Machine Learning

Naureen Ghani

January 1, 2018

1 Computer Science Fundamentals

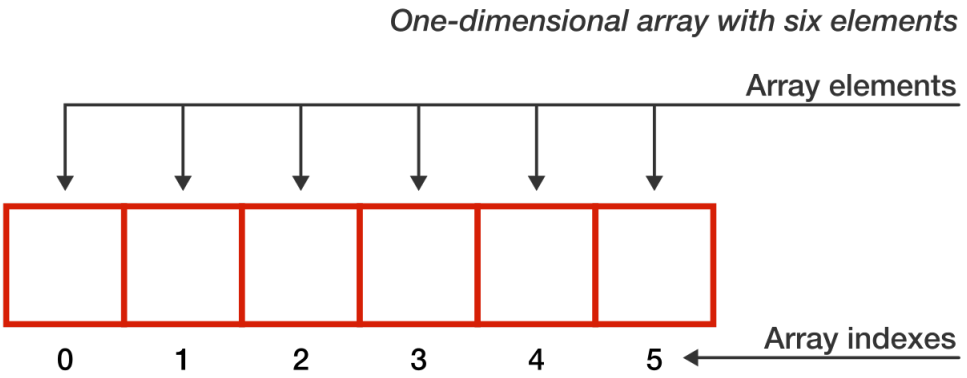
In the digital world, you need to solve problems efficiently. This requires organizing data for storage and analysis. In this section, we will learn the basics of programming.

As in most analytical disciplines, there is more than one answer to the same problem. The crux of mastering computer science is finding the *best* solution. Computers allow us to store and manipulate information, and to perform computations ranging from simple operations like looking up the price of an item in a store to complex modeling like ***artificial neural networks.*** We will discover how algorithms provide a way to organize logic and extract meaning from data.

1.1 Introduction to Algorithms

1.1.1 Arrays

Lists (also known as the ***array abstract data type***) are implemented by the ***array data structure.*** An ***array*** holds an ordered collection of items accessible by an integer index. It is the simplest way to store multiple values. The items in an array can be anything from primitive types such as integers to more complex types like instances of ***classes.***



There are a couple of points which are important to note:

- In this visual representation of an array, you can see that it has a size of six. Once six vlues have been added, no additional values can be added until the array is resized or something is taken out.
- You can also see that each "slot" in the array has an associated number, called an ***index.*** By using these numbers, the programmer can directly index into the array to get specific values, which makes them very efficient.

One potential drawback of an array is that the data is not necessarily sorted or releated in any useful way (other than being mapped to indices). We can develop algorithms to efficiently sort arrays. There are pros and cons to different data types and structures when solving a given problem. Sometimes, simplicity will win; other times, we'll need a more powerful but complex approach.

1.1.2 Searching

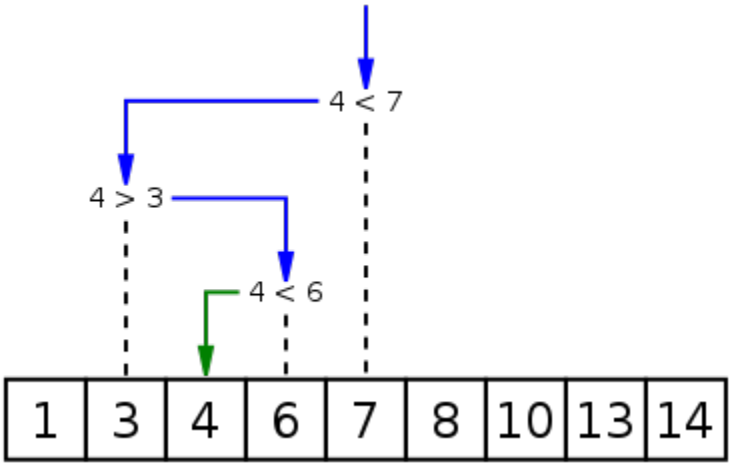
One of the simplest questions one might have about a list is whether or not some item is in the list. For example, if you are writing an algorithm to determine if someone should have access to a members-only website, you can solicit their information and then see if it matches some item in the members list. This is a ***linear search.***

To determine if an element is in an array with n elements using linear search, the number of comparisons we need to make is

- in the best case, just 1 comparison
- in the worst case, n comparisons
- in the average case, given that element is actually in the array, $\frac{n+1}{2}$ comparisons

While the average case might seem not too bad, it's important to think about what this means for large n. While 500,000 comparisons is certainly not as bad as 1,000,000, they're both problematic in that they scale linearly with n; that is, with twice as much data, the algorithm will need twice as many comparisons.

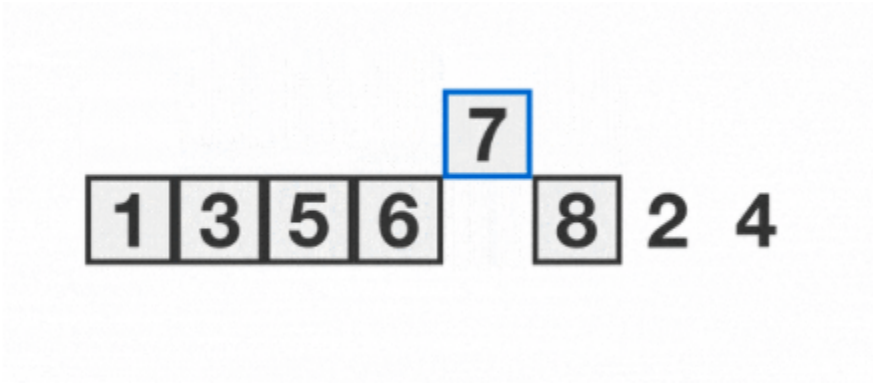
We can do better with a ***binary search.*** With a sorted array, a binary search repeatedly chooses a number in the middle of the remaining possible numbers, and then determines if the desired number would lie to to the left or right of this chosen number (or, if it is the chosen number). In each iteration, the amount of remaining numbers is halved, making binary search very efficient. This is especially important when dealing with a very large array.



1.1.3 Insertion Sort

Organized data can be valuable for certain types of computations, like searching for an element in a list. So, how can we sort a list?

Insertion sort is a sorting algorithm that builds a final sorted array one element at a time. Here is a visual example:



Insertion Sort:

- We start from the left. At first, there is one element (which is, trivially, already sorted).
- For each subsequent element, we compare it to each of the elements to the left of it, moving right to left, until we find where it "fits" in the sorted portion of the array.
- Since we are inserting the new element into the side of the array that is already sorted, the result will still be sorted.
- After we've inserted every element, we have a sorted array.

The best case is that the list is already sorted. Each of the elements other than the first one will be compared to the element preceding it, for a total of $n - 1$ comparisons.

The worst case is that each element is compared to each other element. The second element is compared to the first element, then the third element is compared to those elements, and so on, for a total of $1 + 2 + ... + (n-1) = \frac{(n-1)(n)}{2}$ comparisons. Alternatively, there are n elements and therefore $\binom{n}{2}$ comparisons to be made in the worst case.

In summary, the worst case increases like n^2 for large n, while the best case increases linearly (like n). Unfortunately for insertion sort, the worst case is much more likely, and the average case also increases like n^2. There are more efficient sorting algorithms that we will later learn. Their worst case performance will increase like $n \log n$ for large n.

1.1.4 Big O Notation

In computer science and programming, we're always looking for "good" solutions. In many cases, the good solution is the one which performs the fastest, which is generally related to using the fewest computations.

To meaningfully compare algorithmic performance, we can use ***big O notation***– sometimes referred to as "order of growth." In short, it compares the *asymptotic* behavior of algorithms; that is, how does their performance scale as a function of the input size?

Informally, $f(x) = O(g)$ if f is "less or about the same size" as g and x gets very large. For example, consider two algorithms with step count represented by $f(x) = 4x$ and $g(x) = 10x$. These are "about the same size" as x gets vary large, since the infinite, linear x dominates the 4 and the 10. Thus, $4x = O(10x)$, and $10x = O(4x)$.

On the other hand, compare $4x$ and x^2: $4x$ is less than x^2as x gets very large. Thus, $4x = O(x^2)$, but x^2does not equal $O(4x)$.

Formally speaking, if f and g are functions, then $f = O(g)$ if there exists some constant C such that

$$|f(x)| \leq C \cdot |g(x)|$$

for sufficiently large x.

Another way you could look at this is to divide g by f and see if the result is finite. In the examples just given, $10x/4x = 2.5$, but $x^2/4x = x/4$ which goes to infinity as x gets large.

In the equality case (that is, when $|f(x)| = C \cdot |g(x)|$ for sufficiently large x), it is said that $f = \Theta(g)$.

Since the theta notation is a special case of big O notation, the term "big O notation" can refer to either, even though only one of them uses an O.

1.2 Recursion

1.2.1 Recursion

Recursion is the phenomenon of self-embedded structures or processes. It is an essential tool in the practice of computational analysis for two reasons:

1. it can simplify code
2. it can speed up programs by requiring less information to be stored

A simple, famous example of recursion is the definition of the ***Fibonacci sequence:*** $0, 1, 1, 2, 3, 5, 8...$ We can describe the sequence by the following relation for $n \geq 2$:

$$F_n = F_{n-1} + F_{n-2}$$

For example, $5 = 3 + 2$, $8 = 5 + 3$, and so on; the relation is defined in terms of itself. In recursion, the first few values of a sequence are explicitly defined. This is called a ***base case.*** How would we write a program to yield the Fibonacci sequence using recursion?

```
% Fibonacci Sequence (Recursion)

% Provide base case
n(1) = 1;
n(2) = 1;

k = 3;
while k <= 10
    n(k) = n(k-1)+n(k-2);
    k = k+1;
end
```

First, we must define the base cases to act as seed values. After this, we implement the recurrence relation by calling the function within itself– recursion! The recursive logic is simpler to implement than the iterative solution, and expresses the solution in a way that directly mirrors the nature of the problem. However, it is not necessarily more efficient or less computationally intensive. To illustrate this concept, let us write code using recursion to find the Fibonacci value of a given index:

```
% Fibonacci Term using Recursion

function [result] = fibonacci_term(n)

if n==0||n==1
    result = n;
else
    result = fibonacci_term(n-2)+fibonacci_term(n-1);
end
end
```

Some functions are called multiple times and it isn't as efficient as the non-recursive approach.

A common application of recursion in computer science is ***the binary search algorithm.*** This algorithm is used to find a value in a sorted array of values. It operates roughly as follows:

- Observe the middle element of the array.
- Compare the element to the target.
- If greater than the target, go to the first step, but only consider the first half of the array.
- If less than the target, go to the first step, but only consider the second half of the array.
- If equal to the target, you have found your target!

Given a sorted array

$$[-1, 0, 1, 1, 3, 4, 5, 6, 8, 9, 10, 10, 11, 20, 100]$$

and a target value of 4, we would perform the following steps:

- See 6 as the middle element and $6 > 4$, so we now consider the array $[-1, 0, 1, 1, 3, 4, 5]$.
- See 1 as the middle element $1 < 4$, so we now consider the array $[3, 4, 5]$.
- See 4, and we have found our target

We repeat the same set of steps until some end condition is satisfied: either the element is found, or it is determined that it does not exist in the sorted array. The end condition translates directly to the ***base case*** of the recursion in the algorithm. The following code implements the binary search algorithm:

```
function [index] = binarySearch(A, n, num)

%-------------------------------------------------------------------------
% Syntax:       [index] = binarySearch(A, n, num);
%
% Inputs:       A: Array (sorted) that you want to search
%               n: Length of array A
%               num: Number you want to search in array A
%
% Outputs:      index: Return position in A that A(index) == num
%                      or -1 if num does not exist in A
%
% Description:  This function find number in array (sorted) using binary
%               search
%
% Complexity:   O(1)    best-case performance
%               O(log_2 (n))    worst-case performance
```

```
%                O(1)       auxiliary space
%
% Author:        Trong Hoang Vo
%                hoangtrong2305@gmail.com
%
% Date:          March 31, 2016
%-------------------------------------------------------------------------

left = 1;
right = n;
flag = 0;

while left <= right
    mid = ceil((left + right) / 2);

    if A(mid) == num
        index = mid;
        flag = 1;
        break;
    else
        if A(mid) > num
        right = mid - 1;
        else
            left = mid + 1;
        end
    end
end

if flag == 0
    index = -1;
end

end
```

Recursion is a powerful tool to simplify logic in algorithms. Not only can algorithms be recursive, but so can our basic data structures.

1.2.2 Divide and Conquer

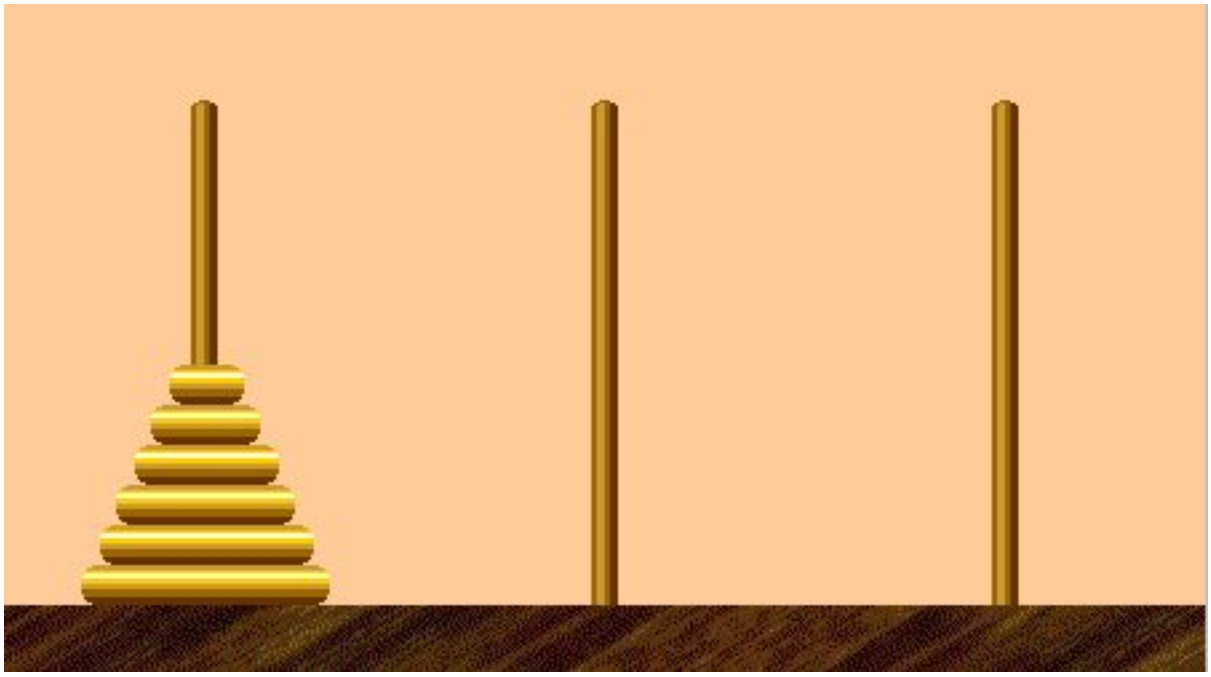
The true power of recursion lies in divide and conquer mechanisms. The ***divide and conquer method*** aims to:

- **divide** a problem into equal sized sub-problems recursively;
- **conquer** each problem separately, solving them recursively;
- **combine** results.

The idea is that each sub-problem is much easier to solve than the original problem, saving us time and simplifying the work involved. The "divide and conquer" technique is useful for, among other things,

- matrix multiplication
- sorting algorithms
- calculating a ***Fourier transform***

The ***Towers of Hanoi*** is a classic puzzle that can be elegantly solved using a divide and conquer algorithm. Take a look at the following configuration of disks and towers:



All the disks are currently on a single tower. Furthermore, every disk is stacked on top of disks that are strictly larger than it. Disks are moved from one tower to another, and they can never be set aside, must be on one of the three towers. And disks must always be stacked on an empty tower on on top of larger disks. The goal of the puzzle is to, by a sequence of disk moves, end up with all the disks stacked in order on another tower. Here is code to solve the Towers of Hanoi puzzle:

```
function [] = towers(n, frompeg, topeg, auxpeg)
% Towers of Hanoi
% Example:
%   towers(5, 'A','C','B') where n = 5 pegs

if n==1
    fprintf('\t move disk 1 from peg %c to peg %c \n', frompeg, topeg);
else
    towers(n-1,frompeg,auxpeg,topeg);
    fprintf('\t move disk %d from peg %c to peg %c \n', n, frompeg, topeg);
    towers(n-1,auxpeg,topeg,frompeg);
end
```

We can also count the number of moves needed in Towers of Hanoi with this:

```
function num_moves = hanoicount(n)
% Counts number of moves needed in Towers of Hanoi
% Example:
%   num_moves = hanoicount(20) for n = 20 pegs

if n==1
    num_moves = 1;
else
    num_moves = 2.*hanoicount(n-1)+1;
end
```

In summary, the "divide and conquer" technique is applicable to a wide array of problems. It's especially useful in certain problems involving wide arrays.

1.2.3 Mergesort

Sorting is one of the most important operations in computer science, so we want it to be as fast as possible. We will use "divide and conquer" in a very efficient sorting algorithm called ***mergesort.*** At a high level, mergesort applies the three steps of ***divide and conquer*** to sort a list of numbers:

1. **Divide:** Split the list into two (approximately) equally-sized lists
2. **Conquer:** Sort each of the two lists separately (using mergesort itself)
3. **Combine:** Given two sorted lists of approximately the same size, merge them into one big sorted list.

Here is code to implement mergesort:

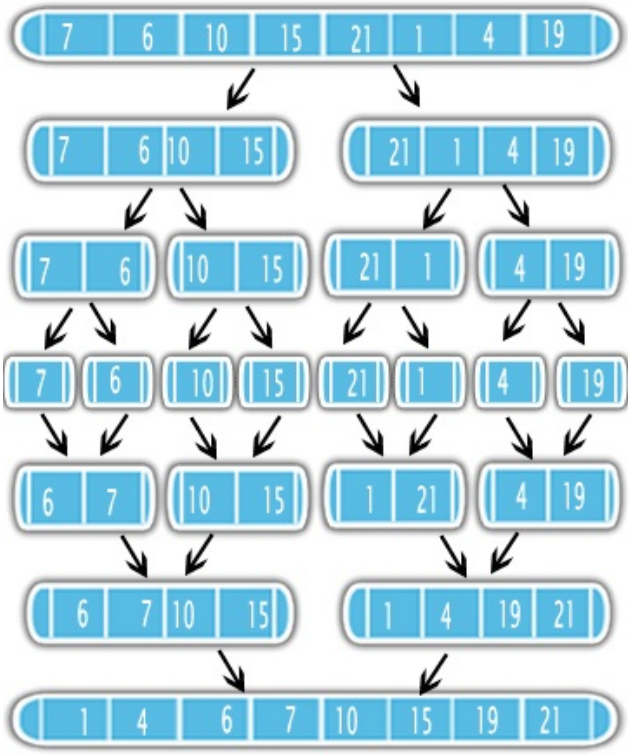
```
function z = merge(x,y)
% x is a row n-vector with x(1) <= x(2) <= ... <= x(n)
% y is a row m-vector with y(1) <= y(2) <= ... <= y(m)
% z is a row (m+n)-vector comprised of all the values in x and y, sorted so
% that z(1) <= ... <= z(m+n)

n = length(x); m = length(y); z = zeros(1,n+m);
ix = 1;      % The index of next x-value to select.
iy = 1;      % The index of next y-value to select.
for iz = 1:(n+m)
    % Determine the iz-th value for the merged array...
    if ix > n
        % All done with x-values. Select the next y-value.
        z(iz) = y(iy); iy = iy+1;
    elseif iy > m
        % All done with y-values. Select the next x-value.
        z(iz) = x(ix); ix = ix+1;
    elseif x(ix) <= y(iy)
        % The next x-value is less than or equal to the next y-value.
        z(iz) = x(ix); ix = ix+1;
    else
        % The next y-value is less than the next x-value.
        z(iz) = y(iy); iy = iy+1;
    end
end

function y = mergeSort(x)
% x is a vector.
% y is a vector consisting of the values in x sorted from smallest to
% largest.
%
% Example:
% a = [4 1 6 3 2 9 5 7 6 0];
% b = mergesSort(a);

n = length(x);
if n==1
    y = x;
else
    m = floor(n/2);
    % Sort the first half...
    y1 = mergeSort(x(1:m));
    % Sort the second half...
    y2 = mergeSort(x(m+1:n));
    % Merge...
    y = merge(y1,y2);
end
```

The nature of the mergesort algorithm lends itself very cleanly to a recursion routine consisting of the following steps:

1. Split the list into two (approximately equal) parts.
2. Sort each of these two lists recursively.

3. Merge the two parts into the new sorted list, by sequentially comparing the first elements of each list, moving the smaller one to the end of a new list.
4. Return the new list.

The approach can be summarized in the following image:

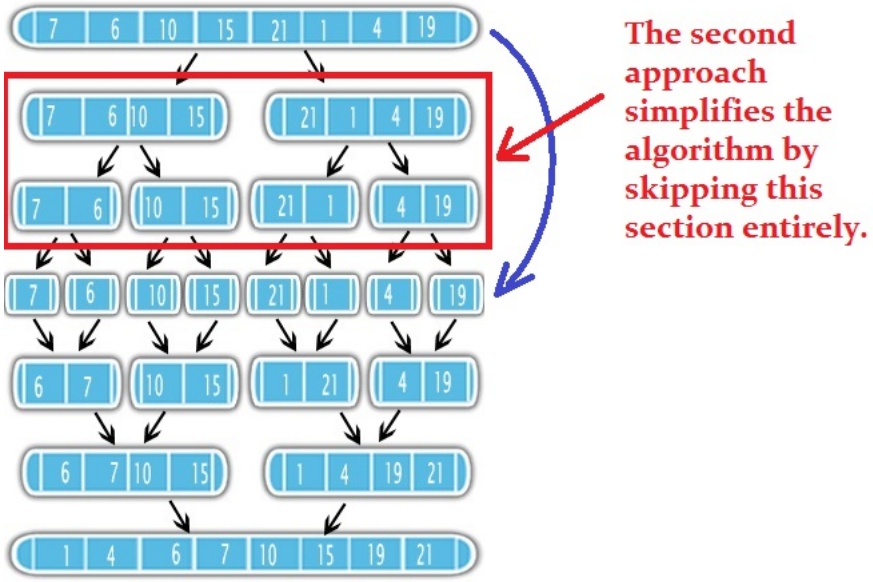


We can also describe the steps of the algorithm a little differently:

1. Split the N elements of the list into N separate lists, each of size one.
2. Pair adjacent lists and merge them, resulting in about half as many lists each about twice the size.
3. Repeat step 2 until you have one list of size N.

This is because after the last recursive calls, we are operating on arrays of size 1, which cannot be split any further and are trivially sorted themselves, thus giving us our base case.

Note that this second approach simplifies the first half of the algorithm, eliminating the divide and conquer algorithm used to separate the list of N elements into N lists each with a single element:



The length of time for the algorithm to run in **big O notation** is given by:

Runtime
$= O((\text{the no. of steps}) \times (\text{worst case no. of comparisons}))$
$= O\left((N) \times (\log_2 N)\right)$
$= O\left(n \log_2 N\right)$

This is asymptotically faster than $O\left(N^2\right)$ for insertion sort. So for $1,000,000$ elements, we could be looking at a speed up factor of $\frac{N}{\log_2 N}$, or $\frac{1{,}000{,}000}{20} = 50,000$. It turns out that the runtime of $O\left(N \log N\right)$ is the best we can do, on average, for a comparison-based sorting algorithm. There are many algorithms with this average case run-time, differing in their approach, space complexity, worst-case runtimes, and "real-world" performance.

However, all of them are valuable even in modern libraries, such as for hybrid approaches like ***Timsort.*** They can also provide us with powerful tools for problem-solving, especially for those that require a "divide and conquer" approach.

1.2.4 Quicksort

Quicksort, just like mergesort, is a "divide and conquer" sorting algorithm that runs in $O\left(N \log_2 N\right)$ time in the average case. In fact, when implemented well, quicksort in practice can be 2~3 times faster than mergesort. Although it has a worst-case runtime of $O\left(N^2\right)$, overall quicksort tends to outperform mergesort in the real world, making it an attractive choice for many programmers.

The main difference between quicksort and mergesort is in where the "heavy lifting" is done. In mergesort, the dividing step is trivial and all the work is in combining the merged sub-lists at the end. However, the opposite is true of quicksort, where the dividing step does all the work, and the combining step is non-existent.

Framed in our "divide and conquer" framework:

- **Divide:** Pick a pivot element (typically the last item in the array). All numbers lower than this value go to the left and all elements higher to the right. For quicksort, this step is commonly known as ***partitioning.***
- **Conquer:** All elements to the left are fed recursively back into the algorithm, as are elements to the right.
- **Combine:** No need to anything at all. At this point, the data should be sorted.

In the ideal case, every time we select the pivot point, it will be such that half (or as close to half as possible) of the numbers get put on the left (i.e. are lower in value than the pivot) and half to the right. This allows for the size of each recursive sub-problem to be half of what it was before, giving us an overall $O\left(N \log_2 N\right)$ runtime.

If, on the other hand, the pivot is close to the minimum or maximum value among the possible choices each time, the overall runtime approaches the worst case of $O\left(N^2\right)$.

The key to a successful sort is picking the right pivot point. The following are some common approaches:

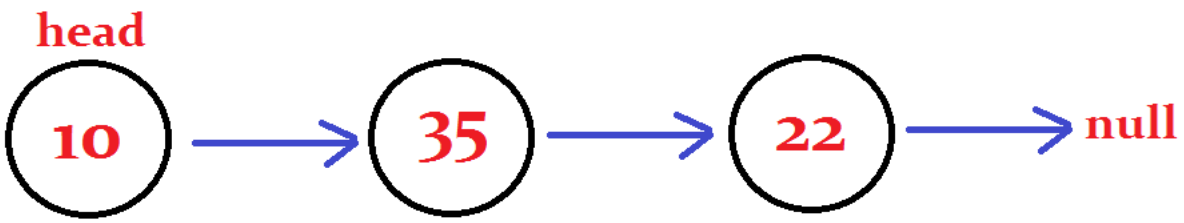
1. Select a random pivot point.
2. Select the first or last element of the array.
3. **"Median of three" method:** choose the first, middle, and last elements, and choose the median.
4. Use an algorithm to find the median.

If the data is sorted randomly, 1 and 2 are equivalent. However, the extra overhead in choosing 3 or 4 can be beneficial.

1.2.5 Linked List

We've explored recursion as a technique in algorithms. It can also be implemented in data structures, which are widely used in computer science.

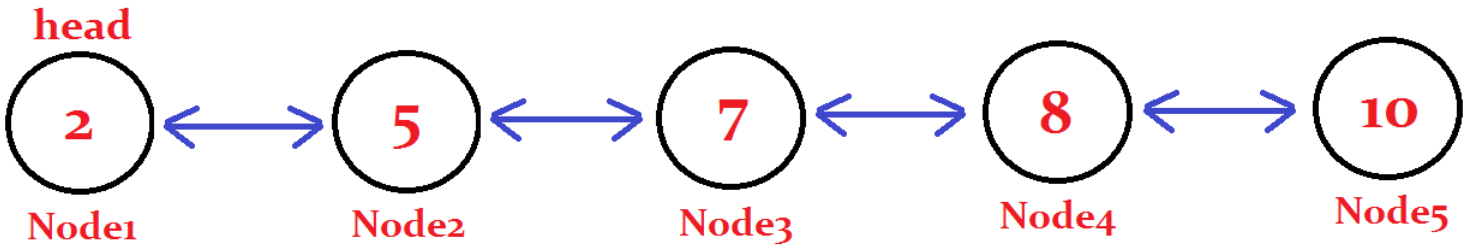
A ***linked list*** is a linear data structure that holds data in nodes. The nodes hold a piece of data as well as a pointer to another node ("the link"). One can picture a linked list as follows:



In the above example, the first node contains the number 10 and points to the second node. The second node contains the number 35 and points to the third node, and the third node contains the number 22 and points to *null*. To implement linked lists in MATLAB, the ***dlnode*** class can be used. This function creates doubly linked lists ("bi-directional") in which each node contains:

- Data array
- Handle to the next node
- Handle to the previous node

This is an illustration of a doubly linked list:



Here is code to implement a doubly-linked list using ***object-oriented programming (OOP):***

```
% Object-Oriented Programming
% Doubly-Linked List
%
% Example:
% Makes doubly-linked list with three nodes with data values 1, 2, and 3:
% n1 = dlnode(1);
% n2 = dlnode(2);
% n3 = dlnode(3);
% n2.insertAfter(n1)    insert n2 after n1
% n3.insertAfter(n2)    insert n3 after n2
% n1.Next               points to n2
% n2.Next.Prev          points back to n2
% n1.Next.Next          points to n3
% n3.Prev.Prev          points to n1

classdef dlnode < handle
% DLNODE A class to represent a doubly-linked list.
% Multiple dlnode objects may be linked together to create linked lists
% Each node contains a piece of data and provides access to the next and
% previous nodes.
    properties
        Data
    end
    properties(SetAccess = private)
        Next
        Prev
    end

    methods
        function node = dlnode(Data)
        % DLNODE Constructs a dlnode object.
        if nargin > 0
            node.Data = Data;
        end
        end

        function insertAfter(newNode, nodeBefore)
            % insertAfter Inserts newNode after newBefore
```

```
            disconnect(newNode);
            newNode.Next = nodeBefore.Next;
            newNode.Prev = nodeBefore;
            if ~isempty(nodeBefore.Next)
            nodeBefore.Next.Prev = newNode;
            end
            nodeBefore.Next = newNode;
        end

        function insertBefore(newNode, nodeAfter)
            % insertBefore Inserts newNode before nodeAfter
            disconnect(newNode);
            newNode.Next = nodeAfter;
            newNode.Prev = nodeAfter.Prev;
            if ~isempty(nodeAfter.Prev)
                nodeAfter.Prev.Next = newNode;
            end
            nodeAfter.Prev = newNode;
        end

        function disconnect(node)
            % DISCONNECT Removes a node from a linked list
            % The node can be reconnected or moved to a different list
            if ~isempty(node.Prev)
                node.Prev.Next = node.Next;
            end
            if ~isempty(node.Next)
                node.Next.Prev = node.Prev;
            end
            node.Next = [];
            node.Prev = [];
        end

        function delete(node)
            % DELETE Deletes a dlnode from a linked list
            disconnect(node);
        end

        function disp(node)
            % DISP Displays a link node
            disp('Doubly-linked list node with data:');
            disp(node.Data);
        end
    end
end
```

Recursion is powerful in the context of data structures because it allows us to quickly manipulate structures via operations that transform them into different or modified instances of the same structure. Furthermore, we can draw from the same set of operations to perform at any part of a recursive data structure, and this will prove incredibly useful in some important algorithms involving another recursive structure, the tree.

2 Introduction to Machine Learning

People are great at finding patterns. Whether it's identifying pictures, or estimating values, humans can do it all without even realizing it's difficult. But everyone has limits, and when there are three hundred variables to keep track of and thousands of elements to process, something new is needed.

Machine learning swoops in where humans fail. ***Machine learning*** is a type of statistics that places emphasis on the use of advanced computational algorithms. We will develop the mathematical basis needed to understand how problems of classification and estimation work. We'll implement these techniques and apply them to real-world problems in neuroscience.

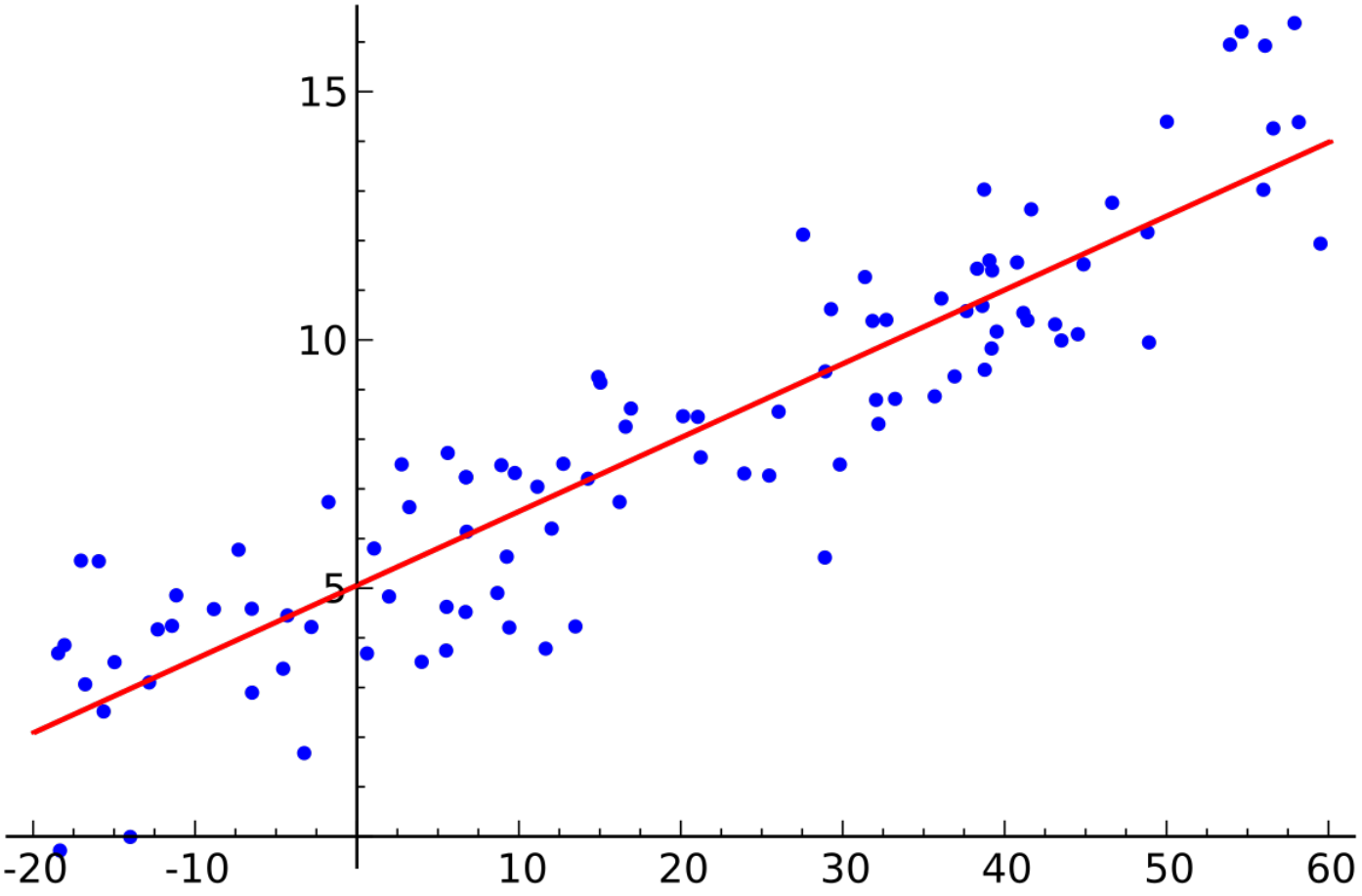
2.1 Linear Regression

2.1.1 Introduction

Many data sets have an approximately linear relationship between variables. In these cases, we can predict one variable using a known value for another using a ***best-fit line,*** a line of the form

$$y = mx + b$$

that follows the trends in the data as closely as possible.



Here, x is called the ***predictor variable*** because it will be used to predict y, while y is often called the ***response variable.*** This technique is known as linear regression, and although it is one of the simplest machine learning techniques, it is often surprisingly powerful.

Linear regression is not limited to only one predictor variable. The key concept behind it is the idea that a change in one or more predictor variables will produce a linear change in the response variable. It is a way to estimate the response variable by summing many weighted predictor variables, each of which has an impact on the final guess proportional to its importance.

2.1.2 Statistics

Machine learning is about taking in information and expanding on it. Techniques from statistics enable us to do so. To find a best-fit line, we begin by calculating five values about our data. If we represent our data sets as collections of points on a scatter plot, these values are the ***means*** of x and y, the ***standard deviations*** of x and y, and the ***correlation coefficient.***

If there are n data points, then the mean of x is simply the sum of all x values divided by n. Correspondingly, the mean of y is the sum of all y values divided by n.

After calculating the means (typically denoted as $\bar{x}$and $\bar{y}$), we can find the standard deviations for the data set through the following formulae:

$$SD_x = \sqrt{\frac{1}{(n-1)} \sum_{i=1}^{n} (x_i - \bar{x})^2}$$

$$SD_y = \sqrt{\frac{1}{(n-1)} \sum_{i=1}^{n} (y_i - \bar{y})^2}$$

The standard deviation of a data set gives a good idea of how close an average data point will be to the mean. A low standard deviation means that data points tend to cluster around the mean, while a large standard deviation implies that they will be more spread out.

After we have SD_x and SD_y, we only have the correlation coefficient, usually denoted by r, left to calculate. This is difficult to calculate by hand, but the process for doing so is quite simple. The first step is to convert x and y to standard units. For $1 \leq i \leq n$, we must put each value x_i through the formula

$$\frac{x_i - \bar{x}}{SD_x},$$

which outputs the number of standard deviations x_i above the mean. Each value y_i should be put through the analogous process with $\bar{y}$ with SD_y. We then compute the correlation coefficient by taking the average of $x$and $y$for each of the points. This is given by

$$r = \frac{1}{n} \cdot \sum_{i=1}^{n} (x_i - y_x)$$

The correlation coefficient indicates how linearly correlated the data is. Its value ranges from -1 to 1. If r is close to zero, then the data is barely correlated at all, at least not with a linear relationship. However, if $r$is close to 1, then the data is correlated and can be approximated well by a best-fit line with a positive slope. Conversely, if $r$is close to -1, the data is correlated and can be approximated well by a best-fit line with a negative slope.

The best-fit line is $y = rx$, where r is the correlation coefficient. It can also be written as

$$y - \bar{y} = \frac{rSD_y}{SD_x} (x - \bar{x})$$

In this representation, the best-fit line has a slope of $\frac{rSD_y}{SD_x}$ and must pass through the point $(\bar{x}, \bar{y})$.

2.1.3 Linear Algebra

Statistical tools are extremely powerful tools when analyzing data, but they are not the only tools. Linear algebra, for instance, often presents a more intuitive way to view the best-fit lines that is easier to generalize. The best-fit line given by the equation

$$y - \bar{y} = \frac{rSD_y}{SD_x} (x - \bar{x})$$

is actually known as the ***least squares regression line,*** which means that if we sum the square of the vertical distance from each data point to the best-fit line, the result will be less than it would be for any other line.

This definition allows us to define an ***error function*** for any given line $y = mx + b$ which outputs the ***sum of squared errors,*** or the sum of the square of each point's vertical distance from a line. This quantity is often abbreviated as ***SSE.*** We use this as a measure of how much a regression line deviates from the actual data set. The least squares

regression line is the line for which the error function is at its minimum value. If we put this in mathematical terms, we find the formula

$$SSE = \sum_{i=1}^{n} (y_i - mx_i - b)^2$$

We can think of our best-fit line as the line with values of m and b that minimize the error function. This is extremely useful because it gives us a concrete set of criteria of our best-fit line which we can expand on to suit our needs. For instance, we can use linear algebra techniques to compute the best-fit line.

Say we have a data set containing n points:

$$(x_i, y_i), (x_2, y_2), ..., (x_n, y_n)$$

We need to find a formula which can give us the least squares regression line for our data set, so a logical first step is to put our variables into linear algebra terms.

First, we realize that since every line can be represented by the equation $y = mx + b$, we can also represent every line with a single, two-dimensional vector:

$$\overline{x} = \begin{bmatrix} m \\ b \end{bmatrix}$$

Now, we define an $n \times 2$ matrix A. For $1 \leq i \leq n$, the i^{th} row of A will contain x_i in the first column and 1 in the second:

$$A = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix}$$

This definition may appear somewhat arbitrary, but it becomes useful when we multiply A by a vector representing a line. For any vector

$$\overrightarrow{x} = \begin{bmatrix} m \\ b \end{bmatrix},$$

the vector given by $A\overrightarrow{x}$ will contain the $y-$values the line represented by $\overrightarrow{x}$ will predict for each $x-$value in our data set. In other words, calculating $A\overrightarrow{x}$ is like feeding the $x-$value from each point into the function represented by $\overrightarrow{x}$.

As a result, if we define a new vector $\overrightarrow{b}$ so that it is the i^{th} element will be y_i from our data set, we can find the vertical distance between each point and the $y-$value predicted for it by subtracting $\overrightarrow{b}$ from $A\overrightarrow{x}$.

We can now find the SSE by individually squaring the values inside $A\overrightarrow{x} - \overrightarrow{b}$ and adding them together. Interestingly, this process is equivalent to squaring the length of $A\overrightarrow{x} - \overrightarrow{b}$, so the SSE is equal to the squared distance in $n-$dimensional space between $A\overrightarrow{x}$ and $\overrightarrow{b}$. In other words,

$$SSE = \left\| A\overrightarrow{x} - \overrightarrow{b} \right\|^2.$$

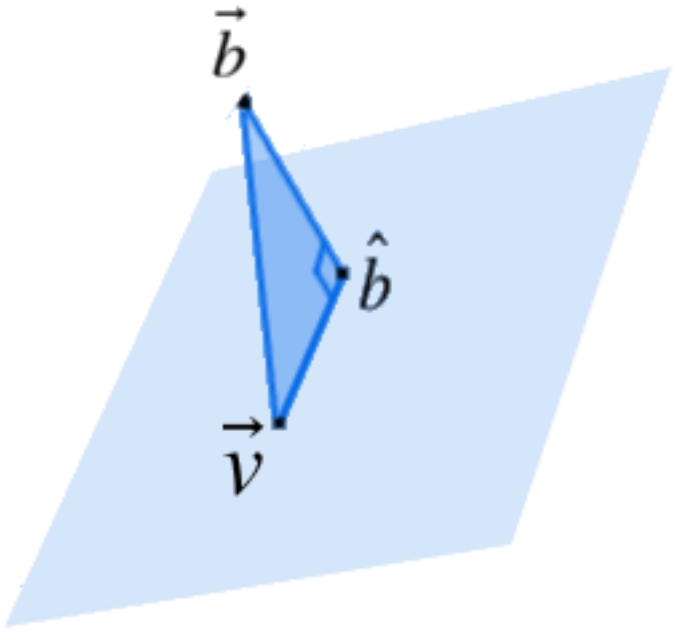
When we are minimizing the SSE, we are minimizing the distance between $A\overrightarrow{x}$ and $\overrightarrow{b}$ since minimizing a squared positive value will also minimize the value itself. In other words, we need to find a vector $\overrightarrow{x}$ for which $A\overrightarrow{x}$ is as close as possible to $\overrightarrow{b}$.

Suppose we have a column space in $\mathbf{R^3}$,W, a vector $\overrightarrow{b}$, and $A\overrightarrow{x}$, the point closest to $\overrightarrow{b}$ on W. Intuitively, it would make sense if $A\overrightarrow{x}$ was equal to $\hat{b}$, the projection of $\overrightarrow{b}$ onto W, and as it turns out there is a simple proof this is true.

We begin by picking an arbitrary point on W, $\overrightarrow{v}$. To get from $\overrightarrow{v}$ to $\overrightarrow{b}$, we can first travel to $\hat{b}$, and then travel perpendicularly from $\hat{b}$ to $\overrightarrow{b}$. $\hat{b}$ is given by drawing a perpendicular line from $\overrightarrow{b}$ to W, so Pythagorean's theorem shows us that

$$\left\| \overrightarrow{v} - \overrightarrow{b} \right\|^2 = \left\| \hat{b} - \overrightarrow{v} \right\|^2 + \left\| \overrightarrow{b} - \hat{b} \right\|^2.$$

This is depicted in the picture below:



This means that no point on W can be closer to $\vec{b}$ than $\hat{b}$, and that $A\vec{x}$ must equal $\hat{b}$. In other words, if we draw a perpendicular line from $\vec{b}$ to W, the point where it intersects with W will be the point on W closest to $\vec{b}$.

2.1.4 Higher Dimensions

One of the benefits of least squares regression is that it is easy to generalize from its use on scatter plots to 3D or even higher dimensional data. Previously, we learned that when least squares regression is used on 2D data, the SSE is given by the formula

$$SSE = \sum_{i=}^{n} (y_i - mx_i - b)^2 .$$

This gives us a good idea of what a higher dimensional error function will look like. We will attempt to modify this formula so that it works for higher dimensional linear regression. Instead of outputting a best-fit line, this formula will now output a best-fit hyperplane– a linear equation in higher dimensions.

We can begin our derivation by representing our best-fit equation with a vector

$$\vec{x} = \begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ m_n \\ b \end{bmatrix} .$$

Now, we must create a matrix A which, when multiplied with $\vec{x}$, outputs a vector containing the predicted value of $y$for each data point in the set.

Previously, we did this by making A's first column the $x-$values of all data points and its second column a line of ones. Now, we can achieve the same results for higher dimensions by adding another column to A for each additional predictor variable. This is shown below for a data set with $n$points and $p$predictor variables:

$$A = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{2p} & 1 \\ x_{21} & x_{22} & \dots & x_{2p} & 1 \\ \vdots & & & & \vdots \\ x_{n1} & x_{n2} & \dots & x_{np} & 1 \end{bmatrix} .$$

At this point, the derivation is exactly the same as before. We have to find the vector $\overrightarrow{x}$ for which $A\overrightarrow{x}$ is as close as possible to b, and once again we can do this by solving the equation

$$A^T\overrightarrow{b} = A^T A\overrightarrow{x}.$$

After that, we have our answer. The elements of $\overrightarrow{x}$ will give the coefficient values for the best-fit hyperplane. But there's one major problem. What if the points in a data set are very predictable, but not in a linear fashion?

As it turns out, there is a simple way to expand on our previous model. We can just add new, nonlinear terms to our function and update the rest of our math accordingly. Generally, this is done by adding powers of the predictor variables, in which case this process is known as ***polynomial regression.***

For instance, say we have a simple data set in which there is one predictor variable $x$and one response variable y. The only twist is that we suspect $y$to be best represented by a second degree polynomial of x. Instead of representing the data with a best-fit line

$$y = mx + b,$$

We should now represent it with a best-fit polynomial

$$y = m_1x^2 + m_2x + b.$$

In many ways, this is the same as creating another predictor variable. We have taken each point in our data set and added another value, x^2. After this step, we can calculate the coefficients as we normally would in higher dimensional linear regression.

2.1.5 Limitations

Linear regression is clearly a very useful tool. Whether you are analyzing crop yields or estimating next year's GDP, it is always a powerful machine learning technique.

However, it does have limitations. The most obvious is that it will not be effective on nonlinear data. Using linear regression means assuming that the response variable changes linearly with the predictor variables.

Outliers are another confounding factor when using linear regression. These are elements of a data set that are far removed from the rest of the data. Outliers are problematic because they are often far enough from the rest of the data that the best-fit line will be strongly skewed by them, even when they are present because of a mistake in recording or an unlikely fluke.

Commonly, outliers are dealt with simply by excluding elements which are too distant from the mean of the data. A slightly more complicated method is to model the data and then exclude whichever elements contribute disproportionately to the error.

It is not impossible for outliers to contain meaningful information though. One should be careful removing test data.

Another major setback to linear regression is that there may be ***multicollinearity*** between predictor variables. That is the term for when several of the input variables appear to be strongly related. Multicollinearity has a wide range of effects. However, the major concern is that it allows many different best-fit equations to appear almost equivalent to a regression algorithm.

As a result, tools such as least squares regression tend to produce unstable results when multicollinearity is involved. There are generally many coefficient values which produce almost equivalent results. This is often problematic, especially if the best-fit equation is intended to extrapolate to future situations where multicollinearity is no longer present.

Another issue is that it becomes difficult to see the impact of single predictor variables on the response variable. For instance, say that two predictor variables x_1 and x_2 are always exactly equal to each other and therefore perfectly correlated. We can immediately see that multiple weightings, such as $mx_1 + m \cdot x_2$ and $2m \cdot x_1 + 0 \cdot x_2$, will lead to the exact result. Now it's nearly impossible to meaningfully predict how much the response variable will change with an increase in x_1 because we have no idea which of the possible weightings best fits reality. This both decreases the utility of our results and makes it more likely that our best-fit line won't fit future situations.

The property of ***heteroscedasticity*** has also been known to create issues in linear regression problems. **Heteroscedastic data sets** have widely different standard deviations in different areas of the data set, which can cause problems when some points end up with a disproportionate amount of weight in regression calculations.

Another classic pitfall in linear regression is ***overfitting,*** a phenomenon which takes place when there are enough variables in the best-fit equation for it to mold itself to the data points almost exactly.

Although this sounds useful, in practice it means that errors in measurement, outliers, and other deviations in the data have a large effect on the best-fit equation. An overfitted function might perform well on the data used to train it, but it will often do very badly at approximating new data. Useless variables may become overvalued in order to more exactly match data points, and the function may behave unpredictably after leaving the space of the training data set.

2.1.6 Alternatives

Least squares linear regression is probably the most well-known types of regression, but there are many other variants which can minimize the problems associated with it.

A common one is ***ridge regression.*** This method is very similar to least squares regression but modifies the error function slightly.

Previously, we used the sum of square errors of the regression line as a measure of error, but in ridge regression we seek to minimize the squared values of coefficients as well. This gives the error function

$$Error = \sum_{i=1}^{n} (y_i - m_1 x_{1i} - m_2 x_{2i} - ... - m_p x_{pi} - b)^2 + \lambda \sum_{i=1}^{p} (m_i)^2 .$$

Here, the value of lambda changes how aggressively coefficients are dampened. Notice that this error function does not penalize the size of the $y-$intercept.

A close relative of ridge regression is simply know as the ***"lasso."*** This also penalizes the size of coefficients in the error function, but does so based on their linear size instead of their squared size. Therefore, error is given by

$$Error = \sum (y_i - m_1 x_{1i} - m_2 x_{2i} - ... - m_p x_{pi} - b)^2 + \lambda \sum_{i=1}^{p} |m_i|$$

It is not at all obvious why lasso would have behavior significantly differing from ridge regression, but there is an interesting geometric reason for the differences. However, to demonstrate this, we must first change the way we view both techniques.

In ridge regression, it turns out that for any values of λ we pick, it's possible to find a value for λ_2 such that minimizing

$$\sum_{i=1}^{n} (y_i - m_1 x_{1i} - m_2 x_{2i} - ... - m_p x_{pi} - b)^2 + \lambda \sum_{i=1}^{p} \left(m_i^2\right)$$

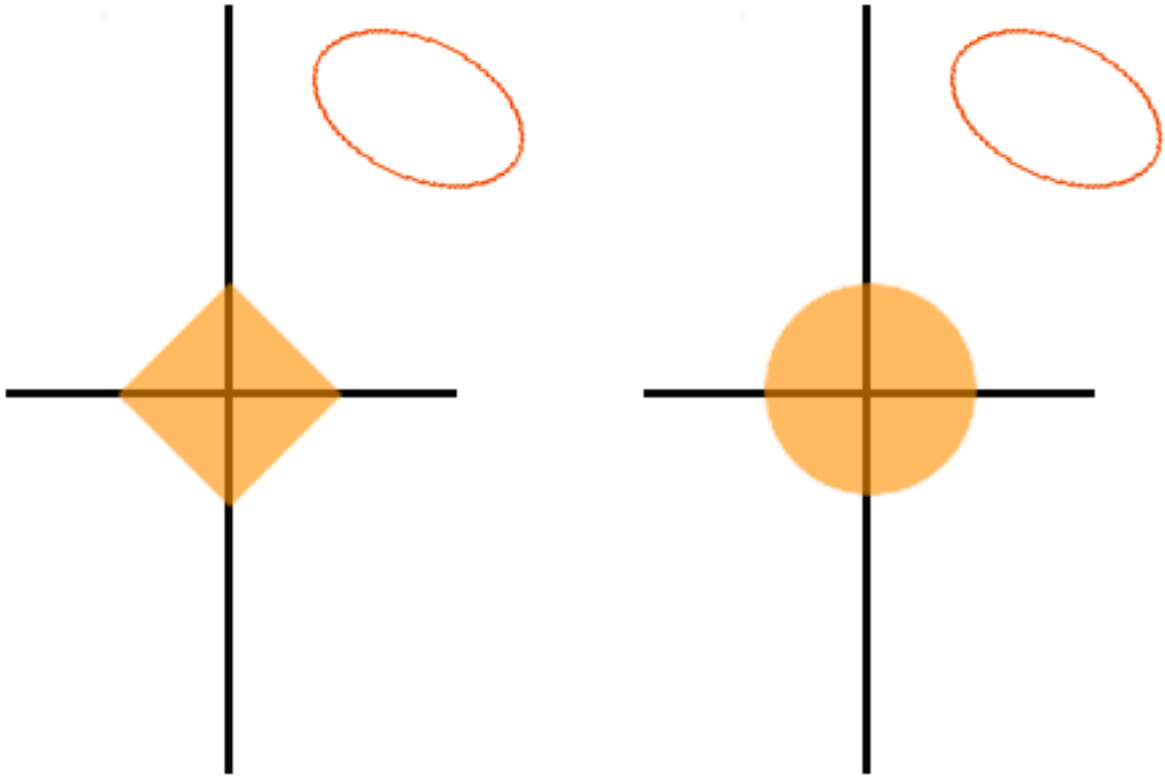
is equivalent to minimizing the SSE when

$$\sum_{i=1}^{p} \left(m_i^2\right) \leq \lambda_2.$$

Similarly, for any value of λ there is some value of λ_2 such that using lasso is equivalent to minimizing the SSE when

$$\sum_{i=1}^{p} |m_i| \leq \lambda_2.$$

A useful way to view the SSE when there are two predictor variables is shown in the pictures below. Here, the $x-$ and $y-$ axes represent the values of coefficients in a best-fit plane, and the ellipses shown all pairs of coefficients which produce a certain value of the SSE for a data set. As the SSE increases, the ellipses get larger.



Also in the pictures are two areas. The diamond represents the coefficient values allowed by lasso. The disk represents the possible coefficient values in ridge regression.

Theres is one important difference between lasso and ridge regression. *Lasso is capable of reducing the weights of predictor variables to zero.* This is useful if one wants to cull predictor variables, among other things. Usually, this is done when there are many predictor variables and using too many in the model will cause overfitting or make it overcomplicated.

Another alternative to linear regression with comparable simplicity is ***K-nearest neighbors regression,*** or ***KNN regression*** for short**.** Say we have an arbitrary point $\overrightarrow{x}$ which holds values for all our predictor variables. We want to estimate the corresponding value of the resultant variable, using the data set and $\overrightarrow{x}$.

Now, we plot the predictor variables for the points in our data set, ignoring the resultants, and pick out the k points geometrically closest to $\overrightarrow{x}$. The estimate KNN regression provides is simply the average of the resultant values for these points.

One useful property of KNN regression is that it makes very few assumptions about the data sets it builds on. Unlike linear regression, which assumes linear relationships, KNN regression can accomodate nearly anything.

Additionally, by adjusting the value of k, we can change the flexibility of KNN regression. If we want to account for even the smallest trends in our data set, we can pick a very small $k-$value. On the other hand, larger values of k will eliminate smaller deviations in favor of larger trends.

2.2 Linear Classification

2.2.1 Indicator Matrix

One of the problems with data is that there is no requirement for it to be continuous. Imagine recording half a heart attack or analyzing an animal to find that it is ninety percent mammal.

We refer to cases like these, in which the resultant variable is qualitative instead of quantitative, as classification problems. These are tricky to deal with using techniques such as linear regression, so before we can deal with classification problems adequately, we must build on our old techniques and even introduce a couple of new ones.

Of course, the first step to solving classification problems mathematically is representing them mathematically. For instance, let's say we run a nature preserve containing lions, tigers, and bears. For each of our animals we have various measurements on appetite, weight, and anything else we could think of, and we want to use this data to teach a machine to differentiate between species.

However, before this can happen we need a way to record what species our various animals are from. Naturally, our first thought is to simply assign a different number to each species. For instance, we could say lions are ones, tigers are twos, and bears are threes.

An alternative school of thought is to represent each animal with three variables, (x_1, x_2, x_3). Each variable will correspond to one type of animal, and will be equal to one if used to record that animal. Otherwise, they will be zero. So for instance, a tiger would be $(0, 1, 0)$ while a bear would be $(0, 0, 1)$. This format is preferred because it carries very few implicit assumptions and won't skew algorithms.

Representing classifications with a vector of Boolean variables is a common practice in machine learning, known as ***hot encoding.*** It is easier to manipulate or interpret classifications in this form.

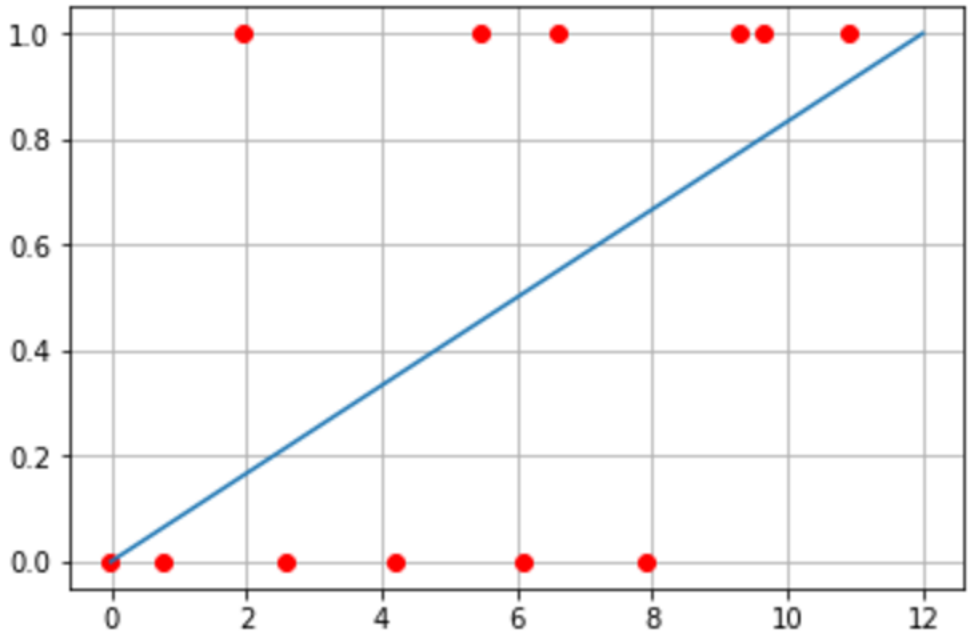
Another common practice, for which the uses will become apparent later on, is to represent a set of one of these hot vectors by placing them into the rows of a matrix known as an ***indicator matrix.*** If we have four animals, a bear, a lion, another bear, and a tiger, the matrix representing their classifications is given by

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

The goal of any classification algorithm is to find the class a data point is most likely belong to. Classification through an indicator matrix does so by estimating the probability that a data point is in each class and picking the class with the highest value.

So, for each of our classes we need a function that generally outputs a value of one for points which meet the class criteria and generally outputs a zero for points that fall outside the criteria. This function isn't exactly a classifier; all it can do is to decide whether a data point is a good fit for a certain class. Conveniently, we can build this function with linear regression.

For a given class, we start by associating a "dummy variable" $y$with each point in our data set. $y$represents whether a given point is in the class; it is set to one if a point is in the class and zero if not. In many ways, this dummy variable is equivalent to the ***response variable,*** and it is what we must predict using a best-fit line. This is demonstrated below for the case of one predictor variable:



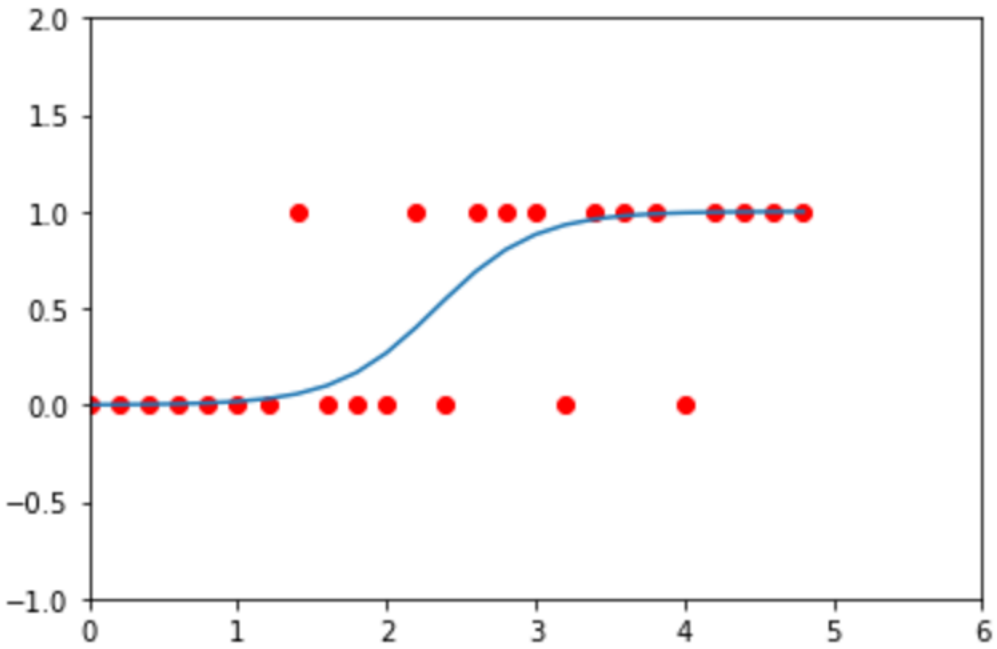
If there are K classes, we can use linear regression on the columns of our indicator matrix to generate K equations, one corresponding to each class. Then, if we are given an arbitrary set of values for our predictor variables, we can simply plug those values into each equation and put them into whichever class corresponds with the highest output.

However, there are some problems with this. One is that although the values our functions output are useful for comparisons between classes, they are awful if taken as actual probabilities. Because they are linear functions, they are fully capable of outputting values greater than one or less than zero, which is ridiculous if we are looking for the likelihood of an event. We can account for these issues by transforming a linear classification function $f(\overrightarrow{x})$ into a sigmoidal classification function of the form of $\sigma(f(\overrightarrow{x})) = \frac{e^{f(\overrightarrow{x})}}{1+e^{f(\overrightarrow{x})}}$.

2.2.2 Logistic Classification

We can take out the middleman and directly calculate a ***best-fit sigmoidal classification function.*** The process is known as ***logistic classification,*** and it is generally much more useful than the linear regression approach we learned earlier, although it is usually only used in the cases of two classes. If there are $n$variables $x_1, x_2, ..., x_n$, using logistic classifiction means calculating weights $m_1, m_2, ..., m_n$ and a bias b such that the sigmoid $\sigma(m_1x_1 + m_2x_2 + ... + m_nx_n + b)$ comes as close as possible to correctly describing our data.

We have shown an example of this below. Here there is one predictor variable, and each data point is given a value of 1 if it is in a certain class and a value of 0 if it is in another. The sigmoid is designed to maximize the likelihood of correctly classifying the entire dataset, and can be used to differentiate between these two classes. Note that this approach is distinct from using an indicator matrix. *There is only one function for two classes, not a function for each class.*



Logistic regression produces the sigmoidal function that best describes the given data. However, we have not properly explained the property that it maximizes.

Say that there are two classes, positive and negative. As with other forms of classification, our best-fit function will be produced by analyzing a large set of data points, each of which is known to be in one of the two classes. Our final function, $p(\overrightarrow{x})$, will give the chance of $\overrightarrow{x}$ being positive.

However, not all probability functions are equal. We differentiate between them by calculating the odds of our data set receiving the classifications it has if a given probability function $p(\overrightarrow{x})$ is correct. This means using $p(\overrightarrow{x})$ to calculate the probability that each point will be in the class it has and then taking the product of the results.

Assign to each point $\overrightarrow{x_i}$, a dummy variable y_i which is set to 1 if $\overrightarrow{x}$ is positive and 0 if negative. Then, expressed mathematically,

$$P(Dataset) = \prod_{i:y_i=1} p(\overrightarrow{x_i}) \prod_{i:y_i=0} 1 - p(\overrightarrow{x_i}).$$

This quantity, the probability that all of the known points have the class they do, is what is maximized by the logistic classification algorithm. The process is known as the ***maximal likelihood method*** because we are finding the highest possible likelihood for a sigmoidal probability function.

So far we've seen how to estimate a probability with logistic regression, but now how to make a classification. Generally, classifications are made by setting a ***threshold probability*** which divides two classes.

The threshold probability is adjusted based on the needs of the situation. With no reason to classify one way or another, a threshold of 0.5 might work well since it would just pick the most probable class. If, on the other hand, we are deciding whether a criminal is guilty, a very high threshold might be called for because of the consequences of an incorrect verdict.

Just as it is useful to turn a linear function into a sigmoidal one for the purpose of calculating probabilities, it is often useful to do the reverse in order to facilitate mathematics.

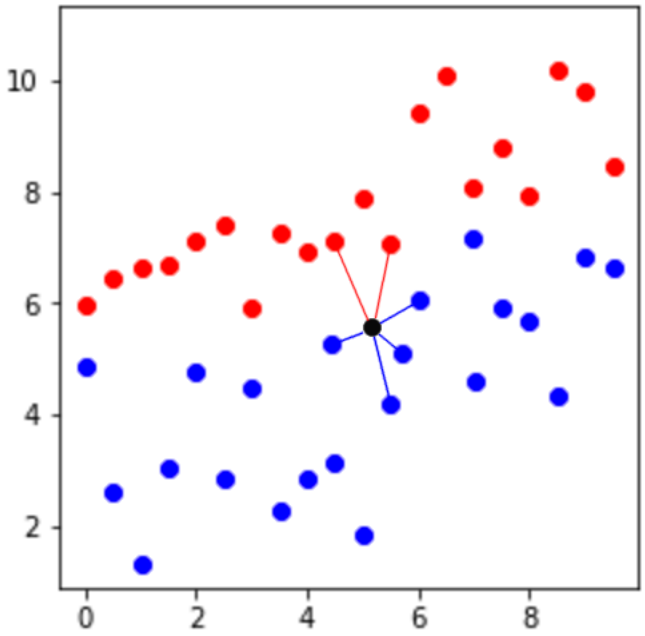
The inverse of the logistic function $\frac{e^x}{1+e^x}$ is the logit $\ln \frac{x}{1-x}$. Taking the logit of a function produced by logistic regression gives the linear function that feeds into the logistic one. Mathematically,

$$logit\left(\frac{e^{m_1x_1+m_2x_2+...+m_nx_n+b}}{1+e^{m_1x_1+m_2x_2+...+m_nx_n+b}}\right) = m_1x_1 + m_2x_2 + ... + m_nx_n + b.$$

This linear function is known as the ***log-odds,*** and although it isn't good for estimating probabilities, it is extremely useful for comparisons and optimizations. Whenever the logistic function isn't behaving, it's often a good idea to convert it to the log-odds.

2.2.3 KNN Classification

The simplest type of classification is $K-$**nearest neighbor classification, *KNN classification*** for short. To use this technique on a given data point $\vec{x}$, we start by identifying the K points nearest to $\vec{x}$ in the data set. Then we classify $\vec{x}$ as whichever class shows up the most frequently out of those K points. Below is an example of KNN classification with $K = 6$.



One issue in KNN classification stems from differences in density between classes. If one class is very common while another is relatively rare, KNN classification will be biased towards the former.

One common solution to this is to weight data points by the inverse of their distance from the point being classified. This is analogous to giving points that are further away fewer shares of the vote in determining a certain point's class.

For KNN classification to be effective, the density of points in a data set must reach a certain level around the point being classified. If we attempt to classify a point in an empty area, we will make our decision based on data points that are far removed from the place we are interested in and get useless results.

2.2.4 Perceptrons

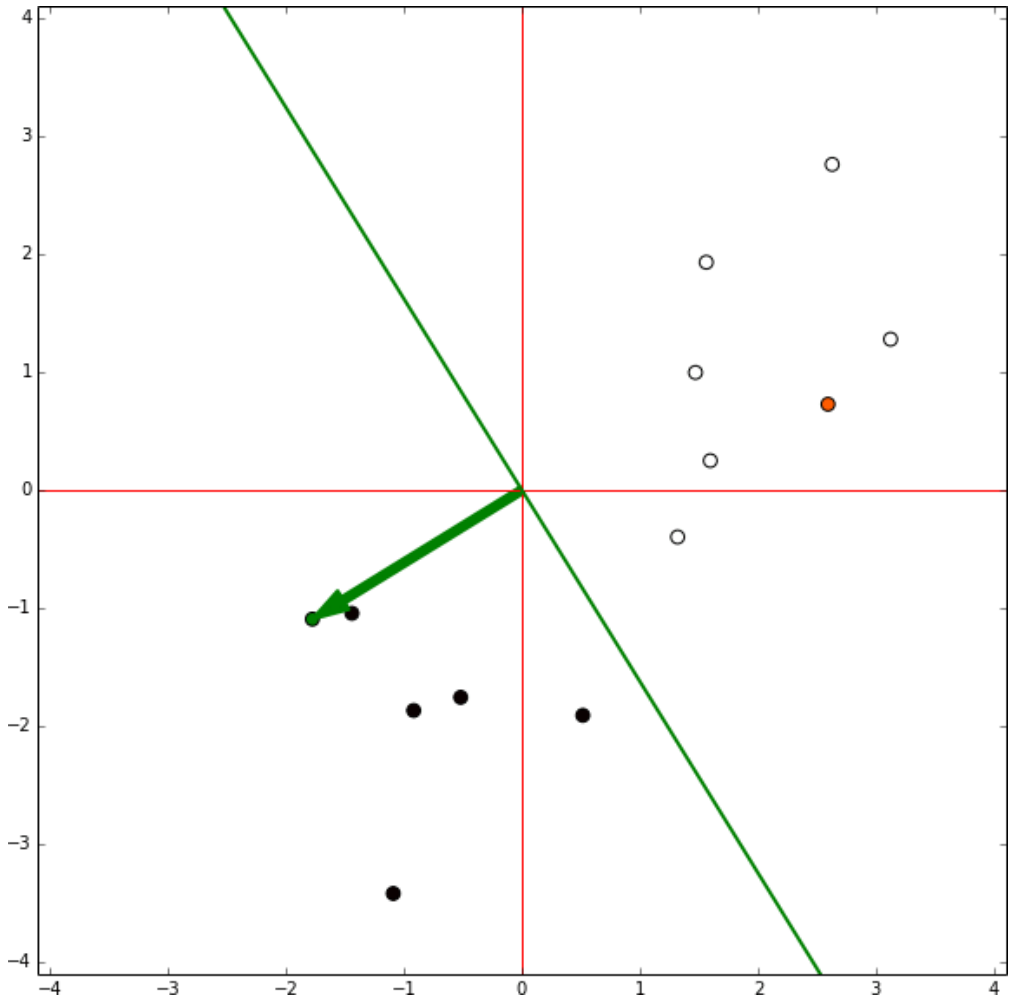
We have put most of our efforts so far into estimating the probability that certain points will be in certain classes. However, there is no rule requiring this approach to finding a boundary line. All we actually need is a division which accurately separates the classes and a function which outputs true on one side of the divider, false on the other.

This function is known as a ***perceptron.*** It can be summed up in two parts, a vector of weights $\vec{w}$ and a bias b. The class a perceptron picks for a point depends on whether the weighted predictor variables of the point outweigh the bias.

We can edit a weight vector to make sure it does a better job classifying a certain point $\vec{x}$. If $\vec{w} \cdot \vec{x}$ is less than b when it should be greater, we can just add $\vec{x}$ to $\vec{w}$ and be assured that it is more correct than it was before. The same logic applies to the opposite case as well. Subtract $\vec{x}$ to make $\vec{w} \cdot \vec{x}$ more likely to be less than b.

Similarly, we can increase the bias by one when $\vec{w} \cdot \vec{x} \geq b$ is done incorrectly, and decrease it by one in the opposite case.

Through this process, repeated enough times, it is always possible to separate the classes that actually are separable. Just go through all of the points in a data set, and if your perceptron classifies one incorrectly, add the point to your weighing vector. Eventually, if it is possible, the perceptron will ***converge*** on a solution.



3 Machine Learning and Neuroscience

Machine learning has two distinct relationships to neuroscience. As with other applied disciplines, modern machine learning methods can be very helpful in data analysis. Some examples of this include sorting the spikes picked up an extracellular electrode into spike trains from different neurons, or trying to predict what object a person is thinking about by combining evidence from different voxels in a functional magnetic resonance imaging (fMRI) scan.

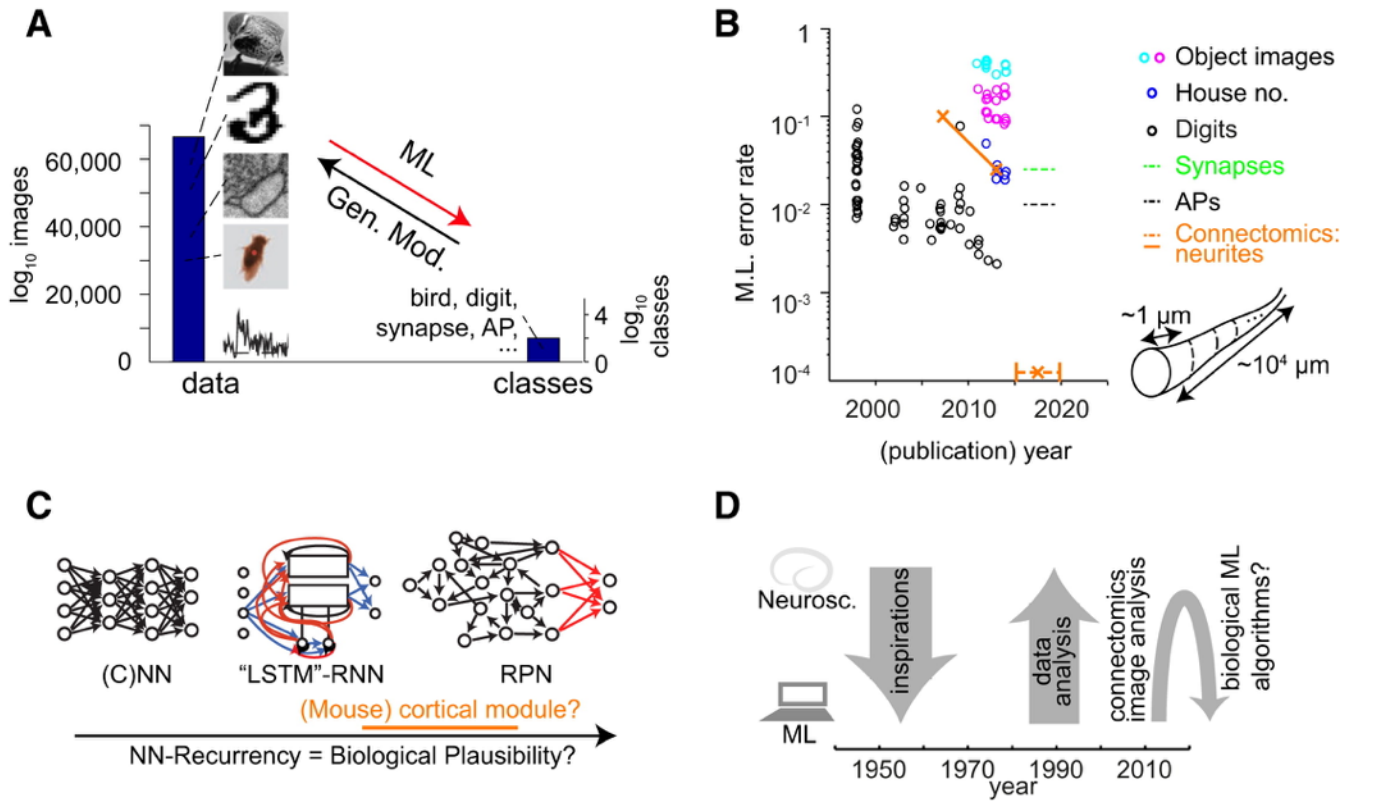
A more interesting relationship is the use of machine learning algorithms as a source of theories about how the brain works (and vice versa). For example, an algorithm called ***td-learning*** inspired a theory on the functional role of dopamine influx in the brain. This algorithm was previously used to predict actions in the game backgammon, where the future contains uncertainty.

Abstract principles that emerged from machine learning can benefit neuroscientists who study learning. One illustration of this is Crick and Mitchison's theory of rapid eye movement (REM) sleep and neural networks. Dreams happen during REM sleep. The pair proposed that a reverse learning mechanism in REM sleep removes certain undesirable modes of interaction in neural networks within the cerebral cortex.

Imagine that sensory input can be divided into broad classes: ***"directed models"*** and ***"undirected models."*** For learning to occur in an undirected model within the brain, it is necessary to include a phase in which the input is ignored and artificial data is generated from the model. Crick and Mitchison extend this argument to propose a computational role for REM sleep. If their theory is correct, then abnormalities of reverse learning might account for some aspects of schizophrenia, mania, and depression.

Most existing machine learning methods for both ***supervised*** and ***unsupervised*** learning are shallow. They do not create multiple layers of adaptive features and are all ***feed-forward.*** However, neuronal networks in the brain are highly recursive. These algorithms are then of limited interest to neuroscientists trying to understand perceptual pathways.

The first widely used method for creating multiple layers of features was the ***supervised back-propagation algorithm***. Unfortunately, its practical performance was disappointing because it required massive amounts of labeled examples (***training data***). For practical applications, it was largely replaced by ***support vector machines.*** More recently, scientists have discovered ***unsupervised*** methods for creating multiple layers of features, one layer at a time, without requiring any labels. One example is an ***autoencoder neural network,*** which is an unsupervised learning algorithm that applies back-propagation. These methods create useful high-level features, so ***deep learning*** is currently being used for tasks such as object and speech recognition. This should enrich the interaction between machine learning and neuroscience.



4 Neural Coding

4.1 Introduction

Neural coding is the study of how computational variables are encoded in neural activity. One early example of this is the legendary Hubel and Wiesel experiments on the cat visual system. The pair began with what was known: light stimulates light-sensing receptor cells in the retina of the eye and different receptor cells respond to stimuli in different parts of the retina's visual field. However, Hubel and Wiesel were studying neurons in higher-functioning areas of the brain not previously studied. They were frustrated by feeble responses in cortical neurons. They were using small spots of light or a black dot on a clear glass slide, projected onto a screen, as a stimulus. One day Hubel accidentally moved the glass slide a little too far, bringing its faint edge into view. Suddenly, those neurons started firing.

In this way, Hubel and Wiesel discovered that cortical neurons didn't respond to simple points of light, but rather to lines of a specific orientation. Some neurons responded to horizontal lines, others to vertical lines, and still others to orientations in between. For their work, the team won the 1981 Nobel Prize in Physiology or Medicine.

The Hubel and Wiesel experiments illustrate neural coding of visual sensory processing. A cat has been anesthetized and placed in front of a screen, with its eyelids open. The tip of a tungsten wire has been placed inside its skull, and lodged next to a neuron in the visual cortex. Even though the cat is no longer conscious, cortical neurons are still active. By connecting an amplifier to the tungsten wire, we can visualize weak electrical responses from the neuron being recorded. The amplified signal can also be connected to a loudspeaker, and we can *hear* neurons.

The technical term for a *spike* in the electrical signal of a neuron is an *action potential.* The frequency of spiking is dependent on the properties of the stimulus. In a *raster plot,* each black bar represents one *spike* or *action potential* and each row of black bars is a *spike train.*

4.2 Represent a spike train in MATLAB

In this tutorial, we will represent spike trains as MATLAB matrices. Let each element of a matrix represent a time interval of 1 ms. If there is a spike in this time interval, then we set the value of the element to 1, else we set it to 0. In other words, a spike train contains binary data. Open up your MATLAB command window, type the following command:

```
myFirstSpikeTrain = [0 1 0 0 0 1 0 0 1 0];
```

If this first spike train starts at time 0 and each element represents a 1 ms bin, then there are 3 spikes in this train. The spikes occur between 1-2 ms, 5-6 ms, and 9-10 ms. Based on this spike train, write code to identify:

1. What is the duration? {Ans: 10 ms}
2. What is the mean firing rate? {Ans: 300 Hz}

4.3 Generate a synthetic spike train in MATLAB

We use a concept in probability theory known as the *Poisson process* to simulate spike trains that have characteristics close to real neurons. For now, we will use a simple formula derived from the Poisson process. Dayan and Abbott explain how to so:

> *Spike sequences can be simulated by using some estimate of the firing rate,* ***fr,*** *predicted from knowledge of the stimulus, to drive a Poisson process. A simple procedure for generating spikes in a computer program is based on the fact that the estimated probability of firing a spike during a short interval of duration* ***dt*** *is* ***fr*dt.*** *The program progresses through time in small steps of size dt and generates, at each time step, a random number* ***x*** *chosen uniformly in the range between 0 and 1. If* ***x*** $<$ ***fr*dt*** *at that time step, a spike is fired; otherwise it is not.*

In this technique, we are marching through time in small steps of size **dt** (i.e. 1 ms) and deciding if there should be a spike at this time point. We can break this down into three steps:

1. Compute the product of **fr*dt.** Let's stimulate a 10 ms long spike train for a neuron firing at 100 Hz. Thus, **fr** = 100 Hz, **dt** = 1 ms and **fr*dt** = 0.1 (remember that Hz is the inverse of s and 1 ms is 1/1000 s, so fr*dt is dimensionless).
2. Generate uniformly distributed random numbers between 0 and 1. In MATLAB, use the function *rand()* to do so. *rand(1, 10)* will generate 10 randon numbers.
3. Compare each random number to fr*dt. If the product is less than fr*dt (x $<$ fr*dt), then there is a spike.

We can summarize these three steps into the code below:

```
function [spikeMat, tVec] = poissonSpikeGen(fr, tSim, nTrials )

dt = 1/1000; % s
nBins = floor(tSim/dt);
spikeMat = rand(nTrials, nBins) < fr*dt;
tVec = 0:dt:tSim-dt;
end
```

An example output is:

```
myPoissonSpikeTrain =

    0 1 0 0 0 0 0 0 1 0
```

Repeat this simulation 20 times. We could just type the commands 20 times but a more efficient way is to create a 2-D matrix. Each row of the matrix will represent one simulation. Here is the code:

```
fr = 100; % Hz
dt = 1/1000; % s
nBins = 10; % 10 ms spike train
nTrials = 20; % number of simulations
spikeMat = rand(nTrials, nBins) < fr*dt;
```

4.4 Write the poissonSpikeGen function in MATLAB

A *function* is way to package code for repeated use. Each function has an input and output. In this code, we have three inputs:

1. firing rate **fr**
2. duration of simulation **tSim**
3. number of trials **nTrials**

We have introduced a new variable here– duration of simulation **tSim.** This is another way to represent **nBins.** The number of bins (**nBins**) is the same as **tSim**/**dt,** where we have chosen **dt** to be 1 ms. Here is the code:

```
function [spikeMat, tVec] = poissonSpikeGen(fr, tSim, nTrials )

dt = 1/1000; % s
nBins = floor(tSim/dt);
spikeMat = rand(nTrials, nBins) < fr*dt;
tVec = 0:dt:tSim-dt;
end
```

This function is automatically saved as poissonSpikeGen.m by MATLAB. It has two outputs:

1. spike matrix **spikeMat**
2. time vector **tVec**

In time vector **tVec**, each time stamp is 1 ms long (**dt**). The last time stamp will start at **tSim - dt,** which represents the time interval between **tSim - dt** and **tSim.**

Let us now simulate 20 spike trains for a neuron firing at 30 Hz for a period of 1 s. First, clear the MATLAB workspacing using the function *clc.* The, we can use our function:

```
[spikeMat, tVec] = poissonSpikeGen(30, 1, 20)
```

4.5 Plot the spike matrix

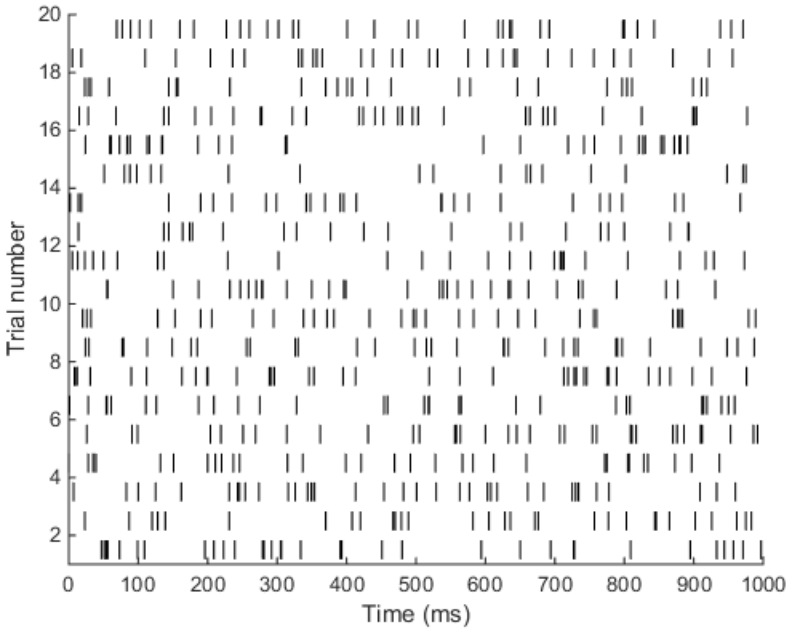
Let's visualize the data by making a raster plot. Here is the code for a function *plotRaster*:

```
function [] = plotRaster(spikeMat, tVec)
% Visualize raster plot
hold all;
for trialCount = 1:size(spikeMat,1)
    spikePos = tVec(spikeMat(trialCount,:));
    for spikeCount = 1:length(spikePos)
        plot([spikePos(spikeCount) spikePos(spikeCount)], [trialCount-0.4
           trialCount+0.4]);
    end
end

ylim([0 size(spikeMat, 1)+1]);
```

This code cycles through each trial, finds all the spike times in that trial, and draws a black line for each spike. Once you have saved this function, use it visualize the spike matrix and label the axes. Here is the code to do so:

```
[spikeMat, tVec] = poissonSpikeGen(30, 1, 20);
plotRaster(spikeMat, tVec*1000);
xlabel('Time (ms)');
ylabel('Trial Number');
```

Here is our raster plot:



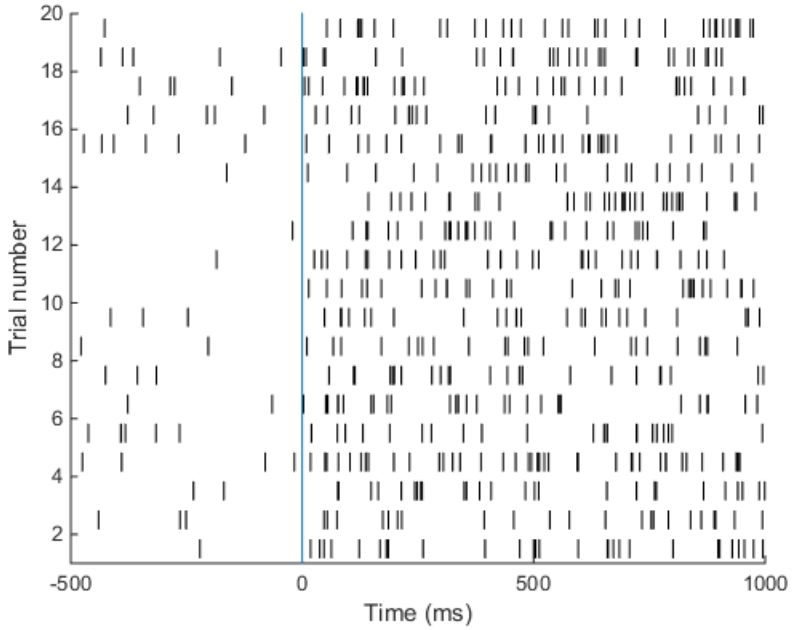
4.6 Quantify the spike matrix in MATLAB

Let's calculate the mean firing rate for each spike train in the raster plot. We can write a *for loop* to repeatedly execute code in MATLAB. To find the mean firing rate, we can calculate the number of action potentials in each trial and divide the time interval (remember to use *s* for finding *Hz*). Here is the code to do so:

```
[numTrials, timeLength] = size(spikeMat);

time_s = timeLength./1000;

numAP = cell(1,1);
for i = 1:numTrials
    numAP{i} = length(find(spikeMat(i,:)==1));
end

numAP = cell2mat(numAP');
mean_firingRate = numAP./time_s;
```

4.7 Plot a realistic spike matrix

Let us apply the functions we wrote to plot a realistic spike matrix. Imagine a neuron in in the cat visual cortex has a *baseline firing rate* of 6 Hz. Upon presentation of a vertical line, the firing rate of this neuron increases to 30 Hz. Make a raster plot representing the firing activity of this neuron. The baseline period is 500 ms and the visual stimulus was presented for 1 second. The plot should look like this:



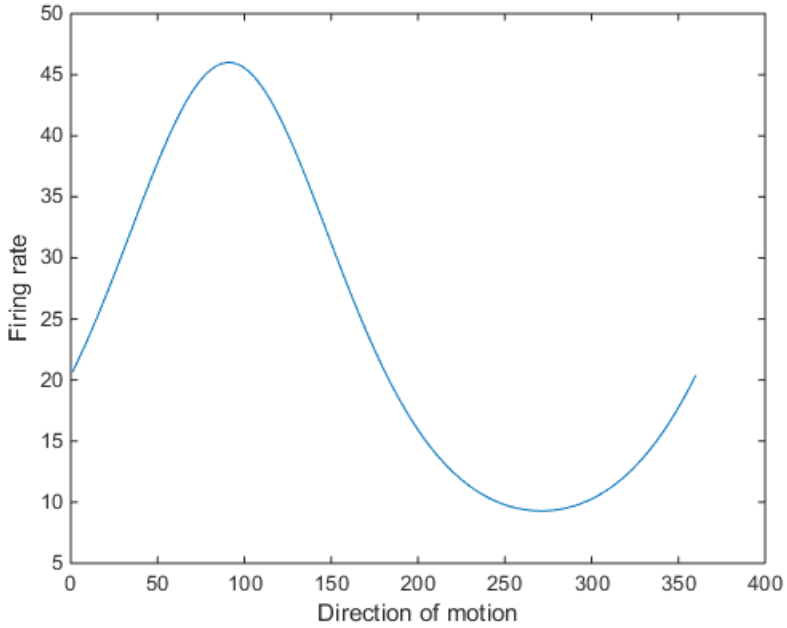
Here is the code to make this plot:

```
% simulate the baseline period
[spikeMat_base, tVec_base] = poissonSpikeGen(6, 0.5, 20);
tVec_base = (tVec_base - tVec_base(end))*1000 - 1;

% simulate the stimulus period
[spikeMat_stim, tVec_stim] = poissonSpikeGen(30, 1, 20);
tVec_stim = tVec_stim*1000;

% put the baseline and stimulus periods together
spikeMat = [spikeMat_base spikeMat_stim];
tVec = [tVec_base tVec_stim];

% plot the raster and mark stimulus onset
plotRaster(spikeMat, tVec);
hold all;
plot([0 0], [0 size(spikeMat, 1)+1]);

% label the axes
xlabel('Time (ms)');
ylabel('Trial number');
```

Simulating spike trains like the ones in the raster plot above requires only one piece of information: *the firing rate of the neuron.*

4.8 Summary

Neural encoding transforms information from the *physical space* to the *neural space.* The physical space consists of physical properties of objects (i.e. direction, shape, speed, color, and loudness). The neural space consists of properties of neurons (firing rate, single cell or population, and dendritic spine density). To understand the encoding of a neuron, we use a *tuning curve.*



A tuning curve is a simple graphical representation of neural encoding. On the *x-axis,* we represent the *physical space.* On the *y-axis,* we represent the *neural space.* Neuroscientists determine the tuning curve of a neuron by presenting the animal with various stimuli (varying along the *x-axis*) while recording the activity of that neuron.

Tuning curves translate physical quantities (direction of motion) into firing rate. The motivation for simulating spike trains is to understand the significance of the firing rate in neural encoding.

5 Signal Processing

5.1 Introduction

Signal processing is used to enhance signal components in noisy measurements. It is especially important in analyzing time-series data in neuroscience. Applications of signal processing include data compression and predictive algorithms.

Data analysis techniques are often subdivided into operations in the ***spatial domain*** and ***frequency domain.*** For one-dimensional time series data, we begin by ***signal averaging*** in the spatial domain. Signal averaging is a technique that allows us to uncover small amplitude signals in the noisy data. It makes the following assumptions:

1. Signal and noise are uncorrelated.
2. The timing of the signal is known.
3. A consistent signal component exists when performing repeated measurements.
4. The noise is truly random with zero mean.

In reality, all these assumptions may be violated to some degree. This technique is still useful and robust in extracting signals.

5.2 Simulation of Signal Averaging

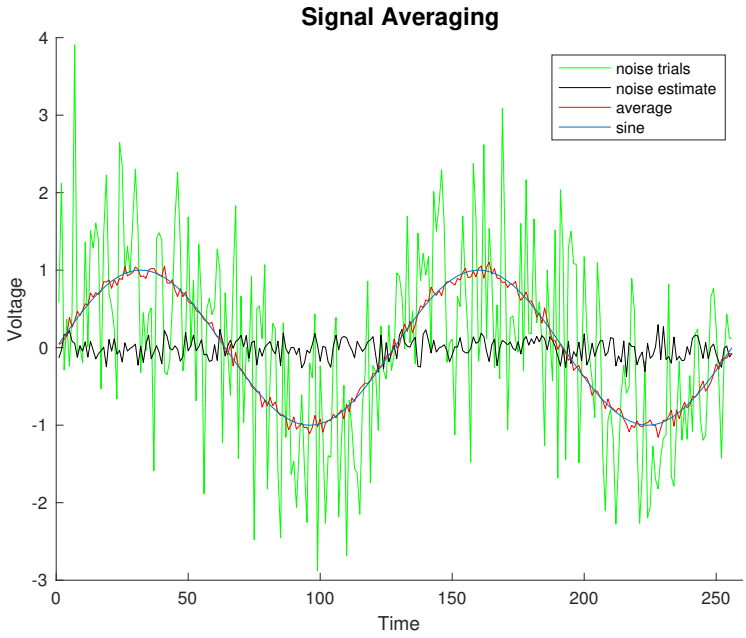
To simulate signal averaging, we will generate a measurement x that consists of a signal s and a noise component n. This is repeated over N trials. For each digitized trial, the kth sample point in the jth trial can be written as

$x_j(k) = s_j(k) + n_j(k)$

Here is the code to simulate signal averaging:

```
% Signal Averaging Simulation

% Generate 256 noisy trials
trials = 256;
noise_trials = randn(256);

% Generate sine signal
sz = 1:trials;
sz = sz/(trials/2);
S = sin(2*pi*sz);

% Add noise to 256 sine signals
for i = 1:trials
    noise_trials(i,:) = noise_trials(i,:) + S;
end

% Average of 256 noisy signals
avg = sum(noise_trials)/trials;
odd_avg = sum(noise_trials(1:2:end,:)/(trials/2));
even_avg = sum(noise_trials(2:2:end,:)/(trials/2));
noise_estimate = odd_avg-even_avg;

% Create Figure
figure;
hold;
plot(noise_trials(1,:),'g');
plot(noise_estimate, 'k');
plot(avg,'r');
plot(S);
legend('noise trials','noise estimate','average','sine');
title('Signal Averaging', 'FontSize', 14);
xlabel('Time');
ylabel('Voltage');
```

```
xlim([0 260]);
hold off;
```



This figure shows how the averaging process results in an estimate of the signal. As compared to the original signal in green, the averaged noise component is reduced in a signal average of 256 trials in red. When averaging real signals, the underlying small-amplitude component may not be as clear. For this reason, the noise estimate is repeated using different parts of the data. One common way to do this is to average all ***odd*** and all ***even*** trials in separate buffers. The average of these two values is the ***noise average***, while the difference of the two is the ***noise estimate***.

5.3 Noise

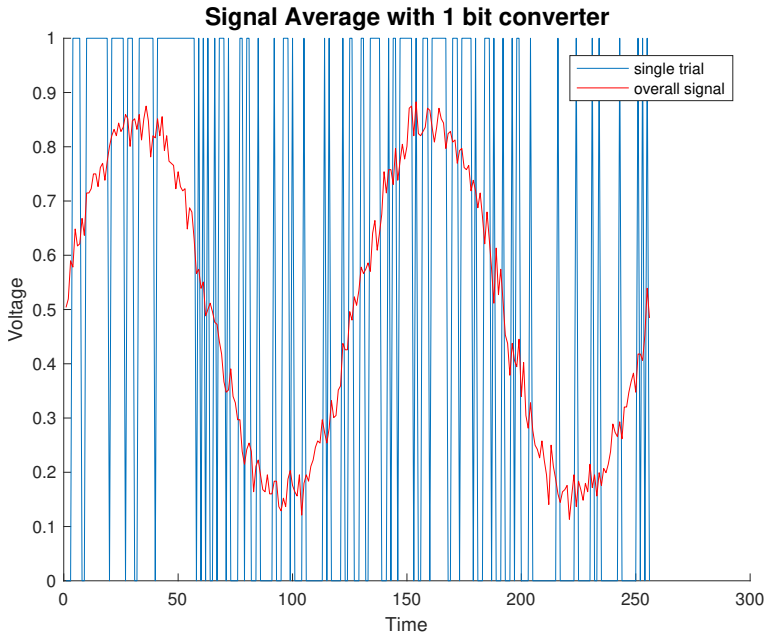
To produce a quality signal measurement, it seems intuitive that a high-precision analog-to-digital converter (ADC) would be needed. However, ADC precision is not critical to signal averaging and noise can be helpful when measuring signals through averaging. In other words, the averaging process, which aims to reduce noise, may work better if some noise is present in signals.

Assume you have a 1-bit ADC. There are only two values for the signal: 0 or 1. In this case, a small deterministic signal without added noise cannot be averaged or measured because it would result in the same uninformative series of 0s and 1s in each trial. If we now add noise to the signal, the probability of finding a 0 or 1 sample is proportional to the signal's amplitude at the time of each sample. By averaging the results of multiple trials, we can obtain a probabilistic representation of the signal.

Let us use the data set of 256 trials in the previous example to simulate signal averaging with a 1-bit ADC. Here is the code as follows:

```
% Analog to Digital Conversion

for k = 1:trials
    for m = 1:trials
        if (noise_trials(k,m) < 0 )
            noise_trials(k,m) = 0;
        else
            noise_trials(k,m)=1;
        end
    end
end

avg2 = sum(noise_trials)/trials;
figure;
hold;
```

```
plot(noise_trials(1,:));
plot(avg2, 'r');
legend('single trial', 'overall signal');
title('Signal Average with 1 bit converter', 'FontSize', 14);
xlabel('Time');
ylabel('Voltage');
hold off;
```



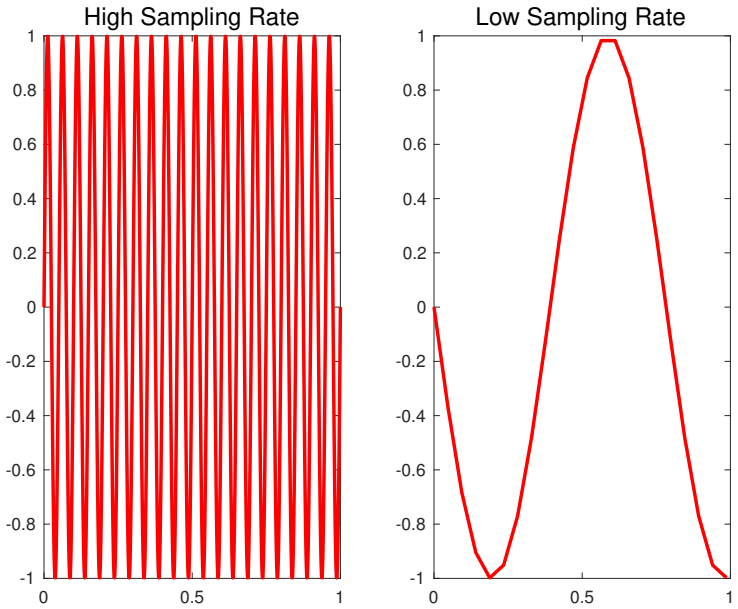
The averaged result using a 1 bit converter is surprisingly similar to the signal average in the previous example. This shows us how reasonable averaging results can be obtained with a low-resolution ADC using the statistical properties of the noise component. This suggests that the ADC resolution may not be a critical component in the signal averaging technique.

5.4 Sampling Rate and Nyquist Frequency

Sampling rate is the number of samples per second taken from a continuous signal to make a discrete or digital signal. For time-domain signals, frequencies are measured in hertz (Hz) or cycles per second. The ***Nyquist-Shannon sampling theorem*** states that perfect reconstruction of a signal is possible when the sampling frequency is greater than twice the maximum frequency of the signal being sampled. For example, if an audio signal has an upper limit of 20,000 Hz (the approximate upper limit of human hearing), a sampling frequency greater than 40,000 Hz will avoid ***aliasing*** and enable signal reconstruction. ***Aliasing*** is a sampling effect that leads to spatial frequencies being falsely interpreted as other spatial frequencies.

Here is a simulation of waveforms at different sampling rates:

```
% Define parameters

t = 0:0.001:1;
f = 20;
signal = sin(2*pi*f*t);

% Simulate different sample rates and plots
figure;
for i = 2:5:50
    subplot(1,2,1); plot(t,signal,'r', 'Linewidth', 2); title('High Sampling
        Rate', 'FontWeight', 'normal', 'FontSize', 14);
    subplot(1,2,2); plot(t(1:i:1000),signal(1:i:1000), 'r', 'Linewidth', 2);
        title('Low Sampling Rate', 'FontWeight', 'normal', 'FontSize', 14);
end
```



5.5 Signal Filtering

A ***filter*** is any process to remove frequencies from a signal. In neurophysiological analysis, we use a linear, continuous-time filter. Data is filtered as it is collected by analog and digital systems. The frequency bands we remove depend on the hardware that is used and on settings determined in the recording software.

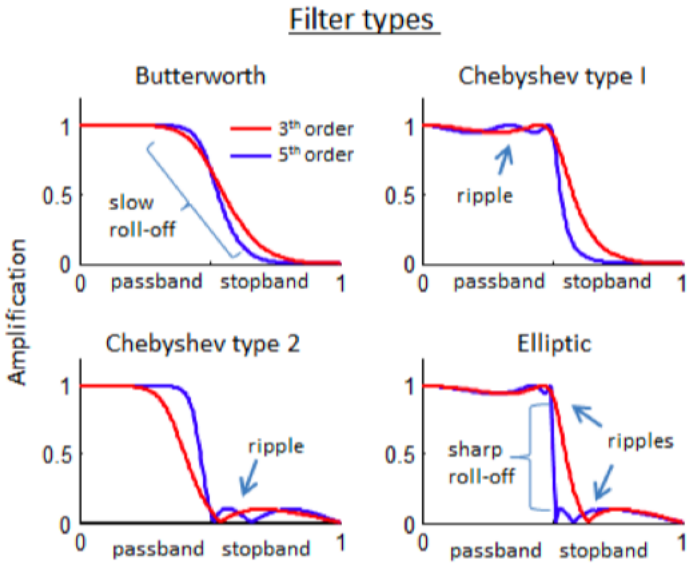
The type of filter we use depends on what we are interested in. If we are interested in higher frequencies, we use a ***high-pass filter***; for lower frequencies, ***a low-pass filter***. ***Band-stop filters*** remove a range of frequencies. A special case of this, the ***notch filter,*** removes only a single frequency. Notch filters are often used to remove ***60 Hz***, which is the frequency of the alternating current coming from the electrical outlets and can contaminate our recordings.

One method to filter a signal is to perform a Fourier transformation, remove coefficients within the stop-band, and then reconstruct the signal with the remaining coefficients. In the strictest sense, this is called a ***truncation filter.*** A common filtering method is to apply a differential equation:

$y(n) = (b_0x(n) + b_1x(n-1) + ... + b_Nx(n-N)) - (a_1y(n-1) + a_2y(n-2) + ... + a_My(n-M))$

Filters are often divided into two classes: ***finite impulse response (FIR)*** and ***infinite impulse response (IIR)***. FIR filters generate a filtered signal using only a fixed interval (using only the ***b coefficients*** in the equation). In contrast, IIR filters use ongoing state changes as the filter slides across samples of the input signal (using only the ***a coefficients*** in the equation).

Bandpass filters use a range of frequencies to filter. Four major types of bandpass filters are ***Butterworth, Chebyshev type I, Chebyshev type II, and Elliptic filters.*** These techniques vary in how they balance the ***"roll-off"*** with the ***"ripple."*** The "roll-off" is the frequencies you do not want (also called the ***stop band***). The "ripple" describes how similar the range of frequencies in the passband are (how "flat" the peaks are).

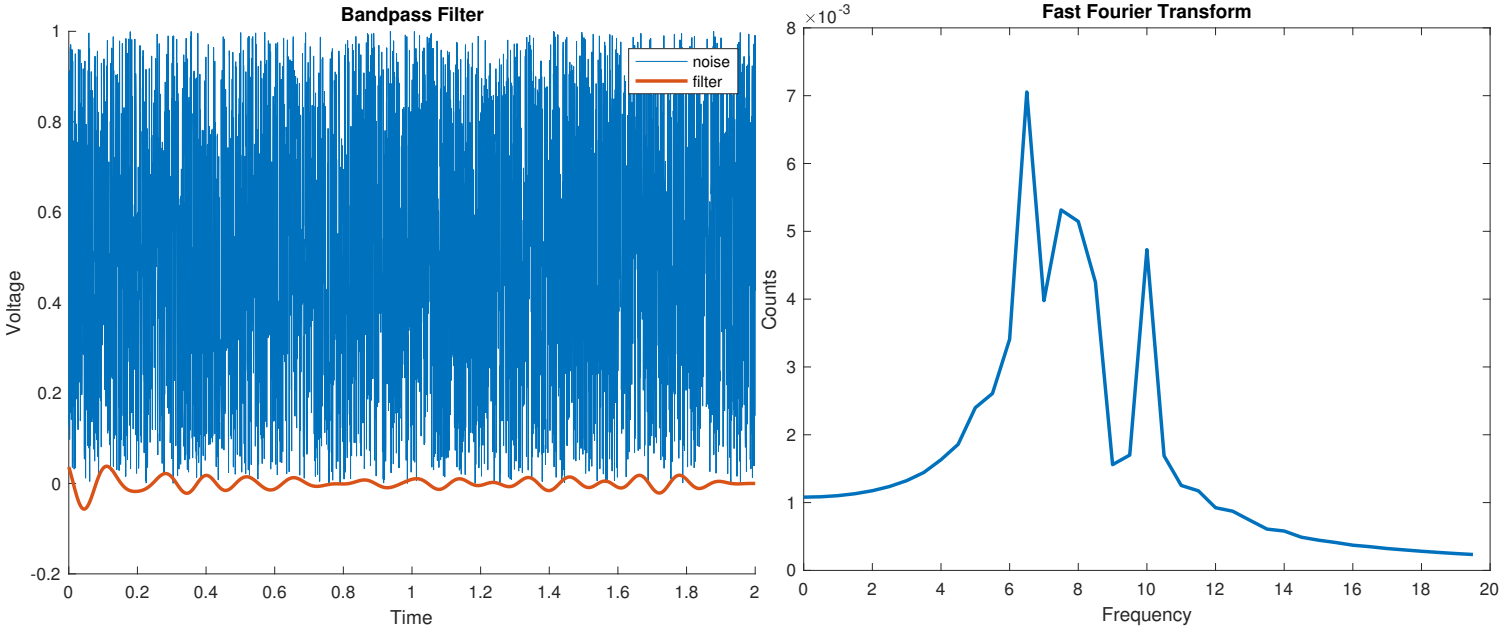


=

An important parameter when setting up a filter is the ***filter order.*** The order refers to the number of coefficients used in determining the filter function. It will determine the balance between time versus. frequency resolution.

In the following simulation, we filter a random signal to generate a new signal that oscillates between 6 to 10 Hz. To confirm that the filter worked, we decompose the signal with a Fourier transform. Here is the code:

```
% Generate Random Noise

noiselfp = rand(1, 4001);
figure;
hold;
plot(linspace(0,2,4001),noiselfp);

% Filter Noise at 6-10 Hz

% Nyquist freq is half the sampling rate and represents
% the highest frequency the data can accurately represent
Nyquist_freq = 2000/2;
low_freq = 6/Nyquist_freq; %lower freq bound
high_freq = 10/Nyquist_freq; %higher freq bound

filter_order = 3; % number of filter exponents
passband = [low_freq high_freq];
% Butterworth filter coefficients
[Bc Ac] = butter(filter_order, passband);
LFP_filt = filtfilt(Bc,Ac, noiselfp);
plot(linspace(0,2,4001), LFP_filt, 'Linewidth',2);


% Label figure
legend('noise','filter');
title('Bandpass Filter');
xlabel('Time'); ylabel('Voltage');
hold off;

% Decompose filtered signal with a Fourier transform
y = fft(LFP_filt)/4001; %divide by num of samples
f = 2000/2 * linspace(0,1,4001/2 + 1);
figure; plot(f(1:40), abs(y(1:40)),'Linewidth',2);
title('Fast Fourier Transform');
xlabel('Frequency'); ylabel('Counts');
```



In this example, the amplitude of each frequency between 6 to 10 Hz is quite variable. This is a result of the short time period of our sample and the chance outcome of our random number generator. The non-zero values before 6 Hz and after 10 Hz demonstrate one tradeoff in using the Butterworth filter: there will be a relatively smooth roll-off in the stopband, with the advantage that we don't introduce ripples.

5.6 Units in Fourier Transform

1. **Record raw signal:** units are ***voltage,*** values are ***real.*** We begin with a signal of time-varying voltage $v(t)$.
2. **Apply Fast Fourier Transform (FFT):** units are ***voltage,*** values are ***complex.*** A signal is always real. However, for a compact representation of the sum of sinusoids, the FFT uses complex exponentials. Since we are representing a real number by a complex-valued entity, we need to add both the complex number and its complex conjugate. For example, if we have a real signal represented as $cos(\theta) + j \times sin(\theta)$, then we also need to add its complex conjugate pair. The complex conjugate is $cos(\theta) - j \times sin(\theta) = exp(-j \times \theta)$. Since $\theta = 2 \times \pi \times f$, we have a ***negative frequency*** when we plot the FFT amplitude (counts) vs. frequency plot. *In summary, negative frequencies are a mathematical side effect of using complex exponentials as our representation of Fourier transformations.*
3. **Take the absolute value:** units are ***voltage,*** values are ***real.*** Magnitude of signal components .
4. **Square the result:** units are $voltage^2$***,*** values are ***real.*** Square of magnitude of signal components. This result can be used to create a plot of ***power spectral density.***

6 Seizure Detection

6.1 Time-Frequency Analysis

How does a signal change over time? This question is often answered by using one of the following three methods:

- Apply a Fourier transform with a sliding window
- Use a wavelet transform
- Filter the signal and apply a Hilbert transform

A ***sliding window*** is a segment of data *("window")* within which the computation (often a Fourier transform) is applied. The window slides through the data, repeating the computation until the entire signal is covered. The built-in **spectrogram** function in MATLAB performs a sliding window, and allows the user to vary window lengths and window overlaps.

A ***wavelet transform*** is an alternative to the Fourier transform. This algorithm computes the similarity between each segment of a signal and a short, wave-like distribution called a ***wavelet.*** The wavelet can be scaled across many widths to capture different frequencies.

The ***Hilbert transform*** is often applied to pre-filtered signals. It can be thought of as a *convolution* (using a distribution to transform the signal) that produces a complex number at each point. These complex numbers can be re-interpreted in terms of phases and amplitudes. Thus, the Hilbert transform is an *instantaneous* Fourier transform.

6.2 Uncertainty Principle

There are limits to how precisely the frequency spectrum of a signal can be computed. In signal processing, the limiting factor is the length of the signal. Dennis Gabor, inventor of the hologram, was the first to realize that the uncertainty principle applies to signals. The exact time and frequency of a signal can never be known simultaneously: a signal cannot plot as a point on the time-frequency plane. *This uncertainty is a property of signals, not a limitation of mathematics.*

Heisenberg's uncertainty principle is often written in terms of the standard deviation of position σ_{x}, the standard deviation of momentum σ_{p}, and the Planck constant h:

$$\sigma_x \sigma_p \geq \frac{h}{4\pi} \approx 5.3 \times 10^{-35} m^2 kg.s^{-1}$$

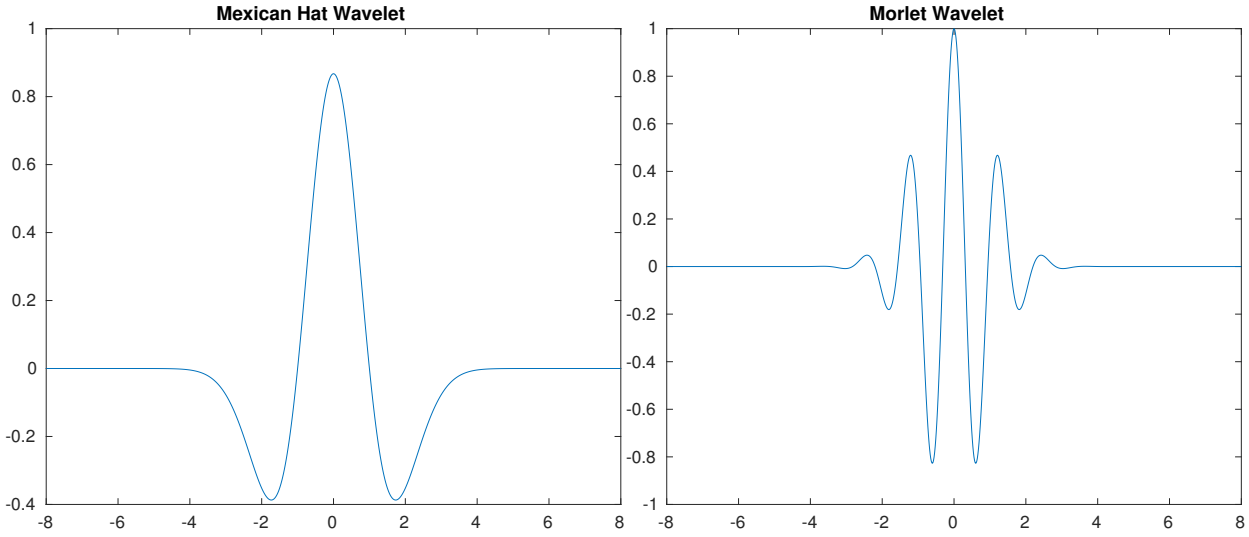
In other words, the product of the uncertainties of position and momentum is small, but not zero. If the standard deviations of the time and frequency estimates are $\sigma_{\mathrm{t}}$and σ_{f}respectively, then we can write ***Gabor's uncertainty principle*** as:

$$\sigma_t \sigma_f \geq \frac{1}{4\pi} \approx 0.8 cycles$$

Thus, the product of the standard deviations of time (ms) and frequency (Hz) must be at least 80 ms-Hz. Regardless of how the transform is computed, we pay for time information with frequency information. Specifically, the product of time uncertainty and frequency uncertainty must be at least $\frac{1}{4\pi}$.

6.3 Types of Wavelets

Time-varying frequency components can be identified by filtering a signal with short distibrutions called ***wavelets.*** Wavelets are brief oscillations. When a filter is applied, these wavelets expand or contract to fit the frequencies of interest.



In neurophysiological signal analysis, the Mexican Hat and Morlet wavelets are commonly used. The Morlet wavelet contains a greater number of cycles, which improves the detection of signal oscillations relative to transient events. It also means that we lost time information in order to increase frequency resolution.

One common error in time-frequency analysis is the misattribution of signal events as oscillations. Neural signals appear as *spikes*. Sharp transitions in the signal are built not of a single frequency component, but a vast range of frequencies. Any filter that is applied to the signal over a spike will give the appearance of a power increase at that moment, and the strength of the power inrease is related to the size of the spike. Thus, noise can lead to mistinterpretation of time-frequency results.

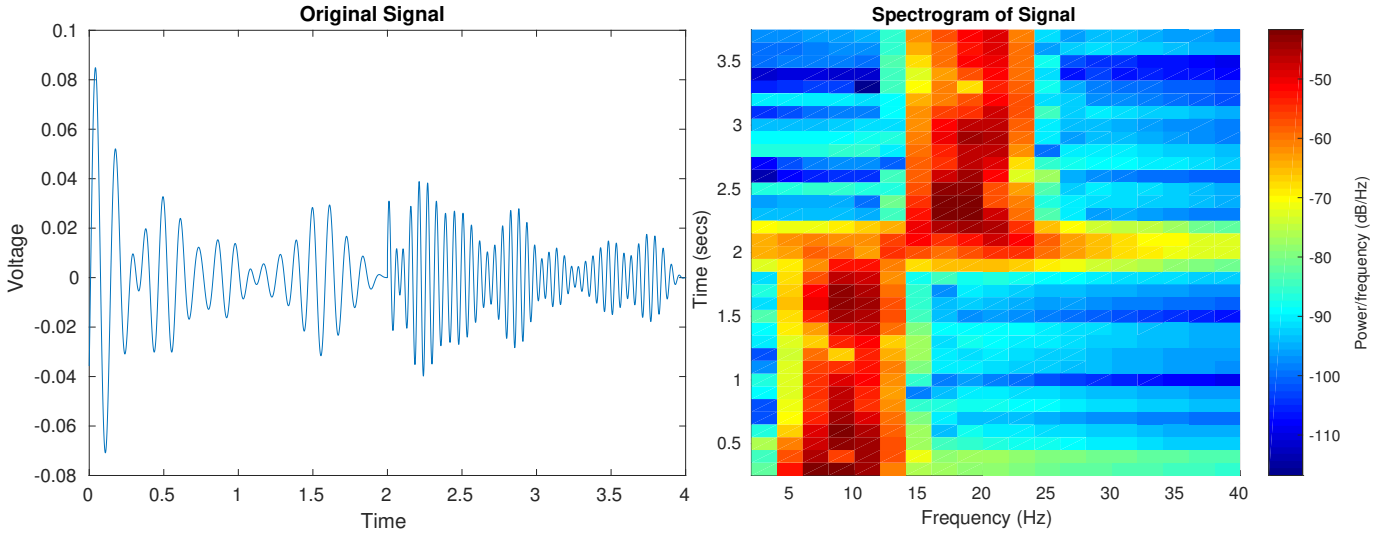
Here is code to pre-filter a signal to remove noise:

```
function fLFP = filt_LFP(sig1, lower_limit, upper_limit, sF)

% filt_LFP uses a butterworth filter to bandpass filter the signal between
% lower and upper limit

% INPUTS:
% sig1 = signal to be filtered
% lower_limit = lower bound of bandpass
% upper_limit = upper bound of bandpass
% sF = sampling frequency (default: 2000 Hz)

% Set default sF = 2000 Hz
if nargin < 4
    sF = 2000;
end
if isempty(sF)
    sF = 2000;
end

Nyquist_freq = sF/2;
lowcut = lower_limit/Nyquist_freq;
highcut = upper_limit/Nyquist_freq;
filter_order = 3;
passband = [lowcut highcut];
[Bc, Ac] = butter(filter_order,passband);
fLFP = filtfilt(Bc, Ac, sig1);
```

6.4 Wavelet Demo

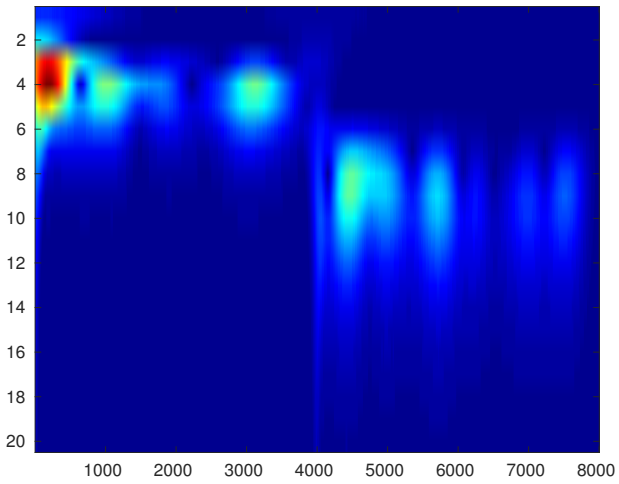
To begin this demo, let's simulate a signal with an 8 to 10 Hz segment followed by a 16 to 20 Hz segment and then apply a sliding window Fourier analysis:

```
% Wavelet Analysis

% Generate a signal with an 6-10 Hz segment
rng(11);
noiselfp = rand(1,4001);
LFP_6to10 = filt_LFP(noiselfp, 6, 10);

% Generate a signal with a 16-20 Hz segment
rng(12);
noiselfp = rand(1,4001);
LFP_16to20 = filt_LFP(noiselfp, 16, 20);

% Concatenate signals
LFP = [LFP_6to10 LFP_16to20];
figure;
plot(linspace(0,4,8002),LFP);
title('Original Signal');
xlabel('Time'); ylabel('Voltage');

% Apply sliding window Fourier Analysis
window = 1000;
noverlap = 800;
F = 2:2:40;
Fs = 2000;
spectrogram(LFP, window, noverlap, F, Fs);
view(2);
colormap jet;
title('Spectrogram of Signal');

% Morlet wavelet
coefsi = cwt(LFP, centfrq('cmor1-1')*Fs./F,'cmor1-1');
f2 = figure;
wm_image = imagesc(abs(coefsi));
colormap jet;

% Mexical hat wavelet
coefsi_hat = cwt(LFP, centfrq('mexh')*Fs./[2:40],'mexh');
imagesc(abs(coefsi_hat));
```

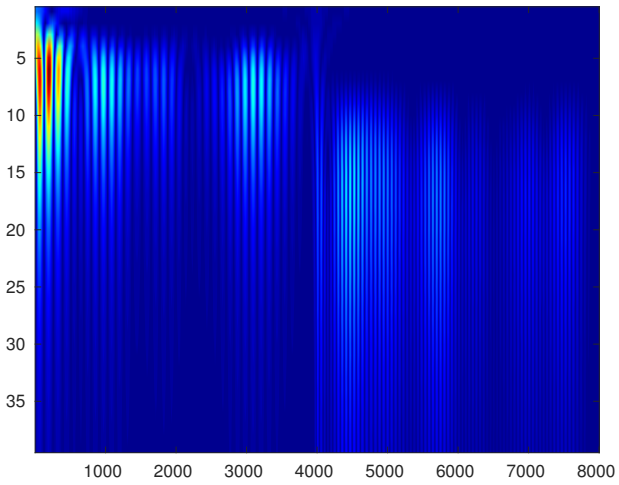


There are two approaches to applying wavelet transformations in MATLAB: discrete and continuous. These differ according to how the set of frequencies (wavelet width) are chosen.

In this exercise, we compute the complex Morlet wavelet. We can obtain meaningful values of amplitude and phase in this way.



We repeat this process for the Mexican hat wavelet. The built-in MATLAB function does not use complex conjugates, so we will only detect upward deflections in the LFP signal at each frequency band.

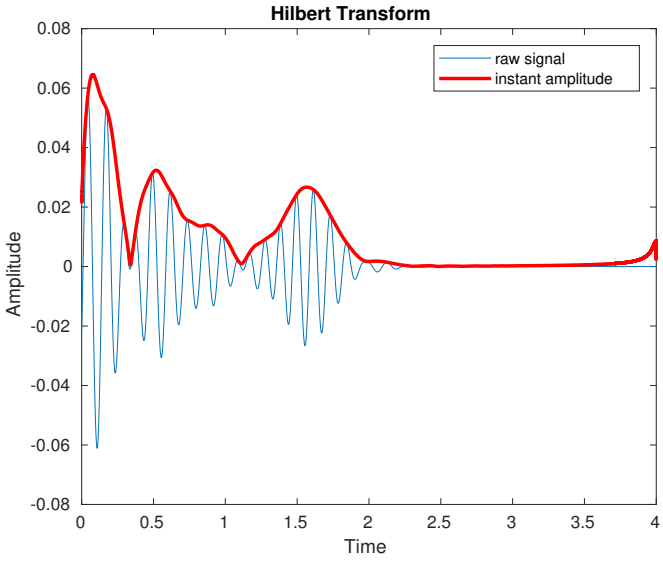


In this image, we see stripes where we previously saw smooth transitions across time bins. The warmer-colored streaks also extend farther across the frequency spectrum, while the resolution appears to be stronger in the temporal domain. This illustrates the *uncertainty principle* described above.

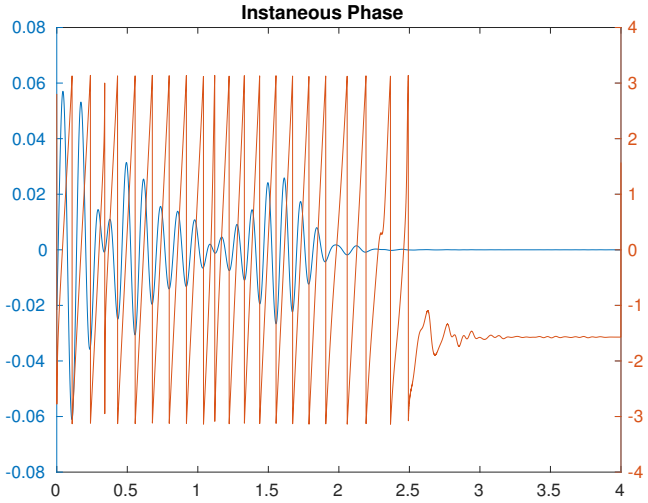
6.5 Hilbert Transform

The Hilbert transform is used to compute the instantaneous phase and amplitude of a signal. It is best to use pre-filtered data. Here is the code to do so:

```
% Hilbert transform

LFP_6to10post = filt_LFP(LFP,6,10,2000);
h_LFP_6to10post = hilbert(LFP_6to10post);
amp_LFP_6to10post = abs(h_LFP_6to10post);

figure;
hold on;
plot(linspace(0,4,8002),LFP_6to10post);
plot(linspace(0,4,8002), amp_LFP_6to10post, 'r','LineWidth',2);
legend('raw signal','instant amplitude');
xlabel('Time'); ylabel('A
```



The red line indicates our instanteous amplitude, which *envelopes* the filtered signal. We could alternatively use the complex conjugate to identify the instantaneous phase of the oscillation in the filtered signal:



6.6 Seizure Simulation

Epilepsy is a chronic neurological disorder characterized by recurrent seizures. In children, many seizures display rhythmin ***spike-wave (SW) discharges*** in the EEG. In a study published by *PLoS One* (*Taylor and Wang, 2014*), a computational model of seizures was built on MATLAB. In this demo, we will explore their simulation:

```
function dudt=AmariTCimpBS(t,u)
%
    %connectivity parameters
    w1 =1.8;%PY  -> PY
    w2 = 4;    %PY -> I
    w3 = 1.5; % I -| PY
    % w4= 0;
    w5 = 10.5; %  TC -> RTN
    w6 = 0.6; % RTN -| TC
    w7 = 3; % PY -> TC
    w8 = 3; %   PY -> RTN
    w9 = 1; %  TC -> PY
    w10= 0.2;% RTN -| RTN

  h_p = -.35; %
  h_i = -3.4; %
```

```
    h_t = -2; %    -2 for bistable for ode45; -2.2 for excitable for ode45.
    h_r = -5; %
    s=2.8;
     a=1;
    % Time scale parameters
    tau1=1*26;    %
    tau2=1.25*26;    %
    tau3=.1*a*26;    %
    tau4=.1*a*26;    %



    sig_py = (1./(1+250000.^-(u(1))));
    sig_in = (1./(1+250000.^-(u(2))));
    sig_tc = (1./(1+250000.^-(u(3))));
    sig_re = (1./(1+250000.^-(u(4))));

    dudt=zeros(4,1);

    dudt(1) = (+h_p -u(1) +w1*sig_py -w3*sig_in + w9*sig_tc )*tau1;
    dudt(2) = (+h_i -u(2) +w2*sig_py                          )*tau2;
    dudt(3) = (+h_t -u(3)  +w7.*sig_py - w6*(s*u(4)+.5)       )*tau3;
    dudt(4)= (+h_r -u(4) +w8.*sig_py + w5*(s*u(3)+.5) -w10*(s*u(4)+.5))*tau4;



end
```

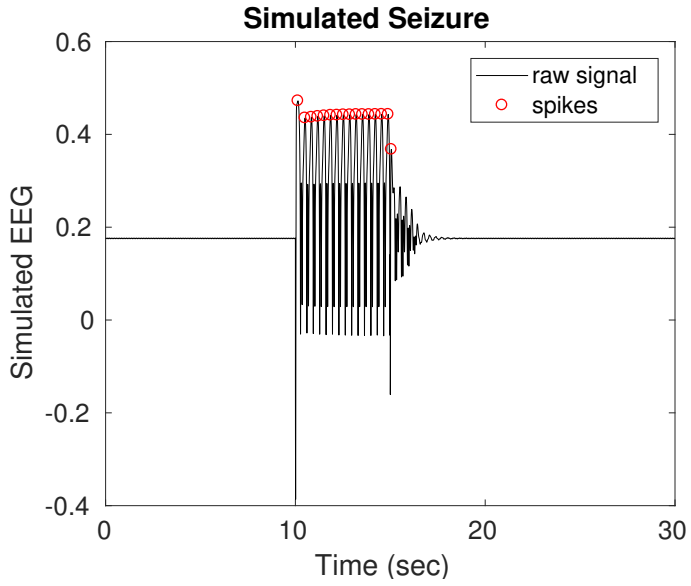
The model describes the temporal evolution of the state of four variables corresponding to the activity of populations of:

- cortical pyramidal neurons (PY)
- cortical inhibitory interneurons (IN)
- thalamo-cortical neurons (TC)
- inhibitory (thalamic) reticular neurons (RE)

There is a background state of normal activity and a rhythmic state of pathalogical activity (SW complex). We can then modify their code to detect peaks by thresholding:

```
%initial condition near the fixed point
fp_approx = [0.1724    0.1787   -0.0818    0.2775];


[t1,u]=ode45(@AmariTCimpBS,[0 10],fp_approx);%background state
[t2,v]=ode45(@AmariTCimpBS,[10 15],u(end,:)-[.3 .3 0 0]);%seizure state
[t3,w]=ode45(@AmariTCimpBS,[15 30],v(end,:)-[.3 .3 0 0]);%background state
%%


PY=[u(:,1); v(2:end,1); w(2:end,1)];
IN=[u(:,2); v(2:end,2); w(2:end,2)];

TC=[u(:,3); v(2:end,3); w(2:end,3)];
RE=[u(:,4); v(2:end,4); w(2:end,4)];

t=[t1;t2(2:end);t3(2:end)];

figure(1)
```

```
plot(t,mean([PY,IN],2),'k')
hold on
m=mean(fp_approx(1:2));
% plot([10 10],[m m-.3],'r','LineWidth',5)
% plot([15 15],[m m-.3],'b','LineWidth',5)
hold off
xlabel('Time (sec)','FontSize',20)
ylabel('Simulated EEG','FontSize',20)
set(gca,'FontSize',15)
%legend('Simulated EEG','Stimulus pulse to induce SWD','Stimulus pulse to terminate
    SWD')

% Use built-in peak detection
time = t;
volt = mean([PY,IN],2);
pks = findpeaks(volt,0.3); % threshold for peaks > 0.3
pks = pks.loc; % convert struct to double
hold on;
scatter(t(pks),volt(pks),'r');
legend('raw signal','spikes');
title('Simulated Seizure');
hold off;
```



References

1. Hinton, G. E. (2011). Machine learning for neuroscience. Neural systems & circuits, 1(1), 12.
2. Helmstaedter, Moritz. "The mutual inspirations of machine learning and neuroscience." Neuron 86, no. 1 (2015): 25-28.
3. Hassabis, D., Kumaran, D., Summerfield, C., & Botvinick, M. (2017). Neuroscience-inspired artificial intelligence. Neuron, 95(2), 245-258.
4. Crick and Mitchison: The function of dream sleep. Nature 1983, 304: 111-114
5. Neural Coding Tutorial: https://praneethnamburi.wordpress.com
6. Signal Processing Tutorial: http://ninsel.net/SignalAnalysisPrimerfortheMATLABNaive_all4chapters_v1p2.pdf

Problem Set 1

Test Your Understanding

Imagine you want to build a personal Twitter feed. You make an array with 100 elements to store the tweets. If you are going to tweet for the 101st time, what should you do so that all of your tweets are stored? (You can assume the array data structure is defined in a language such that all options given in the choices are possible.)

- Resize the array so that it has more than 100 elements
- Loop around, storing the new tweets at the beginning of the array
- Use a pre-made buffer array to store the new tweets

Imagine we're running a restaurant and we've stored the total price paid (as a float, or decimal number) by each customer on a given night in an array. How can we find the **median** amount spent by customers?

- Add up the elements of the array and divide the number of elements in the array
- Sort the elements of the array in increasing order, and take the middle element
- Add the smallest and largest elements of the array and divide by 2

Which of the following measures of central tendency is easily computed given an unsorted array of integers?

- Mean (average)
- Median
- Interquartile Range

Which of the following data would be more naturally represented by a two-dimensional array that a one-dimensional array?

- A chess board
- A digital photograph
- A teacher's gradebook
- All of the above

Assume you have a sorted array of with 1000 elements. To find a specific element, what is the maximum number of paired comparisons needed?

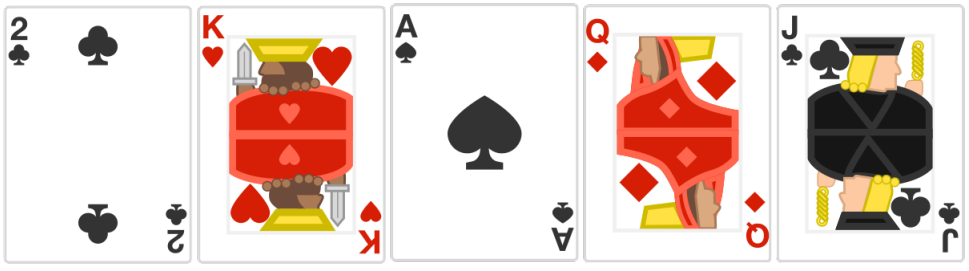
- **Hint:** Pick an element from the list to observe first. If the element we observe is greater than our target, what does this tell us? What if it's lower? With a sorted list, knowing if an element is greater or less than the element we're looking for can be useful.

Suppose you have a sorted array with 100,000,000 elements in it. Assuming the worst case for each method, about how many times more comparisons will linear search make than binary search?

- 4,000
- 40,000
- 400,000
- 4,000,000

Suppose you had the following poker hand dealt to you, and decided to use insertion sort to sort it from smallest to largest. Note that $A > K > Q > J > 2$. How many individual comparisons between cards would need to be made for the sort?

- **Reminder:** First, the K and 2 are compared. Then, the A and K are compared, then the Q and A and so on.



In the worst case for a list with 10 distinct elements, how many comparisons are made? How about for the best case?

Problem Set 2

Problem-Solving

A frog wants to cross a river that is 11 feet across. There are 10 stones in a line leading across the river, separated by 1 foot. He can either jump to the next stone or jump over a stone. He can either jump to the next stone or jump over a stone, and if he jumps over a stone he must land on the next one. The furthest he can jump is 2 feet. Also, he always moves forward (toward the other side of the river). **In how many different ways can he cross the river?**

- **Hint:** One way to cross would be $1, 2, 2, 1, 1, 2, 1, 1$. The total distance jumped must be 11 feet. What are the possible conditions before getting to the n^{th} stone.

Suppose we have 243 identical-looking boxes each containing a drone ready to ship, but we remember that one box is missing instructions! This box will weigh a little less than the rest and there is a scale that can compare weights of any two sets of boxes. Assuming the worst case scenario, what is the minimum number of weighings needed to determine which box has the missing instructions?

Use the **mergesort** algorithm to write the third step of the merge of A and B. Here are the first two steps.

Steps	
Initial State	$A = [2, 4, 9]\, B = [1, 7, 13, 15]\, Results = []$
First Step	$A = [2, 4, 9]\, B = [7, 13, 15]\, Results = [1]$
Second Step	$A = [4, 9]\, B = [7, 13, 15]\, Results = [1, 2]$

How fast is the combine step, i.e. combining two sorted lists into another sorted list containing N elements?

- $O(1)$
- $O(\log N)$
- $O(N)$
- $O\left(N^2\right)$

You would like to sort the following list using the quicksort algorithm described earlier:

$$\{9, 103, 3, 74, 22, 8, 6, 5, 47, 2\}.$$

Let's say we always pick the first element as our "pivot." There will be 5 elements on the left of the pivot after the first pass of partitioning, since there are five values in the list less than the pivot value 9.

At this point, how many elements are guaranteed to be in their "final" sorted position?

Now let's say we are sorting the same list:

$$\{9, 103, 3, 74, 22, 8, 6, 5, 47, 2\},$$

but instead of the first element, we always pick the last element as our pivot. After the first pass of partioning there will be no elements on the left of the pivot, and only the pivot point itself (with the value 2) is guaranteed to be in the right position. (The others may or may not be in their "final" sorted position, but only the pivot value is guaranteed to be.)

Because we placed no elements on the left, we don't recurse on that side. After the next partitioning step, how many elements are guaranteed to be in their "final" sorted position?

Which list will run through quicksort more quickly, assuming we pick either the first or last element as our pivot?

$$A : \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19\}$$

$$B : \{9, 4, 15, 8, 12, 1, 18, 10, 14, 7, 3, 16, 6, 2, 11, 5, 19, 17, 13\}$$

- A
- B
- Same for both

As you might have noticed, an important aspect of quicksort is the selection of the pivot element. Which of the following would provide the optimal pivot selection at each step of the quicksort?

- **A:** The element in the first position
- **B:** The element with the smallest value
- **C:** A randomly selected element
- **D:** The median of the elements

Using the **median of three** method, which value would you select for your pivot point in the following array?

$$[57, 16, 207, 94, 17, 2, 138, 12, 73, 103, 77]$$

Which of the following statements accurately compares quicksort and mergesort?

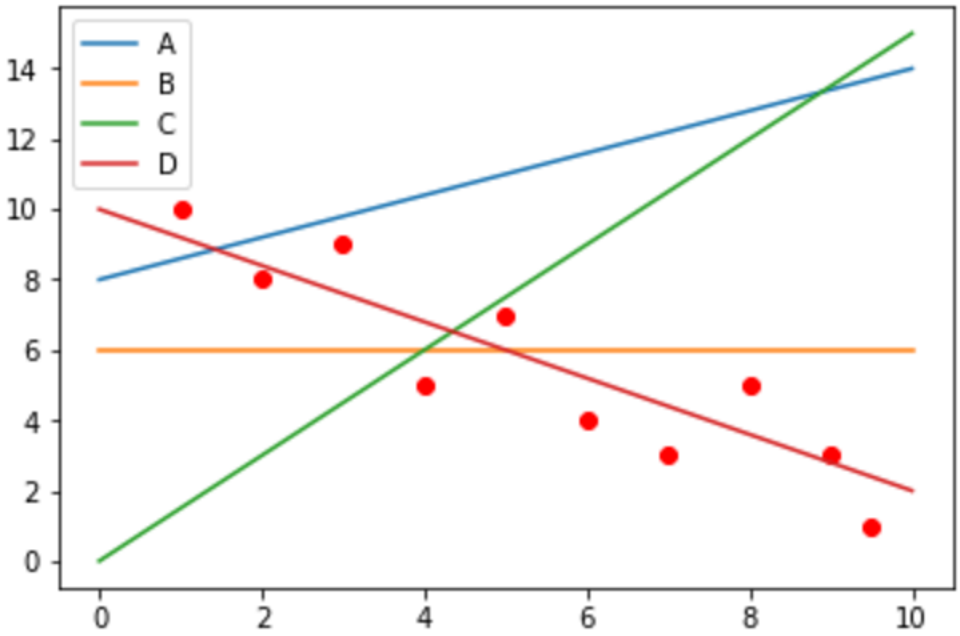
- For large N and randomly distributed data, quicksort is usually faster than mergesort.
- Quicksort runtime ranges from $O\left(N^2\right)$ to $O\left(N \log_2 N\right)$ but mergesort is always $O\left(N \log_2 N\right)$
- The runtime of quicksort is more sensitive to the initial ordering of the array than mergesort is.
- All of these

Problem Set 3

Test Your Understanding

For ten years, Alfred has been planting trees in his exceedingly large backyard. Each year, he records how many seeds he planted in the spring, as well as how many new sprouts there are by fall. With the data points graphed below, which line best represents the relationship between seeds planted and sprouts observed?

- A
- B
- C
- D



Out of the following sets of numbers, which will have the highest standard deviation?

- $[1, 4, 3, 5, 1]$
- $[50, 52, 48, 48, 50]$
- $[90, 90, 90, 90, 90]$
- $[15, 40, 10, 18, 31]$

Say that we are studying the correlation between voltage and a light bulb's brightness. Amazingly, we always know exactly what voltage we are using, but we haven't been so lucky when measuring bulb brightness. Here are our options:

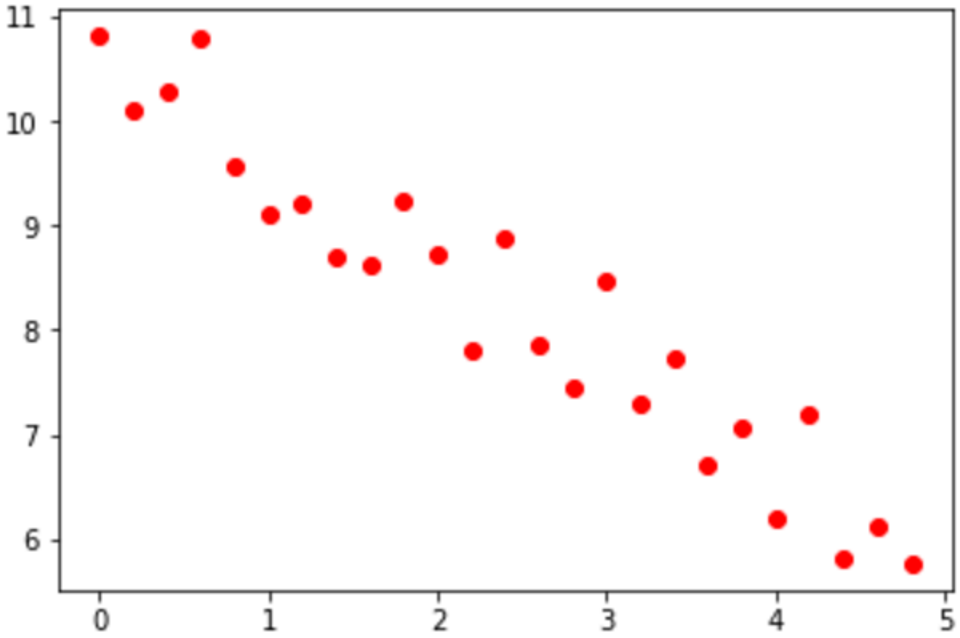
1. A perfectly accurate sensor taped to another light bulb, twice as bright as the one we're measuring. We can't get rid of the second light bulb; we don't know why but the sensor won't work without it.
2. A primitive but effective device from the 1840's, rather like a thermometer. It works, but a human must estimate its readings.
3. A completely broken machine. It always just reads zero.
4. Actual state-of-the-art technology, no quirks attached.

If we always run the light bulb with the same voltage and measure the brightness ten times, which of these devices will collect data with the highest standard deviation, all else being equal?

- Option 1
- Option 2
- Option 3
- Option 4

Which of the following values would be closest to the correlation coefficient of the graph below?

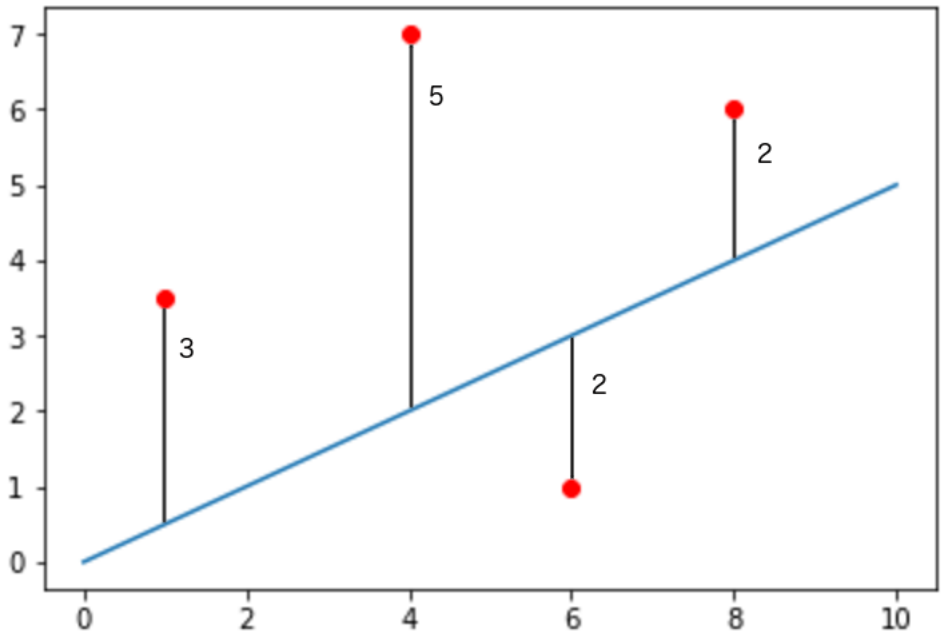
- -1
- 3
- 0.25
- 1



A student has analyzed a data set to find that $\bar{x} = 50, \bar{y} = 30, SD_x = 8, SD_y = 16, r = 0.75$. Which of the following equations gives the best-fit line he needs?

- $y = 1.5x$
- $y - 30 = 1.5\,(x - 50)$
- $y = 0.375x + 48.75$
- $y = 0.66x$

There are several points in the scatter plot shown as well as a best-fit line candidate. We have shown the vertical distance from each point to the line. What is the SSE?



Let's say we have a data set of just three points:

$$(1,0),(3,4),(2,3).$$

Calculate the SSE of line

$$y = 3x + 2$$

Derive an equation to solve for the optimum $\overrightarrow{x}$. Use the fact that $A\overrightarrow{x}$ is perpendicular to $\overrightarrow{b} - A\overrightarrow{x}$ when $\overrightarrow{x}$ is the closest to a solution for $A\overrightarrow{x} = b$. Which of the following choices is correct?

- $A^T\overrightarrow{b} = A^T A\overrightarrow{x}$
- $\overrightarrow{b} = A^T A\overrightarrow{x}$
- $A^T\overrightarrow{b} = A^{-1}\overrightarrow{x}$

If there are $p$predictor variables $\{x_1, x_2, ..., x_p\}$ and one response variable y, then a linear equation which outputs $y$will take the form

$$y = m_1x_1 + m_2x_2 + ... + m_px_p + b.$$

Given this information, what is a reasonable formula for the error when there is more than one predictor variable?

Alfred has data on many types of trees. He has compiled a table of the seeds he planted each spring as well as the number of new sprouts each fall. Using this information, identify the matrix A which he needs to create in the process of calculating a best-fit linear equation.

Oak Seeds	Maple Seeds	New Growths
10	5	9
4	8	7
4	3	5
6	2	4

- $A = \begin{bmatrix} 10 & 5 & 1 \\ 4 & 8 & 1 \\ 4 & 3 & 1 \\ 6 & 2 & 1 \end{bmatrix}$

- $A = \begin{bmatrix} 10 & 1 & 5 & 1 \\ 4 & 1 & 8 & 1 \\ 4 & 1 & 3 & 1 \\ 6 & 1 & 2 & 1 \end{bmatrix}$
- $A = \begin{bmatrix} 10 & 4 & 4 & 6 & 1 \\ 5 & 8 & 3 & 2 & 1 \end{bmatrix}$
- $A = \begin{bmatrix} 10 & 9 & 1 \\ 4 & 7 & 1 \\ 4 & 5 & 1 \\ 6 & 4 & 1 \end{bmatrix}$

Franklin is in the business of building toy race cars and is analyzing the relationship between the weight and top speed of a car when all else is held equal. So far he's managed to collect just five data points, but he's convinced that the relationship should be modeled with a cubic polynomial.

Given the table below, which matrix A must he construct in the process of calculating the best-fit curve?

x	y
5	30
4	26
6	20
3	18
7	15

- Choice 1: $A = \begin{bmatrix} 155 & 1 \\ 84 & 1 \\ 258 & 1 \\ 39 & 1 \\ 399 & 1 \end{bmatrix}$
- Choice 2: $A = \begin{bmatrix} 5 & 5 & 5 & 1 \\ 4 & 4 & 4 & 1 \\ 6 & 6 & 6 & 1 \\ 3 & 3 & 3 & 1 \\ 7 & 7 & 7 & 1 \end{bmatrix}$
- Choice 3: $A = \begin{bmatrix} 125 & 25 & 5 & 1 \\ 64 & 16 & 4 & 1 \\ 216 & 36 & 6 & 1 \\ 27 & 9 & 3 & 1 \\ 343 & 49 & 7 & 1 \end{bmatrix}$

Alfred's done some thinking, and he wants to account for fertilizer in his tree growing efforts. Assume that for every ton of fertilizer he used each seed would be about 1.5 times more likely to grow.

Over the past few years, he has compiled a large data set containing fertilizer used, seeds planted, and tree growth. Is this relationship a good candidate for linear regression analysis?

- No
- Yes

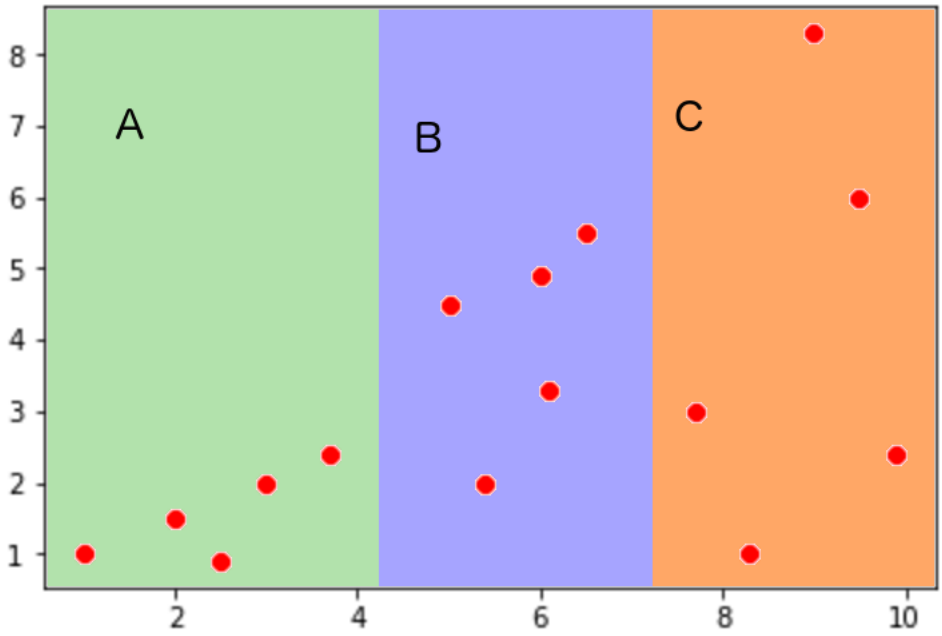
We can see the effects of multicollinearity clearly when we take this problem to its extreme. Say that we have two predictor variables, x_1 and x_2, and one response variable y. Using the test data given in the table below, determine which candidate best-fit equation has the lowest SSE:

x_1	x_2	y
5	10	3
2	4	1
7	14	6
2.5	5	2

- $y = 2x_1 + x_2$

- $y = x_1 + 1.5x_2$
- The sums of squared errors are equivalent

A data set is displayed on the scatterplot below. Which section of the graph will have the greatest weight in linear regression?



- A
- B
- C

A group of scientists wants to analyze bacterial growth in Petri dishes. They have a done of dozen tests, and each time they have recorded every single detail of the environment. The pH levels of the dishes, sugar content of the food, and even the light levels in the room have been recorded. A total of fourteen variables have been taken into account.

In a classic example of overzealous testing, a rogue scientist has added another variable to the mix, his average mood on a scale from zero to ten. When a best-fit equation is generated with this variable included, how will the SSE most likely change? How will the average error on new data change?

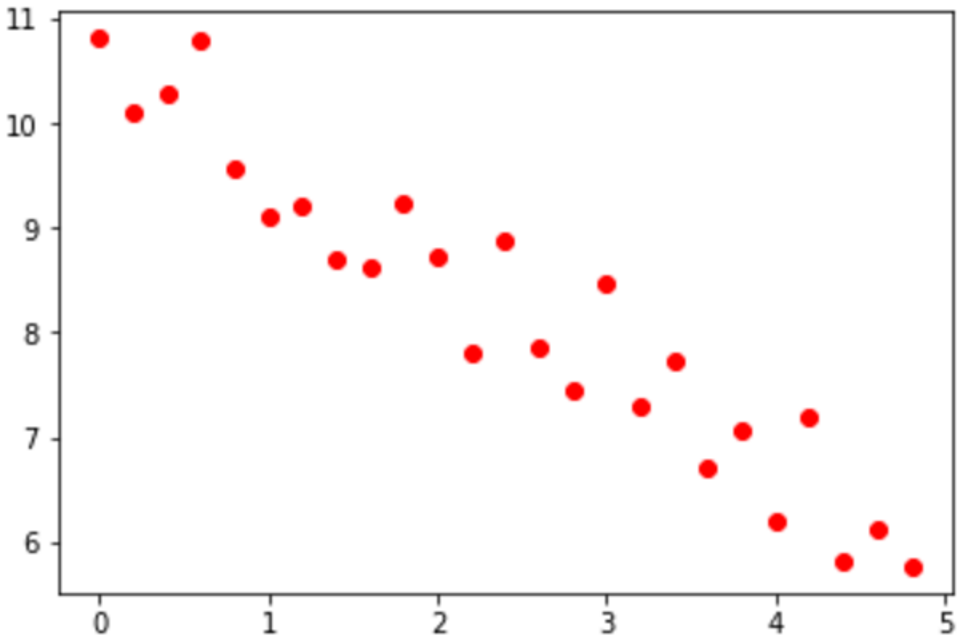
- The SSE will decrease while the error on new data increases
- Everything will remain the same
- The SSE will increase while the error on new data increases
- The SSE and the error on new data will increase

Below, we have three data sets—A, B, C— represented by either tables or scatter plots. We want to reorder them and analyze the first one with K-nearest neighbor regression, the second with lasso, and the third with normal linear regression. Which line-up will give the best results?

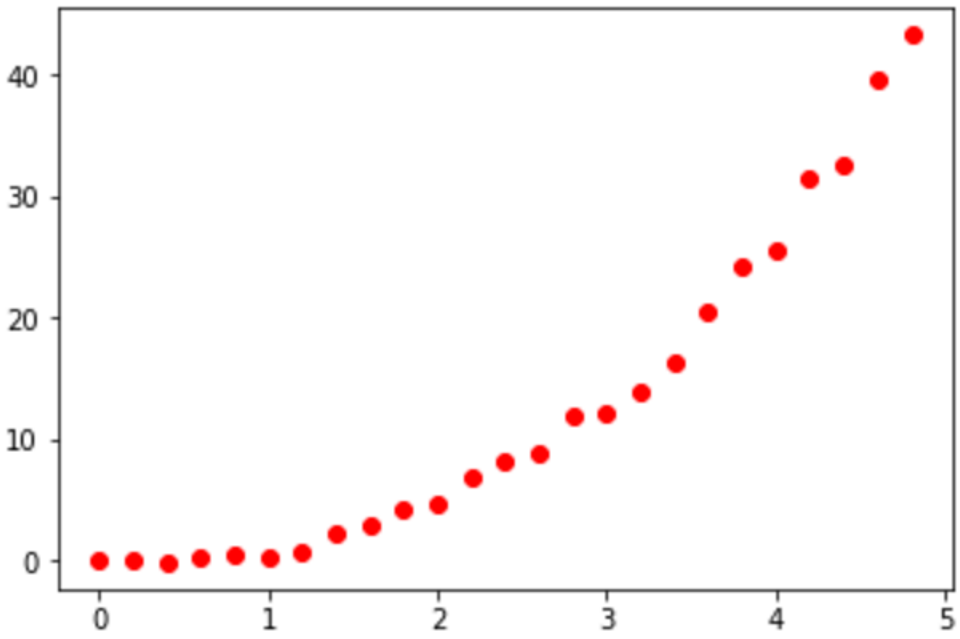
Data Set A:

x_1	x_2	x_3	x_4	y
5	8	97	2	3
2	7	0	2	4
2	6	4	12	14
15	6	-20	5	6
4	8	2	6	5

Data Set B:



Data Set C:



- [A, C, B]
- [B, A, C]
- [C, A, B]
- [C, B, A]

Problem Set 4

Test Your Understanding

Below is a table containing various predictor variables for animals, as well as the one hot representations of their classifications. We want to find an equation which estimates whether a data point represents a tiger or not. As always, we will do so by solving the equation

$$A^T \overrightarrow{b} = A^T A \overrightarrow{x}.$$

Which of our choices gives the correct instantiations of A and $\overrightarrow{b}$ >

Length (ft)	Weight (lbs)	Food (lbs/day)	Lion	Tiger	Bear
10.5	510	17	0	1	0
6.2	430	15	1	0	0
5.7	380	12	1	0	0
7.4	840	27	0	0	1
9.8	470	18	0	1	0

Choices:

1. $A = \begin{bmatrix} 10.5 & 510 & 17 \\ 6.2 & 430 & 15 \\ 5.7 & 380 & 12 \\ 7.4 & 840 & 27 \\ 9.8 & 470 & 18 \end{bmatrix}, \overrightarrow{b} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$

2. $A = \begin{bmatrix} 10.5 & 510 & 17 \\ 9.8 & 470 & 18 \end{bmatrix}, \overrightarrow{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

3. $A = \begin{bmatrix} 10.5 & 510 & 17 & 1 \\ 6.2 & 430 & 15 & 1 \\ 5.7 & 380 & 12 & 1 \\ 7.4 & 840 & 27 & 1 \\ 9.8 & 470 & 18 & 1 \end{bmatrix}, \overrightarrow{b} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$

4. $A = \begin{bmatrix} 10.5 & 510 & 17 & 1 \\ 6.2 & 430 & 15 & 1 \\ 5.7 & 380 & 12 & 1 \\ 7.4 & 840 & 27 & 1 \\ 9.8 & 470 & 18 & 1 \end{bmatrix}, \overrightarrow{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$

A new animal has been brought to the zoo, and our extremely unqualified interm needs help classifying it as a lion, tiger, or bear. The newcomer, Ozzie, is 6 feet long, weighs 850 pounds, and eats 27 pounds of food every day.

After going through the data of zoos all over the country, we have come up with the following functions for each class. x_1, x_2,and x_3 represent length, weight, and food intake, respectively:

$$f_{Lion}(\overrightarrow{x}) = -x_1 - 0.01x_2 - x_3 + 20$$

$$f_{Tiger}(\overrightarrow{x}) = x_1 + 0.01x_2 + x_3 - 10$$

$$f_{Bear}(\overrightarrow{x}) = 0.02x_2 + 3x_3 - 25$$

Using these functions, which animal should we classify Ozzie as?

- Lion
- Tiger
- Bear

Which of the following functions of our linear function $f(\overrightarrow{x})$ solves these problems while maintaining the properties of f that we need for meaningful comparisons between classes?

- $g(\overrightarrow{x}) = sin^2(f(\overrightarrow{x}))$
- $g(\overrightarrow{x}) = \frac{e^{f(\overrightarrow{x})}}{1+e^{f(\overrightarrow{x})}}$
- $g(\overrightarrow{x}) = \frac{f(\overrightarrow{x})^2}{(1-f(\overrightarrow{x}))^2}$
- $g(\overrightarrow{x}) = e^{f(\overrightarrow{x})} - \frac{1}{e^{f(\overrightarrow{x})}}$

A professor wants to estimate whether students will pass his class with their GPAs and ages. He has assigned these to the variables x_1 and x_2, respectively. He has also come up with weights $m_1 = 5$ and $m_2 = 1$, and a bias of $b = -29$.

Recall that the sigmoid function is given by $\sigma(x) = \frac{e^x}{1+e^x}$. What are the odds that a fifteen-year old student with a 3.2 GPA will pass his class, calculated to the nearest hundredth?

- **Note:** Here, it seems like we are only looking at one class, the class of passing students. In reality there are still two classes, students who pass and students who fail.

When classifying data, what is the upper bound of likelihood in the maximal likelihood method? Is this value actually possible to reach?

- 1, Yes
- e, No
- 1, No
- 0, Yes

Suppose we have two classes, red and blue, and a probability function of two predictor variables that is guaranteed to be correct. That is, we have a function, $p(x, y)$, that is guaranteed to give the correct probability that $x$is red.

Additionally, we are given two points, $(3, 2)$ and $(-5, 3)$, and our probability function is given by

$$p(x, y) = \frac{e^{x+y+1}}{1 + e^{x+y+1}}$$

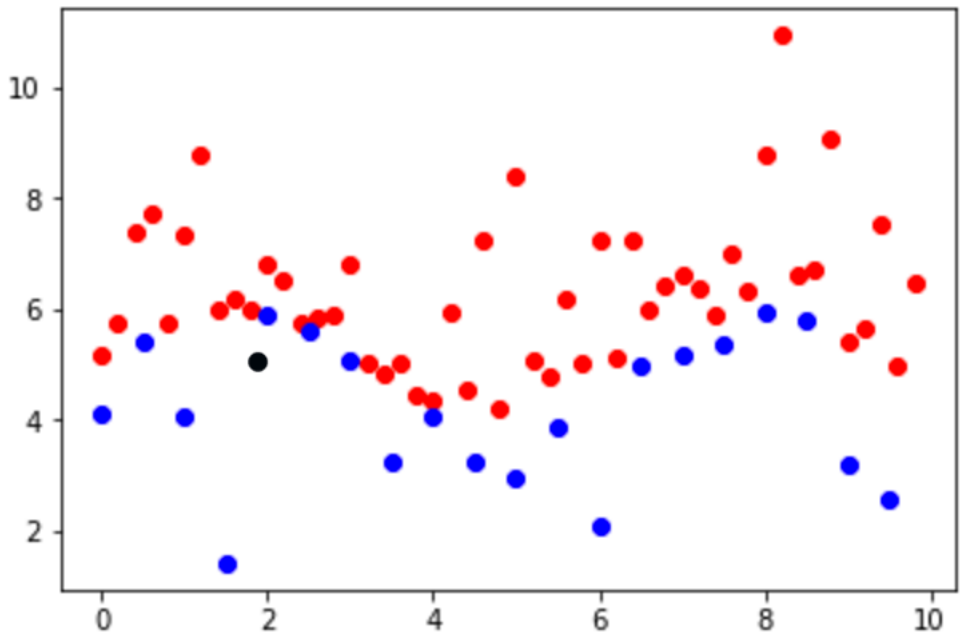
What is the total chance that these two points will have respective classes of red and blue, rounded to the nearest hundredth?

- **Hint:** Assume that all events are independent and that each point is guaranteed to be either red or blue.

A probability function $p(x, y) = \frac{e^{x-y+1}}{1+e^{x-y+1}}$ is found to distinguish between the red and blue cases. With a threshold of 0.7, what will be the curve dividing these two classes? All numbers are rounded to the nearest hundredth.

- $y = \frac{0.45}{x+1}$
- $y = -0.37x - 0.24$
- $2.72^y = -x + 1$
- $y = x + 0.15$

Shown here is a scatterplot representing an arbitrary data set. Using KNN classification, with K = 7, classify the black data point as either the red or blue class.



- Red
- Blue

Say that our data set and the points we are interested in classifying are made up of predictor variables that are always between zero and fifteen. In other words, they are always contained in an $n-$cube with side length fifteen.

If we need 15 points to reach the required density when there is only one predictor variable, about how many points will we need when there are $n$predictor variables?

- $15\sqrt{n}$
- 15^n
- e^{15n}
- $15n$

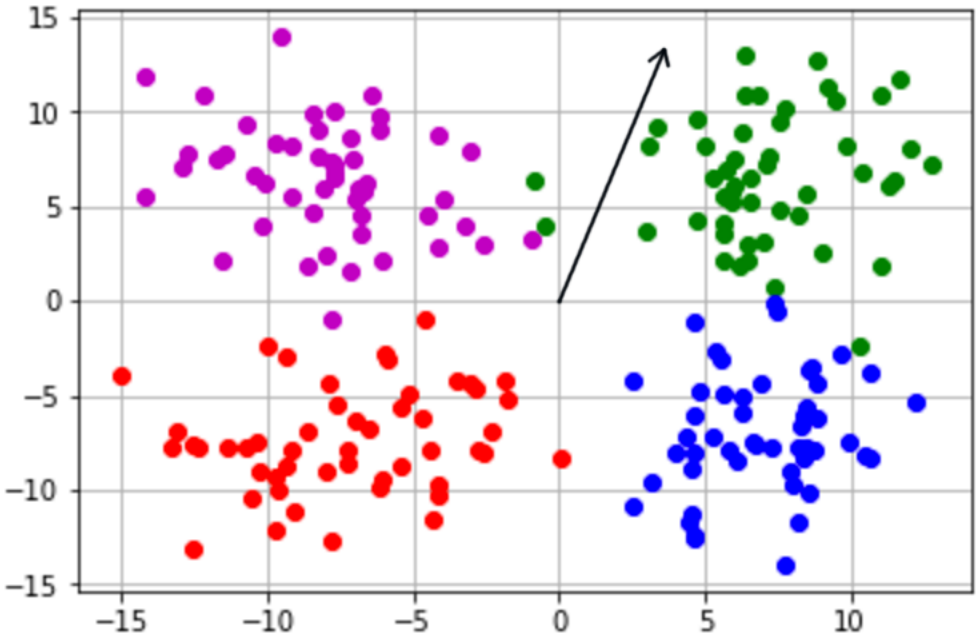
Two scientists want to estimate whether a truck will break down soon based on distance traveled and money spent on maintenance. They have already decided to measure money spent in dollars, but they disagree on how to measure distance. One scientist wants to use inches, while the other wants to use miles. Which of these units will produce a better model?

- Inches
- Miles

Given $\vec{w}$ and b, a perceptron outputs 1 for an input $\vec{x}$ only if

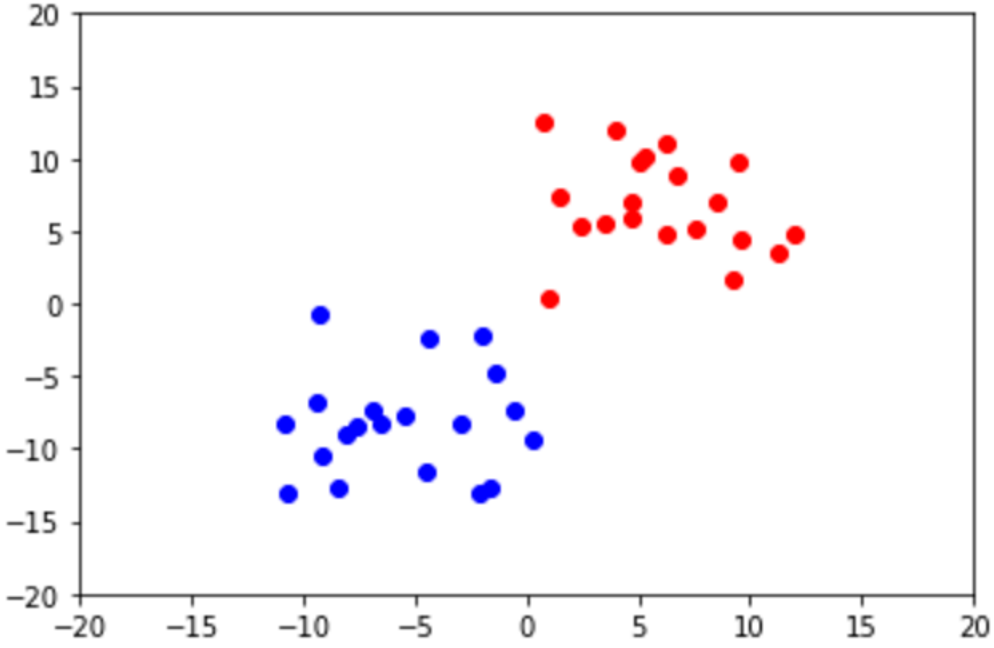
$$\vec{w} \cdot \vec{x} \geq b.$$

Otherwise it outputs 0. With these two outputs, it distinguishes between two classes. Below is the weight factor for a perceptron as well as four groups of points. Which of these groups will entirely be given positive results by the perceptron if the bias is zero?



- Blue
- Green
- Purple
- Red

View the graph below. With a bias of zero, which weighting vector will produce a perceptron that successfully divides the red and blue classes?



- $\vec{w} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
- $\vec{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
- $\vec{x} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$
- $\vec{x} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$

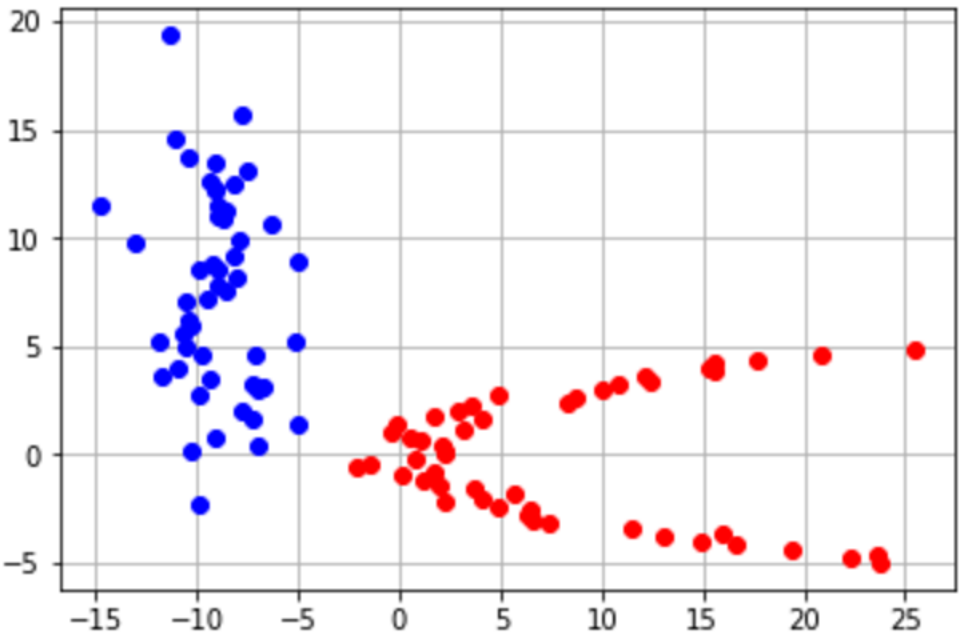
In the process of training a perceptron, we must frequently correct the weights to deal with any misclassified points. For instance, let's say that we have two classes, red and blue, and that our perceptron will see positive results as red and negative results as blue. Generally, if a point $\vec{x}$ is misclassified as blue, we will adjust the weight vector $\vec{w}$ by adding $\vec{x}$ to it.

Which of the following equations relating to this is guaranteed to be true?

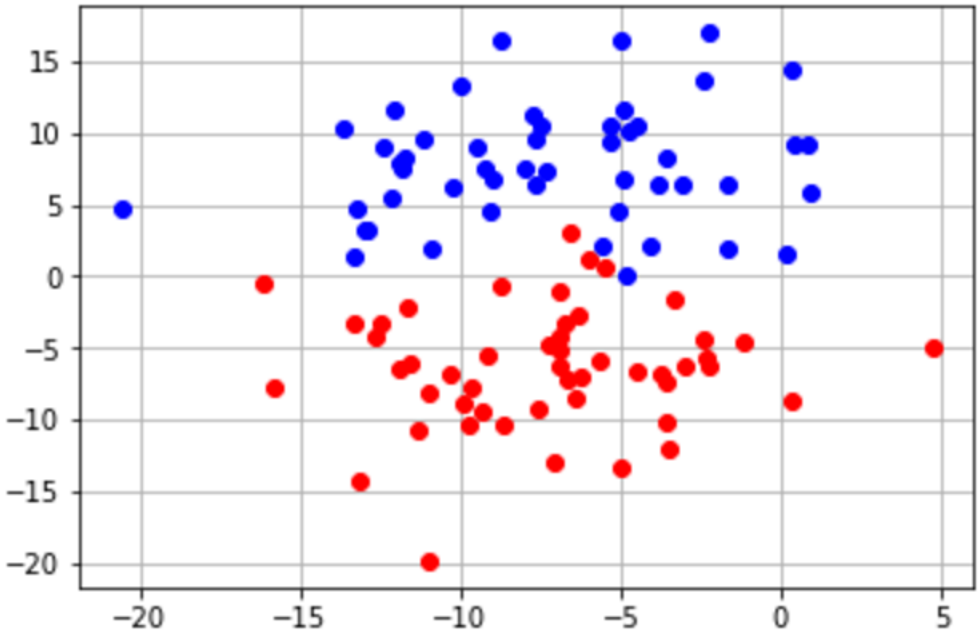
- $(\vec{w} + \vec{x}) \cdot \vec{x} \geq \vec{w} \cdot \vec{x}$
- $|\vec{w} + \vec{x}| \geq |\vec{w}|$
- $(\vec{w} + \vec{x}) \cdot \vec{x} \leq \vec{w} \cdot \vec{x}$
- $(\vec{x} + \vec{x}) \cdot \vec{x} \geq \vec{w} \cdot \vec{x}$

In which of the following sets of points, will a perceptron's vector of weights not converge on a solution?

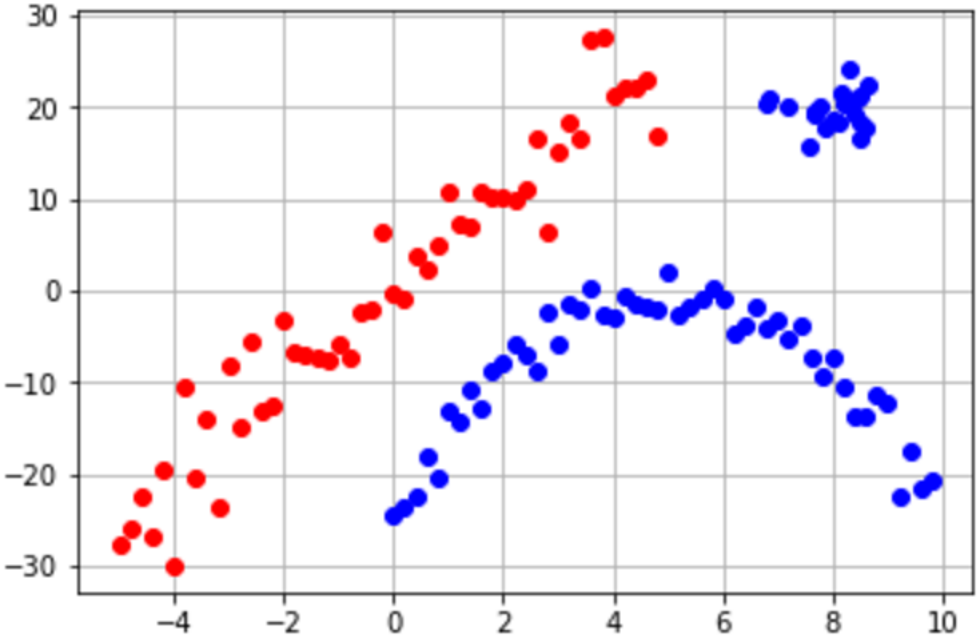
Data Set A:



Data Set B:



Data Set C:



- A
- B
- C

Using regression on an indicator matrix, we have come up with a probability function for each of two classes a and b, based on predictor variables x and y. These are given below with

$$p_a(x) = \frac{e^{10x+5y+12}}{1+e^{10x+5y+12}}$$

$$p_b(x) = \frac{e^{5x+10y+4}}{1+e^{5x+10y+4}}$$

Currently we are making classifications by calculating the probability that a point (x, y) is in each class and then comparing the two values. However, we wish to replace this process with a single perceptron. If this perceptron always gives exactly the same results as our current method, what is the equation for the dividing line which corresponds to it?

- $-5x + 5y = 8$
- $15x + 15y = 16$
- $4x + 3y = 6$
- $1x - 7y = 3$

Data Clustering: A Review

A.K. JAIN

Michigan State University

M.N. MURTY

Indian Institute of Science

AND

P.J. FLYNN

The Ohio State University

Clustering is the unsupervised classification of patterns (observations, data items, or feature vectors) into groups (clusters). The clustering problem has been addressed in many contexts and by researchers in many disciplines; this reflects its broad appeal and usefulness as one of the steps in exploratory data analysis. However, clustering is a difficult problem combinatorially, and differences in assumptions and contexts in different communities has made the transfer of useful generic concepts and methodologies slow to occur. This paper presents an overview of pattern clustering methods from a statistical pattern recognition perspective, with a goal of providing useful advice and references to fundamental concepts accessible to the broad community of clustering practitioners. We present a taxonomy of clustering techniques, and identify cross-cutting themes and recent advances. We also describe some important applications of clustering algorithms such as image segmentation, object recognition, and information retrieval.

Categories and Subject Descriptors: I.5.1 [**Pattern Recognition**]: Models; I.5.3 [**Pattern Recognition**]: Clustering; I.5.4 [**Pattern Recognition**]: Applications—*Computer vision*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Clustering*; I.2.6 [**Artificial Intelligence**]: Learning—*Knowledge acquisition*

General Terms: Algorithms

Additional Key Words and Phrases: Cluster analysis, clustering applications, exploratory data analysis, incremental clustering, similarity indices, unsupervised learning

---

Section 6.1 is based on the chapter "Image Segmentation Using Clustering" by A.K. Jain and P.J. Flynn, *Advances in Image Understanding: A Festschrift for Azriel Rosenfeld* (K. Bowyer and N. Ahuja, Eds.), 1996 IEEE Computer Society Press, and is used by permission of the IEEE Computer Society.

Authors' addresses: A. Jain, Department of Computer Science, Michigan State University, A714 Wells Hall, East Lansing, MI 48824; M. Murty, Department of Computer Science and Automation, Indian Institute of Science, Bangalore, 560 012, India; P. Flynn, Department of Electrical Engineering, The Ohio State University, Columbus, OH 43210.

CONTENTS

1. INTRODUCTION

1.1 Motivation

Data analysis underlies many computing applications, either in a design phase or as part of their on-line operations. Data analysis procedures can be dichotomized as either exploratory or confirmatory, based on the availability of appropriate models for the data source, but a key element in both types of procedures (whether for hypothesis formation or decision-making) is the grouping, or classification of measurements based on either (i) goodness-of-fit to a postulated model, or (ii) natural groupings (clustering) revealed through analysis. Cluster analysis is the organization of a collection of patterns (usually represented as a vector of measurements, or a point in a multidimensional space) into clusters based on similarity. Intuitively, patterns within a valid cluster are more similar to each other than they are to a pattern belonging to a different cluster. An example of clustering is depicted in Figure 1. The input patterns are shown in Figure 1(a), and the desired clusters are shown in Figure 1(b). Here, points belonging to the same cluster are given the same label. The variety of techniques for representing data, measuring proximity (similarity) between data elements, and grouping data elements has produced a rich and often confusing assortment of clustering methods.

It is important to understand the difference between clustering (unsupervised classification) and discriminant analysis (supervised classification). In supervised classification, we are provided with a collection of *labeled* (preclassified) patterns; the problem is to label a newly encountered, yet unlabeled, pattern. Typically, the given labeled (*training*) patterns are used to learn the descriptions of classes which in turn are used to label a new pattern. In the case of clustering, the problem is to group a given collection of unlabeled patterns into meaningful clusters. In a sense, labels are associated with clusters also, but these category labels are *data driven*; that is, they are obtained solely from the data.

Clustering is useful in several exploratory pattern-analysis, grouping, decision-making, and machine-learning situations, including data mining, document retrieval, image segmentation, and pattern classification. However, in many such problems, there is little prior information (e.g., statistical models) available about the data, and the decision-maker must make as few assumptions about the data as possible. It is under these restrictions that clustering methodology is particularly appropriate for the exploration of interrelationships among the data points to make an assessment (perhaps preliminary) of their structure.

The term "clustering" is used in several research communities to describe

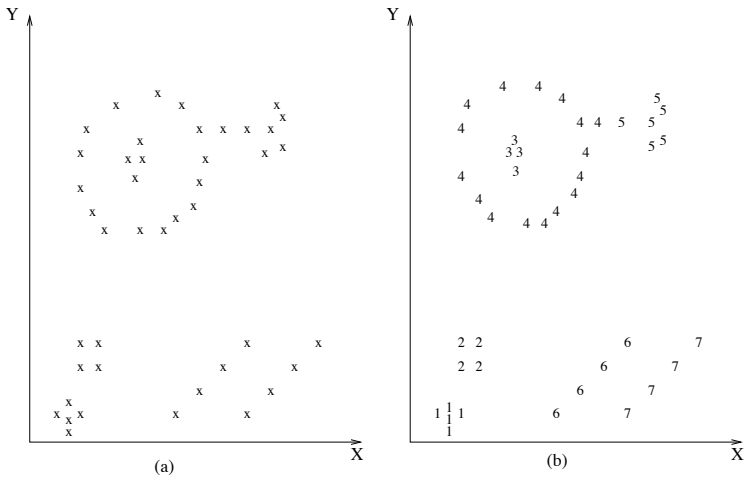


Figure 1. Data clustering.

methods for grouping of unlabeled data. These communities have different terminologies and assumptions for the components of the clustering process and the contexts in which clustering is used. Thus, we face a dilemma regarding the scope of this survey. The production of a truly comprehensive survey would be a monumental task given the sheer mass of literature in this area. The accessibility of the survey might also be questionable given the need to reconcile very different vocabularies and assumptions regarding clustering in the various communities.

The goal of this paper is to survey the core concepts and techniques in the large subset of cluster analysis with its roots in statistics and decision theory. Where appropriate, references will be made to key concepts and techniques arising from clustering methodology in the machine-learning and other communities.

The audience for this paper includes practitioners in the pattern recognition and image analysis communities (who should view it as a summarization of current practice), practitioners in the machine-learning communities (who should view it as a snapshot of a closely related field with a rich history of well-understood techniques), and the broader audience of scientific professionals (who should view it as an accessible introduction to a mature field that is making important contributions to computing application areas).

1.2 Components of a Clustering Task

Typical pattern clustering activity involves the following steps [Jain and Dubes 1988]:

(1) pattern representation (optionally including feature extraction and/or selection),

(2) definition of a pattern proximity measure appropriate to the data domain,

(3) clustering or grouping,

(4) data abstraction (if needed), and

(5) assessment of output (if needed).

Figure 2 depicts a typical sequencing of the first three of these steps, including a feedback path where the grouping process output could affect subsequent feature extraction and similarity computations.

Pattern representation refers to the number of classes, the number of available patterns, and the number, type, and scale of the features available to the clustering algorithm. Some of this information may not be controllable by the

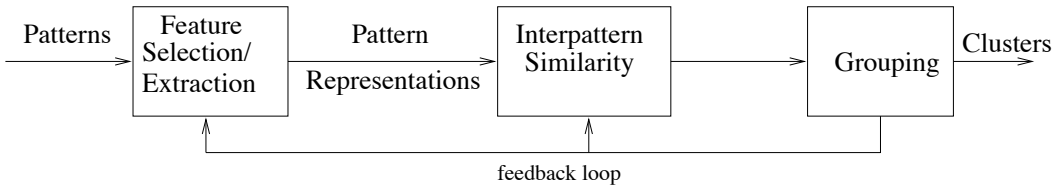


Figure 2. Stages in clustering.

practitioner. *Feature selection* is the process of identifying the most effective subset of the original features to use in clustering. *Feature extraction* is the use of one or more transformations of the input features to produce new salient features. Either or both of these techniques can be used to obtain an appropriate set of features to use in clustering.

Pattern proximity is usually measured by a distance function defined on pairs of patterns. A variety of distance measures are in use in the various communities [Anderberg 1973; Jain and Dubes 1988; Diday and Simon 1976]. A simple distance measure like Euclidean distance can often be used to reflect dissimilarity between two patterns, whereas other similarity measures can be used to characterize the conceptual similarity between patterns [Michalski and Stepp 1983]. Distance measures are discussed in Section 4.

The *grouping* step can be performed in a number of ways. The output clustering (or clusterings) can be hard (a partition of the data into groups) or fuzzy (where each pattern has a variable degree of membership in each of the output clusters). Hierarchical clustering algorithms produce a nested series of partitions based on a criterion for merging or splitting clusters based on similarity. Partitional clustering algorithms identify the partition that optimizes (usually locally) a clustering criterion. Additional techniques for the grouping operation include probabilistic [Brailovski 1991] and graph-theoretic [Zahn 1971] clustering methods. The variety of techniques for cluster formation is described in Section 5.

Data abstraction is the process of extracting a simple and compact representation of a data set. Here, simplicity is either from the perspective of automatic analysis (so that a machine can perform further processing efficiently) or it is human-oriented (so that the representation obtained is easy to comprehend and intuitively appealing). In the clustering context, a typical data abstraction is a compact description of each cluster, usually in terms of cluster prototypes or representative patterns such as the centroid [Diday and Simon 1976].

How is the output of a clustering algorithm evaluated? What characterizes a 'good' clustering result and a 'poor' one? All clustering algorithms will, when presented with data, produce clusters — regardless of whether the data contain clusters or not. If the data does contain clusters, some clustering algorithms may obtain 'better' clusters than others. The assessment of a clustering procedure's output, then, has several facets. One is actually an assessment of the data domain rather than the clustering algorithm itself— data which do not contain clusters should not be processed by a clustering algorithm. The study of *cluster tendency*, wherein the input data are examined to see if there is any merit to a cluster analysis prior to one being performed, is a relatively inactive research area, and will not be considered further in this survey. The interested reader is referred to Dubes [1987] and Cheng [1995] for information.

Cluster validity analysis, by contrast, is the assessment of a clustering procedure's output. Often this analysis uses a specific criterion of optimality; however, these criteria are usually arrived at

subjectively. Hence, little in the way of 'gold standards' exist in clustering except in well-prescribed subdomains. Validity assessments are objective [Dubes 1993] and are performed to determine whether the output is meaningful. A clustering structure is valid if it cannot reasonably have occurred by chance or as an artifact of a clustering algorithm. When statistical approaches to clustering are used, validation is accomplished by carefully applying statistical methods and testing hypotheses. There are three types of validation studies. An *external* assessment of validity compares the recovered structure to an *a priori* structure. An *internal* examination of validity tries to determine if the structure is intrinsically appropriate for the data. A *relative* test compares two structures and measures their relative merit. Indices used for this comparison are discussed in detail in Jain and Dubes [1988] and Dubes [1993], and are not discussed further in this paper.

1.3 The User's Dilemma and the Role of Expertise

The availability of such a vast collection of clustering algorithms in the literature can easily confound a user attempting to select an algorithm suitable for the problem at hand. In Dubes and Jain [1976], a set of admissibility criteria defined by Fisher and Van Ness [1971] are used to compare clustering algorithms. These admissibility criteria are based on: (1) the manner in which clusters are formed, (2) the structure of the data, and (3) sensitivity of the clustering technique to changes that do not affect the structure of the data. However, there is no critical analysis of clustering algorithms dealing with the important questions such as

—How should the data be normalized?

—Which similarity measure is appropriate to use in a given situation?

—How should domain knowledge be utilized in a particular clustering problem?

—How can a vary large data set (say, a million patterns) be clustered efficiently?

These issues have motivated this survey, and its aim is to provide a perspective on the state of the art in clustering methodology and algorithms. With such a perspective, an informed practitioner should be able to confidently assess the tradeoffs of different techniques, and ultimately make a competent decision on a technique or suite of techniques to employ in a particular application.

There is no clustering technique that is universally applicable in uncovering the variety of structures present in multidimensional data sets. For example, consider the two-dimensional data set shown in Figure 1(a). Not all clustering techniques can uncover all the clusters present here with equal facility, because clustering algorithms often contain implicit assumptions about cluster shape or multiple-cluster configurations based on the similarity measures and grouping criteria used.

Humans perform competitively with automatic clustering procedures in two dimensions, but most real problems involve clustering in higher dimensions. It is difficult for humans to obtain an intuitive interpretation of data embedded in a high-dimensional space. In addition, data hardly follow the "ideal" structures (e.g., hyperspherical, linear) shown in Figure 1. This explains the large number of clustering algorithms which continue to appear in the literature; each new clustering algorithm performs slightly better than the existing ones on a specific distribution of patterns.

It is essential for the user of a clustering algorithm to not only have a thorough understanding of the particular technique being utilized, but also to know the details of the data gathering process and to have some domain expertise; the more information the user has about the data at hand, the more likely the user would be able to succeed in assessing its true class structure [Jain and Dubes 1988]. This domain informa-

tion can also be used to improve the quality of feature extraction, similarity computation, grouping, and cluster representation [Murty and Jain 1995].

Appropriate constraints on the data source can be incorporated into a clustering procedure. One example of this is *mixture resolving* [Titterington et al. 1985], wherein it is assumed that the data are drawn from a mixture of an unknown number of densities (often assumed to be multivariate Gaussian). The clustering problem here is to identify the number of mixture components and the parameters of each component. The concept of *density* clustering and a methodology for decomposition of feature spaces [Bajcsy 1997] have also been incorporated into traditional clustering methodology, yielding a technique for extracting overlapping clusters.

1.4 History

Even though there is an increasing interest in the use of clustering methods in pattern recognition [Anderberg 1973], image processing [Jain and Flynn 1996] and information retrieval [Rasmussen 1992; Salton 1991], clustering has a rich history in other disciplines [Jain and Dubes 1988] such as biology, psychiatry, psychology, archaeology, geology, geography, and marketing. Other terms more or less synonymous with clustering include *unsupervised learning* [Jain and Dubes 1988], *numerical taxonomy* [Sneath and Sokal 1973], *vector quantization* [Oehler and Gray 1995], and *learning by observation* [Michalski and Stepp 1983]. The field of spatial analysis of point patterns [Ripley 1988] is also related to cluster analysis. The importance and interdisciplinary nature of clustering is evident through its vast literature.

A number of books on clustering have been published [Jain and Dubes 1988; Anderberg 1973; Hartigan 1975; Spath 1980; Duran and Odell 1974; Everitt 1993; Backer 1995], in addition to some useful and influential review papers. A survey of the state of the art in clustering *circa* 1978 was reported in Dubes and Jain [1980]. A comparison of various clustering algorithms for constructing the minimal spanning tree and the short spanning path was given in Lee [1981]. Cluster analysis was also surveyed in Jain et al. [1986]. A review of image segmentation by clustering was reported in Jain and Flynn [1996]. Comparisons of various combinatorial optimization schemes, based on experiments, have been reported in Mishra and Raghavan [1994] and Al-Sultan and Khan [1996].

1.5 Outline

This paper is organized as follows. Section 2 presents definitions of terms to be used throughout the paper. Section 3 summarizes pattern representation, feature extraction, and feature selection. Various approaches to the computation of proximity between patterns are discussed in Section 4. Section 5 presents a taxonomy of clustering approaches, describes the major techniques in use, and discusses emerging techniques for clustering incorporating non-numeric constraints and the clustering of large sets of patterns. Section 6 discusses applications of clustering methods to image analysis and data mining problems. Finally, Section 7 presents some concluding remarks.

2. DEFINITIONS AND NOTATION

The following terms and notation are used throughout this paper.

—A *pattern* (or *feature vector*, *observation*, or *datum*) $\mathbf{x}$ is a single data item used by the clustering algorithm. It typically consists of a vector of d measurements: $\mathbf{x} = (x_1, \ldots x_d)$.

—The individual scalar components x_i of a pattern $\mathbf{x}$ are called *features* (or *attributes*).

—d is the *dimensionality* of the pattern or of the pattern space.

—A *pattern set* is denoted $\mathcal{X} = \{\mathbf{x}_1, \ldots \mathbf{x}_n\}$. The ith pattern in $\mathcal{X}$ is denoted $\mathbf{x}_i = (x_{i,1}, \ldots x_{i,d})$. In many cases a pattern set to be clustered is viewed as an $n \times d$ *pattern matrix*.

—A *class*, in the abstract, refers to a state of nature that governs the pattern generation process in some cases. More concretely, a class can be viewed as a source of patterns whose distribution in feature space is governed by a probability density specific to the class. Clustering techniques attempt to group patterns so that the classes thereby obtained reflect the different pattern generation processes represented in the pattern set.

—*Hard* clustering techniques assign a *class label* l_i to each patterns $\mathbf{x}_i$, identifying its class. The set of all labels for a pattern set $\mathcal{X}$ is $\mathcal{L} = \{l_1, \ldots l_n\}$, with $l_i \in \{1, \cdots, k\}$, where k is the number of clusters.

—*Fuzzy* clustering procedures assign to each input pattern $\mathbf{x}_i$ a fractional degree of membership f_{ij} in each output cluster j.

—A *distance measure* (a specialization of a proximity measure) is a metric (or quasi-metric) on the feature space used to quantify the similarity of patterns.

3. PATTERN REPRESENTATION, FEATURE SELECTION AND EXTRACTION

There are no theoretical guidelines that suggest the appropriate patterns and features to use in a specific situation. Indeed, the pattern generation process is often not directly controllable; the user's role in the pattern representation process is to gather facts and conjectures about the data, optionally perform feature selection and extraction, and design the subsequent elements of the clustering system. Because of the difficulties surrounding pattern representation, it is conveniently assumed that the pattern representation is available prior to clustering. Nonetheless, a careful investigation of the available features and any available transformations (even simple ones) can yield significantly improved clustering results. A good pattern representation can often yield a simple and easily understood clustering; a poor pattern representation may yield a complex clustering whose true structure is difficult or impossible to discern. Figure 3 shows a simple example. The points in this 2D feature space are arranged in a curvilinear cluster of approximately constant distance from the origin. If one chooses Cartesian coordinates to represent the patterns, many clustering algorithms would be likely to fragment the cluster into two or more clusters, since it is not compact. If, however, one uses a polar coordinate representation for the clusters, the radius coordinate exhibits tight clustering and a one-cluster solution is likely to be easily obtained.

A pattern can measure either a physical object (e.g., a chair) or an abstract notion (e.g., a style of writing). As noted above, patterns are represented conventionally as multidimensional vectors, where each dimension is a single feature [Duda and Hart 1973]. These features can be either quantitative or qualitative. For example, if *weight* and *color* are the two features used, then $(20, black)$ is the representation of a black object with 20 units of weight. The features can be subdivided into the following types [Gowda and Diday 1992]:

(1) Quantitative features: e.g.
   (a) continuous values (e.g., weight);
   (b) discrete values (e.g., the number of computers);
   (c) interval values (e.g., the duration of an event).

(2) Qualitative features:
   (a) nominal or unordered (e.g., color);

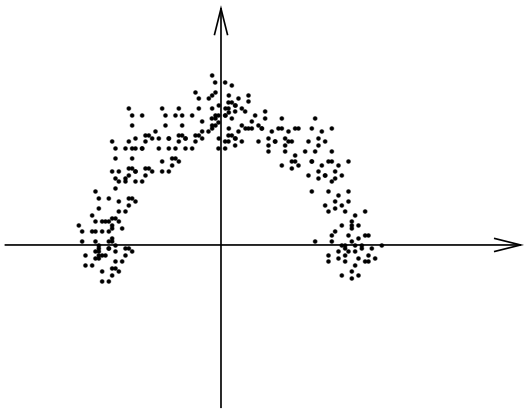


Figure 3. A curvilinear cluster whose points are approximately equidistant from the origin. Different pattern representations (coordinate systems) would cause clustering algorithms to yield different results for this data (see text).

(b) ordinal (e.g., military rank or qualitative evaluations of temperature ("cool" or "hot") or sound intensity ("quiet" or "loud")).

Quantitative features can be measured on a ratio scale (with a meaningful reference value, such as temperature), or on nominal or ordinal scales.

One can also use structured features [Michalski and Stepp 1983] which are represented as trees, where the parent node represents a generalization of its child nodes. For example, a parent node "vehicle" may be a generalization of children labeled "cars," "buses," "trucks," and "motorcycles." Further, the node "cars" could be a generalization of cars of the type "Toyota," "Ford," "Benz," etc. A generalized representation of patterns, called *symbolic objects* was proposed in Diday [1988]. Symbolic objects are defined by a logical conjunction of events. These events link values and features in which the features can take one or more values and all the objects need not be defined on the same set of features.

It is often valuable to isolate only the most descriptive and discriminatory features in the input set, and utilize those features exclusively in subsequent analysis. Feature selection techniques identify a subset of the existing features for subsequent use, while feature extraction techniques compute new features from the original set. In either case, the goal is to improve classification performance and/or computational efficiency. Feature selection is a well-explored topic in statistical pattern recognition [Duda and Hart 1973]; however, in a clustering context (i.e., lacking class labels for patterns), the feature selection process is of necessity ad hoc, and might involve a trial-and-error process where various subsets of features are selected, the resulting patterns clustered, and the output evaluated using a validity index. In contrast, some of the popular feature extraction processes (e.g., principal components analysis [Fukunaga 1990]) do not depend on labeled data and can be used directly. Reduction of the number of features has an additional benefit, namely the ability to produce output that can be visually inspected by a human.

4. SIMILARITY MEASURES

Since similarity is fundamental to the definition of a cluster, a measure of the similarity between two patterns drawn from the same feature space is essential to most clustering procedures. Because of the variety of feature types and scales, the distance measure (or measures) must be chosen carefully. It is most common to calculate the *dissimilarity* between two patterns using a distance measure defined on the feature space. We will focus on the well-known distance measures used for patterns whose features are all continuous.

The most popular metric for continuous features is the *Euclidean distance*

$$d_2(\mathbf{x}_i, \mathbf{x}_j) = \Big(\sum_{k=1}^{d} (x_{i,k} - x_{j,k})^2\Big)^{1/2} = \|\mathbf{x}_i - \mathbf{x}_j\|_2,$$

which is a special case ($p=2$) of the Minkowski metric

$$d_p(\mathbf{x}_i, \mathbf{x}_j) = \left(\sum_{k=1}^{d} |x_{i,k} - x_{j,k}|^p\right)^{1/p}$$

$$= \|\mathbf{x}_i - \mathbf{x}_j\|_p.$$

The Euclidean distance has an intuitive appeal as it is commonly used to evaluate the proximity of objects in two or three-dimensional space. It works well when a data set has "compact" or "isolated" clusters [Mao and Jain 1996]. The drawback to direct use of the Minkowski metrics is the tendency of the largest-scaled feature to dominate the others. Solutions to this problem include normalization of the continuous features (to a common range or variance) or other weighting schemes. Linear correlation among features can also distort distance measures; this distortion can be alleviated by applying a whitening transformation to the data or by using the squared Mahalanobis distance

$$d_M(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i - \mathbf{x}_j)\Sigma^{-1}(\mathbf{x}_i - \mathbf{x}_j)^T,$$

where the patterns $\mathbf{x}_i$ and $\mathbf{x}_j$ are assumed to be row vectors, and Σ is the sample covariance matrix of the patterns or the known covariance matrix of the pattern generation process; $d_M(\cdot\,, \cdot)$ assigns different weights to different features based on their variances and pairwise linear correlations. Here, it is implicitly assumed that class conditional densities are unimodal and characterized by multidimensional spread, i.e., that the densities are multivariate Gaussian. The regularized Mahalanobis distance was used in Mao and Jain [1996] to extract hyperellipsoidal clusters. Recently, several researchers [Huttenlocher et al. 1993; Dubuisson and Jain 1994] have used the Hausdorff distance in a point set matching context.

Some clustering algorithms work on a matrix of proximity values instead of on the original pattern set. It is useful in such situations to precompute all the $n(n-1)/2$ pairwise distance values for the n patterns and store them in a (symmetric) matrix.

Computation of distances between patterns with some or all features being noncontinuous is problematic, since the different types of features are not comparable and (as an extreme example) the notion of proximity is effectively binary-valued for nominal-scaled features. Nonetheless, practitioners (especially those in machine learning, where mixed-type patterns are common) have developed proximity measures for heterogeneous type patterns. A recent example is Wilson and Martinez [1997], which proposes a combination of a modified Minkowski metric for continuous features and a distance based on counts (population) for nominal attributes. A variety of other metrics have been reported in Diday and Simon [1976] and Ichino and Yaguchi [1994] for computing the similarity between patterns represented using quantitative as well as qualitative features.

Patterns can also be represented using string or tree structures [Knuth 1973]. Strings are used in syntactic clustering [Fu and Lu 1977]. Several measures of similarity between strings are described in Baeza-Yates [1992]. A good summary of similarity measures between trees is given by Zhang [1995]. A comparison of syntactic and statistical approaches for pattern recognition using several criteria was presented in Tanaka [1995] and the conclusion was that syntactic methods are inferior in every aspect. Therefore, we do not consider syntactic methods further in this paper.

There are some distance measures reported in the literature [Gowda and Krishna 1977; Jarvis and Patrick 1973] that take into account the effect of surrounding or neighboring points. These surrounding points are called *context* in Michalski and Stepp [1983]. The similarity between two points $\mathbf{x}_i$ and $\mathbf{x}_j$, given this context, is given by

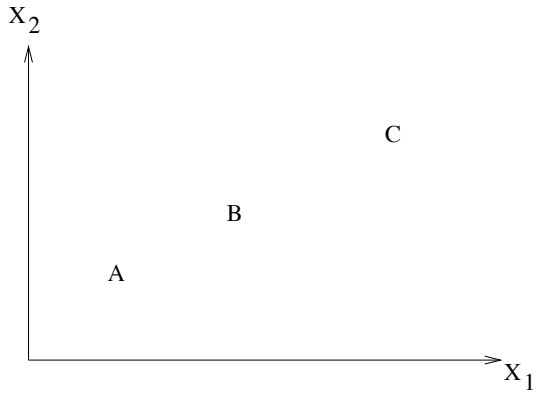


Figure 4. A and B are more similar than A and C.

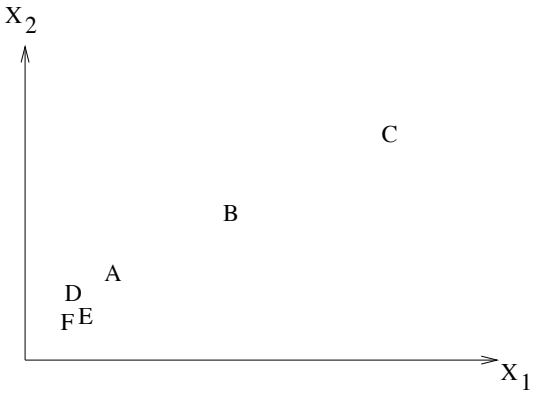


Figure 5. After a change in context, B and C are more similar than B and A.

$$s(\mathbf{x}_i, \mathbf{x}_j) = f(\mathbf{x}_i, \mathbf{x}_j, \mathscr{E}),$$

where $\mathscr{E}$ is the context (the set of surrounding points). One metric defined using context is the *mutual neighbor distance* (MND), proposed in Gowda and Krishna [1977], which is given by

$$MND(\mathbf{x}_i, \mathbf{x}_j) = NN(\mathbf{x}_i, \mathbf{x}_j) + NN(\mathbf{x}_j, \mathbf{x}_i),$$

where $NN(\mathbf{x}_i, \mathbf{x}_j)$ is the neighbor number of $\mathbf{x}_j$ with respect to $\mathbf{x}_i$. Figures 4 and 5 give an example. In Figure 4, the nearest neighbor of A is B, and B's nearest neighbor is A. So, $NN(A, B) = NN(B, A) = 1$ and the MND between A and B is 2. However, $NN(B, C) = 1$ but $NN(C, B) = 2$, and therefore $MND(B, C) = 3$. Figure 5 was obtained from Figure 4 by adding three new points D, E, and F. Now $MND(B, C) = 3$ (as before), but $MND(A, B) = 5$. The MND between A and B has increased by introducing additional points, even though A and B have not moved. The MND is not a metric (it does not satisfy the triangle inequality [Zhang 1995]). In spite of this, MND has been successfully applied in several clustering applications [Gowda and Diday 1992]. This observation supports the viewpoint that the dissimilarity does not need to be a metric.

Watanabe's theorem of the ugly duckling [Watanabe 1985] states:

"*Insofar as we use a finite set of predicates that are capable of distinguishing any two objects considered, the number of predicates shared by any two such objects is constant, independent of the choice of objects.*"

This implies that it is possible to make any two arbitrary patterns equally similar by encoding them with a sufficiently large number of features. As a consequence, any two arbitrary patterns are equally similar, unless we use some additional domain information. For example, in the case of conceptual clustering [Michalski and Stepp 1983], the similarity between $\mathbf{x}_i$ and $\mathbf{x}_j$ is defined as

$$s(\mathbf{x}_i, \mathbf{x}_j) = f(\mathbf{x}_i, \mathbf{x}_j, \mathscr{C}, \mathscr{E}),$$

where $\mathscr{C}$ is a set of pre-defined concepts. This notion is illustrated with the help of Figure 6. Here, the Euclidean distance between points A and B is less than that between B and C. However, B and C can be viewed as "more similar" than A and B because B and C belong to the same concept (ellipse) and A belongs to a different concept (rectangle). The conceptual similarity measure is the most general similarity measure. We

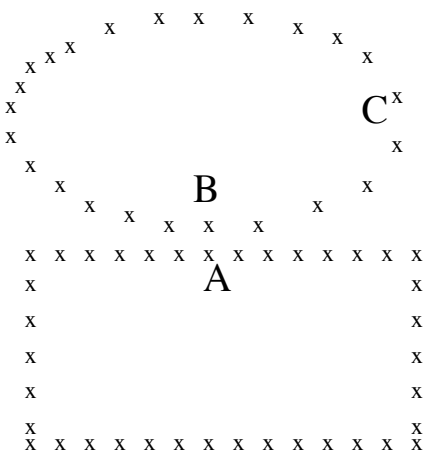


Figure 6. Conceptual similarity between points .

discuss several pragmatic issues associated with its use in Section 5.

5. CLUSTERING TECHNIQUES

Different approaches to clustering data can be described with the help of the hierarchy shown in Figure 7 (other taxonometric representations of clustering methodology are possible; ours is based on the discussion in Jain and Dubes [1988]). At the top level, there is a distinction between hierarchical and partitional approaches (hierarchical methods produce a nested series of partitions, while partitional methods produce only one).

The taxonomy shown in Figure 7 must be supplemented by a discussion of cross-cutting issues that may (in principle) affect all of the different approaches regardless of their placement in the taxonomy.

—Agglomerative *vs.* divisive: This aspect relates to algorithmic structure and operation. An agglomerative approach begins with each pattern in a distinct (singleton) cluster, and successively merges clusters together until a stopping criterion is satisfied. A divisive method begins with all patterns in a single cluster and performs splitting until a stopping criterion is met.

—Monothetic *vs.* polythetic: This aspect relates to the sequential or simultaneous use of features in the clustering process. Most algorithms are polythetic; that is, all features enter into the computation of distances between patterns, and decisions are based on those distances. A simple monothetic algorithm reported in Anderberg [1973] considers features sequentially to divide the given collection of patterns. This is illustrated in Figure 8. Here, the collection is divided into two groups using feature x_1; the vertical broken line V is the separating line. Each of these clusters is further divided independently using feature x_2, as depicted by the broken lines H_1 and H_2. The major problem with this algorithm is that it generates 2^d clusters where d is the dimensionality of the patterns. For large values of d ($d > 100$ is typical in information retrieval applications [Salton 1991]), the number of clusters generated by this algorithm is so large that the data set is divided into uninterestingly small and fragmented clusters.

—Hard *vs.* fuzzy: A hard clustering algorithm allocates each pattern to a single cluster during its operation and in its output. A fuzzy clustering method assigns degrees of membership in several clusters to each input pattern. A fuzzy clustering can be converted to a hard clustering by assigning each pattern to the cluster with the largest measure of membership.

—Deterministic *vs.* stochastic: This issue is most relevant to partitional approaches designed to optimize a squared error function. This optimization can be accomplished using traditional techniques or through a random search of the state space consisting of all possible labelings.

—Incremental *vs.* non-incremental: This issue arises when the pattern set

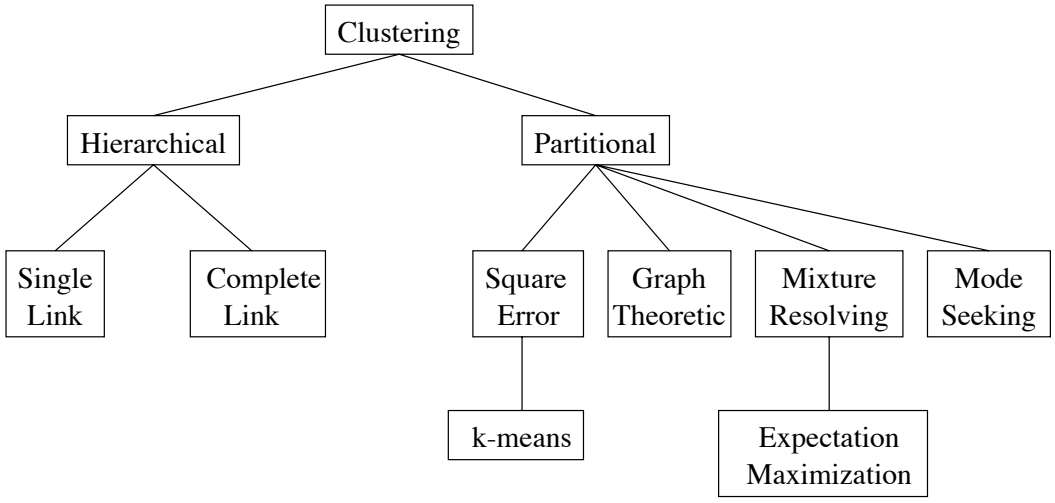


Figure 7. A taxonomy of clustering approaches.

to be clustered is large, and constraints on execution time or memory space affect the architecture of the algorithm. The early history of clustering methodology does not contain many examples of clustering algorithms designed to work with large data sets, but the advent of data mining has fostered the development of clustering algorithms that minimize the number of scans through the pattern set, reduce the number of patterns examined during execution, or reduce the size of data structures used in the algorithm's operations.

A cogent observation in Jain and Dubes [1988] is that the specification of an algorithm for clustering usually leaves considerable flexibilty in implementation.

5.1 Hierarchical Clustering Algorithms

The operation of a hierarchical clustering algorithm is illustrated using the two-dimensional data set in Figure 9. This figure depicts seven patterns labeled A, B, C, D, E, F, and G in three clusters. A hierarchical algorithm yields a *dendrogram* representing the nested grouping of patterns and similarity levels at which groupings change. A dendrogram corresponding to the seven

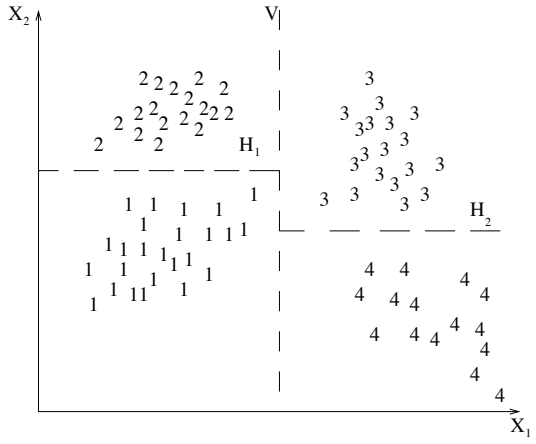


Figure 8. Monothetic partitional clustering.

points in Figure 9 (obtained from the single-link algorithm [Jain and Dubes 1988]) is shown in Figure 10. The dendrogram can be broken at different levels to yield different clusterings of the data.

Most hierarchical clustering algorithms are variants of the single-link [Sneath and Sokal 1973], complete-link [King 1967], and minimum-variance [Ward 1963; Murtagh 1984] algorithms. Of these, the single-link and complete-link algorithms are most popular. These two algorithms differ in the way they characterize the similarity between a pair of clusters. In the single-link method, the distance between two clus-

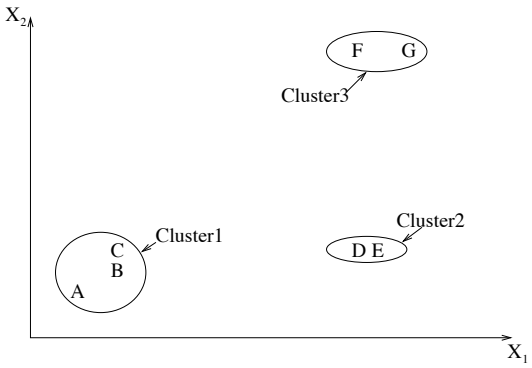


Figure 9. Points falling in three clusters.

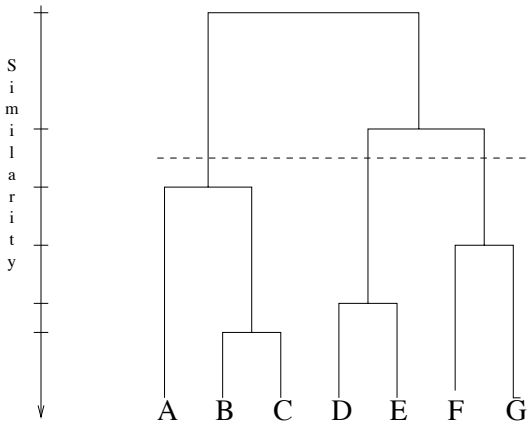


Figure 10. The dendrogram obtained using the single-link algorithm.

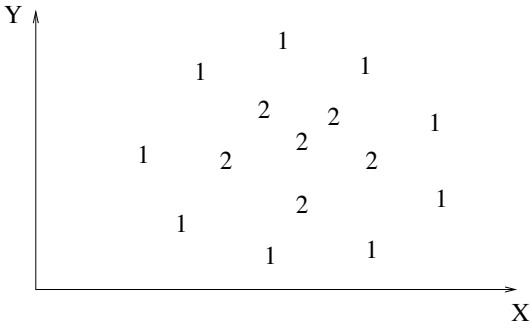


Figure 11. Two concentric clusters.

ters is the *minimum* of the distances between all pairs of patterns drawn from the two clusters (one pattern from the first cluster, the other from the second). In the complete-link algorithm, the distance between two clusters is the *maximum* of all pairwise distances between patterns in the two clusters. In either case, two clusters are merged to form a larger cluster based on minimum distance criteria. The complete-link algorithm produces tightly bound or compact clusters [Baeza-Yates 1992]. The single-link algorithm, by contrast, suffers from a chaining effect [Nagy 1968]. It has a tendency to produce clusters that are straggly or elongated. There are two clusters in Figures 12 and 13 separated by a "bridge" of noisy patterns. The single-link algorithm produces the clusters shown in Figure 12, whereas the complete-link algorithm obtains the clustering shown in Figure 13. The clusters obtained by the complete-link algorithm are more compact than those obtained by the single-link algorithm; the cluster labeled 1 obtained using the single-link algorithm is elongated because of the noisy patterns labeled "*". The single-link algorithm is more versatile than the complete-link algorithm, otherwise. For example, the single-link algorithm can extract the concentric clusters shown in Figure 11, but the complete-link algorithm cannot. However, from a pragmatic viewpoint, it has been observed that the complete-link algorithm produces more useful hierarchies in many applications than the single-link algorithm [Jain and Dubes 1988].

Agglomerative Single-Link Clustering Algorithm

(1) Place each pattern in its own cluster. Construct a list of interpattern distances for all distinct unordered pairs of patterns, and sort this list in ascending order.

(2) Step through the sorted list of distances, forming for each distinct dissimilarity value d_k a graph on the patterns where pairs of patterns closer than d_k are connected by a graph edge. If all the patterns are members of a connected graph, stop. Otherwise, repeat this step.

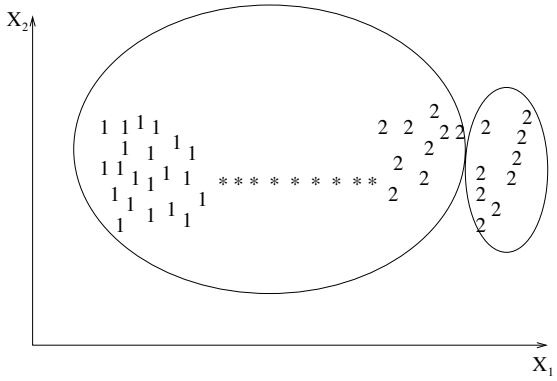


Figure 12. A single-link clustering of a pattern set containing two classes (1 and 2) connected by a chain of noisy patterns (*).

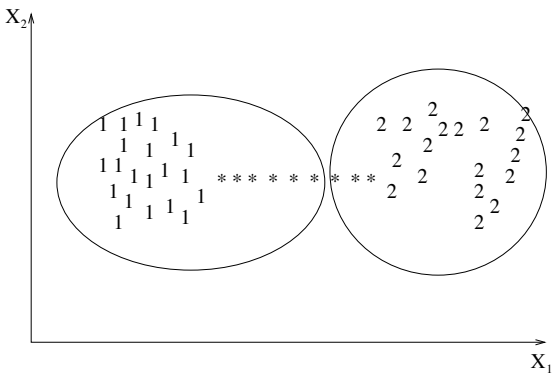


Figure 13. A complete-link clustering of a pattern set containing two classes (1 and 2) connected by a chain of noisy patterns (*).

(3) The output of the algorithm is a nested hierarchy of graphs which can be cut at a desired dissimilarity level forming a partition (clustering) identified by simply connected components in the corresponding graph.

Agglomerative Complete-Link Clustering Algorithm

(1) Place each pattern in its own cluster. Construct a list of interpattern distances for all distinct unordered pairs of patterns, and sort this list in ascending order.

(2) Step through the sorted list of distances, forming for each distinct dissimilarity value d_k a graph on the patterns where pairs of patterns closer than d_k are connected by a graph edge. If all the patterns are members of a completely connected graph, stop.

(3) The output of the algorithm is a nested hierarchy of graphs which can be cut at a desired dissimilarity level forming a partition (clustering) identified by completely connected components in the corresponding graph.

Hierarchical algorithms are more versatile than partitional algorithms. For example, the single-link clustering algorithm works well on data sets containing non-isotropic clusters including well-separated, chain-like, and concentric clusters, whereas a typical partitional algorithm such as the k-means algorithm works well only on data sets having isotropic clusters [Nagy 1968]. On the other hand, the time and space complexities [Day 1992] of the partitional algorithms are typically lower than those of the hierarchical algorithms. It is possible to develop hybrid algorithms [Murty and Krishna 1980] that exploit the good features of both categories.

Hierarchical Agglomerative Clustering Algorithm

(1) Compute the proximity matrix containing the distance between each pair of patterns. Treat each pattern as a cluster.

(2) Find the most similar pair of clusters using the proximity matrix. Merge these two clusters into one cluster. Update the proximity matrix to reflect this merge operation.

(3) If all patterns are in one cluster, stop. Otherwise, go to step 2.

Based on the way the proximity matrix is updated in step 2, a variety of agglomerative algorithms can be designed. Hierarchical divisive algorithms start with a single cluster of all the given objects and keep splitting the clusters based on some criterion to obtain a partition of singleton clusters.

5.2 Partitional Algorithms

A partitional clustering algorithm obtains a single partition of the data instead of a clustering structure, such as the dendrogram produced by a hierarchical technique. Partitional methods have advantages in applications involving large data sets for which the construction of a dendrogram is computationally prohibitive. A problem accompanying the use of a partitional algorithm is the choice of the number of desired output clusters. A seminal paper [Dubes 1987] provides guidance on this key design decision. The partitional techniques usually produce clusters by optimizing a criterion function defined either locally (on a subset of the patterns) or globally (defined over all of the patterns). Combinatorial search of the set of possible labelings for an optimum value of a criterion is clearly computationally prohibitive. In practice, therefore, the algorithm is typically run multiple times with different starting states, and the best configuration obtained from all of the runs is used as the output clustering.

5.2.1 *Squared Error Algorithms*. The most intuitive and frequently used criterion function in partitional clustering techniques is the squared error criterion, which tends to work well with isolated and compact clusters. The squared error for a clustering $\mathcal{L}$ of a pattern set $\mathcal{X}$ (containing K clusters) is

$$e^2(\mathcal{X}, \mathcal{L}) = \sum_{j=1}^{K} \sum_{i=1}^{n_j} \|\mathbf{x}_i^{(j)} - \mathbf{c}_j\|^2,$$

where $\mathbf{x}_i^{(j)}$ is the i^{th} pattern belonging to the j^{th} cluster and $\mathbf{c}_j$ is the centroid of the j^{th} cluster.

The k-means is the simplest and most commonly used algorithm employing a squared error criterion [McQueen 1967]. It starts with a random initial partition and keeps reassigning the patterns to clusters based on the similarity between the pattern and the cluster centers until a convergence criterion is met (e.g., there is no reassignment of any pattern from one cluster to another, or the squared error ceases to decrease significantly after some number of iterations). The k-means algorithm is popular because it is easy to implement, and its time complexity is $O(n)$, where n is the number of patterns. A major problem with this algorithm is that it is sensitive to the selection of the initial partition and may converge to a local minimum of the criterion function value if the initial partition is not properly chosen. Figure 14 shows seven two-dimensional patterns. If we start with patterns A, B, and C as the initial means around which the three clusters are built, then we end up with the partition {{A}, {B, C}, {D, E, F, G}} shown by ellipses. The squared error criterion value is much larger for this partition than for the best partition {{A, B, C}, {D, E}, {F, G}} shown by rectangles, which yields the global minimum value of the squared error criterion function for a clustering containing three clusters. The correct three-cluster solution is obtained by choosing, for example, A, D, and F as the initial cluster means.

Squared Error Clustering Method

(1) Select an initial partition of the patterns with a fixed number of clusters and cluster centers.

(2) Assign each pattern to its closest cluster center and compute the new cluster centers as the centroids of the clusters. Repeat this step until convergence is achieved, i.e., until the cluster membership is stable.

(3) Merge and split clusters based on some heuristic information, optionally repeating step 2.

k-Means Clustering Algorithm

(1) Choose k cluster centers to coincide with k randomly-chosen patterns or k randomly defined points inside the hypervolume containing the pattern set.

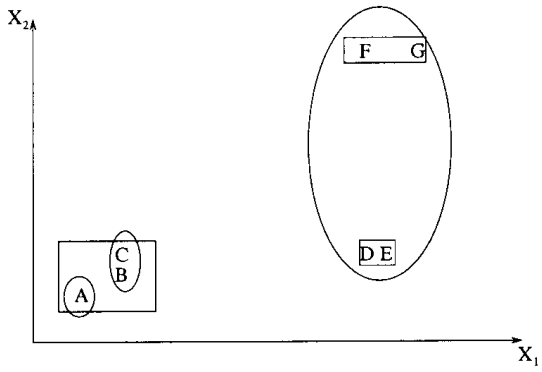


Figure 14. The k-means algorithm is sensitive to the initial partition.

(2) Assign each pattern to the closest cluster center.

(3) Recompute the cluster centers using the current cluster memberships.

(4) If a convergence criterion is not met, go to step 2. Typical convergence criteria are: no (or minimal) reassignment of patterns to new cluster centers, or minimal decrease in squared error.

Several variants [Anderberg 1973] of the k-means algorithm have been reported in the literature. Some of them attempt to select a good initial partition so that the algorithm is more likely to find the global minimum value.

Another variation is to permit splitting and merging of the resulting clusters. Typically, a cluster is split when its variance is above a pre-specified threshold, and two clusters are merged when the distance between their centroids is below another pre-specified threshold. Using this variant, it is possible to obtain the optimal partition starting from any arbitrary initial partition, provided proper threshold values are specified. The well-known ISODATA [Ball and Hall 1965] algorithm employs this technique of merging and splitting clusters. If ISODATA is given the "ellipse" partitioning shown in Figure 14 as an initial partitioning, it will produce the optimal three-cluster partitioning. ISODATA will first merge the clusters {A} and {B,C} into one cluster because the distance between their centroids is small and then split the cluster {D,E,F,G}, which has a large variance, into two clusters {D,E} and {F,G}.

Another variation of the k-means algorithm involves selecting a different criterion function altogether. The *dynamic clustering* algorithm (which permits representations other than the centroid for each cluster) was proposed in Diday [1973], and Symon [1977] and describes a dynamic clustering approach obtained by formulating the clustering problem in the framework of maximum-likelihood estimation. The regularized Mahalanobis distance was used in Mao and Jain [1996] to obtain hyperellipsoidal clusters.

5.2.2 *Graph-Theoretic Clustering*. The best-known graph-theoretic divisive clustering algorithm is based on construction of the *minimal spanning tree* (MST) of the data [Zahn 1971], and then deleting the MST edges with the largest lengths to generate clusters. Figure 15 depicts the MST obtained from nine two-dimensional points. By breaking the link labeled CD with a length of 6 units (the edge with the maximum Euclidean length), two clusters ({A, B, C} and {D, E, F, G, H, I}) are obtained. The second cluster can be further divided into two clusters by breaking the edge EF, which has a length of 4.5 units.

The hierarchical approaches are also related to graph-theoretic clustering. Single-link clusters are subgraphs of the minimum spanning tree of the data [Gower and Ross 1969] which are also the connected components [Gotlieb and Kumar 1968]. Complete-link clusters are maximal complete subgraphs, and are related to the node colorability of graphs [Backer and Hubert 1976]. The maximal complete subgraph was considered the strictest definition of a cluster in Augustson and Minker [1970] and Raghavan and Yu [1981]. A graph-oriented approach for non-hierarchical structures and overlapping clusters is

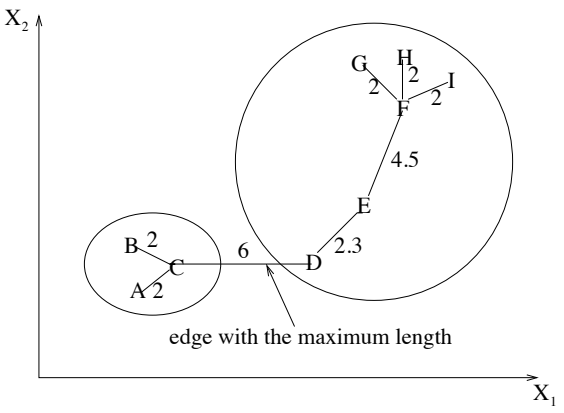


Figure 15. Using the minimal spanning tree to form clusters.

presented in Ozawa [1985]. The *Delaunay graph* (DG) is obtained by connecting all the pairs of points that are Voronoi neighbors. The DG contains all the neighborhood information contained in the MST and the relative neighborhood graph (RNG) [Toussaint 1980].

5.3 Mixture-Resolving and Mode-Seeking Algorithms

The mixture resolving approach to cluster analysis has been addressed in a number of ways. The underlying assumption is that the patterns to be clustered are drawn from one of several distributions, and the goal is to identify the parameters of each and (perhaps) their number. Most of the work in this area has assumed that the individual components of the mixture density are Gaussian, and in this case the parameters of the individual Gaussians are to be estimated by the procedure. Traditional approaches to this problem involve obtaining (iteratively) a maximum likelihood estimate of the parameter vectors of the component densities [Jain and Dubes 1988].

More recently, the Expectation Maximization (EM) algorithm (a general-purpose maximum likelihood algorithm [Dempster et al. 1977] for missing-data problems) has been applied to the problem of parameter estimation. A recent book [Mitchell 1997] provides an accessible description of the technique. In the EM framework, the parameters of the component densities are unknown, as are the mixing parameters, and these are estimated from the patterns. The EM procedure begins with an initial estimate of the parameter vector and iteratively rescores the patterns against the mixture density produced by the parameter vector. The rescored patterns are then used to update the parameter estimates. In a clustering context, the scores of the patterns (which essentially measure their likelihood of being drawn from particular components of the mixture) can be viewed as hints at the class of the pattern. Those patterns, placed (by their scores) in a particular component, would therefore be viewed as belonging to the same cluster.

Nonparametric techniques for density-based clustering have also been developed [Jain and Dubes 1988]. Inspired by the Parzen window approach to nonparametric density estimation, the corresponding clustering procedure searches for bins with large counts in a multidimensional histogram of the input pattern set. Other approaches include the application of another partitional or hierarchical clustering algorithm using a distance measure based on a nonparametric density estimate.

5.4 Nearest Neighbor Clustering

Since proximity plays a key role in our intuitive notion of a cluster, nearest-neighbor distances can serve as the basis of clustering procedures. An iterative procedure was proposed in Lu and Fu [1978]; it assigns each unlabeled pattern to the cluster of its nearest labeled neighbor pattern, provided the distance to that labeled neighbor is below a threshold. The process continues until all patterns are labeled or no additional labelings occur. The mutual neighborhood value (described earlier in the context of distance computation) can also be used to grow clusters from near neighbors.

5.5 Fuzzy Clustering

Traditional clustering approaches generate partitions; in a partition, each pattern belongs to one and only one cluster. Hence, the clusters in a hard clustering are disjoint. Fuzzy clustering extends this notion to associate each pattern with every cluster using a membership function [Zadeh 1965]. The output of such algorithms is a clustering, but not a partition. We give a high-level partitional fuzzy clustering algorithm below.

Fuzzy Clustering Algorithm

(1) Select an initial fuzzy partition of the N objects into K clusters by selecting the $N \times K$ membership matrix U. An element u_{ij} of this matrix represents the grade of membership of object $\mathbf{x}_i$ in cluster $\mathbf{c}_j$. Typically, $u_{ij} \in [0,1]$.

(2) Using U, find the value of a fuzzy criterion function, e.g., a weighted squared error criterion function, associated with the corresponding partition. One possible fuzzy criterion function is

$$E^2(\mathscr{X}, \mathbf{U}) = \sum_{i=1}^{N} \sum_{k=1}^{K} u_{ij} \|\mathbf{x}_i - \mathbf{c}_k\|^2,$$

where $\mathbf{c}_k = \sum_{i=1}^{N} u_{ik}\mathbf{x}_i$ is the k^{th} fuzzy cluster center.

Reassign patterns to clusters to reduce this criterion function value and recompute U.

(3) Repeat step 2 until entries in U do not change significantly.

In fuzzy clustering, each cluster is a fuzzy set of all the patterns. Figure 16 illustrates the idea. The rectangles enclose two "hard" clusters in the data: $H_1 = \{1,2,3,4,5\}$ and $H_2 = \{6,7,8,9\}$. A fuzzy clustering algorithm might produce the two fuzzy clusters F_1 and F_2 depicted by ellipses. The patterns will have membership values in [0,1] for each cluster. For example, fuzzy cluster F_1 could be compactly described as

{(1,0.9), (2,0.8), (3,0.7), (4,0.6), (5,0.55),
(6,0.2), (7,0.2), (8,0.0), (9,0.0)}

and F_2 could be described as

{(1,0.0), (2,0.0), (3,0.0), (4,0.1), (5,0.15),
(6,0.4), (7,0.35), (8,1.0), (9,0.9)}

The ordered pairs (i, μ_i) in each cluster represent the ith pattern and its membership value to the cluster μ_i. Larger membership values indicate higher confidence in the assignment of the pattern to the cluster. A hard clustering can be obtained from a fuzzy partition by thresholding the membership value.

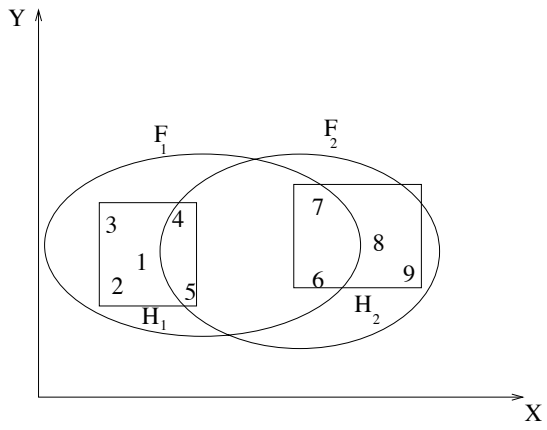


Figure 16. Fuzzy clusters.

Fuzzy set theory was initially applied to clustering in Ruspini [1969]. The book by Bezdek [1981] is a good source for material on fuzzy clustering. The most popular fuzzy clustering algorithm is the fuzzy c-means (FCM) algorithm. Even though it is better than the hard k-means algorithm at avoiding local minima, FCM can still converge to local minima of the squared error criterion. The design of membership functions is the most important problem in fuzzy clustering; different choices include

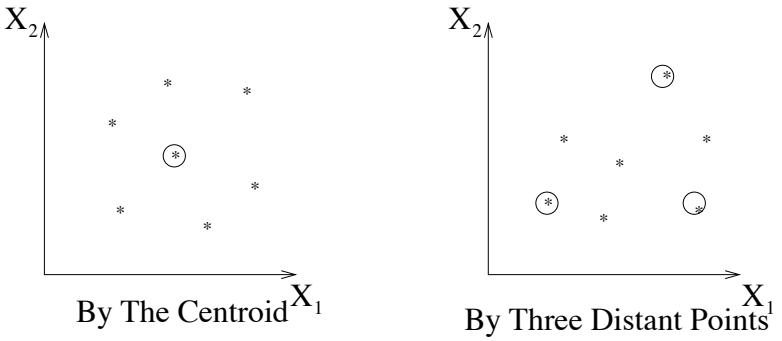


Figure 17. Representation of a cluster by points.

those based on similarity decomposition and centroids of clusters. A generalization of the FCM algorithm was proposed by Bezdek [1981] through a family of objective functions. A fuzzy c-shell algorithm and an adaptive variant for detecting circular and elliptical boundaries was presented in Dave [1992].

5.6 Representation of Clusters

In applications where the number of classes or clusters in a data set must be discovered, a partition of the data set is the end product. Here, a partition gives an idea about the separability of the data points into clusters and whether it is meaningful to employ a supervised classifier that assumes a given number of classes in the data set. However, in many other applications that involve decision making, the resulting clusters have to be represented or described in a compact form to achieve *data abstraction*. Even though the construction of a cluster representation is an important step in decision making, it has not been examined closely by researchers. The notion of cluster representation was introduced in Duran and Odell [1974] and was subsequently studied in Diday and Simon [1976] and Michalski et al. [1981]. They suggested the following representation schemes:

(1) Represent a cluster of points by their centroid or by a set of distant points in the cluster. Figure 17 depicts these two ideas.

(2) Represent clusters using nodes in a classification tree. This is illustrated in Figure 18.

(3) Represent clusters by using conjunctive logical expressions. For example, the expression $[X_1 > 3][X_2 < 2]$ in Figure 18 stands for the logical statement 'X_1 is greater than 3' *and* 'X_2 is less than 2'.

Use of the centroid to represent a cluster is the most popular scheme. It works well when the clusters are compact or isotropic. However, when the clusters are elongated or non-isotropic, then this scheme fails to represent them properly. In such a case, the use of a collection of boundary points in a cluster captures its shape well. The number of points used to represent a cluster should increase as the complexity of its shape increases. The two different representations illustrated in Figure 18 are equivalent. Every path in a classification tree from the root node to a leaf node corresponds to a conjunctive statement. An important limitation of the typical use of the simple conjunctive concept representations is that they can describe only rectangular or isotropic clusters in the feature space.

Data abstraction is useful in decision making because of the following:

(1) It gives a simple and intuitive description of clusters which is easy for human comprehension. In both conceptual clustering [Michalski

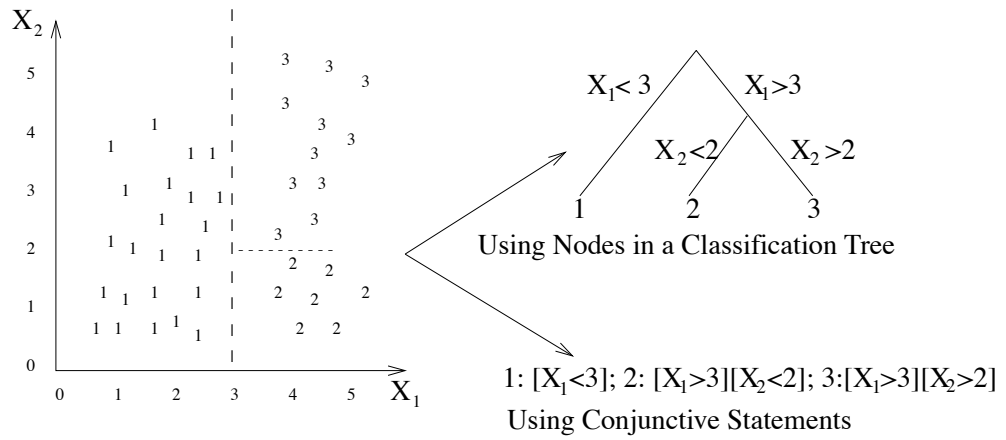


Figure 18. Representation of clusters by a classification tree or by conjunctive statements.

and Stepp 1983] and symbolic clustering [Gowda and Diday 1992] this representation is obtained without using an additional step. These algorithms generate the clusters as well as their descriptions. A set of fuzzy rules can be obtained from fuzzy clusters of a data set. These rules can be used to build fuzzy classifiers and fuzzy controllers.

(2) It helps in achieving data compression that can be exploited further by a computer [Murty and Krishna 1980]. Figure 19(a) shows samples belonging to two chain-like clusters labeled 1 and 2. A partitional clustering like the k-means algorithm cannot separate these two structures properly. The single-link algorithm works well on this data, but is computationally expensive. So a hybrid approach may be used to exploit the desirable properties of both these algorithms. We obtain 8 subclusters of the data using the (computationally efficient) k-means algorithm. Each of these subclusters can be represented by their centroids as shown in Figure 19(a). Now the single-link algorithm can be applied on these centroids alone to cluster them into 2 groups. The resulting groups are shown in Figure 19(b). Here, a data reduction is achieved by representing the subclusters by their centroids.

(3) It increases the efficiency of the decision making task. In a cluster-based document retrieval technique [Salton 1991], a large collection of documents is clustered and each of the clusters is represented using its centroid. In order to retrieve documents relevant to a query, the query is matched with the cluster centroids rather than with all the documents. This helps in retrieving relevant documents efficiently. Also in several applications involving large data sets, clustering is used to perform indexing, which helps in efficient decision making [Dorai and Jain 1995].

5.7 Artificial Neural Networks for Clustering

Artificial neural networks (ANNs) [Hertz et al. 1991] are motivated by biological neural networks. ANNs have been used extensively over the past three decades for both classification and clustering [Sethi and Jain 1991; Jain and Mao 1994]. Some of the features of the ANNs that are important in pattern clustering are:

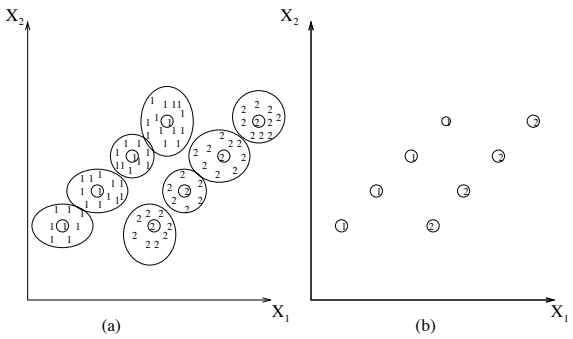


Figure 19. Data compression by clustering.

(1) ANNs process numerical vectors and so require patterns to be represented using quantitative features only.

(2) ANNs are inherently parallel and distributed processing architectures.

(3) ANNs may learn their interconnection weights adaptively [Jain and Mao 1996; Oja 1982]. More specifically, they can act as pattern normalizers and feature selectors by appropriate selection of weights.

Competitive (or winner–take–all) neural networks [Jain and Mao 1996] are often used to cluster input data. In competitive learning, similar patterns are grouped by the network and represented by a single unit (neuron). This grouping is done automatically based on data correlations. Well-known examples of ANNs used for clustering include Kohonen's learning vector quantization (LVQ) and self-organizing map (SOM) [Kohonen 1984], and adaptive resonance theory models [Carpenter and Grossberg 1990]. The architectures of these ANNs are simple: they are single-layered. Patterns are presented at the input and are associated with the output nodes. The weights between the input nodes and the output nodes are iteratively changed (this is called learning) until a termination criterion is satisfied. Competitive learning has been found to exist in biological neural networks. However, the learning or weight update procedures are quite similar to those in some classical clustering approaches. For example, the relationship between the k-means algorithm and LVQ is addressed in Pal et al. [1993]. The learning algorithm in ART models is similar to the leader clustering algorithm [Moor 1988].

The SOM gives an intuitively appealing two-dimensional map of the multidimensional data set, and it has been successfully used for vector quantization and speech recognition [Kohonen 1984]. However, like its sequential counterpart, the SOM generates a suboptimal partition if the initial weights are not chosen properly. Further, its convergence is controlled by various parameters such as the learning rate and a neighborhood of the winning node in which learning takes place. It is possible that a particular input pattern can fire different output units at different iterations; this brings up the *stability* issue of learning systems. The system is said to be stable if no pattern in the training data changes its category after a finite number of learning iterations. This problem is closely associated with the problem of *plasticity*, which is the ability of the algorithm to adapt to new data. For stability, the learning rate should be decreased to zero as iterations progress and this affects the plasticity. The ART models are supposed to be stable and plastic [Carpenter and Grossberg 1990]. However, ART nets are order-dependent; that is, different partitions are obtained for different orders in which the data is presented to the net. Also, the size and number of clusters generated by an ART net depend on the value chosen for the *vigilance threshold*, which is used to decide whether a pattern is to be assigned to one of the existing clusters or start a new cluster. Further, both SOM and ART are suitable for detecting only hyperspherical clusters [Hertz et al. 1991]. A two-layer network that employs regularized Mahalanobis distance to extract hyperellipsoidal clusters was proposed in Mao and Jain [1994]. All these ANNs use a fixed number of output nodes

which limit the number of clusters that can be produced.

5.8 Evolutionary Approaches for Clustering

Evolutionary approaches, motivated by natural evolution, make use of evolutionary operators and a population of solutions to obtain the globally optimal partition of the data. Candidate solutions to the clustering problem are encoded as chromosomes. The most commonly used evolutionary operators are: selection, recombination, and mutation. Each transforms one or more input chromosomes into one or more output chromosomes. A fitness function evaluated on a chromosome determines a chromosome's likelihood of surviving into the next generation. We give below a high-level description of an evolutionary algorithm applied to clustering.

An Evolutionary Algorithm for Clustering

(1) Choose a random population of solutions. Each solution here corresponds to a valid k-partition of the data. Associate a fitness value with each solution. Typically, fitness is inversely proportional to the squared error value. A solution with a small squared error will have a larger fitness value.

(2) Use the evolutionary operators selection, recombination and mutation to generate the next population of solutions. Evaluate the fitness values of these solutions.

(3) Repeat step 2 until some termination condition is satisfied.

The best-known evolutionary techniques are genetic algorithms (GAs) [Holland 1975; Goldberg 1989], evolution strategies (ESs) [Schwefel 1981], and evolutionary programming (EP) [Fogel et al. 1965]. Out of these three approaches, GAs have been most frequently used in clustering. Typically, solutions are binary strings in GAs. In

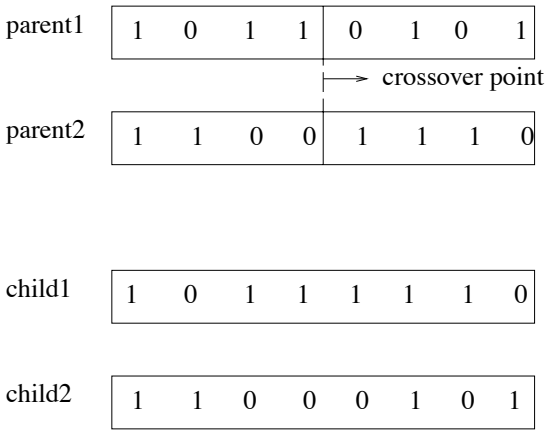


Figure 20. Crossover operation.

GAs, a selection operator propagates solutions from the current generation to the next generation based on their fitness. Selection employs a probabilistic scheme so that solutions with higher fitness have a higher probability of getting reproduced.

There are a variety of recombination operators in use; *crossover* is the most popular. Crossover takes as input a pair of chromosomes (called parents) and outputs a new pair of chromosomes (called children or offspring) as depicted in Figure 20. In Figure 20, a single point crossover operation is depicted. It exchanges the segments of the parents across a crossover point. For example, in Figure 20, the parents are the binary strings '10110101' and '11001110'. The segments in the two parents after the crossover point (between the fourth and fifth locations) are exchanged to produce the child chromosomes. *Mutation* takes as input a chromosome and outputs a chromosome by complementing the bit value at a randomly selected location in the input chromosome. For example, the string '11111110' is generated by applying the mutation operator to the second bit location in the string '10111110' (starting at the left). Both crossover and mutation are applied with some prespecified probabilities which depend on the fitness values.

GAs represent points in the search space as binary strings, and rely on the

crossover operator to explore the search space. Mutation is used in GAs for the sake of completeness, that is, to make sure that no part of the search space is left unexplored. ESs and EP differ from the GAs in solution representation and type of the mutation operator used; EP does not use a recombination operator, but only selection and mutation. Each of these three approaches have been used to solve the clustering problem by viewing it as a minimization of the squared error criterion. Some of the theoretical issues such as the convergence of these approaches were studied in Fogel and Fogel [1994].

GAs perform a globalized search for solutions whereas most other clustering procedures perform a localized search. In a localized search, the solution obtained at the 'next iteration' of the procedure is in the vicinity of the current solution. In this sense, the k-means algorithm, fuzzy clustering algorithms, ANNs used for clustering, various annealing schemes (see below), and tabu search are all localized search techniques. In the case of GAs, the crossover and mutation operators can produce new solutions that are completely different from the current ones. We illustrate this fact in Figure 21. Let us assume that the scalar X is coded using a 5-bit binary representation, and let S_1 and S_2 be two points in the one-dimensional search space. The decimal values of S_1 and S_2 are 8 and 31, respectively. Their binary representations are S_1 = 01000 and S_2 = 11111. Let us apply the single-point crossover to these strings, with the crossover site falling between the second and third most significant bits as shown below.

01!000

11!111

This will produce a new pair of points or chromosomes S_3 and S_4 as shown in Figure 21. Here, S_3 = 01111 and S_4 = 11000. The corresponding decimal values are 15 and 24, respectively. Similarly, by mutating the most significant bit in the binary string 01111 (decimal 15), the binary string 11111 (decimal 31) is generated. These jumps, or gaps between points in successive generations, are much larger than those produced by other approaches.

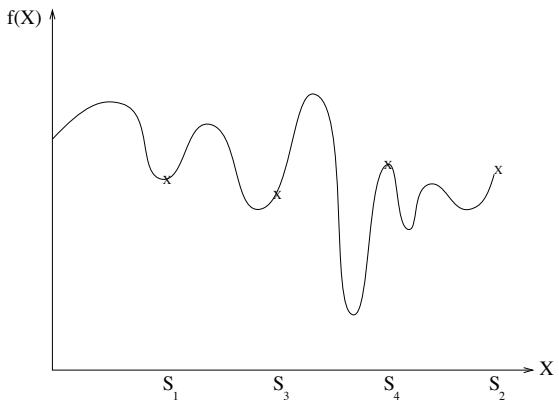


Figure 21. GAs perform globalized search.

Perhaps the earliest paper on the use of GAs for clustering is by Raghavan and Birchand [1979], where a GA was used to minimize the squared error of a clustering. Here, each point or chromosome represents a partition of N objects into K clusters and is represented by a K-ary string of length N. For example, consider six patterns—A, B, C, D, E, and F—and the string 101001. This six-bit binary (K = 2) string corresponds to placing the six patterns into two clusters. This string represents a two-partition, where one cluster has the first, third, and sixth patterns and the second cluster has the remaining patterns. In other words, the two clusters are {A,C,F} and {B,D,E} (the six-bit binary string 010110 represents the same clustering of the six patterns). When there are K clusters, there are K! different chromosomes corresponding to each K-partition of the data. This increases the effective search space size by a factor of K!. Further, if crossover is applied on two good chromosomes, the resulting

offspring may be inferior in this representation. For example, let {A,B,C} and {D,E,F} be the clusters in the optimal 2-partition of the six patterns considered above. The corresponding chromosomes are 111000 and 000111. By applying single-point crossover at the location between the third and fourth bit positions on these two strings, we get 111111 and 000000 as offspring and both correspond to an inferior partition. These problems have motivated researchers to design better representation schemes and crossover operators.

In Bhuyan et al. [1991], an improved representation scheme is proposed where an additional separator symbol is used along with the pattern labels to represent a partition. Let the separator symbol be represented by *. Then the chromosome ACF*BDE corresponds to a 2-partition {A,C,F} and {B,D,E}. Using this representation permits them to map the clustering problem into a permutation problem such as the traveling salesman problem, which can be solved by using the permutation crossover operators [Goldberg 1989]. This solution also suffers from permutation redundancy. There are 72 equivalent chromosomes (permutations) corresponding to the same partition of the data into the two clusters {A,C,F} and {B,D,E}.

More recently, Jones and Beltramo [1991] investigated the use of edge-based crossover [Whitley et al. 1989] to solve the clustering problem. Here, all patterns in a cluster are assumed to form a complete graph by connecting them with edges. Offspring are generated from the parents so that they inherit the edges from their parents. It is observed that this crossover operator takes $O(K^6 + N)$ time for N patterns and K clusters ruling out its applicability on practical data sets having more than 10 clusters. In a hybrid approach proposed in Babu and Murty [1993], the GA is used only to find good initial cluster centers and the k-means algorithm is applied to find the final partition. This hybrid approach performed better than the GA.

A major problem with GAs is their sensitivity to the selection of various parameters such as population size, crossover and mutation probabilities, *etc.* Grefenstette [Grefenstette 1986] has studied this problem and suggested guidelines for selecting these control parameters. However, these guidelines may not yield good results on specific problems like pattern clustering. It was reported in Jones and Beltramo [1991] that hybrid genetic algorithms incorporating problem-specific heuristics are good for clustering. A similar claim is made in Davis [1991] about the applicability of GAs to other practical problems. Another issue with GAs is the selection of an appropriate representation which is low in order and short in defining length.

It is possible to view the clustering problem as an optimization problem that locates the optimal centroids of the clusters directly rather than finding an optimal partition using a GA. This view permits the use of ESs and EP, because centroids can be coded easily in both these approaches, as they support the direct representation of a solution as a real-valued vector. In Babu and Murty [1994], ESs were used on both hard and fuzzy clustering problems and EP has been used to evolve fuzzy min-max clusters [Fogel and Simpson 1993]. It has been observed that they perform better than their classical counterparts, the k-means algorithm and the fuzzy c-means algorithm. However, all of these approaches suffer (as do GAs and ANNs) from sensitivity to control parameter selection. For each specific problem, one has to tune the parameter values to suit the application.

5.9 Search-Based Approaches

Search techniques used to obtain the optimum value of the criterion function are divided into deterministic and stochastic search techniques. Determinis-

tic search techniques guarantee an optimal partition by performing exhaustive enumeration. On the other hand, the stochastic search techniques generate a near-optimal partition reasonably quickly, and guarantee convergence to optimal partition asymptotically. Among the techniques considered so far, evolutionary approaches are stochastic and the remainder are deterministic. Other deterministic approaches to clustering include the branch-and-bound technique adopted in Koontz et al. [1975] and Cheng [1995] for generating optimal partitions. This approach generates the optimal partition of the data at the cost of excessive computational requirements. In Rose et al. [1993], a deterministic annealing approach was proposed for clustering. This approach employs an annealing technique in which the error surface is smoothed, but convergence to the global optimum is not guaranteed. The use of deterministic annealing in proximity-mode clustering (where the patterns are specified in terms of pairwise proximities rather than multidimensional points) was explored in Hofmann and Buhmann [1997]; later work applied the deterministic annealing approach to texture segmentation [Hofmann and Buhmann 1998].

The deterministic approaches are typically greedy descent approaches, whereas the stochastic approaches permit perturbations to the solutions in non-locally optimal directions also with nonzero probabilities. The stochastic search techniques are either sequential or parallel, while evolutionary approaches are inherently parallel. The simulated annealing approach (SA) [Kirkpatrick et al. 1983] is a sequential stochastic search technique, whose applicability to clustering is discussed in Klein and Dubes [1989]. Simulated annealing procedures are designed to avoid (or recover from) solutions which correspond to local optima of the objective functions. This is accomplished by accepting with some probability a new solution for the next iteration of lower quality (as measured by the criterion function). The probability of acceptance is governed by a critical parameter called the temperature (by analogy with annealing in metals), which is typically specified in terms of a starting (first iteration) and final temperature value. Selim and Al-Sultan [1991] studied the effects of control parameters on the performance of the algorithm, and Baeza-Yates [1992] used SA to obtain near-optimal partition of the data. SA is statistically guaranteed to find the global optimal solution [Aarts and Korst 1989]. A high-level outline of a SA based algorithm for clustering is given below.

Clustering Based on Simulated Annealing

(1) Randomly select an initial partition and P_0, and compute the squared error value, E_{P_0}. Select values for the control parameters, initial and final temperatures T_0 and T_f.

(2) Select a neighbor P_1 of P_0 and compute its squared error value, E_{P_1}. If E_{P_1} is larger than E_{P_0}, then assign P_1 to P_0 with a temperature-dependent probability. Else assign P_1 to P_0. Repeat this step for a fixed number of iterations.

(3) Reduce the value of T_0, *i.e.* $T_0 = cT_0$, where c is a predetermined constant. If T_0 is greater than T_f, then go to step 2. Else stop.

The SA algorithm can be slow in reaching the optimal solution, because optimal results require the temperature to be decreased very slowly from iteration to iteration.

Tabu search [Glover 1986], like SA, is a method designed to cross boundaries of feasibility or local optimality and to systematically impose and release constraints to permit exploration of otherwise forbidden regions. Tabu search was used to solve the clustering problem in Al-Sultan [1995].

5.10 A Comparison of Techniques

In this section we have examined various deterministic and stochastic search techniques to approach the clustering problem as an optimization problem. A majority of these methods use the squared error criterion function. Hence, the partitions generated by these approaches are not as versatile as those generated by hierarchical algorithms. The clusters generated are typically hyperspherical in shape. Evolutionary approaches are globalized search techniques, whereas the rest of the approaches are localized search technique. ANNs and GAs are inherently parallel, so they can be implemented using parallel hardware to improve their speed. Evolutionary approaches are population-based; that is, they search using more than one solution at a time, and the rest are based on using a single solution at a time. ANNs, GAs, SA, and Tabu search (TS) are all sensitive to the selection of various learning/control parameters. In theory, all four of these methods are weak methods [Rich 1983] in that they do not use explicit domain knowledge. An important feature of the evolutionary approaches is that they can find the optimal solution even when the criterion function is discontinuous.

An empirical study of the performance of the following heuristics for clustering was presented in Mishra and Raghavan [1994]; SA, GA, TS, randomized branch-and-bound (RBA) [Mishra and Raghavan 1994], and hybrid search (HS) strategies [Ismail and Kamel 1989] were evaluated. The conclusion was that GA performs well in the case of one-dimensional data, while its performance on high dimensional data sets is not impressive. The performance of SA is not attractive because it is very slow. RBA and TS performed best. HS is good for high dimensional data. However, none of the methods was found to be superior to others by a significant margin. An empirical study of k-means, SA, TS, and GA was presented in Al-Sultan and Khan [1996]. TS, GA and SA were judged comparable in terms of solution quality, and all were better than k-means. However, the k-means method is the most efficient in terms of execution time; other schemes took more time (by a factor of 500 to 2500) to partition a data set of size 60 into 5 clusters. Further, GA encountered the best solution faster than TS and SA; SA took more time than TS to encounter the best solution. However, GA took the maximum time for convergence, that is, to obtain a population of only the best solutions, followed by TS and SA. An important observation is that in both Mishra and Raghavan [1994] and Al-Sultan and Khan [1996] the sizes of the data sets considered are small; that is, fewer than 200 patterns.

A two-layer network was employed in Mao and Jain [1996], with the first layer including a number of principal component analysis subnets, and the second layer using a competitive net. This network performs partitional clustering using the regularized Mahalanobis distance. This net was trained using a set of 1000 randomly selected pixels from a large image and then used to classify every pixel in the image. Babu et al. [1997] proposed a stochastic connectionist approach (SCA) and compared its performance on standard data sets with both the SA and k-means algorithms. It was observed that SCA is superior to both SA and k-means in terms of solution quality. Evolutionary approaches are good only when the data size is less than 1000 and for low dimensional data.

In summary, only the k-means algorithm and its ANN equivalent, the Kohonen net [Mao and Jain 1996] have been applied on large data sets; other approaches have been tested, typically, on small data sets. This is because obtaining suitable learning/control parameters for ANNs, GAs, TS, and SA is difficult and their execution times are very high for large data sets. However, it has been shown [Selim and Ismail

1984] that the k-means method converges to a locally optimal solution. This behavior is linked with the initial seed selection in the k-means algorithm. So if a good initial partition can be obtained quickly using any of the other techniques, then k-means would work well even on problems with large data sets. Even though various methods discussed in this section are comparatively weak, it was revealed through experimental studies that combining domain knowledge would improve their performance. For example, ANNs work better in classifying images represented using extracted features than with raw images, and hybrid classifiers work better than ANNs [Mohiuddin and Mao 1994]. Similarly, using domain knowledge to hybridize a GA improves its performance [Jones and Beltramo 1991]. So it may be useful in general to use domain knowledge along with approaches like GA, SA, ANN, and TS. However, these approaches (specifically, the criteria functions used in them) have a tendency to generate a partition of hyperspherical clusters, and this could be a limitation. For example, in cluster-based document retrieval, it was observed that the hierarchical algorithms performed better than the partitional algorithms [Rasmussen 1992].

5.11 Incorporating Domain Constraints in Clustering

As a task, clustering is subjective in nature. The same data set may need to be partitioned differently for different purposes. For example, consider a *whale*, an *elephant*, and a *tuna fish* [Watanabe 1985]. Whales and elephants form a cluster of *mammals*. However, if the user is interested in partitioning them based on the concept of *living in water*, then whale and tuna fish are clustered together. Typically, this subjectivity is incorporated into the clustering criterion by incorporating domain knowledge in one or more phases of clustering.

Every clustering algorithm uses some type of knowledge either implicitly or explicitly. Implicit knowledge plays a role in (1) selecting a pattern representation scheme (e.g., using one's prior experience to select and encode features), (2) choosing a similarity measure (e.g., using the Mahalanobis distance instead of the Euclidean distance to obtain hyperellipsoidal clusters), and (3) selecting a grouping scheme (e.g., specifying the k-means algorithm when it is known that clusters are hyperspherical). Domain knowledge is used implicitly in ANNs, GAs, TS, and SA to select the control/learning parameter values that affect the performance of these algorithms.

It is also possible to use explicitly available domain knowledge to constrain or guide the clustering process. Such specialized clustering algorithms have been used in several applications. Domain concepts can play several roles in the clustering process, and a variety of choices are available to the practitioner. At one extreme, the available domain concepts might easily serve as an additional feature (or several), and the remainder of the procedure might be otherwise unaffected. At the other extreme, domain concepts might be used to confirm or veto a decision arrived at independently by a traditional clustering algorithm, or used to affect the computation of distance in a clustering algorithm employing proximity. The incorporation of domain knowledge into clustering consists mainly of ad hoc approaches with little in common; accordingly, our discussion of the idea will consist mainly of motivational material and a brief survey of past work. Machine learning research and pattern recognition research intersect in this topical area, and the interested reader is referred to the prominent journals in machine learning (e.g., *Machine Learning*, *J. of AI Research*, or *Artificial Intelligence*) for a fuller treatment of this topic.

As documented in Cheng and Fu [1985], rules in an expert system may be clustered to reduce the size of the knowledge base. This modification of clustering was also explored in the domains of universities, congressional voting records, and terrorist events by Lebowitz [1987].

5.11.1 *Similarity Computation*. Conceptual knowledge was used explicitly in the similarity computation phase in Michalski and Stepp [1983]. It was assumed that the pattern representations were available and the dynamic clustering algorithm [Diday 1973] was used to group patterns. The clusters formed were described using conjunctive statements in predicate logic. It was stated in Stepp and Michalski [1986] and Michalski and Stepp [1983] that the groupings obtained by the conceptual clustering are superior to those obtained by the numerical methods for clustering. A critical analysis of that work appears in Dale [1985], and it was observed that monothetic divisive clustering algorithms generate clusters that can be described by conjunctive statements. For example, consider Figure 8. Four clusters in this figure, obtained using a monothetic algorithm, can be described by using conjunctive concepts as shown below:

Cluster 1: $[X \leq a] \wedge [Y \leq b]$

Cluster 2: $[X \leq a] \wedge [Y > b]$

Cluster 3: $[X > a] \wedge [Y > c]$

Cluster 4: $[X > a] \wedge [Y \leq c]$

where $\wedge$ is the Boolean conjunction ('and') operator, and a, b, and c are constants.

5.11.2 *Pattern Representation*. It was shown in Srivastava and Murty [1990] that by using knowledge in the pattern representation phase, as is implicitly done in numerical taxonomy approaches, it is possible to obtain the same partitions as those generated by conceptual clustering. In this sense, conceptual clustering and numerical taxonomy are not diametrically opposite, but are equivalent. In the case of conceptual clustering, domain knowledge is explicitly used in interpattern similarity computation, whereas in numerical taxonomy it is implicitly assumed that pattern representations are obtained using the domain knowledge.

5.11.3 *Cluster Descriptions*. Typically, in knowledge-based clustering, both the clusters and their descriptions or characterizations are generated [Fisher and Langley 1985]. There are some exceptions, for instance,, Gowda and Diday [1992], where only clustering is performed and no descriptions are generated explicitly. In conceptual clustering, a cluster of objects is described by a conjunctive logical expression [Michalski and Stepp 1983]. Even though a conjunctive statement is one of the most common descriptive forms used by humans, it is a limited form. In Shekar et al. [1987], functional knowledge of objects was used to generate more intuitively appealing cluster descriptions that employ the Boolean *implication* operator. A system that represents clusters probabilistically was described in Fisher [1987]; these descriptions are more general than conjunctive concepts, and are well-suited to hierarchical classification domains (e.g., the animal species hierarchy). A conceptual clustering system in which clustering is done first is described in Fisher and Langley [1985]. These clusters are then described using probabilities. A similar scheme was described in Murty and Jain [1995], but the descriptions are logical expressions that employ both conjunction and disjunction.

An important characteristic of conceptual clustering is that it is possible to group objects represented by both qualitative and quantitative features if the clustering leads to a conjunctive concept. For example, the concept *cricket ball* might be represented as

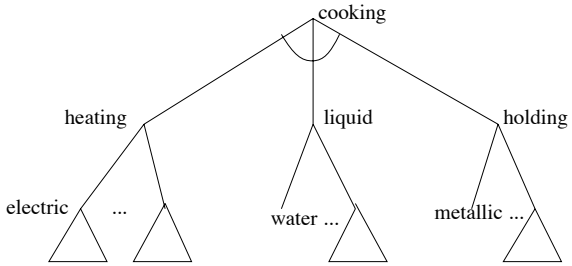


Figure 22. Functional knowledge.

$$\text{color} = \text{red} \wedge (\text{shape} = \text{sphere}) \wedge (\text{make} = \text{leather}) \wedge (\text{radius} = 1.4 \text{ inches}),$$

where radius is a quantitative feature and the rest are all qualitative features. This description is used to describe a cluster of cricket balls. In Stepp and Michalski [1986], a graph (the goal dependency network) was used to group structured objects. In Shekar et al. [1987] functional knowledge was used to group man-made objects. Functional knowledge was represented using and/or trees [Rich 1983]. For example, the function *cooking* shown in Figure 22 can be decomposed into functions like *holding* and *heating* the material in a *liquid* medium. Each man-made object has a primary function for which it is produced. Further, based on its features, it may serve additional functions. For example, a book is meant for *reading*, but if it is heavy then it can also be used as a *paper weight*. In Sutton et al. [1993], object functions were used to construct generic recognition systems.

5.11.4 *Pragmatic Issues*. Any implementation of a system that explicitly incorporates domain concepts into a clustering technique has to address the following important pragmatic issues:

(1) Representation, availability and completeness of domain concepts.

(2) Construction of inferences using the knowledge.

(3) Accommodation of changing or dynamic knowledge.

In some domains, complete knowledge is available explicitly. For example, the *ACM Computing Reviews* classification tree used in Murty and Jain [1995] is complete and is explicitly available for use. In several domains, knowledge is incomplete and is not available explicitly. Typically, machine learning techniques are used to automatically extract knowledge, which is a difficult and challenging problem. The most prominently used learning method is "learning from examples" [Quinlan 1990]. This is an inductive learning scheme used to acquire knowledge from examples of each of the classes in different domains. Even if the knowledge is available explicitly, it is difficult to find out whether it is complete and sound. Further, it is extremely difficult to verify soundness and completeness of knowledge extracted from practical data sets, because such knowledge cannot be represented in propositional logic. It is possible that both the data and knowledge keep changing with time. For example, in a library, new books might get added and some old books might be deleted from the collection with time. Also, the classification system (knowledge) employed by the library is updated periodically.

A major problem with knowledge-based clustering is that it has not been applied to large data sets or in domains with large knowledge bases. Typically, the number of objects grouped was less than 1000, and number of rules used as a part of the knowledge was less than 100. The most difficult problem is to use a very large knowledge base for clustering objects in several practical problems including data mining, image segmentation, and document retrieval.

5.12 Clustering Large Data Sets

There are several applications where it is necessary to cluster a large collection of patterns. The definition of 'large' has varied (and will continue to do so) with changes in technology (e.g., memory and processing time). In the 1960s, 'large'

meant several thousand patterns [Ross 1968]; now, there are applications where millions of patterns of high dimensionality have to be clustered. For example, to segment an image of size 500×500 pixels, the number of pixels to be clustered is 250,000. In document retrieval and information filtering, millions of patterns with a dimensionality of more than 100 have to be clustered to achieve data abstraction. A majority of the approaches and algorithms proposed in the literature cannot handle such large data sets. Approaches based on genetic algorithms, tabu search and simulated annealing are optimization techniques and are restricted to reasonably small data sets. Implementations of conceptual clustering optimize some criterion functions and are typically computationally expensive.

The convergent k-means algorithm and its ANN equivalent, the Kohonen net, have been used to cluster large data sets [Mao and Jain 1996]. The reasons behind the popularity of the k-means algorithm are:

(1) Its time complexity is $O(nkl)$, where n is the number of patterns, k is the number of clusters, and l is the number of iterations taken by the algorithm to converge. Typically, k and l are fixed in advance and so the algorithm has linear time complexity in the size of the data set [Day 1992].

(2) Its space complexity is $O(k + n)$. It requires additional space to store the data matrix. It is possible to store the data matrix in a secondary memory and access each pattern based on need. However, this scheme requires a huge access time because of the iterative nature of the algorithm, and as a consequence processing time increases enormously.

(3) It is order-independent; for a given initial seed set of cluster centers, it generates the same partition of the data irrespective of the order in which the patterns are presented to the algorithm.

However, the k-means algorithm is sensitive to initial seed selection and even in the best case, it can produce only hyperspherical clusters.

Hierarchical algorithms are more versatile. But they have the following disadvantages:

(1) The time complexity of hierarchical agglomerative algorithms is $O(n^2 \log n)$ [Kurita 1991]. It is possible to obtain single-link clusters using an MST of the data, which can be constructed in $O(n \log^2 n)$ time for two-dimensional data [Choudhury and Murty 1990].

(2) The space complexity of agglomerative algorithms is $O(n^2)$. This is because a similarity matrix of size $n \times n$ has to be stored. To cluster every pixel in a 100×100 image, approximately 200 megabytes of storage would be required (assuning single-precision storage of similarities). It is possible to compute the entries of this matrix based on need instead of storing them (this would increase the algorithm's time complexity [Anderberg 1973]).

Table I lists the time and space complexities of several well-known algorithms. Here, n is the number of patterns to be clustered, k is the number of clusters, and l is the number of iterations.

Table I. Complexity of Clustering Algorithms

Clustering Algorithm	Time Complexity	Space Complexity
leader	$O(kn)$	$O(k)$
k-means	$O(nkl)$	$O(k)$
ISODATA	$O(nkl)$	$O(k)$
shortest spanning path	$O(n^2)$	$O(n)$
single-line	$O(n^2 \log n)$	$O(n^2)$
complete-line	$O(n^2 \log n)$	$O(n^2)$

A possible solution to the problem of clustering large data sets while only marginally sacrificing the versatility of clusters is to implement more efficient variants of clustering algorithms. A hybrid approach was used in Ross [1968], where a set of reference points is chosen as in the k-means algorithm, and each of the remaining data points is assigned to one or more reference points or clusters. Minimal spanning trees (MST) are obtained for each group of points separately. These MSTs are merged to form an approximate global MST. This approach computes similarities between only a fraction of all possible pairs of points. It was shown that the number of similarities computed for 10,000 patterns using this approach is the same as the total number of pairs of points in a collection of 2,000 points. Bentley and Friedman [1978] contains an algorithm that can compute an approximate MST in $O(n \log n)$ time. A scheme to generate an approximate dendrogram incrementally in $O(n \log n)$ time was presented in Zupan [1982], while Venkateswarlu and Raju [1992] proposed an algorithm to speed up the ISODATA clustering algorithm. A study of the approximate single-linkage cluster analysis of large data sets was reported in Eddy et al. [1994]. In that work, an approximate MST was used to form single-link clusters of a data set of size 40,000.

The emerging discipline of data mining (discussed as an application in Section 6) has spurred the development of new algorithms for clustering large data sets. Two algorithms of note are the CLARANS algorithm developed by Ng and Han [1994] and the BIRCH algorithm proposed by Zhang et al. [1996]. CLARANS (Clustering Large Applications based on RANdom Search) identifies candidate cluster centroids through analysis of repeated random samples from the original data. Because of the use of random sampling, the time complexity is $O(n)$ for a pattern set of n elements. The BIRCH algorithm (Balanced Iterative Reducing and Clustering) stores summary information about candidate clusters in a dynamic tree data structure. This tree hierarchically organizes the clusterings represented at the leaf nodes. The tree can be rebuilt when a threshold specifying cluster size is updated manually, or when memory constraints force a change in this threshold. This algorithm, like CLARANS, has a time complexity linear in the number of patterns.

The algorithms discussed above work on large data sets, where it is possible to accommodate the entire pattern set in the main memory. However, there are applications where the entire data set cannot be stored in the main memory because of its size. There are currently three possible approaches to solve this problem.

(1) The pattern set can be stored in a secondary memory and subsets of this data clustered independently, followed by a merging step to yield a clustering of the entire pattern set. We call this approach the *divide and conquer* approach.

(2) An incremental clustering algorithm can be employed. Here, the entire data matrix is stored in a secondary memory and data items are transferred to the main memory one at a time for clustering. Only the cluster representations are stored in the main memory to alleviate the space limitations.

(3) A parallel implementation of a clustering algorithm may be used. We discuss these approaches in the next three subsections.

5.12.1 *Divide and Conquer Approach*. Here, we store the entire pattern matrix of size $n \times d$ in a secondary storage space (e.g., a disk file). We divide this data into p blocks, where an optimum value of p can be chosen based on the clustering algorithm used [Murty and Krishna 1980]. Let us assume that we have n/p patterns in each of the blocks.

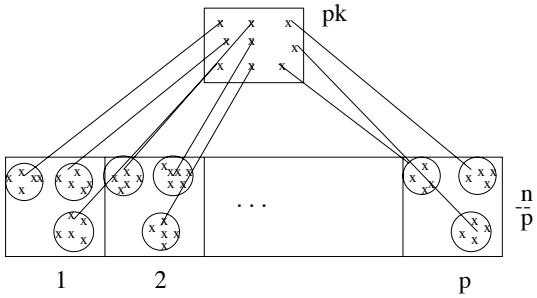


Figure 23. Divide and conquer approach to clustering.

We transfer each of these blocks to the main memory and cluster it into k clusters using a standard algorithm. One or more representative samples from each of these clusters are stored separately; we have pk of these representative patterns if we choose one representative per cluster. These pk representatives are further clustered into k clusters and the cluster labels of these representative patterns are used to relabel the original pattern matrix. We depict this two-level algorithm in Figure 23. It is possible to extend this algorithm to any number of levels; more levels are required if the data set is very large and the main memory size is very small [Murty and Krishna 1980]. If the single-link algorithm is used to obtain 5 clusters, then there is a substantial savings in the number of computations as shown in Table II for optimally chosen p when the number of clusters is fixed at 5. However, this algorithm works well only when the points in each block are reasonably homogeneous which is often satisfied by image data.

A two-level strategy for clustering a data set containing 2,000 patterns was described in Stahl [1986]. In the first level, the data set is loosely clustered into a large number of clusters using the leader algorithm. Representatives from these clusters, one per cluster, are the input to the second level clustering, which is obtained using Ward's hierarchical method.

Table II. Number of Distance Computations (n) for the Single-Link Clustering Algorithm and a Two-Level Divide and Conquer Algorithm

n	Single-link	p	Two-level
100	4,950		1200
500	124,750	2	10,750
100	499,500	4	31,500
10,000	49,995,000	10	1,013,750

5.12.2 *Incremental Clustering*. Incremental clustering is based on the assumption that it is possible to consider patterns one at a time and assign them to existing clusters. Here, a new data item is assigned to a cluster without affecting the existing clusters significantly. A high level description of a typical incremental clustering algorithm is given below.

An Incremental Clustering Algorithm

(1) Assign the first data item to a cluster.

(2) Consider the next data item. Either assign this item to one of the existing clusters or assign it to a new cluster. This assignment is done based on some criterion, *e.g.* the distance between the new item and the existing cluster centroids.

(3) Repeat step 2 till all the data items are clustered.

The major advantage with the incremental clustering algorithms is that it is not necessary to store the entire pattern matrix in the memory. So, the space requirements of incremental algorithms are very small. Typically, they are noniterative. So their time requirements are also small. There are several incremental clustering algorithms:

(1) The leader clustering algorithm [Hartigan 1975] is the simplest in terms of time complexity which is $O(nk)$. It has gained popularity because of its neural network implementation, the ART network [Carpenter and Grossberg 1990]. It is very easy to implement as it requires only $O(k)$ space.

(2) The shortest spanning path (SSP) algorithm [Slagle et al. 1975] was originally proposed for data reorganization and was successfully used in automatic auditing of records [Lee et al. 1978]. Here, SSP algorithm was used to cluster 2000 patterns using 18 features. These clusters are used to estimate missing feature values in data items and to identify erroneous feature values.

(3) The *cobweb* system [Fisher 1987] is an incremental conceptual clustering algorithm. It has been successfully used in engineering applications [Fisher et al. 1993].

(4) An incremental clustering algorithm for dynamic information processing was presented in Can [1993]. The motivation behind this work is that, in dynamic databases, items might get added and deleted over time. These changes should be reflected in the partition generated without significantly affecting the current clusters. This algorithm was used to cluster incrementally an INSPEC database of 12,684 documents corresponding to computer science and electrical engineering.

Order-independence is an important property of clustering algorithms. An algorithm is *order-independent* if it generates the same partition for any order in which the data is presented. Otherwise, it is *order-dependent*. Most of the incremental algorithms presented above are order-dependent. We illustrate this order-dependent property in Figure 24 where there are 6 two-dimensional objects labeled 1 to 6. If we present these patterns to the leader algorithm in the order 2,1,3,5,4,6 then the two clusters obtained are shown by ellipses. If the order is 1,2,6,4,5,3, then we get a two-partition as shown by the triangles. The SSP algorithm, *cobweb*, and the algorithm in Can [1993] are all order-dependent.

5.12.3 *Parallel Implementation*. Recent work [Judd et al. 1996] demonstrates that a combination of algorithmic enhancements to a clustering algorithm and distribution of the computations over a network of workstations can allow an entire 512×512 image to be clustered in a few minutes. Depending on the clustering algorithm in use, parallelization of the code and replication of data for efficiency may yield large benefits. However, a global shared data structure, namely the cluster membership table, remains and must be managed centrally or replicated and synchronized periodically. The presence or absence of robust, efficient parallel clustering techniques will determine the success or failure of cluster analysis in large-scale data mining applications in the future.

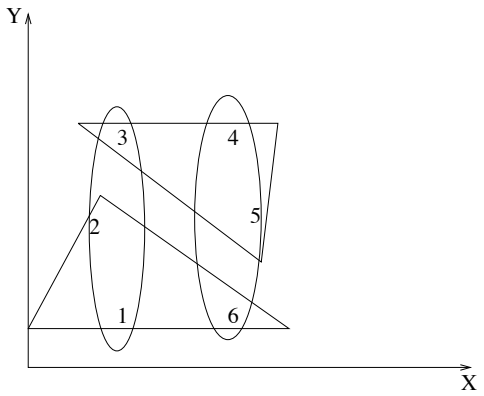


Figure 24. The leader algorithm is order dependent.

6. APPLICATIONS

Clustering algorithms have been used in a large variety of applications [Jain and Dubes 1988; Rasmussen 1992; Oehler and Gray 1995; Fisher et al. 1993]. In this section, we describe several applications where clustering has been employed as an essential step. These areas are: (1) image segmentation, (2) object and character recognition, (3) document retrieval, and (4) data mining.

6.1 Image Segmentation Using Clustering

Image segmentation is a fundamental component in many computer vision