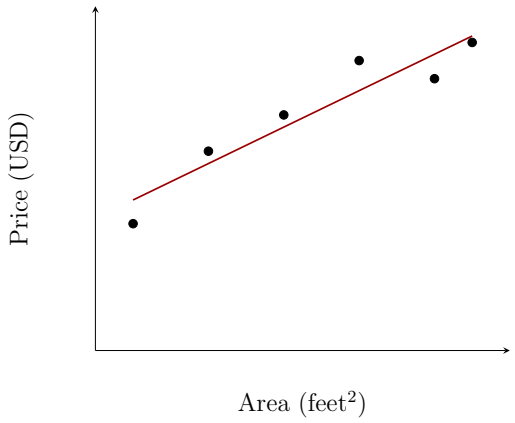
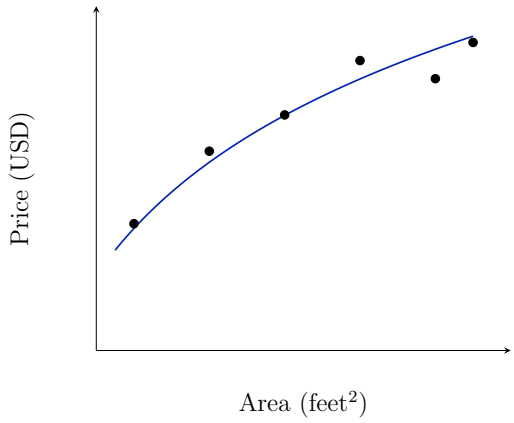
Lecture 2

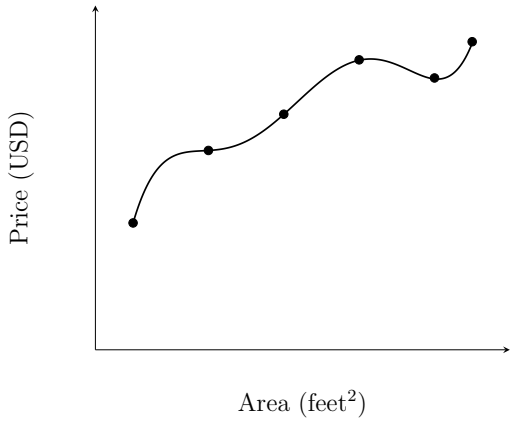
1. Locally weighted regression
2. Linear regression: probabilistic interpretation
3. Logistic regression
4. Perceptron

1 Locally weighted regression

Consider three different models for our previous example.







- underfitting
- good fit
- overfitting

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

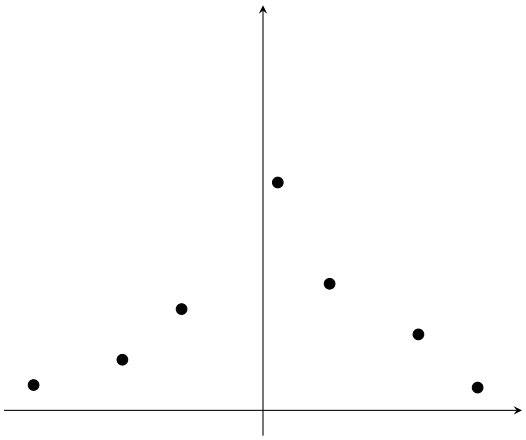
As we can see we can increase number of parameters if we increase the number of samples.

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.

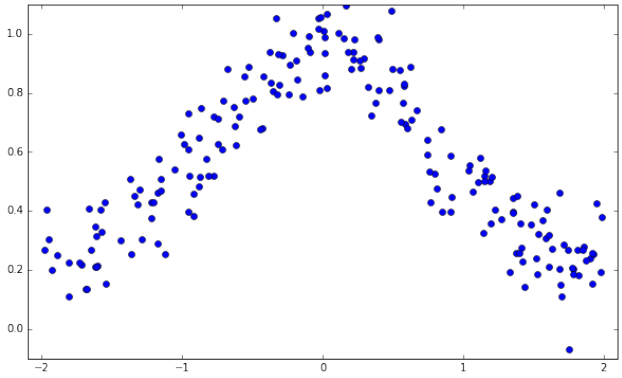
1.1 Python implementation

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]
```

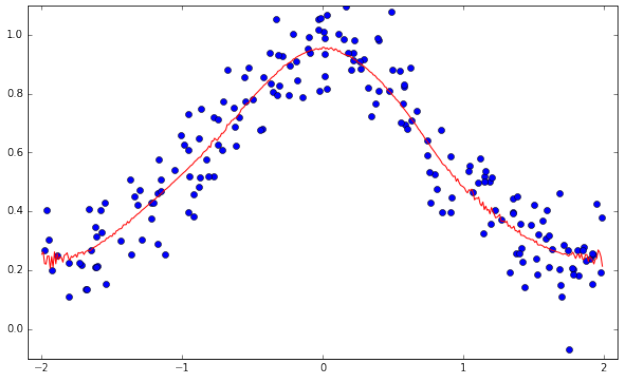
Notice that we should fit the weighted linear regression for each test example separately.

```
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()
```



2 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.

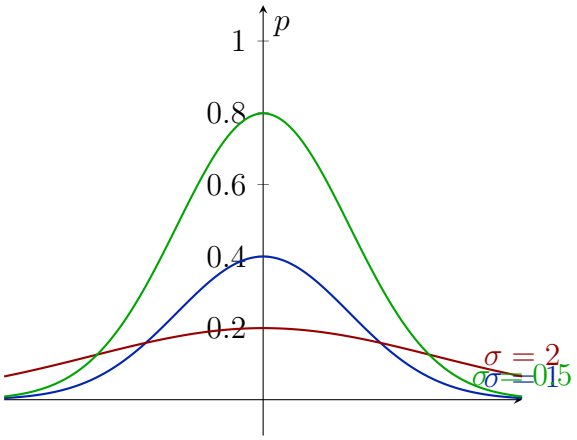
We assume that $y^{(i)} = \theta^T x^{(i)} + \varepsilon^{(i)}$, where $\varepsilon^{(i)}$ is an error term distributed normally:

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

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

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

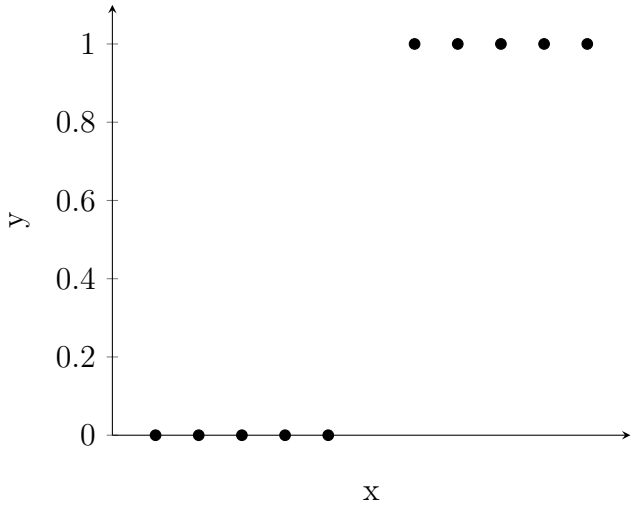
Notice that the standard deviation σ is not important for the optimization.

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?

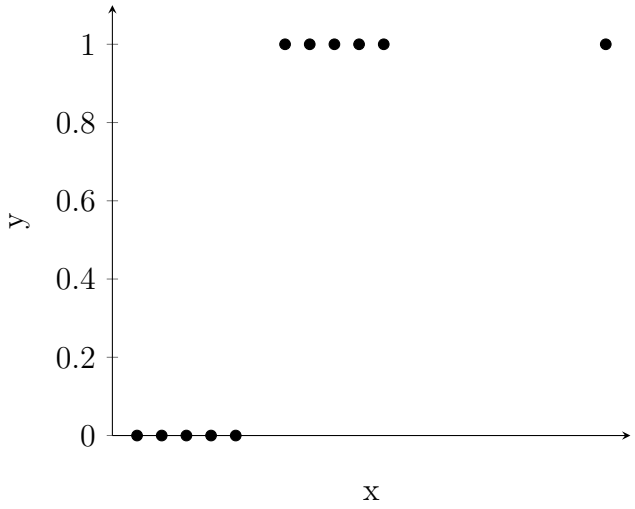
Hint: $p(y|x;\mu) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y-\mu)^2}{2\sigma^2}\right)$

3 Logistic regression

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

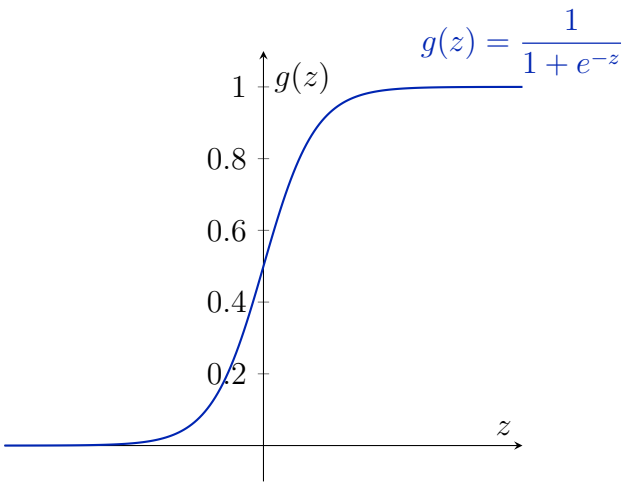


One way to solve this problem is to train linear regression, and after we take some threshold for the straight line. Sometimes it works, but in general it is not a good idea. Here is the example, when this does not work.



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:

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



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

$$p(y = 1|x; \theta) = h_\theta(x)$$
$$p(y = 0|x; \theta) = 1 - h_\theta(x),$$

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

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

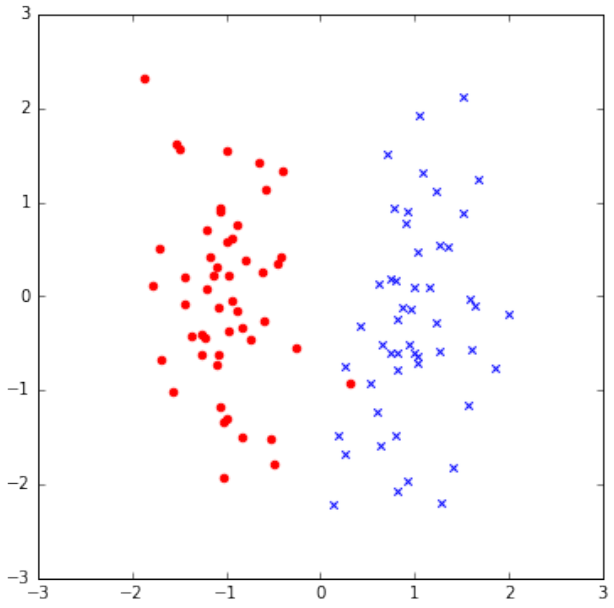
which is exactly the same rule as for least square regression, except that $h_\theta(x)$ is different.

3.1 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()
```



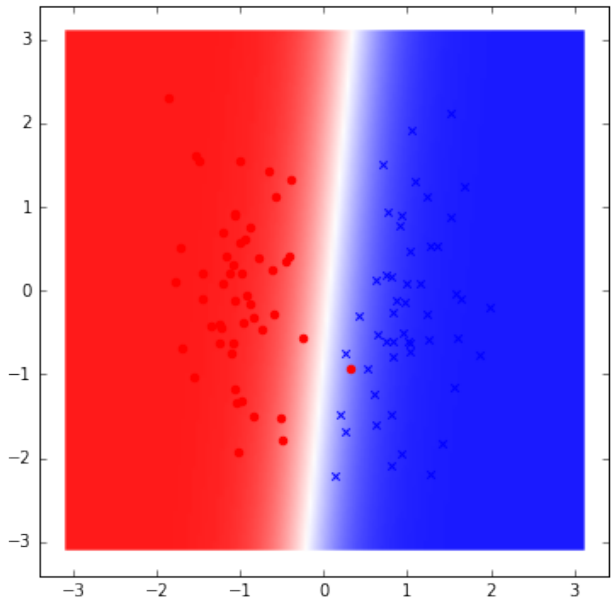
In similar way we train the logistic regression model from `scikit-learn` package:

```
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))
```

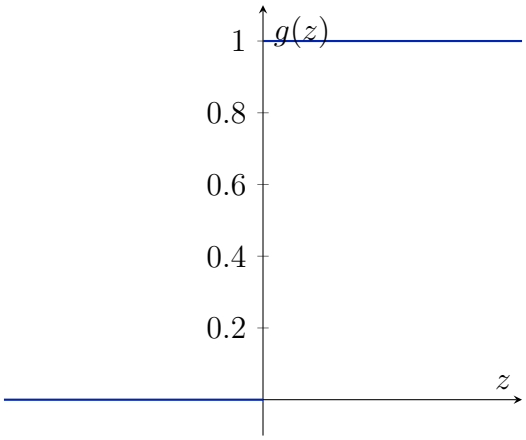
```
x1 = np.arange(-3.1, 3.1, 0.05)
x2 = np.arange(-3.1, 3.1, 0.05)
x1,x2 = np.meshgrid(x1, x2)
y_pred = sigmoid(x1,x2,model.intercept_[0],
                 model.coef_[0][0],model.coef_[0][1])
extent = -3.1, 3.1, -3.1, 3.1

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()
```



4 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.

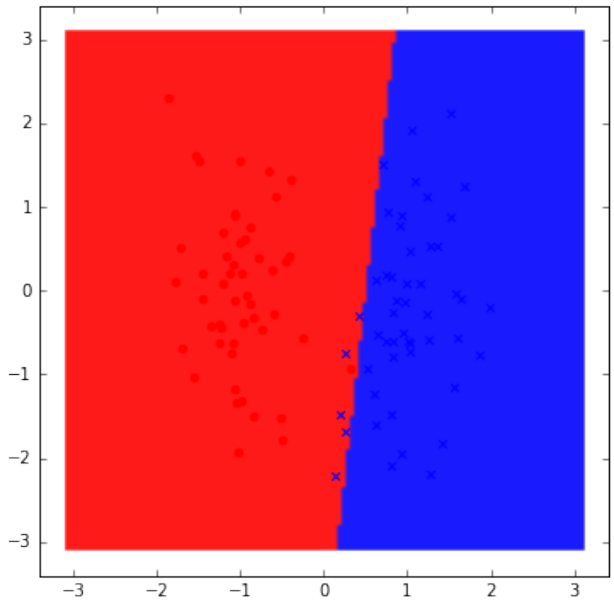
4.1 Python implementation

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()
```


3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3