Web 2.0 and Mobile Interaction - Assignment 1

Olivier Jais-Nielsen

October 7, 2009

1 Approach

I decided to implement different kind of clustering on movies information.

More precisely, my program can download information about the current top rated movies from IMDb, using the API IMDbPy and perform clustering on these films based on the data provided by IMDb : the international titles, several plot summaries and the genres attributed to the films. One can easily add any other data provided by IMDb. I choosed the previous because I think any other data (like actors, directors, languages, ...) doesn't describe the films themeselves but their environment and thus wouldn't be relevant. The selection of the words to take into account was inspired by the script given in lecture 3 : only the words present in at least 10% and at most 50% of the movies are kept.

Both the hierarchical and the k-means clusterings are implemented. The Tanimoto and the Pearson similarity measures can be used as well as the inverse Euclidian distance, defined by : $IE(v_1, v_2) = \exp^{-\|v_1 - v_2\|}$

The program produces a box diagram representation of the clusters, including for each movie some information (title, year, rating) and a thumbnail.

2 Python code

2.1 API

In addition to IMDbPy, several classes and functions have been defined to handle the data and the algorithms involved in clustering. A complete documentation of all the modules is given in the document *modules.pdf*. The most important classes are *WordVector* which defines a vector of word occurences, *Tree* which defines a tree used for the hierarchical clustering and *ClustersSet* which defines a set of clusters for k-mean clustering.

All the source files are in the *Sources* folder.

2.2 Main algorithms

The main algorithms are contained in the module *clustering.py.*

2.2.1 Function *hierarchicalCluster(vectors, simMeasure)*

```
"""
This function computes the hierarchical clustering from the WordVector instances in the "
    vectors" list using the similarity measure "simMeasure". It returns a Tree with
    WordVector labels.
```

```
"""

# We create an empty Tree with the appropriate label merger. It will act has the top tree.
tree = Tree(mergeWordVectorsLabels)

vectPairSimilarities = {}

# For each vector, we create a leaf with the appropriate label and we add it as a branch of the top Tree. These are the initial clusters.
for vectID in vectors.keys():
    branch = Tree();
    branch.setLeaf((vectID, vectors[vectID]))
    tree.addBranch(branch)

# While there is more than one cluster at the top level, we iterate.
while tree.countBranches() > 1:

    # We find the highest similarity among the cluster pairs (we store the similarities in vectPairSimilarities in order to avoid calculating several times the same thing)
    maxSimilarity = -1;
    for i in range(tree.countBranches()):
    vectIID, vectI = tree.getBranchLabel(i)
        for j in range(i + 1, tree.countBranches()):
            vectJID, vectJ = tree.getBranchLabel(j)
            sim = None
            vectIID = str(vectIID)
            vectJID = str(vectJID)
            if (vectIID, vectJID) in vectPairSimilarities.keys():
                sim = vectPairSimilarities[(vectIID, vectJID)]
            elif (vectJID, vectIID) in vectPairSimilarities.keys():
                sim = vectPairSimilarities[(vectJID, vectIID)]
            else:
                sim = simMeasure(vectJ, vectI)
                vectPairSimilarities[(vectIID, vectJID)] = sim
                vectPairSimilarities[(vectJID, vectIID)] = sim
            if sim > maxSimilarity:
                maxSimilarity = sim
                bestI = i
                bestJ = j
                bestVectIID = vectIID
                bestVectJID = vectJID

    # We merge the clusters of the corresponding pair
    tree.mergeBranches(bestI, bestJ)
    del vectPairSimilarities[(bestVectIID, bestVectJID)]
```

```
        del vectPairSimilarities[(bestVectJID, bestVectIID)]

# Once there is only one cluster at the top level, we return it
return tree
```

2.2.2 Function *kMeansClustering(vectors, simMeasure, k)*

```
"""
This function computes the k-mean clustering from the WordVector instances  in the "vectors"
    list using the similarity measure "simMeasure" and "k". It returns a ClusterSet
   containing WordVector instances.
"""

# We create a ClusterSet with no clusters and with the set of vectors      clusters =
   ClusterSet(vectors)
# We create k clusters and fill them arbitrarily : the i-th vector belongs to the (i mod[k])
   -th cluster.
i = 0
for vectID in vectors.keys():
        clusters.attributeVector(vectID, i % k)
        i += 1

# While the composition of the cluster differs from the previous iteration, we iterate.
while clusters.hasChanged():
        # We compute the centroid of each cluster for their current composition.
                          clusters.computeCentroids()

    # For each vector :
        for vectID in vectors.keys():

                # We find the highest similarity between the vector and a centroid
                maxSimilarity = -1
                for clusterID in range(k):
                        currentSim = simMeasure(vectors[vectID], clusters.getCenter(
                           clusterID))
                        if currentSim >= maxSimilarity:
                                maxSimilarity = currentSim
                                bestClusterID = clusterID

                # We assign the vector to the cluster corresponding to the found centroid
                                        clusters.attributeVector(vectID, bestClusterID)

# Once the composition of the clusters stops changing, we return the ClusterSet
return clusters
```

2.2.3 Example application

```python
# We create an interface with IMDb
interface = IMDbInterface("data.db")

# We get the IMDb movie IDs corresponding to the Top 250
movieIDs = getIMDbTop250()

# We prepare dictionaries intended to contain, for each movie ID, an IMDb movie object and a WordVector
movies = {}
wordVectors = {}

# For each movie ID
for i in range(0, 50):
        id = movieIDs[i]

        # We get the IMDb movie object
        movies[id] = interface.getMovie(id)

        # We select the texts concerning the movie that we want to use
        list = [movies[id]["canonical title"]] + movies[id]["akas"] + movies[id]["genres"] + movies[id]["plot"]

        # We remove the annotations from these texts : they can be in a text, after a double colon                list = map (lambda text: text.split("::")[0], list)

        # We compute a word histogram from the texts and a WordVector from it
        wordVectors[id] = WordVector(wordHistogramFromTextList(list))


## HIERARCHICAL CLUSTERING


# We create a PIL image object intended to contain the box diagram of the hierarchical clustering
im = Image.new("RGB", (0, 0), (255, 255, 255))

# We compute the clusters for the choosen similarity measure
clusters = hierarchicalCluster(wordVectors, lambda vect1, vect2: similarity(vect1, vect2, "euclidian inverse"))

# We define an appropriate full label writer : it displays the cannonical title, the year and the rating of the corresponding movie
clusters.setFullLabelWriter(lambda (vectID, vect): (u"%s (Year : %i Rating : %i)" % (movies[vectID]["canonical title"], movies[vectID]["year"], movies[vectID]["rating"])).encode("utf-8"))

```

```
# We define an appropriate thumbnail accessor : it displays the corresponding movie's
    thumbnail    clusters.setThumbnailAccessor(lambda (vectID, vect):
interface.getThumbnail(vectID))

# We resize the image to the appropriate size, we draw the diagram and we save the image
im = im.resize(clusters.boxDimensions())
clusters.drawDiagram(im)
im.save("hierarchical_eucl.jpg")


## K-MEANS CLUSTERING


# We create a PIL image object intended to contain the box diagram of the hierarchical
    clustering
im = Image.new("RGB", (0, 0), (255, 255, 255))

# We compute the clusters for the choosen similarity measure and the choosen k (here 6)
clusters = kMeansClustering(wordVectors, lambda vect1, vect2: similarity(vect1, vect2, "
    euclidian inverse"), 6)

# We define an appropriate full label writer : it displays the cannonical title, the year
    and the rating of the corresponding movie
clusters.setFullLabelWriter(lambda vectID: (u"%s (Year : %i Rating : %i)" % (movies[vectID][
    "canonical title"], movies[vectID]["year"], movies[vectID]["rating"])).encode("utf-8"))

# We define an appropriate thumbnail accessor : it displays the corresponding movie's
    thumbnail
clusters.setThumbnailAccessor(interface.getThumbnail)

# We resize the image to the appropriate size, we draw the diagram and we save the image
        im = im.resize(clusters.boxDimensions())
clusters.drawDiagram(im)
im.save("k-means_eucl.jpg")
```

3 Results

Both kinds of clustering and for all the three similarity measures have been tested on the 50 top rated movies. The results are in the files named according to the format *kind_of_clustering-similarity_measure.jpg* in the *Results* folder.

After performing a hierarchical clustering, we find the major subdivisions that appear and it can give an idea on how many clusters to use with the k-mean clustering. Here, we chose 6 clusters.

Although it is obvious the hierarchical clustering with the inverse Euclidian similarity is a failure, we can also see this similarity measure isn't satisfactory for the k-means clustering : the clusters sizes have no coherence in comparison to the other results. This is explainable as the inverse Euclidian measure quickly reaches extremely low values, when two vectors differ, thus the computational precision is low.

For the other results, it is difficult to evaluate the quality as it is very subjective. One criterium could be to check wether films episodes are in the same clusters. Fo that criterium, the hierarchical clustering with a Pearson similarity is very good while the k-means clustering with the same similarity is not that good.

4 Conclusion

These clustering method are very flexible and therefore can give very accurate results in specific cases. However, in general cases such as this one, it is not easy to appriciate the quality of the results. Moreover, when one criterium is satisfied, like for instance in the hierarchical clustering with a Pearson similarity, there can still be strange aspects : like associating “Amélie Poulain” with “Psycho”...