

▼ 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.

```
1 import numpy as np
2 %matplotlib inline
3 import matplotlib.pyplot as plt

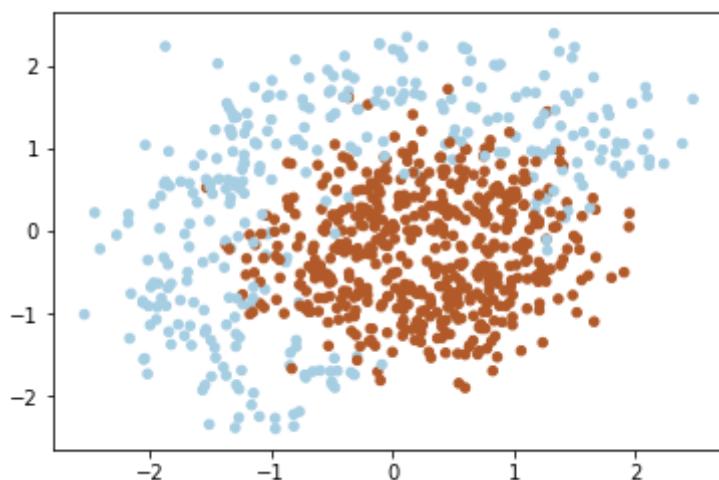
1 import sys
2 sys.path.append("..")
3 import grading
4 grader = grading.Grader(assignment_key="UaHtvpEFEee0XQ6wjK-hZg",
5                          all_parts=["xU7U4", "HyTF6", "uNidL", "ToK7N", "GBdgZ", "dLdHG"])

1 # token expires every 30 min
2 COURSERA_TOKEN = '1Gjdjj7WzDmGPyPr' ### YOUR TOKEN HERE
3 COURSERA_EMAIL = 'knowtech94@gmail.com' ### YOUR EMAIL HERE
```

▼ Two-dimensional classification

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

```
1 with open('train.npy', 'rb') as fin:
2     X = np.load(fin)
3
4 with open('target.npy', 'rb') as fin:
5     y = np.load(fin)
6
7 plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired, s=20)
8 plt.show()
```




```

6         [2.61, -1.28],
7         [-0.59, 2.1]
8     ])
9
10 # call your expand function
11 dummy_expanded = expand(dummy_X)
12
13 # what it should have returned:
14 dummy_expanded_ans = np.array([[ 0.      ,  0.      ,  0.      ,  0.      ,  0.      ,  1.      ],
15                                [ 1.      ,  0.      ,  1.      ,  0.      ,  0.      ,  1.      ],
16                                [ 2.61    , -1.28    ,  6.8121   ,  1.6384   , -3.3408   ,  1.      ],
17                                [-0.59    ,  2.1     ,  0.3481   ,  4.41     , -1.239    ,  1.      ]])
18
19 #tests
20 assert isinstance(dummy_expanded, np.ndarray), "please make sure you return numpy array"
21 assert dummy_expanded.shape == dummy_expanded_ans.shape, "please make sure your shape i
22 assert np.allclose(dummy_expanded, dummy_expanded_ans, 1e-3), "Something's out of order w
23
24 print("Seems legit!")
25

```

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:

$$a(x; w) = \langle w, x \rangle$$

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

```

1 def probability(X, w):
2     """
3     Given input features and weights
4     return predicted probabilities of y==1 given x, P(y=1|x), see description above
5
6     Don't forget to use expand(X) function (where necessary) in this and subsequent fun
7
8     :param X: feature matrix X of shape [n_samples,6] (expanded)
9     :param w: weight vector w of shape [6] for each of the expanded features
10    :returns: an array of predicted probabilities in [0,1] interval.
11    """
12
13    # TODO:<your code here>
14    if X.shape[1] == 2:
15        X = expand(X)
16    net = np.dot(X, w)
17    sigma = 1/(1 + np.exp(-net))
18    prob = np.array([sigma, 1-sigma])
19    return prob

```

```

1 X_expanded.shape

(826, 6)

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

1 ans_part1

array([ 0.38039985])

1 ## GRADED PART, DO NOT CHANGE!
2 grader.set_answer("xU7U4", ans_part1)

1 # you can make submission with answers so far to check yourself at this stage
2 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:

$$l(x_i, y_i, w) = -[y_i \cdot \log P(y_i = 1 | x_i, w) + (1 - y_i) \cdot \log(1 - P(y_i = 1 | x_i, w))]$$

Loss for many samples:

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

```

1 def compute_loss(X, y, w):
2     """
3     Given feature matrix X [n_samples,6], target vector [n_samples] of 1/0,
4     and weight vector w [6], compute scalar loss function L using formula above.
5     Keep in mind that our loss is averaged over all samples (rows) in X.
6     """
7     # TODO:<your code here>
8     prob = probability(X, w)
9     loss = -(y * np.log(prob[0]) + (1-y) * np.log(prob[1]))
10    return np.mean(loss)
11

```

```

1 ans_part2 = compute_loss(X_expanded, y, dummy_weights)

```

```

1 ans_part2

1.0185634030782518

```

```

1 ## GRADED PART, DO NOT CHANGE!
2 grader.set_answer("HyTF6", ans_part2)

```

```
1 # you can make submission with answers so far to check yourself at this stage
2 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).

```
1 def compute_grad(X, y, w):
2     """
3     Given feature matrix X [n_samples,6], target vector [n_samples] of 1/0,
4     and weight vector w [6], compute vector [6] of derivatives of L over each weights.
5     Keep in mind that our loss is averaged over all samples (rows) in X.
6     """
7
8     # TODO<your code here>
9
10    #compute delta
11    delta = y - probability(X, w)[0]
12
13    #compute gradient
14    grad = np.dot(X.transpose(), delta)/X.shape[0]
15
16    return grad
```

```
1 # use output of this cell to fill answer field
2 ans_part3 = np.linalg.norm(compute_grad(X_expanded, y, dummy_weights))
```

```
1 ans_part3

0.64016873021186249
```

```
1 ## GRADED PART, DO NOT CHANGE!
2 grader.set_answer("uNidL", ans_part3)
```

```
1 # you can make submission with answers so far to check yourself at this stage
2 grader.submit(COURSERA_EMAIL, COURSERA_TOKEN)
```

Submitted to Coursera platform. See results on assignment page!

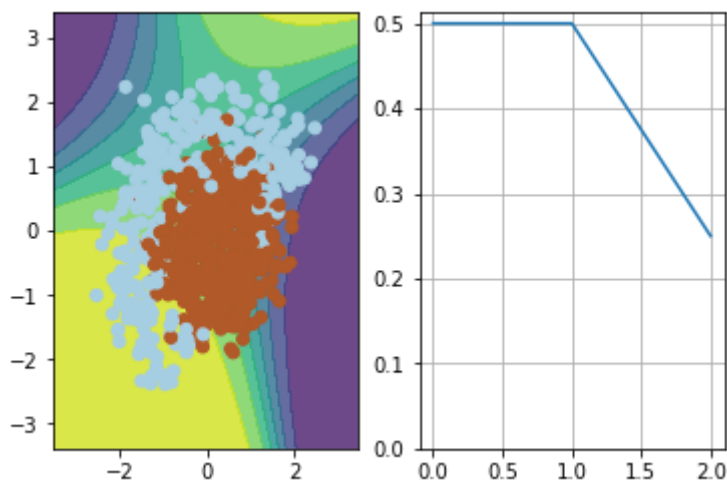
Here's an auxiliary function that visualizes the predictions:

```

1 from IPython import display
2
3 h = 0.01
4 x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
5 y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
6 xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
7
8 def visualize(X, y, w, history):
9     """draws classifier prediction with matplotlib magic"""
10    Z = probability(expand(np.c_[xx.ravel(), yy.ravel()]), w)[0]
11    Z = Z.reshape(xx.shape)
12    plt.subplot(1, 2, 1)
13    plt.contourf(xx, yy, Z, alpha=0.8)
14    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)
15    plt.xlim(xx.min(), xx.max())
16    plt.ylim(yy.min(), yy.max())
17
18    plt.subplot(1, 2, 2)
19    plt.plot(history)
20    plt.grid()
21    ymin, ymax = plt.ylim()
22    plt.ylim(0, ymax)
23    display.clear_output(wait=True)
24    plt.show()

```

```
1 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

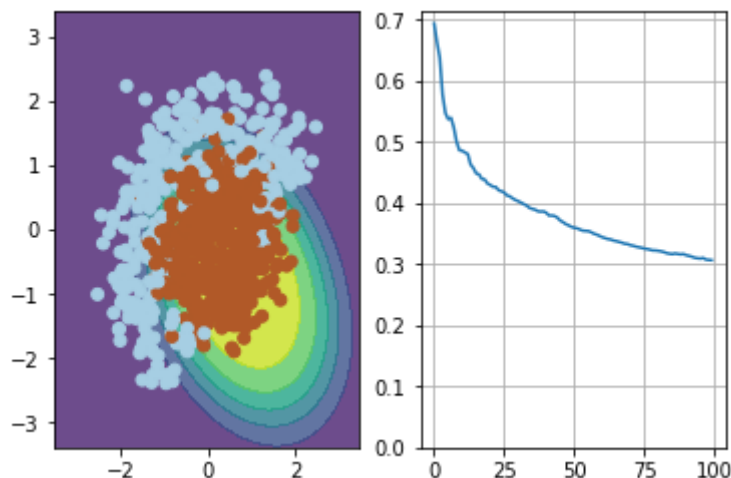
Stochastic gradient descent just takes a random batch of m samples on each iteration, calculates a gradient of the loss on it and makes a step:

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

```

1 # please use np.random.seed(42), eta=0.1, n_iter=100 and batch_size=4 for deterministic
2
3 np.random.seed(42)
4 w = np.array([0, 0, 0, 0, 0, 1])
5
6 eta= 0.1 # learning rate
7
8 n_iter = 100
9 batch_size = 4
10 loss = np.zeros(n_iter)
11 plt.figure(figsize=(12, 5))
12
13 for i in range(n_iter):
14     ind = np.random.choice(X_expanded.shape[0], batch_size)
15     loss[i] = compute_loss(X_expanded, y, w)
16     if i % 10 == 0:
17         visualize(X_expanded[ind, :], y[ind], w, loss)
18
19     # Keep in mind that compute_grad already does averaging over batch for you!
20     # TODO:<your code here>
21     grad = compute_grad(X_expanded[ind, :], y[ind], w)
22     w = w + eta * grad # negative sign begin of gradient
23 visualize(X, y, w, loss)
24 plt.clf()

```



<matplotlib.figure.Figure at 0x7f901cdbd748>

```

1 # use output of this cell to fill answer field
2 ans_part4 = compute_loss(X_expanded, y, w)

```

```
1 ans_part4
```

```
0.30427646989924029
```

```
1 ## GRADED PART, DO NOT CHANGE!
```

```
2 grader.set_answer("ToK7N", ans_part4)
```

```
1 # you can make submission with answers so far to check yourself at this stage
```

```
2 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.

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

$$w_t = w_{t-1} - \nu_t$$

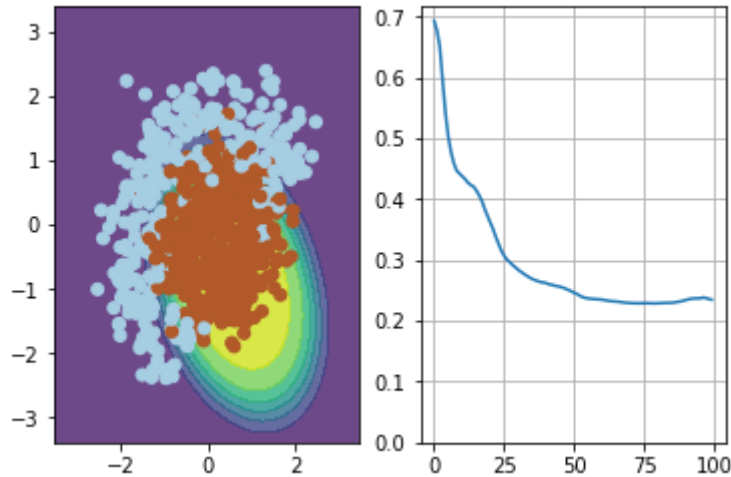
```
1 # please use np.random.seed(42), eta=0.05, alpha=0.9, n_iter=100 and batch_size=4 for d
2 np.random.seed(42)
3 w = np.array([0, 0, 0, 0, 0, 1])
4
5 eta = 0.05 # learning rate
6 alpha = 0.9 # momentum
7 nu = np.zeros_like(w)
8
9 n_iter = 100
10 batch_size = 4
11 loss = np.zeros(n_iter)
12 plt.figure(figsize=(12, 5))
13
14 for i in range(n_iter):
15     ind = np.random.choice(X_expanded.shape[0], batch_size)
16     loss[i] = compute_loss(X_expanded, y, w)
17     if i % 10 == 0:
18         visualize(X_expanded[ind, :], y[ind], w, loss)
19
20     # TODO:<your code here>
21     grad = compute_grad(X_expanded[ind, :], y[ind], w)
22     nu = alpha * nu + eta * grad
23     w = w + nu # negative sign begin to negative gradient
24
```



```

25 visualize(X, y, w, loss)
26 plt.clf()

```



<matplotlib.figure.Figure at 0x7f8fd42f2a90>

```

1 # use output of this cell to fill answer field
2 ans_part5 = compute_loss(X_expanded, y, w)

```

```
1 ans_part5
```

0.23245916420113075

```

1 ## GRADED PART, DO NOT CHANGE!
2 grader.set_answer("GBdgZ", ans_part5)

```

```

1 # you can make submission with answers so far to check yourself at this stage
2 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:

$$G_j^t = \alpha G_j^{t-1} + (1 - \alpha) g_{tj}^2$$

$$w_j^t = w_j^{t-1} - \frac{\eta}{\sqrt{G_j^t + \epsilon}} g_{tj}$$

```

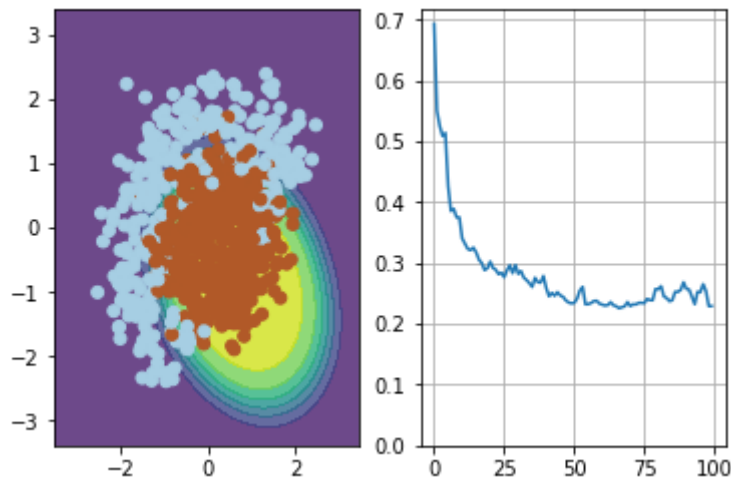
1 # please use np.random.seed(42), eta=0.1, alpha=0.9, n_iter=100 and batch_size=4 for de
2 np.random.seed(42)
3
4 w = np.array([0, 0, 0, 0, 0, 1.])
5
6 eta = 0.1 # learning rate
7 alpha = 0.9 # moving average of gradient norm squared
8 g2 = None # we start with None so that you can update this value correctly on the first

```

```

9 eps = 1e-8
10
11 n_iter = 100
12 batch_size = 4
13 loss = np.zeros(n_iter)
14 plt.figure(figsize=(12,5))
15 for i in range(n_iter):
16     ind = np.random.choice(X_expanded.shape[0], batch_size)
17     loss[i] = compute_loss(X_expanded, y, w)
18     if i % 10 == 0:
19         visualize(X_expanded[ind, :], y[ind], w, loss)
20
21     # TODO:<your code here>
22     grad = compute_grad(X_expanded[ind, :], y[ind], w)
23     if g2 == None:
24         g2 = (1-alpha) * (grad**2)
25     else:
26         g2 = alpha * g2 + (1-alpha) * (grad**2)
27     w = w + eta * grad / np.sqrt(g2 + eps) # negative sign begin to negative gradient
28
29 visualize(X, y, w, loss)
30 plt.clf()

```



<matplotlib.figure.Figure at 0x7f8fd4327748>

```

1 # use output of this cell to fill answer field
2 ans_part6 = compute_loss(X_expanded, y, w)

```

```
1 ans_part6
```

```
0.22383829902910221
```

```

1 ## GRADED PART, DO NOT CHANGE!
2 grader.set_answer("dLdHG", ans_part6)

```

```
1 grader.submit(COURSERA_EMAIL, COURSERA_TOKEN)
```

Submitted to Coursera platform. See results on assignment page!

