Faculty of Arts and Sciences
Department of Computer Science
CMPS 200 – Introduction to Programming
Assignment 12 – Due Dec 5, 2018

Reading material: section 12.1, chapter 13 and chapter 23 from textbook

1. Truck Loading.

In this problem, you need to write a program that emulates loading a truck with a set of items, which presents a variation of the 0/1 Knapsack problem. Given a truck of limited volume, the goal of your program is to load it with the most valuable items. You should first create a class `Item` that models the items that can be loaded into a truck. Each item is characterized by `type`, `value` and `volume`. In addition to getters and setters for accessing the item fields, the class `item` should implement the following methods:

- `__init__()` the constructor, which takes as arguments a string representing type, and two integers representing volume and value, respectively.
- `__str__()`, which returns a string representation of an item in the following format: "type: value, volume" (e.g., couch: 50\$, 50 m^3).

You will then need to create a class called `Truck` that comprises as fields the `id`, a list of loaded items `items` and the maximum volume of the truck, `volume`. In addition to getters and setters for accessing the truck fields, the class should implement the following methods:

- `__init__()` the constructor, which takes as arguments a string representing the truck id and an integer representing its volume, and it should set its items to None.
- `__str__()`, which returns a string representation of a truck in the following format: "id: volume" (e.g., T112: 200 m^3).
- `loadbf()`, which takes as argument the list of items to load the truck with. It uses a brute force strategy by generating the powerset of all the provided items, and then sets the truck's items as the subset of items that provides the highest value and whose total volume does not exceed the truck's volume.
- `loadgreedy()`, which takes as arguments the list of items to load the truck with and a string criteria (i.e., key function), which is used to choose which items to load the truck with. This method should load the truck by picking the items in a greedy manner based on the provided criteria until the truck is full.
- `loaddp()`, which uses dynamic programming to load the truck. Again, this method takes as argument the list of items to load. It should make use of *memoization* to efficiently choose the subset of items to load the truck with, given its volume.

You should implement a program `loadtruck.py` that creates a `Truck` object and loads it using your three different methods. In the process of doing so, you need to prompt the user for the Truck parameters (i.e., id, volume). You also need to create a function that generates a random list of n items, where n is also provided by the user. Feel free to write additional helper functions if you need them!

2. Longest Common Subsequence.

In this problem, you need to write a program that uses dynamic programming to find the longest common subsequence between pairs of DNA sequences. A DNA sequence can be viewed as a string composed of four characters A, C, G and T, representing nucleotides. For example, the following strings can all be viewed as DNA sequences:

- GCGGGTCCCGCTGTTGCCTA
- CGTCCGTAGCGTACAACTTG
- CTCGGGCTCTACCGGTATCA

Your program should consist of the following methods:

- `generate_sequences()`, which takes two arguments, n the number of DNA sequences to generate, and m the length of each sequence. It should generate random strings of length m using the characters A,C,G and T. It should return a list of the n generated DNA sequences.

- `LCS()`, which takes as argument a pair of DNA sequences represented as strings $s1$ and $s2$, and returns the length of their longest common subsequence by building a memoization table L with $len(s1) + 1$ rows and $len(s2) + 1$ columns as follows:

$$L[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L[i-1][j-1] + 1 & \text{if } i,j > 0, s1[i-1] = s2[j-1] \\ max(L[i-1][j], L[i][j-1]) & \text{if } i,j > 0, s1[i-1] \neq s2[j-1] \end{cases}$$

  The length of the longest common subsequence between the two sequences $s1$ and $s2$ can then be found in the cell $L[len(s1)][len(s2)]$.
- `find_LCS()`, which takes as argument a pair of DNA sequences $s1$ and $s2$, and the memoization table L you built using the previous function `LCS()`, and returns the longest common subsequence between $s1$ and $s2$.

Your program should read two command line arguments n and m, representing the number of sequences to generate and their length, respectively. It should then generate n random sequences of length m each and then finds the longest common subsequence between every pair of sequences. Finally, it should print every pair of sequences, along with their longest common subsequence and the length of that subsequence.

For example, assume your run your program as follows `python lcs.py 3 5` and assume that the generated 3 DNA sequences are as follows:

- GTTCG
- GGGGC
- CTCGG

Your program should print the following output:

- LCS(GTTCG,GGGGC) = GG, 2
- LCS(GTTCG,CTCGG) = TCG, 3
- LCS(GGGGC,CTCGG) = GG, 2

3. K-means Clustering.
In this problem, you will implement the k-means clustering algorithm to cluster a set of data instances into K clusters. You should assume that your input would consist of N instances $\{x_1, x_2, \ldots, x_N\}$, each consisting of d real-valued features. The k-means clustering algorithm is given in the following pseudocode:

```
Randomly initialize $K$ cluster centroids $\mu_1, \mu_2, \ldots, \mu_K$
repeat
  for $i = 1, 2, 3 \ldots, N$ do
    assign $x_i$ to cluster $C_k$ where $dist(x_i, \mu_k)$ is minimum
  end for
  for $k = 1, 2, 3, \ldots, K$ do
    $\mu_k = \sum_{x_i \in C_k} \frac{x_i}{|C_k|}$
  end for
until cluster assignment does not change
Return $C_1, C_2, \ldots, C_K$
```

Design and implement a program `kmeans.py` which implements the k-means clustering algorithm described above. You should think about what classes to use to represent data instances and clusters. The centroid of a cluster is computed as the *mean* (i.e., average) of the instances that belong to the cluster, hence the name k-means clustering. You could assume that the distance between two instances or the distance between a data instance and a cluster centroid can be measured using the *Euclidean* distance between their features. You need to think about what methods to use and operations to override to compute distances between instances, to compute centroids of clusters and to assess whenever a centroid of a cluster does not change.

You program should read a .csv file consisting of comma-separated values (N rows and d columns), representing the N instances to be clustered, and a value K representing the number of clusters. To start off your program, you should set the centroids of your K clusters as K instances chosen *randomly* from the provided N instances. Your program should then print out the final clusters into a .csv file, which has the same format and data as the input file but with an additional column at the end representing the cluster each instance belongs to. You can simply use an integer between 1 and K to represent each cluster.

You should test your program on the provided file wine.csv, which you can download from Moodle. The file consists of a set of wine samples represented using 13 features such as the percentage of Alcohol in the sample, Ash, Acidity, Color Intensity, etc. You should *skip* the header row, and use the remaining rows as the data instances to be clustered. To determine the number of clusters K, we will use two different strategies:

1. Given that there are three different types of wine, namely red, white and rosé, you will set $K = 3$ and run your program and save the output as *wine3.csv*.
2. Although there are three main types of wine, there are many variants of each (Sauvignon blanc, Chardonnay, Pinot noir, etc.). Thus, instead of clustering the samples into only three clusters, you will try to find the optimum number of clusters K based on the data. To this end, run your k-means clustering program using different values for K (from 2 till 10) and for each case, compute the *average* distance between the K centroids. Finally, pick K as the one which corresponds to the *least* average distance between the centroids. You should save the output of the clustering with this chosen K as *wineK.csv*, replacing K with its value. How many clusters did you get? Does this number reflect some "plausible" grouping of wine? Provide the answers as comments in your code.

4. Hierarchical Clustering.
Another strategy to determine the optimum number of clusters K is to use the hierarchical Clustering algorithm, which is given in the below pseudocode:

```
Start with each instance in its own cluster
repeat
    Among the current clusters, pick the two clusters $C_i$ and $C_j$ that are most similar
    Replace $C_i$ and $C_j$ with a single cluster $C_i \cup C_j$
until $\min_{i,j} dist(\mu_i, \mu_j) > \tau$
Return the top-level clusters only
```

The algorithm works as follows. You start by creating one cluster for each of the N instances to be clustered, each consisting of a single instance and whose centroid is that instance. It then proceeds by computing the pairwise distances between every two clusters, where the distance between two clusters is again the Euclidean distance between their two centroids. It then merges the pair of clusters whose distance is the smallest. It then repeats the same procedure until the *minimum* distance between every pair of clusters is greater than some threshold value τ. To merge two clusters, your program should create a new cluster, which consists of the instances in both clusters and whose centroid is the mean of the instances in both clusters. You should also remove the two merged clusters from the current set of clusters.

You should write a program `hierarchical.py`, which implements the above algorithm. Your program should take two arguments, the instances to be clustered and the threshold value τ that will be used by the algorithm to determine when to stop the hierarchical clustering. You should make use of the classes and methods you implemented for the k-means approach. You should also test your program using the wine samples in the *wine.csv* file used in the previous problem. The output of your program should be the final set of clusters in the same format as in the previous problem. Your program should try out different values of the threshold τ (10, 100, 1000), and for each outputs a separate file. What do you observe about the effect of the different τ values on the number of clusters? Provide the answer as a comment in your code.

Zip your files in a single archive file `asst12_`***`netid`*** where ***`netid`*** is your AUBnet user name. Your submission to Moodle must be received by noon of the due date. Late submissions will get no grade.