AML_H3T1

April 18, 2023

In this assignment, I leveraged the high RAM spcae on Google Colab, and tested different methods (GD, SGD, BFGS, LBFGS) with different parameters to optimzie the given objective funtion.

In the GD method, the gradient coverges after 300 iterations. However, it doesn't converge in SGD with p=1 and p=100, and the reason is that the number of training point is too small to fit to the model. On the other hand, BFGS and LBFGS have great performance, and the gradient only takes 3 and 1 iteration(s) to converge respectively.The limited amount of memory used in LBFGS makes it more efficient for problems with a large number of variables, while BFGS is generally more suitable for problems with a small number of variables. In this assignment the final gradient value in BFGS is much smaller than LBFGS's. But considering only 500 training points were used in LBFGS method and very short training time it takes, There are still good reasons to choose LBFGS as the most efficient one compared to other three models.

[1]:
```
from psutil import virtual_memory
ram_gb = virtual_memory().total / 1e9
print('Your runtime has {:.1f} gigabytes of available RAM\n'.format(ram_gb))

if ram_gb < 20:
  print('Not using a high-RAM runtime')
else:
  print('You are using a high-RAM runtime!')
```

```
Your runtime has 27.3 gigabytes of available RAM

You are using a high-RAM runtime!
```

[2]:
```
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
from sklearn.metrics.pairwise import rbf_kernel
```

[3]:
```
# Load data
data = scipy.io.loadmat('data1.mat')
X_train = data['TrainingX']
y_train = data['TrainingY'].ravel()
X_test = data['TestX']
y_test = data['TestY'].ravel()

# Compute kernel matrix
```

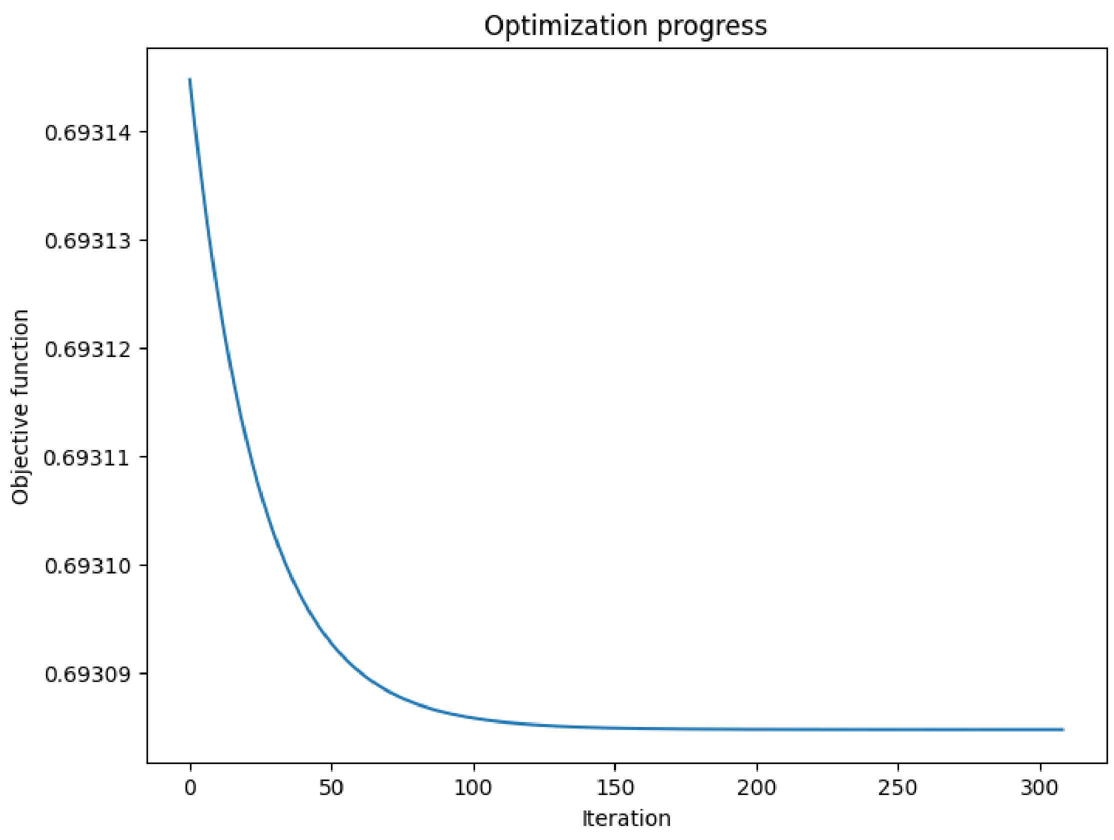
```
N_test = X_test.shape[0]
N_train = X_train.shape[0]
subset_size = 2000 # size of subset used to estimate gamma
gamma = np.sqrt(1 / (subset_size ** 2) * np.sum((X_train[:subset_size, None, :]␣
 ↪- X_train[:subset_size, :]) ** 2))
K_train = rbf_kernel(X_train, X_train, gamma=gamma)
K_test = rbf_kernel(X_test, X_train, gamma=gamma)
gamma
```

[3]:
```
9.529466351243935
```

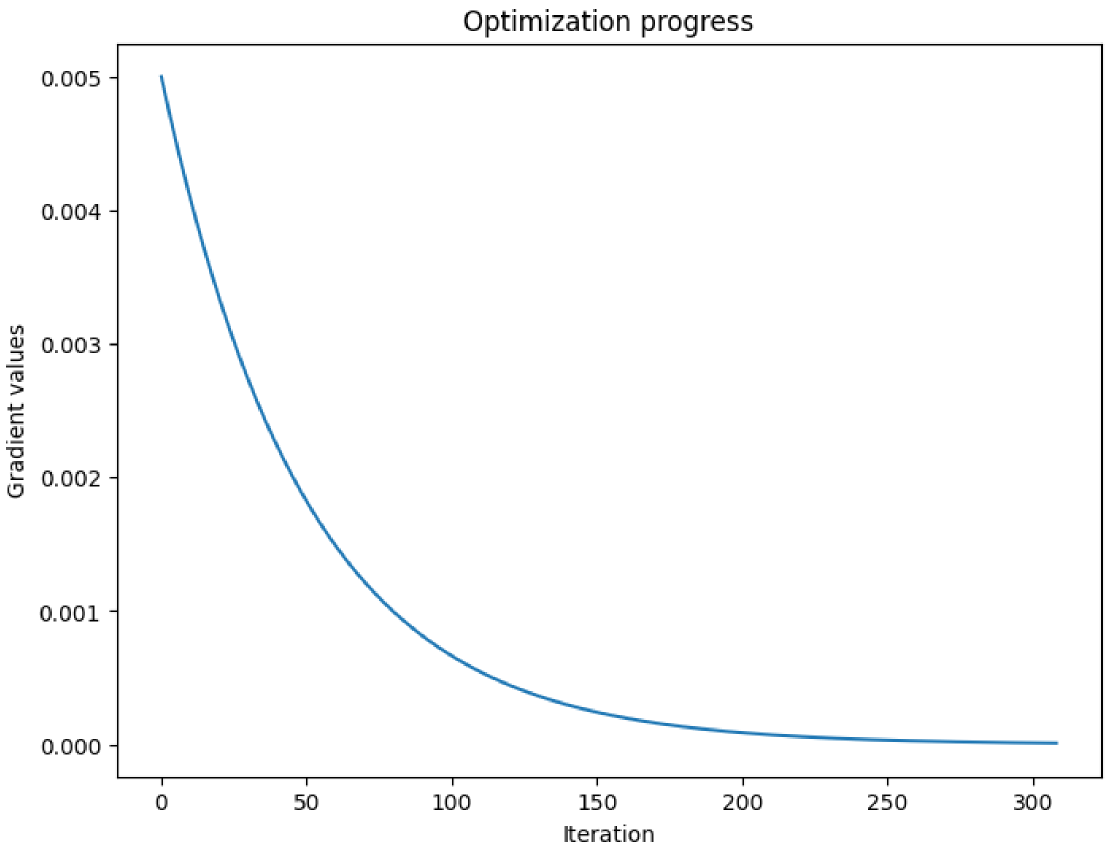
[8]:
```
# Define objective function and gradient function
def objective_function(w, K_train, y_train, lambda_reg):
    z = y_train * K_train.dot(w)
    J = np.mean(np.log(1 + np.exp(-z))) + lambda_reg * np.sum(w ** 2)
    return J

def gradient(w, K_train, y_train, lambda_reg):
    z = y_train * K_train.dot(w)
    grad = -1/N_train * K_train.T.dot(y_train/(1 + np.exp(z))) + 2*lambda_reg*w
    return grad

# Define initial weight vector, learning rate, and regularization parameter
w0 = np.zeros(N_train)
alpha = 0.1  # Learning rate for GD
lambda_reg = 0.1  # Regularization parameter
epsilon = 1e-5

# Train using GD
max_iterations = 500  # Maximum number of iterations for GD
history = {'iter': [], 'J': [], 'grad_norm': []}  # Track optimization progress
w = w0
for i in range(max_iterations):
    grad = gradient(w, K_train, y_train, lambda_reg)
    w = w - alpha * grad
    J = objective_function(w, K_train, y_train, lambda_reg)
    grad_norm = np.linalg.norm(grad)
    history['iter'].append(i)
    history['J'].append(J)
    history['grad_norm'].append(grad_norm)
    if grad_norm < epsilon:
        break
    if i % 20 == 0:
      print(f"Iteration {i}: J={J:.6f}, ||grad||={grad_norm:.6f}")

# Compute test predictions
y_pred = np.sign(K_test.dot(w))
```

```python
y_pred[y_pred == 0] = 1
accuracy = np.mean(y_pred == y_test)
print("Test accuracy:", accuracy)

# Plot optimization progress
plt.figure(figsize=(8, 6))
plt.plot(history['iter'], history['J'])
plt.xlabel('Iteration')
plt.ylabel('Objective function')
plt.title('Optimization progress')
plt.show()
```

```
Iteration 0: J=0.693145, ||grad||=0.005000
Iteration 20: J=0.693111, ||grad||=0.003338
Iteration 40: J=0.693097, ||grad||=0.002228
Iteration 60: J=0.693090, ||grad||=0.001488
Iteration 80: J=0.693087, ||grad||=0.000993
Iteration 100: J=0.693086, ||grad||=0.000663
Iteration 120: J=0.693085, ||grad||=0.000443
Iteration 140: J=0.693085, ||grad||=0.000295
Iteration 160: J=0.693085, ||grad||=0.000197
Iteration 180: J=0.693085, ||grad||=0.000132
Iteration 200: J=0.693085, ||grad||=0.000088
Iteration 220: J=0.693085, ||grad||=0.000059
Iteration 240: J=0.693085, ||grad||=0.000039
Iteration 260: J=0.693085, ||grad||=0.000026
Iteration 280: J=0.693085, ||grad||=0.000017
Iteration 300: J=0.693085, ||grad||=0.000012
Test accuracy: 0.966
```



[10]:
```
plt.figure(figsize=(8, 6))
plt.plot(history['iter'], history['grad_norm'])
plt.xlabel('Iteration')
plt.ylabel('Gradient values')
plt.title('Optimization progress')
plt.show()
```


Optimization progress
Gradient values
0.005
0.004
0.003
0.002
0.001
0.000
Iteration
0
50
100
150
200
250
300

AML_H3T2

April 18, 2023

```
[11]: from psutil import virtual_memory
      ram_gb = virtual_memory().total / 1e9
      print('Your runtime has {:.1f} gigabytes of available RAM\n'.format(ram_gb))

      if ram_gb < 20:
        print('Not using a high-RAM runtime')
      else:
        print('You are using a high-RAM runtime!')
```

```
Your runtime has 27.3 gigabytes of available RAM

You are using a high-RAM runtime!
```

```
[12]: import numpy as np
      import scipy.io
      from sklearn.metrics.pairwise import rbf_kernel
      import random
      import matplotlib.pyplot as plt
```

```
[14]: data = scipy.io.loadmat('data1.mat')
      X_train = data['TrainingX']
      y_train = data['TrainingY'].ravel()
      X_test = data['TestX']
      y_test = data['TestY'].ravel()

      # Compute kernel matrix
      N_test = X_test.shape[0]
      N_train = X_train.shape[0]
      subset_size = 2000 # size of subset used to estimate gamma
      gamma = 9.529466351243935
      K_train = rbf_kernel(X_train, X_train, gamma=gamma)
      K_test = rbf_kernel(X_test, X_train, gamma=gamma)

      # Define objective function and gradient function
      def objective_function(w, K_train, y_train, lambda_reg):
          z = y_train * K_train.dot(w)
          J = np.mean(np.log(1 + np.exp(-z))) + lambda_reg * np.sum(w ** 2)
          return J
```

```python
def gradient(w, K_train, y_train, lambda_reg, p):
    idx = np.random.choice(N_train, p, replace=False)
    K_sub = K_train[idx]
    y_sub = y_train[idx]
    z = y_sub * K_sub.dot(w)
    grad = -1/p * K_sub.T.dot(y_sub/(1 + np.exp(z))) + 2*lambda_reg*w
    return grad

# Define initial weight vector, learning rate, and regularization parameter
w0 = np.zeros(N_train)
alpha = 0.1  # Learning rate for SGD
lambda_reg = 0.1  # Regularization parameter
epsilon = 1e-5

# Train using SGD with p = 1
max_iterations = 500  # Maximum number of iterations for SGD
history1 = {'iter': [], 'J': [], 'grad_norm': []}  # Track optimization progress
w = w0
p = 1
for i in range(max_iterations):
    grad = gradient(w, K_train, y_train, lambda_reg, p)
    w = w - alpha * grad
    J = objective_function(w, K_train, y_train, lambda_reg)
    grad_norm = np.linalg.norm(grad)
    history1['iter'].append(i)
    history1['J'].append(J)
    history1['grad_norm'].append(grad_norm)
    if grad_norm < epsilon:
        break
    if i % 20 == 0:
        print(f"p=1 Iteration {i}: J={J:.6f}, ||grad||={grad_norm:.6f}")

# Compute test predictions
y_pred = np.sign(K_test.dot(w))
y_pred[y_pred == 0] = 1
accuracy = np.mean(y_pred == y_test)
print("Test accuracy:", accuracy)
```

```
p=1 Iteration 0: J=0.693395, ||grad||=0.500000
p=1 Iteration 20: J=0.696715, ||grad||=0.501398
p=1 Iteration 40: J=0.698186, ||grad||=0.502020
p=1 Iteration 60: J=0.698836, ||grad||=0.502296
p=1 Iteration 80: J=0.699121, ||grad||=0.502420
p=1 Iteration 100: J=0.699246, ||grad||=0.502475
p=1 Iteration 120: J=0.699299, ||grad||=0.502499
p=1 Iteration 140: J=0.699407, ||grad||=0.502545
p=1 Iteration 160: J=0.699369, ||grad||=0.502531
```

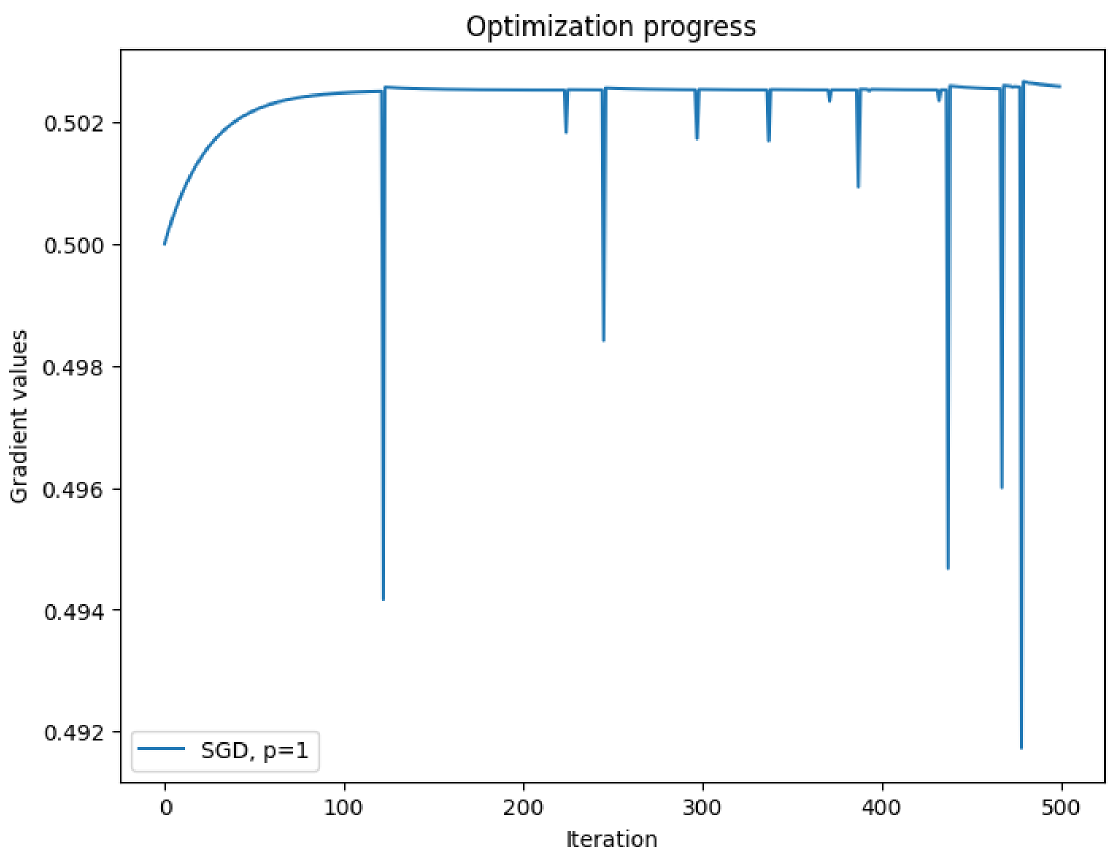
```
p=1 Iteration 180: J=0.699352, ||grad||=0.502524
p=1 Iteration 200: J=0.699344, ||grad||=0.502521
p=1 Iteration 220: J=0.699340, ||grad||=0.502520
p=1 Iteration 240: J=0.699346, ||grad||=0.502523
p=1 Iteration 260: J=0.699388, ||grad||=0.502540
p=1 Iteration 280: J=0.699359, ||grad||=0.502528
p=1 Iteration 300: J=0.699362, ||grad||=0.502529
p=1 Iteration 320: J=0.699348, ||grad||=0.502524
p=1 Iteration 340: J=0.699357, ||grad||=0.502527
p=1 Iteration 360: J=0.699345, ||grad||=0.502523
p=1 Iteration 380: J=0.699343, ||grad||=0.502522
p=1 Iteration 400: J=0.699360, ||grad||=0.502529
p=1 Iteration 420: J=0.699347, ||grad||=0.502523
p=1 Iteration 440: J=0.699490, ||grad||=0.502583
p=1 Iteration 460: J=0.699405, ||grad||=0.502547
p=1 Iteration 480: J=0.699659, ||grad||=0.502653
Test accuracy: 0.927
```

[26]:
```
history2 = {'iter': [], 'J': [], 'grad_norm': []}  # Track optimization progress
w = w0
p = 100
for i in range(max_iterations):
    grad = gradient(w, K_train, y_train, lambda_reg, p)
    w = w - alpha * grad
    J = objective_function(w, K_train, y_train, lambda_reg)
    grad_norm = np.linalg.norm(grad)
    history2['iter'].append(i)
    history2['J'].append(J)
    history2['grad_norm'].append(grad_norm)
    if grad_norm < epsilon:
        break
    if i % 20 == 0:
        print(f"p=100 Iteration {i}: J={J:.6f}, ||grad||={grad_norm:.6f}")

# Compute test predictions
y_pred = np.sign(K_test.dot(w))
y_pred[y_pred == 0] = 1
accuracy = np.mean(y_pred == y_test)
print("Test accuracy:", accuracy)
```

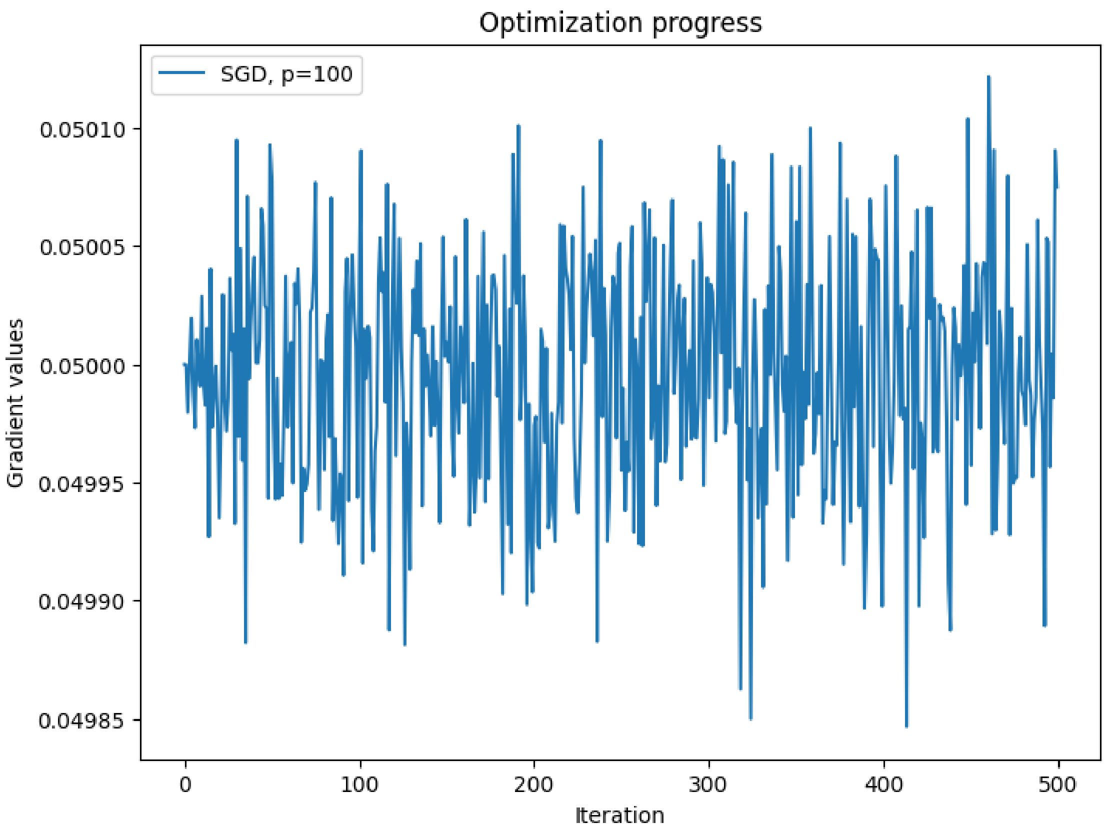
```
p=100 Iteration 0: J=0.693147, ||grad||=0.050000
p=100 Iteration 20: J=0.693148, ||grad||=0.049935
p=100 Iteration 40: J=0.693147, ||grad||=0.050045
p=100 Iteration 60: J=0.693147, ||grad||=0.050001
p=100 Iteration 80: J=0.693147, ||grad||=0.049955
p=100 Iteration 100: J=0.693148, ||grad||=0.050038
p=100 Iteration 120: J=0.693147, ||grad||=0.050068
p=100 Iteration 140: J=0.693147, ||grad||=0.049989
```

```
p=100 Iteration 160: J=0.693147, ||grad||=0.049984
p=100 Iteration 180: J=0.693147, ||grad||=0.050007
p=100 Iteration 200: J=0.693148, ||grad||=0.049974
p=100 Iteration 220: J=0.693148, ||grad||=0.050029
p=100 Iteration 240: J=0.693146, ||grad||=0.050032
p=100 Iteration 260: J=0.693147, ||grad||=0.049924
p=100 Iteration 280: J=0.693146, ||grad||=0.049988
p=100 Iteration 300: J=0.693146, ||grad||=0.049986
p=100 Iteration 320: J=0.693145, ||grad||=0.050028
p=100 Iteration 340: J=0.693146, ||grad||=0.050050
p=100 Iteration 360: J=0.693146, ||grad||=0.049962
p=100 Iteration 380: J=0.693147, ||grad||=0.049981
p=100 Iteration 400: J=0.693147, ||grad||=0.049981
p=100 Iteration 420: J=0.693147, ||grad||=0.049898
p=100 Iteration 440: J=0.693148, ||grad||=0.050024
p=100 Iteration 460: J=0.693145, ||grad||=0.050122
p=100 Iteration 480: J=0.693146, ||grad||=0.049986
Test accuracy: 0.964
```

[27]:
```python
plt.figure(figsize=(8, 6))

plt.plot(history2['iter'], history1['grad_norm'], label='SGD, p=1')
plt.xlabel('Iteration')
plt.ylabel('Gradient values')
plt.title('Optimization progress')
plt.legend()
plt.show()
```



[28]:
```
plt.figure(figsize=(8, 6))

plt.plot(history2['iter'], history2['grad_norm'], label='SGD, p=100')
plt.xlabel('Iteration')
plt.ylabel('Gradient values')
plt.title('Optimization progress')
plt.legend()
plt.show()
```


Optimization progress
SGD, p=100
Gradient values
0.05010
0.05005
0.05000
0.04995
0.04990
0.04985
Iteration
0
100
200
300
400
500

AML_H3T3

April 18, 2023

[1]:
```
from psutil import virtual_memory
ram_gb = virtual_memory().total / 1e9
print('Your runtime has {:.1f} gigabytes of available RAM\n'.format(ram_gb))

if ram_gb < 20:
  print('Not using a high-RAM runtime')
else:
  print('You are using a high-RAM runtime!')
```

```
Your runtime has 27.3 gigabytes of available RAM

You are using a high-RAM runtime!
```

[2]:
```
import numpy as np
import scipy.io
from sklearn.metrics.pairwise import rbf_kernel
from scipy.optimize import minimize
import matplotlib.pyplot as plt
```

[3]:
```
data = scipy.io.loadmat('data1.mat')
X_train = data['TrainingX']
y_train = data['TrainingY'].ravel()
X_test = data['TestX']
y_test = data['TestY'].ravel()

# Compute kernel matrix
N_train = X_train.shape[0]
N_test = X_test.shape[0]
subset_size = 2000 # size of subset used to estimate gamma
gamma = np.sqrt(np.sum((X_train[:subset_size, None, :] - X_train[:subset_size, :
 ↪]) ** 2) / (subset_size ** 2))
K_train = rbf_kernel(X_train, X_train, gamma=gamma)
K_test = rbf_kernel(X_test, X_train, gamma=gamma)
gamma
```

[3]:
```
9.529466351243935
```

[4]:
```
# Define objective function and gradient function
def objective_function(w, K_train, y_train, lambda_reg):
    z = y_train * K_train.dot(w)
    J = np.mean(np.log(1 + np.exp(-z))) + lambda_reg * np.sum(w ** 2)
    return J

def gradient(w, K_train, y_train, lambda_reg):
    z = y_train * K_train.dot(w)
    grad = -1/N_train * K_train.T.dot(y_train/(1 + np.exp(z))) + 2*lambda_reg*w
    return grad


# Define initial weight vector and regularization parameter
w0 = np.zeros(N_train)
lambda_reg = 0.1  # Regularization parameter

# Define function to compute Hessian approximation using BFGS
def hessian_approx(w, *args):
    K_train, y_train, lambda_reg = args
    z = y_train * K_train.dot(w)
    diag = np.exp(z) / ((1 + np.exp(z)) ** 2)
    D = np.diag(diag)
    A = D + 2 * lambda_reg * np.eye(N_train)
    H = K_train.T.dot(A).dot(K_train)
    return H

# Train using BFGS
max_iterations = 1000  # Maximum number of iterations for BFGS

history_bfgs = []
def print_gradient(x):
  grad=gradient(x, K_train, y_train, lambda_reg)
  grad_norm = np.linalg.norm(grad)
  history_bfgs.append(grad_norm)
  print("Gradient:", grad_norm)

#result = minimize(objective_function, w0, method='BFGS', jac=gradient,␣
 ↪tol=1e-5, args=(K_train, y_train, lambda_reg), callback=print_gradient)
result = minimize(objective_function, w0, method='BFGS', jac=gradient, tol=1e-5,␣
 ↪args=(K_train, y_train, lambda_reg),
                  hessp=hessian_approx, options={'maxiter': max_iterations,␣
 ↪'disp': True}, callback=print_gradient)

w = result.x

# Compute test predictions
y_pred = np.sign(K_test.dot(w))
```

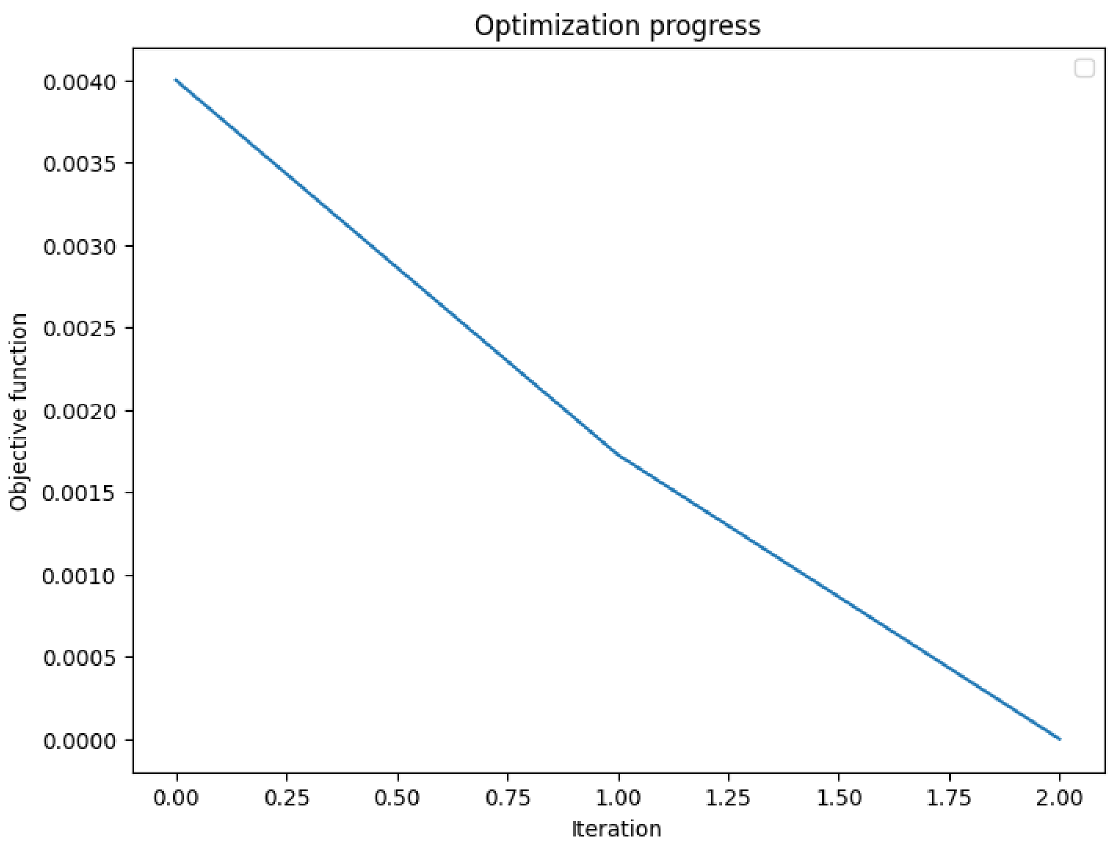
```
y_pred[y_pred == 0] = 1
accuracy = np.mean(y_pred == y_test)
print("Test accuracy:", accuracy)
```

```
/usr/local/lib/python3.9/dist-packages/scipy/optimize/_minimize.py:560:
RuntimeWarning: Method BFGS does not use Hessian-vector product information
(hessp).
  warn('Method %s does not use Hessian-vector product '
```

```
Gradient: 0.0039998750016511184
Gradient: 0.0017270514788925923
Gradient: 1.2556983131731068e-14
Optimization terminated successfully.
         Current function value: 0.693085
         Iterations: 3
         Function evaluations: 4
         Gradient evaluations: 4
Test accuracy: 0.966
```

[6]:
```
# Plot optimization progress
plt.figure(figsize=(8, 6))
plt.plot(history_bfgs)
plt.xlabel('Iteration')
plt.ylabel('Objective function')
plt.title('Optimization progress')
plt.legend()
plt.show()
```

```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note
that artists whose label start with an underscore are ignored when legend() is
called with no argument.
```


Optimization progress
Objective function
Iteration
0.0040
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

AML_H3T4

April 18, 2023

[1]:
```
from psutil import virtual_memory
ram_gb = virtual_memory().total / 1e9
print('Your runtime has {:.1f} gigabytes of available RAM\n'.format(ram_gb))

if ram_gb < 20:
  print('Not using a high-RAM runtime')
else:
  print('You are using a high-RAM runtime!')
```

```
Your runtime has 27.3 gigabytes of available RAM

You are using a high-RAM runtime!
```

[2]:
```
import numpy as np
import scipy.io
from sklearn.metrics.pairwise import rbf_kernel
from scipy.optimize import minimize
import matplotlib.pyplot as plt
```

[7]:
```
data = scipy.io.loadmat('data1.mat')
X_train = data['TrainingX']
y_train = data['TrainingY'].ravel()
X_test = data['TestX']
y_test = data['TestY'].ravel()

# Compute kernel matrix
N_train = X_train.shape[0]
N_test = X_test.shape[0]
subset_size = 500 # size of subset used to estimate gamma
gamma = np.sqrt(np.sum((X_train[:subset_size, None, :] - X_train[:subset_size, :
 ↪]) ** 2) / (subset_size ** 2))
K_train = rbf_kernel(X_train, X_train, gamma=gamma)
K_test = rbf_kernel(X_test, X_train, gamma=gamma)
gamma
```

[7]:
```
9.488604181832654
```

[9]:
```python
# Define objective function and gradient function
def objective_function(w, K_train, y_train, lambda_reg):
    z = y_train * K_train.dot(w)
    J = np.mean(np.log(1 + np.exp(-z))) + lambda_reg * np.sum(w ** 2)
    return J

def gradient(w, K_train, y_train, lambda_reg):
    z = y_train * K_train.dot(w)
    grad = -1/N_train * K_train.T.dot(y_train/(1 + np.exp(z))) + 2*lambda_reg*w
    return grad


# Define initial weight vector and regularization parameter
w0 = np.zeros(N_train)
lambda_reg = 0.1  # Regularization parameter

# Define function to compute Hessian approximation using BFGS
def hessian_approx(w, *args):
    K_train, y_train, lambda_reg = args
    z = y_train * K_train.dot(w)
    diag = np.exp(z) / ((1 + np.exp(z)) ** 2)
    D = np.diag(diag)
    A = D + 2 * lambda_reg * np.eye(N_train)
    H = K_train.T.dot(A).dot(K_train)
    return H

# Train using BFGS
max_iterations = 1000  # Maximum number of iterations for BFGS

history_bfgs = []
def print_gradient(x):
  grad=gradient(x, K_train, y_train, lambda_reg)
  grad_norm = np.linalg.norm(grad)
  history_bfgs.append(grad_norm)
  print("Gradient:", grad_norm)

#result = minimize(objective_function, w0, method='BFGS', jac=gradient,␣
 ↪tol=1e-5, args=(K_train, y_train, lambda_reg), callback=print_gradient)
result = minimize(objective_function, w0, method='L-BFGS-B', jac=gradient,␣
 ↪tol=1e-5, args=(K_train, y_train, lambda_reg),
                  hessp=hessian_approx, options={'maxiter': max_iterations,␣
 ↪'disp': True}, callback=print_gradient)

w = result.x

# Compute test predictions
y_pred = np.sign(K_test.dot(w))
```

```
y_pred[y_pred == 0] = 1
accuracy = np.mean(y_pred == y_test)
print("Test accuracy:", accuracy)
```

```
Gradient: 2.411839400634423e-12
Test accuracy: 0.966
```