ORIE 5270: Big Data Technologies
Homework 0

Instructor: Vasileios Charisopoulos - vc333@cornell.edu

Instructions: This assignment is *optional* and meant to familiarize you with `Python`. Nevertheless, if you submit we will substitute your lowest graded homework with this one (as long as that improves your grade).

The deadline to submit is **Friday, February 19th at 12pm (noon) US EST**. Submit your answers on **Gradescope**. You should submit a *single file per problem.*[1]

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** $y \leq x$.

Write a Python function `binary_search(A, x)` that implements this algorithm.

(*Note: Python already provies a module called* `bisect` *that implements binary search, that you would use if you were writing production code.*)

Problem 2 (Palindromes)**.** We call an integer $x \in \mathbb{Z}$ a *palindrome* if it represents the same value when read from left to right as when it is read from right to left. For example, 121 and 1221 are palindromes, while 122 is not.

(i) Write a Python function `palindrome(x)` that accepts an integer x and returns `True` if it is a palindrome and `False` if not.

(ii) If the value of the argument x passed to your function is not an integer, raise an appropriate exception (`ValueError`) instead.

(*Hint: look for the* `isinstance` *function.*)

Problem 3 (Gradient descent)**.** Gradient descent is one of the oldest algorithms in optimization for solving

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

It works as follows: starting from some initial point $x_0 \in \mathbb{R}^n$, it repeats the following

[1]See https://bit.ly/3oY8s3g for help with submitting a programming assignment.

step:

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

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

(i) Write a Python function that implements gradient descent for T steps starting from a given point x_0, given callables `f` and `gradf` (which return the value of the function and its gradient, respectively) as well as a constant step size η. Your declaration should look like below:

```
def gradient_descent(f, gradf, eta, x_0, T):
    """
    Documentation
    """
```

(ii) In the above, you (most likely) treated `eta` as a number, since the step size was assumed to be a constant. Update your code so that it can accomodate time-varying step sizes. You may assume that the step size may only depend on the iteration index – for example, these are all valid choices:

$$\eta_k \equiv \eta = 0.1, \text{ or } \eta_k = \frac{1}{\sqrt{k}}, \text{ or } \eta_k = 2^{-k}.$$

Problem 4 (Solving a linear system)**.** Linear systems are systems of equations of the form

$$Ax = b, \quad A \in \mathbb{R}^{m \times n},\ b \in \mathbb{R}^m.$$

If the number of equations is larger than the number of unknowns, there is a *unique* solution (if $m > n$, it is possible there is no solution entirely). If $m < n$ (the system is *underdetermined*) the number of possible solutions is infinite, and it is common to look for the *minimum norm* solution, given by

$$x^\sharp := A^\mathsf{T}(AA^\mathsf{T})^{-1}b.$$

(i) Write a Python function (using the `numpy` library) that, given A and b, returns the solution to the system $Ax = b$; if the system is underdetermined, your function should return the minimum norm solution. The solution should be a single NumPy `array` object, and you can assume that A and b will also be given as NumPy matrices.

(ii) If $m > n$ and the system $Ax = b$ has no solutions, raise an appropriate exception instead of returning a value.

Problem 5 (Classes)**.** Implement (in Python) 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 two `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):

- **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.
- **subtraction**: as above, but with `x - y`.
- **iteration**: the `VectorND` object should be iterable:

```
>>> vec = VectorND(1, 2, 3, 4)
>>> for i in vec: print(i)
1
2
3
4
```

  Hint: look up `yield` and the `__iter__` method.

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]
```