ORIE 5270: Big Data Technologies
Homework 0

Deadline: Friday, Feb. 7th at 11:59pm US EST.

This assignment is **optional**. It is meant to get you acquainted with the Gradescope submission / autograding system and give you the chance to write some `Python`.

How to submit: Navigate to the `Files` section of Canvas and download the file called `hw0.zip`; extract its contents to your workspace. It contains one `.py` file for each problem below. After you make all the appropriate edits, create a single .zip file containing these files (and these files only), and submit it to the assignment page on Gradescope. The autograder interface will let you know if your submission was successful and show you some testcases, your code's output on the failed instances, as well as a partial score.

Note: *Your submission* ***must*** *follow the above procedure for the autograder to work. Submissions in other formats, e.g., Jupyter notebooks will be ignored.*

Problem 1 (Binary search)**.** One of the fundamental algorithms in computer science is that of **binary search**. In binary search, we are given a **sorted** array A as well as an element x (that may or may not be part of the array), and wish to find the index of the **largest element** $y \in A$ **such that** $y \leq x$.

Complete the code given in the file `binary_search.py` to implement the above algorithm. If all elements of the array A are larger than the query x, the function should return -1.

Problem 2 (Gradient descent)**.** Gradient descent is one of the oldest algorithms in optimization for solving

$$\min_{x \in \mathbb{R}^n} f(x), \quad \text{where } f \text{ is differentiable.}$$

Starting from some initial point $x_0 \in \mathbb{R}^n$, gradient descent iterates:

$$x_{k+1} := x_k - \eta_k \nabla f(x_k), \quad k \geq 0, \tag{1}$$

where η_k is the so-called *step size* and $\nabla f(x_k)$ is the gradient of f evaluated at x_k.

In this problem, you will implement gradient descent to minimize the least-squares loss function:

$$f(x) = \frac{1}{2} \|Ax - b\|^2, \tag{2}$$

where $\|x\|$ is the standard Euclidean norm. You are encouraged to use `NumPy` for the implementation.

(i) In the file `gradient_descent.py`, fill in the code for the functions `loss(A, b, x)` and `gradient(A, b, x)`. The first function evaluates the loss $f(x)$ from (2) and the second function evaluates its gradient $\nabla f(x)$.

(ii) In the same file, fill in the missing code for `gradient_descent(A, b, x_0, T)`, which implements the iteration from (1) for a fixed number of steps T using the so-called *Polyak step size*:

$$\eta_k := \frac{f(x_k) - \min f}{\|\nabla f(x_k)\|^2}.$$

You can assume there is an exact solution to the least squares problem, so that $\min f = 0$.

This function should return the final iterate x_T as well as the norm of the gradient evaluated at x_T, $\|\nabla f(x_T)\|$.

Note: When minimizing smooth functions, the norm of the gradient is commonly used to decide termination of the algorithm.

Problem 3 (Classes)**.** Implement a class called `VectorND` that represents a real-valued vector of the form $[x_1, \ldots, x_N]$. In particular, the dimension N should not be fixed, but your class should allow it to be determined when constructing an instance. For example, both of the following should construct `VectorND` instances:

```
>>> vec1 = VectorND(1, 2, 3, 4)
>>> vec2 = VectorND(1, 2, 3)
```

Your class should support the following operations (by implementing the appropriate class methods):

- **length**: if `vec` is a `VectorND` instance, writing `len(vec)` should return its length. For example:

```
>>> vec = VectorND(0, 1, 2)
>>> len(vec)
3
```

- **equality**: two `VectorND` instances are equal if they represent the same vector:

```
>>> vec_1 = VectorND(0, -1, 0)
>>> vec_2 = VectorND(0, -1, 0)
>>> vec_1 == vec_2
True
```

```
>>> vec_3 = VectorND(0, 1, 0)
>>> vec_1 == vec_3
False
```

- **addition**: given two `VectorND` instances `x` and `y`, writing `x + y` should return a `VectorND` object with elements $[x_1 + y_1, \ldots, x_N + y_N]$. If the vector lengths are different, it should raise an appropriate exception. For example:

```
>>> vec1 = VectorND(1, -1, 2, 0)
>>> vec2 = VectorND(0, 1, -2, 1)
>>> vec1 + vec2   # should be equivalent to VectorND(1, 0, 0, 1)
```

- **substraction**: as above, but with `x - y`.

In addition, it should provide a printable representation of the object, as below:

```
>>> vec = VectorND(1, 2, 3, 4)
>>> vec
Vector: [1, 2, 3, 4]
```

All necessary changes should be made in the file `vector_nd.py`, which contains some skeleton code for your convenience.