Discriminant Analysis and Naive Bayes

Michael

School of Mathematics and Statistics, UCD

School of Economics, University of Nottingham

1 Discriminant Analysis

Suppose $f_k(x)$ is the class-conditional density of X in class $Y = k$, and let π_k be the prior probability of class k, with $\sum_{k=1}^{K} \pi_k = 1$. A simple application of Bayes theorem gives us

$$\mathbb{P}(Y = k|X = x) = \frac{f_k(x)\pi_k}{\sum_{l=1}^{K} f_l(x)\pi_l} \tag{1.1}$$

We see that in terms of ability to classify, having the $f_k(x)$ is almost equivalent to having the quantity $\mathbb{P}(Y = k|X = x)$. Many techniques are based on models for the class densities:

- linear and quadratic discriminant analysis use Gaussian density;
- more flexible mixtures of Gaussians allow for nonlinear decision boundaries;
- general nonparametric density estimates for each class density allow the most flexibility
- *Naive Bayes* models are a variant of the previous case, they assume that the inputs are conditionally independent in each class.

Suppose we we model each class density as multivariate Gaussian

$$f_k(x) = \frac{1}{(2\pi)^{p/2}|\Sigma_k|^{1/2}} e^{-\frac{1}{2}(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k)} \tag{1.2}$$

Linear discriminant analysis (LDA) arises in the special case when we assume that the classes have a common covariance matrix $\Sigma_k = \Sigma \ \forall k$. The denominator in equation (1.1) does not depend on k, therefore we can write it as

$$\mathbb{P}(Y = k|X = x) = \frac{f_k(x)\pi_k}{\mathbb{P}(X = x)} = C \times f_k(x)\pi_k, \quad C = 1/\mathbb{P}(X = x)$$

Substitute the density function in (1.2) and take the log, we can get

$$\log[\mathbb{P}(Y = k|X = x)] = \log C + \log \pi_k - \frac{1}{2}(x - \mu_k)^T\Sigma^{-1}(x - \mu_k)$$

This is the same for every category, k. So we want to find the maximum of this over k, in which it is equivalent to

$$\delta_k(x) = \log \pi_k - \frac{1}{2}\mu_k^T \Sigma^{-1} \mu_k + x^T \Sigma^{-1} \mu_k$$

At an input x, we predict the response with the highest $\delta_k(x)$. In practice we do not know the parameters of the Gaussian distributions, and will need to estimate them using our training data:

- $\hat{\pi}_k = N_k/N$
- $\hat{\mu}_k = \sum_{g_i=k} x_i/N_k$;
- $\hat{\Sigma} = \Sigma_{k=1}^K \Sigma_{g_i=k}(x_i - \hat{\mu}_k)(x_i - \hat{\mu}_k)^T/(N-K)$

If the Σ_k are not assumed to be equal, then we can get *quadratic discriminant function* (QDA),

$$\delta_k(x) = -\frac{1}{2}\log|\Sigma_k| - \frac{1}{2}(x-\mu_k)^T \Sigma_k^{-1}(x-\mu_k) + \log \pi_k$$

Here is the code.

```
# Function to fit LDA
def fitLDA(X, Y):
    '''
    X and Y are dataframe
    need to be transfomed into:
    Input: X - n by m matrix, containing all features
           Y - n by 1 matrix, contanning the information of class
    Causion: the orders of rows of X and Y are assumed to be corresponding
                                                     to
             each other, which means inforamtion of any row (say row 10) of
                                                                X
             and information of same row (row 10) of Y are from same
             observation
    Output: prior_probability: estimated probability: pi_k (dataframe)
            classmean: estiamted mean of different classes: mu_k (dataframe
                                                            )
            sigma: estimated covariance matrix (matrix)
    '''
    n = X.shape[0]   # get the number of total observation
    columnsIndex = X.columns
    m = X.shape[1]   # get the number of covariate variables (features)
    X = np.asmatrix(X).reshape(n, m)
    Y = np.asarray(Y).reshape(-1, 1)   # Y has to be an array, not matrix
    yunique = np.unique(Y)
    prior_probability = {}  # initialize a distionary for prior probability
    mu_k = {}
    sigma = {}

    for i in yunique:
        rowindex = np.where(Y == i)[0]
        classlabel = 'class' + str(i)
        prior_probability[classlabel] = len(rowindex)/n
```

```
        filteredx = X[rowindex, :]
        mean_vector = np.mean(filteredx, axis=0)
        mu_k[classlabel] = mean_vector
        mean_maxtrix = np.repeat(mean_vector, filteredx.shape[0], axis=0)
        diff_matrix = filteredx - mean_maxtrix
        sigma_k = diff_matrix.T @ diff_matrix
        # calcluate within-class covariance
        sigma[classlabel] = sigma_k

    # tidy the output
    sigma_sum = np.zeros([m, m])
    for i in sigma:
        sigma_sum += sigma[i]
    sigma = sigma_sum/(n - len(yunique))  # estimate final sigma
    prior_probability = pd.DataFrame(list(prior_probability.values()),
                                     index=prior_probability.keys(),
                                     columns=['Prior Probability'])
    mean_dataframe = []
    for v in mu_k:
        mean_dataframe.extend(np.array(mu_k[v]))
    mu_k = pd.DataFrame(mean_dataframe, index=mu_k.keys(),
                        columns=columnsIndex)

    return(prior_probability, mu_k, sigma)


# Function to classify LDA
def classifyLDA(featureX, priorpro, mu, sigma, critical=False):
    '''
    This is the classification for multi-categories case
    and it only takes the binary case as a special one

    Input: 1)featureX: n by m dataframe, where n is sample size, m is
                                              number
           of covariatle variables

           2)mu: k by m dataframe, where k is the number of classes or
           categories m is the number of covariatle variables. mu is
           taken from fitLDA() function.

           3) priorpro: k by 1 dataframe, prior probabilty, it is taken
                                                     from
           fitLDA() function.

           4) sigma: k by k covariance matrix, it is also taken from fitLDA
                                                     ()

           5) critical=Faulse, if it is true, then it should be the case
                                                     that
              number of calsses = number of features. Otherwise, there is
              no solution for critical values.
              WARNING: in this function, the critical value calculation
                                                          only
                       applies for the binar case with one feature
    Output:
           Classification results: n by 1 vector and newdataframe with
                                  extra column called 'LDAClassification'
           and k by 1 vector of critical values of X
```

```
    '''
    newX = pd.DataFrame.copy(featureX)
    classLabels = priorpro.index  # get class labes from dataframe
    featureLabels = featureX.columns
    meanLabels = mu.columns
    X = np.asmatrix(featureX)
    priorpro = np.asmatrix(priorpro)
    if all(featureLabels == meanLabels):
        delta = np.zeros([featureX.shape[0], 1])
        for v in range(len(classLabels)):
            Probabilty = np.array(priorpro[v, :]).reshape(-1, 1)
            # get prior probabilty for class k
            mean_vector = np.array(mu.iloc[v, :]).reshape(-1, 1)
            # get mean vector for class k
            deltaX = (X @ np.linalg.inv(sigma) @ mean_vector
                      - 1/2 * mean_vector.T @
                      np.linalg.inv(sigma) @ mean_vector
                      + math.log(Probabilty))
            delta = np.hstack([delta, np.asmatrix(deltaX).reshape(-1, 1)])

        delta = delta[:, 1:]
        classificationResults = np.argmax(delta, axis=1)
        # maximize the delta over k
        newX['LDAClassification'] = classificationResults.reshape(-1, 1)
    else:
        print('Pleasre make sure that featured X and mean vector\
              have the same covariate variables')
        sys.exist(0)

    if critical is True:
        if len(classLabels) < len(featureLabels):
            print('There is no solutions for critical values\
                  as dimension of classes is less than dimension\
                  of covariate variables')
            sys.exist(0)
        else:
            # calculate the critical values
            mean_i = np.array(mu.iloc[0, :]).reshape(-1, 1)
            mean_j = np.array(mu.iloc[1, :]).reshape(-1, 1)
            prob_i = np.array(priorpro[0, :]).reshape(-1, 1)
            prob_j = np.array(priorpro[1, :]).reshape(-1, 1)
            xcritical = sigma/(mean_j - mean_i)*(
                math.log(prob_i/prob_j) + (mean_j**2
                                           - mean_i**2)/(2*sigma))
            return(classificationResults, newX, xcritical)
    else:
        return(classificationResults, newX)
```

2 Naive Bayes

The Naive Bayes[1] algorithm is a classification algorithm based on Bayes rule and a set of conditional independence assumptions. Given the goal of learning $\mathbb{P}(Y|X)$ where $X =< X_1, \cdots, X_n >$, the Naive Bayes algorithm makes the assumption that each X_i

[1]Reference: https://www.cs.cmu.edu/~tom/mlbook/NBayesLogReg.pdf

is conditionally independent of each of the other X_ks given Y, also independent of each subset of the other X_k's given Y.

The expression for the probability that Y will take on its kth possible value, according to Bayes rule, is

$$\mathbb{P}(Y = y_k|X_1, \cdots, X_n) = \frac{\mathbb{P}(Y = y_k)\mathbb{P}(X_1, \cdots, X_n|Y = y_k)}{\sum_j \mathbb{P}(Y = y_j)\mathbb{P}(X_1, \cdots, X_n|Y = y_j)}$$

Assuming the X_i are conditionally independent given Y, we can write the above equation as

$$\mathbb{P}(Y = y_k|X_1, \cdots, X_n) = \frac{\mathbb{P}(Y = y_k)\prod_i \mathbb{P}(X_i|Y = y_k)}{\sum_j \mathbb{P}(Y = y_j)\prod_i \mathbb{P}(X_i|Y = y_i)}$$

If we are interested only in the most probable value of Y, then we have the Naive Bayes classification rule:

$$Y \leftarrow \arg\max_{y_k} \frac{\mathbb{P}(Y = y_k)\prod_i \mathbb{P}(X_i|Y = y_k)}{\sum_j \mathbb{P}(Y = y_j)\prod_i \mathbb{P}(X_i|Y = y_i)}$$

which simplifies to the following (because the denominator does not depend on y_k)

$$Y \leftarrow \arg\max_{y_k} \mathbb{P}(Y = y_k)\prod_i \mathbb{P}(X_i|Y = y_k) \tag{2.1}$$

To solve the model, we need estimate $\mathbb{P}(Y = y_k)$ and $\mathbb{P}(x_i|Y = y_k)$.

There are several naive Bayes models, one could read the detailed explanations from this website.

Here is the code.

```
def classifyNBG(featureX, priorpro, mu, sigma):
    '''
    Same Input, same out
    But algorithm is different, we need employ the pdf of Gaussian Normal
    '''
    # calcluate probability from Gaussian Nomral
    newX = pd.DataFrame.copy(featureX)
    classLabels = priorpro.index  # get class labes from dataframe
    featureLabels = featureX.columns
    meanLabels = mu.columns
    X = np.asmatrix(featureX)
    m = featureX.shape[1]
    delta = np.zeros([featureX.shape[0], 1])
    deltaX = np.zeros([featureX.shape[0], 1])
    if all(featureLabels == meanLabels):
        for v in classLabels:
            probabilty = np.array(priorpro.loc[v, :])
            mean_vector = np.array(mu.loc[v, :]).reshape(1, -1)
            sigma_k = sigma[v]
            for i in range(featureX.shape[0]):
                x_rowvector = X[i, :]
                x_diff = (x_rowvector - mean_vector).reshape(1, -1)
                zmod = np.sqrt(np.power((2*math.pi),m)
                               *np.linalg.det(sigma_k))
```

```
                post_prob = 1/zmod*np.exp(-0.5*x_diff@np.linalg.inv(sigma_k
                                                       )@x_diff.T)
                delta[i, :] = post_prob * probabilty

            deltaX = np.hstack([deltaX, delta])

        deltaX = deltaX[:, 1:]
        deltaX = np.divide(deltaX, np.sum(deltaX, axis=1).reshape(-1, 1))
        classificationResults = np.argmax(deltaX, axis=1)
        # maximize the delta over k
        newX['LDAClassification'] = classificationResults.reshape(-1, 1)

    else:
        print('Pleasre make sure that featured X and mean vector\
              have the same covariate variables')
        sys.exist(0)

    return(classificationResults, newX)
```