MTH 594: Machine Learning

Dmitry Efimov

November 28, 2016

Contents

1 Introduction

Prerequisites: basic linear algebra (matrices and vectors, eigenvectors), basic statistics and probability theory, basic programming skills (Python).

Online resources:

- UCI Machine learning repository, which contains a large collection of standard datasets for testing learning algorithms:
  archive.ics.uci.edu/ml
- examples of recent work in machine learning, start by taking a look at the conferences NIPS: **old.nips.cc** and ICML. Some other related conferences include UAI, AAAI, IJCAI.
- platform for data mining competitions: **www.kaggle.com**

Goals of the course:

1. Convey my own excitement;
2. Teach how to apply algorithms;
3. Give a background for research in machine learning.

Final project:

- You can form study groups (max is 2 students per group)
- Two options for project:
  - investigate some aspect of machine learning;
  - apply machine learning algorithm to the problem you are interested in.

Course organization:

Machine Learning is one of the most interesting interdisciplinary area. It touches many different industries and you will find the stuff we take in this class very useful. During the last 15-20 years we have obtained much more capabilities for machine learning. Some examples of machine learning problems are:

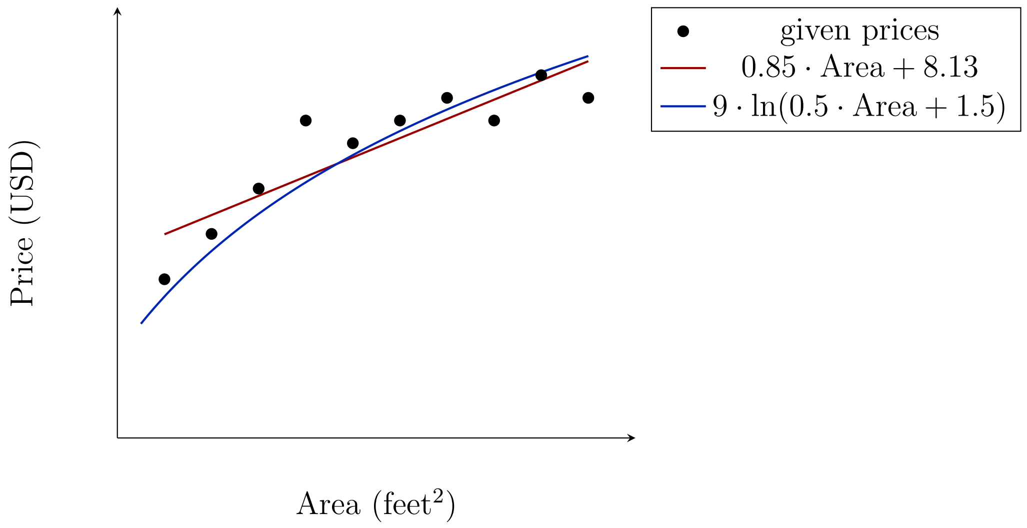
- digits recognition (amount on cheques)
- database mining (patient medical records)

- recommender systems (YouTube movies)

To solve all these problems we need learning algorithms. The course is divided in 4 sections:

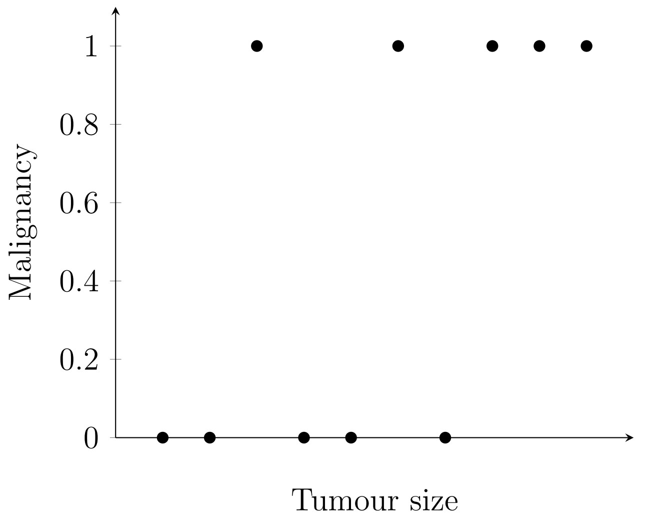
- **Supervised learning**

  For example, we can collect housing prices (in USD) and areas (in feet^2). The simplest way to visualise is to put them on the graph:

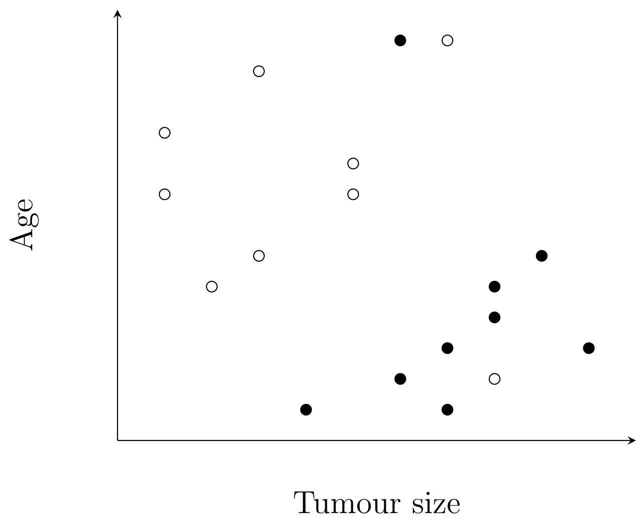


Now we can try to find some linear or more complicated function to predict the price based on the given area. We call it **regression problem**.

For another example we consider the problem where the tumour malignancy should be predicted based on the tumour size. The output for this problem is not continuous, but rather discrete: 0 means that the tumour is malignant, 1 - benign. This is a **classification problem.** Then the picture looks different:



More general example is when number of inputs is bigger than one: you may want to predict malignancy based on tumour size and age:



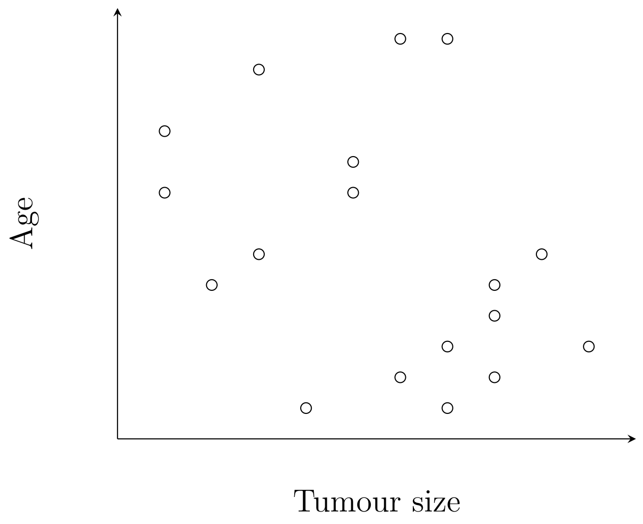
The general idea is that we should split classes by some curve. In many problems the data could have dimensions bigger than 3 and it is impossible to show it on the graph. But there are a lot of fascinating algorithms (like support vector machine) that can handle these complicated cases.

- **Learning theory**

  This part requires a lot efforts of mathematicians. We try to understand how algorithms work, what is the guarantee that the algorithm will work? The example for the previous problem is: how much training data do we need to predict prices?

- **Unsupervised learning**

  In this section we should find interesting structures in the data, for example, clusters.



Some areas that involve the unsupervised learning technique are geophysics, image processes, social network analysis, market segmentation. Another example of unsupervised learning is cocktail party problem: we should split voices of different people based on audio records. The solution of this problem could be obtained by the independent component analysis (ICA).

- **Recommender systems**

  In this part we have the set of users and set of objects (for example, set of movies from YouTube) with ratings given by users to the objects. If we have missed ratings, we should complete them, i.e. how would user rate the given movie? The example of the algorithm that solves this kind of problems is factorization machine.

2 Basic notations

- m: number of training examples
- x: input variables or features (notice that x is a column vector)
- y: output variable or target
- (x, y): training example (sample)
- $(x^{(i)}, y^{(i)})$: i-th training example

Part I: Supervised Learning

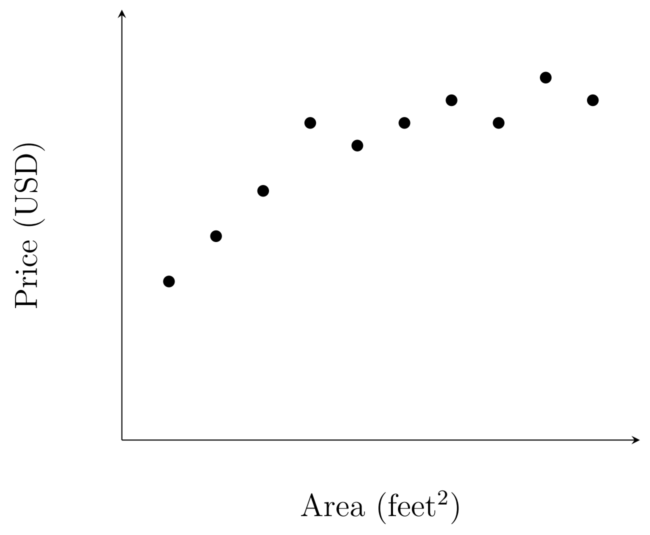
- Arthur Samuel (1959). Machine Learning: field of study that gives computers the ability to learn without being explicitly programmed. He wrote checker program computer against himself. Computer learned how to play checkers better than Arthur Samuel.
- Tom Mitchell (1998) Well-posed learning problem: A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.

The general workflow for the supervised learning can be visualized by the following diagram:

$$\begin{tikzcd}
\boxed{\text{Training set}} \arrow[r] & \boxed{\text{Learning algorithm}} \arrow[r] & \boxed{\text{Hypothesis } h}
\end{tikzcd}$$

As an example, we consider predicting the housing prices: given the living area in feet^2 the goal is to predict the price. The hypothesis represents the relationship between living area and price ($m = 10$):

Living area (feet^2)	**USD** ($\times$1000)
2104	400
1416	232
1534	315
852	178
1940	240
...	...



3 Linear regression

Linear regression hypotheses that the output variable or target can be predicted via the following model:

$$h(x) = \sum_{i=0}^{n} \theta_i x_i = \theta^T x$$

where $x_0 = 1$, n is a number of features, θ_i are unknown **parameters**. When $n = 1$, the hypothesis reduces to:

$$h(x) = \theta_0 + \theta_1 x$$

For $n = 2$, $h(x)$ assumes the form:

$$h(x) = h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

The task of learning algorithm is to learn parameter values from the training set, i.e. estimate θ from the given data values. The accuracy of the learning algorithm is usually measured by a cost (error) function which is naturally chosen to be:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \tag{3.1}$$

where $y^{(i)}$ are the observed target values. $J(\theta)$ is minimized with respect to θ to find the best parameter estimate.

3.1 Analytical minimization: normal equations

Due to the linearity of the hypothesized model, minimization of $J(\theta)$ in equation (3.1) can produce analytical solution. The solution can be derived using Linear Algebra terminology. But we need to take derivatives with respect to matrices.

Given the function we need to find

$$\nabla_\theta J = \begin{bmatrix} \dfrac{\partial J}{\partial \theta_0} \\ \vdots \\ \dfrac{\partial J}{\partial \theta_n} \end{bmatrix} \in \mathbb{R}^{n+1}$$

Gradient descent iteration transforms to

$$\theta := \theta - \alpha \nabla_\theta J,$$

where left-hand and right-hand sides are $n+1$-dimensional vectors.

Definition. $f : \mathbb{R}^{m\times n} \to \mathbb{R}$ or $f(A), A \in \mathbb{R}^{m\times n}$. Then

$$\nabla_A f(A) = \begin{bmatrix} \dfrac{\partial f}{\partial A_{11}} & \cdots & \dfrac{\partial f}{\partial A_{1n}} \\ \vdots & & \vdots \\ \dfrac{\partial f}{\partial A_{m1}} & \cdots & \dfrac{\partial f}{\partial A_{mn}} \end{bmatrix}$$

Some facts from the Linear Algebra:

- $\operatorname{tr} AB = \operatorname{tr} BA$
- $\operatorname{tr} ABC = \operatorname{tr} CAB = \operatorname{tr} BCA$
- If $f(A) = \operatorname{tr} AB$, then $\nabla_A \operatorname{tr} AB = B^T$
- $\operatorname{tr} A = \operatorname{tr} A^T$
- If $a \in \mathbb{R}$ then $\operatorname{tr} a = a$
- $\nabla_A \operatorname{tr} ABA^TC = CAB + C^TAB^T$

Denote

$$X\theta = \begin{bmatrix} ---(x^{(1)})^T--- \\ ---(x^{(2)})^T--- \\ \cdots \\ ---(x^{(m)})^T--- \end{bmatrix} \theta = \begin{bmatrix} (x^{(1)})^T\theta \\ (x^{(2)})^T\theta \\ \cdots \\ (x^{(m)})^T\theta \end{bmatrix} = \begin{bmatrix} h_\theta(x^{(1)}) \\ h_\theta(x^{(2)}) \\ \cdots \\ h_\theta(x^{(m)}) \end{bmatrix}$$

Remember that

$$y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix},$$

then

$$X\theta - y = \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \cdots \\ h_\theta(x^{(m)}) - y^{(m)} \end{bmatrix}.$$

Recall $z^T z = \sum_i z_i^2$. Then

$$\frac{1}{2}(X\theta - \vec{y})^T(X\theta - \vec{y}) = \frac{1}{2}\sum_{i=1}^{m}(h(x^{(i)}) - y^{(i)})^2 = J(\theta)$$

Now we should set gradient to zero:

$$\nabla_\theta J(\theta) = \vec{0}$$

It means

$$\nabla_\theta \left(\frac{1}{2}(X\theta - y)^T(X\theta - y)\right) = \frac{1}{2}\nabla_\theta \operatorname{tr}\left(\theta^T X^T X\theta - \theta^T X^T y - y^T X\theta + y^T y\right) =$$
$$= \frac{1}{2}\left(\nabla_\theta \operatorname{tr}\theta\theta^T X^T X - \nabla_\theta \operatorname{tr} y^T X\theta\right).$$

But

$$\nabla_\theta \operatorname{tr}\theta I\theta^T X^T X = X^T X\theta I + X^T X\theta I$$

and

$$\nabla_\theta \operatorname{tr} y^T X\theta = X^T y,$$

then

$$\nabla_\theta J(\theta) = \frac{1}{2}\left[X^T X\theta + X^T X\theta - X^T y - X^T y\right] = X^T X\theta - X^T y = 0.$$

The right hand side is called **normal equations**:

$$X^T X\theta = X^T y$$

and finally

$$\theta = (X^T X)^{-1} X^T y.$$

3.2 Python implementation

There are many useful packages in Python. To load them we use the following commands:

```
In [1]: from mpl_toolkits.mplot3d import Axes3D
        from matplotlib import cm
        import matplotlib.pyplot as plt
        import numpy as np
        import sklearn.datasets as ds
        from sklearn.linear_model import LinearRegression
```

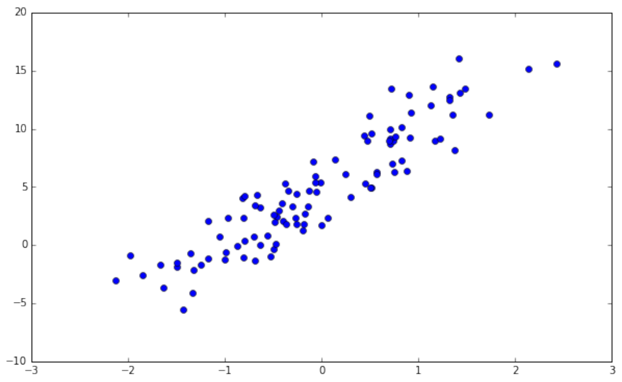
We use the package `matplotlib` for plotting, the package `numpy` for fast matrix calculations, the package `sklearn` to work with data and use different built-in methods.

Before we train our simple linear regression model we should get some data. For this example we generate data with 100 samples and 1 feature using method `make_regression` from the package `sklearn`.

```
In [2]: X_train,y_train = ds.make_regression(n_samples=100,
                                             n_features=1,
                                             n_informative=1,
                                             noise=20.0,
                                             bias=50,
                                             random_state=2016)
        y_train = y_train/10 # scale target to have nicer pictures
```

To visualize the data we use method `plot` from the `matplotlib` package.

```
In [3]: fig = plt.figure(figsize=(10,6))
        ax = plt.axes(xlim=(-3, 3), ylim=(-10, 20))
        ax.plot(X_train[:,0],y_train,'o')
        plt.show()
```



Now we build our first model using the method `LinearRegression` from the package `scikit-learn`. Notice that when you work with `scikit-learn` training different models is very similar. Usually, training contains the following steps:

1. Create a class for the model and define all parameters
2. Use the method `fit` to train the model on the training examples
3. Use the method `predict` or `predict_proba` to obtain predictions on the test examples.

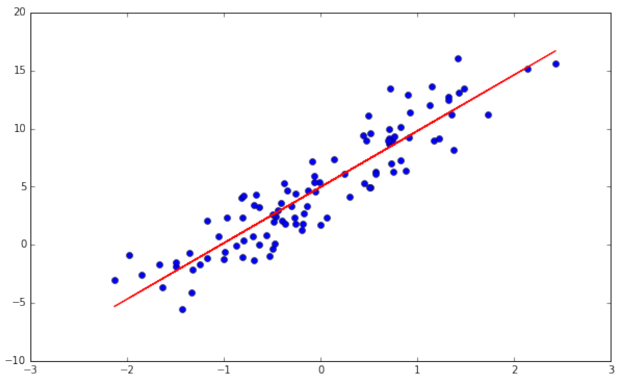
In our example we do not have any test examples, that is why we train the linear regression model and obtain the coefficients from our model.

```
In [4]: model = LinearRegression() # create an object of the LinearRegression class
        model.fit(X_train,y_train) # train the model on the training examples
        intercept = model.intercept_
        slope = model.coef_[0]
        best_fit = X_train[:,0] * slope + intercept
        print "Parameters thetas: " + str(round(model.intercept_,2)) +\
              ", " + ", ".join([str(round(x,2)) for x in model.coef_])
```

```
Parameters thetas: 4.98, 4.83
```

Finally, we visualize our data along with the regression line.

```
In [5]: fig = plt.figure(figsize=(10,6))
        ax = plt.axes(xlim=(-3, 3), ylim=(-10, 20))
        ax.plot(X_train[:,0],y_train,'o')
        line, = ax.plot([], [], lw=2)
        plt.plot(X_train[:,0], best_fit, 'k-', color = "r")
        plt.show()
```



3.3 Minimization of $J(\theta)$: gradient descent

The gradient descent minimizes $J(\theta)$ iteratively by starting from an initial random value of θ (say $\theta = \vec{0}$) and updating the parameter values using certain step size. The updated point could be obtained by choosing the direction of steepest descent (because the aim is to reach the minimum as quickly as possible). A drawback of the procedure is that, for some complicated $J(\theta)$ with multi minima, it could converge to a local minimum of $J(\theta)$, i.e., starting from different initial point could lead to a different optimum value (check the Python implementation section).

At each iteration, the value θ is updated by the following rule

$$\theta_i := \theta_i - \alpha \frac{\partial}{\partial \theta_i} J(\theta)$$

where α is the step size or the **learning rate**. If α is too small, the minimization algorithm takes long time to converge, if α is too big, the algorithm can diverge.

For the general linear regression hypothesis, the partial derivative with respect to θ_i is given by:

$$\frac{\partial}{\partial \theta_i} J(\theta) = \frac{\partial}{\partial \theta_i} \left(\frac{1}{2} (h_\theta(x) - y)^2 \right) = 2 \cdot \frac{1}{2} \cdot (h_\theta(x) - y) \cdot \frac{\partial}{\partial \theta_i} (h_\theta(x) - y) =$$
$$= (h_\theta(x) - y) \cdot \frac{\partial}{\partial \theta_i} (\theta_0 x_0 + \ldots + \theta_n x_n - y) = (h_\theta(x) - y) \cdot x_i$$

Then the updated parameter value is:

$$\theta_i := \theta_i - \alpha(h_\theta(x) - y) \cdot x_i$$

and the algorithm becomes very simple:

Algorithm 1 Batch Gradient Descent

```
1: repeat
2:     for $i = 0$ to $n$ do
3:         $\theta_i := \theta_i - \alpha \sum_{j=1}^{m} (h_\theta(x^{(j)}) - y^{(j)}) \cdot x_i^{(j)}$
4: until convergence
```

Notice again that the second part in the line 3 of the Algorithm 1 is just $\frac{\partial}{\partial \theta_i} J(\theta)$.

In our problem, $J(\theta)$ does not have a complicated form, it is just quadratic surface. It converges reasonably rapidly. The gradient is decreasing (the step becomes smaller and smaller).

The name of the algorithm Batch Gradient Descent came from the idea that we look at the whole training set every step. But when training set is very big (for example, $m > 1000000$), then we need to look at a huge amount of training samples each iteration. The alternative for this is called Stochastic Gradient Descent.

Algorithm 2 Stochastic Gradient Descent

```
1: repeat
2:     for $j = 1$ to $m$ do
3:         for $i = 0$ to $n$ do
4:             $\theta_i := \theta_i - \alpha(h_\theta(x^{(j)}) - y^{(j)}) \cdot x_i^{(j)}$
5: until convergence
```

In this algorithm we update weights based on the first training example only, after on the second training example only and so on. For large datasets the Stochastic Gradient Descent is much faster. But the problem that you do not walk to the minimum in the fastest way but you still approaching to it (check the Python implementation section).

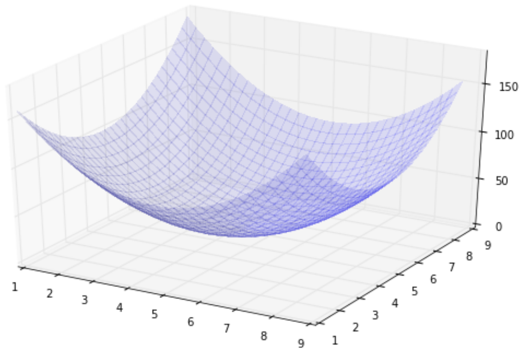
3.4 Python implementation

For our previous example with one feature we add the bias term by adding the column of 1's to the design matrix:

```
In [6]: X_train_bias = np.c_[np.ones(X_train.shape[0]), X_train]
```

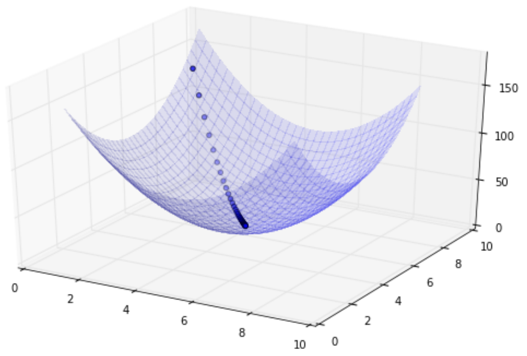
Next step will be to define the cost function $J(\theta_0, \theta_1)$ and visualize it as a function of two variables:

```
In [7]: def cost_function(x, y, theta):
            y = y.reshape((-1,1))
            theta = np.array(theta).reshape((2,-1))
            return np.sum((np.dot(x, theta) - y) ** 2, axis=0)/(2*10)

        fig = plt.figure(figsize=(10, 6))
        ax = fig.gca(projection='3d')
        plt.hold(True)
        a = np.arange(1, 9, 0.25)
        b = np.arange(1, 9, 0.25)
        a, b = np.meshgrid(a, b) # make a grid of values from a and b
        # we use some vectorization to speed up the calculations
        c = cost_function(X_train_bias, y_train,\
              np.c_[a.ravel(), b.ravel()].T).reshape(a.shape)

        surf = ax.plot_surface(a, b, c, rstride=1, cstride=1, alpha=0.1,
                                 linewidth=0, antialiased=False)
        ax.set_zlim(-0.01, 180.01)

        plt.show()
```



The following function is an implementation of Batch Gradient descent:

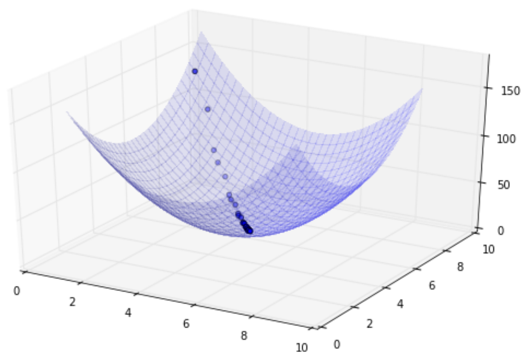
```
In [8]: def batch_gradient_descent(x, y, theta0, iters, alpha):
            theta = theta0    # initial values for theta's
            history = []      # array of theta's
            costs = []        # array of costs
            # main loop by number of iterations:
            for i in range(iters):
                history.append(theta)
                cost = cost_function(x, y, theta)[0]
                costs.append(cost)
                pred = np.dot(x, theta)
                error = pred - y
                gradient = x.T.dot(error)
                theta = theta - alpha * gradient  # update
            return history, costs

        history, costs = batch_gradient_descent(X_train_bias, y_train,
                                                theta0 = [1.2, 8],
                                                iters = 30,
                                                alpha = 0.001)
        fig = plt.figure(figsize=(10, 6))
        ax = fig.gca(projection='3d')
        plt.hold(True)
        surf = ax.plot_surface(a, b, c, rstride=1, cstride=1, alpha=0.1,
                               linewidth=0, antialiased=False)
        ax.set_zlim(-0.01, 180.01)

        t0 = np.array([x[0] for x in history])
        t1 = np.array([x[1] for x in history])
        ax.scatter(t0, t1, costs, color="k");

        plt.show()
```



As you can see, the algorithm found the fastest path to the minimum. Stochastic Gradient Descent does not find the shortest path, but in most cases it is much faster because it does not use all training examples to update the parameters:

```
In [9]: def stochastic_gradient_descent(x, y, theta0, iters, alpha):
            theta = theta0     # initial values for theta's
            history = []       # array of theta's
            costs = []         # array of costs
            m = y.size         # number of training examples
```

```
    # main loop by number of iterations:
    for i in range(iters):
        for j in range(m):
            pred = np.dot(x[j,:], theta)
            error = pred - y[j]
            gradient = error * x[j,:]
            theta = theta - alpha * gradient  # update
            if j % 40 == 0:
                history.append(theta)
                cost = cost_function(x, y, theta)[0]
                costs.append(cost)
    return history, costs

history, costs = stochastic_gradient_descent(X_train_bias, y_train,
                                             theta0 = [1.2, 8],
                                             iters = 10,
                                             alpha = 0.005)

fig = plt.figure(figsize=(10, 6))
ax = fig.gca(projection='3d')
plt.hold(True)
surf = ax.plot_surface(a, b, c, rstride=1, cstride=1, alpha=0.1,
                       linewidth=0, antialiased=False)
ax.set_zlim(-0.01, 180.01)

t0 = np.array([x[0] for x in history])
t1 = np.array([x[1] for x in history])
ax.scatter(t0, t1, costs, color="k");

plt.show()
```

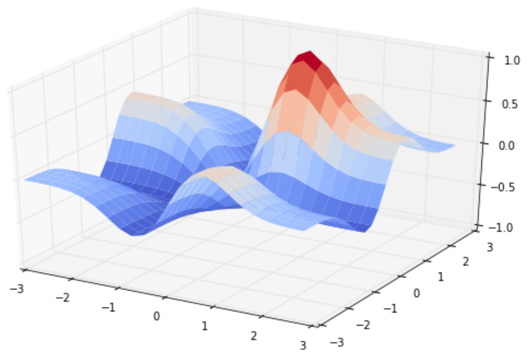


When we apply the Gradient Descent algorithm to the linear regression our straight line gradually approaching to the line which fits our data in the best way. We apply the above method `batch_gradient_descent` to fit the coefficients in the linear regression.

```
In [10]: alpha = 0.001                 # learning rate
         iters = 100                   # number of iterations
         theta = np.random.rand(2) # initial guess for theta's
         history, cost = batch_gradient_descent(X_train_bias,
                                                y_train,
                                                theta,
                                                iters,
                                                alpha)
         theta = history[-1]
         print "Parameters thetas after gradient descent: " +\
               ", ".join([str(round(x,2)) for x in theta])
]
```

```
Parameters thetas after gradient descent: 4.98, 4.82
```

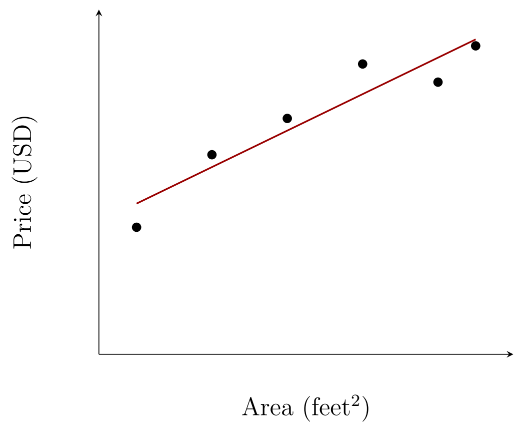
Notice that the parameters from the manual implementation are very close to the parameters from `scikit-learn`. Finally, we show the function for which gradient descent can give different results for different initial values of θ.

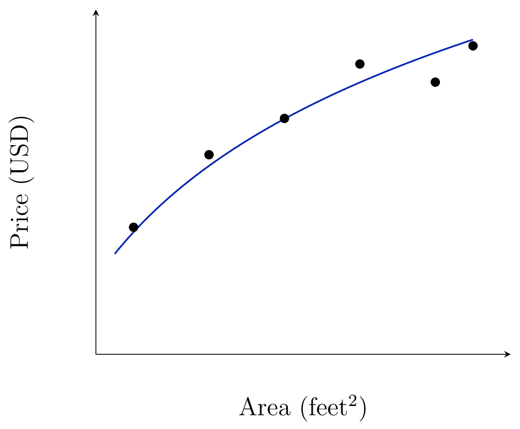


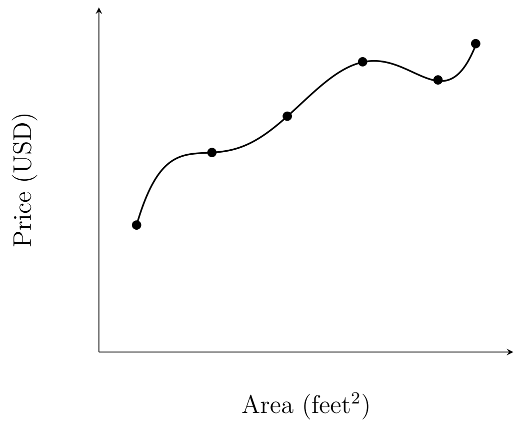
4 Locally weighted regression

Consider three different models for our previous example.

In the first case we fit using hypothesis $h_\theta(x) = \theta_0 + \theta_1 x$ and obviously that our model is not very accurate (in machine learning we call it **underfitting**). For the second case we fit the model with hypothesis $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$ and it seems that this model will be accurate enough. For the last example we fit the polynomial of the fifth order and it will fit







- underfitting
- good fit
- overfitting

data exactly, but our intuition says us that this model is not good. In machine learning we call it **overfitting**.

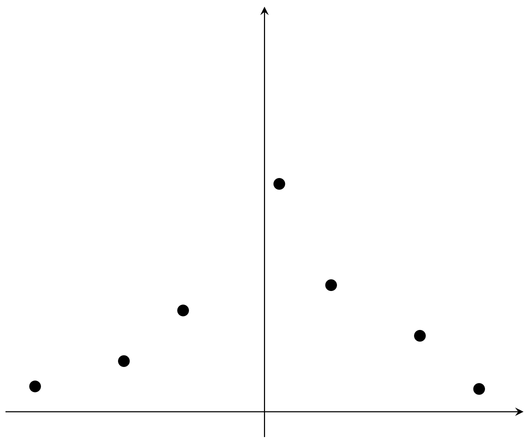
As we can see we can increase number of parameters if we increase the number of samples.

Definition. Parametric learning algorithm is the algorithm with fixed number of parameters (for example, linear regression).

Definition. Nonparametric learning algorithm is the algorithm where number of parameters grows with m.

In this section we consider the example of nonparametric algorithm: locally weighted regression (loess or lowess).

Given the following training set



In linear regression to evaluate h at certain x we fit θ to minimize $\sum_i \left(y^{(i)} - \theta^T x^{(i)}\right)^2$ and return $\theta^T x$.

In loess we look at the points from dataset that are closest to x and fit linear regression for these points only. In mathematical language we fit θ to minimize

$$\sum_i w^{(i)} \left(y^{(i)} - \theta^T x^{(i)}\right)^2$$

where $w^{(i)} = \exp\left(-\frac{(x^{(i)} - x)^2}{2\tau^2}\right)$ and τ is bandwidth parameter.

If $|x^{(i)} - x|$ is small, then $w^{(i)} \approx 1$.

If $|x^{(i)} - x|$ is large, then $w^{(i)} \approx 0$

The weight is proportional to the height of the bell-shaped curve. If τ is large the bell shape is wider.

The disadvantage of loess is that we need to fit the linear regression each time we want to predict.

4.1 Python implementation

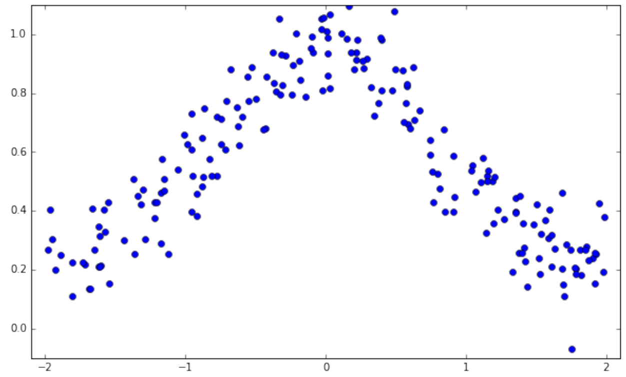
Loading necessary libraries:

```
In [1]: from matplotlib import cm
        import matplotlib.pyplot as plt
        import numpy as np
        import sklearn.datasets as ds
```

To implement the weighted linear regression we generate some nonlinear data first:

```
In [2]: X_train = np.random.uniform(low = -2, high = 2, size=200)
        y_train = 1.0/(1.0 + np.power(X_train,2)) +\
                  np.random.normal(size=len(X_train), scale=0.1)

        fig = plt.figure(figsize=(10,6))
```

```
ax = plt.axes(xlim=(-2.1, 2.1), ylim=(-0.1, 1.1))
ax.plot(X_train,y_train, 'o')
plt.show()
```



We will use the stochastic gradient descent to find the best values of θ.

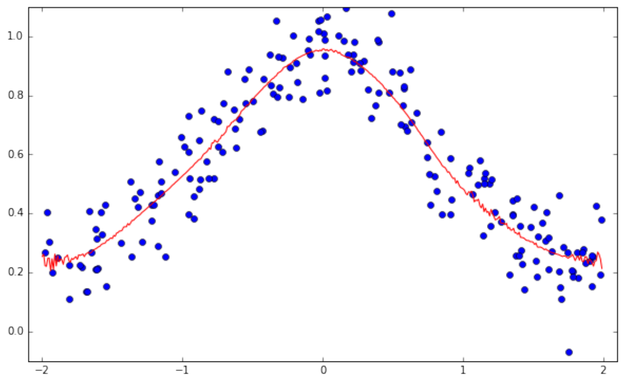
```
In [3]: def fit_loess(X_train, y_train, x, iters, alpha, tau):
            # add bias column to the design matrix
            X_train_bias = np.c_[np.ones(X_train.shape[0]), X_train]
            y_train = y_train.reshape((-1, 1))
            ### calculate weights for given parameter tau
            w = np.exp(-1.0/(2.0*tau**2)*(X_train - x)**2).reshape((-1,1))
            m = y_train.size # number of training examples
            theta = np.random.rand(2).reshape((-1,1)) # random start
            # the main loop by the number of iterations:
            for i in range(iters):
                pred = np.dot(X_train_bias, theta)
                error = np.multiply(w, pred - y_train)
                gradient = X_train_bias.T.dot(error)/m
                theta = theta - alpha * gradient  # update weights
            return np.dot(np.array([1.0, x]).reshape((1,-1)), theta)[0]
```

Notice that we should fit the weighted linear regression for each test example separately.

```
In [4]: alpha = 0.1 # set step-size
        iters = 1000 # set number of iterations
        tau = 0.2
        X_test = np.arange(-2, 2, 0.01)
        y_pred = [fit_loess(X_train, y_train, x, iters, alpha, tau)
                  for x in X_test]
```

Finally, we plot our predictions along with the training examples.

```
In [5]: fig = plt.figure(figsize=(10,6))
        ax = plt.axes(xlim=(-2.1, 2.1), ylim=(-0.1, 1.1))
        ax.plot(X_train, y_train, 'o')
        ax.plot(X_test, y_pred, color='r')
        plt.show()
```



5 Linear regression: probabilistic interpretation

In the previous lecture we agreed that the cost function $J(\theta)$ is defined as a sum of squared differences. The reasonable question is why we choose the function $J(\theta)$ in such form? In this section we look at the same problem from different point of view.

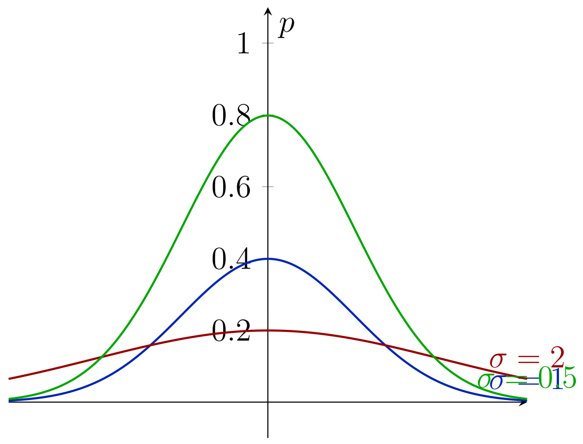
We assume that $y^{(i)} = \theta^T x^{(i)} + \varepsilon^{(i)}$, where $\varepsilon^{(i)}$ is an error term distributed normally:

$$\varepsilon^{(i)} \sim N(0, \sigma^2) \text{ (gaussian distribution).}$$

The density for this error is defined as

$$p(\varepsilon^{(i)}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\varepsilon^{(i)})^2}{2\sigma^2}\right),$$

where σ is a standard deviation and shows the width of the bell-shaped density function.



Then

$$p(y^{(i)}|x^{(i)}, \theta) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right)$$

In other words,

$$y^{(i)}|x^{(i)}, \theta \sim N(\theta^T x^{(i)}, \sigma^2)$$

If we assume that θ is not a random variable, but some value, we say that we parametrize distribution by θ. Notice that $\varepsilon^{(i)}$'s are independently identically distributed. We define the likelihood as

$$L(\theta) = p(y|x;\theta) = \prod_{i=1}^{m} p(y^{(i)}|x^{(i)};\theta) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right).$$

The idea of likelihood is that we take $p(y|x;\theta)$ and consider it as a function of θ. How to choose parameters θ?

Maximum likelihood estimation: choose θ to maximize $L(\theta) = p(y|x;\theta)$. We define

$$\begin{aligned} l(\theta) &= \ln L(\theta) = \ln \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right) = \\ &= \sum_{i=1}^{m} \ln\left[\frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right)\right] = \\ &= m \ln \frac{1}{\sqrt{2\pi}\sigma} + \sum_{i=1}^{m} -\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}. \end{aligned}$$

So maximizing $l(\theta)$ is the same as minimizing

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(y^{(i)} - \theta^T x^{(i)})^2$$

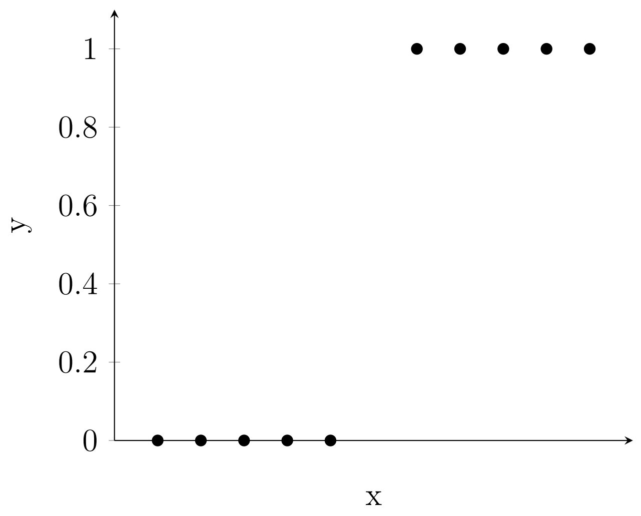
Notice that the standard deviation σ is not important for the optimization.

Exercise. Assuming that $y \sim N(\mu, \sigma^2)$ (that means that we predict constant value μ for all test examples), what is the best value of μ?

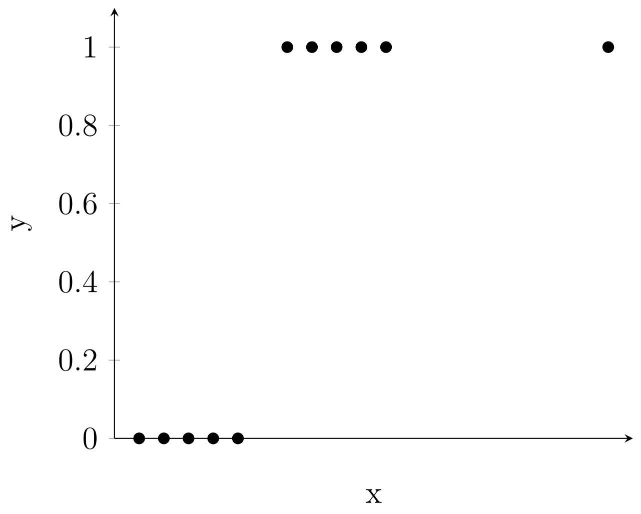
Hint: $p(y|x;\mu) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y-\mu)^2}{2\sigma^2}\right)$

6 Logistic regression

Let us build our first classification algorithm. The simplest classification problem claims that the output $y \in \{0, 1\}$.

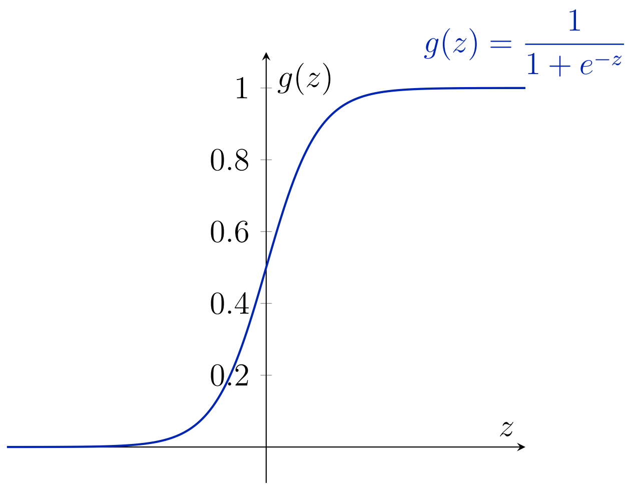


One way to solve this problem is to train linear regression, and after we take some threshold for the straight line. Sometimes it works, but in general it is not a good idea. Here is the example, when this does not work.



For the second case linear regression gives very bad result. The good idea is to change our hypothesis such that $h_\theta(x) \in [0,1]$. But with this assumption linear function is not the best choice for $h_\theta(x)$. We choose $h_\theta(x)$ as follows:

$$\begin{aligned} h_\theta(x) &= g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}, \\ g(z) &= \frac{1}{1+e^{-z}} - \text{ sigmoid function or logistic function.} \end{aligned}$$



To understand the logistic regression from the probabilistic point of view we define the probability density function as

$$\begin{aligned} p(y=1|x;\theta) &= h_\theta(x) \\ p(y=0|x;\theta) &= 1 - h_\theta(x), \end{aligned}$$

then

$$p(y|x;\theta) = h_\theta(x)^y \left(1 - h_\theta(x)\right)^{1-y}.$$

As before, the likelihood

$$L(\theta) = p(y|X;\theta) = \prod_{i=1}^{m} p(y^{(i)}|x^{(i)};\theta) = \prod_{i=1}^{m} h_\theta(x^{(i)})^{y^{(i)}} \left(1 - h_\theta(x^{(i)})\right)^{1-y^{(i)}}$$

and we maximize the log-likelihood

$$l(\theta) = \ln L(\theta) = \sum_{i=1}^{m} y^{(i)} \ln h_\theta(x^{(i)}) + (1 - y^{(i)}) \ln(1 - h_\theta(x^{(i)})).$$

To do find the value for θ that maximizes the log-likelihood we apply gradient ascent:

$$\theta := \theta + \alpha \nabla_\theta l(\theta),$$

where

$$\frac{\partial}{\partial \theta_j} l(\theta) = \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}.$$

Then gradient ascent can be written as

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)},$$

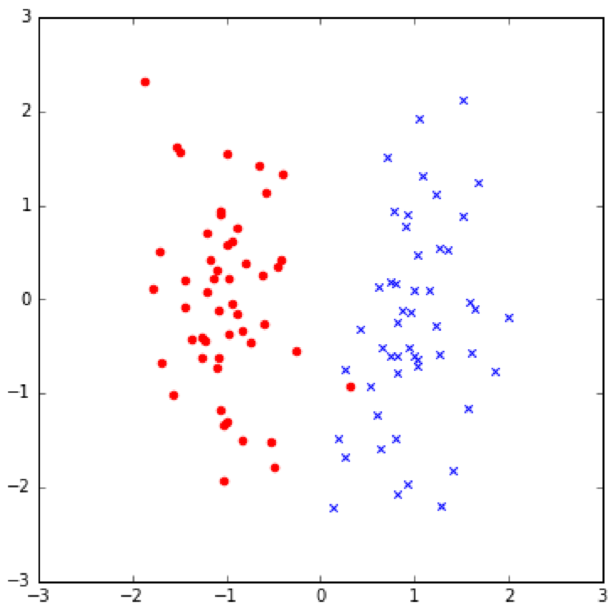
which is exactly the same rule as for least square regression, except that $h_\theta(x)$ is different.

6.1 Python implementation

```
In [6]: from sklearn.linear_model import LogisticRegression
```

To illustrate the logistic regression algorithm we generate data with 100 samples and 2 features:

```
In [7]: X_train, y_train = ds.make_classification(n_features=2,
                                                  n_redundant=0,
                                                  n_informative=1,
                                                  n_clusters_per_class=1,
                                                  random_state=3216)
        ix0 = [i for i,x in enumerate(y_train) if x == 0]
        ix1 = [i for i,x in enumerate(y_train) if x == 1]
        fig = plt.figure(figsize=(6,6))
        plt.scatter(X_train[ix0,0],X_train[ix0, 1],marker='o',color='red')
        plt.scatter(X_train[ix1,0],X_train[ix1, 1],marker='x',color='blue')
        plt.show()
```



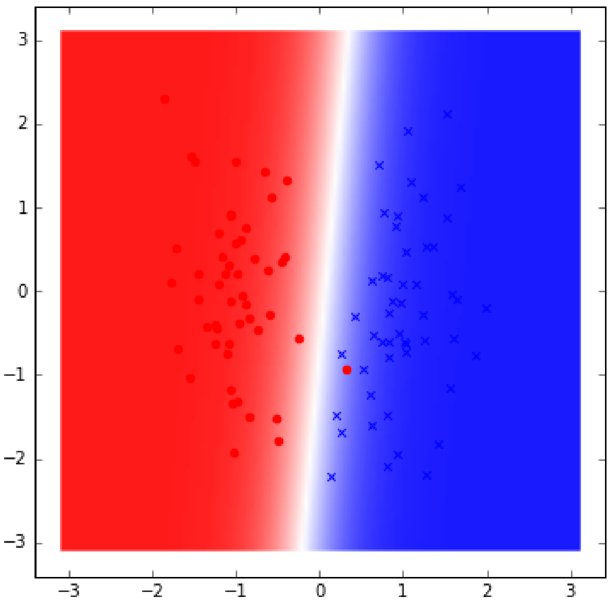
In similar way we train the logistic regression model from `scikit-learn` package:

```
In [8]: model = LogisticRegression()
        model.fit(X_train,y_train)
```

and visualize the results:

```
In [9]: def sigmoid(x1, x2, th0, th1, th2):
            return -1.0/(1+np.exp(-th0 - x1*th1 - x2*th2))
```

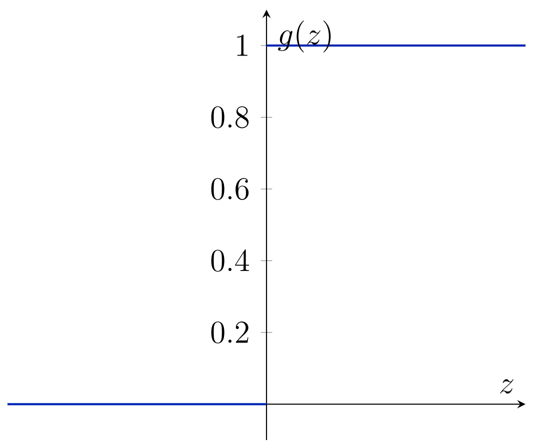
```
x1 = np.arange(-3.1, 3.1, 0.05)
x2 = np.arange(-3.1, 3.1, 0.05)
x1,x2 = np.meshgrid(x1, x2)
y_pred = sigmoid(x1,x2,model.intercept_[0],
                model.coef_[0][0],model.coef_[0][1])
extent = -3.1, 3.1, -3.1, 3.1

fig = plt.figure(figsize=(10,6))
plt.imshow(y_pred, cmap=cm.bwr, alpha=.9, interpolation='bilinear',
           extent = extent, origin='lower')
plt.scatter(X_train[ix0,0],X_train[ix0, 1],marker='o',color='red')
plt.scatter(X_train[ix1,0],X_train[ix1, 1],marker='x',color='blue')
plt.show()
```



7 Perceptron

Define the function

$$g(z) = \begin{cases} 1 & \text{if } z \geqslant 0, \\ 0 & \text{otherwise.} \end{cases}$$



If the hypothesis is expressed as $h_\theta(x) = g(\theta^T x)$, then learning rule becomes:

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)}$$

It turns out that it is very difficult to use probabilistic semantics in this learning rule. But in the future we are going to use this rule as a building block of some algorithms.

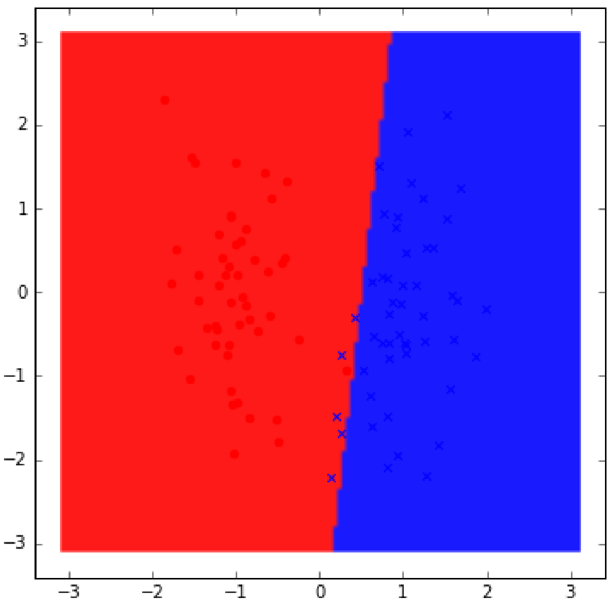
7.1 Python implementation

The perceptron algorithm is trained on the data we used for the logistic regression:

```
In [10]: from sklearn.linear_model import Perceptron

In [11]: model = Perceptron()
         model.fit(X_train,y_train)
```

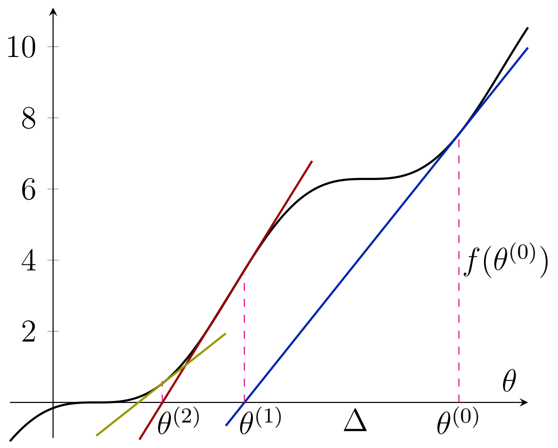
The results of the perceptron algorithm can be visualized in the following way:

```
In [12]: def heaviside(x1, x2, th0, th1, th2):
             return -np.sign(th0 + x1*th1 + x2*th2)

         x1 = np.arange(-3.1, 3.1, 0.05)
         x2 = np.arange(-3.1, 3.1, 0.05)
         x1,x2 = np.meshgrid(x1, x2)
         y_pred = heaviside(x1,x2,model.intercept_[0],
                            model.coef_[0][0],model.coef_[0][1])
         extent = -3.1, 3.1, -3.1, 3.1

         fig = plt.figure(figsize=(10,6))
         plt.imshow(y_pred, cmap=cm.bwr, alpha=.9, interpolation='bilinear',
                       extent = extent)
         plt.scatter(X_train[ix0,0],X_train[ix0, 1],marker='o',color='red')
         plt.scatter(X_train[ix1,0],X_train[ix1, 1],marker='x',color='blue')
         plt.show()
```



8 Newton's method

Newton's method is an algorithm that help to solve an equation $f(\theta) = 0$.



We start from some random $\theta^{(0)}$, by definition:

$$f'(\theta^{(0)}) = \frac{f(\theta^{(0)})}{\Delta},$$

which implies that $\Delta = \dfrac{f(\theta^{(0)})}{f'(\theta^{(0)})}$. Then

$$\theta^{(1)} = \theta^{(0)} - \frac{f(\theta^{(0)})}{f'(\theta^{(0)})}$$

or, in general,

$$\theta^{(t+1)} = \theta^{(t)} - \frac{f(\theta^{(t)})}{f'(\theta^{(t)})}.$$

To maximize likelihood $l(\theta)$ we find θ such that $l'(\theta) = 0$, then one iteration of the Newton's methods is written as

$$\theta^{(t+1)} = \theta^{(t)} - \frac{l'(\theta^{(t)})}{l''(\theta^{(t)})}.$$

This algorithm has a quadratic convergence. It means that error decreasing as square after each iteration. For example, in linear regression we need just 3 iterations to find the parameters with very good accuracy.

When the problem is to find several parameters, one iteration of the Newton's methods can be written as a vector equation:

$$\theta^{(t+1)} = \theta^{(t)} - H^{-1}\nabla_\theta l,$$

where θ is a vector and H is the Hessian matrix:

$$H_{ij} = \frac{\partial^2 l}{\partial\theta_i\partial\theta_j}.$$

Notice that if we want to minimize something (instead of maximization) the algorithm does not change.

9 Exponential family

In this section we generalise the ideas of linear and logistic regressions. Before we considered two cases:

- $y \in \mathbb{R} \Rightarrow$ Gaussian distribution $N(\mu, \sigma^2) \Rightarrow$ linear regression
- $y \in \{0, 1\} \Rightarrow$ Bernoulli distribution with parameter φ such that $p(y = 1|\varphi) = \varphi \Rightarrow$ logistic regression

We will show that in both cases we deal with particular cases of general class of exponential family distributions:

$$p(y; \eta) = b(y) \exp\left(\eta^T T(y) - a(\eta)\right),$$

where η is a natural parameter, $T(y)$ is a sufficient statistics (in many cases $T(y) = y$).

For different choices of a, b and T we will have different distributions:

1. Bernoulli distribution with φ: $p(y = 1; \varphi) = \varphi$. Then

$$p(y; \varphi) = \varphi^y(1-\varphi)^{1-y} = \exp\left(\ln\left(\varphi^y(1-\varphi)^{1-y}\right)\right) =$$
$$= \exp\left(y\ln\varphi + (1-y)\ln(1-\varphi)\right) = \exp\left(\ln\frac{\varphi}{1-\varphi}y + \ln(1-\varphi)\right).$$

By introducing the notations $\eta = \ln \frac{\varphi}{1-\varphi}$, $T(y) = y$, $-a(\eta) = \ln(1-\varphi)$, $b(y) = 1$, we obtain the exponential family formula. Notice that we can find

$$\varphi = \frac{1}{1+e^{-\eta}},$$

then $a(\eta) = \ln(1+e^{\eta})$.

2. Gaussian distribution $N(\mu, \sigma^2)$. For simplicty we set $\sigma^2 = 1$. Then

$$p(y;\mu) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}(y-\mu)^2\right) = \ldots =$$
$$= \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}y^2\right) \exp\left(\mu y - \frac{1}{2}\mu^2\right).$$

By introducing the notations $b(y) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}y^2\right)$, $\eta = \mu$, $T(y) = y$, $a(\eta) = \frac{1}{2}\eta^2$, we obtain the exponential family formula.

10 Generalized Linear Models (GLM)

The Generalized Linear Models use the exponential family distributions to build different algorithms in one general framework. Assuming some distribution for the output y we obtain the form for the hypothesis $h_\theta(x)$ and find the parameters by finding the maximum of log-likelihood. We assume the following:

- $y \mid x; \theta \sim \text{ExpFamily}(\eta)$
- given x our goal is to output $E[T(y) \mid x]$, in other words,

$$h(x) = E[T(y) \mid x]$$

- $\eta = \theta^T x$ (in more general case if $\eta \in \mathbb{R}^k$, then $\eta_i = \theta_i^T x$)

Examples of GLM:

1. **Bernoulli distribution**

   - $y \mid x; \theta \sim \text{ExpFamily}(\eta)$
   - for fixed x and θ, the algorithm output is

$$h_\theta(x) = E[y \mid x;\theta] = p(y=1 \mid x;\theta) = \varphi = \frac{1}{1+e^{-\eta}} = \frac{1}{1+e^{-\theta^T x}}$$

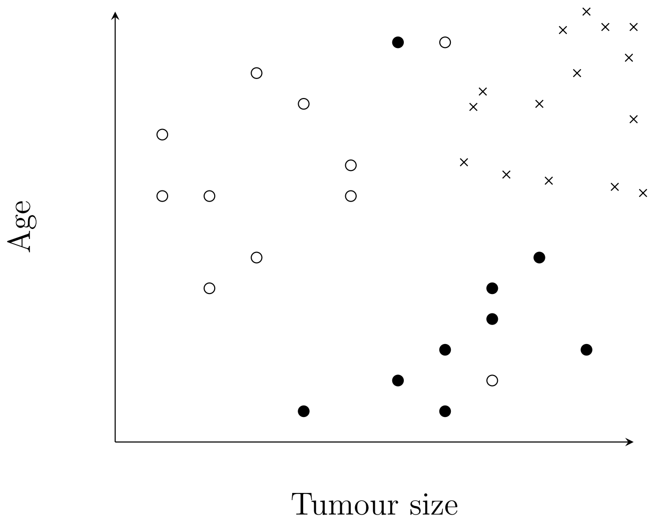
Definition.

$g(\eta) = E[y \mid \eta] = \frac{1}{1+e^{-\eta}}$ is a canonical response function
g^{-1} is a canonical link function

2. **Gaussian distribution**

   Exercise for the homework

3. **Multinomial distribution**: $y \in \{1, \dots, k\}$



Parameters φ_1, φ_2, ..., φ_k are defined as$p(y = i) = \varphi_i$. Such formulation is redundant, because φ_k can be expressed as $\varphi_k = 1 - \varphi_1 - \ldots - \varphi_{k-1}$. That is why we do not take φ_k as a parameter. We introduce function $T(y), y \in \{1, \dots, k\}$ as follows:

$$T(1) = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \; T(2) = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \dots, T(k-1) = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

and

$$T(k) = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \; T(1), T(2), \dots, T(k) \in \mathbb{R}^{k-1}.$$

The indicator function $\mathbb{1}\{True\} = 1$, $\mathbb{1}\{False\} = 0$, for example, $\mathbb{1}\{2 = 3\} = 0$. Then $T(y)_i = \mathbb{1}\{y = i\}$. The distribution:

$$\begin{aligned} P(y) &= \varphi_1^{\mathbb{1}\{y=1\}} \cdot \varphi_2^{\mathbb{1}\{y=2\}} \cdot \ldots \cdot \varphi_k^{\mathbb{1}\{y=k\}} = \\ &= \varphi_1^{T(y)_1} \cdot \varphi_2^{T(y)_2} \cdot \ldots \cdot \varphi_{k-1}^{T(y)_{k-1}} \cdot \varphi_k^{1 - \sum\limits_{j=1}^{k-1} T(y)_j} = \ldots = \\ &= b(y) \exp\left(\eta^T T(y) - a(\eta)\right), \end{aligned}$$

where

$$\eta = \begin{bmatrix} \ln(\varphi_1/\varphi_k) \\ \vdots \\ \ln(\varphi_{k-1}/\varphi_k) \end{bmatrix} \in \mathbb{R}^{k-1}, \; a(\eta) = -\ln \varphi_k, \; b(y) = 1.$$

We could solve these equations with respect to η and obtain

$$\varphi_i = \frac{e^{\eta_i}}{1+\sum\limits_{j=1}^{k-1} e^{\eta_j}} = \frac{e^{\theta_i^T x}}{1+\sum\limits_{j=1}^{k-1} e^{\theta_j^T x}}, i = 1 \dots k-1.$$

The purpose of these transformation is to obtain the learning algorithm:

$$h_\theta(x) = E[T(y) \mid x;\theta] = E\left[\begin{array}{c} \mathbb{1}\{y=1\} \\ \vdots \\ \mathbb{1}\{y=k-1\} \end{array} \middle| x;\theta\right] =$$

$$= \begin{bmatrix} \varphi_1 \\ \vdots \\ \varphi_{k-1} \end{bmatrix} = \begin{bmatrix} \dfrac{e^{\theta_1^T x}}{1+\sum\limits_{j=1}^{k-1} e^{\theta_j^T x}} \\ \vdots \\ \dfrac{e^{\theta_{k-1}^T x}}{1+\sum\limits_{j=1}^{k-1} e^{\theta_j^T x}} \end{bmatrix}$$

This algorithm is called **softmax regression** - generalisation of logistic regression for k classes. To fit the parameters for the given training set $(x^{(1)}, y^{(1)})$, ..., $(x^{(m)}, y^{(m)})$, we find the likelihood in the form

$$L(\theta) = \prod_{i=1}^{m} p(y^{(i)} \mid x^{(i)};\theta) = \prod_{i=1}^{m} \varphi_1^{\mathbb{1}\{y^{(i)}=1\}} \cdot \varphi_2^{\mathbb{1}\{y^{(i)}=2\}} \cdot \dots \cdot \varphi_k^{\mathbb{1}\{y^{(i)}=k\}},$$

where $\varphi_1 = \dfrac{e^{\theta_1^T x}}{1+\sum\limits_{j=1}^{k-1} e^{\theta_j^T x}}$. The last step will be to find the log-likelihood, find the derivatives and use stochastic gradient descent to maximize the log-likelihood function with respect to parameters θ.

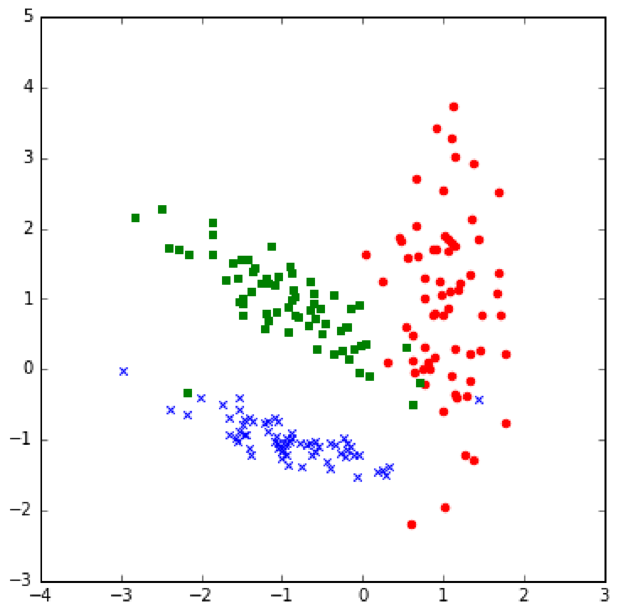
10.1 Python implementation

Loading necessary libraries:

```
In [1]: from matplotlib import cm
        import matplotlib.pyplot as plt
        import numpy as np
        import sklearn.datasets as ds
```

The softmax regression in `scikit-learn` can be trained using LogisticRegression class. We generate data with 3 classes and 2 features first:

```
In [2]: from sklearn.linear_model import LogisticRegression

In [3]: X,y = ds.make_classification(n_features=2,
                                     n_redundant=0,
                                     n_informative=2,
                                     n_clusters_per_class=1,
                                     n_classes=3,
                                     n_samples=200,
                                     random_state=3216)
        ix0 = [i for i,x in enumerate(y) if x == 0]
        ix1 = [i for i,x in enumerate(y) if x == 1]
        ix2 = [i for i,x in enumerate(y) if x == 2]
        fig = plt.figure(figsize=(6,6))
        #ax = plt.axes(xlim=(-3.1, 3.1), ylim=(-3.1, 3.1))
        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.scatter(X[ix2,0],X[ix2, 1],marker='s',color='green')
        plt.show()
```



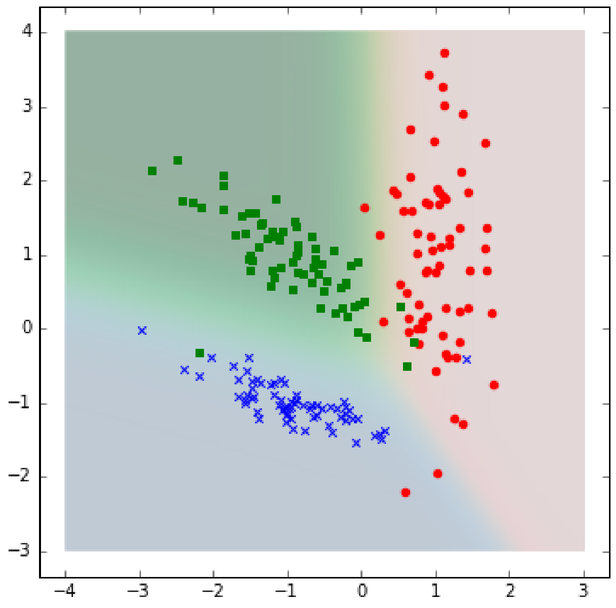
Training the softmax regression is easy as before:

```
In [4]: model = LogisticRegression(multi_class='multinomial', solver='newton-cg')
        model.fit(X,y)
        print 'Intercepts:'
        print model.intercept_
        print 'Coefficients:'
        print model.coef_
```

```
Intercepts:
[-0.10153312 -0.56865552  0.67018864]
Coefficients:
[[ 2.80913084  1.10436996]
 [-1.8280927  -2.14320504]
 [-0.98103814  1.03883507]]
```

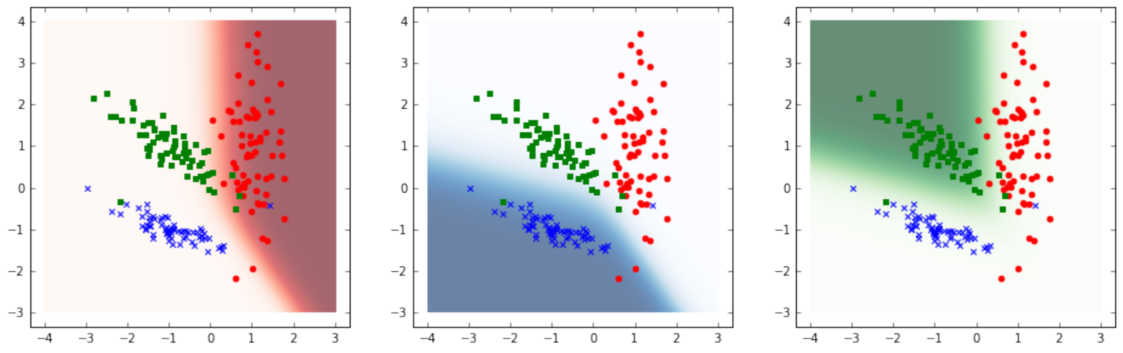
It is possible to get probabilities from the model using the method `predict_proba()`, but I have implemented the function `softmax_probs()` to evaluate probabilities with `model.intercept_` and `model.coeff_` attributes.

```
In [5]: def softmax_probs(X, model):
            Xb = np.hstack((np.ones((X.shape[0],1)), X))
            thetas = np.hstack((model.intercept_.reshape((-1,1)), model.coef_))
            probs = np.exp(np.dot(Xb, np.transpose(thetas)))
            probs_sums = probs.sum(axis=1)
            probs = probs / probs_sums[:, np.newaxis]
            return probs

        x1 = np.arange(-4.0, 3.0, 0.05)
        x2 = np.arange(-3.0, 4.0, 0.05)
        x1,x2 = np.meshgrid(x1, x2)
        y_pred = softmax_probs(np.c_[x1.ravel(), x2.ravel()], model)
        extent = -4.0, 3.0, -3.0, 4.0

        fig = plt.figure(figsize=(10,6))
        plt.imshow(y_pred[:,0].reshape(x1.shape), cmap=cm.Reds, alpha=.4,
                   interpolation='bilinear',extent = extent, origin='lower')
        plt.imshow(y_pred[:,1].reshape(x1.shape), cmap=cm.Blues, alpha=.4,
                   interpolation='bilinear',extent = extent, origin='lower')
        plt.imshow(y_pred[:,2].reshape(x1.shape), cmap=cm.Greens, alpha=.4,
                   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')
        plt.scatter(X[ix2,0],X[ix2, 1],marker='s',color='green')
        plt.show()
```

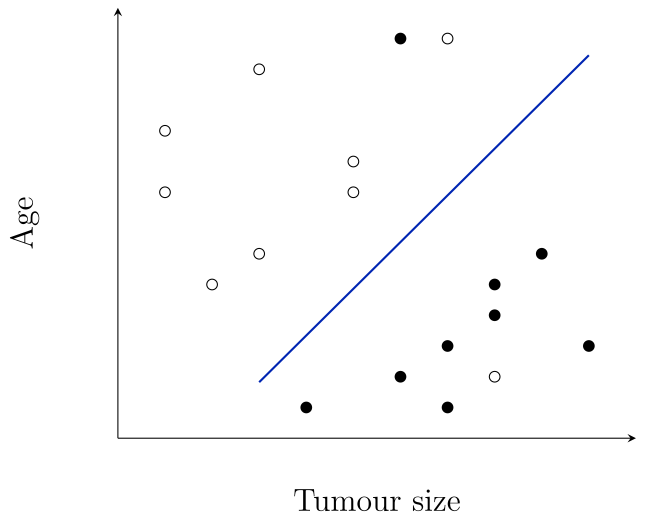


And another way to visualize it:

```
In [6]: fig = plt.figure(figsize=(16,8))
        fig.add_subplot(131)
        plt.imshow(y_pred[:,0].reshape(x1.shape), cmap=cm.Reds, alpha=.6,
                   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')
        plt.scatter(X[ix2,0],X[ix2, 1],marker='s',color='green')

        fig.add_subplot(132)
        plt.imshow(y_pred[:,1].reshape(x1.shape), cmap=cm.Blues, alpha=.6,
                   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')
        plt.scatter(X[ix2,0],X[ix2, 1],marker='s',color='green')

        fig.add_subplot(133)
        plt.imshow(y_pred[:,2].reshape(x1.shape), cmap=cm.Greens, alpha=.6,
                   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')
        plt.scatter(X[ix2,0],X[ix2, 1],marker='s',color='green')

        plt.show()
```



11 Generative learning algorithm



The logistic regression tries to find the straight line between two classes. This algorithm belongs to the class of **discriminative algorithms**. The idea of properties for discriminative algorithms is the following:

- learns $p(y \mid x)$
- or learns $h_\theta(x) \in \{0, 1\}$

Now we are going to talk about different approach. Assuming that we have two classes, we build two models: one - on positive examples only, the second - on negative examples only. When we get new sample we try to understand which model this sample match better and based on this matching we make a conclusion. This algorithm belongs to the class of **generative algorithms**. The idea of the generative algorithms is

- learn $p(x \mid y)$ and $p(y)$
- use Bayes rule to get $p(y = 1 \mid x) = \dfrac{p(x \mid y = 1)p(y)}{p(x)}$, where

$$p(x) = p(x \mid y = 0)p(y = 0) + p(x \mid y = 1)p(y = 1)$$

12 Gaussians

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

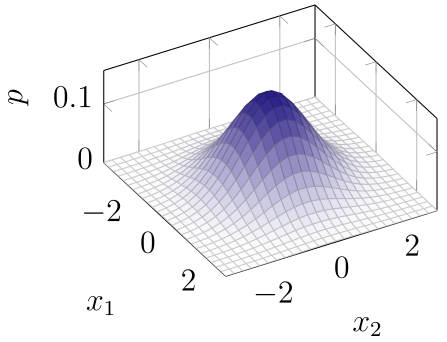
$$p(x) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\vec{\mu})^T\Sigma^{-1}(x-\vec{\mu})\right),$$

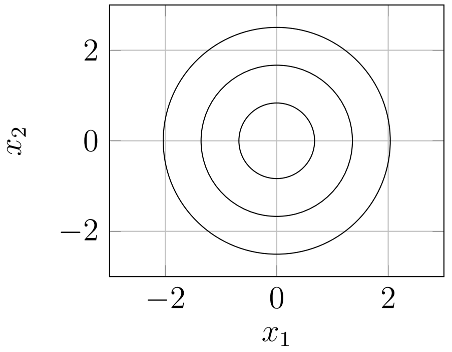
where $\vec{\mu}$ is a mean vector, and Σ is a symmetrical covariance matrix:

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

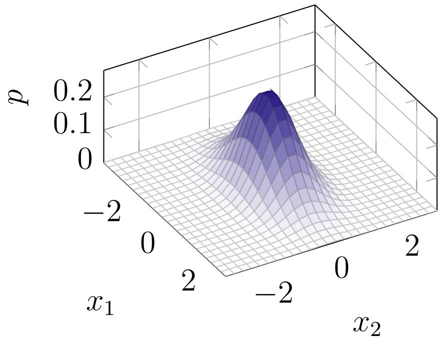
The density function $p(x)$ in case of two variables $x = (x_1, x_2)$ for multivariate gaussian distribution can be plotted as a surface. Consider the distributions with zero mean vectors and different covariance matrices:

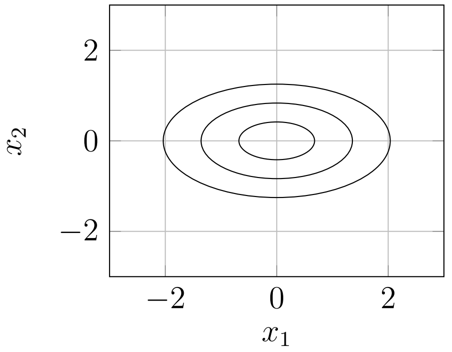
1. $\Sigma = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$



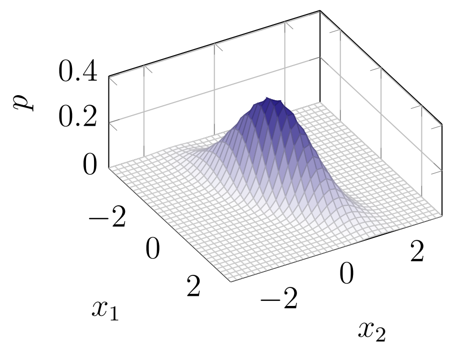


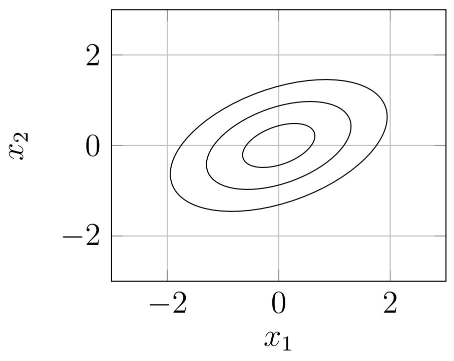
2. $\Sigma = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 0.3 \end{bmatrix}$





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





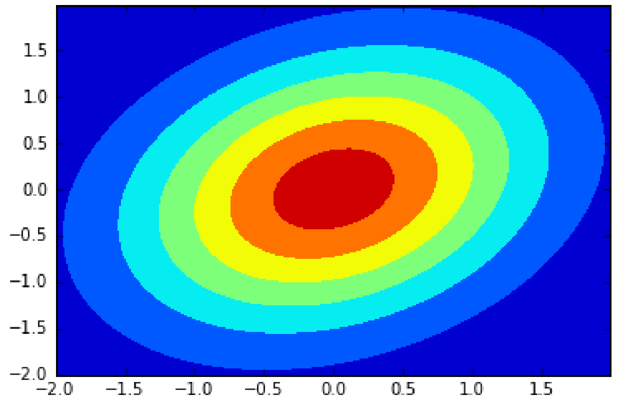
12.1 Python implementation

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()
```

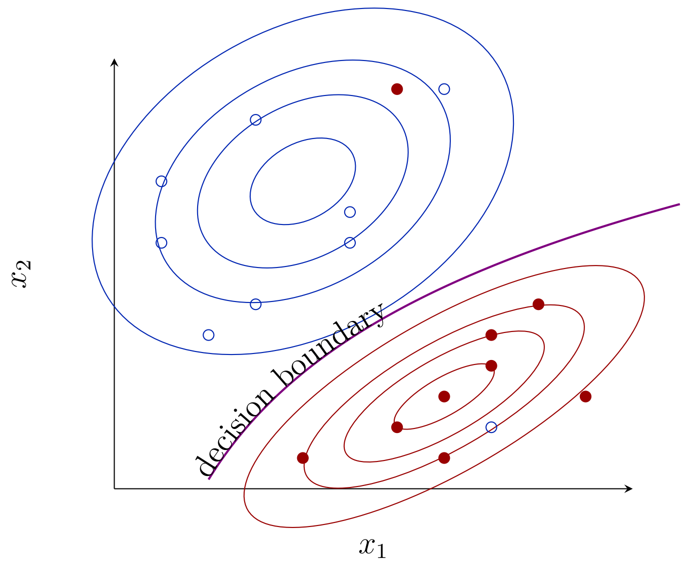


13 Gaussian discriminant analysis

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:

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

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

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

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

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

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

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:

$$\arg\max_y p(y \,|\, x) = \arg\max_y p(x \,|\, y).$$

13.1 Python implementation

Loading necessary libraries:

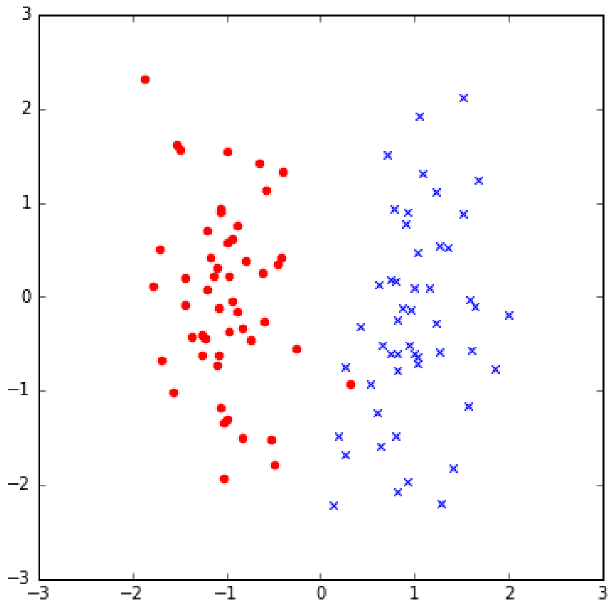
```
In [3]: from matplotlib import cm
        import matplotlib.pyplot as plt
        import numpy as np
        import sklearn.datasets as ds
```

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()
```

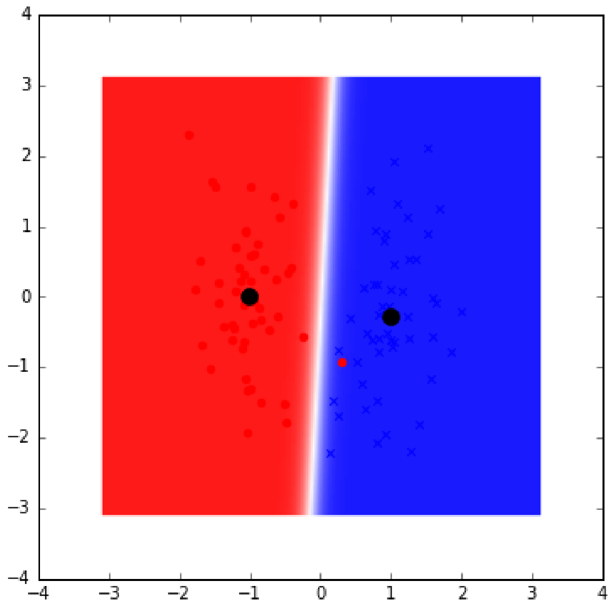


There are two implementations of Gaussian Discriminant Analysis in `scikit-learn`: Linear Discriminant Analysis (LDA) and Quadratic Discriminant Analysis (QDA). We try to use both and compare the results. First, we train a model for Linear Discriminant Analysis:

```
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()
```



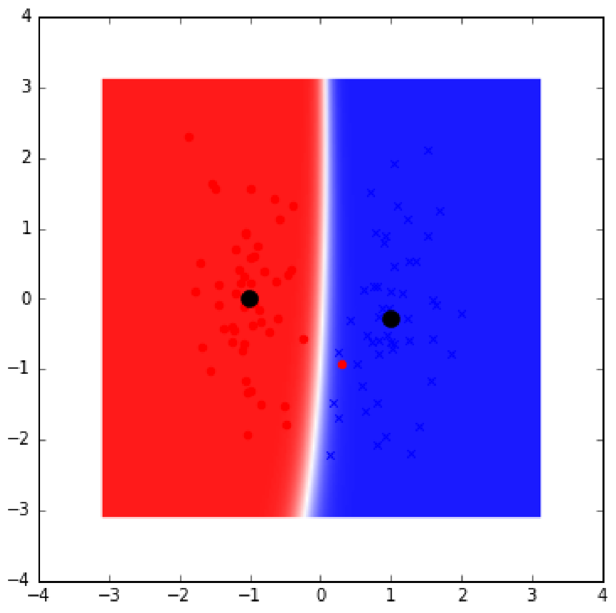
We do the same for Quadratic Discriminant Analysis:

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

and visualize the results:

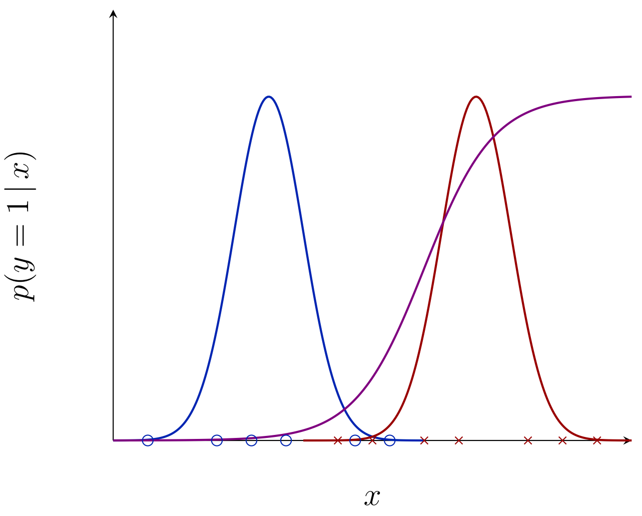
```
In [9]: x1, x2 = np.meshgrid(np.linspace(-3.1, 3.1, 200),
                             np.linspace(-3.1, 3.1, 200))
        y_pred = 1.0 - qda.predict_proba(np.c_[x1.ravel(), x2.ravel()])
        y_pred = y_pred[:, 1].reshape(x1.shape)
        extent = -3.1, 3.1, -3.1, 3.1

        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()
```



14 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.

15 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 bag 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:

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

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

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

15.1 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
```

16 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

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

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

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

17 Event models

In the previous section we assumed that all x_i are binary, $x_i \in \{0, 1\}$. To generalize the previous model we assume that $x_i \in \{1, \ldots, k\}$, then the generative learning algorithm implies that we model

$$p(x\,|\,y) = \prod_{i=1}^{n} p(x_i\,|\,y),$$

where on the right-hand side we have multinomial probabilities (rather than Bernoulli as before). In case of continuous features this model can be also implemented if you apply the procedure of discretization:

$x_1 < 1$	$1 \leqslant x_1 < 10$	$10 \leqslant x_1 < 50$	$50 \leqslant x_1 < 100$	$100 \leqslant x_1$
0	1	2	3	4

The model with more than two possible values for each feature is called **multinomial event model** (compared to the multivariate Bernoulli event model we had earlier).

We consider the spam classification problem from different point of view. One email will be described by the vector $(x_1^{(i)}, x_2^{(i)}, \ldots, x_{d_i}^{(i)})$, where d_i is a number of words in the email i and $x_j \in \{1, 2, \ldots, n\}$, where n is a size of the bag of words and could be a huge number, for example, $n = 50\,000$. In other words, we assign some number to each word and takes these numbers to the feature vector. The main difference in such formulation is that feature vectors have different lengths for different training examples (because email lengths are different).

The joint distribution for x and y can be written as

$$p(x, y) = \left(\prod_{j=1}^{d} p(x_j\,|\,y)\right) p(y),$$

where x_j is j-th word in the email, d is a number of words in the email. We identify the parameters for this model and apply the maximum likelihood estimation to find the values for them. Parameters are defined by the equations:

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

For example, the parameter $\varphi_{k\,|\,y=1}$ defines the probability of having word k in the spam email on any position j and so on.

The likelihood is defined as

$$\begin{aligned}
l(\varphi_{k\,|\,y=1}, \varphi_{k\,|\,y=0}, \varphi_y) &= \ln \prod_{i=1}^{m} p(x^{(i)}, y^{(i)}, \varphi_{k\,|\,y=1}, \varphi_{k\,|\,y=0}, \varphi_y) = \\
&= \ln \prod_{i=1}^{m} p(x^{(i)}\,|\,y^{(i)}, \varphi_{k\,|\,y=1}, \varphi_{k\,|\,y=0}) \cdot p(y^{(i)}; \varphi_y)
\end{aligned}$$

and the maximization gives the following list of parameters

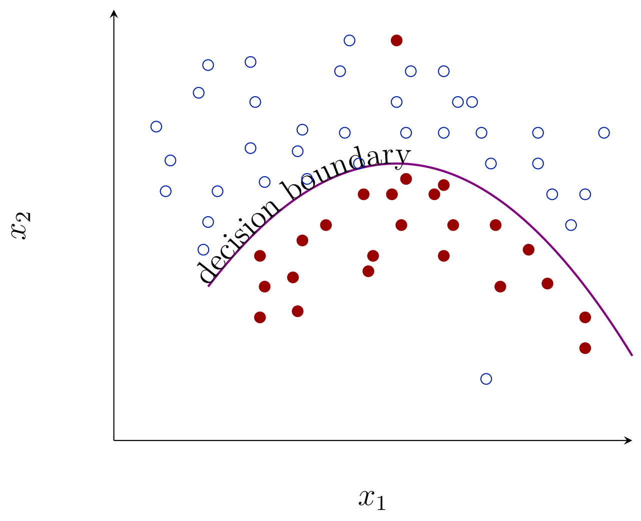
$$\varphi_{k\,|\,y=1} = \frac{\sum\limits_{i=1}^{m} \left(\mathbb{1}\{y^{(i)} = 1\} \sum\limits_{j=1}^{d_i} \mathbb{1}\{x_j^{(i)} = k\} \right) + 1}{\sum\limits_{i=1}^{m} (\mathbb{1}\{y^{(i)} = 1\} \cdot d_i) + n},$$

$$\varphi_{k\,|\,y=0} = \frac{\sum\limits_{i=1}^{m} \left(\mathbb{1}\{y^{(i)} = 0\} \sum\limits_{j=1}^{d_i} \mathbb{1}\{x_j^{(i)} = k\} \right) + 1}{\sum\limits_{i=1}^{m} (\mathbb{1}\{y^{(i)} = 0\} \cdot d_i) + n}.$$

The meaning of the first parameter is the number of appearances of the word k in all spam emails divided by the total number of words in all spam emails. Notice that we use the general Laplace smoothing described in the previous lecture.

18 Neural networks

The following example shows that in some cases we need non linear decision boundary which means that we should build nonlinear classifiers.



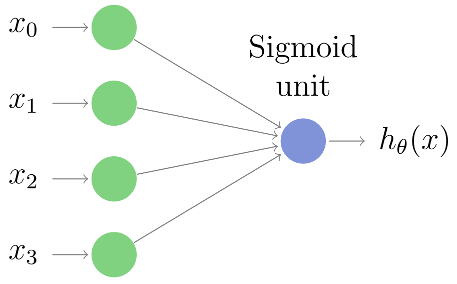
One example of nonlinear classifiers we took before was the logistic regression with hypothesis $h_\theta(x) = \dfrac{1}{1+e^{-\theta^T x}}$. Another example is generative learning algorithm with the assumptions

$$x \,|\, y = 1 \sim \text{ExpFamily}(\eta_1),$$
$$x \,|\, y = 0 \sim \text{ExpFamily}(\eta_0).$$

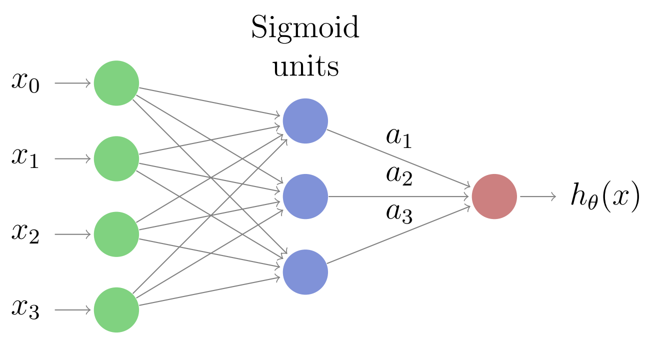
One way to build nonlinear classifier is to fit the hypothesis with higher degree function instead of $\theta^T x$. For example, we can consider the hypothesis

$$h_\theta(x) = \frac{1}{1+e^{-\theta_0-\theta_1 x_1-\theta_2 x_2-\theta_3 x_1 x_2-\theta_4 x_1^2 x_2-\dots}}.$$

Another way is to visualize the logistic regression algorithm in some fancy way and try to generalize the idea:



For this diagram we are going to use the biological terminology: the circles are **neurons** or **units**, the set of green circles (representing the features) is an **input layer**, the blues circle with sigmoid function is an **output layer** with **sigmoid activation function**. The natural way to generalize this diagram is to add more neurons:



On this diagram we have 3 layers: input layer (green), hidden layer (blue) with sigmoid activation functions and output layer (red) with sigmoid activation function. Let $g(z) = \dfrac{1}{1+e^{-z}}$ be a sigmoid function then

$$a_1 = g(x^T\theta_1^{(1)}),\ a_2 = g(x^T\theta_2^{(1)}),\ a_3 = g(x^T\theta_3^{(1)}),\ h_\theta(x) = g(a^T\theta^{(2)}),$$

where

$$a = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}.$$

The number of parameters (**weights** in the neural network terminology) for this network equals to the number of edges for this diagram: $12+3=15$. In other words, we should learn 4 vectors of parameters $\theta_1^{(1)}$, $\theta_2^{(1)}$, $\theta_3^{(1)}$ and $\theta^{(2)}$. Obviously we could add more hidden layers and our hypothesis becomes more complicated.

To fit the parameters we recall that the log-likelihood for logistic regression has a form

$$l(\theta) = \sum_{i=1}^{m} y^{(i)} \ln h_\theta(x^{(i)}) + (1-y^{(i)}) \ln(1-h_\theta(x^{(i)}))$$

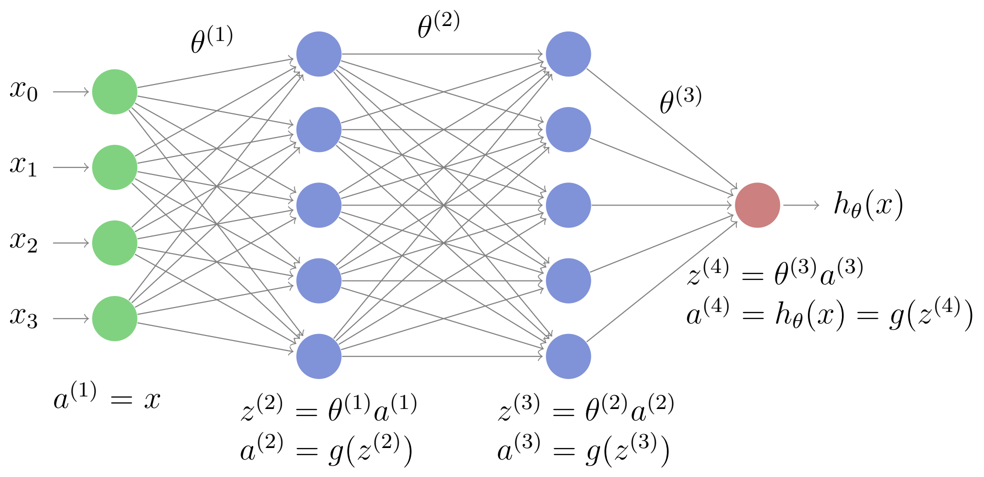
and we obtain the neural network as a generalization of logistic regression. This is the reason that for the neural network we are going to use the following loss function

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \ln h_\theta(x^{(i)}) + (1-y^{(i)}) \ln(1-h_\theta(x^{(i)})),$$

and find the minimum of this function using the gradient descent. For the neural networks the gradient descent algorithm has a special name: **backpropagation**. The procedure of predictions in the neural network is called **forward propagation**.

To have the convenient notations we denote by $\theta^{(k)}$ the matrix of parameters from the layer k to the layer $k+1$. The element $\theta_{ij}^{(k)}$ defines the weight from the i-th neuron of layer $k+1$ to the j-th neuron of layer k. Let $a_i^{(k)}$ be the activation function of the neuron i in layer k. To use the gradient descent algorithm we should compute all derivatives $\dfrac{\partial J(\theta)}{\partial \theta_{ij}^{(k)}}$.

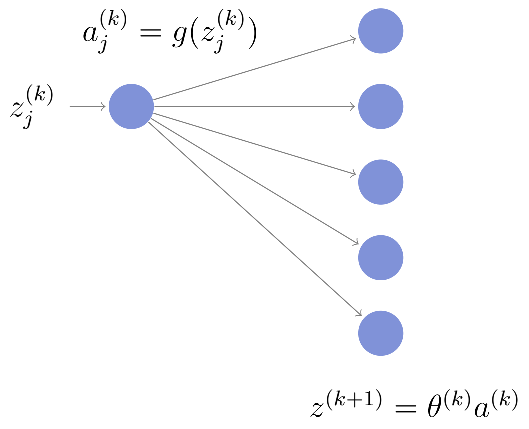
Given one training example (x, y) consider one forward propagation step for the following network configuration:



The backpropagation procedure includes the calculation of partial derivatives

$$\delta_j^{(k)} = \frac{\partial J(\theta)}{\partial z_j^{(k)}}.$$

Consider the unit j in the layer k and the next layer $k+1$:



Assuming that we know derivatives $\delta_i^{(k+1)} = \dfrac{\partial J(\theta)}{\partial z_i^{(k+1)}}$ and

$$z_i^{(k+1)} = \ldots + \theta_{ij}^{(k)} a_j^{(k)} + \ldots = \ldots + \theta_{ij}^{(k)} g(z_j^{(k)}) + \ldots, \tag{18.1}$$

where terms from other units of layer k are denoted by dots. Then the derivative $\delta_j^{(k)}$ can be calculated using Chain Rule:

$$\delta_j^{(k)} = \frac{\partial J(\theta)}{\partial z_j^{(k)}} = \sum_i \frac{\partial J(\theta)}{\partial z_i^{(k+1)}} \cdot \frac{\partial z_i^{(k+1)}}{\partial z_j^{(k)}} =$$
$$= \sum_i \frac{\partial J(\theta)}{\partial z_i^{(k+1)}} \cdot \theta_{ij}^{(k)} \cdot g'(z_j^{(k)}) = \sum_i \delta_i^{(k+1)} \theta_{ij}^{(k)} \cdot g'(z_j^{(k)}).$$

More general, for all units of the layer k the formula can be written as

$$\delta^{(k)} = (\theta^{(k)})^T \delta^{(k+1)} g'(z^{(k)}) = (\theta^{(k)})^T \delta^{(k+1)} a^{(k)} (1 - a^{(k)})$$

(remember that $g(z)$ is a sigmoid function and its derivatives can be calculated in a very simple way).

When all "deltas" are calculated it is easy to calculate our target derivatives (again using Chain Rule and the equation (18.1)):

$$\frac{\partial J(\theta)}{\partial \theta_{ij}^{(k)}} = \frac{\partial J(\theta)}{\partial z_i^{(k+1)}} \cdot \frac{\partial z_i^{(k+1)}}{\partial \theta_{ij}^{(k)}} = \delta_i^{(k+1)} \cdot g(z_j^{(k)}) = \delta_i^{(k+1)} \cdot a_j^{(k)}.$$

Now we can summarize the backpropagation algorithm (we are using stochastic gradient descent to update weights):

Algorithm 3 Backpropagation algorithm for neural networks

```
 1: Set number of hidden layers $K$ and number of neurons $m_k$ in the layer $k$
 2: Initialize matrices $\theta^{(k)}, k = 0, \dots, K$ of random weights
 3: repeat
 4:     for all training examples $(x, y)$ do
 5:         Set $a^{(0)} = x$
 6:         Run the forward step to compute $a^{(k)}, k = 1, \dots, K+1$
 7:         Set $\delta^{(K+1)} = a^{(K+1)} - y$ (here $a^{(K+1)}$ is a prediction)
 8:         for $k = K$ downto 1 do
 9:             Set $\delta^{(k)} = (\theta^{(k)})^T \delta^{(k+1)} a^{(k)} (1 - a^{(k)})$
10:             for all $i = 1, \dots, m_{k+1}$ and $j = 1, \dots, m_k$ do
11:                 Perform the stochastic gradient descent step
                        $\theta_{ij}^{(k)} := \theta_{ij}^{(k)} - \alpha \cdot \delta_i^{(k+1)} \cdot a_j^{(k)}$
12: until convergence
```

18.1 Python implementation

Import necessary libraries:

```
In [1]: import numpy as np
        import random
        import math
        import sklearn.datasets as ds
        from matplotlib import cm
        import matplotlib.pyplot as plt

        from nnvisual import *
```

We have implemented the following functions: `sigmoid()` is an activation function, `forward()` implements forward propagation step, `backward()` implements backpropagation step. The function `plot_decision_boundary()` draws the decision boundary for two classes. The function `train_neural_network()` runs the main loop of forward and backward propagations, and the function `predict_neural_network()` runs the forward loop to obtain the predictions. Notice that we have added the regularization term, which means that instead of the loss function (we will talk about the regularization techniques later in our lectures):

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\ln h_\theta(x^{(i)}) + (1-y^{(i)})\ln(1-h_\theta(x^{(i)}))\right)$$

we minimize the function

$$J(\theta) = -\frac{1}{m}\left(\sum_{i=1}^{m} y^{(i)}\ln h_\theta(x^{(i)}) + (1-y^{(i)})\ln(1-h_\theta(x^{(i)}))\right) + \frac{\beta}{2}\sum_{k=0}^{K}\sum_{i=1}^{m_{k+1}}\sum_{j=1}^{m_k}\left(\theta_{ij}^{(k)}\right)^2.$$

```
In [2]: def sigmoid(z):
            return 1.0/(1+np.exp(-z))

        def forward(X, thetas):
            Xb = np.hstack((np.ones((X.shape[0],1)), X))
            a = np.transpose(Xb)
            alist = [a]
            for i in range(L):
                a = sigmoid(np.dot(thetas[i], a))
                if i < L-1:
                    a = np.vstack((np.ones((1, a.shape[1])), a))
                alist.append(a)
            return alist

        def backward(a, y, thetas, alpha=0.1, beta=0.01, batch_size=1):
            start = 0
            end = start + batch_size
            while True:
                if end > y.shape[1]:
                    end = y.shape[1]
                # a[L] are predictions
                deltas = a[L][:,start:end] - y[:,start:end]
                for i in range(L-1, -1, -1):
                    deltas_prev = np.multiply(
                        np.multiply(
                            np.dot(thetas[i].T, deltas),
                            a[i][:,start:end]),
                        1-a[i][:,start:end])
                    step = np.sum(np.array([
                                  np.outer(deltas[:,k], a[i][:,start:end][:,k])
                                  for k in range(deltas.shape[1])
                                  ]), axis=0)
                    thetas[i][:,1:] = thetas[i][:,1:] - alpha*step[:,1:]
                                      - beta*thetas[i][:,1:]
                    thetas[i][:,0] = thetas[i][:,0] - alpha*step[:,0]
                    deltas = deltas_prev[1:, :]
                start = end
                end = start + batch_size
                if start == y.shape[1]:
                    break
            return thetas



        def plot_decision_boundary(X, y, thetas, fig, pos,
                                   xmin=-0.1, xmax=1.1, ymin=-0.1, ymax=1.1):
            x1, x2 = np.meshgrid(np.linspace(xmin, xmax, 200),
                                 np.linspace(ymin, ymax, 200))
            ypred = forward(np.c_[x1.ravel(), x2.ravel()], thetas)[L][0]
            ypred = ypred.reshape(x1.shape)
            extent = xmin, xmax, ymin, ymax

            fig.add_subplot(1,4,pos)
            plt.imshow(ypred, cmap=cm.bwr, alpha=.9,
                       interpolation='bilinear', extent = extent,
                       origin='lower')
            plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=cm.bwr)

        def train_neural_network(X, y, alpha, beta, batch_size, seed, niter):
            ### initialize weights
            thetas = []
            np.random.seed(seed)
            for i in range(L):
                thetas.append(20*(np.random.rand(nn[i+1], nn[i]+1)-0.5))
            print 'Thetas:'
            print thetas

            ### the main loop of training
            fig = plt.figure(figsize=(16,32))
            count_plot = 0
            for it in range(niter):
                a = forward(X, thetas)
                thetas = backward(a, y, thetas, alpha=alpha, beta=beta,
                                    batch_size=batch_size)
                if it % round(niter/4) == 0:
                    count_plot = count_plot + 1
                    plot_decision_boundary(X, y, thetas, fig, count_plot,
                                           xmin=np.min(X[:,0])-0.1,
                                           xmax=np.max(X[:,0])+0.1,
                                           ymin=np.min(X[:,1])-0.1,
                                           ymax=np.max(X[:,1])+0.1)
        plt.show()
        return thetas

    def predict_neural_network(X, thetas):
        return forward(X, thetas)[L][0]
```

We use our neural network to solve the famous XOR problem.

$\mathbf{x_1}$	$\mathbf{x_2}$	XOR $(\mathbf{x_1}, \mathbf{x_2})$
1	1	1
1	0	0
0	1	0
0	0	1

The XOR function detects if x_1 and x_2 are the same. The known fact is that it is impossible to solve this problem using logistic regression and we need at least one hidden layer to get correct solution. First, we define our dataset: 4 training examples with XOR as a target variable y.

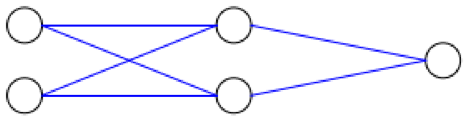
```
In [3]: X_train = np.array([[1, 1], [1, 0], [0, 1], [0, 0]]).astype(np.float64)
        y_train = np.array([1, 0, 0, 1]).reshape((1, -1))
```

Define the neural network configuration (the last number is the dimension of y):

```
In [4]: nn = np.array([X_train.shape[1], 2, y_train.shape[0]]).astype(int)
        L = len(nn)-1 # L = number of hidden layers + 1
```

We are using nnvisual.py code to show how my neural network looks like. This code is modified code from here: https://github.com/miloharper/visualise-neural-network (thanks to miloharper for it). I have changed the orientation of the neural network in the original file.

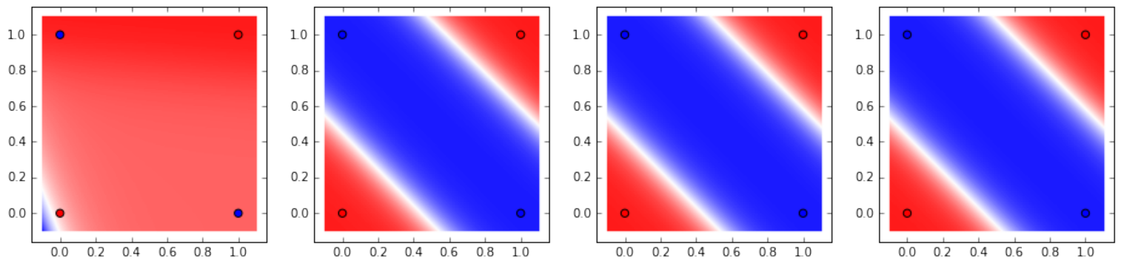
```
In [5]: network = NeuralNetwork()
        for l in nn:
            network.add_layer(l)
        network.draw()
```



Now we check different initializations and different parameters:

```
In [6]: thetas = train_neural_network(X_train, y_train,
                                      alpha=0.1, beta=0.0001,
                                      batch_size=2, seed=131214, niter=10000)
        print "Predictions:"
        predict_neural_network(X_train, thetas)
```

```
Thetas:
[array([[1.67236953, 7.19523078, 2.85081229],
       [-8.56741528, 0.63708182, 8.25788355]]),
 array([[5.62926985, 5.52618743, 5.01061296]])]
```



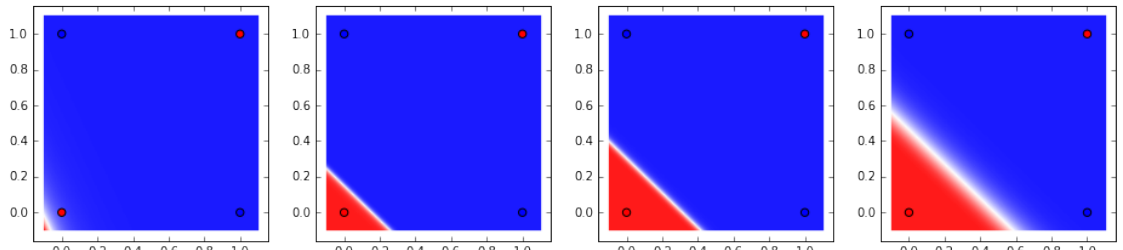
```
Predictions:
```

```
Out[6]: array([0.97663021, 0.02344909, 0.0234481, 0.97639398])
```

Here the influence of the learning rate alpha. We can see that our neural network cannot find the correct minimum with big learning rate. Notice that in this case all predictions are very close to each other.

```
In [7]: thetas = train_neural_network(X_train, y_train,
                                      alpha=10.0, beta=0.0001,
                                      batch_size=2, seed=131214, niter=10000)
        print "Predictions:"
        predict_neural_network(X_train, thetas)
```

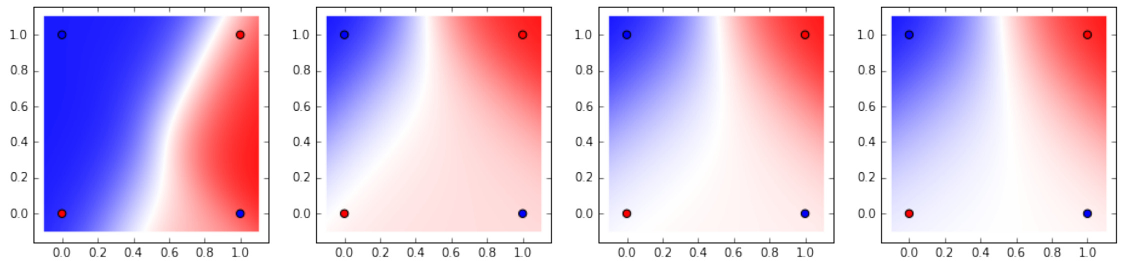
```
Thetas:
[array([[1.67236953, 7.19523078, 2.85081229],
       [-8.56741528, 0.63708182, 8.25788355]]),
 array([[5.62926985, 5.52618743, 5.01061296]])]
```



```
Predictions:

Out[7]: array([0.999996, 0.999996, 0.999996, 1.])
```

The initialization weights are also very important. The different seed can give different results:

```
In [8]: thetas = train_neural_network(X_train, y_train,
                                      alpha=0.01, beta=0.0001,
                                      batch_size=2, seed=114, niter=10000)
        print "Predictions:"
        predict_neural_network(X_train, thetas)

Thetas:
[array([[-6.96736066, 7.37490561, 8.51767526],
       [-0.23173645, 5.30985877, -2.76355142]]),
 array([[-7.335906, 0.43388152, 4.42122827]])]
```

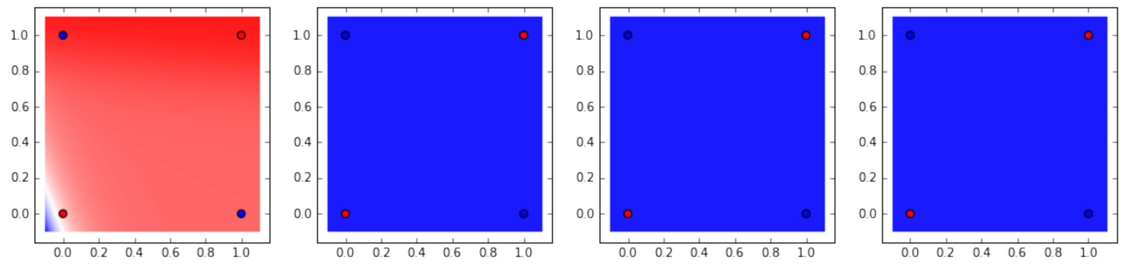


```
Predictions:

Out[8]: array([0.8485083, 0.52529168, 0.15702184, 0.46377799])
```

The regularization parameter beta is also important. For wrong values of beta the neural network can fail as well:

```
In [9]: thetas = train_neural_network(X_train, y_train,
                                      alpha=0.1, beta=0.01,
                                      batch_size=2, seed=131214, niter=10000)
        print "Predictions:"
        predict_neural_network(X_train, thetas)
```

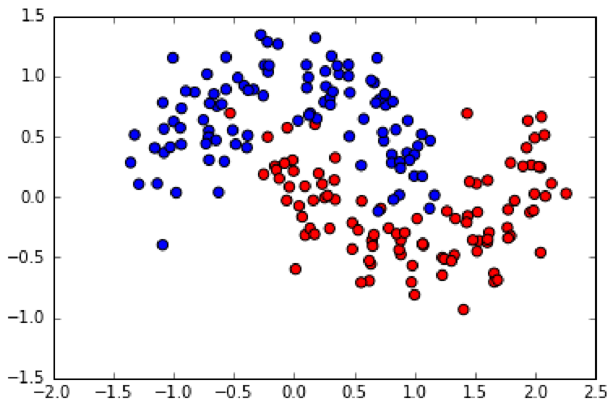
```
Thetas:
[array([[1.67236953, 7.19523078, 2.85081229],
        [-8.56741528, 0.63708182, 8.25788355]]),
 array([[5.62926985, 5.52618743, 5.01061296]])]
```



```
Predictions:

Out[9]: array([0.5, 0.5, 0.5, 0.5])
```

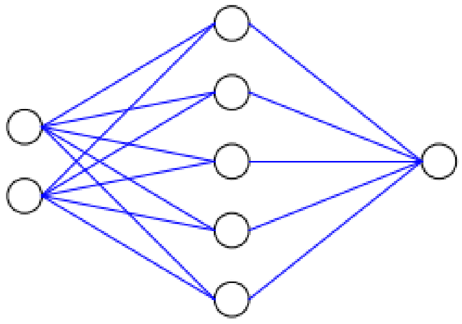
As an another example we consider the `make_moons` dataset from `scikit-learn` package:

```
In [10]: np.random.seed(0)
         X_train, y_train = ds.make_moons(200, noise=0.20)
         y_train = y_train.reshape((1, -1))
         plt.scatter(X_train[:,0], X_train[:,1], s=40, c=y_train, cmap=cm.bwr)
         plt.show()
```

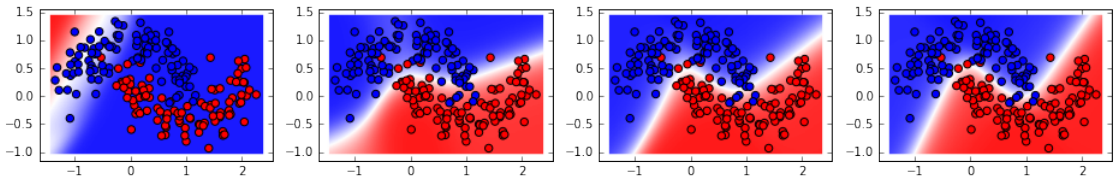


Initialize our neural network and visualize it:

```
In [11]: nn = np.array([X_train.shape[1], 5, y_train.shape[0]]).astype(int)
         L = len(nn)-1 # L = number of hidden layers + 1

In [12]: network = NeuralNetwork()
         for l in nn:
             network.add_layer(l)
         network.draw()
```

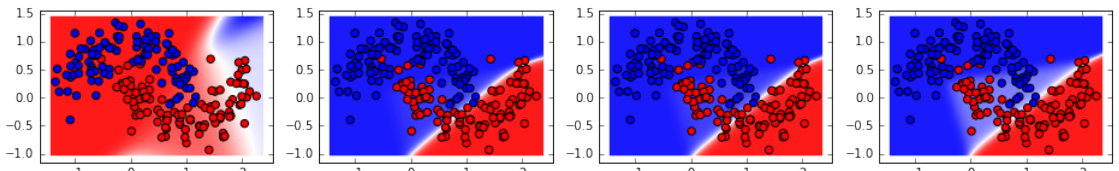


```
In [13]: thetas = train_neural_network(X_train, y_train,
                                       alpha=0.01, beta=0.0001,
                                       batch_size=10, seed=1104, niter=1000)

Thetas:
[array([[1.22741958, -1.53385074, 4.2008582 ],
       [-3.76905412, 6.57801497, 1.09413825],
       [ 0.05969385, -3.51291599, 6.83412737],
       [-8.07433437, -3.75662311, 2.07674147],
       [-1.92150795, -4.73883283, 2.7290175]]),
 array([[-9.78380152, 1.20918211, -8.0590909, -2.00016833, 9.60846917,
         9.05818477]])]
```



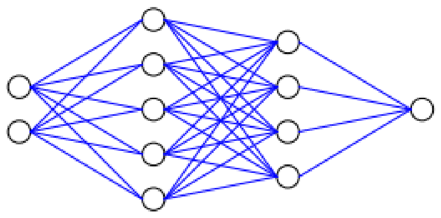
Wrong learning rate:

```
In [14]: thetas = train_neural_network(X_train, y_train,
                                       alpha=0.1, beta=0.0001,
                                       batch_size=10, seed=1104, niter=1000)

Thetas:
[array([[1.22741958, -1.53385074, 4.2008582 ],
       [-3.76905412, 6.57801497, 1.09413825],
       [ 0.05969385, -3.51291599, 6.83412737],
       [-8.07433437, -3.75662311, 2.07674147],
       [-1.92150795, -4.73883283, 2.7290175]]),
 array([[-9.78380152, 1.20918211, -8.0590909, -2.00016833, 9.60846917,
         9.05818477]])]
```

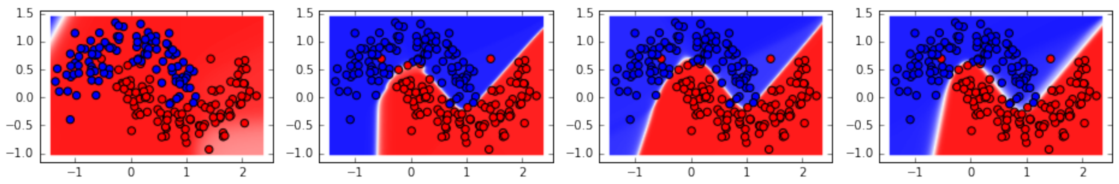


Add more layers to the network:

```
In [15]: nn = np.array([X_train.shape[1], 5, 4, y_train.shape[0]]).astype(int)
         L = len(nn)-1 # L = number of hidden layers + 1
         network = NeuralNetwork()
         for l in nn:
             network.add_layer(l)
         network.draw()
```



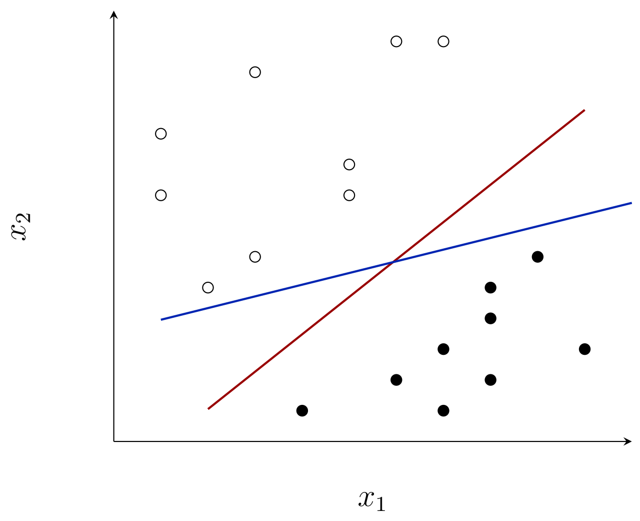
```
In [16]: thetas = train_neural_network(X_train, y_train,
                                      alpha=0.1, beta=0.0001,
                                      batch_size=10, seed=1104, niter=1000)

Thetas:
[array([[1.2274, -1.5338, 4.2008],
       [-3.7690, 6.5780, 1.0941],
       [ 0.0596, -3.5129,  6.8341],
       [-8.0743, -3.7566, 2.0767],
       [-1.9215, -4.7388, 2.7290]]),
 array([[-9.7838, 1.2091, -8.0590, -2.0001, 9.6084, 9.0581],
        [2.7658, 6.1862, 5.8740, 3.1083, 4.1896, -9.6227],
        [7.9800, -5.7013, -2.8169, 1.3702, 1.6314, 2.6471],
        [0.9133, 5.7996, -6.8791, 3.4554, -4.2520, 8.1307]]),
 array([[1.5958, -6.1230, -9.7194, -6.3800, 1.2106]])]
```



19 Support vector machines: intuition

In this section we develop another nonlinear algorithm. There are two intuitions behind it:

- Logistic regression computes $\theta^T x$ and predict 1 if $\theta^T x > 0$, 0 - otherwise. If $\theta^T x >> 0$, then the algorithm is very "confident" that $y = 1$. If $\theta^T x << 0$, the algorithm is very "confident" that $y = 0$. Our aim is to obtain the algorithm that gives $\theta^T x^{(i)} >> 0$ for any i such that $y^{(i)} = 1$, and $\theta^T x^{(i)} << 0$ for any i such that $y^{(i)} = 0$.
- If we assume that classes are linearly separable, then we prefer the red line as our decision boundary (the intuition is that blue line is not good decision boundary):



For this algorithm we should use slightly different notations. First of all, we assume that $y \in \{-1, 1\}$ which means that output values could be 1 or -1. Second of all, we are using the hypothesis

$$h_{w,b}(x) = \text{sign}(w^T x + b),$$

where

$$\text{sign}(z) = \begin{cases} 1 & \text{if } z \geqslant 0, \\ -1 & \text{otherwise} \end{cases}$$

We removed the convention $x_0 = 1$ and θ_0 is replaced by b, θ is replaced by vector w.

Definition. Functional margin of a hyperplane $w^T x + b = 0$ with respect to $(x^{(i)}, y^{(i)})$ is:

$$\hat{\gamma}^{(i)} = y^{(i)}(w^T x^{(i)} + b).$$

Notice that if $y^{(i)} = 1$ then the algorithm should give $w^T x^{(i)} + b >> 0$, and if $y^{(i)} = -1$ the algorithm should give $w^T x^{(i)} + b << 0$. In both cases $\hat{\gamma}^{(i)} = y^{(i)}(w^T x^{(i)} + b) > 0$. Our aim is to build the algorithm that gives the functional margin positive for all training examples.

Definition. Minimal functional margin is

$$\hat{\gamma} = \min_i \hat{\gamma}^{(i)}.$$

Based on our intuition we should claim from the algorithm that the worst functional margin should be large.

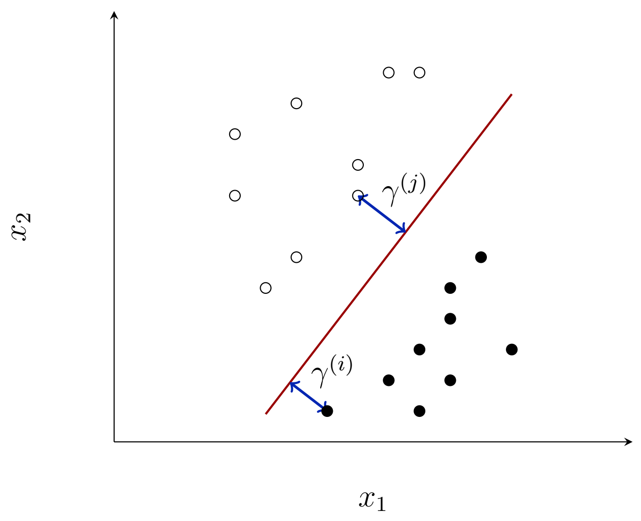
Definition. A **geometric margin** $\gamma^{(i)}$ is the distance between training example $x^{(i)}$ to the decision boundary $w^T x + b = 0$:

$$\gamma^{(i)} = y^{(i)} \left[\left(\frac{w}{||w||} \right)^T x^{(i)} + \frac{b}{||w||} \right].$$

Notice that the formula for the distance between point and hyperplane from Calculus is:

$$d = \left(\frac{w}{||w||} \right)^T x^{(i)} + \frac{b}{||w||},$$

and d has different signs for points on different sides of the plane. We multiply this distance by $y^{(i)}$ to have positive value for all training examples (remember that $y^{(i)} \in \{-1, 1\}$).



Definition. Minimal geometric margin is

$$\gamma = \min_i \gamma^{(i)}.$$

It is easy to see that functional and geometric margins are connected by $\gamma^{(i)} = \dfrac{\hat{\gamma}^{(i)}}{||w||}$. If $||w|| = 1$, then $\hat{\gamma}^{(i)} = \gamma^{(i)}$. The disadvantage of the functional margin is that it is not normalized: for example, if we double w and b, then we double functional margin, but the hyperplane $w^T x + b = 0$ does not change. To simplify our following calculations we can assume that $||w|| = 1$ or $|w_1| = 1$.

We can formulate two equivalent optimization problems (**optimal margin classifier**):

1. $\max\limits_{\gamma, w, b} \gamma$

$$\text{subject to } y^{(i)}(w^T x^{(i)} + b) \geqslant \gamma,\ i = 1 \ldots m,$$
$$||w|| = 1.$$

2. $\max\limits_{\hat{\gamma},w,b} \dfrac{\hat{\gamma}}{||w||}$

$$\text{subject to } y^{(i)}(w^T x^{(i)} + b) \geqslant \hat{\gamma},\ i = 1 \dots m.$$

Notice that for the first formulation the functional and geometric margins are the same. Also this is an example of non-convex optimization.

Consider the second optimization problem, as we mentioned before functional margin can be increasing, but the decision boundary stays the same. To avoid this situation we impose additional constraint on $\hat{\gamma}$:

$$\hat{\gamma} = 1 \text{ (scaling constraint).}$$

With this constraint the second optimization problem transforms to the **Support Vector Machine (SVM) classifier** problem

$$\boxed{\begin{array}{c} \min\limits_{w,b} \dfrac{||w||^2}{2} \text{ (the same as } \max\limits_{w,b} \dfrac{1}{||w||}) \\ \text{subject to } y^{(i)}(w^T x^{(i)} + b) \geqslant 1,\ i = 1 \dots m. \end{array}}$$

This is the quadratic programming problem, because our objective function is a quadratic function and all our constraints are linear. Recall that after we solve this optimization problem the prediction for the test point x is calculated as

$$\hat{y} = \text{sign}(w^T x + b).$$

Example. For the following dataset

x_1	x_2	y
-1	-2	1
1	-1	1
0	2	-1
1	3	-1

the SVM classifier optimization problem is formulated as:

$$\min_{w,b} \frac{1}{2}(w_1^2 + w_2^2)$$

subject to

$$\begin{aligned} &-w_1 - 2w_2 + b \geqslant 1, \\ &w_1 - w_2 + b \geqslant 1, \\ &-2w_2 - b \geqslant 1, \\ &-w_1 - 3w_2 - b \geqslant 1. \end{aligned}$$

20 Primal/dual optimization problem

To introduce the Support Vector Machine algorithm for the problems with nonlinearly separable classes we should recall the notion of primal and dual optimization problems. The method of Lagrange multipliers in the Multidimensional Calculus helps to solve the problem of optimization with additional constraints:

$$\min_w f(w)$$

subject to $h_i(w) = 0, i = 1 \dots l$, or $h(w) = \begin{bmatrix} h_1(w) \\ h_2(w) \\ \vdots \\ h_l(w) \end{bmatrix} = \vec{0}$.

The Lagrangian is defined as

$$L(w, \beta) = f(w) + \sum_i \beta_i h_i(w),$$

where β are Lagrange multipliers. The solution of the original optimization problem can be found by solving the system of equations

$$\frac{\partial L}{\partial w} = 0, \ \frac{\partial L}{\partial \beta_i} = 0$$

with respect to w and β.

The primal problem by tradition is formulated in more general form:

$$\min_w f(w)$$

subject to

$$\begin{aligned} g_i &\leqslant 0, \ i = 1, \dots, k, \\ h_i(w) &= 0, \ i = 1, \dots, l, \end{aligned}$$

or with vector notations:

$$\begin{aligned} g(w) &\leqslant \vec{0}, \\ h(w) &= \vec{0} \end{aligned}$$

For this problem the Lagrangian is defined by

$$L(w, \alpha, \beta) = f(w) + \sum_{i=1}^{k} \alpha_i g_i(w) + \sum_{i=1}^{l} \beta_i h_i(w)$$

and by definition

$$\theta_P(w) = \max_{\alpha, \beta, \alpha_i \geqslant 0} L(w, \alpha, \beta) = \begin{cases} f(w), \ \text{if conditions for } g, h \text{ satisfies} \\ \infty, \ \text{otherwise.} \end{cases}$$

Notice that if $g_i(w) > 0$, then $\theta_P(w) = \infty$; if $h_i(w) \neq 0$, then $\theta_P(w) = \infty$; otherwise, $\theta_P(w) = f(w)$.

With this definition the original problem is transformed to the **primal problem**:

$$p^* = \min_w \theta_P(w) = \min_w \max_{\alpha, \beta, \alpha_i \geqslant 0} L(w, \alpha, \beta).$$

The natural way to modify this problem is to switch max and min and formulate the **dual problem**:

$$d^* = \max_{\alpha \geqslant 0, \beta} \theta_D(\alpha, \beta) = \max_{\alpha \geqslant 0, \beta} \min_w L(w, \alpha, \beta),$$

where by definition

$$\theta_D(\alpha, \beta) = \min_w L(w, \alpha, \beta).$$

It is easy to show that $d^* < p^*$, because $\max\min \leqslant \min\max$.

Example.

$$\max_{y \in \{0,1\}} \min_{x \in \{0,1\}} \mathbb{1}\{x = y\} \leqslant \min_{x \in \{0,1\}} \max_{y \in \{0,1\}} \mathbb{1}\{x = y\}$$

Notice that $\min\limits_{x \in \{0,1\}} \mathbb{1}\{x = y\} = 0$ and $\max\limits_{y \in \{0,1\}} \mathbb{1}\{x = y\} = 1$.

The important theorem from optimization theory tells that under certain conditions: $d^* = p^*$ and we can solve dual problem instead of primal problem.

Theorem. Let

1) f is convex (if function f convex and hessian H exists then $H \geqslant 0$);

2) h_i is affine ($h_i(w) = a_i^T w + b_i$);

3) constraints g_i are strictly feasible (there exist w such that for any i $g_i(w) < 0$).

Then

1) there exists w^*, α^* and β^* such that w^* solves primal problem and α^*, β^* solve the dual problem and $p^* = d^* = L(w^*, \alpha^*, \beta^*)$;

2) $\dfrac{\partial L}{\partial w}(w^*, \alpha^*, \beta^*) = 0$, $\dfrac{\partial L}{\partial \beta}(w^*, \alpha^*, \beta^*) = 0$;

3) $\alpha_i^* g_i(w^*) = 0$ (Karush-Kuhn-Tucker (KKT) complementarity condition).

Moreover, by definition $\alpha_i^* \geqslant 0$ and from the initial conditions $g_i(w^*) \leqslant 0$, which means that if $\alpha_i^* > 0$, then KKT condition implies $g_i(w^*) = 0$. In most cases,

$$\alpha_i^* > 0 \Leftrightarrow g_i(w^*) = 0$$

($g_i(w)$ is an **active constraint**).

21 SVM dual problem

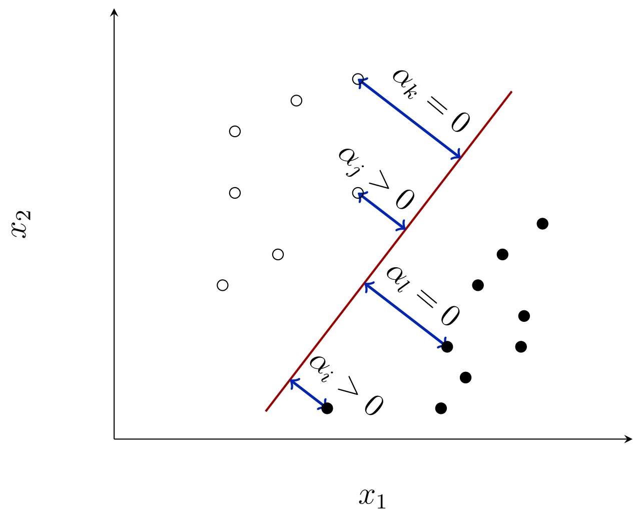
In this section we apply the idea of Lagrange multipliers to the SVM optimization problem and formulate a SVM dual problem. The SVM optimization problem has been formulated in the previous lecture as

$$\min_{w,b} \frac{||w||^2}{2},$$

subject to

$$y^{(i)}(w^T x^{(i)} + b) \geqslant 1, \ i = 1, \ldots, m.$$

We define $g_i(w,b) = -y^{(i)}(w^T x^{(i)} + b) + 1 \leqslant 0$. Notice that we do not have coefficients β as there are no constraints for h. If $\alpha_i > 0$, then $g_i(w,b) = 0$ (active constraint) and implies that the training example $(x^{(i)}, y^{(i)})$ has a functional margin equals to 1.



The Lagrangian has the form

$$L(w,b,\alpha) = \frac{||w||^2}{2} - \sum_{i=1}^{m} \alpha_i (y^{(i)}(w^T x^{(i)} + b) - 1)$$

and dual problem is

$$\theta_D(\alpha) = \min_{w,b} L(w,b,\alpha).$$

In order to minimize the Lagrangian we find derivatives and set them to zero:

$$\nabla_w L(w,b,\alpha) = w - \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} = 0 \Rightarrow w = \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} \tag{21.1}$$

and

$$\frac{\partial L}{\partial b} = -\sum_{i=1}^{m} y^{(i)} \alpha_i = 0.$$

Substitute these conditions back to the Lagrangian:

$$\begin{aligned} L(w,b,\alpha) &= \frac{1}{2} w^T w - \sum_{i=1}^{m} \alpha_i (y^{(i)}(w^T x^{(i)} + b) - 1) = \\ &= \frac{1}{2} \left(\sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} \right)^T \left(\sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} \right) - \sum_{i=1}^{m} \alpha_i (y^{(i)}(w^T x^{(i)} + b) - 1) = \\ &= \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \left\langle x^{(i)}, x^{(j)} \right\rangle - \sum_{i=1}^{m} \sum_{j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \left\langle x^{(i)}, x^{(j)} \right\rangle + \sum_{i=1}^{m} \alpha_i = \\ &= \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \left\langle x^{(i)}, x^{(j)} \right\rangle = W(\alpha), \end{aligned}$$

where $\langle \cdot, \cdot \rangle$ is a notation for the dot product of two vectors.

Finally, the **SVM dual problem** is to find

$$\boxed{\begin{array}{l}\max_\alpha W(\alpha) = \max_\alpha \left(\sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m y^{(i)} y^{(j)} \alpha_i \alpha_j \left\langle x^{(i)}, x^{(j)} \right\rangle \right) \\ \text{subject to} \\ \qquad \alpha_i \geqslant 0, \\ \qquad \sum_i y^{(i)} \alpha_i = 0. \end{array}}$$

Notice that if $\sum_i y^{(i)} \alpha_i \neq 0$, then $\theta_D(\alpha) = -\infty$, otherwise, $\theta_D(\alpha) = W(\alpha)$.

Example. The SVM dual problem for the dataset (see the previous sections):

x_1	x_2	y
-1	-2	1
1	-1	1
0	2	-1
1	3	-1

is formulated as:

$$\begin{aligned}&\max_\alpha(\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 - \frac{5}{2}\alpha_1^2 - \alpha_2^2 - 2\alpha_3^2 - 5\alpha_4^2 \\ &-\alpha_1\alpha_2 - 4\alpha_1\alpha_3 - 7\alpha_1\alpha_4 - 2\alpha_2\alpha_3 - 3\alpha_2\alpha_4 - 6\alpha_3\alpha_4)\end{aligned}$$

subject to

$$\begin{aligned}&\alpha_i \geqslant 0, \\ &\alpha_1 + \alpha_2 - \alpha_3 - \alpha_4 = 0.\end{aligned}$$

After we find the solution α^*, the coefficients can be found as

$$w = \sum_{i=1}^m \alpha_i^* y^{(i)} x^{(i)} \tag{21.2}$$

and we use the worst positive and negative training examples to find b:

$$b = \frac{\max_{i:y^{(i)}=-1} w^T x^{(i)} + \min_{i:y^{(i)}=1} w^T x^{(i)}}{2}.$$

With the equation (21.1) the hypothesis for the new test point x is expressed in terms of dot products:

$$h_{w,b} = \operatorname{sign}(w^T x + b) = \operatorname{sign}\left(\sum_{i=1}^m \alpha_i y^{(i)} \left\langle x^{(i)}, x \right\rangle + b \right) \tag{21.3}$$

22 Kernels

The idea of kernels is that often features are high dimensional ($x^{(i)} \in \mathbb{R}^m$) but instead of feature representations it is enough to find dot products.

Example. Assuming we have the problem with one feature $x \in \mathbb{R}$ only, the polynomial regression of the fourth order can be represented as a linear regression with the following list of features:

$$\varphi(x) : x \to \begin{bmatrix} x \\ x^2 \\ x^3 \\ x^4 \end{bmatrix}$$

For the SVM optimization problem the hypothesis (21.3) is replaced by

$$h_{w,b} = \operatorname{sign}(w^T\varphi(x) + b) = \operatorname{sign}\left(\sum_{i=1}^{m} \alpha_i y^{(i)} \left\langle \varphi(x)^{(i)}, \varphi(x) \right\rangle + b\right),$$

and in all following calculations we should replace the dot product $\left\langle x^{(i)}, x^{(j)} \right\rangle$ by $\left\langle \varphi(x^{(i)}), \varphi(x^{(j)}) \right\rangle$.

There are no any restrictions for the mapping $\varphi(x)$, in fact it is possible to have infinite dimensional $\varphi(x) \in \mathbb{R}^\infty$. Fortunately, for many different φ we can specify the function (**kernel**) that defines the dot product:

$$K(x^{(i)}, x^{(j)}) = \left\langle \varphi(x^{(i)}), \varphi(x^{(j)}) \right\rangle.$$

In such situations we do not need to compute $\varphi(x)$ explicitly, but we should compute the kernel $K(x, z)$ (which is less computationally expensive than computing $\varphi(x)$).

22.1 Kernel examples

1. $K(x, z) = (x^T z)^2$, where $x, z \in \mathbb{R}^n$. We try to transform this kernel to the exact form of dot product:

$$K(x, z) = (x^T z)^2 = \left(\sum_{i=1}^{n} x_i z_i\right)\left(\sum_{j=1}^{n} x_j z_j\right) = \sum_{i=1}^{n}\sum_{j=1}^{n}(x_i x_j)(z_i z_j),$$

that can be interpreted as a dot product of vectors that contains all possible combinations of x and z components. For example, if $n = 3$, then

$$\varphi(x) : x \to \begin{bmatrix} x_1x_1 \\ x_1x_2 \\ x_1x_3 \\ x_2x_1 \\ x_2x_2 \\ x_2x_3 \\ x_3x_1 \\ x_3x_2 \\ x_3x_3 \end{bmatrix}$$

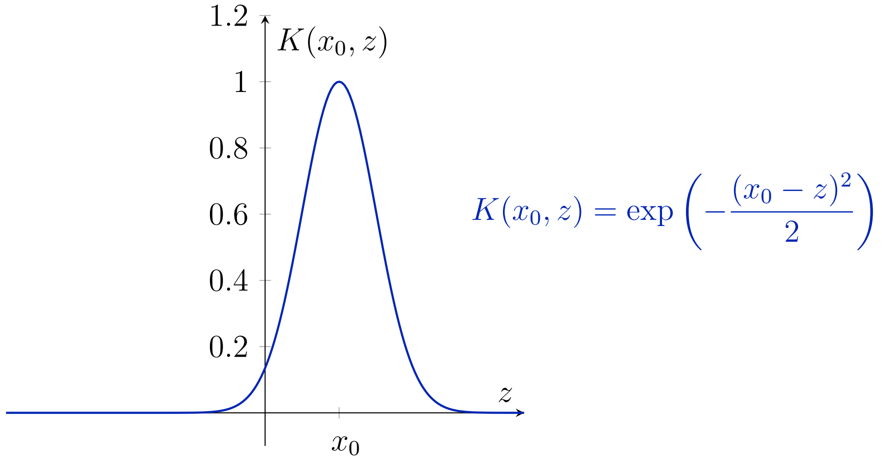
To compute the dot product $\langle \varphi(x), \varphi(z) \rangle$ for two training examples we need $O(2n^2 + n)$ operations. If we use kernel for that we need $O(n)$ operations only (because we just calculate dot product of two vectors $x^T z$ and take square of it).

2. $K(x,z) = (x^Tz + c)^2$ corresponds to

$$\varphi(x) : x \to \begin{bmatrix} x_1x_1 \\ x_1x_2 \\ x_1x_3 \\ x_2x_1 \\ x_2x_2 \\ x_2x_3 \\ x_3x_1 \\ x_3x_2 \\ x_3x_3 \\ \sqrt{2c}\cdot x_1 \\ \sqrt{2c}\cdot x_2 \\ \sqrt{2c}\cdot x_3 \\ c \end{bmatrix}$$

3. $K(x,z) = (x^Tz + c)^d$ corresponds to $\binom{n+d}{d}$ features of all monomials up to degree d.

4. $K(x,z) = \exp\left(-\frac{||x-z||^2}{2\sigma^2}\right)$ (**radial basis function (RBF) kernel**) corresponds to the transformation of feature space into an infinite dimensional Hilbert space. The intuition of this kernel is that if x and z are very similar than they will be pointing to the same direction and dot product should be large. In contrast if x and z are very different, then the dot product should be very small. If we fix x and consider $K(x,z)$ as a function of z the graph of this function is a bell shaped function:



22.2 Kernel testing

Assuming that we have chosen some function $K(x,z)$ as a kernel. The main question is: does there exist some $\varphi(x)$ such that $K(x,z) = \langle\varphi(x), \varphi(z)\rangle$?

Definition. For the given set of points $\{x^{(i)}, \dots, x^{(m)}\}$ a **kernel matrix $\mathbf{K} \in \mathbb{R}^{m\times m}$** is defined by

$$\mathbf{K}_{ij} = K(x^{(i)}, x^{(j)}), \tag{22.1}$$

where K is a kernel function.

Theorem (Mercer). Let $K(x, z)$ be given. Then K is a valid (Mercer) kernel (i.e. there exists φ such that $K(x, z) = \langle\varphi(x), \varphi(z)\rangle$) if and only if for all $\{x^{(i)}, \dots, x^{(m)}\}$ the kernel matrix $\mathbf{K} \in \mathbb{R}^{m\times m}$ is symmetric positive semi-definite.

Indeed, for any vectors $x, z \in \mathbb{R}^n$

$$\begin{aligned} z^T\mathbf{K}z &= \sum_i\sum_j z_i\mathbf{K}_{ij}z_j = \sum_i\sum_j z_i\varphi(x^{(i)})^T\varphi(x^{(j)})z_j = \\ &= \sum_i\sum_j z_i \sum_k(\varphi(x^{(i)}))_k(\varphi(x^{(j)}))_k\, z_j = \sum_k\sum_i\sum_j z_i\, (\varphi(x^{(i)}))_k(\varphi(x^{(j)}))_k\, z_j = \\ &= \sum_k\left(\sum_i z_i\varphi(x^{(i)})_k\right)^2 \geqslant 0. \end{aligned}$$

Here we used a fact that $a^Tb = \sum\limits_k a_kb_k$.

Example. $K(x, z) = -1$ is not a valid kernel function.

22.3 SVM with kernels

We can reformulate the SVM dual problem from the previous section as follows:

$$\max_\alpha W(\alpha) = \max_\alpha\left(\sum_{i=1}^m \alpha_i - \frac{1}{2}\sum_{i=1}^m\sum_{j=1}^m y^{(i)}y^{(j)}\alpha_i\alpha_jK\left(x^{(i)}, x^{(j)}\right)\right)$$

subject to

$$\alpha_i \geqslant 0,$$
$$\sum_i y_i\alpha_i = 0,$$

with the prediction for the new test point x

$$h_{w,b} = \operatorname{sign}\left(\sum_{i=1}^m \alpha_iy^{(i)}K(x^{(i)}, x) + b\right), \tag{22.2}$$

where K is a chosen kernel function.

The last remark is that the kernel idea is more general than SVM and we can formulate many algorithms in terms of dot products.

23 Soft margin

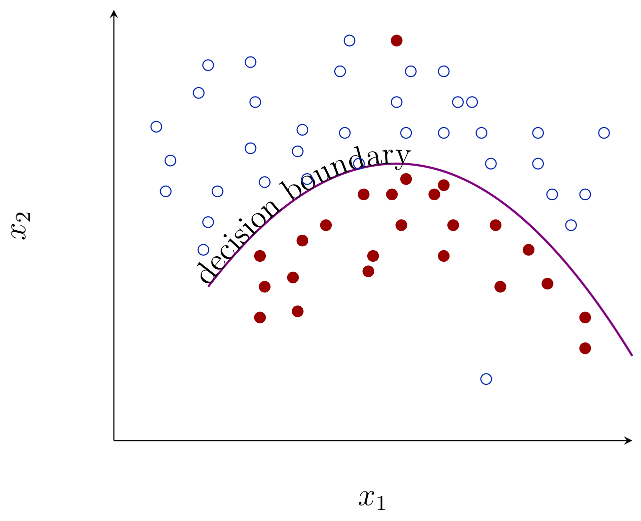
In case of non linear decision boundaries the SVM algorithm is called L_1 **norm soft margin SVM** and formulated as follows:

$$\min_w \frac{||w||^2}{2} + C\sum_{i=1}^m \xi_i$$

subject to

$$y^{(i)}(w^T x^{(i)} + b) \geqslant 1 - \xi_i, \ \xi_i \geqslant 0, \ i = 1, \dots, m.$$

Such formulation is useful for non linear separable datasets, for example, in the next picture we cannot find the hyperplane that separates two classes.



Remember that if $y^{(i)}(w^T x^{(i)} + b) > 0$, then the example is classified correctly. With the above formulation we allow the algorithm to misclassify something (because of the term $1 - \xi_i$), but we encourage the algorithm not to do it, because it will increase the objective function by $\sum\limits_{i=1}^{m} \xi_i$. Notice that this is also convex optimization problem.

As before we find the derivatives of Lagrangian

$$L(w, b, \xi, \alpha, r) = \frac{1}{2}||w||^2 + C\sum_i \xi_i - \sum_{i=1}^{m} \alpha_i (y^{(i)}(w^T x^{(i)} + b) - 1 + \xi_i) - \sum_{i=1}^{m} r_i \xi_i$$

and equate them to zero:

$$\begin{aligned}
&\nabla_w L(w, b, \xi, \alpha, r) = w - \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} = 0 \Rightarrow w = \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)}, \\
&\frac{\partial L}{\partial b} = -\sum_{i=1}^{m} \alpha_i y^{(i)} = 0, \\
&\frac{\partial L}{\partial \xi_i} = C - \alpha_i - r_i = 0.
\end{aligned}$$

We can also add the KKT conditions:

$$\begin{aligned}
&\alpha_i (y^{(i)}(w^T x^{(i)} + b) - 1 + \xi_i) = 0, \\
&r_i \xi_i = 0.
\end{aligned}$$

Taking into consideration all these conditions we derive

$$\alpha_i = 0 \Rightarrow r_i = C > 0 \Rightarrow \xi_i = 0 \Rightarrow y^{(i)}(w^T x^{(i)} + b) \geqslant 1 \tag{23.1}$$

$$\alpha_i = C \Rightarrow y^{(i)}(w^T x^{(i)} + b) = 1 - \xi_i \Rightarrow y^{(i)}(w^T x^{(i)} + b) \leqslant 1 \tag{23.2}$$

$$0 < \alpha_i < C \Rightarrow r_i > 0 \Rightarrow \xi_i = 0 \Rightarrow y^{(i)}(w^T x^{(i)} + b) = 1 \tag{23.3}$$

To obtain the dual problem we substitute all these conditions to the Lagrangian:

$$\begin{aligned}
&L(w,b,\xi,\alpha,r) = \frac{1}{2}w^T w + C\sum_i \xi_i - \sum_{i=1}^m \alpha_i(y^{(i)}(w^T x^{(i)}+b) - 1 + \xi_i) - \sum_{i=1}^m r_i\xi_i = \\
&= \frac{1}{2}\left(\sum_{i=1}^m \alpha_i y^{(i)} x^{(i)}\right)^T \left(\sum_{i=1}^m \alpha_i y^{(i)} x^{(i)}\right) - \sum_{i=1}^m \alpha_i(y^{(i)}(w^T x^{(i)}+b) - 1) - \sum_i (C - \alpha_i - r_i)\xi_i = \\
&= \frac{1}{2}\sum_{i=1}^m\sum_{j=1}^m y^{(i)}y^{(j)}\alpha_i\alpha_j \left\langle x^{(i)}, x^{(j)}\right\rangle - \sum_{i=1}^m\sum_{j=1}^m y^{(i)}y^{(j)}\alpha_i\alpha_j \left\langle x^{(i)}, x^{(j)}\right\rangle + \sum_{i=1}^m \alpha_i = \\
&= \sum_{i=1}^m \alpha_i - \frac{1}{2}\sum_{i=1}^m\sum_{j=1}^m y^{(i)}y^{(j)}\alpha_i\alpha_j \left\langle x^{(i)}, x^{(j)}\right\rangle = W(\alpha).
\end{aligned}$$

Finally, the dual optimization problem with the kernel idea is stated as

$$\max_\alpha W(\alpha) = \sum_{i=1}^m \alpha_i - \frac{1}{2}\sum_{i=1}^m\sum_{j=1}^m y^{(i)}y^{(j)}\alpha_i\alpha_j K(x^{(i)}, x^{(j)}) \tag{23.4}$$

subject to

$$\begin{aligned}
&\sum_{i=1}^m y^{(i)}\alpha_i = 0, \\
&0 \leqslant \alpha_i \leqslant C,\ i = 1,\dots,m.
\end{aligned} \tag{23.5}$$

24 SMO algorithm

In this section we come up with an efficient algorithm that solves the SVM optimization problem. First, consider the problem

$$\max_\alpha W(\alpha_1,\dots,\alpha_m)$$

without constraint on α's.

Algorithm 4 Coordinate ascent algorithm

```
1: repeat
2:     for $i = 1$ to $m$ do
3:         $\alpha_i = \arg\max_{\hat\alpha_i} W(\alpha_1,\dots,\alpha_{i-1},\hat\alpha_i,\alpha_{i+1},\dots,\alpha_m)$ (freeze all variables except $\alpha_i$)
4: until convergence
```

Compared to the gradient descent this algorithm takes much more steps, but for many optimization problems it is very easy to make step by one parameter.

We apply it for our SVM dual optimization problem. Unfortunately, this algorithm does not work in a straight way, because of the condition $\sum_{i=1}^m y^{(i)}\alpha_i = 0$ (if we fix all α's except one, then we can find the last α explicitly). That's why we try to optimize two α's per step.

Algorithm 5 Sequential Minimal Optimization (SMO) algorithm

```
1: define the way how to choose pairs of $\alpha$'s for the following loop
2: repeat
3:     for pairs $\alpha_i$ and $\alpha_j$ do
4:         $\alpha_i, \alpha_j = \arg\max_{\hat{\alpha}_i, \hat{\alpha}_j} W(\alpha_1, \ldots, \hat{\alpha}_i, \ldots, \hat{\alpha}_j, \ldots, \alpha_m)$ (freeze all variables except $\alpha_i$ and $\alpha_j$)
5: until convergence
```

We elaborate more about the implementation of this algorithm. Without loss of generality we update α_1 and α_2. The general case could be obtained in the same manner by replacing α_1 by α_i and α_2 by α_j. We assume that we have α_i^{old} from the previous step of the SMO algorithm, for which conditions (23.5) hold:

$$\begin{aligned}&\sum_{i=1}^m y^{(i)}\alpha_i^{old} = 0,\\ &0 \leqslant \alpha_i^{old} \leqslant C.\end{aligned}$$

The first equation can be transformed to

$$\alpha_1^{old}y^{(1)} + \alpha_2^{old}y^{(2)} = -\sum_{i=3}^m \alpha_i^{old}y^{(i)} \Rightarrow \alpha_1^{old} + \alpha_2^{old}y^{(2)}y^{(1)} = -y^{(1)}\sum_{i=3}^m \alpha_i^{old}y^{(i)},$$

and the right-hand side of the last equation will be denoted by ζ, then

$$\begin{aligned}&\alpha_1^{old} + s\alpha_2^{old} = \alpha_1 + s\alpha_2 = \zeta,\\ &\text{where } s = y^{(1)}y^{(2)}.\end{aligned}$$

Notice that ζ does not change after one step of the SMO algorithm.

We introduce the following notations (using (22.1)):

$$\begin{aligned}&h(x) = \sum_{i=1}^m \alpha_i y^{(i)} K(x^{(i)}, x) + b,\\ &v_j = \sum_{i=3}^m y^{(i)}\alpha_i \mathbf{K}_{ij} = h(x^{(j)}) - b - \alpha_1 y^{(1)}\mathbf{K}_{1j} - \alpha_2 y^{(2)}\mathbf{K}_{2j}.\end{aligned}$$

Then

$$\begin{aligned}&W(\alpha_1, \alpha_2, \ldots, \ldots) = \alpha_1 + \alpha_2 - \frac{1}{2}(\alpha_1)^2\mathbf{K}_{11} - \frac{1}{2}(\alpha_2)^2\mathbf{K}_{22} - y^{(1)}y^{(2)}\alpha_1\alpha_2\mathbf{K}_{12}\\ &-\alpha_1 y^{(1)}\sum_{i=3}^m y^{(i)}\alpha_i\mathbf{K}_{i1} - \alpha_2 y^{(2)}\sum_{i=3}^m y^{(i)}\alpha_i\mathbf{K}_{i2} + V(\alpha_3, \ldots, \alpha_m) =\\ &= \alpha_1 + \alpha_2 - s\alpha_1\alpha_2\mathbf{K}_{12} - \frac{1}{2}(\alpha_1)^2\mathbf{K}_{11} - \frac{1}{2}(\alpha_2)^2\mathbf{K}_{22}\\ &-\alpha_1 y^{(1)}v_1 - \alpha_2 y^{(2)}v_2 + V(\alpha_3, \ldots, \alpha_m) =\end{aligned}$$

substitute the expression $\alpha_1 = \zeta - s\alpha_2$:

$$\begin{aligned}&= \zeta - s\alpha_2 + \alpha_2 - s(\zeta - s\alpha_2)\alpha_2\mathbf{K}_{12} - \frac{1}{2}(\zeta - s\alpha_2)^2\mathbf{K}_{11} - \frac{1}{2}(\alpha_2)^2\mathbf{K}_{22}\\ &-(\zeta - s\alpha_2)y^{(1)}v_1 - \alpha_2 y^{(2)}v_2 + V(\alpha_3, \ldots, \alpha_m).\end{aligned}$$

We have obtained the quadratic function with respect to α_2. To find the maximum we find the derivative and equate it to zero:

$$\begin{aligned}&W'_{\alpha_2} = -s + 1 - s\zeta\mathbf{K}_{12} + 2\alpha_2\mathbf{K}_{12}\\&+\zeta s\mathbf{K}_{11} - \alpha_2\mathbf{K}_{11} - \alpha_2\mathbf{K}_{22} + y^{(2)}v_1 - y^{(2)}v_2 = 0.\end{aligned}$$

Then

$$2\alpha_2\mathbf{K}_{12} - \alpha_2\mathbf{K}_{11} - \alpha_2\mathbf{K}_{22} = s - 1 + s\zeta\mathbf{K}_{12} - \zeta s\mathbf{K}_{11} - y^{(2)}v_1 + y^{(2)}v_2.$$

Substitute the expressions for v_1, v_2 and $\zeta = \alpha_1^{old} + s\alpha_2^{old}$:

$$\begin{aligned}&\alpha_2(2\mathbf{K}_{12} - \mathbf{K}_{11} - \mathbf{K}_{22}) = s - 1 + s(\alpha_1^{old} + s\alpha_2^{old})(\mathbf{K}_{12} - \mathbf{K}_{11})\\&-y^{(2)}(h(x^{(1)}) - b - \alpha_1^{old}y^{(1)}\mathbf{K}_{11} - \alpha_2^{old}y^{(2)}\mathbf{K}_{12})\\&+y^{(2)}(h(x^{(2)}) - b - \alpha_1^{old}y^{(1)}\mathbf{K}_{12} - \alpha_2^{old}y^{(2)}\mathbf{K}_{22}) =\\&= s - 1 + \alpha_2^{old}(2\mathbf{K}_{12} - \mathbf{K}_{11} - \mathbf{K}_{22}) - y^{(2)}h(x^{(1)}) + y^{(2)}h(x^{(2)})\end{aligned}$$

and finally, using $s - 1 = y^{(2)}(y^{(1)} - y^{(2)})$:

$$\alpha_2 = \alpha_2^{old} - y^{(2)}\frac{(h(x^{(1)}) - y^{(1)}) - (h(x^{(2)}) - y^{(2)})}{2\mathbf{K}_{12} - \mathbf{K}_{11} - \mathbf{K}_{22}} \tag{24.1}$$

and

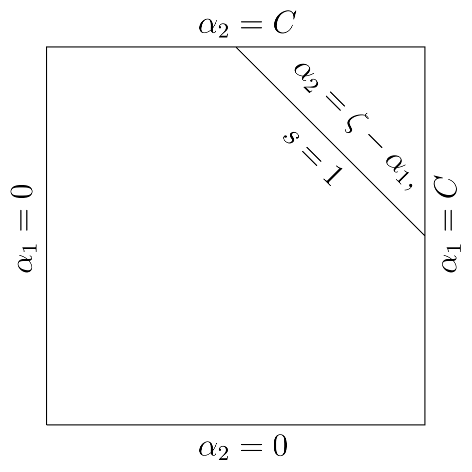
$$\alpha_1 = \zeta - s\alpha_2 = \alpha_1^{old} + s(\alpha_2^{old} - \alpha_2) \tag{24.2}$$

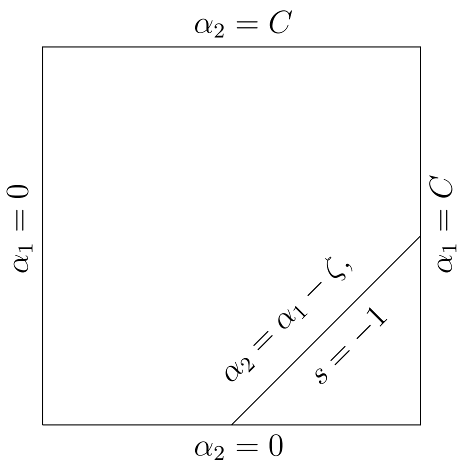
Notice that h in the formula (24.1) is calculated for old values of α's.

We have not used the conditions $0 \leqslant \alpha_i \leqslant C$ yet. Remember that ζ is a constant during each step of the SMO algorithm, which means that we can consider the equation

$$\alpha_1 + s\alpha_2 = \zeta \Rightarrow \alpha_2 = s\zeta - s\alpha_1$$

as an equation of the straight line.





There are two possible cases (see the figures):

- If $s = 1$, then $\alpha_2 = \zeta - \alpha_1$. Then we will have the following chain of implications:

$$0 \leqslant \alpha_1 \leqslant C \Rightarrow \zeta - C \leqslant \zeta - \alpha_1 \leqslant \zeta \Rightarrow \zeta - C \leqslant \alpha_2 \leqslant \zeta,$$

which means that $\max(0, \zeta - C) \leqslant \alpha_2 \leqslant \min(\zeta, C)$, or using $\zeta = \alpha_1^{old} + s\alpha_2^{old}$ ($s = 1$):

$$\max(0, \alpha_1^{old} + \alpha_2^{old} - C) \leqslant \alpha_2 \leqslant \min(\alpha_1^{old} + \alpha_2^{old}, C) \tag{24.3}$$

- If $s = -1$, then $\alpha_2 = \alpha_1 - \zeta$. In this case:

$$0 \leqslant \alpha_1 \leqslant C \Rightarrow -\zeta \leqslant \alpha_1 - \zeta \leqslant C - \zeta \Rightarrow -\zeta \leqslant \alpha_2 \leqslant C - \zeta,$$

which means that $\max(0, -\zeta) \leqslant \alpha_2 \leqslant \min(C-\zeta, C)$, or using $\zeta = \alpha_1^{old} + s\alpha_2^{old}$ $(s = -1)$:

$$\max(0, \alpha_2^{old} - \alpha_1^{old}) \leqslant \alpha_2 \leqslant \min(C + \alpha_2^{old} - \alpha_1^{old}, C) \tag{24.4}$$

The KKT conditions also give the formula to calculate b. Assuming that after one step of the SMO algorithm we got $0 < \alpha_2 < C$, then

$$y^{(2)}(w^T x^{(2)} + b) = 1 \Rightarrow w^T x^{(2)} + b = y^{(2)},$$

implies

$$\begin{aligned} & b_2 = y^{(2)} - w^T x^{(2)} = y^{(2)} - \sum_{i=1}^{m} y^{(i)} \alpha_i \mathbf{K}_{i2} = \\ & = y^{(2)} - y^{(1)}\alpha_1 \mathbf{K}_{12} - y^{(2)}\alpha_2 \mathbf{K}_{22} - \sum_{i=3}^{m} y^{(i)} \alpha_i \mathbf{K}_{i2} = \\ & = y^{(2)} - y^{(1)}\alpha_1 \mathbf{K}_{12} - y^{(2)}\alpha_2 \mathbf{K}_{22} - (h(x^{(2)}) - b - \alpha_1^{old} y^{(1)} \mathbf{K}_{12} - \alpha_2^{old} y^{(2)} \mathbf{K}_{22}) = \\ & = b^{old} - (h(x^{(2)}) - y^{(2)}) - y^{(2)} \mathbf{K}_{22}(\alpha_2 - \alpha_2^{old}) - y^{(1)} \mathbf{K}_{12}(\alpha_1 - \alpha_1^{old}) \end{aligned} \tag{24.5}$$

Similarly, if $0 < \alpha_1 < C$:

$$b_1 = b^{old} - (h(x^{(1)}) - y^{(1)}) - y^{(2)} \mathbf{K}_{12}(\alpha_2 - \alpha_2^{old}) - y^{(1)} \mathbf{K}_{11}(\alpha_1 - \alpha_1^{old}) \tag{24.6}$$

If none of the conditions $0 < \alpha_1 < C$ and $0 < \alpha_2 < C$ is true, then we can take the average $\dfrac{b_1 + b_2}{2}$ (any b between b_1 and b_2 satisfies to the KKT conditions).

When we switch to the general case and take any pair of α_i, α_j, first we choose α_j such that it does not satisfy the KKT condition (23.3) (with some tolerance γ):

$$0 < \alpha_j < C \Rightarrow y^{(j)}(w^T x^{(j)} + b) = 1 \Rightarrow y^{(j)}(h(x^{(j)}) - y^{(j)}) = 0 \tag{24.7}$$

Also notice that if $\alpha_j = C$, then we could have $y^{(j)}(h(x^{(j)}) - y^{(j)}) < 0$ and if $\alpha_j = 0$, then we could have $y^{(j)}(h(x^{(j)}) - y^{(j)}) > 0$. The following algorithm summarizes all our calculations with references to the formulas.

Algorithm 6 SMO algorithm for Support Vector Machine

```
1: set $C$, $\gamma$, initial values $\alpha_i = 0$, $b = 0$
2: repeat
3:   for $j = 1$ to $m$ do
4:     evaluate $E_j = h(x^{(j)}) - y^{(j)}$
5:     if $(y^{(j)}E_j < -\gamma$ and $\alpha_j < C)$ or $(y^{(j)}E_j > \gamma$ and $\alpha_j > 0)$ then    ▷ (24.7)
6:       repeat
7:         choose $\alpha_i, i \neq j$, randomly
8:         evaluate $E_i = h(x^{(i)}) - y^{(i)}$
9:         if $y^{(i)} \cdot y^{(j)} > 0$ then
10:          $L = \max(0, \alpha_i + \alpha_j - C)$    ▷ (24.3)
11:          $H = \min(\alpha_i + \alpha_j, C)$    ▷ (24.3)
12:        else
13:          $L = \max(0, \alpha_j - \alpha_i)$    ▷ (24.4)
14:          $H = \min(C + \alpha_j - \alpha_i, C)$    ▷ (24.4)
15:        if $L == H$ then continue
16:        evaluate $\eta = 2\mathbf{K}_{ij} - \mathbf{K}_{ii} - \mathbf{K}_{jj}$
17:        if $\eta == 0$ then continue
18:        evaluate $\alpha_j^{new} = \min(\max(\alpha_j - y^{(j)}\frac{E_i - E_j}{\eta}, L), H)$    ▷ (24.1)
19:        if $|\alpha_j^{new} - \alpha_j| < 10^{-5}$ then continue
20:        evaluate $\alpha_i^{new} = \alpha_i + y^{(j)}y^{(i)}(\alpha_j - \alpha_j^{new})$    ▷ (24.2)
21:        if $(\alpha_j^{new} > 0$ and $\alpha_j^{new} < C)$ then
22:          $b = b - E_j - y^{(j)}\mathbf{K}_{jj}(\alpha_j^{new} - \alpha_j) - y^{(i)}\mathbf{K}_{ij}(\alpha_i^{new} - \alpha_i)$    ▷ (24.5)
23:        else
24:          if $(\alpha_i^{new} > 0$ and $\alpha_i^{new} < C)$ then
25:            $b = b - E_i - y^{(j)}\mathbf{K}_{ij}(\alpha_j^{new} - \alpha_j) - y^{(i)}\mathbf{K}_{ii}(\alpha_i^{new} - \alpha_i)$    ▷ (24.6)
26:          else
27:            $b = b - 0.5 \cdot (E_i + E_j$
               $+y^{(j)}(\mathbf{K}_{jj} + \mathbf{K}_{ij})(\alpha_j^{new} - \alpha_j)$
               $+y^{(i)}(\mathbf{K}_{ij} + \mathbf{K}_{ii})(\alpha_i^{new} - \alpha_i))$
                                                                        ▷ (24.5), (24.6)
28:      until False
29: until convergence
30: return $\alpha$, $b$
```

24.1 Python implementation

Loading necessary libraries:

```
In [1]: import numpy as np
        import random
        import math
        import sklearn.datasets as ds
        from matplotlib import cm
        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D
```

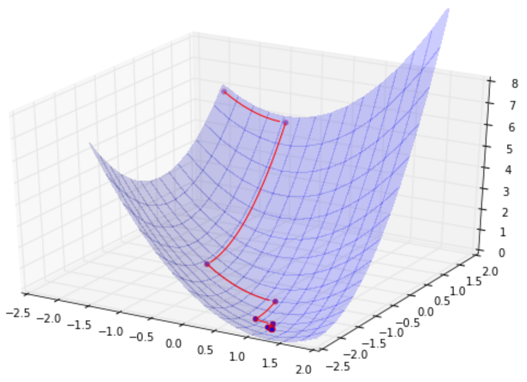
The coordinate ascent algorithm for the function

$$z = x^2 + y^2 + 2x + xy$$

can be implemented as follows:

```
In [2]: def fun(a,b):
            return a**2 + b**2 + 2*b + a*b

        def dfun(a,b,var):
            if var == 0:
                return -b/2.0
            else:
                return - 1.0 - 0.5*a

        def coordinate_ascent(theta0, iters):
            history = [np.copy(theta0)] # to store all thetas
            theta = theta0 # initial values for thetas
            for i in range(iters): # number of iterations
                for j in range(2): # number of variables
                    theta[j] = dfun(theta[0], theta[1], j)
                    history.append(np.copy(theta))
            return history

        history = coordinate_ascent(theta0 = [-1.8, 1.6], iters = 7)

        fig = plt.figure(figsize=(10, 6))
        ax = fig.gca(projection='3d')
        plt.hold(True)
        a = np.arange(-1.85, 1.85, 0.25)
        b = np.arange(-1.85, 1.85, 0.25)
        a, b = np.meshgrid(a, b)
        c = fun(a,b)
        surf = ax.plot_surface(a, b, c, rstride=1, cstride=1, alpha=0.2,
                               linewidth=0, antialiased=False)
        ax.set_zlim(-0.01, 8.01)

        a = np.array([x[0] for x in history])
        b = np.array([x[1] for x in history])
        c = fun(a,b)
        ax.scatter(a, b, c, color="r");
```

```
    for i in range(len(history)-1):
        if history[i][0] == history[i+1][0]:
            b = np.arange(history[i][1], history[i+1][1],
                          np.sign(history[i+1][1] - history[i][1])*0.1)
            a = np.ones(len(b)) * history[i][0]
            c = fun(a,b)
            ax.plot(a, b, c, color="r")
        else:
            a = np.arange(history[i][0], history[i+1][0],
                          np.sign(history[i+1][0] - history[i][0])*0.1)
            b = np.ones(len(a)) * history[i][1]
            c = fun(a,b)
            ax.plot(a, b, c, color="r")

    plt.show()
```



There are several implementations of SVM in the `scikit-learn` package: SVC (for classification) and SVR (for regression).

```
In [3]: from sklearn.svm import SVC
```

As in the previous lecture we try to solve the XOR problem, but this time we are going to use Support Vector Machine for it.

$\mathbf{x_1}$	$\mathbf{x_2}$	XOR $(\mathbf{x_1}, \mathbf{x_2})$
1	1	1
1	0	0
0	1	0
0	0	1

The XOR function detects if x_1 and x_2 are the same. Define our dataset: 4 training examples with XOR as a target variable y.

```
In [4]: X_train = np.array([[1, 1], [1, 0], [0, 1], [0, 0]]).astype(np.float64)
        y_train = np.array([1, -1, -1, 1])
```

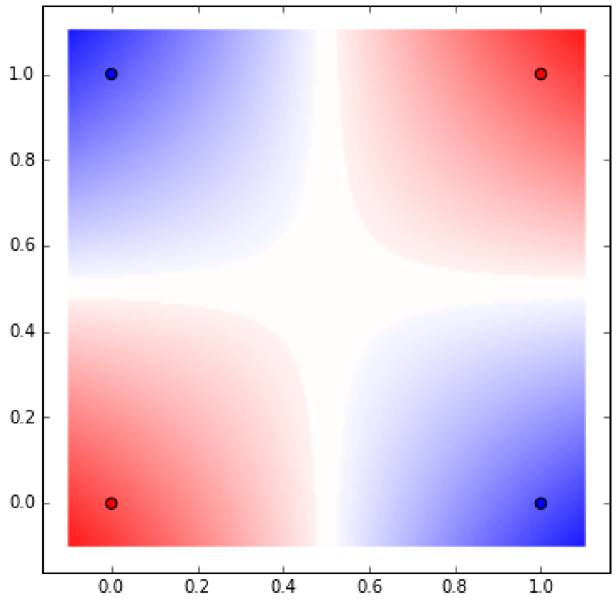
Train the SVC model:

```
In [5]: clf = SVC(probability=True, C=100.0, gamma=0.1, kernel='rbf')
        clf.fit(X_train, y_train)
```

We are going to use the function `plot_decision_boundary()` to draw the decision boundary.

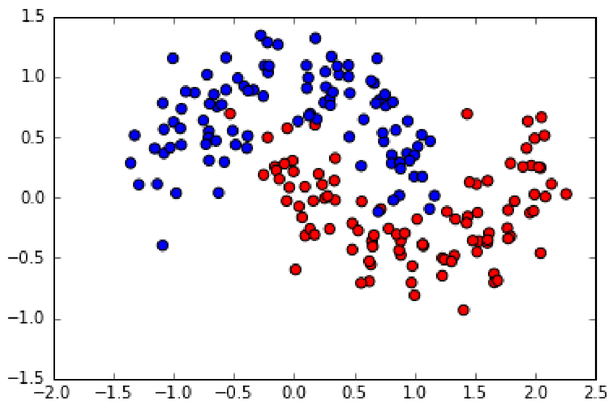
```
In [6]: def plot_decision_boundary(X, y, model,
                                   xmin=-0.1, xmax=1.1,
                                   ymin=-0.1, ymax=1.1):
            fig = plt.figure(figsize=(6,6))
            x1, x2 = np.meshgrid(np.linspace(xmin, xmax, 200),
                                 np.linspace(ymin, ymax, 200))
            ypred = model.predict_proba(np.c_[x1.ravel(), x2.ravel()])[:,0]
            ypred = ypred.reshape(x1.shape)
            extent = xmin, xmax, ymin, ymax

            plt.imshow(ypred, cmap=cm.bwr, alpha=.9, interpolation='bilinear',
                       extent = extent, origin='lower')
            plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=cm.bwr)
```

```
In [7]: plot_decision_boundary(X_train, y_train, clf)
```



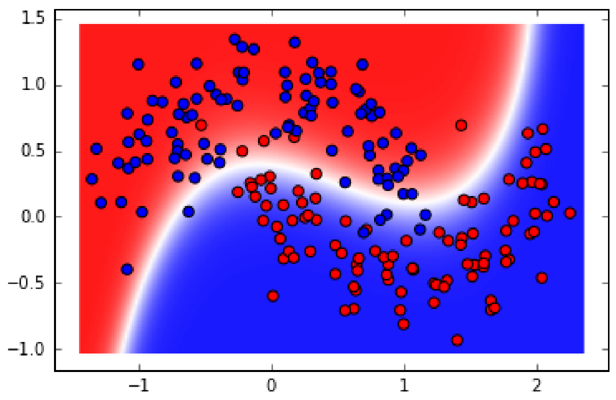
For the `make_moons` benchmark from the `scikit-learn`:

```
In [8]: np.random.seed(0)
        X_train, y_train = ds.make_moons(200, noise=0.20)
        plt.scatter(X_train[:,0], X_train[:,1], s=40, c=y_train, cmap=cm.bwr)
        plt.show()
```



```
In [9]: clf = SVC(probability=True, C=100.0, gamma=0.1, kernel='rbf')
        clf.fit(X_train, y_train)
```

```
In [10]: plot_decision_boundary(X_train, y_train, clf,
                                xmin=np.min(X_train[:,0])-0.1,
                                xmax=np.max(X_train[:,0])+0.1,
                                ymin=np.min(X_train[:,1])-0.1,
                                ymax=np.max(X_train[:,1])+0.1)
```



We can get some useful information from the model, for example, number of support vectors (number of i such that $\alpha_i \neq 0$):

```
In [11]: print "Number of support vectors: " + str(len(clf.support_))
```

```
Number of support vectors: 51
```

We have also implemented the simplified version of SMO algorithm. First we define the function that calculates a kernel. We are going to use RBF kernel, which means that $\mathbf{K}_{ii} = 1$ for any i.

```
In [12]: def kernel(x1, x2):
             return np.exp(-np.sum(np.power([i-j for i,j in zip(x1,x2)], 2))/2.0)
```

When we predict the output for the test dataset, we should calculate the kernel function for each pair (test example, support vector). It becomes very computationally expensive if you use loops for that. I have written the function that use some vectorized operations from numpy:

```
In [13]: def h(X, y, alpha, b, Xtest):
             ix = [i for i,a in enumerate(alpha) if a!=0]
             X_magnitude = 0.5*np.sum(np.power(X[ix,:], 2), axis=1)
             Xtest_magnitude = 0.5*np.sum(np.power(Xtest, 2), axis=1)
             dists = np.multiply(np.exp(np.dot(Xtest, X[ix,:].T) -
                                        np.add.outer(Xtest_magnitude,
                                                     X_magnitude)),
                                  y[ix])
             return np.sum(dists, axis=1) + b
```

The main function is smo_step() that takes X, y, α, b and index j as inputs and returns new values α and new value b. Notice that if this function fails it returns previous values for α and b.

```
In [14]: def smo_step(X, y, alpha, b, j):
             Ej = h(X, y, alpha, b, X[j,:].reshape((1,-1)))[0] - y[j]
             if ((y[j]*Ej < -gamma and alpha[j]<C) or
                 (y[j]*Ej > gamma and alpha[j]>0)):
                 ilist = [x for x in range(X.shape[0]) if x != j]
                 while True:
                     if len(ilist) == 0:
                         break
                     i = random.sample(ilist, 1)[0]
                     ilist.remove(i)
                     Ei = h(X, y, alpha, b, X[i,:].reshape((1,-1)))[0] - y[i]
                     if y[i]*y[j] > 0:
                         L = max(0.0, alpha[i] + alpha[j] - C)
                         H = min(alpha[i] + alpha[j], C)
                     else:
                         L = max(0.0, alpha[j] - alpha[j])
                         H = min(C + alpha[j] - alpha[i], C)
                     if L == H: continue
                     Kij = kernel(X[i,:], X[j,:])
                     eta = 2.0*Kij - 2.0
                     if eta == 0: continue
                     alpha_j = np.clip(alpha[j] - y[j]*(Ei - Ej)/eta, L, H)
                     if np.abs(alpha_j - alpha[j]) < 0.00001: continue
```

```
                    alpha_i = alpha[i] + y[i]*y[j]*(alpha[j] - alpha_j)
                    if (alpha_j > 0 and alpha_j < C):
                        b = b - Ej - y[j]*(alpha_j - alpha[j])
                                   - y[i]*Kij*(alpha_i - alpha[i])
                    else:
                        if (alpha_i > 0 and alpha_i < C):
                            b = b - Ei - y[j]*Kij*(alpha_j - alpha[j])
                                       - y[i]*(alpha_i - alpha[i])
                        else:
                            b = b - 0.5*(Ei + Ej
                                        + y[j]*(1.0 + Kij)*(alpha_j - alpha[j])
                                        + y[i]*(1.0 + Kij)*(alpha_i - alpha[i]))
                    alpha[i] = alpha_i
                    alpha[j] = alpha_j
                    break
            return alpha, b
```

One more auxiliary function plot_decision_boundary(). The main difference is that now we are using function h for the prediction.

```
In [15]: def plot_decision_boundary(X, y, alpha, b,
                                    xmin=-0.1, xmax=1.1,
                                    ymin=-0.1, ymax=1.1):
             x1, x2 = np.meshgrid(np.linspace(xmin, xmax, 200),
                                  np.linspace(ymin, ymax, 200))
             ypred = np.array(h(X, y, alpha, b, np.c_[x1.ravel(), x2.ravel()]))
             ypred = ypred.reshape(x1.shape)
             extent = xmin, xmax, ymin, ymax

             plt.imshow(ypred, cmap=cm.bwr, alpha=.9, interpolation='bilinear',
                        extent = extent, origin='lower')
             plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=cm.bwr)
```

As an example I have implemented the simple function that returns the number of support vectors:

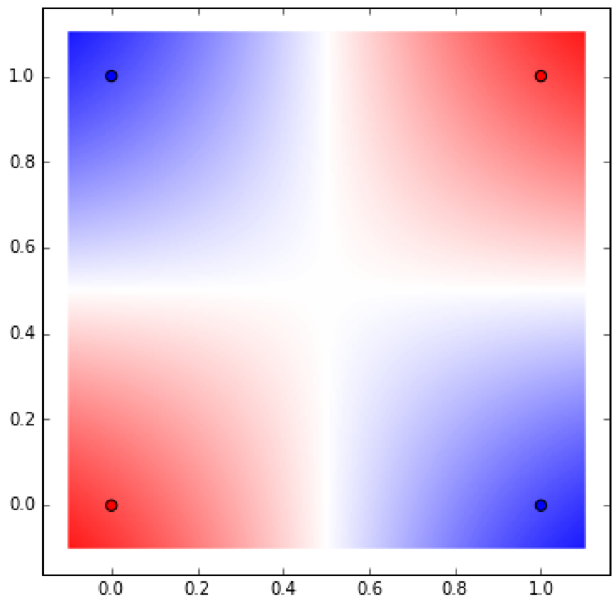
```
In [16]: def support_vector_count(alpha):
             return len([i for i,a in enumerate(alpha) if a!=0])
```

We solve the XOR problem using our implemented SVM.

```
In [17]: X_train = np.array([[1, 1], [1, 0], [0, 1], [0, 0]]).astype(np.float64)
         y_train = np.array([1, -1, -1, 1])
```

```
In [18]: ### define the parameters and initialize values for alpha and b
         C = 10.0
```

```
        gamma = 0.1
        b = 0.0
        alpha = np.zeros(X_train.shape[0])
        niter = 10

        ### parameters for plotting
        fig = plt.figure(figsize=(6,6))

        ### our main loop
        for it in range(niter):
            for j in range(X_train.shape[0]):
                alpha, b = smo_step(X_train, y_train, alpha, b, j)
        plot_decision_boundary(X_train, y_train, alpha, b,
                               xmin=np.min(X_train[:,0])-0.1,
                               xmax=np.max(X_train[:,0])+0.1,
                               ymin=np.min(X_train[:,1])-0.1,
                               ymax=np.max(X_train[:,1])+0.1)
        plt.show()
```



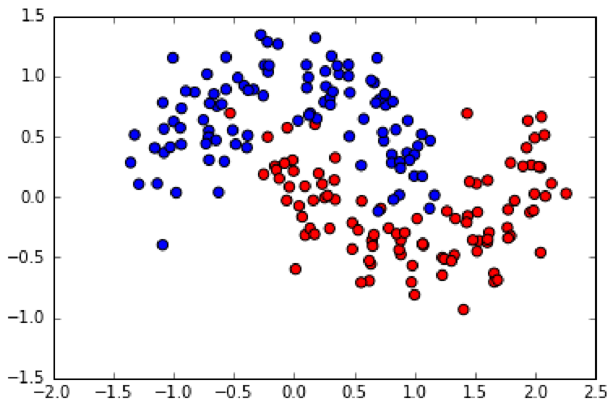
```
In [19]: print "alpha: " + str(alpha)
         print "b: " + str(b)
```

```
alpha: [ 10.  10.  10.  10.]
b: -0.368625049269
```

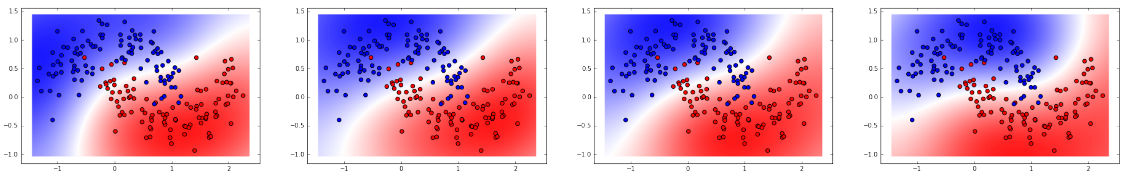
We check our implementation for `make_moons` data:

```
In [20]: np.random.seed(0)
         X_train, y_train = ds.make_moons(200, noise=0.20)
```

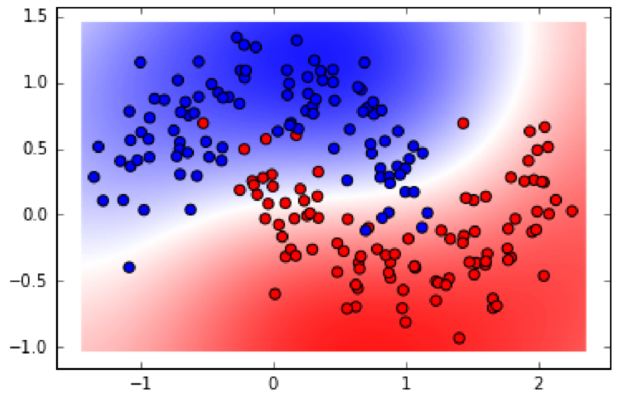
```
        y_train[y_train==0] = -1
        plt.scatter(X_train[:,0], X_train[:,1], s=40, c=y_train, cmap=cm.bwr)
        plt.show()
```



```
In [21]: ### define the parameters and initialize values for alpha and b
         C = 1000.0
         gamma = 0.001
         b = 0.0
         alpha = np.zeros(X_train.shape[0])
         niter = 600

         ### parameters for plotting
         iter_to_plot = [1, niter/3, 2*niter/3, niter-1]
         count_plot = 0
         fig = plt.figure(figsize=(32,32))

         ### our main loop
         for it in range(niter):
             for j in range(X_train.shape[0]):
                 alpha, b = smo_step(X_train, y_train, alpha, b, j)
             if it in iter_to_plot:
                 count_plot = count_plot + 1
                 fig.add_subplot(1,len(iter_to_plot),count_plot)
                 plot_decision_boundary(X_train, y_train, alpha, b,
                                        xmin=np.min(X_train[:,0])-0.1,
                                        xmax=np.max(X_train[:,0])+0.1,
                                        ymin=np.min(X_train[:,1])-0.1,
                                        ymax=np.max(X_train[:,1])+0.1)
         plt.show()
```



```
In [22]: fig = plt.figure(figsize=(6,6))
         plot_decision_boundary(X_train, y_train, alpha, b,
                                xmin=np.min(X_train[:,0])-0.1,
                                xmax=np.max(X_train[:,0])+0.1,
                                ymin=np.min(X_train[:,1])-0.1,
                                ymax=np.max(X_train[:,1])+0.1)
         plt.show()
```



```
In [23]: print "Number of support vectors: " + str(support_vector_count(alpha))

Number of support vectors: 88
```

The final remark is that built-in algorithm uses a lot of optimization techniques (for example, it does not choose pair of α_i, α_j randomly but utilizes some heuristic for that). As we can see even without these techniques our algorithm gives pretty good result.

25 Generalized additive models (GAM)

In the previous lectures we talked about the generalized linear models (GLM), where we assumed that target variable y has a distribution from exponential family and the prediction is an expected value μ of this distribution. The link function g^{-1} has been introduced such that

$$g^{-1}(\mu) = f(\mu) = \theta^T x$$

(for convenience, we denoted $f = g^{-1}$). Recall that:

- if $f(\mu) = \mu$ and $y \sim N(\mu, \sigma^2)$, then the resulted model is a linear regression;
- if $f(\mu) = \ln\left(\dfrac{\mu}{1-\mu}\right)$ (or, $\mu = \dfrac{1}{1+e^{-\eta}}$) and $y \sim \mathrm{Ber}(\mu)$, then the resulted model is a logistic regression.

The generalized additive models can be introduced as a modification of GLM, we assume that

$$f(\mu) = \alpha + f_1(x_1) + f_2(x_2) + \ldots + f_n(x_n),$$

where $f_j, j = 1, \ldots, n$, are nonlinear differentiable functions, α is a constant (notice that this is nonparametric model). Consider two cases: GAM for regression and GAM for classification.

25.1 GAM for regression

We assume that $f(\mu) = \mu$ and $y \sim N(\mu, \sigma^2)$, in other words,

$$y|x;\alpha;f_j \sim N(\alpha + f_1(x_1) + f_2(x_2) + \ldots + f_n(x_n), \sigma^2).$$

We find the best value for α using maximum likelihood estimation. The log-likelihood has a form

$$l(\alpha, f_1, \ldots, f_n) = \ln \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \alpha - \sum\limits_{j=1}^{n} f_j(x_j^{(i)}))^2}{2\sigma^2} \right).$$

The derivative with respect to α is

$$l'_\alpha = \frac{1}{\sigma^2} \sum_{i=1}^{m} \left(y^{(i)} - \alpha - \sum_{j=1}^{n} f_j(x_j^{(i)}) \right) = 0.$$

Because of the term $\sum\limits_{j=1}^{n} f_j(x_j^{(i)})$, α cannot be found explicitly. Assuming the additional condition

$$\sum_{i=1}^{m} f_j(x_j^{(i)}) = 0 \text{ for any } j$$

(mean of f_j along the data is zero), we will find

$$\alpha = \frac{1}{m} \sum_{i=1}^{m} y^{(i)}$$

(average of all targets in the dataset).

The next step will be to find the estimations for f_j. If we have already found the estimations for $f_j, j \neq 1$ we will find the best estimation for f_1 (general case can be considered by analogy):

$$l(\alpha, f_j) = \ln \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \alpha - f_1(x_1^{(i)}) - \sum\limits_{j=2}^{n} f_j(x_j^{(i)}))^2}{2\sigma^2} \right)$$

Denote

$$z^{(i)} = y^{(i)} - \alpha - \sum_{j=2}^{n} f_j(x_j^{(i)}),$$

then the log-likelihood equals to

$$l(\alpha, f_j) = \ln \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(z^{(i)} - f_1(x_1^{(i)}))^2}{2\sigma^2}\right).$$

We reformulate the problem: find the best estimation for f_1 with condition

$$z|x; f_1 \sim N(f_1(x_1), \sigma^2).$$

The advantage of this formulation is that the model with only one feature should be trained. It gives a rise to the **backfitting algorithm**.

Algorithm 7 Backfitting algorithm for GAM regression

```
1: set initial values $\alpha = \frac{1}{m}\sum_{i=1}^{m} y^{(i)}$, $f_j = 0$ for all $j = 1, \dots, n$
2: repeat
3:   for $j = 1$ to $n$ do
4:     evaluate working targets $z^{(i)} = y^{(i)} - \alpha - \sum_{k=1, k\neq j}^{n} f_k(x_k^{(i)})$
5:     train model with feature $x_j$ and target $z$ to estimate $f_j$
6: until convergence
7: return $\alpha$, $f_j$
```

For the line 5 of this algorithm we should choose a single variable model. Simple nonparametric approaches could be used (for example, weighted linear regression or cubic splines).

25.2 Python implementation

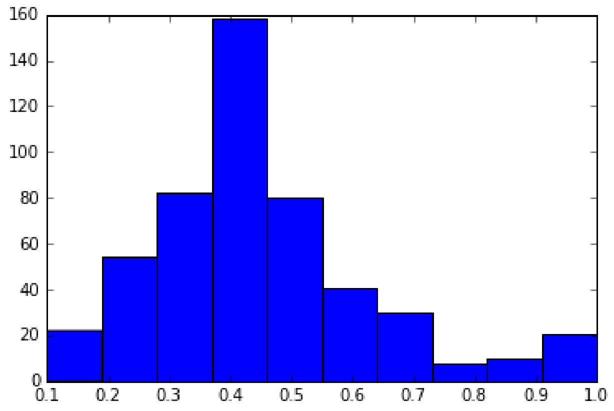
Import necessary libraries:

```
In [1]: import numpy as np
        import pandas as pd
        import random
        import math
        import sklearn.datasets as ds
        from sklearn.cross_validation import train_test_split
        from ml_metrics import rmse
        from matplotlib import cm
        import matplotlib.pyplot as plt
```

As a benchmark we will use `boston` data from the `scikit-learn` package.

```
In [2]: boston = ds.load_boston()
        X = boston.data
        y = boston.target/50.
        X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                            test_size=0.33,
                                                            random_state=42)
```

```
In [3]: print 'Number of training examples: ' + str(X.shape[0])
        print 'Number of features: ' + str(X.shape[1])

Number of training examples: 506
Number of features: 13

In [4]: plt.hist(y)
        plt.show()
```



To implement backfitting algorithm we will use cubic spline approximation as our building block algorithm (function `UnivariateSpline` from `scipy` package). Now we train generalized additive model with backfitting algorithm.

```
In [5]: from scipy.interpolate import UnivariateSpline

In [6]: # number of training examples
        m = X_train.shape[0]
        # number of features
        n = X_train.shape[1]
        # bias term for GAM
        alpha = np.mean(y_train)
        # list of trained splines
        fj = []
        # number of iterations
        niter = 100
        for it in range(niter):
            for var in range(n):
                # evaluate values for previous splines
                fj_eval = [f(X_train[:,j]) for j,f in enumerate(fj)
                           if j != var]
                # convert values to numpy array
                fj_eval = np.array(fj_eval).T.reshape((m, -1))
                # calculate working target
                z = y_train - alpha - np.sum(fj_eval, axis=1)
                # prepare var and z for UnivariateSpline:
                df = pd.DataFrame()
                df['x'] = np.round(X_train[:,var], 1)
                df['y'] = z
                df.sort_values(by='x', inplace=True)
                df = df.groupby('x')['y'].mean().reset_index()
                # fit UnivariateSpline model:
                if it == 0:
                    # add splines to fj
                    fj.append(UnivariateSpline(df.x.values,
                                               df.y.values,
                                               k=min(len(df)-1,3)))
                else:
                    # replace splines in fj
                    fj[var] = UnivariateSpline(df.x.values,
                                               df.y.values,
                                               k=min(len(df)-1,3))
```

The result will be the set of cubic splines for each variable saved in the list `fj`. Now we check the accuracy on the test set:

```
In [7]: # evaluate spline values for test points
        fj_eval = [f(X_test[:,j]) for j,f in enumerate(fj)]
        # convert spline values to numpy array
        fj_eval = np.array(fj_eval).T.reshape((X_test.shape[0], -1))
        # calculate predictions
        y_pred = alpha + np.sum(fj_eval, axis=1)
        # print accuracy
        print rmse(y_test, y_pred)
```

```
0.100600863408
```

25.3 GAM for classification

For the classification we assume $f(\mu) = \ln\left(\frac{\mu}{1-\mu}\right)$ and Bernoulli distribution for the target y, in other words:

$$y|x;\alpha;f_j \sim \text{Ber}(g(\alpha + f_1(x_1) + f_2(x_2) + \ldots + f_n(x_n))),$$

where $g(z) = \frac{1}{1+e^{-z}}$ is a sigmoid function. Denote

$$\eta = \alpha + f_1(x_1) + f_2(x_2) + \ldots + f_n(x_n),$$

then the log-likelihood can be written as

$$l(\eta) = \sum_{i=1}^{m} y^{(i)} \ln \mu^{(i)} + (1 - y^{(i)}) \ln(1 - \mu^{(i)}).$$

To find the initial value for α we assume that $f_j = 0$ for all $j = 1, \ldots, n$:

$$l'_\eta = \sum_{i=1}^{m} y^{(i)}(1-\mu^{(i)}) - (1-y^{(i)})\mu^{(i)} = 0 \Rightarrow \alpha = \ln\left(\frac{\mu}{1-\mu}\right),$$

where

$$\mu = \frac{1}{m}\sum_{i=1}^{m} y^{(i)}.$$

To find f_j we will use the following procedure. First, we use Newton's method with respect to η:

$$\eta^{new} = \eta^{old} - \frac{l'_\eta(\eta^{old})}{l''_\eta(\eta^{old})} \text{(for each training example).}$$

But

$$\begin{aligned} l'(\eta) &= y(1-\mu) - (1-y)\mu = y - \mu, \\ l''(\eta) &= -\mu(1-\mu) \end{aligned}$$

(remember that $\mu = g(\eta)$). Then the updating formula for η is

$$\eta^{new} := \eta^{old} + \frac{y - \mu^{old}}{\mu^{old}(1-\mu^{old})}.$$

The second step will be to use backfitting algorithm for GAM regression (previous section) with target values η^{new}.

The important difference between linear case and logistic case is that in the linear case we assumed that variance σ^2 for all training examples is constant. In the logistic case the variance for the Bernoulli distribution is calculated as $\mu(1-\mu)$. It means that we should normalize variances for the training examples to the same value; it can be easily done by using weights $\mu(1-\mu)$ in backfitting algorithm (**weighted backfitting algorithm**):

Algorithm 8 GAM for classification with weighted backfitting algorithm

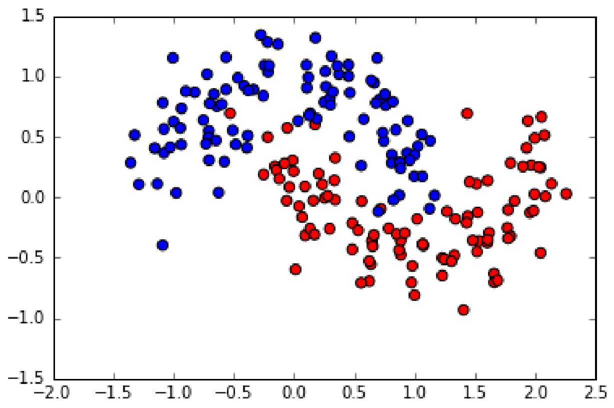
```
1: set initial values $\mu = \frac{1}{m}\sum_{i=1}^{m} y^{(i)}$, $\alpha = \ln\left(\frac{\mu}{1-\mu}\right)$, $f_j = 0$, $j = 1, \ldots, n$
2: repeat
3:     evaluate $\eta^{(i)} = \alpha + \sum_{j=1}^{n} f_j(x^{(i)})$ and $\mu^{(i)} = \frac{1}{1+e^{-\eta^{(i)}}}$
4:     make the Newton's step $\eta^{(i)} := \eta^{(i)} + \frac{y^{(i)} - \mu^{(i)}}{\mu^{(i)}(1-\mu^{(i)})}$
5:     evaluate weights $w^{(i)} = \mu^{(i)}(1-\mu^{(i)})$
6:     evaluate new value for $\alpha = \frac{1}{m}\sum_{i=1}^{m} \eta^{(i)}$
7:     for $j = 1$ to $n$ do
8:         evaluate working targets $z^{(i)} = \eta^{(i)} - \alpha - \sum_{k=1,k\neq j}^{n} f_k(x_k^{(i)})$
9:         train model with feature $x_j$, target $z$ and weights $w$ to estimate $f_j$
10: until convergence
11: return $\alpha$, $f_j$
```

As before for the step 9 of the algorithm we can choose some nonparametric model, for example, cubic spline fitting or weighted linear regression.

25.4 Python implementation

For classification we will use `make_moons` data from the `scikit-learn` package.

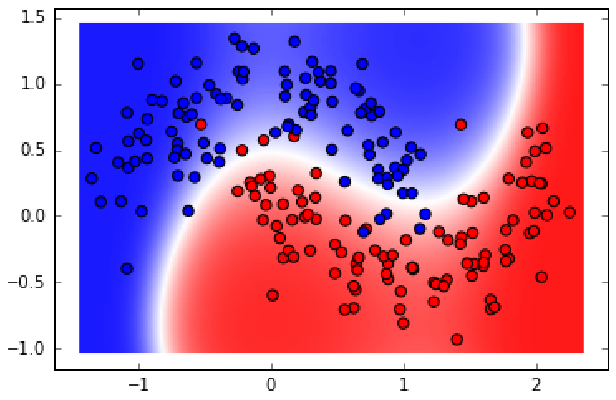
```
In [8]: np.random.seed(0)
        X_train, y_train = ds.make_moons(200, noise=0.20)
        plt.scatter(X_train[:,0], X_train[:,1],
                    s=40, c=y_train, cmap=cm.bwr)
        plt.show()
```



Now we implement GAM for logistic regression and weighted backfitting algorithm.

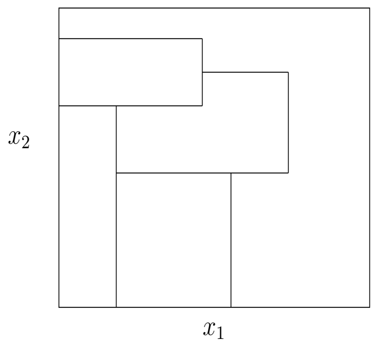
```
In [9]: # number of training examples
        m = X_train.shape[0]
        # number of features
        n = X_train.shape[1]
        # list of trained splines
        fj = []
        # number of iterations
        niter = 100
        for it in range(niter):
            y_mean = np.mean(y_train)
            alpha = np.log(y_mean/(1-y_mean))
            if it == 0:
                eta = np.array([alpha]*m)
            else:
                fj_eval = np.array([f(X_train[:,j])
                                    for j,f in enumerate(fj)
                                    if j != var]).T.reshape((m, -1))
                eta = np.array([alpha]*m) + np.sum(fj_eval, axis=1)
            mu = 1.0/(1 + np.exp(-1.0*eta))
            z = eta + (y_train - mu)/(mu*(1-mu))
            w = np.multiply(mu, 1.0-mu)
            alpha_z = np.mean(z)
            for var in range(n):
                fj_eval = np.array([f(X_train[:,j])
                                    for j,f in enumerate(fj)
                                    if j != var]).T.reshape((m, -1))
                zz = z - alpha_z - np.sum(fj_eval, axis=1)
                df = pd.DataFrame()
                df['x'] = np.round(X_train[:,var], 1)
                df['y'] = zz
                df['w'] = w
                df.sort_values(by='x', inplace=True)
                df = df.groupby('x')[['y', 'w']].mean().reset_index()
                # fit UnivariateSpline model:
                if it == 0:
                    # add splines to fj
                    fj.append(UnivariateSpline(df.x.values,
                                               df.y.values,
                                               df.w.values,
                                               k=min(len(df)-1,3)))
                else:
                    # replace splines in fj
                    fj[var] = UnivariateSpline(df.x.values,
                                               df.y.values,
                                               df.w.values,
                                               k=min(len(df)-1,3))
            alpha = alpha_z

In [10]: def plot_decision_boundary(X, y, alpha, fj,
                                    xmin=-0.1, xmax=1.1,
                                    ymin=-0.1, ymax=1.1):
             fig = plt.figure(figsize=(6,6))
             x1, x2 = np.meshgrid(np.linspace(xmin, xmax, 200),
                                  np.linspace(ymin, ymax, 200))
             Xt = np.c_[x1.ravel(), x2.ravel()]
             fj_eval = np.array([f(Xt[:,j])
                                 for j,f in enumerate(fj)]).T.reshape((Xt.shape[0], -1))
             ypred = 1.0/(1.0 + np.exp(-alpha - np.sum(fj_eval, axis=1)))
             ypred = ypred.reshape(x1.shape)
             extent = xmin, xmax, ymin, ymax

             plt.imshow(ypred, cmap=cm.bwr, alpha=.9, interpolation='bilinear',
                        extent = extent, origin='lower')
             plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=cm.bwr)
```

```
In [11]: plot_decision_boundary(X_train, y_train, alpha, fj,
                                xmin=np.min(X_train[:,0])-0.1,
                                xmax=np.max(X_train[:,0])+0.1,
                                ymin=np.min(X_train[:,1])-0.1,
                                ymax=np.max(X_train[:,1])+0.1)
```

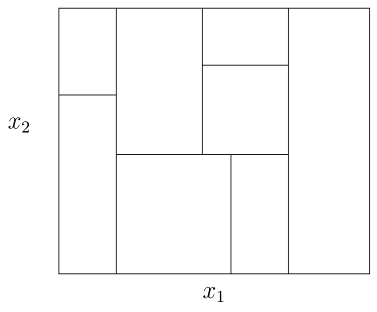


26 Tree-based methods

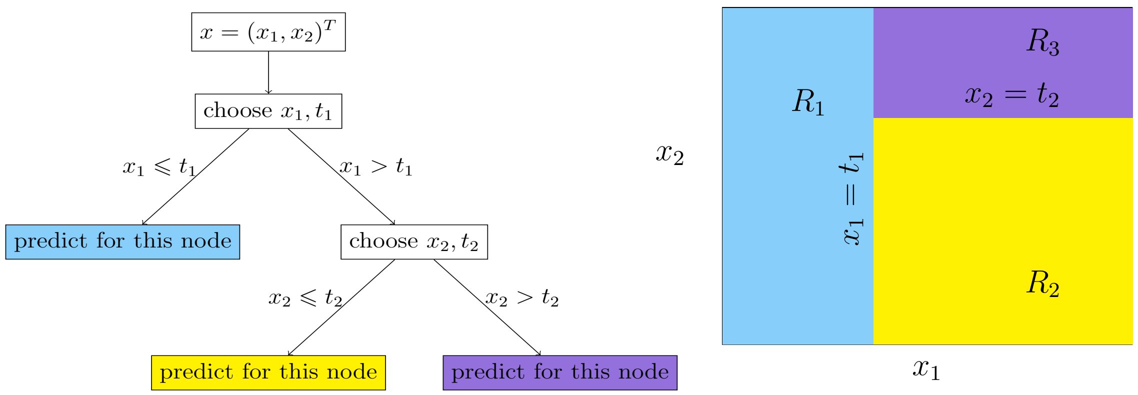
Another nonparametric approach utilizes the structure called decision tree. In many cases we should split the feature space in small regions and predict the target variable y for each region separately. The example of splitting is shown on the following picture:



Unfortunately, it is difficult to describe such splitting analytically. We consider simpler type of partition - binary partitions:



We can describe the last splitting using **decision trees**. The methods that utilize the decision trees are called tree-based methods. Consider the problem with two features $x = (x_1, x_2)^T$, the simplest decision tree with according binary partition could look like this:



Usually, the prediction is constant for each **terminal node** or **leaves** (coloured nodes on the picture). The advantage of the decision tree structure is that the hypothesis function $h_{c,t}(x)$ can be expressed as a linear combination of indicator functions. For the previous example, the hypothesis can be written as

$$h_{c,t}(x) = c_1 \cdot \mathbb{1}\{x_1 \leqslant t_1\} + c_2 \cdot \mathbb{1}\{x_1 > t_1, x_2 \leqslant t_2\} + c_3 \cdot \mathbb{1}\{x_1 > t_1, x_2 > t_2\}.$$

In this model we have 5 parameters: c_1, c_2, c_3 are predictions for the terminal nodes, t_1, t_2 are splitting points.

In general case if we have n features $x = (x_1, \dots, x_n)^T$, the hypothesis function can be written as

$$h_{c,t}(x) = \sum_{k=1}^{K} c_k \mathbb{1}\{x \in R_k\}.$$

Here K is a number of terminal nodes and R_k is the region of the feature space that corresponds to the terminal node k. The main advantage of this model is interpretability.

26.1 Regression trees

We consider in details how to build the decision trees for regression problems. Assuming that the prediction c_k for each region R_k is constant it is easy to prove that the best prediction would be the average of $y^{(i)}$ for the training examples in this region.

Indeed, we could use the probabilisitic approach in this case. Consider the simplest situation when we have one feature and one target. If the variable x is splitted by the number c and μ_1 is a constant prediction for $x < c$, μ_2 is a constant prediction for $x \geqslant c$, then

$$y|x, c, \mu_1, \mu_2 \sim N((\mu_2 - \mu_1)\mathbb{1}\{x \geqslant c\} + \mu_1, \sigma^2).$$

The log-likelihood

$$l(\mu_1, \mu_2, c) = m \ln \frac{1}{\sqrt{2\pi}\sigma} - \sum_{i=1}^{m} \frac{(y^{(i)} - (\mu_2 - \mu_1)\mathbb{1}\{x^{(i)} \geqslant c\} - \mu_1)^2}{2\sigma^2}.$$

Derivatives with respect to μ_1 and μ_2 give

$$\mu_1 = \frac{\sum\limits_{i=1}^{m} y^{(i)} \mathbb{1}\{x^{(i)} < c\}}{\sum\limits_{i=1}^{m} \mathbb{1}\{x^{(i)} < c\}},$$
$$\mu_2 = \frac{\sum\limits_{i=1}^{m} y^{(i)} \mathbb{1}\{x^{(i)} \geqslant c\}}{\sum\limits_{i=1}^{m} \mathbb{1}\{x^{(i)} \geqslant c\}}.$$

Differentiation with respect to c becomes more complicated, because the indicator function $\mathbb{1}$ is discontinuous. Another problem appears as in majority of cases there are more than one feature, and we should build log-likelihood with respect to all features. It becomes computationally infeasible to optimize the error function for the hypothesis $h_{c,t}(x) = \sum\limits_{k=1}^{r} c_k \mathbb{1}\{x \in R_k\}$ explicitly.

To resolve this problem we introduce the measure that will identify the best variable and the best split

$$Q_k(T) = \frac{1}{m_k} \sum_{i \in R_k} (y^{(i)} - \mu_k)^2,$$

where m_k is a number of training examples in the node R_k, and apply greedy algorithm that optimizes the tree step by step (not the hypothesis in general).

Algorithm 9 Greedy optimization algorithm for decision trees

```
1: initialize root node $R_1 = \mathbb{R}^n$, where $n$ is a number of features
2: initialize a list of terminal nodes $R = \{R_1\}$
3: repeat
4:   for each region $R_k$ from $R$ do
5:     for $j = 1$ to $n$ do
6:       for all splitting points $c$ do
7:         define
               $R_{kl} = \{x^{(i)} \in R_k \mid x_j^{(i)} < c\},$
               $R_{kr} = \{x^{(i)} \in R_k \mid x_j^{(i)} \geqslant c\}$
8:         evaluate $\mu_1 = \frac{1}{m_{kl}} \sum_{R_{kl}} y^{(i)}$, $\mu_2 = \frac{1}{m_{kr}} \sum_{R_{kr}} y^{(i)}$
9:         evaluate $\varepsilon = m_{kl} Q_{kl}(T) + m_{kr} Q_{kr}(T)$
10:    choose $j, c = \arg\min_{j,c} \varepsilon$
11:    remove $R_k$ from $R$; add $R_{kl}$, $R_{kr}$ to $R$
12: until convergence
13: return list of terminal nodes $R$
```

This algorithm when run till the end gives a huge decision tree with small number of training examples in the terminal nodes. Obviously, that the resulted error on the training set will be equal to zero. Remember that we called such situation overfitting (excellent predictions on the training set and very bad prediction on the test set). To avoid such situation we can introduce additional stopping criteria:

- minimal size of terminal node: for example, we can require to have at least 10 training examples in each terminal node
- minimal change of error: for example, we can require not to split the node in case if the error does not decrease more than 0.01

Then the above algorithm should be modified by adding one of these conditions (or both) before the lines 10 and 11.

Usually, the decision tree is constructed using two steps. First, we build the maximal decision tree and second, we start pruning the tree by removing terminal nodes one by one. The procedure of prunning is stopped when the following function reaches the minimal value:

$$C_\alpha(T) = \sum_{k=1}^{K} m_k Q_k(T) + \alpha K.$$

This function creates some trade-off between the tree size and its goodness of fit to the data. If α is zero then K increases till the errors $Q_k = 0$ for all k. But if $\alpha > 0$, then big value of K implies the big value for the second term in $C_\alpha(T)$, for example, if there exists big α such that the minimal value of C_α is obtained if $K = 1$. Usually, α is defined using cross-validation procedure.

26.2 Python implementation

First, we use the built-in implementation of the decision tree in Python.

```
In [12]: from sklearn.tree import DecisionTreeRegressor, export_graphviz
         from graphviz import Source
         import pydot
```

As a benchmark we use `boston` dataset from the `scikit-learn` package.

```
In [13]: boston = ds.load_boston()
         X = boston.data
         y = boston.target/50. # just for convenience
         X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                             test_size=0.33,
                                                             random_state=42)
```

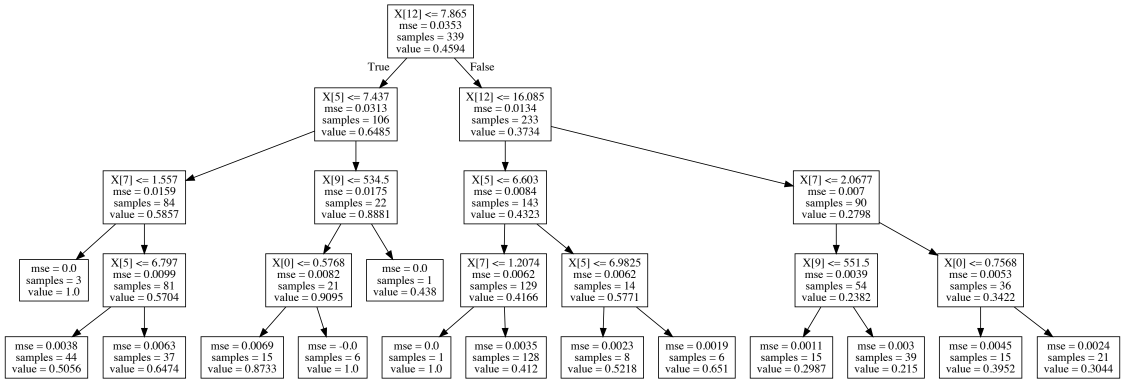
Fit the model (notice that we use parameter `max_depth` to restrict the tree size):

```
In [14]: clf = DecisionTreeRegressor(max_depth=4)
         clf = clf.fit(X_train, y_train)
```

And visualize the resulted tree:

```
In [15]: # produces dot file
         export_graphviz(clf, out_file='regression_tree_scikit.dot')
         dot = open('regression_tree_scikit.dot').read()
         dot = dot.replace('node [shape=box] ;',
              'node [shape=box] ;\ngraph [size="10.3,10.3!"];')
         dot = Source(dot)
         dot
```

```
Out[15]:
```



The predictions can be obtained as follows:

```
In [16]: y_pred = clf.predict(X_test)
         print rmse(y_test, y_pred)
```

```
0.0771597318202
```

We want to show a simple implementation of decision tree algorithm from scratch. There are different possibilities to implement the decision tree in Python. We will create two classes for node and tree, accordingly. `Node` class contains the description of each node and pointer to father and daughter nodes, and `RegressionTree` class contains all nodes and functions for splitting and predicton. We have added the function that saves tree structure in the dot file.

```
In [17]: class Node():
             def __init__(self, id, samples, prediction,
                          error, father_node=None):
                 if father_node is None: # the node is a root node
                     self.depth = 0
                 else:
                     self.depth = father_node.depth + 1
                 self.father_node = father_node
                 self.id = id
```

```
            self.samples = np.array(samples)
            self.prediction = prediction
            self.error = error
            self.j = None
            self.t = None

    class RegressionTree():
        def __init__(self, X, y, max_depth=5, min_samples_leaf = 1):
            self.X = X
            self.y = y
            self.max_depth = max_depth
            self.min_samples_leaf = min_samples_leaf
            self.nodes = []
            mu, err = self.error(y)
            self.nodes.append(Node(0, [x for x in range(X.shape[0])],
                                   mu, err))
            self.nodes_to_split = [0]

        def error(self, y):
            p = np.mean(y)
            return p, np.mean(np.power(y-p, 2))

        def split_next_node(self):
            if len(self.nodes_to_split) == 0:
                return
            node = self.nodes[self.nodes_to_split[0]]
            del self.nodes_to_split[0]
            if (node.depth >= self.max_depth or
                len(node.samples)<=2*self.min_samples_leaf-1):
                node.left = None
                node.right = None
                return
            best_j = -999
            best_t = -999
            best_se1 = node.error*len(node.samples)
            best_se2 = node.error*len(node.samples)
            best_mu1 = -999
            best_mu2 = -999
            for j in range(self.X.shape[1]):
                Xj = self.X[node.samples,j]
                Xj_min = np.min(Xj)
                Xj_max = np.max(Xj)
                for t in np.linspace(Xj_min, Xj_max, 300):
                    y_left = self.y[node.samples][Xj<t]
                    y_right = self.y[node.samples][Xj>=t]
                    if (len(y_left)<self.min_samples_leaf or
                        len(y_right)<self.min_samples_leaf):
                        continue
                    mu1, mse1 = self.error(y_left)
                    mu2, mse2 = self.error(y_right)
                    se1 = mse1*len(y_left)
                    se2 = mse2*len(y_right)
                    if (se1 + se2) < (best_se1 + best_se2):
                        best_j = j
                        best_t = t
                        samples_left = node.samples[Xj<t]
                        samples_right = node.samples[Xj>=t]
                        best_mu1 = mu1
                        best_mu2 = mu2
                        best_se1 = se1
                        best_se2 = se2
                        best_mse1 = mse1
                        best_mse2 = mse2
            if best_j == -999:
                node.left = None
                node.right = None
            else:
                node.left = Node(len(self.nodes), samples_left,
                                 best_mu1, best_mse1, node)
                node.right = Node(len(self.nodes)+1, samples_right,
                                  best_mu2, best_mse2, node)
                node.j = best_j
                node.t = best_t
                self.nodes.append(node.left)
                self.nodes.append(node.right)
                self.nodes_to_split.append(len(self.nodes)-2)
                self.nodes_to_split.append(len(self.nodes)-1)

        def predict(self, X):
            preds = []
            for i in range(X.shape[0]):
                node = self.nodes[0]
                while True:
                    if node.j is None:
                        preds.append(node.prediction)
                        break
                    if X[i, node.j] < node.t:
                        node = node.left
                    else:
                        node = node.right
            return preds

        def save_tree_to_dot(self, filename):
            def get_edge(x):
                if x.j is None:
                    label = 'mse = ' + str(np.round(x.error,3)) +\
                            '\n samples = ' + str(len(x.samples)) +\
                            '\n value = ' + str(np.round(x.prediction,3))
                else:
                    label = 'X[' + str(x.j) + '] < ' + str(np.round(x.t)) +\
                            '\n mse = ' + str(np.round(x.error,3)) +\
                            '\n samples = ' + str(len(x.samples)) +\
                            '\n value = ' + str(np.round(x.prediction,3))
                if x.father_node is None:
                    return str(x.id) + ' [ label="' + label + '"] ;\n'
                else:
                    return str(x.id) + ' [ label="' + label + '"] ;\n' +\
                                     str(x.father_node.id) +\
                                     ' -> ' + str(x.id) + ' ;\n'

            f = open(filename, 'w')
            f.write('digraph Tree {\n node [shape = box] ;\n')
            f.write(' graph [size="9.7,8.3!"];\n')

            for x in self.nodes:
                f.write(get_edge(x))
            f.write('}')
            f.close()
```

```
In [18]: tree = RegressionTree(X_train, y_train,
                               max_depth=4, min_samples_leaf = 4)
         while len(tree.nodes_to_split)>0:
             tree.split_next_node()
         tree.save_tree_to_dot('regression_tree_manual.dot')
```

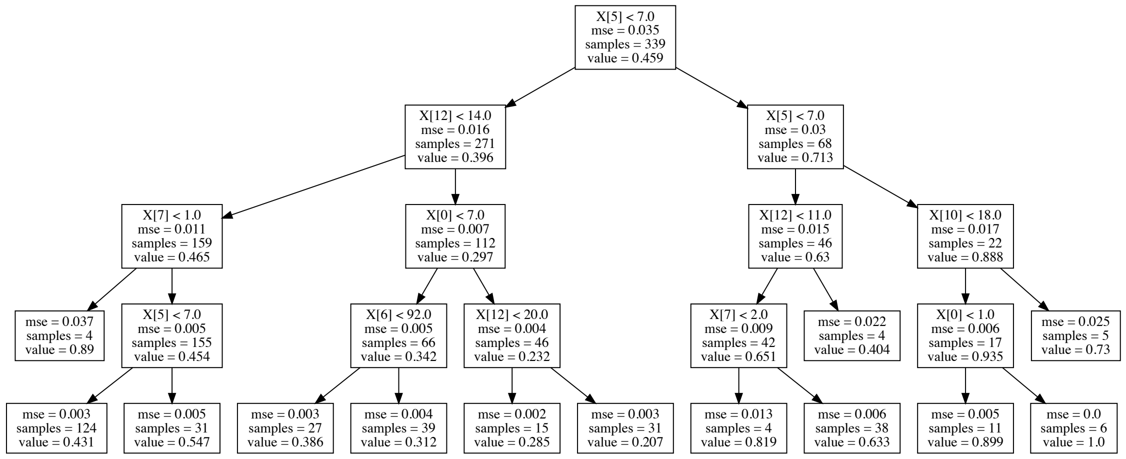
Check the error on the test set:

```
In [19]: rmse(y_test, tree.predict(X_test))

Out[19]: 0.07209792520517698
```

We can also draw the tree from the dot file (remember that we saved tree in the dot file already):

```
In [20]: dot = Source(open('regression_tree_manual.dot').read())
         dot
```

```
Out[20]:
```



26.3 Classification trees

The main difference between regression and classification decision trees is the way to define the quality of split. If we have several classes $1, 2, \ldots, D$, then it does not make sense to calculate the mean squared error $Q_k(T)$ like we did for the regression trees. Consider the regions $R_1, R_2, \ldots, R_K$, obtained by the decision tree. We could evaluate the quantities

$$p_{kd} = \frac{1}{m_k} \sum_{i \in R_k} \mathbb{1}\{y^{(i)} = d\} \text{ (percentage of class } d \text{ in the region } R_k),$$

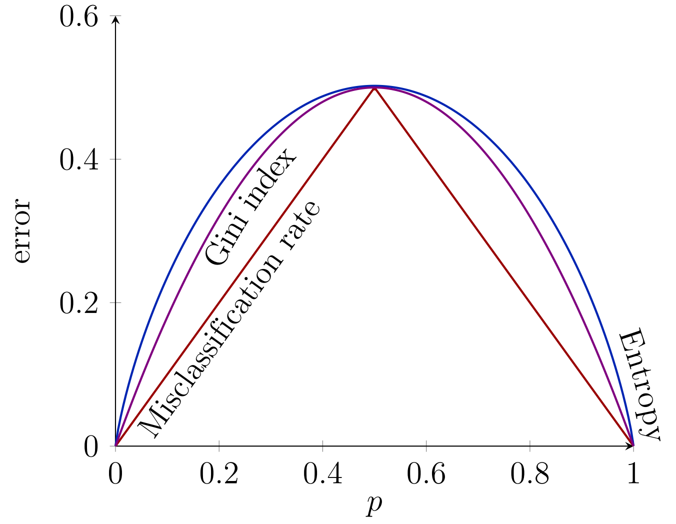
where as before m_k is a number of training examples in the region R_k, and $d \in \{1, 2, \ldots, D\}$. There are several criteria for quality of split can be used:

- **Misclassification error**: denote $d(k) = \arg\max_d p_{kd}$ (the majority class in the region R_k), then
$$Q_k(T) = \frac{1}{m_k} \sum_{i \in R_k} \mathbb{1}\{y^{(i)} \neq d(k)\} = 1 - p_{k\,d(k)}.$$
- **Gini index**:
$$Q_k(T) = \sum_{d=1}^{D} p_{kd}(1 - p_{kd}).$$
- **Cross-entropy (or deviance)**:
$$Q_k(T) = -\sum_{d=1}^{D} p_{kd} \ln p_{kd}.$$

For example, if we have two classes only $(D = 2)$ and $p_{k1} = p$, then

- Misclassification error: $\min(p, 1 - p)$

- Gini index: $2p(1-p)$
- Cross-entropy: $-p\ln p - (1-p)\ln(1-p)$

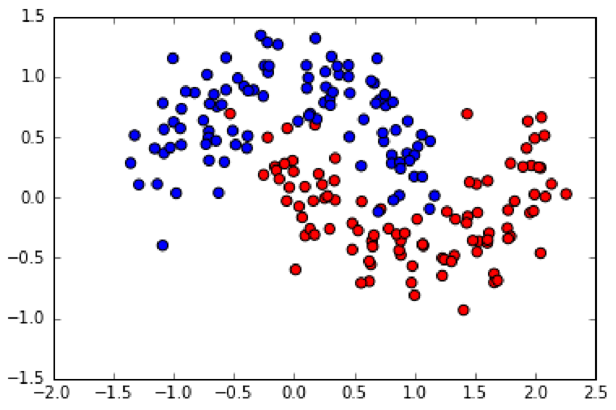


To build the classification decision tree we should choose the error $Q_k(T)$ and run the Algorithm 3.

26.4 Python implementation

As a benchmark we use `make_moons` data:

```
In [21]: np.random.seed(0)
         X, y = ds.make_moons(200, noise=0.20)
         plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=cm.bwr)
         X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                             test_size=0.33,
                                                             random_state=42)
```

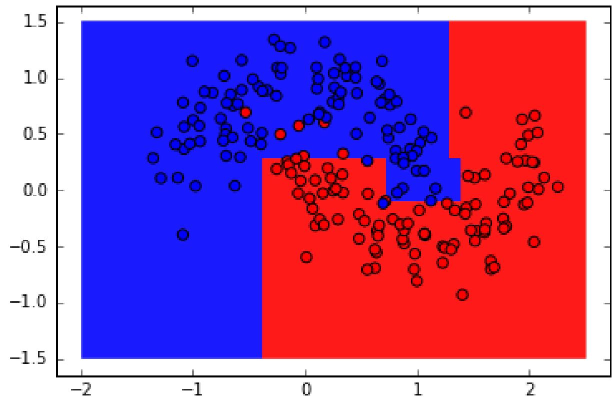


The following function `plot_decision_boundary()` visualizes the result of our classification:

```
In [22]: def plot_decision_boundary(model, xmin=-0.1, xmax=1.1,
                                    ymin=-0.1, ymax=1.1):
             fig = plt.figure(figsize=(6,6))
             x1, x2 = np.meshgrid(np.linspace(xmin, xmax, 500),
                                  np.linspace(ymin, ymax, 500))
             ypred = model.predict(np.c_[x1.ravel(), x2.ravel()])
             ypred = ypred.reshape(x1.shape)
             extent = xmin, xmax, ymin, ymax

             plt.imshow(ypred, cmap=cm.bwr, alpha=.9,
                        interpolation='bilinear',
                        extent = extent, origin='lower')
             plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=cm.bwr)
```

The `scikit-learn` has its own implementation of classification decision trees:

```
In [23]: from sklearn.tree import DecisionTreeClassifier

In [24]: clf = DecisionTreeClassifier(max_depth=10,
                                      min_samples_leaf=4)
         clf = clf.fit(X_train, y_train)

In [25]: plot_decision_boundary(clf, xmin=-2.0, xmax=2.5,
                                ymin=-1.5, ymax=1.5)
```



The main difference between regression and classification decision trees is the metrics used to estimate the quality of split. In regression decision tree we used a mean squared error, in classification decision tree we will use Gini index, cross-entropy or misclassification rate:

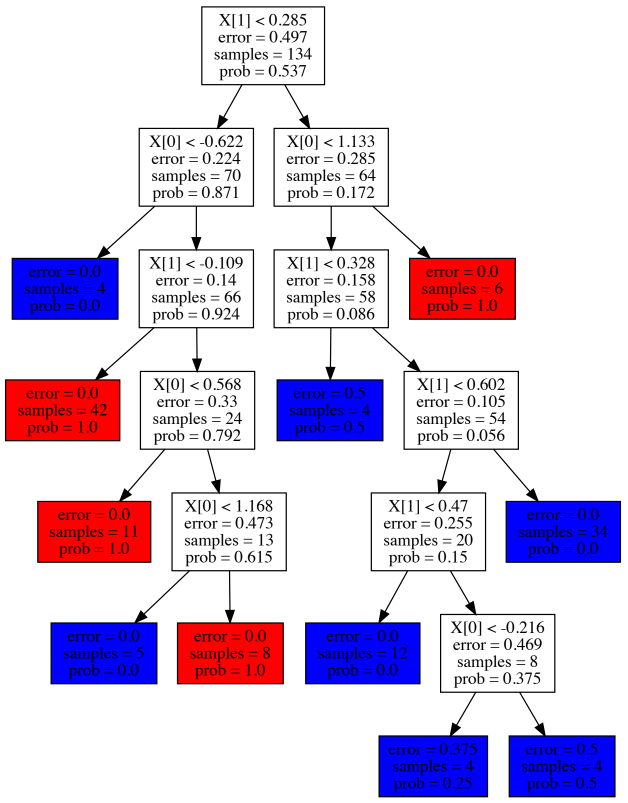
```
In [26]: class ClassificationTree():
             def __init__(self, X, y, error_type='gini',
                          max_depth=5, min_samples_leaf = 1):
                 self.X = X
```

```
            self.y = y
            self.error_type = error_type
            self.max_depth = max_depth
            self.min_samples_leaf = min_samples_leaf
            self.nodes = []
            mu, err = self.error(y)
            self.nodes.append(Node(0, [x for x in range(X.shape[0])],
                                   mu, err))
            self.nodes_to_split = [0]

        def error(self, y):
            p = np.mean(y)
            if self.error_type == 'entropy':
                p = np.clip(p, 0.01, 0.99)
                return p, -1.0*p*np.log(p)-(1-p)*np.log(1-p)
            elif self.error_type == 'gini':
                return p, 2.0*p*(1.0-p)
            elif self.error_type == 'misclass':
                return p, min(p, 1-p)

        def split_next_node(self):
            if len(self.nodes_to_split) == 0:
                return
            node = self.nodes[self.nodes_to_split[0]]
            del self.nodes_to_split[0]
            if (node.depth >= self.max_depth
                or len(node.samples)<=2*self.min_samples_leaf-1
                or node.error == 0.0):
                node.left = None
                node.right = None
                return
            best_j = -999
            best_t = -999
            best_weighted_err1 = node.error*len(node.samples)
            best_weighted_err2 = node.error*len(node.samples)
            best_mu1 = -999
            best_mu2 = -999
            for j in range(self.X.shape[1]):
                Xj = self.X[node.samples,j]
                Xj_min = np.min(Xj)
                Xj_max = np.max(Xj)
                for t in np.linspace(Xj_min, Xj_max, 300):
                    y_left = self.y[node.samples][Xj<t]
                    y_right = self.y[node.samples][Xj>=t]
                    if (len(y_left)<self.min_samples_leaf
                        or len(y_right)<self.min_samples_leaf):
                        continue
                    mu1, err1 = self.error(y_left)
                    mu2, err2 = self.error(y_right)
                    weighted_err1 = err1*len(y_left)
                    weighted_err2 = err2*len(y_right)
                    if (weighted_err1 + weighted_err2 <
                        best_weighted_err1 + best_weighted_err2):
                        best_j = j
                        best_t = t
                        samples_left = node.samples[Xj<t]
                        samples_right = node.samples[Xj>=t]
                        best_mu1 = mu1
                        best_mu2 = mu2
                        best_err1 = err1
                        best_err2 = err2
                        best_weighted_err1 = weighted_err1
                        best_weighted_err2 = weighted_err2
            if best_j == -999:
                node.left = None
                node.right = None
            else:
                node.left = Node(len(self.nodes), samples_left,
                                 best_mu1, best_err1, node)
                node.right = Node(len(self.nodes)+1, samples_right,
                                  best_mu2, best_err2, node)
                node.j = best_j
                node.t = best_t
                self.nodes.append(node.left)
                self.nodes.append(node.right)
                self.nodes_to_split.append(len(self.nodes)-2)
                self.nodes_to_split.append(len(self.nodes)-1)

        def predict(self, X, probability = False):
            preds = []
            for i in range(X.shape[0]):
                node = self.nodes[0]
                while True:
                    if node.j is None:
                        preds.append(node.prediction)
                        break
                    if X[i, node.j] < node.t:
                        node = node.left
                    else:
                        node = node.right
            if probability == False:
                preds = np.round(preds)
            return np.array(preds)

        def save_tree_to_dot(self, filename):
            def get_edge(x):
                if x.j is None:
                    label = 'error = ' + str(np.round(x.error,3)) +\
                            '\n samples = ' + str(len(x.samples)) +\
                            '\n prob = ' + str(np.round(x.prediction,3))
                    if x.prediction > 0.5:
                        color = 'red'
                    else:
                        color = 'blue'
                else:
                    label = 'X['+str(x.j)+'] < '+str(np.round(x.t,3))+\
                            '\n error = '+str(np.round(x.error,3))+\
                            '\n samples = '+str(len(x.samples))+\
                            '\n prob = '+str(np.round(x.prediction,3))
                    color = 'white'
                if x.father_node is None:
                    return str(x.id)+' [ style=filled fillcolor='+color+\
                           ' label="'+label+'"] ;\n'
                else:
                    return str(x.id)+' [ style=filled fillcolor='+color+\
                           ' label="'+label+'"] ;\n'+\
                           str(x.father_node.id)+' -> '+str(x.id)+' ;\n'

            f = open(filename, 'w')
            f.write('digraph Tree {\n node [shape = box] ;\n')
            for x in tree.nodes:
                f.write(get_edge(x))
            f.write('}')
            f.close()
```

```
In [27]: tree = ClassificationTree(X_train, y_train,
                                   error_type='gini',
                                   max_depth=10, min_samples_leaf = 4)
         while len(tree.nodes_to_split)>0:
             tree.split_next_node()
         tree.save_tree_to_dot('classification_tree_manual.dot')
```

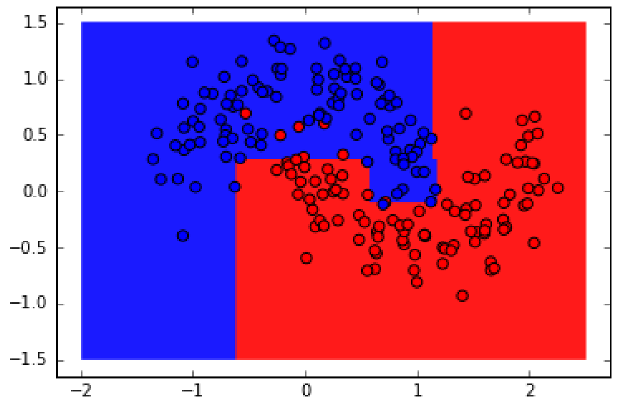
```
In [28]: dot = Source(open('classification_tree_manual.dot').read())
         dot
```

```
Out[28]:
```

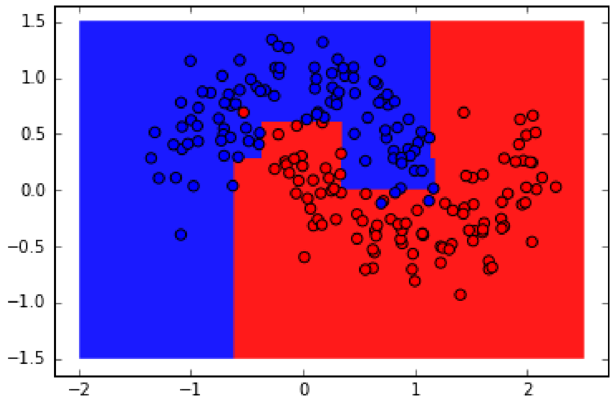


Compare the decision boundaries for different error types:

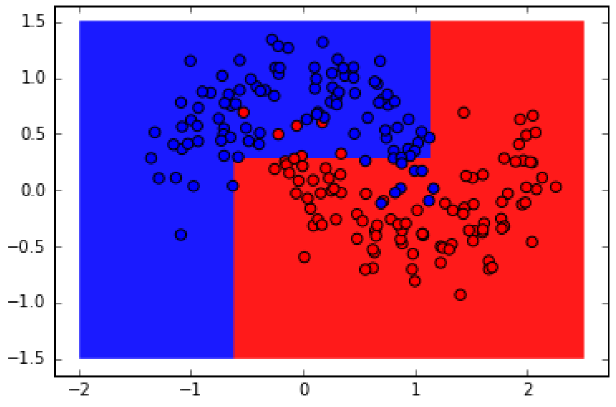
```
In [29]: tree = ClassificationTree(X_train, y_train,
                                    error_type='gini', max_depth=10,
                                    min_samples_leaf = 4)
         while len(tree.nodes_to_split)>0:
             tree.split_next_node()
         plot_decision_boundary(tree,
                                xmin=-2.0, xmax=2.5,
                                ymin=-1.5, ymax=1.5)
```



```
In [30]: tree = ClassificationTree(X_train, y_train,
                                   error_type='entropy', max_depth=10,
                                   min_samples_leaf = 4)
         while len(tree.nodes_to_split)>0:
             tree.split_next_node()
         plot_decision_boundary(tree,
                                xmin=-2.0, xmax=2.5,
                                ymin=-1.5, ymax=1.5)
```



```
In [31]: tree = ClassificationTree(X_train, y_train,
                                   error_type='misclass', max_depth=10,
                                   min_samples_leaf = 4)
         while len(tree.nodes_to_split)>0:
             tree.split_next_node()
         plot_decision_boundary(tree,
                                xmin=-2.0, xmax=2.5,
                                ymin=-1.5, ymax=1.5)
```



27 Boosting

27.1 Exponential loss

The natural generalization for the additive models can be obtained by

$$\eta = \sum_{k=1}^{K} \beta_k b_k(x, \theta_k). \tag{27.1}$$

For classification problems as before we find the probability by applying the sigmoid function:

$$\mu = g\left(\sum_{k=1}^{K} \beta_k b_k(x, \theta_k)\right) = g(\eta),$$

where

$$g(z) = \frac{1}{1 + e^{-2z}}$$

(the coefficient 2 is introduced for convenience).

As before we assume

$$y \mid \beta_k; b_k; \theta_k \sim \operatorname{Ber}(\mu),$$

and log-likelihood is expressed as:

$$l(\beta_k, b_k, \theta_k) = \sum_{i=1}^{m} y^{(i)} \ln \mu^{(i)} + (1 - y^{(i)}) \ln(1 - \mu^{(i)}).$$

Consider the function

$$h(\mu) = y \ln \mu + (1 - y) \ln(1 - \mu).$$

We introduce the transformed target by $z = 2y - 1 \Leftrightarrow y = \dfrac{z+1}{2}$ (with this notation if $y \in \{0, 1\}$, then $z \in \{-1, 1\}$), then

$$\begin{aligned} h(\mu) &= \frac{z+1}{2} \ln \frac{1}{1 + e^{-2\eta}} + \left(1 - \frac{z+1}{2}\right) \ln \left(1 - \frac{1}{1 + e^{-2\eta}}\right) = \\ &= \frac{1}{2} \left(-z \ln(1 + e^{-2\eta}) - \ln(1 + e^{-2\eta}) - 2\eta + 2z\eta - \ln(1 + e^{-2\eta}) + z \ln(1 + e^{-2\eta})\right) = \\ &= -(\ln(1 + e^{-2\eta}) + \eta(1 - z)) = -\ln\left((1 + e^{-2\eta}) e^{\eta(1-z)}\right) = -\ln\left(1 + e^{-2z\eta}\right). \end{aligned}$$

In the last transition we use the fact that $z \in \{-1, 1\}$:

$$\begin{aligned} &\text{if } z = 1, \text{ then } \ln\left((1 + e^{-2\eta}) e^{\eta(1-z)}\right) = \ln\left(1 + e^{-2\eta}\right) = \ln\left(1 + e^{-2z\eta}\right), \\ &\text{if } z = -1, \text{ then } \ln\left((1 + e^{-2\eta}) e^{\eta(1-z)}\right) = \ln\left(e^{2\eta} + 1\right) = \ln\left(1 + e^{-2z\eta}\right). \end{aligned}$$

With the obtained expression for the likelihood we can formulate the maximum likelihood optimization problem in the simpler way:

$$\arg \max_{\beta_k, b_k, \theta_k} l(\beta_k, b_k, \theta_k) = \arg \min_{\beta_k, b_k, \theta_k} e^{-z\eta},$$

where η is defined with (27.1) and $z = 2y - 1$ is transformed target. The function

$$L(z, \eta) = \sum_{i=1}^{m} \exp\left(-z^{(i)}\eta^{(i)}\right)$$

is called an **exponential loss**.

27.2 Adaboost

Boosting is the example of algorithm ensembling. The idea of ensembling is to combine different algorithms to increase the prediction accuracy. The simplest example of ensembling is to take the output from two algorithms (for example, SVM and neural net) and take the average of two predictions. Boosting is a very powerful algorithm that helps to combine "weak" models (the models with accuracy just a little bit higher than random guess).

Consider the equation (27.1) where we replace basis functions $b_k(x, \theta)$ by some classifiers $G_k(x) \in \{-1, 1\}$:

$$\eta = \sum_{k=1}^{K} \beta_k G_k(x). \tag{27.2}$$

We will use stagewise approach to find weights β_k and functions G_k in (27.2), the idea is similar to the backfitting algorithm for generalized additive models. Assuming that we have built the models $G_k(x)$ with weights β_k, $k = 1, \dots, K$, we try to find the next weight β and classifier $G(x)$ using exponential loss function:

$$\begin{aligned}
&\arg\min_{\beta, G} L(z, \eta) = \arg\min_{\beta, G} \sum_{i=1}^{m} \exp\left(-z^{(i)}\eta^{(i)}\right) = \\
&= \arg\min_{\beta, G} \sum_{i=1}^{m} \exp\left(-z^{(i)}\left(\sum_{k=1}^{K} \beta_k G_k(x^{(i)}) + \beta G(x^{(i)})\right)\right) = \\
&= \arg\min_{\beta, G} \sum_{i=1}^{m} w^{(i)} \exp\left(-z^{(i)}\beta G(x^{(i)})\right),
\end{aligned}$$

where

$$w^{(i)} = \exp\left(-z^{(i)} \sum_{k=1}^{K} \beta_k G_k(x^{(i)})\right) \tag{27.3}$$

are constants during the algorithm step and can be considered as weights.

If the classifier G predicts the training example i correctly, then

$$z^{(i)} G(x^{(i)}) = 1,$$

otherwise,

$$z^{(i)} G(x^{(i)}) = -1.$$

Hence,

$$\begin{aligned}
&\arg\min_{\beta, G} \left(\sum_{i=1}^{m} w^{(i)} \exp\left(-z^{(i)}\beta G(x^{(i)})\right)\right) = \\
&= \arg\min_{\beta, G} \left(\sum_{i:\, G(x^{(i)}) = z^{(i)}} w^{(i)} e^{-\beta} + \sum_{i:\, G(x^{(i)}) \neq z^{(i)}} w^{(i)} e^{\beta}\right) = \\
&= \arg\min_{\beta, G} \left((e^{\beta} - e^{-\beta}) \cdot \sum_{i=1}^{m} w^{(i)} \mathbb{1}\{z^{(i)} \neq G(x^{(i)}\} + e^{-\beta} \cdot \sum_{i=1}^{m} w^{(i)}\right).
\end{aligned}$$

This formula gives the following results:

- if β is fixed, then $G = \arg\min_G \sum_{i=1}^{m} w^{(i)} \mathbb{1}\{z^{(i)} \neq G(x^{(i)})\}$. In other words, our classifier G should minimize the sum of weights for wrongly predicted training examples.
- if G is fixed, then calculating the derivative with respect to β and assigning it to zero gives:

$$\beta = \frac{1}{2} \ln \frac{1-r}{r}, \text{ where } r = \frac{\sum_{i=1}^{m} w^{(i)} \mathbb{1}\{z^{(i)} \neq G(x^{(i)})\}}{\sum_{i=1}^{m} w^{(i)}}. \tag{27.4}$$

  This result has a perfect common sense: if $r \to \frac{1}{2}$ (random classifier), then $\beta \to 0$; if $r \to 0$ (perfect classifier), then $\beta \to \infty$. As we can see the algorithm will give higher weight β to more accurate classifier G.

We can also notice that weights $w^{(i)}$ can be updated from previous step with the formula (27.3). Indeed,

$$w^{(i)}_{new} = \exp\left(-z^{(i)} \sum_{k=1}^{K-1} \beta_k G_k(x^{(i)}) - z^{(i)} \beta_K G_K(x^{(i)})\right) =$$
$$= w^{(i)}_{old} \cdot \exp\left(-z^{(i)} \beta_K G_K(x^{(i)})\right),$$

but $-z^{(i)} G_K(x^{(i)}) = 2 \cdot \mathbb{1}\{z^{(i)} \neq G_K(x^{(i)})\} - 1$, thus

$$w^{(i)}_{new} = w^{(i)}_{old} \cdot e^{\alpha_K \mathbb{1}\{z^{(i)} \neq G_K(x^{(i)})\}} \cdot e^{-\beta_K},$$

where $\alpha_K = 2\beta_K$ and finally, we can remove the term $e^{-\beta_K}$, because it multiplies all weights by the same factor:

$$w^{(i)}_{new} = w^{(i)}_{old} \cdot e^{\alpha_K \mathbb{1}\{z^{(i)} \neq G_K(x^{(i)})\}} \tag{27.5}$$

If we start fitting with the constant weights for all training examples (for example, $w^{(i)} = \frac{1}{m}$ for all $i = 1, \dots, m$, then after each iteration k we multiply weight for the training example i by the factor

$$e^{\alpha_k \mathbb{1}\{z^{(i)} \neq G_k(x^{(i)})\}} = \begin{cases} 1, \text{ if } G_k \text{ classifies the example } i \text{ correctly} \\ e^{\alpha_k}, \text{ otherwise.} \end{cases}$$

Now we can formulate the first boosting algorithm (Adaboost.M1):

Algorithm 10 Adaboost.M1

```
1: Initialize weights $w^{(i)} = \frac{1}{m}$, $i = 1, 2, \dots, m$
2: for $k = 1$ to $K$ do
3:     Fit a classifier $G_k(x)$ to the training data with weights $w^{(i)}$
4:     Compute $r = \frac{\sum_{i=1}^{m} w^{(i)} \mathbb{1}\{z^{(i)} \neq G(x^{(i)})\}}{\sum_{i=1}^{m} w^{(i)}}$        ▷ (27.4)
5:     Compute $\alpha_k = \ln \frac{1-r}{r}$ (weight for the classifier $G_k$)
6:     Set $w^{(i)} := w^{(i)} \cdot \exp\left(\alpha_k \cdot \mathbb{1}\{z^{(i)} \neq G_k(x^{(i)})\}\right)$, $i = 1, \dots, m$        ▷ (27.5)
7: return $G(x) = \operatorname{sign}\left(\sum_{k=1}^{K} \alpha_k G_k(x)\right)$
```

28 Boosting trees

28.1 Gradient boosting

In this section we combine the ideas of backfitting algorithm and decision tree. First, we introduce some loss function $L(f)$, for different problems we can choose squared loss function (for regression)

$$L(f) = \frac{1}{2m} \sum_{i=1}^{m} (y^{(i)} - f(x^{(i)}))^2,$$

or cross-entropy loss (for classification)

$$L(f) = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \ln f(x^{(i)}) + (1 - y^{(i)}) \ln(1 - f(x^{(i)})) \right),$$

or any other convenient loss function. The comparison of the Algorithms 1 and 2 can give the following observations:

- In the Algorithm 2 we try to find minimum of the loss function as a function of f. Replacing the Newton's method by Gradient Descent gives rise to the updating step:

$$f^{new} = f^{old} - \alpha \frac{dL}{df}\bigg|_{f=f^{old}}, \tag{28.1}$$

  where f^{old} is a vector of current values of f on all training examples, $\frac{dL}{df}\Big|_{f=f^{old}}$ is a derivative of the loss function $L(f)$ with respect to f calculated at all current values f^{old}, α is a learning rate.

- In the Algorithm 1 we find the new value for f as
$$f^{new} = f^{old} + g, \tag{28.2}$$
where g is fitted using some building block algorithm (for example, univariate cubic splines) with previous residues as as working target.

The equations (28.1) and (28.2) give a new idea: find the function g by fitting the building block algorithm to the working target $r = -\frac{dL}{df}\Big|_{f=f^{old}}$ and find the coefficient α as

$$\alpha = \arg\min_{\alpha} L(f^{old} + \alpha g).$$

This idea is called **gradient boosting**.

28.2 Gradient tree boosting

Gradient boosting can be easily implemented when the building block algorithm is a classification or decision tree.

Algorithm 11 Gradient Tree Boosting

```
1: Initialize $f_0(x) = \arg\min_{\mu} \sum_{i=1}^{m} L(y^{(i)}, \mu)$
2: for $k = 1$ to $K$ do
3:     Compute working target $r_k^{(i)} = -\left(\frac{dL}{df}\right)\Big|_{f=f_{k-1}(x^{(i)})}$ for all $i = 1, \dots, m$
4:     Fit a regression tree to the targets $r_k^{(i)}$ with terminal nodes $R_{kj}$, $j = 1, \dots, J_k$
5:     Compute
           $$\gamma_{kj} = \arg\min_{\gamma} \sum_{x^{(i)} \in R_{kj}} L(y^{(i)}, f_{k-1}(x^{(i)}) + \gamma)$$
       for all $j = 1, \dots, J_k$
6:     Update $f_k(x) = f_{k-1}(x) + \sum_{j=1}^{J_k} \gamma_{kj} \mathbb{1}\{x \in R_{kj}\}$
7: return $f_K(x)$
```

In the line 5 of the algorithm we should find the best value for γ in each terminal node (for squared loss function it will be just an average of working target).

The classification algorithm is similar except that in the line 4 we should fit a classification tree and also we should repeat lines 3-6 for each class.

Exercise. Write down the pseudocode for the Gradient Tree Boosting for the classification problem with

a) two classes;

b) D classes.

28.3 Python implementation

As a benchmark we use the `boston` data from the `scikit-learn` package.

```
In [32]: boston = ds.load_boston()
         X = boston.data
         y = boston.target/50.
         X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                              test_size=0.33,
                                                              random_state=42)
```

```
In [33]: print 'Number of training examples: ' + str(X.shape[0])
         print 'Number of features: ' + str(X.shape[1])
```

```
Number of training examples: 506
Number of features: 13
```

Now we train the gradient trees boosting model implemented in the package `xgboost`:

```
In [34]: import xgboost as xgb

         param = {}
         param['objective'] = 'reg:linear'
         param['eta'] = 0.03
         param['max_depth'] = 10
         param['eval_metric'] = 'rmse'
         param['silent'] = 1
         param['nthread'] = 8
         param['gamma'] = 1.0
         param['lambda'] = 0.0
         param['min_child_weight'] = 1
         param['subsample'] = 0.8
         param['colsample_bytree'] = 1.0
         num_round = 1000
         param['seed'] = 2657
         plst = list(param.items())

         xgmat_train = xgb.DMatrix(X_train, label=y_train,
                                   missing = -999.0)
         bst = xgb.train( plst, xgmat_train, num_round );
         xgmat_test = xgb.DMatrix(X_test, missing = -999.0)
         y_pred = bst.predict( xgmat_test )
         print rmse(y_test, y_pred)
```

```
0.108290326706
```

We will implement gradient tree boosting using `DecisionTreeRegressor` class from the `scikit-learn`. Notice that we did not use a lot of tricks (like shrinkage, regularization and so on).

```
In [35]: from sklearn.tree import DecisionTreeRegressor

In [36]: def dLdf(X, y, f0, f):
             if len(f) > 0:
                 ypred = np.sum(np.array([fj.predict(X) for fj in f]).T,
                                axis=1) + f0
             else:
                 ypred = np.ones_like(y) * f0
             return y - ypred

         def gradient_boosting_trees(X, y, ntrees=100, max_depth=3):
             f0 = np.mean(y)
             nsamples = X.shape[0]
             f = []
             for k in range(ntrees):
                 # define the working target
                 r = dLdf(X, y, f0, f)
                 clf = DecisionTreeRegressor(max_depth=max_depth)
                 clf.fit(X, r)
                 f.append(clf)
             return f0, f

In [37]: f0, f = gradient_boosting_trees(X_train, y_train,
                                         ntrees=100, max_depth=2)

In [38]: ypred = np.sum(np.array([fj.predict(X_test) for fj in f]).T,
                        axis=1) + f0
         rmse(y_test, ypred)

Out[38]: 0.075087791829280118
```

Compared to the single decision tree, the accuracy is better:

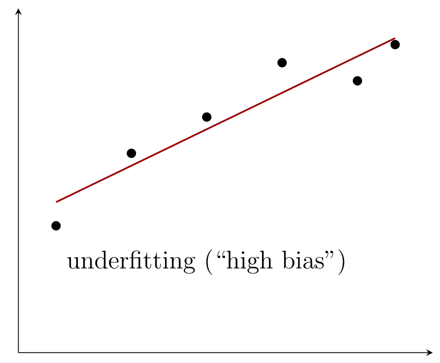
```
In [39]: clf = DecisionTreeRegressor(max_depth=5)
         clf.fit(X_train, y_train)
         ypred = clf.predict(X_test)
         rmse(y_test, ypred)

Out[39]: 0.080012682023411025
```

Learning theory

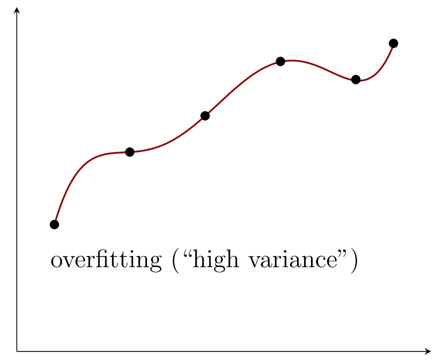
29 Bias / variance

In most cases to apply supervised machine learning algorithms we can use some packages where these algorithms are already implemented. But even when we use built-in algorithms we should know how it works. In this lecture we try to obtain some theoretical results about machine learning algorithms in general (without concentration on some particular algorithm).

In the Lecture 2 we talked about overfitting and underfitting. Let us recall these notions:

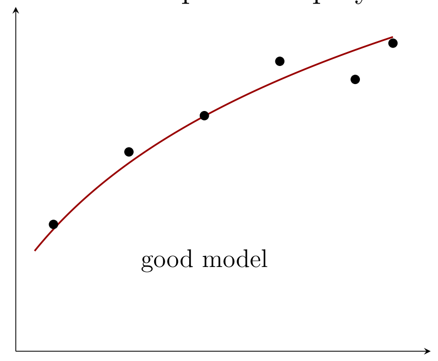


$\theta_0 + \theta_1 x$



$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

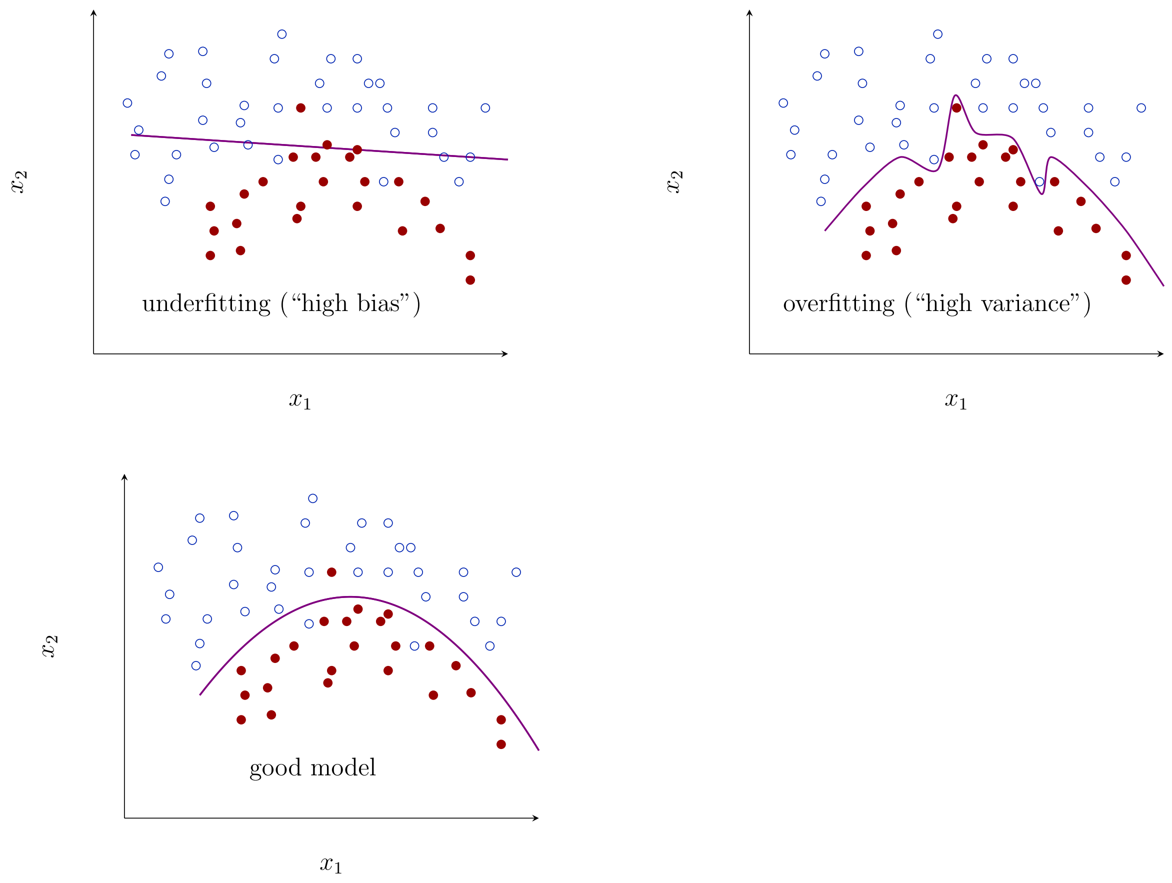
On the first picture we see that even when we take more data our error does not decrease, we say that this model has "high bias", or underfitting. On the second picture the error is zero, but if we add more data the prediction will be very bad, we say that this model has "high variance", or overfitting. The best model lies somewhere between, for example, it could be some quadratic polynomial in our example:



$\theta_0 + \theta_1 x + \theta_2 x^2$

The similar situation is happening for classification problems, for example, the next two figures show the underfitting and overfitting models for 2-class classification problem:

On the first picture we fit logistic regression with linear function (underfitting, or "high bias"), on the second we fit logistic regression with higher order polynomial (overfitting, or "high variance"). Our purpose is to find the best model somewhere between, for example, as before the good model could be the logistic regression with quadratic polynomial:



30 Empirical risk minimization (ERM)

To formalize the problem, consider a simple linear classification problem with the hypothesis:

$$h_\theta(x) = g(\theta^T x),$$

where

$$g(z) = \mathbb{1}\{z \geqslant 0\}$$

and $y \in \{0, 1\}$. Denote the training dataset by $S = \{x^{(i)}, y^{(i)}\}_{i=1}^m$ and assume that $(x^{(i)}, y^{(i)})$ are independent identically distributed sampled from some distribution D.

The **training error**, or **risk**, for this problem is defined as

$$\hat{\varepsilon}(h_\theta) = \hat{\varepsilon}_S(h_\theta) = \frac{1}{m} \sum_{i=1}^{m} \mathbb{1}\{h_\theta(x^{(i)}) \neq y^{(i)}\} \tag{30.1}$$

The **empirical risk minimization (ERM)** problem is a non-convex optimization problem that can be formulated as

$$\hat{\theta} = \arg\min_\theta \hat{\varepsilon}_S(h_\theta).$$

With this formulation we can define the parametric models only. To cover nonparametric models we formulate the problem in more general way. Let the hypothesis class $\mathcal{H} = \{h_\theta, \theta \in$

$\mathbb{R}^{n+1}\}$ includes the hypotheses $h_\theta : X \to \{0,1\}$. The hypothesis class $\mathcal{H}$ can contain any set of functions, for example, functions, produced by neural networks, or piecewise functions, produced by tree-based methods. Then ERM problem is equivalent to

$$\hat{h} = \arg\min_{h\in\mathcal{H}} \hat{\varepsilon}_S(h).$$

The **generalization error** for the hypothesis h is defined as

$$\varepsilon(h) = P(h(x) \neq y \,|\, (x,y) \sim D) \tag{30.2}$$

(probability that hypothesis h does not classify training example x correctly, given that (x, y) are drawn from the distribution D). The main difference between training error and generalization error is that **training error is calculated on the training set** S only, but **generalization error is calculated for any sample drawn from the distribution** D. The problem of generalization error minimization is very complicated and cannot be solved in majority of cases. Usually, this problem is replaced by the ERM problem. The main question what is the relationship between training error $\hat{\varepsilon}$ and generalization error ε.

31 Union bound / Hoeffding inequality

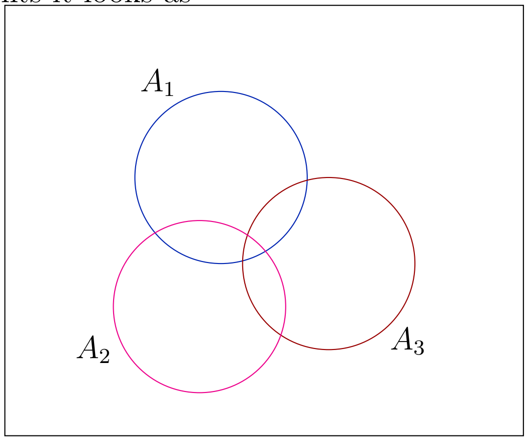
Two facts from the probability theory are necessary for our next results.

Theorem (union bound). Let $A_1, A_2, \ldots, A_k$ be k probabilistic events (not necessarily independent). Then

$$P(A_1 \cup A_2 \cup \ldots \cup A_k) \leqslant P(A_1) + P(A_2) + \ldots + P(A_k),$$

where $\cup$ defines the union of events $A_1, \ldots, A_k$ (operator "OR").

The illustration for the union bound theorem is the Venn diagram, for example, for 3 events it looks as



Theorem (Hoeffding inequality). Let $z_1, \ldots, z_m$ be m independent identically distributed Bernoulli random variables with mean φ, i.e.

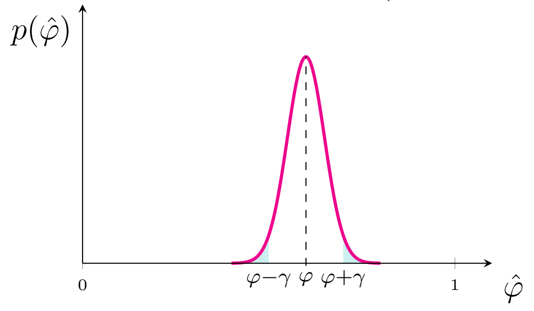
$$P(z_i = 1) = \varphi \text{ for all } i,$$

and $\hat{\varphi} = \frac{1}{m}\sum_{i=1}^{m} z_i$, then

$$P(|\hat{\varphi} - \varphi| > \gamma) \leqslant 2e^{-2\gamma^2 m}$$

for any fixed number $\gamma > 0$.

This theorem means that if we choose a number φ, then the density function for $\hat{\varphi}$ has a shape of normal distribution (the distribution of $\hat{\varphi}$ is roughly normal):



When γ is fixed then the probability to have $\hat{\varphi}$ on the tails (shaded regions) is less than $2e^{-2\gamma^2 m}$, and it is decreasing exponentially with m.

32 Uniform convergence

Consider the case when the hypothesis class $\mathcal{H}$ is finite: $\mathcal{H} = \{h_1, \dots, h_k\}$ (we have just k hypotheses in the hypothesis class $\mathcal{H}$). The solution for the ERM problem is straightforward: for each hypothesis $h_j \in \mathcal{H}$ calculate the training error $\hat{\varepsilon}$ on the training set S and choose the hypothesis with the minimal error:

$$\hat{h} = \arg\min_{h_j \in \mathcal{H}} \hat{\varepsilon}_S(h_j).$$

We prove that such simple procedure helps to minimize generalization error ε as well. For any $h_j \in \mathcal{H}$, we define

$$z^{(i)} = \mathbb{1}\{h_j(x^{(i)}) \neq y^{(i)}\} \in \{0, 1\}.$$

By definition (30.2),

$$P(z_i = 1) = \varepsilon(h_j),$$

which means that z_i are independent identically distributed Bernoulli random variables with mean $\varepsilon(h_j)$. The formula (30.1) implies

$$\hat{\varepsilon}(h_j) = \frac{1}{m}\sum_{i=1}^{m} z_i = \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}\{h_j(x^{(i)}) \neq y^{(i)}\},$$

then by Hoeffding inequality

$$P(|\varepsilon(h_j) - \hat{\varepsilon}(h_j)| > \gamma) \leqslant 2e^{-2\gamma^2 m}.$$

The obvious consequence we can obtain from this inequality: if m becomes large then the difference between training error and generalization error becomes smaller.

For any hypothesis $h_j \in \mathcal{H}$ we define the random event A_j that occurs if $|\varepsilon(h_j) - \hat{\varepsilon}(h_j)| > \gamma$. We proved that

$$P(A_j) \leqslant 2e^{-2\gamma^2 m}.$$

Consider the probability

$$\begin{aligned} &P(\exists h_j \in \mathcal{H} \text{ such that } |\varepsilon(h_j) - \hat{\varepsilon}(h_j)| > \gamma) = \\ &= P(A_1 \cup A_2 \cup \ldots \cup A_k) \leqslant \sum_{j=1}^{k} P(A_j) = \sum_{j=1}^{k} 2e^{-2\gamma^2 m} = 2ke^{-2\gamma^2 m} \end{aligned}$$

by the union bound theorem. The probability of opposite event

$$P(\forall h_j \in \mathcal{H} : \, |\varepsilon(h_j) - \hat{\varepsilon}(h_j)| \leqslant \gamma) \geqslant 1 - 2ke^{-2\gamma^2 m}.$$

The last result can be reformulated as: with probability $1 - 2ke^{-2\gamma^2 m}$ the training error $\hat{\varepsilon}(h)$ will be within γ of generalization error $\varepsilon(h)$ for all $h \in \mathcal{H}$. This statement is called **uniform convergence**.

There are three parameters of interest:

- γ: how far the training error from the generalization error (**error bound**)
- m: how many training examples we have
- δ: what is the range for probability

We analyse the uniform convergence result:

- Given γ and m, the probability $\delta = 2ke^{-2\gamma^2 m}$.
- Given γ and δ, what should be the size m of the training set?

  $$\delta = 2ke^{-2\gamma^2 m} \Leftrightarrow m = \frac{1}{2\gamma^2} \ln \frac{2k}{\delta},$$

  which means that if the number of training examples $m \geqslant \frac{1}{2\gamma^2} \ln \frac{2k}{\delta}$, then with probability $(1 - \delta)$ we have $|\varepsilon(h) - \hat{\varepsilon}(h)| \leqslant \gamma$ for all $h \in \mathcal{H}$ (**"sample complexity" bound**).
- Given m and δ, what will be the error bound γ?

  $$\delta = 2ke^{-2\gamma^2 m} \Leftrightarrow \gamma = \sqrt{\frac{1}{2m} \ln \frac{2k}{\delta}},$$

  which means that with probability $(1 - \delta)$

  $$|\varepsilon(h) - \hat{\varepsilon}(h)| \leqslant \sqrt{\frac{1}{2m} \ln \frac{2k}{\delta}}$$

  for any $h \in \mathcal{H}$.

Notice that m grows as $\ln k$ and if we add huge number of hypothesis to $\mathcal{H}$, then m does not grow a lot.

Let us see how the uniform convergence result is related to ERM problem. We assume

$$|\varepsilon(h) - \hat{\varepsilon}(h)| < \gamma \tag{32.1}$$

$$\hat{h} = \arg\min_{h\in\mathcal{H}} \hat{\varepsilon}(h) \text{ (ERM problem)} \tag{32.2}$$

$$h^* = \arg\min_{h\in\mathcal{H}} \varepsilon(h) \text{ (best hypothesis)} \tag{32.3}$$

The equation (32.3) defines the best hypothesis for the specific problem. As we have limited resources usually we cannot find it, but we can approximate it by the hypothesis $\hat{h}$.

Theorem. Let $|\mathcal{H}| = k$ and let m, δ be fixed, then with probability $1-\delta$

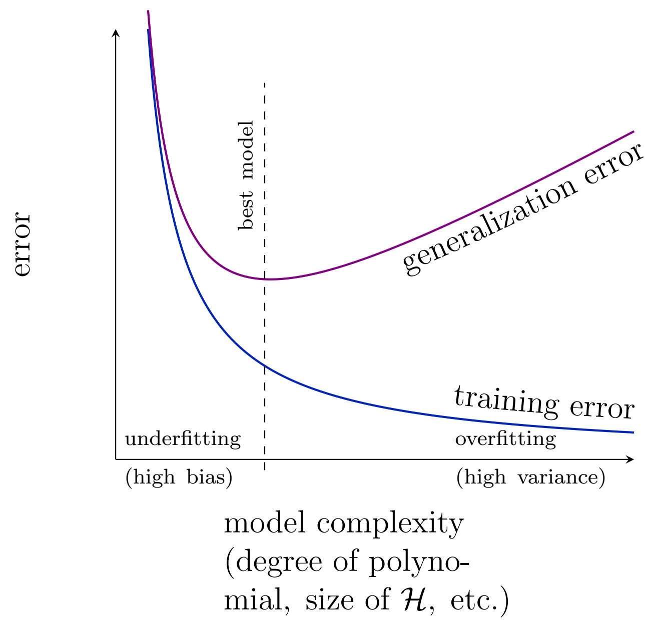
$$\varepsilon(\hat{h}) \leqslant \left(\min_{h\in\mathcal{H}} \varepsilon(h)\right) + 2\sqrt{\frac{1}{2m}\ln\frac{2k}{\delta}}. \tag{32.4}$$

Proof. By equation (32.1) and (32.2)

$$\varepsilon(\hat{h}) \overset{(32.1)}{\leqslant} \hat{\varepsilon}(\hat{h}) + \gamma \overset{(32.2)}{\leqslant} \hat{\varepsilon}(h^*) + \gamma \overset{(32.1)}{\leqslant} \varepsilon(h^*) + 2\gamma \overset{(32.2)}{=} \left(\min_{h\in\mathcal{H}} \varepsilon(h)\right) + 2\gamma.$$

Before we proved that $\gamma = \sqrt{\frac{1}{2m}\ln\frac{2k}{\delta}}$ and the equation (32.1) holds with probability $(1-\delta)$ for any hypothesis $h \in \mathcal{H}$, that proves the theorem. □

How we can use the obtained result? We could start solving the machine learning problem in the class of linear hypothesis $\mathcal{H}$ and after expand the class $\mathcal{H}$ to the class $\mathcal{H}'$ of quadratic hypothesis. What happens with our errors? Obviously, that $\varepsilon(h^*)$ becomes better (the best hypothesis on the class of quadratic functions is better then on the class of linear functions). But simultaneously, the number of hypothesis k increases. If we say that the first term $\min_{h\in\mathcal{H}} \varepsilon(h)$ in the equation (32.4) represents the "bias" and the second term $2\sqrt{\frac{1}{2m}\ln\frac{2k}{\delta}}$ represent the "variance", then if we expand the hypothesis class $\mathcal{H}$ our "bias" term is decreasing and "variance" term is increasing. This behaviour can be visualized with the graph for fixed number m of training examples:



Corollary. Let $|\mathcal{H}| = k$ and let δ and γ be fixed. Then to have

$$\varepsilon(\hat{h}) \leqslant \min_{h \in \mathcal{H}} \varepsilon(h) + 2\gamma$$

with probability $(1 - \delta)$, it is sufficient that

$$m \geqslant \frac{1}{2\gamma^2} \ln \frac{2k}{\delta} = O\left(\frac{1}{\gamma^2} \ln \frac{k}{\delta}\right).$$

Example. If $k = 100$ (number of hypotheses), $\gamma = 0.1$ (the desired difference between training error and generalization errors), $\delta = 0.8$ (the probability), then $m = \frac{1}{0.1^2} \ln \frac{100}{0.8} \approx 483$.

33 VC dimension

Definition. Given a set of points $S = \{x^{(1)}, \ldots, x^{(m)}\}$, we say a hypothesis class $\mathcal{H}$ **shatters** S if $\mathcal{H}$ can realize any labelling on it (for any set of labels for m points there exists the hypothesis $h \in \mathcal{H}$ such that it maps points perfectly on the chosen set of labels).

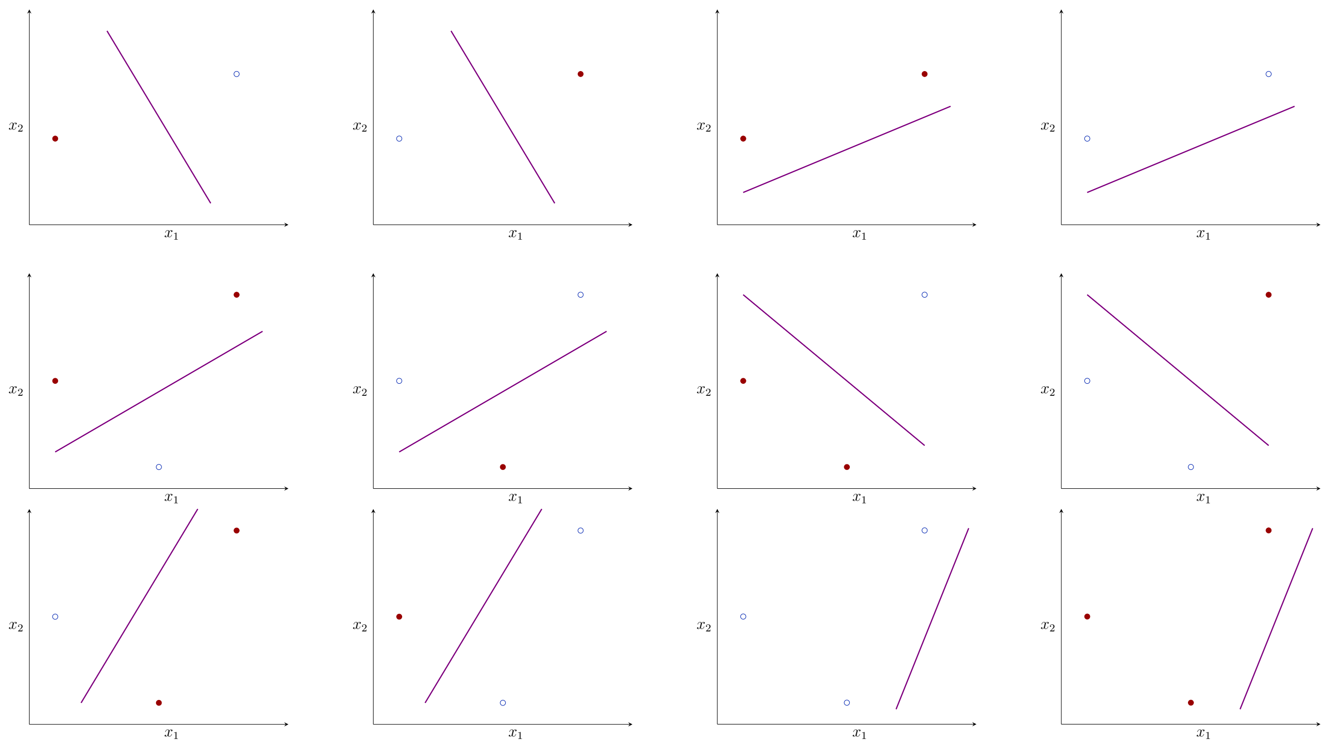
Examples.

1. $\mathcal{H}$ is a class of linear classifiers in 2D and $S = \{x^{(1)}, x^{(2)}\}$, then all possible labelings (assuming that $y \in \{0, 1\}$) are

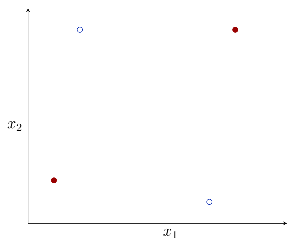
   For all labelings we can find the linear classifier which means that $\mathcal{H}$ shatters S.

2. $\mathcal{H}$ is a class of linear classifiers in 2D and $S = \{x^{(1)}, x^{(2)}, x^{(3)}\}$, then all possible labelings are

   As before for all labelings we can find the linear classifier which means that $\mathcal{H}$ shatters S.



3. $\mathcal{H}$ is a class of linear classifiers in 2D and $S = \{x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}\}$, then we can find labelling for which we cannot find the linear classifier that perfectly splits our classes:

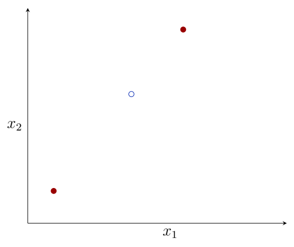


In this situation we say that $\mathcal{H}$ does not shatter data S.

Definition. The **Vapnik-Chervonenkis dimension** $VC(\mathcal{H})$ of the hypothesis class $\mathcal{H}$ is the size of the largest set shattered by $\mathcal{H}$.

Example. From the previous examples we can say that if $\mathcal{H}$ is a set of linear classifiers in 2D, then $VC(\mathcal{H}) = 3$.

Notice that there is a set of 3 points on the plane that cannot be shattered by linear classifiers $\mathcal{H}$:



but it is enough to have at least one set of 3 points such that it is shattered by $\mathcal{H}$. For 4 points we can choose any set and for any set of 4 points it cannot be shattered by $\mathcal{H}$.

More generally, in the n-dimensional space

$$VC(\{\text{linear classifiers in } \mathbb{R}^n\}) = n + 1.$$

Theorem. Let $\mathcal{H}$ be given and $VC(\mathcal{H}) = d$. Then with probability $(1 - \delta)$, we have that for all $h \in \mathcal{H}$

$$|\varepsilon(h) - \hat{\varepsilon}(h)| \leqslant O\left(\sqrt{\frac{d}{m}\ln\frac{m}{d} + \frac{1}{m}\ln\frac{1}{\delta}}\right).$$

Thus, with probability $(1 - \delta)$ we also have

$$\varepsilon(\hat{h}) \leqslant \varepsilon(h^*) + O\left(\sqrt{\frac{d}{m}\ln\frac{m}{d} + \frac{1}{m}\ln\frac{1}{\delta}}\right).$$

Corollary. In order to guarantee that

$$\varepsilon(\hat{h}) \leqslant \varepsilon(h^*) + 2\gamma$$

with probability $(1 - \delta)$ it suffices that

$$m = O_{\gamma,\delta}(d).$$

(the last means that if we fix γ and δ, then $m = O(d)$).

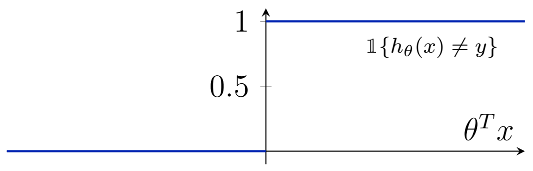
In many cases VC dimension is very close to the number of parameters, but there is one interesting case when it is not true. For the large margin classifiers (for example, SVM) the following property holds.

Theorem. Let $\mathcal{H}$ be the hypothesis class of the large margin classifiers and $||x||_2 \leqslant R$, then

$$VC(\mathcal{H}) \leqslant \left\lceil \frac{R^2}{4\gamma^2} \right\rceil + 1,$$

where γ is a minimal distance to the decision boundary.

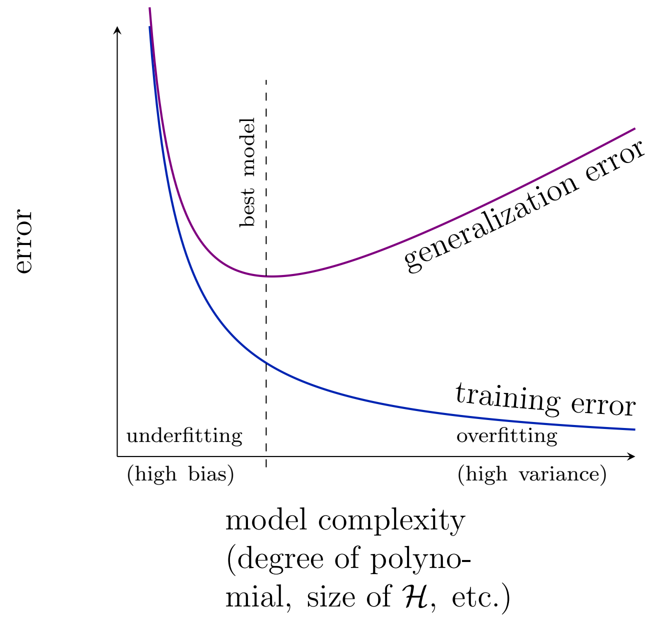
The last remark is that in this lecture we considered step function as our risk function for the ERM problem.



It turns out that logistic regression and SVM are just approximations to the ERM problem with logistic loss and hinge loss accordingly (as step function is not convex function). And this fact means that all the theory we have developed is correct for logistic regression and SVM classifiers.

34 Model selection

In previous lecture we showed the trade-off between bias and variance as follows



The main question is how to choose the model complexity: it could be degree of polynomial, bandwidth parameter in locally weighted linear regression or parameter C in SVM. Assuming that we have list of models

$$\mathcal{M} = \{M_1, M_2, M_3, \ldots\},$$

the wrong way would be to choose the model with the lowest error on the training set, because in such a way the most complicated model will be chosen. Consider different techniques to choose the best model (as before let S be our training set).

- **Hold-out cross validation**

  1. Split S into S_{train} (70%) and S_{val} (30%).
  2. Train each model $M \in \mathcal{M}$ on S_{train} and predict on S_{val}.
  3. Pick the model $M \in \mathcal{M}$ with the lowest error on S_{val}.

- **K-fold cross validation**

  1. Split S in K pieces (**folds**).
  2. Train on $K-1$ folds and predict on the remaining fold. Average the accuracy for all K folds.
  3. Pick the model $M \in \mathcal{M}$ with the lowest average error.

- **Leave-one-out cross validation**

  This technique is similar to K-fold cross validation but number of folds K equals to the size of the training set S. This method is very computationally expensive.

34.1 Python implementation

Import necessary libraries:

```
In [1]: import numpy as np
        import pandas as pd
        import random
        import math
        import sklearn.datasets as ds
        %matplotlib inline
```

Cross validation is one of the most important procedure, when you select the model or tune the parameters. There are several possibilities to perform cross validation in the `scikit-learn`.

```
In [2]: from sklearn.cross_validation import train_test_split
        from sklearn.cross_validation import StratifiedKFold
        from sklearn.cross_validation import LeaveOneOut
```

The simplest way to build cross validation splits is to use the method `train_test_split`:

```
In [3]: boston = ds.load_boston()
        X = boston.data
        y = boston.target/50.
        X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                             test_size=0.33,
                                                             random_state=42)
```

```
In [4]: print 'Number of training examples: ' + str(X_train.shape[0])
        print 'Number of validation examples: ' + str(X_test.shape[0])
```

```
Number of training examples: 339
Number of validation examples: 167
```

Another way is to use K-fold or leave-one-out cross validation procedures. Notice that we use the method `StratifiedKFold` that guarantees that the distribution of target variable stays the same for different folds.

```
In [5]: skf = StratifiedKFold(y, n_folds=3, shuffle=True, random_state=21387)
        for train_index, test_index in skf:
            X_train = X[train_index]
            X_test = X[test_index]
            print 'Number of training examples: ' + str(X_train.shape[0])
            print 'Number of validation examples: ' + str(X_test.shape[0])
```

```
Number of training examples: 323
Number of validation examples: 183
Number of training examples: 332
Number of validation examples: 174
Number of training examples: 357
Number of validation examples: 149
```

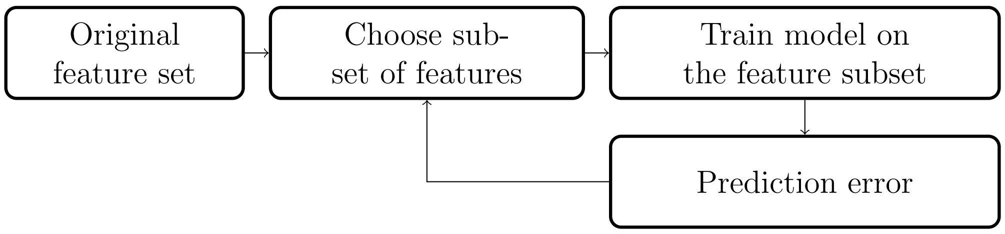
Using the method `LeaveOneOut` we can perform leave-one-out cross validation procedure in similar way.

35 Feature selection

Another approach to model selection is called feature selection. For many machine learning problems feature space is a very high-dimensional. To reduce it we should choose the subset of features: if we have n features then the number of possible subsets is 2^n. Feature selection methods can be splitted in three groups: wrappers, filters and embedded methods.

- Wrappers.

  Wrappers are model-based methods for feature selection and are considered to be the most effective and computationally intractable algorithms. The main principle is shown on the picture.



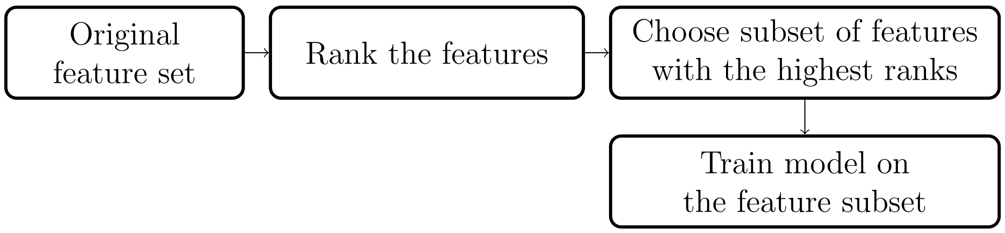
  To find the most relevant and informative subset of features, the model is trained for different subsets of features. The subset with the lowest error is selected as best set of features. Because wrapper methods utilize the model algorithm to extract relationships and causality between features, they are considered more effective and hence more desired than filters and embedded methods. To increase the speed of wrapper methods we can use different search methods such as forward selection, backward elimination and stepwise regression are available.

  – **Forward selection** begins with no variables and progressively adds features until maximum reduction in prediction error is reached.
  – **Backward elimination** begins with all features and progressively removes features having smallest contributions.
  – **Stepwise selection** starts by adding features until reaching some stopping criteria. Then the algorithm starts dropping features until reaching another stopping criteria and so on.

Notice that these techniques do not guarantee the selection of the global optimal feature set.

- Filters.

  Filters evaluate feature importance as a preprocessing operation to model training as depicted in the figure.


Original feature set
Rank the features
Choose subset of features with the highest ranks
Train model on the feature subset

  The main difference between filters and wrappers is that filters do not use the training procedure to capture the relationship between features. Rather, they use some information metric to calculate feature ranking from the data without direct input from the target. Popular information metrics include t-statistic, p-value, Pearson correlation coefficient and other correlation measures. One more example of such metric is mutual information

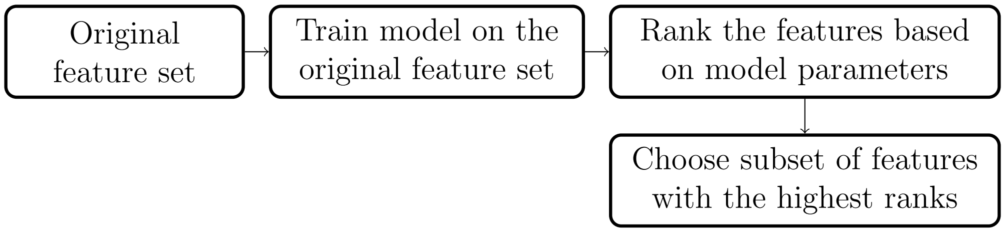
$$MI(x_i, y) = \sum_{x_i \in \{0,1\}} \sum_{y \in \{0,1\}} p(x_i, y) \ln \frac{p(x_i, y)}{p(x_i)p(y)} = KL(p(x_i, y)||p(x_i)p(y)).$$

  When all features are ranked the top k features are picked (k is chosen with the cross validation procedure).

  Computationally, filters are more efficient than wrappers as they require only the computation of n scores for n features. They are also more robust against overfitting than wrappers.

- Embedded methods.

  Embedded methods use training procedure to obtain feature rankings:


Original feature set
Train model on the original feature set
Rank the features based on model parameters
Choose subset of features with the highest ranks

  Regularization methods are the most common forms of embedded methods.

36 Bayesian approach and regularization

In previous lectures we have maximized the likelihood to get coefficients for the model, for example, for linear regression the maximum likelihood is

$$\theta = \arg\max_\theta \prod_{i=1}^m p(y^{(i)} \,|\, x^{(i)}, \theta).$$

In this technique we do not assume anything about parameters θ. Bayesian approach contains additional step: we add prior on θ, for example,

$$\theta \sim N(0, \tau^2 I),$$

where τ^2 is a vector of variances and I is a unit matrix. The Bayes rule gives

$$p(\theta \,|\, S) = \frac{p(S \,|\, \theta) p(\theta)}{p(S)},$$

where $S = \{(x^{(i)}, y^{(i)})\}_{i=1}^m$ is a training set. This formula can be transformed to

$$p(\theta \,|\, S) \propto \left(\prod_{i=1}^m p(y^{(i)} \,|\, x^{(i)}, \theta) \right) p(\theta), \tag{36.1}$$

because

$$p(S \,|\, \theta) = \prod_{i=1}^m p((x^{(i)}, y^{(i)}) \,|\, \theta) = \prod_{i=1}^m p(y^{(i)} \,|\, x^{(i)}, \theta) \cdot p(x^{(i)} \,|\, \theta).$$

Left side of the formula (36.1) is called **a "posterior"**. For the new test example (x, y) we obtain

$$p(y \,|\, x, S) = \int\limits_\theta p(y \,|\, x, \theta) \cdot p(\theta \,|\, S) \, d\theta$$

(here we treat θ as a random variable, that's why we put comma instead of semicolon) and in particularly

$$E\left[y \,|\, x, S\right] = \int\limits_y y \cdot p(y \,|\, x, S) \, dy.$$

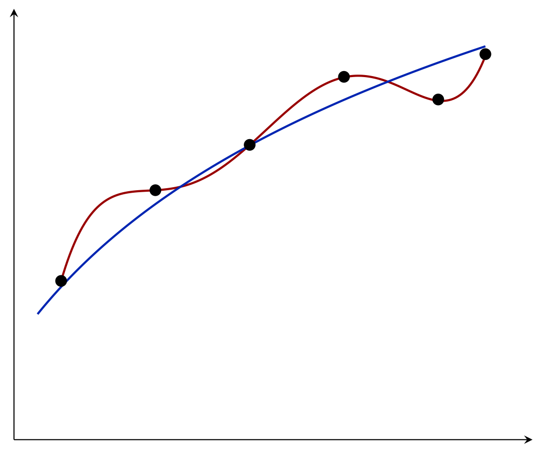
For many problems last two steps are computationally difficult, because usually, θ is many dimensional vector and the integration in $\mathbb{R}^n$ is not simple task. Easier way is to calculate

$$\hat{\theta}_{MAP} = \arg\max_\theta p(\theta \,|\, S) = \arg\max_\theta \left(\prod_{i=1}^m p(y^{(i)} \,|\, x^{(i)}, \theta) \right) p(\theta)$$

where "MAP" stands for "maximum a posteriori". After we found the set of parameters the prediction is calculated as before:

$$h_{\hat{\theta}_{MAP}}(x) = \hat{\theta}_{MAP}^T x.$$

The intuition behind the Bayesian technique is the following: if $\theta \sim N(0, \tau^2 I)$, then almost all parameters should be close to zero. Applying Bayesian regularization (blue line on the picture) will lead to reducing high variance compared to the model without regularization (red line on the picture):



To compare models without and with regularization we can look at the objective functions. For example, for linear regression without regularization the algorithm tries to solve the following optimization problem

$$\min_{\theta} \sum_i ||y^{(i)} - \theta^T x^{(i)}||^2,$$

then for Bayesian linear regression (or linear regression with regularization) the optimization problem becomes

$$\min_{\theta} \sum_i ||y^{(i)} - \theta^T x^{(i)}||^2 + \lambda||\theta||^2,$$

where λ is a regularization parameter.

The last remark is that Bayesian linear regression can be considered as embedded feature selection method where the size of weights θ defines the importance of the corresponding variables (zero weight means that feature is not important for the model).

36.1 Python implementation

In this section we show how regularization procedure affects the weights of the linear regression model. We generate data with 10 features with 6 out of 10 informative features:

```
In [6]: X,y = ds.make_regression(n_samples=100,
                                 n_features=10,
                                 n_informative=6,
                                 noise=1.0,
                                 bias=0,
                                 random_state=2016)
        y = y/10.
```

Different types of regularizations are implemented in the `scikit-learn` package. We will use `Lasso` class that implements L1 regularization.

```
In [7]: from sklearn.linear_model import Lasso
```

Now let us observe the model coefficients with respect to different values of the regularization term (alpha).

```
In [8]: for a in [0.0, 0.1, 0.5, 1.0, 10.0]:
            print 'alpha = ' + str(a) + ':'
            model = Lasso(alpha=a, normalize=False, max_iter=1000000)
            model.fit(X,y)
            print model.coef_
            print '***************'
```

```
alpha = 0.0:
[ -5.52107945e-03  -3.67860089e-03   7.64629577e+00   1.73280696e-01
   2.82826703e-01   2.28833747e+00   5.71482110e-03   4.49741291e+00
   5.56384364e+00   1.73914964e-03]
***************
alpha = 0.1:
[ 0.          0.          7.51109466  0.09366317  0.14180965  2.18308828
 -0.          4.39417767  5.4250924  -0.        ]
***************
alpha = 0.5:
[ 0.          0.          7.00822133  0.          0.          1.77766907
 -0.          3.9454249   4.87546516 -0.        ]
***************
alpha = 1.0:
[ 0.          0.          6.39221165  0.          0.          1.27978247
 -0.          3.36381678  4.19028074 -0.        ]
***************
alpha = 10.0:
[ 0.  0.  0.  0. -0.  0.  0.  0.  0. -0.]
***************
```

37 Online learning

The process of online learning can be described by the following diagram

$$x^{(1)} \to \hat{y}^{(1)} \to y^{(1)} \to x^{(2)} \to \hat{y}^{(2)} \to y^{(2)} \to x^{(3)} \to \ldots$$

Total online error is defined as

$$\sum_{i=1}^{m} \mathbb{1}\{\hat{y}^{(i)} \neq y^{(i)}\}.$$

Examples.

- Logistic regression with stochastic gradient descent.

    1. Initialize $\theta = 0$.
    2. After i-th example, update the parameters

    $$\theta := \theta + \alpha(y^{(i)} - h_\theta(x^{(i)})) \cdot x^{(i)}$$

    (stochastic gradient descent step).

- FTRL (Follow The Regularized Leader) algorithm.

38 Advice for apply ML algorithms

The first step of designing a learning system and solving the machine learning problem is look at the data and plot it. It gives understanding about what features are given and how to work with these features.

When you solve some practical problem you can obtain different results with the same algorithm (for example, xgboost). One of the reason could be the randomness of the algorithm. To reproduce the results of the algorithm usually we fix seed when we train the algorithm. Fixed seed can help to generate constant sequence of pseudorandom numbers.

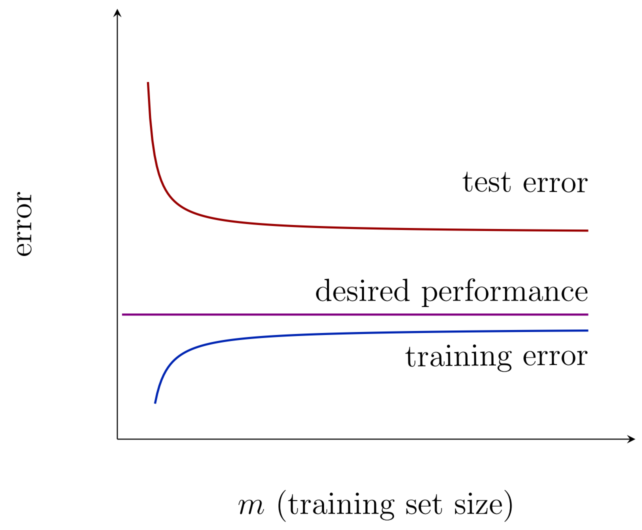
In this section we try to understand how to debug the learning algorithms. Consider some machine learning problem and choose some algorithm, as an example we choose Bayesian logistic regression implemented with the gradient descent:

$$\max_\theta \sum_{i=1}^m \ln p(y^{(i)} \,|\, x^{(i)}, \theta) - \lambda ||\theta||^2.$$

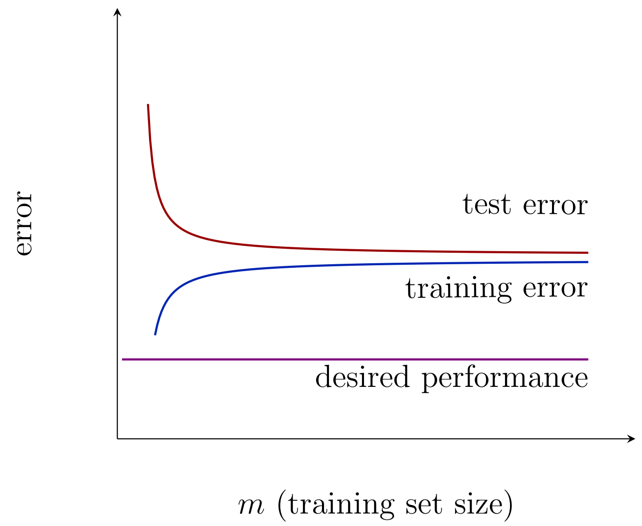
Using cross validation procedure we get 20% test error which is unacceptably high. The main question how to reduce this error. Several possibilities are

- get more training examples (fixes high variance)
- smaller set of features (fixes high variance)
- larger set of features (fixed high bias)
- change the features: for example, email header vs. email body features (fixes high bias)
- run gradient descent with more iterations (fixes optimization algorithm)
- replace gradient descent by Newton's method (fixes optimization algorithm)
- use different value for λ (fixes optimization objective)
- use different algorithm (for example, SVM) (fixes optimization objective)

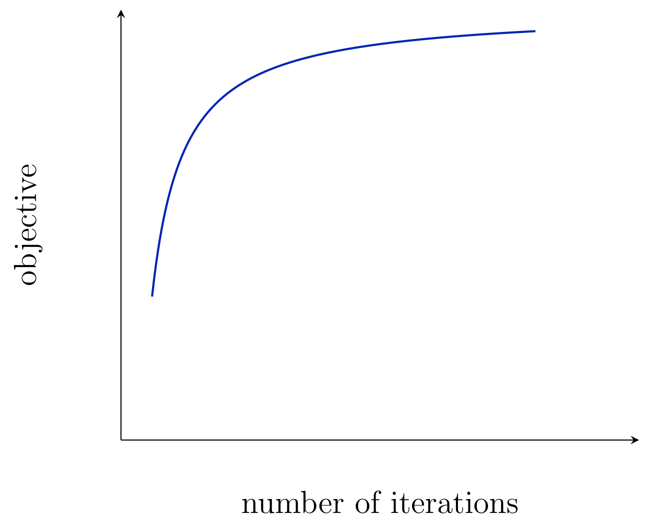
Consider some of these possibilities in more details. For example, we suspect a high variance (overfitting), then the training error will be much lower than the test error and this case is shown on the following graph (notice that gap between training and test errors is pretty big):



If we suspect a high bias (underfitting), the picture will be different:



Another diagnostics is related to the algorithm convergence. The following graph shows how the objective function changes when we increase the number of iterations:



With this graph we can understand if the algorithm converges (or if we use the correct number of iterations for the algorithm).

When optimization technique is applied we should have the correct objective function. The wrong objective function can also ruin the test error. One general technique is to add weights to the objective function

$$J(\theta) = \sum_i w^{(i)} \mathbb{1}\{h_\theta(x^{(i)}) = y^{(i)}\}$$

(for example, weights $w^{(i)}$ could be higher for non-spam than for spam).

Finally, you should carefully tune the parameters of the algorithm (for example, λ for the Bayesian logistic regression or C for the support vector machine).

In general, there are two approaches to machine learning problems solving:

- Careful design:
  - spend a long term designing exactly the right features, collecting the right dataset, and designing the right algorithmic architecture
  - implement it and hope it works

  The benefit of this approach is in the nicer and, perhaps, more scalable algorithms. We may also come up with new, elegant learning algorithms and contribute to basic research in machine learning.

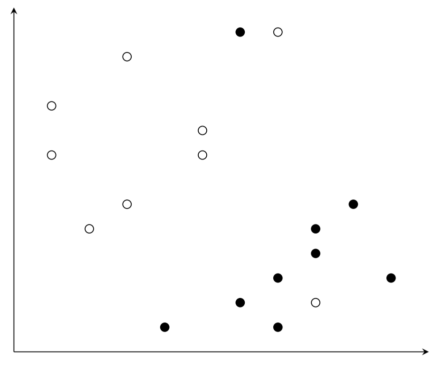
- Build and fix:
  - implement something quick-and-dirty
  - run error analysis and diagnostics to see what's wrong with it and fix its errors

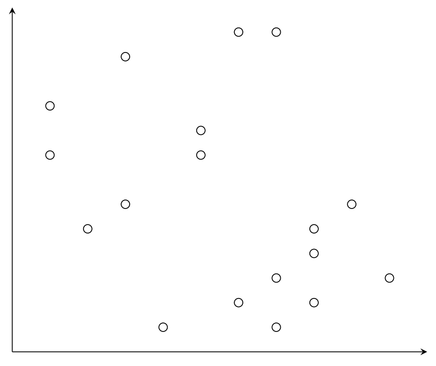
  The benefit of this approach is in the consumed time. Usually, it gets faster to make your application problem working, that could be important for the industrial sector.

Unsupervised learning

39 Clustering (k-means)

We start new part of our course that we call **unsupervised learning**. The difference between supervised and unsupervised learning can be explained by the following pictures:


Supervised learning


Unsupervised learning

In unsupervised learning problems no labels are given and job of the algorithm is to discover structure of the data.

The first example of unsupervised learning algorithm is called **clustering**. Clustering algorithm can be applied to identify groups of customers, groups of documents (for example, find groups of related articles on news.google.com), image segmentation and so on. The most popular example of clustering algorithm is **k-means algorithm**. Consider the unlabelled dataset:

$$\{x^{(1)}, \ldots, x^{(m)}\},\ x^{(i)} \in \mathbb{R}^n.$$

The k-means algorithm is listed as follows.

Algorithm 12 K-means algorithm
```
1: Set the number of clusters $k$
2: Initialize set of points (cluster centroids) $\mu_1, \ldots, \mu_k \in \mathbb{R}^n$
3: repeat
4:     Set $c^{(i)} = \arg\min_j ||x^{(i)} - \mu_j||$ (assigning point to the cluster $j$)
5:     Update cluster centroids to the means of points from clusters:
           $\mu_j := \frac{\sum_{i=1}^{m} \mathbb{1}\{c^{(i)} = j\} \cdot x^{(i)}}{\sum_{i=1}^{m} \mathbb{1}\{c^{(i)} = j\}}$
6: until convergence
7: return $c^{(i)}$
```

We want to say few words about the convergence of k-means algorithm. It turns out that it is easy to prove that this algorithm converges in some sense. By introducing the distortion function (this function is non-convex)

$$J(c, \mu) = \sum_{i=1}^{m} ||x^{(i)} - \mu_{c^{(i)}}||^2,$$

we can reformulate the steps of k-means algorithm using the coordinate ascent algorithm on J.

Exercise. Prove that this statement is correct.

Non-convexity of the distortion function does not guarantee finding the global minimum during the optimization algorithm. In practice, you should start k-means algorithm several times with different seed, it could help to identify the best set of clusters.

The final remark concerns the number of clusters (first step of the algorithm). What should be the correct number k of clusters. Usually this number is defined manually (there exist some automatic ways to define the number of clusters, in practice, the correct number of clusters is very ambiguous).

39.1 Python implementation

Import necessary libraries:

```
In [1]: import numpy as np
        import pandas as pd
        import random
        import math
        import sklearn.datasets as ds
        import matplotlib.pyplot as plt
```

K-means algorithm is implemented in the `scikit-learn` package:

```
In [2]: from sklearn.cluster import KMeans
```

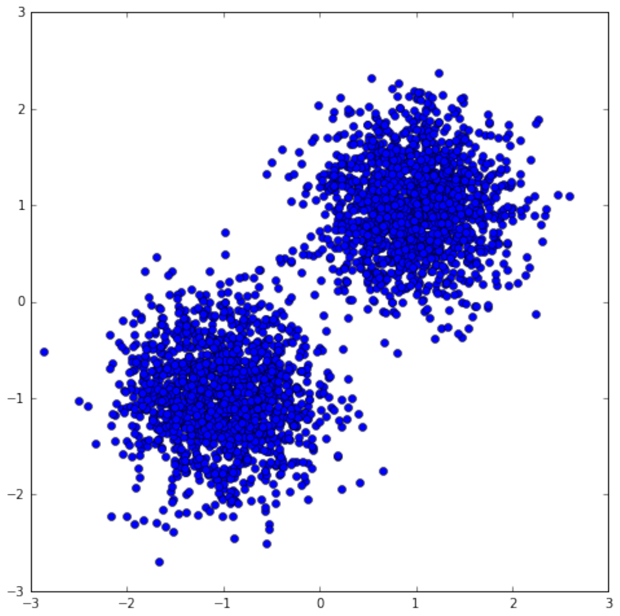
To illustrate how you can train this algorithm we generate some toy data and fit the algorithm on these data.

```
In [3]: from sklearn.datasets.samples_generator import make_blobs
        np.random.seed(0)
        centers = [[1, 1], [-1, -1]]
        n_clusters = len(centers)
        X, labels_true = make_blobs(n_samples=3000,
                                    centers=centers,
                                    cluster_std=0.5)
```

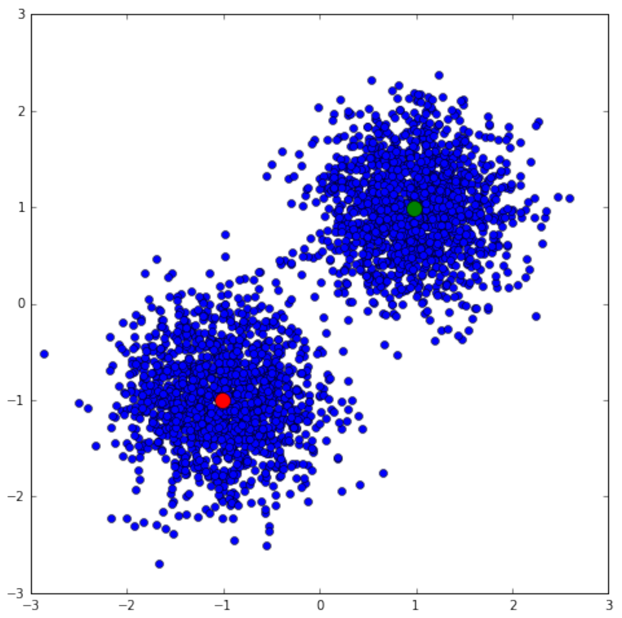
Helper function for data visualization:

```
In [4]: def plot_cluster_data(X, c=[1]*X.shape[0], mu=None):
            fig = plt.figure(figsize=(8, 8))
            ax = fig.add_subplot(1, 1, 1)
            if len(np.unique(c)) == 1:
                ax.plot(X[:,0], X[:,1], 'o')
            else:
                ix = np.where(c==1)
                ax.plot(X[ix,0], X[ix,1], 'o',
                        markerfacecolor='red')
                ax.plot(mu[0,0], mu[0,1], 'o',
                        markerfacecolor='red',
                        markersize=12)
                ix = np.where(c==0)
                ax.plot(X[ix,0], X[ix,1], 'o',
                        markerfacecolor='green')
                ax.plot(mu[1,0], mu[1,1], 'o',
                        markerfacecolor='green',
                        markersize=12)
            if not mu is None:
                ax.plot(mu[0,0], mu[0,1], 'o',
                        markerfacecolor='red',
                        markersize=12)
                ax.plot(mu[1,0], mu[1,1], 'o',
                        markerfacecolor='green',
                        markersize=12)
            plt.show()
```

```
In [5]: plot_cluster_data(X)
```



```
In [6]: clst = KMeans(n_clusters=2, random_state=2342)
        clst.fit(X)
        mu = clst.cluster_centers_
        plot_cluster_data(X, mu = mu)
```

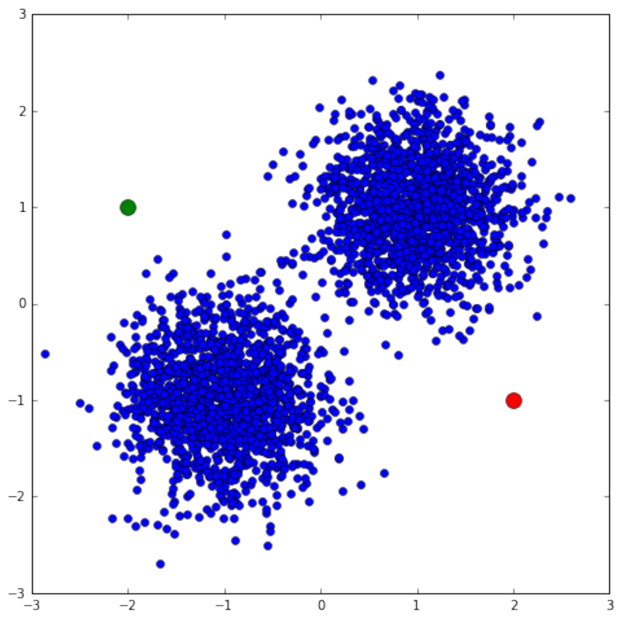


Now we show what happens on each step of the algorithm. We will use two additional functions that corresponds to the two steps of the k-means algorithm.

```
In [7]: def update_labels(X, mu):
            c = np.argmax(np.c_[np.sum(np.power(X - mu[0,:], 2), axis=1),
                                np.sum(np.power(X - mu[1,:], 2), axis=1)],
                          axis=1)
            return c

        def update_cluster_centers(X, c):
            ix = np.where(c==1)
            mu[0,:] = np.mean(X[ix,:], axis=1)
            ix = np.where(c==0)
            mu[1,:] = np.mean(X[ix,:], axis=1)
            return mu
```

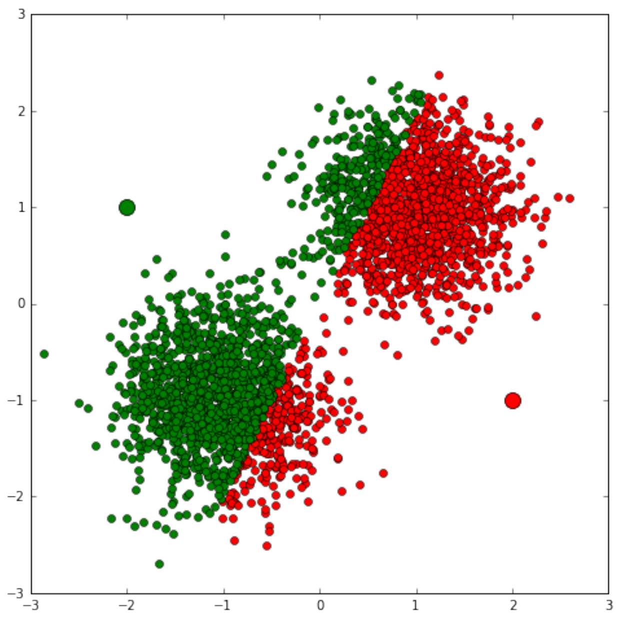
First step of the algorithm is to choose the number of clusters k and cluster centers randomly.

```
In [8]: k = 2
        mu = np.array([[2.0,-1.0], [-2.0,1.0]])
        plot_cluster_data(X, mu=mu)
```

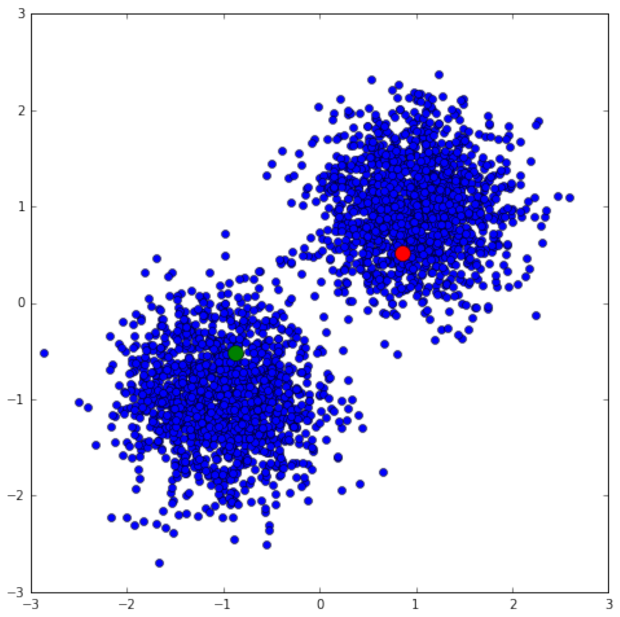


For each point we calculate what is the closest center:

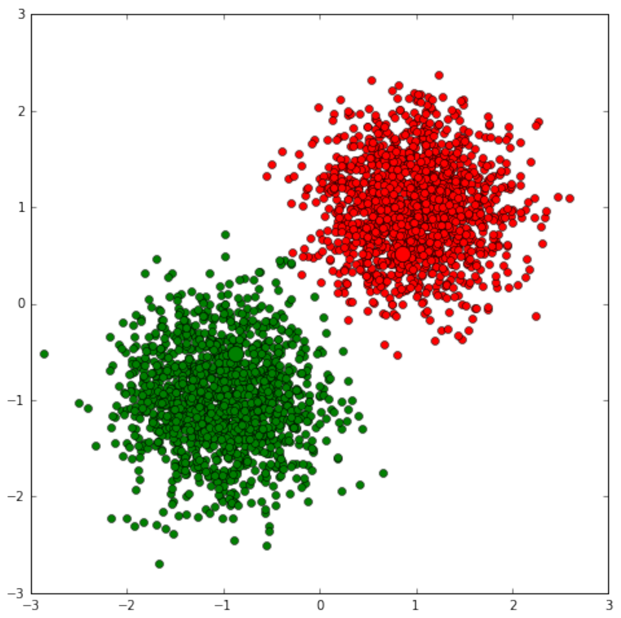
```
In [9]: niter = 2
        for it in range(niter):
            print 'Iteration ' + str(it) + ':'
            c = update_labels(X, mu)
            print '...updating labels:'
            plot_cluster_data(X, c=c, mu=mu)
            print '...updating centers:'
            mu = update_cluster_centers(X, c)
            plot_cluster_data(X, mu=mu)

Iteration 0:
...updating labels:
```



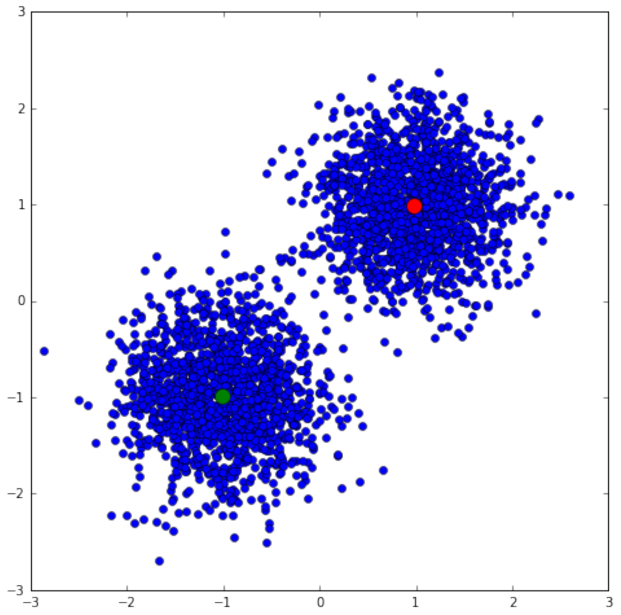
```
...updating centers:
```



```
Iteration 1:
...updating labels:
```

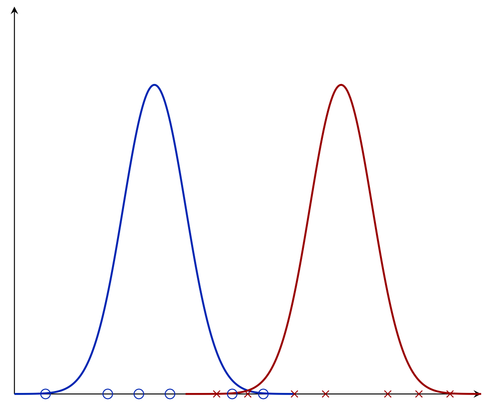


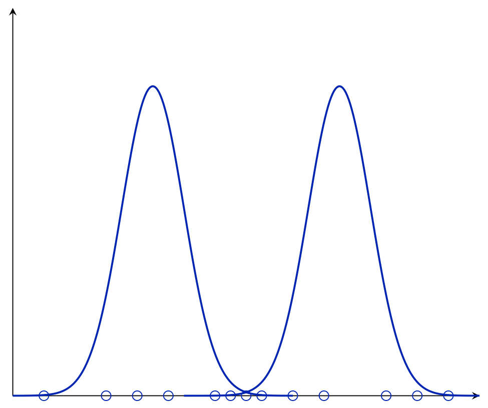
```
...updating centers:
```



40 Mixture of Gaussians

In some problems we do not need to find the clusters, but to estimate the density. Assuming we have the training set $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$, our aim is to find the probability density function $p(x)$. To understand the idea we compare this problem with Gaussian discriminant analysis algorithm. In the simplest case when $x^{(i)} \in \mathbb{R}$ (training examples are described by one feature) our aim is to build two Gaussian distributions:


Gaussian discriminant analysis


Mixture of Gaussians

The main difference between Gaussian discriminant analysis and mixture of Gaussians is that in the first case we can easily estimate the parameters of our distributions using class labels: for example, if we assume that

$$x^{(i)} \,|\, (y^{(i)} = j) \sim N(\mu_j, \Sigma_j),\ j \in \{0, 1\},$$
$$y^{(i)} \sim \text{Bernoulli}(\varphi),$$

then parameter φ can be calculated as

$$\varphi = \frac{1}{m} \sum_{i=1}^{m} \mathbb{1}\{y^{(i)} = 1\}$$

and maximum likelihood estimation gives the values for μ_j, Σ_j:

$$\mu_j = \frac{\sum\limits_{i=1}^{m} \mathbb{1}\{y^{(i)} = j\} \cdot x^{(i)}}{\sum\limits_{i=1}^{m} \mathbb{1}\{y^{(i)} = j\}},$$

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

It becomes more complicated for the mixture of Gaussians, because now labels are not given. We introduce a latent (hidden, unobserved) random variable z (this variable plays a role of y in Gaussian discriminant algorithm). By analogy we make the following assumptions:

$$z^{(i)} \sim \ \text{Multinomial}(\varphi),\ \varphi_j \geqslant 0,\ \sum_{j=1}^{k} \varphi_j = 1,$$
$$x^{(i)} \,|\, (z^{(i)} = j) \sim N(\mu_j, \Sigma_j),\ j \in \{1, \ldots, k\},$$

where k is a number of Gaussian distributions in our mixture. For $k = 2$ the random variable $z^{(i)} \sim$ Bernoulli(φ). We can write the log-likelihood

$$l(\varphi, \mu, \Sigma) = \sum_{i=1}^{m} \ln p(x^{(i)}, z^{(i)}; \varphi, \mu, \Sigma).$$

Notice that the probability in the right hand side is a joint distribution of $x^{(i)}$, $z^{(i)}$ that can be calculated as

$$p(x^{(i)}, z^{(i)}) = p(x^{(i)} \mid z^{(i)}) \cdot p(z^{(i)}),$$

which implies

$$l(\varphi, \mu, \Sigma) = \sum_{i=1}^{m} \ln \sum_{j=1}^{k} p(x^{(i)} \mid z^{(i)} = j; \mu, \Sigma) \cdot p(z^{(i)} = j; \varphi).$$

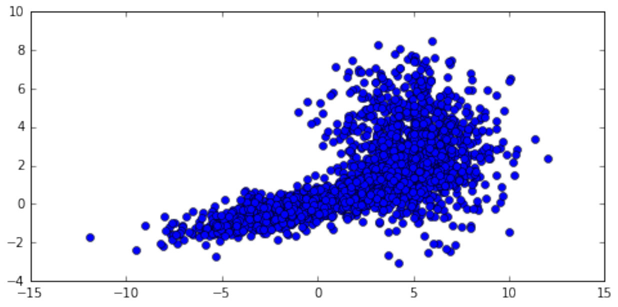
If we know the values of $z^{(i)}$ for all training example then we can maximize this log-likelihood by analogy with Gaussian discriminant analysis. Unfortunately, $z^{(i)}$ and parameters φ are unknown and MLE does not give the result in some closed form, to find the solution we should apply additional step to guess what is the distribution of z. We formulate our problem as

$$\arg\max_{\varphi,\mu,\Sigma} \sum_{i=1}^{m} \ln \sum_{j=1}^{k} p(x^{(i)} \mid z^{(i)} = j; \mu, \Sigma) \cdot p(z^{(i)} = j; \varphi) \tag{40.1}$$

40.1 Python implementation

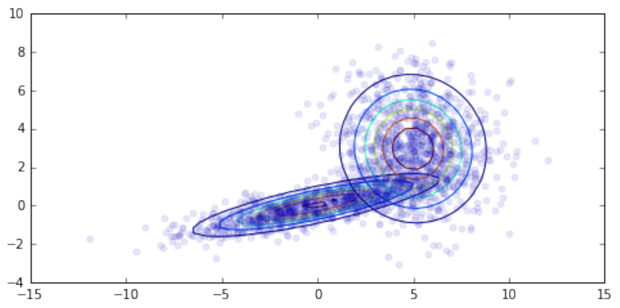
As an example we generate data based on two different gaussian distributions

```
In [10]: np.random.seed(0)
         n_samples = 1000
         X1 = 2.0*np.random.randn(n_samples, 2) + np.array([5, 3])
         C = np.array([[0., -0.5], [3.5, .7]])
         X2 = np.dot(np.random.randn(n_samples, 2), C)
         X_train = np.vstack([X1, X2])

         fig = plt.figure(figsize=(8, 8))
         ax = fig.add_subplot(1, 1, 1)
         ax.plot(X_train[:,0], X_train[:,1], 'o')
         plt.gca().set_aspect('equal', adjustable='box')
         plt.show()
```



and train mixture of Gaussians model on these data:

```
In [11]: from sklearn.mixture import GMM
In [12]: np.random.seed(1)
         model = GMM(n_components=2, covariance_type='full')
         model.fit(X_train)

         fig = plt.figure(figsize=(8, 8))
         ax = fig.add_subplot(1, 1, 1)
         ax.plot(X_train[:,0], X_train[:,1], 'o', alpha=.1, ms=5)

         for i in range(2):
             mu = model.means_[i]
             sigma = model.covars_[i]
             sigma_inv = np.linalg.inv(sigma)
             sigma_det = np.linalg.det(sigma)
             x = np.linspace(-15.0, 15.0)
             y = np.linspace(-4.0, 10.0)
             X, Y = np.meshgrid(x, y)
             XX = np.array([X.ravel(), Y.ravel()]).T
             XX = np.dot(np.dot(XX - mu, sigma_inv),
                         np.transpose(XX - mu))
             P = np.exp(-0.5*np.diagonal(XX))/(2*np.pi*sigma_det**0.5)
             P = P.reshape(X.shape)
             CS = plt.contour(X, Y, P)
         plt.gca().set_aspect('equal', adjustable='box')
         plt.show()
```



41 Jensen's inequality

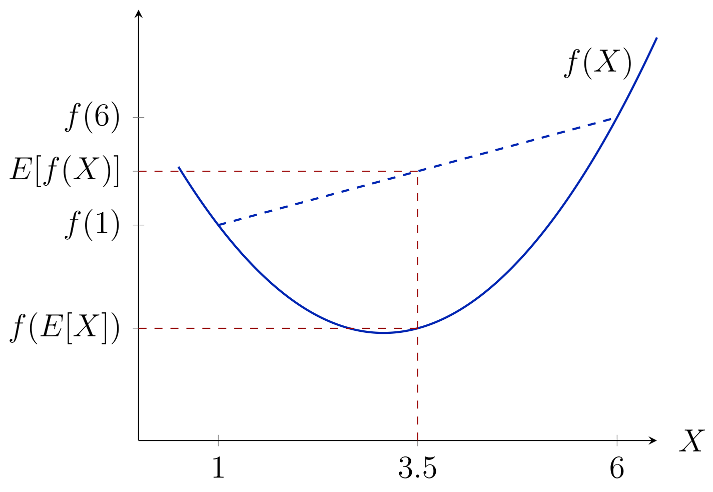
Theorem 1. Let f be a convex function (e.g. $f''(x) \geqslant 0$) and X be a random variable. Then

$$f(E[X]) \leqslant E[f(X)].$$

Example. Consider the random variable X with the density function

$$p(X) = \begin{cases} 0.5, & \text{if } X = 1, \\ 0.5, & \text{if } X = 6. \end{cases}$$

Then for the convex function $f(X)$ the Jensen's inequality is illustrated on the picture



Theorem 2. If $f''(x) > 0$ (f is strictly convex), then

$$E[f(X)] = f(E[X]) \text{ if and only if } X = E[X] \text{ with probability 1.}$$

Theorem 3. If $f''(x) \leqslant 0$ (concave downward), then

$$f(E[X]) \geqslant E[f(X)].$$

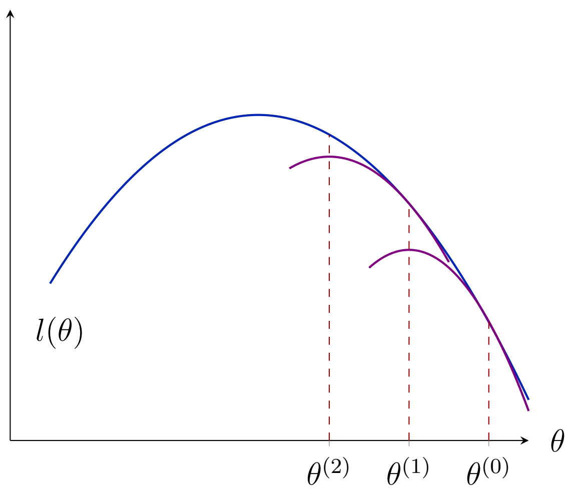
42 General EM algorithm

The algorithm that helps to solve the optimization problem (40.1) for mixture of Gaussians model is called **EM (expectation-maximization) algorithm**.

The general optimization problem that can be solved using EM algorithm is formulated as follows. Assuming that we have some model for $p(x, z; \theta)$, where x are observed and z are latent, our goal is to maximize log-likelihood with respect to parameters θ:

$$l(\theta) = \sum_{i=1}^{m} \ln p(x^{(i)}; \theta) = \sum_{i=1}^{m} \ln \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}; \theta).$$

EM algorithm is an iterative algorithm that constructs lower bound for the log-likelihood on each step and maximizes this lower bound instead of the original log-likelihood (violet curves on the picture are lower bounds):



Now we make this idea more formal. The maximum log-likelihood can be transformed as

$$\begin{aligned}\max_\theta l(\theta) = \max_\theta \sum_{i=1}^m \ln p(x^{(i)};\theta) = \sum_{i=1}^m \ln \sum_{z^{(i)}} p(x^{(i)}, z^{(i)};\theta) = \\ = \sum_{i=1}^m \ln \sum_{z^{(i)}} Q_i(z^{(i)}) \frac{p(x^{(i)}, z^{(i)};\theta)}{Q_i(z^{(i)})},\end{aligned} \tag{42.1}$$

where $Q_i(z^{(i)})$ is some unknown probability density function for $z^{(i)}$ with properties $Q_i(z^{(i)}) \geqslant 0$, $\sum\limits_{z^{(i)}} Q_i(z^{(i)}) = 1$. The expression inside the logarithm is the expectation of $\dfrac{p(x^{(i)}, z^{(i)};\theta)}{Q_i(z^{(i)})}$ given that $z^{(i)}$ drawn from the distribution Q_i:

$$\sum_{z^{(i)}} Q_i(z^{(i)}) \frac{p(x^{(i)}, z^{(i)};\theta)}{Q_i(z^{(i)})} = E_{z^{(i)} \sim Q_i}\left[\frac{p(x^{(i)}, z^{(i)};\theta)}{Q_i(z^{(i)})}\right] \tag{42.2}$$

(by definition, if the random variable Z has a density p, then

$$E[g(Z)] = \sum_z p(z) g(z)$$

for any function $g(Z)$).

Then Jensen's inequality

$$\ln E[X] \geqslant E[\ln X]$$

for the concave function $\ln X$ and equation (42.2) applied to the right hand side of the equation (42.1) gives

$$\begin{aligned}\sum_{i=1}^m \ln \sum_{z^{(i)}} Q_i(z^{(i)}) \frac{p(x^{(i)}, z^{(i)};\theta)}{Q_i(z^{(i)})} = \sum_{i=1}^m \ln E_{z^{(i)} \sim Q_i}\left[\frac{p(x^{(i)}, z^{(i)};\theta)}{Q_i(z^{(i)})}\right] \geqslant \\ \geqslant \sum_{i=1}^m E_{z^{(i)} \sim Q_i}\left[\ln \frac{p(x^{(i)}, z^{(i)};\theta)}{Q_i(z^{(i)})}\right] = \sum_{i=1}^m \sum_{z^{(i)}} Q_i(z^{(i)}) \ln \frac{p(x^{(i)}, z^{(i)};\theta)}{Q_i(z^{(i)})}.\end{aligned}$$

We just showed that

$$l(\theta) \geqslant \sum_{i=1}^{m} \sum_{z^{(i)}} Q_i(z^{(i)}) \ln \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})},$$

where the right hand side is a lower bound from the previous picture for the log-likelihood $l(\theta)$. It makes sense to have this lower bound as close to the log-likelihood as possible, which means that we should replace $\geqslant$ by $=$. But the theorem 2 from the previous section implies that we have equation instead of inequality if and only if the random variable $\frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}$ is constant, or

$$Q_i(z^{(i)}) \propto p(x^{(i)}, z^{(i)}; \theta).$$

By definition, $\sum\limits_{i=1}^{m} Q_i(z^{(i)}) = 1$, then

$$Q_i(z^{(i)}) = \frac{p(x^{(i)}, z^{(i)}; \theta)}{\sum\limits_{z^{(i)}} p(x^{(i)}, z^{(i)}; \theta)} = \frac{p(x^{(i)}, z^{(i)}; \theta)}{p(x^{(i)}; \theta)} = p(z^{(i)} \mid x^{(i)}; \theta).$$

Algorithm 13 General EM algorithm

```
1: repeat
2:     E-step: set
           $Q_i(z^{(i)}) = p(z^{(i)} \mid x^{(i)}; \theta)$
3:     M-step: optimize the lower bound with respect to $\theta$
           $\theta := \arg\max_{\theta} \sum_{i=1}^{m} \sum_{z^{(i)}} Q_i(z^{(i)}) \ln \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}$
4: until convergence
5: return $\theta$
```

By analogy with the k-means algorithm, define the distortion function

$$J(Q, \theta) = \sum_{i=1}^{m} \sum_{z^{i)}} Q_i(z^{i)}) \ln \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{i)})},$$

then $l(\theta) \geqslant J(Q, \theta)$ and EM can be viewed as a coordinate ascent algorithm on J where on the E-step we maximize J with respect to Q and on the M-step we maximize it with respect to θ.

Exercise. Prove that the algorithm converges, i.e. if we have consider $\theta^{(t)}$ and $\theta^{(t+1)}$ from two consequent iterations of the algorithm, then

$$l(\theta^{(t+1)}) \geqslant l(\theta^{(t)}).$$

43 EM algorithm for the mixture of Gaussians

To solve the optimization problem (40.1) we apply EM algorithm.

- **E-step** (guess values of $z^{(i)}$). With the assumptions

$$x^{(i)} \,|\, (z^{(i)} = j) \sim N(\mu_j, \Sigma_j),$$
$$z^{(i)} \sim \text{Multinomial}(\varphi) \Rightarrow p(z^{(i)} = j) = \varphi_j$$

  and the Bayes rule we determine the distribution

$$Q_i(z^{(i)} = j) = p(z^{(i)} = j \,|\, x^{(i)}; \varphi, \mu, \Sigma) = \frac{p(x^{(i)} \,|\, z^{(i)} = j) \cdot p(z^{(i)} = j)}{\sum\limits_{l=1}^{k} p(x^{(i)} \,|\, z^{(i)} = l) \cdot p(z^{(i)} = l)}.$$

  Denote

$$w_j^{(i)} = Q_i(z^{(i)} = j).$$

- **M-step** (update estimates for the parameters).

$$\begin{aligned} J(Q, \varphi, \mu, \Sigma) &= \sum_{i=1}^{m} \sum_{z^{(i)}} Q_i(z^{(i)}) \ln \frac{p(x^{(i)}, z^{(i)}; \varphi, \mu, \Sigma)}{Q_i(z^{(i)})} = \\ &= \sum_{i=1}^{m} \sum_{z^{(i)}} w_j^{(i)} \ln \frac{p(x^{(i)} \,|\, z^{(i)}; \mu, \Sigma) \cdot p(z^{(i)}; \varphi)}{w_j^{(i)}} = \\ &= \sum_{i=1}^{m} \sum_{j=1}^{k} w_j^{(i)} \ln \frac{\frac{1}{(2\pi)^{n/2}|\Sigma_j|^{1/2}} \exp\left(-\frac{1}{2}(x^{(i)} - \mu_j)^T \Sigma_j^{-1} (x^{(i)} - \mu_j)\right) \cdot \varphi_j}{w_j^{(i)}}. \end{aligned}$$

  Remember that there is an additional constraint

$$\sum_{j=1}^{k} \varphi_j = 1.$$

  To use this constraint we can write down the Lagrangian

$$L = J(Q, \varphi, \mu, \Sigma) - \beta \left(\sum_{j=1}^{k} \varphi_j - 1 \right)$$

  and set it derivatives with respect to parameters to zero. Doing this gives the estimates of parameters:

$$\nabla_{\varphi_j} L = 0 \Rightarrow \varphi_j = \frac{1}{m} \sum_{i=1}^{m} w_j^{(i)},$$
$$\nabla_{\mu_j} L = 0 \Rightarrow \mu_j = \frac{\sum\limits_{i=1}^{m} w_j^{(i)} x^{(i)}}{\sum\limits_{i=1}^{m} w_j^{(i)}},$$
$$\nabla_{\Sigma_j} L = 0 \Rightarrow \Sigma_j = \frac{\sum\limits_{i=1}^{m} w_j^{(i)} \cdot (x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum\limits_{i=1}^{m} w_j^{(i)}}.$$

The algorithm is summarized as follows:

Algorithm 14 EM algorithm for the mixture of Gaussians

```
1: Set the number of components $k$
2: Initialize values for the parameters $\varphi_j$, $\mu_j$, $\Sigma_j$, $j = \{1, \dots, k\}$
3: repeat
4:     E-step: set
           $$w_j^{(i)} = \frac{\frac{1}{(2\pi)^{n/2}|\Sigma_j|^{1/2}} \exp\left(-\frac{1}{2}(x^{(i)} - \mu_j)^T \Sigma_j^{-1} (x^{(i)} - \mu_j)\right) \cdot \varphi_j}{\sum\limits_{l=1}^{k} \frac{1}{(2\pi)^{n/2}|\Sigma_l|^{1/2}} \exp\left(-\frac{1}{2}(x^{(i)} - \mu_l)^T \Sigma_l^{-1} (x^{(i)} - \mu_l)\right) \cdot \varphi_l}$$
5:     M-step: update the parameters
           $$\varphi_j := \frac{1}{m} \sum_{i=1}^{m} w_j^{(i)}$$
           $$\mu_j := \frac{\sum_{i=1}^{m} w_j^{(i)} x^{(i)}}{\sum_{i=1}^{m} w_j^{(i)}}$$
           $$\Sigma_j := \frac{\sum_{i=1}^{m} w_j^{(i)} \cdot (x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum_{i=1}^{m} w_j^{(i)}}$$
6: until convergence
7: return $\varphi_j$, $\mu_j$, $\Sigma_j$, $j = \{1, \dots, k\}$
```

We will show how to derive φ_j in the M-step of the algorithm:

$$\nabla_{\varphi_j} L = 0 \Leftrightarrow \sum_{i=1}^{m} \frac{w_j^{(i)}}{\varphi_j} - \beta = 0 \Leftrightarrow \sum_{i=1}^{m} w_j^{(i)} - \beta \cdot \varphi_j = 0 \text{ for all } j.$$

Sum obtained equations by j:

$$\sum_{j=1}^{k} \sum_{i=1}^{m} w_j^{(i)} - \beta \sum_{j=1}^{k} \varphi_j = 0 \Leftrightarrow \sum_{i=1}^{m} \sum_{j=1}^{k} w_j^{(i)} - \beta = 0,$$

but $\sum\limits_{j=1}^{k} w_j^{(i)} = 1$, because $w_j^{(i)} = p(z^{(i)} = j \,|\, x^{(i)}; \varphi, \mu, \Sigma)$ are probabilities for multinomial random variable. Finally,

$$\beta = \sum_{i=1}^{m} 1 = m \Rightarrow \sum_{i=1}^{m} w_j^{(i)} - m\varphi_j = 0 \Rightarrow \varphi_j = \frac{1}{m} \sum_{i=1}^{m} w_j^{(i)}.$$

We compare Gaussian discriminant analysis and mixture of Gaussians in the following table.

Param.	**Gaussian discriminant analysis**	**Mixture of Gaussians**
φ_j	$\frac{1}{m}\sum_{i=1}^{m}\mathbb{1}\{y^{(i)}=j\}$	$\frac{1}{m}\sum_{i=1}^{m}w_j^{(i)}$
μ_j	$\frac{\sum_{i=1}^{m}\mathbb{1}\{y^{(i)}=j\}\cdot x^{(i)}}{\sum_{i=1}^{m}\mathbb{1}\{y^{(i)}=j\}}$	$\frac{\sum_{i=1}^{m}w_j^{(i)}\cdot x^{(i)}}{\sum_{i=1}^{m}w_j^{(i)}}$
Σ_j	$\frac{1}{m}\sum_{i=1}^{m}(x^{(i)}-\mu_{y^{(i)}})\cdot(x^{(i)}-\mu_{y^{(i)}})^T$	$\frac{\sum_{i=1}^{m}w_j^{(i)}\cdot(x^{(i)}-\mu_j)(x^{(i)}-\mu_j)^T}{\sum_{i=1}^{m}w_j^{(i)}}$

We can conclude that in the mixture of Gaussians we replace the percentage of each class by the probability $w_j^{(i)}$.

43.1 Python implementation

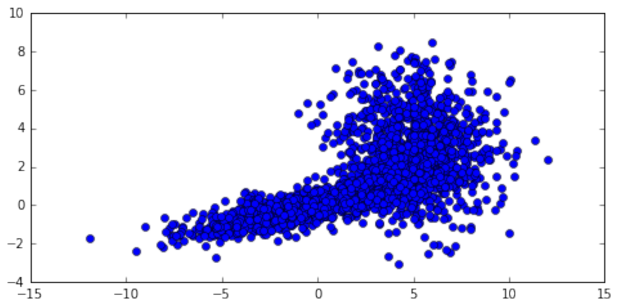
We implement EM algorithm for the mixture of Gaussians from scratch. First, we define the functions for E-step and M-step. Notice that we have tried to use some built-in `numpy` functions to avoid loops.

```
In [13]: def Estep(mu, sigma, phi):
             # calculate determinants of sigma's
             det_sigma = np.array([[np.linalg.det(sigma[i])]
                                   for i in range(k)])
             # calculate inverse matrices for sigma's
             inv_sigma = np.array([np.linalg.inv(sigma[i])
                                   for i in range(k)]).reshape(sigma.shape)
             # calculate Q(z) = p(x|z)*p(z)/p(x)
             pxz = np.array([
                     np.exp(
                         -0.5*np.diagonal(
                             np.dot(
                                 np.dot(X_train - mu[i,:], inv_sigma[i]),
                                 np.transpose(X_train - mu[i,:])
                             )
                         )
                     )/((2.0*np.pi)**(n/2.0)*det_sigma[i,0]**0.5)*phi[i,0]
                     for i in range(k)]).T
             pz = pxz/np.sum(pxz, axis=1).reshape((-1, 1))
             return pz
```

```
def Mstep(pz):
    pz_sum = np.sum(pz, axis=0).reshape((-1,1))
    # update parameters
    phi_new = pz_sum/m
    mu_new = np.transpose(np.dot(X_train.T, pz)/pz_sum.T)
    sigma_new = np.array([
            np.dot(np.array([
                        np.outer(X_train[j,:] - mu_new[i,:],
                                 X_train[j,:] - mu_new[i,:])
                        for j in range(m)]).reshape((m, -1)).T,
                   pz[:,i]).reshape((n,n))/pz_sum[i,0]
            for i in range(k)]).reshape((k,n,n))
    return mu_new, sigma_new, phi_new
```

We use the data from the previous section (generated as a mixture of two gaussian distributions):

```
In [14]: fig = plt.figure(figsize=(8, 8))
         ax = fig.add_subplot(1, 1, 1)
         ax.plot(X_train[:,0], X_train[:,1], 'o')
         plt.gca().set_aspect('equal', adjustable='box')
         plt.show()
```



Define the variables and parameters for our EM algorithm:

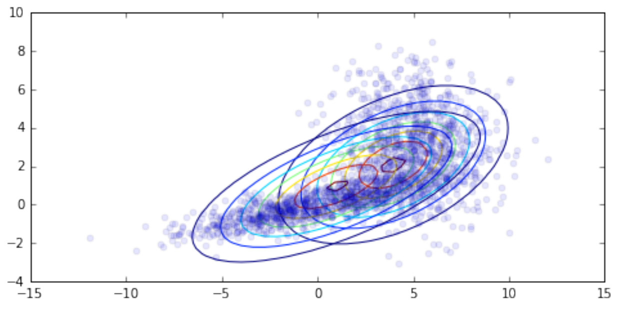
```
In [15]: # number of components
         k = 2
         # number of features
         n = X_train.shape[1]
         # number of training examples
         m = X_train.shape[0]
         # number of iterations
         niter = 10
         # initial values of phi
         phi = np.array([1.0/k]*k).reshape((k,-1))
         # initial values for mu and sigma
```

```
        mu = []
        sigma = []
        np.random.seed(234)
        for cl in range(k):
            mu.append(np.mean(X_train[np.random.choice(m, m/2),:], axis=0))
            sigma.append(np.identity(n))
        mu = np.array(mu).reshape((k, n))
        sigma = np.array(sigma).reshape((k, n, n))
```

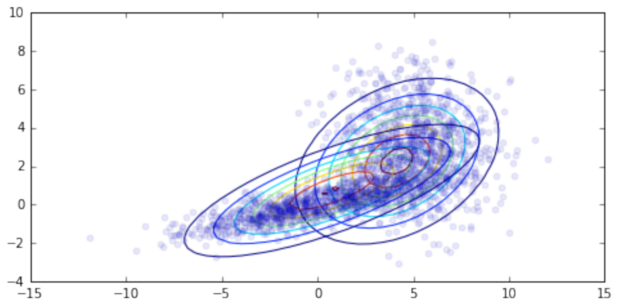
We train the mixture of Gaussians model and plot the fitted distributions after each iteration:

```
In [16]: for nit in range(niter):
             print 'Iteration ' + str(nit+1) + ':'
             pz = Estep(mu, sigma, phi)
             mu, sigma, phi = Mstep(pz)
             fig = plt.figure(figsize=(8, 8))
             ax = fig.add_subplot(1, 1, 1)
             ax.plot(X_train[:,0], X_train[:,1], 'o', alpha=.1, ms=5)
             for i in range(k):
                 mu_i = mu[i,:]
                 sigma_i = sigma[i]
                 sigma_i_inv = np.linalg.inv(sigma_i)
                 x = np.linspace(-15.0, 15.0)
                 y = np.linspace(-4.0, 10.0)
                 X, Y = np.meshgrid(x, y)
                 XX = np.array([X.ravel(), Y.ravel()]).T
                 P = np.exp(-0.5*np.diagonal(
                         np.dot(
                             np.dot(XX - mu_i, sigma_i_inv),
                             np.transpose(XX - mu_i)
                         )))/(2*np.pi*np.linalg.det(sigma_i)**0.5)
                 P = P.reshape(X.shape)
                 CS = plt.contour(X, Y, P)
             plt.gca().set_aspect('equal', adjustable='box')
             plt.show()
```

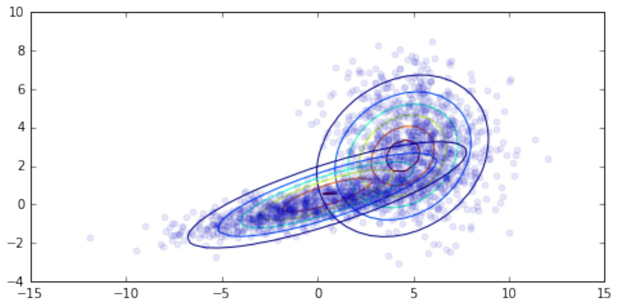
```
Iteration 1:
```



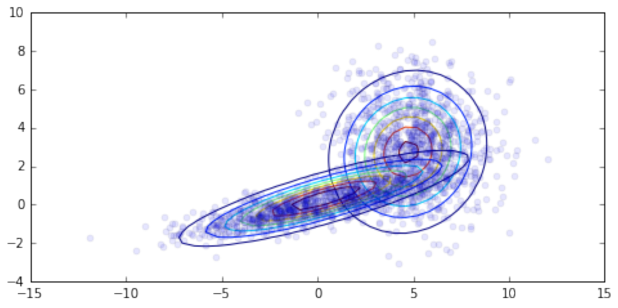
```
Iteration 2:
```



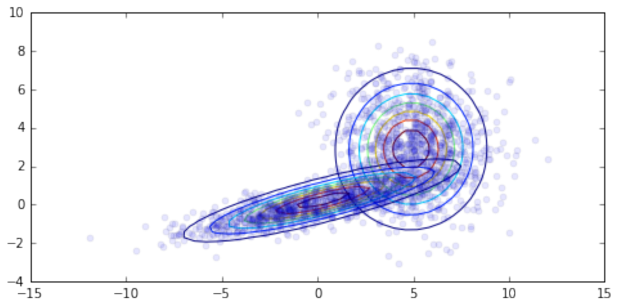
```
Iteration 3:
```



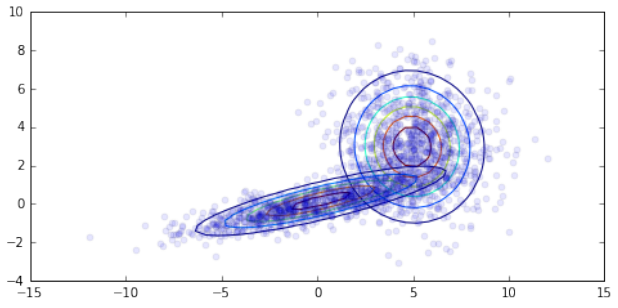
```
Iteration 4:
```



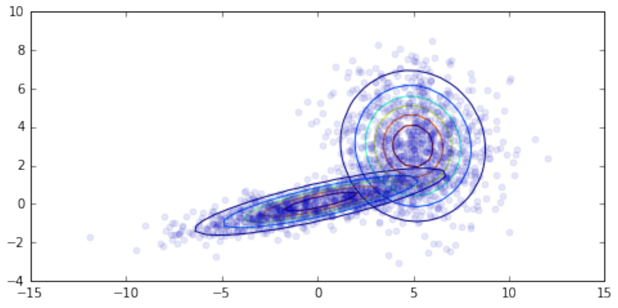
```
Iteration 5:
```



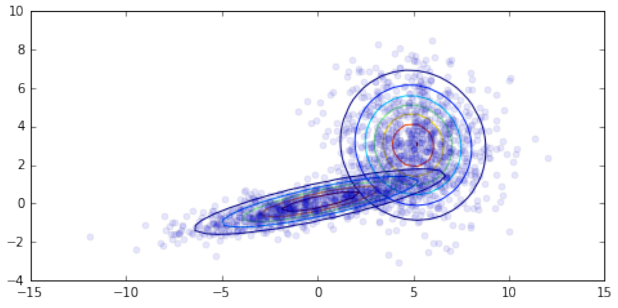
```
Iteration 6:
```



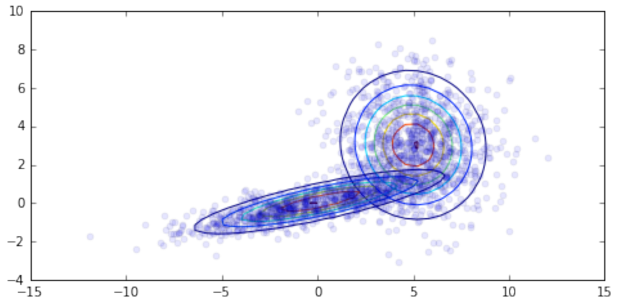
```
Iteration 7:
```



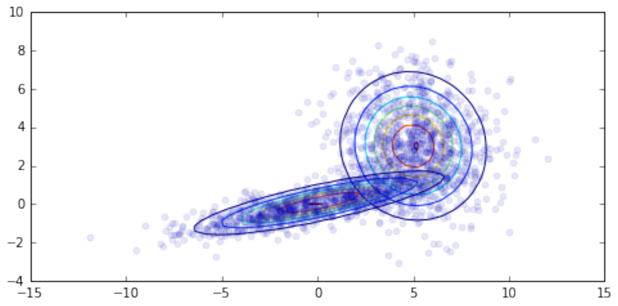
```
Iteration 8:
```



```
Iteration 9:
```



```
Iteration 10:
```



44 EM algorithm for the mixture of Naive Bayes

In the previous lectures we talked about Naive Bayes model applied to the text classification problem. Assuming we have a dataset of m documents: $\{x^{(1)}, \ldots, x^{(m)}\}$, $x^{(i)} \in \mathbb{R}^n$, and $x_j^{(i)} = \mathbb{1}\{\text{word } j \text{ appears in the document } i\}$. It means that each document in the database can be represented by the features vector

$$x^{(i)} = \begin{array}{c} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ \vdots \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \end{array} \begin{array}{c} a \\ as \\ advert \\ and \\ \vdots \\ claim \\ \vdots \\ light \\ \vdots \\ zip \end{array}$$

In this section we apply EM algorithm to build a mixture of Naive Bayes models. As before instead of class labels $y^{(i)}$ we introduce a latent random variable $z^{(i)} \in \{0, 1\}$, which means that we assume two clusters for the text dataset and

$$z^{(i)} \sim \text{Bernoulli}(\varphi).$$

We also make Naive Bayes assumption

$$p(x^{(i)} \,|\, z^{(i)}) = \prod_{j=1}^{n} p(x_j^{(i)} \,|\, z^{(i)}),$$

or more specifically,

$$p(x_j^{(i)} = 1 \,|\, z^{(i)} = 0) = \varphi_{j\,|\,z=0},$$
$$p(x_j^{(i)} = 1 \,|\, z^{(i)} = 1) = \varphi_{j\,|\,z=1}.$$

EM algorithm contains two steps:

- **E-step**.

$$w^{(i)} = p(z^{(i)} = 1 \,|\, x^{(i)}; \varphi_{j\,|\,z}, \varphi)$$

- **M-step**.

$$\varphi_{j\,|\,z=1} = \frac{\sum\limits_{i=1}^{m} w^{(i)} \mathbb{1}\{x_j^{(i)} = 1\}}{\sum\limits_{i=1}^{m} w^{(i)}}$$

$$\varphi_{j\,|\,z=0} = \frac{\sum\limits_{i=1}^{m} (1 - w^{(i)}) \mathbb{1}\{x_j^{(i)} = 1\}}{\sum\limits_{i=1}^{m} (1 - w^{(i)})}$$

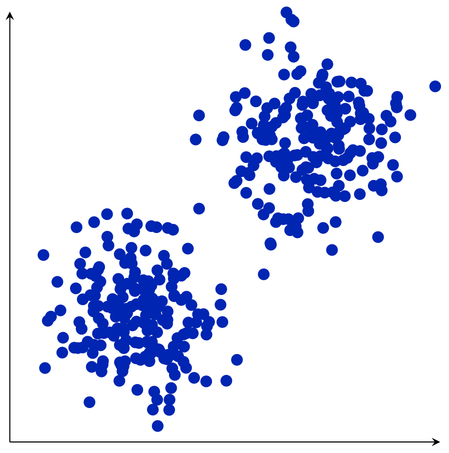
$$\varphi = \frac{\sum\limits_{i=1}^{m} w^{(i)}}{m}$$

The final remark is that if we apply this algorithm $w^{(i)}$'s will be close either to 0 or 1.

45 Factor analysis

45.1 Intuition

For the following dataset the mixture of Gaussians model can be very effective:



If the number of training example m is much bigger than the number of features n then the mixture of Gaussians is the best choice. In practice the situation when the number of features greater (or approximately the same) than number of training examples is very often. In this section we consider the algorithm that works fine for the case $n >> m$ or $n \approx m$.

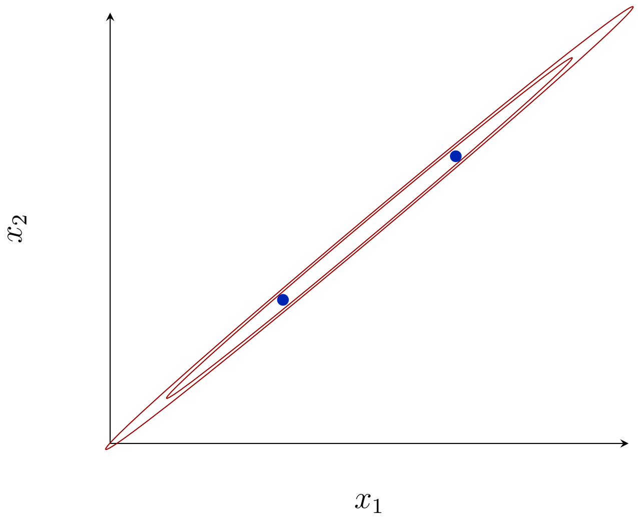
Our goal is to estimate the probability density $p(x)$ given a training set $\{x^{(1)}, \ldots, x^{(m)}\}$.

If we assume that $x \sim N(\mu, \Sigma)$, where $\Sigma \in \mathbb{R}^{n\times n}$, then maximum likelihood estimation gives a solution

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)},$$

$$\Sigma = \frac{1}{m}\sum_{i=1}^{m} (x^{(i)} - \mu)(x^{(i)} - \mu)^T.$$

If the number of samples is small the matrix Σ is singular (contains zero eigenvalues), which means that it is not invertible and calculation of the probability density gives $\frac{0}{0}$ undetermined form. The simple illustration of this situation is:



In this case $m = n = 2$ and the estimated density will be infinitely "long" along the axis through these two points. To fix this problem we can add some restrictions for Σ.

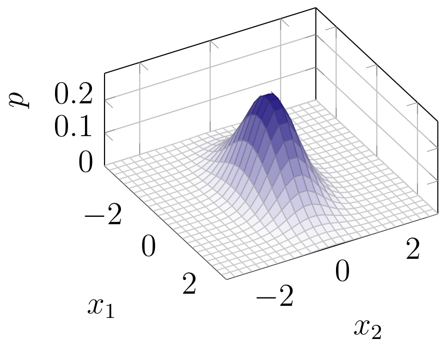
First option is to make Σ to be diagonal:

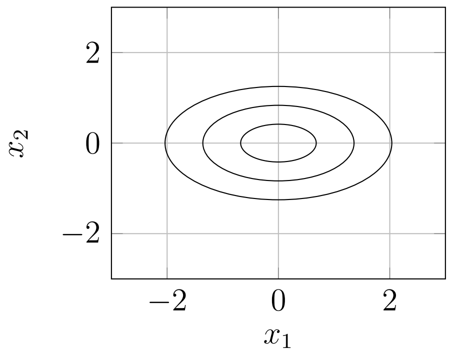
$$\Sigma = \begin{pmatrix} \sigma_1^2 & 0 & \cdots & 0 & 0 \\ 0 & \sigma_2^2 & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & \sigma_{n-1}^2 & 0 \\ 0 & 0 & \cdots & 0 & \sigma_n^2 \end{pmatrix} \in \mathbb{R}^{n\times n},$$

then maximum likelihood estimates gives

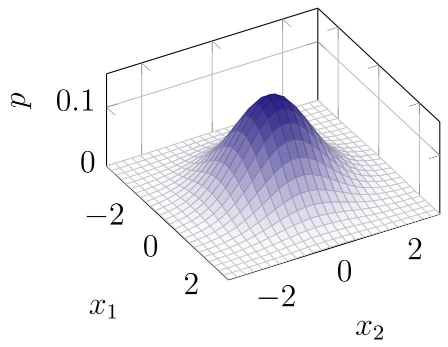
$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2$$

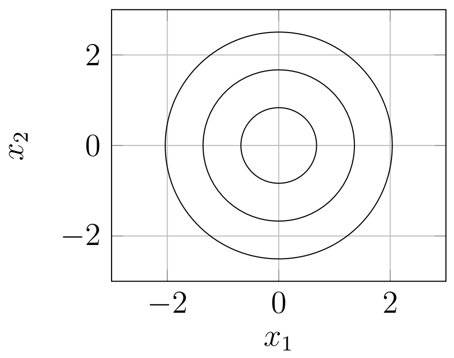
and main axes of Gaussian distribution will parallel to the coordinate axes:





More restrictive assumption could be $\Sigma = \sigma^2 I$, where I is a unit matrix, in this case the cross sections of Gaussian probability density become circular:





If we add such restrictions for the covariance matrix then we also assume that features in our training dataset are uncorrelated or independent which is not true in many cases. In the factor analysis we add some restrictions to avoid the singularity we described before, but we also try to catch some correlations between features in the dataset.

We introduce the latent variables $z \in N(\vec{0}, I)$, $z \in \mathbb{R}^d (d < n)$ and make the following assumptions:

$$x \,|\, z \sim N(\mu + \Lambda z, \Psi),$$

or, equivalently,

$$x = \mu + \Lambda z + \varepsilon,$$

where

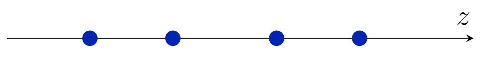
$$\varepsilon \sim N(\vec{0}, \Psi).$$

The parameters of the model are $\mu \in \mathbb{R}^n$, $\Lambda \in \mathbb{R}^{n\times d}$, $\Psi \in \mathbb{R}^{n\times n}$. We introduce an additional assumption that Ψ is a diagonal.

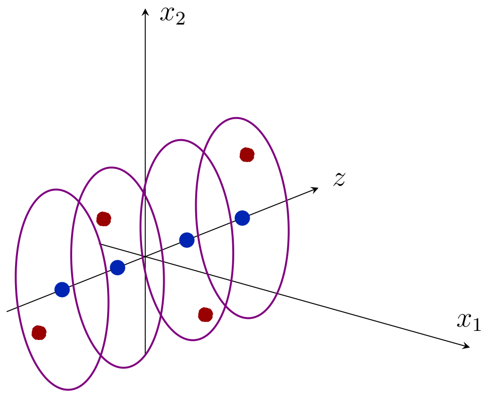
Example. For the case $z \in \mathbb{R}$, $x \in \mathbb{R}^2$ with the parameters

$$\Lambda = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \; \Psi = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}, \; \mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

the idea of the factor analysis is visualized as



$z \in \mathbb{R}$ (blue points)



$x \in \mathbb{R}^2$ (red points)

Similarly, we can take $z \in \mathbb{R}^2$ and $x \in \mathbb{R}^3$. Then the transformation $x = \mu + \Lambda z$ is a mapping of the 2D space to the 3D space, the transformation $x = \mu + \Lambda z + \varepsilon$ is the same mapping but with the added Gaussian noise.

45.2 Marginal and conditionals for Gaussians

Consider the vector partitioning of the gaussian random variable X

$$X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \sim N(\mu, \Sigma),$$

where $X_1 \in \mathbb{R}^r$, $X_2 \in \mathbb{R}^s$, $X \in \mathbb{R}^{r+s}$. Then

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}$$

and

$$\Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} = \begin{bmatrix} E[(X_1-\mu_1)(X_1-\mu_1)^T] & E[(X_1-\mu_1)(X_2-\mu_2)^T] \\ E[(X_2-\mu_2)(X_1-\mu_1)^T] & E[(X_2-\mu_2)(X_2-\mu_2)^T] \end{bmatrix},$$

where $\mu_1 \in \mathbb{R}^r$, $\mu_2 \in \mathbb{R}^s$, $\Sigma_{11} \in \mathbb{R}^{r\times r}$, $\Sigma_{12} \in \mathbb{R}^{r\times s}$, $\Sigma_{21} \in \mathbb{R}^{s\times r}$ and $\Sigma_{22} \in \mathbb{R}^{s\times s}$.

Then the random variable $X_1 \sim N(\mu_1, \Sigma_{11})$ is a gaussian random variable with the probability density

$$p(x_1) = \int\limits_{x_2} p(x_1, x_2)dx_2.$$

The conditional density function can be calculated as

$$p(x_1 \,|\, x_2) = \frac{p(x_1, x_2)}{p(x_2)},$$

where numerator and denominator are known. Using this formula it can be shown that

$$X_1 \,|\, X_2 \sim N(\mu_{1\,|\,2}, \Sigma_{1\,|\,2}),$$

where

$$\mu_{1\,|\,2} = \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \mu_2) \tag{45.1}$$

and

$$\Sigma_{1\,|\,2} = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}. \tag{45.2}$$

These facts will be useful for the next section.

45.3 Factor analysis model

To find the solution for the factor analysis model we combine the latent random variable $z \sim N(\vec{0}, I)$ and the observed variable $x = \mu + \Lambda z + \varepsilon$, $\varepsilon \sim N(\vec{0}, \Psi)$ to

$$\begin{pmatrix} z \\ x \end{pmatrix} \sim N(\mu_{zx}, \Sigma).$$

Then $\mu_{zx} = E\begin{bmatrix} z \\ x \end{bmatrix} = \begin{bmatrix} Ez \\ Ex \end{bmatrix} = \begin{bmatrix} \vec{0} \\ \mu \end{bmatrix}$ is a $(d+n)$-dimensional vector and covariance matrix

$$\Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix},$$

where

$$\begin{aligned}\Sigma_{11} &= E[(z-Ez)(z-Ez)^T] = \mathrm{Cov}(z) = I,\\ \Sigma_{12} &= E[(z-Ez)(x-Ex)^T] = \Lambda^T,\\ \Sigma_{21} &= E[(x-Ex)(z-Ez)^T] = \Lambda,\\ \Sigma_{22} &= E(x-Ex)(x-Ex)^T = \Lambda\Lambda^T+\Psi.\end{aligned}$$

As an example, we show how to prove the formula for Σ_{21}:

$$\begin{aligned}\Sigma_{21} = E[(x-Ex)(z-Ez)^T] = E(\mu+\Lambda z+\varepsilon-\mu)\cdot z^T) =\\ = E\left[\Lambda zz^T\right] + E[\varepsilon z^T] = \Lambda E[zz^T] = \Lambda.\end{aligned}$$

In other words,

$$\begin{pmatrix} z \\ x \end{pmatrix} \sim N\left(\begin{bmatrix} \vec{0} \\ \mu \end{bmatrix}, \begin{bmatrix} I & \Lambda^T \\ \Lambda & \Lambda\Lambda^T+\Psi \end{bmatrix}\right),$$

and

$$x \sim N(\mu, \Lambda\Lambda^T+\Psi). \tag{45.3}$$

To find the parameters for this model we should maximize the log-likelihood given the training set:

$$\begin{aligned}l(\Lambda,\mu,\Psi) = \sum_{i=1}^m \ln p(x^{(i)};\Lambda,\mu,\Psi) =\\ = \sum_{i=1}^m \frac{1}{(2\pi)^{n/2}\cdot|\Lambda\Lambda^T+\Psi|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^T(\Lambda\Lambda^T+\Psi)^{-1}(x-\mu)\right).\end{aligned}$$

The standard approach (finding derivatives and setting them to zero) does not work in this case: it is impossible to solve analytically the obtained system of equations.

45.4 EM steps for factor analysis

In order to find the maximum of the likelihood function we apply the EM algorithm:

- **E-step**. As before
$$Q_i(z^{(i)}) = p(z^{(i)} \,|\, x^{(i)};\Lambda,\mu,\Psi),$$
where
$$z^{(i)} \,|\, x^{(i)} \sim N(\mu_{z^{(i)}\,|\,x^{(i)}}, \Sigma_{z^{(i)}\,|\,x^{(i)}}).$$
To calculate $\mu_{z^{(i)}\,|\,x^{(i)}}$ and $\Sigma_{z^{(i)}\,|\,x^{(i)}}$ we use the formulas (45.1) and (45.2):
$$\mu_{z^{(i)}\,|\,x^{(i)}} = \Lambda^T(\Lambda\Lambda^T+\Psi)^{-1}(x^{(i)}-\mu) \tag{45.4}$$
and
$$\Sigma_{z^{(i)}\,|\,x^{(i)}} = I - \Lambda^T(\Lambda\Lambda^T+\Psi)^{-1}\Lambda. \tag{45.5}$$
- **M-step**. To maximize the log-likelihood we maximize the lower bound for the log-likelihood
$$\Lambda,\mu,\Psi = \arg\max_{\Lambda,\mu,\Psi} \sum_{i=1}^m \int_{z^{(i)}} Q_i(z^{(i)}) \ln \frac{p(x^{(i)},z^{(i)};\Lambda,\mu,\Psi)}{Q_i(z^{(i)})} dz^{(i)}.$$

Notice that we have replaced sum by integration because $z^{(i)}$ is a continuous random variable now. By definition of the expected value, we have

$$\int\limits_{z^{(i)}} Q_i(z^{(i)}) \ln \frac{p(x^{(i)}, z^{(i)}; \Lambda, \mu, \Psi)}{Q_i(z^{(i)})} dz^{(i)} = E_{z^{(i)} \sim Q_i} \left[\ln \frac{p(x^{(i)}, z^{(i)}; \Lambda, \mu, \Psi)}{Q_i(z^{(i)})} \right] =$$
$$= E_{z^{(i)} \sim Q_i} \left[\ln p(x^{(i)} \mid z^{(i)}; \Lambda, \mu, \Psi) \right] + E_{z^{(i)} \sim Q_i} \left[\ln \frac{p(z^{(i)})}{Q_i(z^{(i)})} \right].$$

The second term in the last expression does not depend on the parameters ($Q_i(z^{(i)})$ is a distribution that is fixed on the E-step) which means that our goal is to maximize

$$\sum_{i=1}^{m} E_{z^{(i)} \sim Q_i} \left[\ln p(x^{(i)} \mid z^{(i)}; \Lambda, \mu, \Psi) \right] =$$
$$= \sum_{i=1}^{m} E \left[\ln \frac{1}{(2\pi)^{n/2} |\Psi|^{1/2}} \exp \left(-\frac{1}{2} (x^{(i)} - \mu - \Lambda z^{(i)})^T \Psi^{-1} (x^{(i)} - \mu - \Lambda z^{(i)}) \right) \right] =$$
$$= \sum_{i=1}^{m} E \left[-\frac{n}{2} \ln(2\pi) - \frac{1}{2} \ln |\Psi| - \frac{1}{2} (x^{(i)} - \mu - \Lambda z^{(i)})^T \Psi^{-1} (x^{(i)} - \mu - \Lambda z^{(i)}) \right]$$

(by the equation (45.3)). As an example, we show how to maximize with respect to Λ, in this case the first two terms are constants and the third term can be replaced by trace because Ψ is a diagonal matrix:

$$\nabla_\Lambda \sum_{i=1}^{m} E \left[\frac{1}{2} (x^{(i)} - \mu - \Lambda z^{(i)})^T \Psi^{-1} (x^{(i)} - \mu - \Lambda z^{(i)}) \right] =$$
$$= \sum_{i=1}^{m} \nabla_\Lambda E \left[-\operatorname{tr} \frac{1}{2} {z^{(i)}}^T \Lambda^T \Psi^{-1} \Lambda z^{(i)} + \operatorname{tr} {z^{(i)}}^T \Lambda^T \Psi^{-1} (x^{(i)} - \mu) \right] =$$
$$= \sum_{i=1}^{m} \nabla_\Lambda E \left[-\operatorname{tr} \frac{1}{2} \Lambda^T \Psi^{-1} \Lambda z^{(i)} {z^{(i)}}^T + \operatorname{tr} \Lambda^T \Psi^{-1} (x^{(i)} - \mu) {z^{(i)}}^T \right] =$$
$$= \sum_{i=1}^{m} E \left[-\Psi^{-1} \Lambda z^{(i)} {z^{(i)}}^T + \Psi^{-1} (x^{(i)} - \mu) {z^{(i)}}^T \right].$$

In these transformations we used the facts

$$\nabla_A \operatorname{tr} ABA^T C = CAB + C^T AB,$$
$$\operatorname{tr} AB = \operatorname{tr} BA.$$

By solving an equation $\nabla_\Lambda = 0$ we obtain

$$\sum_{i=1}^{m} \Lambda E \left[z^{(i)} {z^{(i)}}^T \right] = \sum_{i=1}^{m} (x^{(i)} - \mu) E \left[{z^{(i)}}^T \right]$$
$$\Rightarrow \Lambda = \left(\sum_{i=1}^{m} (x^{(i)} - \mu) E \left[{z^{(i)}}^T \right] \right) \left(\sum_{i=1}^{m} E \left[z^{(i)} {z^{(i)}}^T \right] \right)^{-1}.$$

But on the E-step we calculated (formulas (45.4) and (45.5))

$$E \left[{z^{(i)}}^T \right] = \mu_{z^{(i)} \mid x^{(i)}}$$

and for any random variable $z \sim N(\mu, \Sigma)$

$$\Sigma = E\left[zz^T\right] - (Ez)(Ez)^T \Rightarrow E\left[zz^T\right] = \Sigma + (Ez)(Ez)^T,$$

which implies

$$E\left[z^{(i)}{z^{(i)}}^T\right] = \Sigma_{z^{(i)}\,|\,x^{(i)}} + \mu_{z^{(i)}\,|\,x^{(i)}}\mu^T_{z^{(i)}\,|\,x^{(i)}}.$$

We leave the calculation of μ and Ψ as an exercise. Final formulas for all parameters on the M-step become

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)},$$

$$\Lambda = \left(\sum_{i=1}^{m}(x^{(i)} - \mu)\mu^T_{z^{(i)}\,|\,x^{(i)}}\right)\left(\sum_{i=1}^{m}\Sigma_{z^{(i)}\,|\,x^{(i)}} + \mu_{z^{(i)}\,|\,x^{(i)}}\mu^T_{z^{(i)}\,|\,x^{(i)}}\right)^{-1},$$

$$\Psi = \operatorname{diag}\frac{1}{m}\sum_{i=1}^{m}\Big(x^{(i)}{x^{(i)}}^T - x^{(i)}\mu^T_{z^{(i)}\,|\,x^{(i)}}\Lambda^T - \Lambda\mu_{z^{(i)}\,|\,x^{(i)}}{x^{(i)}}^T +$$

$$+\Lambda\left(\mu_{z^{(i)}\,|\,x^{(i)}}\mu^T_{z^{(i)}\,|\,x^{(i)}} + \Sigma_{z^{(i)}\,|\,x^{(i)}}\right)\Lambda^T\Big).$$

Exercise. Get an expressions for μ and Ψ by analogy with Λ.

45.5 Python implementation

Import necessary libraries:

```
In [1]: import numpy as np
        import pandas as pd
        import random
        import math
        import sklearn.datasets as ds
        import matplotlib.pyplot as plt
```

Factor analysis algorithm is implemented in the `scikit-learn` package:

```
In [2]: from sklearn.decomposition import FactorAnalysis
```

To illustrate how you can train this algorithm we use `boston` data from the `scikit-learn`.

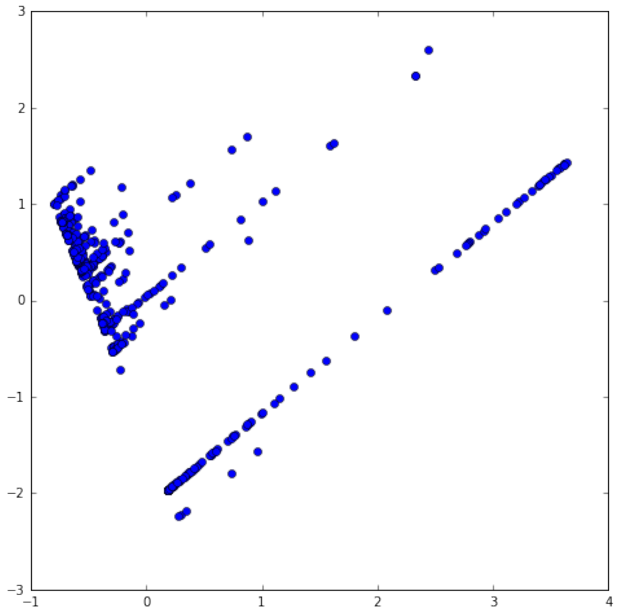
```
In [3]: boston = ds.load_boston()
        X = boston.data
        X.shape

Out[3]: (506, 13)
```

Fit the model:

```
In [4]: model = FactorAnalysis(n_components=2, random_state=324)
        model.fit(X)
        X_reduced = model.transform(X)
```

and visualize the results:

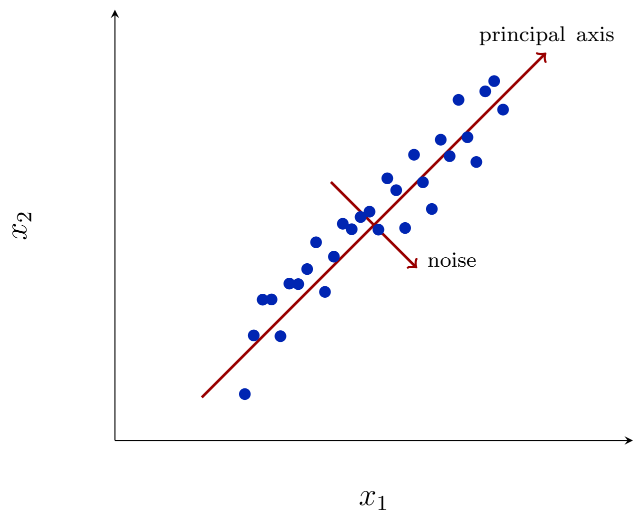
```
In [5]: fig = plt.figure(figsize=(8, 8))
        ax = fig.add_subplot(1, 1, 1)
        ax.plot(X_reduced[:,0], X_reduced[:,1], 'o')
        plt.show()
```



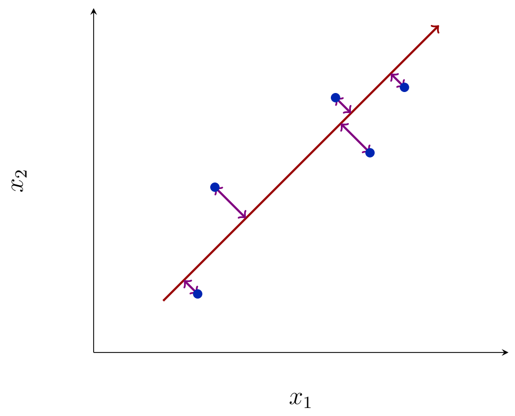
46 Principal Component Analysis

Compared to the factor analysis, the principal component analysis is not probabilistic algorithm. The main purpose of this algorithm is dimensionality reduction (though there are a lot of other applications like data vizualization, data compression, anomaly detection and distance calculation).

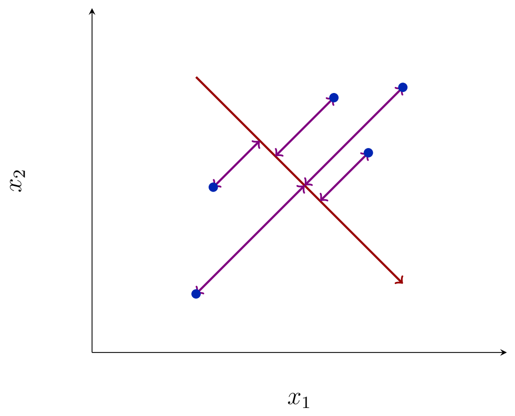
Assuming that our given data are n-dimensional: $\{x^{(1)}, \ldots, x^{(m)}\}$, $x^{(i)} \in \mathbb{R}^n$, our goal is to reduce these data to k-dimensional data ($k < n$). The intuition of such reduction is the following:



On this picture 2-dimensional dataset could be reduced to the 1-dimensional. Consider the simple dataset that can give the idea how to complete this procedure:



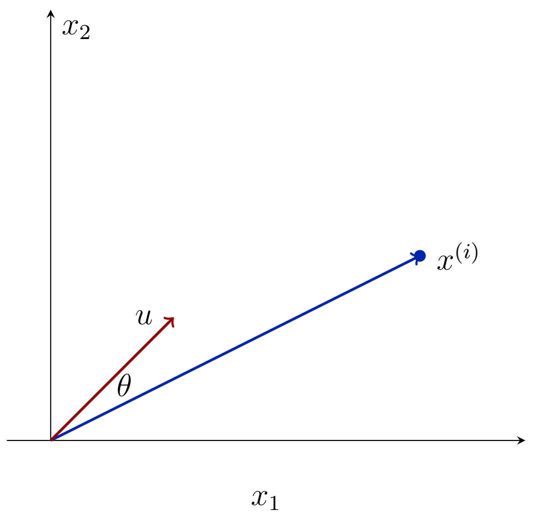
Good principal axis



Bad principal axis

We could observe that the best principle axis has the minimal sum of distances from the training data points. But a minimising the distance between point $x^{(i)}$ to the principle axis is equivalent to minimizing the angle between radius vector of $x^{(i)}$ and the principal axis. Let u be unit directional vector of the principal axis, then the cosine of angle between radius vector $x^{(i)}$ and u can be calculated as

$$\cos\theta = \frac{{x^{(i)}}^T \cdot u}{||x^{(i)}||},\ ||u|| = 1.$$



Our main goal is to find u with the following condition:

$$u = \arg\max_{u:||u||=1} \frac{1}{m}\sum_{i=1}^{m}({x^{(i)}}^T u)^2 = \arg\max_{u:||u||=1} \frac{1}{m}\sum_{i=1}^{m}(u^T x^{(i)})({x^{(i)}}^T u) =$$

$$= \arg\max_{u:||u||=1} u^T \left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}{x^{(i)}}^T\right] u.$$

Denote

$$\Sigma = \sum_{i=1}^{m} x^{(i)}{x^{(i)}}^T = X^T X \in \mathbb{R}^{n\times n},$$

then we should solve the optimization problem

$$\max_u u^T \Sigma u, \text{ given } u^T u = 1.$$

The Lagrangian

$$L(u,\lambda) = u^T\Sigma u - \lambda(u^T u - 1)$$

and it derivative with respect to u gives

$$\nabla_u L = \Sigma u - \lambda u = 0 \Leftrightarrow \Sigma u = \lambda u.$$

It implies that u is the principal eigenvector of $\Sigma = \dfrac{1}{m}\displaystyle\sum_{i=1}^{m} x^{(i)}{x^{(i)}}^T$.

More generally, if we want k-dimensional subspace, choose $u_1,\ldots,u_k$ to be k top eigenvectors of Σ (corresponding to k highest eigenvalues). Then for each training point $x^{(i)} \in \mathbb{R}^n$ the new representation will be

$$z^{(i)} = \begin{bmatrix} u_1^T x^{(i)} \\ u_2^T x^{(i)} \\ \vdots \\ u_k^T x^{(i)} \end{bmatrix} \in \mathbb{R}^k.$$

One of the most efficient way to find the eigenvectors of the matrix Σ is to use **singular value decomposition (SVD)** for the design matrix $X \in \mathbb{R}^{m \times n}$:

$$X = UDV^T,$$

where $U \in \mathbb{R}^{m \times m}$, $V \in \mathbb{R}^{n \times n}$ and $D \in \mathbb{R}^{m \times n}$ is a diagonal matrix:

$$D = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & \sigma_{n-1} & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & \sigma_n & 0 & \cdots & 0 \end{bmatrix}$$

(matrix D has this form if $n > m$, otherwise, we have rows of zeros below the last nonzero row). Additionally, columns of the matrix U are eigenvectors of the matrix XX^T and columns of the matrix V are eigenvectors of the matrix X^TX. In most cases, finding SVD for the matrix X is very fast that allows to find top k eigenvectors of the matrix $\Sigma = X^TX$ very effectively. Another interesting fact is that for many applications a lot of singular values become zero that gives us natural dimensionality reduction (we could take number of nonzero eigenvalues as a dimensionality k of the reduced space). Finally, we formulate the PCA algorithm (notice that we have some preprocessing step in this algorithm, which is very important to implement before we calculate SVD for the matrix X).

Algorithm 15 PCA algorithm

```
1: Define the dimensionality $k$ for the reduced space
2: **Zero out mean**. Calculate
       $$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$
   and replace $x^{(i)}$ with $x^{(i)} - \mu$
3: **Normalization to unit variance**. Calculate
       $$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m} \left(x_j^{(i)}\right)^2$$
   and replace $x_j^{(i)}$ with $\frac{x_j^{(i)}}{\sigma_j}$ for all $j \in \{1, \dots, n\}$
4: Find SVD for the design matrix $X$:
       $$X = UDV^T$$
5: Set $V = [k$ left columns of the matrix $V]$
6: Calculate the matrix product
       $$Z = XV$$
7: **return** $Z$ (new design matrix of size $m \times k$)
```

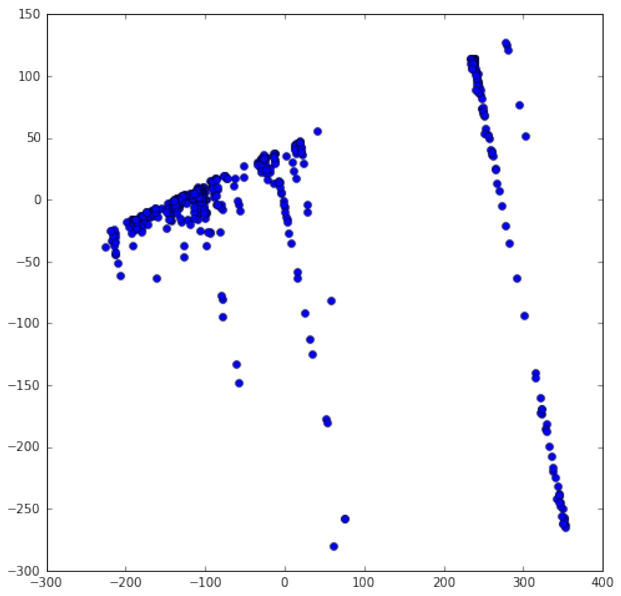
To compare different unsupervised algorithm we discussed in the lectures we combine them in the following table

	Probabilistic	**Not probabilistic**
Subspace	Factor analysis	PCA
Clustering	Mixture of Gaussians	K-means

46.1 Python implementation

The results of PCA algorithm is very close to the results obtained from the Factor Analysis algorithm:

```
In [6]: from sklearn.decomposition import PCA

In [7]: model = PCA(n_components=2)
        model.fit(X)
        X_reduced = model.transform(X)

In [8]: fig = plt.figure(figsize=(8, 8))
        ax = fig.add_subplot(1, 1, 1)
        ax.plot(X_reduced[:,0], X_reduced[:,1], 'o')
        plt.show()
```



47 Latent Semantic Indexing (LSI)

Latent Semantic Indexing (LSI) is the Principal Component Analysis algorithm applied to the text data. The main approach to the text machine learning problem is to construct binary feature vector where each feature represents the presence of some specific word:

$$x^{(i)} = \begin{array}{c} \left[\begin{array}{c} 0 \\ 1 \\ 0 \\ 1 \\ \vdots \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{array}\right] \end{array} \begin{array}{c} a \\ as \\ advert \\ and \\ \vdots \\ claim \\ \vdots \\ light \\ \vdots \\ zip \end{array}$$

The feature vector could be very high dimensional, for example, $x^{(i)} \in \mathbb{R}^{10000}$, then in the Principal Component Analysis algorithm we should find the eigenvalues of the matrix $\Sigma = X^T X \in \mathbb{R}^{10000\times 10000}$, which is extremely large matrix. Notice that when we apply PCA to such kind of data usually we skip preprocessing steps (zero our mean and normalization to unit variance).

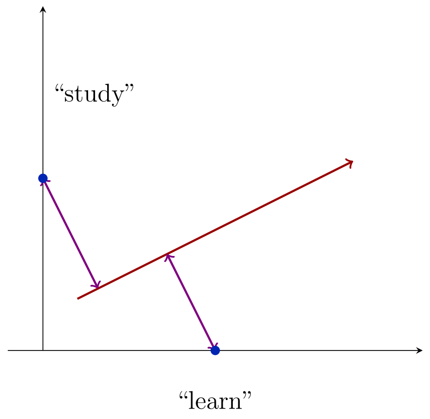
We can use SVD and find the principal components for our text data. Notice that the main goal of LSI is not a dimensionality reduction but rather calculating the similarity between documents. One general approach to measuring the similarity is to calculate cosine distance: given two documents $x^{(i)}$ and $x^{(j)}$ the cosine distance is determined as

$$\text{similarity}(x^{(i)}, x^{(j)}) = \cos\theta = \frac{{x^{(i)}}^T \cdot x^{(j)}}{||x^{(i)}|| \cdot ||x^{(j)}||},$$

where θ is the angle between two documents (angle between radius vectors of $x^{(i)}$ and $x^{(j)}$). The numerator of this similarity is

$${x^{(i)}}^T \cdot x^{(j)} = \sum_{l=1}^{n} x_l^{(i)} \cdot x_l^{(j)} = \sum_{l=1}^{n} \mathbb{1}\{\text{documents } i \text{ and } j \text{ both contain word } l\}.$$

One of the disadvantage of this approach is that if we have different words of the same meaning, for example, if one documents has a word “study” and the second document contains a word “learn”, then the similarity will be zero. When we apply PCA to our text data and find the principal components we would have something like



After the projection on the principal axes similar documents become closer to each other.

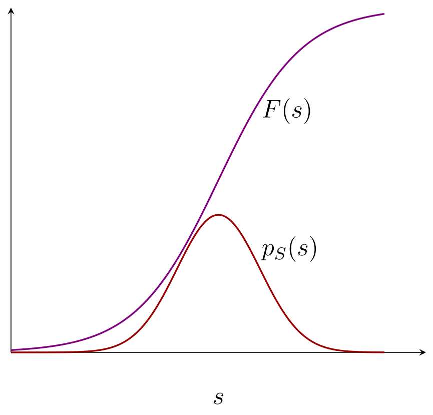
48 Independent Component Analysis (ICA)

For the given random variable S with the probability density function $p_S(s)$, the **cumulative distribution function (cdf)** is defined by

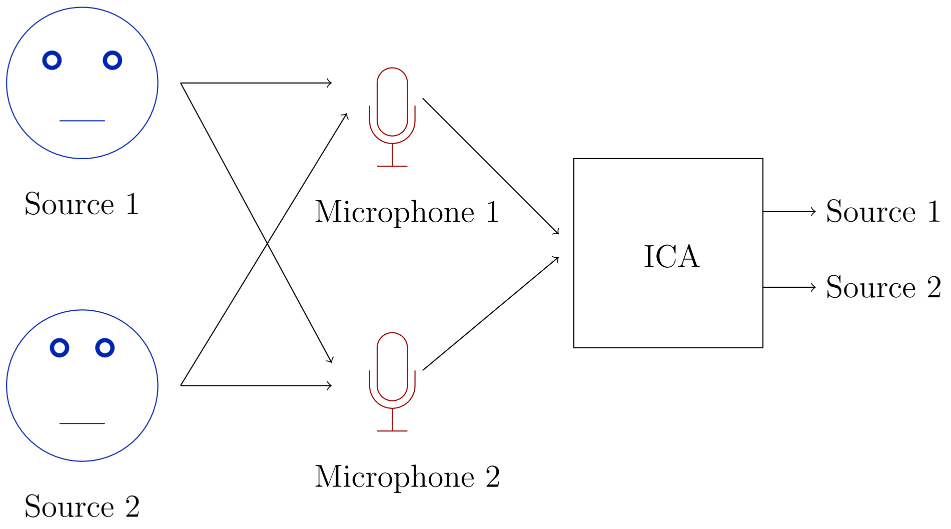
$$F(s) = P(S \leqslant s) = \int\limits_{-\infty}^{s} p_S(t)dt.$$

From this definition the obvious connection between pdf and cdf is

$$p_S(s) = F'(s). \tag{48.1}$$



Consider the following problem: we use n microphones to record n speakers placed in the same room, the resulted signal at time i is denoted by $x^{(i)} \in \mathbb{R}^n$, where $x_j^{(i)}$ is a signal on the microphone j at time i. Also denote by $s_j^{(i)}$ a signal from speaker j at time i, $s^{(i)} \in \mathbb{R}^n$ (originally, $s^{(i)}$ are not known).



We assume that

$$x^{(i)} = As^{(i)},$$

where A is an unknown square matrix (**mixing matrix**), i.e.

$$x_j^{(i)} = \sum_k A_{jk} s_k^{(i)}.$$

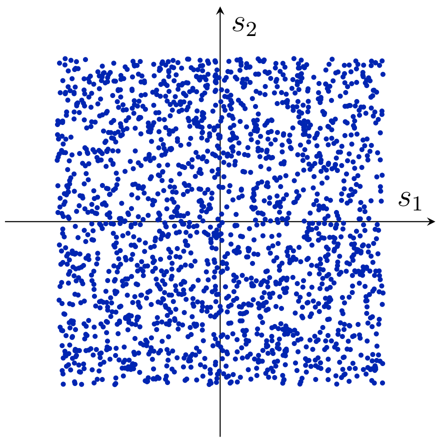
Our goal is to find **unmixing matrix** $W = A^{-1}$:

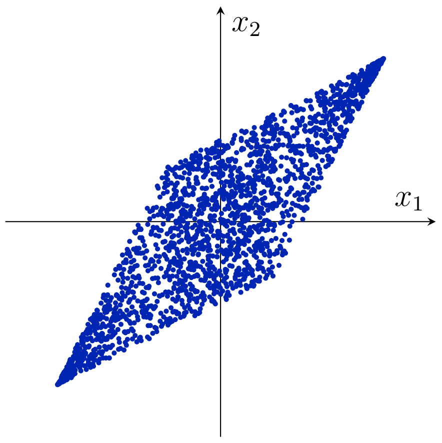
$$W = \begin{bmatrix} \longleftarrow w_1^T \longrightarrow \\ \vdots \\ \longleftarrow w_n^T \longrightarrow \end{bmatrix},$$

so that $s^{(i)} = Wx^{(i)}$.

We assume that

$$s_j^{(i)} \sim \text{Uniform } [-1, 1]$$





Analysing the problem can give an idea about ambiguities we have obtained. The first ambiguity is related to the fact that if we shuffle speakers then the output signal does not change. The second ambiguity is related to the signal sign: we cannot define if the signal is positive or negative. It turns out that if the original sources s are not Gaussian, then we do not have other ambiguities and we will be able to recover the matrix W. Otherwise, if the original signals s are Gaussians, we will have another ambiguity that is related to the invariance of Gaussian distribution to different rotations. In fact, in this situation we will have an arbitrary rotational component that does not allow us to recover the original sources.

Exercise. Show that if $s^{(i)} \sim N(0, I)$, then it is impossible to recover the original sources using ICA.

Assuming that we know the density function for $s \in \mathbb{R}^n$ and $s = Wx \Leftrightarrow x = W^{-1}s = As$, the density function for x is defined as

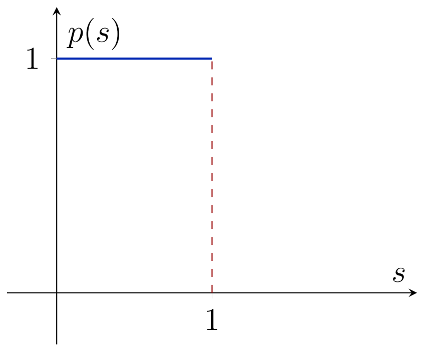
$$p_X(x) = p_S(Wx) \cdot |W|.$$

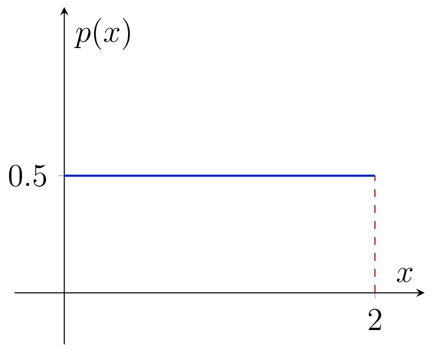
Example. Consider the uniform random variable $s \sim$ Uniform $[0,1]$ with density

$$p_S(s) = \mathbb{1}\{0 \leqslant s \leqslant 1\}$$

and the random variable $x = 2s$. In this example, $A = 2$, $W = \frac{1}{2}$, then

$$p_X(x) = \mathbb{1}\{0 \leqslant x \leqslant 2\} \cdot \frac{1}{2}.$$





Finally, we can formulate the Independent Component Analysis algorithm. For the independent original sources we have

$$p(s) = \prod_{j=1}^{n} p_S(s_j),$$

then

$$p(x) = \left[\prod_{j=1}^{n} p_S(w_j^T x)\right] \cdot |W|,$$

where $W = A^{-1} = \begin{bmatrix} \longleftarrow w_1^T \longrightarrow \\ \vdots \\ \longleftarrow w_n^T \longrightarrow \end{bmatrix}$ and $s_j = w_j^T x$.

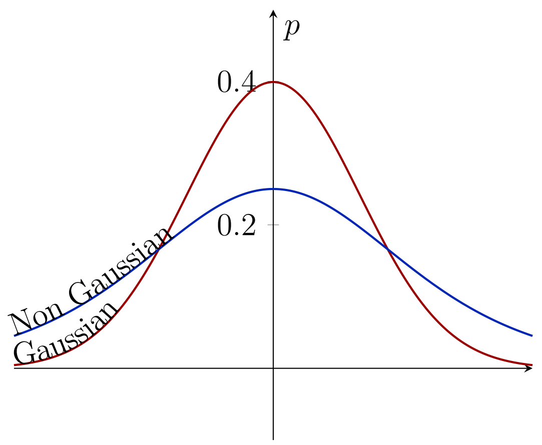
When we choose $p_S(s_j)$ we should make sure that it is not Gaussian. One of the option is to choose cdf first and after use the formula (48.1). It is convenient to choose

$$F(s) = \frac{1}{1 + e^{-s}},$$

then

$$p(s) = F'(s) = \frac{e^{-s}}{(1 + e^{-s})^2}$$

(compared to the Gaussian distribution the density of this distribution has fatter tails):



Another option is to choose Laplace distribution $p(s) = \frac{1}{2}e^{-|s|}$.

Given the set of output signals $\{x^{(1)}, \dots, x^{(m)}\}$ we write down the log-likelihood for our parameters:

$$l(W) = \sum_{i=1}^{m} \ln \left(\left[\prod_{j=1}^{n} p_S(w_j^T x^{(i)}) \right] \cdot |W| \right) = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} \ln p_S(w_j^T x^{(i)}) + \ln |W| \right).$$

By taking derivatives with respect to W and using the fact

$$\nabla_W |W| = |W| \left(W^{-1}\right)^T$$

we obtain:

$$\nabla_W l(W) = \sum_{i=1}^{m} \left(\begin{bmatrix} 1 - \dfrac{2}{1+\exp(-w_1^T x^{(i)})} \\ \vdots \\ 1 - \dfrac{2}{1+\exp(-w_n^T x^{(i)})} \end{bmatrix} {x^{(i)}}^T + (W^T)^{-1} \right).$$

Using this gradient we can write down one step of the stochastic gradient descent for the output signal $x^{(i)}$

$$W := W + \alpha \cdot \left(\begin{bmatrix} 1 - \dfrac{2}{1+\exp(-w_1^T x^{(i)})} \\ \vdots \\ 1 - \dfrac{2}{1+\exp(-w_n^T x^{(i)})} \end{bmatrix} {x^{(i)}}^T + (W^T)^{-1} \right).$$

After we find W we can recover unknown original signals as $s^{(i)} = W x^{(i)}$.

The final remark is there are a lot of applications of the ICA algorithm:

- EEG cap: split signals from different parts of the brain (for example, split brain signal to heart-beat signal, eyeblink signal and so on);
- independent component of images.

References

[1] J. Platt. "Fast Training of Support Vector Machines using Sequential Minimal Optimization", *in Advances in Kernel Methods - Support Vector Learning, B. Scholkopf, C. Burges, A. Smola, eds., MIT Press*, 1998.

[2] T. Hastie and R. Tibshirani. "Generalized additive models", *in Statistical Science, Vol.1, No.3*, 1998, pp. 297-318