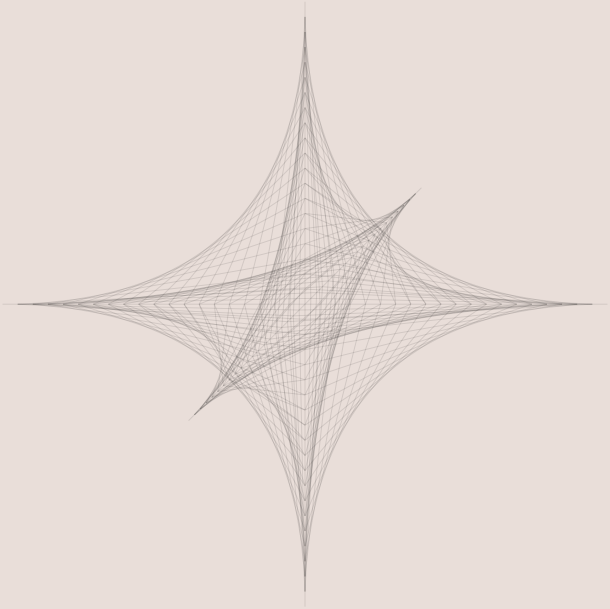
Introduction to Data Science

by Khallil Benyattou



りっぱ
立派

March 3, 2024

Contents

Disclaimer

I wrote these notes for my own personal use while closely following the lectures of **MIT 6.0002 (Fall 2016)**. There are some bits missing (diagrams I haven't generated yet) but I've managed to create some of them with Ti*k*Z diagrams.

If anybody is trying to sell you these notes:

- ~~I am flattered.~~
- These notes are freely available for public access and should not, in any circumstance, be sold or distributed for profit.

These notes are hosted in only one location, my website:

https://kbenyattou.github.io/notes/

All errors herein are my own.

Chapter 1

Optimisation

The main part of this course is on computational models. How do we use computation to understand the world in which we live? A model can be thought of as an experimental device that can help us explain things that have already happened or predict what things will be like in the future. Science has had an increasing reliance on computation to supplement traditional experimentation. We'll discuss 3 kinds of models: **optimisation**, **statistical** and **simulation** models.

Optimisation problems have a function that we wish to maximise (or minimise) while abiding by a (possibly empty) set of constraints.

Some classic optimisation problems are:

- Shortest path problems
- The travelling salesman - Given a number of cities and costs to travel from city to city by aeroplane, what's the least cost round trip you can find?
- Bin packing - Filling up some container with objects of varying size and shape. (Very important in shipping)
- Sequence alignment - These frequently crop up in biology and natural language processing. For example, aligning DNA sequences.

It's useful to have an inventory of previously solved problems. This is because when solving new problems, we can reduce them by mapping them onto old problems that other people have already found solutions for. Most optimisation problems do not have fast solutions. We'll begin by focusing on the knapsack problem.

1.1 The Knapsack (Backpack) Problem

The knapsack problem is usually formulated in terms of a burglar who wishes to steal items of the greatest possible value from a house under the constraint that everything he steals must fit in their backpack.

There are two variants of the knapsack problem:

- The continuous knapsack problem allows for fractions of items to be taken.

  This version of the problem is very easy to solve with a greedy algorithm: Take as much of the highest value item as you can until your bag either runs out of space or you've taken all of the highest value item. In this case, move onto the next highest value item and repeat until the bag is full.

- The discrete "0/1" knapsack problem is more interesting. In this version, you have the choice to either take the entirety of an item or leave it behind. You aren't

allowed to take, for example, 3/5 of a television.

The discrete knapsack problem is more interesting because any choice you make will non-trivially affect all future choices.

Formalisation of the Discrete Knapsack Problem

- Each item is represented by a pair `<value, weight>`
- The knapsack can accommodate items with a total weight of no more than w
- A vector L of length n will represent the set of available items. Each element of the vector is an item.
- A vector v of length n will represent whether each item has been taken or not with a 1 or 0 respectively

Thus, we're looking for a v that

$$\begin{cases} \text{maximises } \sum_{i=0}^{n-1} v[i] * L[i].\text{value} \\ \text{subject to the constraint that } \left(\sum_{i=0}^{n-1} v[i] * L[i].\text{weight} \right) \leqslant w \end{cases}$$

The obvious method is via brute force. Enumerate all possible combinations of items (generate the power set $\wp(L)$ of L i.e. the set of all possible subsets), remove all combinations whose total weights exceed the weight constraint and choose any one combination with the largest value. Since the power set of a set L has size $2^{\text{len(L)}}$, this is clearly not very practical. So is there another candidate for an algorithm that would give us a solution in less than exponential time? Sadly not. The knapsack problem and many other optimisation problems are inherently exponential. However, there are some good solutions.

1.2 Greedy Algorithms

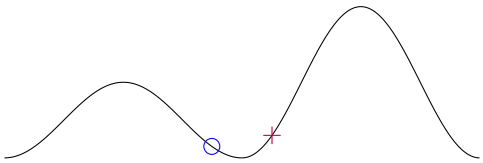
In pseudocode, a greedy algorithm takes the following form:

Algorithm 1 A greedy knapsack algorithm.

```
1: while knapsack is not full do
2:     put "best" available item into the knapsack
3: end while
```

The algorithm depends on how we define the "best" available item.

Greedy algorithms make a sequence of local optimisations. Take, for example, a greedy algorithm that describes how to climb a hill: "If you can increase your altitude, do it. If you can't, stop the process."



According to our greedy algorithm, if we start at ∘ we make our way to the top of the left hill making locally optimal decisions. This peak is a local maximum. If we start at +, we make our way to the top of the hill on the right - a global maximum. This is the problem with greedy algorithms. By making locally optimal choices, there's a possibility that we get stuck at a local maximum and don't maximise our function.

Implementation for the Discrete Knapsack Problem

We begin by defining a new abstract data type for each item. This gives us an easy way, via methods like `.get_name()`, to retrieve attributes like the item's name etc.

```
class Possession:
    """Each possession has a name (str), value (float) and weight (float).
    These can be accessed with the .get_name() and .get_value() methods"""

    def __init__(self, name, value, weight):
        self.name = str(name)
        self.value = float(value)
        self.weight = float(weight)

    def get_name(self):
        return self.name

    def get_value(self):
        return self.value

    def get_weight(self):
        return self.weight

    def __str__(self):
        return str(self.name) + "\t" + str(self.value) + "\t" +
    str(self.weight)

    # So that the items in the list can be printed out
    def __repr__(self):
        return "<" + str(self.name) + ", " + str(self.value) + ", " +
    str(self.weight) + ">"
```

We build the list of items with the following code:

```
def item_selection(names, values, weights):
    """Assumes all inputs are lists of equal size"""
    selection = []
    for i in range(len(names)):
        selection.append(Possession(names[i],values[i],weights[i]))
```

```
    return selection

names = ["wine", "beer", "pizza", "burger", "fries", "cola", "apple",
    "donut"]
values = [89, 90, 95, 100, 90, 79, 50, 10]
weights = [123, 154, 258, 354, 365, 150, 95, 195]

L = item_selection(names, values, weights)
```

We leave the choice of how to define the "best" next item up to the user in the form of a formal parameter called `keyFunction`.

```
def greedy_algorithm(selection, max_weight, keyFunction):
    total_value = 0
    total_weight = 0
    selectionCopy = sorted(selection, key=keyFunction, reverse=True)
    print(f"{selectionCopy}\n")
    optimal_choices = []

    for i in range(len(selectionCopy)):
        if selectionCopy[i].get_weight() + total_weight <= max_weight:
            total_weight += selectionCopy[i].get_weight()
            total_value += selectionCopy[i].get_value()
            optimal_choices.append(selectionCopy[i])

    print(f"The total value of the items stolen is {total_value}.\nThe bag
    has a maximum capacity of {max_weight} of which {total_weight} has been
    used.\nThe items chosen are:")
    for choice in optimal_choices:
        print(f"\t{choice}")
```

Finally, we initialise a value for the maximum available weight and call the algorithm with the user's chosen `keyFunction`. In this case, we use `Possession.get_value` to arrange the items' values in descending order:

```
max_weight = 1000
print(f"Using the greedy algorithm (by descending value) to allocate
    {max_weight} calories of items.")
greedy_algorithm(L, max_weight, Possession.get_value)
```

Pros of greedy algorithms:

- Easy to implement.
- Computationally efficient ($n \log n$ time complexity).

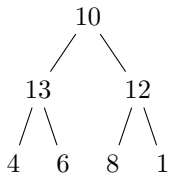
Cons:

- These algorithms don't actually solve the problem.
- Our "solution" may or may not even be a good approximation to an optimal solution.

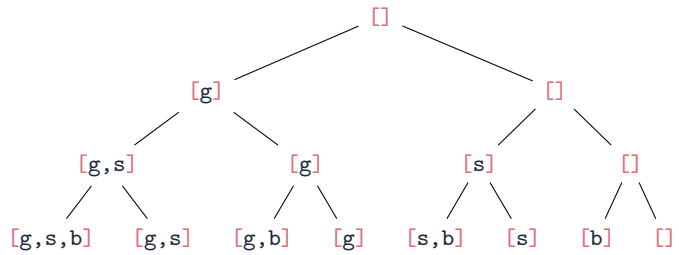
1.2.1 Brute Force Implementation

One particular brute force implementation uses a search tree. We'll search the tree we construct in a depth-first manner. Namely, we explore one path totally until moving onto the next.

e.g. The order in which the nodes on the tree below are visited is `[10, 13, 4, 6, 12, 8, 1]`:



Below is an example of an appropriate search tree for the knapsack problem with 3 items.



For this tree, if a node has children it will have two and the left node will represent 'taking the item' and the right node will mean that we've left the item.

The code for such a greedy algorithm is as follows:

```
def maxval(items, remaining_weight):
    if items == [] or remaining_weight == 0:
        result = (0, ())
    elif items[0].get_weight() > remaining_weight:
        # In this case, only explore the right branch because the current
    item cannot be added
        result = maxval(items[1:], remaining_weight)
    else:
        next_item = items[0]
        # Consider the left branch's value
        value_with_item, solution_with_item = maxval(items[1:],
    remaining_weight - next_item.get_weight())
        value_with_item += next_item.get_value()
        # Consider the right branch's value
        value_without_item, solution_without_item = maxval(items[1:],
    remaining_weight)
        # Compare the branches
        if value_with_item >= value_without_item:
            result = (value_with_item, solution_with_item + (next_item,))
```

```
            else:
                result = (value_without_item, solution_without_item)
        return result

def testmaxval(items, max_weight, printItems = True):
    print(f"Use search tree to allocate {max_weight} calories")
    val, taken = maxval(items, max_weight)
    print(f"Total value of items taken = {val}")
    if printItems:
        for item in taken:
            print("   ", item)
```

The computational complexity of the brute force depth-first method depends on the number of nodes generated in the tree. For n items to choose from, there are $n+1$ levels in the tree and each level i contains 2^i nodes. In total, there are $2^{n+1}-1$ nodes so this algorithm has a time complexity of $\mathcal{O}(2^{n+1})$ for an input of length n.

An obvious optimisation is to not explore parts of the tree that violate the constraint. This, however, doesn't change the worst-case complexity.

So is this all hopeless? Are we doomed to an inherently exponential solution to such optimisation problems? In theory, yes. In practice, there's an idea called dynamic programming that can cut the time taken down a lot.

1.3 Dynamic Programming

Take the fibonacci sequence as a motivation idea. Defined recursively, we have that $\text{fib}(0) = 1, \text{fib}(1) = 1$ and $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$ for integers $n > 1$. The growth in time taken to compute a result is roughly proportional to the growth in value of the result.

Consider the recursive call tree of `fib(6)`. We compute `fib` of 3 three times and each of these creates four additional calls of `fib`. It's bad enough to do something once. To do the same thing over and over again is wasteful. No matter how many times we compute `fib(3)`, we will get the same result. To avoid doing the same work over and over again, we can store the answer and look it up when it's needed. This is the key idea behind dynamic programming. We trade time for space with this method. There are two ways of storing values:

- A top-down solution involves storing the results from subproblems in a list. This technique is called **memoisation**.
- A bottom-up solution involves starting from the bottom (smallest possible value) and calculating values in order while holding temporary values in variables. This technique is called is called **tabulation**.

Examples of these two are below:

```
def top_down_fib(n, memo={}):
    if n == 0 or n == 1:
        return 1
```

```
    elif n in memo:
        return memo[n]
    else:
        memo[n] = top_down_fib(n-1, memo) + top_down_fib(n-2, memo)
        return memo[n]

def bottom_up_fib(n):
    if n == 0 or n == 1:
        return 1
    else:
        a, b  = 0, 1
        for i in range(n):
            a, b = b, a+b
        return b
```

The time complexity of the bottom-up approach is $\mathcal{O}(n)$ and its space complexity is constant (in $\mathcal{O}(1)$) because we only use two variables to track the intermediate results.

The time complexity of the bottom-up approach is also $\mathcal{O}(n)$ because we only solve each subproblem once. However, its space complexity is linear (in $\mathcal{O}(n)$) because we store our results in an array of size $n + 1$.

This is a tremendous improvement on the exponential time complexity of the recursively defined fibonacci function:

```
def recursive_fib(n):
    if n == 0 or n == 1:
        return 1
    else:
        return recursive_fib(n-1) + recursive_fib(n-2)
```

When can one use memoisation?

The problems it can help with (by finding an optimal solution) have two things:

- **Optimal substructure**: when a globally optimal solution can be found by combining optimal solutions to local subproblems e.g. `fib(x)` can be found by solving for `fib(x-1)`, `fib(x-2)` and then combining them.
- **Overlapping subproblems**: when finding an optimal solution involves solving the same problem multiple[1] times e.g. computing `fib(x)` many times.

Merge sort was an example of a problem with optimal substructure but we'd expect that most of the time, it wouldn't have overlapping subproblems unless the list repeated itself in parts. Dynamic programming can't be used to improve merge sort. Oh well. Nothing is a silver bullet.

The knapsack problem does have optimal substructure because we're comparing the take and not take branches. It doesn't have overlapping subproblems for the

pizza, beer, burger

[1] You can create a memo for a problem without overlapping subproblems but every time you look in the memo it's empty because you're only solving each subproblem once.

search tree because we never have the same contents. We'd get no speed-up. However, a different menu with more than 1 beer would give us the same problem to solve more than once. However, you don't need multiple copies of the same item to have overlapping subproblems e.g.

Chapter 2

Graphs

We now broaden the class of models to talk about graphs.

Definition 2.0.1 *A **graph** $G = (V, E)$ is a collection of **nodes** or **vertices** V (that have information associated with them) and **edges** E that connect nodes.*

- *If its edges are unidirectional, a graph is called **directed** (or a **digraph**). Otherwise, it's **undirected**.*
- *If there's an edge from a node n to a node m, we call n the **source** (or parent) node and m the **destination** (or child) node.*
- *If a value (or weight) is associated with each edge in a graph, the graph is called **weighted**.*

Trees are a special type of directed graph in which any pair of nodes is connected by a single path. There are no loops. We used a decision tree to solve the discrete knapsack problem.

In computer science, we mostly use Australian trees - they're upside down. The roots are the top and the leaves are at the bottom.

Graphs are useful because not only can we use them to model all sorts of networks based on relationships (like computer networks, transportation grids, financial networks etc.) but we can also use them to infer things about these structures:

- Finding sequences of links between elements: Is there a path from A to B?
- Finding the least expensive path between nodes
- Partitioning a graph into sets of connected elements
- Finding the most efficient way to separate sets of highly connected sets (sub-graphs). An example of this is the min-cut/max-flow problem.

2.1 Constructing Graphs

We begin by creating custom data types for nodes and edges. These will be later implemented into a digraph data type. For the time being, we simply initialise a name for each node.

```
class Node:
    def __init__(self, name):
```

```
        self.name = name

    def getName(self):
        return self.name

    def __str__(self):
        return self.name
```

An edge is a connection between a source node and a destination node. We add the functionality to retrieve these nodes. We hold off on adding a weight (float) to each edge for now.

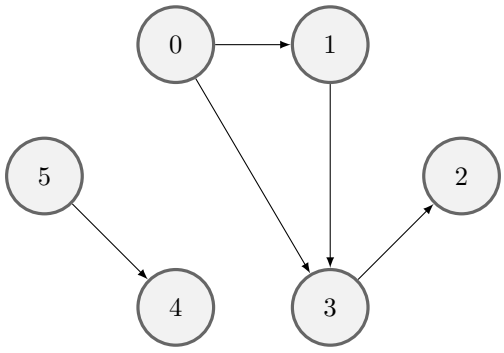
```
class Edge:
    def __init__(self, src, dest):
        self.src = src
        self.dest = dest

    def getSource(self):
        return self.src

    def getDestination(self):
        return self.dest

    def __str__(self):
        return self.src.getName() + "->" + self.dest.getName()
```

There are several ways to model a digraph. Take the following digraph for example:



We can model a graph using what is known as an **adjacency matrix**. If we think of the source and destination nodes as two separate lists `src` and `dest`, then the rows and columns of our adjacency matrix $A = (a_{ij})$ are indexed by `src` and `dest` respectively. If an edge from a parent node `src[i]` to a child node `dest[j]` exists, then $a_{ij} = 1$. Otherwise, $a_{ij} = 0$.

$$\begin{array}{c} \\ ⓪ \\ ① \\ ② \\ ③ \\ ④ \\ ⑤ \end{array} \begin{array}{c} \begin{array}{cccccc} ⓪ & ① & ② & ③ & ④ & ⑤ \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Notice that the adjacency matrix for our digraph is not symmetric. A more special type of graph is an undirected graph. The adjacency matrices of undirected graphs are symmetric[1] because if there's an edge from `src[i]` to a child node `dest[j]` i.e. $a_{ij} = 1$, then there is also an edge in the other direction $a_{ji} = 1$.

For an adjacency matrix, if the number of edges is particularly small compared to the number of nodes then the matrix is mainly 0s and so we have a very inefficient way of storing only a few values. To this end, we consider an adjacency list instead.

$$\begin{array}{ccc} 0 & \longmapsto & [1,3] \\ 1 & \longmapsto & [3] \\ 2 & \longmapsto & [\ ] \\ 3 & \longmapsto & [2] \\ 4 & \longmapsto & [\ ] \\ 5 & \longmapsto & [4] \end{array}$$

We represent an adjacency list as a mapping from source nodes to a list of destination nodes for which edges from the source node exist.

In Python, we can use a dictionary to formalise this structure - the source nodes will serve as keys in the dictionary and the lists will be the values.

```
class Digraph:
    def __init__(self):
        # We initialise an empty dictionary to store the edges
        self.edges = {}

    def addNode(self, node):
        if node in self.edges:
            raise ValueError("Duplicate node")
        else:
            self.edges[node] = []

    def addEdge(self, edge):
        src = edge.getSource()
        dest = edge.getDestination()
        if not (src in self.edges and dest in self.edges):
            raise ValueError("Node not in graph")
        self.edges[src].append(dest)
```

[1] A matrix $A = (a_{ij})$ is symmetric if and only if it is equal to its transpose i.e. for every i and j, $a_{ij} = a_{ji}$.

```
    def childrenOf(self, node):
        return self.edges[node]

    def hasNode(self, node):
        return node in self.edges

    def getNode(self, name):
        for n in self.edges:
            if n.getName() == name:
                return n
        raise NameError(name)

    def __str__(self):
        result = ""
        for src in self.edges:
            for dest in self.edges[src]:
                result = result + src.getName() + "->" + dest.getName() +
"\n"
                return result[:-1] #omit final newline
```

2.2 Depth First Search

- Start at an initial node
- Consider all edges that leave that node (in some order)
- Follow the first edge and check to see if we're at the goal node

↑ If not, repeat the process from the new node

- Continue until either you've found the goal node or run out of options
    - When you run out of options, backtrack to the previous node and try the next edge, repeating the process.

The code for a breadth first search is as follows:

```
def DFS(graph, start, goal, path = [], shortest_path):
    path += [start]
    if start == goal:
        return path
    else:
        for child in childrenOf(start):
            if child not in path:
                # The following condition shortest_path == None means that
    we don't yet have a solution
                if shortest_path == None or len(path) <= len(shortest_path):
                    new_path = DFS(graph, child, goal, path, shortest_path)
                    if new_path != None:
                        # If there is a solution, set it to shortest_path
                        shortest_path = new_path
                else:
                    print(f"{child} has already been visited")
            return shortest_path
```

2.3 Breadth-First Search

- Start at an initial node
- Consider all edges that leave that node (in some order)
- Follow the first edge and check to see if we're at the goal node

↑ If not, try the next edge from the current node

- Continue until either you've found the goal node or run out of options
    - When you run out of edge options, move to the next node at the same distance from the start, and repeat.
    - When you run out of node options, move to the next level in the graph (all nodes one step further from the start), and repeat.

Since we're searching the graph level by level, if we find a solution then we know that it's the shortest path.

The code for a breadth first search is as follows:

```
def BFS(graph, start, goal):
    initial_path = [start]
    # We need some way to keep track of the paths we're yet to explore and
    we use a queue for this
    path_queue = [initial_path]
    # This while loop condition says that as long as I have a path to
    explore and I'm yet to find a solution, proceed with the loop's
    contents:
    while len(path_queue) != 0:
        temp_path = path_queue.pop(0)
        last_node = temp_path[-1]
        if last_node == goal:
            return temp_path
        else:
            for next_node in graph.childrenOf(last_node):
                new_path = temp_path + [next_node]
                path_queue.append(new_path)
    return None
```

Chapter 3

Random Walks

Why are we looking at random walks?

Random walks are important in:

- There are many people who believe that movement in stock prices are best-modelled by a random walk
- Modelling physical processes like diffusion (heat, molecules etc.)

Random walks are also good illustrations of how to use simulations to understand the world around us (and a good excuse to practise classes and plotting).

3.1 Drunkards In A Field

Say you have a drunk in a field that we model as $\mathbb{Z}^2$ and he can only move randomly in 4 directions (up, down, left and right). Is there an interesting relationship between the number of steps our drunkard takes and the distance he is from the origin at the end of those steps?

Suppose he takes only 1 step. The drunkard is then, irrespective of the direction he steps in, always 1 step away from the origin. WLOG, let's assume the first step he takes is right. After 2 steps, his possible locations are $(0, 0), (1, 1), (2, 0)$ and $(1, -1)$.

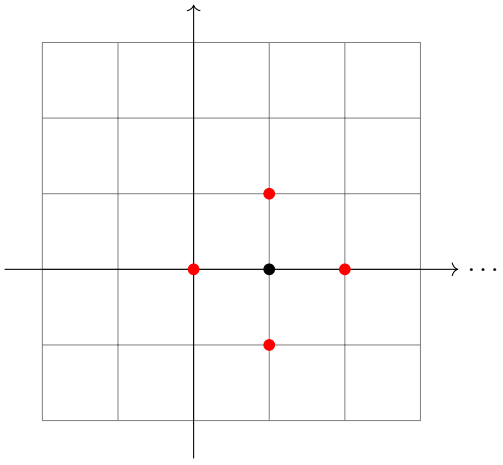


Figure 3.1: Possible locations (in red) after two steps assuming the first is east.

On average, our drunkard is $(1 + \sqrt{2})/2 \approx 1.207$ steps away - a little further away from the origin after two steps than one.

How about after 1000 steps? I refuse to calculate it directly so we resort to a simulation. We'll structure it exactly the same way in which we've been structuring our simulations. We'll model a walk with k-steps, repeat the process n-times and report back the average distance from the origin of the n-walks.

We begin by defining an umbrella class of drunks, under which we'll create special types of drunkards that behave differently.

```
class Drunk:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return f"{self.name}"
```

Let's say that a regular drunk is equally likely to move in any of the four compass directions but a homing drunk has a tendency to move east. Perhaps his home nation lies to the east. The homing drunk defines a **biased random walk**.

```
class RegularDrunk(Drunk):
    def takestep(self):
        possiblesteps = [(0,1), (0,-1), (-1,0), (1,0)]
        return random.choice(possiblesteps)

class HomingDrunk(Drunk):
    def takestep(self):
        possiblesteps = [(0,1), (0,-1), (-0.9,0), (1.1,0)]
        return random.choice(possiblesteps)
```

Now we define a class that we can use to report and modify the location of a drunkard. There are the usual getters and setters and we also include a method to return the distance between two locations.

```
class Location:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def make_move(self, dx, dy):
        return Location(self.x + dx, self.y + dy)

    def distance(self, other):
        xdiff = self.x - other.x
        ydiff = self.y - other.y
        return (xdiff**2 + ydiff**2)**(0.5)

    def get_x(self):
        return self.x

    def get_y(self):
        return self.y
```

```
    def __str__(self):
        return f"<{self.x}, {self.y}>"
```

We need a way to keep track of the drunkards and their locations in the field. To this end, we define our field as a dictionary mapping drunkards (these are immutable data types so we can use them as keys) to their locations.

```
class Field:
    def __init__(self):
        self.drunkards = {}

    def add_drunkard(self, drunk, location):
        if drunk in self.drunkards:
            raise ValueError(f"{str(drunk)} is already in the field")
        else:
            self.drunkards[drunk] = location

    def move_drunkard(self, drunk):
        if drunk not in self.drunkards:
            raise ValueError("Drunk is not present.")
        dx, dy = drunk.nextstep()
        self.drunkards[drunk] = self.drunkards[drunk].make_move(dx, dy)

    def get_location(self, drunk):
        if drunk not in self.drunkards:
            raise ValueError("The drunk is not in the field.")
        return self.drunkards[drunk]
```

Finally, we define one function to calculate the displacement at the end of a single k-step walk and another to average n trials of a single k-walk.

```
def walk(f, k, drunk):
    for i in range(k):
        f.move_drunkard(drunk)
    return f.get_location(drunk)

def averagedistance(f, n, k, drunk):
    start = Location(0,0)
    total_distance = 0.0
    for i in range(n):
        total_distance += walk(f, k, drunk).distance(start)
    return total_distance/n
```

Plot where they end up and maybe a relationship between steps and distance. sqrt(n) for reg and 0.05 for homing

Table 3.1: Recorded displacements for both a regular and a homing drunk over $n = 100$ trials.

k	Regular	Homing
1	1.000	1.020
2	1.307	0.775
10	2.762	2.445
100	7.115	10.089
1000	28.103	49.967

Chapter 4

Monte Carlo Simulation

Sometimes the combinatorics of probability are too complicated. Instead, one can repeat an experiment many times and calculate the average rate at which an event occurs. This is the idea of Monte Carlo Simulation, invented by Ulam and implemented by von Neumann.

Definition 4.0.1

- ***Monte Carlo Simulation*** *is a technique used to approximate the probability of an event by running the same simulation multiple times and averaging the results.*
- *A program is* ***deterministic*** *if whenever it's run on the same input, it produces the same output. Many aspects of the world can only be accurately modelled by a stochastic process.*
- *A process is* ***stochastic*** *if its next state can depend upon on some random element.*

Given an experiment, the population can be thought of as the universe of all possible examples (the outcome space). We take this population and sample it by drawing a proper subset. Then we make an inference about the population by running some statistics on the sample.

How can we guarantee (or at least reasonably suppose) that our inferences are representative of the population? By sampling the population at random.

A sample chosen at random tends to exhibit the same properties as the population.

If the sample isn't chosen at random, it would be unreasonable to suggest that our inferences are meaningful.

e.g. Flipping Coins

Suppose that you flip a coin once and it lands on heads. How confident are you that the next flip yields heads? Probably not very.

Now suppose that you've flipped a coin 100 times. They all land on heads. At this point, you're probably suspicious. This may not be a fair coin or it's weighted in some way that is (very, very) biased. Your best guess for the 101^{st} flip based on prior observations is clearly heads.

Suppose instead that out of 100 flips, you get 52H and 48T. The best guess for the probability of the 101^{st} flip landing on heads is 52/100. How confident are you in this guess? You shouldn't have much confidence compared to when the coin landed on heads

100 times in a row. Why is this? The answer lies in the variability (or **variance**) of the observations. With 52H and 48T, the next flip could really go either way. Thus, the key observation here is that:

As the variance grows, we need a larger sample to have the amount of confidence.

roulette example

4.1 Confidence Intervals

> This is a sticking point for a lot of people and I don't personally think the presentation of the LLN in this course is a statistically/mathematically rigorous one.
>
> Will revisit and add a better version when I get to it in my own studies.
>
> Me

Theorem 4.1.1 (**The Law of Large Numbers**) *In repeated independent tests with the same actual probability p of a particular outcome in each text, the chance that the fraction of times that outcome occurs differs from p converges to 0 as the number of trials $\to \infty$.*

This law is misunderstood and it's so frequently misunderstood that we give it a name, the **Gambler's Fallacy**. People believe that if deviations from the expected outcome occur, they'll be evened out in the future. For example, you may have heard in a baseball game that the commentator says something like "He's struck out 6 times, he's due for a hit this time (because he's usually a pretty good hitter)"

A very famous example comes from an eventful night at a Monte Carlo casino. At a roulette table, black appeared 25 times in a row. This is an extremely unlikely event with a probability of 1/. This prompted people to gather around and bet a lot of money on red showing up. They all believed that "surely the next will be red" to "even out" the ridiculous sequence of black appearing. However, they failed to understand that the 26^{th} roll is independent of all prior rolls.

$$\mathbb{P}\big(\{26^{\text{th}} \text{ is red} \,\big|\, 25 \text{ blacks in a row}\}\big) = \mathbb{P}\big(\{26^{\text{th}} \text{ is red}\}\big) = \frac{1}{2}$$

The house always wins and it definitely won on that night.

The Gambler's Fallacy isn't entirely off the mark. An idea often confused with it is called:

Theorem 4.1.2 (**Regression to the Mean**) *Following an extreme event, the next random event is likely to be less extreme.*

By less extreme, it is **not** meant that the next outcome will compensate or correct prior observations - only that the next event will be less extreme. To see the distinction between the two, consider 20 roulette spins, the first 10 of which are red.

- The Gambler's fallacy would suggest that the next 10 spins will have fewer than 5

reds (below the mean, to even out the unusually high amount of reds in the first 10 spins)

- Regression to the mean, on the other hand, would suggest that the next 10 spins will have less than 10 reds and be closer to the mean. This outcome is less extreme.

Whenever one is sampling, we're never guaranteed to get perfect accuracy. It's always possible that you'll get a weird sample. Alternatively, you may get the exact right answer.

We need to be able to differentiate between what happens to be true and what we know in a rigorous sense (or have real good reason to believe it) is true. The brings up a fundamental question in computational statistics - How many samples do we need to look at before we can have justifiable confidence in our answer?

Our confidence, as seen earlier, will depend on the variance of our observations.

Definition 4.1.3

- *The **variance** of a random variable X is a measure of the spread of our data and is defined as*

$$\mathrm{Var}(X) = \sum_{x \in X} \frac{(x-\mu)^2}{|X|}$$

  - *μ is the mean of our data set*
  - *We normalise the variance by dividing by the number of members of our data set. We do this to avoid making conclusions like "this set has high variance by virtue of having many members"*
  - *Squaring means that we don't care about whether the distance from the mean is positive or negative. We only care about the spread. However, we could've considered $|x - \mu|$ if the sign was all we cared about. Instead, squaring gives outliers extra emphasis.*

- *The **standard deviation** σ of a random variable X is given by the square root of the variance of X.*

The standard deviation on its own is a meaningless number. We need to think about it in the context of the mean. For example, $\sigma(X) = 100$ and $\mu = 100$ means that the standard deviation is very large. On the other hand, if $\sigma(X) = 100$ and $\mu = 10^9$, we can clearly see that σ is relatively small.

When trying to estimate an unknown value from a random variable, we can use the mean. However, it's more informative to consider more information in the form of a confidence interval. A confidence interval is a pair (range, confidence) where the range is a collection of values likely to contain an unknown value x and the confidence is a percentage likelihood that $x \in$ range.

We calculate confidence intervals with the empirical rule. The empirical rule requires two assumptions under which it will always hold:

- The mean estimation error is zero (so one is just as likely to guess as high as low

i.e. there's no reason to be systematically off in one direction or another) so there's no bias in our errors.

- The distribution of errors in estimates is normal i.e. follows a Gaussian distribution

A distribution captures the notion of the relative frequency with which some random variable takes on different values. There are two kinds of random variable:

- Discrete random variables are drawn from a finite set of values e.g. a coin flip has the outcome space $\{H, T\}$ and we can fully describe our distribution with $p(\{H\}) = a, p(\{T\}) = b$.
- Continuous random variables are drawn from a set of reals between two numbers. Take $[0, 1]$ as an example. We couldn't possibly enumerate the probability for each number in this interval because there are infinitely many of them, each of which has measure 0. Thus, the probability of a particular x occurring is 0. Instead, we define a probability density function that gives the probability of a random variable lying between 2 values e.g. $p(X \in [a, b])$.

e.g.

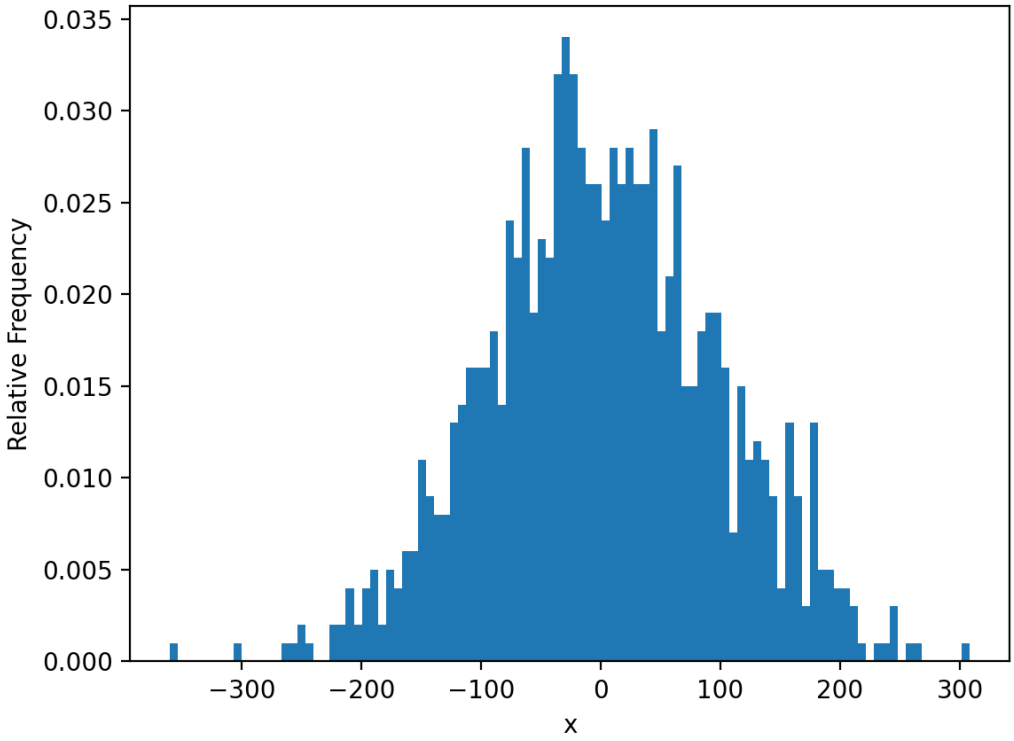
$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right)$$

This is the probability density function of a Gaussian distribution with mean μ and standard deviation σ. We also call this a Normal(μ, σ) distribution.

```
import pylab
import random
num_samples = 1000
dist = []
for i in range(num_samples):
    dist.append(random.gauss(0,100))

graph = pylab.hist(dist, bins=100, weights = [1/num_samples]*len(dist))
pylab.xlabel('x')
pylab.ylabel('Relative Frequency')
pylab.show()

print(f'Fraction within ~200 of the mean = {sum(graph[0][30:70])}')
```



What do the values of p represent? Plotting the pdf of a Normal(0, 1) distribution between -4 and 4 gives a peak to the graph at 0.4. If we changed the values of μ and σ appropriately, the peak of the y-axis will exceed 1. This means that the values of p are definitely not probabilities. Instead, p is the derivative of the cumulative distribution function and so we integrate p between certain values to find the probabilities of events.

some calculations verifying the empirical rule for the roulette example

The empirical rule works for the roulette example (20:00 in the video) but the roulette example is uniformly distributed (each outcome is equally likely in a single spin) and not normally distributed. Why is this the case? This is because we were reasoning about the return of several spins, not a single spin. As soon as we end up talking about the mean of multiple events, we can apply something called the central limit theorem.

Theorem 4.1.4 (**Central Limit Theorem**) *Given a sufficiently large sample:*

- *The means of the samples in a set of samples $\{\mu_1, \ldots, \mu_n\}$ (called the sample mean) will be approximately normally distributed.*
- *This normal distribution will have a mean that is close to the mean of the population.*
- *The variance of the sample means will be close to the variance of the population divided by the sample size.*

Chapter 5

Harnessing Randomness

Randomness can be very useful for computing quantities that are not inherently random. For example, the value of π can be approximated using areas. Buffon and Laplace proposed an algorithm for computing π. Consider a square $\mathcal{S}$ of side-length 2 and a circle $\mathcal{C}$ of radius 1 inscribed inside the square. Buffon proposed "shooting a bunch of arrows at random at the square." Ignoring those that land out of the square, we can compute the ratio:

$$\frac{\text{arrows that land in } \mathcal{C}}{\text{arrows that land in } \mathcal{S}} \approx \frac{\text{area}(\mathcal{C})}{\text{area}(\mathcal{S})} = \frac{\pi}{4}$$

Re-arranging implies that

$$\pi \approx \frac{4 \times \text{arrows that land in } \mathcal{C}}{\text{arrows that land in } \mathcal{S}}$$

Thus, we can define a function:

```
import random
def shootarrows(num_arrows):
    in_circle = 0
    for arrow in range(1, num_arrows + 1):
        x = random.random()
        y = random.random()
        if (x*x + y*y)**(0.5) <= 1:
            in_circle += 1
    return 4 * (in_circle/float(num_arrows))
```

This method can also be adapted to perform numerical integration i.e. estimating the area of a region R enclosed by a curve f and the x-axis. The method goes as follows:

- Pick a region $E \supseteq R$ whose area is easy to calculate,
- Generate a random set of points that fall in E,
- Let F be the fraction of points that fall within R,
- Return the area of E multiplied by F to get an estimate for R.

I'll stick to a non-negative function $f(x) = \sin(x)$ over $[0, \pi]$. Then $R = \int_0^\pi \sin(x)\ \mathrm{d}x$.

```
import random
import math

def num_int(f, a, b, p):
    '''f is a non-negative function over a region [a,b], where a and b are
    the endpoints of integration. p (an integer) is the number of points
    used for the estimation'''
```

```
    inside_region = 0
    enclosing_area = (b-a)*1
    for arrow in range(1, p):
        x = random.uniform(a,b)
        y = random.random()
        if y <= f(x):
            inside_region += 1
    return (inside_region/p)*enclosing_area
```

Running this code for 10, 100, 1000 and 10000 points gives the following:

```
>>>  for i in [10**d for d in range(1,5)]:
>>>      print(num_int(lambda x: math.sin(x), 0, math.pi, i))

2.5132741228718345
2.0420352248333655
2.001194520336698
1.9782608939654929
```

You can see the approximations gathering around 2 which is a promising sign because we can verify the area using basic[1] integration. Sometimes it might not be so straightforward to see if our algorithm is an accurate representation of reality. Oftentimes we can only tell if our simulation is reproducible instead of accurate.

5.1 Random Stats Knowledge

Inferential statistics is concerned with making inferences about a population (of things) by examining one or more random samples drawn from that population.

We used Monte Carlo simulation to generate many random samples to compute confidence intervals. This is all good and well when doing simulations but what happens when you're trying to do something real like taking an election poll by sampling the population?

[1]
$$\int_0^{\pi} \sin(x)\ \mathrm{d}x = -\cos(x)\Big|_0^{\pi} = -((-1) - 1) = 2$$

Definition 5.1.1

- *The* ***empirical rule***[a] *states that for a normal distribution, independent of the mean and standard deviation, the number of standard deviations from the mean required to encompass a fixed fraction of the data is constant.*
- *The* ***standard error of the mean*** *is defined as*

$$SE = \frac{\sigma}{\sqrt{n}}$$

where n is the size of the sample and σ is the standard deviation.

[a] The empirical rule is also called the $\mathbf{68-95-99.7}$ **rule**.

Chapter 6

Modelling Experimental Data

One way to understand experimental (physical, biological, social etc.) data is by fitting a model to it. We intend for the model to explain the underlying mechanism and allow us to make predictions about the behaviour in new settings.

It's always a good start to plot the data we have to see if there's an obvious relationship between the independent and dependent variables. Say we're investigating Hooke's Law: the relationship between the displacement of a spring and the weight placed on it.

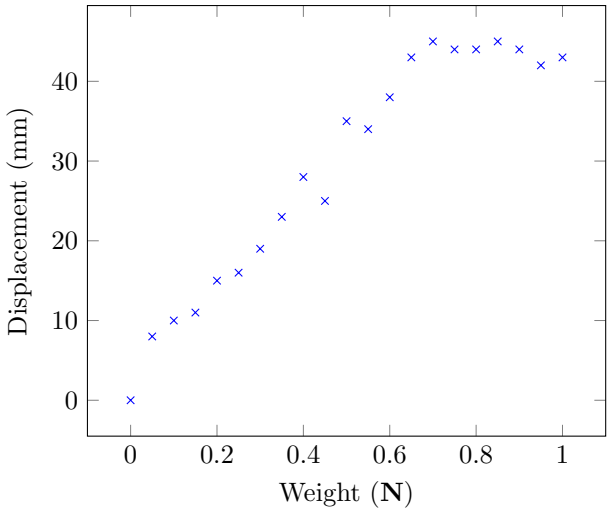


Figure 6.1: Plotting spring displacement against weight.

Experiments naturally carry errors as we can see above but it would be ideal if we could find a curve (model) that best fits the data and accounts for the uncertainty in our measurements.

A straight line would fit the first part of the data fairly well but the displacements plateau after around 0.7N (when the spring reaches its elastic limit).

A polynomial[1] would fit somewhat better. To this end, we'll be capitalising on a method **polynomial linear regression** - a special type of regression that models the relationship between a dependent variable and an independent variable as an n^{th} degree polynomial.

In order to find a curve that fits the data, we first need a way to measure how closely

[1] Why not another type of function? As a result of the Weierstrass approximation theorem, the set of polynomial functions is uniformly dense in $\mathcal{C}([0,1],\mathbb{R})$ i.e. dense with respect to the metric defined by the uniform norm $\|\cdot\|_\infty$.

our model predictions would match the observed data.

6.1 Measuring Fit

We can define functions to quantitatively measure the "difference" between the observed points and the points predicted by our curve. These are known as **objective functions**. A curve that best fits the data will ideally minimise the objective function (so this can be framed as an optimisation problem). The most commonly used objective function is called **least squares** and it turns out to be the function that is used for one of Python's polynomial linear regression algorithms.

Let[2]

- `x` represent the range of values that the independent variable takes,
- `y` be the corresponding observed values of the dependent variable, and
- `p` represent the corresponding data points that our model/curve predicts.

We define the least-squares function by:

$$\sum_{i=0}^{\texttt{len(y)}-1} \left(\texttt{y[i]} - \texttt{p[i]}\right)^2.$$

For different sets of data, a different summand might be more appropriate. For example, we could measure the horizontal distance h, the vertical distance v or the closest distance s from the observed points to the curve ℓ of best fit. s is usually helpful in classification techniques.

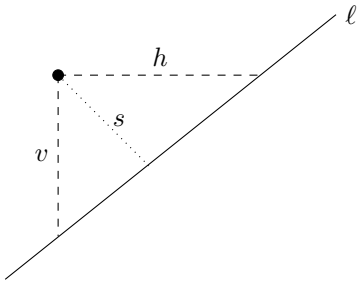


Figure 6.2: Different ways to measure fit.

For our spring example, h doesn't make much sense because the horizontal axis measures our independent variable. Since we're trying to predict the displacement given a particular mass, our uncertainty/error in measurement is precisely v.

6.2 Polynomial Linear Regression

Since polynomials are entirely defined by their coefficients $p(x) = \sum_{i=0}^{n} a_i x^i$, the task of minimising our objective function reduces to determining a set of appropriate coefficients

[2] All three lists are ordered and their length are equal.

$\{a_i\}_{0 \leqslant i \leqslant n}$. The in-built function `polyfit` from the `polyval` module can be used to find these coefficients.

Let's consider the simplest case of fitting a degree 1 polynomial to some data. The set of all possible lines we can consider is $\{\ell(x) = ax+b \colon (a,b) \in \mathbb{R}^2\}$. We can shift perspective slightly and instead consider a surface whose height at each point (a,b) is the value of the objective function at $\ell(x)$ we note that the set of all possible lines $\ell(x) = ax + b$ can be represented as a surface above the $x-y$ plane whose height above the point (a,b) is the value of the objective function.

Since our objective function is a sum of squares, our surface always has a concave shape. The `polyfit` method implements a linear algorithm starting at some point in the surface and "walks downhill" (following ∇f) until it reaches the bottom[3]. This point is where the objective function is minimised and `polyfit` returns the coefficients of our model in an array. Then we can use `polyval` to create an array of predicted values given our model and the independent variable's range of values.

```
linear_model = pylab.polyfit(xvals, yvals, 1)
linear_prediction = pylab.polyval(linear_model, xvals)
pylab.plot(xvals, linear_prediction, 'r--', label = "Linear Model")
```

6.3 Fit Comparison

Now that we have a way to generate models using polynomial linear regression, it'd be a good idea to develop some tools to decide on which polynomial best fits our data.

Sometimes there'll be a theory, like Hooke's law, that helps contextualise how good our model is. In the event that we don't have a theory, we can use our objective-function to compare the **mean squared error** (or MSE) of our fits:

$$\text{MSE}(\texttt{p}) = \frac{1}{\texttt{len(y)}} \sum_{\texttt{i}} (\texttt{y[i]} - \texttt{p[i]})^2.$$

```
def MSE(observed, predicted):
    error = 0.0
    for i in range(len(observed)):
        error += (observed[i] - predicted[i])**2
    return error/len(observed)
```

As a measure of goodness of fit, the mean squared error isn't scale independent so comparing numbers like 12000 and 1500 is somewhat meaningless. To address this, we can use a quantity known as the **coefficient of determination**. We denote it by R^2

[3]This point is unique due to the concavity of our surface.

and define it as follows:

$$R^2 = 1 - \frac{\sum_{\mathtt{i}}(\mathtt{y[i]} - \mathtt{p[i]})^2}{\sum_{\mathtt{i}}(\mathtt{y[i]} - \mu)^2} \qquad \text{where } \mu = \frac{1}{\mathtt{len(y)}} \sum_{\mathtt{i}} \mathtt{y[i]}.$$

This ratio is scale-independent and is defined to measure which portion of the variability in the data is accounted for by the model. For a model obtained via polynomial linear regression, $R^2 \in [0, 1]$. If the model completely accounts for the variation in the observations, then the numerator of the fraction is 0 i.e. $R^2 = 1$. On the other hand, if $R^2 = 0$ then there is no relationship between the values predicted by the model and the way the data is distributed around the mean.

To simplify the calculation of R^2, we can multiply the numerator and denominator by $\frac{1}{\mathtt{len(y)}}$:

$$R^2 = 1 - \frac{\text{MSE}(\mathtt{p})}{\text{Var}(\mathtt{y})}$$

The coefficient of determination should not be the only metric by which you choose a model. There's one glaring danger that we'll elaborate on by considering the following mystery set of data.

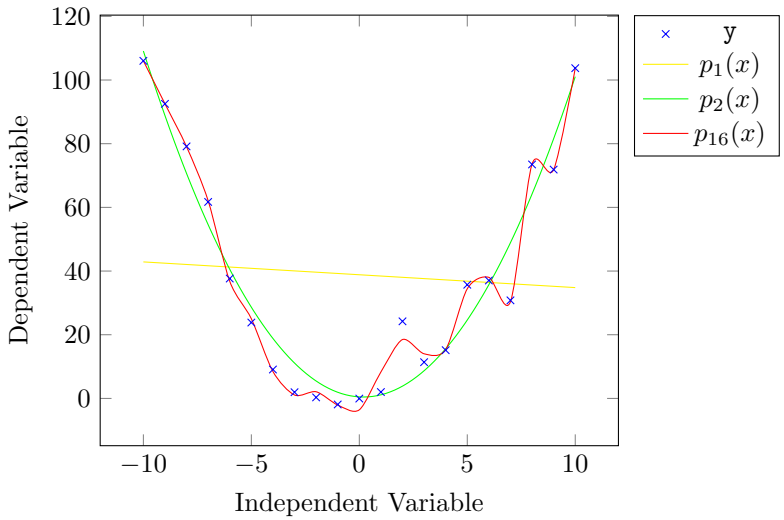


Figure 6.3: Our mystery set of data along with best-fit plots of orders 1, 2 and 16.

The line p_1 clearly accounts for variation above and below the line but it's a very poor fit for the data. The quadratic model p_2 looks a lot better by eye and follows the data reasonably well. How about higher order models? There's no underlying theory for this mystery set of data so we can rely on R^2. Let's calculate them for the following 3 models:

```
Order 1:    0.0047933537856069
```

```
Order 2:    0.9440861459030232
Order 16:   0.9961232940782585
```

The line p_1 is truly shambolic. The order 16 polynomial p_{16} has the highest value of R^2 but it's not drastically better than the quadratic model p_2. Does this make it the best model to use?

We're building models to explain the underlying phenomenon and to make predictions. Can you name any physical processes that have a 16^{th} order variation? I'd be hard-pressed to find one. Looking more closely, the higher value of R^2 for p_{16} is a result of the model fitting to not only the underlying trend but also the noise/uncertainty in the measurements. This is what we call **overfitting**.

It turns out that the mystery data was generated by taking a quadratic polynomial and adding noise ω that follows a normal distribution with $\mu = 0$ and $\sigma = 8$.

$$\mathbb{Z} \cap [-10, 10] \ni x \longmapsto (x^2 + \omega) \text{ where } \omega \sim N(0, 8)$$

This begs the question of whether or not[4] there's a better way to decide how complicated a model we need than simply eyeballing our data.

6.4 Cross-Validation

So far, we've only been seeing how well our models have performed on the (training) data from which they were learned. At the very least we'd like a small training error but as we've already seen, this isn't sufficient to get a great model. We'd also like our model to work on other datasets generated by the same process. To this end, we can generate models from one dataset and test them on another. This is known as **cross-validation**.

We expect the testing error to be greater than the training error. Let's verify this for two randomly generated datasets generated using the same quadratic process with noise as before.

```
Degree 1 model:
        Learned from dataset 1, tested on 1
                R^2 = 0.0047933537856068575
        Learned from dataset 1, tested on 2
                R^2 = -0.04280359897546249
        Learned from dataset 2, tested on 2
                R^2 = 0.01507240473346294
        Learned from dataset 2, tested on 1
                R^2 = -0.047073424270508424
Degree 2 model:
        Learned from dataset 1, tested on 1
                R^2 = 0.9440861459030232
        Learned from dataset 1, tested on 2
                R^2 = 0.8733121831961159
        Learned from dataset 2, tested on 2
```

[4]Spoiler: There is. Keep reading.

```
                R^2 = 0.9353443579003733
        Learned from dataset 2, tested on 1
                R^2 = 0.8884947292270357
Degree 16 model:
        Learned from dataset 1, tested on 1
                R^2 = 0.9961232940782585
        Learned from dataset 1, tested on 2
                R^2 = 0.8360920085887131
        Learned from dataset 2, tested on 2
                R^2 = 0.9745480042571426
        Learned from dataset 2, tested on 1
                R^2 = 0.8720430891997337
```

In all cases, we can see that the training error is less than the testing error. We also note that the value of R^2 for p_{16} makes a sharper decline than the quadratic model when we train it on one dataset and test it on another. On the other hand, the quadratic fares slightly better.

Thus, cross-validation seems to be a useful method of exposing overfitting for a model.

A small note on `polyval`: In the case of perfect data (with no measurement errors/noise), increasing the degree of the polynomial to fit the data has no effect on the coefficients returned. In the event of data with noise, after a certain point, adding higher order terms to our model approximates the noise, potentially overfitting to the training data. Despite R^2 increasing in this case, the predictive ability of our model has the potential to decline. The ideal would be to find a balance between:

- an insufficiently complex model that won't explain the data,
- and an overly complex model that overfits to the training data.

6.5 Finding the Right Model

So how do we actually go about finding an appropriate model?

One could begin with a linear model, look at the R^2 values and see how well it accounts for the data. Then increase the order and repeat the process that until you find a point at which the model does a good job of both fitting the data and predicting new data. Keep increasing the order and the metrics will eventually fall off, leaving you with a good idea of an appropriate model complexity.

In the event that you don't have a theory (like Hooke's Law) to guide you, using cross-validation is useful.

- If the dataset is small, we can use "leave-one-out" cross-validation.
- If the dataset is larger, we can use "k-fold" cross-validation (or repeated random sampling).

6.6 Mini Project

The lecturer said one could model the variation of mean daily high temperature over many years.

- Get means for each year.
- Plot them.
- Try and fit models to them:
    - For each dimensionality (linear, quadratic, cubic etc.):
        * Train on one half of the data, test on the other half. Record R^2 on the test data.
        * Report the average R^2 for each dimensionality.

Chapter 7

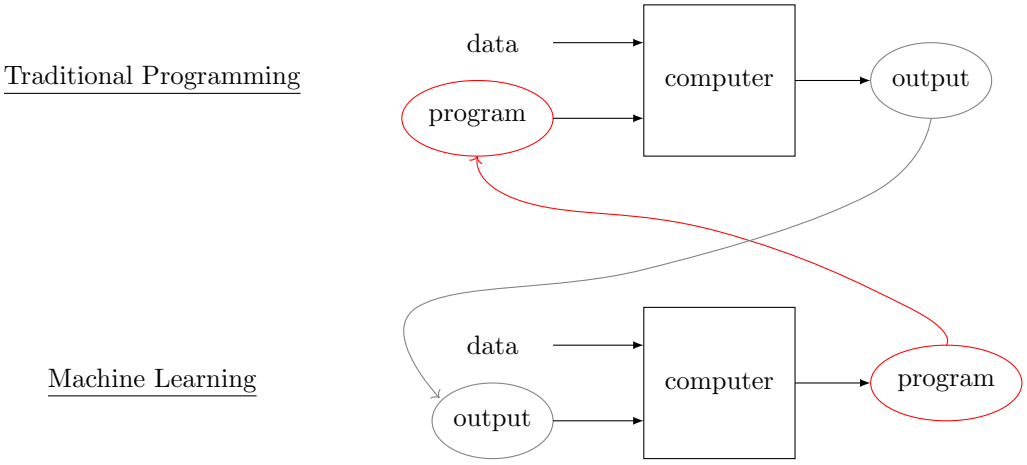
Introduction to Machine Learning

One could certainly argue that all programs learn something. In the traditional sense of programming, a program takes in data and learns the result of computing what it was instructed to do. Machine learning takes a slightly different approach and what the program learns is how to optimise itself to better achieve a certain purpose.

An early definition of machine learning can be attributed to Arthur Samuel[1] back in 1959:

> "[Machine learning is] a field of study that gives computers the ability to learn *without being explicitly programmed.*"

How can we have the computer learn without being explicitly programmed? One way to think of this is to compare how we traditionally program and what is typical from machine learning algorithms.



Traditional programming involves writing a program and inputting it to the computer so that it can take data and produce some output. Machine learning algorithms, on the other hand, entail giving the computer some output (examples of what I want the program to produce) like labels on data or characterisations of different classes of items. In turn, we want the computer to create a program that one can use to infer new information about things. This creates a nice loop in red and grey.

There are 5 essential questions that must be decided on in all machine learning methods:

[1] Computer pioneer that wrote the first self-learning program which played checkers.

1. What is the training data and how we are going to evaluate the success of the algorithm?
2. How will we represent each example i.e. what are the right features?
3. Which definition for measuring distances (similarity) between these examples will we use?
4. How can we build more detailed methods of measuring clustering/similarities to find an objective function to minimise to find the best cluster to use?
5. What is the best optimisation method to use to learn the model?

7.1 Feature Representation

Vectors will be our chosen structure for storing information about examples. They are ordered and allow for interesting manipulations by matrices (linear maps). Feature engineering comes down to deciding which features that we'd like to include in **feature vectors**. As an example, somebody's birth month is irrelevant to predicting their upcoming exam scores but their historical performance is very relevant. With more features, the danger of overfitting increases which leads to drawing less meaningful conclusions. Another way of saying this is that we want to maximise the ratio of useful to irrelevant input (the **signal-to-noise ratio**).

Once a feature has been represented in a vector, the next thing that must be decided on is how to measure similarity between examples. This comes in the form of deciding on a **metric** (distance measuring function) on our vectors. Such functions allow us to assign differing weights to certain features based on their relative importance.

> On the topic of relative importance, if the dynamic range of some features are much greater than others they will tend to dominate distance measures. For example, say we'd like to cluster people in a room given:
> - their gender which we assign 1 for male and 0 for female,
> - whether they are bespectacled (1) or not (0), and
> - their weight in kilograms.
>
> If we choose to measure distance using the Euclidean metric, their weight in kilograms would completely dominate our results. To counter this, we can scale our feature via z-scaling. For a set of features $X = \{x_i\}$ we can rescale it by replacing it with
>
> $$Y = \left\{ \frac{x_i - \overline{X}}{\sigma(X)} \right\}.$$
>
> This rescaled set has a mean $\overline{Y} = 0$ and standard deviation $\sigma(Y) = 1$.
>
> Alternatively, we can perform a linear interpolation on X. Rename our smallest and largest features to 0 and 1 respectively, and scale the remaining values linearly in $(0, 1)$.
>
> This gets all of our features in the same ballpark so we can compare them.

A commonly used metric is called the Minkowski metric. Given two feature vectors $v = (v_1, \ldots, v_n)$ and $w = (w_1, \ldots, w_n)$ of equal length, and $p > 0$ we define the Minkowski

metric d_p by

$$d_p(v, w) = \left(\sum_{i=1}^{n} (v_i - w_i)^p \right)^{1/p}.$$

If $p = 2$, we have what is the usual Euclidean metric for measuring the straight line distance between two points in space. Different values of p offer different ways of measuring the "distance" between features.

Data can be either labelled or unlabelled. This partitions machine learning algorithms into two very broad classes.

7.2 Unsupervised Learning

Unsupervised learning concerns itself with trying to group unlabelled data into "natural" groups/clusters. The most popular technique for this is known as **clustering**.

7.2.1 Clustering Overview

Clustering at its heart is an optimisation problem. The goal is to find a set of clusters that minimises an objective function under some constraints. The objective function should measure the dissimilarity of examples within a cluster and this depends on the choice of distance metric. One such measure is called variability:

Definition 7.2.1

- *We define the **variability** of a cluster C by the sum*

$$\sum_{e \in C} (d(mean(C), e))^2$$

where $mean(C)$ is the Euclidean mean of the feature vectors in a cluster and is carried out component-wise.

- *The **dissimilarity** of a set of clusters $\mathcal{C}$ is defined by the sum of each cluster's variability:*

$$\sum_{C \in \mathcal{C}} variability(C).$$

If we divided the variability by the size of C, we'd have the variance. By not normalising, we penalise big and highly diverse clusters more than smaller, highly diverse clusters.

So is our clustering problem simply about minimising the dissimilarity measure of a set of clusters? Place each example in its own unique cluster. Now the variability is 0 and the dissimilarity of the set of clusters is 0. Minimised!

Clearly this is a terrible solution and we require some constraints. We can either declare a minimum distance between clusters or an upper limit on the number of possible clusters.

There are two popular methods for unsupervised learning:

7.2.2 Hierarchical Clustering

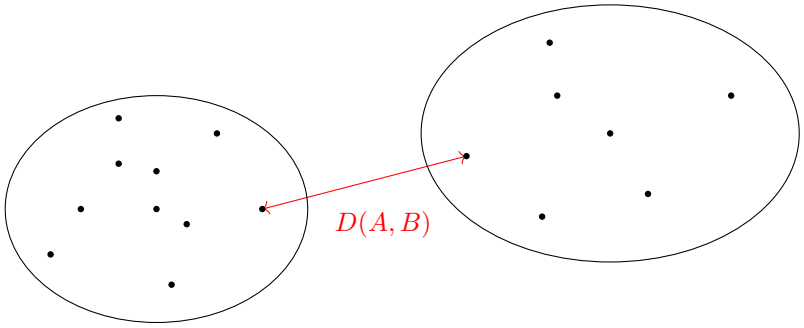
Suppose that we begin with n-items.

1. Define n clusters and place each item in a single cluster.
2. Merge the two *closest* clusters into a single cluster.
3. Continue the process until all items are collected into a single cluster of size n.

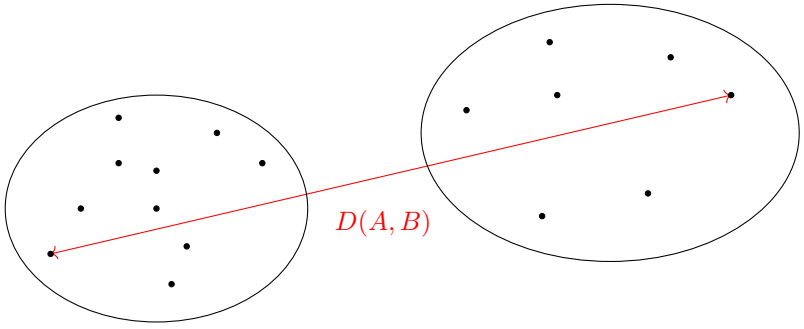
Typically, we stop this process at some point before completion. This is called agglomerative (bottom-up) hierarchical clustering. The results of hierarchical clustering are often presented as a dendrogram.

Depending on the *distance measure* used, different results are obtained. We refer to these measures as **linkage metrics**. There are several different kinds. Let A and B be two clusters.

- **Single-linkage**:
  One defines the distance between two clusters as the distance between the closest pair of objects, where only pairs consisting of one object from each cluster is considered i.e.
  $$D(A, B) := \min_{a \in A,\, b \in B} d(a, b)$$



- **Complete-linkage**:
  One calculates the distance between all pairs with each object in a different cluster and returns the greatest distance i.e.
  $$D(A, B) := \max_{a \in A,\, b \in B} d(a, b)$$



- **Average-linkage**:
  The distance between two clusters is defined as the average of the distances between all pairs of items, with each pair being made up of an item in each cluster i.e.

$$D(A, B) = \frac{\sum_{a \in A,\, b \in B} d(a, b)}{|A| * |B|}$$

Advantages of hierarchical clustering:

- You get the entire history of the process with dendrograms.
- It's a deterministic method (so doesn't depend on any random elements).
- The method is flexible with respect to the chosen linkage criterion.

Disadvantages of hierarchical clustering:

- The final result may not be optimal because the algorithm is inherently greedy in the sense that locally optimal decisions at each point don't necessarily lead to a globally optimal solution.
- It's potentially really slow, in the order of $\mathcal{O}(n^3)$ for the naive approach. For single-linkage, there do exist algorithms with quadratic complexity but this is still not very good.

A much faster greedy algorithm for clustering is known as the K-means algorithm. It is most useful when you know how many clusters you would like at the end, as opposed to the hierarchical approach.

7.2.3 K-Means Clustering

Suppose that we begin with n-items. The goal of K-means clustering is to partition our items into k clusters in way that minimises the dissimilarity of the clusters and that each example is in the cluster whose centroid is the closest to that example. We execute it as follows[2]:

1. Randomly choose k examples as initial centroids.
2. While True:
    - Create k clusters by assigning each example to its closest centroid.
    - Compute k new centroids by averaging the examples in each cluster.
    - If none of the centroids are different from the previous iteration:
        * break and return the current set of clusters

There are some issues with k-means clustering but we also list ways to ameliorate them:

- Choosing k poorly can lead to strange/unexpected results.
    - We can use a-priori knowledge about the application domain to inform our choice of k e.g. if you know there are 5 variants of bacteria

[2]There should also be a condition that raises an error if one of our clusters is empty.

- We can also search for a good k by trying different values and evaluating the quality of the results. Alternatively, we can run a hierarchical clustering on a subset of the data to get a sense of the structure underlying the data, get a k and then try k-means with that value of k.

- The results can depend quite heavily upon the initial centroids because, unlike hierarchical clustering, k-means is non-deterministic as our initial choice of centroids is random.
  - To avoid unlucky centroids, one could try and select centroids that are distributed evenly over the space.
  - A common practice for any randomised greedy algorithm involves trying multiple sets of randomly chosen centroids and choosing the best results by minimising dissimilarity of the clusters.

7.3 Supervised Learning

Supervised learning concerns itself with looking for a rule to predict the label associated with unseen input based on labelled examples which are in the form of (feature vector, label) pairs. Supervised learning can be divided into two categories:

- **Classification**: Predicting a discrete value (label) associated with a feature vector.
- **Regression**: Predicting[3] a real number associated with a feature vector.

7.3.1 k-Nearest Neighbours

One of the simplest ways to classify unseen examples is to build up a **distance matrix** of the labelled data, find the nearest existing example to the unseen one, and predict its label.

$$\begin{array}{c} \\ x \\ y \\ z \end{array} \begin{array}{c} \begin{array}{ccc} x & y & z \end{array} \\ \begin{pmatrix} 0 & d(x,y) & d(x,z) \\ d(y,x) & 0 & d(y,z) \\ d(z,x) & d(z,y) & 0 \end{pmatrix} \end{array}$$

Figure 7.1: The distance matrix of the set of examples $\{x, y, z\}$ for the metric d.

A downside of the nearest neighbour approach is that if the data is particularly noisy, you can get the wrong label by matching the unseen example to a noisy data point e.g. for character recognition, this would manifest as matching the character '0' to a particularly poorly rendered '9'.

To counter this, we can consider a (usually odd) number of nearest neighbours to our new example and "let them vote" as illustrated below:

This is called the **k-nearest neighbours** approach.

[3] We did this with polynomial linear regression.

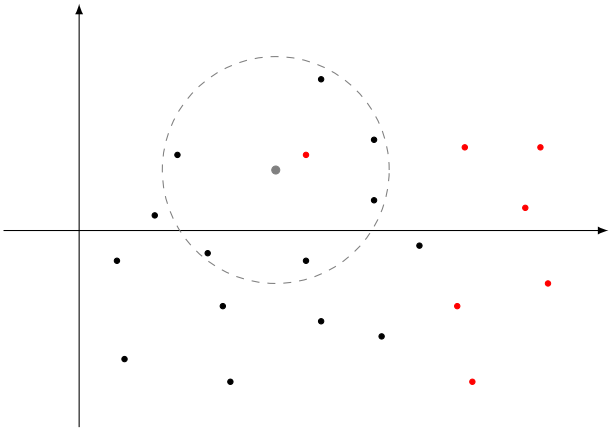


Figure 7.2: The nearest neighbour to our unseen point in grey is red but taking the 7 nearest neighbours lets the black points outvote the reds.

Downsides include:

- Efficiency: The time taken increases with larger values of k
- If k is too large, our result ends up being dominated by the most prevalent label in the training data set e.g. the black points above.

How does one choose k? We can do something similar to the k-means clustering example. Namely, we can split our training data into two parts and do cross-validation.

As a general rule, machine learning algorithms generally have a parameter (like k) and we can find an optimal parameter by searching the data and performing cross-validation on it.

Advantages of k-nearest neighbours include:

- There's no theory required.
- It's easy to explain the methods and results.
- The learning is fast and there's no explicit training in the sense that we need only remember the values in the distance matrix.

Disadvantages of k-nearest neighbours include:

- The process is memory intensive and predictions can take a long time because we need to make a lot of comparisons.
- There are better algorithms to find approximate KNN than brute force.
- We aren't getting any information on what generated the data (as we aren't finding a model like with polynomial linear regression).

7.4 Metrics For Evaluating Our Learning

The accuracy of our predictions is not a particularly meaningful measure when there's a great class imbalance e.g. A model for predicting that you don't have a rare disease (with an occurrence rate of 0.01%) has an accuracy of 0.999 but it's totally useless as a model.

Other metrics include:

- **Sensitivity** (or recall) is a measure of how good the model is at identifying the positive cases and is defined by

$$\text{sensitivity} = \frac{\text{true positives}}{\text{true positives } + \text{false negatives}}$$

- **Specificity** (or precision) is defined by

$$\text{specificity} = \frac{\text{true negatives}}{\text{true negatives } + \text{false positives}}$$

- If we say that somebody is positive for a test, what is the probability that they are actually positive? This is given by the **positive predictive value** and is defined by

$$\text{ppp} = \frac{\text{true positives}}{\text{true positives } + \text{false positives}}$$

- and the negative predictive value is just ppp with positive replaced by negative.

e.g. Consider the scenario where a clinic is running a screening test for breast cancer and they're trying to find the people who should have a more extensive examination. We'd like our model to most emphasise sensitivity since we're sending people on for future tests and we don't want to miss somebody who has cancer.

e.g. Say we're deciding on who is so sick that we should do open heart surgery on them. We'd want a high specificity because the risks of surgery are high. We don't want to be operating on people who don't need it.

As you can see, depending on the application we end up having to choose a balance of these statistics.

7.5 Testing A Classifier

Before building a classifier, it's important to have a protocol in place for testing one. There are two popular methods for this:

- **Leave-One-Out**:
  Typically used when one has a small number of examples. Consider a set of n-examples. For each example, remove it, train on the remaining $n-1$ and then test on the example you initially removed. Then average the results of this loop.
- **Repeated Random Sampling**:
  This is typically used when one has a large set of examples and it's possible to split the data, say, 80 : 20. We train on the larger subset and test on the smaller.

7.6 Logistic Regression

The final section of these notes is based on logistic regression. A logistic model is used to model the probability of an event[4] that can only take on some finite set of values (usually 0 or 1 e.g. survived/died).

Logistic regression finds weights for each feature that we'll use to make predictions. The weights can be thought of as the coefficients in linear regression.

- A positive weight implies (almost) that the variable is positively correlated with the outcome (e.g. having scales is positively correlated with an animal being a reptile).
- A negative weight implies a negative correlation.
- The absolutely magnitude is related to the strength of the correlation.

There is an optimisation process used to calculate these weights from the training data. It's quite complex and uses a *log*arithmic function. Hence, *log*istic.

To actually use it, we can import the `sklearn.linear_module` module. Inside of this module, there's a class called `LogisticRegression` which has 3 methods:

- `fit(sequence_of_feature_vectors, sequence_of_labels)`
  returns an object of type `LogisticRegression`
- `coef_`
  returns the weights of the features
- `predict_proba(feature_vector)`
  returns the probabilities of different labels

The following code builds a logistic regression model which is simply a set of weights for each of the features.

```
def buildModel(examples):
    featureVecs, labels = [], []
    for e in examples:
        featureVecs.append(e.getFeatures())
        labels.append(e.getLabel())
    LogisticRegression = sklearn.linear_model.LogisticRegression
    model = LogisticRegression().fit(featureVecs, labels)
    return model
```

Applying the model entails building a vector of features associated with a test set, making a prediction for them with `model.predict_proba` and then calculating the true/false positives/negatives given a probability:

```
def applyModel(examples):
    testFeatureVecs = [e.getFeatures() for e in testSet]
    probs = model.predict_proba(testFeatureVecs)
    truePos, trueNeg, falsePos, falseNeg = 0, 0, 0, 0
```

[4] We don't use linear regression for this purpose. Linear regression can return numbers outside of the range $[0, 1]$ which is nonsense in a probabilistic setting.

```
    for i in range(len(probs)):
        if probs[i][1] > prob:
            if testSet[i].getLabel() == label:
                truePos += 1
            else:
                falsePos += 1
        else:
            if testSet.getLabel() != label:
                trueNeg += 1
            else:
                falseNeg += 1
    return truePos, trueNeg, falsePos, falseNeg
```

What we get out of logistic regression is a probability of something having a label so we then need to build a classifier given a threshold. Then it can all be combined into a single function e.g.

```
def lr(trainingData, testData, label, prob):
    model = buildModel(trainingData)
    results = applyModel(model, testData, label, prob)
    return results
```

To access the classes we can use `model.classes_`.

Features are often correlated with one another so you can't interpret them one at a time. There are two main ways of doing logistic regression:

- L1 - focuses on finding some weights and driving them to zero which is useful for highly dimensional problems relative to the number of examples.

  e.g. If you have 100 variables and 1000 examples, you're very likely to overfit. L1 is designed to avoid overfitting. If you have two correlated features, L1 may drive one to zero making it look unimportant.

- L2 - spreads the weight across the variables so if you have a bunch of correlated variables, it might look like none of them are important.

Determining the probability that best suits one's needs can be done by varying p for a single model and accumulating a bunch of results for the metric you're interested in.

Chapter 8

Stock Simulation

The main impetus for me going through the introductory MIT computer science courses was to simulate stock behaviour. The graphs look cool and it seemed like there was a bit of statistics to it (something I haven't learned much of so why not). Now it's finally here:

There are two basic strategies for investing in the stock market:

- You could own stocks in all companies listed on the stock market. In this case, the investments you make depend on the health of the stock market as a whole. This is called having an **indexed portfolio**.
    - They don't require a lot of thought.
    - They have a low expense ratio i.e. you don't need to hire somebody brilliant to manage your portfolio.
- The other option is what I typically think of when it comes to stocks. You hire somebody intelligent and pay them to pick winners for you and outperform the stock market. This is called a **managed portfolio**.

Since the goal is to simulate the market, we need some assumptions upon which we build our model. We'll assume that the market is **informationally efficient**[1] i.e. "current prices of stocks reflect all publicly known information and are therefore unbiased." This is what is known as the **efficient market hypothesis**.

This effectively means that the market is memoryless. It doesn't matter what the stock price was yesterday. Today it's priced given the best known information and so tomorrow it's equally likely to increase or decrease in value relative to the whole[2] market. Under this hypothesis, no stock is more or less likely to outperform the market because all known information is already incorporated into the price...

That sounds off. Some stocks are certainly more volatile than others and can be modelled by different distributions. The model we'll choose for a stock is a **random walk**. Then we can build up a custom market class as a collection of stocks and methods.

Several things are incorporated into modelling a stock:

- The initial price
- Some way of storing the price history of a stock
- A way of keeping track of the most recent change in value

[1] If people thought a stock was underpriced, they would've bought more of it and the stock price would've already risen and been accounted for in the model.

[2] It's well known that over periods of multiple decades that the market has a tendency to go up so there's an upward bias to the stock market (seemingly contrary to media portrayal).

- Some way of capturing the behaviour of a stock's price movement according to how volatile it is.
    - We can simulate a stock's movement by pulling random samples from a distribution. For example, we can choose a volatility constant (that can be thought of as a percentage move per day) and create a distribution which whenever called returns a selected value between ±volatility.
- Implementing a way to change the price of a stock
- A method for plotting the history of a stock's movements

```
class Stock:
    def __init__(self, price, distribution):
        self.price = price
        self.history = [price]
        self.distribution = distribution
        self.lastchange = 0.0

    def setprice(self, newprice):
        self.price = newprice

    def getprice(self):
        return self.price

    def makemove(self, marketbias, momentum):
        oldprice = self.price
        basemove = self.distribution() + marketbias
        self.price *= (1.0 + basemove)
        if momentum:
            self.price += momentum*random.gauss(.25,.25)*self.lastchange
        if self.price < 0.01:
            self.price = 0.0
        self.history.append(self.price)
        self.lastchange = self.price - oldprice

    def showhistory(self):
        pylab.plot(self.history)
        pylab.title('Stocks')
        pylab.xlabel('Day')
        pylab.ylabel('Points')
```

There are some important points to make about `makemove`:

- The price adjustment in line 17 is multiplicative as opposed to additive because stock price changes are not independent[3] of stock price. As an example, a Google stock priced at 300 points is much more likely to move 10 points than a stock priced at 5 points.
- Some modellers believe in the idea of **momentum** in the stock market i.e. what's more likely to happen today is what happened yesterday. This brings up an interesting question on whether or not stock prices are memoryless (also known as

[3]This is informed by looking at historical data. The amount a stock moves tends to be proportional to the price of the stock and, interestingly enough, the percentage moves for expensive and inexpensive stocks don't differ much.

Poisson[4]).

[4]A Poisson process is one in which past behaviour has no effect on future behaviour.