COMPX521-19A Assignment 1

Using k-means clustering to learn representations for image classification

Due date: Monday, 8 April, 9am

The k-means clustering algorithm, fast and relatively simple to implement, is one of the oldest algorithms in machine learning. It is surprisingly useful and widely used in practice, e.g., for feature extraction from images: once features have been extracted by clustering image patches taken from a collection of training images, they can be used to transform these images into a representation that is suitable for standard classification algorithms (Coates & Ng, 2012). This transformation process is closely related to what happens in convolutional neural networks—widely used for deep learning.

The basic method for extracting features from images using k-means is simple: split the training images into equal-size "patches", represent each patch as a vector of numbers (consisting, for example, of RBG values: first the intensity values from the red channel, then those from the green one, and finally, the ones from the blue channel), and cluster these vectors using k-means. The final cluster centres, which can be viewed as representatives of the collection of image patches, are the extracted features.

Once features have been extracted, we can transform any given image by using each feature as a "filter" that is run across the image: we calculate the cross-correlation between the filter and subregions of the image as we slide the filter across it, which yields a single number for each subregion, and push this number through a non-linear ("activation") function. Assembling these numbers appropriately yields a a "feature map"—a new "image" showing how well the corresponding feature aligns with each subregion of the input image.

Note that there will be one feature map (corresponding to a "channel") for each feature, i.e., cluster centre. The set of filters producing the different feature maps is called a "filter bank". Each feature map can be represented as a high-dimensional vector, just like the intensity values from a colour channel in the original input image, and we can concatenate the vectors for all feature maps to make a very high-dimensional vector that we can use as a training instance (also called a "feature vector") for a standard classification algorithm. We simply append the class label of the input image at the end of this vector.

In practice, we need to reduce the vector's dimensionality to make this process workable. There are two mechanisms for doing this: 1) when a filter is run across an image, we can specify a step size—also called "stride"—so that the size of the resulting feature map is reduced, and 2) after the feature maps have been created, we can reduce their size by splitting them into regions and pooling values from each region—for example, we can simply average all the numeric values in each region.

It turns out that applying the above filtering process to raw image data does not work that well; to make it work better, the input values from each raw image are normalised and decorrelated using a process called "whitening".

The paper (Coates & Ng, 2012) describes all this in detail. The pseudo code for the preprocessing steps and the clustering algorithm is given on page 6 of this paper. Note that we apply a variant of k-means called "spherical" k-means, which ensures that the vectors representing each cluster centre all have length 1. Section 4 of the paper describes how to process images using the above striding and pooling process based on a filter bank—the set of centroids created by k-means, which is also called a "dictionary" (each cluster centre aka feature is viewed as a code word in a dictionary).

Your task in this assignment is to implement the method from (Coates & Ng, 2012) as a WEKA pre-processing tool (aka WEKA "filter") (Frank, Hall & Witten, 2016), but you do not need to implement the deep learning method in Section 5 of the paper. The filter must read a dataset with a string attribute containing the file names of the images to process and one other attribute, the (nominal) class attribute. The output of the filter needs to be the processed form of the images, suitable for other WEKA learning algorithms.

Some starter code for the new class, called `KMeansImageFilter`, is given at the course website on Moodle. To complete this starter code, you will need to add k-means and the generation of feature vectors. Note that you will need to implement spherical k-means from scratch: WEKA's `SimpleKMeans` is not spherical and much too slow. Use the MTJ library for Java to do the heavy-duty matrix algebra required for the implementation. It is included in WEKA and used by schemes such as `PrincipalComponents`. Three .jar files are part of MTJ: `mtj.jar`, `arpack_combined_all.jar`, and `core.jar`. All three must be included in your project's classpath/library files to use MTJ. These .jar files are actually inside in the `weka.jar` file that contains the WEKA software. You can extract them using the jar utility that comes with Java, i.e., a command like `jar -xf weka.jar`. Information on how to set up a project in the IntelliJ IDE for this assignment is given in Appendix A.

To make MTJ use native matrix algebra implementations, for greater speed, use the WEKA package manager to install the appropriate netlibNative* WEKA package for your system. (For even better speed on Linux and Windows, compile a native library such as OpenBLAS for your specific system and set up netlib-Java to use it.[1] On Macs, the default system matrix algebra package, "veclib", is already optimised, and the netilibNativeOSX WEKA package will use it automatically.) Note that you may have to run your code outside IntelliJ so that WEKA is actually able to use these native libraries.

Make sure your code is well documented. `KMeansImageFilter` implements command-line option handling so that it can be run from the command-line, as

[1] Instructions are at https://github.com/fommil/netlib-Java

well as getter and setter methods for the parameters so that it can be used from WEKA's GUIs. Parameters are: `size`, specifying the width and height of the image patches to extract for clustering, `numPatches`, the number of patches to extract from each image for clustering, `K`, the number of cluster centres, `stride`, the stride for the application of the filters when feature vectors are computed, `pool`, the width and height of the subregions used for pooling in this process, and `S`, the seed for the random number generator used in the program.

An important implementation note is that to achieve reasonable runtime when filters are applied to compute feature values for an image that is being processed, you need to make sure you multiply the entire filter matrix D with a particular image patch in one go; do not apply each feature (i.e., cluster centre) individually to an image patch!

Another note: in the paper, the authors state that 10 iterations of spherical k-means are usually enough. However, given that running spherical k-means does not take that long on the data we will use, I recommend to just keep performing iterations until the sum of squared errors has converged. There is a method in the starter code that you can use to compute the sum of squared errors efficiently.

The first subsection in Appendix A.1 of the textbook has succinct information on basic concepts in matrix algebra (transpose of a vector/matrix, matrix product, and Euclidean norm are the concepts relevant for the assignment, don't worry about the tensor product and the Hadamard product). Note that a vector can be viewed as a matrix with a single column. Appendix B below has additional info relevant to the assignment.

Once you have implemented the filter, your task is to run some experiments with it in conjunction with WEKA's `FilteredClassifier` and an appropriate base learner, on some image classifications datasets. The datasets are available on the Linux file system at `/home/ml/521/assignment1` (and also via `https://www.cs.waikato.ac.nz/ml/521/assignment1/`). You can assume that all images are the same size and square.

The feature extraction process has some parameters. For the patch size, the paper recommends 6 or 8; for the pool size, it recommends 2 or 3. Just use 10 for the ϵ_{norm} parameter and 0.1 for the ϵ_{zca} parameter. You can hardcode those. Also, there is the α parameter for the activation function `f()` that is used in the feature calculation. Just leave that (hardcoded) at 0 for the experiments.

For the MNIST[2] dataset (with 28x28 pixel images), the following values for the patch size, stride, and pool size are reasonable:

```
patch size = 8, stride = 4, pool size = 2
```

With 1000 clusters, this should give 9,000 features. (The assumption is that the pools do not overlap and that they exhaust the input completely: we only have a stride parameter for the application of the filters to the image before

[2]http://yann.lecun.com/exdb/mnist/

pooling; we do not have a stride parameter for pooling.) Use the same set-up for the Fashion-MNIST[3] dataset.

For CIFAR-10[4] (with 32x32 pixel images), the following values for the patch size, stride, and pool size are reasonable:

```
patch size = 8, stride = 3, pool size = 3
```

With 1000 clusters, this should again give 9,000 features. Use the same set-up for the SVHN[5] dataset.

Use `functions.SGD` as the base classifier in the `FilteredClassifier`. An example command-line for the MNIST problem, assuming the default settings in `KMeansImageFilte` are appropriate for MNIST, would be:

```
java -Xmx6g weka.Run .FilteredClassifier -t training.arff -T testing.arff -F .KMeansImageFilter W .MultiClassClassifier -- \
                                                                        -M 3 -W .SGD -- -N -M -F 0 -L 0.0001 -E 100=
```

Once the experiments are done, write a report on your findings. Please submit your report as a PDF file. Make a `.zip` or `.tar.gz` archive with your code (sources and compiled classes) and the report and submit it on Moodle. Once extracted, it must be possible to run your implementation by adding the top level of the extracted directory structure to the Java `CLASSPATH` (assuming the main WEKA 3.8 `weka.jar` file is already in the `CLASSPATH`).

Grades will be based on the quality of the report and the program. The report should be approximately three pages long in standard LaTeX report format. It should contain an introduction, a section explaining the algorithm that was implemented, a section with results and a corresponding discussion, and some conclusions. Relevant references to the literature should also be included (e.g., the paper describing the algorithm, a reference for WEKA, etc.). Results should be presented as tables or graphs when appropriate, with corresponding captions and references in the text that discusses them. The program should be readable and correct.

Make sure you start the assignment early. Debugging machine learning algorithms and running experiments takes a lot of time! There will be no extensions, except for documented medical reasons.

References

Coates, A. & Ng, A. Y. (2012). Learning feature representations with k-means. In *Neural networks: Tricks of the trade* (pp. 561–580). Springer.

Frank, E., Hall, M. A. & Witten, I. H. (2016). The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques", Morgan Kaufmann, Fourth Edition, 2016.

[3]https://github.com/zalandoresearch/fashion-mnist
[4]https://www.cs.toronto.edu/ kriz/cifar.html
[5]http://ufldl.stanford.edu/housenumbers/

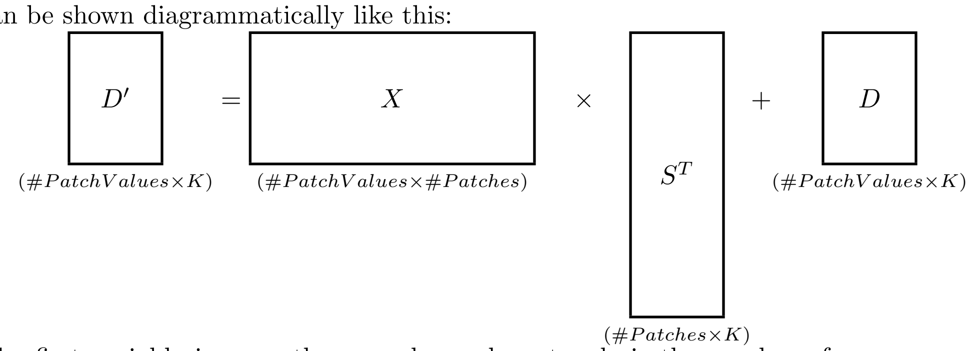
Appendix A: Setting Things up in IntelliJ

Below are step-by-step instructions for setting up an IntelliJ project for Assignment 1. The instructions are based on IntelliJ community edition for the Mac. They should also work on Windows or Linux. The `weka.jar` and `weka-src.jar` files are simply included as libraries in the project. The three MTJ .jar files also need to be included as libraries.

1. Start IntelliJ and select "Create New Project" or "File → New → Project ...".
2. Click "Next" and then select "Next".
3. Enter "MLAssignment1" as the project name and click "Finish".
4. Select "Project Structure" from the "File" menu and select "Libraries".
5. Click on "+" and choose "Java".
6. Navigate to the `weka.jar` file from the WEKA distribution you have downloaded, select it, click "Open" and then "OK".
7. Repeat steps 5-6 for `mtj.jar`, `arpack_combined_all.jar`, and `core.jar`, which you have extracted from weka.jar using the jar utility.
8. Click on "+" at the bottom of the panel on the right.
9. Navigate to the `weka-src.jar` file from the WEKA distribution and select it.
10. Click "Open", then untick "src/test/java" and select "OK".
11. Select "OK".
12. Right-click on "src" under "MLAssignment1" and select "New" → "File".
13. Enter "weka/filters/unsupervised/attribute/KMeansImageFilter.java" and click "OK".
14. Copy and paste the starter code downloaded from the course webpage in Moodle.
15. Select "Edit Configurations..." from the "Run" menu.
16. Add a new configuration for "KMeansImageFilter" (select "+" and "Application") specifying "weka.filters.unsupervised.attribute.KMeansImageFilter" as the "Main class" and "-h" as the "Program Arguments"; click "Apply".
17. Select "Run 'KMeansImageFilter'" from the "Run" menu.
18. Check that output stating the available command-line options for the filter is given in the IntelliJ terminal. That means you are set!

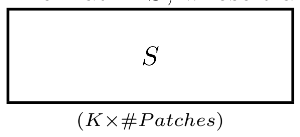
Appendix B: The matrices in spherical K-means

The update rule for the cluster centroids/dictionary D in spherical K-means can be shown diagrammatically like this:


D'
$=$
X
$\times$
S^T
$+$
D
$(\#PatchValues \times K)$
$(\#PatchValues \times \#Patches)$
$(\#Patches \times K)$
$(\#PatchValues \times K)$

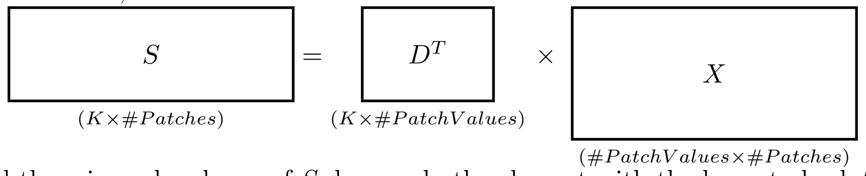
The first variable in parentheses under each rectangle is the number of rows of the corresponding matrix, the second variable corresponds to the number of columns. K is the number of centroids (i.e., the number of dictionary elements/filters to be learned), and each column of D is a centroid. X is the whitened data matrix, with one patch/instance per column. The variable $\#PatchValues$ holds the number of intensity values per patch/centroid (three times the number of pixels in the patch/centroid, and the variable $\#Patches$ holds the total number of training patches/instances (including patches from all images).

The matrix S, whose transpose is S^T, has the format



Each column $s^{(i)}$ of this matrix S has exactly one non-zero element $s_j^{(i)}$, namely for the row j corresponding to the centroid $D^{(j)}$ that is most closely related to the image patch stored in column $x^{(i)}$ of X (as measured by the absolute value of the dot product between $D^{(j)}$ and $x^{(i)}$). The exact value of this non-zero element indicates how strongly the patch is aligned with the centroid. This value is the outcome of the dot product.

Note that you can calculate S very efficiently by using matrix multiplication instead of taking lots of individual dot products between column vectors in D and X. First, calculate S as


S
$=$
D^T
$\times$
X
$(K \times \#Patches)$
$(K \times \#PatchValues)$
$(\#PatchValues \times \#Patches)$

and then, in each column of S, keep only the element with the largest absolute value and set all other elements to 0.

PS: In the algorithm for spherical K-means, $\|p\|_2$ stands for the L_2 norm (aka Euclidean norm) of the vector p.