Machine Learning HW2 – Linear Models for Classification

NCTU EE 0310128 游騰杰

Programming language: Python

Environment: Python 2.7.13 under Homebrew @ Mac OS X 10.11.6

Python module used: numpy, pandas, math, Pillow, random

Plot: Matlab

1. Use the training images to build “Multi-class Probabilistic Generative Model” and “Multi-class Probabilistic Discriminative Model”.

(1) Multi-class Probabilistic Generative Model:

Formula [Reference] Textbook P.198 equation (4.62)

Multi-class generalization sigmoid function (normalized exponential)

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{\sum_j p(\mathbf{x}|\mathcal{C}_j)p(\mathcal{C}_j)} = \frac{\exp(a_k)}{\sum_j \exp(a_j)} \tag{4.62}$$

```
214
215 def classify_image(attributes, inputVector, prior):
216     # inputVector is a row vector with dimension equal to pca dimension
217     # i.e. a image vector reduced by pca
218     # calClassProbability
219     numClass = len(attributes)
220     dimension = len(inputVector)
221
222     probabilities = []
223     for itr_class in range(numClass):
224         tmp_probability = 1
225
226         for itr_dim in range(dimension):
227             mean, stdev = attributes[itr_class][itr_dim]
228             x = inputVector[itr_dim]
229             #print "Class: {:1d} | Dim: {:1d} | mean = {:4.3f}, std = {:4.3f}" \
230             #     .format(itr_class + 1, itr_dim + 1, mean, stdev)
231             tmp_probability *= gaussian_probability(x, mean, stdev)
232
233         probabilities.append(tmp_probability * abs(math.log(prior[itr_class])))
234         #probabilities.append(tmp_probability)
235     # makePrediction
236     return np.array(probabilities).argmax()
237
```

Formula [Reference] Textbook P.198 equation (4.64)

Class-conditional densities [under assumption of Gaussian distribution] of class C_k

$$p(\mathbf{x}|\mathcal{C}_k) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\mathbf{\Sigma}|^{1/2}} \exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_k)^{\mathrm{T}}\mathbf{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_k)\right\}. \tag{4.64}$$

```
191
192 def gaussian_probability(x, mean, stdev):
193     exponent = math.exp(-(math.pow(x - mean,2) / (2 * math.pow(stdev,2))))
194
195     return (1 / (2 * math.pi * stdev)) * exponent
196
```

Formula [Reference] Textbook P.202 equation (4.81)

Naïve Bayes class-conditional distributions

$$p(\mathbf{x}|\mathcal{C}_k) = \prod_{i=1}^{D} \mu_{ki}^{x_i}(1-\mu_{ki})^{1-x_i} \tag{4.81}$$

```
def classify_image(attributes, inputVector, prior):
    # inputVector is a row vector with dimension equal to pca dimension
    # i.e. a image vector reduced by pca
    # calClassProbability
    numClass = len(attributes)
    dimension = len(inputVector)

    probabilities = []
    for itr_class in range(numClass):
        tmp_probability = 1

        for itr_dim in range(dimension):
            mean, stdev = attributes[itr_class][itr_dim]
            x = inputVector[itr_dim]
            #print "Class: {:1d} | Dim: {:1d} | mean = {:4.3f}, std = {:4.3f}" \
            #    .format(itr_class + 1, itr_dim + 1, mean, stdev)
            tmp_probability *= gaussian_probability(x, mean, stdev)

        probabilities.append(tmp_probability * abs(math.log(prior[itr_class])))
        #probabilities.append(tmp_probability)
    # makePrediction
    return np.array(probabilities).argmax()

```

(2) Multi-class Probabilistic Discriminative Model:

Formula: [Reference] Textbook P.207 equation (4.92)

Newton-Raphson iterative optimization

$$\mathbf{w}^{(\text{new})} = \mathbf{w}^{(\text{old})} - \mathbf{H}^{-1}\nabla E(\mathbf{w}). \tag{4.92}$$

Formula: [Reference] Textbook P.208 equation (4.100)

z is an N-dimensional vector with elements

$$\mathbf{z} = \mathbf{\Phi}\mathbf{w}^{(\text{old})} - \mathbf{R}^{-1}(\mathbf{y} - \mathbf{t}). \tag{4.100}$$

Formula: [Reference] Textbook P.209 equation (4.104)

Softmax transformation for multi-class classification

$$p(\mathcal{C}_k|\boldsymbol{\phi}) = y_k(\boldsymbol{\phi}) = \frac{\exp(a_k)}{\sum_j \exp(a_j)} \tag{4.104}$$

```
def softmax(w_trans, x):
    numClass = w_trans.shape[0]
    numTest = x.shape[0]

    classify_result = np.array([], dtype=np.int)
    # Evaluate every test data
    for itr_test in range(numTest):
        p_Ck_x = []

        aff_trans = affine_transform(w_trans, x[itr_test])
        # P(Ck|phi), phi: phi(x)
        denominator = 0
        for itr_aff in aff_trans:
            denominator += math.exp(itr_aff)
        for itr_aff in aff_trans:
            p_Ck_x.append(math.exp(itr_aff) / denominator)
        # Find the most possible class based on probability
        classify_result = np.append(classify_result, \
                                    np.array(p_Ck_x).argmax())

    return classify_result

```

Formula: [Reference] Textbook P.208 equation (4.99)

Newton-Raphson update formula for the logistic regression model

$$\begin{aligned}\mathbf{w}^{(\text{new})} &= \mathbf{w}^{(\text{old})} - (\mathbf{\Phi}^{\mathrm{T}}\mathbf{R}\mathbf{\Phi})^{-1}\mathbf{\Phi}^{\mathrm{T}}(\mathbf{y}-\mathbf{t}) \\ &= (\mathbf{\Phi}^{\mathrm{T}}\mathbf{R}\mathbf{\Phi})^{-1}\left\{\mathbf{\Phi}^{\mathrm{T}}\mathbf{R}\mathbf{\Phi}\mathbf{w}^{(\text{old})} - \mathbf{\Phi}^{\mathrm{T}}(\mathbf{y}-\mathbf{t})\right\} \\ &= (\mathbf{\Phi}^{\mathrm{T}}\mathbf{R}\mathbf{\Phi})^{-1}\mathbf{\Phi}^{\mathrm{T}}\mathbf{R}\mathbf{z} \end{aligned} \tag{4.99}$$

```
263
264 def newton_raphson(numClass, dsgMtx, training_class, w_init, converge):
265     # Using Newton-Raphson to find optimized w
266     # Textbook P.207 Eq 4.95
267     print "Optimized with Newton-Raphson's Method"
268     bNewtonRaphson = [True] * numClass
269     numTrain, dimension = dsgMtx.shape
270
271     dsgMtx_T = dsgMtx.transpose()
272     classify_oneOfk = expand_oneOfk(numClass, np.array(training_class))
273
274     # Initialize w
275     w_trans = [np.array([random.uniform(w_init, w_init)] * dimension) \
276                for itr in range(numClass)]
277
278     # Iterative found solution
279     for itr_class in range(numClass):
280         NR_itr = 0
281
282         # Target value
283         t = classify_oneOfk[:, itr_class]
284         t = t.reshape(t.shape[0], 1)
285         while bNewtonRaphson[itr_class]:
286             w = w_trans[itr_class].reshape(dimension, 1)
287             # Predict value
288             # y = p(Ck | phi(x)) = sigmoid(w.t * phi(x))
289             y = posterior_probability(w, dsgMtx)
290             # Weighing matrix
291             R = np.zeros((y.shape[0], y.shape[0]))
292             for itr in range(R.shape[0]):
293                 R[itr, itr] = y[itr] * (1 - y[itr])
294             #
295             w_curr = w - np.dot(np.dot(np.linalg.pinv( \
296                                        np.dot(dsgMtx_T.dot(R), dsgMtx)), \
297                                 dsgMtx_T), \
298                                 (y - t))
299
300             #print "Newton-Raphson Class {:1d}, itr: {:3d}:\nw = {:s}"\
301             #      .format(itr_class + 1, NR_itr + 1, str(w_curr.transpose()))
302
303             w_trans[itr_class] = w_curr
304
305             if NR_itr == converge[itr_class]:
306                 bNewtonRaphson[itr_class] = False
307             NR_itr += 1
308
309     return np.array(w_trans)
310
```

Formula: [Reference] Textbook P.209 equation (4.105)

Activations ($\mathbf{a}_k$) in Eq. 4.104

$$a_k = \mathbf{w}_k^{\mathrm{T}}\phi. \tag{4.105}$$

```
435 def affine_transform(w_trans, x):
436     # pass phi(x) as x
437     # a_k = w.T * x
438     numClass = w_trans.shape[0]
439
440     aff_trans = []
441     for itr_class in range(numClass):
442         w = w_trans[itr_class].reshape(w_trans[0].shape[0], 1)
443         x = x.reshape(x.shape[0], 1)
444
445         aff_trans.append(np.dot(w.transpose(), x))
446
447     return aff_trans
448
```

Formula: Basis function (phi function) 2-Dimension n^{th}-order polynomials

```
221
222  def polynomial(x, y, order, adj):
223      # Polynomial basis function
224      phi = []
225
226      for n in range(order + 1):
227          for x_n in range(n + 1):
228              phi.append([pow(x, x_n) * pow(y, n-x_n) / adj])
229
230      return np.hstack(phi)
231
```

```
231
232  def phi_transform(data, trainAmt, order, adj):
233      # Transform data to phi domain
234      numClass = len(trainAmt)
235      phi = []
236
237      offset = 0
238      for itr_class in range(numClass):
239          for itr_element in range(trainAmt[itr_class]):
240              phi.append(polynomial(data[offset + itr_element, 0], \
241                                    data[offset + itr_element, 1],
242                                    order, adj))
243          offset += trainAmt[itr_class]
244
245      return np.vstack(phi)
```

Performance:

```
$ python2 HW2_Face_Classification_Discriminative.py
Training info:
 - Class 1: 1000 files in total.
            30.0 percent of data are reserved for verifying.
 - Class 2: 1000 files in total.
            30.0 percent of data are reserved for verifying.
 - Class 3: 1000 files in total.
            30.0 percent of data are reserved for verifying.
 - Shuffle data    : False
 - Cross-Validation: False
 - Balanced training data.
 - Reducing dimension from 900 to 2.

 ===== Summary =====
Error rate:   10.89% (  98 of  900)

In-Class error rate:
 - Class 1:    5.33% (  16 of  300)
 - Class 2:    0.00% (   0 of  300)
 - Class 3:   27.33% (  82 of  300)
```

```
$ python2 HW2_Face_Classification_Generative.py
Training info:
 - Class 1: 1000 files in total.
            10.0 percent of data are reserved for verifying.
 - Class 2: 1000 files in total.
            10.0 percent of data are reserved for verifying.
 - Class 3: 1000 files in total.
            10.0 percent of data are reserved for verifying.
 - Shuffle data    : False
 - Cross-Validation: False
 - Balanced training data.
 - Reducing dimension from 900 to 2.

 ===== Summary =====
Error rate:   13.00% (  39 of  300)

In-Class error rate:
 - Class 1:    5.00% (   5 of  100)
 - Class 2:   21.00% (  21 of  100)
 - Class 3:   13.00% (  13 of  100)
```

2. Use Newton-Raphson iterative optimization in "Multi-class Probabilistic Discriminative Model".

To find the iteration that Newton-Raphson converges for each class, I print out those **w**' s and observe the values to obtain the number of iteration when the **w** stop changes.

```
$ python2 HW2_Face_Classification_Discriminative.py
Training info:
 - Class 1: 1000 files in total.
            10.0 % of data are reserved for verifying.
 - Class 2: 1000 files in total.
            10.0 % of data are reserved for verifying.
 - Class 3: 1000 files in total.
            20.0 % of data are reserved for verifying.
 - Shuffle data    : True
 - Cross-Validation: False
 - Unbalanced training data
 - Reducing dimension from 900 to 2.
Optimized with Newton-Raphson's Method
Newton-Raphson Class 1, total iteration:  10
Newton-Raphson Class 2, total iteration:  10
Newton-Raphson Class 3, total iteration:  10
```

3. You can separate some images for testing your models and note the error rate in the report. Choosing how many data for testing is up to you.

```
# Constants / Parameter
cv_part = 5
cv_enable = False
cv_iteration = cv_part if cv_enable else 1

random_suffle = False
pca_dimension = 2
reserved_part = [0.20, 0.10, 0.10]
converge = [10, 10, 10]
order = 3
adj = pow(10.0, 7)
w_init = pow(10.0, -10)

Train_Path = "Data_Train/"
Demo_Path  = "Demo/"
#
numClass = sum(os.path.isdir(os.path.join(Train_Path, itr_dir)) \
                        for itr_dir in os.listdir(Train_Path))

numDemo = sum(os.path.isfile(os.path.join(Demo_Path, itr_file)) \
                        for itr_file in os.listdir(Demo_Path))

```

To parametrize and to make it easier for adjusting the portion used for training, testing, or even cross-validation in the future, I first define some parameters to improve the maintainability and generalizability of the code.

Name	Function
cv_part	Determine the amount of n in n-fold cross validation.
cv_enable	Enable or disable the cross-validation functioning in the code.
cv_iteration	Determine the times of iteration based on previous two values.
random_suffle	Determine whether to shuffle the input data. Since some features might hide in some certain data, randomly choosing the training data and iterate several times to attenuate the possible bias when we simply choose the training data by their sequence, the average performance of the model can be a better indication for evaluating the model.
pca_dimension	Determine the dimension of PCA reduces to.
reserved_part	Define the portion of the data for each class that saved for testing (the number * 100 equal percentage of given data).
converge	A list of iterations that a **w** of each class required to converge.
order	The order of polynomial basis function.
adj	Scaling parameter to avoid overflow.
w_init	The seed to initialize the **w**.
numClass	Scan the number of given classes by number of folders in the corresponding working directory.
numDemo	Scan the number of demo data in the corresponding folder.

```
94      training_list = []
95      testing_list = []
96      for itr_class in range(numClass):
97          # Generate training data path name
98          file_dir = "Data_Train/Class{:d}/".format(itr_class + 1)
99          tmp_rand = []
100
101         for itr_file in range(trainAmt[itr_class] + verifAmt[itr_class]):
102             # Generate training data name
103             file_name = "faceTrain{:d}_{:d}.bmp".format(itr_class + 1, itr_file + 1)
104
105             # Convert the image to np.array for dimension reduction
106             tmp_img = np.array(Image.open(file_dir + file_name))
107             # reshape the image to column vectors
108             tmp_img = tmp_img.reshape(1, tmp_img.shape[0] * tmp_img.shape[1])
109
110             # Without Suffle - Divide data to training and testing data
111             # Append to the data matrix
112             if (not random_suffle) and (itr_file < trainAmt[itr_class]):
113                 training_list.append(tmp_img)
114             elif (not random_suffle) and (itr_file >= trainAmt[itr_class]):
115                 testing_list.append(tmp_img)
116             # Random Suffle - Divide data to training and testing data
117             elif random_suffle:
118                 tmp_rand.append(tmp_img)
119
120         # Random Suffle - Divide data to training and testing data
121         if random_suffle:
122             # Shuffle the data
123             tmp_data = np.random.permutation(np.vstack(tmp_rand))
124             training_list.append(tmp_data[:trainAmt[itr_class]])
125             testing_list.append(tmp_data[-1 * verifAmt[itr_class]:])
126
127     return np.vstack(training_list), np.vstack(testing_list)
128
```

Within each iteration (Line 96: one iteration per class,) if the random_suffle is asserted, the pre-loaded data will first be shuffled using numpy.random.permutation() to randomly permute the rows (namely, shuffle the order of images) in the preloaded data, and then stack it to the returning variables (training_data and testing_data.)

```
86      # Check if the data in each class are balanced.
87      b_balanced = True
88      for itr in range(numClass):
89          b_balanced = b_balanced and (trainAmt[itr] == np.array(trainAmt).mean())
90
91      print " - {:s} training data".format("Balanced" if b_balanced else "Unbalanced")
92
```

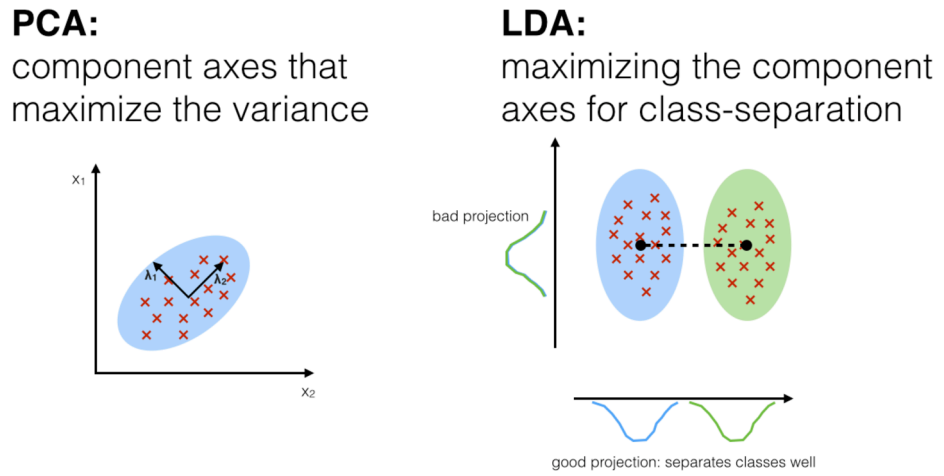
This block is to check whether the current setting for training and testing data is between classes is balanced or unbalanced.

Cross-Validation Error Rate	
Generative = 14.78%	Discriminative: 10.67%

```
$ python2 HW2_Face_Classification_Generative.py
Training info:
 - Class 1: 1000 files in total.
            20.0 percent of data are reserved for verifying.
 - Class 2: 1000 files in total.
            20.0 percent of data are reserved for verifying.
 - Class 3: 1000 files in total.
            20.0 percent of data are reserved for verifying.
 - Shuffle data    : True
 - Cross-Validation: True
 - Balanced training data.
 - Reducing dimension from 900 to 2.
 Iteration: #1
Error rate:   15.33% (  92 of  600)
 Iteration: #2
Error rate:   15.33% (  92 of  600)
 Iteration: #3
Error rate:   13.67% (  82 of  600)

*** Average error rate:   14.78% ***
```

```
$ python2 HW2_Face_Classification_Discriminative.py
Training info:
 - Class 1: 1000 files in total.
            10.0 % of data are reserved for verifying.
 - Class 2: 1000 files in total.
            10.0 % of data are reserved for verifying.
 - Class 3: 1000 files in total.
            10.0 % of data are reserved for verifying.
 - Shuffle data    : True
 - Cross-Validation: True
 - Balanced training data
 - Reducing dimension from 900 to 2.
Optimized with Newton-Raphson's Method
 Iteration: #1
Error rate:   11.33% (  34 of  300)
Optimized with Newton-Raphson's Method
 Iteration: #2
Error rate:   10.00% (  30 of  300)
Optimized with Newton-Raphson's Method
 Iteration: #3
Error rate:   10.67% (  32 of  300)

*** Average error rate:   10.67% ***
```

4. Please use Principal component analysis (PCA) to map data down to 2 dimensions. You can use other dimension reduction methods to make your report be better.



The goal of PCA (or all kinds of dimension reduction) is to project a dataset onto a lower-dimensional space with good class-separability in order avoid overfitting (curse of dimensionality) and also reduce computational costs.

PCA (Principal Component Analysis) versus LDA (Linear Discriminant Analysis)

Both LDA and PCA are linear transformation techniques that are commonly used for dimension reduction. Since PCA ignores the class labels of the data, in other words, class labels are not involved when computing the PCA projection matrix, it is an "unsupervised" algorithm. The goal of PCA is to find the directions (the so-called principal components) that maximize the variance within the entire dataset (inter-data separation). Contrary to PCA, LDA is a "supervised" algorithm and it computes the directions that maximize the separation between multiple classes (inter-class separation).

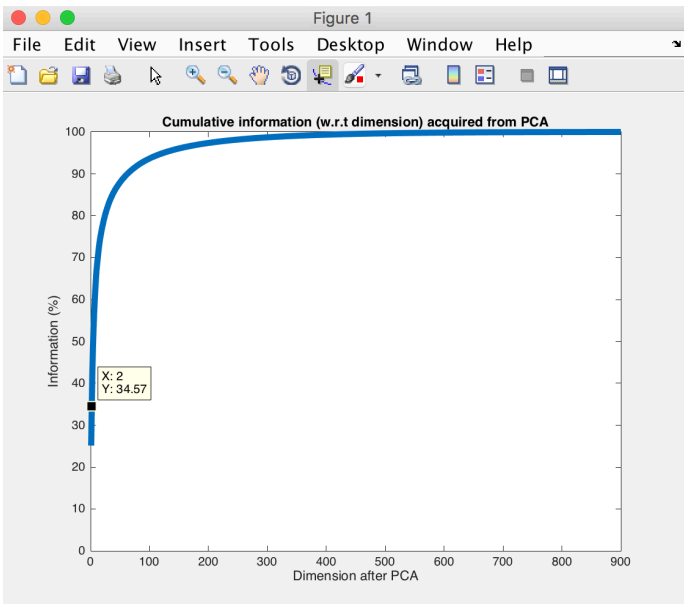
Choosing dimension that PCA reduces to

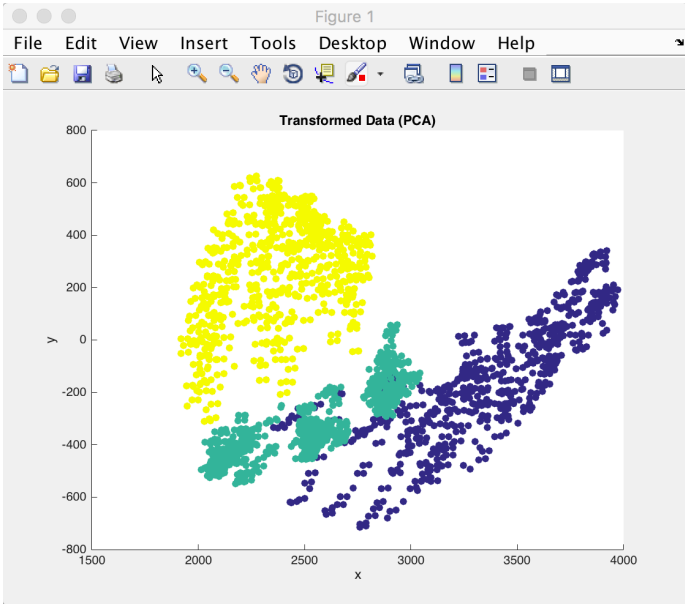
Picking the dimension that PCA reduces to is also of great importance, discarding too much information (choosing very less dimension) may make the model acting poorly, choosing higher dimension also means we are adding more noise, unwanted, or unimportant information to our feature space.

We will compute eigenvectors (the "proper" components) from our dataset and collect them in a so-called scatter-matrices (i.e., the in-between-class scatter matrix and within-class scatter matrix). If we would observe that all eigenvalues have a similar magnitude, then this may be a good indicator that our data is already projected on a "good" feature space. On

the other hand, if eigenvalues have distinguishing magnitudes, we might be interested in keeping only eigenvectors whose eigenvalues are much larger than others, since they contain more information about our data distribution. Vice versa, small eigenvalues are less informative and we might consider dropping them from our feature subspace.

Analysis on feature subspace





The figure on the left shows the portion of information compared to the original dataset when the feature subspace is composed of two highest eigenvectors in the scatter matrix is around 34.57%. I am not sure if this is enough for building a good model, but the result I obtain seems to indicate that it is close enough (error rate 10~15%), for further improvement, I think the modification of the probability model and some trick on the PCA or in combination with LDA might be useful.

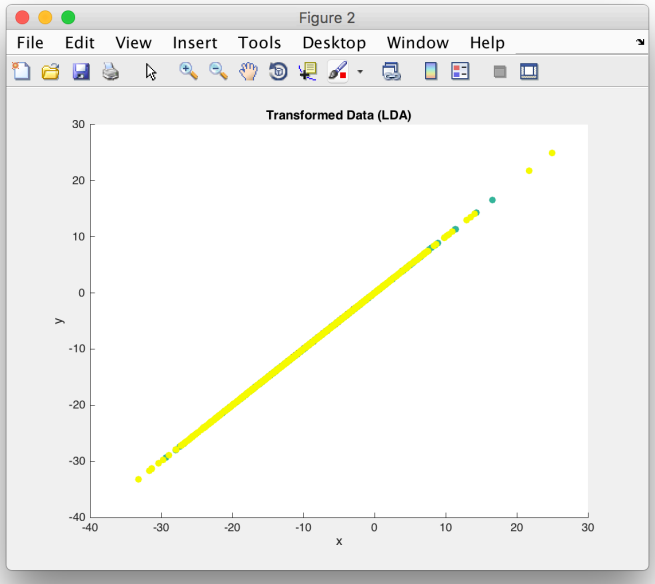
The figure on the right shows that the separation between classes is not enough, some portion of the class overlapped, without further processing the data, it is foreseeable that this feature subspace might not give a high accuracy result (error rate <5% or even <1%)

Reference:

Implementing a Principal Component Analysis (PCA) – in Python, step by step
http://sebastianraschka.com/Articles/2014_pca_step_by_step.html

```python
def lda(data, n, trainAmt, training_class):
    dimension = data.shape[1]
    numClass = len(trainAmt)

    # 1. Computing the d-dimensional mean vectors
    # mean_vectors = [mx_k, my_k] for every class
    offset = 0
    mean_vectors = []
    for itr_class in range(numClass):
        tmp_mean = []
        for itr_dim in range(dimension):
            mean = data[offset:offset + trainAmt[itr_class], itr_dim].mean()
            tmp_mean.append(mean)
        mean_vectors.append(np.array(tmp_mean))
        offset += trainAmt[itr_class]

    # 2. Computing the Scatter Matrices
    #   2.1 - Within-class scatter matrix SW
    offset = 0
    S_W = np.zeros((dimension, dimension))
    for itr_class in range(numClass):...

    #   2.2 - Between-class scatter matrix SB
    overall_mean = np.mean(data, axis=0)

    S_B = np.zeros((dimension, dimension))
    for itr_class, mean_vec in enumerate(mean_vectors):...

    # 3. Solving the generalized eigenvalue problem by SW_inv * SB
    eig_vals, eig_vecs = np.linalg.eig(np.linalg.inv(S_W).dot(S_B))

    # 4. Selecting linear discriminants for the new feature subspace
    # 4.1. Sorting the eigenvectors by decreasing eigenvalues
    # Make a list of (eigenvalue, eigenvector) tuples
    eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:,i]) for i in range(len(eig_vals))]

    # Sort the (eigenvalue, eigenvector) tuples from high to low
    eig_pairs = sorted(eig_pairs, key=lambda k: k[0], reverse=True)

    # 4.2. Choosing k eigenvectors with the largest eigenvalues
    proj_mtx = np.hstack((eig_pairs[itr][1].reshape(dimension, 1) \
                          for itr in range(n)))
    eig_vals = np.hstack((eig_pairs[itr][0] for itr in range(dimension)))

    return proj_mtx, eig_vals
```

The above figure shows the code which implements Linear Discriminant Analysis with python, unfortunately, I have no idea dealing with the complex value in eigenvalues and eigenvectors, the figure on the right show the feature subspace with real part only, which indicates that the LDA subspace should be a plane orthogonal to the real plane.



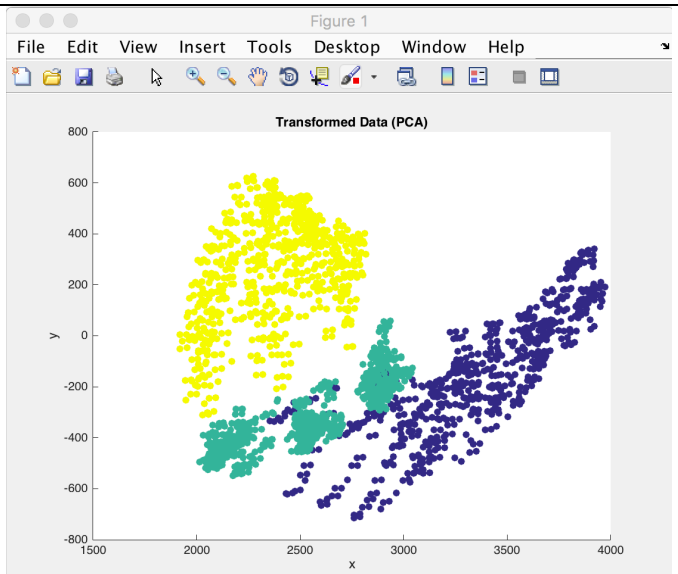
Reference:

Linear Discriminant Analysis – Bit by Bit

http://sebastianraschka.com/Articles/2014_python_lda.html

5. Draw the pictures of your decision region. (Scatter plot)

Dataset in feature subspace



6. Test unbalanced data how to affect your models.

(1) Generative

```
$ python2 HW2_Face_Classification_Generative.py
Training info:
 - Class 1: 1000 files in total.
            10.0 percent of data are reserved for verifying.
 - Class 2: 1000 files in total.
            10.0 percent of data are reserved for verifying.
 - Class 3: 1000 files in total.
            10.0 percent of data are reserved for verifying.
 - Shuffle data    : True
 - Cross-Validation: False
 - Balanced training data.
 - Reducing dimension from 900 to 2.

 ===== Summary =====
Error rate:   18.33% (  55 of  300)

In-Class error rate:
 - Class 1:   15.00% (  15 of  100)
 - Class 2:   24.00% (  24 of  100)
 - Class 3:   16.00% (  16 of  100)
```

Error rate = 18.33%

```
$ python2 HW2_Face_Classification_Generative.py
Training info:
 - Class 1: 1000 files in total.
            20.0 percent of data are reserved for verifying.
 - Class 2: 1000 files in total.
            10.0 percent of data are reserved for verifying.
 - Class 3: 1000 files in total.
            10.0 percent of data are reserved for verifying.
 - Shuffle data    : False
 - Cross-Validation: False
 - Unbalanced training data.
 - Reducing dimension from 900 to 2.

 ===== Summary =====
Error rate:    7.00% (  28 of  400)

In-Class error rate:
 - Class 1:    0.00% (   0 of  200)
 - Class 2:   28.00% (  28 of  100)
 - Class 3:    0.00% (   0 of  100)
```

Error rate = 7.00%

```
$ python2 HW2_Face_Classification_Generative.py
Training info:
 - Class 1: 1000 files in total.
            10.0 percent of data are reserved for verifying.
 - Class 2: 1000 files in total.
            20.0 percent of data are reserved for verifying.
 - Class 3: 1000 files in total.
            10.0 percent of data are reserved for verifying.
 - Shuffle data    : False
 - Cross-Validation: False
 - Unbalanced training data.
 - Reducing dimension from 900 to 2.

 ===== Summary =====
Error rate:   29.50% ( 118 of  400)

In-Class error rate:
 - Class 1:    0.00% (   0 of  100)
 - Class 2:   40.00% (  80 of  200)
 - Class 3:   38.00% (  38 of  100)
```

Error rate = 29.50%

```
$ python2 HW2_Face_Classification_Generative.py
Training info:
 - Class 1: 1000 files in total.
            10.0 percent of data are reserved for verifying.
 - Class 2: 1000 files in total.
            10.0 percent of data are reserved for verifying.
 - Class 3: 1000 files in total.
            20.0 percent of data are reserved for verifying.
 - Shuffle data    : False
 - Cross-Validation: False
 - Unbalanced training data.
 - Reducing dimension from 900 to 2.

 ===== Summary =====
Error rate:   27.25% ( 109 of  400)

In-Class error rate:
 - Class 1:    0.00% (   0 of  100)
 - Class 2:   33.00% (  33 of  100)
 - Class 3:   38.00% (  76 of  200)
```

Error rate = 27.25%

(2) Discriminative:

```
$ python2 HW2_Face_Classification_Discriminative.py
Training info:
 - Class 1: 1000 files in total.
            10.0 % of data are reserved for verifying.
 - Class 2: 1000 files in total.
            10.0 % of data are reserved for verifying.
 - Class 3: 1000 files in total.
            10.0 % of data are reserved for verifying.
 - Shuffle data    : True
 - Cross-Validation: False
 - Balanced training data
 - Reducing dimension from 900 to 2.
Optimized with Newton-Raphson's Method

 ===== Summary =====
Error rate:   11.00% (  33 of  300)

In-Class error rate:
 - Class 1:    6.00% (   6 of  100)
 - Class 2:    6.00% (   6 of  100)
 - Class 3:   21.00% (  21 of  100)
```

Error rate = 11.00%

```
$ python2 HW2_Face_Classification_Discriminative.py
Training info:
 - Class 1: 1000 files in total.
            20.0 % of data are reserved for verifying.
 - Class 2: 1000 files in total.
            10.0 % of data are reserved for verifying.
 - Class 3: 1000 files in total.
            10.0 % of data are reserved for verifying.
 - Shuffle data    : True
 - Cross-Validation: False
 - Unbalanced training data
 - Reducing dimension from 900 to 2.
Optimized with Newton-Raphson's Method

 ===== Summary =====
Error rate:   11.25% (  45 of  400)

In-Class error rate:
 - Class 1:    7.00% (  14 of  200)
 - Class 2:    6.00% (   6 of  100)
 - Class 3:   25.00% (  25 of  100)
```

Error rate = 11.25%

```
$ python2 HW2_Face_Classification_Discriminative.py
Training info:
 - Class 1: 1000 files in total.
            10.0 % of data are reserved for verifying.
 - Class 2: 1000 files in total.
            20.0 % of data are reserved for verifying.
 - Class 3: 1000 files in total.
            10.0 % of data are reserved for verifying.
 - Shuffle data    : True
 - Cross-Validation: False
 - Unbalanced training data
 - Reducing dimension from 900 to 2.
Optimized with Newton-Raphson's Method

 ===== Summary =====
Error rate:   12.25% (  49 of  400)

In-Class error rate:
 - Class 1:    3.00% (   3 of  100)
 - Class 2:    9.00% (  18 of  200)
 - Class 3:   28.00% (  28 of  100)
```

Error rate = 12.25%

```
$ python2 HW2_Face_Classification_Discriminative.py
Training info:
 - Class 1: 1000 files in total.
            10.0 % of data are reserved for verifying.
 - Class 2: 1000 files in total.
            10.0 % of data are reserved for verifying.
 - Class 3: 1000 files in total.
            20.0 % of data are reserved for verifying.
 - Shuffle data    : True
 - Cross-Validation: False
 - Unbalanced training data
 - Reducing dimension from 900 to 2.
Optimized with Newton-Raphson's Method

 ===== Summary =====
Error rate:   16.25% (  65 of  400)

In-Class error rate:
 - Class 1:    4.00% (   4 of  100)
 - Class 2:    4.00% (   4 of  100)
 - Class 3:   28.50% (  57 of  200)
```

Error rate = 16.25%