Data Structures and Algorithms with Python

Workshop 5

This is the final workshop of the module — it is intended both for recap and for exploring new things using Python. You can for example:

1. Study the knapsack problem and Fantasy Football using greedy algorithms (sections 1-2 below)
2. Explore open data and food hygiene in South Kensington restaurants (section 3)
3. Explore the Python library `networkx` for solving BFS and many other graph problems (section 4)
4. Discuss the group project with your group members
5. Ask any questions about anything we've covered

The undercurrent on the first couple of ideas is that Python makes it really easy to access the web and scrape interesting data for a variety of purposes. Fantasy Football, Food Hygiene, and stock prices are just a couple of examples...

1 The knapsack problem

During the lecture, we started looking at the knapsack problem. The problem is simple: you want to pick the most valuable items to fill out a 'knapsack' with limited capacity. But the applications are plentiful: many problems with budget constraints in finance and operations management are versions of the knapsack problem or closely related to it.

More precisely, we start with a set of n items that each have values v_i and weights w_i and are trying to maximize the total value of items to pick, subject to a capacity constraint W:

$$\max_{x_i \in \{0,1\}} \sum_i x_i v_i \tag{1}$$

$$\text{s.t. } \sum_i x_i w_i \leq W \tag{2}$$

This is the binary (0/1) version of the knapsack problem, where we either pick each item or do not.

So our goal is to fill a knapsack (a budget) with generic items (eg players in fantasy football or projects for a firm). To implement this in Python, our object-oriented instincts suggest creating a `class Item`.

```
class Item():
    def __init__(self, n, v, w):
        self.name = n
        self.value = float(v)
        self.weight = float(w)
    # code...
```

The constructor function takes as input the name, value, and weight of an item. The complete implementation of the class is given in the Python file.

1.1 Filling the knapsack

Before going into the binary knapsack problem, let's first look at the *fractional* version of it. In the fractional version, we assume that items are divisible (say, you're travelling to Asia and want to bring back different kinds of tea leaves). We developed *greedy approaches* for this choice in the lecture. Roughly speaking, a greedy choice is myopic: it seeks to maximize the current value added, without considering the future consequences of the choice. We looked at several greedy choices: the maximum-value item, the minimum-weight item, and the best item in terms of bang-for-buck. The function `greedy` contains a skeleton code for an implementation of this choice.

```
def greedy(items, maxWeight, keyFunction):
    """Assumes Items a list, maxWeight >= 0,
    keyFunction maps elements of Items to floats, see eg density above"""
    # Sort items
    sortedItems = sortByFunction(items,keyFunction)
    result = []
    totalValue = 0.0 # knapsack value
```

```
    totalWeight = 0.0 # knapsack weight <= maxWeight
    # more code...
```

The function takes three inputs: a list of items, a capacity constraint, and `keyFunction`, a function to that defines your greedy choice. Notice that Python allows us to pass a function as an argument to another function in this convenient way.

The function `greedy` then proceeds to pass the `keyFunction` on to Python's `sorted`-function This function conveniently sorts the list of items based on the function you've defined.

```
    return sorted(items, key=keyFunction, reverse = True)
```

The Python file also includes three functions you can use to sort the items: `value`, `weightInverse` and `density`. Try out sorting your list of items by the different functions.

Now you're ready to complete the function `greedy`. The idea is simple: after sorting the list of items based on the desired function, the function should pick each item in turn, then finally returning:

- `result`, a list containing the items in the knapsack.
- `totalValue`, the sum of values of items in your knapsack

Solve the knapsack problem with the different greedy choices. We saw in the lecture that the bang-for-buck greedy algorithm is optimal here. How different are your results with the different greedy approaches?

Try your greedy algorithm out on different sets of items. Note that the greedy algorithm is not actually optimal in the binary case. But what is the optimal solution?

For small problems at least, we can implement a 'brute force' solution of the knapsack problem, which loops through all possible subsets that you can create from the set of items you have and finds the best feasible one. The problem is that there are exponentially many possible subsets. For this purpose, the code contains a function `powerSet` that creates all possible subsets of a list of items L. It uses a clever Python concept called a generator: it does not store all of the subsets in memory, but creates a `generator` object that can be used to iterate through this set one by one. To do this, it uses the keyword `yield`. Read more about generators in chapter 8.3.1 of the textbook.

The result is that the output from the function `powerSet` can simply be iterated through. This is done in the `bruteForce` function that follows. Make sure you understand how the brute force solution goes through all subsets of the problem.

Try solving the optimal solution of the brute force algorithm for small samples. How fast is the greedy solution? How large problems can you solve via brute force? What is the complexity of the brute-force solution?

2 Fantasy football

You'll now get to apply your algorithm to a potentially useful real-world application: fantasy football (FF). In fantasy football, you act as a manager of a football team in eg the Premier League and try to build the best team of players possible using a limited budget. FF leagues are popular among football fans: for example, the Premier League organizes an official league on its website, with significant prizes for best 'managers'.

How is the the success of your FF team determined? The Premier League has a set of rules for this: for example, your players will get a certain number of points for scoring/assisting goals, keeping an empty sheet, being named man of the match, and so on. The league also determines the costs of acquiring different players into your team based on their popularity and performance.

You're probably starting to see this as a knapsack problem: we have a limited budget, as well as costs for each player. However, we're missing the 'values' of different players. Determining a player's expected future value for your team (how many goals will they score in following games) may be more art than science, but one simple way to try to value them is to use their past performance (in terms of points already accumulated). The Premier League helpfully publishes the current scores of each player on their website. If we download this data, we have everything we need to solve a knapsack problem.

This data (current as of 28/9) is available on the Hub in a html file downloaded from the website. The Python file further includes helper functions for you to read this data into a list of players:

- The `readPlayersHTML` takes a html page downloaded from the Premier League website containing the players values, and parses it to create lists of player attributes.
- The `class Player` is very similar to the `Item` class, but adapted for player attributes
- The `buildPlayers` function takes the lists of player attributes and creates a list of players

In order to allow you to start building your team, we won't go into much detail here — you can just run the functions. But going through these functions later is a useful exercise to develop your skills in using code written by others.

Go ahead and solve the problem using the greedy algorithm on the list of players and a budget constraint. Try out different budget constraints. What is the result? Is this what a football team looks like? What went wrong?

You'll probably want to have exactly 11 players in your team, and at specific places too (at most a single goalkeeper too and so on). In the language of optimisation, this is a *cardinality constraint* on the knapsack problem. That is, the optimisation problem would have an additional constraint

$$\sum_i x_i \leq 11. \tag{3}$$

This is a somewhat more challenging problem. The greedy heuristic provided presents a naive way of dealing with this: simply picking in the greedy order and then making sure the solution does not break any of the constraints.

This is probably not the best solution you could find. What could you do to improve your solution? A brute-force algorithm would find the correct solution, but there are many players so it will be quite slow. There are several optimisation methods that could help you, but these are beyond the scope of this course so they are left for you to explore. For example, you could look into dynamic programming and genetic algorithms. Guttag's book shows you a dynamic programming algorithm for the knapsack problem (without the extra constraints on the number of players etc); for genetic algorithms, you could start by searching online for "genetic algorithms fantasy football python"...

3 Python for your safety

Let's turn from algorithms to something completely different.

One of the main advantages of Python is that once you've got a hang of the syntax, it's easy to get useful work done quickly. It is therefore the go-to language for practical problem-solving in many different application areas in business, science, all kinds of analytics, journalism, etc.

A common thing that Python makes really easy is accessing data online. This is also becoming increasingly useful as both governments and companies are opening their data for anyone to access. The London Datastore, for instance, provides access to hundreds of datasets from the Greater

London Authority.[1] With more and more fascinating data sets available online, this is a womderful time to be learning Python for analytics.

Here we'll look at a problem of some practical significance: finding out food hygiene in restaurants close to us. The UK Food Standards Agency provides data on restaurant hygiene inspections in its webpage (http://ratings.food.gov.uk/open-data/en-GB) free for anyone to access.

The data is provided in a format called XML (short for extensible markup language). XML is used for many purposes — here it is convenient for representing data that is not completely "structured", that is, not all possible fields of information are available for all data points. To analyse XML data, we'll first need to convert it to a more practical form, here a csv file.

There are many Python libraries that provide us with easy access to web data. The Python file on the Hub provides one (rather crude) way to read the relevant data online and then read XML into a csv file. This code is provided for your convenience; you can explore it at your leisure. For now, you can move straight to analysing the data.

The csv file contains the following fields:

- BusinessName, FHRSID, BusinessType, RatingDate, AddressLine — these are self-explanatory
- RatingValue — the hygiene rating for the establishent (5 is the highest ie best rating)
- Hygiene, Structural, ConfiDenceInManagement — more specific scores where the lower the better
- Longitude, Latitude — geolocation

Read the csv file using the provided commands and write Python functions to answer the following questions:

1. What are the best and worst restaurants in terms of their hygiene? What fraction of the restaurants would you deem are doing well in terms of their hygiene? Do you think there's a connection to what their business type is?

2. Imperial College's position is (-0.1749,51.4988). What are the closest restaurants? Write a function that gives you the distances from Imperial to all the restaurants. Assume that the world is flat.

[1] See http://data.london.gov.uk/. A large list of other open data sets can be found here: https://github.com/caesar0301/awesome-public-datasets. Sometimes you have to register as a developer to get convenient access to data: for example Transport for London and Twitter provide fairly comprehensive access to their data through their APIs.

3. What are the closest restaurants that you would advice your classmates to avoid?

Note that many of the businesses are missing some of the data. You may want to ignore these in your analysis – one way to do this is using the `dropna()` function in pandas.

If you wanted to turn this into a project, you might next try to figure out how to automatically download all data from the website, then visualize it, and perhaps also layer it over a map...

4 Graphs using Python libraries

There is a useful Python library called `networkx` which you can use to solve not only the above shortest-path problems but many other relevant graph problems. You can use `import networkx` in Python to gain access to it. To get started, find the official documentation at http://networkx.readthedocs.io/en/latest/.

Work through the tutorial on the website and recreate the weighted graph from the lecture. Plot the graph. Use the shortest-path algorithm from the library to calculate shortest paths (see Reference -> Algorithms -> Shortest paths in the documentation).

Here's a very basic sample use of the module:

```
import networkx as nx
G=nx.Graph()
G.add_nodes_from([1,2,3,4])
G.add_edge(1,2)
G.add_edge(2,3)
G.add_edge(3,4)
n=nx.shortest_path_length(G,1,4)
```