MAT342 NUMERICAL ANALYSIS FINAL PROJECT

OPTIMIZATION OF MACHINE LEARNING CLASSIFIERS WITH DIRECT SEARCH ALGORITHMS

Rachel You
Gordon College
May 17, 2017

1 Introduction

In machine learning, besides good choices of classifiers and abundant data, good parameters for the classifier chosen are also essential in determining the goodness of classification. Finding the best parameters is an optimization problem. Optimization problems can also all be transformed into minimization problems, and be resolved with minimization algorithms. Minimizing functions in a multivariate case can be complicated when we are not able to take the derivative of the function. Nelder-Mead Algorithm and Method of Simulated Annealing are two methods to minimize a function without taking the derivative. We are going to examine each algorithm and apply each to tuning Support Vector Machine (SVM).

2 Nelder-Mead Algorithm

Nelder-Mead Algorithm starts with an initial simplex, and proceeds by iterations of operations of reflection, expansion, contraction, and shrinkage. Eventually, the simplex will reach the optimal solution. Simplex is a generalization of the notion of a triangle or tetrahedron to arbitrary dimensions. Specifically, a k-simplex is a k-dimensional polytope which is the convex hull of its $k+1$ vertices.

2.1 Algorithm

Suppose we are solving a problem with n-simplex. At each iteration, we use n+1 points, denoted by $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_{n+1}$, with $f(\mathbf{x}_1) \leq f(\mathbf{x}_2) \leq ... \leq f(\mathbf{x}_{n+1})$, where $\mathbf{x}_1$ is the best and $f(\mathbf{x}_{n+1})$ is the worst. A simplex S_k is denoted by $S_k =< x_1, x_2, ...x_{n+1} >$. In two-dimensional, it will be 3 points, forming a 3-simplex, a triangle.

Trial steps are generated by the operations of reflection, expansion, contraction, and shrinkage. A reflected vertex is computed by reflecting the worst vertex, $\mathbf{x}_{n+1}$, through the centroid of the remaining vertices.

$$x_r = (1+\alpha)\bar{x} - \alpha\mathbf{x}_{n+1},$$

where $\alpha = 1$, and $\bar{x}$ is the centroid defined by

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i.$$

The reflected vertex is accepted if $f(\mathbf{x}_1) \leq f(x_r) \leq f(\mathbf{x}_n)$.

If the $f(\mathbf{x}_r) \leq f(\mathbf{x}_1)$, then we produce an expansion.

$$x_e = \gamma x_r + (1-\gamma)\bar{x},$$

where $\gamma = 2$. The expansion vertex is accepted if $f(x_e) < f(\mathbf{x}_1)$, otherwise the reflected vertex is accepted.

If $f(\mathbf{x}_n) \leq f(x_r)$, then a contraction is computed. If $f(\mathbf{x}_{n+1}) \leq f(x_r)$, then the internal contraction vertex is computed as

$$x_c = \beta \mathbf{x}_{n+1} + (1-\beta)\bar{x},$$

otherwise, the external contraction vertext is computed as

$$\hat{x_c} = \beta x_r + (1-\beta)\bar{x},$$

where $\beta = \frac{1}{2}$. The contraction vertex is accepted if it has a lower function value than $\mathbf{x}_n$.

If both reflection vertex and contraction vertex are rejected, then the simplex is shrunk. Each vertex $\mathbf{x}_i$, except the best point $\mathbf{x}_1$, is replaced by the point halfway between $\mathbf{x}_i$ and $\mathbf{x}_1$.

$$x_i \leftarrow \frac{\mathbf{x}_i + \mathbf{x}_1}{2}.$$

Finally, function values of the accepted points are sorted with the remaining point(s), and the next iteration begins.

2.2 Stopping Criteria

There are multiple stopping criteria that may be used. Here we introduce four, the first three discussed by Dennis and Woods, and the last one discussed by Cheney and Kincaid.

The first one is to halt when the standard error of the function values falls below some threshold value:

$$\frac{1}{n}\sum_{i=1}^{n+1}(f(x_i) - \bar{f})^2 < \epsilon_1,$$

where $\bar{f}$ is the average of the function values and $\epsilon_1 > 0$ is a preset tolerance value.

Another stopping criterion bases on how far the simplex moves at an iteration, halting when:

$$\frac{1}{n}\sum_{i=1}^{n} ||\mathbf{x}_i^k - x_i^{k+1}||^2 < \epsilon_2,$$

where $\epsilon_2 > 0$ and x_i^{k+1} is the i^{th} unordered point in the $k+1^{\text{st}}$ simplex. The main objection to this method is that the left-hand side for a shrinkage step will be greater than the value for a contraction step, and shrinkage occurs frequently when the simplex is in a neighborhood of a local minimizer.

Therefore, the third stopping criterion is introduced by Woods:

$$\frac{1}{\Delta}\max_{2\leq i\leq n+1} ||\mathbf{x}_i - \mathbf{x}_1|| \leq \epsilon_3,$$

where $\Delta = \max(1, ||(x)_1)$ and $\epsilon > 0$. This is a measure of the relative size of the simplex.[1]

The fourth stopping criterion tests whether the relative flatness is small[2], which is

$$\frac{F(x_0) - F(x_n)}{|F(x_0)| - |F(x_n)|} < \epsilon_4$$

2.3 Advantages

- Robustness: It tolerates noise in the function values.
- Simplicity in programming: Trail points are obtained using very simple algebraic manipulations and these points are accepted or rejected based only on their function values.
- Low overhead in storage and computation: When the number of variable is small, this algorithm is often competitive with much more complex algorithms that require a great deal of overhead in storage and algebraic manipulations.
- Don't require gradient information.

2.4 Disadvantage

- Stopping criteria is not guaranteed to converge.
- Can stuck at local minimum instead of absolute minimum.

3 Method of Simulated Annealing

Originally, the name and inspiration of Simulated Annealing come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. Method of Simulated Annealing (SA) is good at minimizing difficult functions, especially discrete functions. It uses probabilities to assign new values to each data point and evaluate to get to another iteration.

3.1 Algorithm

The algorithm generates a sequence of points $x_1, x_2, x_3, ...$ and it's hoped the the minimum of function values of the points will converge.

For iteration k, we first generate random points $u_1, u_2, ..., u_m$ in a large neighborhood of x_k. The minimum function value will be selected:

$$F(u_j) = \min\{F(u_1), F(u_2), ..., F(u_m)\}$$

If the newly computed function value $F(u_j)$ is less than $F(x_k)$, then set $x_{k+1} = u_j$. If $F(u_j)$ is not better than $F(x_k)$, we assign a probability p_i to u_i by formula:

$$p_i = e^{\alpha[F(x_k)-F(u_i)]}(1 \leq i \leq m)$$

where α is a positive parameter set by user code. The probabilities are normalized by dividing each by their sum.

$$S = \sum_{i=1}^{m} p_i$$

$$p_i \leftarrow p_i/S$$

Finally, a point is chosen randomly from points $u_1, u_1, ..., u_m$ with probabilities p_i taking into account.[2]

3.2 Advantages

- Avoid getting stuck at local optimums: the complicated choice of x_{k+1} can help considering points that are not stuck at the same local minimum.[3]
- Don't require gradient information.

3.3 Disadvantages

- Repeatedly annealing with a $1/\log(k)$ schedule is very slow, especially if the cost function is expensive to compute.
- For problems where the energy landscape is smooth, or there are few local minima, SA is overkill — simpler, faster methods (e.g., gradient descent) will work better. But generally don't know what the energy landscape is for a particular problem.
- Heuristic methods, which are problem-specific or take advantage of extra information about the system, will often be better than general methods, although SA is often comparable to heuristics.
- The method cannot tell whether it has found an optimal solution. Some other complimentary method (e.g. branch and bound) is required to do this.[4]

4 Python Implementations and Simple Examples

Here a small example is implemented to visualize both methods in 2D.

The function we are using is $f(x, y) = x^2 + y^2$, for which we can easily know the minimum to be at $(0, 0)$.

One thing to be noted is that although the function used here has 2 variables, the implementation is generalized and can be used for higher dimensions.

4.1 Nelder-Mead Implementation with 2D Example

Here we use the third stopping criterion, just by choice.

```
import numpy as np
import matplotlib.pyplot as plt

# define the function
def f(x):
    return (np.power(x[0],4)-2*np.power(x[0],2)+x[0])
        +(np.power(x[1],4)-2*np.power(x[1],2)+x[1])

# define constants to be used
alpha = 1.0
gamma = 2.0
beta = 0.5
epsilon = 0.000001

# initialize x array and other variables
x = np.array([[1.0,1.5],[1.2,0.9],[1.1,1.3]])
fx = np.array([f(x_) for x_ in x])
fxsort = fx.argsort()
fx = fx[fxsort]
x = x[fxsort]
n = 2
count = 0

# draw contour plot
xlist = np.linspace(-2.0, 2.0, 100) # Create 1-D arrays for x,y dimensions
ylist = np.linspace(-2.0, 2.0, 100)
X,Y = np.meshgrid(xlist, ylist) # Create 2-D grid xlist,ylist values
Z = f(np.array([X,Y]))
plt.contour(X, Y, Z, [-5.0,-4.0,-3.9,-3.8,-3.7,-3.6,-3.5,-3.0,-2.0,-1.0,
        0.0,1.0,2.0,3.0], colors = 'b', linestyles = 'solid')

# plot the initial simplex
p = plt.Polygon(x, closed=True)
ax = plt.gca()
ax.add_patch(p)
```

```
# iteration
# reflection
while True:
    count += 1
    xnew = []
    xbar = 1/float(n)*np.sum(x[:n-1,:],axis=0) # centroid
    xr = (1+alpha)*xbar - alpha*x[n]
    fxr = f(xr)
    if fx[0] <= fxr <= fx[n-1]:
        xnew = xr #reflection_accepted = true
        print "reflection"
    elif fxr <= fx[0]:
        # expansion
        xe = gamma*xr + (1-gamma)*xbar
        if f(xe) < fx[0]:
            xnew = xe # expansion_accepted = true
            print "expansion"
        else:
            xnew = xr # reflection_accepted = true
            print "reflection"
    else: # fx[n-1] <= fxr:
        # contraction
        if fx[n] <= fxr:
            # internal contraction
            xc = beta*fx[n]+(1-beta)*xbar
        else:
            # external contraction
            xc = beta*xr + (1-beta)*xbar
        if f(xc) < fx[n-1]:
            xnew = xc
            print "contraction"
        else: # both reflection vertex and contraction vertex are rejected
            # shrinkage
            for i in range(1,n):
                x[i] = (x[i] + x[0])/2.0
            xnew = (x[n] + x[0])/2.0
            print "shrinkage"
    x[n] = xnew
    # resort the array
    fx = np.array([f(x_) for x_ in x])
    fxsort = fx.argsort()
    fx = fx[fxsort]
    x = x[fxsort]
    # plot the simplex
    p = plt.Polygon(x, closed=True, fill=False)
    ax = plt.gca()
    ax.add_patch(p)
    Delta = max(1.0,np.sum(np.abs(x[0])**2,axis=-1)**(1./2))
    norm_array = np.array([np.sum(np.abs(x[i] - x[0])**2,axis=-1)**(1./2)
        for i in range(1,n+1)])
    rel_size = 1.0/Delta*max(norm_array)
    if rel_size < epsilon:
        break

bestX = x[0]
bestF = fx[0]

print "number of iterations:"
print count
print "Best variable result:"
print bestX
print "Best function value:"
print bestF

annotation = "f(%3.8f,%3.8f)=%3.8f" % (bestX[0],bestX[1],bestF)
plt.title("Nelder-Mead Simple 2D Example with f = (x^4-2x^2+x)+y^4-2y^2+y")
plt.xlabel("x")
plt.ylabel("y")
plt.plot(bestX[0],bestX[1],'^')
plt.annotate(annotation, (bestX[0],bestX[1]))
plt.show()
```

Output shows the operations performed in each iteration and the total number of iterations taken. (Plot: Figure 1)

```
expansion
contraction
contraction
expansion
reflection
reflection
shrinkage
```

```
shrinkage
contraction
contraction
shrinkage
shrinkage
shrinkage
shrinkage
shrinkage
shrinkage
number of iterations:
16
Best variable result:
[−1.11923475 −1.09887952]
Best function value:
−4.11119256687
```

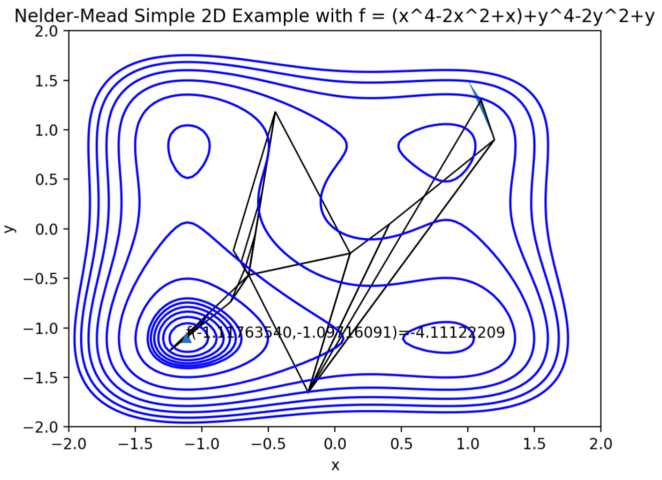

Figure 1: Nelder Mead Example, I.C. (1.0,1.5), (1.2,0.9), (1.1,1.3)

This first initial condition (1.0,1.5), (1.2,0.9), (1.1,1.3) gives us a pretty good result -4.11119256687, as the minimum calculated by WolframAlpha is -4.11235 at (-1.10716,-1.10716).

However, with a slightly different initial condition (1.0,1.5), (1.0,1.4), (1.1,1.5), the result gets worse. With 23 iterations, we get -3.96500547771 at (-0.928125 -1.103125) in Figure 2. It is not clear why the result doesn't get better, since we are close to the global minimum instead of stuck at a local minimum.

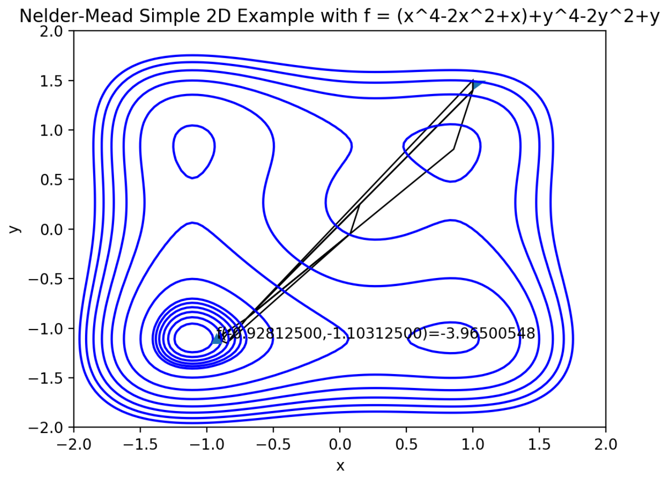

Figure 2: Nelder Mead Example, I.C. (1.0,1.5), (1.0,1.4), (1.1,1.5)

4.2 Simulated Annealing Implementation with 2D Example

Since Simulated Annealing does not have a good stopping criterion, multiple trials have to be performed to find a good number of iterations.

```
import numpy as np
import matplotlib.pyplot as plt

# define the function
def f(x):
    return (np.power(x[0],4)−2*np.power(x[0],2)+x[0])
        +(np.power(x[1],4)−2*np.power(x[1],2)+x[1])

# define constants to be used
n = 2
iterations = 30
m = 100
radius = 2.0
sigma = 5.0
alpha = 1.0

# define initial conditions
x = np.zeros((iterations,2))
fx = np.zeros(iterations)
```

```
x[0] = [1.0,1.5]
fx[0] = f(x[0])

# draw contour plot
xlist = np.linspace(-2.0, 2.0, 100) # Create 1-D arrays for x,y dimensions
ylist = np.linspace(-2.0, 2.0, 100)
X,Y = np.meshgrid(xlist, ylist) # Create 2-D grid xlist,ylist values
Z = f(np.array([X,Y]))
plt.contour(X, Y, Z, [-5.0,-4.0,-3.9,-3.8,-3.7,-3.6,-3.5,-3.0,-2.0,-1.0,
        0.0,1.0,2.0,3.0], colors = 'b', linestyles = 'solid')

#iteration
for k in range(0,iterations-1):
    u = np.zeros((m,n))
    count = 0
    while count < m:
        unew = np.zeros(n)
        for i in range(0,n):
            unew[i] = np.random.normal(x[k,i],sigma)
        distance = np.sum(np.abs(unew-x[k])**2,axis=-1)**(1./2)
        if distance < radius:
            u[count] = unew
            count += 1

    fu = np.array([f(u_) for u_ in u])
    j = np.argmin(fu)

    # accept the new variable if function value gets better
    if fu[j] < fx[k]:
        x[k+1] = u[j]
        fx[k+1] = f(x[k+1])
    else:
        p = np.zeros(m)
        for i in range(0,m):
            p[i] = np.exp(alpha*(fx[k]-fu[i]))
        S = np.sum(p)
        p = p/S
        xi = np.random.rand()
        for i in range(0,m):
            if xi < np.sum(p[:i]):
                x[k+1] = u[i]
                fx[k+1] = f(x[k+1])

bestX = x[np.argmin(fx)]
bestF = np.min(fx)

# print the function values changing
print fx
print "Best␣variable␣result:"
print bestX
print "Best␣function␣value:"
print bestF

annotation = "f(%3.8f,%3.8f)=%3.8f" % (bestX[0],bestX[1],bestF)
# plot
plt.plot(x[:,0],x[:,1])
plt.title("Simulated␣Annealing␣Simple␣2D␣Example␣\n
␣␣␣␣␣␣␣␣with␣f␣=␣x^4−2x^2+x+y^4−2y^2+y")
plt.xlabel("x")
plt.ylabel("y")
plt.plot(bestX[0],bestX[1],'^')
plt.annotate(annotation, (bestX[0],bestX[1]))
plt.show()
```

Output with initial condition (1.0,1.5): (Plot: Figure 3)

```
[  2.0625      −1.21778208  −3.92876839   9.58461764  −3.94375237
  −3.95570613  −3.02678706  −3.60393895  −3.96656617  −2.28259712
  −4.08328778  −3.73289963  −4.10746298  −3.50612466  −4.10595261
  −1.33336042  −3.49883171  −3.86614938  −3.96198828   9.32012524
  −3.58886781  −4.05768169   5.40391305  −3.8853161   −4.07253786
  29.39800908  −4.0900892    0.22363373  −3.73468861  −4.0945986 ]
Best variable result:
[−1.09787143 −1.0780657 ]
Best function value:
−4.10746297788
```

Output with initial condition (1.1,1.2): (Plot: Figure 4)

```
[  0.5377      −1.21965853  −3.50153028  −4.09333512  25.7548202
  −3.96067359  −3.9814316   −4.09616206   5.57256979  −4.00986528
   7.61192087  −3.99912214  −4.09068264  −0.61914329  −3.7425246
  −3.86116518  −3.94834034  −4.11181432  −0.87453229  −3.82497098
  −3.8932599   −3.90816478  −0.32069523  −3.80752951  −3.9597208
```

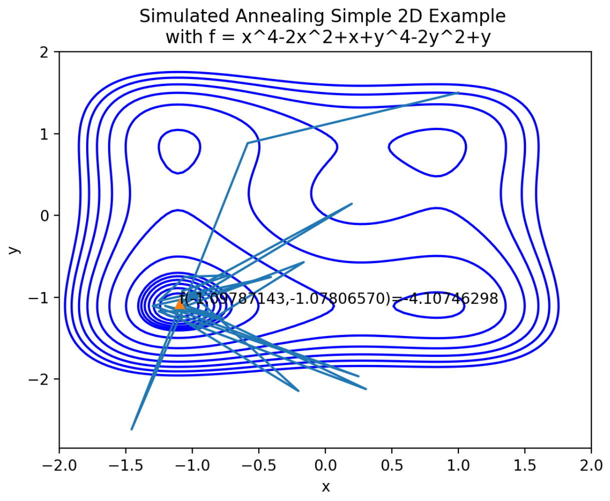

Figure 3: Simulated Annealing Example with I.C. (1.0,1.5)

```
   0.13929305  −3.93950367  −4.00272135  −4.02458903  −4.04561459]
Best variable result:
[−1.10531862 −1.09732896]
Best function value:
−4.11181431633
```

For Simulated Annealing, it is not likely to stuck at a local minimum or some weird point, but it is also hard to control the accuracy.

4.3 Comparison

Nelder-Mead Algorithm gets small error when having a good initial condition, but gets larger error when having a bad initial condition. What Simulated Annealing gets relatively more stable, although the accuracy is not guaranteed as it's using random numbers.

Also, Nelder-Mead Algorithm converges much faster, while Simulated Annealing doesn't converge.

Method	steps	I.C.	Best [x y]	error
NM	16	[[1.0,1.5],[1.2,0.9],[1.1,1.3]]	[-1.11923475 -1.09887952]	0.00115743313
NM	22	[[1.0,1.5],[1.0,1.4],[1.1,1.5]]	[-0.928125 -1.103125]	0.14734452229
SA	30	[1.0,1.5]	[-1.09787143 -1.0780657]	0.00488702212
SA	30	[1.1,1.2]	[-1.10531862 -1.09732896]	0.00053568367

Table 1: Nelder-Mead vs. Simulated Annealing

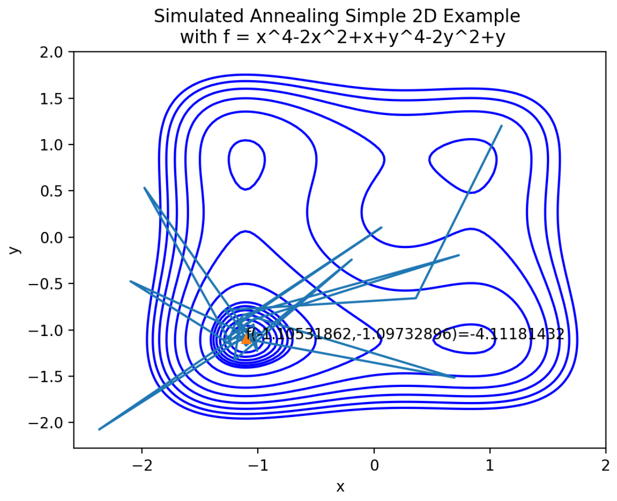

Figure 4: Simulated Annealing Example with I.C. (1.1,1.2)

5 Application to SVM

Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane, to maximize distances to nearest point (margin).

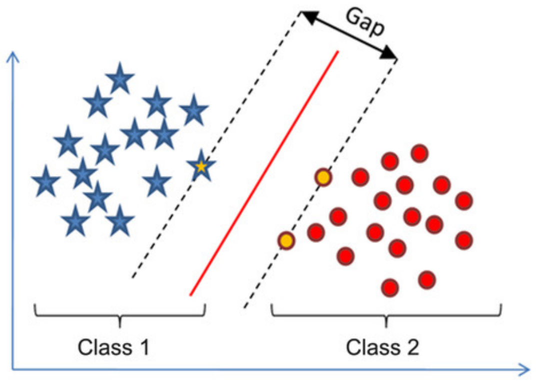

Figure 5: SVM Visualization

Two essential numerical parameters are C and gamma. C controls tradeoff between smooth decision boundary and classifying training points correctly, and larger C tends to make training points more correct. γ defines how far the influence of a single training example reaches, with low values meaning "far" and high values meaning "close". Here, we are going to use both Nelder-Mead Algorithm and Simulated Annealing to find the best parameters for an SVM with python package ”sklearn” that recognizes hand written numerical digits that are represented in 28×28-pixel arrays.

Since the methods originally are designed to find minimums, we take the negatives of accuracy scores to turn the maximization problem into a minimization problem.

5.1 Nelder-Mead

With Nelder-Mead method, it seems that starting with random initial conditions will lead local optimum.

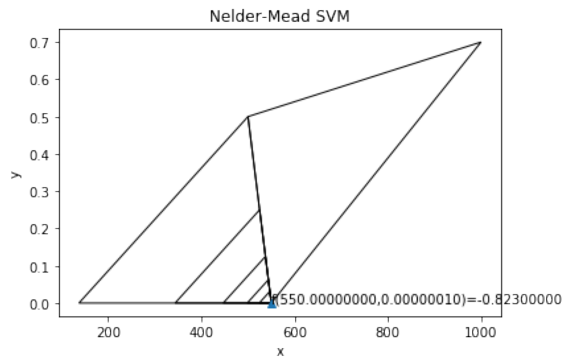

Figure 6: I.C. $x = [[100, 1], [500, 0.5], [1000, 0.7]]$, epsilon = 0.05

When we start with reasonable initial condition $x = [[1000, 1], [5000, 0.1], [10000, 0.01]]$, it in the end finds the best fit at $[10000, 0.01]$ with accuracy score 0.96433333 after 8 rounds.

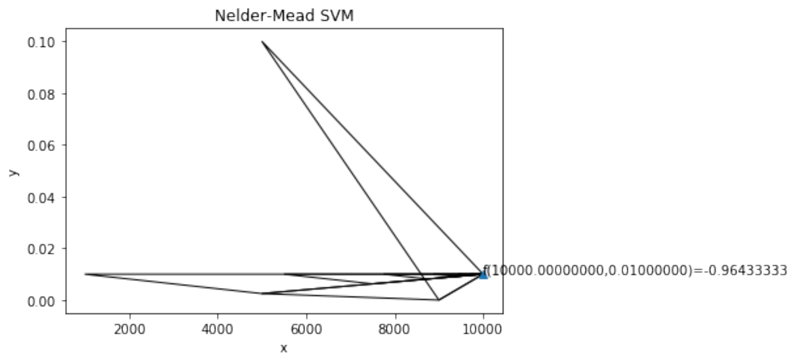

Figure 7: I.C. $x = [[1000, 1], [5000, 0.1], [10000, 0.01]]$, epsilon = 0.05

In fact, Nelder-Mead depends so much on initial condition in this case especially when we don't use a very large epsilon value. If we change the initial condition to $x = [[10002, 1.1], [5050, 0.09], [9999, 0.02]]$, it gives a better accuracy score 0.967 at $[9999, 0.02]$.

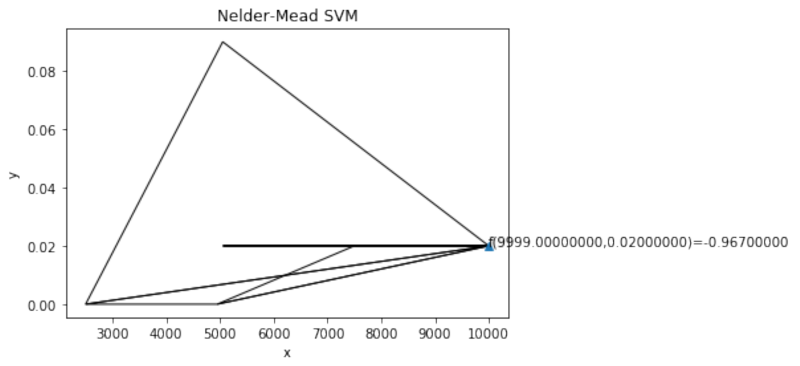

Figure 8: I.C. $x = [[10002, 1.1], [5050, 0.09], [9999, 0.02]]$, epsilon = 0.05

5.2 Simulated Annealing

Simulated Annealing is taking really long time when having the classifier to fit the data, since it generates a lot of random points each iteration. Randomly selected initial value does not always work well. An initial condition too far off can make it not know which direction to go.

When the initial parameters is set to a ridiculous point $x = [100.0, 150.0]$, the accuracy score never goes to more than 0.10833333.

If we use a reasonable value $x = [10000.0, 0.1]$ as initial condition, generating 5 iterations with each iteration taking 10 random points, we are able to get an accuracy score of 0.96766666667.

5.3 Comparison to GridSearch

Comparing to sklearn's GridSearch algorithm that selects the best fit from a set of values, Nelder-Mead might be able to find a better fit given a smaller epsilon, since the values to be tested are not constrained by the few choices provided. However, the initial conditions still have to be reasonable.

6 Conclusion and Future Research Ideas

Nelder-Mead Algorithm and Simulated Annealing are useful for finding optimized values for functions that we are not able to calculate the derivatives. Each still has its own disadvantages. Nelder-Mead Algorithm can suffer from getting stuck at local minimums, while

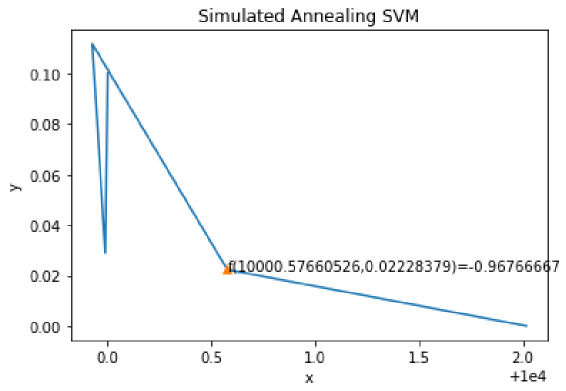

Figure 9: I.C. $x = [10000.0, 0.1]$, 5 iterations with 10 random points each iteration

Simulated Annealing sometimes require too much computation resources when calculating complicated functions.

Better implementations might be available to reduce repetitive calculations of functions and thus reduce time needed. Rescaling functions might also allow the algorithms to find better results when each axes are not of the same scale.

In selecting parameters for SVM, it might be better to first make use of sklearn's GridSearch with exponentially different points, finding the relatively best point, and then use Nelder-Mead or Simulated Annealing with a larger neighborhood at that specific area to find better results. This will allow us to reduce getting stuck at local minimum and spending too much time because of the computationally expensiveness of Simulated Annealing. Also, we could modify Simulated Annealing to use exponentially different radius to accommodate this specific type of problem.

References

[1] Dennis, John E. JR. and Woods, Daniel J. Optimization on Microcomputers: The Nelder-Mead Simplex Algorithm. *New Computing Environments: Microcomputers in Large-scale Computing*, 116-121.

[2] Cheney, Ward. and Kincaid, David *Numerical Mathematics and Computing*, 580-582.

[3] Simulated Annealing
`http://bamboo.ee.ntu.edu.tw/LabWebsite/Frame/DO/Simulated%20annealing.pdf`

[4] Disadvantages of Simulated Annealing
`https://cs.adelaide.edu.au/~paulc/teaching/montecarlo/node140.html`