Lecture 4

1. Gaussians
2. Gaussian discriminant analysis
3. Generative vs Discriminative comparison
4. Naive Bayes
5. Laplace Smoothing

1 Gaussians

Definition. $x \sim N(\vec{\mu}, \Sigma)$ if

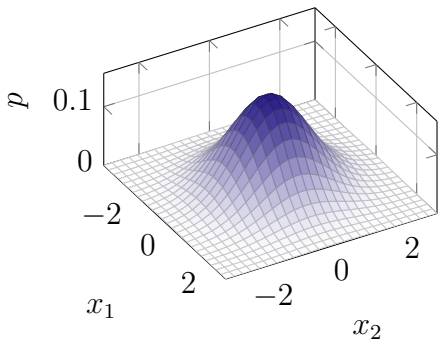
$$p(x) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\vec{\mu})^T\Sigma^{-1}(x-\vec{\mu})\right),$$

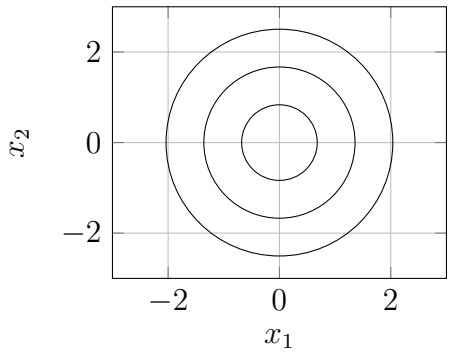
where $\vec{\mu}$ is a mean vector, and Σ is a symmetrical covariance matrix:

$$\Sigma = E\left[(x-\vec{\mu})(x-\vec{\mu})^T\right].$$

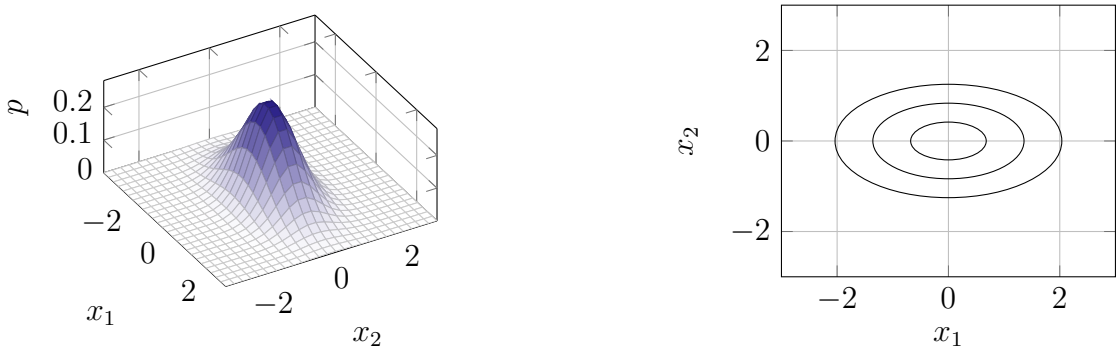
The density function $p(x)$ in case of two variables $x = (x_1, x_2)$ for multivariate gaussian distribution can be plotted as a surface. Consider the distributions with zero mean vectors and different covariance matrices:

1. $\Sigma = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$

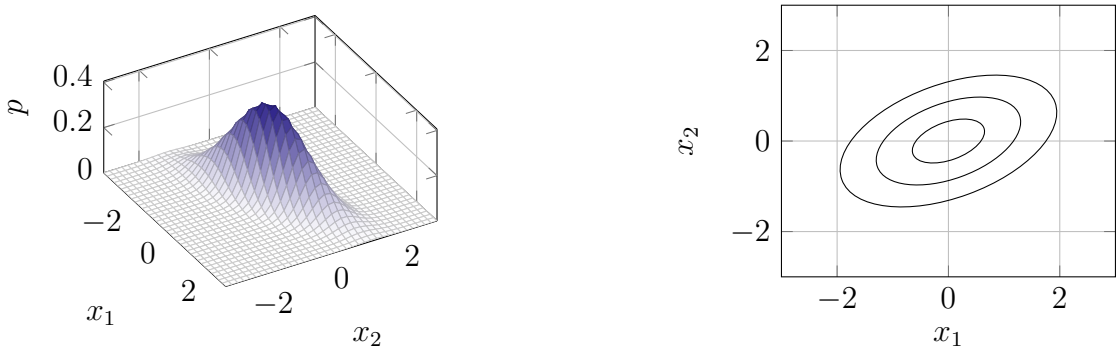




2. $\Sigma = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 0.3 \end{bmatrix}$



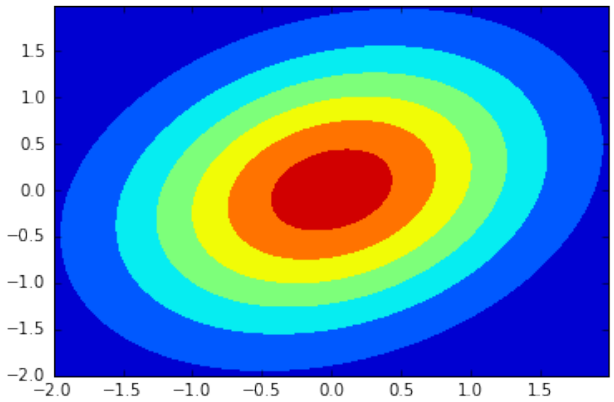
3. $\Sigma = \begin{bmatrix} 1.0 & 0.3 \\ 0.3 & 0.5 \end{bmatrix}$



1.1 Python implementation

We can visualize Gaussians in Python:

```
In [1]: sigma = [[1.0, 0.75], [0.25, 1.0]]
        print np.array(sigma)

[[ 1.    0.75]
 [ 0.25  1.  ]]

In [2]: from scipy.stats import multivariate_normal
        x, y = np.mgrid[-2:2:.01, -2:2:.01]
        pos = np.empty(x.shape + (2,))
        pos[:, :, 0] = x; pos[:, :, 1] = y
        rv = multivariate_normal([0.0, 0.0], sigma)
        plt.contourf(x, y, rv.pdf(pos))
        plt.show()
```

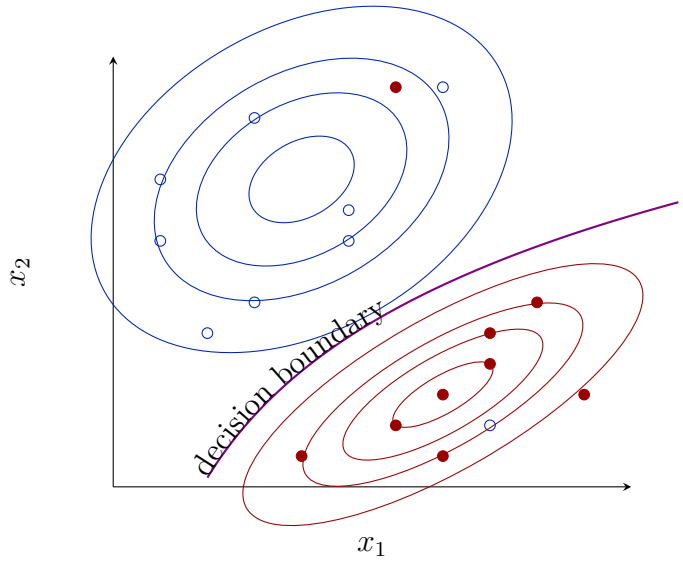


2 Gaussian discriminant analysis

Gaussian discriminant analysis is the first example of generative learning algorithm. Assuming that $x \in \mathbb{R}^n$, continuous-valued and $x|y$ is Gaussian:

$$x|y \sim N(\vec{\mu}, \Sigma).$$

For the classification problem (e.g. with two classes) we fit the gaussian distributions on features for each class separately. Visualization for the problem with two features and two classes looks as follows:



As before we use the Bernoulli pdf for two classes:

$$p(y) = \varphi^y (1-\varphi)^{1-y}$$

Based on our assumption about normality of x:

$$p(x \mid y=0) = \frac{1}{(2\pi)^{n/2}|\Sigma_0|^{1/2}} \exp\left(-\frac{1}{2}(x-\vec{\mu_0})^T \Sigma_0^{-1}(x-\vec{\mu_0})\right)$$

and

$$p(x \mid y=1) = \frac{1}{(2\pi)^{n/2}|\Sigma_1|^{1/2}} \exp\left(-\frac{1}{2}(x-\vec{\mu_1})^T\Sigma_1^{-1}(x-\vec{\mu_1})\right).$$

Notice that $\varphi, \vec{\mu_0}, \vec{\mu_1}, \Sigma_0$ and Σ_1 are parameters for our model. For the case of two classes and two features the number of parameters is $1+2+2+4+4=13$. This formulation is the most general and sometimes it refers as Quadratic Discriminant Analysis (as the resulted decision boundary will have the shape of quadratic function). We can simplify this model by assuming that two classes share the same $\Sigma_0 = \Sigma_1 = \Sigma$, which leads to the linear decision boundary (Linear Discriminant Analysis).

The log-likelihood for this model can be written as

$$l(\varphi, \mu_0, \mu_1, \Sigma) = \ln \prod_{i=1}^{m} p(x^{(i)}, y^{(i)}) = \ln \prod_{i=1}^{m} p(x^{(i)} \mid y^{(i)}) \cdot p(y^{(i)})$$

This function has a different form compared to the logistic regression, where we had

$$l(\theta) = \ln \prod_{i=1}^{m} p(y^{(i)} \mid x^{(i)}; \theta).$$

Maximization of the log-likelihood with respect to parameters (for the case of Linear Discriminant Analysis) gives the following values for the parameters:

$$\varphi = \frac{\sum\limits_i y^{(i)}}{m} = \frac{\sum\limits_i \mathbb{1}\{y^{(i)}=1\}}{m} \text{ (average of values for the target variable);}$$

$$\mu_0 = \frac{\sum\limits_i \mathbb{1}\{y^{(i)}=0\} \cdot x^{(i)}}{\sum\limits_i \mathbb{1}\{y^{(i)}=0\}} \text{ (average of } x^{(i)} \text{ inside the class 0);}$$

$$\mu_1 = \frac{\sum\limits_i \mathbb{1}\{y^{(i)}=1\} \cdot x^{(i)}}{\sum\limits_i \mathbb{1}\{y^{(i)}=1\}} \text{ (average of } x^{(i)} \text{ inside the class 1);}$$

$$\Sigma_0 = \Sigma_1 = \Sigma = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu_{y^{(i)}}) \cdot (x^{(i)} - \mu_{y^{(i)}})^T.$$

The prediction step uses the Bayes rule to maximize $p(y \mid x)$:

$$\arg\max_y p(y \mid x) = \arg\max_y \frac{p(x \mid y)p(y)}{p(x)} = \arg\max_y p(x \mid y)p(y).$$

In case when the target classes are equally balanced that means

$$p(y=1) = p(y=0) = \frac{1}{2},$$

the prediction step is transformed to:

$$\arg\max_y p(y \mid x) = \arg\max_y p(x \mid y).$$

2.1 Python implementation

Loading necessary libraries:

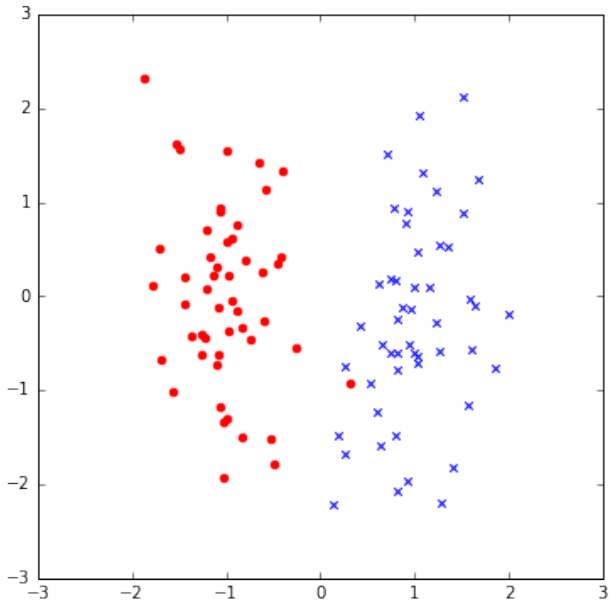
```
In [3]: from matplotlib import cm
        import matplotlib.pyplot as plt
        import numpy as np
        import sklearn.datasets as ds
        %matplotlib inline
```

Gaussian Discriminant Analysis is implemented in the `scikit-learn` package:

```
In [4]: from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
        from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
```

Generate training examples with two classes and two features:

```
In [5]: X,y = ds.make_classification(n_features=2,
                                     n_redundant=0,
                                     n_informative=1,
                                     n_clusters_per_class=1,
                                     random_state=3216)
        markers = ['o', 'x']
        ix0 = [i for i,x in enumerate(y) if x == 0]
        ix1 = [i for i,x in enumerate(y) if x == 1]
        fig = plt.figure(figsize=(6,6))
        plt.scatter(X[ix0,0],X[ix0, 1],marker='o',color='red')
        plt.scatter(X[ix1,0],X[ix1, 1],marker='x',color='blue')
        plt.show()
```

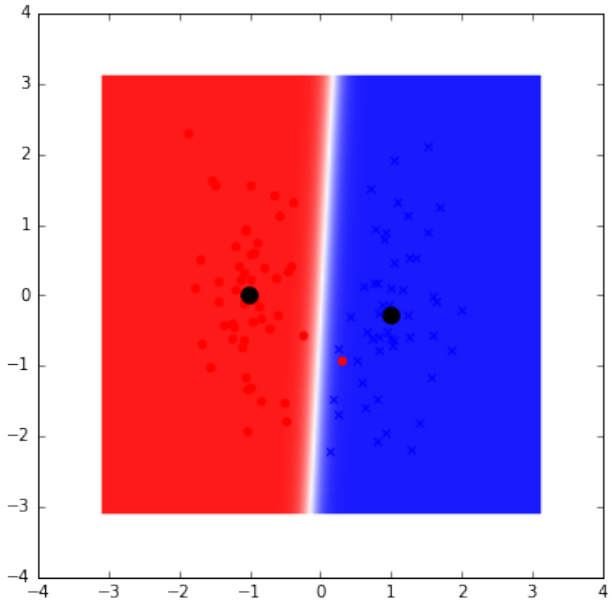


There are two implementations of Gaussian Discriminant Analysis in `scikit-learn`: Linear Discriminant Analysis (LDA) and Quadratic Discriminant Analysis (QDA). We try to use both and compare the results. First, we train a model for Linear Discriminant Analysis:

```
In [6]: lda = LinearDiscriminantAnalysis(solver="svd", store_covariance=True)
        y_pred = lda.fit(X, y).predict(X)
```

Visualize the result:

```
In [7]: x1, x2 = np.meshgrid(np.linspace(-3.1, 3.1, 200),
                            np.linspace(-3.1, 3.1, 200))
        y_pred = 1.0 - lda.predict_proba(np.c_[x1.ravel(), x2.ravel()])
        y_pred = y_pred[:, 1].reshape(x1.shape)
        extent = -3.1, 3.1, -3.1, 3.1

        fig = plt.figure(figsize=(6,6))
        plt.imshow(y_pred, cmap=cm.bwr, alpha=.9,
                   interpolation='bilinear', extent = extent, origin='lower')
        plt.scatter(X[ix0,0],X[ix0, 1],marker='o',color='red')
        plt.scatter(X[ix1,0],X[ix1, 1],marker='x',color='blue')

        #means
        plt.plot(lda.means_[0][0], lda.means_[0][1],
                 'o', color='black', markersize=10)
        plt.plot(lda.means_[1][0], lda.means_[1][1],
                 'o', color='black', markersize=10)

        plt.show()
```

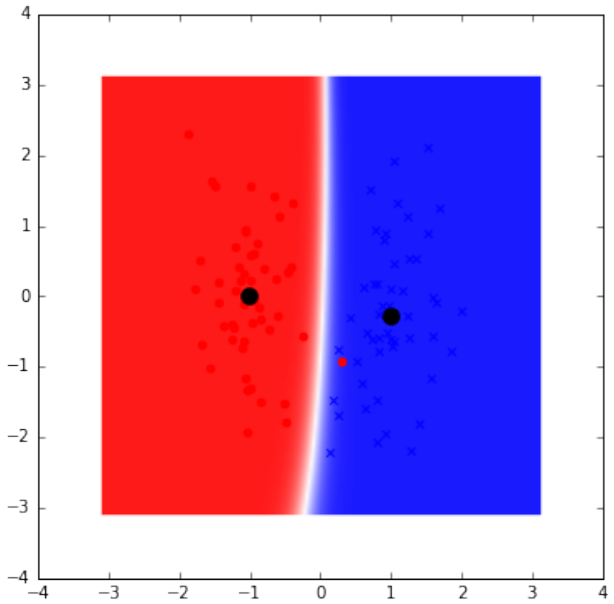


We do the same for Quadratic Discriminant Analysis:

```
In [8]: qda = QuadraticDiscriminantAnalysis(store_covariances=True)
        y_pred = qda.fit(X, y).predict(X)
```

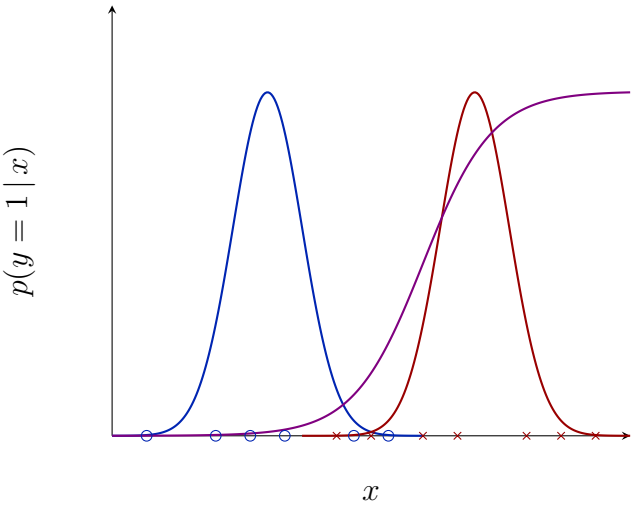
and visualize the results:

```
In [9]: x1, x2 = np.meshgrid(np.linspace(-3.1, 3.1, 200),
                             np.linspace(-3.1, 3.1, 200))
        y_pred = 1.0 - qda.predict_proba(np.c_[x1.ravel(), x2.ravel()])
        y_pred = y_pred[:, 1].reshape(x1.shape)
        extent = -3.1, 3.1, -3.1, 3.1

        fig = plt.figure(figsize=(6,6))
        plt.imshow(y_pred, cmap=cm.bwr, alpha=.9,
                   interpolation='bilinear', extent = extent, origin='lower')
        plt.scatter(X[ix0,0],X[ix0, 1],marker='o',color='red')
        plt.scatter(X[ix1,0],X[ix1, 1],marker='x',color='blue')

        #means
        plt.plot(lda.means_[0][0], lda.means_[0][1],
                 'o', color='black', markersize=10)
        plt.plot(lda.means_[1][0], lda.means_[1][1],
                 'o', color='black', markersize=10)

        plt.show()
```



3 Generative vs Discriminative comparison

For the simplest case of one feature and two classes, we fit two univariate normal distributions for each class:



The class with $y^{(i)} = 0$ is denoted by circles, the class with $y^{(i)} = 1$ - by crosses. The violet curve on the picture shows the graph of the function

$$p(y = 1 \mid x) = \frac{p(x \mid y = 1)p(y)}{p(x)} \text{ (Bayes rule),}$$

and as we can see, it has the shape of the sigmoid function with respect to x. This fact can be formulated as follows:

Theorem. If $x \mid y \sim N(\mu, \Sigma)$, then the posterior $p(y = 1 \mid x)$ has a logistic shape.

The opposite statement is not correct. In fact, if we have, for example, if $x \mid y = 1 \sim$ Poisson(λ_1) and $x \mid y = 0 \sim$ Poisson(λ_0), then the posterior $p(y = 1 \mid x)$ is logistic as well. And more generally,

Theorem. If $x \mid y = 1 \sim$ ExpFamily(η_1) and $x \mid y = 0 \sim$ ExpFamily(η_0), then the posterior $p(y = 1 \mid x)$ is logistic.

The preliminary condition for using the Gaussian Discriminant Analysis is our confidence that $x \mid y$ is Gaussian, otherwise, we should use the logistic regression. In fact, the assumption about normality of input variables is a very strong assumption and does not hold for the majority of data. Because of this assumption the Gaussian Discriminant Analysis needs less data to build a model.

4 Naive Bayes

Another example of generative learning algorithm is called Naive Bayes. Consider the email spam filtering problem with two classes $y \in \{0, 1\}$, 0 means that the email is not a spam, 1 - otherwise. To build the feature vector we collect the bug of words from all emails. For the given email we compose the feature vector as a list of indicator functions for each word. For example, this email

"Dear Purchasing Manager,
Thank you for taking the time to read my email.
Our Policy is to delivery quality as well as having very good price, please, find attached a price list is both interior and exterior lighting products, including the LED high bay, LED tube and LED Panel light.
Please contact with me more the informations, specifications and price, shipment etc.
Best Regards, Sunny Sales Manager"

will be transformed to the following feature vector:

$$x = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ \vdots \\ 1 \\ \vdots \\ 0 \\ \vdots \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} a \\ as \\ advert \\ and \\ \vdots \\ bay \\ \vdots \\ buy \\ \vdots \\ claim \\ \vdots \\ light \\ \vdots \\ zip \end{matrix}$$

The number of features n equals to the number of all words: $x \in \{0,1\}^n$. The number of all combinations is 2^n, for example, if $n = 50\,000$, then we have $2^{50\,000}$ combinations and 50 000 parameters. To fit the model $p(x \,|\, y)$ with such huge number of parameters we make a strong assumption to simplify it. In Naive Bayes algorithm we assume that x_i's are conditionally independent given y (which means that the appearance of one word does not depend on the existence of another word in the email):

$$\begin{aligned} &p(x_1, x_2, \ldots, x_n \,|\, y) = p(x_1 \,|\, y) \cdot p(x_2 \,|\, y, x_1) \cdot \ldots \cdot p(x_n \,|\, y, x_1, \ldots, x_{n-1}) = \\ &= p(x_1 \,|\, y) \cdot p(x_2 \,|\, y) \cdot \ldots \cdot p(x_n \,|\, y) = \prod_{j=1}^{n} p(x_j \,|\, y). \end{aligned}$$

Obviously, this assumption is false. However, the model with this assumption gives decent results. The parameters of the model are defined by

$$\begin{aligned} &\varphi_{j\,|\,y=1} = p(x_j = 1 \,|\, y = 1), \\ &\varphi_{j\,|\,y=0} = p(x_j = 1 \,|\, y = 0), \\ &\varphi_y = p(y = 1). \end{aligned}$$

As before, to fit the parameters we write down the joint likelihood:

$$L(\varphi_y, \varphi_{j\,|\,y=1}, \varphi_{j\,|\,y=0}) = \prod_{i=1}^{m} p(x^{(i)}, y^{(i)}).$$

and complete the maximum likelihood estimation (MLE), which gives the values for the parameters:

$$\varphi_y = \frac{\sum_{i=1}^{m} \mathbb{1}\{y^{(i)} = 1\}}{m},$$
$$\varphi_{j\,|\,y=1} = \frac{\sum_{i=1}^{m} \mathbb{1}\{x_j^{(i)} = 1, y^{(i)} = 1\}}{\sum_{i=1}^{m} \mathbb{1}\{y^{(i)} = 1\}},$$
$$\varphi_{j\,|\,y=0} = \frac{\sum_{i=1}^{m} \mathbb{1}\{x_j^{(i)} = 1, y^{(i)} = 0\}}{\sum_{i=1}^{m} \mathbb{1}\{y^{(i)} = 0\}}.$$

Notice that $\sum_{i=1}^{m} \mathbb{1}\{y^{(i)} = 1\}$ is a number of spam emails, $\sum_{i=1}^{m} \mathbb{1}\{x_j^{(i)} = 1, y^{(i)} = 1\}$ is a number of spam emails with word j and so on. The final step will be using these parameters to infer $p(y\,|\,x)$ as

$$p(y = 1\,|\,x) = \frac{p(x\,|\,y = 1) \cdot p(y = 1)}{p(x\,|\,y = 1) \cdot p(y = 1) + p(x\,|\,y = 0) \cdot p(y = 0)} \text{ (Bayes rule).}$$

4.1 Python implementation

There are several packages in Python with implemented Naive Bayes. The package `scikit-learn` has several implementations: `GaussianNB`, `MultinomialNB` and `BernoulliNB`. In this section we are going to use different implementation from `nltk` package that is convenient for text classification. We are going to use the `movie_reviews` dataset from this package:

```
In [10]: import nltk
         from nltk.corpus import movie_reviews
         import random
         documents = [(list(movie_reviews.words(fileid)), category)
                      for category in movie_reviews.categories()
                      for fileid in movie_reviews.fileids(category)]
         random.shuffle(documents)
```

This dataset contains 2000 movie reviews manually labeled by 'pos' (positive review) or 'neg' (negative review).

```
In [11]: labels = [x[1] for x in documents]
         print ("Number of positive reviews: " +
                str(len([x for x in labels if x == 'pos'])))
         print ("Number of negative reviews: " +
                str(len([x for x in labels if x == 'neg'])))
```

```
Number of positive reviews: 1000
Number of negative reviews: 1000
```

We choose the 2000 most frequent words and define the function that generates binary feature vector x based on these 2000 words for each document.

```
In [12]: all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())
         word_features = list(all_words)[:2000]

         def document_features(document):
             document_words = set(document)
             features = {}
             for word in word_features:
                 features['contains({})'.format(word)] = (word in document_words)
             return features
```

Here is the example how this function works:

```
In [13]: from itertools import islice
         feature = document_features(['batmans', 'inquires', 'rags'])
         for key in islice(feature, 10):
             print key + ': ' + str(feature[key])
```

```
contains(corporate): False
contains(barred): False
contains(batmans): True
contains(menacing): False
contains(rags): True
contains(inquires): True
contains(nosebleeding): False
contains(playhouse): False
contains(peculiarities): False
contains(kilgore): False
```

The next step builds train and test sets. We add randomly 1900 documents to the train set and other 100 documents to the test set.

```
In [14]: featuresets = [(document_features(d), c) for (d,c) in documents]
         train_set, test_set = featuresets[100:], featuresets[:100]
         print "Size of train set is: " + str(len(train_set))
         print "Size of test set is: " + str(len(test_set))
```

```
Size of train set is: 1900
Size of test set is: 100
```

The final step is to train Navie Bayes model and check the accuracy:

```
In [15]: classifier = nltk.NaiveBayesClassifier.train(train_set)
         print "Accuracy: " + str(nltk.classify.accuracy(classifier, test_set))
```

```
Accuracy: 0.63
```

We can also check the most important words based on their probabilities:

```
In [16]: classifier.show_most_informative_features(10)
```

```
Most Informative Features
          contains(sans) = True              neg : pos    =      8.9 : 1.0
    contains(mediocrity) = True              neg : pos    =      7.6 : 1.0
     contains(dismissed) = True              pos : neg    =      7.1 : 1.0
   contains(bruckheimer) = True              neg : pos    =      6.3 : 1.0
     contains(uplifting) = True              pos : neg    =      5.9 : 1.0
           contains(ugh) = True              neg : pos    =      5.7 : 1.0
   contains(overwhelmed) = True              pos : neg    =      5.7 : 1.0
       contains(topping) = True              pos : neg    =      5.7 : 1.0
        contains(doubts) = True              pos : neg    =      5.5 : 1.0
  contains(effortlessly) = True              pos : neg    =      5.3 : 1.0
```

5 Laplace smoothing

The last formula should be elaborated more. Assuming that we never meet some word in spam emails, for example, $p(x_{20\,000} = 1|y = 1) = 0$, the computation of $p(y = 1 \,|\, x)$ becomes:

$$p(y = 1 \,|\, x) = \frac{p(x \,|\, y = 1) \cdot p(y = 1)}{p(x \,|\, y = 1) \cdot p(y = 1) + p(x \,|\, y = 0) \cdot p(y = 0)} = \\ = \frac{\prod_{i=1}^{n} p(x_i = 1 \,|\, y = 1) \cdot p(y = 1)}{p(x \,|\, y = 1) \cdot p(y = 1) + p(x \,|\, y = 0) \cdot p(y = 0)} = 0.$$

In the last expression the numerator is zero, because one of term in the product $p(x_{20\,000} = 1|y = 1) = 0$. Based on the common sense we would say that if we do not have a spam email with some word, then we cannot imply that in the future we will not have a spam email with this word. Similarly, we could have the situation that $p(x_i = 1|y = 0) = 0$ for some i, then the denominator in the above expression becomes zero.

To avoid these situations we apply the Laplace smoothing:

$$p(y = 1) = \frac{\#1s + 1}{\#1s + 1 + \#0s + 1}$$

in contrast with the previous section where we had

$$p(y = 1) = \frac{\#1s}{\#1s + \#0s}.$$

With the Laplace smoothing the parameters for Naive Bayes are transformed to

$$\varphi_{j\,|\,y=1} = \frac{\sum_{i=1}^{m} \mathbb{1}\{x_j^{(i)} = 1, y^{(i)} = 1\} + 1}{\sum_{i=1}^{m} \mathbb{1}\{y^{(i)} = 1\} + 2},$$
$$\varphi_{j\,|\,y=0} = \frac{\sum_{i=1}^{m} \mathbb{1}\{x_j^{(i)} = 1, y^{(i)} = 0\} + 1}{\sum_{i=1}^{m} \mathbb{1}\{y^{(i)} = 0\} + 2}.$$

In general, if we have more classes: $y \in \{1, \ldots, k\}$, then

$$p(y = j) = \frac{\sum_{i=1}^{m} \mathbb{1}\{y^{(i)} = j\} + 1}{m + k}.$$