Graduate School Form 30
Updated 1/15/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Hossein Karimy Dehkordy

Entitled
AUTOMATED IMAGE CLASSIFICATION VIA UNSUPERVISED FEATURE LEARNING BY K-MEANS

For the degree of Master of Science

Is approved by the final examining committee:

Dr. Murat Dundar
Chair

Dr. Fengguang Song

Dr. Yuni Xia

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Dr. Murat Dundar

Approved by: Dr.Shiaofen Fang 6/26/2015

Head of the Departmental Graduate Program Date

AUTOMATED IMAGE CLASSIFICATION

VIA UNSUPERVISED FEATURE LEARNING BY K-MEANS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Hossein Karimy Dehkordy

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 2015

Purdue University

Indianapolis, Indiana

For my parents, who always support me and to whom I owe all my life.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

Convolutional Neural Networks:	CNN
Feature Learning:	FL
Neural Networks:	NN
Open Multi-Processing:	OpenMP
Principal component analysis:	PCA
Support Vector Machine:	SVM

ABSTRACT

Karimy Dehkordy, Hossein. M.S., Purdue University, August 2015. Automated Image Classification via Unsupervised Feature Learning by K-means. Major Professor: Murat Dundar.

Research on image classification has grown rapidly in the field of machine learning. Many methods have already been implemented for image classification. Among all these methods, best results have been reported by neural network-based techniques. One of the most important steps in automated image classification is feature extraction. Feature extraction includes two parts: feature construction and feature selection. Many methods for feature extraction exist, but the best ones are related to deep-learning approaches such as network-in-network or deep convolutional network algorithms. Deep learning tries to focus on the level of abstraction and find higher levels of abstraction from the previous level by having multiple layers of hidden layers. The two main problems with using deep-learning approaches are the speed and the number of parameters that should be configured. Small changes or poor selection of parameters can alter the results completely or even make them worse. Tuning these parameters is usually impossible for normal users who do not have super computers because one should run the algorithm and try to tune the parameters according to the results obtained. Thus, this process can be very time consuming.

This thesis attempts to address the speed and configuration issues found with traditional deep-network approaches. Some of the traditional methods of unsupervised learning are used to build an automated image-classification approach that takes less time both to configure and to run.

CHAPTER 1. INTRODUCTION

1.1 Motivation

Nowadays, data are growing at an exponential rate, requiring us to face big issues related to processing and extracting information from data. One possibility is to increase computer performance and processing ability, thereby increasing output. Another is to design and develop new algorithms or approaches that can operate better on big data and analyze them faster with appropriate performance.

A well-known big-data domain is image classification. By growing the technology and using pictures and images from many fields, we might find a solution that can process and classify big data everywhere without the need for supercomputers. Many algorithms for image classification exist, but some are weak and have low accuracy. The rest need supercomputers to run, an option not available for everyday users with modest budgets. One way to address this is to use classic and powerful methods along with some tricks that could do the classification at the same level as the new methods. Reviewing the latest powerful methods in image classification, one can see that most of them use unsupervised feature learning, getting help from neural networks (NNs). From the old and powerful algorithms, I selected K-means as the best choice for unsupervised learning, a choice also adopted by several papers in the literature. This study explores the utility of K-means under different settings for automated image classification.

1.2 Types of Learning Algorithms

Currently available machine-learning algorithms can be split into four major categories: supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning.

1.2.1 Supervised Learning

When we have labeled data we can use supervised learning for the task of inference. In this method, each data point or instance includes two parts: “input,” which contains the feature variables, and “output” or “label,” which is the desired value for this data point. With this type of data organization, we can run training algorithms over the given input/output pairs to build a model that will predict the labels for future unlabeled test data. Figure 1 shows this process.

Supervised learning is divided into two main categories:

1. Classification deals with data sets that are partitioned into different categories or classes. In this type of learning the output has a finite set of values called labels or categories. The most popular classification methods are as follows:

    - Neural networks
    - Support vector machine (SVM)
    - Nearest neighbors (KNN)
    - Decision trees
    - Naïve Bayes classifier

2. Regression deals with continuous-valued data. The output value in this type of learning can be any real value. The most popular regression methods are as follows:

- Generalized linear models
- Linear regression
- Nonlinear regression

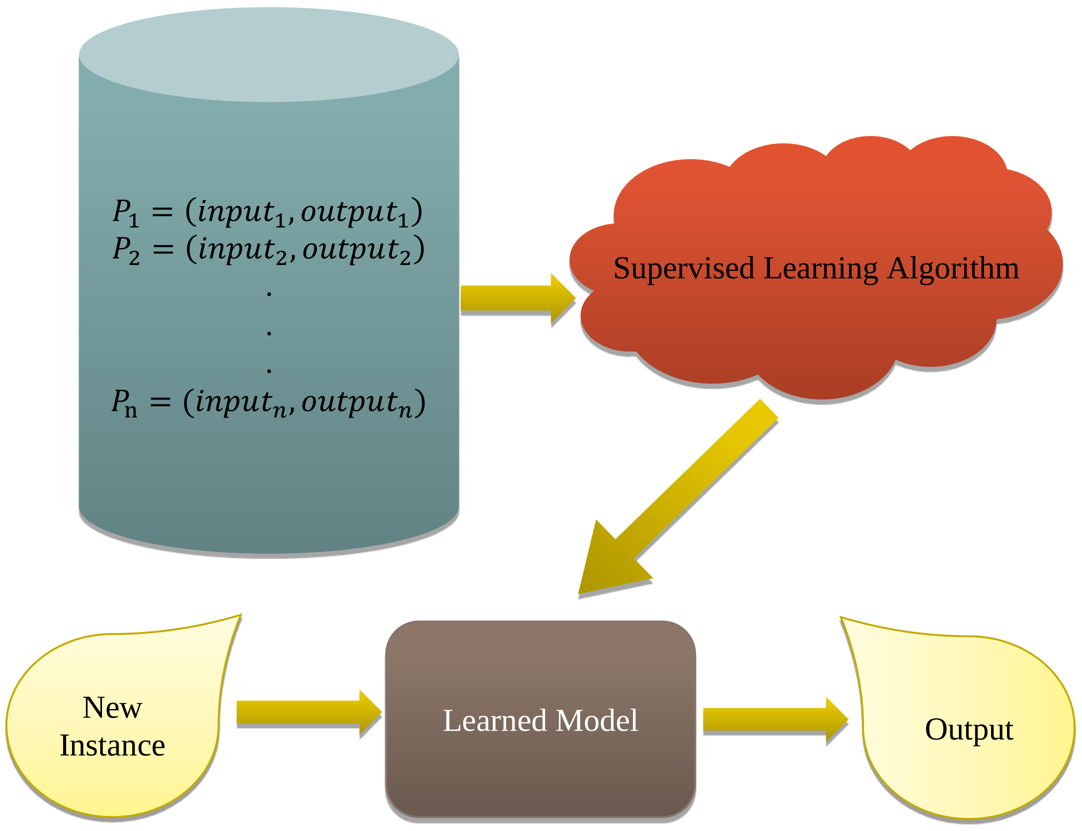

$P_1 = (input_1, output_1)$
$P_2 = (input_2, output_2)$
.
.
.
$P_n = (input_n, output_n)$
Supervised Learning Algorithm
New Instance
Learned Model
Output

Figure 1: Supervised learning model

1.2.2 Unsupervised Learning

When the given training data have no labels, we use unsupervised learning to make inference from the data. This method is based on statistics but also commonly used in data mining, whose main goal is to look for patterns in the given data and group them into different clusters. We can see how this process works in Figure 2. This can be achieved by different algorithms, with some of the most popular ones being:

- Neural networks
- K-means clustering
- Hierarchical clustering
- Gaussian mixture models
- Self-organizing maps
- Hidden Markov models

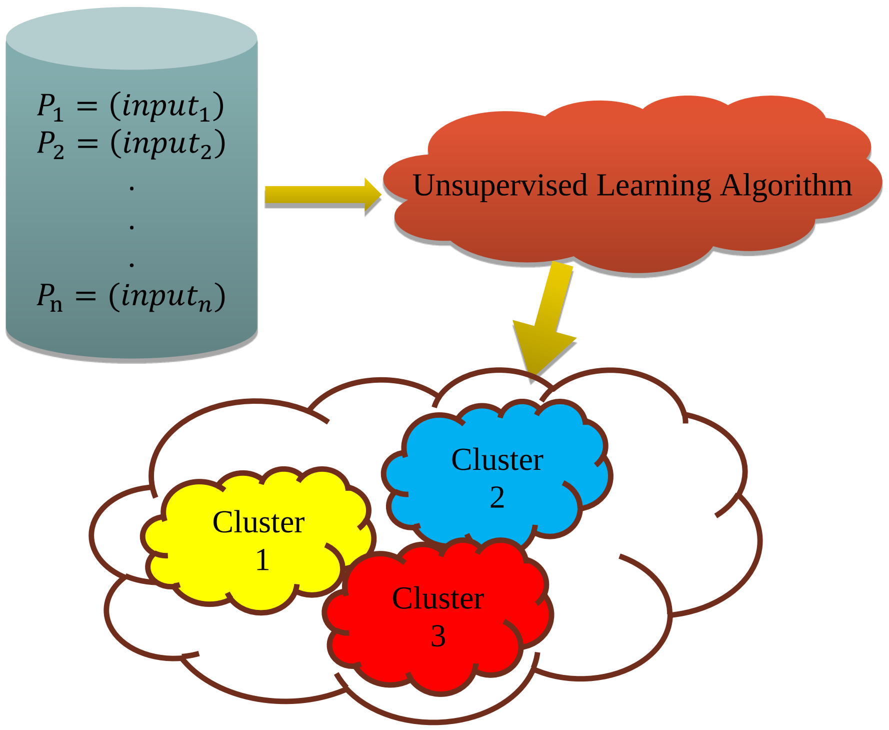


Figure 2: Unsupervised learning model

1.2.3 Semi-Supervised Learning

This method is employed when we have both labeled and unlabeled data, to develop a model that is better than models created using each type alone. In this method we use a large amount of unlabeled data to learn and build a classification model for a small set of labeled data. The process is showing in Figure 3. The major algorithms in this method are:

- Self-Training
- Generative Models
- Transductive Learning
- Graph-Based Algorithms

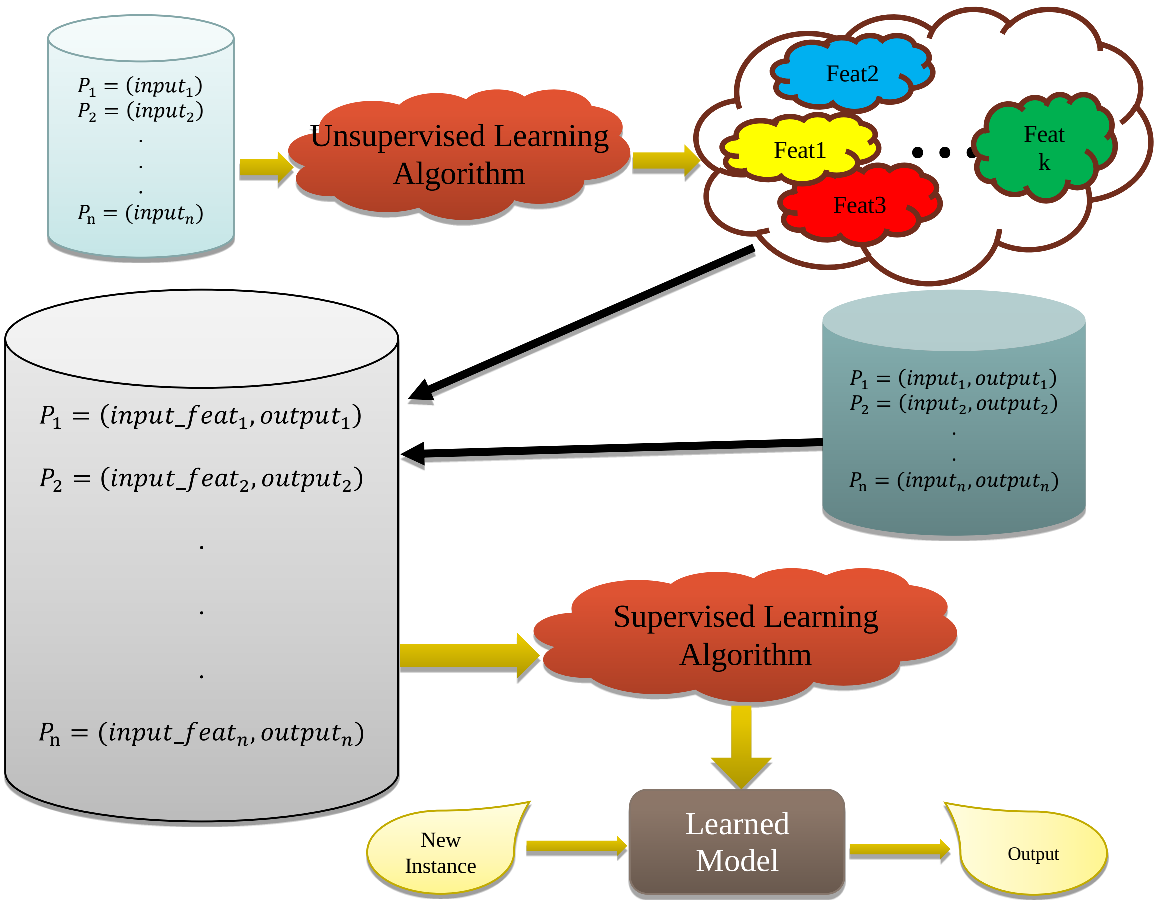

$P_1 = (input_1)$
$P_2 = (input_2)$
$P_n = (input_n)$
Unsupervised Learning Algorithm
Feat2
Feat1
Feat3
Feat k
$P_1 = (input\_feat_1, output_1)$
$P_2 = (input\_feat_2, output_2)$
$P_n = (input\_feat_n, output_n)$
$P_1 = (input_1, output_1)$
$P_2 = (input_2, output_2)$
$P_n = (input_n, output_n)$
Supervised Learning Algorithm
Learned Model
New Instance
Output

Figure 3: Semi-supervised learning model

1.2.4 Reinforcement Learning

In this method, the machine or software is allowed to automatically adapt and determine the best behavior or action respective to its environment. This method is used by most agent-based fields such as games, multi-agent systems, swarm intelligence, and genetic algorithms.

Usually the algorithm consists of “rules” and “actions” that depend on these rules; by analyzing the environment, the system tries to find the best way of doing actions to achieve the highest rewards. The agents in this model interact with the environment to get feedback and make decision for the next action.

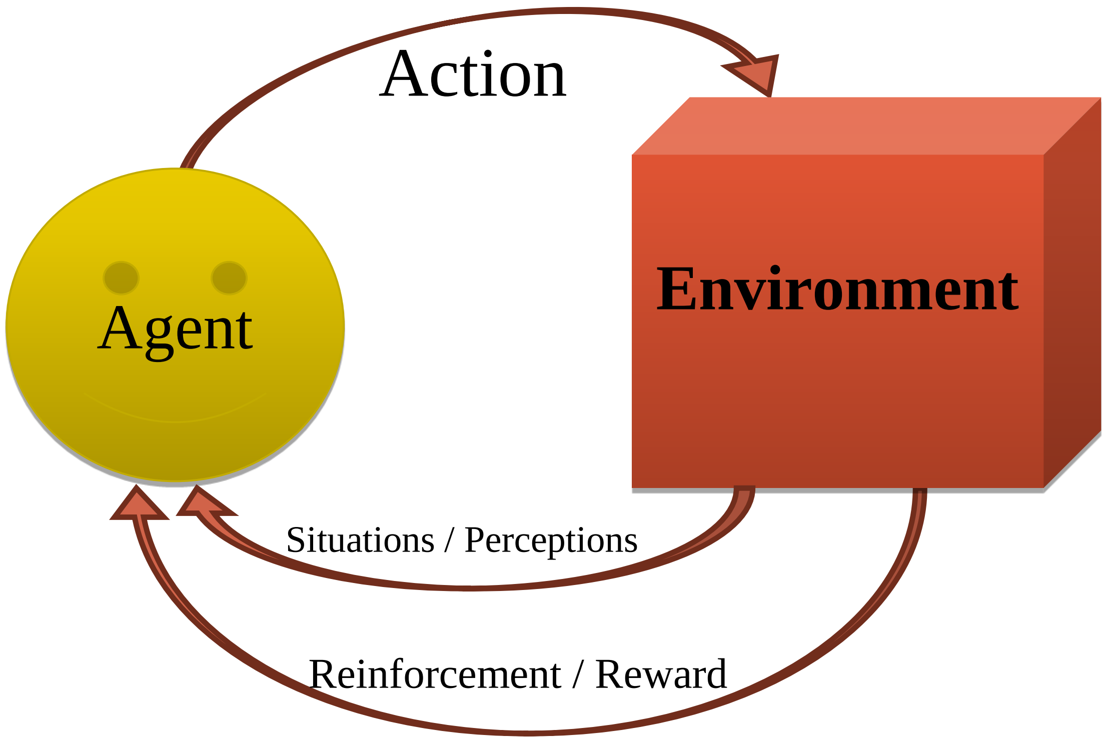


Figure 4: Reinforcement learning model

CHAPTER 2. LITERATURE REVIEW AND RELATED WORK

2.1 Learning Feature Representations

The main steps in machine learning classification are *feature selection, feature extraction*, and *classification*. For good performance we should have good feature representation from the raw data. There are many feature representation methods; each one focuses on certain special factors and deemphasizes others based on the domain of the data. Generally, the methods for feature representation are either flat or hierarchical and may be learned in supervised or unsupervised fashion. In this thesis we focus on unsupervised feature-learning methods. The most important ones are deep learning-based, which use a multilayer architecture for representation-learning, and *K-means-based*, which use a single layer.

2.1.1 Deep Learning

Introduced in 1960, the first learning algorithm, “Perceptron,” was a linear classifier over a simple feature extractor. By 2006, deep learning or hierarchical learning became well-known to machine-learning experts, who found that neural networks could be more powerful than previously thought. By connecting multiple levels of networks one could build a model with a higher level of abstraction from the raw data even using non-linear transformations and complex networks.[1-3]

Generally, deep learning is a hierarchy of representations in which the level of abstraction is increased at each level of the network. For instance, in an image domain for recognizing an object we can assume that the following levels would be learned: [4]

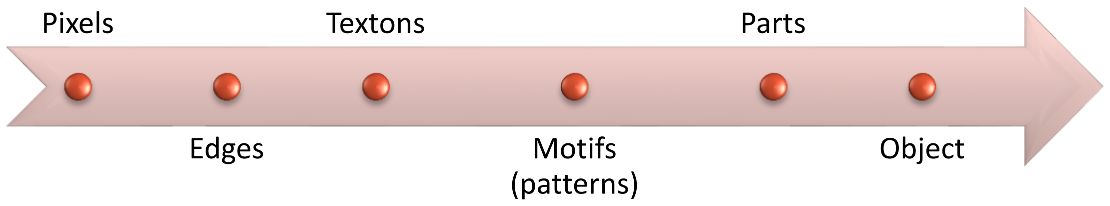


Figure 5: Levels of abstraction in the process of object recognition.

Three different categories are available in the neural network method:

- Feed-Forward NN: The connection is from input to output without any cycle. This is the first and simplest type of neural network and could have zero or more hidden layers[5]. The most important feed-forward algorithms are:
    - Multilayer Neural Networks[5]
    - Convolutional Networks[6]

- Feed-Back NN: At each level of the network or entire network, coding and decoding components can transmute and send errors back to previous levels for optimization in the next iteration. The most popular algorithms in the methods are:
    - Stacked Predictive Sparse Coding[7]
    - Deconvolutional Network[8, 9]

- Bi-Directional NN: In this method there is an interaction between the input and output of each layer. In other words, we can say that this method has no direction. . The most important algorithms in this method are:
    - Stacked Convolutional Auto-Encoders[10]
    - Deep Boltzmann Machines[11]

Some of these mentioned algorithms are very powerful in the image classification field. A majority of recent papers are based on algorithms such as stacked auto-encoders [10, 12-14] and convolutional neural networks (CNNs) [15-17]. This thesis focuses mainly on CNNs, which are partially connection NNs.

2.1.2 K-means Clustering

One of the most popular unsupervised learning algorithms is K-means, which clusters data based on the similarities between data points. The data are divided into K groups or clusters by their distances [18]. The usual first step is to select K random points as representative instances of the groups. Then in each iteration the algorithm calculates the distance of each input data point from those representative instances and assigns the point to a cluster. After that, the mean of the members in a given cluster would be the new representative instance of that cluster. Adam et al. [19] use K-means as an unsupervised feature-learning algorithm and compare its results with those of other methods.

The K-means algorithm is a simple but powerful approach for data clustering. The only issue with this approach is the speed, which is comparatively slow owing to the large number of distance calculations, especially on big data. There are some approaches to improve the speed; the best one uses the triangle inequality [20].

In the triangle inequality, for any three points $x, y, and\, z$ we know that $\|\overrightarrow{xz}\| \leq \|\overrightarrow{xy}\| + \|\overrightarrow{yz}\|$. According to that inequality, Elkan [20] proved that for each point x and centers b and c we have

1. If $\left\|\overrightarrow{bc}\right\| \geq 2 * \left\|\overrightarrow{xb}\right\|$ then $\|\overrightarrow{xc}\| \geq \left\|\overrightarrow{xb}\right\|$
2. $\|\overrightarrow{xc}\| \geq \max\left\{0, \left\|\overrightarrow{xb}\right\| - \left\|\overrightarrow{bc}\right\|\right\}$

These two lemmas make the K-means faster through having fewer calculations and using a caching technique. In this thesis work I have developed a software for K-means based on the triangle inequality technique. In addition, we enhance this method by using parallel computing, either CPU or GPU versions, which I will explain in greater detail in Chapter 4.

2.2 Data Augmentation

One technique that we can use for enhancing classification is data augmentation. Recent research has shown that augmenting data may increase the classification accuracy [21, 22].

A number of ways exist to apply augmentation on the data: transmission, rotation, scaling, and mirroring. Using each of these, one can augment the data and produce a new data set (or in our case, new images) with more training samples available for classification.

We can use augmentation for both training and test data sets. In the training phase, by augmenting the data we enlarge the data-set size and give the classification methods more data for learning. In the test phase, we can use augmentation by a voting technique.

First we predict the output of a test point and its augmented points, and then use voting to find which output is more frequent.

The main issue with data augmentation is that growing the size of the data set causes a drop in speed. A tradeoff between data augmentation and algorithm speed is necessary; the data are augmented as much as possible without sacrificing acceptable and reasonable speed.

2.3 Feature Concatenation

When we extract two or more sets of features from a data point, we can concatenate them and build a feature vector for each point of data. This combination will let the classifier work in higher dimensions and have more freedom in classification. This method usually works for classifiers such as SVM, which will be trained better by having more dimensionality. Zhang [23] and Damoulas and Girolami [24] explain this method and show that the combination of different feature sets provides better results than using each set separately.

I used this technique by combining extracted features related to different patch sizes into one feature vector for each data point. I will describe this in Chapter 3.

CHAPTER 3. IMAGE CLASSIFICATION PIPELINE

3.1 Image Preprocessing

One of the most important parts of any data-classification task, especially an image-classification task, is preprocessing. Raw data in the real world are usually dirty, which means:

- Incomplete: Parts of the data are missing or only partially complete
- Noisy: Data have errors, outliers, or misrepresentations
- Inconsistent: Data have some incongruities in either formatting or labeling.

Preprocessing is mainly a combination of data cleaning, to fill in missing values and remove noise and outliers by data transformation or normalization. Preprocessing is meant to eliminate as many anomalies as possible from the data. Coates et al. [19] show in their paper that whitening, which is a method of preprocessing applied by SVD or PCA, has a significant and positive effect on the result.

My research for this thesis focused on the CIFAR-10 [25] data set, which has no incompleteness or inconsistency, I will explain further in Chapter 4. Therefore, no data cleaning is required, and we may focus on data transformation.

Multiple methods are available for data transformation or normalization. Owing to the needs of the chosen data set, three different approaches were used. These approaches are outlined in the following sections.

3.1.1 Contrast Normalization

Contrast normalization, also called contrast stretching, is a method to normalize data values and stretch their histogram. Instead of obtaining centralized pixel values by subtracting the mean and dividing by the range of values, we can stretch the contrast and normalize the values based on a new value range:

$$if\ X \subseteq [a, b] \text{ then we need to normalize to } [c, d], \text{so we have}$$

$$X_{new} = (X - a) * \frac{d - c}{b - a} + c$$

For example, if the images are grayscale and there are 8 bits for each pixel, we can normalize the $minimum$ value min and the $maximum$ value max as follows:

$$I_{new} = (I_{old} - min\ ) * \frac{2^8 - 1}{max - min}$$

The change can be seen visually in Figure 6 and Figure 7.

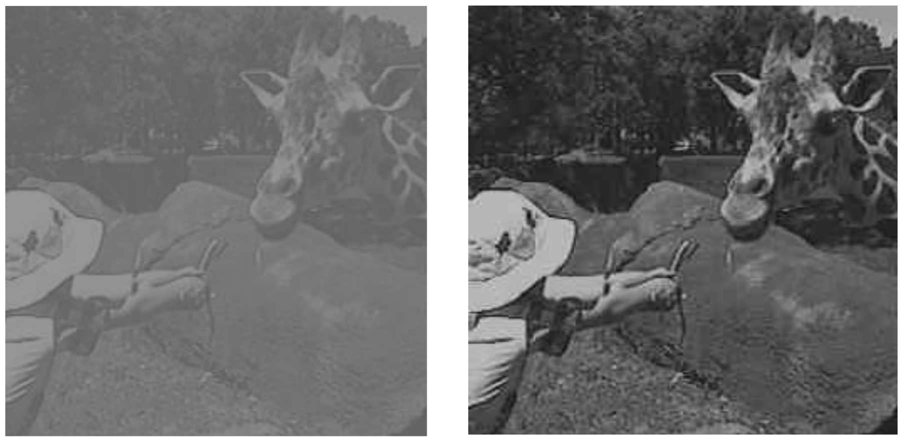


Figure 6: Grayscale image before and after contrast stretching

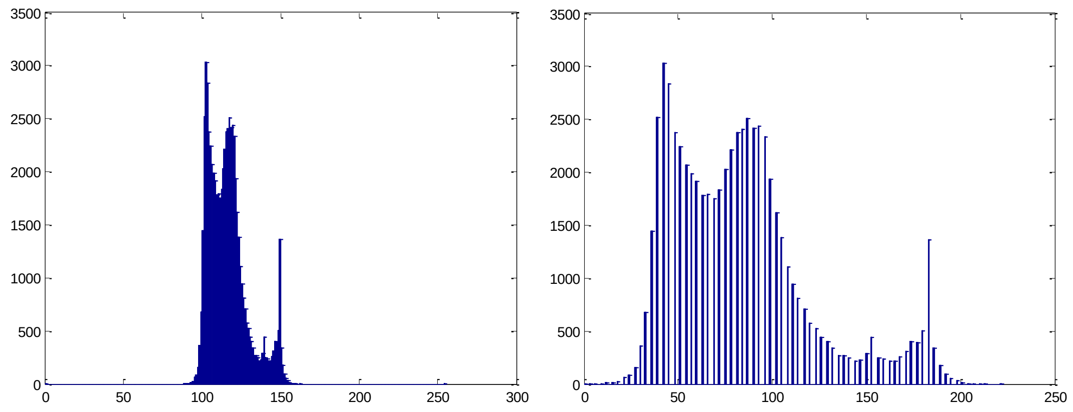


Figure 7: Histogram of grayscale image from Figure 6 before and after contrast normalization.

3.1.2 Zero Mean and Unit Variance (Z-Score)

Z-scores, explained in Abdi [26], are a standardization method that gives the data zero mean and unit variance. Briefly, in this method we simply eliminate the data mean and then divide the data by their standard deviation as follows:

$$X_{new} = \frac{X_{old} - \bar{X}}{s}$$

$$\text{where } \begin{cases} \bar{X} & is\ the\ mean\ of\ the\ data \\ s & is\ standard\ deviation\ of\ data \end{cases}$$

By applying this method to the previous image we obtain the following:

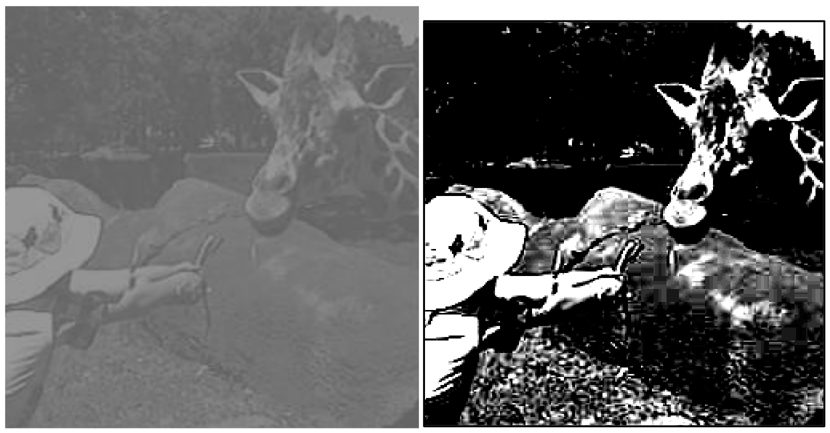


Figure 8: Applying z-score to an image

3.1.3 Principal Component Analysis (PCA) Whitening

Principal component analysis (PCA) is a linear transformation that tries to find the direction (components) of the data in a way that maximizes the variance of the data. Basically it is a dimensionality-reduction method, and the number of principal components is less than or equal to number of original variables.

Assume that we have data matrix X and know that

$$C = EDE^T \quad \begin{cases} C: & Covariance \\ E: & Eigenvectors\ in\ columns \\ D: & Eigenvector\ on\ diagonal \end{cases}$$

PCA whitening is a transformation of data as follows:

$$PCA_w(X) = D^{-\frac{1}{2}}E^T$$

The main problem with this method is that PCA whitening is not a unique one-time process; whitened data can be whitened again [26].

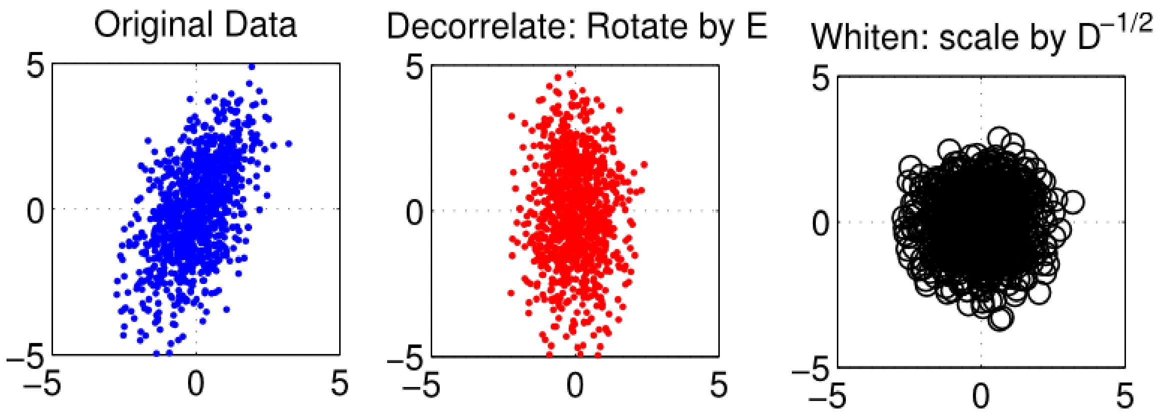


Figure 9: Effect of PCA whitening on data [27]

3.1.4 Zero-phase Whitening (ZCA)

Zero-phase whitening (ZCA) was introduced by Bell and Sejnowski in 1996 [28]. They used E and stacked together two eigenvectors to calculate an orthogonal matrix:

$$ZCA_w(X) = ED^{-\frac{1}{2}}E^T$$

The result of this type of whitening has been proved to be as close as possible to the original data and has a significant effect on the image classification [19]. I used this in addition to contrast normalization and Z-scores.

3.2 Feature Learning Process

The next step after preprocessing, which includes normalization and whitening, is finding a way to learn features and mining them from the raw data. For this process I focused on K-means, an unsupervised method and a powerful clustering algorithm.

Feature learning by K-means was proposed by Coates and Ng in [29]. Using the patching technique in their approach, they showed that K-means could be a good method for feature-learning problems.

The following sections describe the K-means process for feature learning.

3.2.1 Patching Technique

For feature learning the raw data are divided into small patches. As shown in Figure 10, we have two parameters: *receptive field* size p, the size of the patches, and *stride* s, the step size of the patching process.

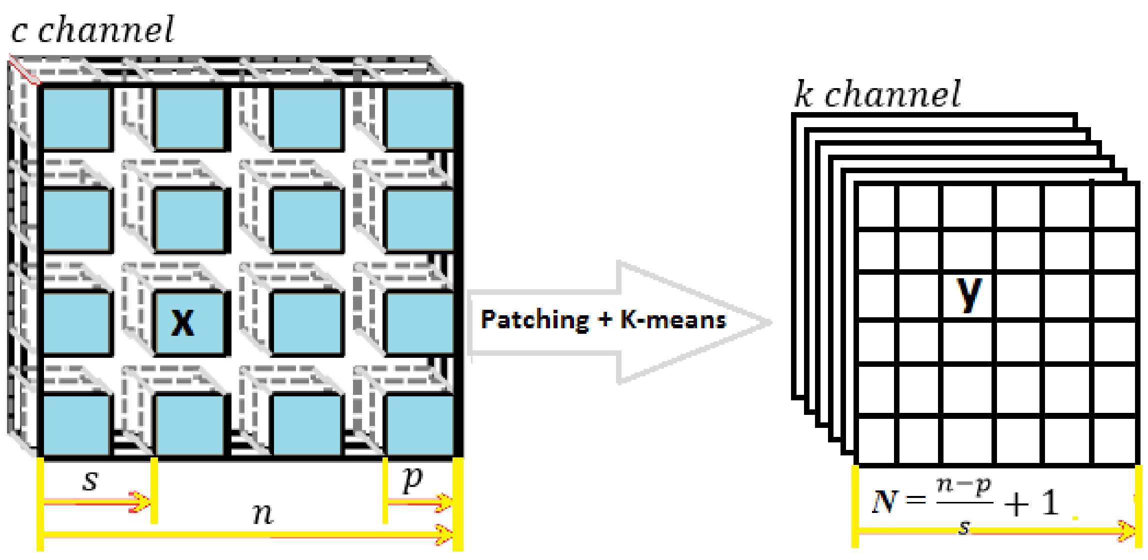


Figure 10: Applying k-means by using patching.

Figure 10 shows the process of extracting features from the data X with size $n \times n$ by using $p \times p$ patches separated by s pixels, and then applying K centroids, which are calculated by K-means, to generated new feature vectors for the data Y.

The only problem with using this technique is that dimensionality will not be decreased. The new width N is actually less than the old width n, but on the other hand, usually $K \gg d$. This means that the new feature space will be much larger than the original feature space.

$$N \times N \times K \gg n \times n \times d$$

Implementing this process means that we would have a bigger data set whose size would make training the classifier a problem. There are some techniques for overcoming this issue. One is using the pooling method, which will be described in Section 3.3.2.

3.2.2 K-means Clustering

K-means is a powerful unsupervised method that performs clustering. As explained in the previous section, the patching technique for feature learning produces a bunch of small patches. We can use K-means for grouping these patches and finding their centroids as their representative instances. The only parameter for this algorithm is K, the number of clusters or learned features.

As described before, I used the K-means which is described by Elkan in his paper[20] and is based on the triangle inequality. I implemented a multi-thread CPU–based software that runs extremely faster than that one. This software can take an input file as its data set plus some other parameters such as the receptive field size k and calculate the centroids. We can call the software from a command line. This is a good feature as it can be used for other software such as MATLAB.

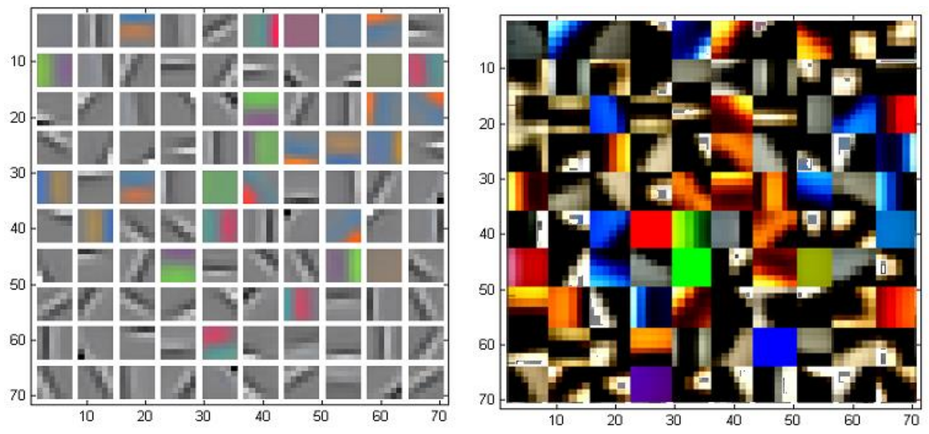


Figure 11: Learned features by k-means with and without whitening

There are two ways to initialize:

1. Using randomly generated centers: In this approach we can generate K points by using uniformly-distributed random numbers. This method is faster, and for my thesis had better results.

2. In the second approach, we select the mean of the points as the first center, and then calculate all the distances. We select another point from the data as next point, and group the data based on the distances to each center, and so on.

The second approach is slower than the first, and needs an initialization process to find K centroids before the main body of the algorithm is started.

3.3 Feature Extraction

After finding and learning features, we use them to extract similar features from the raw data. Feature extraction is completely different from feature selection. Feature extraction consists of two phases: feature construction and feature selection. The purpose of feature construction is to integrate different attributes of the raw data to build a higher level of abstraction. This could happen on hidden layers of the neural network or even by using K-means to find those representations. In some cases doing preprocessing can work as feature construction, although that is not the case here. Feature selection is the process of selecting related and informative features. Using the learned features as filters or using any other algorithm such as Chi-squared or TF-IDF are some typical approaches of feature selection.[30]

There are many ways to do feature extraction. In image processing we use these methods to find edges, corners, textons, motifs, and so on. I performed feature extraction in two steps using those learned centroids from K-means. The main and first part is convolution. Since we do feature extraction to reduce the volume of data, especially large data, in order to describe them in a better way, I used the pooling technique for dimensionality reduction. I'll describe both ways in the following sections.

3.3.1 Convolution

Convolution is one of the most important methods in signal processing. Convolution by filters can change and convert the data input into a new representation. Mathematically it is a dot product of the input by filters as follows:

$$Output = Input \cdot Filter$$

Hence in the $2D$ domain each output cell at position x and y is calculated by the following equation which F is filter $n \times m$ pixels:

$$Output(x, y) = input(x, y) \cdot F_{n\times m} = \sum_{i=1}^{n}\sum_{j=1}^{m} X(i + x, j + y) * F(i, j)$$

As we can see, the result will be a scalar value for a given filter F.

We can use this method to convolve the extracted filters (centroids) with the original data, and generate their impulse response. Assume that we have images 16×16 pixels in size and that we extracted 10 features using K-means with $P = 3$.

Therefore, by using convolution we will have a new image (or data point) $16 - 3 + 1 = 14$ pixels in each side. We can see the convolution process in Figure 12, which shows a 3×3–pixel filter. For the first convolution we have

$$Y(1,1) = X(1{:}3,1{:}3) \cdot F(1{:}3,1{:}3)$$

$$\begin{aligned} Y(1,1) = {} & 1*0+0*1+0*0+ \\ & 1*1+0*2+1*1+ \\ & 1*0+0*1+0*0 = 2 \end{aligned}$$

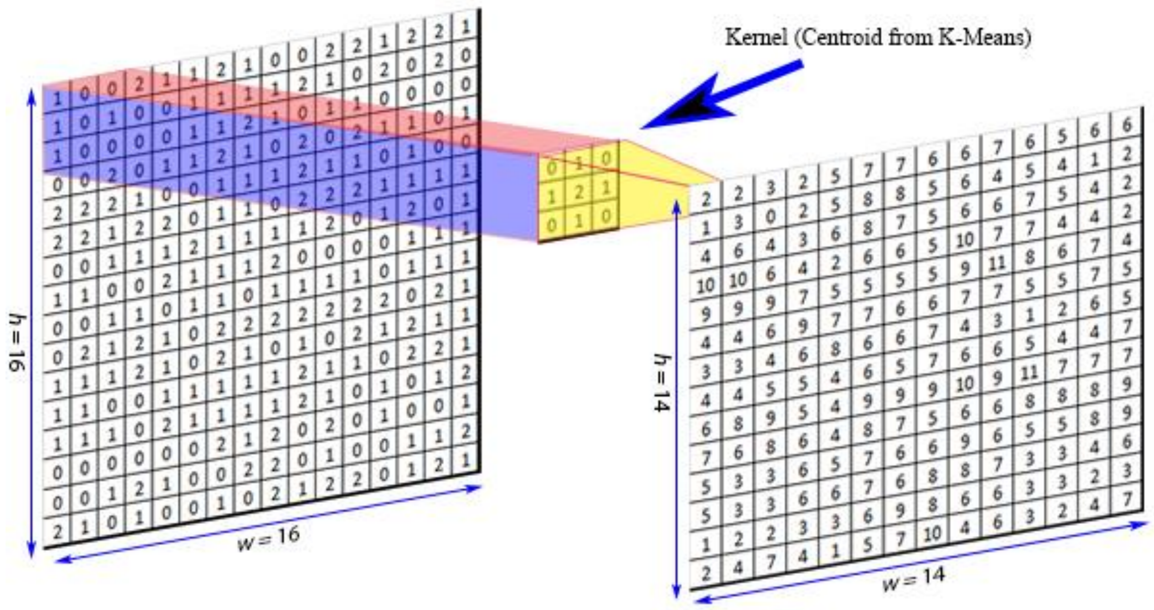


Figure 12: Convolution process

Using convolution is popular in image classification. The literature contains many papers dealing with a convolutional neural network (CNN) model that results in good accuracy.[15, 31, 32]

To obtain several feature map vector from centroids of the K-means phase, there are two methods for generating a feature-map vector[19]:

- Hard assignment: In this approach we set 1 for closest center to the given patch of input data and 0 for the rest. For instance, in the previous example we have 10 centers or filters; applying all of them in each input image for every cell of output data gives a

(1×10) feature-map vector. Each element of this feature vector could be assigned by the following equation:

$$feat\_vector_k = \begin{cases} 1 & if\ k = argmin\|c_j - x\|_2^2 \\ 0 & otherwise \end{cases}$$

- Soft assignment: In this method, all filters have an effect on the feature-map vector with respect to their distance subtracted from the mean of all of them. The following equation is a mathematical definition of this method:

$$feat_{vector_k} = \max\{0, \mu_z - z_k\}$$

where

$$z_k = \|x - c_k\|_2$$
$$\mu_z = \underset{k}{\text{mean}}\, z_k$$

Since this method retains more information, it should have better results than the alternative.

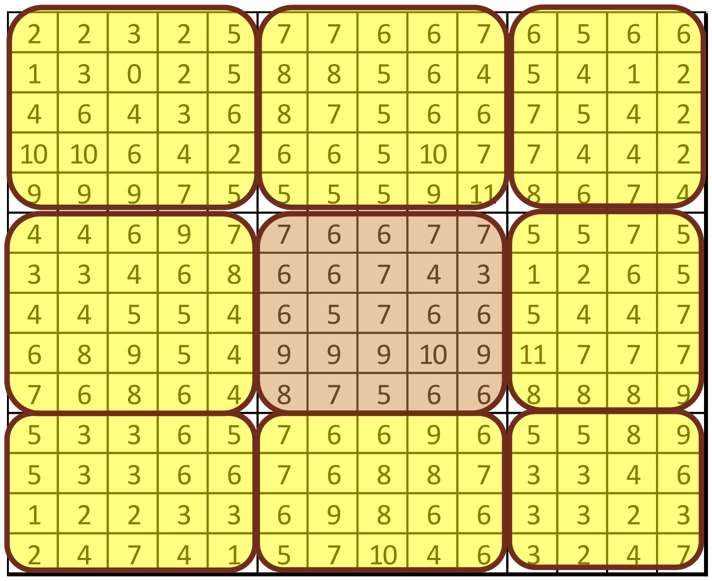
3.3.2 Pooling Technique

Using convolution and having many learned features results in a very large data set. In the previous section, we assumed that there is a grayscale-image data set with each image size equal to $16 \times 16 \times 1$ = 256 pixels. By learning ten features of size $6 \times 6 \times 1$ in pixels and then doing convolution for each input image, we will have an output of size $14 \times 14 \times 10 = 1960$, seven times larger than the input image, and our data set will be seven times of its original volume. Here, the pooling technique is a useful method to reduce the volume size or dimensionality.

Pooling reduces a block of $a \times b$ units to one value. The purpose is to get spatial invariance by reducing the dimensions of the feature-map matrix [6].

There are two type of pooling functions:

1. Max pooling is getting the maximum value of a specific block. In the following figure, using max pooling 5×5 a value of 10 would be selected as the output for the red block.



2	2	3	2	5	7	7	6	6	7	6	5	6	6
1	3	0	2	5	8	8	5	6	4	5	4	1	2
4	6	4	3	6	8	7	5	6	6	7	5	4	2
10	10	6	4	2	6	6	5	10	7	7	4	4	2
9	9	9	7	5	5	5	5	9	11	8	6	7	4
4	4	6	9	7	7	6	6	7	7	5	5	7	5
3	3	4	6	8	6	6	7	4	3	1	2	6	5
4	4	5	5	4	6	5	7	6	6	5	4	4	7
6	8	9	5	4	9	9	9	10	9	11	7	7	7
7	6	8	6	4	8	7	5	6	6	8	8	8	9
5	3	3	6	5	7	6	6	9	6	5	5	8	9
5	3	3	6	6	7	6	8	8	7	3	3	4	6
1	2	2	3	3	6	9	8	6	6	3	3	2	3
2	4	7	4	1	5	7	10	4	6	3	2	4	7

Figure 13: Max-pooling sample

The process is based on the following equation:

$$y_{out} = \max_{i \in [1,a]} \left(\max_{j \in [1,b]} input\_block(i,j) \right)$$

For each input block function we will get the maximum value.

2. Subsampling is a more powerful function that takes all the neighborhoods into account by getting an average of the inputs of a given block, multiplies it by a trainable scalar β and add bias b. Then using a non-linear function such as $tanh$ as follows[33]:

$$y_j = \tanh\left(\beta \sum_{N\times N} a_i^{n\times n} + b\right)$$

The pooling technique thus reduces the amount of data to be processed; for example, dividing the feature-map matrix into2×2 regions reduces the data set to $1/4$ of the original number of elements.

3.4 Classification

The final step in image classification is classification, which follows preprocessing, feature learning (FL), and feature selection. During the classification phase we choose a classifier to build a model, and then use the model classifying test cases. Among the many types of classification algorithms already available, I preferred to use a support vector machine (SVM)[34].

SVM is a powerful classifier specifically for high dimensions. It is based on the concept of margin-maximizing hyperplane that seperates two classes to maximize the margin between them. SVM is a binary classifier that can assign a point only to one of two groups. It builds a model based on the given training data set and then using that model will predict and classify new samples of the data.

If the data are linearly separable, linear SVM can find a maximum-margin hyperplane that separates the data into two groups. In this case, the hyperplane equation would be $\boldsymbol{w \cdot x - b = 0}$ and the regions are bounded by $\boldsymbol{w \cdot x - b = 1}$ and $\boldsymbol{w \cdot x - b = -1}$. This type of margin is called a hard margin.

Cortes and Vapnik [34] proposed a new idea that can handle mislabeled instances. In other words, if we could not find any hyperplane to separate two classes, their technique, known as soft-margin SVM, can find the best hyperplane that achieves maximal separation of the data. It does this by introducing a slack variable ξ_i that controls the degree of misclassification.

3.4.1 Types of SVM

SVM supports both regression and classification for numerical and categorical variables. Based on these, SVM models are classified into the following groups [35]:

- Classification SVMs
  - C-SVM classification

    C-SVM is the first soft-margin method in SVM using the parameter ξ. C controls how much misclassifying is acceptable. For large values of C the SVM will choose the smaller-margin hyperplane, and for small values of C the optimizer finds the larger-margin hyperplane.

$$Error\ function : \frac{1}{2}w^t w + C\sum_{i=1}^{N}\xi_i$$

$$Constraints: \begin{cases} y_i(w^t\phi(x_i) + b) \geq 1 - \xi_i \ \ \forall\, i = 1..N \\ \xi \geq 0 \end{cases}$$

- ν-SVM classification

  ν-SVM, proposed by Scholkopf et al. [36], has a parameter ν instead of C. This parameter is used to control the number of instances that are support vectors and that lie on the wrong side of that hyperplane.

$$Error\ function: \frac{1}{2}w^t w - \nu\rho + \frac{1}{N}\sum_{i=1}^{N}\xi_i$$

$$Constraints: \begin{cases} y_i(w^t\phi(x_i) + b) \geq \rho - \xi_i \ \ \forall\, i = 1..N \\ \xi \geq 0 \\ \rho \geq 0 \end{cases}$$

➢ Regression SVM: Like the classification method, regression SVM tries to find the generalization bounds for regression. In other words, regression SVM will predict a new sample according to the nature of the regression concept [37].

- ϵ-SVM regression

  In this method, instead of having a square loss function in the least-square regression, the error function is based on the ϵ. Generally we can say that errors less than ϵ will be ignored.

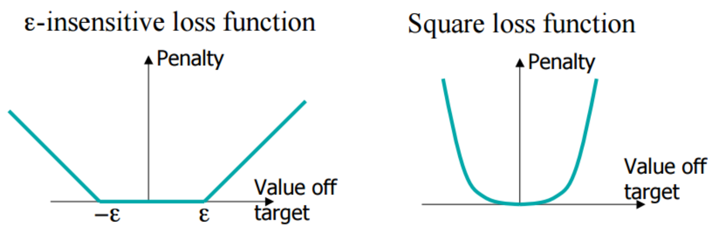


Figure 14 : Regression loos functions

$$Error\ function: \frac{1}{2}w^t w + C\sum_{i=1}^{N}(\xi_i + \xi_i^*)$$

$$Constraints: \begin{cases} (w^t\phi(x_i) + b) - y_i \leq \epsilon + \xi_i^* & \forall\ i = 1..N \\ -[(w^t\phi(x_i) + b) - y_i] \leq \epsilon + \xi_i & \forall\ i = 1..N \\ \xi_i \geq 0 \quad and \quad \xi_i^* \geq 0 \end{cases}$$

- ν-SVM regression

  Like the classification method, it employs ν to control the number of supported vectors.

$$Error\ function: \frac{1}{2}w^t w - C\left(\nu\epsilon + \frac{1}{N}\sum_{i=1}^{N}(\xi_i + \xi_i^*)\right)$$

$$Constraints: \begin{cases} (w^t\phi(x_i) + b) - y_i \leq \epsilon + \xi_i & \forall\ i = 1..N \\ -[(w^t\phi(x_i) + b) - y_i] \leq \epsilon + \xi_i^* & \forall\ i = 1..N \\ \xi_i \geq 0 \quad and \quad \xi_i^* \geq 0 \quad and\ \epsilon > 0 \\ 0 \end{cases}$$

3.4.2 Kernel functions

By nature SVM is linear but it can accept a kernel function to work in either non-linear form or in higher dimensions. Some of the major kernel functions for SVM are as follows:

- Linear: $X_i \cdot X_j$
- Polynomial: $(\gamma X_i \cdot X_j + C)^d$

- RBF: $e^{\left(-\gamma|X_i - X_j|^2\right)}$

- Sigmoid: $\tanh\left(\gamma X_i \cdot X_j + C\right)$

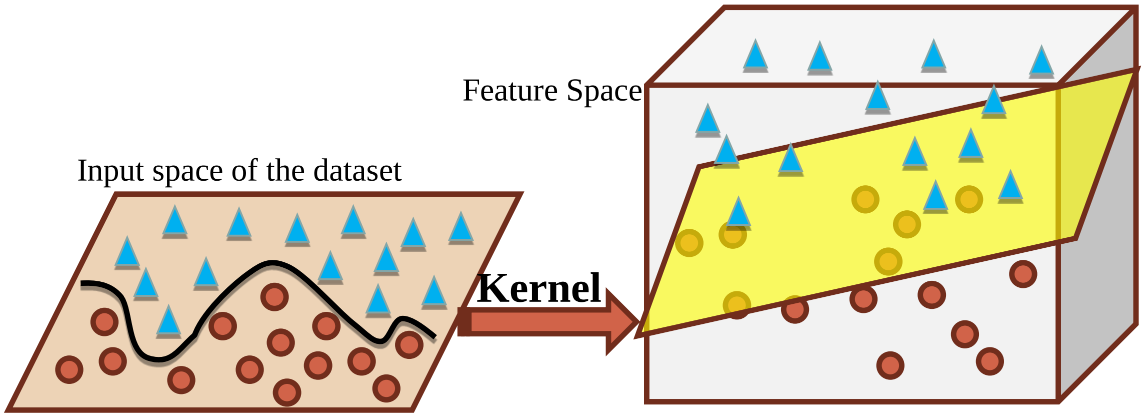


Figure 15: Effect of kernels in SVM

3.4.3 Multiclass SVM

SVM is a binary classification method. There are some approaches for using SVM to solve multi-class problems. Of these, two methods are most important:[38]

- One against one: For K classes, this method builds $\frac{K(k-1)}{2}$ different SVM models, each model generated by training with data from each pair of classes. The best way to test is using a voting strategy to determine which class has the highest vote and should be selected.

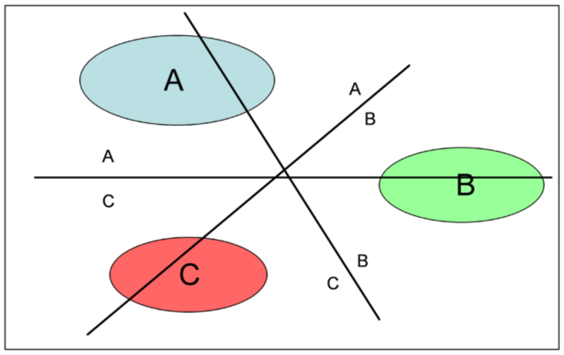


Figure 16: SVM model for one-against-one approach

- One against all: For K classes, this method builds K models; each $model_i$ is generated from training between instances of $class_i$ with all other instances from other classes. For the testing step, we can select $class_j$, which has a maximum value of $\left(W^j\right)^T \phi(x) + b^j$

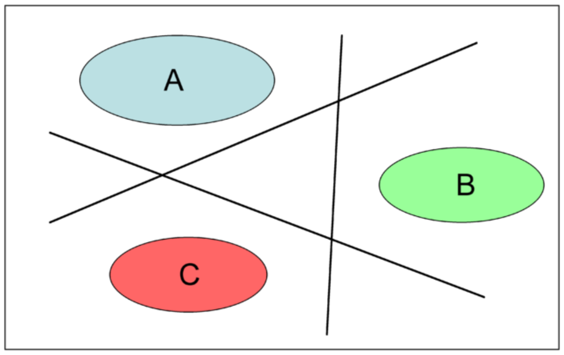


Figure 17: SVM model for one-against-all approach

3.4.4 Cross Validation

Sometimes we need to estimate the accuracy of a model for an independent data set. In classification, before the testing phase we need to know how accurate and trustable our model is. Validating is a major step to verify that the model is ready to use. For the testing step we ensure that we have acceptable accuracy without any major over-fitting.

Cross-validation is the process of validating the model before testing. The first step of cross-validation is to divide the training data set into complementary subsets. The next step is to train the model on the first subset and test with the second. This process is repeated using a different pair of train/test sets and the results are recorded each time. Finally, the average of those validations results will be an estimation of the model based on the given parameters.

In k-fold cross-validation, the training data set is divided into k subsamples. In each iteration, one of those k subsets is used for validation of the model and the remainder are used for training. After k iterations the average of those validations would be the estimation of the accuracy of the model. Two-fold cross-validation is a simple form of k-fold in which the two sets are of equal size.

CHAPTER 4. THESIS EXPERIMENT SETTINGS

4.1 Data Set

For my thesis, I focused on the data set CIFAR-10 [25], which contains 60,000 RGB images, each 32×32 pixels, in 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck; 50,000 images were used as training data and 10,000 as test data.

CIFAR-10 is a popular data set utilized by many papers in order to have a common benchmark for comparison. Therefore, I used it to be able to compare my results to those of other methods as well.

Table 1: CIFAR-10 specifications

Data set type	Images data
Size of each instance	32×32 pixels
Number of channels	3 channels RGB
Number of classes:	10 classes
Number of training instances	50,000 images
Number of test instances	10,000 images

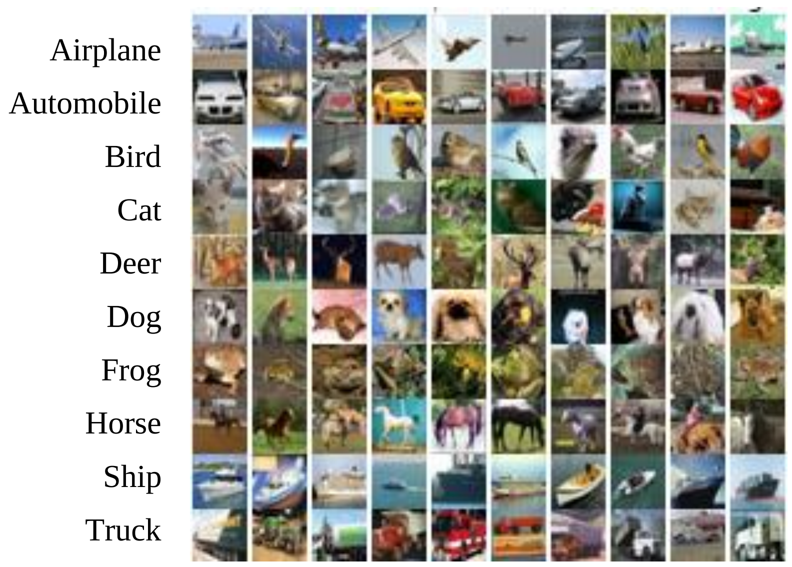


Figure 18: Categories of CIFAR 10

4.2 Process Pipeline

Figure 19 shows my process pipeline, beginning with preprocessing, passing through feature extraction and feature selection, and ending with classification. Each of these will be described in the following sections.

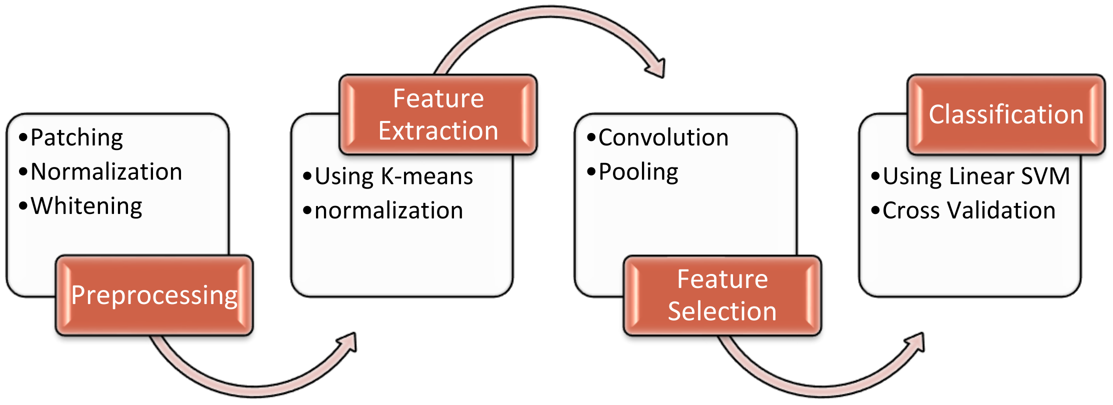


Figure 19: Process pipeline

4.3 Preprocessing

The CIFAR-10 data set is a complete and standard data set that does not have any incomplete values or any inconsistencies. Therefore, data cleaning is not required. For data normalization and whitening I used MATLAB to do the following:

1. To begin, I used the patching technique with receptive field sizes 6, 8, 10, 12, and 16 to split each image into smaller patches.
2. After patching, I applied a Z-score to each patch to give it zero mean and unit variance.
3. Finally, I applied ZCA over all the patches with the following piece of script:

```
C = cov(patches);
M = mean(patches);
[V,D] = eig(C);
P = V * diag(sqrt(1./(diag(D) + 0.1))) * V';
data = bsxfun(@minus, patches, M) * P;
```

Parts 1 and 2 run very slowly in MATLAB, so I developed a software in C++ that accepts images and breaks them into the desired receptive field sizes and then applied the Z-scores over them [39]. Part 3 was done in MATLAB and does not take so much time to process. We can see relevant codes in Appendix A.

4.4 Feature Learning

After splitting the training images into small patches and applying normalization and whitening, the next step is feature learning. As already mentioned, I used K-means as the unsupervised feature-learning approach for this thesis. I implemented a K-means software based on the triangle inequality [20] using the power of multi-threading from CPU and GPU. My system GPU was not very successful because it is too slow and lacks sufficient memory for this project. But in contrast, the CPU version is quite fast and takes less than one hour for this data set [39]. This software runs in command-line mode and can take the input data set and K as its basic parameters. I can also run in random initialization or processing mode and in the end generate 3 main files:

- Centers (-co), which contains the centroids
- Selected clusters (-clo), which includes the selected clusters for each data point
- Distance values (-dio), which contains the distance values between each point and its cluster centroid.

Using my software, I can send the prepared patches to software and define the number of features (centroids) that I want. I set $k = 100, 200, 500, 1000,$ and 2000. Figure 20 shows the results for some different receptive field sizes.

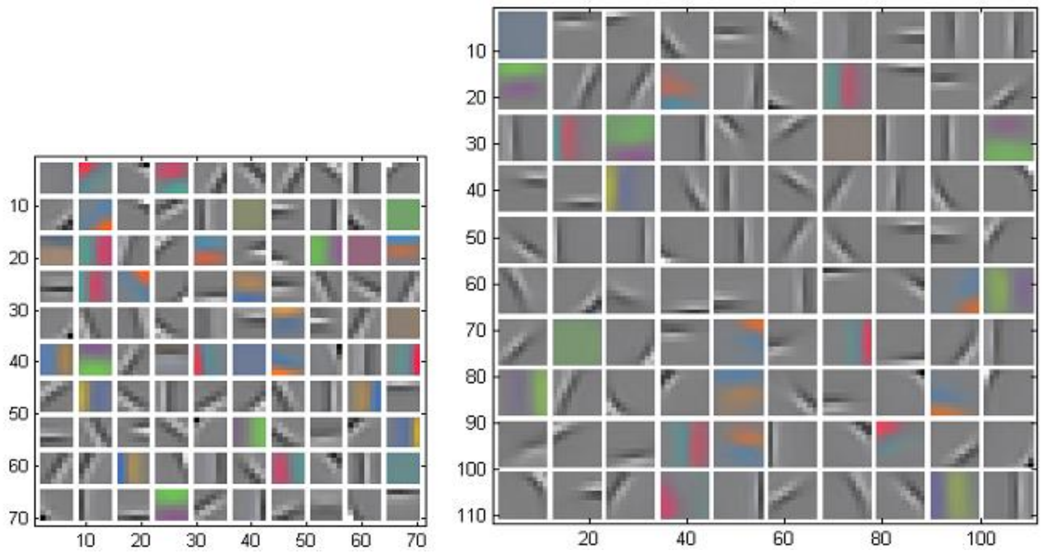


Figure 20: 100 Learned features with patch size 6 (left) and 10 (right)

4.5 Feature Extraction

After learning the features and extracting them from the images, we use them to convolute into the original images and generate the feature vector maps for each image. As explained in Section 3.3.1, convoluting each centroid or filter into each image will generate a new matrix with smaller width and height with respect to receptive field size. Hence, by applying all the features we obtain a k-convoluted matrix for each image, as we can see in Figure 10. For instance, with receptive field size equal to 6 and $k = 100$ we have a new multi-dimensional matrix $27 \times 27 \times 100$ for each given image, considerably bigger than the input data, which were only $32 \times 32 \times 3$. To handle this enlargement we can use the pooling technique.

The pooling technique, described in Section3.3.2, is a way of integrating a block of neighborhood units into one unit. In my experiments I used 2×2 and 3×3 regions for each generated feature vector with max-pooling methods that reduced the dimensionality to $1/4$ and $1/9$ of the originals, respectively.

For the whole convolution and pooling process, I wrote another software program that used GPU for faster performance [40].

After generating the feature vectors I applied another normalization over each attribute among all the images to convert them into normal distribution.

4.6 Classification

The last step is classification. As mentioned before, I used SVM for classification. This is one of the most powerful classifications and runs quickly. I used LibLinear library [41], which is one of the fastest and most accurate tools currently available. I tried different variations of parameters for it; the best results were achieved with $s = 0\ and\ c = 1$. I also used cross-validation, which LibLinear supports, with $v = 5$.

4.7 Environment configurations

I developed all the software and wrote all the scripts on my laptop system, whose configuration is given in Table 2. Then I ran experiments on two systems. One was our laboratory PC (specifications in Table 2); the other, used for K-means with large k, was an Amazon $EC2$ server with Windows OS (specifications in Table 2);.

Table 2: Environment configurations

System	CPU	Hard Disc	Memory	GPU
My laptop	Core $i7\ 720qm$	128 GB SSD	4 GB	GeForce $GT230M$ 1024 CUDA cores 1GB Mem
Lab PC	Intel Xeon $W3550$	500 GB	12 GB	Quadro FX 580 512 CUDA cores 1GB Mem
Amazon EC2 g2.2xlarge	Intel Xeon $E5$-2670	60 GB SSD	15 DB	NVDIA Cluster 1536 CUDA cores 4GB MEM

CHAPTER 5. OTHER METHODS

5.1 Data Augmentation

After a standard process of whitening, feature extraction, and classification, I preferred to use data augmentation. As explained in Section 2.2, we can use the augmentation technique to prepare some data points similar to the original given data. By doing this, we can generate a new data set that simulates new data that are different from what we originally had. An image can be augmented by the following methods:

- Transmission: can be applied vertically or horizontally and in each direction. By moving the picture one can simulate cases in which the object might have been on another part of the image.
- Scaling: may be used for enlarging or shrinking. Either way, we try to simulate cases in which an object could have another size.
- Rotation: can be clockwise or counterclockwise. In this case, objects with some rotations can be simulated. This rotation would be very useful if the image shows the top of an object.
- Mirroring: we can mirror or flip the image to generate an object with reflected properties.

I used all four data augmentations as follows:

1. I did transmission in both vertical and horizontal directions. For directions I moved the images by values $[-10, -5, +5, +10]$. I also did a mixed transmission by moving in diagonal directions with those values.

By doing those transmissions I generated a data set whose volume was $4 \times 4 = 16$ times the original one.

2. For scaling I resized the images by these percentages: $[60,\ 80, 120, 150]$

   This grew the data set to 4 times its original volume.

3. I made some small rotations because the images show the front of the objects and rotation would be meaningless in this case. Therefore I used just the following values: $[-5°, +5°]$

4. And finally for the mirroring, I flipped the images only horizontally because vertical mirroring is incorrect for images showing the front of an object such as ones which exist on out dataset.

I applied all the above augmentations and generated a new data set that was 23 times bigger than the original. Handling data sets of this size with SVM requires more memory than I could access. Therefore, instead of running all the data, I subsampled the data set by selecting 400 instances for each class from the 5,000 images actually available in each class. These augmentations were applied on both training and test data.

5.2 Feature Concatenation

Feature concatenation is another method of adding more information to a data set for classification. By concatenating the features extracted by feature-learning algorithms we can combine the information from two or more different feature vectors into one and give the classifier a better chance to learn the model.

For this purpose I concatenated feature vectors that were generated based on the different receptive field sizes. For example, I concatenated feature vectors generated from $p = 6$ and $p = 10$; the result is shown in the next chapter.

5.3 Multilayer Feature Selection

This method is the most interesting one to me. The basic concept is repeating the feature-selection process multiple times before classification. In deep learning, the layers are performing the same process. They get input data, learn the weights, and assign the outputs. The proposed pipeline in Figure 19 consists of four parts. The first three parts are a group of feature-learning steps and the fourth is simply the final classification. If we repeat the feature-learning package multiple times, each layer will get a higher level of abstraction features from the data. Doing the process for 2 iterations on part of the data gave me better results than a single iteration did. Figure 21 shows that process in which the learning package would be placed in each level and all levels are the same.

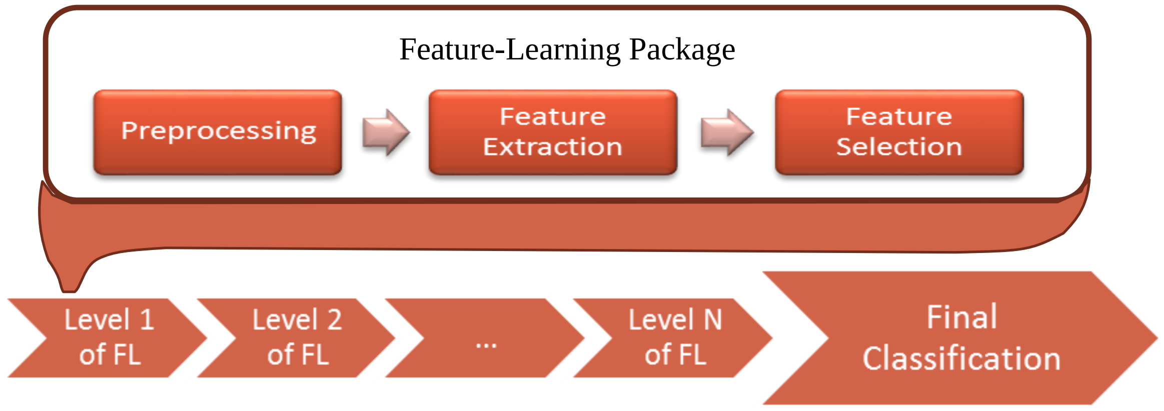


Figure 21: Multi-layer feature learning

CHAPTER 6. RESULTS

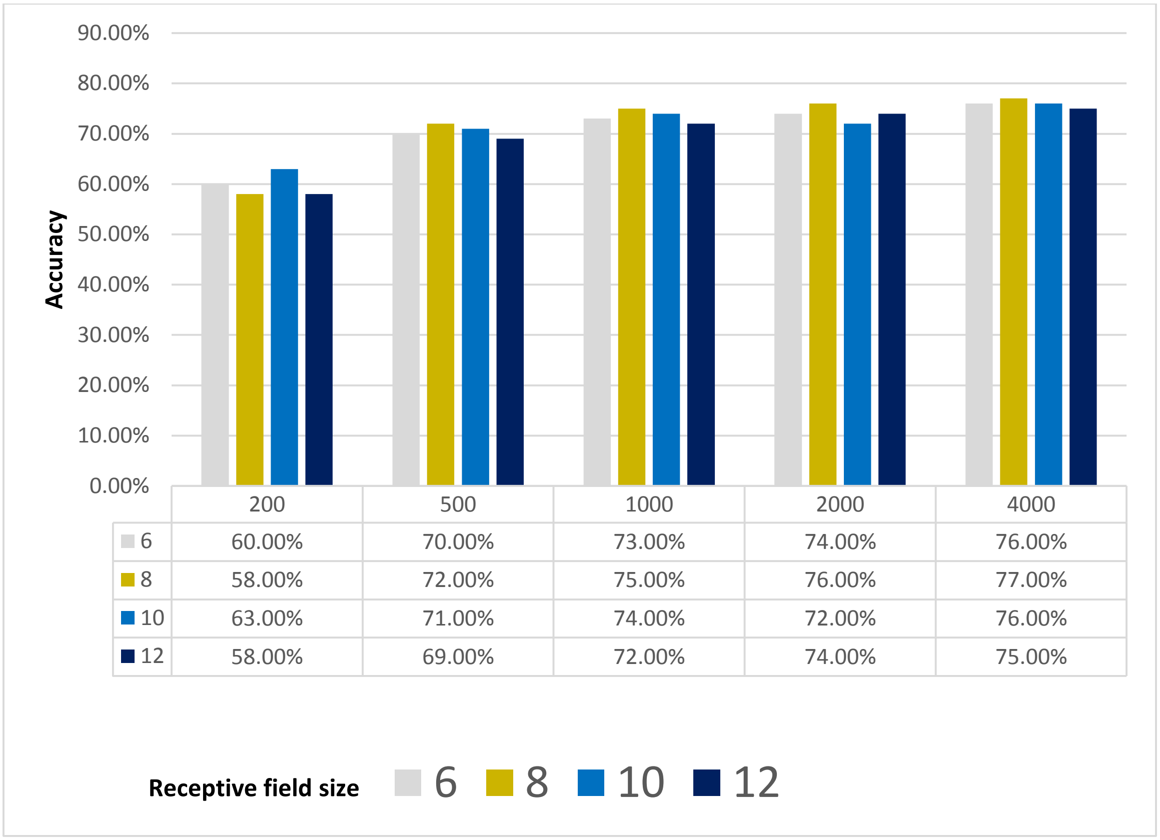
This chapter discusses the final results of my project. Many of these results were obtained from runs lasting a couple of hours. I used unsupervised learning methods and tried to utilize most of the tricks that I mentioned in previous chapters. The main problem for the project was shortage of time. I needed to run massive algorithms over big data sets. The patch parts compelled me to develop software and achieve the benefits of multi-threading. In some cases, I used data samples for faster results, but in most the entire data set was used.

After a variety of experiments, I generated the following four sets of results, which will be described in the following sections.

6.1 Different Patch Sizes and Ks

Figure 22 shows the results of using a different patch size and a different number of centroids. Note that increasing the size of the patches may result in a slight reduction in accuracy. Therefore, this parameter must be neither too small nor too large; in my experience the best choice was a value of ~ 8.

On the other hand, the number of centroids, i.e., the value of k, in K-means has a direct effect on accuracy. When the value of k increases, so does the accuracy. However, the number of centroids for $k > 1000$ does not have any significant effect. Therefore, a k value of ~ 1000 for this data set would be a good selection.

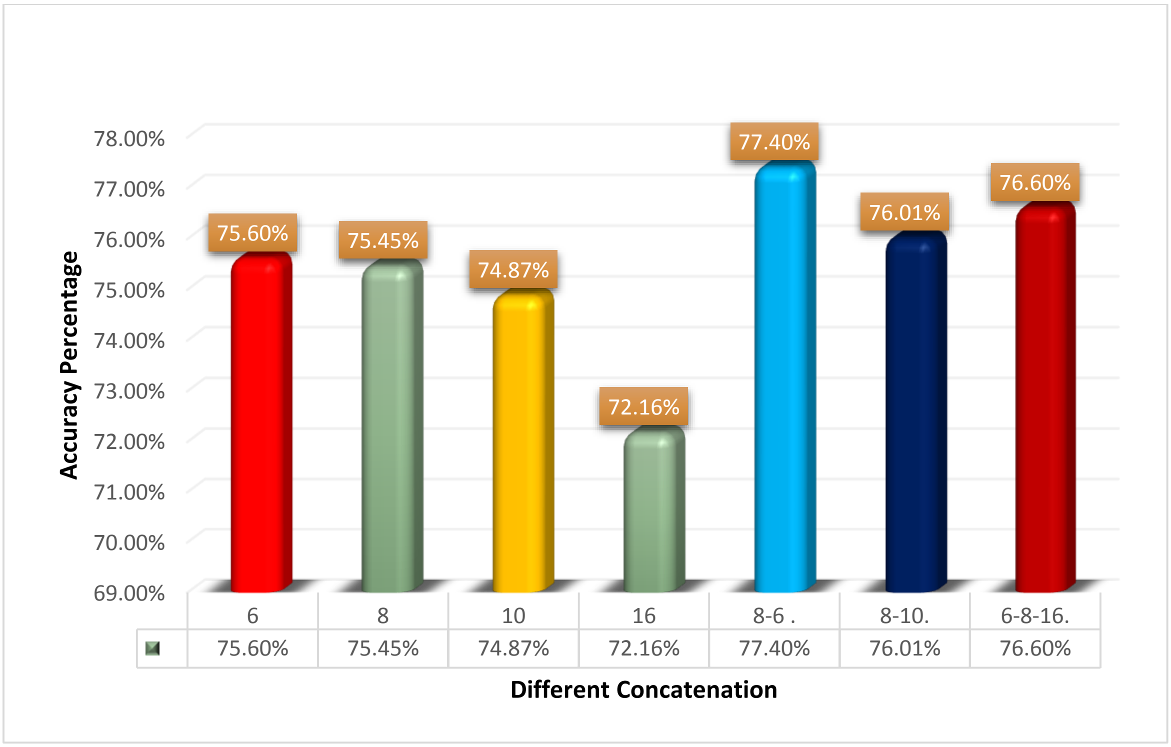


	200	500	1000	2000	4000
■ 6	60.00%	70.00%	73.00%	74.00%	76.00%
■ 8	58.00%	72.00%	75.00%	76.00%	77.00%
■ 10	63.00%	71.00%	74.00%	72.00%	76.00%
■ 12	58.00%	69.00%	72.00%	74.00%	75.00%

Figure 22: Result of different p and k

6.2 Using Concatenation

Concatenating the feature vectors was my second test case. I concatenated the feature vectors of patches size $6, 8, 10$ and 16 in three different testing cases. We can see that the accuracy percentage of concatenating is not so different in compare to single feature vectors and this result is interesting. This type of concatenating provided no improvement in accuracy. The reason for this is that in the first level of feature learning we have only the edges as features. For small images such as those in CIFAR-10, the size of these edges are not very important, and there is no big different between filters with 6×6 and 10×10 pixels. Figure 23 shows these results.



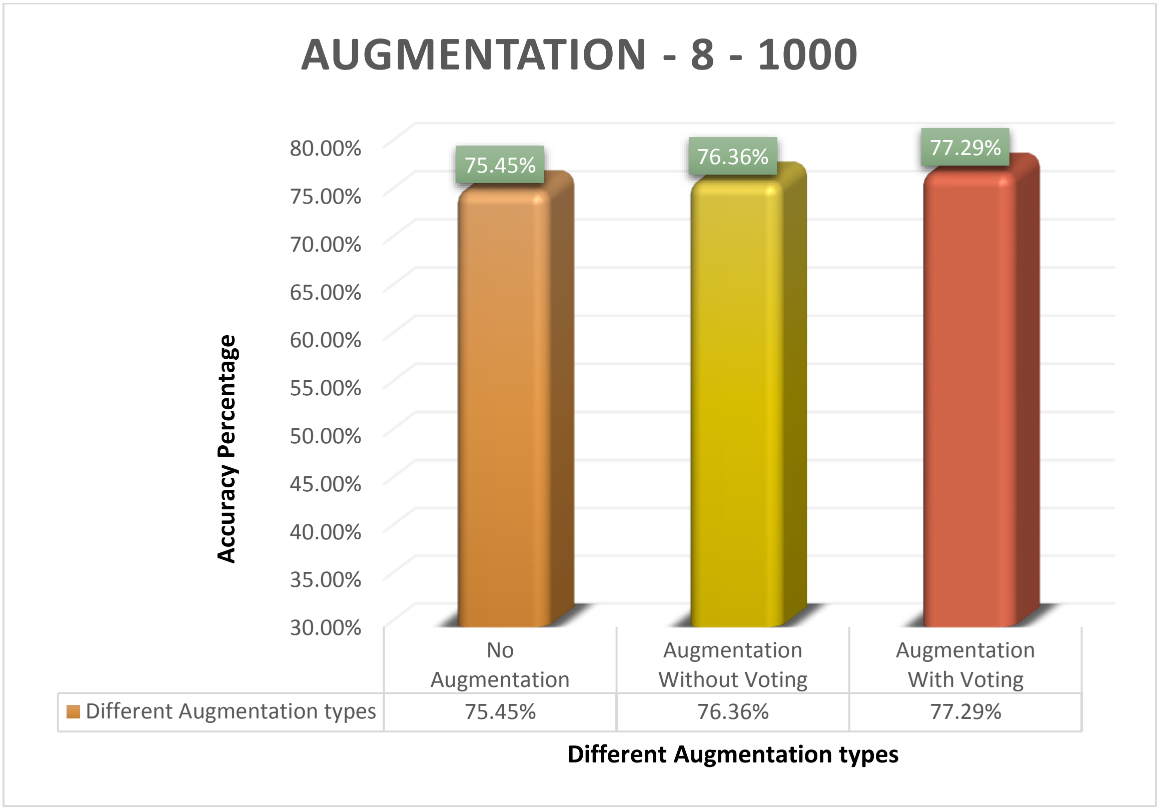
	6	8	10	16	8-6 .	8-10.	6-8-16.
▨	75.60%	75.45%	74.87%	72.16%	77.40%	76.01%	76.60%

Figure 23: Result of concatenating

6.3 Data Augmentations

The third experiment concerned augmenting the data. Because data augmentation requires so much memory, I did this experiment with receptive field size 8 and $k = 1{,}000$.

I ran the experiment with and without the voting technique; we can see that voting delivers better accuracy. Figure 24 shows the results of this experiment.



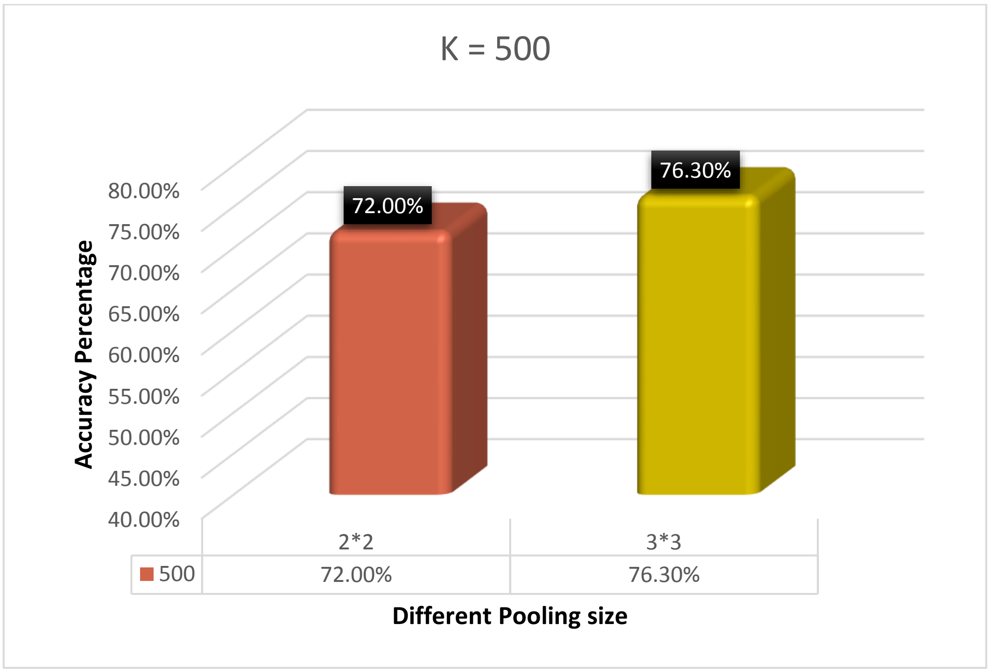
	No Augmentation	Augmentation Without Voting	Augmentation With Voting
🟧 Different Augmentation types	75.45%	76.36%	77.29%

Figure 24: Result of augmenting the data

6.4 Different Number of Regions for Pooling

The final test case was based on the pooling technique. I tried to test the effect of the region count used for pooling. It is obvious that more regions mean more information in each data point. As we can see in Figure 25 and Figure 26, increasing the number of regions increases the accuracy.

An interesting point is that when the number of centroids is increased, the effect becomes smaller. For instance, when K was equal to 500 the accuracy increased ~4% in going from 2×2 regions to 3×3 regions, but when K was equal to 1,000 the accuracy increased only ~1%, even when the regions were increased to 4×4 regions. This means that a higher number of centroids will keep more information, and we do not need to keep information from so many neighborhoods.



	2*2	3*3
🟥 500	72.00%	76.30%

Figure 25: Result of different region number in pooling for K=500

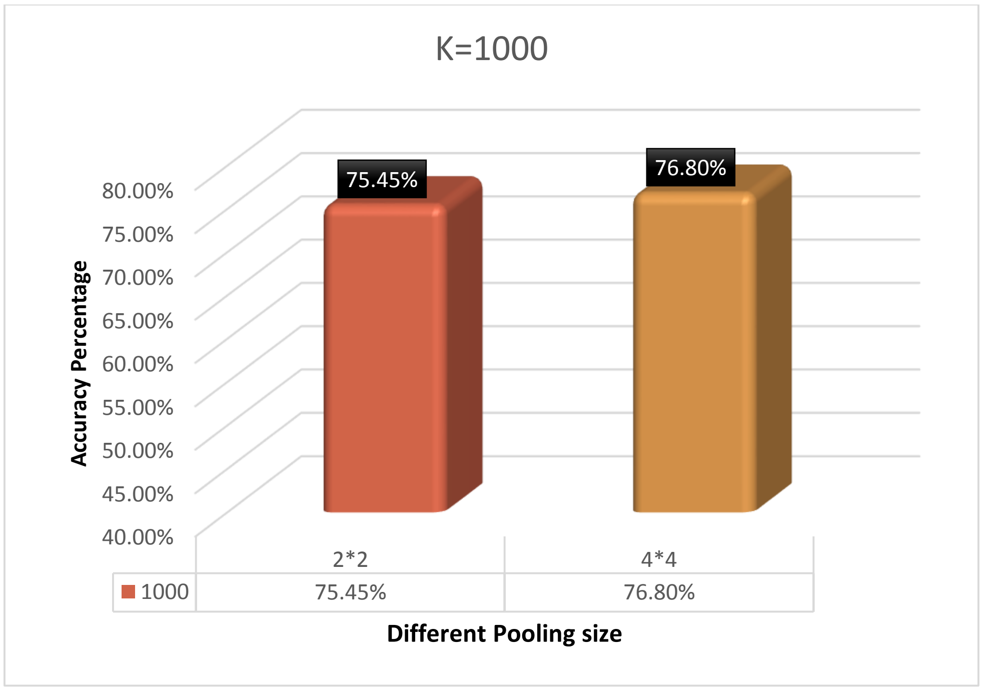


Figure 26: Result of different region number in pooling for K=1000

CHAPTER 7. CONCLUSION

7.1 Summary

In this thesis, I tried to use different approaches and build a model for automating image classification using K-means as an unsupervised feature-learning method. Feature learning by K-means is not at all a new idea. The popularity of deep learning shows that K-means has found a new effective usage. In this thesis, I explored some ideas such as patching technique, data augmentation, feature concatenations, feature convolution, pooling method, and multilevel feature learning to get the benefits of K-means for the task of image classification. Future work may focus on multilevel feature learning by K-means. Preliminary results with two levels of learning suggest multi-layer model can produce higher accuracy than a single layer model.

The following conclusions can be drawn from experimental results:

- K-means with a single layer can achieve accuracy that is comparable to single-layer deep learning techniques.
- Receptive field size does not seem to be a critical parameter for the CIFAR data set possibly because the small size (32 x 32) of the images in the data set.
- Data augmentation and voting improve accuracy compared to baseline.

- The concept of feature-vector concatenation should play a big role in classification; but for this data set it did not have a very significant effect on the accuracy.
- Pooling tries to reduce the dimensionality and extract the most important information from a set of neighborhoods. Decreasing the size of each block and increasing the number of regions increased the accuracy for this data set.

This summary of our results suggest K-means is a promising techniques for unsupervised feature learning and it could potentially do well if implemented in a multi-layer framework. We showed that K-means-based unsupervised feature learning will have less complexity and faster speed in comparison to many others methods.

7.2 Future Work

The research presented in this thesis seems to have raised more questions than it has answered. Several lines of research arising from this work should be pursued. Some of these questions can be more easily answered by using a powerful computer system with very large memory and good processing throughput. For example the part on data augmentation definitely needs more memory then I had available both for feature learning and for classification. I tried to use an Amazon EC2 with 60GB of memory, but even that much memory was insufficient for SVM classification and I was forced to use a subset of the data. Data augmentation together with the voting technique should have a better result than what was observed. This process in deep learning is very time consuming because the learning process by the neural network is very slow. In contrast, by using K-means the processing time significantly improves.

A second line of research that follows from Section 5.3 is to use my model and test it for more levels. I used it for only two states because of time limitations.

In addition to these two major research paths alternative clustering and feature encoding techniques can be explored.

LIST OF REFERENCES

LIST OF REFERENCES

[1] L. Deng and D. Yu, *Deep Learning: Methods and Applications*: Now Publishers, 2014.

[2] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: a review and new perspectives," *IEEE Trans Pattern Anal Mach Intell,* vol. 35, pp. 1798-828, Aug 2013.

[3] G. E. Hinton, "Learning multiple layers of representation," *Trends Cogn Sci,* vol. 11, pp. 428-34, Oct 2007.

[4] B. Julesz, "Textons, the elements of texture perception, and their interactions," *Nature,* vol. 290, pp. 91-7, Mar 12 1981.

[5] D. Svozil, V. Kvasnicka, and J. Pospichal, "Introduction to multi-layer feed-forward neural networks," *Chemometrics and intelligent laboratory systems,* vol. 39, pp. 43-62, 1997.

[6] C. Nebauer, "Evaluation of convolutional neural networks for visual recognition," *IEEE Trans Neural Netw,* vol. 9, pp. 685-96, 1998.

[7] C. Hang, Z. Yin, P. Spellman, and B. Parvin, "Stacked Predictive Sparse Coding for Classification of Distinct Regions in Tumor Histopathology," in *Computer Vision (ICCV), 2013 IEEE International Conference on*, 2013, pp. 169-176.

[8] M. D. Zeiler, D. Krishnan, G. W. Taylor, and R. Fergus, "Deconvolutional networks," in *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, 2010, pp. 2528-2535.

[9] M. D. Zeiler, G. W. Taylor, and R. Fergus, "Adaptive deconvolutional networks for mid and high level feature learning," in *Computer Vision (ICCV), 2011 IEEE International Conference on*, 2011, pp. 2018-2025.

[10] J. Masci, U. Meier, D. Cireşan, and J. Schmidhuber, "Stacked convolutional auto-encoders for hierarchical feature extraction," in *Artificial Neural Networks and Machine Learning–ICANN 2011*, ed: Springer, 2011, pp. 52-59.

[11] R. Salakhutdinov and G. E. Hinton, "Deep boltzmann machines," in *International Conference on Artificial Intelligence and Statistics*, 2009, pp. 448-455.

[12] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *The Journal of Machine Learning Research,* vol. 11, pp. 3371-3408, 2010.

[13] L. Bo, X. Ren, and D. Fox, "Unsupervised feature learning for RGB-D based object recognition," in *Experimental Robotics*, 2013, pp. 387-402.

[14] W. Wang, B. C. Ooi, X. Yang, D. Zhang, and Y. Zhuang, "Effective multi-modal retrieval based on stacked auto-encoders," 2014.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097-1105.

[16] D. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, 2012, pp. 3642-3649.

[17] W. Zhao, R. Chellappa, P. J. Phillips, and A. Rosenfeld, "Face recognition: A literature survey," *Acm Computing Surveys (CSUR),* vol. 35, pp. 399-458, 2003.

[18] J. MacQueen, "Some methods for classification and analysis of multivariate observations," 1967.

[19] A. Coates, A. Y. Ng, and H. Lee, "An analysis of single-layer networks in unsupervised feature learning," in *International Conference on Artificial Intelligence and Statistics*, 2011, pp. 215-223.

[20] C. Elkan, "Using the triangle inequality to accelerate k-means," in *ICML*, 2013.

[21] K. K. Wang, "Image Classification with Pyramid Representation and Rotated Data Augmentation on Torch 7."

[22] X. Lu, B. Zheng, A. Velivelli, and C. Zhai, "Enhancing Text Categorization with Semantic-enriched Representation and Training Data Augmentation," *Journal of the American Medical Informatics Association : JAMIA,* vol. 13, pp. 526-535, Oct 2006.

[23] D. Zhang, *Advances in machine learning applications in software engineering*: IGI Global, 2006.

[24] T. Damoulas and M. A. Girolami, "Combining feature spaces for classification," *Pattern Recognition,* vol. 42, pp. 2671-2683, 2009.

[25] TorontoUniv., "The CIFAR-10 dataset", V. N. Alex Krizhevsky, Geoffrey Hinton. Toronto University: http://www.cs.toronto.edu/~kriz/cifar.html (Accessed: Dec 10, 2014)

[26] "Stanford Univ.", *Whitening*. Available: http://ufldl.stanford.edu/wiki/index.php/Whitening (Accessed: 16 March 2015)

[27] Dustin Stansbury, *The Statistical Whitening Transform*. Available: https://theclevermachine.wordpress.com/tag/principal-components-analysis-pca/ (Accessed: 10 April 2015)

[28] A. J. Bell and T. J. Sejnowski, "Edges are the" independent components" of natural scenes."

[29] A. Coates and A. Y. Ng, "Learning feature representations with k-means," in *Neural Networks: Tricks of the Trade*, ed: Springer, 2012, pp. 561-580.

[30] I. Guyon and A. Elisseeff, "An introduction to feature extraction," in *Feature Extraction*, ed: Springer, 2006, pp. 1-25.

[31] Y. LeCun, "LeNet-5, convolutional neural networks."

[32] J. Ngiam, Z. Chen, D. Chia, P. W. Koh, Q. V. Le, and A. Y. Ng, "Tiled convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2010, pp. 1279-1287.

[33] D. Scherer, A. Müller, and S. Behnke, "Evaluation of pooling operations in convolutional architectures for object recognition," in *Artificial Neural Networks–ICANN 2010*, ed: Springer, 2010, pp. 92-101.

[34] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, pp. 273-297, 1995.

[35] Dell, *Support Vector Machines*. Available: http://www.statsoft.com/textbook/support-vector-machines (Accessed: 20 April 2015)

[36] B. Schölkopf, A. J. Smola, R. C. Williamson, and P. L. Bartlett, "New support vector algorithms," *Neural computation*, vol. 12, pp. 1207-1245, 2000.

[37] O. L. Mangasarian and D. R. Musicant, "Robust linear and support vector regression," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 22, pp. 950-955, 2000.

[38] C.-W. Hsu and C.-J. Lin, "A comparison of methods for multiclass support vector machines," *Neural Networks, IEEE Transactions on*, vol. 13, pp. 415-425, 2002.

[39] Hossein Karimy, "K-means which patching and normalization," vol. https://bitbucket.org/H_Soshiant/kmeans, ed, 2015.

[40] Hossein Karimy, "Feature Extraction (Convolution+Pooling)," vol. https://bitbucket.org/H_Soshiant/feature-extraction, ed, 2015.

[41] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A library for large linear classification," *The Journal of Machine Learning Research*, vol. 9, pp. 1871-1874, 2008.

APPENDICES

Appendix A. K-Means C++ Software

This software was written in C++ on Visual Studio 2013 with two main parts: Normalization and K-means.

The software was built based on the open multi-processing (OpenMP) technique, which is a multi-platform API used for multi-processing programming in C, C++, and other programming languages. The OpenMP is a very fast and easy to use CPU-based API and is integrated with Visual Studio. The source code is located on Bitbucket at https://bitbucket.org/H_Soshiant/kmeans (Accessed: 20 May 2015)

In the normalization part, the given data set can be broken into patches and Z-score normalization applied. In the K-means part, the given input data or the output of the normalization part can be imported and applied through the K-means process. This process may start with predefined centroids as initialization, or the software can calculate them by an internal procedure. There are several input parameters and four main files as output, which are described as follows:

Table 3: K-means parameters

Inputs:

Variable	Value	Description
-i	Input File	The path of the input data.
-it	File format	Can take the following values: • 1 : Full space seperated file • 2 : Binary file: row by row full binary file
-t	File data type	Can take the following values: • 0 : boolean • 1 : unsigned integer 8 bits • 2 : unsigned integer 16 bits • 3 : float • 4 : double
-ic	Predefined centers file	Path to the predefined centers for initialization

Table 3, continued

-r	Record count	Number of records in the given data
-d	Image side size	Size of the image for each side. Picture should be square
-k	K value	Number of needed clusters
-ub	-	Using this variable will force software to use buffering, which needs much more memory
-nq	-	Used to work in silent mode without asking any questions
-b	Batch count	Number of batches for spliting the input data
-bn	Batch number	Starting batch number for resuming

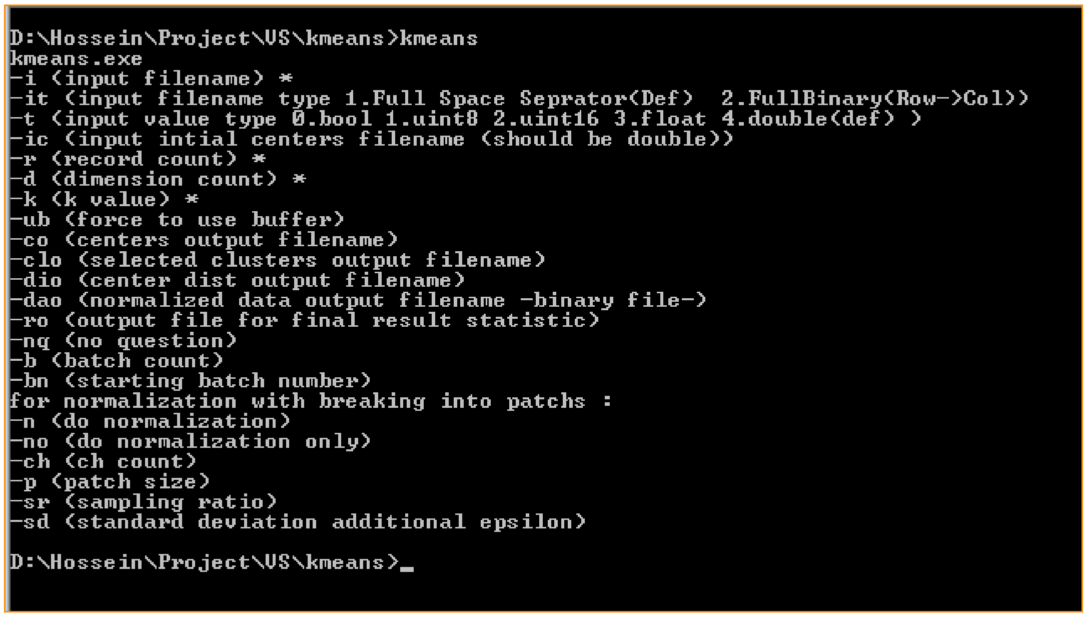
Normalization Parameters

-n	-	Tell to do normalization
-no	-	Using this variable causes the software to do only normalization
-ch	Channel count	Number of input channels for images e.g ., 3 for RGB
-p	Patch size	Size of receptive field
-sd	Epsilon value	Used for adding in standard deviation calculation

Output Parameters

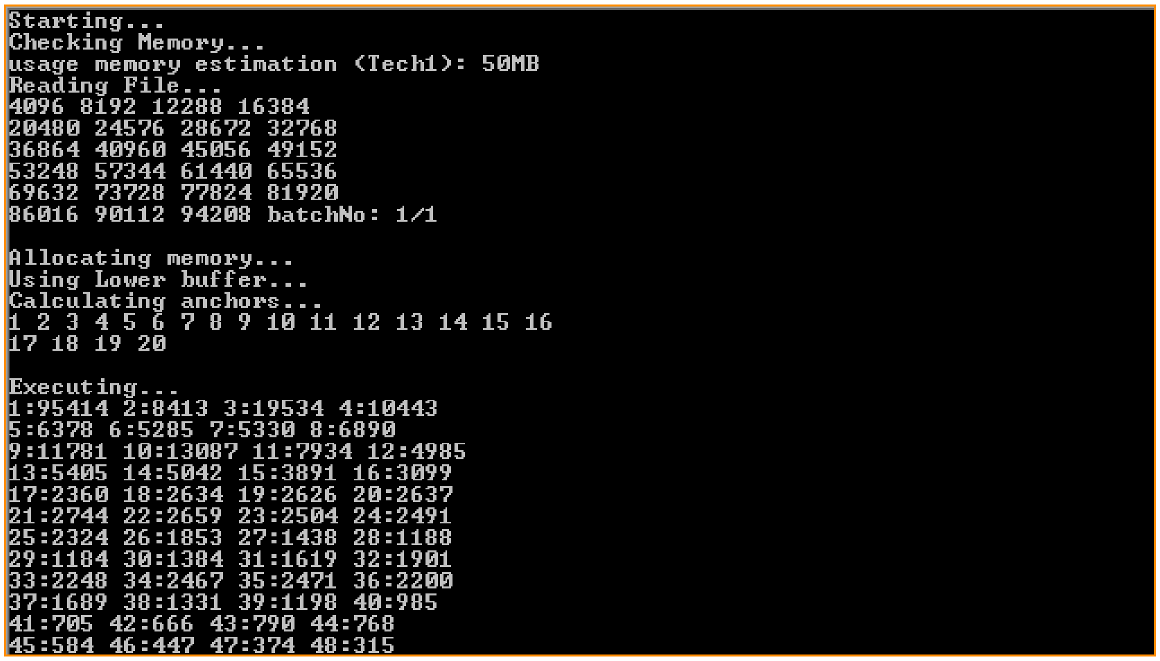
-co	Centers filename	Filename for writing centers
-clo	Clusters filename	Filename for writing selected clusters for each point
-dio	Distances filename	File name for writing distances between each point and related centroid
-dao	Data filename	Filename for writing normalized data/patches

The following figures show some snapshots of the software:



```
D:\Hossein\Project\VS\kmeans>kmeans
kmeans.exe
-i (input filename) *
-it (input filename type 1.Full Space Seprator(Def)  2.FullBinary(Row->Col))
-t (input value type 0.bool 1.uint8 2.uint16 3.float 4.double(def) )
-ic (input intial centers filename (should be double))
-r (record count) *
-d (dimension count) *
-k (k value) *
-ub (force to use buffer)
-co (centers output filename)
-clo (selected clusters output filename)
-dio (center dist output filename)
-dao (normalized data output filename -binary file-)
-ro (output file for final result statistic)
-nq (no question)
-b (batch count)
-bn (starting batch number)
for normalization with breaking into patchs :
-n (do normalization)
-no (do normalization only)
-ch (ch count)
-p (patch size)
-sr (sampling ratio)
-sd (standard deviation additional epsilon)

D:\Hossein\Project\VS\kmeans>_
```

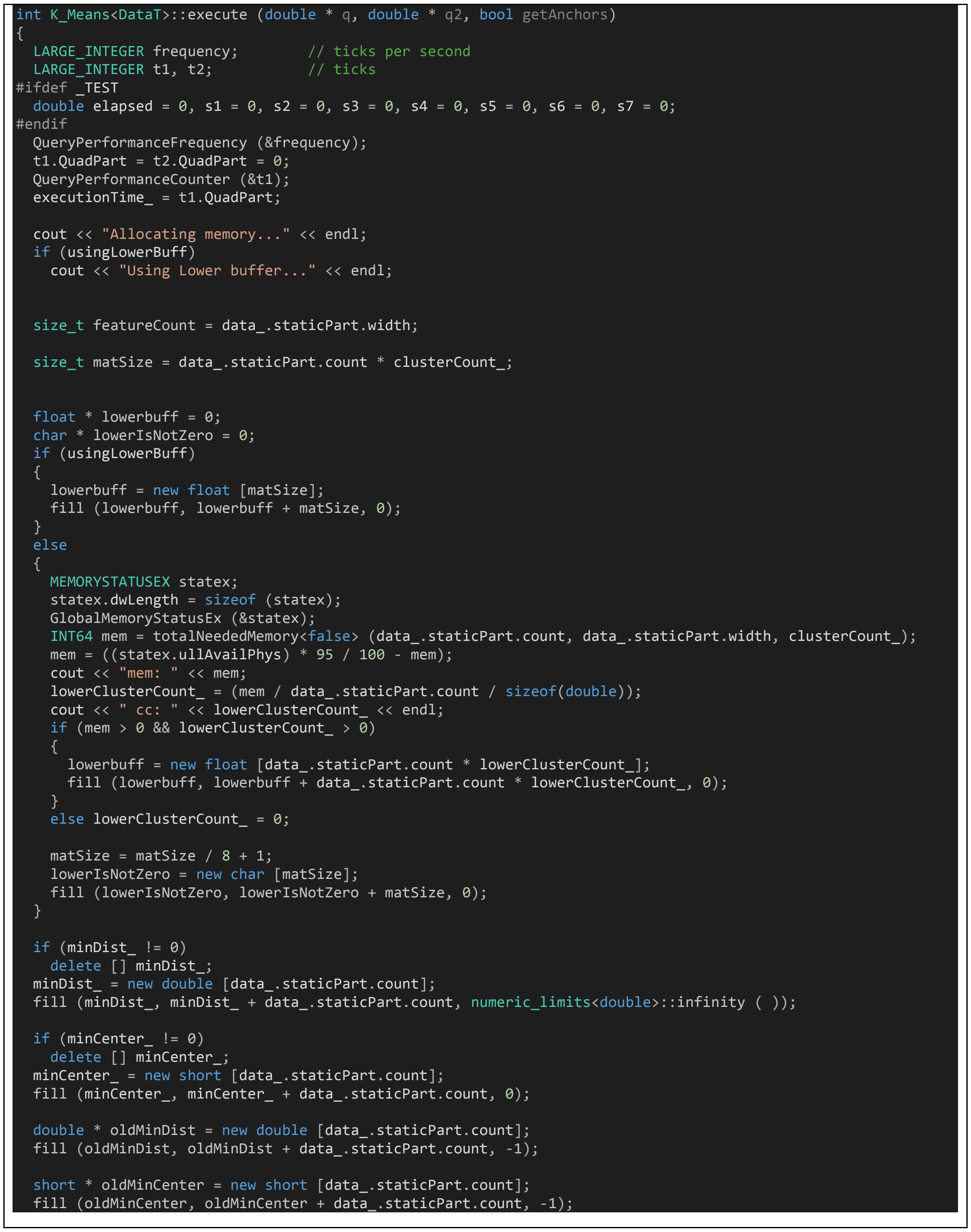
Figure 27 : List of parameters in k-means software



```
Starting...
Checking Memory...
usage memory estimation (Tech1): 50MB
Reading File...
4096 8192 12288 16384
20480 24576 28672 32768
36864 40960 45056 49152
53248 57344 61440 65536
69632 73728 77824 81920
86016 90112 94208 batchNo: 1/1

Allocating memory...
Using Lower buffer...
Calculating anchors...
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20

Executing...
1:95414 2:8413 3:19534 4:10443
5:6378 6:5285 7:5330 8:6890
9:11781 10:13087 11:7934 12:4985
13:5405 14:5042 15:3891 16:3099
17:2360 18:2634 19:2626 20:2637
21:2744 22:2659 23:2504 24:2491
25:2324 26:1853 27:1438 28:1188
29:1184 30:1384 31:1619 32:1901
33:2248 34:2467 35:2471 36:2200
37:1689 38:1331 39:1198 40:985
41:705 42:666 43:790 44:768
45:584 46:447 47:374 48:315
```

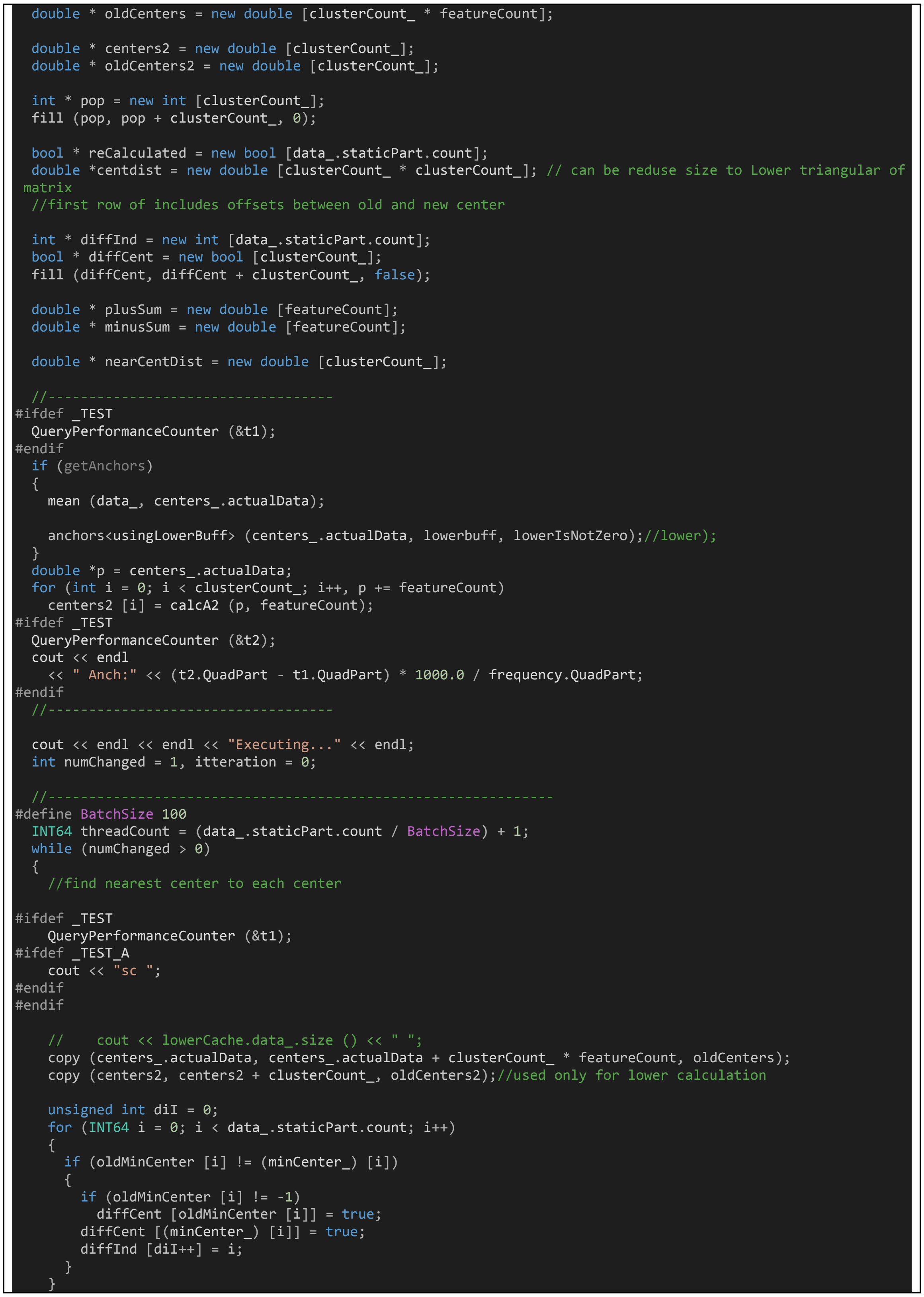
Figure 28 :Running k-means

And the last part is the source code of the main method in the software that is for calculating K-means:



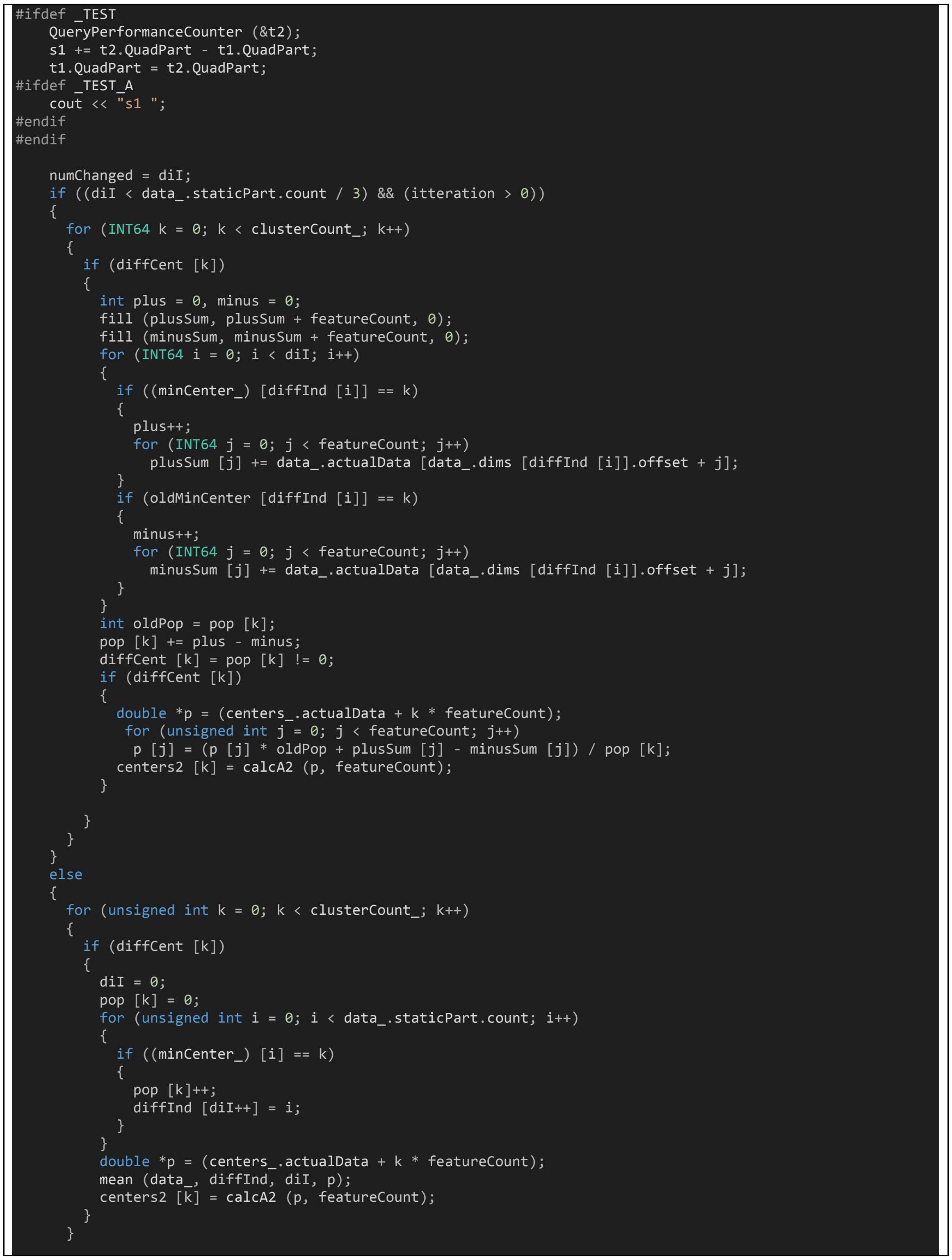
```cpp
int K_Means<DataT>::execute (double * q, double * q2, bool getAnchors)
{
  LARGE_INTEGER frequency;        // ticks per second
  LARGE_INTEGER t1, t2;           // ticks
#ifdef _TEST
  double elapsed = 0, s1 = 0, s2 = 0, s3 = 0, s4 = 0, s5 = 0, s6 = 0, s7 = 0;
#endif
  QueryPerformanceFrequency (&frequency);
  t1.QuadPart = t2.QuadPart = 0;
  QueryPerformanceCounter (&t1);
  executionTime_ = t1.QuadPart;

  cout << "Allocating memory..." << endl;
  if (usingLowerBuff)
    cout << "Using Lower buffer..." << endl;


  size_t featureCount = data_.staticPart.width;

  size_t matSize = data_.staticPart.count * clusterCount_;


  float * lowerbuff = 0;
  char * lowerIsNotZero = 0;
  if (usingLowerBuff)
  {
    lowerbuff = new float [matSize];
    fill (lowerbuff, lowerbuff + matSize, 0);
  }
  else
  {
    MEMORYSTATUSEX statex;
    statex.dwLength = sizeof (statex);
    GlobalMemoryStatusEx (&statex);
    INT64 mem = totalNeededMemory<false> (data_.staticPart.count, data_.staticPart.width, clusterCount_);
    mem = ((statex.ullAvailPhys) * 95 / 100 - mem);
    cout << "mem: " << mem;
    lowerClusterCount_ = (mem / data_.staticPart.count / sizeof(double));
    cout << " cc: " << lowerClusterCount_ << endl;
    if (mem > 0 && lowerClusterCount_ > 0)
    {
      lowerbuff = new float [data_.staticPart.count * lowerClusterCount_];
      fill (lowerbuff, lowerbuff + data_.staticPart.count * lowerClusterCount_, 0);
    }
    else lowerClusterCount_ = 0;

    matSize = matSize / 8 + 1;
    lowerIsNotZero = new char [matSize];
    fill (lowerIsNotZero, lowerIsNotZero + matSize, 0);
  }

  if (minDist_ != 0)
    delete [] minDist_;
  minDist_ = new double [data_.staticPart.count];
  fill (minDist_, minDist_ + data_.staticPart.count, numeric_limits<double>::infinity ( ));

  if (minCenter_ != 0)
    delete [] minCenter_;
  minCenter_ = new short [data_.staticPart.count];
  fill (minCenter_, minCenter_ + data_.staticPart.count, 0);

  double * oldMinDist = new double [data_.staticPart.count];
  fill (oldMinDist, oldMinDist + data_.staticPart.count, -1);

  short * oldMinCenter = new short [data_.staticPart.count];
  fill (oldMinCenter, oldMinCenter + data_.staticPart.count, -1);
```

Figure 29 : Major method of k-means calculation



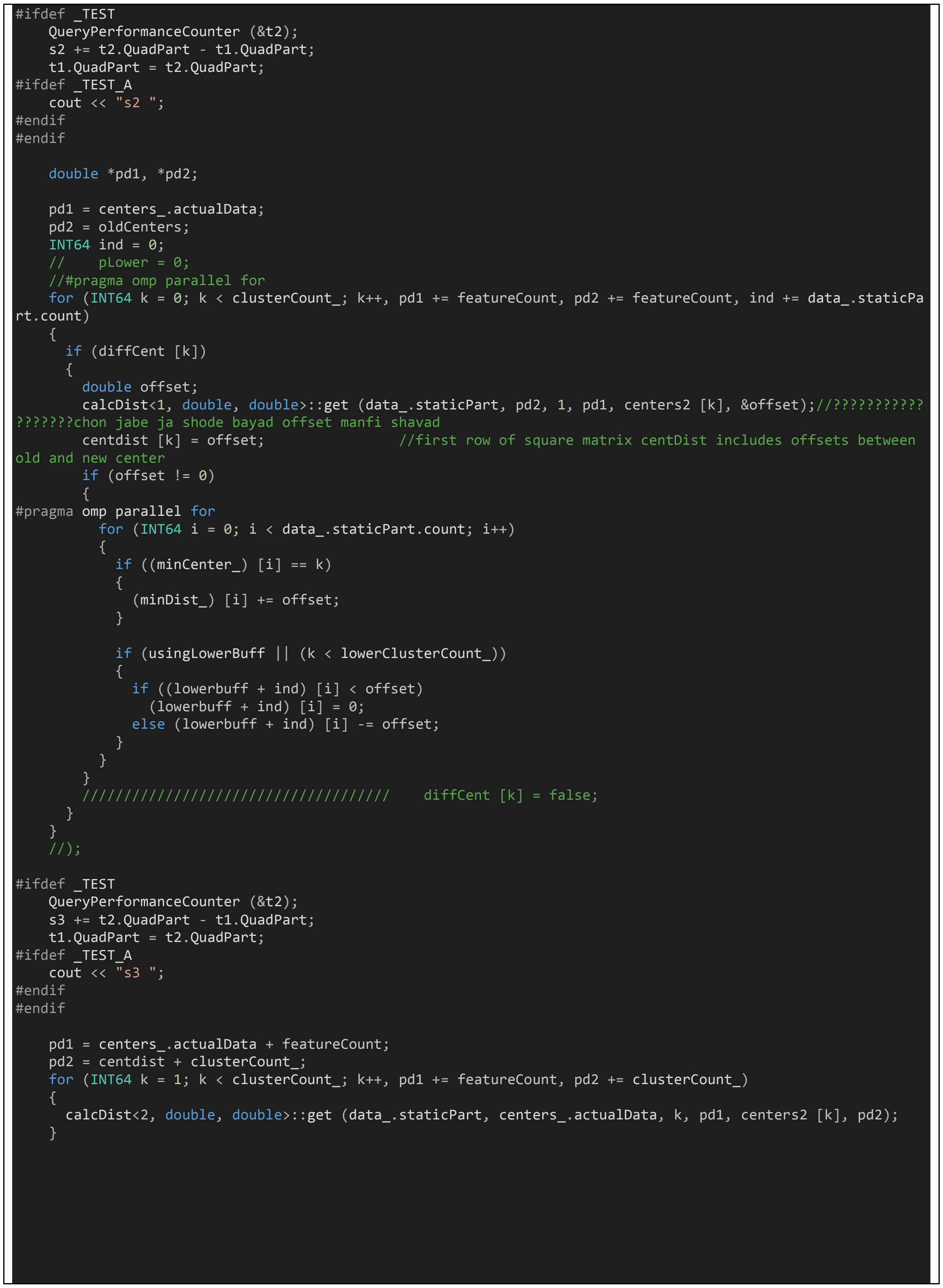
```
  double * oldCenters = new double [clusterCount_ * featureCount];

  double * centers2 = new double [clusterCount_];
  double * oldCenters2 = new double [clusterCount_];

  int * pop = new int [clusterCount_];
  fill (pop, pop + clusterCount_, 0);

  bool * reCalculated = new bool [data_.staticPart.count];
  double *centdist = new double [clusterCount_ * clusterCount_]; // can be reduse size to Lower triangular of
 matrix
  //first row of includes offsets between old and new center

  int * diffInd = new int [data_.staticPart.count];
  bool * diffCent = new bool [clusterCount_];
  fill (diffCent, diffCent + clusterCount_, false);

  double * plusSum = new double [featureCount];
  double * minusSum = new double [featureCount];

  double * nearCentDist = new double [clusterCount_];

  //-----------------------------------
#ifdef _TEST
  QueryPerformanceCounter (&t1);
#endif
  if (getAnchors)
  {
    mean (data_, centers_.actualData);

    anchors<usingLowerBuff> (centers_.actualData, lowerbuff, lowerIsNotZero);//lower);
  }
  double *p = centers_.actualData;
  for (int i = 0; i < clusterCount_; i++, p += featureCount)
    centers2 [i] = calcA2 (p, featureCount);
#ifdef _TEST
  QueryPerformanceCounter (&t2);
  cout << endl
    << " Anch:" << (t2.QuadPart - t1.QuadPart) * 1000.0 / frequency.QuadPart;
#endif
  //-----------------------------------

  cout << endl << endl << "Executing..." << endl;
  int numChanged = 1, itteration = 0;

  //-------------------------------------------------------------
#define BatchSize 100
  INT64 threadCount = (data_.staticPart.count / BatchSize) + 1;
  while (numChanged > 0)
  {
    //find nearest center to each center

#ifdef _TEST
    QueryPerformanceCounter (&t1);
#ifdef _TEST_A
    cout << "sc ";
#endif
#endif

    //    cout << lowerCache.data_.size () << " ";
    copy (centers_.actualData, centers_.actualData + clusterCount_ * featureCount, oldCenters);
    copy (centers2, centers2 + clusterCount_, oldCenters2);//used only for lower calculation

    unsigned int diI = 0;
    for (INT64 i = 0; i < data_.staticPart.count; i++)
    {
      if (oldMinCenter [i] != (minCenter_) [i])
      {
        if (oldMinCenter [i] != -1)
          diffCent [oldMinCenter [i]] = true;
        diffCent [(minCenter_) [i]] = true;
        diffInd [diI++] = i;
      }
    }
```

Figure 29, continued.



```
#ifdef _TEST
    QueryPerformanceCounter (&t2);
    s1 += t2.QuadPart - t1.QuadPart;
    t1.QuadPart = t2.QuadPart;
#ifdef _TEST_A
    cout << "s1 ";
#endif
#endif

    numChanged = diI;
    if ((diI < data_.staticPart.count / 3) && (itteration > 0))
    {
      for (INT64 k = 0; k < clusterCount_; k++)
      {
        if (diffCent [k])
        {
          int plus = 0, minus = 0;
          fill (plusSum, plusSum + featureCount, 0);
          fill (minusSum, minusSum + featureCount, 0);
          for (INT64 i = 0; i < diI; i++)
          {
            if ((minCenter_) [diffInd [i]] == k)
            {
              plus++;
              for (INT64 j = 0; j < featureCount; j++)
                plusSum [j] += data_.actualData [data_.dims [diffInd [i]].offset + j];
            }
            if (oldMinCenter [diffInd [i]] == k)
            {
              minus++;
              for (INT64 j = 0; j < featureCount; j++)
                minusSum [j] += data_.actualData [data_.dims [diffInd [i]].offset + j];
            }
          }
          int oldPop = pop [k];
          pop [k] += plus - minus;
          diffCent [k] = pop [k] != 0;
          if (diffCent [k])
          {
            double *p = (centers_.actualData + k * featureCount);
             for (unsigned int j = 0; j < featureCount; j++)
              p [j] = (p [j] * oldPop + plusSum [j] - minusSum [j]) / pop [k];
            centers2 [k] = calcA2 (p, featureCount);
          }

        }
      }
    }
    else
    {
      for (unsigned int k = 0; k < clusterCount_; k++)
      {
        if (diffCent [k])
        {
          diI = 0;
          pop [k] = 0;
          for (unsigned int i = 0; i < data_.staticPart.count; i++)
          {
            if ((minCenter_) [i] == k)
            {
              pop [k]++;
              diffInd [diI++] = i;
            }
          }
          double *p = (centers_.actualData + k * featureCount);
          mean (data_, diffInd, diI, p);
          centers2 [k] = calcA2 (p, featureCount);
        }
      }
```

Figure 29, continued.

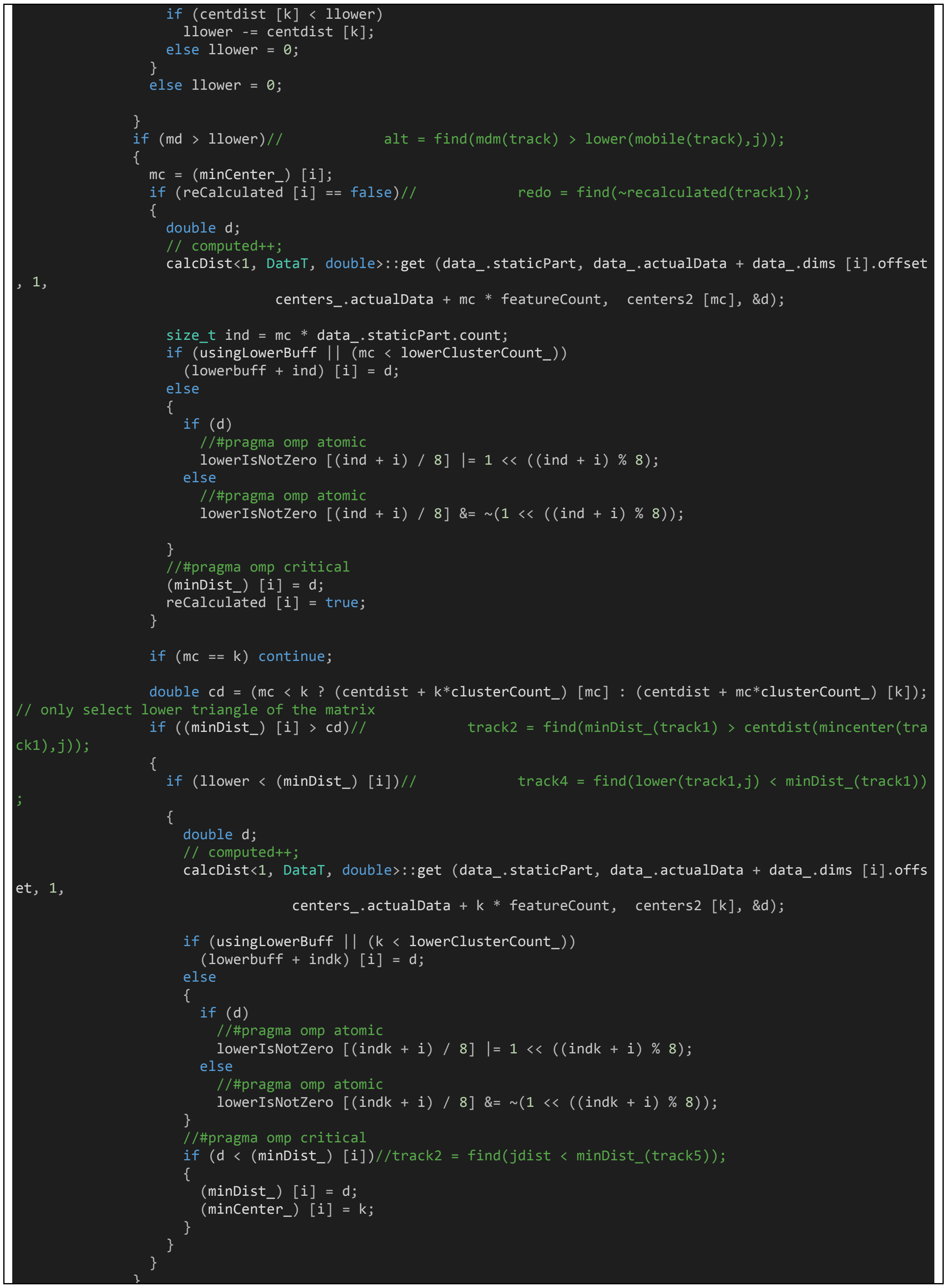


```
#ifdef _TEST
    QueryPerformanceCounter (&t2);
    s2 += t2.QuadPart - t1.QuadPart;
    t1.QuadPart = t2.QuadPart;
#ifdef _TEST_A
    cout << "s2 ";
#endif
#endif

    double *pd1, *pd2;

    pd1 = centers_.actualData;
    pd2 = oldCenters;
    INT64 ind = 0;
    //    pLower = 0;
    //#pragma omp parallel for
    for (INT64 k = 0; k < clusterCount_; k++, pd1 += featureCount, pd2 += featureCount, ind += data_.staticPa
rt.count)
    {
      if (diffCent [k])
      {
        double offset;
        calcDist<1, double, double>::get (data_.staticPart, pd2, 1, pd1, centers2 [k], &offset);//???????????
???????chon jabe ja shode bayad offset manfi shavad
        centdist [k] = offset;                //first row of square matrix centDist includes offsets between
old and new center
        if (offset != 0)
        {
#pragma omp parallel for
          for (INT64 i = 0; i < data_.staticPart.count; i++)
          {
            if ((minCenter_) [i] == k)
            {
              (minDist_) [i] += offset;
            }

            if (usingLowerBuff || (k < lowerClusterCount_))
            {
              if ((lowerbuff + ind) [i] < offset)
                (lowerbuff + ind) [i] = 0;
              else (lowerbuff + ind) [i] -= offset;
            }
          }
        }
        /////////////////////////////////////    diffCent [k] = false;
      }
    }
    //);

#ifdef _TEST
    QueryPerformanceCounter (&t2);
    s3 += t2.QuadPart - t1.QuadPart;
    t1.QuadPart = t2.QuadPart;
#ifdef _TEST_A
    cout << "s3 ";
#endif
#endif

    pd1 = centers_.actualData + featureCount;
    pd2 = centdist + clusterCount_;
    for (INT64 k = 1; k < clusterCount_; k++, pd1 += featureCount, pd2 += clusterCount_)
    {
      calcDist<2, double, double>::get (data_.staticPart, centers_.actualData, k, pd1, centers2 [k], pd2);
    }
```

Figure 29, continued.

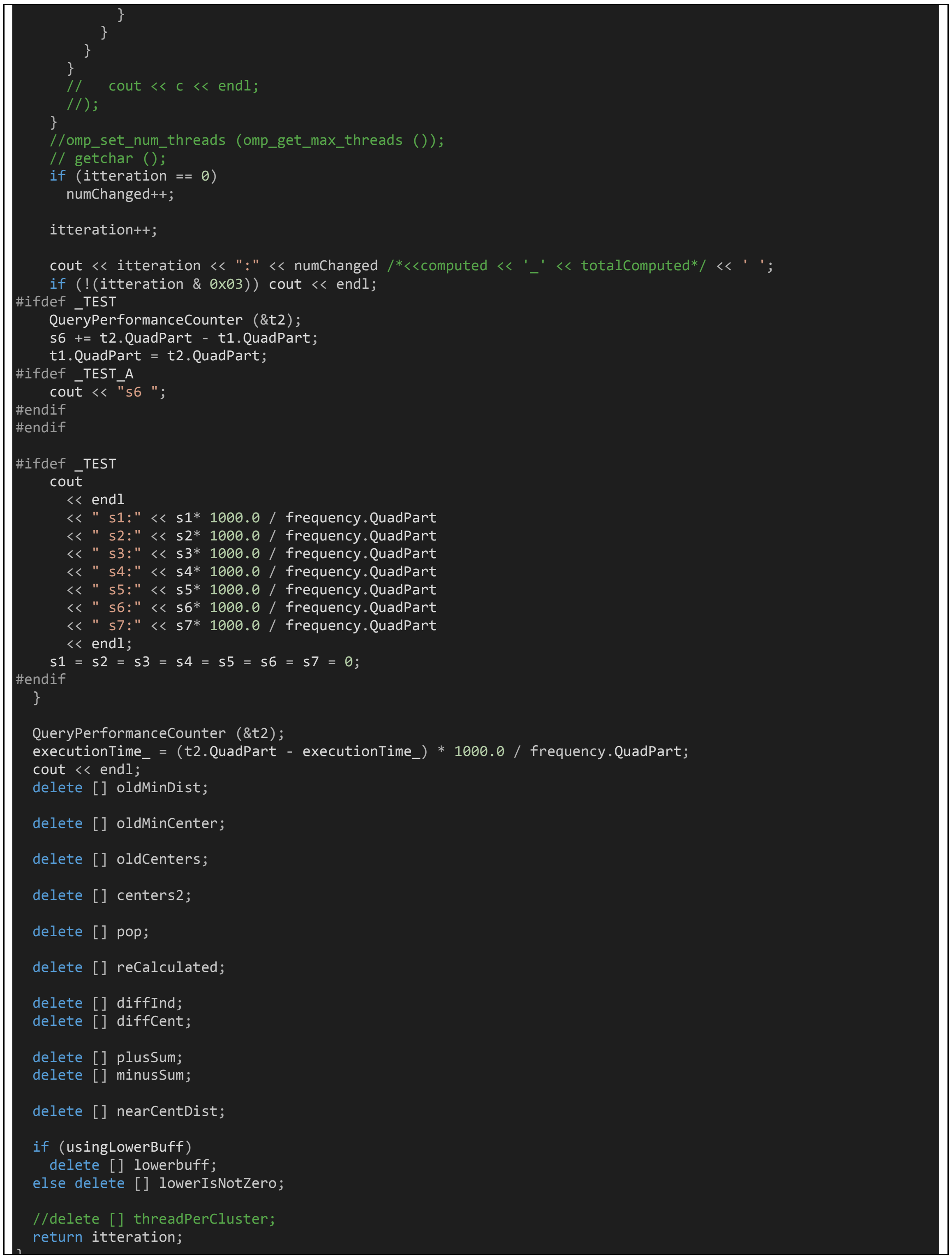
```
#ifdef _TEST
    QueryPerformanceCounter (&t2);
    s4 += t2.QuadPart - t1.QuadPart;
    t1.QuadPart = t2.QuadPart;
#ifdef _TEST_A
    cout << "s4 ";
#endif
#endif

    //Prepare for next itteration

    for (unsigned int k = 0; k < clusterCount_; k++)
    {
      pd2 = centdist + k*clusterCount_;
      nearCentDist [k] = *pd2;
      for (unsigned int k2 = 0; k2 < clusterCount_; k2++, pd2 += k < k2 ? clusterCount_ : 1)
      {
        if (nearCentDist [k] > *pd2)
          nearCentDist [k] = *pd2;
      }
    }
#ifdef _TEST
    QueryPerformanceCounter (&t2);
    s5 += t2.QuadPart - t1.QuadPart;
    t1.QuadPart = t2.QuadPart;
#ifdef _TEST_A
    cout << "s5 ";
#endif
#endif

    copy ((minCenter_), (minCenter_) +data_.staticPart.count, oldMinCenter);
    copy ((minDist_), (minDist_) +data_.staticPart.count, oldMinDist);
    fill (reCalculated, reCalculated + data_.staticPart.count, false);

    //---------------------------------------------------------------------------
    for (INT64 k = 0; k < clusterCount_; k++)
    {
      size_t indk = k * data_.staticPart.count;
      //      if (threadPerCluster [k] == 0) continue;

      //omp_set_num_threads (threadPerCluster [k]);

      // size_t c = 0;
#pragma omp parallel for schedule(static)
      for (INT64 ii = 0; ii < threadCount; ii++)
      {
        INT64 m = ii * BatchSize;
        INT64 i = m;
        for (m += BatchSize; (i < m) && (i <data_.staticPart.count); i++)
        {
          unsigned int mc = oldMinCenter [i];
          double md = oldMinDist [i];// +(mc == k ? offset : 0);
          if (md > nearCentDist [mc])
          {
            double cd = (mc < k ? (centdist + k*clusterCount_) [mc] : (centdist + mc*clusterCount_) [k]);
            if (md > cd)//            track = find(mdm > centdist(mcm,j));
            {
              // c++;
              if (((minCenter_) [i] == k) && reCalculated [i]) continue;
              double llower;

              //#pragma omp critical
              if (usingLowerBuff || (k < lowerClusterCount_))
                llower = (lowerbuff + indk) [i];
              else
              {
                if (k == 0 || lowerIsNotZero [(indk + i) / 8] & 1 << ((indk + i) % 8))
                {
                  calcDist<1, DataT, double>::get (data_.staticPart, data_.actualData + data_.dims [i].offset
, 1,
                                                  oldCenters + k * featureCount, oldCenters2 [k]/* calcA2 (p,
 featureCount)*/, &llower);
```

Figure 29, continued.



```
                  if (centdist [k] < llower)
                    llower -= centdist [k];
                  else llower = 0;
                }
                else llower = 0;

              }
              if (md > llower)//            alt = find(mdm(track) > lower(mobile(track),j));
              {
                mc = (minCenter_) [i];
                if (reCalculated [i] == false)//            redo = find(~recalculated(track1));
                {
                  double d;
                  // computed++;
                  calcDist<1, DataT, double>::get (data_.staticPart, data_.actualData + data_.dims [i].offset
, 1,
                               centers_.actualData + mc * featureCount,  centers2 [mc], &d);

                  size_t ind = mc * data_.staticPart.count;
                  if (usingLowerBuff || (mc < lowerClusterCount_))
                    (lowerbuff + ind) [i] = d;
                  else
                  {
                    if (d)
                      //#pragma omp atomic
                      lowerIsNotZero [(ind + i) / 8] |= 1 << ((ind + i) % 8);
                    else
                      //#pragma omp atomic
                      lowerIsNotZero [(ind + i) / 8] &= ~(1 << ((ind + i) % 8));

                  }
                  //#pragma omp critical
                  (minDist_) [i] = d;
                  reCalculated [i] = true;
                }

                if (mc == k) continue;

                double cd = (mc < k ? (centdist + k*clusterCount_) [mc] : (centdist + mc*clusterCount_) [k]);
// only select lower triangle of the matrix
                if ((minDist_) [i] > cd)//            track2 = find(minDist_(track1) > centdist(mincenter(tra
ck1),j));
                {
                  if (llower < (minDist_) [i])//            track4 = find(lower(track1,j) < minDist_(track1))
;
                  {
                    double d;
                    // computed++;
                    calcDist<1, DataT, double>::get (data_.staticPart, data_.actualData + data_.dims [i].offs
et, 1,
                                 centers_.actualData + k * featureCount,  centers2 [k], &d);

                    if (usingLowerBuff || (k < lowerClusterCount_))
                      (lowerbuff + indk) [i] = d;
                    else
                    {
                      if (d)
                        //#pragma omp atomic
                        lowerIsNotZero [(indk + i) / 8] |= 1 << ((indk + i) % 8);
                      else
                        //#pragma omp atomic
                        lowerIsNotZero [(indk + i) / 8] &= ~(1 << ((indk + i) % 8));
                    }
                    //#pragma omp critical
                    if (d < (minDist_) [i])//track2 = find(jdist < minDist_(track5));
                    {
                      (minDist_) [i] = d;
                      (minCenter_) [i] = k;
                    }
                  }
                }
              }
```

Figure 29, continued.



```
      }
     }
    }
   }
   //  cout << c << endl;
   //);
  }
  //omp_set_num_threads (omp_get_max_threads ());
  // getchar ();
  if (itteration == 0)
   numChanged++;

  itteration++;

  cout << itteration << ":" << numChanged /*<<computed << '_' << totalComputed*/ << ' ';
  if (!(itteration & 0x03)) cout << endl;
#ifdef _TEST
  QueryPerformanceCounter (&t2);
  s6 += t2.QuadPart - t1.QuadPart;
  t1.QuadPart = t2.QuadPart;
#ifdef _TEST_A
  cout << "s6 ";
#endif
#endif

#ifdef _TEST
  cout
   << endl
   << " s1:" << s1* 1000.0 / frequency.QuadPart
   << " s2:" << s2* 1000.0 / frequency.QuadPart
   << " s3:" << s3* 1000.0 / frequency.QuadPart
   << " s4:" << s4* 1000.0 / frequency.QuadPart
   << " s5:" << s5* 1000.0 / frequency.QuadPart
   << " s6:" << s6* 1000.0 / frequency.QuadPart
   << " s7:" << s7* 1000.0 / frequency.QuadPart
   << endl;
  s1 = s2 = s3 = s4 = s5 = s6 = s7 = 0;
#endif
 }

 QueryPerformanceCounter (&t2);
 executionTime_ = (t2.QuadPart - executionTime_) * 1000.0 / frequency.QuadPart;
 cout << endl;
 delete [] oldMinDist;

 delete [] oldMinCenter;

 delete [] oldCenters;

 delete [] centers2;

 delete [] pop;

 delete [] reCalculated;

 delete [] diffInd;
 delete [] diffCent;

 delete [] plusSum;
 delete [] minusSum;

 delete [] nearCentDist;

 if (usingLowerBuff)
  delete [] lowerbuff;
 else delete [] lowerIsNotZero;

 //delete [] threadPerCluster;
 return itteration;
}
```

Figure 29, continued.

Appendix B. Feature-Extraction C++ Software

This software was designed and implemented for calculating the feature vectors and is GPU (graphical processing unit)-based software. I used the Nvidia CUDA programming API for this software and wrote it using C++ in Visual Studio 2013.

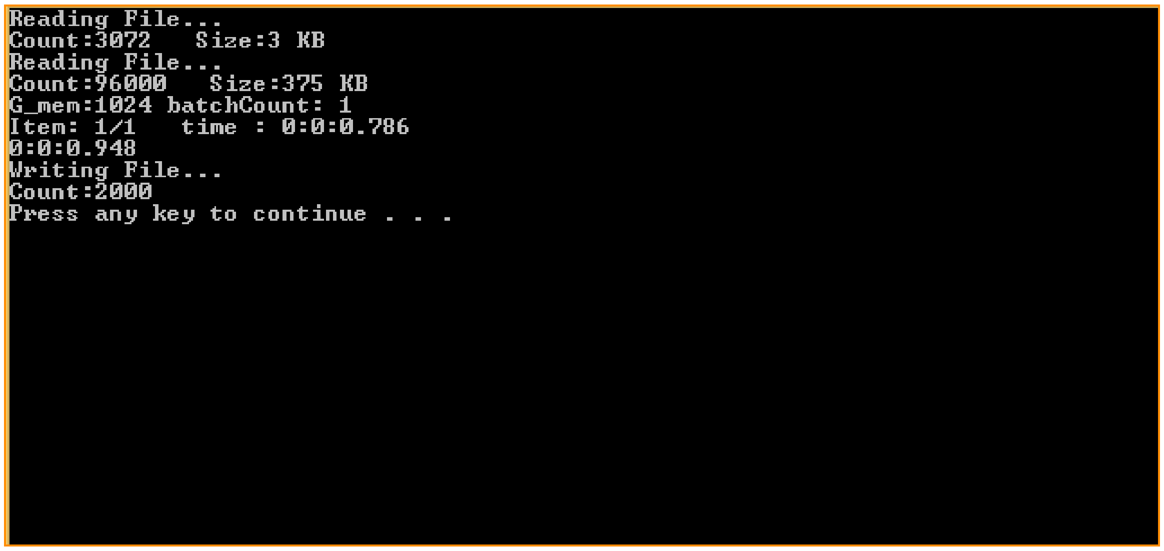
The following table includes the input and output parameters:

Table 4 : Parameters of feature extraction software Inputs

Variable	**Value**	**Description**
-i	Input File	The path of the input data.
-imgC	Number of images	Image count
-imgD	Image Size	Width of the images
-c	Centroids filename	The file that includes the centroids
-k	K value	Number of centroids
-p	Patches size	Size of receptive field
-reg	Region count	Number of regions for pooling
-ch	Channel count	Number of channels

Output Parameter

-r	Result filename	Filename for writing feature-vector result



```
Reading File...
Count:3072   Size:3 KB
Reading File...
Count:96000   Size:375 KB
G_mem:1024 batchCount: 1
Item: 1/1   time : 0:0:0.786
0:0:0.948
Writing File...
Count:2000
Press any key to continue . . .
```

Figure 30: Feature-extraction software

Appendix C. MATLAB Scripts

For the overall process I used MATLAB and a few scripts as follows:

- Preprocessing: this part of the code is used for reading the data, formalization, and patching phase. The source code is as follows:

```
function [data, M, P] = preprocessing( data, k, wSize, kmeansPath, path, dims, sampleRatio)
%data is the input raw data
%k is the number of needed clusters
%wSize is the size of the patches
%kmeansPath is path to k-means software
%dims is a vector includes the image's dimensions
%sampleRatio is used for sampling from the data


    chCount = dims(3);
    imgSize = dims(1);

    %% write data to file
    disp('write data to file...');
    iFN = [path, 'data.dat'];
    fid = fopen(iFN, 'Wb');
    fwrite(fid, data', 'float');
    fclose(fid);
    %% get normalize patches
    disp('get normalize patches...');
    totalSize = size(data,1);
    daFN = [path,'step1\',num2str(wSize),'\',num2str(k),'\Data.dat'];
    cmd = [kmeansPath ...
        ,' -r ',num2str(totalSize)...
        ,' -it 2 -ub -t 3 '...
        ,' -d ',num2str(imgSize)...
        ,' -no -p ',num2str(wSize)...
        ,' -sr ', num2str(sampleRatio)...
        ,' -sd 10 '...
        ,' -ch ',num2str(chCount)...
        ,' -i "',iFN ,'" ' ...
        ,' -dao "',daFN,'" '...
        ,' -k ',num2str(k)];
    system(cmd);

    %% read patches
    fInfo=dir(daFN);
    totalSize = fInfo.bytes / (wSize^2*chCount)/ 8;
    fis=fopen(daFN, 'rb');
    data = fread(fis, [wSize^2*chCount, totalSize], 'double')';
    fclose (fis);

    %delete(daFN);

    %% whitening

    disp('whitening...');
    C = cov(data);
    M = mean(data);
    [V,D] = eig(C);
    P = V * diag(sqrt(1./(diag(D) + 0.1))) * V';
    data = bsxfun(@minus, data, M) * P;

    disp('done');
end
```

Figure 31 : MATLAB script for preprocessing

➢ Model Layer: This script is used for the feature-learning package and could also be used for multi-layer learning.

```matlab
function [centers] = modelLayer(layerNo, data, k, wSize, kmeansPath, path, onlyRead, centM)
%layerNo is used for separating the output path
%data is patches
%k is the value of K
%wSize is the receptive field size
%kmeansPath is path of the k-means software
%path is used for result files
%onlyread is used for reading last results without new calculation
    [r d]=size(data);

    coFN = [path,'step1\L_',num2str(layerNo),'\',num2str(wSize),'\',num2str(k),'\Centers.dat'];
    clFN = [path,'step1\L_',num2str(layerNo),'\',num2str(wSize),'\',num2str(k),'\Clusters.dat'];
    daFN = [path,'step1\L_',num2str(layerNo),'\',num2str(wSize),'\',num2str(k),'\Data.dat'];
    rFN = [path,'step1\L_',num2str(layerNo),'\',num2str(wSize),'\',num2str(k),'\Result.dat'];
    if(~onlyRead)
        centers = randn(k,d)*centM;%X(randsample(size(X,1), k), :);
        icFN = [path,'initcenter.dat'];
        fid = fopen(icFN, 'Wb');
        fwrite(fid, centers' , 'double');
        fclose(fid);
        %%
        iFN = [path, 'data.dat'];
        fid = fopen(iFN, 'Wb');
        fwrite(fid, data', 'float');
        fclose(fid);

        %% starting kmeans
        cmd = [kmeansPath ...
            ,' -r ',num2str(r)...
            ,' -it 2 -ub -t 3 '...
            ,' -d ',num2str(d)...
            ,' -ch 1 ' ...
            ,' -i "',iFN ,'" ' ...
            ,' -ic "',icFN ,'" ' ...
            ,' -co "',coFN,'" ' ... y
            ,' -clo "',clFN,'" ' ...
            ,' -dao "',daFN,'" '...
            ,' -ro "',rFN,'" '...
            ,' -k ',num2str(k)];

        system(cmd);
    end
    disp(['done']);
    beep on
    beep
    centers =dlmread(coFN);

end
```

Figure 32 : MATLAB script of k-means

- Convolution and Pooling: The final script is for feature selection, which consists of convolution and pooling sections.

```
function [res] = conv_pool(images, centroids, dims, pw, M, P)
%images contains the original images
%centroids contains the extracted and calculated centroids from K-means
%dims includes the dimensions of the images
%pw is pooling size
%M is mean of last whitening
%P is the svd matrix of last whitening

    whitening = (nargin == 6);
    if(~whitening)
       M=0;
       P=0;
    end

    d2= dims(1)*dims(2);
    chCount = dims(3);
    wSize = (sqrt(size(centroids,2)/chCount));
    S2=wSize^2;

    numCentroids = size(centroids,1);
    %hw = (wSize-1)/2;
    %vs = dims(1) + 2 * hw;
    dp = dims(1)-wSize+1;
    dp2 = dp^2;
    ndp2=ceil(dp/pw)^2;

    res = zeros(size(images,1),ndp2 * numCentroids);
    for i=1:size(images,1)

        if (mod(i,200) == 0) fprintf('Extracting features: %d / %d\n', i,
size(images,1)); end

        img = zeros (dp2,S2*dims(3));

        for ch=1:dims(3)
            img(:, (ch-1)*S2+1:ch*S2) = im2col(reshape(images(i,(ch-
1)*d2+1:ch*d2),dims(1:2)),[wSize wSize])';
        end

        img = bsxfun(@rdivide, bsxfun(@minus, img, mean(img,2)),
sqrt(var(img,[],2)+10));
        % whiten
        if (whitening)
            img = bsxfun(@minus, img, M) * P;
        end

        % compute 'triangle' activation function
        xx = sum(img.^2, 2);
        cc = sum(centroids.^2, 2)';
        xc = img * centroids';

        z = sqrt( bsxfun(@plus, cc, bsxfun(@minus, xx, 2*xc)) ); % distances

        mu = mean(z, 2); % average distance to centroids for each patch
        img = max(bsxfun(@minus, mu, z), 0);
```

Figure 33 : Convolution and Pooling

```
        % patches is now the data matrix of activations for each patch
        if(pw~=1)
            img = reshape(img, dp, dp, numCentroids);
            img2=zeros(1,numCentroids*ndp2);
            for c=1:numCentroids
                [pp, ix] = MaxPooling(img(:,:,c), [pw pw]);
                img2((c-1)*ndp2+1:c*ndp2)=reshape(pp,1,ndp2);
            end
            res(i, :) = img2;%reshape(img2,1,nd2*numCentroids);
        else
            res(i, :) = reshape(img,1,ndp2*numCentroids);
        end
    end
    disp('done');
end
```

Figure 33, continued.