BACHELOR DEGREE IN DATA SCIENCE & ENGINEERING
UNIVERSITAT POLITÈCNICA DE CATALUNYA

CONSTRAINED OPTIMIZATION

LAB ASSIGNMENT
SUPPORT VECTOR CLASSIFIER IN AMPL

REPORT

Adrián Cerezuela Hernández
Ramon Ventura Navarro

INDEX

1. IMPLEMENTATION OF PRIMAL AND DUAL QUADRATIC FORMULATIONS

In this project we are asked to develop a Support Vector Classifier in AMPL, which purpose is to find two parallel hyperplanes ($w^T x + \gamma$) separating two classes such that the classification errors are minimized and the margin between the hyperplanes is maximized. These opposite objectives are weighted by parameter $\nu \in \mathbb{R}$ (we will call it c in our AMPL codes). More specifically, we are asked to implement its primal and dual quadratic formulations, then to check their correctness and finally to analyze the results obtained.

Firstly, the primal quadratic formulation is the following one:

$$min_{(w,\gamma,s)\in\mathbb{R}^{n+1+m}} \frac{1}{2} w^T w + \nu \sum_{i=1}^{m} s_i$$

$$\text{s. to } y_i(w^T x_i + \gamma) + s_i - 1 \leq 0 \qquad i = 1, \ldots, m$$

$$-s_i \leq 0 \qquad i = 1, \ldots, m$$

Based on this formulation, our AMPL code implements it as follows:

```
#parameters
param m > 0;
param n > 0;

param c >= 0;

param y{1..m};
param x{{1..m},{1..n}};

#variables
var w{1..n};
var gamma;
var s{1..m} >= 0;

#objective function: primal quadratic formulation
minimize objf1 : 0.5*sum{i in {1..n}}(w[i]^2) + c*sum{i in {1..m}}s[i];

#constraint
subject to c1{i in {1..m}}: -y[i]*(sum{j in {1..n}}(x[i,j]*w[j])+gamma)
                            -s[i]+1 <= 0;
```

On the other hand, the dual quadratic formulation is the next one:

$$max_{\lambda\in\mathbb{R}^m} \sum_{i=1}^{m} \lambda_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \lambda_i y_i \lambda_j y_j K_{ij}$$

$$\text{s.to } \sum_{i=1}^{m} \lambda_i \, y_i = 0$$

$$0 \leq \lambda_i \leq \nu \qquad i = 1, \ldots, m$$

And our AMPL code implements it as follows:

```
#parameters
param m > 0;
param n > 0;

param c >= 0;

param y{1..m};
param x{{1..m},{1..n}};

#variables
var lambda{1..m} >= 0, <= c;

#objective function: primal quadratic formulation
#linear kernel is used: Kij -> sum{k in 1..n}A[i,k]*A[j,k]
maximize objf2: sum{i in {1..m}}lambda[i]
                - 0.5*sum{i in {1..m}, j in {1..m}}(lambda[i]*y[i]*
                lambda[j]*y[j]*(sum{k in {1..n}}(x[i,k]*x[j,k])));

#constraints
subject to c2 : sum{i in {1..m}}(y[i]*lambda[i]) = 0;
```

2. MODEL VALIDATION AND ANALYSIS OF THE RESULTS

Parameters and datasets

Once we have our primal and dual problems modeled, we need the data that will help us checking the correctness of our models.

We will divide our evaluation in two main datasets, one generated by ourselves and another one downloaded from the internet. Each one will of them will be divided into two parts: training and test. Later we will explain their uses to obtain results.

The parameters used are m (number of instances), n (number of features) and c. That last parameter c is usually an integer number between 0 and 10, according to the SVC Python manual. After some tries, we have decided c to take as value 2.

We are provided of a random point generator, which brings us a dataset of p points $\in \mathbb{R}^4$, generated with a specific seed. The training set, consisting of 100 points, will be generated using one of our DNIs' number, and the second one, consisting of 50 points, using the other DNI number. It can be seen that some classification values have an asterisk. That means that are wrongly classified and don't belong to the class assigned.

As we mentioned before, to make sure our SVM implementation works well we will apply it to a dataset different from the ones obtained with the generator. Using as our source the OpenML website, we are going to be working with a dataset related to diabetes diagnostics, that decides through a binary-valued variable whether the patient shows signs of diabetes according to World Health Organization criteria. It contains 768 instances, of which we will take the first 700 (350 for training – 350 for testing), and has 8 numeric features and one symbolic. The symbolic feature determines the class value.

Although we have the data, it requires a certain format for an AMPL *.dat* file. That is why we have developed a simple Pyhton code that handles that. It will be included in the zip containing all the coding-related files.

Compute of accuracies and model validation

In order to fulfill a proper model validation, two additional executable files have been created (*svm_primal.run* and *svm_dual.run*). The main idea of these two codes is to calculate both accuracies for the training and testing datasets, independently for the primal and dual formulations of the problem. Both programs work either way for the generated dataset or the one found on the internet, we just simply have to change the one we are willing to use in each *.run* file. Let's firstly take a look at the proper details of the codes and then study the conclusions for each dataset, which we are now going to refer to them as *dataset1* for the generated one and *dataset2* as the internet-found set (*tr* stands for training and *te* stands for testing).

Right below, the implementation of the accuracies based on the primal problem:

```
# ---PRIMAL PROBLEM : TRAINING SET---
reset;
model svm_primal.mod;

# Choose dataset
#data data1_tr.dat; # generated training dataset
data data2_tr.dat; # internet found training dataset


option solver gurobi;

solve;
display w, s, gamma;
```

```
# ---TRAINING ACCURACY---

# We calculate the predicted values for the training dataset
param predictions_tr{i in 1..m};
for{i in 1..m}{
    if (sum{j in 1..n} w[j]*x[i,j] + gamma) > 0 then let predictions_tr[i] := 1;
    else let predictions_tr[i] := -1;
    #display predictions_tr[i];
}

# Once we have the predicted values, we compare them to the real ones
var matching_values_tr := 0;
for{i in 1..m}{
    # For every matching value we add one to the total amount
    if predictions_tr[i] == y[i] then let matching_values_tr := matching_values_tr + 1;
}

# We now compute the training accuracy percentage
var accuracy_tr = (matching_values_tr / m) * 100;
display accuracy_tr;

# ---ACCURACY VALIDATION---

# Definition of the new parameters
param m_te >= 1, integer;
param y_te{i in 1..m};
param x_te{1..m,1..n};

# Choose dataset accordingly
#data data1_te.dat # generated testing dataset
data data2_te.dat; # internet found training dataset

# We calculate the predicted values
param predictions_te{i in 1..m_te};
for{i in 1..m_te}{
    if (sum{j in 1..n} w[j]*x_te[i,j] + gamma) > 0 then let predictions_te[i] := 1;
    else let predictions_te[i] := -1;
    #display predictions_te[i];
}

# Once we have the predicted values, we compare them to the real ones
var matching_values_te := 0;
for{i in 1..m_te}{
    # For every matching value we add one to the total amount
    if predictions_te[i] == y_te[i] then let matching_values_te := matching_values_te + 1;
}

# We now compute the validation accuracy
var accuracy_te = (matching_values_te / m_te) * 100;
display accuracy_te;
```

As we can see, this program firstly solves the primal problem using the previously coded *svm_primal.mod* and compares if each predicted value matches the datasets real value for y. Once for the training dataset and once for the testing one. Then it finally prints both accuracies by comparing how many predicted values match with the real ones.

In this program we find no problems as we count with each of the variables needed to decide the value for each prediction following the next criteria:

$$w^T \cdot x + \gamma = 0$$

Where the value of the prediction will be -1 if the expression is negative and 1 if it turns to be positive.

Let's now take a look at *svm_dual.run*:

```
# ---DUAL PROBLEM : TRAINING SET---
reset;
model svm_dual.mod;

# Choose dataset
#data data1_tr.dat; # generated training dataset
data data2_tr.dat; # internet found training dataset

option solver gurobi;

solve;
display lambda;

# ---FINDING NECESSARY VARIABLES---

# In order to compute the training and test accuracies
# we find 'w' and 'gamma' from the dual problem solution
param w{1..n};
let {j in {1..n}} w[j] := sum{i in {1..m}} lambda[i]*x[i,j]*y[i];

param gamma;
for {i in {1..m}} {
    if lambda[i] > 0.01 and lambda[i] < c*0.1 then {
        # A support vector point was found
        let gamma := 1/y[i] - sum{j in {1..n}} w[j]*x[i,j];
        break;
    }
}
display w, gamma;

# ---TRAINING ACCURACY---

# We calculate the predicted values for the training dataset (data1.dat)
param predictions_tr{i in 1..m};
for{i in 1..m}{
    if (sum{j in 1..n} w[j]*x[i,j] + gamma) > 0 then let predictions_tr[i] := 1;
    else let predictions_tr[i] := -1;
    #display predictions_tr[i];
}

# Once we have the predicted values, we compare them to the real ones
var matching_values_tr := 0;
for{i in 1..m}{
    # For every matching value we add one to the total amount
    if predictions_tr[i] == y[i] then let matching_values_tr := matching_values_tr + 1;
}

# We now compute the training accuracy percentage
var accuracy_tr = (matching_values_tr / m) * 100;
display accuracy_tr;

# ---ACCURACY VALIDATION---

# Definition of the new parameters
param m_te >= 1, integer;
param y_te{i in 1..m};
param x_te{1..m,1..n};
```

```
# Choose dataset accordingly
#data data1_te.dat # generated testing dataset
data data2_te.dat; # internet found training dataset

# We calculate the predicted values
param predictions_te{i in 1..m_te};
for{i in 1..m_te}{
    if (sum{j in 1..n} w[j]*x_te[i,j] + gamma) > 0 then let predictions_te[i] := 1;
    else let predictions_te[i] := -1;
    #display predictions_te[i];
}

# Once we have the predicted values, we compare them to the real ones
var matching_values_te := 0;
for{i in 1..m_te}{
    # For every matching value we add one to the total amount
    if predictions_te[i] == y_te[i] then let matching_values_te := matching_values_te + 1;
}

# We now compute the validation accuracy
var accuracy_te = (matching_values_te / m_te) * 100;
display accuracy_te;
```

Similarly to the previous program, we solve the dual model and then compute both accuracies for the training and testing dataset. However, this time we don't count with w and γ, we only have the values for λ. So the only difference remains on having to previously calculate w and γ to then be able to apply the same process.

To validate the presented model with a specific dataset, we would need to check that when executing both codes for a same dataset, the results of training and testing accuracies for the primal and dual formulation do respectively match, which we will study for both datasets right now.

Analysis of the obtained results for each dataset

When calculating the accuracies for *dataset1* we can see how the primal and dual solutions coincide, validating the model. The accuracy found is 93% for the training dataset and 88% for the testing dataset. These results seem to be reasonable. Also, the outputs for w and γ are the following:

```
w
2.11816
3.64411
2.06454
2.42442

gamma = -5.15359
```

The amount of values of λ is too big to be displayed here, but in a general sense, we can see that almost half of them are equal to 2, coinciding with the value of c.

Again, the model gets validated by this second dataset, coinciding both accuracies for both of the dual solutions. However, *dataset2*, which has a significantly larger amount of data, turns out to be harder to predict with a 77% accuracy for the training set and 81% for the testing one. This time, these are the values for w and γ.

```
      w
 0.0873381
 0.0233228
-0.00437026
-0.00545071
-0.000253404
 0.0643663
 1.28381
 0.000759762
```

```
gamma = -6.07384
```

Again, values for λ are excessive to be printed in this document, but we can see how more than half of them are equal to $c = 2$.

3. DUAL MODEL IMPLEMENTATION WITH RBF KERNEL

Parameters and datasets

In this section we will study an alternative implementation of the dual problem, using an RBF or Gaussian kernel instead of the linear one. By definition, given two points $x, y \in \mathbb{R}^n$, the RBF kernel is computed as:

$$K(x, y) = e^{-\frac{||x-y||^2}{2\sigma^2}}$$

where σ is a free parameter. However, for our implementation we won't use sigma, but gamma. Gamma is equal to $\frac{1}{n}$, where n is our n, the number of features. Alternatively, gamma is also equal to $\frac{1}{2\sigma^2}$.

Similarly to the dual one shown above, our implementation of the RBF kernel model is as follows:

```
#parameters
param m;
param n;

param c;

param y{1..m};
param x{1..m,1..n};

#variables
var lambda{1..m} >= 0, <= c;
#gaussian kernel is used: Kij -> exp(-(1/(2*(sigma)^2))*(sum{k in i..n}(x[i,k]-x[j,k])^2))
maximize objf3 : sum{i in 1..m} lambda[i] - 0.5*
            sum{i in 1..m,j in 1..m} lambda[i]*y[i]*lambda[j]*y[j]*
            exp(-1/n*(sum{k in 1..n}(x[i,k]-x[j,k])^2));


#constraints
subject to c3 : (sum{i in {1..m}}lambda[i]*y[i]) = 0;
```

This model will be applied to a linearly non-separable dataset, which will be generated through the function *sklearn.datasets.make_swiss_roll(m)* from the python *sklearn* library, where m is the number of instances.

For us, in this dataset m will be 350, using 300 on the training one and 50 on the testing one, and n will be 3 so the points generated will be $\in \mathbb{R}^3$. The Python code that gives us that dataset (twice, one for training and one for testing) directly in *.dat* format will be included in the zip containing all the coding stuff, but the main part of the code is the following:

```
m = 350
n = 3

# same for "data3_te.dat"
f = open("data3_tr.dat", "w", encoding='utf-8')
data = sklearn.datasets.make_swiss_roll(m)
mean = numpy.mean(data[1])
classes = [1 for i in range(m)]
y = []

# same for "data3_te.dat but
# range(0,m-50) -> range(300,m)"
for i in range(0,m-50):
    f.write(str(i+1))

    for j in range(0,n):
        f.write("  ")
        f.write(str(data[0][i][j]))
    f.write('\n')
f.write(";")
f.write('\n')

for j in range (0, m-50):
    f.write(str(j+1))
    f.write("  ")
    f.write(str(y[j]))
    f.write("\n")
```

As it can be seen, the values are classified in function of the mean of the vector t.

Compute of accuracies and analysis of the results

For this new implementation of the dual formulation, in order to compute the training and testing accuracies we will have to find a proper value for gamma, and then follow a similar process to the previous codes. This time we have to take into account that, because we don't know the transformation function φ and so the separation hyperplane, w and γ are found like the following:

$$\phi(x)^T w + \gamma = 0$$

$$w = \sum_{i=1}^{m} \lambda_i y_i K(x_i, x)$$

$$\gamma = \frac{1}{y_i} - \sum_{i=1}^{m} \lambda_j y_j K(x_i, x_j)$$

```
# ---DUAL PROBLEM : TRAINING SET---
reset;
model svm_dual_rbfkernel.mod;

data data3_tr.dat; # linearly nonseparable training dataset

option solver gurobi;

solve;
display lambda;

# ---FINDING NECESSARY VARIABLES---

# In order to compute the training and test accuracies we find 'gamma'
param gamma;
for {i in 1..m} {
    if  (lambda[i] > 0.01 and lambda[i] < c*0.1) then {
        # A support vector point was found
        let gamma := (y[i]- (sum{j in 1..m} lambda[j]*y[j]*
        exp(-1/n*(sum{k in 1..n}(x[i,k]-x[j,k])^2 ) ) ) ) ;
        break;
    }
}

display gamma;

# ---TRAINING ACCURACY---

# We calculate the predicted values for the training dataset
param predictions_tr{i in 1..m};
for{i in 1..m}{
    if ((sum{j in 1..m} lambda[j]*y[j]*exp(-1/n*(sum{k in 1..n}(x[i,k]-x[j,k])^2))) + gamma) > 0
    then let predictions_tr[i] := 1;
    else let predictions_tr[i] := -1;
    #display predictions_tr[i];
}

# Once we have the predicted values, we compare them to the real ones
var matching_values_tr := 0;
for{i in 1..m}{
    # For every matching value we add one to the total amount
    if predictions_tr[i] == y[i] then let matching_values_tr := matching_values_tr + 1;
}
```

```
# We now compute the training accuracy percentage
var accuracy_tr = (matching_values_tr / m) * 100;
display accuracy_tr;

# ---ACCURACY VALIDATION---

# Definition of the new parameters
param m_te >= 1, integer;
param y_te{i in 1..m};
param x_te{1..m,1..n};
data data3_te.dat; # testing dataset

# We calculate the predicted values
param predictions_te{i in 1..m_te};
for{i in 1..m_te}{
    if ((sum {j in {1..m}} lambda[j] * y[j] *
    exp(-(1/n)*(sum {k in 1..n} (x_te[i,k] - x[j, k])^2))) + gamma) > 0
    then let predictions_te[i] := 1;
    else let predictions_te[i] := -1;
    #display predictions_te[i];
}

# Once we have the predicted values, we compare them to the real ones
var matching_values_te := 0;
for{i in 1..m_te}{
    # For every matching value we add one to the total amount
    if predictions_te[i] == y_te[i] then let matching_values_te := matching_values_te + 1;
}

# We now compute the validation accuracy
var accuracy_te = (matching_values_te / m_te) * 100;
display accuracy_te;
```

```
gamma = 0.115279
```

When analyzing the results of this execution for the new linearly nonseparable dataset we find a training accuracy of 100% and a testing accuracy of 98%, being this the most precise of all the implementations studied. We can also appreciate how this time, the values for λ are closer to 0 than our chosen c.