Programming assignment (Linear models, Optimization)

In this programming assignment you will implement a linear classifier and train it using stochastic gradient descent modifications and numpy.

In [1]:
```
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
```

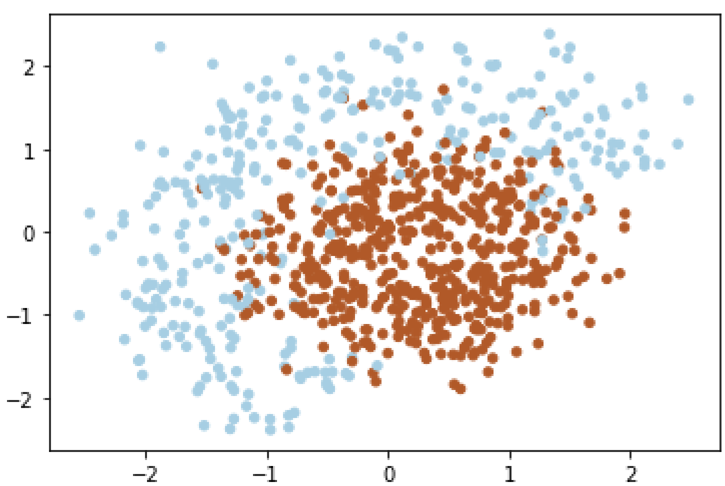
In [2]:
```
import sys
sys.path.append("..")
import grading
grader = grading.Grader(assignment_key="UaHtvpEFEee0XQ6wjK-hZg",
                      all_parts=["xU7U4", "HyTF6", "uNidL", "ToK7N", "GBdgZ", "dl
```

In [3]:
```
# token expires every 30 min
COURSERA_TOKEN = 'q1AzcK0aD21Hi8xh'
COURSERA_EMAIL = 'a.dushyant@gmail.com'
```

Two-dimensional classification

To make things more intuitive, let's solve a 2D classification problem with synthetic data.

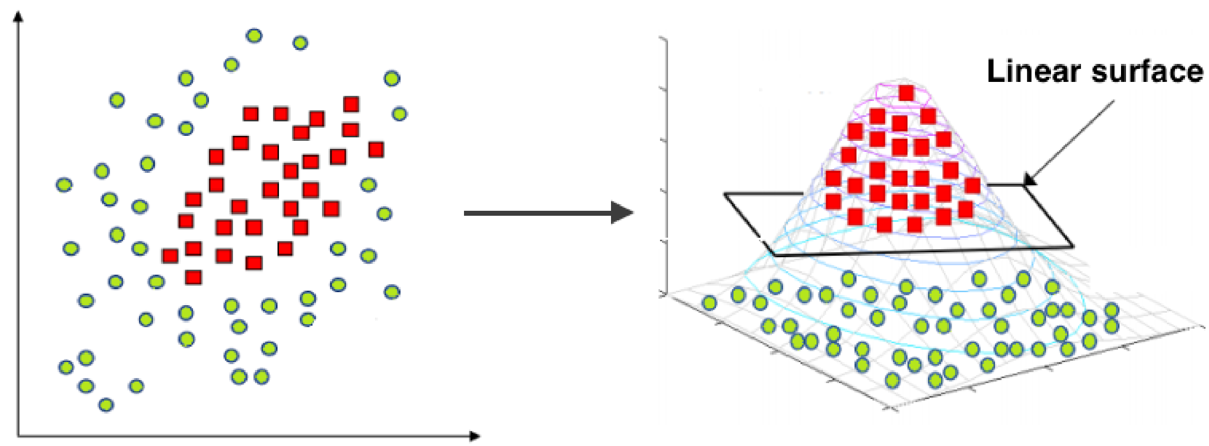
In [4]:
```
with open('train.npy', 'rb') as fin:
    X = np.load(fin)

with open('target.npy', 'rb') as fin:
    y = np.load(fin)

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired, s=20)
plt.show()
```



Task

Features

As you can notice the data above isn't linearly separable. Since that we should add features (or use non-linear model). Note that decision line between two classes have form of circle, since that we can add quadratic features to make the problem linearly separable. The idea under this displayed on image below:



In [5]:
```python
def expand(X):
    """
    Adds quadratic features.
    This expansion allows your linear model to make non-linear separation.

    For each sample (row in matrix), compute an expanded row:
    [feature0, feature1, feature0^2, feature1^2, feature0*feature1, 1]

    :param X: matrix of features, shape [n_samples,2]
    :returns: expanded features of shape [n_samples,6]
    """
    X_expanded = np.zeros((X.shape[0], 6))

    # TODO:<your code here>
    X_expanded[:,0], X_expanded[:,1] = X[:,0],X[:,1]
    X_expanded[:,2], X_expanded[:,3]= X[:,0]**2, X[:,1]**2
    X_expanded[:,4], X_expanded[:,5] = X[:,0]*X[:,1], np.ones(X.shape[0])

    return X_expanded
```

In [6]:
```python
X_expanded = expand(X)
```

Here are some tests for your implementation of `expand` function.

In [7]:
```
# simple test on random numbers

dummy_X = np.array([
        [0,0],
        [1,0],
        [2.61,-1.28],
        [-0.59,2.1]
    ])

# call your expand function
dummy_expanded = expand(dummy_X)

# what it should have returned:   x0       x1       x0^2     x1^2     x0*x1    1
dummy_expanded_ans = np.array([[ 0.    ,  0.    ,  0.    ,  0.    ,  0.    ,  1.
                               [ 1.    ,  0.    ,  1.    ,  0.    ,  0.    ,  1.
                               [ 2.61  , -1.28  ,  6.8121,  1.6384, -3.3408,  1.
                               [-0.59  ,  2.1   ,  0.3481,  4.41  , -1.239 ,  1.

#tests
assert isinstance(dummy_expanded,np.ndarray), "please make sure you return numpy
assert dummy_expanded.shape == dummy_expanded_ans.shape, "please make sure your s
assert np.allclose(dummy_expanded,dummy_expanded_ans,1e-3), "Something's out of c

print("Seems legit!")
```

```
Seems legit!
```

Logistic regression

To classify objects we will obtain probability of object belongs to class '1'. To predict probability we will use output of linear model and logistic function:

$$a(x; w) = \langle w, x \rangle$$
$$P(y = 1 \;\big|\; x, \, w) = \frac{1}{1 + \exp(-\langle w, x \rangle)} = \sigma(\langle w, x \rangle)$$

In [8]:
```
def probability(X, w):
    """
    Given input features and weights
    return predicted probabilities of y==1 given x, P(y=1|x), see description abo

    Don't forget to use expand(X) function (where necessary) in this and subseque

    :param X: feature matrix X of shape [n_samples,6] (expanded)
    :param w: weight vector w of shape [6] for each of the expanded features
    :returns: an array of predicted probabilities in [0,1] interval.
    """

    # TODO:<your code here>
    z = np.dot(X,w)
    a = 1./(1+np.exp(-z))
    return np.array(a)
```

In [9]:
```
dummy_weights = np.linspace(-1, 1, 6)
ans_part1 = probability(X_expanded[:1, :], dummy_weights)[0]
```

In [10]:
```
## GRADED PART, DO NOT CHANGE!
grader.set_answer("xU7U4", ans_part1)
```

In [11]:
```
# you can make submission with answers so far to check yourself at this stage
grader.submit(COURSERA_EMAIL, COURSERA_TOKEN)
```

```
Submitted to Coursera platform. See results on assignment page!
```

In logistic regression the optimal parameters w are found by cross-entropy minimization:

Loss for one sample:
$$l(x_i, y_i, w) = - \left[{y_i \cdot log P(y_i = 1 \, | \, x_i,w) + (1-y_i) \cdot log (1-P(y_i = 1\, | \, x_i,w))}\right]$$

Loss for many samples:
$$L(X, \vec{y}, w) = {1 \over \ell} \sum_{i=1}^\ell l(x_i, y_i, w)$$

In [12]:
```
def compute_loss(X, y, w):
    """
    Given feature matrix X [n_samples,6], target vector [n_samples] of 1/0,
    and weight vector w [6], compute scalar loss function L using formula above.
    Keep in mind that our loss is averaged over all samples (rows) in X.
    """
    # TODO:<your code here>
    l = X.shape[0]
    a = probability(X, w)

    cross_entropy = y*np.log(a) +(1-y)*np.log(1-a)
    cost = -np.sum(cross_entropy)/float(l)

    cost = np.squeeze(cost)      # To make sure your cost's shape is what we expe
    assert(cost.shape == ())

    return cost
```

In [13]:
```
# use output of this cell to fill answer field
ans_part2 = compute_loss(X_expanded, y, dummy_weights)
```

In [14]:
```
## GRADED PART, DO NOT CHANGE!
grader.set_answer("HyTF6", ans_part2)
```

In [15]:
```
# you can make submission with answers so far to check yourself at this stage
grader.submit(COURSERA_EMAIL, COURSERA_TOKEN)
```

```
Submitted to Coursera platform. See results on assignment page!
```

Since we train our model with gradient descent, we should compute gradients.

To be specific, we need a derivative of loss function over each weight [6 of them].

$$\nabla_w L = \frac{1}{\ell} \sum_{i=1}^{\ell} \nabla_w l(x_i, y_i, w)$$

We won't be giving you the exact formula this time — instead, try figuring out a derivative with pen and paper.

As usual, we've made a small test for you, but if you need more, feel free to check your math against finite differences (estimate how L changes if you shift w by 10^{-5} or so).

In [16]:
```
def compute_grad(X, y, w):
    """
    Given feature matrix X [n_samples,6], target vector [n_samples] of 1/0,
    and weight vector w [6], compute vector [6] of derivatives of L over each wei
    Keep in mind that our loss is averaged over all samples (rows) in X.
    """

    # TODO<your code here>
    m = X.shape[0]
    A = probability(X, w)
    dZ = A - y
    #cost = compute_loss(X, y, w)
    dW = np.dot(dZ, X) / float(m)

    return dW
```

In [17]:
```
# use output of this cell to fill answer field
ans_part3 = np.linalg.norm(compute_grad(X_expanded, y, dummy_weights))
```

In [18]:
```
## GRADED PART, DO NOT CHANGE!
grader.set_answer("uNidL", ans_part3)
```

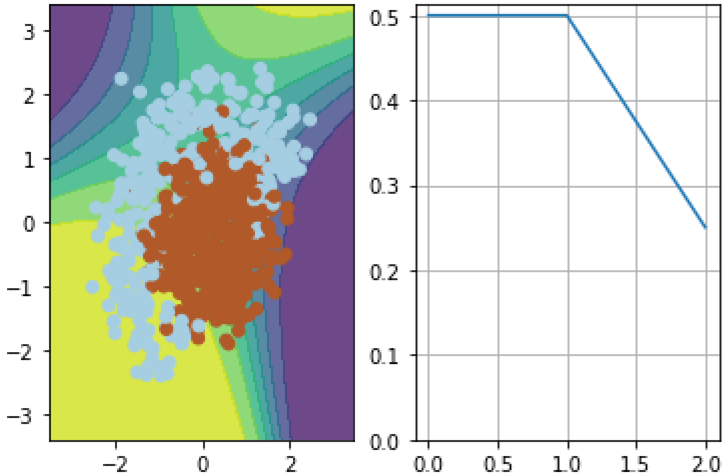
In [19]:
```
# you can make submission with answers so far to check yourself at this stage
grader.submit(COURSERA_EMAIL, COURSERA_TOKEN)
```

```
Submitted to Coursera platform. See results on assignment page!
```

Here's an auxiliary function that visualizes the predictions:

In [20]:
```
from IPython import display

h = 0.01
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

def visualize(X, y, w, history):
    """draws classifier prediction with matplotlib magic"""
    Z = probability(expand(np.c_[xx.ravel(), yy.ravel()]), w)
    Z = Z.reshape(xx.shape)
    plt.subplot(1, 2, 1)
    plt.contourf(xx, yy, Z, alpha=0.8)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())

    plt.subplot(1, 2, 2)
    plt.plot(history)
    plt.grid()
    ymin, ymax = plt.ylim()
    plt.ylim(0, ymax)
    display.clear_output(wait=True)
    plt.show()
```

In [21]:
```
visualize(X, y, dummy_weights, [0.5, 0.5, 0.25])
```



Training

In this section we'll use the functions you wrote to train our classifier using stochastic gradient descent.

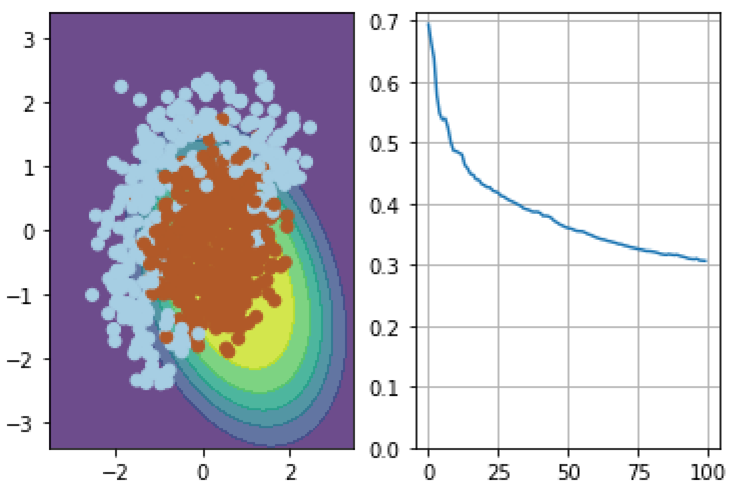
You can try change hyperparameters like batch size, learning rate and so on to find the best one, but use our hyperparameters when fill answers.

Mini-batch SGD

Stochastic gradient descent just takes a random batch of m samples on each iteration, calculates a gradient of the loss on it and makes a step:

$$w_t = w_{t-1} - \eta \dfrac{1}{m} \sum_{j=1}^{m} \nabla_w l(x_{i_j}, y_{i_j}, w_t)$$

In [22]:
```
# please use np.random.seed(42), eta=0.1, n_iter=100 and batch_size=4 for determi

np.random.seed(42)
w = np.array([0, 0, 0, 0, 0, 1])

eta= 0.1 # learning rate

n_iter = 100
batch_size = 4
loss = np.zeros(n_iter)
plt.figure(figsize=(12, 5))

for i in range(n_iter):
    ind = np.random.choice(X_expanded.shape[0], batch_size)
    loss[i] = compute_loss(X_expanded, y, w)
    if i % 10 == 0:
        visualize(X_expanded[ind, :], y[ind], w, loss)

    # Keep in mind that compute_grad already does averaging over batch for you!
    # TODO:<your code here>
    dW = compute_grad(X_expanded[ind, :], y[ind], w)
    w = w - eta * dW
    visualize(X, y, w, loss)
    plt.clf()
```



```
<Figure size 432x288 with 0 Axes>
```

In [23]:
```
# use output of this cell to fill answer field

ans_part4 = compute_loss(X_expanded, y, w)
```

In [24]:
```
## GRADED PART, DO NOT CHANGE!
grader.set_answer("ToK7N", ans_part4)
```

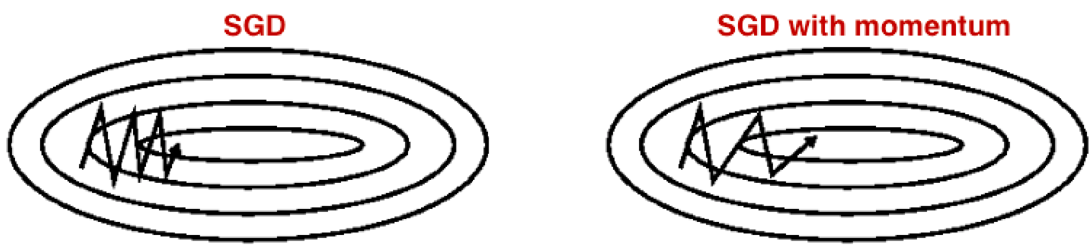
In [25]:
```
# you can make submission with answers so far to check yourself at this stage
grader.submit(COURSERA_EMAIL, COURSERA_TOKEN)
```

```
Submitted to Coursera platform. See results on assignment page!
```

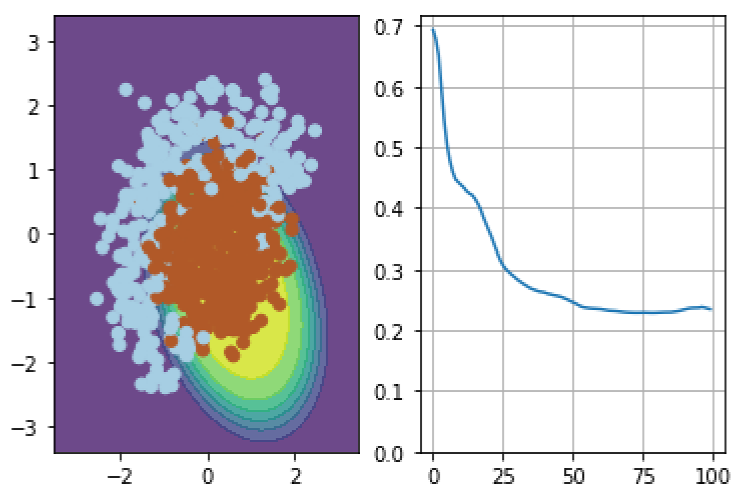
SGD with momentum

Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in image below. It does this by adding a fraction α of the update vector of the past time step to the current update vector.

$$v_t = \alpha v_{t-1} + \eta \frac{1}{m} \sum_{j=1}^{m} \nabla_w l(x_{i_j}, y_{i_j}, w_t)$$
$$w_t = w_{t-1} - v_t$$



In [26]:
```
# please use np.random.seed(42), eta=0.05, alpha=0.9, n_iter=100 and batch_size=4
np.random.seed(42)
w = np.array([0, 0, 0, 0, 0, 1])

eta = 0.05 # learning rate
alpha = 0.9 # momentum
nu = np.zeros_like(w)

n_iter = 100
batch_size = 4
loss = np.zeros(n_iter)
plt.figure(figsize=(12, 5))

for i in range(n_iter):
    ind = np.random.choice(X_expanded.shape[0], batch_size)
    loss[i] = compute_loss(X_expanded, y, w)
    if i % 10 == 0:
        visualize(X_expanded[ind, :], y[ind], w, loss)

    # TODO:<your code here>
    dW = compute_grad(X_expanded[ind, :], y[ind], w)
    nu = alpha*nu+eta*dW
    w = w - nu
    visualize(X, y, w, loss)
    plt.clf()
```




```
<Figure size 432x288 with 0 Axes>
```

In [27]:
```
# use output of this cell to fill answer field

ans_part5 = compute_loss(X_expanded, y, w)
```

In [28]:
```
## GRADED PART, DO NOT CHANGE!
grader.set_answer("GBdgZ", ans_part5)
```

In [29]:
```
# you can make submission with answers so far to check yourself at this stage
grader.submit(COURSERA_EMAIL, COURSERA_TOKEN)
```

```
Submitted to Coursera platform. See results on assignment page!
```

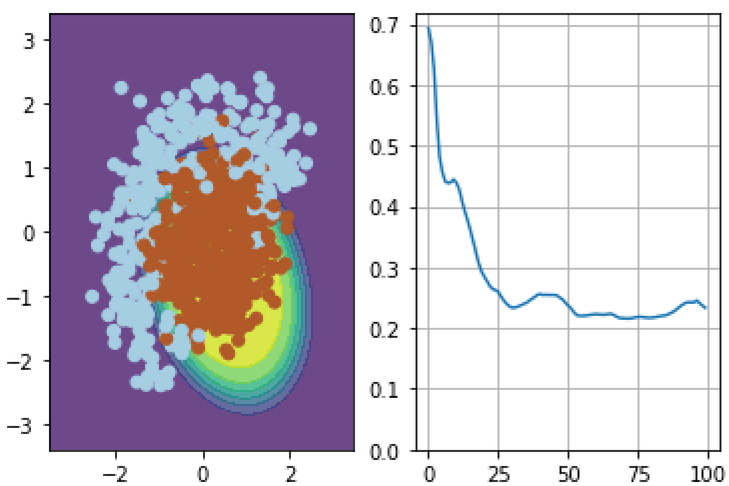
RMSprop

Implement RMSPROP algorithm, which use squared gradients to adjust learning rate:

$$G_j^t = \alpha G_j^{t-1} + (1-\alpha) g_{tj}^2$$
$$w_j^t = w_j^{t-1} - \frac{\eta}{\sqrt{G_j^t + \varepsilon}} g_{tj}$$

In [31]:

```
# please use np.random.seed(42), eta=0.1, alpha=0.9, n_iter=100 and batch_size=4
np.random.seed(42)

w = np.array([0, 0, 0, 0, 0, 1.])

eta = 0.1 # learning rate
alpha = 0.9 # moving average of gradient norm squared
g2 = None # we start with None so that you can update this value correctly on the
eps = 1e-8

n_iter = 100
batch_size = 4
loss = np.zeros(n_iter)
plt.figure(figsize=(12,5))
for i in range(n_iter):
    ind = np.random.choice(X_expanded.shape[0], batch_size)
    loss[i] = compute_loss(X_expanded, y, w)
    if i % 10 == 0:
        visualize(X_expanded[ind, :], y[ind], w, loss)

    # TODO:<your code here>
    dW = compute_grad(X_expanded[ind, :], y[ind], w)
    nu = alpha*nu+eta*dW
    w = w - nu

visualize(X, y, w, loss)
plt.clf()
```




```
<Figure size 432x288 with 0 Axes>
```

In [32]:
```
# use output of this cell to fill answer field
ans_part6 = compute_loss(X_expanded, y, w)
```

In [33]:
```
## GRADED PART, DO NOT CHANGE!
grader.set_answer("dLdHG", ans_part6)
```

In [34]:
```
grader.submit(COURSERA_EMAIL, COURSERA_TOKEN)
```

```
Submitted to Coursera platform. See results on assignment page!
```

In [ ]: