Machine Learning

Naureen Ghani

January 1, 2018

1 Computer Science Fundamentals

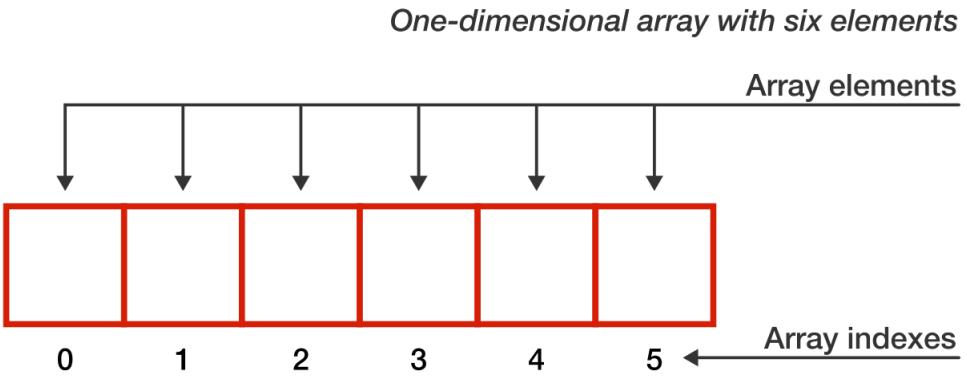
In the digital world, you need to solve problems efficiently. This requires organizing data for storage and analysis. In this section, we will learn the basics of programming.

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

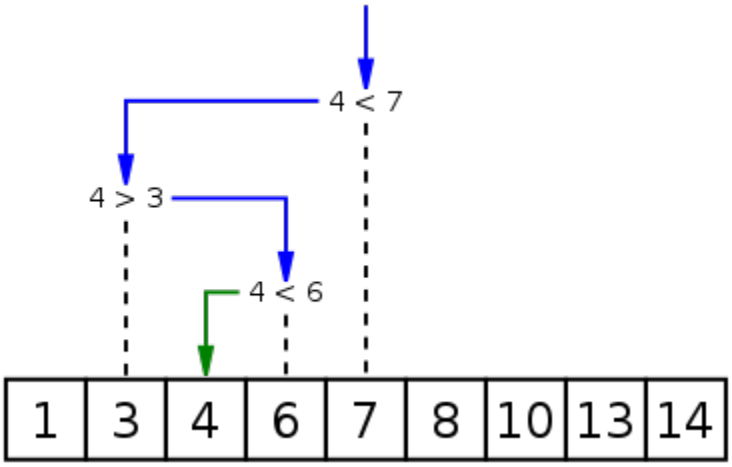
One of the simplest questions one might have about a list is whether or not some item is in the list. For example, if you are writing an algorithm to determine if someone should have access to a members-only website, you can solicit their information and then see if it matches some item in the members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

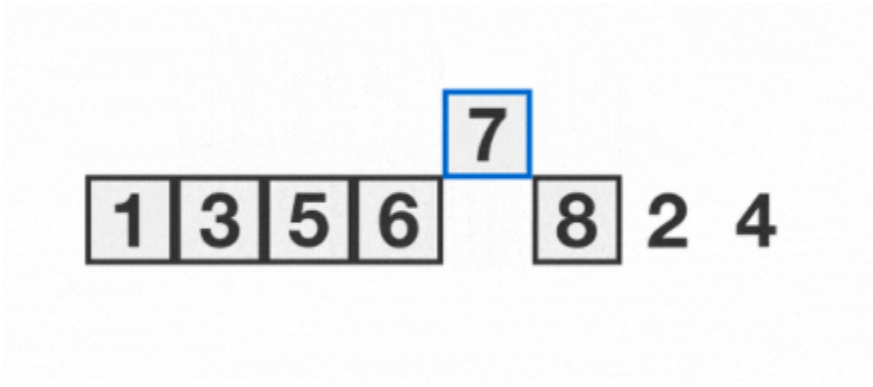
We can do better with a ***binary search.*** With a sorted array, a binary search repeatedly chooses a number in the middle of the remaining possible numbers, and then determines if the desired number would lie to to the left or right of this chosen number (or, if it is the chosen number). In each iteration,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

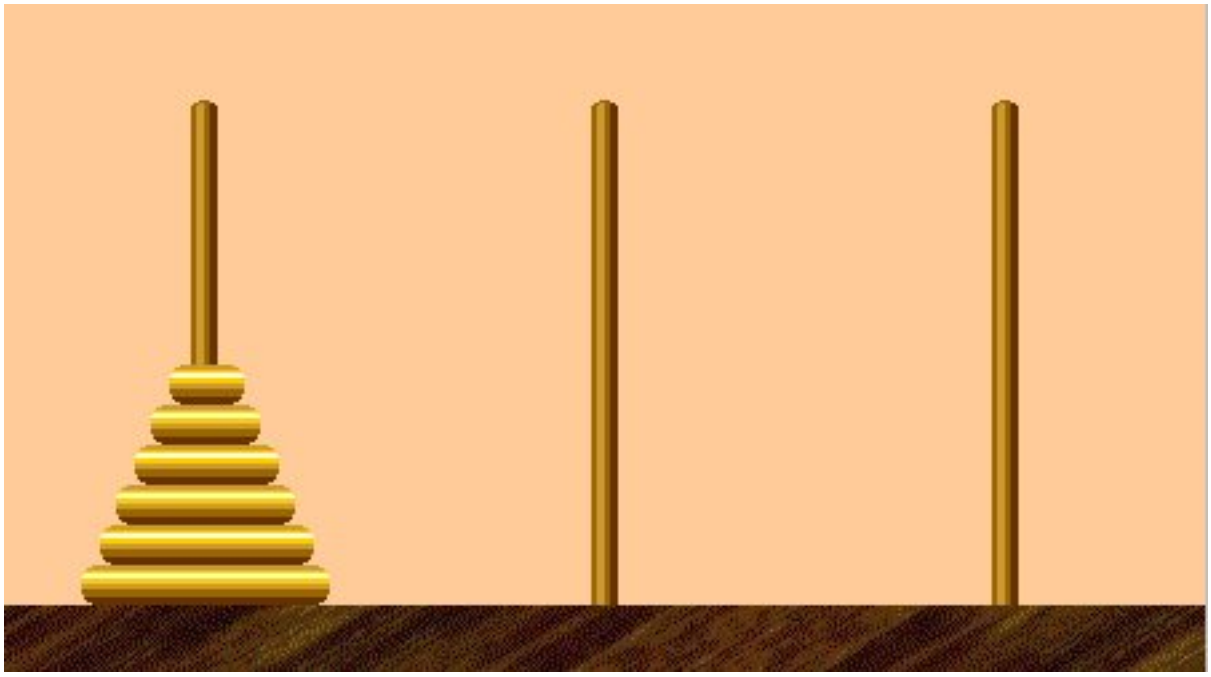
The true power of recursion lies in divide and conquer mechanisms. The ***divide and conquer method*** aims to:

- **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:

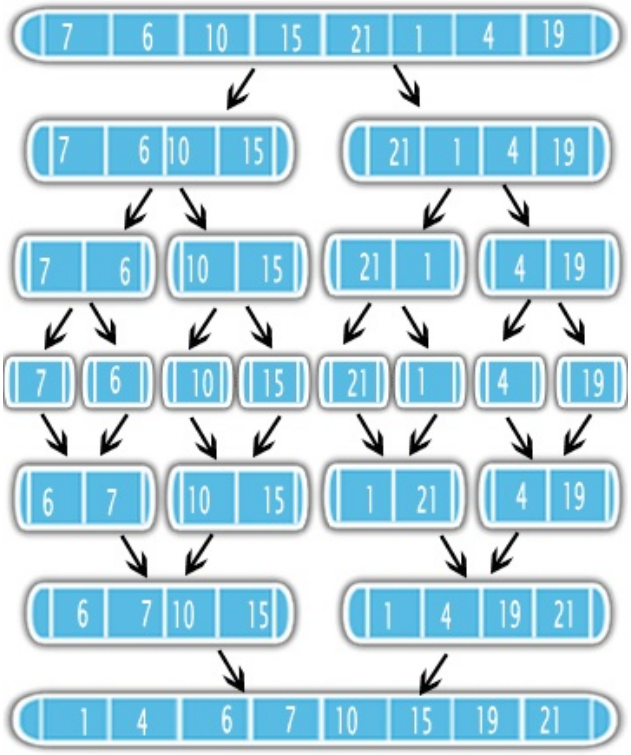
```
function z = merge(x,y)
% x is a row n-vector with x(1) <= x(2) <= ... <= x(n)
% y is a row m-vector with y(1) <= y(2) <= ... <= y(m)
% z is a row (m+n)-vector comprised of all the values in x and y, sorted so
% that z(1) <= ... <= z(m+n)

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:

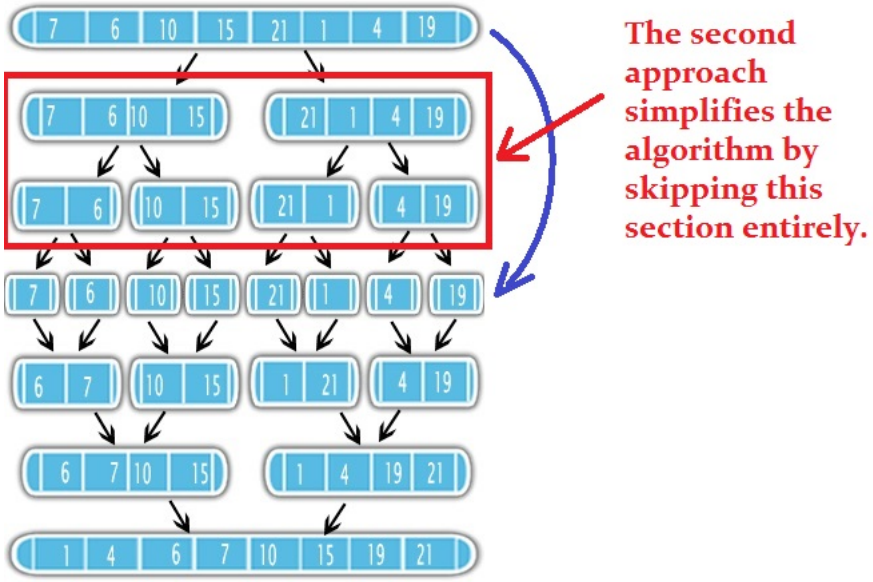


We can also describe the steps of the algorithm a little differently:

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:

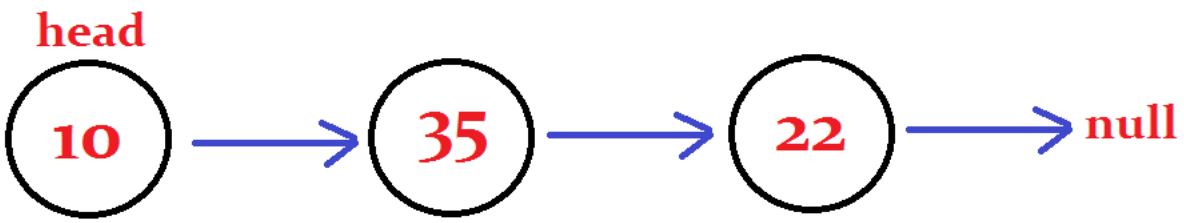
1. Select a random pivot point.
2. Select the first or last element of the array.
3. **"Median of three" method:** choose the first, middle, and last elements, and choose the median.
4. Use an algorithm to find the median.

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.

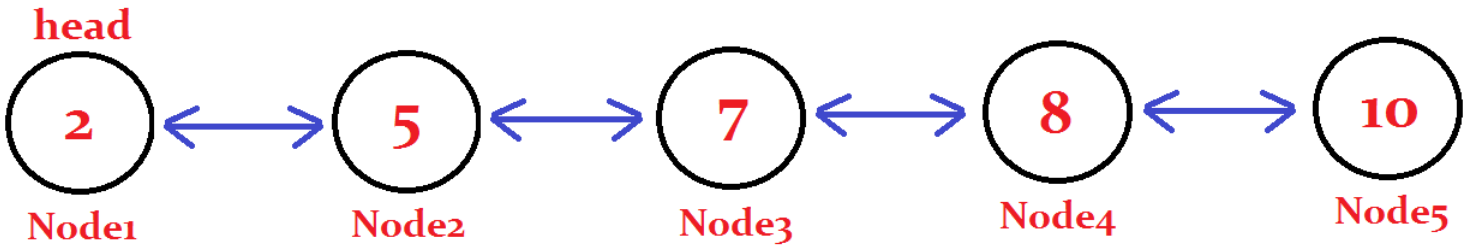
A ***linked list*** is a linear data structure that holds data in nodes. The nodes hold a piece of data as well as a pointer to another node ("the link"). One can picture a linked list as follows:



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.

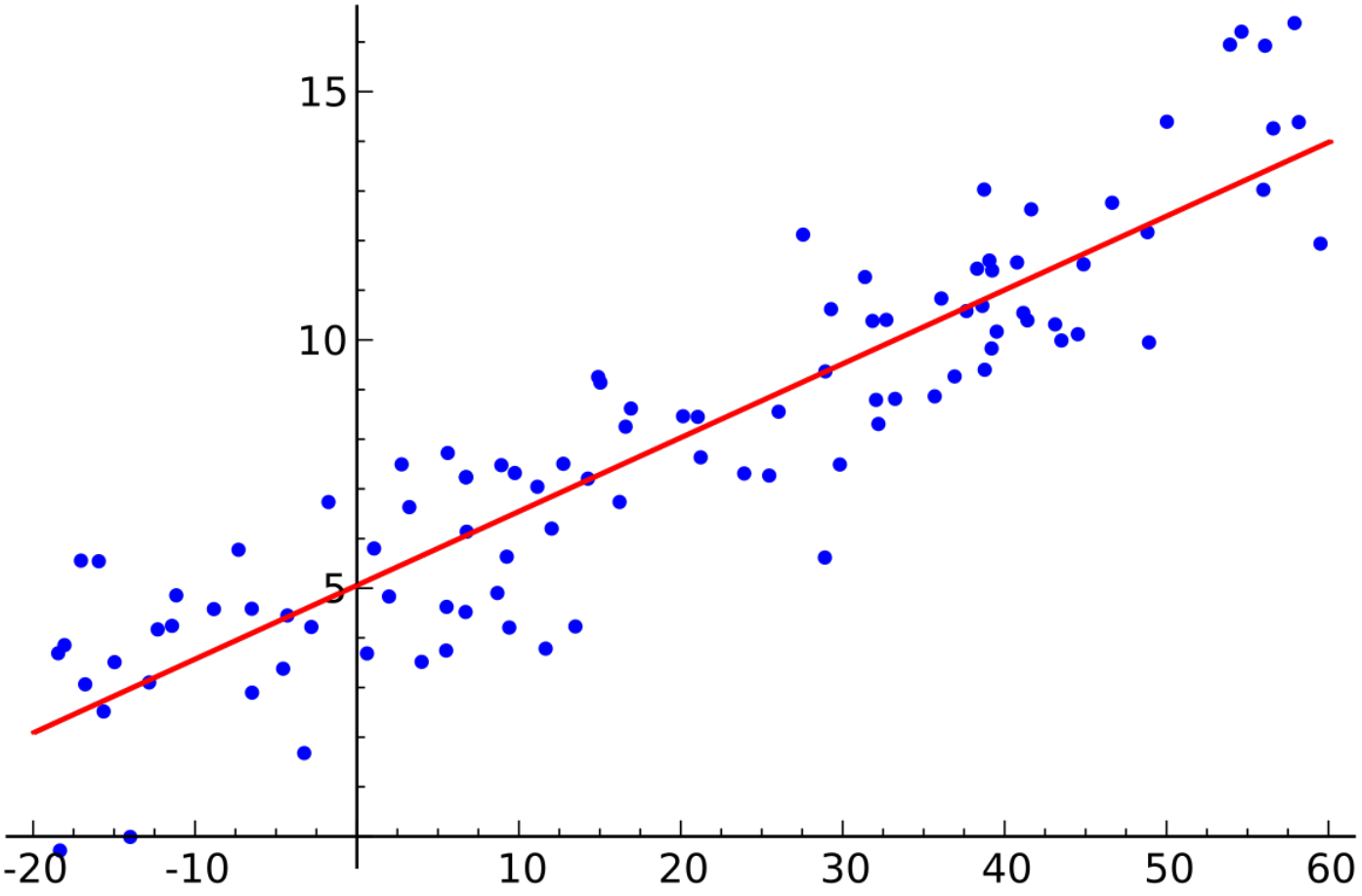
2.1 Linear Regression

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,

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

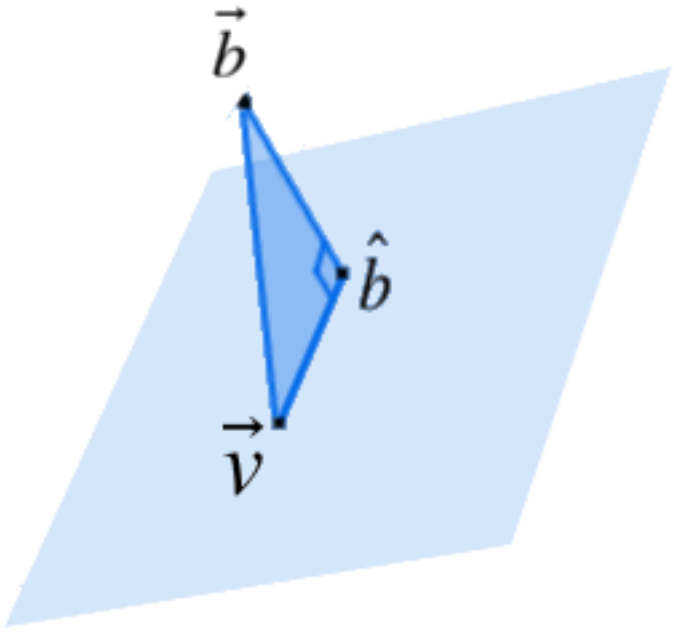
When we are minimizing the SSE, we are minimizing the distance between $A\overrightarrow{x}$ and $\overrightarrow{b}$ since minimizing a squared positive value will also minimize the value itself. In other words, we need to find a vector $\overrightarrow{x}$ for which $A\overrightarrow{x}$ is as close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

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

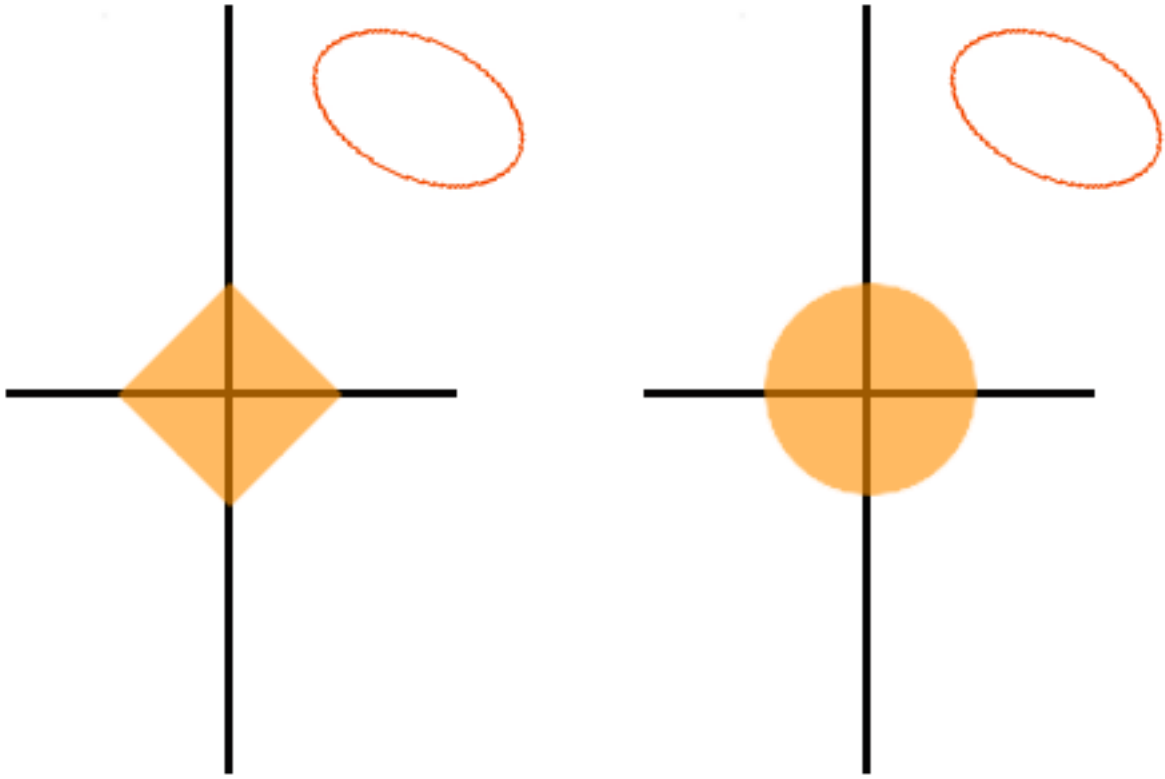
is equivalent to minimizing the SSE when

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

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.

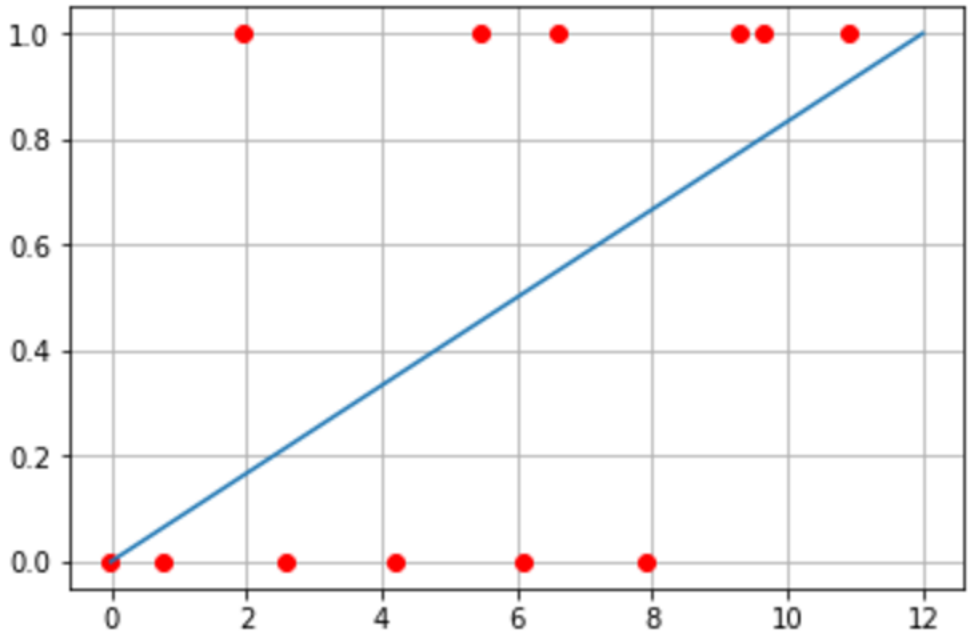
Another common practice, for which the uses will become apparent later on, is to represent a set of one of these hot vectors by placing them into the rows of a matrix known as an ***indicator matrix.*** If we have four animals, a bear, a lion, another bear, and a tiger, the matrix representing their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



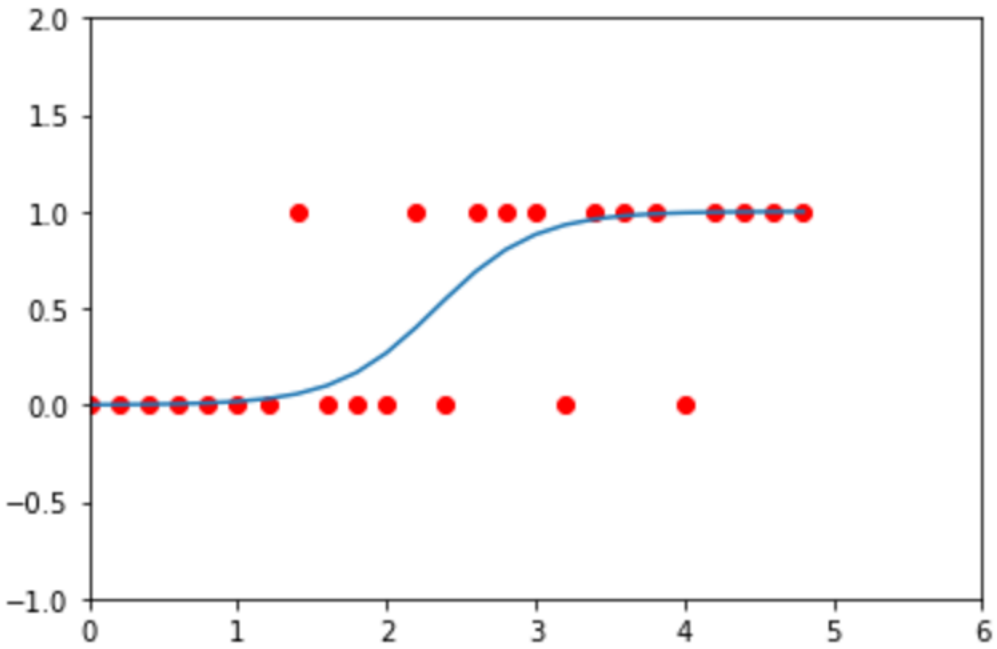
If there are K classes, we can use linear regression on the columns of our indicator matrix to generate K equations, one corresponding to each class. Then, if we are given an arbitrary set of values for our predictor variables, we can simply plug those values into each equation and put them into whichever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

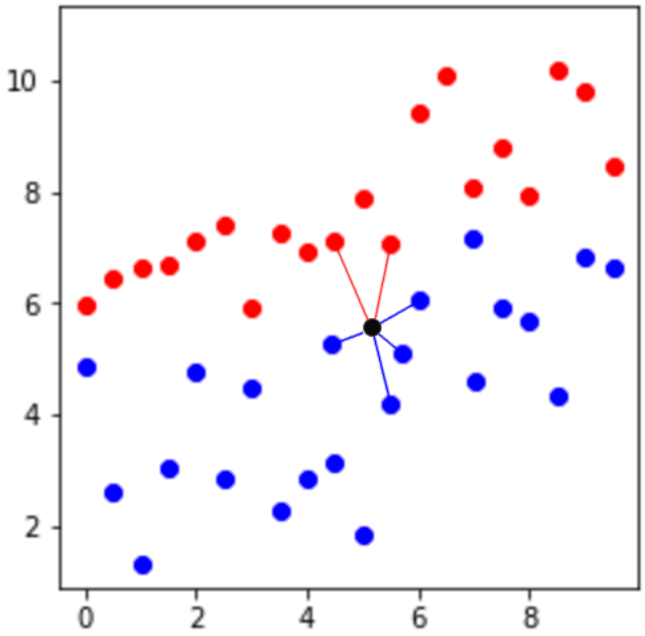
The inverse of the logistic function $\frac{e^x}{1+e^x}$ is the logit $\ln \frac{x}{1-x}$. Taking the logit of a function produced by logistic regression gives the linear function that feeds into the logistic one. Mathematically,

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

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

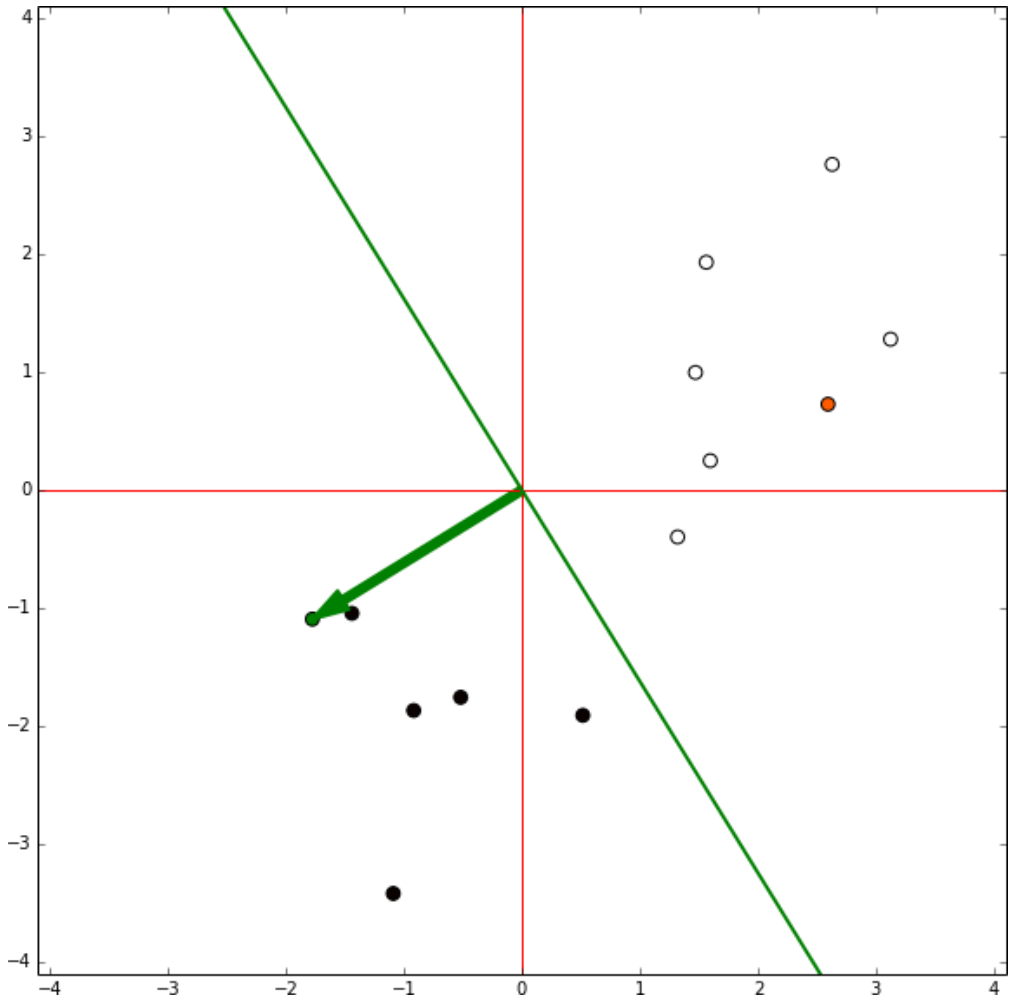
We have put most of our efforts so far into estimating the probability that certain points will be in certain classes. However, there is no rule requiring this approach to finding a boundary line. All we actually need is a division which accurately separates the classes and a function which outputs true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

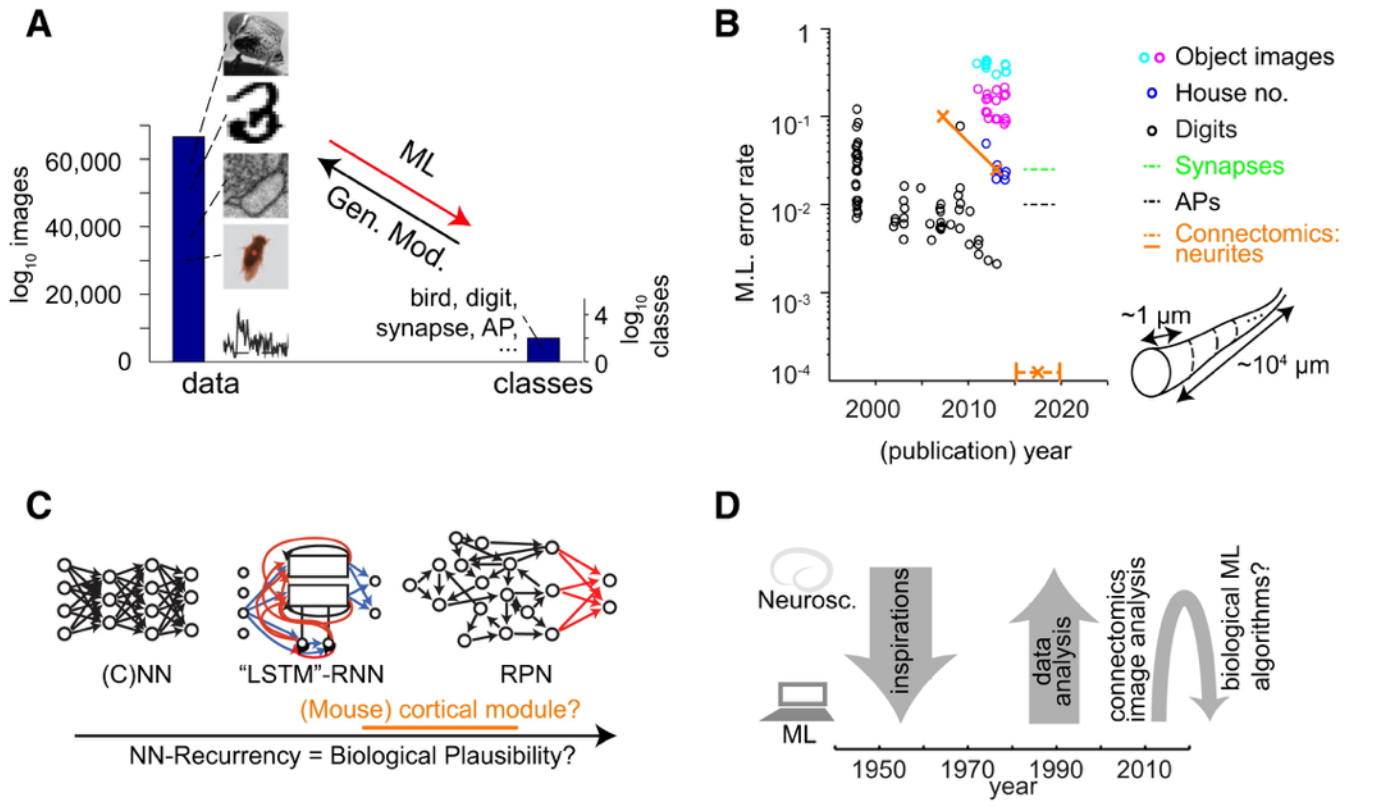
A more interesting relationship is the use of machine learning algorithms as a source of theories about how the brain works (and vice versa). For example, an algorithm called ***td-learning*** inspired a theory on the functional role of dopamine influx in the brain. This algorithm was previously used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



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?

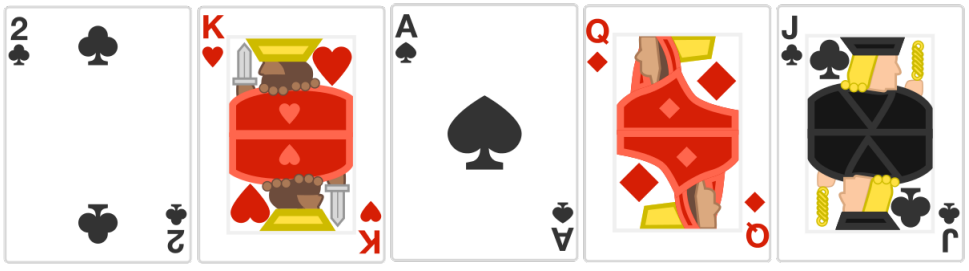
- **Hint:** Pick an element from the list to observe first. If the element we observe is greater than our target, what does this tell us? What if it's lower? With a sorted list, knowing if an element is greater or less than the element we're looking for can be useful.

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?

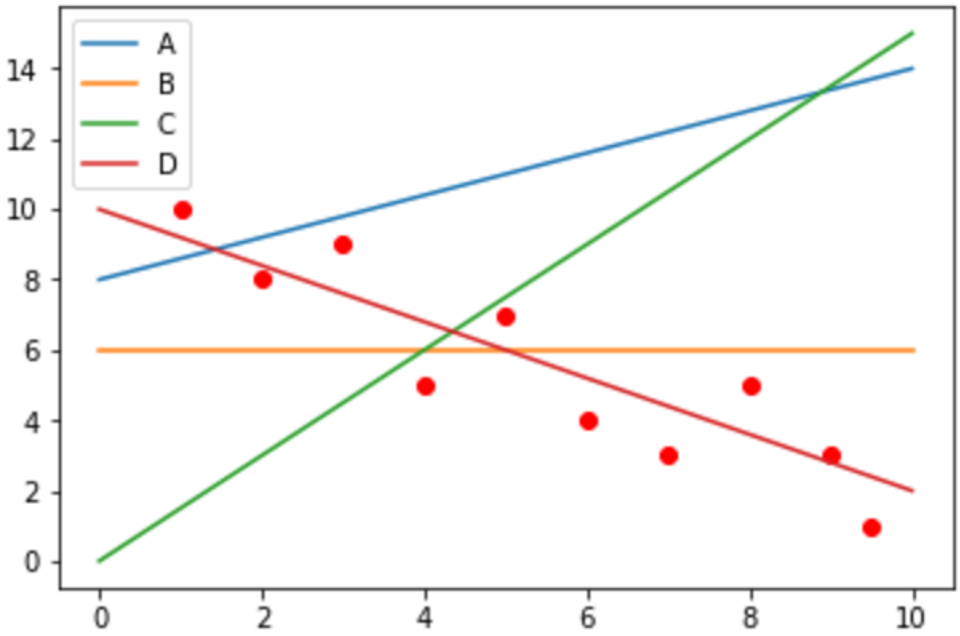
- For large N and randomly distributed data, quicksort is usually faster than mergesort.
- Quicksort runtime ranges from $O\left(N^2\right)$ to $O\left(N \log_2 N\right)$ but mergesort is always $O\left(N \log_2 N\right)$
- The runtime of quicksort is more sensitive to the initial ordering of the array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

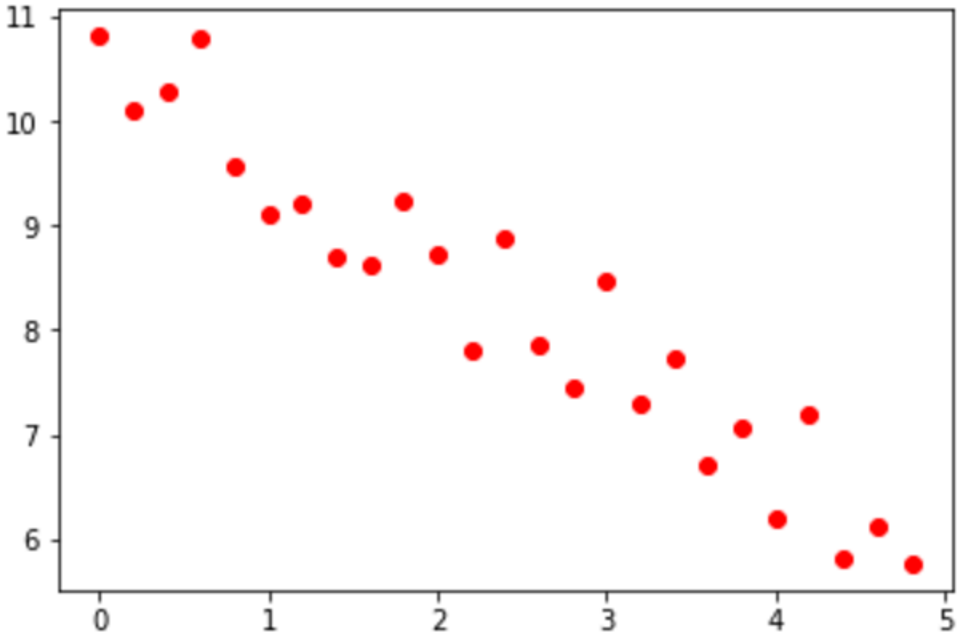
1. A perfectly accurate sensor taped to another light bulb, twice as bright as the one we're measuring. We can't get rid of the second light bulb; we don't know why but the sensor won't work without it.
2. A primitive but effective device from the 1840's, rather like a thermometer. It works, but a human 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?

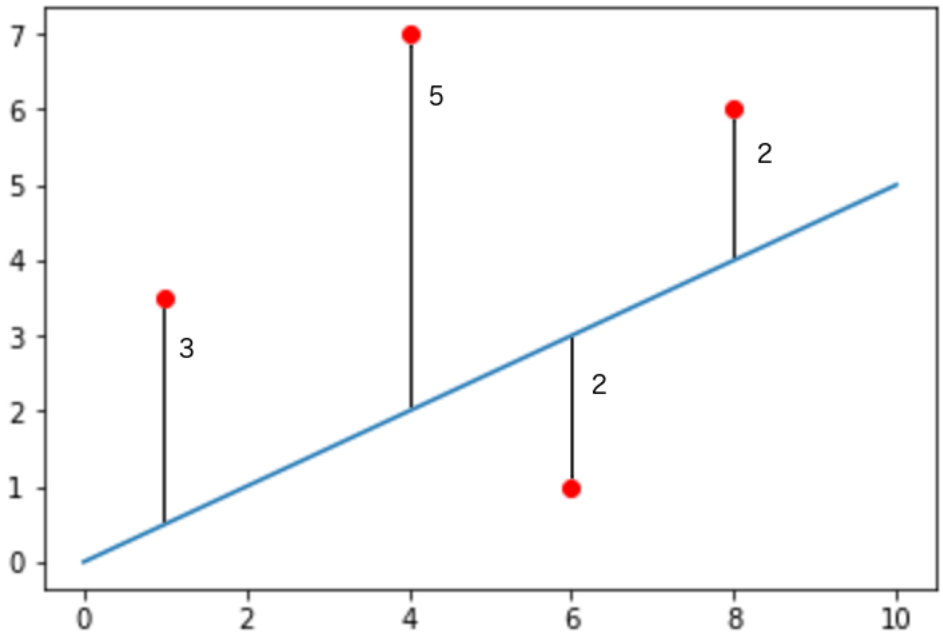
- -1
- 3
- 0.25
- 1



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

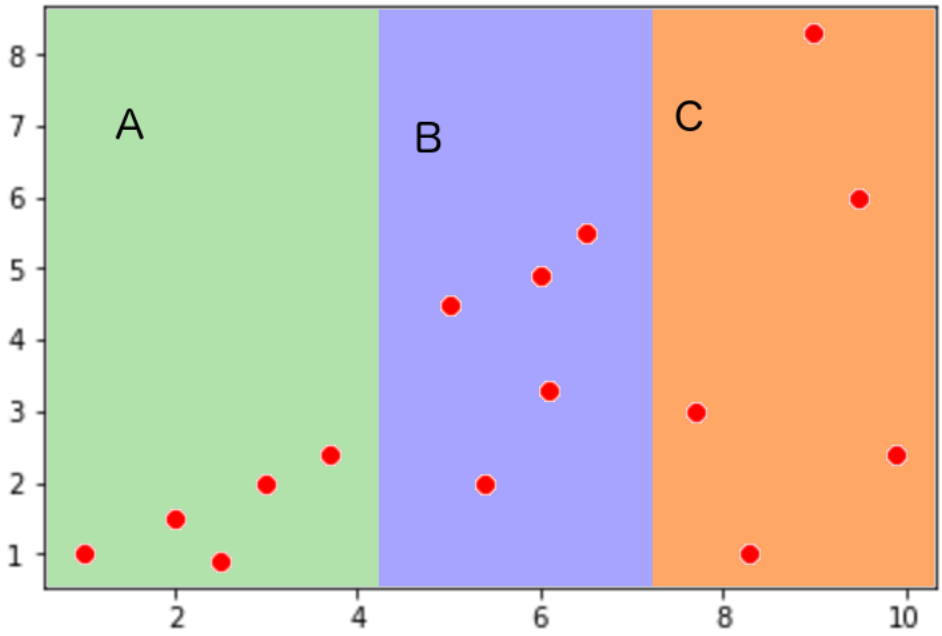
We can see the effects of multicollinearity clearly when we take this problem to its extreme. Say that we have two predictor variables, x_1 and x_2, and one response variable y. Using the test data given in the table below, determine which candidate best-fit equation has the lowest SSE:

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?

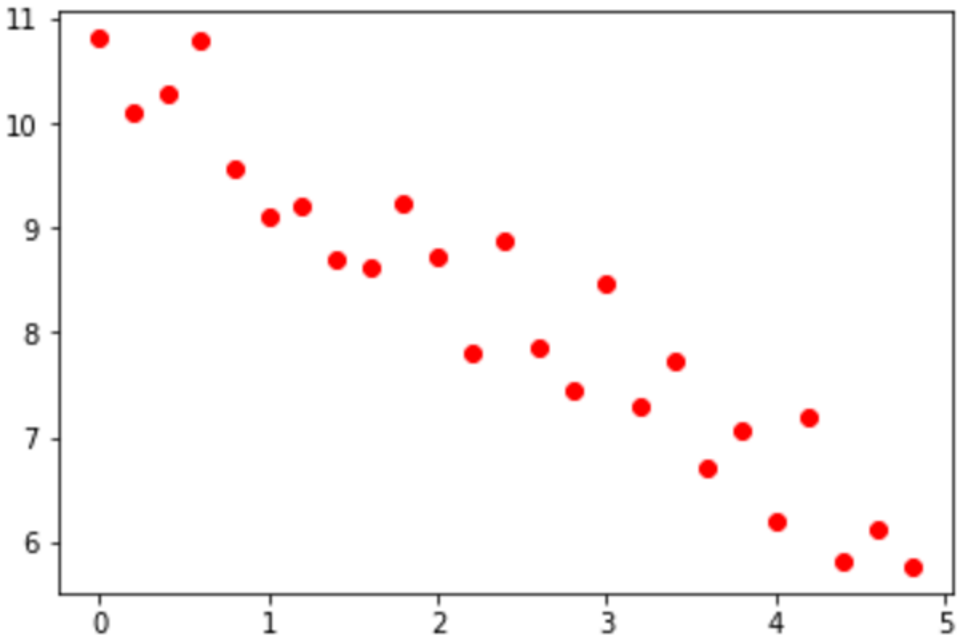
- The SSE will decrease while the error on new data increases
- Everything will remain the same
- The SSE will increase while the error on new data increases
- The SSE and the error on new data will increase

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?

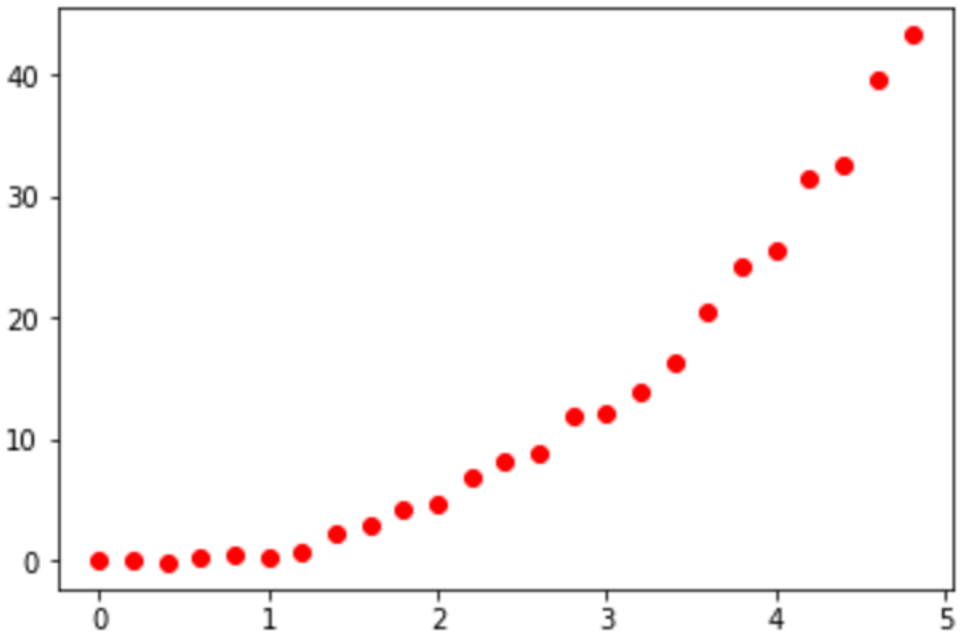
Data Set A:

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

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

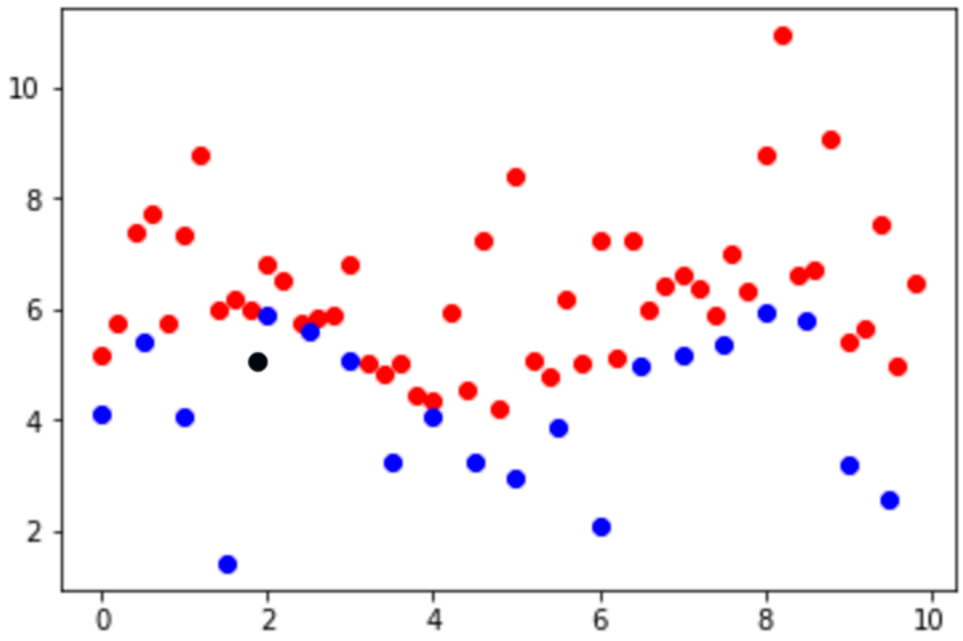
What is the total chance that these two points will have respective classes of red and blue, rounded to the nearest hundredth?

- **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}
- $15n$

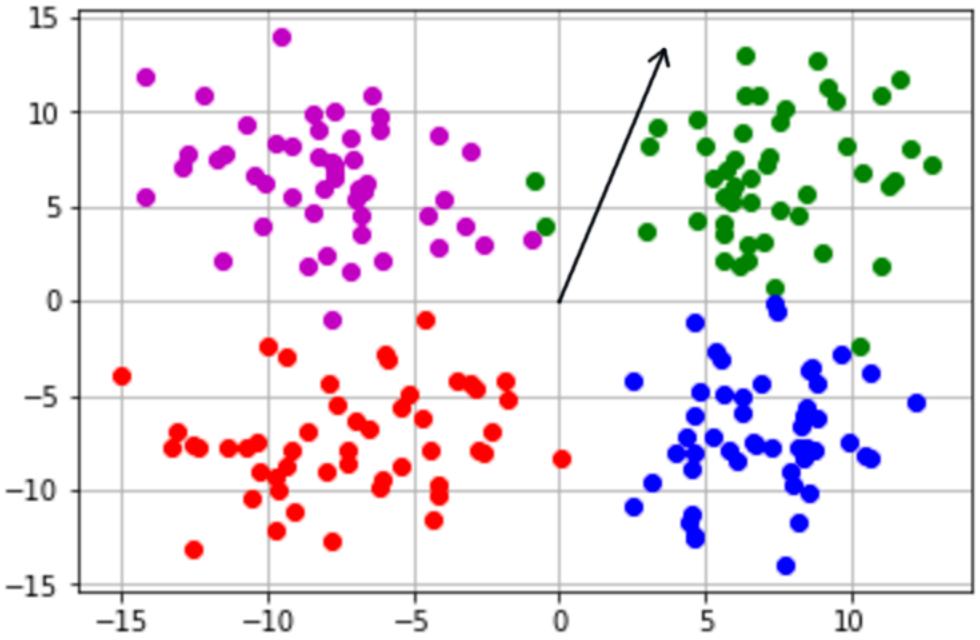
Two scientists want to estimate whether a truck will break down soon based on distance traveled and money spent on maintenance. They have already decided to measure money spent in dollars, but they disagree on how to measure distance. One scientist wants to use inches, while the other wants to use miles.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

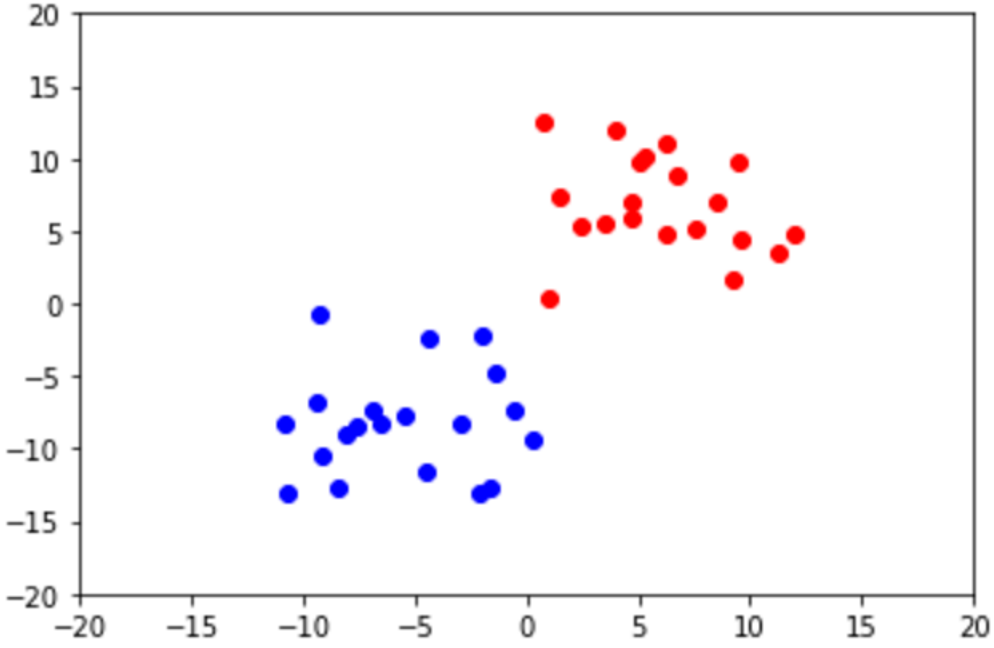
$$\vec{w} \cdot \vec{x} \geq b.$$

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?



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

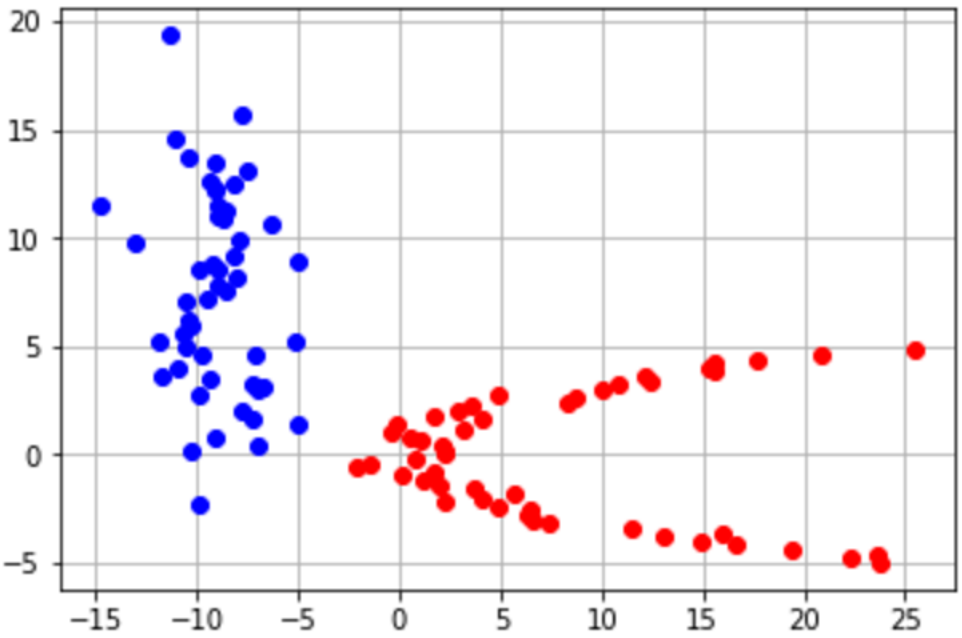
In the process of training a perceptron, we must frequently correct the weights to deal with any misclassified points. For instance, let's say that we have two classes, red and blue, and that our perceptron will see positive results as red and negative results as blue. Generally, if a point $\vec{x}$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

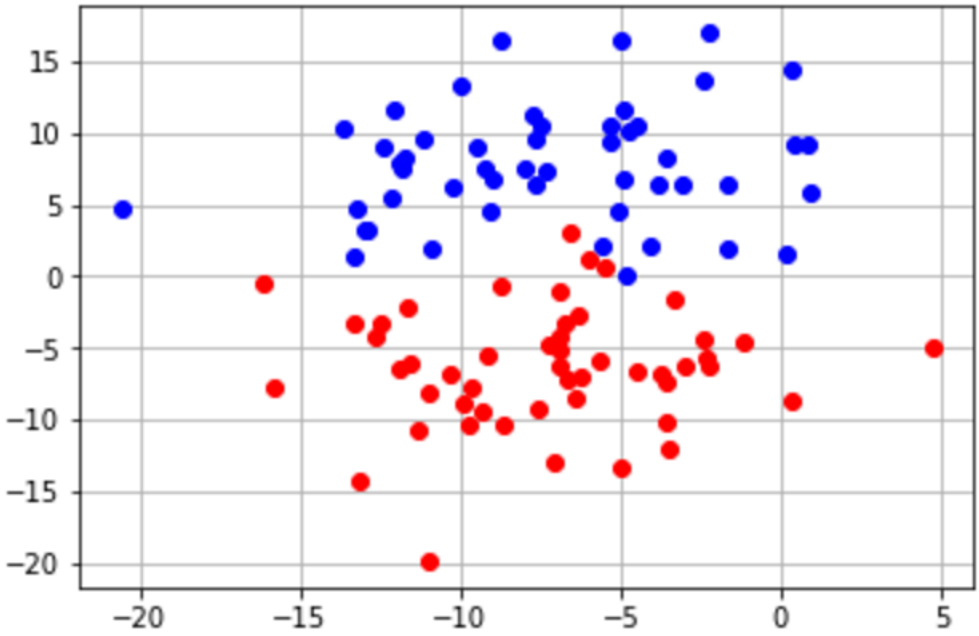
- $(\vec{w} + \vec{x}) \cdot \vec{x} \geq \vec{w} \cdot \vec{x}$
- $|\vec{w} + \vec{x}| \geq |\vec{w}|$
- $(\vec{w} + \vec{x}) \cdot \vec{x} \leq \vec{w} \cdot \vec{x}$
- $(\vec{x} + \vec{x}) \cdot \vec{x} \geq \vec{w} \cdot \vec{x}$

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

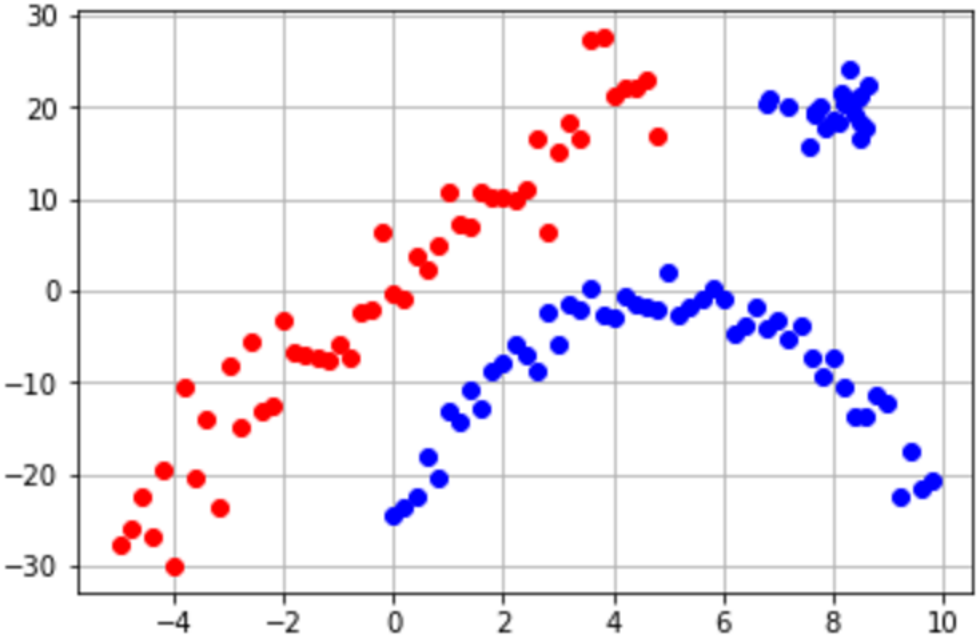
Data Set A:



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?

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