Lecture 11 - Feb 14

Linear Discriminant Analysis
Quadratic Discriminant Analysis

Reading

Week 7 Notes in GitHub

Elements of Statistical Learning
4.3 Discriminant Analysis

O. Ledoit and M. Wolf. Improved estimation of the covariance matrix of stock returns with an application to portfolio selection. *Journal of Empirical Finance*, 10(5), 603-621. (2003)

O. Ledoit and M. Wolf. Honey, I shrunk the sample covariance matrix. *Journal of Portfolio Management*, 30 (4), pp. 110-119. (2004)

Upcoming Deadlines

Homework 2 (Feb 18)
Project 1 Proposal (coming soon)

Let $f_k(x)$ be the density of X conditioned on $Y=k$ and π_k prior probability of class k

By Bayes' Theorem + Law of Total Probability,

$$P(Y=j \mid X=x) = \frac{f_j(x)\pi_j}{\sum_{l=1}^{k} f_l(x)\pi_l}$$

- LDA/QDA use Gaussian densities
- Gaussian mixtures are more flexible
- Bayes classifier may assume multivariate Gaussian densities or other estimates
- Naive Bayes assumes densities are products of marginal densities.

Suppose the densities are multivariate Gaussian

$$f_k(x) = \frac{1}{(2\pi)^{d/2}|\Sigma_k|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_k)^T \Sigma_k^{-1}(x-\mu_k)\right)$$

Linear discriminant analysis (LDA) is the special case where all classes have the same covariance matrix $\Sigma_k = \Sigma$.

Log-odds

$$\ln\left(\frac{\mathbb{P}(Y=i\,|\,X=x)}{\mathbb{P}(Y=j\,|\,X=x)}\right) = \ln\left(\frac{f_i(x)\pi_i}{\sum_{k=1}^{K} f_k(x)\pi_k} \cdot \frac{\sum_{k=1}^{K} f_k(x)\pi_k}{f_j(x)\pi_j}\right)$$

$$= \ln\left(\frac{f_i(x)}{f_j(x)}\right) + \ln\left(\frac{\pi_i}{\pi_j}\right)$$

$$= \ln\left(\frac{\frac{1}{(2\pi)^{p/2}\sqrt{|\Sigma|}}\exp\left(\frac{-1}{2}(x-\mu_i)^T\Sigma^{-1}(x-\mu_i)\right)}{\frac{1}{(2\pi)^{p/2}\sqrt{|\Sigma|}}\exp\left(\frac{-1}{2}(x-\mu_j)^T\Sigma^{-1}(x-\mu_j)\right)}\right) + \ln\left(\frac{\pi_i}{\pi_j}\right)$$

$$= \frac{-1}{2}(x-\mu_i)^T\Sigma^{-1}(x-\mu_i) + \frac{1}{2}(x-\mu_j)^T\Sigma^{-1}(x-\mu_j) + \ln\left(\frac{\pi_i}{\pi_j}\right)$$

$$= \frac{-1}{2}x^T\Sigma^{-1}x + \frac{1}{2}x^T\Sigma^{-1}\mu_i + \frac{1}{2}\mu_i^T\Sigma^{-1}x - \frac{1}{2}\mu_i^T\Sigma^{-1}\mu_i$$

$$+ \frac{1}{2}x^T\Sigma^{-1}x - \frac{1}{2}x^T\Sigma^{-1}\mu_j - \frac{1}{2}\mu_j^T\Sigma^{-1}x + \frac{1}{2}\mu_j^T\Sigma^{-1}\mu_j$$

$= \delta_j(x)$ discriminant for class j

$= \delta_i(x)$ discriminant

0 diff. from

δ_i - discriminant function for class i

$+\ln(\pi_i) + \ln(\pi_j)$

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

By assuming $\Sigma_k = \Sigma$, the log-odds are linear in x. Otherwise, it would be quadratic in x, and quadratic discriminant analysis (QDA) is used.

$$\delta_i(x) = \frac{-1}{2} \ln(|\Sigma_i|) - \frac{1}{2}(x-\mu_i)^T \Sigma_i^{-1} (x-\mu_i) + \ln(\pi_i)$$

QDF

The predicted class for x is the k such that the discriminant function $\delta_k(x)$ is maximized

LDA/QDA Implementation

```python
class LDA:
    def fit(self, X, Y):
        # find the unique labels
        uniqueY = np.unique(Y)

        # find the dimensions
        n = X.shape[0]
        self.d = X.shape[1]
        self.k = uniqueY.shape[0]

        # initialize the variables
        self.prior = np.zeros([self.k, 1])
        self.mu = np.zeros([self.k, self.d])

        # compute the covariance matrix
        mu = np.mean(X, axis = 0)
        Xbar = X - mu
        self.Sig = (1/n) * Xbar.T @ Xbar
        self.invCov = np.linalg.inv(self.Sig)

        # compute class means and prior probabilities
        for i, y in enumerate(uniqueY):
            # extract a class of datapoints from X
            Xi = X[Y == y]

            # compute the size of each class
            ni = Xi.shape[0]

            # compute the priors
            self.prior[i] = ni / n

            # compute the sample mean
            self.mu[i] = np.mean(Xi, axis = 0)
```

```python
class QDA:
    def fit(self, X, Y):
        # find the unique labels
        uniqueY = np.unique(Y)

        # find the dimensions
        n = X.shape[0]
        self.d = X.shape[1]
        self.k = uniqueY.shape[0]

        # initialize the variables
        self.prior = np.zeros([self.k, 1])
        self.mu = np.zeros([self.k, self.d])
        self.Sigma = np.zeros([self.k, self.d, self.d])

        # compute class means and prior probabilities
        for i, y in enumerate(uniqueY):
            # extract a class of datapoints from X
            Xi = X[Y == y]

            # compute the size of each class
            ni = Xi.shape[0]

            # compute the priors
            self.prior[i] = ni / n

            # compute the sample mean
            self.mu[i] = np.mean(Xi, axis = 0)

            # compute the centered data
            XiBar = Xi - self.mu[i]

            # compute the sample covariance
            self.Sigma[i] = (1/ni) * XiBar.T @ XiBar
```

```python
def predict(self, X):
    n = X.shape[0]

    discriminants = np.zeros([n, self.k])

    for i, x in enumerate(X):
        x = np.atleast_2d(x).T

        for j in range(self.k):
            discriminants[i][j] = x.T @ self.invCov @ self.mu[j].T - (1/2) * self.mu[j] @ self.invCov @ self.mu[j].T + np.log(self.prior[j])


    predictions = np.argmax(discriminants, axis = 1)

    return predictions
```

```python
def predict(self, X):
    n = X.shape[0]

    discriminants = np.zeros([n, self.k])

    for i, x in enumerate(X):
        x = np.atleast_2d(x).T

        for j in range(self.k):
            invCov = np.linalg.inv(self.Sigma[j])
            discriminants[i][j] = x.T @ invCov @ self.mu[j].T - (1/2) * self.mu[j] @ invCov @ self.mu[j].T + np.log(self.prior[j])

    predictions = np.argmax(discriminants, axis = 1)

    return predictions
```

QDA vs. Bayes

Identical!

```python
class QDA:
    def fit(self, X, Y):
        # find the unique labels
        uniqueY = np.unique(Y)

        # find the dimensions
        n = X.shape[0]
        self.d = X.shape[1]
        self.k = uniqueY.shape[0]

        # initialize the variables
        self.prior = np.zeros([self.k, 1])
        self.mu = np.zeros([self.k, self.d])
        self.Sigma = np.zeros([self.k, self.d, self.d])

        # compute class means and prior probabilities
        for i, y in enumerate(uniqueY):
            # extract a class of datapoints from X
            Xi = X[Y == y]

            # compute the size of each class
            ni = Xi.shape[0]

            # compute the priors
            self.prior[i] = ni / n

            # compute the sample mean
            self.mu[i] = np.mean(Xi, axis = 0)

            # compute the centered data
            XiBar = Xi - self.mu[i]

            # compute the sample covariance
            self.Sigma[i] = (1/ni) * XiBar.T @ XiBar
```

```python
class BayesClassifier:
    def fit(self, X, Y):
        # find the unique labels
        uniqueY = np.unique(Y)

        # find the dimensions
        n = X.shape[0]
        self.d = X.shape[1]
        self.k = uniqueY.shape[0]

        # initialize the outputs
        self.prior = np.zeros([self.k, 1])
        self.mu = np.zeros([self.k, self.d])
        self.Sigma = np.zeros([self.k, self.d, self.d])

        # compute class prior probabilities, sample means, and sample covariances
        for i, y in enumerate(uniqueY):
            # split the X into its classes
            Xi = X[Y == y]

            # compute the size of each class
            ni = Xi.shape[0]

            # compute the priors
            self.prior[i] = ni / n

            # compute the sample mean
            self.mu[i] = np.mean(Xi, axis = 0)

            # compute the centered data
            XiBar = Xi - self.mu[i]

            # compute the sample covariance
            self.Sigma[i] = (1/ni) * XiBar.T @ XiBar
```

```python
    def predict(self, X):
        n = X.shape[0]

        discriminants = np.zeros([n, self.k])

        for i, x in enumerate(X):
            x = np.atleast_2d(x).T

            for j in range(self.k):
                invCov = np.linalg.inv(self.Sigma[j])
                discriminants[i][j] = x.T @ invCov @ self.mu[j].T - (1/2) * self.mu[j] @ invCov @ self.mu[j].T + np.log(self.prior[j])

        predictions = np.argmax(discriminants, axis = 1)

        return predictions
```

```python
    def predict(self, X):
        n = X.shape[0]

        posteriorPre = np.zeros([n, self.k])

        # compute the pdf term of the posterior probabilities
        for i in range(n):
            for j in range(self.k):
                posteriorPre[i][j] = scipy.stats.multivariate_normal.pdf(X[i], self.mu[j], self.Sigma[j], allow_singular = True)

        # compute a vector proportional to the posterior probabilities
        posterior = posteriorPre * self.prior.T

        # find the label for each datapoint by choosing the most probable class
        predictions = np.argmax(posterior, axis = 1)

        return predictions
```

Friedman (1989) proposed a compromise between LDA and QDA, which allows one to shrink the separate covariances of QDA toward a common covariance as in LDA. These methods are very similar in flavor to ridge regression. The regularized covariance matrices have the form

$$\hat{\mathbf{\Sigma}}_k(\alpha) = \alpha\hat{\mathbf{\Sigma}}_k + (1-\alpha)\hat{\mathbf{\Sigma}}, \tag{4.13}$$

where $\hat{\mathbf{\Sigma}}$ is the pooled covariance matrix as used in LDA. Here $\alpha \in [0,1]$ allows a continuum of models between LDA and QDA, and needs to be specified. In practice α can be chosen based on the performance of the model on validation data, or by cross-validation.

Similar modifications allow $\hat{\mathbf{\Sigma}}$ itself to be shrunk toward the scalar covariance,

$$\hat{\mathbf{\Sigma}}(\gamma) = \gamma\hat{\mathbf{\Sigma}} + (1-\gamma)\hat{\sigma}^2\mathbf{I} \tag{4.14}$$

for $\gamma \in [0,1]$. Replacing $\hat{\mathbf{\Sigma}}$ in (4.13) by $\hat{\mathbf{\Sigma}}(\gamma)$ leads to a more general family of covariances $\hat{\mathbf{\Sigma}}(\alpha,\gamma)$ indexed by a pair of parameters.

hyper

Regularization + Shrinkage DA Implementation

The scikit-learn implementations do not allow both as suggested in *ESL*, so we can write our own.

```
from sklearn.base import BaseEstimator, ClassifierMixin, TransformerMixin

class DA(ClassifierMixin, BaseEstimator):
    def __init__(self, equalCovariances = True, alpha = 1, gamma = 1):
        # if True, LDA
        # if False, QDA
        self.equalCovariances = equalCovariances

        # if less than 1, regularized DA (requires QDA)
        self.alpha = alpha

        # if less than 1, shrinkage
        self.gamma = gamma
```

```
    def fit(self, X, Y):
        # find the unique labels
        uniqueY = np.unique(Y)

        # find the dimensions
        n = X.shape[0]
        self.d = X.shape[1]
        self.k = uniqueY.shape[0]

        # initialize the variables
        self.prior = np.zeros([self.k, 1])
        self.mu = np.zeros([self.k, 1, self.d])

        # compute the covariance matrix
        if self.equalCovariances or self.gamma < 1 or self.alpha < 1:
            mu = np.mean(X, axis = 0)
            Xbar = X - mu
            self.Sig = (1/n) * Xbar.T @ Xbar

            # shrinkage
            if self.gamma < 1:
                self.Sig = self.gamma * self.Sig + (1 - self.gamma) * np.diag(self.Sig)

            self.invCov = np.linalg.inv(self.Sig)

        if not self.equalCovariances:
            self.Sigma = np.zeros([self.k, self.d, self.d])
```

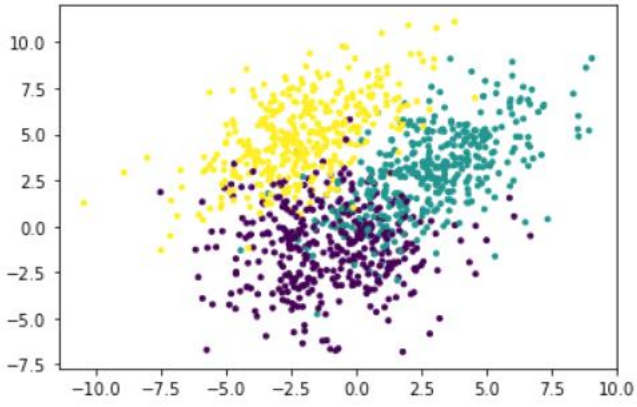
```
        for i, y in enumerate(uniqueY):
            # extract a class of datapoints from X
            Xi = X[Y == y]

            # compute the size of each class
            ni = Xi.shape[0]

            # compute the priors
            self.prior[i] = ni / n

            # compute the feature means within the class
            self.mu[i] = np.mean(Xi, axis = 0)

            # compute separate covariances for QDA
            if not self.equalCovariances:
                # compute the centered data
                XiBar = Xi - self.mu[i]

                # compute the class sample covariance
                self.Sigma[i] = (1/ni) * XiBar.T @ XiBar

            # regularization
            if self.alpha < 1:
                self.Sigma[i] = self.alpha * self.Sigma[i] + (1 - self.alpha) * self.Sig
```

```
def predict(self, X):
    n = X.shape[0]

    discriminants = np.zeros([n, self.k])

    for i, x in enumerate(X):
        x = np.atleast_2d(x).T

        for j in range(self.k):
            if not self.equalCovariances:
                self.invCov = np.linalg.inv(self.Sigma[j])

            discriminants[i][j] = x.T @ self.invCov @ self.mu[j].T - (1/2) * self.mu[j] @ self.invCov @ self.mu[j].T + np.log(self.prior[j])


    predictions = np.argmax(discriminants, axis = 1)

    return predictions

def score(self, X, y, sample_weight = None):
    return accuracy_score(y, self.predict(X), sample_weight = sample_weight)
```

So CV works

Example: Random Points

```
# number of points to generate
numberOfPoints = 500

# generate points from class 0
mean1 = np.array([-1, -1])
covariance1 = np.array([[5, 0], [0, 5]])
X1 = np.random.multivariate_normal(mean1, covariance1, numberOfPoints)

# generate points from class 1
mean2 = np.array([3, 3])
covariance2 = np.array([[5, 3], [3, 5]])
X2 = np.random.multivariate_normal(mean2, covariance2, numberOfPoints)

# generate points from class 2
mean3 = np.array([-2, 5])
covariance3 = np.array([[5, 3], [3, 5]])
X3 = np.random.multivariate_normal(mean3, covariance3, numberOfPoints)

# stack the points
X = np.vstack((X1, X2, X3))

# create a vector of the labels
Y = np.hstack((numberOfPoints * [0], numberOfPoints * [1], numberOfPoints * [2]))

# randomly choose 75% of the data to be the training set and 25% for the testing set
trainX, testX, trainY, testY = train_test_split(X, Y, test_size = 0.25, random_state = 1)

# plot the training set
plt.scatter(trainX[:,0], trainX[:,1], c = trainY, marker = '.')
```



```
# initialize accuracy and hyperparameter list
bestAccuracy = [0, 0, 0]

# test regularization hyperparameters 0.00, 0.01, ..., 0.19
for i in range(1, 11):
    for j in range(1, 11):
        alpha = i/10
        gamma = j/10

        # build the QDA classifier
        model = DA(False, alpha, gamma)

        # fit the QDA classifier to the training data
        model.fit(trainX, trainY)

        # compute the test predictions
        predictedY = model.predict(testX)

        # find the mean cross-validation accuracy
        mean_cv_scores = np.mean(cross_val_score(model, trainX, trainY, cv = 5))

        # print quality metrics
        print('Mean CV accuracy for parameters', alpha, gamma, 'is', mean_cv_scores)

        # save the hyperparameter reg_param if better than found before
        if mean_cv_scores > bestAccuracy[0]:
            bestAccuracy = [mean_cv_scores, alpha, gamma]

print('\nThe best dev accuracy', bestAccuracy[0], 'occured with alpha =', bestAccuracy[1], 'and gamma =', bestAccuracy[2])
```

```
# build the QDA classifier
model = DA(False, bestAccuracy[1], bestAccuracy[2])

# fit the QDA classifier to the training data
model.fit(trainX, trainY)

# predict the labels of the test set
predictedY = model.predict(testX)

# print quality metrics
print('\nTest Classification Report for the best hyperparameters:\n\n', classification_report(testY, predictedY))

print('\nTest Confusion Matrix:\n')
sn.heatmap(confusion_matrix(testY, predictedY))
```

```
Mean CV accuracy for parameters 1.0 0.9 is 0.8604444444444445
Mean CV accuracy for parameters 1.0 1.0 is 0.8604444444444445

The best dev accuracy 0.8666666666666666 occured with alpha = 0.6 and gamma = 1.0

Test Classification Report for the best hyperparameters:

              precision    recall  f1-score   support

           0       0.82      0.82      0.82       120
           1       0.87      0.85      0.86       127
           2       0.90      0.91      0.91       128

    accuracy                           0.86       375
   macro avg       0.86      0.86      0.86       375
weighted avg       0.86      0.86      0.86       375
```

Ledroit-Wolf Shrinkage

Excerpts of their paper…

Our philosophy is different. Consider the sample covariance matrix S and a highly structured estimator, denoted by F. We find a compromise between the two by computing a convex linear combination $\delta F + (1-\delta)S$, where δ is a number between 0 and 1. This technique is called *shrinkage*, since the sample covariance matrix is 'shrunk' towards the structured estimator. The number δ is referred to as the *shrinkage constant*.[3] Intuitively, it measures the weight that is given to the structured estimator. Shrinkage estimators have a long and successful history in statistics. The beauty of the principle is that by properly combining two 'extreme' estimators one can obtain a 'compromise' estimator that performs better than either extreme. To make a somewhat sloppy anal-

Any shrinkage estimator has three ingredients: An estimator with no structure, an estimator with a lot of structure, and a shrinkage constant. The estimator without structure is generally quite obvious, given the context. For us it is the sample covariance matrix. Less obvious are the choice of the structured estimator, or shrinkage target, and the shrinkage constant.

3.2 Shrinkage Target

The shrinkage target should fulfill two requirements at the same time: it involves only a small number of free parameters (that is, a lot of structure) but it also reflects important characteristics of the unknown quantity being estimated. Ledoit and Wolf (2003) suggest the single-factor matrix of Sharpe (1963) as the shrinkage target. In this paper we make a different suggestion: the *constant correlation model*. In our experience, it gives comparable performance but is easier to implement. The model says that all the (pairwise) correlations are identical.[4] The estimation of the model is straightforward. The average of all the sample correlations is the estimator of the common constant correlation. This number together with the vector of sample variances implies our shrinkage target, denoted by F in the remainder of the paper. A formal description of the shrinkage target is provided in Appendix A; in particular, see equation (3).

The population and sample correlations between the returns on stocks i and j are given by

$$\varrho_{ij} = \frac{\sigma_{ij}}{\sqrt{\sigma_{ii}\sigma_{jj}}} \qquad \text{and} \qquad r_{ij} = \frac{s_{ij}}{\sqrt{s_{ii}s_{jj}}}$$

The average population and sample correlations are given by

$$\bar{\varrho} = \frac{2}{(N-1)N}\sum_{i=1}^{N-1}\sum_{j=i+1}^{N}\varrho_{ij} \qquad \text{and} \qquad \bar{r} = \frac{2}{(N-1)N}\sum_{i=1}^{N-1}\sum_{j=i+1}^{N} r_{ij}$$

Define the population constant correlation matrix Φ by means of the population variances and the average population correlation:

$$\phi_{ii} = \sigma_{ii} \qquad \text{and} \qquad \phi_{ij} = \bar{\varrho}\sqrt{\sigma_{ii}\sigma_{jj}}$$

Correspondingly, define the sample constant correlation matrix F by means of the sample variances and the average sample correlation:

$$f_{ii} = s_{ii} \qquad \text{and} \qquad f_{ij} = \bar{r}\sqrt{s_{ii}s_{jj}} \tag{3}$$

This matrix F is the shrinkage target introduced in Subsection 3.2.

3.3 Shrinkage Constant

The obvious practical problem is which value to choose for the shrinkage constant. Any choice of δ strictly between 0 and 1 would yield a compromise between S and F. But this results in infinitely many possibilities. Intuitively, there is an 'optimal' shrinkage constant. It is the one that minimizes the expected distance between the shrinkage estimator and the true covariance matrix. Call this number δ^*. Appendix B derives a formula for estimating δ^*. The estimated optimal shrinkage constant is denoted $\hat{\delta}^*$; see equation (5) in Appendix B. Our operational shrinkage estimator of the covariance matrix Σ is now ready for use[5]:

$$\hat{\Sigma}_{Shrink} = \hat{\delta}^* F + (1-\hat{\delta}^*)S \tag{2}$$

(See the paper for the formula -- pages 12-15)