Deep Learning on Historical Manuscripts

This project investigates the application of deep learning algorithms for image analysis problems on historical manuscripts. Nowadays, large collections of manuscripts exist in libraries around the world. There has been much interest in their digitization for preservation reasons, because many of these manuscripts are degraded and very fragile. However, their content is largely unexplored by researchers and unknown to the interested public, as digitizing a manuscript does not mean that its content is easily exploitable. Digitization creates only a digital image of the manuscript, and an additional transcription step is needed so that the manuscript's content can be electronically retrieved and analyzed. Many methods have been proposed in the literature to automatically transcribe a manuscript. All of them are based on annotated/labeled data. In other words, to be able to learn to automatically transcribe a manuscript, we need to have (part of) the transcription already done.

The aim of this project is to develop deep learning methodologies to analyze historical manuscripts and to retrieve re-occuring patterns (characters, words, text lines, etc) in an unsupervised way, such that subsequent transcription of these manuscripts requires as little human assistance as possible. The goal is to apply large-scale, unsupervised deep learning algorithms in order to automatically learn features and patterns specifically tuned to text images. It is important to note here that the project does not address the problem of automatic text recognition, due to lack of training lables and transcription.

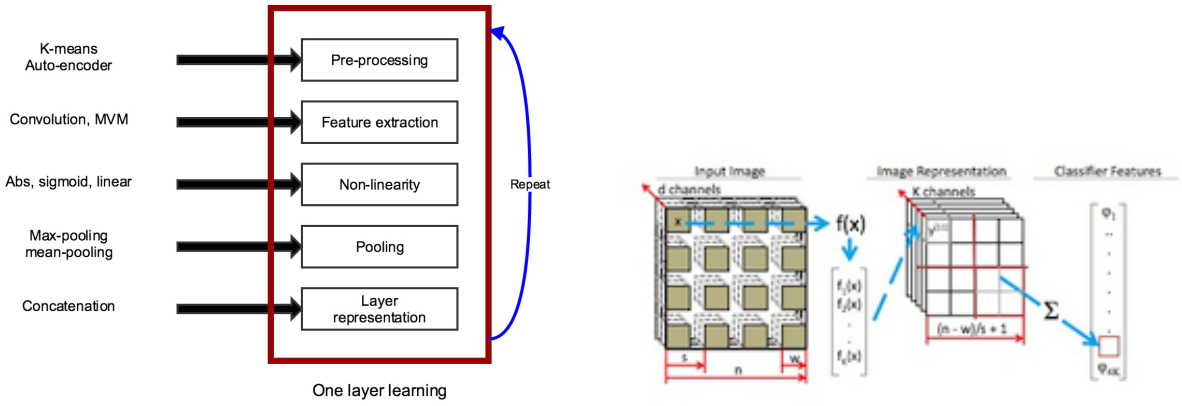
The project can be based on several deep learning algorithms. However, one very promising and general architecture is the one proposed in the paper "Emergence of Object-Selective Features in Unsupervised Feature Learning" [1]. Example diagrams of the deep learning architecture are shown in Figure 1. The above architecture contains simple algorithms that can be easily parallelized to large scales. The Deep Learning architecture to be implemented in this project will be based on it. A very suitable framework to implement these algorithms is the Apache Spark, which already contains some Linear Algebra and Machine Learning functionality.

One application of the proposed Machine Learning architecture, which is suitable for the unsupervised data at our disposal, is that of *Word Spotting*: The user gives as input to the algorithm an image of a specific word. The goal is to identify in the searched manuscripts instances of the same word. This way, the document can be indexed and searched online. A diagram of the word spotting example is shown in Figure 2. Some pre-processing steps have been already conducted that make the above problems easier to tackle. One example is the step of text line extraction. By restricting the processing on text lines, we make sure that there are no overlapping words in the image, a fact that makes the application of learning algorithms much more difficult.

1 Data

We have at our disposal a significant amount of handwritten high-res digitized pages (> 20000) in .tiff format from one Swiss writer, C.F. Ramuz. The images have been provided by the Bilbiothèque Cantonale et Universitaire de Lausanne (BCU) (`http://www.bcu-lausanne.ch/`) at the University of Lausanne (UNIL). The data rests for the moment on a local HDD, and it is about 300 GB. The images are property of the Ramuz Foundation, however, their usage is allowed inside an institutional environment for research purposes.

In Figure 3 we show some samples of the images available in the dataset. As mentioned before, we do not have at our disposal a diplomatic transciption of the data. However, we are in the process of creating some



(b) Taken from [2]

Figure 1: Examples of unsupervised deep learning architectures.

ground truth transcriptions with the collaboration of UNIL, which will be used exclusively for evaluation purposes. No training data will be used during the learning procedure of our algorithm.

2 Project Description

In this section we provide a detailed description of the project steps. The project can be divided into three important sub-projects:

1. Data extraction/processing.
2. Feature learning.
3. Experimental evaluation.

In the following, we give a detailed description of each of the above steps:

2.1 Data extraction/processing

The first part of the project is the data extraction from the original high-res .tiff images of handwritten text. For that purpose, we need to extract a large amount of patches from the image files. We divided the process into two main stages :

1. Segmenting the text lines for each page.
2. Extracting the patches for each text line.

For both of these steps, the first issue was to process image data from Spark. For this task, we implemented a serializable class `ImageData`, which automatically switches between a compressed binary representation of an image and its decompressed accessible version. This approach, which uses the encoding/decoding capabilities of OpenCV, allows us to load/save/shuffle our image data efficiently.

In order to segment the text lines for each page, we adapted the very good pipeline created by Andrea Mazzei (former member of the DHLAB). The original code in C++ for Windows was adapted to our case. We made it runnable on a UNIX system, callable from Java and we added some safety mechanisms to process such high resolution images. The pipeline (Fig. 4) is made of three segments : *cropping* (removing the scanned

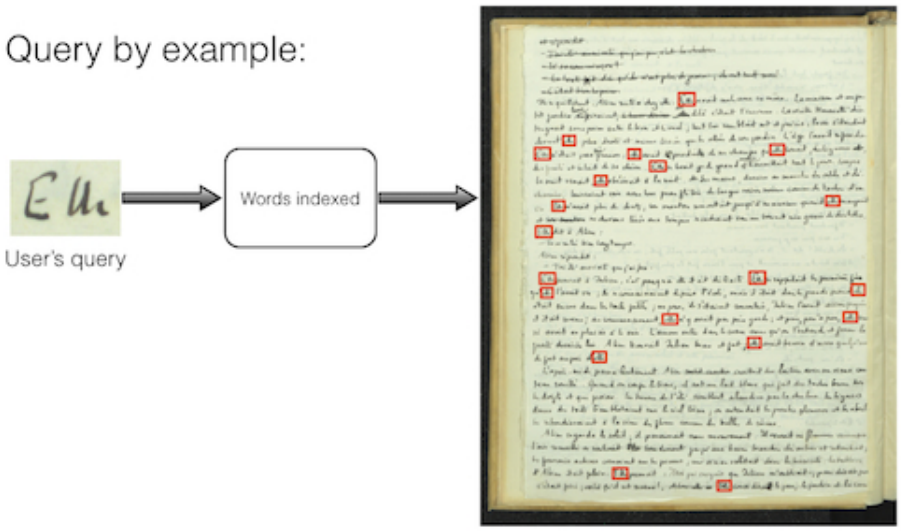


Figure 2: Word Spotting diagram.

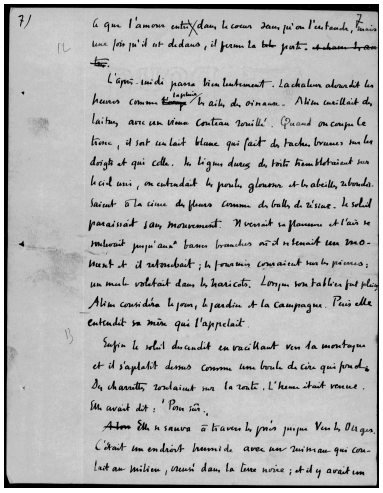
boundaries of the pages, altering the original size), *binarizing* (separating the ink from the background), and *detecting the text lines* (clustering the foreground in separated lines of text). We output the results in four different folders: the cropped images, the images that failed to be processed, the JSON files encoding the detected text lines, and a visual representation of the detection (see Fig. 5). Another technical hurdle of the project was to properly incorporate the native libraries (of OpenCV and our segmentation pipeline) on the Spark cluster. In order to avoid compilation from the user (which for OpenCV is not straightforward and no Java oriented compiled version is accessible directly), we created our own repository where maven downloads the corresponding jar and native library and then packs it all together in the same jar, allowing us to easily send it to Spark thereafter. Since manually sending the native libraries to nodes was not very practical, this ensures that each node has all the necessary code at the start of the process[1]. Schematic can be seen on Fig. 6.

The results of the image segmentation pipeline are saved in .json files. These files are input to the patch extraction process. We extract patches of a specific size from each text line. This way, we eliminate noisy patches that contain parts from two consecutive text lines. This is important, especially for the candidate patch extraction of the word spotting experiment. For training the algorithm, we extract patches of size 32×32 and 64×64. For the testing phase, the size of the patch will be the same as the size of the query word image. We have implemented three approaches for the patch extraction:

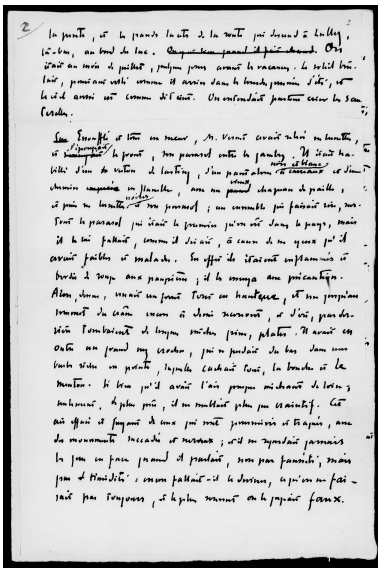
1. In the first approach, we extract patches from a rectangular bounding box that surrounds a text line. This approach can produce some redundant patches that fall outside the actual precise boundaries of the text lines. To eliminate those patches, and also the blank ones that fall between words, we discard patches whose standard deviation is below a threshold.

2. In the second approach, we use the complete boundary coordinates information for each text line. Hence, instead of scanning over the whole rectangular bounding box, we restrict our scan only to the relevant parts of the text line. This approach, as the first one, uses a sequential sliding window methodology with an horizontal step size equal to the patch size and a vertical step size being half of the patch's size.

[1]It must be noted that even then, Java cannot load native libraries from the jar directly. These need to be unpacked to a temporary folder before loading them.



(a) C.F. Ramuz, Aline, p. 7, 1905.



(b) C.F. Ramuz, Peintre Vaudois, p. 2, 1911.

Figure 3: Sample pages from the dataset.

3. In the third approach, we extract random patches from each text line, until we obtain a certain amount of patches per text line.

For the project, we used the second approach as it was the fastest of all with the lowest number of false-positive and false-negative patches.

The output of the patch extraction procedure is a dataset of vector tuples: The second vector contains the vectorized patch itself and the first vector consists of meta-data information for each patch in the following format: [page_id, line_id, x, y, w, h], where

1. page_id: ID of the page to which this patch belongs to
2. line_id: ID of the text line to which this patch belongs to
3. x: top-left x-coordinate of the extracted patch in the text line (used to locate the patch)
4. y: top-left y-coordinate of the extracted patch in the text line
5. w: width of the patch
6. h: height of the patch

In Figure 2.1 we show the complete diagram of the data extraction and processing pipeline.

2.2 Feature Learning

This part is the core of the deep learning architecture. As seen in Figure 1a, the chosen deep learning architecture can be broken into four main parts:

1. **Data pre-processing**. In this project we implemented the ZCA whitening algorithm, which projects the original data points onto a space where they become de-correlated. ZCA pre-processing is a necessary step for the learning algorithm to work well and learn useful filters.

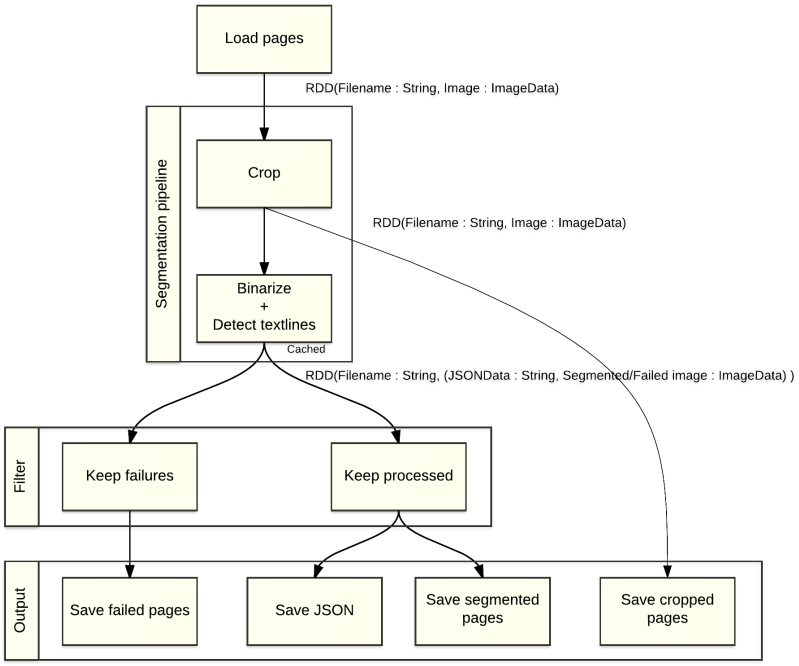


Figure 4: Segmentation flowchart.

2. **Feature learning with an un-supervised learning algorithm**. Two algorithms have been used: (1) K-means (implementation from Spark) and (2) Autoencoders (our distributed implementation).

3. **Feature extraction with Matrix-Vector multiplications (MVMs) and FFT-based convolution**. We use FFT-based convolutional feature extraction for the first layer and MVMs for the remaining layers. We have two implementations of FFT: (1) A naive one based on `http://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm#Pseudocode` and (2) an adaptation of optimized FFT code from `http://www.nayuki.io/res/free-small-fft-in-multiple-languages/Fft.java`. For the MVMs we use the wrapper Spark methods for the BLAS calls.

4. **Non-linear transformations**. After the feature extraction process, the new representations pass through some non-linear functions, such as the absolute value, or the sigmoid.

5. **Pooling operations**. We have implemented max pooling over non-overlapping regions of a specific size. By concatenating all these pooled values for each learned filter, we obtain the final data representations for the current layer.

The input parameters necessary for the different modules of the learning architecture are encoded into a protocol buffer. A simple example of a protocol buffer file is given below:

```
message ConfigPreprocess {
              required double eps_1 = 1;
              required double eps_2 = 2;
}

message ConfigKMeans {
       required int32 number_of_clusters = 1;
```

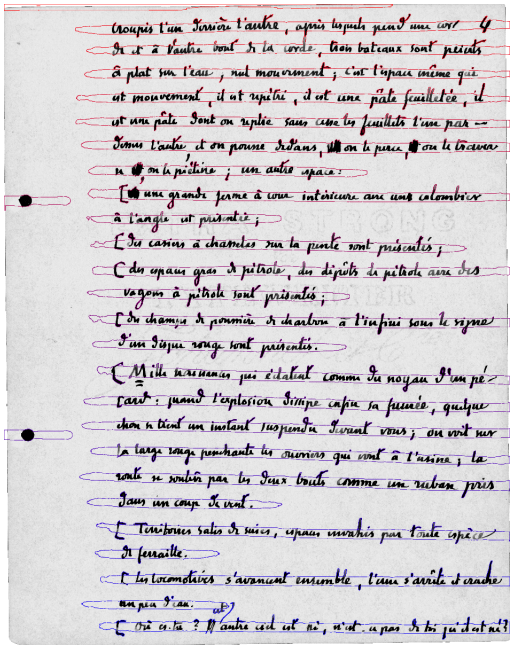


Figure 5: Example of segmented image.

```
        required int32 number_of_iterations = 2;
}

message ConfigAutoencoders {
        required int32 number_of_units = 1;
}

message ConfigFeatureExtractor {
  required int32 input_dim1 = 1;
  required int32 input_dim2 = 2;
  required int32 feature_dim1 = 3;
  required int32 feature_dim2 = 4;
  enum NonLinearity {
    NONE = 0;
        ABS = 1;
        SOFT = 2;
  }
  optional NonLinearity non_linearity = 5 [default = NONE];
  optional double soft_threshold = 6;
}

message ConfigPooler {
  required int32 pool_size = 1;
  enum PoolType {
    MAX = 0;
    AVERAGE = 1;
  }
  optional PoolType pool_type = 2 [default = MAX];
```

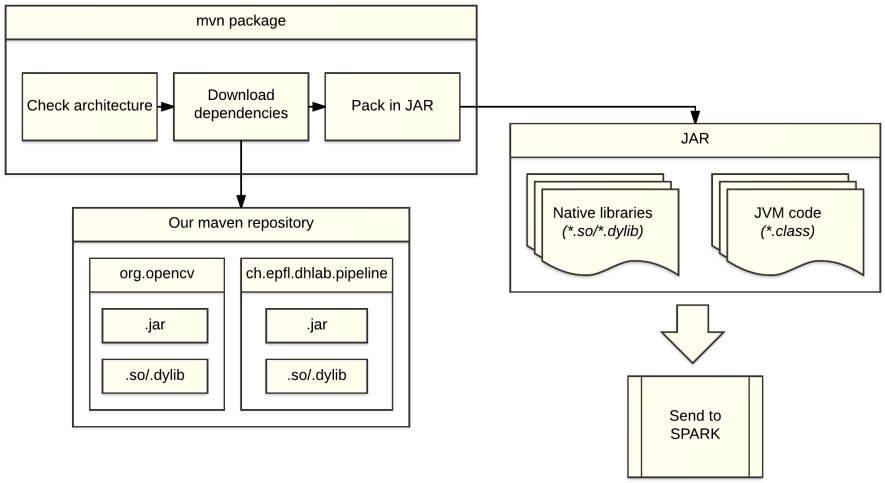


Figure 6: How we dealt with dependencies.

```
}

message ConfigBaseLayer {
  optional ConfigPreprocess config_preprocess = 1;
  optional ConfigKMeans config_kmeans = 2;
  optional ConfigAutoencoders config_autoencoders = 3;
  optional ConfigFeatureExtractor config_feature_extractor = 4;
  required ConfigPooler config_pooler = 5;
}

message ConfigManuscripts {
    repeated ConfigBaseLayer config_layer = 1;
}
```

In the above proto buffer configuration, we have created six messages. The messages `ConfigProcess`, `ConfigKMeans`, `ConfigAutoencoders`, `ConfigFeatureExtractor` and `ConfigPooler` contain the basic input parameters for all the steps of the learning algorithm. The message `ConfigBaseLayer` brings together all the sub-messages and represents the configuration of one learning layer of the system. Finally, the message `ConfigManuscripts` contains a list of configurations for each learning layer of the system. A realization of the above configuration with specific parameters is given in the following block of code:

```
config_layer {
  config_preprocess {
    eps_1 : 10
    eps_2 : 0.1
  }
  config_kmeans {
    number_of_clusters: 10
    number_of_iterations: 20
  }
  config_feature_extractor {
    input_dim1: 128
```

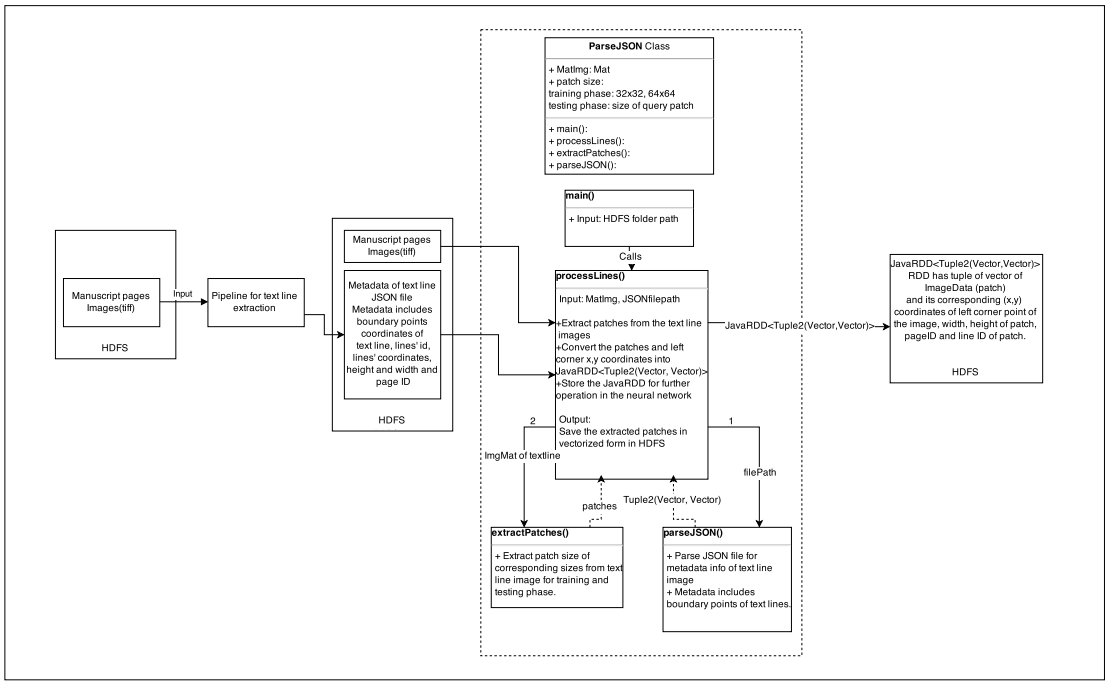


Figure 7: UML-type diagram of the data extraction and processing pipeline.

```
    input_dim2: 32
    feature_dim1: 32
    feature_dim2: 32
  }
  config_pooler {
    pool_size: 2
  }
}
}
```

In the above example, we have created a one-layer architecture, where we do a ZCA pre-processing with regularization parameters 10 and 0.1, we run K-means with 10 clusters and 20 iterations, we do an FFT-based convolutional feature extraction on patches of size 128×32 with filter sizes 32×32 and we perform pooling over 2×2 blocks.

A high-level flow of a two-layer architecture of our system is shown in Figure 2.2. Two datasets of small and large patches are input to a `Learner` and `Extractor` interface. After the pooling operation, the resulting data representations become input to the second layer of learning. The same steps are computed as in the first layer. After the final pooling operation, the final represnetations of the data are the output of the learning system.

A more detailed UML diagram of our design is shown in Figure 2.2. There is a main interface called `DeepLearningLayer`, which provides method definitions for the pre-processing, learning, feature extraction and pooling operations. The methods `train` and `test` combine the different steps into a sequential pipeline. Moreover, there are methods for saving and loading the trained models. The class `BaseLayer` implements this interface and calls the necessary methods of the learning procedure. The interfaces `PreProcessor, Learner, Extractor, Pooler` contain method definitions for the corresponding parts of the learning architecture. The classes `PreProcessZCA, KMeansLearner, AutoencoderLearner, MultiplyExtractor, FFTConvolutionExtractor, MaxPooler, MaxPoolerExtended` implement the corresponding interfaces. The

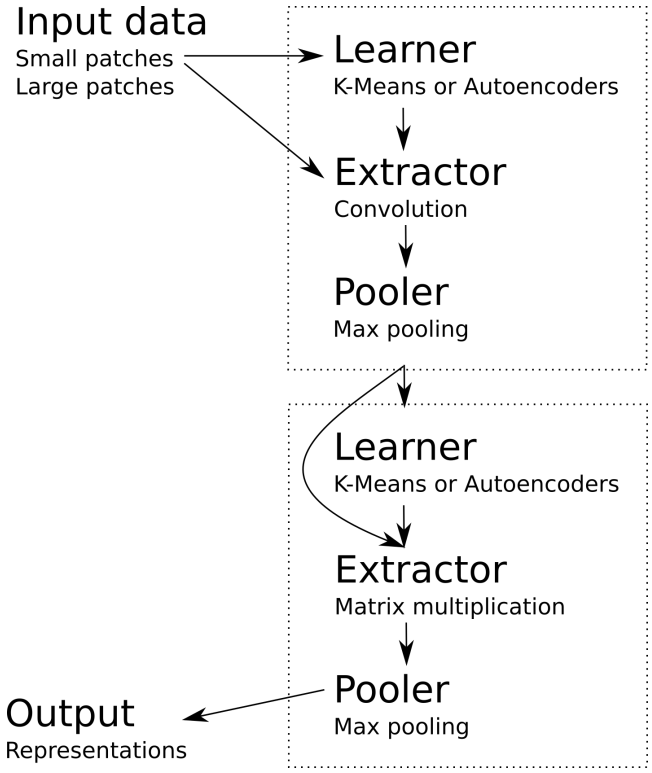


Figure 8: Flow diagram of the learning system.

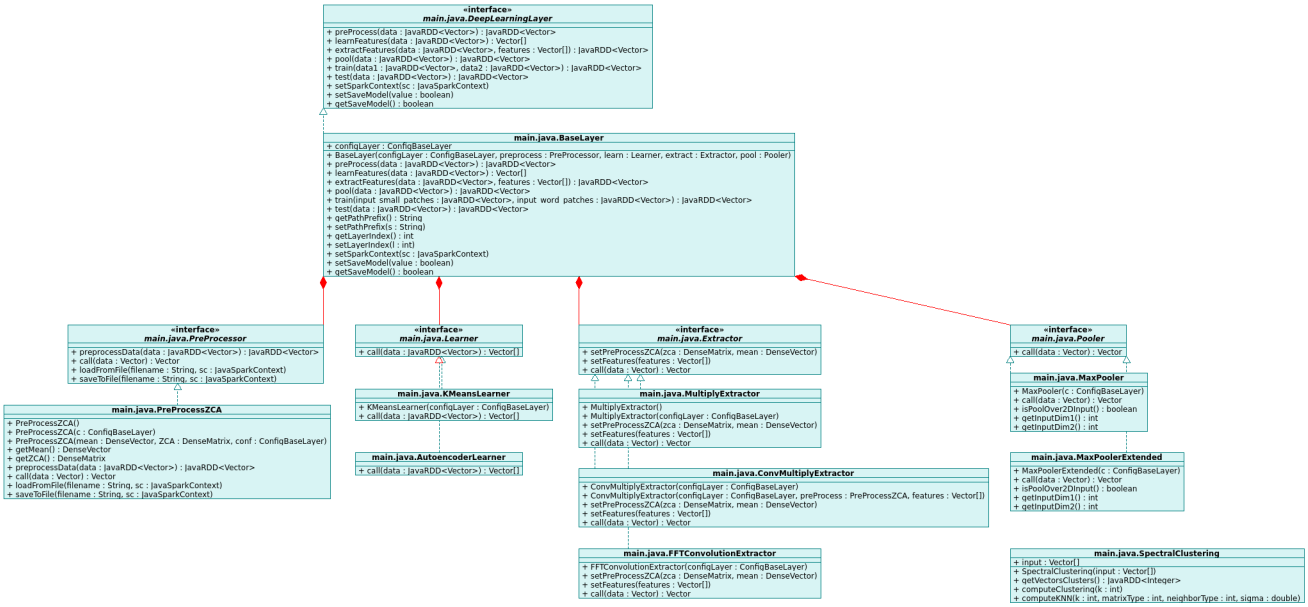


Figure 9: UML diagram of the learning architecture.

class `SpectralClustering` implements the spectral clustering method.

2.3 Experimental Evaluation

In this part we evaluate the results of our learning algorihtm. After the learning is done, we use our learned system for the problem of Word Spotting on several query images. Depending on the size of the input word, an approach is implemented that extracts candidate patches from the source text images. The query together with the candidate patches will pass through our algorithm to create new representations. The final representation of the query image will be compared with the representations of the candidate patches. The patches whose representations are the most similar to the query's representation will be the output of our experiment.

3 Team members and work assignments

The project consists of the following members:

1. Nikolaos Arvanitopoulos (Team leader)
2. Viviana Petrescu
3. Isinsu Katircioglu
4. Benoit Seguin
5. Radu-Christian Ionescu
6. Arun Balajee Vasudevan
7. Ashish Ranjan Jha
8. Arttu Voutilainen
9. Alexander Vostriakov

The work assignments for the first milestone have been decided in a hierarchical way as follows:

- One team consisting of Isinsu, Benoit, Arun and Ashish worked on the data extraction. More specifically:
  - Benoit was responsible for integrating OpenCV inside our Spark architecture to enable image processing functionality in the system. Furthermore, he integrated Andrea Mazzei's pipeline (binarization, text line extraction, word segmentation) into Spark and applied it on the available data. The output from this process was saved as .json files that describe the whole segmentation result.
  - Ashish and Isinsu were responsible for the data extraction procedure. The input to this module is the .json files which were created in the previous step. The output of this step are text (or object) files saved in HDFS, which contain the extracted patches to be used for the learning algorithm. This part also involved the extraction of candidate patches for the word spotting experiment.
  - Arun was responsible for building a meta-data system, which keeps track of necessary information for the patch extraction. Example meta-data include page id, patch location and size, etc.
- The other team consisting of Nikolaos, Viviana, Radu, Arttu and Alexander worked on the learning pipeline. More specifically:
  - Nikolaos implemented the ZCA pre-processing part, feature extraction based on MVMs and non-linear transformations.

- Arttu implemented the FFT-based convolutional feature extraction, based on a naive implementations and an adaptation of optimized code.
- Radu implemented the sparse auto-encoder as an alternative to K-means. Furthermore, he was responsible for running experiments on the cluster.
- Viviana implemented the max-pooling operation and example code for unit testing.
- Alexander implemented a version of Spectral Clustering, a clustering procedure based on graphs.

Arttu and Viviana decided on the used design for the implementation of the learning architecture. Furthermore, they provided a lot of support in debugging the system. Details of the classes implemented are shown in the UML diagram of Figure 2.2.

4 Experiments

References

[1] Adam Coates, Andrej Karpathy, and Andrew Y. Ng. Emergence of object-selective features in unsupervised feature learning. In P. Bartlett, F.c.n. Pereira, C.j.c. Burges, L. Bottou, and K.q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2690–2698, 2012.

[2] Adam Coates and AndrewY. Ng. Learning feature representations with k-means. In Grégoire Montavon, GenevièveB. Orr, and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade*, volume 7700 of *Lecture Notes in Computer Science*, pages 561–580. Springer Berlin Heidelberg, 2012.