Data Mining and Machine Learning - Exercise 3

Johan Rodhe

October 6, 2016

1

1.1

Generating data points was something I struggled a bit with in python in the first exercise. Now it all seems so simple. The code can be seen in Figure 1 below. It is pretty much the same as in previous exercises. 100 points from class 1 and 100 points from class 2 was generated.

```
7   def loadData(N):
8       mu1 = [1, 0.5]
9       sigma1 = [[1, 0.5], [0.5, 1]]
10      mu2 = [-1, -0.5]
11      sigma2 = [[1, -0.5], [-0.5, 1]]
12      x1 = np.random.multivariate_normal(mu1, sigma1, 100)
13      x2 = np.random.multivariate_normal(mu2, sigma2, 100)
14
15      return x1, x2
```

Figure 1: Code for generating the points. The library numpy was imported as np.

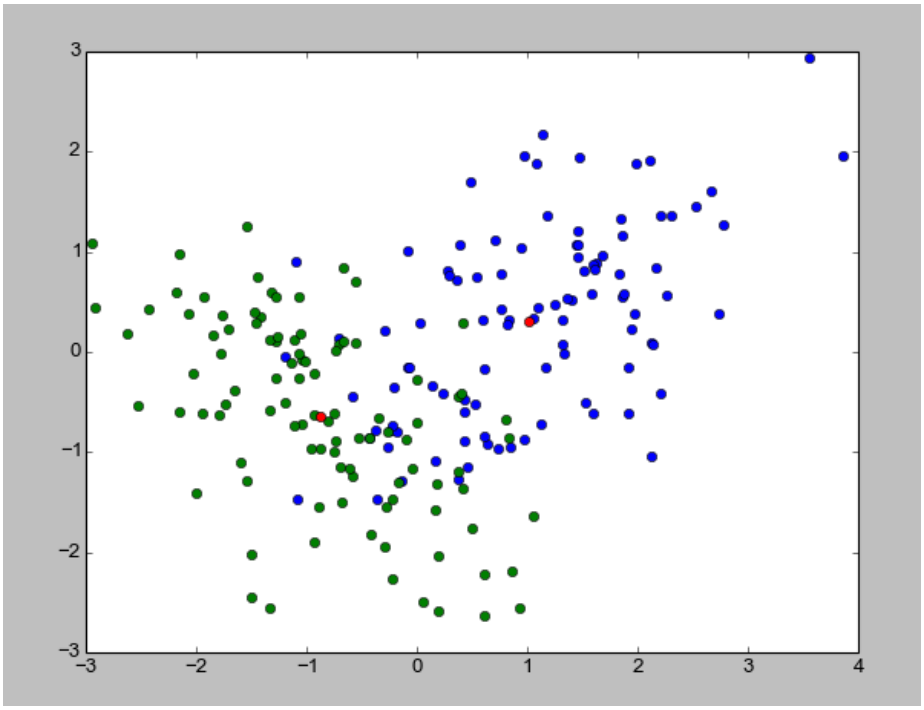

Figure 2: Plot of the points. Blue points belonging to class 1 and green to class 2. The red dots are the means.

Some clustering can be seen in around the means in Figure 2. The red dot to the left is the mean for class 2 and the right for class 1.

The covariance was not added to the plot since I could not figure out how to visually represent it. I found a way to draw an ellipse of the covariance matrix error using eigenvalues and eigenvectors. But since this is done later in the exercise I assumed this was not what you were asking for.

1.2

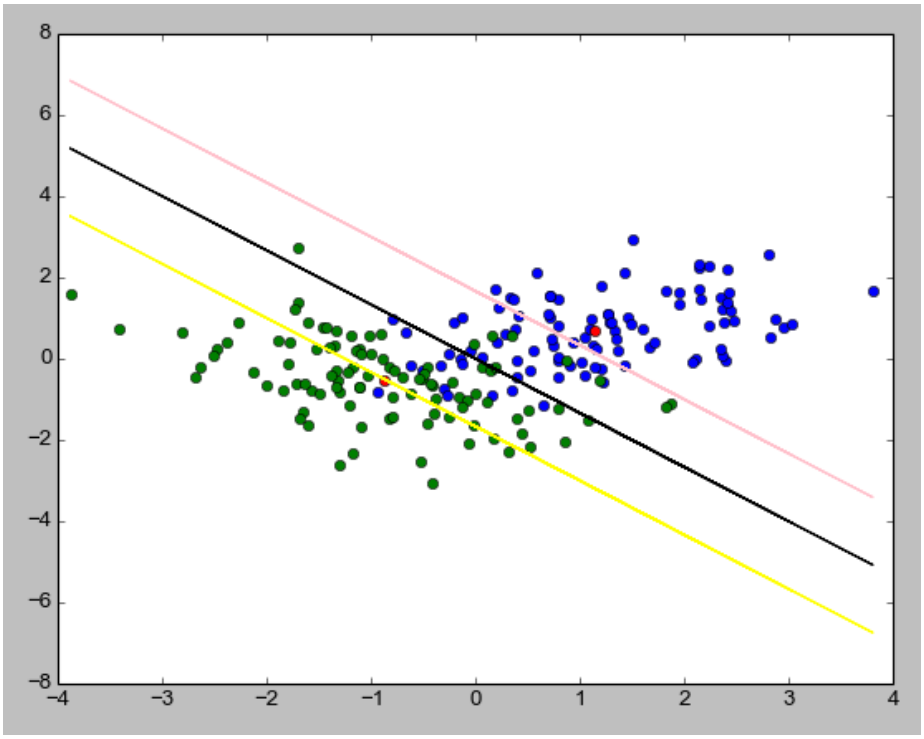

Figure 3: The three decision boundaries where $w_0 = 0$ is the black line, $w_0 = 1$ the yellow one and $w_0 = -1$ the pink.

w_0	class 1 misclassifications	class 2 misclassifications	total misclassifications
0	13	6	19
1	3	47	50
-1	41	0	41

Table 1: Table of misclassification rates

From Table 1 we can see that the total misclassifications is the lowest when $w_0 = 0$. This would seem like the best choice. In a scenario where it is very important not to misclassify a point from class 2 but not so important if a point from class 1 is misclassified $w_0 = -1$ would be a much better decision boundary.

Below in Figure 4 is the code for the discriminant functions.

```
18  def lda(x1, x2, w):
19      x = np.concatenate((x1, x2), axis=0)
20      w0 = 0
21      w1 = 1
22      w2 = -1
23      error1 = 0
24      error2 = 0
25      y = -(w[0] * x + w0) / w[1]
26      y1 = -(w[0] * x + w1) / w[1]
27      y2 = -(w[0] * x + w2) / w[1]
28      for i in range(len(x1)):
29          p = np.dot(np.array(w), np.array(x1[i])[np.newaxis].T)
30          if p < -w0: error1 += 1
31          p = np.dot(np.array(w), np.array(x2[i])[np.newaxis].T)
32          if p > -w0: error2 += 1
33      print error1, error2
34      return x, y, y1, y2
```

Figure 4: The three different decision boundaries are returned.

1.3

Using Fisher's criterion on the 2 dimensional input vectors the goal is to reduce them to one dimension using:

$$y = \mathbf{w}^T \mathbf{x} \tag{1}$$

To classify a threshold is placed on y and if $y \geq -w_0$ then it is classified as C_1 and otherwise as C_2. The weight vector $\mathbf{w}$ is adjusted so a projection with maximized class separation is selected. The idea is to maximize a function that will give a large separation between classes but a small variance within the classes. The within class covariance matrix is given by the sum of the covariance matrices of the two classes.

$$S_W = \sum_{n \in C_1} (x_n - m_1)(x_n - m1)^T + \sum_{n \in C_2} (x_n - m_2)(x_n - m_2)^T = S_1 + S_2 \tag{2}$$

where $m_1 and m_2$ are the means of the two classes. The between class covariance is given by:

$$S_B = (m_2 - m_1)(m_2 - m_1)^T \tag{3}$$

The solution is then obtained from the so called generalized eigen-value problem:

$$S_W^{-1} S_B \mathbf{w} = \lambda I \tag{4}$$

```
37  def fisher(x1, x2):
38      mean1 = x1.mean(axis=0)[np.newaxis].T
39      mean2 = x2.mean(axis=0)[np.newaxis].T
40      cov_matrix1 = np.cov(x1, rowvar=0)
41      cov_matrix2 = np.cov(x2, rowvar=0)
42      within_cov_matrix = cov_matrix1 + cov_matrix2
43      between_cov_matrix = np.dot((mean1 - mean2), (mean1 - mean2).T)
44      A = np.dot(la.inv(within_cov_matrix), between_cov_matrix)
45      eig_values, eig_vectors = la.eig(A)
46      eig_pairs = [(np.abs(eig_values[i]), eig_vectors[:,i]) for i in range(len(eig_values))]
47      eig_pairs = sorted(eig_pairs, key=lambda k: k[0], reverse=True)
48      eig_vec1 = np.array(eig_pairs[0][1])
49
50      return eig_vec1
```

Figure 5: Code for Fisher's criterion to get weights. Line 38-43 is calculating the means and covariance matrices. 44-48 is solving the generalized eigen-value problem. The function returns the eigen vector corresponding to the highest eigen value.

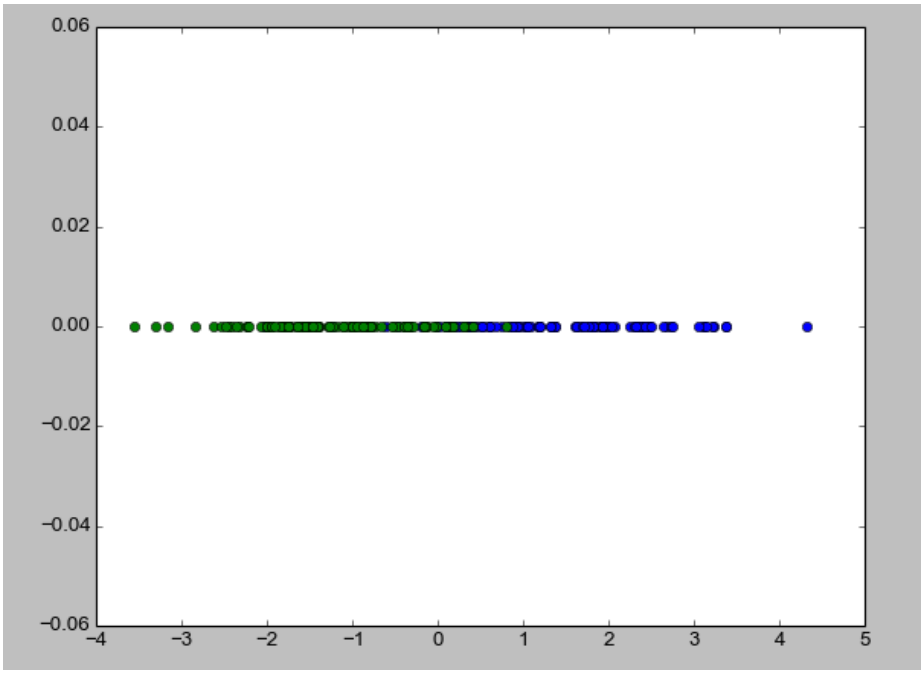

Figure 6: The input vectors reduced to one dimension using Fisher's criterion. To calculate the misclassification rate we need to first construct a discriminant from this by adding a threshold y_0 so that if $y(x) \geq y_0$ then it belongs to C_1 and otherwise to C_2. For example the misclassification rate if we use $y_0 = 0$ is 13 for class 1 and 6 for class 2.

1.4

```
8  def loadIris():
9      data = pd.read_csv(filepath_or_buffer='iris.data',  header=None,  sep=',')
10
11     data.columns=['sepal_len', 'sepal_wid', 'petal_len', 'petal_wid', 'class']
12     data.dropna(how="all", inplace=True)
13
14     data.tail()
15     feature_matrix = data.ix[0:99,0:4].values
16     labels = data.ix[0:99, 4].values
17     return feature_matrix, labels
```

Figure 7: The Iris dataset was loaded using the pandas module.

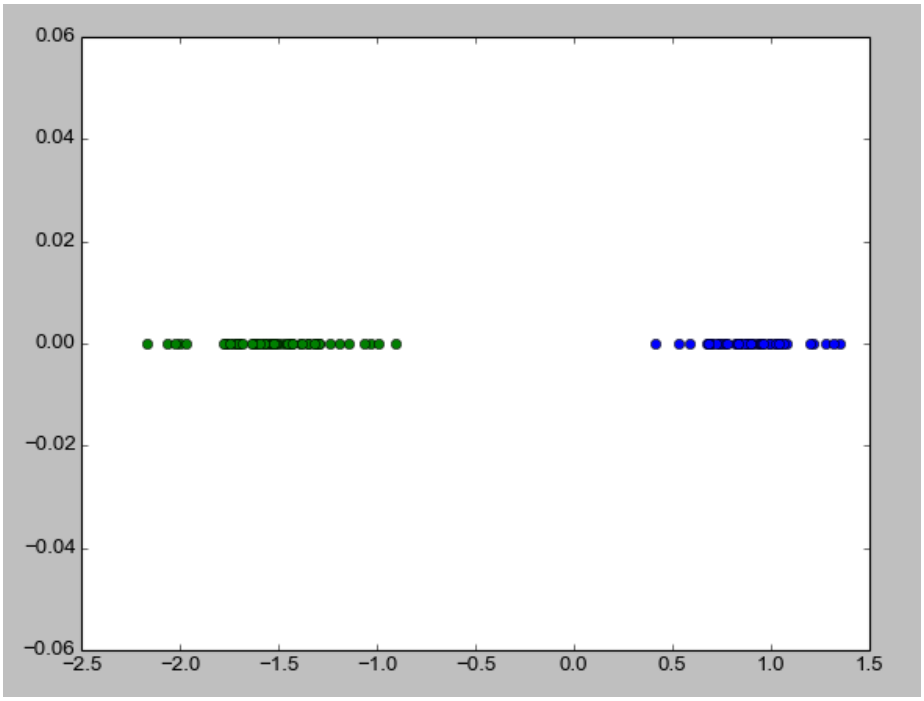

Figure 8: The result of applying the method to the Iris dataset. Green dots represent the Iris-versicolor class, and blue dots Iris-setosa.

From the figure it is clear that the dimensionality reduction using Fisher's criterion is effective. If a threshold $y_0 = 0$ is added then the classifier will classify the points with 0 misclassification rate.

If we want to add another class the discriminant function has to change. We instead get:

$$y_k(x) = \mathbf{w}_k^T \mathbf{x} + w_{k0} \tag{5}$$

where k = 1, 2, 3. Then we assign $\mathbf{x}$ to C_k if $y_k(x) < y_j(x)$ for all $j \neq k$.

2

2.1

The discriminant function for Normal density functions is:

$$g_i(x) = -\frac{1}{2}(\mathbf{x} - m_i)^T \Sigma_i^{-1}(\mathbf{x} - m_i) - \frac{d}{2} ln 2\pi - \frac{1}{2} ln|\Sigma_i| + lnP(w_i) \tag{6}$$

If we assume that the prior probabilities $P(w_i)$ are the same for all classes then the term $lnP(w_i)$ can be ignored. This assumption was made when constructing the three different classifiers.

- Case 1: $\Sigma_i = \sigma^2 \mathbf{I}$
  In this case $|\Sigma_i| = \sigma^{2d}$ and $\Sigma_i^{-1} = \frac{1}{\sigma^{2d}}\mathbf{I}$ and the $\frac{d}{2} ln 2\pi$ term is not dependent on i they can be ignored. When we have all of these conditions the problem of classifying becomes simple. We compute the euclidean distance from each $\mathbf{x}$ to the class mean vectors and assign it to the class of the nearest mean.

```
52      for i in range(len(testSet)):
53          e_dist1 = euclideanDistance(testSet[i], mean_vector1)
54          e_dist2 = euclideanDistance(testSet[i], mean_vector2)
55          e_dist3 = euclideanDistance(testSet[i], mean_vector3)
56          if e_dist1 < e_dist2 and e_dist1 < e_dist3: y_pred.append("Iris-setosa")
57          elif e_dist2 < e_dist1 and e_dist2 < e_dist3: y_pred.append("Iris-versicolor")
58          elif e_dist3 < e_dist2 and e_dist3 < e_dist1: y_pred.append("Iris-virginica")
59
```

Figure 9: The minimum distance classifier. The function euclideanDistance is the same on as used in earlier exercises.

- Case 2: $\Sigma_i = \Sigma$
  We now instead have the discriminant function:

  $$g_i(x) = -\frac{1}{2}(\mathbf{x} - m_i)^T \Sigma^{-1}(\mathbf{x} - m_i) \tag{7}$$

  The covariance matrix is calculated with:

  $$S = \frac{N_1}{N} S_1 + \frac{N_2}{N} S_2 \tag{8}$$

  Once again the problem of classifying can be stated fairly simple. For each $\mathbf{x}$ calculate the Mahalanobis distance $(\mathbf{x} - m_i)^T \Sigma^{-1}(\mathbf{x} - m_i)$ to each of the mean vectors of the classes.

```
    S = (1/3) * cov1 + (1/3) * cov2 + (1/3) * cov3
    for i in range(25):
        y_actual.append("Iris-setosa")
    for i in range(25):
        y_actual.append("Iris-versicolor")
    for i in range(25):
        y_actual.append("Iris-virginica")
    for i in range(len(testSet)):
        m_dist1 = mahalanobisDistance(testSet[i], mean_vector1, S)
        m_dist2 = mahalanobisDistance(testSet[i], mean_vector2, S)
        m_dist3 = mahalanobisDistance(testSet[i], mean_vector3, S)
        if m_dist1 < m_dist2 and m_dist1 < m_dist3: y_pred.append("Iris-setosa")
        elif m_dist2 < m_dist1 and m_dist2 < m_dist3: y_pred.append("Iris-versicolor")
        elif m_dist3 < m_dist2 and m_dist3 < m_dist1: y_pred.append("Iris-virginica")
```

Figure 10: The linear Gaussian classifier. S is the covariance matrix calculated with (8). Then the Mahalanobis distance to the mean vectors is calculated and x is assigned to the class with the closest mean vector.

```
def mahalanobisDistance(x, mean, cov):
    diff = x - mean
    m_dist = np.dot(np.dot(diff.T, la.inv(cov)), diff)

    return m_dist
```

Figure 11: The function for calculating Mahalanobis distance.

- Case 3: $\Sigma_i = arbitrary$
  Now the only term that can be ignored is $\frac{d}{2}ln2\pi$. Our discriminant function is in this case:

$$g_i(x) = \mathbf{x}^T\mathbf{W}_i\mathbf{x} + \mathbf{w}_i^T\mathbf{x} + w_{i0} \tag{9}$$

  where

$$\mathbf{W}_i = -\frac{1}{2}\Sigma_i^{-1} \tag{10}$$

  and

$$\mathbf{w}_i = \Sigma_i^{-1}\mathbf{m}_i \tag{11}$$

```
    W1 = -(1/2) * la.inv(cov1)
    W2 = -(1/2) * la.inv(cov2)
    W3 = -(1/2) * la.inv(cov3)
    w1 = np.dot(la.inv(cov1), mean_vector1)
    w2 = np.dot(la.inv(cov2), mean_vector2)
    w3 = np.dot(la.inv(cov3), mean_vector3)
    w1_0 = -(1/2) * np.dot(np.dot(mean_vector1.T, la.inv(cov1)), mean_vector1)
    w2_0 = -(1/2) * np.dot(np.dot(mean_vector2.T, la.inv(cov2)), mean_vector2)
    w3_0 = -(1/2) * np.dot(np.dot(mean_vector3.T, la.inv(cov3)), mean_vector3)
    for i in range(len(testSet)):
        g1 = np.dot(np.dot(testSet[i].T, W1), testSet[i]) + np.dot(w1.T, testSet[i]) + w1_0
        g2 = np.dot(np.dot(testSet[i].T, W2), testSet[i]) + np.dot(w2.T, testSet[i]) + w2_0
        g3 = np.dot(np.dot(testSet[i].T, W3), testSet[i]) + np.dot(w3.T, testSet[i]) + w3_0
        if g1 > g2 and g1 > g3: y_pred.append("Iris-setosa")
        elif g2 > g1 and g2 > g3: y_pred.append("Iris-versicolor")
        elif g3 > g2 and g3 > g1: y_pred.append("Iris-virginica")
```

Figure 12: The quadratic Gaussian classifier for three classes. Equations (9), (10), (11) are used to calculate the discriminant function.

2.2

As can be seen in Table 2 below the training set misclassification is fairly low for all the classes. Especially the Linear- and Quadratic Gaussian classifiers. It can be said that the classifiers are doing well. But it does not mean that the classifiers will do well with other data since the classifiers was here tested on the same data that was used to construct them. So it does not say anything about how the classifiers are doing in general.

$Classifier$	Class 1	Class 2	Class 3	Total
Minimum Distance	0	1	5	6
Linear Gaussian	0	1	0	1
Quadratic Gaussian	0	1	0	1
Rate	0	$\frac{3}{25}$	$\frac{5}{25}$	$\frac{8}{25}$

Table 2: Table of training set misclassification rates for the three different classifiers.

2.3

For the training set I use the first 25 lines of each class. For the test set I use the other 25 lines. I created confusion tables using the pandas module.

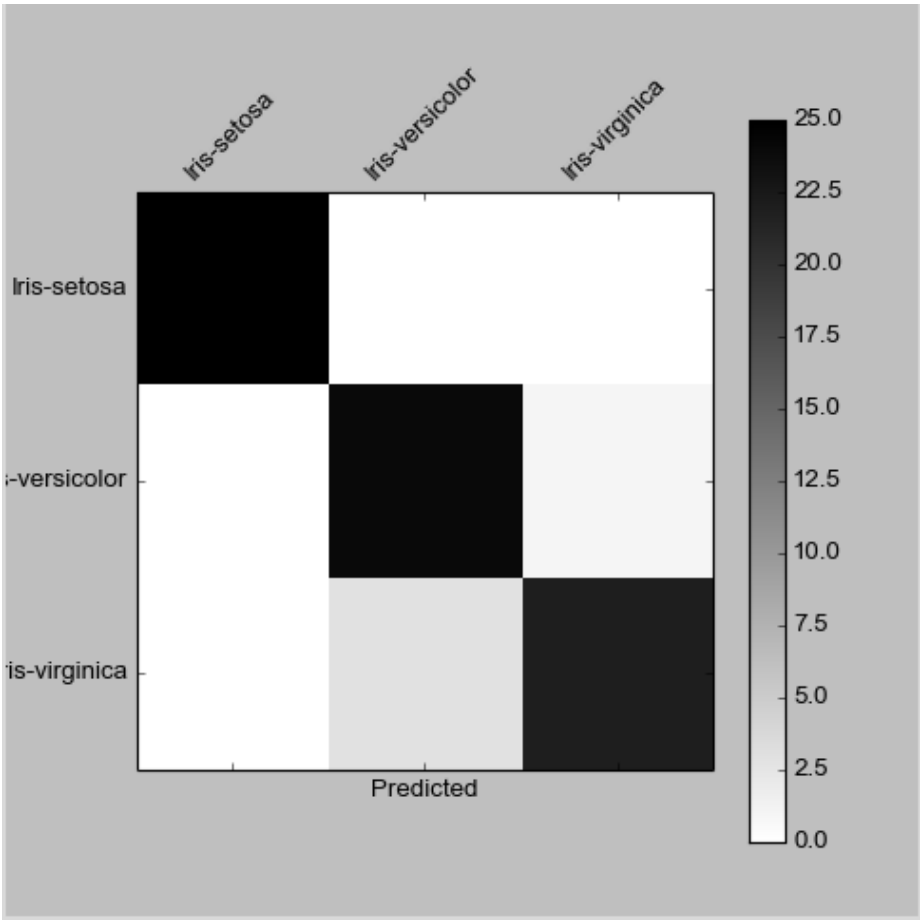

Figure 13: Confusion table for the minimum distance classifier. Actual values are seen on the right-hand side and predicted on the top.

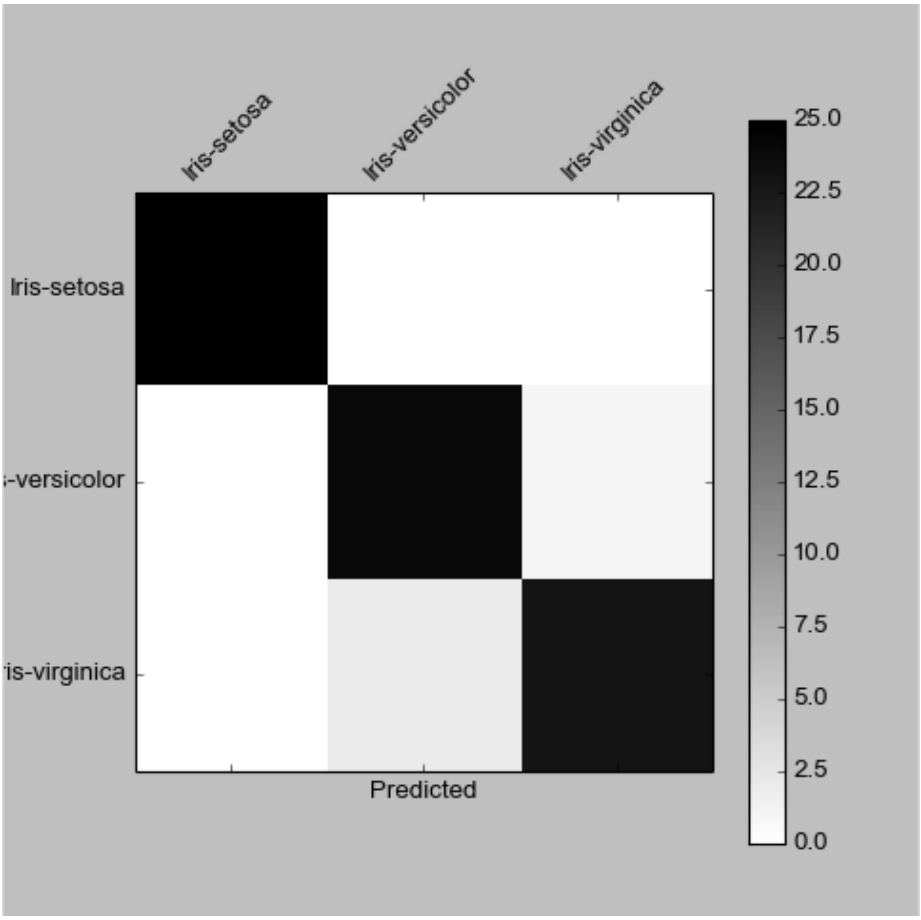

Figure 14: Confusion table for the linear Gaussian classifier.

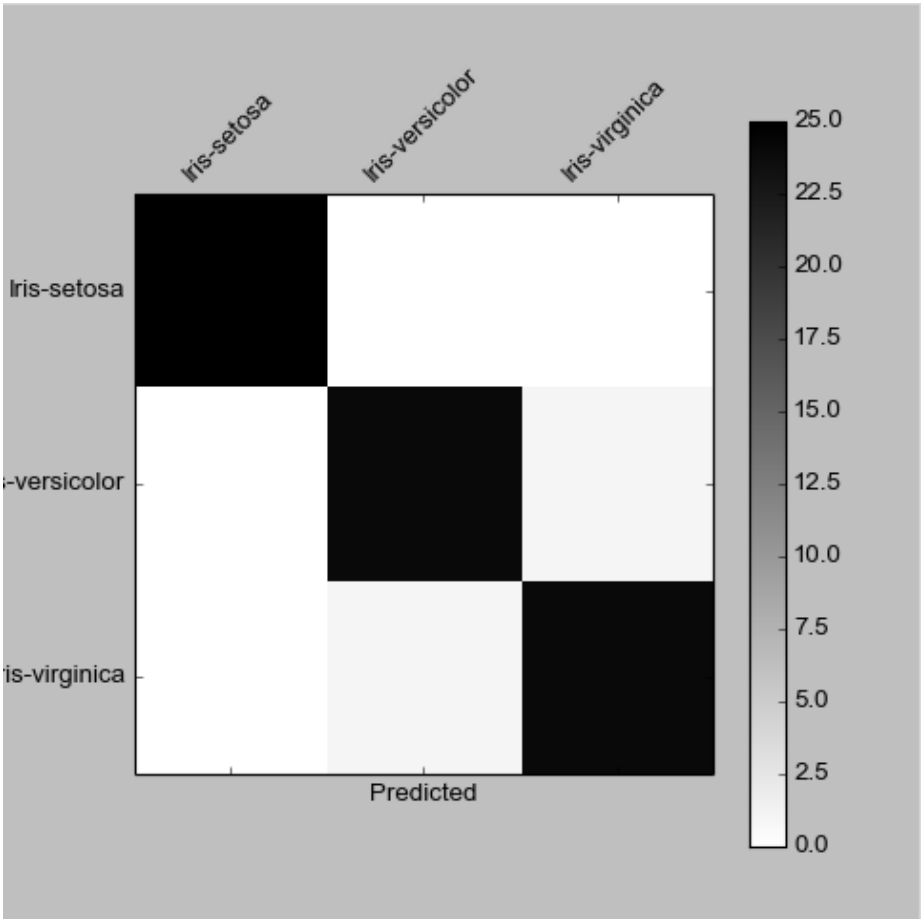

Figure 15: Confusion table for the quadratic Gaussian classifier.

It is a little bit hard to see the difference since the misclassification rate is relatively low for all the classifiers so I will produce a table as well. I just thought they looked nice.

Predicted / *Actual*	Iris-setosa	Iris-versicolor	Iris-virginica	Misclassification rate
Iris-setosa	25	0	0	0
Iris-versicolor	0	24	1	$\frac{1}{25}$
Iris-virginica	0	3	22	$\frac{3}{25}$

Table 3: Confusion table for minimum distance classifier.

Total misclassification rate: $\frac{4}{75}$

Predicted / *Actual*	Iris-setosa	Iris-versicolor	Iris-virginica	Misclassification rate
Iris-setosa	25	0	0	0
Iris-versicolor	0	24	1	$\frac{1}{25}$
Iris-virginica	0	2	23	$\frac{2}{25}$

Table 4: Confusion table for linear Gaussian classifier.

Total misclassification rate: $\frac{3}{75}$

Predicted / *Actual*	Iris-setosa	Iris-versicolor	Iris-virginica	Misclassification rate
Iris-setosa	25	0	0	0
Iris-versicolor	0	24	1	$\frac{1}{25}$
Iris-virginica	0	1	24	$\frac{1}{25}$

Table 5: Confusion table for quadratic Gaussian classifier.

Total misclassification rate: $\frac{2}{75}$

From the tables above the test set misclassification rates for each class can be seen. We notice that the test set errors is not very different from the training set errors which means that the classifiers generalized very well.

3

Title: A decision support system for detecting products missing from the shelf based on heuristic rules

Authors: Dimitrios Papakiriakopoulos, Katerina Pramatari, Georgios Doukidis

For a store to build customer loyalty and to keep their customers satisfied product availability is a critical issue. It increases sales and profitability. The term "out-of-shelf" is used in grocery retailing to describe when a customer does not find the product he or she is looking for on the shelf of a supermarket. It is a more broad term then "out-of-stock". When the "out-of-shelf" situation occurs a customer might chose to go to a different store to buy the product. The out-of-stock problem has been more widely studied and out-of-shelf problem is no less harmful to the business of the supermarket.

So far when a product is out-of-shelf it is first noticed by staff walking around the store and checking. Obviously this is expensive and stores would want a more effective way of doing it. The study proposes to use a decision support system for automatic detection of out-of-shelf products.

The idea is to discriminate the products into two sets corresponding to the availability (EXISTS or OOS(out-of-shelf)). To do this algorithms such as Naive Bayes has been developed to classify new examples based on patterns from earlier known cases.

The study proposes a decision system based on heuristic rules and the system actually compares different types of classifiers such as neural but I will focus on describing how the Naive Bayes classifier is used in the process.

To develop the rules they use an iterative process consisting of three steps.

- Physical Audit: A researcher collects data by going in to stores on different occasions to detect OOS situations.
- Building Classifiers: The classifier gets known examples classified as EXISTS or OOS as input. From this it makes a classification model that classifies products based on their attributes.
- Validation: The last step validates the classification model. Here the accuracy of the model is examined. Both how accurate the model was for available examples and how effective it is for new and unclassified examples.

The result and conclusions of the study show that even though the OOS-problem is very difficult to detect automatically the suggested decision system provides good results. But there is a trade-off situation between support and accuracy. To reach 90% accuracy the system only detects one third of the OOS situations. The study also notes that when following the same procedure on a different retail store the same classifiers was not the result. This suggests that the same system can not be used for different retail chains.