CS 540: Introduction to Artificial Intelligence
Homework Assignment #1

Assigned: 9/11
Due: 9/25 before class

Hand in your homework:

This homework includes a written portion and a programming portion. Please type the written portion and hand in a printed hard-copy in class. All pages should be stapled together, and the first sheet must include a header with: your name, Wisc username, class section, HW #, date — and, if handed in late, how many days late it is. The programming portion will be written in Java. All files needed to run the code (including any support files you've written) should be collected and compressed as one zip file named <Wisc username>_HW1.zip. This file should then be uploaded to the appropriate place on the course Moodle website.

Late Policy:

All assignments are due **at the beginning of class** on the due date. One (1) day late, defined as a 24-hour period from the deadline (weekday or weekend), will result in 10% of the total points for the assignment deducted. So, for example, if a 100-point assignment is due on a Wednesday 11 a.m., and it is handed in between Wednesday 11 a.m. and Thursday 11 a.m., 10 points will be deducted. Two (2) days late, 25% off; three (3) days late, 50% off. No homework can be turned in more than three (3) days late. Written questions and program submission have the same deadline. A total of two (2) free late days may be used throughout the semester without penalty.

Assignment grading questions must be raised with the instructor within one week after the assignment is returned.

Collaboration Policy:

You are to complete this assignment individually. However, you are encouraged to discuss the general algorithms and ideas with classmates, TAs, and instructor in order to help you answer the questions. You are also welcome to give each other examples that are not on the assignment in order to demonstrate how to solve problems. But we require you to:

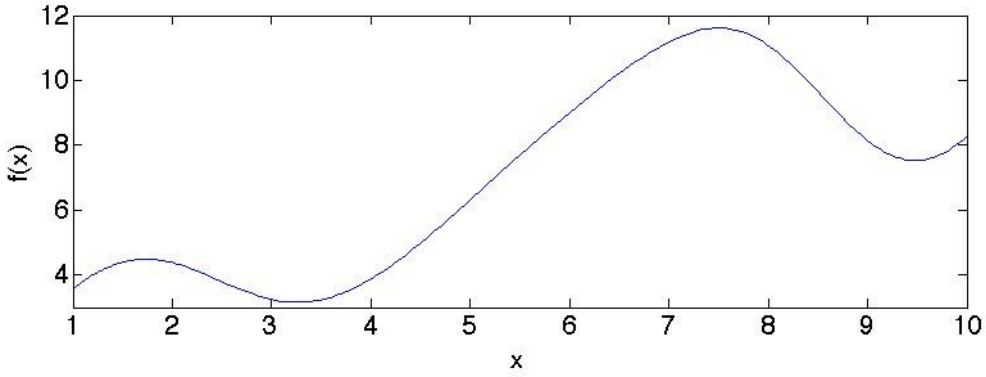
- not explicitly tell each other the answers
- not to copy answers or code fragments from anyone or anywhere
- not to allow your answers to be copied
- not to get any code on the Web

In those cases where you work with one or more other people on the general discussion of the assignment and surrounding topics, we suggest that you specifically record on the assignment the names of the people you were in discussion with.

Question 1: Warm Up Math Questions [20]

Part a)

a.1) [5] Take the derivative with respect to x: $f(x) = e^{\sin x} + x \ln(3 + \cos x)$



a.2) [5] Use a calculator to verify: $f'(x) \approx (f(x + \varepsilon) - f(x - \varepsilon)) / (2\varepsilon)$, with $x = 5$, $\varepsilon = 0.001$.

Part b) You are invited to play on Monty Hall's game show. In one of the games, you are asked to choose one of three doors. Behind one of the doors is a prize, while nothing is behind the other two. First: you choose one of the doors, but that door isn't opened. Instead, Monty opens one of the other two doors you have not chosen. Importantly, Monty knows the location of the prize and will only ever open a door which **does not** contain the prize. Monty then asks you if you would like to change your guess, to the remaining unopened door, or to stay with your original guess.

b.1) [5] What is the probability that the door you originally chose is the one with the prize (after Monty has opened one door)? In other words, what is the probability that you win if you stay with your original choice? (Hint: use Bayes rule)

b.2) [5] What is the probability that you will find the prize if you switch your choice to the other unopened door (after Monty has opened one door)? Should you switch?

Question 2: Hierarchical Clustering [20]

a) [10] Given a dataset with five points $\{x_1,...,x_5\} = \{(0,0), (1,1), (2,1), (0,3), (2,3)\}$, run Hierarchical Clustering by hand (you can use a calculator) using complete-linkage and L_1 distance (Manhattan distance). For each iteration, show the cluster membership of each point. The easiest way to do this is by drawing a dendrogram where in iteration 1, all points are in their own clusters.

b) [10] An expert insists that points (0,0) and (2,1) **MUST NOT** be in the same cluster at any time. Repeat part a) with this constraint in mind, again starting from the first iteration.

Question 3: k-means [20]

a) [6] Given the 6 data points $\{x_1 =(1,2), x_2 =(2,2), x_3=(4,2), x_4=(1,1), x_5=(2,1), x_6=(4,1)\}$, run k-means clustering by hand with k = 2 (you can use a calculator), with the initial centroids at the same positions as x_2 and x_5. Use L_2 distance (Euclidean distance).

b) [6] Repeat part a) with initial centroids at the same positions as data points x_2 and x_3.

c) [8] Show the objective function of k-means. Which clustering leads to a better result, part a) or b), in terms of your objective function? Justify your choice.

Question 4: Programming: k-NN Classification [40]

For this problem, you will be building a k-NN classifier and measuring its performance for different k.

The Data:

The data you will be classifying a set of words, each word the name of either a machine, a fruit or a nation. For each word, a set of 3 real valued features have been calculated (using a technique known as a deep neural network)[1]. This kind of representation of words in a continuous, real valued feature space is known as *word embedding*. Each word is *embedded* in this three dimension space, allowing us to calculate "distances" between words.

In addition to the embedding feature values, we can consider each word as belonging to one of three categories: *machine*, *fruit* or *nation*. A total of 80 words were sampled, their word embedding features calculated and their categories assigned. This set of instances was then split into training and test sets (60 for training and 20 for test data) and are provided to you.

Figure 1 shows the provided training data (reduced to two dimensions for display). Words are colored according to their category (*red*, *blue* or *green* for categories *machine*, *fruit* or *nation*, respectively). Note that the actual feature space is 3-dimensional (The data was reduced to two dimensions using a technique called Multidimensional Scaling).

You will write a k-NN classifier to learn from a set of training data and return the classifications for a test set.

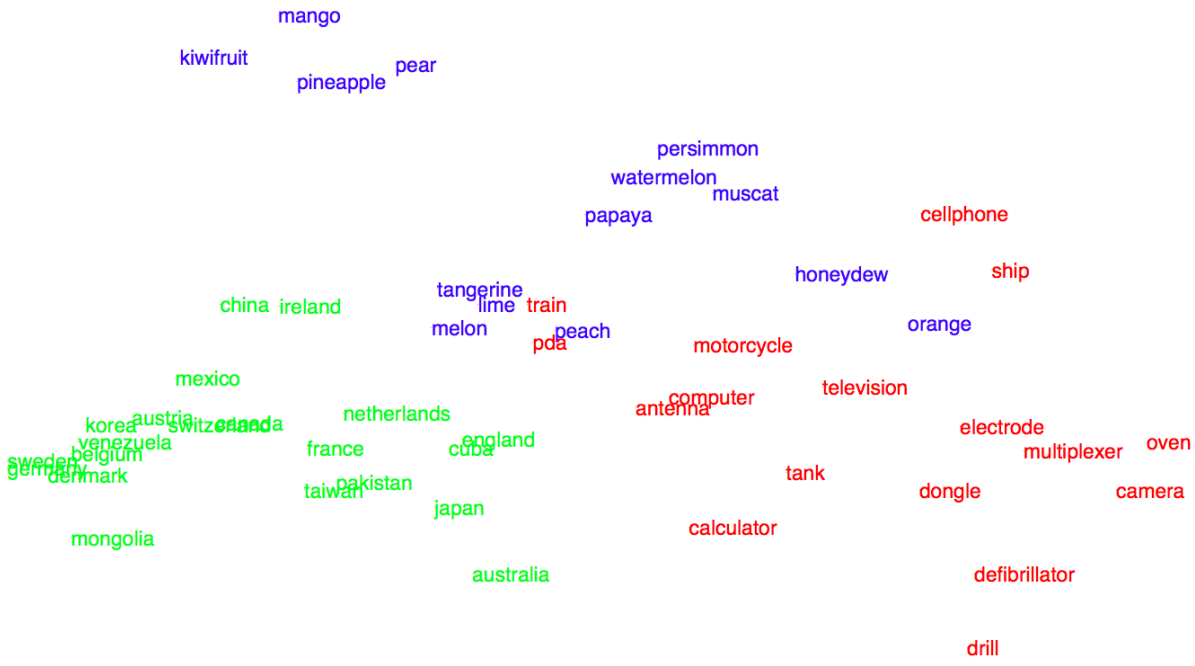


Figure 1

1 R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. 2011. Natural language processing (almost) from scratch. Journal of Machine Learning Research, 12:2493–2537.

Code:

You are provided a set of skeleton code and sample data in "`HW1.zip`", located on the course website. The provided code handles input and output. Your job will be to implement a single empty method, `KNN.classify()`, described below, which will perform k-NN classification on a set of data and return several pieces of information related to that categorization.

IMPORTANT: Please **DO NOT** modify any of the provided code outside of this one method. You can of course add any supporting methods or classes you might need, though note that a set of utility methods are provided. Please take care to include any new `.java` files in your hand in zip file which are necessary for your code to run.

This `HW1.zip` package includes the files:

```
HW1/src/HW1.java       % wrapper class for running the classifier
   |    Item.java      % each training or test item is an instance of Item
   |    KNN.java       % THIS CLASS CONTAINS THE METHOD YOU MUST MODIFY
   |    KNNResult.java % a class to hold the results of classification
   |    Utils.java     % a set of potentially useful utility methods
   |
   /data/train.txt      % a sample set of training items
         test.txt       % a sample set of test items
   /HW1.pdf             % this document
```

When you open the file `src/KNN.java` you will see the method you must complete:

```
public class KNN {
  public KNNResult classify (  Item[] trainingData,
                               Item[] testData,
                               int k ) {

    /* ... YOUR CODE GOES HERE ... */

    }
}
```

Arguments:

`Item[] trainingData`
An array of the training instances. Each element in this array represents an instance in the training data, which includes the instance's category, name, and feature vector. The `Item` class is described in more detail below.

`Item[] testData`
An array of the test instances that your classifier must provide a classification for. The test instance array has the same structure as the training array. NOTE: the test data includes the correct categorization for each item. This "ground truth" category for each test instance is also stored in the corresponding `Item` element in the `testData` array. This information is used when calculating the accuracy of your classifier (categorization of an item is correct when the predicted category is equal to the ground truth category). You should IGNORE the correct category labels when you first make predictions (ie don't peek!) and only use this information when calculating accuracy.

`int k`
The integer for nearest neighbor selection.

Returns:

The method `KNN.classify()` must return an object of type :

```
public class KNNResult {
  double accuracy;             % accuracy on the test set (#correct/total)
  String[] categoryAssignment; % one of 'machine','fruit', or 'nation'
  String[][] nearestNeighbors; % 2D array of names of k nearest neighbors
                               %  for each test item, one row per test item
}
```

defined in `KNNResult.java`. Within your `classify()` method, you will need to write code which fills in these values. This object contains information about performance of the classifier on the test set. We will use the contents of this object to verify the correctness of your classification algorithm.

Each `KNNResult` object includes:

`double accuracy`
The performance on the test set. This is the accuracy (number of correctly classified test instances divided by the total number of test instances). Contains a value between 0 and 1.

`String[] categoryAssignment`
An array of assigned categories for the test set, in the same order as the instances in the input test file.

`String[][] nearestNeighbor`
An array of names of the k nearest neighbors for each test instance, again with each row corresponding to a test instance in the order of the test file input. The columns must contain the k nearest neighbors to that item, ordered in distance from nearest to farthest. The number of columns depends on the value of k provided.

Additional Classes:

The method `KNN.classify()` accepts an array of training instances (or items) and an array of test instances. Each training item is represented as an instance of the Item class, defined in `Item.java`:

```
public class Item {
  String category;    % the category of this item
  String name;        % the name of the item (the word itself)
  double[] features;  % array of 3 feature values
}
```

The remaining class, `Utils.java`, contains helper methods which you may find useful as you complete the classifier method.

To run:

The HW1 class contains a main method which takes the following arguments:

java HW1 <trainingFileName> <testFileName> <k> <outputFlag>

The outputFlag controls what the program outputs:

1 – The value of accuracy given k
2 – The array of category assignments
3 – The array of k nearest neighbor assignments

For example:

java HW1 train.txt test.txt 5 1

with use train.txt with k=5 to classify items in test.txt and print the accuracy.

Notes:

- In all cases, where distance must be calculated, use Euclidean distance.

- In some cases a tie may occur. For example, if K = 5, and the 5 nearest neighbors for an instance are 2 items with category "fruit", 2 "machine" and 1 "nation", there is a tie between "fruit" and machine". It's not clear which category to apply. When ties occur, break them using the priority order: **nation, machine, fruit**. That is, if there is a tie, choose "nation" first, then "machine", and lastly "fruit". In our example, you would choose "machine", because "machine" has higher priority than "fruit".

- You are **NOT** responsible for implementing any file input or console output. This is already written in the code provided to you.

- Again, ake a look at the `Utils` (Utils.java) class before you start. There may be methods implemented there that you'll want to use.

- Your code must compile by calling "javac *.java" in the folder of the unzipped code skeleton.

- To make the problem more concrete, here are the steps the grader will take:
1. unzip the provided zip file
2. remove any "class" files
3. call "javac *.java"
4. call "java HW1 ..." with different sets of arguments and different training and test sets

- **IMPORTANT:** Remember, **DO NOT MODIFY** the existing methods and classes. We will use this structure to evaluate your work. You can add methods or classes as needed to facilitate your classification procedure, but please remember to include them in your final hand in submission.

Data:

Sample training and test set files are provided. Again, note that the provided "test.txt" includes the correct, "ground truth" categorizations so that you can evaluate the performance of your code. When grading your code we will be using different sets of training and test data.

You are strongly encouraged to make training/test sets of your own. Data set files are tab-separated with each line in the data set consists of a category, an instance and their feature representations as below, separated by tabs:

<category> <instance name> <feature dim. 1> <dim.2> <dim.3>

When creating your own training/test sets, you can simply make up feature values which seem reasonable to you and which you can use to test your code.

Deliverables:

Submit a zip file containing your modified version of the code skeleton along with any supporting code you've written. **This file should be named <Wisc username>_HW1.zip.**

For example, bgibson_HW1.zip might contain:

```
bgibson_HW1/HW1.java
            Item.java
            KNN.java            % with my newly coded classify()
            KNNResult.java
            Utils.java          % with some added helper methods
```

Testing:

You will of course want to do some testing of your code. Use both the data files provided and some files of your own creation.

Some questions to ask while testing:

- What should the best k be, measured by accuracy, on this dataset?
- What should the k nearest neighbors be for a particular instance?
- Are there any data points you can expect the classifier to get wrong?
- Does the accuracy make sense given the data?
- and many more...