University of Burgundy

MsCV

SSI Pattern Recognition

Homework 2 Linear Classification

by

Gopikrishna Erabati

Supervisor: Dr. Desire Sidibe

NOTE:

1. There is a README.txt file in code folder which explains how to run the code. I tried my best to well extensively comment the code, to help the user better understand it. 'main.m' is the main file, it took 79 sec to get the total output for problem1 and problem2 (after getting regularization parameter values for problem 1).

2. In my report, the implementations of steps explained in the report can be found at corresponding functions in the code, (specified in yellow color) for easy access.

1. Logistic Regression

1.1 Introduction to problem

We want to build a classifier to filter spam emails. We were given training and test data sets of emails[1]. Each email has been processed and a set of 57 features were extracted.

- 48 features, in [0; 100], giving the percentage of words in a given message which match a given word on a predefined list (called vocabulary).
- 6 features, in [0; 100], giving the percentage of characters in the email that match a given character on the list.
- Feature 55: the average length of an uninterrupted sequence of capital letters.
- Feature 56: the length of the longest uninterrupted sequence of capital letters.
- Feature 57: the sum of the lengths of uninterrupted sequence of capital letters.

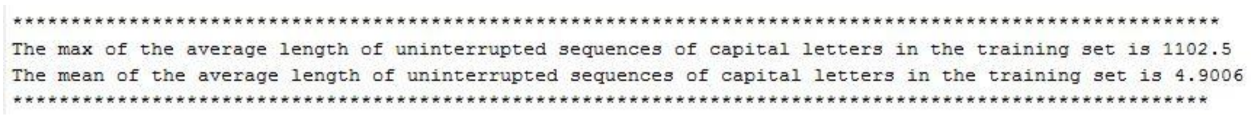
The data is stored in *'data_train'* for training data, '*data_train_lab*' for training data labels, '*data_test*' for test data and '*data_test_lab*' for test data labels in the code.

1.2 Tasks

1.2.1 What are the max and mean of the average length of uninterrupted sequences of capital letters in the training set?

This task is implemented by function
[data_max, data_mean] = getMaxMean(data, feature)
More details explained in function.

This can be computed by getting mean and max of 55th column of *'data_train'*. The result obtained is as shown in Fig. 1.



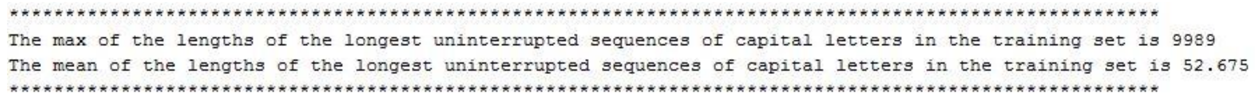
```
*****************************************************************************************************
The max of the average length of uninterrupted sequences of capital letters in the training set is 1102.5
The mean of the average length of uninterrupted sequences of capital letters in the training set is 4.9006
*****************************************************************************************************
```

Fig. 1 Result of task 1.2.1

1.2.2 What are the max and mean of the lengths of the longest uninterrupted sequences of capital letters in the training set?

This task is implemented by function described above.

This can be computed by getting mean and max of 56th column of *'data_train'*. The result obtained is as shown in Fig. 2.



```
*****************************************************************************************************
The max of the lengths of the longest uninterrupted sequences of capital letters in the training set is 9989
The mean of the lengths of the longest uninterrupted sequences of capital letters in the training set is 52.675
*****************************************************************************************************
```

Fig.2 Result of task 1.2.2

1.2.3 Preprocessing

This task is implemented by function

[data_process, data_mean, data_std] = getPreProcess(data , type)

1.2.3.1 Standardize the columns so they all have mean 0 and unit variance

In the above function, type = 1for standardize.

When features differ by order of magnitude, it is important to perform feature scaling or normalization. This is done by subtracting the mean of each feature from the dataset, and scale the feature values by their respective standard deviation.

$$for\ every\ feature\ f_i, f_i \leftarrow \frac{f_i - \bar{f_i}}{\sigma_{f\,i}}$$

This way, each feature is now centered (zero mean) and standardized (unit variance).

The mean and standard deviation of data is also taken as output to be used later.

The standardized data for train is stored in *'data_stand'*, and for test in *'data_test_stand'*.

1.2.3.2 Transform the features using log(x_{ij} + 0.1)

In the above function, type = 2 for transformed features by above relation.

the transformed data is stored in *'data_trans'*, and for test in *'data_test_trans'*.

1.2.3.3 Binarize the features using I(x_{ij} > 0)

In the above function, type = 3 for binarize features.

The binarized data is stored in '*data_bin'*, and for test in *'data_test_bin'*.

1.2.4 Fitting a Logistic Regression Model for each version of data

Priorly, a column of ones is added to train and test data to take care of bias term.

In order to fit the logistic regression model to data, we need to get the regularization parameter, λ. We use regularization parameter to avoid over fitting[2].

We use cross-validation (training data) to get the regularization parameter.

This is implemented by function
lambda = getlambda(data_train, data_train_lab)

I implemented this function, where it takes the train data and train data labels and it automatically calculates training error and low validation error and gives lambda as output.
Cross-validation is performed by taking:

- 80% of training data to train (called training data, here) and
- 20% of training data to validate (called validation data, here)

We define a range of lambda to get the minimum validation and train error for data.

The process goes as follows:

1. For each lambda value
    a. Get the regression parameters
    This is implemented by function
    regs_para = getRegPara(data, labels, lambda)

    In order to get the regression parameters(w), we use MATLAB built-in function 'fminunc' to get optimal value of regression parameters. So, the function requires a cost function to optimize. A function that, when given the training set and a particular 'w', computes the cost function and the gradient with respect to 'w' for the dataset [2].
    This cost function is implemented by
    [e grad] = costFunction_regu(X, y, w, lambda)

    The cost function is defined as,

$$E(w) = -\sum_{n=1}^{N}\{y_n \log(\phi_n) + (1-y_n)\log(1-\phi_n)\} + \frac{\lambda}{2}\sum_{k=1}^{M} w_k^2$$

    We do not regularize the bias term w_0.
    where, $\phi_n = y(\phi_n) = \sigma\left(w^T \phi(x_n)\right)$
    with, $\sigma(x) = \frac{1}{1+e^{-a}}$ is sigmoid function.
    The gradient of the cost function is a vector where the *i*th element is defined as:
    if *i* = 0

$$\frac{\partial E(w)}{\partial w_0} = \sum_{n=1}^{N}(y(\phi_n) - y_n)\phi_n$$

    else

$$\frac{\partial E(w)}{\partial w_i} = \sum_{n=1}^{N}(y(\phi_n) - y_n)\phi_n + \lambda w_i$$

b. After we get the regression parameters, we predict the class for validation set and training set.
   This is implemented by function
   [y] = predict(X,w)

   This prediction is done by using sigmoid function where we pass w^TX to the sigmoid function to get the prediction.
   As sigmoid function values are between 0 and 1, whose values can be interpreted as probabilities.

$$\sigma(a) > 0.5 \rightarrow a \text{ belongs to class } I$$
$$\sigma(a) < 0.5 \rightarrow a \text{ belongs to class } II$$

c. After we predict the validation labels and training labels, we compare with the true validation labels and train labels to get validation error and train error respectively.
   And this, error is stored for each lambda

2. After getting all errors for all lambda values, the minimum error is chosen and that particular index lambda value is returned by the program.

After we get the regularization parameter, we compute the regression parameters on the full training data and get the regression parameters. The process is as explained in (1.a) step above.

Get train Error : After getting the regression parameters from training data, we apply it on training data to get training error.
This is implemented by function
error_per = getError(data, labels, regs_para)

Here, we predict the class by using training data and regression parameters and assign a class as explained in (1.b) step above. Finally, we compare the predicted class to the true class we were given to find the error in prediction.

Get test Error: The same method above is applied on test data to get the test error.

1.2.5 Results and Discussion (logistic regression)

1.2.5.1 Choosing lambda for standardize data

I have tested various values of lambda to get the validation and train errors.
The result is as shown in Fig. 3

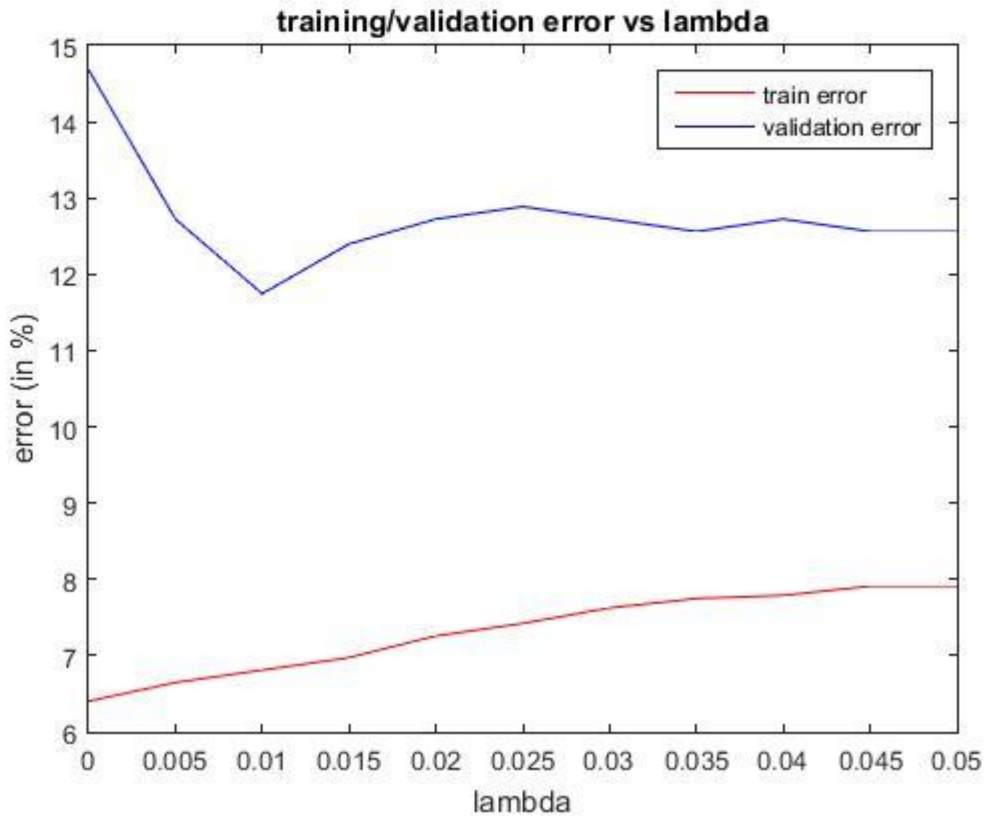


Fig. 3 Train and validation error for standardized data (to get lambda)

From the Fig.3, we can infer that for the lambda value of 0.01, the validation and train errors are optimal. So lambda for standardized data is chosen as 0.01.

	Train Error (in %)	Validation Error (in %)
Lambda = 0.01	6.81	11.74

1.2.5.2 Choosing lambda for transformed data

The same approach is been followed on transformed data.

The result is as shown in Fig. 4.

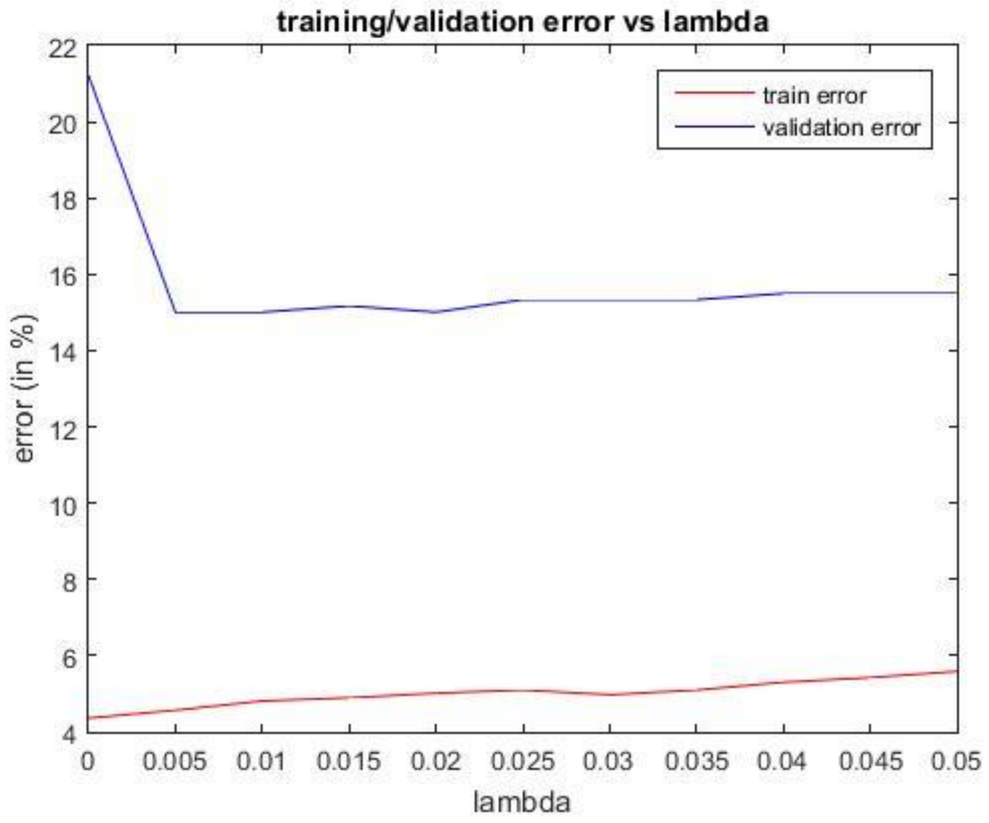


Fig. 4 Train and validation error for transformed data (to get lambda)

From the Fig.4, we can infer that for the lambda value of 0.005, the validation and train errors are optimal. So lambda for standardized data is chosen as 0.005.

	Train Error (in %)	Validation Error (in %)
Lambda = 0.01	4.56	15

1.2.5.3 Choosing lambda for binarized data

The same approach is been followed on binarizeded data.

The result is as shown in Fig. 5.

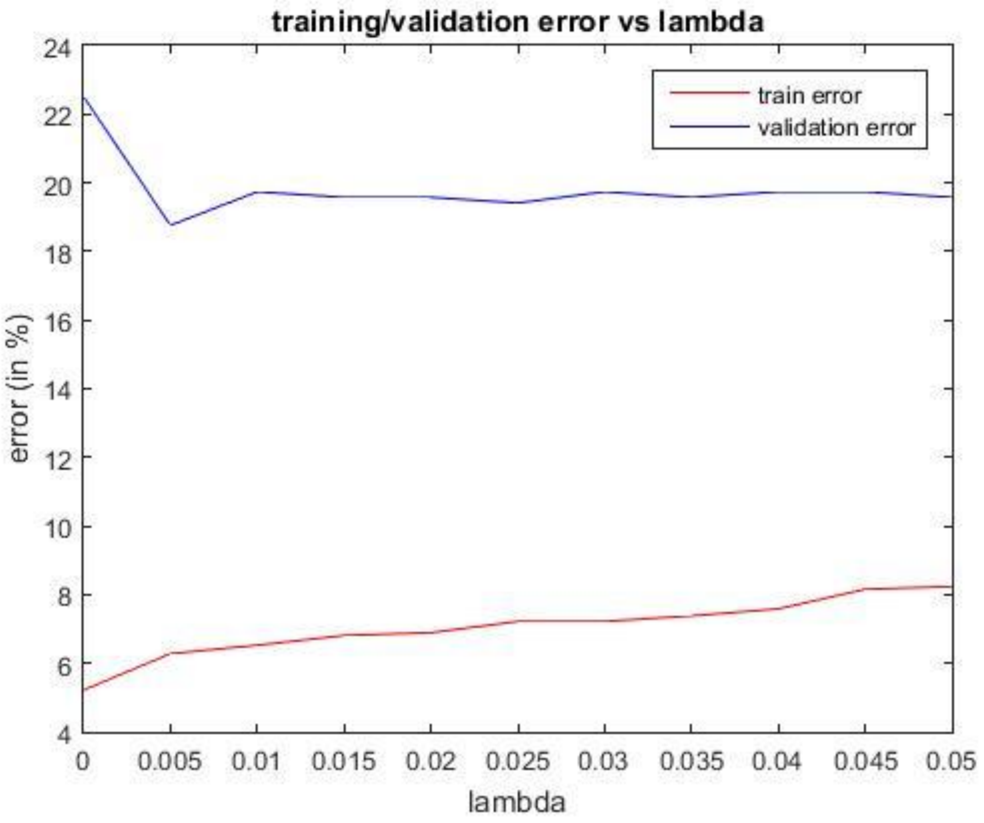


Fig. 5 Train and validation error for binarized data (to get lambda)

From the Fig.5, we can infer that for the lambda value of 0.005, the validation and train errors are optimal. So lambda for standardized data is chosen as 0.005.

	Train Error (in %)	Validation Error (in %)
Lambda = 0.01	6.28	18.76

1.2.6 Naive Bayes Classifier

The Naive Bayes Classifier is implemented by function
test_lab_naive = getNaiveBayes(data_train, data_train_lab, data_test)
This is my own implementation of Naive Bayes.

The main idea is, when we were given a test data to classify it, we will check the probability of the class I given test data(X), p(C_1|X) and also probability of class II given test data, p(C_2|X) and assign a class which has greater probability of two.

This is computed by following :

$$p(C_1|X) \propto p(X|C_1)p(C_1)$$

where, $p(C_1)$ can be found from training data as,

$$p(C_1) = \frac{1}{N}\sum_{n=1}^{N} t_{n_{C1}} = \frac{N_1}{N}$$

and

$$p(X|C_1) = p(X_1, X_2, \dots, X_m\,|C_1)$$

by assumption of i.i.d, we get

$$p(X|C_1) = p(X_1|C_1)p(X_2|C_1) \dots . p(X_m|C_1)$$

so finally,

$$p(C_1|X) \propto p(X_1|C_1)p(X_2|C_1) \dots . p(X_m|C_1)p(C_1)$$

$$p(C_1|X) \propto \prod_{n=1}^{m} p(X_n|C_1)\, p(C_1)$$

where, each $p(X_i|C_1)$ can be a normal distribution $\mathcal{N}(X_i;\ \mu_1, \sigma_1) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-1}{2}\left(\frac{(X_i-\mu_1)^2}{\sigma^2}\right)}$

The same applies to class II to get $p(C_2|X)$

$$if\ p(C_1|X) \geq\ p(C_2|X) assign\ class\ I$$
$$else\ assign\ class\ II$$

I have also tried to use Naive Bayes of MATLAB, by the function getErrorNBMatlab()

The results of my implementation and Matlab's have a close match as seen in Tab. 1.

1.2.6 Overall Train and Test Errors

TABLE I Train and Test Errors

DataType	**λ**	**Train Error (in %)**	**Test Error (in %)**			
			With preprocessing of test data	With preprocessing of test data using train data (only for method1)	Naive Bayes (my implementation)	Naive Bayes (Matlab)
Standadized Data	0.01	8.32	8.65	8.789	18.94	18.88
Transformed Data	0.005	5.18	6.05	-	18.099	18.099
Binarized Data	0.005	6.98	7.55	-	38.76	-

The Tab. 1 gives the overall train and test errors using logistic regression and Naive Bayes for all sets of data.

A common good practice in classification is to do feature normalization or data standardization of the features, i.e., center the data subtracting the mean and normalize it dividing by the variance (or standard deviation too). For self containment and to my understanding we do this to achieve two main things:

1. Avoid extra small model weights for the purpose of numerical stability.
2. Ensure quick convergence of optimization algorithms like e.g. Conjugate Gradient so that the large magnitude of one feature dimension w.r.t. the others doesn't lead to slow convergence.

If testing data is expected to strongly vary from the training data, it might be also interesting to use the test data for normalization. This might slightly go into the direction of domain adaption. However, that the applicability of the approach highly depends on whether our problem setting admits such a treatment (e.g. often in classification, we only have one test point at a time at our disposal, which renders the test mean approach inapplicable). So, in this case we take the mean and standard deviation from train data and apply it on test data.

But, in this data, the transformed feature data (using log(x+0.1)) gives less error, as data in certain columns have high order of information, so when we apply logarithm to such data, it would give us ess numbers as any other columns of data. So, now all the adta would be in proper order to get better results.

2. Cats and dogs classification using SVM

2.1 Introduction to Problem

We were given cat and dog data of 80 images each of dimension 64 x 64 [1]. total 160 images were divided into two sets training and validation set in ratio of 80:20 .

The data is in the dimension of 4096 x 80, where each column corresponds to each image. So, here I wrote a function to get images from the data. the function is
imageMatrix = getImageMatrix(data)

2.2 Feature Descriptor

I have tried using different feature descriptors to test on the data.
this is implemented by using the function
[catFeatures, dogFeatures] = getFeatures(cat, dog, featureType)

featureTypes:

1. Intensity descriptor
The basic feature is the color of cats and dogs, as the color of both of them vary. I tried to classify the data using SVM with this feature descriptor.

2. Corner descriptor
I used Harris corner detector to detect the corners of images and extract features in the images using the corner points.

3. Histogram of Oriented Gradients (HOG) feature descriptor
The **histogram of oriented gradients (HOG)** is a feature descriptor used in computer vision and image processing for the purpose of object detection. This method is similar to that of edge orientation histograms, scale-invariant feature transform descriptors, but differs in that it is computed on a dense grid of uniformly spaced cells and uses overlapping local contrast normalization for improved accuracy.

4. FAST Features
I used FAST feature detector to detect the keypoints and extract feature descriptor around these keypoints in the image. Features from accelerated segment test (FAST) is a corner detection method, which could be used to extract feature points. The most promising advantage of the FAST corner detector is its computational efficiency. Referring to its name, it is fast and indeed it is faster than many other well-known feature extraction methods, such as difference of Gaussians (DoG).

5. SURF Features
Speeded up robust features (SURF) is a local feature detector and descriptor. The standard version of SURF is several times faster than SIFT and claimed by its authors to be more robust against different image transformations than SIFT.

6. SIFT Feature
Scale-invariant feature transform (SIFT) is an algorithm in computer vision to detect and describe local features in images. For any object in an image, interesting points on the object can be extracted to provide a "feature description" of the object. This description, extracted from a training image, can then be used to identify the object when attempting to locate the object in a test image containing many other objects. To perform reliable recognition, it is important that the features extracted from the training image be detectable even under changes in image scale, noise and illumination. Such points usually lie on high-contrast regions of the image, such as object edges.

Out of all the feature descriptors, I would prefer **Histogram of oriented Gradients (HOG)** feature descriptor, because it is purely gradient based, and captures the object shape information well. HOG compute edge gradient of a whole image and find orientation of each pixel so it can generate a histogram. For global feature extraction HOG works well. One can think of SIFT = HOG of the detected key points. SIFT is typically computed at interest points.

2.3 SVM

Support vector machines are supervised learning models with associated learning algorithms that analyze data used for classification [3]. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other. . An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall.
After training of SVM, we will keep only a subset of training data. these data points are called support vectors. At the heart of SVMs is the notion of margins [2]. A margin is the smallest signed distance between decision boundary and training data. A good classifier should produce a decision boundary with large margin.

I used the MTLABs SVM classifier function to train the classifier.
The implementation of SVM when given features using cross validation is :
[error_train, error_valid] = trainSVMAndGetError(cat, dog)

We split the data into training and validation data as said before and train the SVM classifier using the train data with gaussian kernel.

Why Kernel ?
In many cases, computing the kernel is easy, but computing the feature vector corresponding to the kernel is really hard. Many machine learning algorithms can be written to *only* use dot products, and then we can replace the dot products with kernels. By doing so, we don't have to use the feature vector at all. This means that we can work with highly complex, efficient-to-compute, and yet high performing kernels without ever having to write down the huge dimensional feature vector. Thus, if not for the ability to use kernel functions directly, we would be stuck with relatively low dimensional, low-performance feature vectors. This "trick" is called the *kernel trick.*

2.4 Results

TABLE II Train and test errors for SVM classification

Feature	Train error (in %)	Validation error (in %)	Time (in sec)
Intensity	0	37.5	0.911
Harris Corner	5.46	21.87	12.68
HOG	0.78	12.5	1.3
FAST Features	7.03	34.37	13.5
SURF Features	5.46	43.75	2.49
SIFT Features	3.125	40.62	1046

From the above table, we can see that our choice of HOG feature descriptor gives less test validation error as compared to other methods and it takes less time for the execution of descriptor and also the SVM classifier.

References

[1] Problem statement of Homework2 by Dr. Desire Sidebe
[2] Lecture slides of Dr. Desire Sidebe
[3] SVM, Wikipedia