

INT304: LAB2 REPORT

Seungyeop Kang
Student ID: 1825478

ABSTRACT

Discriminant function is a classification method applicable for the case when the dependent variable or criterion is categorical, and the independent variable or prediction is interval in nature. Non-parametric classification is a method that does not require strong assumptions about the form of mapping functions. In this report, two methods, Modified Quadratic Discriminant Function and Parzen Window Method, will be used to check their performance and properties with UCI Iris dataset.

1 INTRODUCTION

Pattern recognition is one of the most important areas in Machine learning. Therefore, various pattern classification techniques have been developed for decades. Parametric probability density estimation is a commonly used method, but it is sometimes challenging to estimate the probability density function without knowing its parametric form. In this case, non-parametric density estimation can be applied because it can determine the best fit of the training data for the mapping function while maintaining the ability to generalize for unknown data (Brownlee, 2016). Parzen window method is one of the non-parametric density estimation technique. By using a kernel function, it can approximate the distribution of the given dataset, estimating densities of each classes and choosing the class with maximal posterior probability (Brown, 1999).

Discriminant analysis (DA) is a parametric classification technique to classify two or more groups when groups are known a priori. The earliest adoption of DA is Linear Discriminant Analysis (LDA) in 1936. Through a generalization of Fisher's linear discriminant, it is widely used for feature extraction in pattern classification (Dash, 2021). After that, Quadratic Discriminant Analysis (QDA) was introduced based on the evolution of LDA for non-linear classification. In LDA, the covariance matrix was assumed as a constant, but QDA assumes that the dispersion is not same for all categories. Thus, it follows that each category differs for the position of its centroid and covariance matrix. However, there are several issues of QDA. The first issue is that the performance is degraded due to the parameter estimation errors. Secondly, the performance will be worse if the feature size keeps increasing over the appropriate size. Lastly, it will require significant computation time and storage when the dimension is considerably high (Kimura et al., 1987). Hence, the Modified Quadratic Discriminant Function (MQDF) was introduced to solve these problems and improve the performance.

2 METHODOLOGY

2.1 IRIS DATASET

Iris dataset is a famous flower dataset provided by University of California Irvine (UCI) in 1936. This is known as the best known database for pattern recognition. This dataset includes 150 samples, with 4 attributes. The four features have same units and in all numeric. There are three classes with equally distributed samples without any missing data. Thus, it is easy to implement without time-consuming for data preparation (Zhang, 2020).

2.2 PARZEN WINDOW

As mentioned, Parzen Window is a non-parametric estimation method through kernel density estimation because the type of the underlying distribution is unknown. To implement this technique, the prior class probability $P(w_k)$ of each class can be calculated as follows:

$$P(w_k) = \frac{n_k}{\sum_{i=1}^K n_i} \quad (1)$$

where n_k is the number of samples in the class w_k and K is the total number of classes.

And then, the conditional probability of a test sample x can be computed by using a kernel function. There are various types of kernel functions such as uniform, triangular, cosine, quartic and so on. In this report, the gaussian

kernel is used according to the given requirement. However, the gaussian kernel is not appropriate to compute multi-dimensional data. Hence, the data with high dimension will be computed by multivariate gaussian distribution as follows when $u = \frac{x-x_i}{h}$:

$$\phi_1(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{u^2}{2}} \quad \& \quad \phi_2(u) = \frac{\exp(-\frac{1}{2}u^T \Sigma^{-1} u)}{\sqrt{(2\pi)^d |\Sigma|}} \quad (2)$$

where x is a test sample, x_i is a train data, h is the kernel size, and d is the dimension of the dataset (Sellner, 2017). Then, based on these kernel functions, the conditional probability of a test sample x can be computed as follows:

$$P(x|w_k) = \frac{1}{n_k} \sum_{x_i \in w_k} \frac{1}{h^d} \phi\left(\frac{x-x_i}{h}\right) \quad (3)$$

Now, the posterior probability can be calculated according to the Bayesian theorem:

$$P(w_k|x) = \frac{P(x|w_k)P(w_k)}{P(x)} \propto P(x|w_k)P(w_k) \quad (4)$$

2.3 MODIFIED QUADRATIC DISCRIMINANT FUNCTION (MQDF)

According to equation 4, the a posteriori probability is calculated based on the class probability density function (pdf) $P(x|w_k)$ and a priori probability $P(w_k)$ because the mixture density function $P(x)$ is independent of class label. Then, the discriminant function for classification can be defined as:

$$g(x, w_i) = P(x|w_i)P(w_i) \quad (5)$$

By applying the multivariate gaussian density to $P(x|w_i)$, the QDF can be obtained as follows:

$$g_0(x, w_i) = (x - \mu_i)^T \Sigma_i^{-1} (x - \mu_i) + \log |\Sigma_i|. \quad (6)$$

In equation 6, the covariance matrix Σ_i can be diagonalized as $\Sigma_i = \Phi_i \Lambda_i \Phi_i^T$ by K-L transform where Φ_i is the descending ordered eigenvectors and Λ_i is the descending ordered eigenvalues. Then, the QDF can be modified as follows:

$$g_0(x, w_i) = [\Phi_i^T (x - \mu_i)]^T \Lambda_i^{-1} \Phi_i^T (x - \mu_i) + \log |\Lambda_i| = \sum_i \frac{1}{\lambda_{ij}} [\Phi_i^T (x - \mu_i)]^2 + \sum_{j=1}^d \log \lambda_{ij}. \quad (7)$$

Here, by replacing the i th eigenvalues to the i th eigenvalues with an appropriate constant, the MQDF1 can be obtained with the improved performance. It can be further improved if the minor eigenvalues in the descending order are replacing with a constant δ_i . Then, the MQDF2 can be defined as follows:

$$\begin{aligned} g_2(x, w_i) &= \sum_{j=1}^k \frac{1}{\lambda_{ij}} [\phi_{ij}^T (x - \mu_i)]^2 + \sum_{j=k+1}^d \frac{1}{\delta_i} [\phi_{ij}^T (x - \mu_i)]^2 + \sum_{j=1}^k \log \lambda_{ij} + (d-k) \log \delta_i \\ &= \sum_{j=1}^k \frac{1}{\lambda_{ij}} [\phi_{ij}^T (x - \mu_i)]^2 + \frac{1}{\delta_i} r_i(x) + \sum_{j=1}^k \log \lambda_{ij} + (d-k) \log \delta_i \end{aligned} \quad (8)$$

where $r_i(x) = \|x - \mu_i\|^2 - \sum_{j=1}^k [(x - \mu_i)^T \phi_{ij}]^2$ when we use the invariance of Euclidean distance for $\|x - \mu_i\|^2 = \sum_{j=1}^d [\phi_{ij}^T (x - \mu_i)]^2$. In this report, the δ_i will be calculated by the average of minor eigenvalues which is provided by Moghaddam and Pentland (Liu et al., 2004). Then, the computation time and storage can be reduced significantly when the dimension of dataset is high. Here, MQDF3 will not be considered because the 5-fold cross validation used in this report has the train set 4 times greater than the test set. Thus, Regularized Discriminant Analysis (RDA) is not required.

3 DESIGN AND RESULTS

In this section, the detailed steps and process of the self-designed algorithms will be explained step-by-step with the clipped running images. Since it is restricted to use any third-party library in the core part of the code, the self-defined functions are firstly designed as follows.

Algorithm 1 Function to visualize the iris dataset: **twoD_plot()** & **fourD_plot()**

```

1: Input: 'filename' for twoD_plot()
2: Input: each feature data for fourD_plot()
3: twoD_plot: read the 'filename' data via pandas
4: plot the 2D analysis result via seaborn
5: fourD_plot: assign each feature to axes
6: x=sepal length, y=sepal width, z=petal length
7: c = petal width
8: plot the 3D scatter plot with colour gradient (c)

```

Algorithm 3 Function to predict the class of test data: **pz_predict()**

```

1: Input: x_len, np_array
2:
3: for i in range(x_len) do
4:   max = max(cond. prob. of features)
5:   if max == np_array[0][i] then
6:     pred = 'Iris-setosa'
7:   else if max == np_array[1][i] then
8:     pred = 'Iris-versicolor'
9:   else
10:    pred = 'Iris-virginica'
11:   end if
12:   Append pred → x_pred[]
13: end for
14: Return x_pred

```

Algorithm 5 Kernel function to compute normal distribution: **normal_kernel()**

```

1: Input: test data  $x$ , train data  $x_i$ , hyper-parameter  $h$ 
2:  $u = \frac{x-x_i}{h}$ 
3: Return  $\frac{1}{\sqrt{2\pi}} e^{-\frac{u^2}{2}}$ 

```

Algorithm 6 Function to calculate the classification accuracy: **pz_accuracy()**

```

1: Input: predicted class result  $pred\_class$ , real class names  $class\_x$ 
2: for index, pred in enumerate(pred_class) do
3:   if pred == class_x[index] then
4:     acc += 1
5:   else
6:     pass
7:   end if
8: end for
9: Return acc / len(pred_class) * 100

```

Algorithm 2 Kernel function to compute multivariate normal distribution: **multivariate_normal_kernel()**

```

1: Input: test data  $x$ , train data  $x_i$ ,
2:   hyper-parameter  $h$ , covariance matrix
3: det_cov: determinant of the covariance matrix
4: inv_cov: inverse of the covariance matrix
5:  $u = \frac{x-x_i}{h}$ 
6: numer =  $\exp(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu))$ 
7: denom =  $\sqrt{(2\pi)^k} |\Sigma|$ 
8: Return numer / denom

```

Algorithm 4 Function to implement Parzen Window Method: **parzen_window()**

```

1: Input: test data  $test$ , train data  $train$ ,
2:   hyper-parameter  $h$ , dimension  $d$ 
3: cov = identity matrix scaled with  $d$ 
4: if  $d == 1$  then
5:   p = normal_kernel() / h
6:   (Input: test, train, d)
7: Return p
8: else
9:   p = multivariate_normal_kernel() /  $h^d$ 
10:  (Input: test, train, h, cov)
11: Return p
12: end if

```

Here, Algorithm 1 can be used to visualize the iris dataset to check its properties clearly as Fig. 1 and Fig. 2. Note that the colour gradient in Fig. 1 denotes the petal width. For Parzen Window method, Algorithm 4 can be implemented first to check the dimension. Depends on the dimension of the dataset, the proper kernel function will be used between

Algorithm 2 and Algorithm 5. After computing the a posterior probability of test samples, the classification can be conducted by Algorithm 3 and the prediction accuracy will be calculated by Algorithm 6.

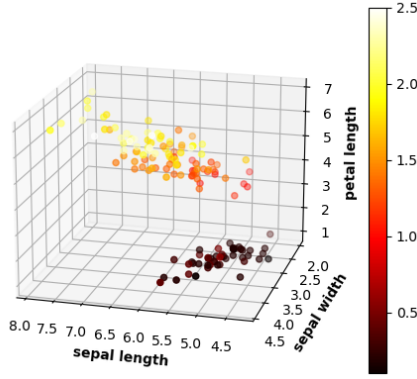


Figure 1: Iris dataset in 3D with colour gradient.

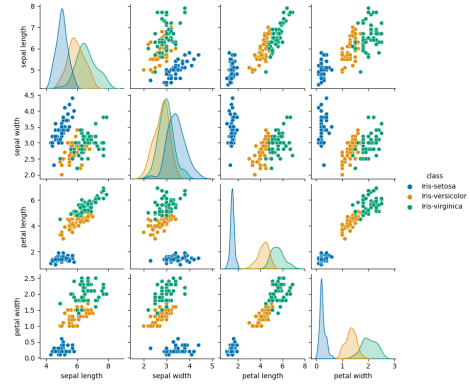


Figure 2: Iris dataset in 2D analysis.

Algorithm 7 Class for Data preprocessing according to the requirements: **Data_process()**

- 1: **__init__(self)**: filename (iris.data), line_comp = [] and iris_list = []
 - 2: **load_data(self)**: Method to load the dataset and store them in a list
 - 3: **shuffle(self)**: Method to shuffle the stored dataset
 - 4: **separate_data(self)**: Method to separate the dataset into five subsets for 5-fold cross validation
 - 5: **combine_train(self, index, total_data)**:
 - 6: Method to separate the combined sets to a train set and a test set.
 - 7: **separate_class(self, dataset)**: Method to separate the dataset into three given classes
 - 8: **numeric_n_name(self, nested_list)**: Method to separate the numeric data and class names
 - 9: **data_analyzer(self, info)**:
 - 10: Method to plot the 2D and 4D figures of the given dataset to analyse the properties of the iris data
 - 11: **prior_prob(self, dataset)**: Method to calculate the prior probabilities of each class
-

Algorithm 7 is a Class that can be used to process the iris dataset and prepare the test and train sets. Since the structures of the defined methods are simple, the details can be checked in Appendix. Then, based on these self-designed functions and class, the Parzen Window method can be implemented as Algorithm 8.

```
train set: [[array([[4.7, 3.2, 1.6, 0.2],
[5.5, 3.5, 1.3, 0.2],
[4.5, 2.3, 1.3, 0.3],
[4.9, 3.1, 1.5, 0.1],
[4.9, 3.1, 1.5, 0.1],
[5.2, 4.1, 1.5, 0.1],
[4.8, 3.4, 1.6, 0.2],
[4.6, 3.1, 1.5, 0.2],
[5.1, 3.7, 1.5, 0.4],
[4.4, 3. , 1.3, 0.2],
```

Figure 3: Shuffled train dataset.

```
test set: [[5.5 4.2 1.4 0.2]
[6.7 3.3 5.7 2.5]
[6.7 2.5 5.8 1.8]
[4.8 3.4 1.9 0.2]
[5.4 3.4 1.7 0.2]
[4.6 3.2 1.4 0.2]
[5. , 3.4 1.6 0.4]
[5.8 4. , 1.2 0.2]
[5.5 2.3 4. , 1.3]
[6.4 3.2 4.5 1.5]
```

Figure 4: Shuffled test dataset.

```
P(wk): (0.3, 0.35, 0.35)
Actual labels of test set: ['Iris-setosa', 'Iris-setosa',
Process finished with exit code -1
```

Figure 5: $P(w_k)$ & test labels.

Based on these preprocessed datasets, the Parzen Window algorithm can be implemented. Firstly, the kernel function will be iterated for the whole train samples in each class. After that, $P(x|w_k)$ can be computed based on the accumulated outputs of the kernel function. Additionally, the log-likelihood function is applied here to find the optimal kernel size during the 5-fold cross validation of Parzen Window algorithm:

$$L^i(h) = \sum_{x \in np_x} \log(P(x|w_k)). \quad (9)$$

Later, $L^i(h)$ will be divided by the number of cross validation sets (=5 in this report) to obtain the averaged result. The following figures show the clipped results of these steps.

```

Output of the kernel function: 1.2004281212431078e-08
Output of the kernel function: 0.03056187107456834
Output of the kernel function: 0.03056187107456834
Output of the kernel function: 0.03056190805035759
Output of the kernel function: 0.030561945026146838
Output of the kernel function: 3.563032559486036
Output of the kernel function: 3.5630396058242164
Output of the kernel function: 3.563039605975328
Output of the kernel function: 3.61342759309665
Output of the kernel function: 3.6134275930967075
Output of the kernel function: 3.6140618842459005

```

Figure 6: Output of the kernel function $\phi_2(u)$ in equation 2.

```

P(x|wk): 0.3015773950530228
L(h): -0.5206012144324176
P(x|wk): 4.946922662101319e-121
L(h): -120.82626609357403
P(x|wk): 9.824144725817652e-119
L(h): -238.83397134185748
P(x|wk): 0.4423748497603017
L(h): -239.18818091370733
P(x|wk): 0.7648545588174582
L(h): -239.30460205411993

```

Figure 7: $P(x|w_k)$ & $L^i(h)$.

Then, finally, the posterior probability of each test samples can be computed by $P(x|w_k)P(w_k)$ and stored in each class array as Fig. 8. Based on these results, we can predict the class of each test sample as Fig. 9 and calculate the accuracy of each iteration. After completing 5-fold cross validation, the average accuracy and $L^i(h)$ can be also obtained to validate the final result as Fig. 10. Since this result will be influenced by the kernel size h , this process was repeated from $h = 0.2$ to $h = 3$. Then, the optimal kernel size will be the point where the output of $L^i(h)$ is the highest. The final result will be shown in the next section to analyse the obtained graphs.

Algorithm 8 Main part of the Parzen Window Method code

```

1: if __name__ == '__main__':
2:   Data preprocessing via iris = Data_process()
3:   for h in range(0.2, 3, 0.1): do
4:     for index in range(len(five subsets)) do
5:       total_subset: [0]=train set & [1]=test set via iris.combine_train(index, five subsets)
6:       sep_dataset: [0]=set., [1]=ver. & [2]=vir. via iris.separate_class(total_subset[0])
7:       sep_data = [sep_dataset[0], sep_dataset[1], sep_dataset[2]]
8:        $P(w_k)$ : iris.prior_prob(sep_dataset)
9:       np_se, np_ver, np_vir = np. numeric data of three classes
10:      train = [np_se, np_ver, np_vir] in float
11:      x = numeric data of the test set & np_x = x in float
12:      d = dimension, x_len = len(np_x) & class_x = classes of test data
13:      for name in train do (Start the Parzen Window Method)
14:        for x in np_x do
15:          for x_i in name do
16:             $p_x = \sum \frac{\exp(-\frac{1}{2}u^T \Sigma^{-1}u)}{h^d \sqrt{(2\pi)^k |\Sigma|}}$  when  $u = \frac{x-x_i}{h}$  and  $d > 1$ 
17:          end for
18:           $P(x|w_k) = \frac{p_x}{n_k}$  & opt_size += log(p_xw)
19:          if cn == 0 then
20:            Append  $P(w_k)P(x|w_k) \rightarrow p\_se$ 
21:          else if cn == 1 then
22:            Append  $P(w_k)P(x|w_k) \rightarrow p\_ver$ 
23:          else
24:            Append  $P(w_k)P(x|w_k) \rightarrow p\_vir$ 
25:          end if
26:        end for
27:        cn += 1
28:      end for
29:      prob_array = np.array([p_se, p_ver, p_vir]) & pred_class = pz_predict(x_len, prob_array)
30:      pz_acc = pz_accuracy(pred_class, class_x)
31:    end for
32:  end for

```

Algorithm 9 Function to delete two less important features from the iris data: **delete_two_col()**

```

1: Input: numpy array of iris data
2: d1: Deleted the first column of iris data & d2: Deleted the first column of d1
3: Return d2

```

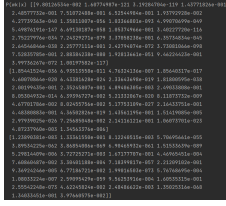


Figure 8: $P(w_k|x)$ of 30 test samples.

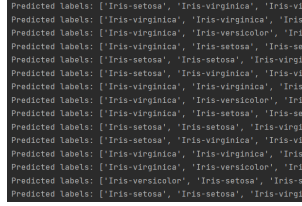


Figure 9: Predicted labels of test samples for h iterations.

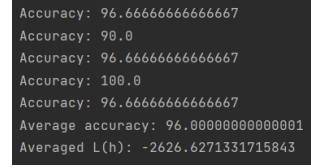


Figure 10: Parzen Window result of 5-fold cross validation.

Additionally, the Parzen Window algorithm was conducted again, but only with two most sensitive features according to Fig. 1 and Fig. 2. The two figures clearly show that the petal length and petal width have high sensitivity to classify the dataset. Thus, Algorithm 9 will delete the sepal length and sepal width from the dataset. Then, we can visualize the relationship between the predicted class and $P(w_k|x)$.

Algorithm 10 Function to calculate the mean and covariance matrix of the dataset: **QDF_model()**

- 1: **Input:** train set *train*, # of classes
- 2: **for** i in **range**(# of classes) **do**
- 3: train[i] = np.matrix(ith train data)
- 4: Append [mean of i th train].T \rightarrow mean[]
- 5: Append cov. matrix of [i th train].T \rightarrow cov_matrix
- 6: **end for**
- 7: **Return** mean, cov_matrix

Algorithm 11 Function to compute train parameters of MQDF2: **MQDF2_model()**

- 1: **Input:** covariance matrix, d , k , # of classes
- 2: **for** i in **range**(# of classes) **do**
- 3: covMat = covariance matrix of the i th class
- 4: eigvals, eigvecs = eigenval. & eigenvect. of covMat
- 5: id = index of the ascending order of the eigenval.
- 6: id = id[::-1] (convert to the descending order)
- 7: eigvals = eigvals[id] & eigvecs = eigvecs[:, id]
- 8: Append eigenvect. from $j = 1$ to $k \rightarrow$ eigenvectors[]
- 9: Append eigenval. from $j = 1$ to $k \rightarrow$ eigenvalues[]
- 10: delta[i] = $\frac{1}{d-k} \sum_{j=k+1}^d \lambda_{ij}$
- 11: **end for**
- 12: **Return** eigenvalue, eigenvector, delta

Algorithm 10 and Algorithm 11 are designed to obtain the trained parameters of the MQDF2 model. Firstly, Algorithm 10 will compute the mean and covariance matrix of train dataset. And then, these outputs will be used in Algorithm 11 to obtain the eigenvalues and eigenvectors of the train dataset. Here, the order of eigenvalues and eigenvectors will be covered into the descending order to remove the minor eigenvalues defined by k . The other training parameter δ_i will be calculated as follows:

$$\delta_i = \frac{tr(\Sigma_i) - \sum_{j=1}^k \lambda_{ij}}{d-k} = \frac{1}{d-k} \sum_{j=k+1}^d \lambda_{ij} \quad (10)$$

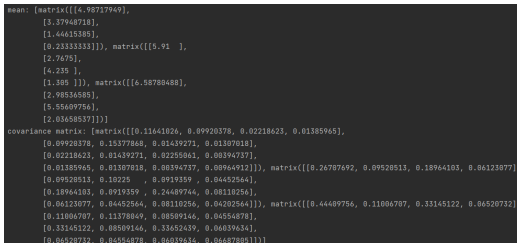


Figure 11: Mean and Covariance matrix of train set.

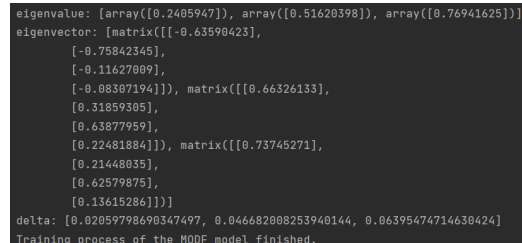


Figure 12: Trained parameters of MQDF2.

After completing the training process of MQDF, these trained parameters will be used to predict the samples in test set. The general process of the MQDF prediction is explained in Algorithm 12. These procedures are simply to calculate equation 8. From this algorithm, $g_2(x, w_i)$ of the test sample can be computed for three classes, and the class with the maximal value of $g_2(x, w_i)$ will be chosen for the predicted class label.

```

g2(x, w1) = 6.709646167715402
g2(x, w1) = -167.53706495733977
g2(x, w1) = -190.87466266224828
g2(x, w1) = -1127.5088950279455

```

Figure 13: $g_2(x, w_i)$ of each test sample.

```

Predicted labels: [0, 2, 2, 0,
Predicted labels: [2, 2, 1, 1,
Predicted labels: [2, 1, 0, 0,
Predicted labels: [2, 0, 0, 1,

```

Figure 14: Predicted labels of test set during iterations.

Since the predicted labels of test samples are obtained, the classification accuracy can be computed for 5-fold cross validation as Fig. 15. However, these accuracy results are influenced by the parameter k . Thus, several iterations were conducted to determine an appropriate value of k . These results will be shown in the next section with analysis. Note that the main part of MQDF code is similar to the Parzen Window code, instead used the self-defined functions for MQDF. Details of the code are explained through comments in MQDF.py.

```

Training process of the MQDF model finished.
1 th accuracy: 96.66666666666667
Training process of the MQDF model finished.
2 th accuracy: 90.0
Training process of the MQDF model finished.
3 th accuracy: 96.66666666666667
Training process of the MQDF model finished.
4 th accuracy: 100.0
Training process of the MQDF model finished.
5 th accuracy: 100.0
Averay accuracy of 5-fold cross-validation when k = 1 : 96.66666666666667
-----

```

Figure 15: MQDF result of 5-fold cross validation.

Algorithm 12 Function to predict classes based on MQDF2 trained parameters: **predict_MQDF2()**

```

1: Input: d, test_set, # of classes, k, mean, eigenvalue, eigenvector, delta
2: Assert error when  $k < d$  &  $k > 0$ 
3: for sample in test_set do
4:   test_x = transposed np.matrix(sample) & max_g2 =  $-\infty$ 
5:   for i in range(# of classes) do
6:     dis =  $\|x - \mu_i\|^2$  & euc_dis = [0] * k
7:     for j in range(k) do
8:       euc_dis[j] =  $\sum_{j=1}^k [\phi_{ij}^T (x - \mu_i)]^2$  & ma_dis[j] =  $\sum_{j=1}^k [(x - \mu_i)^T \phi_{ij}]^2$ 
9:     end for
10:    g2 = 0
11:    for j in range(k) do
12:      g2 +=  $\sum_{j=1}^k \frac{1}{\lambda_{ij}} [\phi_{ij}^T (x - \mu_i)]^2 + \sum_{j=1}^k \log \lambda_{ij}$ 
13:    end for
14:    g2 +=  $\frac{1}{\delta_i} r_i(x) + (d - k) \log \delta_i$  when  $r_i(x) = \|x - \mu_i\|^2 - \sum_{j=1}^k [(x - \mu_i)^T \phi_{ij}]^2$ 
15:    g2 = -g2
16:    if g2 > max_g2 then
17:      max_g2 = g2 & prediction = i
18:    else if g2 == max_g2 then
19:      error
20:    else
21:      pass
22:    end if
23:  end for
24:  Append prediction  $\rightarrow$  pred_label[]
25: end for
26: Return pred_label

```

4 ANALYSIS

In the Design and Results section, details of the important parts in the designed codes are explained, and clipped running images are provided. These results proved that those algorithms are correctly designed and have high performance. However, it is still not explained that how to determine the appropriate value of the kernel size h in Parzen Window Method and k in MQDF algorithm. Hence, this section will discuss the proper way to choose these values by analysing the obtained results.

To choose the optimal kernel size of Parzen Window Method, the designed algorithm was iterated from 0.2 to 3 with 0.1 increment. Then, two graphs can be obtained as follows:

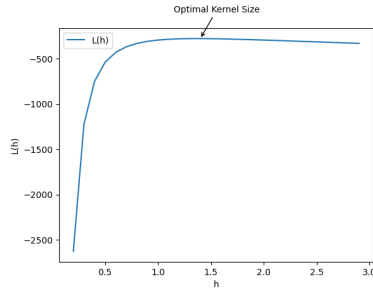


Figure 16: Optimal kernel size estimation.

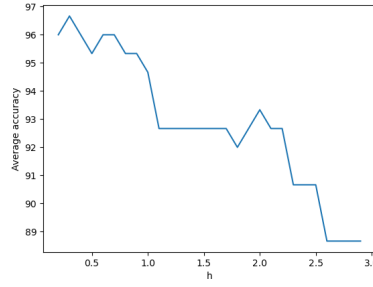


Figure 17: Average accuracy of Parzen Window.

```
Optimal Kernel Size: [1.4]
Average accuracy when h = [1.4] : 92.66666666666666 %
Process finished with exit code 0
```

Figure 18: Average accuracy with optimal kernel size.

According to the above figures, the optimal kernel size is chosen at 1.4. When h is equal to 1.4, the average accuracy is 92.67%. It means that the optimal kernel size and highest accuracy points are different. It is possible when the total dataset is not large. Since the iris dataset has 150 samples and the samples have few differences, the accuracy will be higher if the kernel size is smaller. However, this could be dangerous for other unknown data because the accuracy will decrease rapidly if the sample has slightly more difference.

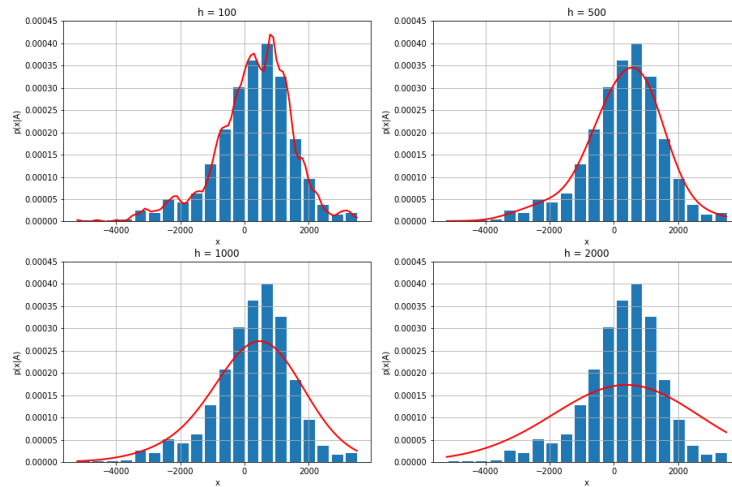


Figure 19: Influence of h for kernel density estimation (Spetrad, 2020).

The above figure shows that the estimation is too tight when the kernel size is small. In this case, the classification performance will be low for samples with high difference. Thus, the appropriate size will be the second one. Therefore, the optimal kernel size chosen in this experiment is proper to be used.

Fig 20 and Fig 21 show the visualized $P(w_k|x)$ for two high sensitive features when $h = 0.2$ and $h = 1.4$, respectively. Although $h = 1.4$ is not an accurate optimal kernel size for this case, it still shows the clear classification of three

classes, and similar to the actual relationship of petal length and petal width shown in Fig 2. Additionally, the average accuracy of 5-fold cross validation with $h = 1.4$ is also increased to 94.67% when the two less sensitive features, sepal length and sepal width, are removed. Note that the red denotes 'setosa', yellow denotes 'versicolor' and green denotes 'virginica' in these two figures.

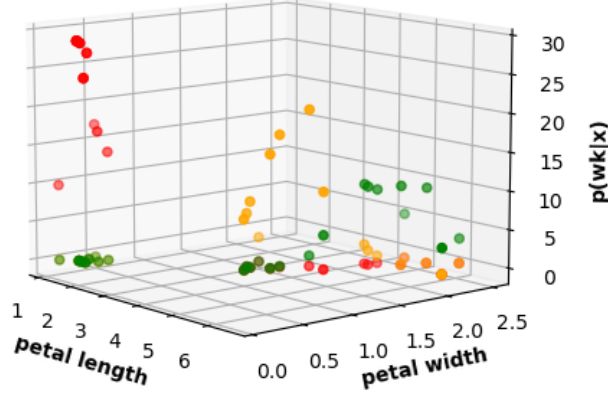


Figure 20: Parzen window when $h = 0.2$.

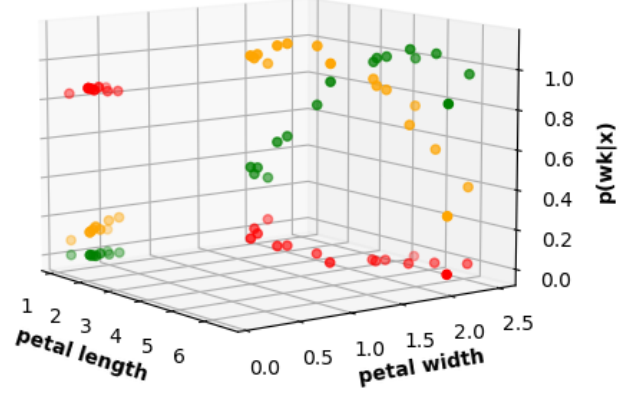


Figure 21: Parzen window when $h = 1.4$.

Now, the optimal value of k for MQDF2 can be also determined based on iterations from 1 to 3 with increment 1. Fig 22 shows the average accuracy of MQDF prediction when the value of δ_i is set based on equation 10. However, unlike the result shown in Kimura et al. (1987), the accuracy is decreased when $k = 2$. Since the difference of the designed algorithm in this paper and Kimura et al. (1987) is δ_i , the value of δ_i is set to a constant value. Then, the result can be obtained as Fig 23. Although the second result shows more close result to the paper, the first method is still theoretically more accurate because it can find the proper δ_i without experimental results. Thus, in this experiment, the proper value of k can be 1 because it has high accuracy and causes low computation time and storage than $k = 3$.

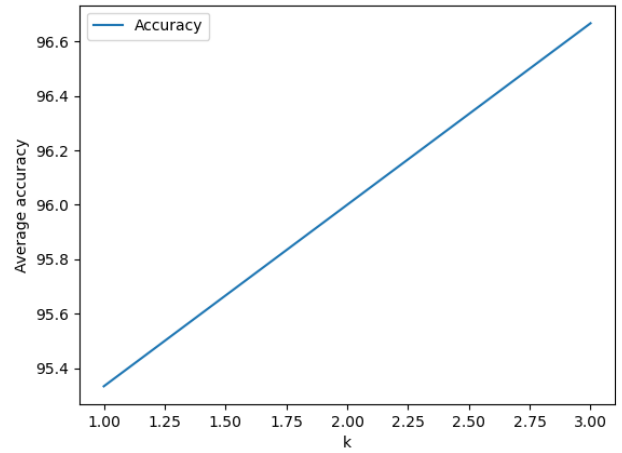
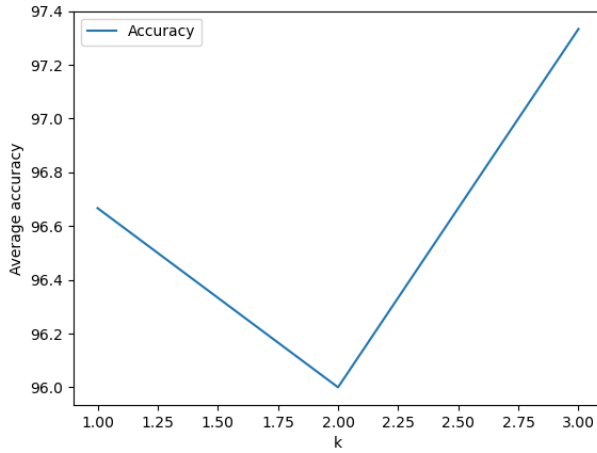


Figure 22: Average accuracy of MQDF with flexible δ_i . Figure 23: Average accuracy of MQDF with a constant δ_i .

The theoretical advantages of Parzen Window is already mentioned in the previous section as Non-parametric estimation. However, the disadvantage of this technique is that it will take high computation cost and time since it stores the whole training samples. On the other hand, MQDF2 will only store reduced training samples. Also, by applying RDA, MQDF3 can be properly used if the training samples are not enough compared to the test samples. Therefore, if the data is Gaussian distribution, MQDF will be a good choice for classification. However, although it is time-consuming, Parzen Window method will be a good technique if the underlying distribution is unknown.

In this report, by setting the iteration of h in Parzen Window equal to the iteration of k in MQDF, the running time of the code is measured to check the advantage of MQDF. Note that this result just shows the approximate result because

some self-defined functions could cause more time than others although the two techniques are designed similarly.

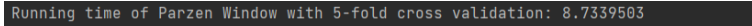


Figure 24: Running time of Parzen Window from $h = 1$ to $h = 3$.

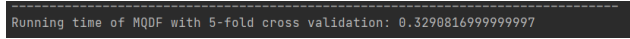


Figure 25: Running time of MQDF2 from $k = 1$ to $k = 3$.

From Fig 24 and Fig 25, it is clearly shown that the MQDF2 will take significantly lower time than Parzen Window. Therefore, it is verified that MQDF2 will take low computation time and storage. However, the low value of k will decrease the accuracy when the dimension is high. Thus, k should be determined according to the appropriate accuracy.

5 CONCLUSION

In this report, the two important techniques in pattern recognition were introduced. Parzen Window method is a non-parametric estimation technique which can be used without knowing the probability distribution or discriminant function. By using only the labeled data and kernel function, it can estimate the probability distribution. Thus, it is simple and easy to be implemented. However, since it should store the whole training dataset, it will consume long time and large storage.

On the other hand, MQDF is a parametric estimation technique that is a modified algorithm of QDF. Unlike QDF, it can reduce the running time and storage by replacing the small eigenvalues to a proper constant. Also, it can have low effect of the estimation error of non-dominant eigenvectors by using a pseudo-Bayesian estimate of the covariance matrix. Per the derived equations above, the classification can be conducted by calculated $g_2(x, w_i)$ in terms of k . However, this method is not suitable for real time-prediction, and k cannot be too low for high dimension dataset because it will influence the accuracy negatively.

The previous section shows the results of these two algorithms with detailed analysis. Firstly, the optimal kernel size h of Parzen Window was determined in Fig. 16 through the averaged log-likelihood function over 5-fold cross validation. And then, the average accuracy of Parzen Window in terms of h was provided in Fig. 17. However, these two figures illustrated that the optimal kernel size does not have the highest accuracy. Therefore, the detailed analysis was conducted via Fig. 19, and proved that the optimal kernel size $h = 1.4$ is appropriate. Following two figures, Fig. 20 and Fig. 21, also proved that the optimal kernel size can provide more similar result of the actual iris dataset. Thus, it is more proper to classify other unknown dataset with high variance.

After the analysis of Parzen Window, the results of MQDF illustrated the effect of k and δ_i to the accuracy. According to the results, it is clear that the accuracy will be generally increased when k increases. However, using the average of minor eigenvalues for δ_i , the accuracy could be changed per how proper value is chosen for δ_i . Lastly, the running time of two algorithms proved the advantage of MQDF2 to reduce the time consumption.

REFERENCES

- M. Brown. Parzen window, 1999. [Online: accessed 24-April-2022].
- J. Brownlee. Parametric and nonparametric machine learning algorithms, 2016. [Online: accessed 24-April-2022].
- S. K. Dash. A brief introduction to linear discriminant analysis, 2021. [Online: accessed 24-April-2022].
- F. Kimura, K. Takashina, S. Tsuruoka, and Y. Miyake. Modified quadratic discriminant functions and the application to chinese character recognition. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, (1):149–153, 1987.
- C. L. Liu, H. Sako, and H. Fujisawa. Discriminative learning quadratic discriminant function for handwriting recognition. *IEEE Trans. on Neural Networks*, 15(2):430–444, 2004.
- J. Sellner. Introduction to kernel density estimation (parzen window method), 2017. [Online: accessed 25-April-2022].
- Spetlrad. Non-parametric probability density estimation-parzen window, 2020. [Online: accessed 26-April-2022].
- D. Zhang. Exploring classifiers with python scikit-learn iris dataset, 2020. [Online: accessed 25-April-2022].

Appendices

A SELF-DEFINED FUNCTIONS IN PARZEN_WINDOW_MAIN.PY

```
def twoD_plot(filename): # To check the general properties of the dataset in 2D (
                          # Additional task)
    data = pd.read_csv(filename, names=["sepal length", "sepal width", "petal length",
                                         "petal width", "class"])
    data.head(5)
    data.describe()
    data.groupby('class').size()
    sns.pairplot(data, hue="class", height=2, palette='colorblind');
    plt.show()

def fourD_plot(p1, p2, p3, p4): # To check the general properties of the dataset
                                # in 4D (Additional task)
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    x = p1.astype(float) # sepal length
    y = p2.astype(float) # sepal width
    z = p3.astype(float) # petal length
    c = p4.astype(float) # petal width
    img = ax.scatter(x, y, z, c=c, cmap=plt.hot()) # The 4D datasets will be shown
                                                    # in the 3D coordinate with color gradient
    fig.colorbar(img)
    # Add axis
    ax.set_xlabel('sepal length', fontweight='bold')
    ax.set_ylabel('sepal width', fontweight='bold')
    ax.set_zlabel('petal length', fontweight='bold')
    plt.show()

def multivariate_normal_kernel(x, xi, h, cov): # Kernel function for datasets
                                                # higher than 1D
    det_cov = np.linalg.det(cov) # Determinant of the covariance matrix
    inv_cov = np.linalg.inv(cov) # Inverse of the covariance matrix
    u = np.matrix((x - xi)/h) # Compute the distance
    numer = pow(e, -0.5 * u * inv_cov * np.transpose(u)) # Numerator of the
                                                           # multivariate gaussian distribution
    denom = pow(pow(2*np.pi, len(x)) * det_cov, 1/2) # Denominator of the
                                                         # multivariate gaussian distribution
    return numer / denom

def normal_kernel(x, xi, h): # Kernel function for 1D datasets
    u = (x - xi) / h # Compute the distance
    return np.exp(-(np.abs(u) ** 2) / 2) / (np.sqrt(2 * np.pi))

def parzen_window(test, train, h, d): # Parzen Window function to output the
                                       # conditional probability
    cov = np.identity(d) # Create an identity matrix scaled with the dimension of
                           # the dataset for the covariance matrix
    if d == 1: # For 1D, apply the normal distribution
        p = normal_kernel(test, train, h) / h
        return p
    else: # For higher dimension, apply the multivariate normal distribution
        p = multivariate_normal_kernel(test, train, h, cov) / pow(h, d)
        return p

def pz_predict(x_len, np_array): # To find the predicted class of the test set
    x_pred = []
    for i in range(x_len):
        max = np.max(np_array[:,i]) # Get the maximum probability in the ith column (
                                     # ith data of x)
        if max == np_array[0][i]: # If 'max' is equal to the ith value in setosa array
```

```

    pred = 'Iris-setosa'
    elif max == np_array[1][i]: # If 'max' is equal to the ith value in versicolor
                                array
        pred = 'Iris-versicolor'
    else: # If 'max' is equal to the ith value in virginica array
        pred = 'Iris-virginica'
    x_pred.append(pred) # Store the predicted class in the order of the test
                        datasets

return x_pred

def pz_accuracy(pred_class, class_x): # To obtain the accuracy of the predicted
                                     result
    acc = 0 # Initialize the accuracy
    for ind, pred in enumerate(pred_class):
        if pred == class_x[ind]: # Compare the predicted classes with the actual
                                classes of the test set
            acc += 1 # Increase the accuracy parameter if it is correct
        else:
            pass # If not correct, pass
    return (acc / len(pred_class)) * 100

```

B CLASS IN PARZEN_WINDOW_MAIN.PY

```

class Data_process: # Class for data pre-processing
    def __init__(self):
        self.filename = "iris.data" # Dataset folder name
        # Predefined parameters
        self.line_comp = []
        self.iris_list = []

    def load_data(self): # Method to load the dataset and store them in a list
        with open(self.filename) as f:
            for line in f:
                text_lines = line.strip()
                line_comp = text_lines.split(',')
                self.iris_list.append(line_comp)
        del self.iris_list[-1] # Remove the empty element of the list
        return self.iris_list

    def shuffle(self): # Method to shuffle the stored dataset
        random.seed(97) # Define the seed value first to keep the shuffled data same
        random.shuffle(self.iris_list) # Shuffle the list
        return self.iris_list

    def separate_data(self): # Method to separate the dataset into five parts for 5-
                            fold cross validation
        length = int(len(self.iris_list) / 5) # Cutting length of the list
        data1 = self.iris_list[:length]
        data2 = self.iris_list[length:length * 2]
        data3 = self.iris_list[length * 2:length * 3]
        data4 = self.iris_list[length * 3:length * 4]
        data5 = self.iris_list[length * 4:length * 5]
        return data1, data2, data3, data4, data5

    def combine_train(self, ind, total_data): # Method to separate combined train
                                             sets and a test set
        train = []
        for i in range(len(total_data)): # According to the index, the test set will
                                         be chosen among the five subsets
            if ind == i:
                test = total_data[i]

```

```

        else:
            train += total_data[i]
        return train, test

def separate_class(self, dataset): # Method to separate dataset into three given
                                   classes

    setosa = []
    versicolor = []
    virginica = []
    for info in dataset:
        if info[4] == 'Iris-setosa':
            setosa.append(info)
        elif info[4] == 'Iris-versicolor':
            versicolor.append(info)
        else:
            virginica.append(info)
    return setosa, versicolor, virginica

def numeric_n_name(self, nested_list): # Method to separate the numeric data and
                                       class_names

    num_list = []
    class_list = []
    for instance in nested_list:
        num_data = instance[:4] # Extract the numeric data
        class_name = instance[4:] # Extract the class names of the data sets
        num_list.append(num_data)
        class_list += class_name
    return num_list, class_list # Numeric data can be converted into numpy array

def data_analyzer(self, info): # Method to plot the 2D and 4D figures of the
                               given dataset to analyze the properties

    np_info = np.array(info)
    sepal_length = np_info[:,0]
    sepal_width = np_info[:,1]
    petal_length = np_info[:,2]
    petal_width = np_info[:, 3]

    fourD_plot(sepal_length, sepal_width, petal_length, petal_width)
    twoD_plot(self.filename)

def prior_prob(self, dataset): # Method to calculate the prior probabilities of
                               each class

    prior_prob_se = len(dataset[0]) / (len(dataset[0]) + len(dataset[1]) + len(
        dataset[2])) # Setosa
    prior_prob_ve = len(dataset[1]) / (len(dataset[0]) + len(dataset[1]) + len(
        dataset[2])) # Versicolor
    prior_prob_vi = len(dataset[2]) / (len(dataset[0]) + len(dataset[1]) + len(
        dataset[2])) # Virginica
    return prior_prob_se, prior_prob_ve, prior_prob_vi

```

C SELF-DEFINED FUNCTIONS IN MQDF.PY

```

def QDF_model(train_data, class_num): # Modified function from QDF to obtain the
                                      required parameters for MQDF

    mean = [] # Initialize a list to store the mean
    cov_matrix = [] # Initialize a list to store covariance matrices
    for i in range(class_num): # For three classes
        train[i] = np.matrix(train_data[i], dtype=np.float64) # Convert the ith data
                                                                to a matrix & use 64-bit precision
        mean.append(train_data[i].mean(0).T) # Store the mean for four features in ith
                                                class

```

```

    # np.cov consider a row as one feature. Thus, train_data[i] should be
    # transposed
    cov_matrix.append(np.matrix(np.cov(train_data[i].T))) # Store the covariance
    # matrix of ith class

return mean, cov_matrix

def MQDF2_model(cov, d, k, class_num): # Function to obtain the trained parameters
    # of MQDF2
    eigenvalue = [] # Initialize a list to store eigenvalues
    eigenvector = [] # Initialize a list to store eigenvectors
    delta = [0] * class_num # Each delta value of classes will be stored here
    for i in range(class_num): # For three classes
        covMat = cov[i] # Get the covariance matrix of ith class
        eigvals, eigvecs = np.linalg.eig(covMat) # Obtain eigenvalues and eigenvectors
        # from the cov. matrix

        # Disordering the eigenvalues
        id = eigvals.argsort() # Get the ascending order of the eigenvalue indexes
        id = id[::-1] # Convert id to the descending order of the eigenvalues
        eigvals = eigvals[id] # Descending the eigenvalues
        eigvecs = eigvecs[:, id] # Descending the eigenvectors

        eigenvector.append(eigvecs[:, 0:int(k)]) # Store the eigenvectors from j=1 to
        # k
        eigenvalue.append(eigvals[0:int(k)]) # Store the eigenvalues from j=1 to k
        # Compute delta as the mean of minor values
        delta[i] = sum(eigvals[int(k):]) / (d - k) # Recommended method from Moghaddam
        # and Pentland
        # delta[i] = 0.2 # constant delta to check the influence of delta on the
        # accuracy (theoretically not appropriate)

    return eigenvalue, eigenvector, delta

def predict_MQDF2(d, np_x, class_num, k, mean, eigenvalue, eigenvector, delta): #
    # Function to perform classification based on
    # the MQDF2 trained parameters
    assert (k < d and k > 0) # Assertion error when k greater or equal to d and
    # negative k
    pred_label = [] # Initialize a list to store the predicted classes
    for sample in np_x: # For the number of test samples
        test_x = np.matrix(sample, np.float64).T # Convert a sample data to a matrix
        max_g2 = -float('inf') # The initial value of max_g2 is set to the negative
        # infinity
        for i in range(class_num): # For three classes
            dis = np.linalg.norm(test_x.reshape((d,)) - mean[i].reshape((d,))) ** 2 #
            # Compute the distance between the sample
            # data and the mean, and then square it
            # Second term of the residual of subspace projection
            euc_dis = [0] * int(k) # Initialization
            ma_dis = [0] * int(k) # Initialization
            for j in range(int(k)): # For the range of k
                euc_dis[j] = ((eigenvector[i][:, j].T * (test_x - mean[i]))[0, 0]) ** 2
                ma_dis[j] = (((test_x - mean[i]).T * eigenvector[i][:, j])[0, 0]) ** 2

            g2 = 0 # Initialize the MQDF2
            for j in range(int(k)): # For the range of k
                # Firstly, compute the terms including j and add them to g2
                g2 += (euc_dis[j] * 1.0 / eigenvalue[i][j]) + math.log(eigenvalue[i][j])

            # Secondly, compute the terms only including i and add them to g2
            g2 += ((dis - sum(ma_dis)) / delta[i]) + ((d - int(k)) * math.log(delta[i]))
            g2 = -g2 # Convert the g2 values to minus to find the maximum value

            if g2 > max_g2: # If the current g2 > previous max g2
                max_g2 = g2 # Replace the g2 value
                prediction = i # Store the class id of current g2
            elif g2 == max_g2:

```

```

        print(i, "==", prediction) # Error if two g2 values are equal
    else:
        pass # Ignore current g2 if it's smaller than max_g2
    pred_label.append(prediction) # After for loop, append the current max g2
                                   class id
return pred_label

def mqdf_accuracy(predic, class_x): # Function to calculate the prediction
                                   accuracy of MQDF2
    conv_pred = []
    for pred in predic: # For the predicted id numbers of test dataset
        if pred == 0: # If the predicted value is '0'
            conv_pred.append('Iris-setosa') # It is setosa
        elif pred == 1: # If it is '1'
            conv_pred.append('Iris-versicolor') # It is versicolor
        elif pred == 2: # If it is '2'
            conv_pred.append('Iris-virginica') # It is virginica
        else: # Out of range, there is an error
            print('Wrong prediction') # Error when index out of the range

    accuracy = pz_accuracy(conv_pred, class_x) # Accuracy can be calculated using '
                                              pz_accuracy()'
    return accuracy

```

D MAIN PART OF MQDF.PY

```

if __name__ == '__main__':
    iris = Data_process() # Define class
    irist_data = iris.load_data() # Load the iris dataset
    div_data = iris.numeric_n_name(irist_data) # Separate numeric dataset and class
                                              names
    init_data = iris.shuffle() # Shuffle the dataset
    five_data = iris.separate_data() # Divide the dataset for 5-fold cross
                                    validation

    # Predefined parameters
    classNum = 3 # Three classes
    d = len(div_data[0][0]) # Feature numbers = 4

    k_list = []
    mqdf_acc_list = []

    start = timeit.default_timer() # To measure the running time of MQDF2, start
                                    timer
    for k in range(1, d): # 'k' should be always 'integer' and smaller than d
        mqdf_sum_avg_acc = 0
        cnt = 1
        for index in range(len(five_data)): # 5-fold cross validation
            total_subset = iris.combine_train(index, five_data) # Index denotes the
                                                                array for testing
            sep_dataset = iris.separate_class(total_subset[0]) # Return separated train
                                                                datasets by three classes
            sep_data = [sep_dataset[0], sep_dataset[1], sep_dataset[2]] # Nested list of
                                                                the three datasets

            # Only extract the numeric data from the datasets
            np_se = np.array(iris.numeric_n_name(sep_data[0])[0])
            np_ver = np.array(iris.numeric_n_name(sep_data[1])[0])
            np_vir = np.array(iris.numeric_n_name(sep_data[2])[0])
            # Prepare the train dataset by converting the numbers in 'str' to 'float'
            train = [np_se.astype(float), np_ver.astype(float), np_vir.astype(float)]

```

```

mean, cov = QDF_model(train, classNum) # Obtain the mean and covariance
                                         matrices

eigval, eigvec, delta = MQDF2_model(cov, d, k, classNum) # Obtain the
                                                         eigenvalues, eigenvectors and delta

# mean, eigenvalues, eigenvectors, k and delta will be the trained
# parameters of MQDF2 for prediction
print('Training process of the MQDF model finished.')

# Prepare the test dataset
x = np.array(iris.numeric_n_name(total_subset[1])[0]) # numeric data of
                                                         test set

np_x = x.astype(float)
x_len = len(np_x)

class_x = iris.numeric_n_name(total_subset[1])[1] # Real class names of
                                                    each test set

predic = predict_MQDF2(d, np_x, classNum, k, mean, eigval, eigvec, delta) #
Input the trained parameters to predict

MQDF_accuracy = mqdf_accuracy(predic, class_x) # Based on the prediction
result, compute the classification accuracy
mqdf_sum_avg_acc += MQDF_accuracy
print(cnt, 'th accuracy:', MQDF_accuracy)
cnt += 1

MQDF_avg_acc = mqdf_sum_avg_acc / len(five_data) # Calculate the average
accuracy of 5-fold cross validation
print('Averay accuracy of 5-fold cross-validation when k =', k, ':',
      MQDF_avg_acc)

print("_____")

k_list.append(k)
mqdf_acc_list.append(MQDF_avg_acc)
stop = timeit.default_timer() # Stop timer
print('Running time of MQDF with 5-fold cross validation:', stop - start) #
Running time of MQDF2

```