MITx 6.00.2x

Introduction to Computational Thinking and Data Science

Lecture Videos - Notes

Table of Contents

UNIT 1

A **computational model** is an experiment modeled by computers to help us understand the world in which we live (such as by helping us understand things that have happened or will happen in the future). Three common types of computational models are *optimization models*, *statistical models*, and *simulation models*.

Optimization models are computational models useful for solving problems that have an upper or lower limit to their answer. Optimization models consist of (1) an objective function that is to maximized or minimized (e.g. a problem that asks a "*what is the largest possible…"* or "*what is the smallest possible…*", etc. type of question), and (2) a set of constraints (which may be none) that determine within which limits the problem is to be solved.

- Although optimization models can be used to solve many different problems, they are often computationally challenging because they can take long periods of time or large amounts of computing power to solve a problem. Because of this, many optimization models use "greedy" algorithms to approximate the solution to a problem.

The **knapsack problem** is an example of a problem that can be solved through an optimization model. In the knapsack problem, there is a knapsack (or some sort of container) that can hold a finite number of things and carry a finite amount of weight, and there is a set of available items

of varying value and weight that are to be put in the knapsack. However, there are more items available than the knapsack can accommodate. The goal of the knapsack problem (in the *0/1 variation* of the knapsack problem, where *whole* items must be put into the knapsack, not a half or a quarter of item, for instance, as in the *continuous* or *fractional variation*) is to find the combination of items in a set that will fit in the knapsack and have the highest value.

- A possible but impractical method of solving the knapsack problem is to use a brute force algorithm to calculate every possible combination of items from the items available (called generating subsets of the *power set*, or total available items), eliminate all sets with a weight greater than the knapsack can carry, and select from the remaining sets the set with the highest value. This solution, however, is inherently exponential, meaning for a power set of 100 items, for instance, the total number of possible subsets would be 1,267,650,600,228,229,401,496,703,205,376 (something that would take a very long time to calculate!).

- Another possible method to solve this problem is to use a **greedy algorithm** (an algorithm that makes the “best” possible choice at any given moment, regardless of whether that choice will prove to actually be the “best” choice in the long run). Solutions that use greedy algorithms may seem like decent options since they are often computationally efficient (unlike a brute force algorithm) but they do not always give optimal solutions to the problem at hand. In fact, it is often difficult to even approximate how close a greedy algorithm’s solution is to the truly optimal solution.

- Although at times impractical, using a brute-force algorithm to solve the knapsack problem is still a possible method that will definitely give a truly optimal solution. One way to implement a brute-force algorithm to solve the knapsack problem is by using a **search tree** to enumerate all possible combinations of objects in the knapsack. A search tree is a data structure that begins at a root (which is in this case, all of the possible items that can be put into the knapsack), and branches downward into nodes (possibilities of combinations of items that can be put into the knapsack) and then eventually into leaves (the bottommost nodes in the search tree). Once every possible combination of items has been represented by the search tree, the combination that obeys the constraints of the knapsack problem and has the highest value can be selected as the truly optimal solution.

  (In a search tree (in the context of the knapsack problem), the first item from the group of available items is selected. If this item can be put into the knapsack, a node showing the possibility of taking that item is drawn downwards to the *left* of the root. Then, the next item in the group of available items is selected. If this item can be put into the knapsack, a node showing the possibility of taking that item (and any items already in the knapsack) is drawn downwards to the *left* of the previous node. This process is repeated until there are no more items available that will fit into the knapsack, at which point one must work up the search tree and fill in the *right* sides of the nodes with the possibilities

of not taking the items taken by the *left* sides of the nodes. Essentially, nodes drawn to the *left* of a previous node reflect the consequences of putting an item into the knapsack while the nodes drawn to the *right* of a previous node reflect the consequences of not putting that same item into the knapsack. This process of drawing left and right nodes continues under every left and right node until there are no more available items to take or not take.)

However, as mentioned before, this solution is exponential, since the time that it takes to construct a search tree depends on the number of nodes generated, which depends on the number of levels in the search tree (the number of nodes at level i is 2^i), which is the same as the number of available items. An algorithm that would construct a search tree would be of the order 2^{n+1}.

…

While using a search tree algorithm does provide a definite globally optimal solution, it is not a time efficient solution for large problems because of its exponential growth rate. However, some solutions that have large growth rates (like this search tree algorithm) can be optimized using **dynamic programming**, particularly those that are recursive in nature. For example, consider the following code:

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

The rather simple function shown above uses recursion to find the *n*th number in the Fibonacci sequence. This function works fine for small values of *n*, but quickly takes more and more time to calculate larger values of *n* (at some point becoming too inefficient to be useful). The reason the function is so inefficient is because it continually has to recalculate Fibonacci numbers it has already calculated (because of its recursive nature). Using dynamic programming, this issue can be resolved by creating a list that stores all of the Fibonacci numbers already calculated by the function so that the function does not have to redundantly recalculate the same values:

```
def fib(n, memo = []):
    if n == 0 or n == 1:
        return 1
    try:
        return memo[n]
    except KeyError:
        result = fib(n - 1, memo) + fib(n - 2, memo)
        memo = result
```

```
        return result
```

The function shown above checks to see whether or not the value it is supposed to calculate is already stored in the list `memo`. If it is, it returns that value, if not, it calculates that value and then stores it in the list. This process of storing and retrieving values from some sort of table in order to reduce the complexity of an algorithm is called **memoization**. Because of this, the function above will almost instantaneously return solutions that would have taken years to calculate using the previous function.

Dynamic programming does not always simplify complex algorithms, however. Dynamic programming works best when (1) finding the globally optimal solution to the problem can be found by combining locally optimal solutions from smaller, local subproblems (this is called having an **optimal substructure**, and in this case, was the *n*th Fibonacci number being found by finding the smaller Fibonacci numbers before it), and (2) finding the optimal solution to the problem involves solving the same problem many times (in this case, solving for the same Fibonacci number multiple times).

"In essence, many problems of practical importance can be formulated as optimization problems, the solutions to which can be found adequately (but not necessarily optimally) by greedy algorithms. Although finding the optimal solutions to such problems by using greedy algorithms is often inherently exponential, dynamic programming can be used to drastically improve the performance of a subclass of such optimization problems—those with an optimal substructure and overlapping subproblems. This will always give us an optimal solution and, under the right circumstances, have good performance as well." (Paraphrased.)

…

Graphs are visual representations of data that are useful for laying out information and solving problems. Graphs often consist of *nodes* (or points) that have some sort of properties associated with them, and *edges* (or arcs) that connect those nodes to each other. *Undirected graphs* have no symmetrical structure between their nodes while *directed graphs* have a definite, symmetrical structure between their nodes (as in a tree diagram, where there is a single parent node and numerous child notes and each pair of nodes is connected by a single path). Since the world is filled with networks based on relationships, graphs can be used to efficiently solve real-world problems because they abstract away irrelevant details and leave us with data we can actually use to find solutions to optimization problems.

- The **depth-first search algorithm** is an important algorithm used to solve graph problems. In a depth-first search, the algorithm begins at a particular node and travels to one of the children of that node, and then to one of the children of that node, and then to one of the children of that node, and so on and so forth until there are no more children to travel to. After evaluating this path, the algorithm travels up to the most recently travelled node and travels down any children of that node which haven't been visited yet

to create a new path to evaluate. If all the children of a particular node have been visited, the algorithm travels up to the most recently travelled node and repeats this process. In this way, all possible paths in a graph can be found and evaluated to find a solution to a problem (or, each node in a graph can be checked to see if it is the solution).

In the case that there are cycles in the graph (nodes that connect to other nodes that connect back to the previous nodes in some way), the algorithm may keep track of which nodes have already been visited to safeguard against looping back to the same nodes over and over again.

- The **breadth-first search algorithm** is another important algorithm used to solve graph problems. In a breadth-first search, the algorithm explores multiple paths at once (instead of a single path like the depth-first search algorithm) by exploring and evaluating the path created by each child node of a parent node separately before moving on to exploring the child nodes of those child nodes afterwards (e.g. the algorithm will evaluate each individual path made from a parent node to each of its child nodes to see if it has found a solution before checking each individual path made from each one of those child nodes to each one of its child nodes to find a solution there, and so on and so forth). In this way, a solution to a problem may be found without actually exploring every path in a graph to its fullest extent (since, once a solution is found, there is no need to continue searching for solutions in the child nodes of the solution node).

…

In Python, a **lambda** is a small, anonymous function that can take any number of arguments but only have one expression. They are written using the following syntax:

```
lambdaName = lambda arg1, arg2, arg3 : <expression>
```

The expression after the colon indicates what value should be returned by the lambda. For example, the following lambdas return the sum of an argument and an integer, the product of two arguments, and the sum of three arguments, respectively:

```
addTen = lambda x : x + 10

multiply = lambda x, y : x * y

addThese = lambda x, y, z : x + y + z
```

Lambdas are called just like regular functions:

```
addTen(10)      # will return '20'
```

```
multiply(4, 8)      # will return '32'

addThese(7, 8, 9)      # will return '24'
```

Conditionals can be used in lambda expressions to create more complicated expressions:

```
lambda x: "positive" if (x >= 0) else "negative"
```

The lambda shown above returns the string `"positive"` if the argument it is passed is greater than or equal to zero; otherwise, it returns the string `"negative"`. As shown, conditional expressions within lambdas are often surrounded by parentheses to indicate when the conditional expression begins and ends.

Although it is possible to write very long or complex lambda expressions, it is generally a good practice to use a regular function instead of a lambda if the lambda expression is longer than a line of code.

...

Both the `.sort()` and `sorted()` list methods have an extra parameter called `key` that accepts a single function as an argument, applies that function to each element in the iterable being sorted by the `.sort()` or `sorted()` methods (prior to the methods being run), and then allows the `.sort()` or `sorted()` methods to sort the iterable based on the function given as the argument for the `key` parameter. Although the function given as the argument for the `key` parameter is applied once to every item in the iterable being sorted, the items in the final sorted iterable are the same as they were in the original iterable (they are only sorted, not modified by the function). For example:

```
sorted("Everything is awesome!".split())
```

The above line of code returns a sorted list of the words in the string `"Everything is awesome!"` (since the `.split()` method splits the string at every whitespace). However, the word *Everything* is placed before the word *awesome!* even though there aren't in alphabetical order because `Everything` begins with an uppercase letter:

```
['Everything', 'awesome!', 'is']
```

In order to sort the words of the string in alphabetical order in spite of their uppercase and lowercase letters, we can pass the `.lower()` method as the parameter `key`:

```
sorted("Everything is awesome!".split(), key=str.lower())
```

The `key=str.lower()` will apply the `.lower()` function to each word in `"Everything is awesome!"` and the `sorted()` method will then sort the string according to lowercase letters of every word (alphabetical order):

```
['awesome!', 'Everything', 'is']
```

Since the `key` parameter only accepts a function as an argument, it is common to pass lambda expressions to the key parameter instead when there are no functions available to achieve the intended result:

```
student_tuples = [
    ('john', 'A', 15),
    ('jane', 'B', 12),
    ('dave', 'B', 10),
]
sorted(student_tuples, key=lambda student: student[2])   # sort by
age
```

The above code example from the Python 3 documentation uses a lambda to sort a tuple of lists according to the item at the index `2` in each list in the tuple (in other words, the age of each student in the example), outputting:

```
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

...

Bitwise operators are operators in Python that perform special operations on binary numbers. The bitwise `AND` operator (represented by `&`) compares the digits of two binary numbers and returns a binary number whose digits are ones if the corresponding numbers in the original two binary numbers are ones or zeroes if the corresponding numbers in the original two binary numbers are either a one and a zero each or both zeroes:

```
a = 50        # 110010
b = 25        # 011001

c = a & b     # 010000
```

In the above example, the first digit of the binary value of the variable `c` is `0` because the first digits of binary values in the variables `a` and `b` were `1` and `0`, respectively. The second digit of the binary value of the variable `c` is `1` because the second digits of binary values in the variables `a` and `b` were `1` and `1`, respectively. (And so on and so forth for the rest of the number.)

The bitwise `OR` operator (represented by `|`) is similar to the bitwise `AND` operator except that when comparing the digits of two binary numbers, it returns a binary number whose digits are zeroes if the corresponding numbers in the original two binary numbers are zeros or ones if the corresponding numbers in the original two binary numbers are either a one and a zero each or both ones (the opposite of what the bitwise operator `AND` does):

```
a = 50        # 110010
b = 25        # 011001

c = a | b     # 111011
```

In the above example, the first digit of the binary value of the variable `c` is `1` because the first digits of binary values in the variables `a` and `b` were `1` and `0`, respectively. The fourth digit of the binary value of the variable `c` is `0` because the fourth digits of binary values in the variables `a` and `b` were `0` and `0`, respectively. (And so on and so forth for the rest of the number.)

The `<<` bitwise operator shifts all of the digits in a binary number left by a number of specified spaces. The empty spaces created on the right of the binary number are filled in with zeros:

```
a = 50        # 110010

c = a << 2    # 001000
```

In the above example, all of the digits of the variable `a` are shifted right two spaces (as indicated by the `2` in `c = a << 2`) to the left. *The result is the same as multiplying* a *by* 2^b *(as in* `a * (2**b)`*).*

The `>>` bitwise operator shifts all of the digits in a binary number right by a number of specified spaces. The empty spaces created on the left of the binary number are filled in with zeros:

```
a = 50        # 110010

c = a >> 2    # 001100
```

In the above example, all of the digits of the variable `a` are shifted right two spaces (as indicated by the `2` in `c = a >> 2`) to the right. *The result is the same as dividing* a *by* 2^b *(as in* `a // (2**b)`*).*

The `~` operator flips the zeros and ones of a binary number, making all of the zeros ones and all the ones zeros:

```
a = 50        # 110010
```

```
c = ~a        # 001101
```

...

In Python, recursive functions have a recursion limit (a limit to the number of times a recursive call can be made) that will output a `RecursionError` if exceeded. The sys module contains two functions (`getrecursionlimit` and `setrecursionlimit`) that retrieve the default recursion limit and update its value, respectively:

```
import sys

getrecursionlimit()      # returns the default recursion limit

setrecursionlimit(2048)      # sets the recursion limit to '2048'
```

UNIT 2

For many years, philosophers and scientists have debated as to whether the physical world may be inherently predictable or unpredictable. Some have argued that while certain processes in the world are highly likely to occur, they can never occur for certain, while others have argued that certain processes in the world can be accurately predicted based on logic and reasoning. Whether the world may be inherently predictable or unpredictable, our lack of knowledge often prevents us from making accurate predictions, and so in many cases, it may be useful to simply treat the physical world as inherently unpredictable. This method of thinking is known as *predictive nondeterminism*. The following are a few different types of models and their respective characteristics:

- **Deterministic Model** - "is one whose behavior is entirely predictable. Every set of variable states is uniquely determined by parameters in the model and by sets of previous states of these variables. Therefore, these models perform the same way for a given set of initial conditions, and it is possible to predict precisely what will happen." *Essentially, a model that has an entirely predictable behavior based on a certain set of inputs.*

- **Stochastic Model** - "is one in which randomness is present, and variable states are not described by unique values, but rather by probability distributions. The behavior of this model cannot be entirely predicted." *A model with some degree of predictability, but an element of randomness as well.*

- **Static Model** - "does not account for the element of time. In this type of model, a simulation will give us a snapshot at a single point in time." *A model that doesn't account for time.*

- **Dynamic Model** - "does account for the element of time. This type of model often contains state variables that change over time." *A model that does account for time, often having variables that change over time.*

- **Discrete Model** - "does not take into account the function of time. The state variables change only at a countable number of points in time, abruptly from one state to another." *A model that does not account for time, but having variables that change at certain points over time (yet the points at which they change are abrupt and random).*

- **Continuous Model** - "does take into account the function of time, typically by modelling a function f(t) and the changes reflected over time intervals. The state variables change in an unbroken way through an infinite number of states." *A model that does account for*

time, having variables that change at certain, predictable points in time (the points at which they change are in a sequence and unlikely to be random).

Whenever randomness is introduced into a model, the subject of **probability** becomes important. Probability can be calculated using mathematical formulas and counting (*actual probability*) and/or by using programs to run simulations and record the results to find an estimate of the actual probability (*sample probability*). The sample probability of a model usually closely matches the actual probability of the model, although when measuring the frequency of the occurrence of a rare event, it can take a large number of trials to get an accurate estimate of the sample probability.

A **simulation model** is a collection of computations that provides useful information about the particular system being modeled. Simulation models are useful because they allow us to model systems that would be difficult or laborious to model using simple equations and mathematics and can successively be refined (such as by using conditionals or "what if" questions). However, simulation models are only an approximation of reality, and, as George Box said it, "*All models are wrong, but some are useful.*" A simple way to begin exploring simulation models is by working with **random walks**, or, processes that are completely random and are subject to little order in their behavior.

…

The built-in Python library random contains a number of useful features for incorporating random elements into Python code. The `.randrange()` function has parameters similar to the `range()` function (`start`, `stop`, `step`) but returns a random integer between the range of values specified (and does not actually build range object):

```
import random

random.randrange(0, 101, 2)   # returns a random int between 1 and
100
```

The `.randint()` function does nearly the same thing as the `.randrange()` function, returning an integer equal to or greater than/less than its two parameters:

```
random.randint(0, 101)      # returns a random int between 0 and 101
```

The `.uniform()` function does nearly the same thing as the `.randint()` function, returning a *float* equal to or greater than/less than its two parameters:

```
random.uniform(0, 10)      # returns a random float between 0 and 10
```

The `.choice()` function returns a random element form a non-empty sequence:

```
random.choice(["spaghetti", "lasagne", "pizza", "calzone"]
```

The `.random()` function returns a random float from `0.0` up to (but not including) `1.0`:

```
random.random()
```

The `random.sample()` function takes in a sequence and returns a number (specified by its second argument) of random items in that sequence (while leaving the sequence unchanged):

```
random.sample(my_list, 3)       # returns 3 random items from my_list
```

One useful method for debugging is the `.seed()` method, which, when called with an integer argument of `0`, makes the `.random()` function (as well as many other functions in the `random` module) always return the same set of "random" numbers each time the code is run (or at least each time `.seed(0)` is called):

```
random.seed(0)
```

UNIT 3

Inferential statistics is a branch of statistics that seeks to estimate certain characteristics of a **population** (a group of examples) based on characteristics gathered from one or more **samples** (sections) of that population. If a sample is taken at random from the population, that sample tends to exhibit the same characteristics as that of the entire population. The confidence placed in such estimates often depends on the size of the sample (e.g. the larger the sample the more accurate representation of the population) and the variance of the sample (i.e. varying amounts of variance in a sample may lead to different conclusions about the population). The individual elements of a sample are called **sample points**.

The **law of large numbers** (also known as *Bernoulli's Law*) says that the difference between the average of all the outcomes of a series of probability experiments and the actual probability of the experiments will come closer to (and eventually become) zero as the number of trials of those experiments approaches infinity. This means that when performing a series of probability experiments, deviations from the average results may occur but will eventually be offset by more average results in the future (this holds true even for extreme deviations—results that differ significantly from the expected average—called **outliers**). This means that while unexpected results may occur in a series of experiments, as the number of trials of experiments increases, more and more expected results will occur and the total of all the results of the experiments (both unexpected and expected) will eventually equal the actual probability of those things happening. This does not mean that extreme deviations will be eventually offset by opposite extreme deviations later on (this common misconception is known as the *gambler's fallacy*)—only that extreme deviations will be eventually offset average deviations over time (a concept known as *regression to the mean*, which states that following an extreme random event, the next event is likely to be less extreme).

- It is important to keep in mind that it is never possible to guarantee complete *accuracy* of an estimate of the characteristics of a population through simply looking at a sample (unless the entire population is examined and compared to the estimates), but an estimate may still be *precisely correct* (i.e. when flipping a fair coin it is safe to say that either side has a 50% chance of landing face up).

Variance is the overall "spread" of the values that make up a group. The square root of the variance is known as the **standard deviation**. The standard deviation of a group of values can be calculated by squaring the difference between each value and the mean value, and then taking the square root of the average of those squares. A group of values having most of its values close to the mean value will have a low standard deviation, while a group of values having most of its values far from the mean value will have a high standard deviation.

- It is important to note that the standard deviation should always be viewed relative to the mean, since the size of the standard deviation may be big or small depending on how large the mean is (e.g. if the mean is 10 and the standard deviation is 0.2, the standard deviation from the mean would be rather small; however, if the mean is 1,000,000 and the standard deviation is 0.2, the standard deviation from the mean would be extremely small, meaning most of the values are very close in value to each other.

Often, instead of estimating a single value for a certain characteristic of a population, a range in which the actual value is likely to be contained is estimated instead, accompanied by a measure of how likely the actual value will be contained within that range. The range of the values of where the actual value may fall into is known as a **confidence interval**, and the measure of how likely that value will fall within that range of values is known as the **confidence level**. The confidence interval is often determined by the **margin of error**, a positive/negative value of how far from the estimated value the actual value may be (e.g. if the estimated value is 12.5 and the margin of error is +/-1.2, the confidence interval is 11.3 - 13.7; a 95% confidence level that the expected value will fall within that range means that if this experiment were performed an infinite number of times, the expected value will fall within that range 95% percent of the time).

- The confidence level of a confidence interval can be determined by the **empirical rule**, which says that around 68% of the values in a group are within one standard deviation of the mean, around 95% of the values in a set are within two (or more technically, 1.96) standard deviations of the mean, and around 99.7% of the values in a set are within three standard deviations of the mean (e.g. if the standard deviation of a set is 0.8 and the mean is 3.2, around 68% of the values of that set are within the range 2.4 - 4.0). This means that by looking at a random sample drawn from a population and calculating the standard deviation of that sample, one can figure out in what confidence interval and with what confidence level a particular value can be found. The empirical rule holds true for nearly all cases where (1) the standard distribution of the values in a set is *normal* (see below) and (2) the average error of the estimates of the data is also equal to zero.

When determining a **probability distribution** (i.e. the distribution of a group of probabilities, say, for a particular value), there are two kinds of probability distributions to consider: a *discrete probability distribution*, which maps exact values of probability (e.g. the probability of rolling a particular number on dice, a scenario in which the probability is an exact number) from a finite set of values, and a *continuous probability distribution*, which maps an estimation of particular values of probability within a range in which those estimations are likely to be found (e.g. the probability of a car moving at an exact speed such as 50.000000… mph—a scenario in which the speed of the car is constantly changing and the probability of that car moving at that exact speed is extremely low). Probability distributions are often represented visually through **probability density functions** (PDFs), which show the probability of some variable lying between two values. A PDF defines a curve where the range of the x-axis is marked by the maximum and minimum values the variable can be and the area below the curve represents the probability of the variable falling within that range of values (i.e. the higher the curve at a

particular point in the x-axis, the more likely the unknown value will be at that same point in the x-axis, and vice versa).

- **Normal distribution** (also called *gaussian distribution*) simply means that, in a graph of a set of data points, there is a bell-shaped curve centered around the mean of the data and whose sides gradually slope downward (asymptotically) towards zero). In contrast to normal distribution, a graph of **uniform distribution** simply has a straight line along the x-axis of the graph throughout the graph since in uniform distributions, all values have an equal probability (of, say, being chosen). There are several other types of distributions, such as exponential, poisson, and skellam distributions.

Besides the empirical rule, the other perhaps most important concept in statistics is known as the **central limit theorem**, which states that given a large enough number of samples, the individual means each of those samples from the large number of samples (called *sample means*) will be approximately normally distributed when plotted with the rest of the sample means gathered. This means that although all of the values in, say, a uniform distribution have an equal probability of being chosen, the means of enough random samples taken from that uniform distribution will still be normally distributed (e.g. if you took a random sample from that uniform distribution and calculated its mean, then repeated this process a large number of times, the means that you would get would be *normally distributed* (for example, when graphed)—even though they were all taken from a *uniform distribution*). This is because the samples (whose means are calculated and plotted) are drawn at random from the population, meaning the sample points that make up those samples are also drawn at random, therefore producing a normal distribution when plotting the means of all of the samples taken.

- Additionally, the central limit theorem states that the mean of this normal distribution will be close to the mean of the original population and that each of the sample means will have a variation close to the variance of the original distribution divided by the size of the sample. The central limit theorem applies not only to uniform distributions but to all distributions (including those that are exponential, poissonian, etc.), *meaning that as long as there are a sufficient number of random samples, the shape of the distribution from which they are taken from does not matter—data inferences can still be made about the mean of the original population by using confidence intervals and confidence levels defined by the empirical rule applied to the normal distribution of the sample means taken from that population.*

Monte Carlo simulations are a class of computational algorithms that perform repeated simulations of a probabilistic model (with certain parameters changing at random in each simulation) in order to obtain numerical results which can be systematically analyzed. Monte Carlo simulations are often used to solve probabilistic problems that would be difficult or impossible to solve otherwise. For example, a Monte Carlo simulation can be executed by setting up a model with some elements(s) of randomness, running that model a large number of

times with the randomly changing parameters, and analyzing the results obtained (such as to make statistical inferences from the stochastic model).

In *probability sampling*, each element in a population has a non-zero chance of being chosen in a random sample (e.g. every element has *some* probability of being chosen). In some probabilistic models, this means that each sample in the population has an equal probability of being chosen as a sample to be examined. This, however, is not always desirable—such as in cases where the population is divided into smaller subgroups of varying sizes and choosing a random number of samples from the population may result in more samples being chosen from larger subgroups, leaving some of the smaller subgroups underrepresented in proportion to the size of the entire population. In response to this, **stratified sampling** divides a population into subgroups and then takes a random sample from each subgroup proportional to the size of the subgroup in relation to the rest of the population.Those samples can then be summed up proportionately to get an accurate estimate of a characteristic of the entire population.

When taking a large number of random samples from a population and using the central limit theorem to make inferences from the means of those samples, the larger the size of each of the random samples, the smaller the confidence interval will be that can be taken from a plot of the means of those samples. (A useful way to visualize confidence intervals is by using **error bars**, which are small dots that represent the mean of the sample with (usually) vertical bars that extend above and below those dots showing the extent of the positive and the negative values of the confidence interval.) *This means that taking a smaller number of random samples with a larger sample size of each sample usually makes a much bigger difference in making the confidence interval of the means of those samples smaller as opposed to taking a larger number of random samples with a smaller sample size for each of the samples.*

Although useful, it is sometimes not practical (or even possible) to take many random samples from a population and make inferences about the population from those samples. Oftentimes, inferences about a population must be made from a single random sample drawn from the population. This can be done by choosing a single random sample of an appropriate sample size (as determined by an estimate of the *skew* of the population), calculating the mean and standard deviation of the sample, using that information to estimate the *standard error*, and then using the estimated standard error to make confidence intervals around the mean of the sample.

- The **standard error of the mean** is the standard deviation of a distribution of means from random samples taken from a population. It can be calculated by taking the standard deviation of the population and dividing it by the square root of the size of the random sample taken from the population. As in the steps shown above, the standard deviation of the population is not always known, meaning to calculate the standard error of the mean, the standard deviation of the population must be estimated using the standard deviation of the random sample (just like the mean of the population can be estimated using the mean of the random sample). As long as the random sample is of a

sufficient size (not necessarily in proportion to the population), the standard deviation of the sample will be close to the standard deviation of the population.

- However, choosing a sufficient sample size for the standard deviation of a random sample to closely match that of the standard deviation of the population from which it was chosen from, the distribution of the population makes a difference. This difference is determined by the **skew** (the measure of symmetry in a data plot, say, like a histogram) of the data samples of the population (e.g. a sample drawn from a population with a uniform distribution will often have a standard deviation closer to that of the population than a sample drawn from a population with an exponential distribution, since the former has a more symmetrical distribution than the latter). The size of the population has little effect on closeness of the standard deviation of the sample to that of the population (most of the difference depends on the skew of the population).

...

`NaN` means “not a number” and is useful in some instances for returning something with no particular value in Python. NaN can be created using the following syntax:

```
float(“NaN”)      # or float(“nan”) or float(“NAN”)... etc.

import math

math.nan
```

The `.gauss()` function from the `random` module returns a random number sampled from a normal distribution of which the mean value is the first argument and the standard deviation is the second argument of the `.gauss()` function:

```
import random

random.gauss(0, 30)      # returns a random number from a normal
                           distribution having a mean value of 0 and a
                           standard deviation of 30
```

UNIT 4

When gathering data from the world around us, it is often useful to plot that data and find some sort of curve that fits the data plotted (this is known as **curve fitting**). However, because the data we collect may not always be accurate due to various reasons, the results that are plotted may not match that of our expectations. For instance, we may expect a plot of our data to resemble that of a linear line but instead find the data points in our plot to appear rather randomly distributed and to only vaguely resemble a linear line. At other times, we may have no idea of what sort of curve will fit a plot of our data the best—and so there are actually mathematical ways of going about figuring out what sort of curve fits the plot of our data the best (i.e. linear, quadratic, exponential, etc.).

- The goal of this optimization problem is to find a line where the sum of the distances between the line itself and the data points of the plot is as small as possible (e.g. the less the distance there is between the data points and the line, the closer the line will fit the curve of the data points of the plot). In this case, there are three ways to measure the distance between the line and data points of the plot: measuring the distance from the line to a particular data point along the x-axis, measuring the distance from the line to a particular data point along the y-axis, or measuring the distance perpendicular to the line itself from the line to a particular data point.

    - How we choose to measure the distance between a line and the data points in a plot depends on the variables being plotted. Often (in real-world data), a set of independent variables (values that are known and can be accurately measured or calculated) are mapped to a set of dependent variables (values that are known but cannot be accurately measured or calculated). The point of this mapping to find a line that fits the plotted data best and use that line as an estimate of the actual values of the dependent variables (e.g. if we are trying to find out the trend of some data containing values that may not be completely accurate, we may plot that data, find a line that best fits the trend of that data, and use that line as our estimate of the actual trend of the data).

    - If then, for instance, the variables we are trying to predict (the dependent variables) are graphed along the y-axis, then we will choose to measure the distance between a line and the data points by the y-axis.

- One of the most common formulas (called *objective functions*) used to calculate the distance between the data points in a plot and a curve is known as the **method of the least squares**. In this method, the value of a point in the curve is subtracted from the value of a corresponding data point, resulting in a difference known as the **residual** (e.g.

the value of some data point is the *minuend*, the value of the corresponding point of where the line "says" the data point should be is the *subtrahend*, and the residual is the *difference*). This is done for each data point and corresponding point in the curve, and the results are added to each other and the result is squared (squaring the sum of the residuals makes the result positive).

- The different types of curves examined to see which best fits a plot of data are often represented by **polynomials** (e.g. the polynomial `y = ax + b` is the equation for a linear line, the polynomial $y = ax^2 + bx + x$ is the equation for a quadratic curve, etc.). To find where (or at what angle, etc.) a particular curve fits a data plot best, one must determine which coefficients of the polynomial of the curve will produce a curve with the closest fit to the data plots (according to the method of the least squares). There are many algorithms that can be used to do this, one being `pylab`'s `polyfit()` function.

When determining how well a particular curve fits a data plot, one can either measure how well that curve fits the data plot relative to another curve (e.g. by using the method of the least squares to figure out how far from each data point each corresponding point in the curve is, dividing the sum of those distances by the number of data points to obtain the average error of the points of the curve, and comparing the averages of the error of those two curves to each other) or how well that curve fits the data plot in a more absolute sense.

- The latter method of measuring how well a curve fits a data plot in an absolute sense can be determined by value known as the **coefficient of determination** (or, R^2, for short). The coefficient of determination can be calculated by dividing the sum of each of the values of the data points (minus the values of the corresponding points on a curve) squared by the sum of each of the values of the data points (minus the mean value of the data points) squared and subtracting the result from one (in the fraction, the numerator can be thought of as the amount of error in each of the estimates from the curve and the denominator can be thought of as the variance in each of the values from the mean).

    - Because coefficient of determination compares the errors in the estimation provided by the curve to the variability of the data points in the plot, it is intended to capture the amount of variability captured in the data points that is accounted for by the model of the curve. In a *linear regression* (a linear curve), an R^2 of 1 means the curve accounts for all of the variability in the data plot, an R^2 of 0 means there is no relationship between the curve and the data plot, and an R^2 of 0.5 means the curve accounts for only half of the variability in the data plot.

One of the most important parts to understanding data gathered from the world around us is knowing how the data we gathered came about (i.e. where it came from, what affected it, how did we record it, etc.). This will helps us make logical conclusions about what our data should

look like as a whole and help us understand why our data and certain characteristics of our data are. In other words, "To model data effectively, it is important to understand the underlying model that describes the data."

When fitting curves to plots of data, we often want to ensure that the curve generated not only fits this particular plot of data but other plots of the same sort of data so that we can make predictions about those plots of data as well. One method of doing this is by using **cross-validation**, or, testing how well a curve generated by one set of data fits an actual curve represented by another set of the same sort of data (and vice versa). This can be done by generating a curve to fit one set of data, and then measuring how much error is produced by fitting the same curve to a second set of data (as to be expected, there will usually be a little more error in fitting the curve to the second set of data is opposed to the first), then fitting a curve generated to fit the second set of data to the first set of data.

- Generally, when generating a curve to fit a plot of data, generating a curve that *exactly* fits the plot of data may not always be the best thing to do since the same curve may not (and oftentimes, will not) exactly fit another plot of the same sort of data (e.g. the curve may **overfit** the plot of data currently being analyzed while missing the underlying curve pattern present in this plot of data as well as others like it). On the other hand, however, it is possible to generate a curve that is too simple and does not do a good job in representing the actual underlying curve in a set of data and others like it (an example of the curve **underfitting** the plot of data). *Finding a balance between the two is important to generate a good curve to fit that will not only fit one plot of data, but others like it as well.* (Typically, curves generated by polynomials of higher order tend to "fit" the actual curve generated by a plot of data points better than those of lower order but do not quite "fit" curves generated by other plots of the same sort of data points very well because they "fit" the curve based on the plot of the data points they were trained upon so closely.)

- There are several methods of deciding what order of a polynomial generates the best-fitting curve to a set of data and others like it, such as the **one-one-out cross-validation** (a method where all of the examples except for one example in a data set are curve-fitted and the results analyzed to estimate the best-fitting curve for the entire set of data), **k-fold cross-validation** (a method all of the examples of a data set are partitioned into subsections and all the subsection except for one subsection are curve-fitted and the results analyzed to estimate the best-fitting curve for the entire set of data), or the **repeated-random-sampling validation** (a method where a random section of all the examples in a data set are curve-fitted and the results analyzed to estimate the best-fitting curve for the entire set of data).

…

The `polyfit()` function from the module `pylab` accepts three arguments: a list of x-values, a list (of equal length) of y-values, and an integer representing the degree of the polynomial to consider (a polynomial of degree 1 produces a linear line, one of degree 2 produces a quadratic curve, etc.). `polyfit()` returns a tuple of the coefficients for the specified polynomial that best fits the data points plotted by the x and y-values given by the first two arguments:

```
pylab.polyfit(<x-values>, <y-values>, <order_of_polynomial>)
```

The `polyval()` function from the module `pylab` accepts two arguments: a tuple of the coefficients of a polynomial (such as is returned by the `polyfit()` function), and a list of values. The `polyval()` function returns an array of corresponding (y-)values from a polynomial with the coefficients provided by the first argument for the list of values it is given in the second argument (`polyval()` determines the order of the polynomial by the number of coefficients it is given):

```
pylab.polyval((<tuple_of_coefficients>), <values>)
```

```
pylab.polyval((3, 4), [1, 2, 3, 4, 5])
```

```
# the above call of polyval() returns [7, 10, 13, 16, 19] as the
# corresponding values for the values [1,2, 3, 4, 5] in the
polynomial # y = 3x + 4
```

UNIT 5

Machine learning has been said to be the "*field of study that gives computers the ability to learn without explicitly being programmed*." In other words, while traditional computing may be described as giving a computer a program and some data and getting some sort of output from the computer, machine learning can be described as giving a computer some data and the output associated with that data and getting a program from the computer. New data can then be given as input to the program, and the computer will output what it "thinks" should be associated with data.

All machine learning methods require some sort of representation of the examples being analyzed (these representations are usually vectors with one or more attributes known as **features**), some sort of system of measurement to measure how related or unrelated features are to each other, an objective function (accompanied by a set of constraints) that can be used to actually measure the degree of similarity between features, an optimization method to "learn" the model being built by the features and their similarities, and some sort of method to evaluate the models that the optimization method finds.

Broadly speaking, machine learning can be categorized by two models. In **supervised machine learning**, computers are given a set of values and features, and must determine which values should be matched to which features based on information on how values and features should be paired (this is very similar to a subcategory of supervised machine learning known as the *classification model* (the other popular subcategory of supervised machine learning is the *regression model*)). In **unsupervised machine learning**, computers aren't given any values for classifying or grouping features and must instead sort features based on values or similarities of those features it finds on its own (this is very similar to a subcategory of unsupervised machine learning known as *clustering*, which defines a metric that captures how related or unrelated features are to each other, and then groups features based on that metric).

- Helping computers determine how to sort features based on their degree of similarity to other features requires careful balance (since, usually, we don't want computers to categorize things too broadly but we also don't want them to categorize things too exactly, either). The **signal-to-noise ratio** is the ratio of the amount of useful information that can be used to make a decision about sorting or pairing a feature to a value to the amount of useless information that cannot be used to make a good decision about sorting or pairing a feature to a value. One of the goals of machine learning is to make this ratio as high as possible to make the best possible decision about sorting features.

- One widely-used method of deciding how to sort and group features (in *supervised machine learning*) is known as **k-nearest neighbors**. The k-nearest neighbors method

works for features in a distance matrix (imagine the features plotted as points on a two-dimensional plane) by choosing a feature with an unknown label and evaluating the labels of the *k* number (typically, an odd number) of closest features to that feature. The feature with the unknown label is then classified with the features from the *k* number of closest features with the most number of the same labels. In this way, features are labelled based on the labels of the features surrounding them.

    - The k-nearest neighbors method is a relatively simple method of classifying features that requires little theory to understand and doesn't require additional training for the computer (since the computer is only finding the nearest neighbors of a particular feature). The downsides to this method are that predictions can take a long time to be made (it is often memory-intensive) and it does not reveal very much about the process that generated the data it is examining (i.e. why the features were placed where they were).

    - When comparing features of individual vectors, the scaling of the values of the features of the vectors has a huge impact on how those vectors may be sorted and grouped based on their features (e.g. vectors with features that aren't properly scaled may have features that have a disproportionate amount of influence over how those vectors are classified compared to other features of the same vector). Two methods of scaling the features of vectors more properly are known as Z-scaling (where all of the values of the features of a vector are scaled so that their mean is 0 and they have a standard deviation of 1) and interpolation (where the smallest value of a vector's features is mapped to 0 and the largest value of a vector's features is mapped to 1 so that all of the values of the vector's features are linearly scaled).

- One widely-used method of deciding how to sort and group vectors (in *unsupervised machine learning*) is known as **k-means clustering**. The k-means clustering method works for features in a distance matrix (imagine the features plotted as points on a two-dimensional plane) and works by choosing a *k* number of random features in the matrix and uses those selected features as initial centroids (geographical centers of a cluster). k-means clustering then groups the rest of the features in the matrix with the closest centroid to form a *k* number of clusters (because of the *k* number of initial centroids). The values of the features of each of those clusters are then averaged and the centroid of each cluster moved to the mean value of the average of the features of that cluster. The algorithm then runs again and groups all of the features of the matrix with the nearest centroid (because the way features are grouped may change, since the value of centroids was changed) and the clusters are again averaged and the centroids moved to the mean of the average of the clusters. This process is repeated until no features move after the centroids are moved, meaning all of the features are as close as possible to each individual cluster.

- One downside to k-means clustering is that when the features whose values will be the values for the initial centroids are randomly selected, a “bad” set of features that does not accurately represent the actual clusters in the matrix may be selected. One solution to this problem is to run a k-means clustering algorithm many times (perhaps with varying values of *k*) and select from the results the best representation of the clusters in the matrix. Unlike supervised machine learning, there are no definite “correct” ways of sorting features in unsupervised machine learning, since the features used to sort and group features is determined by the computer itself.

…

Unfortunately, while statistics can be for many good purposes, they can (and are) sometimes used to misrepresent the truth (a.k.a lying). One way to avoid making inaccurate conclusions about data is visually represent them (such as by using graphs) to get a better picture of the data being analyzed. However, graphs can be manipulated to make correct data appear disproportional compared to the way it originally was. To ensure that data is being interpreted correctly, one must be careful to note the labels and x and y axes of a graph so that they know how to view the data being represented. Other ways that statistics can be misused is by believing in statistical fallacies or by simply doing a poor job in analyzing the data in the first place (as the saying goes, *garbage in, garbage out*—making mistakes or encountering errors that aren’t accounted for by statistical laws or principles won’t magically produce correct results).

The way populations are sampled has a huge impact on whether the results of analyzing a sample taken from that population are credible. To ensure that we can trust the data we collect and make accurate inferences from that data, we must understand how that data was collected and whether or not we can make assumptions about that data that so many principles in statistics rely on (such as the empirical rule, the central limit theorem, and more). There are many ways that our collection of data may indeed be biased with/without our knowledge (e.g. such as in “convenience” sampling, where examples are sampled based on how easy they are to collect, an example of which is “survivor” bias, where data is only sampled from a population that *survived* a process (not the entire population that *began* the process, which may be more).

Another thing to be wary about when interpreting data gathered from statistics is that context matters. Numbers gathered from the analysis of data mean little without a context in which to interpret them. For example, percentages without a baseline (some sort of figure the population percentages apply to) mean nearly nothing. Having an understanding of the context and figures surrounding data is key to interpreting it correctly.

Enjoy 6.00.1x!

🍏🍕🦒

ApplePieGiraffe

🍏🍰🦒