Computational Machine Learning, Fall 2015
Homework 4: k-means, bag-of-features

Due: Friday, November 13th, 2015, at noon (submit via email)
Preparation:

- install the software packages and material related to the Scikit-Learn lecture available here `https://github.com/amueller/nyu_ml_lectures`
- read Chapter 22 in [2] (Sec. 22.1-22.2) about the k-means algorithm

Instructions: submit the link to your Jupyter notebook.

1 Prologue

The purpose of the homeworks is to provide guidance for you to implement *yourself* useful machine learning algorithms, and use the real-world data related to your project as a playground to use your own codes and observe their performance.

In this homework, you will first implement the k-means algorithm, as well as a an accelerated and more stable version known as *k-means++*. Then, you will implement the bag-of-features representation and use it on the real-world data related to your project. All the support Python code needed was presented in the Scikit-Learn lectures. Take advantage of this assignment to make progress on your project write-up.

It is essential that you play by the rules and implement *your own codes*, genuinely and pouring your heart into it. *Should you have any question on this homework assignment, please feel free to post it on Piazza.*

2 k-means

In this section, you will focus on the so-called "k-means++" algorithm, described in [1].

2.1 k-means: a first version

First of all, read carefully Sec. 22.1-22.2 in [2], dedicated to the k-means algorithm, as well as:

`http://scikit-learn.org/stable/modules/clustering.html#k-means`

Then, write a function `mykmeans` that implements the k-means algorithm described in Sec. 22.2.1 and takes as input:

- the set of data-points $p_1, \ldots, p_m$), where $p_i \in \mathbf{R}^d$ for all $i = 1, \ldots, m$
- the maximum number of clusters k
- the maximum number of iterations `max_iter`, from now set to 50

The centroids (or cluster centers) are initialized randomly. The "repeat until convergence" while-loop is replaced by a for-loop with the maximum number of iterations `max_iter`.

Set the seed of the random number generator to a particular value, corresponding to a date of your choice. For instance, run `random.seed(11062015)` if the date is Nov. 6th 2015. Of course, do not use that date, just pick a date that is special to you.

Load the Iris dataset, and select the data-points corresponding to a particular class/category. Run `mykmeans` on this dataset you just created, with k the number of clusters (number of centroids) set to $k = 3$.

Plot the data-points and the obtained centroids, using different colors for each cluster. Play with the number of clusters, vary k from 1 to a large number. What happens when k is large? What happens when k is small? Is it easy for you to draw conclusions from the different runs of your algorithm on this fixed dataset? What's problematic with your current algorithm?

2.2 k-means: a better version

The current function `mykmeans` starts from a particular set of centroids, drawn randomly *once* before the algorithm runs. Update your function with an outer-loop, so that you run the previous algorithm multiple times, yet starting from different random draws of the initial centroids. The updated algorithm `mykmeans-multi` now should output the centroids corresponding to the run which reached the smallest value of the k-means objective function, over all runs.

Interlude The k-means objective function is called the *distortion*. A purpose of clustering is to represent each data-point in a cluster by the corresponding centroid, often also called prototype. The further a datapoint is from the corresponding prototype, the more it is distorted when represented by the prototype. Thus, for a particular set of data-points and the corresponding centroids/prototypes output by k-means, the distortion measures how much the set data-points is distorted when represented by the centroids/prototypes.

Run again the experiments on your datasets with `mykmeans-multi`, setting the number of runs to $r = 100$. Note that the number of runs should now be an input to your algorithm. What do your observe? How is this new version preferable to the first version?

Plot the curve of distortion versus the number of iterations, for the run that was picked after the multiple runs. The iteration counter is the one involved in the for-loop corresponding to the while-loop "repeat until convergence" in [2]. Display the visualisation of the clustering along the iterations, using different colors for each cluster. Highlight the trajectory of the centroids along the iterations, by joining the corresponding points with a line.

At this point, you have an implementation of k-means `mykmeans-multi`, which works fine and yields nice results. Yet, having to run the k-means algorithm multiple somehow lacks elegance. A more elegant solution, called *k-means++*, was proposed.

2.3 k-means++

Read Sec.1-2 of the article [1]. Write a function `mykmeans++` that implements the k-means++ algorithm. The function `mykmeans++` wraps the previous function `mykmeans`.

Run again the experiments on your datasets with `mykmeans-multi`. Vary the parameters k, `max_iter`. Plot the curve of distortion versus the number of iterations, for the initialization obtained at the end of step 1.c in the algorithm described in Sec. 2.2 of [1]. Display the visualisation of the clustering along the iterations, using different colors for each cluster. Highlight the trajectory of the centroids along the iterations, by joining the corresponding points with a line.

What do your observe? How is this new version preferable to the previous versions?

At this point, you have your own, state-of-the-art, implementation of k-means: `mykmeans++`. So far, we only focused on the *clustering quality*. The *clustering speed* was not the focus.

2.4 Reading Scikit-Learn

Compare the running time of `mykmeans++` to the Scikit-Learn's `sklearn.cluster.KMeans`, for different values of the number of clusters and different size of of datasets. If you are running your experiments on a cluster or HPC, set the number of jobs n_jobs to $n_jobs = 1$, for a fair comparison.

Pull the source code of `sklearn.cluster.KMeans`, and read it carefully. Try to parse what's going on in that code. The code includes several programming tricks to improve the speed the algorithm. Pick at least one trick, focus on it, understand what they do, and use a similar trick for your own function `mykmeans++`.

Explain in words the trick you used and what Python property (data structure, code optimization, etc.) it is based on, and write it down in Latex (1-2 pages). Write the kind of documentation you wish you had when reading code in general. Run again the comparison of your improved code with Scikit-Learn's `sklearn.cluster.KMeans`. Is the difference significant? In what terms? Quality? Speed?

3 Bag-of-features

In this section, you will focus on the bag-of-features representation.

3.1 Bag-of-words: the original

First of all, do the Scikit-Learn tutorial on text classification:

`http://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html`

Use a k-nearest neighbors classifier `neighbors.KNeighborsClassifier`, in place of Naive-Bayes; see Sec. "Training classifier" of the tutorial. Do Exercise 2 "Sentiment analysis on movie reviews". Feel free to go beyond the tutorial instructions, play with the parameters, and report your experimental results.

3.2 Bag-of-features

At this point, you have worked and experimented with the bag-of-words feature representation of text. The bag-of-feature representation is the extension of this approach beyond text data.

Overview The bag-of-words feature representation works as follows. A text is a "bag of words", assuming everything else than words has been discarded beforehand, using text pre-processing, tokenizing and filtering. A text is then represented with a feature vector. Each entry of the feature vector corresponds to a word (or word category) frequency, to which a normalizing transformation is applied such as tf-idf.

The bag-of-features representation extends this as a metaphor for non-textual data. Let us use audio signals as an illustration. The case of images was extensively covered during the course. An audio signal is viewed as "bag of local features". A local feature is a vector encoding the signal properties within a temporal window of the original audio signal. A temporal window a portion of the original audio signal that starts and ends a particular time-stamps. Local features could be FFT-type features that compute characteristics in the frequency domain, cepstral features that are appropriate for speech processing, etc. [1]

In order to build a counterpart of the bag-of-words feature representation, one first need to define a vocabulary, that is a set of "words" is when dealing with non-textual data. The idea is quite simple yet powerful. Take a "bunch" of local feature vectors extracted from real-world data, perform a clustering on all these local feature vectors, get the centroids. The obtained centroids define a "vocabulary"; each centroid defines a "word". In order to avoid over-fitting downstream the pipeline, the bunch of local feature vectors should be extracted from real-world data that is related yet distinct for the data further used for the classification/retrieval task at hand. For instance, if one works on music retrieval on a collection of a specific dataset of jazz recordings, one should use a different dataset of music recordings (preferably, jazzy) to build the vocabulary.

We now have a kind of vocabulary of "words" for audio signals. Note that the vocabulary is specific to the local features that were used. If one changes the type of local features, then another vocabulary has to be learned. Each "word" is a cluster centroid. Now, consider a audio signal of the dataset for the task at hand. Break this audio signal into a bunch of (overlapping) temporal windows. For each temporal window, extract a local feature vector. The audio signal is therefore turned into a bunch of local feature vectors. For each local feature vector, compute the cluster assignment. Compute the occurrences for each "word" over the bunch of local feature vectors, and build the corresponding histogram of frequencies. Apply a power-normalization or tf-idf, and you have obtained the bag-of-features representation of your audio signal.

Building the bag-of-features In this section, you will focus on the data related to your project. Depending on your project, *signal* means either text, audio signal, image, etc. In order to instantiate this section to your particular case, please use the personalised glossary I am sending you by email. If you are missing information, please immediately reach out to me through piazza.

Write a function `learnvocabulary` that, for a bunch of signals, extracts local features and learn a vocabulary using your own k-means algorithm, implemented in the previous section. If relevant, the function takes as input the length of the temporal/spatial window, the number of local features, the type of local characteristics (FFT, SIFT, etc.), the number of clusters, the max. number of iterations.

Write a function `getbof` that, given a vocabulary, computes the corresponding bag-of-features (BOF) feature vector for any signal. The function takes as input the vocabulary, and if relevant, the function takes as input the length of the temporal/spatial window, the number of local features, the type of local characteristics (FFT, SIFT, etc.).

[1] Local features are usually implemented in specialized toolboxes. Details are domain and application-dependent, hence beyond the scope of this course.

At this point, you have your own state-of-the-art, implementation of bag-of-features. Let's see how you can improve it.

3.3 Reading Scikit-Learn

Pull the source codes of involved for computing bag-of-words representation, and read them carefully. Try to parse what's going on in that code. The code includes several programming tricks to improve the speed the algorithm. Pick at least one trick, focus on it, understand what they do, and use a similar trick for your own functions.

Explain in words the trick you used and what Python property (data structure, code optimization, etc.) it is based on, and write it down in Latex (1-2 pages). Write the kind of documentation you wish you had when reading code in general. Run again the comparison of your improved code with the previous version. Is the difference significant? In what terms? Quality? Speed?

4 Project

The previous sections allowed you to write you own codes to compute a bag-of-features representation for the data involved in your project. Equipped with these codes and Scikit-Learn's k-nearest neighbors classifier `neighbors.KNeighborsClassifier`, you can now implement a first approach for your project.

Launch experiments for various settings of the parameters (size of windows, number of clusters, type of local characteristics, etc.), plot the experimental results. Perform a grid-search on the parameters to find near-optimal settings. Report and plot the experimental results.

Document and write down your experiments: settings, results, plots, etc. Should you have any question on this homework assignment, please feel free to post it on Piazza.

5 Feedback

1. Approximately how long did it take to complete this assignment?
2. Is there a concept you feel you still don't quite grasp? Detail what remains unclear.
3. Is there a terminology/concept that is recurringly used during the course, but you still have no idea what it means?
4. Is there a concept you would love to see covered during the course? (deep learning is not eligible; you can just take my course next semester)

References

[1] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 1027–1035, 2007. 2, 2.3

[2] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014. (document), 2.1, 2.2