Gaussian Discriminant Analysis

CS6140: Machine Learning

Uzair Ahmad

Introduction

Previously, we've discussed the Naïve Bayes Model, where components of the input vector x_i are discrete-valued. When features of x_i are continuous-valued random variables, instead of using the Gaussian Naïve Bayes Model, we could use the Gaussian Discriminant Analysis Model (GDA), in which we assume that, given the label y_i , the input x_i follows a multivariate Gaussian distribution. The GDA Model is also a generative model that can be applied to classification tasks.

GDA model parameters

GDA is also a generative model, thus we have $P(y_i|\mathbf{x_i}) = \frac{P(\mathbf{x_i}|y_i)P(y_i)}{P(\mathbf{x_i})}$. Since the task is to classify an example, here we could make an assumption of the label such that y_i follows a Bernoulli distribution specified by parameter π. $y_i \sim Bernoulli(\pi)$.

Then, given the fact that the label is known, we can make another assumption about the input variables such that $x_i|y_i = 1$ and $x_i|y_i = 0$ follow the multivariate Gaussian distribution specified by (μ_0, Σ) and (μ_1, Σ), respectively.

$x_i|y_i = 0 \sim N(\mu_1, \Sigma)$

$xi|yi = 1 \sim N(\mu_0, \Sigma)$

Given the distribution parameters are known, we have

$$P(y_i) = \pi^{y_i}(1-\pi)^{1-y_i}$$

$$P(\mathbf{x_i}|\mathbf{y_i = 0}) = \frac{\mathbf{1}}{\mathbf{(2\pi)^{\frac{m+1}{2}}|\Sigma|^{\frac{1}{2}}}}\mathbf{exp(-\frac{1}{2}(x_i - \mu_0)^T\Sigma^{-1}(x_i - \mu_0))}$$

$$P(\mathbf{x_i}|\mathbf{y_i = 1}) = \frac{\mathbf{1}}{\mathbf{(2\pi)^{\frac{m+1}{2}}|\Sigma|^{\frac{1}{2}}}}\mathbf{exp(-\frac{1}{2}(x_i - \mu_1)^T\Sigma^{-1}(x_i - \mu_1))}$$

where $\mu_0, \mu_1 \in R^m$ is the mean vector; $\mathbf{\Sigma} \in \mathbf{R^{m\times m}}$ is the covariance matrix; and the $|\mathbf{\Sigma}| \in \mathbf{R}$ is the determinant of $\mathbf{\Sigma}$.

Notice that computing the probability of x under each class conditional density is equivalent to calculating the distance from x to the center of each class, μ_i, using Mahalanobis distance.

$d = \sqrt{(x_i - \mu_1)^T\Sigma^{-1}(x_i - \mu_1)}$

Therefore, GDA can be thought of as a nearest centroids classifier.

Training phase

According to the assumptions we've made above, GDA has the following parameters.

- π, which specifies $P(y_i)$;
- μ_0, Σ which specifies $P(\mathbf{x_i}|y_i = 0)$;
- μ_1, Σ which specifies $P(\mathbf{x_i}|y_i = 1)$;

Thus, we could write down the log-likelihood of the data $l(\pi, \mu_0, \mu_1, \Sigma) = ln \prod_{i=1}^{N} P(\mathbf{x_i}|y_i)P(y_i)$. By Maximizing l, we could get the optimal parameters as followings

$$\pi^* = \frac{\sum_{i=1}^{N} \mathbb{1}(yi=1)}{N}$$

$$\mu_0^* = \frac{\sum_{i=1}^{N} \mathbf{x_i}\mathbb{1}(yi=0)}{\sum_{i=1}^{N} \mathbb{1}(yi=0)}$$

$$\mu_1^* = \frac{\sum_{i=1}^{N} \mathbf{x_i}\mathbb{1}(y_i=1)}{\sum_{i=1}^{N} \mathbb{1}(y_i=1)}$$

$$\sum_{y_i}^* = \frac{\sum_{i=1}^{N} (\mathbf{x_i}-\mu_{y_i})(\mathbf{x_i}-\mu_{y_i})^T}{N}$$

where $\mathbb{1}(y_i = 0)$ is the indicator function such that if $y_i = 0$,the $\mathbb{1}(yi = 0)$ outputs 1, otherwise 0.

Prediction phase

After a model is trained, we can make prediction on the label of a given data such that $y_i = argmax_{y_i} P(\mathbf{x_i}|y_i, \theta)P(y_i)$.

If class priors are uniform, then the test data point can be classified finding

$$P(x|y_i, \theta) = argmin_{y_i} P(\mathbf{x}|y_i) = (\mathbf{x} - \mu_i)^T \Sigma_{y_i}^{-1}(\mathbf{x} - \mu_i)$$

Here we assume that the covariance is common among all classes. In case each class has a different covariance, the resulting boundary will be quadratic also known as **Quadratic Discriminant Analysis**.

```
class GDA():
    def __init__(self):
        self.pi = None
        self.mu0 = None
        self.mu1 = None
        self.sigma = None

    def fit(self, x, y):
        self.pi = np.mean(y)
        self.mu0 = np.mean(x[y[:,0]==0], axis=0)
        # centroid of class 0
        self.mu1 = np.mean(x[y[:,0]==1], axis=0)
        # centroid of class 1

        n_x = x[y[:,0] == 0] - self.mu0
        p_x = x[y[:,0] == 1] - self.mu1

        self.sigma = ((n_x.T).dot(n_x) + (p_x.T).dot(p_x))/x.shape[0]
        self.sigma_inv = np.linalg.inv(self.sigma)

    def predict(self, x):
        p0 = np.sum(np.dot((x-self.mu0),self.sigma_inv)*(x-self.mu0),axis=1)*self.pi
        p1 = np.sum(np.dot((x-self.mu1),self.sigma_inv)*(x-self.mu1),axis=1)*self.pi
        return p1 >= p0
```