Micro Problem Set

Handed out: Monday December 10, 2018
Due: Wednesday December 12, 2018 at 4:30 PM

Important Reminders:

This micro-pset will be administered in lieu of the quiz, previously scheduled for December 12th. As such it will contribute to the final grades in the same proportion as the quiz would have. This means it will account for 40% of the final grade for 6.0002 students and 20% for 6.00 students. This micro-pset differs from previous psets in many regards, the most notable ones are listed below:

- You will not be allowed to use late days for this pset.
- You MAY NOT consult with others on this problem set. **Collaboration violations of any sort will result in a 0 on the pset.**
- There will be no office hours for the duration of this pset.
- You can post private questions on Piazza to ask for clarifications, but no other posts will be permitted
- There will be no checkoff for this pset.

Problem 1 (25%)

Fill in your answer to this question in **`q1.py`**. Write a function that meets the following specification.

```
def average_streak_count(n, streak, num_trials):
    """
    n: number of coin flips in one trial
    streak: a string of length > 0, representing the sequence of
            heads or tails in one trial. Heads represented as "H" and tails as "T"
    num_trials: number of trials in the simulation
    Runs a Monte Carlo simulation with a number of trials 'num_trials'. For each
    trial, it tracks the number of times 'streak' occurs when a fair coin is
    flipped 'n' times. After 'num_trials' number of trials, it returns a tuple of:
    (1) the average number of times the streak occurs
    (2) the width of the 95% confidence interval rounded to 3 decimal places
        (from the mean to one side, only)
    """
    # You are given this function - do not modify
    def get_mean_std(X):
        mean = sum(X)/len(X)
        tot = 0.0
        for x in X:
            tot += (x - mean)**2
        std = (tot/len(X))**0.5
        return (mean, std)

    # YOUR CODE HERE
```

For example:

`n = 6`
`streak = "HTH"`
If one trial yields: `"HTHTHH"` then the streak occurs 2 times.
If one trial yields: `"HTHHTH"` then the streak occurs 2 times.
If one trial yields: `"THTHHH"` then the streak occurs 1 times.
And `average_streak_count(6,"HTH",10000)` returns something close to `(0.5, 1.25)`

Problem 2 (25%)

Fill in your answer to this question in **q2.py**. Write a function that meets the following specification.

```
def exchange_money(exchange_rates, amount, currency_from, currency_to):
    """
    exchange_rates: list of lists for all exchange rates, where one exchange rate
            is represented as: [currency_from, currency_to, exchange_rate]
    exchange_rate: positive and non-zero float s.t.
            amount_of_currency_from*exchange_rate = amount_of_currency_to
            amount_of_currency_to*(1/exchange_rate) = amount_of_currency_from
    currency_from, currency_to: str
    amount: float representing the amount of money you wish to exchange
    currency_from: string representing the currency that 'amount' is in
    currency_to: string representing the currency you wish to change 'amount' to

    Returns a float, rounded to 2 decimal places, representing the maximum amount
    of currency_to that can be achieved by using the exchange rates in exchange_rates to
    exchange the amount of currency_from.

    If there is no path from curency_from to currency_to, returns None

    Hint: you should utilize classes from previous problem sets.
    """
    # YOUR CODE HERE
```

For example, the following should return the result `20.26`:

```
>>> exchange_rates = [["USD", "EUR", 0.88], ["MXN", "JPY", 5.57], ["JPY", "EUR",
0.0078]]
>>> print(exchange_money(exchange_rates, 1, "USD", "MXN"))
>>> 20.26
```

Problem 3 (25%)

In this question, you'll figure out how best to transport cows across space in a spaceship with limited weight capacity. Fill in your answer to this question in **q3.py**.

You will be implementing a greedy algorithm. The algorithm will always pick the lightest cow that will fit onto the spaceship first. You'll add cows until you can't add any more, then transport them. You may need to make multiple trips, with each trip picking cows according to this greedy algorithm, until you transport all the cows that can be transported. The function you write returns a list of lists, where each inner list represents a single trip and contains the names of cows taken on that trip.

Assumptions:

- All the cows are between 1 and 100 tons in weight.
- All the cows have unique names.
- If multiple cows weigh the same amount, break ties arbitrarily.
- The spaceship has a cargo weight limit (in tons), which is passed into the function as a parameter.

```
def greedy_cow_transport(cows,limit=10):
    """
    cows: a dictionary of name (string), weight (int) pairs
    limit: weight limit of the spaceship (an int)
    Uses a greedy heuristic to determine an allocation of cows that attempts to
    minimize the number of spaceship trips needed to transport all the cows.
    The greedy heuristic should follow the following method:
    1. As long as the current trip can fit another cow, add the lightest
       cow that will fit to the trip
    2. Once the trip is full, begin a new trip to transport some of the remaining cows
    Does not mutate the dictionary cows.
    Returns a list of lists, with each inner list containing the names of cows
    transported on a particular trip. The list should be in the order of the trips.
    """
    # YOUR CODE HERE
```

For example, suppose the spaceship has a weight limit of 10 tons and the set of cows to transport is `{"A": 6, "B": 3, "C": 2, "D": 5}`. The greedy algorithm will first pick `C` as the lightest cow for the first trip. There is still space for `8` tons on the first trip. Then it picks the next lightest cow, `B`, and now there are `5` tons left for the first trip. Then it picks `D`, and there are `0` tons left for the first trip. So the first trip is `[C, B, D]`. For the second trip, you take only `A`.

`If cows = {"A": 6, "B": 3, "C": 2, "D": 5}` then `print(greedy_cow_transport(cows, 10))` prints `[["C", "B", "D"], ["A"]]`

Problem 4 (25%)

Fill in your answer to this question in **`q4.py`**.

The diameter of a cluster is the distance between the two points in the cluster that are furthest apart. Change the code in the box below so that `trykmeans` returns the clustering with the smallest sum of the squares of the diameters of the clusters in the clustering.

For example:

```
players = buildPatriotsData(pats, True)
bestClustering = trykmeans(players, 4 , 40)
round(dissimilarity(bestClustering), 4)
```

Should return something close to `6.2278`

```
import numpy as np
import random

def minkowskiDist(v1, v2, p):
    #Assumes v1 and v2 are equal length arrays of numbers
    dist = 0
    for i in range(len(v1)):
        dist += abs(v1[i] - v2[i])**p
    return dist**(1/p)

class Example(object):
    def __init__(self, name, features, label = None):
        #Assumes features is an array of floats
        self.name = name
        self.features = features
        self.label = label
    def dimensionality(self):
        return len(self.features)
    def getFeatures(self):
        return self.features[:]
    def getLabel(self):
        return self.label
    def getName(self):
        return self.name
    def distance(self, other):
        return minkowskiDist(self.features, other.getFeatures(), 2)
    def __str__(self):
        return self.name +':'+ str(self.features) + ':'\
               + str(self.label)

class Cluster(object):
    def __init__(self, examples):
```

```
        """Assumes examples a non-empty list of Examples"""
        self.examples = examples
        self.centroid = self.computeCentroid()
    def update(self, examples):
        """Assume examples is a non-empty list of Examples
           Replace examples; return amount centroid has changed"""
        oldCentroid = self.centroid
        self.examples = examples
        self.centroid = self.computeCentroid()
        return oldCentroid.distance(self.centroid)
    def computeCentroid(self):
        vals = np.array([0.0]*self.examples[0].dimensionality())
        for e in self.examples: #compute mean
            vals += e.getFeatures()
        centroid = Example('centroid', vals/len(self.examples))
        return centroid
    def getCentroid(self):
        return self.centroid
    def variability(self):
        totDist = 0
        for e in self.examples:
            totDist += (e.distance(self.centroid))**2
        return totDist
    def members(self):
        for e in self.examples:
            yield e
    def __str__(self):
        names = []
        for e in self.examples:
            names.append(e.getName())
        names.sort()
        result = 'Cluster with centroid '\
               + str(self.centroid.getFeatures()) + ' contains:\n  '
        for e in names:
            result = result + e + ', '
        return result[:-2] #remove trailing comma and space

def dissimilarity(clusters):
    """Assumes clusters a list of clusters
       Returns a measure of the total dissimilarity of the
       clusters in the list"""
    totDist = 0
    for c in clusters:
        totDist += c.variability()
    return totDist

def scaleAttrs(vals):
    vals = np.array(vals)
    mean = sum(vals)/len(vals)
    sd = np.std(vals)
    vals = vals - mean
    return vals/sd
```

```
def kmeans(examples, k):
    #Get k randomly chosen initial centroids, create cluster for each
    initialCentroids = random.sample(examples, k)
    clusters = []
    for e in initialCentroids:
        clusters.append(Cluster([e]))

    #Iterate until centroids do not change
    converged = False
    numIterations = 0
    while not converged:
        numIterations += 1
        #Create a list containing k distinct empty lists
        newClusters = []
        for i in range(k):
            newClusters.append([])

        #Associate each example with closest centroid
        for e in examples:
            #Find the centroid closest to e
            smallestDistance = e.distance(clusters[0].getCentroid())
            index = 0
            for i in range(1, k):
                distance = e.distance(clusters[i].getCentroid())
                if distance < smallestDistance:
                    smallestDistance = distance
                    index = i
            #Add e to the list of examples for appropriate cluster
            newClusters[index].append(e)

        for c in newClusters: #Avoid having empty clusters
            if len(c) == 0:
                raise ValueError('Empty Cluster')

        #Update each cluster; check if a centroid has changed
        converged = True
        for i in range(k):
            if clusters[i].update(newClusters[i]) > 0.0:
                converged = False
    return clusters

def trykmeans(examples, numClusters, numTrials):
    """Calls kmeans numTrials times and returns the result with the
          lowest dissimilarity"""
    best = kmeans(examples, numClusters)
    minDissimilarity = dissimilarity(best)
    trial = 1
    while trial < numTrials:
        try:
            clusters = kmeans(examples, numClusters)
        except ValueError:
```

```
            continue #If failed, try again
        currDissimilarity = dissimilarity(clusters)
        if currDissimilarity < minDissimilarity:
            best = clusters
            minDissimilarity = currDissimilarity
        trial += 1
    return best

def printClustering(clustering):
    """Assumes: clustering is a sequence of clusters
       Prints information about each cluster"""
    for c in clustering:
        numPts = 0
        for p in c.members():
            numPts += 1
        print('Cluster of size', numPts)
        for p in c.members():
            print('   ' + p.__str__())

#Patriots' example
edelman = ['edelman', 70, 200]
hogan = ['hogan', 73, 210]
gronkowski = ['gronkowski', 78, 265]
gordon = ['gordon', 75, 225]
hollister = ['hollister', 76, 245]
dorsett = ['dorsett', 70, 192]
allen = ['allen', 75, 265]
cannon = ['cannon', 77, 335]
brown = ['brown', 80, 380]
shelton = ['shelton', 74, 345]
guy = ['guy', 76, 315]
mason = ['mason', 73, 310]
thuney = ['thuney', 77, 305]
karras = ['karras', 76, 305]
mason = ['mason', 73, 310]
michel = ['michel', 71, 215]
white = ['white', 70, 205]
patterson = ['patterson', 74, 228]
develin = ['develin', 75, 255]
brady = ['brady', 76, 225]

receivers = [edelman, hogan, gordon, dorsett]
dLine = [guy, shelton]
oLine = [brown, cannon, mason, karras, thuney]
tights = [gronkowski, hollister, allen]
backs = [michel, white, patterson, develin, brady]

pats = []
for p in receivers:
    pats.append((p, 'receiver'))
for p in dLine:
    pats.append((p, 'Dline'))
```

```
for p in oLine:
    pats.append((p, 'Oline'))
for p in tights:
    pats.append((p, 'tight end'))
for p in backs:
    pats.append((p, 'back'))

class Player(Example):
    pass

def buildPatriotsData(players, toScale = False):
    heightList, weightList = [],[]
    for p in players:
        heightList.append(p[0][1])
        weightList.append(p[0][2])
    if toScale:
        heightList = scaleAttrs(heightList)
        weightList = scaleAttrs(weightList)
    #Build points
    points = []
    for i in range(len(players)):
        features = np.array([heightList[i], weightList[i]])
        features = np.array([heightList[i], weightList[i]])
        points.append(Player(players[i][0][0], features, players[i][1]))
    return points
```

Handin Procedure:

1. **Save** each problem in their respective .py files.

2. **Run** your file to make sure it has no syntax errors. When you run your file, it should not output anything. That is, comment out any code that calls the function we ask you to write.

3. **Upload** each file: q1.py, q2.py, q3.py, q4.py to the 6.00 submission site. You may upload new versions of each file until the deadline, but anything uploaded after that time will not be counted.

4. **Verify** that each of your files have been successfully uploaded. If you do not see their content on the micropset submission site, then they were not successfully uploaded.