Machine learning : HW 05 王上澤 111598067

1. For the problem of finding the fastest way to the island on pp 2 of 19_Gradient_descent_basis.
    1. formulate it as a constrained optimization problem with variables x and y.
        1. Find the minimum of time spend on running and swimming

$$j(x,y) = \frac{x}{8} + \frac{y}{3}$$

where

$$x < 6$$

and

$$y = \sqrt{(6-x)^2 + 2^2}$$

        2. So we can make j(x, y) into

$$j(x) = \frac{x}{8} + \frac{\sqrt{(6-x)^2 + 2^2}}{3}$$

        3. Then we do the derivative

$$\frac{d}{dx}(\frac{x}{8} + \frac{\sqrt{(6-x)^2 + 2^2}}{3})$$

into

$$\frac{1}{8} + \frac{1}{6 \times \sqrt{x^2 - 12x + 40}} \times (2x - 12)$$

        4. Then we going to find out the x that make $j'(X) = 0$

$$55x^2 - 660x + 1944 = 0$$

$$x = 6 \pm, \frac{\sqrt{7920}}{110}$$

because $x < 6$, so

$$x = 6 - \frac{\sqrt{7920}}{110} \approx 5.19$$

so the fastest way to go to the island is first run 5.19 miles and swim the rest of route.

    2. Analytically solve this problem by using the Lagrange multiplier method.
        1. Our objective is to minimize the $f(x,y) = \frac{x}{8} + \frac{y}{3}$ subject to $y^2 - (6-x)^2 = 4$ we write the auxiliary equation as

$$\mathcal{L}(x,y,\lambda) = \frac{x}{8} + \frac{y}{3} + \lambda(y^2 - (6-x)^2 - 4)$$

Then find the partial derivative of

$$\frac{\partial}{\partial x}\mathcal{L}(x,y,\lambda)=0$$

$$\frac{\partial}{\partial y}\mathcal{L}(x,y,\lambda)=0$$

$$\frac{\partial}{\partial \lambda}\mathcal{L}(x,y,\lambda)=0$$

2. After the calculation, we can find the

$$\frac{\partial}{\partial x}\mathcal{L}=\frac{1}{8}+12\lambda-2\lambda x=0$$

$$\frac{\partial}{\partial y}\mathcal{L}=\frac{1}{3}+2\lambda y=0$$

$$\frac{\partial}{\partial \lambda}\mathcal{L}=y^2-(6-x)^2-4=0$$

and

$$x=\frac{1}{16\lambda}+6$$

$$y=\frac{-1}{6\lambda}$$

lets put x and y into

$$y^2-(6-x)^2-4=0$$

and we can find out

$$\lambda=\pm\frac{\sqrt{55}}{96}$$

However, due the contraint of $x<6$ and y should be a positive number(its nonsense to swim away from island), so the λ should be

$$\lambda=-\frac{\sqrt{55}}{96}$$

and put λ into x and y

$$x=6-\frac{6}{\sqrt{55}}\approx 5.19$$

$$y=\frac{16}{\sqrt{55}}$$

the same answer to the last question's.

2. Follow the treatment of the L2 regularization (pp. 45 – 48) to find out the influence of the L1 regularization on the updating of weights.

1. Do in gradient descent with a modified cost function

$$\nabla \mathcal{J}(w) = \nabla f(w) + \nabla \lambda g(w)$$

where

$$\nabla \lambda g(w) = \frac{d}{dw}(|w|) = \frac{w}{|w|}$$

so the update formula should be

$$w(k+1) = w(k) - \eta(\nabla f(w(k) - sign(w(k))w(k)))$$

where

$$sign(w(k)) = \{\ 1\ if\ w(k) > 0,\ 0\ if\ w(k) = 0,\ -1\ if\ w(k) < 0$$

3. We mention in the lecture about how to use a penalty function to convert a constrained optimization problem to an unconstrained one. However, the obtained solution is not exactly the same as that from the original problem. The following illustrates this situation.

   1. Analytically find the values of x and y to maximize the cost function

$$\mathcal{J}(x, y) = x + y - 100(x^2 + y^2 - 1)^2$$

   To maximize the cost function, I write a python code to do the gradient descent.

```
import numpy as np

def cost_function(x, y):
    return x + y - 100 * (x**2 + y**2 - 1)**2

def gradient(x, y):
    dx = 1 - 400 * x * (x**2 + y**2 - 1)
    dy = 1 - 400 * y * (x**2 + y**2 - 1)
    return np.array([dx, dy])

def gradient_descent(learning_rate, num_iterations):
    # Initial values
    x = 0.5
    y = 0.5

    # Perform gradient descent
    for i in range(num_iterations):
        grad = gradient(x, y)
        x += learning_rate * grad[0]
        y += learning_rate * grad[1]

    return x, y

# Set hyperparameters
learning_rate = 0.001
num_iterations = 10000
```

```
# Run gradient descent
x_optimized, y_optimized = gradient_descent(learning_rate,
num_iterations)

# Print the optimized values
print("Optimized values:")
print("x =", x_optimized)
print("y =", y_optimized)
```

and the output tell me the `x` and `y` for the max $J(x, y)$ is

```
Optimized values:
x = 0.7083401048248115
y = 0.7083687630644799
```

2. Is your answer the same as the solution of the original problem (the example in the lecture notes)?

   This is not as the same number as $(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2})$, but close enough.

3. How to change *J*(x, y) if we want to find the minimum of the function

$$f(x, y) = x + y \; subject \; to \; x^2 + y^2 = 1$$

   Just change the sign of learning rate from `-` to `+`

```
import numpy as np

def cost_function(x, y):
    return x + y - 100 * (x**2 + y**2 - 1)**2

def gradient(x, y):
    dx = 1 - 400 * x * (x**2 + y**2 - 1)
    dy = 1 - 400 * y * (x**2 + y**2 - 1)
    return np.array([dx, dy])

def gradient_ascent(learning_rate, num_iterations):
    # Initial values
    x = 0.5
    y = 0.5

    # Perform gradient descent
    for i in range(num_iterations):
        grad = gradient(x, y)
        x -= learning_rate * grad[0]
        y -= learning_rate * grad[1]

    return x, y
```

```
# Set hyperparameters
learning_rate = 0.001
num_iterations = 10000

# Run gradient descent
x_optimized, y_optimized = gradient_ascent(learning_rate,
num_iterations)

# Print the optimized values
print("Optimized values:")
print("x =", x_optimized)
print("y =", y_optimized)
```

output:

```
Optimized values:
x = -0.0025000312511719337
y = -0.0025000312511719337
```

4. Repeat problem 3 (a), but use the gradient ascent algorithm to find the solution. To have a unique answer, use $x0 = y0 = 1$ and $\eta = 0.0005$.

    1. Here comes the code

```
import numpy as np

def cost_function(x, y):
    return x + y - 100 * (x**2 + y**2 - 1)**2

def gradient(x, y):
    dx = 1 - 400 * x * (x**2 + y**2 - 1)
    dy = 1 - 400 * y * (x**2 + y**2 - 1)
    return np.array([dx, dy])

def gradient_ascent(learning_rate, num_iterations):
    # Initial values
    x = 1.0
    y = 1.0

    # Perform gradient descent
    for i in range(num_iterations):
        grad = gradient(x, y)
        x += learning_rate * grad[0]
        y += learning_rate * grad[1]

    return x, y

# Set hyperparameters
```

```
learning_rate = 0.0005
num_iterations = 1000

# Run gradient descent
x_optimized, y_optimized = gradient_ascent(learning_rate,
num_iterations)

# Print the optimized values
print("Optimized values:")
print("x =", x_optimized)
print("y =", y_optimized)
```

and output:

```
Optimized values:
x = 0.7083534821584602
y = 0.7083534821584602
```

2. Is your numerical solution close to the analytic solution in problem 3? Yes!
3. If we increase the penalty constant from 100 to 10,000, what side effect(s) may result in your program?
   The double scalar will overflow!

5. Use the LDA to reduce the feature dimension from 4 to 2 for the Iris data set. As usual, take 70% of the samples as the training set to perform dimensionality reduction. Use 5-NN to classify the test set and then report the average accuracy after 10 trials.

```
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.decomposition import FactorAnalysis
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis



url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data"
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width',
'Class']
dataset = pd.read_csv(url, names=names)

# preprocessing
X = dataset.drop('Class', axis = 1)
y = dataset['Class']
# 10 times iteration
```

```
for i in range(10):
    # Splitting the dataset into the Training set and Test set
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

    fa = LinearDiscriminantAnalysis(n_components=2)
    X_train = fa.fit_transform(X_train, y_train)
    X_test = fa.transform(X_test)

    classifier = KNeighborsClassifier(n_neighbors=5)
    classifier.fit(X_train, y_train)

    # Predicting the Test set results
    y_pred = classifier.predict(X_test)
    print('Accuracy ' + str(accuracy_score(y_test, y_pred)), str(i),
"iteration")
```

output:

```
Accuracy 1.0 0 iteration
Accuracy 1.0 1 iteration
Accuracy 0.9777777777777777 2 iteration
Accuracy 0.9555555555555556 3 iteration
Accuracy 0.9777777777777777 4 iteration
Accuracy 0.9111111111111111 5 iteration
Accuracy 0.9555555555555556 6 iteration
Accuracy 0.9555555555555556 7 iteration
Accuracy 0.9777777777777777 8 iteration
Accuracy 0.9777777777777777 9 iteration
```