CSE 232 Fall 2017

Programming Project #7

## Assignment Overview

This project focuses on the use of maps. It is worth 50 points (5% of your overall grade). It is due Monday 10/30 before midnight

## The Problem

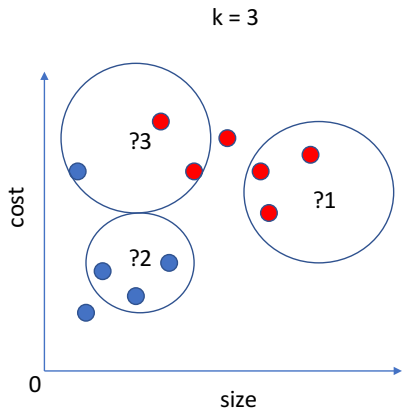
Machine learning is a common term used these days. One of the many areas where machine learning is used is in pattern recognition, specifically in the area of classifying a "thing" as to its type. This is the overall problem we are going to address today.

### The k-nearest neighbor classifier

A relatively simple, though potentially computationally expensive, algorithm to classifying an unknown element in terms of known elements is called a k-nearest neighbor classifier, a knn for short. It works as follows.

Imagine you have two measurements you have performed on a set of objects: let's say size and cost. We call these the *features* of the classification problem. You plot (in 2D for two features, in higher dimensions for more features) the location of all the known objects with respect to their feature values on a graph. You also *label* each of the known objects as to their known class, here indicated by the colors red and blue.

We then place an unknown object (represented by ?), whose class we do not know, into the graph based on its features and find its k nearest neighbors, in this case k=3. The situation is shown below



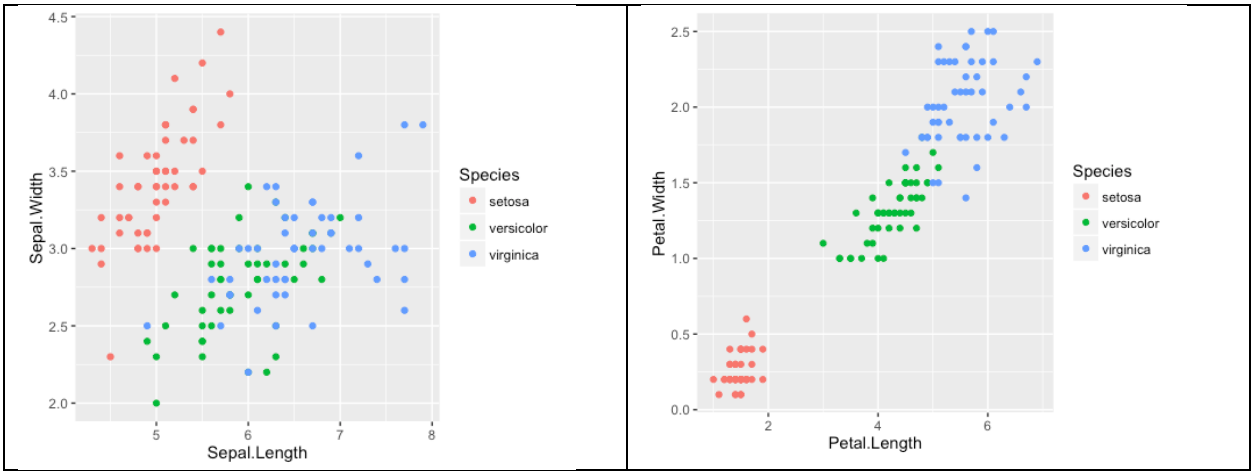

We can then answer the following questions as to what "class" the unknowns are based on their three nearest neighbors in the 2 feature space.

- ?1 is a "red" thing
- ?2 is a "blue" thing
- ?3 is 2/3 red and 1/3 blue

Pretty easy. Computationally expensive because to find the 3-nn or 5-nn of an element we must measure the distance between the unknown and *every other element* in our known set, then find the 3 or 5 (depending on the value of k) closest.

**The Iris dataset**

When you do classification, the first standard data set you run into is called the Iris data set (https://en.wikipedia.org/wiki/Iris_flower_data_set) . It was generated in 1936 by Ronald fisher and has 4 features used in classifying three types of iris flowers, 50 examples of the three classes. Here is a nice plot of sepal length vs width and petal length vs width for each of the 150 flower examples, each colored as to is iris type (from https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor/ ).




You can tell that it seems quite possible to distinguish the 3 iris types based on these 4 features. This is why this set is often used as a test set. Only 4 features, 3 easily separated classes. If your approach can't do iris, then your approach isn't very good.

**The overall approach**

The approach we are going to take is not really the most efficient, using STL maps to store information, but it isn't that bad either and we need the practice. If you are tempted to do it another way (perhaps by finding a solution on the internet????) then be aware that, because `proj07_functions.h` is very specific, you will have to use STL maps. There are a couple of ideas we need to understand.

**Map of vectors to strings**

It turns out that you can use a `map` of the form `std::map<vector<double>, string>` where the `vector`, consisting of `doubles` for each feature in our problem is the ***key*** and the `string` is the ***classification*** of this known element. If we read in data from a data set, we will store the `vector<double>` of features in the `map` as the key and use the `string` as a classification. More about this later.

**Euclidean distance**

We need to measure the "distance" of one object (represented by its features, a `vector<double>`) and all the other objects in the `map`. There are various measures of distance, but the easiest is Euclidean distance (https://en.wikipedia.org/wiki/Euclidean_distance ). The general formula is in an n-dimensional space would be:

$$\mathrm{d}(\mathbf{p},\mathbf{q}) = \mathrm{d}(\mathbf{q},\mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$

$$= \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}.$$

**Test**
The best way to test how well your data set is might work with a knn and a specific k is to do some testing. A good testing approach is called a "leave-one-out" strategy. It works like this. For every object you have in your known set, for example for all 150 examples in the iris test set, you take one out of the set and test it against the remaining set, for iris the remaining 149. As the leave-one-out element is also known, you can measure the accuracy of each element of the known elements. The overall accuracy, the accuracy of all 150 elements in the iris set for example, gives you an overall accuracy of the data set for a knn and a particular k.

**Your Tasks**
Complete the Project 7 by writing code for the following functions. Details of type for the functions can be found in `proj07_functions.h` (provided for you, see details below). There are some supporting functions here that will help you solve and debug the problem.

**function split** : `vector<string> split(const string &s, char delim)`
You supposedly wrote this for lab06. Takes in a string, splits it into its parts as separated by the character `delim`. In the header the default for `delim` is a space (note again you only do the defaults in the header, not in the code itself). Returns a `vector<string>` for each string in the `delim` separated string `s`.

**function read_data** : `void read_data(map<vector<double>, string> &m, unsigned int feature_count, ifstream &inf)`
Reads in every line of data from an already opened `ifstream` reference and fills in map `m`. Each line of the file consists of `feature_count` features (for iris, 4). Each feature is a double and each is separated by a comma. The last element is a string, the class type for this feature combination. An empty map is passed in by reference and filled by the function.

- uses `split`

**function pair_to_string**: `string pair_to_string(const pair<vector<double>, string> &p)`
Takes in a single element of the map, a `pair<vector<double>, string>` , and returns a string. Look at Mimir for the format

- uses fixed and setprecision(3)

**function print_map**: `void print_map(const map<vector<double>, string> &m, ostream &out)`
Takes in our map and prints it to the already opened `ostream & out` (could be `cout`, could be a file).

- uses `pair_to_string`

**function distance**: `double distance(const vector<double> &v1, const vector<double> &v2, unsigned int feature_count)`
Takes in two feature vectors, both of the size `feature_count`, and returns the Euclidean distance between those to vectors as a double.

**function k_neighbors :** `map<vector<double>, string> k_neighbors(const map<vector<double>, string> &m, const vector<double> &test, int k)`
This is the big one! Takes in our map `m` (containing the contents of the file we read in) and a feature `vector<double>` called `test`, and returns a smaller version of the original map which contains only the `k` elements, the `k` nearest neighbors, found in `m` when measured against `test` **excluding** the case `test`.

- uses `distance`

**function test_one** : `double test_one(const map<vector<double>, string> &m, pair<vector<double>, string> test, int k)`
Takes in our map `m` and one particular map element (a pair) `test`. Runs `k_neighbors` to find the `k` nearest neighbors. Measures whether each neighbor found in the k-neighbors matches with the label of the `test` element. Returns a double which is the percentage of times the near-neighbors were the correct label (the one that came in with `test`). Range of values should be from 0.0 (non matched) to 1.0 (they all matched).

- uses `k_neighbors`

**function test_all** : `double test_all(const map<vector<double>, string> &m, int k)`
This is the leave_one_out test. Tests every element in the map `m` against the remaining elements in `m`. Returns an overall accuracy, the average accuracy of each element of the map when tested agains the rest of the map `m`.

- **test_all** is special! It is **always hidden** in Mimir and uses a different file than little.txt (which is what most of the tests use). It is also extra credit in that the full 50 points is already available in the other tests (plus the Manual points).

## Test Cases
Test cases are provided in Mimir as always.

## Assignment Notes
1. You are given the following files the proj07 directory:
    a. `proj07_functions.h` – This file is the header file for the project. It will be used as is in Mimir testing.
    b. `iris.data.txt`, the full iris data set
    c. `little.txt`, a small subset of the full iris, much easier to work with and used in many of the test cases on Mimir
2. You will write and turn in only `proj07_functions.cpp`

## Deliverables
`proj07_functions.cpp` -- your completion of the functions `proj07_functions.h`

1. Remember to include your section, the date, project number and comments.
2. Please be sure to use the specified full directory and file name, i.e. `proj07/proj07_functions.cpp`
3. Submit to Mimir.