CS-E4830 Kernel Methods in Machine Learning
Assignment 3

Kunal Ghosh, 546247

October 20, 2016

Q1. $J_w = l(w) + \lambda/2 \, \|w\|_2$. finding the update direction for stochastic gradient descent. when $l(w)$ is.

a). Quadratic loss $\Rightarrow$ $l(w) = \sum_{i=1}^{N} (w^T x_i - y_i)^2$.

$$\therefore \quad J_w = \sum_{i=1}^{N} (w^T x_i - y_i)^2 + \frac{\lambda}{2} \|w\|_2 .$$

Writing the objective function as an average of N datapoints.

$$(J_{w,\lambda})_i = \left\{ (w^T x_i - y_i)^2 + \frac{\lambda'}{2} \|w\|_2 \right\}$$

$$J_{w,\lambda} = \sum_{i=1}^{N} (J_{w,\lambda})_i$$

where $\lambda' = \lambda/N$

$$\frac{\partial (J_{w,\lambda})_i}{\partial w} = 2 (w^T x_i - y_i)^T x_i + \lambda' w$$

$$\therefore \quad w^{(K+1)} = w^{(K)} - t^{(K)} \left(2 (w^T x_i - y_i)^T x_i + \lambda' w \right)$$

b). Using Logistic loss. $l(w) = \sum_{i=1}^{N} \log(1 + \exp(-y_i (w^T x_i)))$

Writing the objective function as an average of. N data points.

$$(J_{w,\lambda})_i = \left\{ \log(1 + \exp(-y_i (w^T x_i))) + \frac{\lambda'}{2} \|w\|_2 \right\}$$

also here $\lambda' = \lambda/N$.

$$J_{w,\lambda} = \sum_{i=1}^{N} (J_{w,\lambda})_i$$

$$\frac{\partial (J_{w,\lambda})_i}{\partial w} = \frac{1 * \exp(-y_i (w^T x_i))}{1 + \exp(-y_i (w^T x_i))} \cdot -(y_i * x_i) + \lambda' w .$$

$$\therefore \quad w^{(K+1)} = w^{(K)} - t^{(K)} \left(\lambda' w - \frac{y_i \, x_i \, \exp(-y_i (w^T x_i))}{1 + \exp(-y_i (w^T x_i))} \right) .$$

Q2.a). for the dual SVM problem.

$$\max_{\alpha} J(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j K(x_i, x_j).$$

$$\text{s.t} \quad 0 \leq \alpha_i \leq C \quad \forall \; i = 1, \ldots, N.$$

Taking the gradient of the objective w.r.t one of the dual variables. α_K.

$$\nabla_{\alpha_K} J(\alpha) = \left\{ 1 - \underbrace{\frac{1}{2}\sum_{j=1}^{N} \alpha_j y_i y_j K(x_i, x_j)}_{i = K.} - \underbrace{\frac{1}{2}\sum_{i=1}^{N} \alpha_i y_i y_j K(x_i, x_j)}_{j = K.}, \quad 0 \right\}.$$

$$\therefore \nabla_{\alpha_K} J(\alpha) = \begin{cases} 1 - y_i \sum_{j=1}^{N} y_j \alpha_j K(x_i, x_j) & K = i \text{ or } K = j \\ 0 & K \neq i \text{ and } K \neq j \end{cases}$$

$1 - y_i \sum_{j=1}^{N} y_j \alpha_j K(x_i, x_j)$ is a scalar and can be. > 0 or < 0 and. after normalizing. becomes 1. if $1 - y_i \sum_{j=1}^{N} y_j \alpha_j K(x_i, x_j) > 0$ & -1 otherwise.

Now since we are finding the update direction (normalized). w.r.t only <u>one</u> of the directions we disregard the gradient in the other directions or set them as zeros.

$$\therefore \quad \Delta\alpha_j = 0 \quad \forall \; j \neq i$$

$$\Delta\alpha_i = \begin{cases} 1 & \text{if } 1 - y_i \sum_{j=1}^{N} y_j \alpha_j K(x_i, x_j) \\ -1 & \text{otherwise.} \end{cases}$$

for the dual SVM problem. $t = \dfrac{\Delta\alpha^T(1-H\alpha)}{\Delta\alpha^T H\Delta\alpha}$

Q2 (b)

if only one ~~dimension~~ element in $\Delta\alpha$ is getting updated then $\Delta\alpha$ looks like $[0, 0, \dots, \Delta\alpha_i, 0, 0 \dots, 0]^T$

$$\begin{array}{cc} & \begin{array}{cccc} y_1y_2K_{11} & K_{12} & K_{13} & \cdots K_{1N} \end{array} \\ H = N & \begin{bmatrix} \cancel{\quad} & & & \vdots \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ K_{N1} & \cdots & \cdots & K_{NN} \end{bmatrix} \\ & N \end{array} \qquad \alpha = \underset{N\times 1}{\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \vdots \\ \alpha_N \end{bmatrix}}$$

$$H\alpha = N\times 1. \quad \left[y_1\sum_{i=1}^{N} y_i K_{1i}\alpha_i, \; y_2\sum_{i=1}^{N} y_i K_{2i}\alpha_i, \dots, y_N\sum_{i=1}^{N} y_i K_{Ni}\alpha_i\right]$$

$$1 - H\alpha = \left[1 - y_1\sum_{i=1}^{N} y_i K_{1i}\alpha_i, \; 1 - y_2\sum_{i=1}^{N} y_i K_{2i}\alpha_i, \dots, 1 - y_N\sum_{i=1}^{N} y_i K_{Ni}\alpha_i\right]$$

Then. $\Delta\alpha^T(1-H\alpha)$

$$= \left[\Delta\alpha_1\left(1 - y_1\sum_{j=1}^{N} y_j\alpha_j K_{1j}\right), \dots, \Delta\alpha_N\left(1 - y_N\sum_{j=1}^{N} y_j\alpha_j K_{Nj}\right)\right].$$

$\therefore$ for the i^{th} dual variable $\Delta\alpha^T(1-H\alpha)$ would be

$$\Delta\alpha_i\left(1 - y_i\sum_{j=1}^{N} y_j\alpha_j K_{ij}\right).$$

Where $K_{ij} = K(x_i, x_j)$.

$$\therefore\; t = \frac{\Delta\alpha_i\left(1 - y_i\sum_{j=1}^{N} y_j\alpha_j K_{ij}\right)}{\cancel{\Delta\alpha_i\left(1 - y_i\sum_{j=1}^{N} y_j\alpha_j K_{ij}\right)} \;\; \Delta\alpha^T H\Delta\alpha}.$$

Simplifying the denominator now,

$$H\Delta\alpha_i = \left[y_1y_iK_{1i}\Delta\alpha_i, \; y_2y_iK_{2i}\Delta\alpha_i, \dots, y_Ny_iK_{Ni}\Delta\alpha_i\right]$$

$$\Delta\alpha_i^T H\Delta\alpha_i = y_iy_iK_{ii}\Delta\alpha_i\Delta\alpha_i$$

$$\therefore\; t = \frac{\Delta\alpha_i\left(1 - y_i\sum_{j=1}^{N} y_j\alpha_j K_{ij}\right)}{\Delta\alpha_i\Delta\alpha_i y_iy_iK_{ii}}$$

$$\therefore\; t = \Delta\alpha_i\left(1 - y_i\sum_{j=1}^{N} y_j\alpha_j K_{ij}\right) / K_{ii}$$

Since $\Delta\alpha_i$ is just the direction. for $\Delta\alpha_i \neq 0$ it is either 1 or -1 $\therefore\; \Delta\alpha_i\Delta\alpha_i = 1$. Similarly for the case when $y_i = \pm 1$ then $y_iy_i = 1$

Q2. c). The duality gap is given as.

$$dg(\alpha) = f_0(w, \xi_{i}) - g(\alpha).$$

where $g(\alpha)$ = dual softmargin SVM for any dual feasible α.

$$\therefore g(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j K(x_i, x_j).$$

primal softmargin SVM

$$= \text{minimize} \quad \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{N} \xi_i$$

subject to: $1 - y_i(\langle w, \phi(x_i)\rangle) - \xi_i \leq 0 \Rightarrow 1 - y_i(\langle w, \phi(x_i)\rangle) \leq \xi_i$

$\xi_i \geq 0$

for all $i = 1 \ldots N$.

$-\xi_i \leq 0$.

or $\xi_i \geq \max(0, 1 - y_i (\langle \alpha^T \phi(x_i), \phi(x_j)\rangle))$

$$\max\left(0, 1 - y_i \sum_{j=1}^{N} \alpha_j y_j \phi(x_j)^T \phi(x_i)\right).$$

$$\max\left(0, 1 - y_i \sum_{j=1}^{N} \alpha_j y_j K(x_i, x_j)\right).$$

$$\boxed{\text{optimal } w. = \sum_{i=1}^{N} \alpha_i y_i \phi(x_i)}$$

$$= \text{minimize} \quad \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{N} \max\left(0, 1 - y_i \sum_{j=1}^{N} \alpha_j y_j K(x_i, x_j)\right)$$

also. $\frac{1}{2} w^T w \Rightarrow \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \phi(x_i)^T \phi(x_j)$.

$$\therefore f_0(w, \xi_i) \Rightarrow \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j K(x_i, x_j)$$

$$+ C \sum_{i=1}^{N} \max\left(0, 1 - y_i \sum_{j=1}^{N} \alpha_j y_j K(x_i, x_j)\right).$$

$$\therefore \quad f_0(\omega, \xi_i) - g(\alpha).$$

$$= \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j K(x_i, x_j) + C\sum_{i=1}^{N}\max\left(0, 1 - y_i\sum_{j=1}^{N}\alpha_j y_j K(x_i, x_j)\right)$$

$$- \sum_{i=1}^{N}\alpha_i + \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j K(x_i\, x_j).$$

$$= \sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j K(x_i, x_j) + C\sum_{i=1}^{N}\max\left(0, 1 - y_i\sum_{j=1}^{N}\alpha_j y_j K(x_i, x_j)\right) - \sum_{i=1}^{N}\alpha_i$$

Duality Gap

re-arranging a few terms.

$$\Rightarrow \boxed{-\sum_{i=1}^{N}\alpha_i + \sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j K(x_i, x_j) + C\sum_{i=1}^{N}\max\left(0, 1 - y_i\sum_{j=1}^{N}\alpha_j y_j K_{ij}\right).}$$

$$= -\sum_{i=1}^{N}\alpha_i\left(1 - y_i\sum_{j=1}^{N}\alpha_j y_j K_{ij}\right) + C\sum_{i=1}^{N}\max\left(0, 1 - y_i\sum_{j=1}^{N}\alpha_j y_j K_{ij}\right)$$

if $g_i = 1 - y_i \sum_{j=1}^{N} y_j \alpha_j K_{ij}$ Then the above Equation simplifies to.

$$= \boxed{-\sum_{i=1}^{N}\alpha_i g_i + C\sum_{i=1}^{N}\max(0, g_i)}$$

Duality Gap in terms of the gradient g_i , alpha_i and C.

Solution to Question 3(a)

Implementation of the `Stochastic Dual Coordinate Ascent for SVM` as described in the Assignment handout.

```python
########################################################################
# Dual soft-margin SVM problem
# $max_a J(a) = \sum_{i=1}^N a_i + 0.5 \sum_{i=1}^N \sum_{j=1}^N a_i a_j y_i y_j k(x_i, x_j)$
#
# s.t. $0 \leq a_i \leq C$ for all $i = 1 \ldots N$
########################################################################
from __future__ import division
import numpy as np
import pdb

def svmTrainDCGA(K, y, C, N=None):
    """
    Solves the dual softmargin SVM using the
    Stochastic Dual Coordinate Ascent algorithm.

    params K: kernel matrix (dim: (N,N))
    params y: Label vectors (dim: (N,1))
    params C: Regularization parameter (dim: scalar)
    params N: Number of datapoints (dim: scalar)
    return a: Dual SVM variables (dim: (N,1))
    """

    if not N:
        # usually the size of training data
        N = K.shape[1]

    # Initialization
    a = np.zeros((N,1))
    count = 0 # number of iterations
    threshold = np.exp(-3)* np.sqrt(np.trace(K))

    grad = np.ones_like(a)
    duality_gap = np.inf

    y_mat = np.expand_dims(y,axis=1)
    yK_full = np.repeat(y_mat, K.shape[1],axis=1) * K

    condition = True
    while condition:
        count += 1
        # Select a random training example (get a random index in 0:N-1)
        idx = np.random.randint(K.shape[1])

        # calculate the update direction delta_a
        delta_ai = 0
        yK = y * K[idx, :]
        grad_i = 1 - y[idx] * np.dot(a.T,yK)
        delta_ai = 1 if grad_i >= 0 else -1

        # calculate step size t
        t = (1.0*delta_ai * grad_i)/K[idx,idx]

        if t != 0:
            ai_old = np.copy(a[idx])
```

```
            # update gradient
            t_del_a = np.clip(t*delta_ai, -ai_old, C-ai_old)
            grad -= (y[idx] * t_del_a * yK).reshape((grad.shape))

            a[idx] += t_del_a
            if count >= N:
                # # update gradient

                # calculate duality gap
                dg = []
                dg.append(np.dot(a.T,grad))
                dg.append(np.sum(np.maximum(grad, 0)))
                duality_gap = -1 * dg[0] + C * dg[1]
                count = 0
                # since duality gap is only changing here
                # we can also update the condition to stop
                # the iteration here.
                condition = (duality_gap >= threshold)
                # print("{} dg_1 = {} dg_2 = {} {} grad_max = {} grad_min = {} amax {} amin
                ↪   {}".format(duality_gap, dg[0], dg[1], threshold, np.max(grad),
                ↪   np.min(grad),np.max(a), np.min(a)))
    return a
```

Solution to Question 3(b)

In this section we have used the Quadratic program solver of the `CVXOPT` package in python, to solve dual soft-margin SVM (equation 1 as given in the assignment handout).

```
################################################
# Solving the dual soft margin SVM             #
# using quadratic programming solver           #
# from a package (CVXOPT)                      #
################################################

import pdb
import numpy as np
from cvxopt import matrix
from cvxopt import solvers


def svmTrainQP(K, y, C):
    """
    params K: (N_train,N_train) kernel matrix.
    params y: (N_train,) training label vector.
    params C: Scalar

    solvers.qp solves a quadratic program
    of the form:
    min 0.5 x' P x + q' x
     x

    s.t. Gx <= h

    where x' implies x tranpose
    """
    y = np.expand_dims(y, axis=1)
```

```
    N = K.shape[0]

    P = matrix(np.dot(y,y.T) * K, tc='d')
    q = matrix(-1*np.ones_like(y), tc='d')
    G = matrix(np.concatenate((
                np.eye(N)*-1,
                np.eye(N)
                )))
    h = matrix(np.concatenate((
                np.zeros_like(y),
                np.ones_like(y)*C
                )))

    sol = solvers.qp(P,q,G,h)
    return sol['x']
```

Solution to Question 3(c)

We have used the following code snippet to run the svmTrainDCGA and svmTrainQP for different amounts of training data. It also contains the code to time the execution, calculate prediction accuracy and generate plots. The plot thus generated are discussed in the next section.

```
from __future__ import division
# coding: utf-8

# Evaluate performance of algorithm on a fixed test dataset based
# on stochastic dual gradient ascent svmTrainDCGA.py
# and svmTrainQP.y for different training size
# $N_{tr} = [100, 500, 1000, 2000, 3000]$ for $C = 1$

import sys
import time
import pdb
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

from svmTrainDCGA2 import svmTrainDCGA
from svmTrainQP import svmTrainQP

def svmTest(K_test_train, y, alpha):
    return np.sign(np.dot(K_test_train.T,y*alpha))

# load data
print("Loading data ..")
X_test  = np.loadtxt("X_test.txt")
# Adding in a column of ones, since we don't include a bias term
X_test  = np.concatenate((np.zeros((X_test.shape[0],1)),X_test),axis=1)

X_train = np.loadtxt("X_train.txt")
# Adding in a column of ones, since we don't include a bias term
X_train = np.concatenate((np.zeros((X_train.shape[0],1)),X_train),axis=1)

y_test  = np.loadtxt("y_test.txt")
y_train = np.loadtxt("y_train.txt")

# constants
```

```python
C = 1
Ntr = [100,500,1000,2000,3000]

train_Nx, train_Nd = X_train.shape

def linear_kernel(X, Z):
    """
    X is (n,d)
    Z is (m,d)

    Dimension of kernel_mat is (n,m)
    """
    n,d = X.shape
    m,d = Z.shape
    kernel_mat = np.zeros((n,m))
    for row in xrange(n):
        kernel_mat[row,:] = np.dot(Z,X[row,:]).T
    return kernel_mat

dcga_errors = []
qp_errors = []

dcga_time = []
qp_time = []

for ntr in Ntr:
    print("Number of datapoints : {}".format(ntr))
    # get ntr random indexes
    ntr_idxs = np.random.permutation(train_Nx)[:ntr]
    # get subset of data
    X_train_ = X_train[ntr_idxs,:]
    y_train_ = y_train[ntr_idxs]
    print("Computing Kernel ..")
    K_train_train = linear_kernel(X_train_, X_train_)
    K_test_train = linear_kernel(X_train_, X_test)

    print("Running SVM Train ..")
    timeSt = time.clock()
    a_dcga = svmTrainDCGA(K_train_train, y_train_, C)
    time1 = time.clock()
    a_qp = svmTrainQP(K_train_train, y_train_, C)
    time2 = time.clock()

    dcga_time.append(time1-timeSt)
    qp_time.append(time2-time1)

    print("DCGA time : {} QP time : {}".format(dcga_time[-1], qp_time[-1]))
    print("Testing ...")
    y_train_ = np.expand_dims(y_train_, axis=1)
    y_test_ = np.expand_dims(y_test, axis=1)
    # DCGA
    y_pred = svmTest(K_test_train, y_train_, a_dcga)
    dcga_errors.append(100 - (np.sum(y_pred != y_test_)*100/len(y_test_)))
    # QP
    y_pred = svmTest(K_test_train, y_train_, a_qp)
    qp_errors.append(100 - (np.sum(y_pred != y_test_)*100/len(y_test_)))

# Plotting code
plt.plot(Ntr,dcga_errors,'r-')
plt.plot(Ntr, qp_errors,'g-')
plt.xlabel("Number of datapoints.")
```

```
plt.ylabel("Accuracy in percent.")
plt.legend(["DCGA","QP"])


plt.figure()
plt.plot(Ntr, dcga_time,'r-')
plt.plot(Ntr, qp_time,'g-')
plt.xlabel("Number of datapoints.")
plt.ylabel("execution time (in seconds).")
plt.legend(["DCGA","QP"])

plt.show()
```

Solution to Question 3(d)

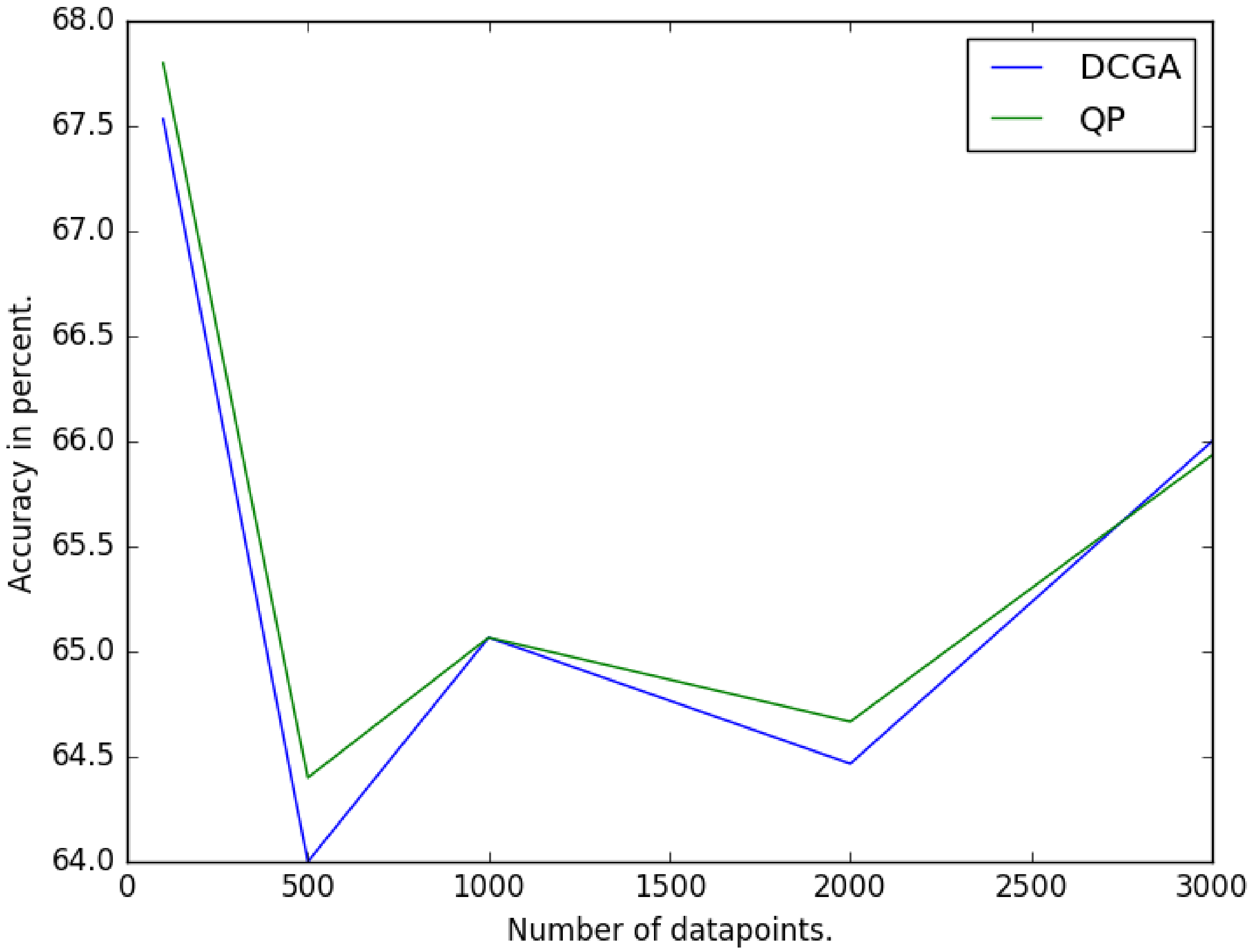

Figure 1: *This figure shows the test accuracy in percentage as a function of the training dataset size. It can be seen that the* `Stochastic Dual Coordinate Ascent (DCGA)` *algorithm performs better than the* `Quadratic Program` *solver for the full dataset but has lower accuracy for small dataset size.* ***Note!*** *that the same trend wasn't visible over multiple runs (see Figure 2 for another run). However, for the full dataset Stochastic DCGA did perform better than the QP version consistently.*

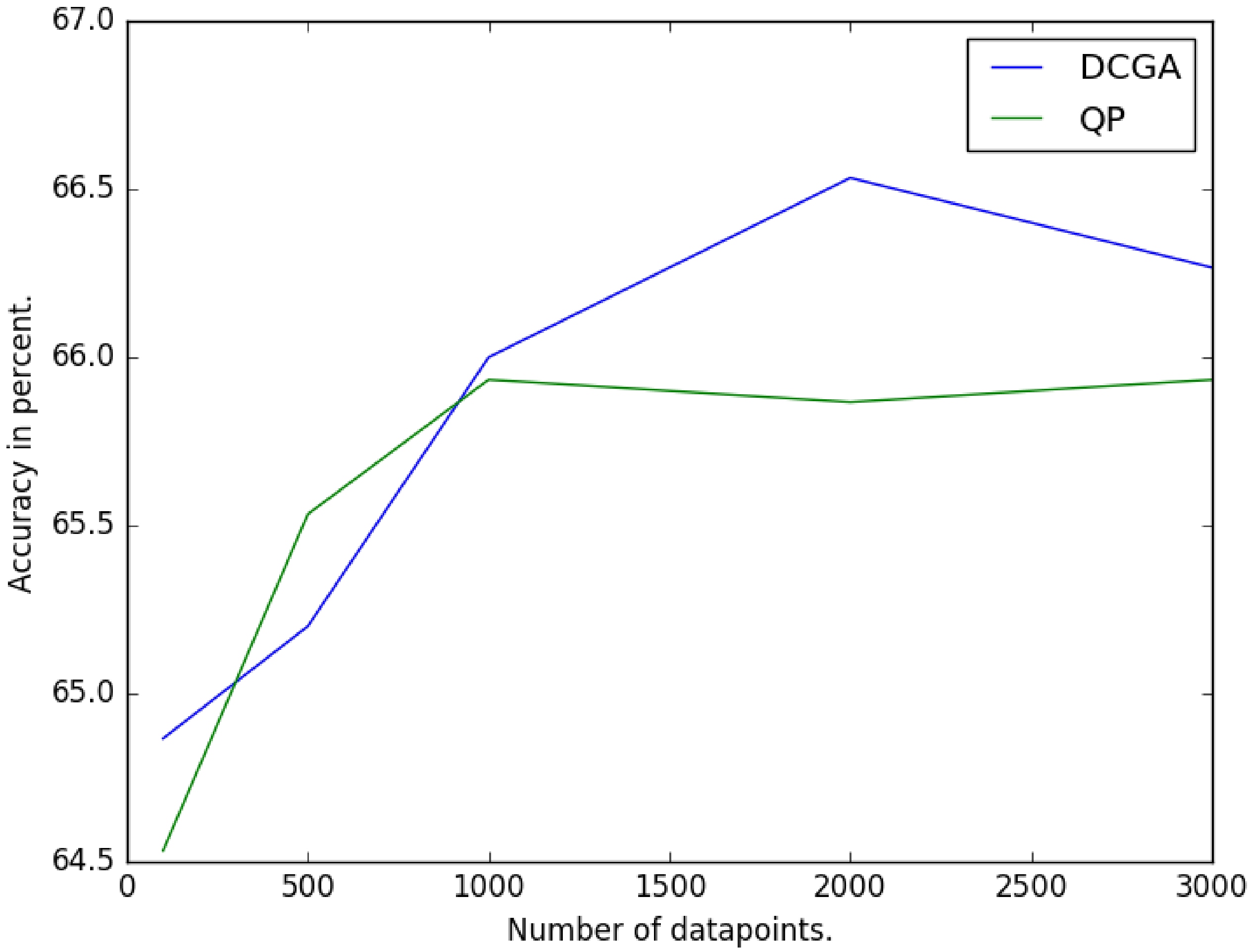

Figure 2: *This figure shows the test accuracy in percentage as a function of the training dataset size. This is an alternate run of the same code used to generate Figure 1, it is clear that* `Stochastic DCGA` *performs bettern than the* `QP` *for a given fixed dataset (all 3000 datapoints) but one cannot make the same conclusion for smaller dataset sizes as there is quite a lot of variance in the accuracy as is seen by comparing this figure with Figure 1.*

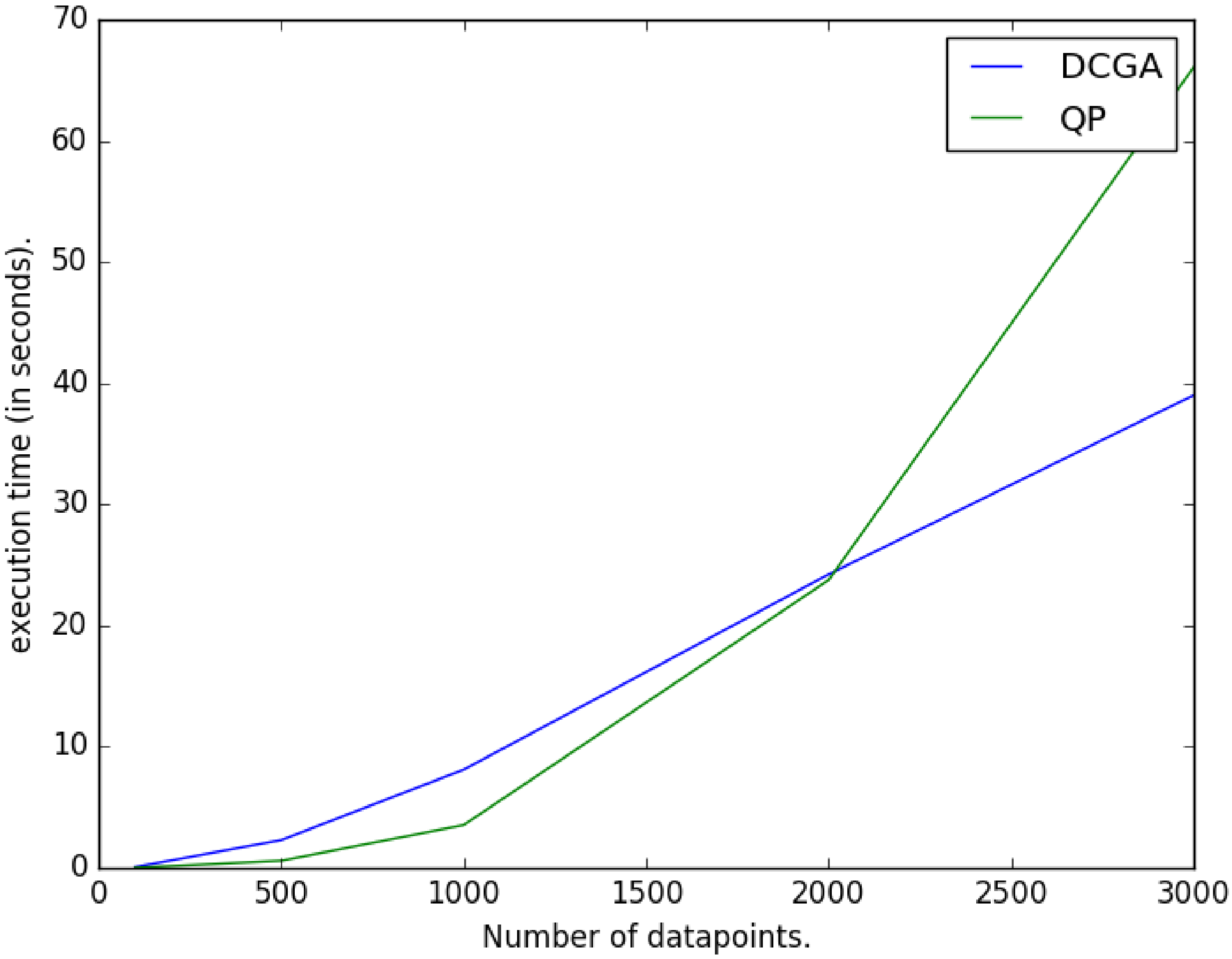

Figure 3: *This figure shows the execution time (seconds), lower is better, as a function of the number of training datapoints. It is quite clear that the Quadratic programming* `(QP)` *solution is faster for small to moderate dataset sizes as compared to the* `Stochastic DCGA` *implementation. However as the dataset size increases* `Stochastic DCGA` *starts to run much quicker than the* `QP`*.*