Sorting Challenge Project

COT 4400, Summer 2017

June 12, 2017

1 Overview

The *Dunning-Kruger effect* is a psychological bias that causes people evaluate their own competence incorrectly. In particular, those who are incompetent at a task generally lack the understanding to recognize how incompetent they are, while those who are very competent may assume that tasks they find easy are easy for everyone else, as well. In this project, you will develop and implement an algorithm to sort a collection of objects whose comparator functions exhibit behavior consistent with the Dunning-Kruger effect.

2 Sorting the objects

Each object in the array has a competence rating in the range of 0 to 100, and your goal is to sort these objects in order of increasing competence. However, your challenge is that the comparator function for these objects is unreliable in a way consistent with the Dunning-Kruger effect—each object holds a *belief* about its own competence that it compares against another's competence to determine whether it is more or less competent than the other. When making this comparison, each object compares its *belief* value with the other object's true *competence*. Objects are only unreliable in evaluating themselves.

As an example, suppose that object a has competence 20 and believed competence of 40 and object b has competence 25 and belief 35. Object a would report that it is more competent than b, since its perceived competence of 40 is more than 25, while object b would report that it is more competent than a since $35 > 20$. Since these two disagree on who is more competent, you would need to use information from other objects' comparisons in order to determine the correct order between a and b.

You may assume that all of the following are true about the beliefs of these objects:

1. Objects with competence of at least 25 and less than 90 will have belief equal to their competence.
2. Objects with competence less than 25 will overestimate their competence.
3. Such objects will overestimate their competence by at least 1 but will not believe their competence to be 90 or more.
4. Below 25, objects with *lower* competence will have a *higher* belief in their competence. (In other words, competence and belief are negatively correlated in low performers.)
5. Objects with competence of 90 or more will underestimate their competence.
6. Such objects will underestimate their competence by at least 1, though they will never believe their competence to be less than 25.
7. At or above 90, objects with *higher* competence will have *higher* belief in their competence. (Competence and belief are positively correlated in high performers.)
8. If any two objects agree that one is more competent than the other, they are both correct.

For the sake of simplicity, you may also assume that no two elements have exactly the same competence values.

3 Provided code

You have been provided with C++ code containing a main function, a scoring function, and two implementations of the Subject class to test your proposed sorting algorithms. Outside of changing a few constants, you should not need to modify any of these files, though their functions and purpose are described here, so that you understand the environment in which your algorithm is being executed.

In the interests of time, this project has only been adapted to C++, and only C++ solutions will be considered.

3.1 Subject and derived subclasses (`dunningkruger.h` and `dunningkruger.cpp`)

Subjects are the objects you are attempting to sort. They have several methods, but for your purposes, the most important one is:

```
bool considersThemselfBetterThan(const Subject& oth) const
```

This function returns whether the invoking Subject believes their competence to be higher than the argument's competence. More details about Subjects' beliefs are provided in Section 2. You may also use the operators $>$ and $<$ to compare Subjects; these are identical to invoking this function on the left Subject, with the right Subject as an argument (and negating for $<$).

You have also been provided with a `swap` function that swaps two given Subjects in an array. For example, `swap(arr[0], arr[1])` will swap the first two elements in the array `arr`.

For debugging purposes, Subject also features a `getCompetence` method, as it is not feasible to debug your function without having some way of "peeking" at the true competence value. The `getCompetence` function **will not be present** when your submission is graded, however, so if your sorting function invokes `getCompetence`, your code will not compile, and you will receive no credit.

In addition to the base Subject class, there are two different derived classes, SlightlyOff and Delusional, which provided two different implementations of how these Subjects might assess their own competence. These are provided as a way to make sure that your algorithm is general to the problem and does not rely on a particular implementation of how these beliefs are derived. As might be inferred from their names, SlightlyOff objects are much more accurate in estimating their competence than Delusional objects, which represent a more extreme case. As a result, Delusional objects may be more difficult to sort accurately. (The test implementation may be more or less extreme than these examples.)

3.2 Main and utility functions (`driver.cpp` and `dunningkruger.cpp`)

The driver constructs a collection of Subjects with competence values evenly distributed between 0 and 100. It shuffles the array and executes the `dksort` function 5 times, computing the average timing and accuracy rating for `dksort`. In order to ensure that every run of the program has the same outcome, it uses the same set of seeds for the random number generator every run. *Important*: if you decide to create a randomized algorithm, please generate random numbers using the `rand()` function from the `random` library so that this consistency is preserved.

The implementation provided constructs an array of 100 SlightlyOff objects; however, this behavior can be modified by changing the `POP_SIZE` and `CLASS_TO_TEST` constants at the top of `driver.cpp`.

The scoring function compares all pairs of elements in the array to find which ones are out of order and returns a score between 0 and 1 to represent how sorted the array is. The full details of this function are not important, but perfectly sorted arrays get a score of 1, random arrays generally have a score between 0.1 and 0.2, and your goal should be a score of 0.9 or greater. This function is a better indicator of performance for large n. An array with 10000 or 100000 elements can give a good measure of your algorithm's performance, though it's better to start with $n = 100$ for testing/debugging purposes.

3.3 Compiling the provided code

In order to compile the code, you will need to implement the following function in `sort.cpp`:

```
void dksort(Subject* arr, int n)
```

This function does not have to do anything, but the code will not compile without it.

For Windows and Mac users, you should be able to import all of the provided cpp and h files into a project in your favorite IDE, add `sort.cpp`, and compile from there. Linux users, you have been provided with a Makefile for the project. By typing `make` in the project directory, the code will be compiled automatically, assuming you have `make` and `g++` installed.

4 Submission

For this project, you should submit a file named `sort.cpp` that implements the function

```
    void dksort(Subject* arr, int n)
```

which sorts an array of n objects based on their comparator functions. You can read more about the implementation of this class in section 3.1. Your submission may also include any helper functions required to implement `dksort`; however, only `dksort` will be evaluated.

5 Grading

You may receive up to 7 bonus points based on the accuracy of your `dksort` function, as reflected by the `score` function. For full credit, your function should consistently achieve a score above 0.9 when sorting large inputs. Reasonably accurate submissions may receive up to 3 additional bonus points based on speed. These points will be awarded relative to the speed of "standard" solutions implemented by the instructor.

These bonus points will be added to any bonus points you have accrued in class.