

# ITI 1121. Introduction to Computing II

## Winter 2014

### Assignment 1 [ PDF ]

(Last modified on January 29, 2014)

**Deadline: Monday January 27, 2014, 18:00**

## Solution

- [a1-solution.jar](#)

## Learning objectives

- Implementing complex data types using fixed-size arrays
- Recognizing the role of reference variable to create class associations
- Applying basic object oriented programming principles to solve problems
- Editing, compiling, debugging, and running Java programs
- Raising awareness concerning the university policies for academic fraud

As a collective, (...) humanity produces five zettabytes of data every year: 40,000,000,000,000,000,000 (forty sextillion) bits. (...) If you wrote out all five zettabytes that humans produce each year by hand, you would reach the galactic core of the Milky Way.

**The Predictive Power of Big Data**  
by Erez Aiden and Jean-Baptiste Michel  
Newsweek, December 25, 2013

## Background information

Enormous amounts of data are generated by sensors, online transactions, scientific experiments, and social media to name but a few examples. Analytics <sup>1</sup> aims to extract meaningful information (knowledge) from large and complex data-sets. This information helps business leaders make rapid and precise decisions, and scientists make discoveries.

Machine learning (ML) is a branch of Computer Science. “[It] is the systematic study of algorithms and systems that improve their knowledge or performance with experience” [1]. The machine learning community develops powerful algorithms enabling data analytics. One of the first tasks to be performed when studying a new data-set is often clustering, which is sometimes referred to as unsupervised learning,

---

<sup>1</sup> A term akin to “Big Data” and “Data Mining”

as it aims to uncover hidden groups (clusters) within the data. Herein, we look at representing data in a systematic machine-readable form, and implementing a well known algorithm, *k*-means (aka Lloyd's algorithm) to uncover clusters.

## Problem statement

**Clustering** consists of organizing examples into groups such that examples in a given group are similar one to another. In order to give a precise definition of similarity, we first need to look at the data representation.

### 1 FeatureVector (35 marks)

Information must be represented in a form that is easily amenable to automated analyses. Often, examples are represented as fixed-length feature vectors.

- **Marketing:** Examples can be surveys and the features are answers to questions. In this case, market segmentation into groups allows to better understand the relationships between individuals to possibly identify large groups that are likely to buy a given product.
- **Transcriptomics:** Examples can be genes and the features can be their expression level under various physiological conditions or sampled from different tissues. In this case, clustering may reveal groups of genes working together (metabolic pathways).

Consider the following data about car makes<sup>2</sup>. The data consists of 38 makes of cars represented using features including mileage per gallon (MPG), weight (WT), horse power (HP), and number of cylinders (Cyl).

	MPG	WT	HP	Cyl
Buick Estate	16.9	4.360	155.0	8.0
Ford Country	15.5	4.054	142.0	8.0
Chevy Malibu	19.2	3.605	125.0	8.0
Chrysler	18.5	3.940	150.0	8.0
...				
BMW 320i	21.5	2.600	110.0	4.0
VW Rabbit	31.9	1.925	71.0	4.0

#### 1.1 Implementation

Each **FeatureVector**<sup>3</sup> has a name (String). A **FeatureVector** uses a fixed-length array to store the values of the features of this example. For this assignment, the values of the features are of type **double**. To facilitate the debugging of our classes, **FeatureVector** has the following behaviour: Following a call to **FeatureVector.setVerbose(true)**, all the calls to the method **toString()** will return a detailed description of the object (see below). A call **FeatureVector.setVerbose(false)** disables this behaviour.

---

<sup>2</sup><http://www.stat.berkeley.edu/classes/s133/Cluster2a.html>

<sup>3</sup>More precisely, "each object of the class FeatureVector...".

## 1.2 FeatureVector

The class **FeatureVector** declares two constructors.

- **FeatureVector(String name, int size)**: Initializes the attributes of this object. Specifically, **name** is a String that represents the name of this example (FeatureVector). A **FeatureVector** created with this constructor has **size** features, each one initialized to 0.0.
- **FeatureVector(String name, double[] elems)**: Initializes the attributes of this object. Specifically, **name** is a String that represents the name of this example (FeatureVector). The parameter **elems** designates an array of values to be used as features for this example. The values must be copied, not shared.

## 1.3 Methods

- **String getName()**: Returns the **name** of this **FeatureVector**.
- **int getSize()**: Returns the **size** of this **FeatureVector**, i.e. the number of features.
- **double featureAt(int index)**: Returns the value of the feature at position **index** of this **FeatureVector**.
- **void featureSet(int index, double value)**: Sets the value of the feature at position **index** to **value**.
- **FeatureVector copy()**: Returns a new **FeatureVector** that is a copy of this one. The method must implement a “deep-copy”, i.e. this object and the copy do not share references to mutable objects.
- **double getDistance(FeatureVector other)**: Returns the Euclidean distance between this feature vector and that of **other**. Specifically, given  $v$  and  $w$ , two feature vectors of length  $m$ , the Euclidean distance is defined as,

$$\sqrt{\sum_{i=1}^m (v_i - w_i)^2}$$

- **void plus(FeatureVector other)**: Adds the feature values of **other** to this feature vector.
- **void div(FeatureVector other)**: Divides these feature values by that of **other**.
- **void div(double value)**: Divides these feature values by **value**.
- **String toString()**: Returns a **String** representation of this object. If the **verbose** option is not set, the method simply returns the **name** of the object, otherwise it returns the **name** of the object, followed by colon, followed by the feature values, separated by comas, and enclosed between curly braces.

```
FeatureVector v;  
v = new FeatureVector("G", new double[] { 7.0, 5.0 });  
System.out.println(v);  
FeatureVector.setVerbose(true);  
System.out.println(v);
```

The above statements display the following on the console:

G  
G: {7.0, 5.0}

- **boolean equals(*FeatureVector* other)**: Returns **true** if this and **other** have the same number of features and the respective values are equal.
- **void setVerbose(*boolean* value)**: After a call to **setVerbose(true)** calls to the method **toString()** will return the verbose representation of the object.

## 2 Cluster (30 marks)

A **Cluster** is a collection of examples (objects of the class **FeatureVector**).

### 2.1 Implementation

A **Cluster** uses a fixed-size array, of size **capacity**, to store **FeatureVector** objects.

### 2.2 Cluster

- **Cluster(int capacity)**: The constructor initializes this **Cluster** to store a maximum of **capacity** **FeatureVector** objects.

### 2.3 Methods

- **int getSize()**: returns the number of **FeatureVector** objects that are currently stored in this **Cluster**.
- **boolean add(*FeatureVector* elem)**: Returns **false** if this **Cluster** has reached its maximum capacity, i.e. it currently stores **capacity** objects. Otherwise, **elem** is stored at the next available position of the fixed-size array, and the method returns **true**. The convention is that the first added element is stored at index 0, the next element at position 1, etc.
- **FeatureVector getElementAt(int index)**: returns the **FeatureVector** stored at position **index** of this **Cluster**.
- **FeatureVector getCentroid()**: Returns a **FeatureVector** representing the centroid of this **Cluster**. A centroid, is a **FeatureVector** such that each of its feature is the average value of the corresponding feature of all the feature vectors of this cluster. By convention, the method returns **null** if this **Cluster** is empty.
- **double getVariance()**: Returns the variance of **this Cluster**. Specifically, let  $v_i$  be the  $i$ -th feature vector of this cluster,  $l$  the number of feature vectors currently stored in this cluster,  $c$  be the centroid, and  $d(v_i, c)$  be the distance between the feature vectors  $v_i$  and  $c$ . The variance is defined as follows,

$$\sqrt{\sum_{i=1}^l d(v_i, c)^2}$$

- **String toString()**: Returns a **String** representation of this **Cluster**. Specifically, the representation starts with the word “Cluster”, followed by colon, followed by the **String** representation of all the examples of the cluster, separated by comas, and enclosed in curly braces.

```

Cluster c;
c = new Cluster(10);
c.add(new FeatureVector("A", new double[] { 0.0, 0.0 }));
c.add(new FeatureVector("B", new double[] { 1.0, 1.0 }));
c.add(new FeatureVector("C", new double[] { 1.0, 0.0 }));
c.add(new FeatureVector("D", new double[] { 2.0, 0.0 }));
FeatureVector.setVerbose(true);
System.out.println(c);

```

The above statements will display the following on the console:

```
Cluster: {A: {0.0, 0.0}, B: {1.0, 1.0}, C: {1.0, 0.0}, D: {2.0, 0.0}}
```

### 3 Clustering (25 marks)

The class **Clustering** implements a  $k$ -means algorithm adapted from [1, 2]. The class has a single public method:

- **static Cluster[] kmeans(FeatureVector[] examples, int k)**

This problem is known to be difficult to solve exactly when the number of examples is large. The  $k$ -means algorithm is an approximate method that makes the “best” local choice at each iteration without consideration of the global solution of the problem. We call this a “greedy” strategy. Comparing the examples to the centroid of the clusters reduces considerably the total number of comparisons as it avoids comparing all-against-all examples and the number of centroids (clusters) is generally small. This makes the  $k$ -means algorithm quite efficient.

#### Input:

The input consists of  $n$  examples (objects of the class **FeatureVector**) and  $k$ , the number of clusters to be produced.

#### Output:

A partition of the  $n$  input examples into  $k$  groups (clusters). The partition is such that the elements of the same cluster are closer to the centroid of the group than they are from the centroid of other clusters.

#### Algorithm:

- Initialization
  - Create  $k$  clusters and let  $D$  designate this set of clusters.
  - Randomly select <sup>4</sup>, without replacement,  $k$  examples from the  $n$  input examples. Each randomly selected example becomes the centroid of a cluster<sup>5</sup>.

---

<sup>4</sup>An object of the class **java.util.Random** has a method **nextInt(int n)** that returns a randomly generated number in the range  $[0, n[$

<sup>5</sup>When a **Cluster** has only one example, that example is centroid!

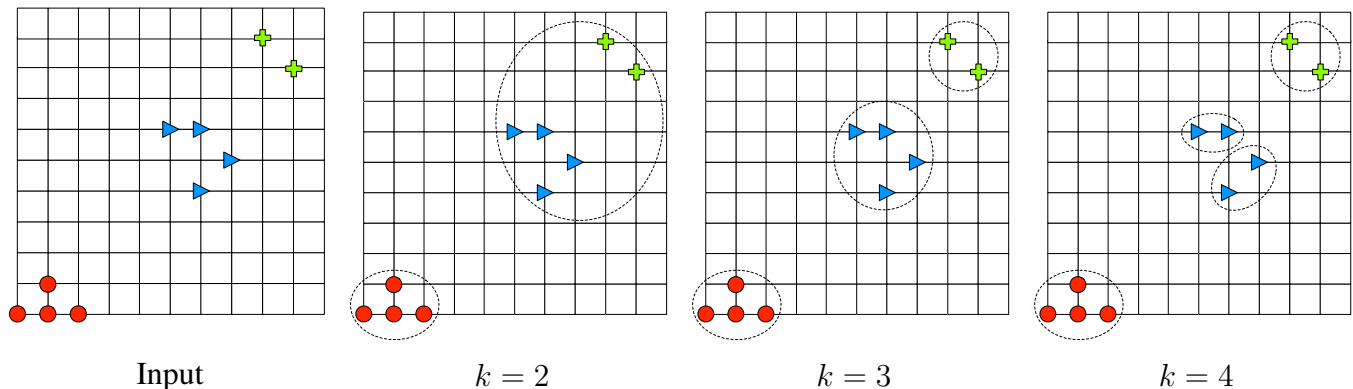
- Repeat
  - Create  $k$  clusters and let  $D'$  designate this set of clusters.
  - For each input example  $e$ , let  $i$  be the cluster of  $D$  such that the distance  $e$  to the centroid of  $D_i$  is minimum. Assign  $e$  to  $D'_i$ , where  $D'_i$  designates the  $i$ -th cluster of  $D'$ .
  - Updating  $D$ .
    - \* Compare the corresponding clusters of  $D$  and  $D'$ .
      - If the size or the centroid has changed, then the respective cluster of  $D'$  replaces that of  $D$ .
    - \* If none of the clusters have changed, the algorithm terminates and returns the current clusters.
    - \* Otherwise, the algorithm resumes its outer loop.

## Utilities.run

Since the algorithm is “greedy”, the solution that it returns is not necessarily optimal. Since examples are randomly selected in the initialization step, each run potentially returns a different solution. For these reasons, we will select the best run out of many. The method **Utilities.run**(**FeatureVector[]** examples, **int k**, **int trials**) returns the solution with lowest cluster variance.

## Synthetic example

**RunSynthetic** illustrates the concepts. Each example has two features, which makes it easy to visualize the data on a two-dimensional surface.



```
FeatureVector[] examples;
examples = new FeatureVector[10];

examples[0] = new FeatureVector("A", new double[] { 0.0, 0.0 });
examples[1] = new FeatureVector("B", new double[] { 1.0, 1.0 });
examples[2] = new FeatureVector("C", new double[] { 1.0, 0.0 });
examples[3] = new FeatureVector("D", new double[] { 2.0, 0.0 });
examples[4] = new FeatureVector("E", new double[] { 6.0, 4.0 });
examples[5] = new FeatureVector("F", new double[] { 5.0, 6.0 });
examples[6] = new FeatureVector("G", new double[] { 7.0, 5.0 });
```

```

examples[7] = new FeatureVector("H", new double[] { 6.0, 6.0 });
examples[8] = new FeatureVector("I", new double[] { 8.0, 9.0 });
examples[9] = new FeatureVector("J", new double[] { 8.0, 9.0 });

FeatureVector.setVerbose(true);

Utilities.run(examples, 3, 10);

```

Executing the above statements produces the following output <sup>6</sup>.

```

Cluster: {A: {0.0, 0.0}, B: {1.0, 1.0}, C: {1.0, 0.0}, D: {2.0, 0.0}}
Cluster: {I: {8.0, 9.0}, J: {8.0, 9.0}}
Cluster: {E: {6.0, 4.0}, F: {5.0, 6.0}, G: {7.0, 5.0}, H: {6.0, 6.0}}

```

## Car makes example

**RunCars** is a more involved example consisting of 38 car makes, each one represented using 6 features.

## 4 Rules and regulation (10 marks)

Follow all the directives available on the [assignment directives web page](#), and submit your assignment through the on-line submission system [uottawa.blackboard.com](http://uottawa.blackboard.com).

You must preferably do the assignment in teams of two, but you can also do the assignment individually. Pay attention to the directives and answer all the questions.

## 5 Academic fraud

This part of the assignment is meant to raise awareness concerning plagiarism and academic fraud. Please read the following documents:

- <http://web5.uottawa.ca/admingov/regulations.html#r71>
- <http://web5.uottawa.ca/mcs-smc/academicintegrity/>
- <http://www.uottawa.ca/plagiarism.pdf>

Cases of plagiarism will be dealt with according to the university regulations.

**By submitting this assignment, you acknowledge:**

1. **having read the above documents, and**
2. **understanding the consequences of plagiarism**

<sup>6</sup>The order of the clusters may vary from one run to another since the examples are randomly selected in the initialization step of the algorithm.

## References

- [1] Peter Flach. *Machine Learning: The Art and Science of Algorithms that Make Sense of Data*. Cambridge University Press, 2012.
- [2] John V Guttag. *Introduction to Computation and Programming Using Python*. MIT Press, revised and expanded edition, 2013.

## Files

- README.txt
- FeatureVector.java
- Cluster.java
- Clustering.java
- RunSynthetic.java
- RunCars.java
- Utilities.java
- StudentInfo.java

Here are some tests, but remember that “testing shows the presence, not the absence of bugs”, Edsger W. Dijkstra.

- FeatureVectorTest.java
- ClusterTest.java

## A Frequently Asked Questions (FAQ)

1. **“Can I use classes from the Java Collection (ArrayList, LinkedList, etc.) in my implementation?”**

No. In a real world situation, you would be using the Collection. However, in ITI1121, you must demonstrate your ability to implement these classes yourself.

2. **“Do we have to handle error situations?”**

No. Until we have seen exceptions in class, which is the mechanism for handling error situations in Java, you can assume that all the input values are valid. However, you do have to make sure that your programs can handle all the valid situations.

3. **“Can we add new methods?”**

You **cannot** add **public** methods, as this would be changing the public interface of the classes, but you can add (and are encouraged to) as many **private** methods as you want.

4. **“When I run RunCars, I get an output that is almost identical to the expected one, but with 1 car misplaced. Do you think this is caused by an error in my code or this can be attributed to the random choice of input examples?”**

The output of the  $k$ -means depends on the initial choice of centroids. Since the initial centroids are randomly selected, the algorithm potentially returns a different solution for each run. I initially



underestimated the minimum number of trials needed to find an “optimal” solution. I ran some tests, printing the total variance, and found that it sometimes take up to 25 runs to get a solution with a total variance of “17516.363363154764”. I have now increased the number of runs from 10 to 100.

It is also possible that there are multiple solutions with the same total variance.

Finally, I wrote “optimal” with quotes, because we don’t know if the solution is optimal. I did 1,000,000 runs, and could not find a better solution. Therefore, this solution is likely to be optimal, but we don’t know for sure. Algorithms like  $k$ -means are said to be heuristic algorithms. This is in contrast with “exact” algorithms that provably always return the optimal solution.

5. **“I can’t seem to compile/run the JUnit tests in DrJava, can you help?”**

Indeed, I had “tested” the tests in Eclipse only. I have now published new tests on the assignment Web page that work in DrJava, Eclipse, as well as the command line.

6. **“In the definition of the cluster variance, your refer to  $v_i - c$  and I am not quite sure what you mean by that.”**

This was an error, I meant the distance between  $v_i$  and  $c$ , where  $v_i$  is the  $i$ -th feaatue vector of a cluster, and  $c$  is the centroid of that cluster.

7. **“Could you post the slides about Assignment #1 that you presented in class?”**

Here is the link:

- <http://www.site.uottawa.ca/turcotte/teaching/iti-1121/assignments/01/iti1121-a1.pdf>

8. **“I am having trouble grasping the concept of having two constructors”**

As you will see, the concept of having two constructors is not very difficult to grasp.

When you are implementing a class, you can think that you are providing services to a user (another programmer).

Having multiple constructors, each with a different signature, is simply to provide flexibility to the user, many ways to create objects from that class.

Lets take the example of a class to represent time.

```
public class Time {

    private int hours;
    private int minutes;
    private int seconds;

    public Time() {
        this.hours = 12;
        this.minutes = 30;
        this.seconds = 15;
    }

    public Time(int hours, int minutes, int seconds) {
        this.hours = hours;
        this.minutes = minutes;
    }
}
```

```

        this.seconds = seconds;
    }

    // ...
}

```

Now, a programmer using the class `Time` has two ways to create objects of the class `Time`.

```

Time t1, t2;

t1 = new Time();
t2 = new Time(13, 45, 0);

```

When using the first constructor, an object is constructed with default values 12:30:15, where as for using the second constructor, the programmer needs to provide values for the attributes.

The compiler can easily select the appropriate constructor by simply matching the list of actual and effective parameters.

#### 9. “My calculation of the variance differs from yours, and I can’t seem to find the error”

Two things. First, there was an error in my method `getVariance()`, where the result of taking the square root of the final sum was not saved. When fixing this, the total variance for the cars example becomes 255.17409734965472. Secondly, I have seen some student solutions where the square root was applied to the terms of the sum, rather than the sum.

**Last Modified: January 29, 2014**