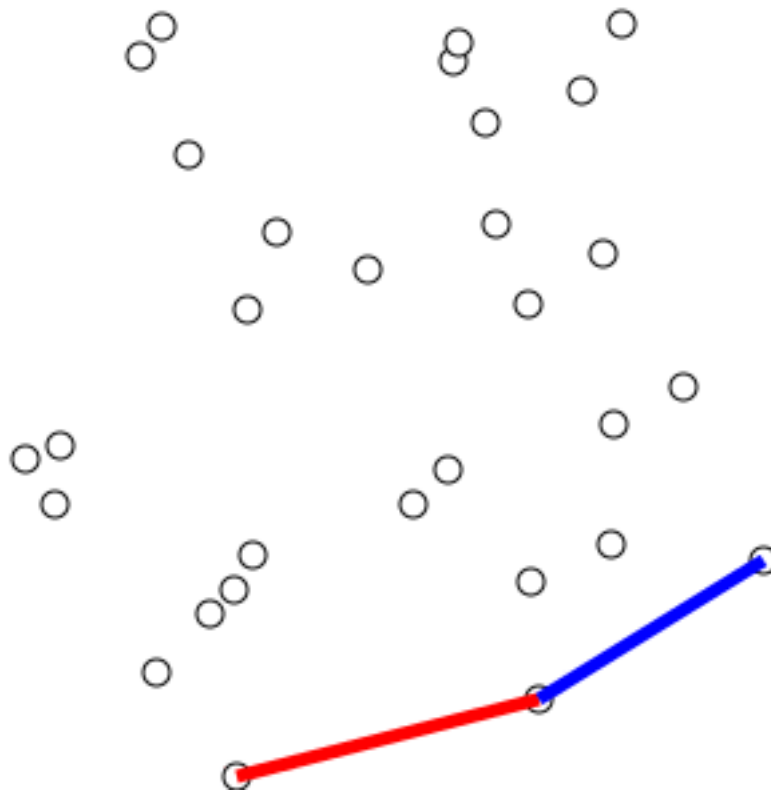


Now, it is easy to imagine that the SVM classifier is nothing but the linear separator that bisects the line joining these convex hulls exactly **mid-point**.

How to determine the convex hull?

The algorithm used to determine the convex hull of a set of points, in action is called **Graham's scan**. The algorithm finds all vertices of the convex hull ordered along its boundary. It uses a stack to detect and remove concavities in the boundary efficiently.

Now, the question is how efficient this algorithm is i.e. what is the time complexity of Graham's scan method?



Visual representation of Graham's scan to find the convex hull

It turns out that the time complexity of Graham's scan depends on the underlying **sort algorithm** that it needs to employ for finding the right set of points which constitute the convex hull! But what is it sorting to begin with?

The fundamental idea of this scanning techniques comes from two properties of a convex hull,

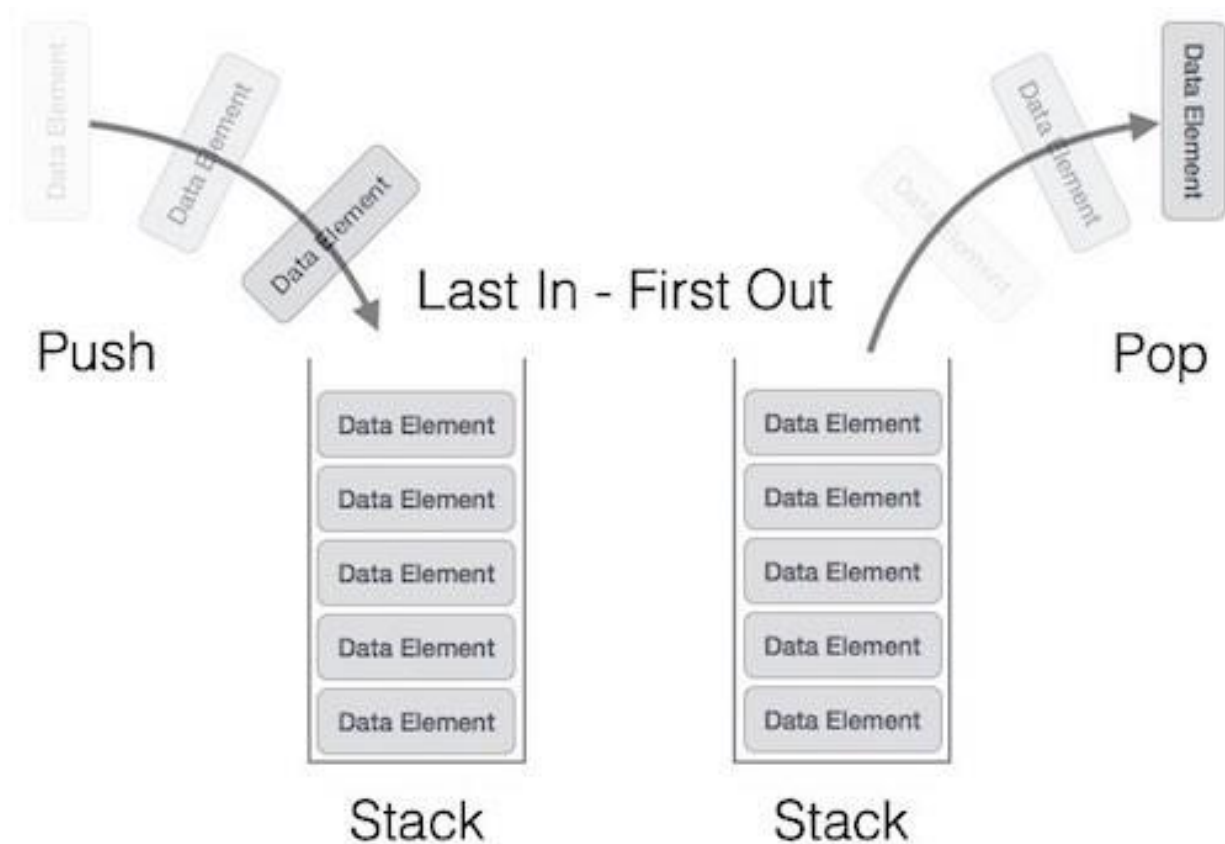
- Can traverse the convex hull by making only counter clockwise turns
- The vertices of convex hull appear in increasing order of polar angle
- with respect to point p with lowest y-coordinate.

First the points are stored in an array points. Therefore, the algorithm starts by locating a reference point. This is the point with the lowest y coordinate (in case of ties, we break the tie by selecting the point with

the lowest y coordinate and the lowest x coordinate). Once we locate the reference point, we move that point to beginning of points by making it trade places with the first point in the array.

Next, we sort the remaining points by their polar angle relative to the reference point. After sorting, the points with the lowest polar angle relative to the reference point will be at the beginning of the array, and the points with the largest polar angle will be at the end.

With the points properly sorted, we can now run the main loop in the algorithm. The loop makes use of a second list that will grow and shrink as we process the points in the main array. Basically, we push points, which appear in counter clockwise rotation, on to the stack and reject points (pop from the stack) if the rotation becomes clockwise. The second list starts out empty. At the end of the algorithm, the points that make up the convex boundary will appear in the list. A stack data structure is used for this purpose.



So, the time complexity of Graham's scan depends on the **efficiency of the sort algorithm**. Any general purpose sort technique can be used but there is a big difference between using a $O(n^2)$ and a $O(n \log(n))$ algorithm (as illustrated in the following animation).

	 Insertion	 Selection	 Bubble	 Shell	 Merge	 Heap	 Quick	 Quick3
 Random								
 Nearly Sorted								
 Reversed								
 Few Unique								

Algorithm	Time Complexity		
	Best	Average	Worst
Selection Sort	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Heap Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$
Quick Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n^2)$
Merge Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$
Bucket Sort	$\Omega(n+k)$	$\theta(n+k)$	$O(n^2)$
Radix Sort	$\Omega(nk)$	$\theta(nk)$	$O(nk)$
Count Sort	$\Omega(n+k)$	$\Omega(n+k)$	$\Omega(n+k)$

References

- <https://towardsdatascience.com/understanding-the-difference-between-ai-ml-and-dl-cceb63252a6c>
- <https://www.oreilly.com/content/how-graph-algorithms-improve-machine-learning/>
- <https://towardsdatascience.com/data-scientists-the-five-graph-algorithms-that-you-should-know-30f454fa5513>
- <https://datascience.aero/machine-learning-graphs/>
- <https://www.analyticsvidhya.com/blog/2017/09/common-machine-learning-algorithms/>
- <https://analyticsindiamag.com/role-of-dynamic-programming-in-machine-learning/>
- <https://medium.datadriveninvestor.com/reinforcement-learning-as-heuristic-search-analogy-31d92b06dadd>
- <https://arxiv.org/abs/1805.04272>
- <https://towardsdatascience.com/how-the-good-old-sorting-algorithm-helps-a-great-machine-learning-technique-9e744020254b>