Homework 1: Math Foundations for ML

The is the coding potion of Homework 1. The homework is aimed at testing the ability to code up mathematical operations using Python and the `numpy` library.

For each problem, we provide hints or example test cases to check your answers (see the `assert` statements below). Your full submission will be autograded on a larger batch of randomly generated test cases.

Note on the autograding process

For this assignment, we are using `nbgrader` for autograding. We recommend that you use JupyterLab or Jupyter notebook to complete this assignment for compatibility.

The cells containing example test cases also serve as placeholders for the autograder to know where to inject additional random tests. Notice that they are always after your solution; moving/deleting them will cause the tests to fail, and we'd have to manually regrade it. They are marked with `DO NOT MOVE/DELETE` and set to read-only just in case.

The autograder tests will call the functions named `solve_system`, `split_into_train_and_test`, `closest_interval`. You may not change the function signature (function name and argument list), but otherwise feel free to add helper functions in your solution. You can also make a copy of the notebook and use that as a scratchpad.

To double check your submission format, restart your kernel (Menu bar -> Kernel -> Restart Kernel); execute all cells from top to bottom, and see if you can pass the example test cases.

```
import numpy as np
```

Part 1: Systems of linear equations

Given n equations with n unknown variables ($n \leq 4$), write a function `solve_system` that can solve this system of equations and produce an output of value for each variable such that the system of equations is satisfied.

The system of equations will be provided as a list of strings as seen in `test_eq`.

You may assume that the variables are always in $\{a, b, c, d\}$, the system has a unique solution, and all coefficients are integers.

```
def solve_system(equations):
    """
    Takes in a list of strings for each equation.
    Returns a numpy array with a row for each equation value
    """
    def strnum(s): # string to numeric coefficient
```

```
        cur_num, is_pos, eq_seen = 0, 1, 0
        n = len(s)
        res = []
        i = 0
        while(i < n):
            while(i < n and '0' <= s[i] and s[i] <= '9'):
                cur_num = cur_num * 10 + ord(s[i]) - ord('0')
                i += 1
            if(eq_seen and cur_num):
                res.append(cur_num if is_pos else -cur_num)
            if(i == n):
                return res
            c = s[i]
            if(c == '+'):
                is_pos = 1
            elif(c == '-'):
                is_pos = 0
            elif('a' <= c <= 'd'):
                res.append(cur_num if is_pos else -cur_num)
                if(not res[-1]):
                    res[-1] = 1
                cur_num = 0
            elif(c == '='):
                eq_seen = 1
            i += 1
        return res
    a, b = [], []
    eq_n = len(equations)
    for i in range(eq_n):
        _a = strnum(equations[i])
        len_a = len(_a)
        a.append(_a[:len_a - 1])
        for j in range(len_a - 1, eq_n):
            a[-1].append(0)
        b.append(_a[len(_a) - 1])
    sol = np.linalg.solve(a, b)
    sol_n = len(sol)
    res = []
    for i in range(sol_n):
        res.append([sol[i]])
    return res

# === DO NOT MOVE/DELETE ===
# This cell is used as a placeholder for autograder script injection.

def test_eq(sys_eq):
    results = solve_system(sys_eq)
    expected = np.array([[3],[5],[2],[4]])
    assert np.allclose(expected, results)
```

```
test_eq([
    '2 a + b - 3 c + d = 9',
    '-5 a + b - 4 c + d = -14',
    'a + 2 b - 10 c = -7',
    'a + 2 b = 13',
])
```

Part 2: Split a dataset into test and train

(For this question, using an existing implementation (e.g. `sklearn.model_selection.train_test_split`) will give 0 points.)

In supervised learning, the dataset is usually split into a train set (on which the model is trained) and a test set (to evaluate the trained model). This part of the homework requires writing a function `split_into_train_and_test` that takes a dataset and the train-test split ratio as input and provides the data split as an output. The function takes a `random_state` variable as input which when kept the same outputs the same split for multiple runs of the function.

Note: if `frac_test` does not result in an integer test set size, round down to the nearest integer.

Hints:

- The input array `x_all_LF` should not be altered after the function call.
- Running the function with the same seed multiple times should yield the same results.
- Every element in the input array should appear either in the train or test set, but not in both.

```
def split_into_train_and_test(x_all_LF, frac_test=0.5, seed=None):
    ''' Divide provided array into train and test sets along first
dimension
    User can provide random number generator object to ensure
reproducibility.
    Args
    ----
    x_all_LF : 2D np.array, shape = (n_total_examples, n_features) (L,
F)
        Each row is a feature vector
    frac_test : float, fraction between 0 and 1
        Indicates fraction of all L examples to allocate to the "test"
set
        Returned test set will round UP if frac_test * L is not an
integer.
        e.g. if L = 10 and frac_test = 0.31, then test set has N=4
examples
    seed : integer or None
        If int, will create RandomState instance with provided value
as seed
        If None, defaults to current numpy random number generator
```

```
np.random.
    Returns
    -------
    x_train_MF : 2D np.array, shape = (n_train_examples, n_features)
(M, F)
        Each row is a feature vector
        Should be a separately allocated array, NOT a view of any
input array
    x_test_NF : 2D np.array, shape = (n_test_examples, n_features) (N,
F)
        Each row is a feature vector
        Should be a separately allocated array, NOT a view of any
input array
    Post Condition
    --------------
    This function should be side-effect free. Provided input array
x_all_LF
    should not change at all (not be shuffled, etc.)
    Examples
    --------
    >>> x_LF = np.eye(10)
    >>> xcopy_LF = x_LF.copy() # preserve what input was before the
call
    >>> train_MF, test_NF = split_into_train_and_test(
    ...     x_LF, frac_test=0.201,
random_state=np.random.RandomState(0))
    >>> train_MF.shape
    (7, 10)
    >>> test_NF.shape
    (3, 10)
    >>> print(train_MF)
    [[0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
     [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
     [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
     [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
     [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
     [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
     [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]]
    >>> print(test_NF)
    [[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
     [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
     [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]]
    ## Verify that input array did not change due to function call
    >>> np.allclose(x_LF, xcopy_LF)
    True
    References
    ----------
    For more about RandomState, see:
    https://stackoverflow.com/questions/28064634/random-state-pseudo-
random-numberin-scikit-learn
```

```
    '''
    if seed is None:
        rng = np.random.RandomState()
    rng = np.random.default_rng(seed)
    copy = x_all_LF.copy()
    rng.shuffle(copy)
    L = np.shape(x_all_LF)[0]
    import math
    N = math.floor(frac_test * L) # n_test_examples
    x_test_NF, x_train_MF = [], []
    for i in range(N):
        x_test_NF.append(copy[i].copy())
    for i in range(N, L):
        x_train_MF.append(copy[i].copy())
    return np.array(x_train_MF), np.array(x_test_NF)

# === DO NOT MOVE/DELETE ===
# This cell is used as a placeholder for autograder script injection.

N = 10
x_LF = np.eye(N)
xcopy_LF = x_LF.copy() # preserve what input was before the call
train_MF, test_NF = split_into_train_and_test(x_LF, frac_test=0.2,
seed=0)
```

Part 3: Solving a Search Problem

Given a list of N intervals, for each interval $[a, b]$, we want to find the closest non-overlapping interval $[c, d]$ greater than $[a, b]$.

An interval $[c, d]$ is greater than an non-overlapping interval $[a, b]$ if $a<b<c<d$.

The function `closest_interval` takes in the list of intervals, and returns a list of indices corresponding to the index of the closest non-overlapping interval for each interval in the list. If a particular interval does not have a closest non-overlapping interval in the given list, return `-1` corresponding to that element in the list.

```
from typing import List
from dataclasses import dataclass
from functools import cmp_to_key

@dataclass
class pair:
    a: int
    b: int

def cmp(p1: pair, p2: pair) -> int:
    return p1.a - p2.a

def lower_bound(a: List[pair], x: pair, n: int, cmp) -> int:
```

```
    low, high = 0, n
    while(low < high):
        mid = low + (high - low) // 2
        if(cmp(x, a[mid]) < 0):
            high = mid
        else:
            low = mid + 1
    if(low < n and cmp(a[low], x) < 0):
        low += 1
    return low

def closest_interval(intervals):
    n = len(intervals)
    sort_a = [pair(0, 0) for _ in range(n)]
    for i in range(n):
        sort_a[i].a = intervals[i][0]
        sort_a[i].b = i
    sort_a = sorted(sort_a, key = cmp_to_key(cmp))
    res = n * [-1]
    for i in range(n):
        j = lower_bound(sort_a, pair(intervals[i][1], 0), n, cmp)
        if(j != n):
            res[i] = sort_a[j].b
    return res

# === DO NOT MOVE/DELETE ===
# This cell is used as a placeholder for autograder script injection.

intervals = np.array([
    [1, 4],
    [2, 5],
    [8, 9],
    [6, 8],
    [9, 10],
    [3, 4],
    [7, 9],
    [5, 7],
])

expected_closest_intervals = closest_interval(intervals)

# Evaluate
results = np.array([7, 3, -1, 4, -1, 7, -1, 2])
assert np.allclose(expected_closest_intervals, results)
```