OTDM: Support vector machine

Gabriel Zarate, Ximena Moure

November 2023

Contents

1 Introduction

The aim of this project is to develop a Support Vector Machine (SVM) in both its primal and dual quadratic formulations, utilizing AMPL. Our goal is to validate the performance of these SVM models across various datasets, ensuring consistency in the results. A key part of this validation process will be to confirm that the separating hyperplane derived from the dual formulation aligns accurately with the one obtained from the primal model.

2 Primal formulation

The primal optimization problem can be formulated as follows:

$$\begin{aligned} \underset{(w,\gamma,s)\in R^{N+1+m}}{\text{minimize}} \quad & \frac{1}{2}w^T w + \nu e^T s \\ \text{subject to} \quad & -Y(Aw + \gamma e) - s + \epsilon \leq 0, \\ & -s \leq 0. \end{aligned} \tag{1}$$

In order to implement it in AMPL we used the scalar form:

$$\begin{aligned} \underset{(w,\gamma,s)\in R^{n+m+1}}{\text{minimize}} \quad & \frac{1}{2}\sum_{j=1}^{n} w_j^2 + \nu \sum_{i=1}^{m} s_i \\ \text{subject to} \quad & -Y_i \left(\sum_{j=1}^{n} A_{ij} w_j + \gamma \right) - s_i + 1 \leq 0, \quad \forall i \in \{1, \ldots, m\}, \\ & -s_i \leq 0, \quad \forall i \in \{1, \ldots, m\}. \end{aligned} \tag{2}$$

3 Dual formulation

The dual optimization problem can be formulated as follows:

$$\begin{aligned} & \max_{\lambda} \quad \lambda^T e - \frac{1}{2}\lambda^T Y A \lambda Y^T A^T \\ & \text{subject to:} \\ & \quad \lambda^T Y e = 0 \\ & \quad 0 \leq \lambda \leq \nu \end{aligned} \tag{3}$$

As mentioned before, to solve it in AMPL we need to use the scalar form:

$$\begin{aligned} & \max_{\lambda} \quad \sum_{i=1}^{m} \lambda_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \lambda_i \lambda_j y_i y_j K_{ij} \\ & \text{subject to:} \\ & \quad \sum_{i=1}^{m} \lambda_i y_i = 0 \\ & \quad 0 \leq \lambda \leq \nu \end{aligned} \tag{4}$$

where K_{ij} is the Kernel matrix that is equal to AA^T.

4 Implementation over generated dataset

4.1 Generating the dataset

Two datasets were generated, one consisting of 8000 rows for training, and one consisting of 2000 rows for testing. Both datasets were generated using a seed value of 1234. The generation of these datasets was accomplished using the following commands:

```
./gensvmdat train_data.txt 8000 1234
./gensvmdat test_data.txt 2000 1234
```

4.2 Primal

In order to implement it in AMPL the following files were created:

- primal.mod: implements the primal as an optimization problem. See 1
- dataset.dat: it reads the train and test files and loads them, also it sets the values for the parameters like m (number of rows), n (number of columns) and nu (tradeoff). See 2
- primal.run: is a script that executes the model over the generated dataset, it sets cplex as the solver and after solving it calculates the train and test accuracy using the calculated variables (w and gamma). See 3

To calculate the accuracy, it is essential to determine the sign of the decision function, which is expressed as follows:

$$y_i = \begin{cases} 1 & \text{if } \sum_{j=1}^{n} A_{ij} w_j + \gamma \geq 0, \\ -1 & \text{otherwise.} \end{cases} \tag{5}$$

```
param m_train;
param m_test;
param n;

param nu;

param y_train {1..m_train};
param A_train {1..m_train, 1..n};

param y_test {1..m_test};
param A_test {1..m_test, 1..n};


# Variables

var w {1..n};
var gamma;
var s {1..m_train};

# Problem

minimize primal:
    0.5 * sum{j in 1..n}(w[j]^2)
    + nu * sum{i in 1..m_train}(s[i]);
```

```

subject to primal_c1 {i in 1..m_train}:
    -y_train[i]*sum{j in 1..n}(A_train[i,j]*w[j] + gamma) -s[i] + 1 <= 0;

subject to primal_c2 {i in 1..m_train}:
    -s[i] <= 0;
```

Listing 1: AMPL code for primal.mod

```

param m_train := 8000;
param m_test := 2000;
param n := 4;

param nu := 30;

read {i in 1..m_train}(
A_train[i,1], A_train[i,2], A_train[i,3], A_train[i,4], y_train[i]
) < data/train_data.txt;

read {i in 1..m_test}(
A_test[i,1], A_test[i,2], A_test[i,3], A_test[i,4], y_test[i]
) < data/test_data.txt;
```

Listing 2: Code for dataset.dat

```
reset;

option solver cplex;

model primal.mod

data ./data/dataset.dat;

solve;

display w, gamma;

print "Train accuracy";

param y_pred_tr {1..m_train};
let {i in {1..m_train}} y_pred_tr[i] :=
    if sum{j in {1..n}}(A_train[i,j]*w[j] + gamma) > 0 then 1 else -1;

param correct_tr = sum{i in {1..m_train}}(
    if y_pred_tr[i] = y_train[i] then 1 else 0
);

param accuracy_tr = correct_tr / m_train;

display accuracy_tr;

print "Test accuracy";

param y_pred_te {1..m_test};
let {i in {1..m_test}} y_pred_te[i] :=
    if sum{j in {1..n}}(A_test[i,j]*w[j] + gamma) > 0 then 1 else -1;
```

```

param correct_te = sum{i in {1..m_test}}(
    if y_pred_te[i] = y_test[i] then 1 else 0
);

param accuracy_te = correct_te / m_test;

display accuracy_te;
```

Listing 3: Code for primal.run

To execute the code the following command was executed on AMPL IDE:

```
include primal.run;
```

4.3 Dual

In order to implement it in AMPL the following files were created:

- dual.mod: implements the dual as an optimization problem. See 4
- dual.run: See 5. It is a script that executes the model over the generated dataset (the same dataset created for the primal), it sets cplex as the solver and it solves the problem. From this we get λ, but we need to compute the hyperplane. We know that this is possible by using the following equations:

$$w_j = \sum_{i=1}^{m} \lambda_i y_i A_{ij}, \qquad j \in \{1, \dots, n\}$$

$$\gamma = \frac{1}{n}\left(y_i - \sum_{j=1}^{n} w_j A_{ij} \right), \qquad j \in \{1, \dots, n\},\ i \in SV$$

  where SV is the set of support vectors (i.e., $s_i = 0, \lambda_i > 0$).
  Once w and γ are computed we can calculate the train and test accuracy.

```
param m_train;
param m_test;
param n;

param nu;

param y_train {1..m_train};
param A_train {1..m_train, 1..n};

param y_test {1..m_test};
param A_test {1..m_test, 1..n};

var lambda {1..m_train} >=0 ,  <=nu;  # lagrange multipliers

maximize svm_dual:
        sum{i in 1..m_train} lambda[i]
        -1/2*sum{i in 1..m_train, j in 1..m_train}lambda[i]
        *y_train[i]*lambda[j]*y_train[j]
        *(sum{k in 1..n}A_train[i,k]*A_train[j,k]);
```

```

subject to dual_constraint:
        sum{i in 1..m_train}lambda[i]*y_train[i] =0;
```

Listing 4: AMPL code for dual.mod

```
reset;

print "SOLVING DUAL WITH GENERATED DATASET";

option solver cplex;

model dual.mod;

data ./data/dataset.dat;

solve;

display lambda;

display nu;

 # Define a tolerance level for determining support vectors
param tol_accuracy := 1e-6;


# Calculate the weights w[j]
param w {1..n};
let {j in 1..n} w[j] := sum {i in 1..m_train} lambda[i] * y_train[i] *
    A_train[i,j];

display w;

# Variable to store the value of gamma
var gamma_val;

# Loop over the training set to find a support vector and calculate gamma
for {i in 1..m_train} {
    if lambda[i] > tol_accuracy and lambda[i] < nu - tol_accuracy then {
        let gamma_val := y_train[i] - sum {j in 1..n} w[j] * A_train[i,j];
    }
}

# Divide by the number of features, if gamma has been set
let gamma_val := gamma_val / n;

display gamma_val;

# Predict and calculate accuracy on the training set
param y_pred_tr {1..m_train};
let {i in 1..m_train} y_pred_tr[i] :=
    if sum {j in 1..n} (A_train[i,j] * w[j] + gamma_val) > 0 then 1 else -1;

param correct_tr := sum {i in 1..m_train} (
    if y_pred_tr[i] = y_train[i] then 1 else 0
);

```

```
param accuracy_tr := correct_tr / m_train;

display accuracy_tr;

# Predict and calculate accuracy on the test set
param y_pred_te {1..m_test};
let {i in 1..m_test} y_pred_te[i] :=
    if sum {j in 1..n} (A_test[i,j] * w[j] + gamma_val) > 0 then 1 else -1;

param correct_te := sum {i in 1..m_test} (
    if y_pred_te[i] = y_test[i] then 1 else 0
);

param accuracy_te := correct_te / m_test;

display accuracy_te;
```

Listing 5: AMPL code for dual.mod

4.4 Results

4.4.1 Primal

The results for the primal are the following:

```
CPLEX 22.1.1.0: optimal solution; objective 72426.20237
26 separable QP barrier iterations
No basis.
w [*] :=
1  5.1156
2  4.98397
3  5.04391
4  5.00287
;

gamma = -2.52409

Train accuracy
accuracy_tr = 0.94175

Test accuracy
accuracy_te = 0.933
```

4.4.2 Dual

The results for the dual are the following:

```
CPLEX 22.1.1.0: optimal solution; objective 72426.20236
30 QP barrier iterations
No basis.
nu = 30

w [*] :=
1  5.1156
```

```
2  4.98397
3  5.04391
4  5.00287
;

gamma_val = -2.52409

accuracy_tr = 0.94175

accuracy_te = 0.933
```

4.4.3 Comparison

It can be seen that the dual and primal gave the same results, the values of w and γ are the same in both cases.
It is worth mentioning that in order to achieve the same results we tried different values for the tolerance used to mitigate numerical errors. We found that the tolerance value that yields the same result as the primal is the value 10^{-6}.

5 Application to other dataset

5.1 Implementation

The selected dataset to test the implemented code was the Iris dataset, that can be found on the following link Iris dataset, or it can also be directly imported to Python, which it was done.

The dataset consists on a group of instances that describe a type of plant and each entry has 4 numerical features:

- sepal length
- sepal width
- petal length
- petal width

And its target variable, which has 3 levels are:

- Iris Setosa
- Iris Versicolour
- Iris Virginica

To convert this dataset into a suitable input por AMPL and into a suitable problem for our implemented code, the dataset had to be transformed into a binary problem and to do so, dataGen.py (see 9 was implemented. The new decided problem was to classify if the plant is Iris Virginica or not.

There first the dataset was imported, then the target variable values were converted into 1 and -1 (if it is Iris Virginica or not respectively). With that, the dataset was split into train and test (80-20) using stratify to get the target balanced on each split. And finally these were exported into train_iris.txt and test_iris.txt.

To execute the AMPL implemented code with this dataset it was needed to create new .dat and .run files:

1. dataset.dat: where the only changes where over the m sizes and the filenames where the data is imported, this because this dataset n was the same as the generated one previously. See 6.
2. primal_iris.run: the only change was the data command parameter, changing the .dat file. See 7.
3. dual_iris.run: the only change was the data command parameter, changing the .dat file.

```
param m_train := 120;
param m_test := 30;
param n := 4;

param nu := 1;

read {i in 1..m_train}(
A_train[i,1], A_train[i,2], A_train[i,3], A_train[i,4], y_train[i]
) < iris_data/train_iris.txt;

read {i in 1..m_test}(
A_test[i,1], A_test[i,2], A_test[i,3], A_test[i,4], y_test[i]
) < iris_data/test_iris.txt;
```

Listing 6: Code for dataset.run

```
reset;

option solver cplex;

model primal.mod

data ./iris_data/dataset.dat;

solve;

display w, gamma;

print "Train accuracy";

param y_pred_tr {1..m_train};
let {i in {1..m_train}} y_pred_tr[i] :=
    if sum{j in {1..n}}(A_train[i,j]*w[j] + gamma) > 0 then 1 else -1;

param correct_tr = sum{i in {1..m_train}}(
    if y_pred_tr[i] = y_train[i] then 1 else 0
);

param accuracy_tr = correct_tr / m_train;

display accuracy_tr;

print "Test accuracy";

```

```
param y_pred_te {1..m_test};
let {i in {1..m_test}} y_pred_te[i] :=
    if sum{j in {1..n}}(A_test[i,j]*w[j] + gamma) > 0 then 1 else -1;

param correct_te = sum{i in {1..m_test}}(
    if y_pred_te[i] = y_test[i] then 1 else 0
);

param accuracy_te = correct_te / m_test;

display accuracy_te;
```

Listing 7: Code for primal_iris.run

```
reset;

print "SOLVING DUAL WITH GENERATED DATASET";

option solver cplex;

model dual.mod;

data ./iris_data/dataset.dat;

solve;

display lambda;

display nu;

param tol_accuracy := 1e-6; # Define a tolerance level for determining
    support vectors


# Calculate the weights w[j]
param w {1..n};
let {j in 1..n} w[j] := sum {i in 1..m_train} lambda[i] * y_train[i] *
    A_train[i,j];

display w;

# Variable to store the value of gamma
var gamma_val;

# Loop over the training set to find a support vector and calculate gamma
for {i in 1..m_train} {
    if lambda[i] > tol_accuracy and lambda[i] < nu - tol_accuracy then {
        let gamma_val := y_train[i] - sum {j in 1..n} w[j] * A_train[i,j];
    }
}

# Divide by the number of features, if gamma has been set
let gamma_val := gamma_val / n;

display gamma_val;

# Predict and calculate accuracy on the training set
```

```
param y_pred_tr {1..m_train};
let {i in 1..m_train} y_pred_tr[i] :=
    if sum {j in 1..n} (A_train[i,j] * w[j] + gamma_val) > 0 then 1 else -1;

param correct_tr := sum {i in 1..m_train} (
    if y_pred_tr[i] = y_train[i] then 1 else 0
);

param accuracy_tr := correct_tr / m_train;

display accuracy_tr;

# Predict and calculate accuracy on the test set
param y_pred_te {1..m_test};
let {i in 1..m_test} y_pred_te[i] :=
    if sum {j in 1..n} (A_test[i,j] * w[j] + gamma_val) > 0 then 1 else -1;

param correct_te := sum {i in 1..m_test} (
    if y_pred_te[i] = y_test[i] then 1 else 0
);

param accuracy_te := correct_te / m_test;

display accuracy_te;
```

Listing 8: Code for dual_iris.run

Listing 9: Code for dataGen.py

```
from ucimlrepo import fetch_ucirepo
from sklearn.model_selection import train_test_split
import pandas as pd

iris = fetch_ucirepo(id=53)

X = iris.data.features
y = iris.data.targets

class_counts = y.value_counts()

y_binary = (y == 'Iris-virginica').astype(float) * 2 - 1
X_train, X_test, y_train, y_test = train_test_split(X, y_binary,
    test_size=0.2, random_state=42, stratify=y_binary)

train_data = pd.concat([X_train, y_train], axis=1)
train_data.to_csv('train_iris.txt', sep=' ', header=False, index=False)

# Export test data to test_iris.txt
test_data = pd.concat([X_test, y_test], axis=1)
test_data.to_csv('test_iris.txt', sep=' ', header=False, index=False)
```

5.2 Results

5.2.1 Primal

The results obtained for the primal over the iris datset are the following:

```
CPLEX 22.1.1.0: optimal solution; objective 13.51679642
15 separable QP barrier iterations
No basis.
w [*] :=
1  -0.687044
2  -0.937784
3   2.12455
4   1.56317
;

gamma = -1.51585

Train accuracy
accuracy_tr = 0.983333

Test accuracy
accuracy_te = 1
```

5.2.2 Dual

The results obtained for the dual over the iris datset are the following:

```
CPLEX 22.1.1.0: optimal solution; objective 13.51679636
17 QP barrier iterations
No basis.
nu = 1

w [*] :=
1  -0.687044
2  -0.937784
3   2.12455
4   1.56317
;

gamma_val = -1.51585

accuracy_tr = 0.983333

accuracy_te = 1
```

5.2.3 Comparison

The dual and primal gave the same results, the values of w and γ are the same in both cases.
It is worth mentioning that we tried with different values for v. We could also have done a cross validation in order to get the best value, but since we achieve good results with just trying a few values we did not consider that doing cross validation was necessary.

6 Linearly non-separable dataset

6.1 Implementation

We wanted to analyse the performance of the classifier in a linearly non-separable dataset. Thus, we use the Swiss roll synthetic dataset.

In order to convert it to a binary classification problem we first establish a threshold. The threshold is set to the mean of the scalar values, providing a central value that effectively divides the dataset into two groups. Data points with scalar values below this mean are assigned a label of -1, indicating one class, while those above the mean are assigned a label of 1, indicating the other class. This binary labeling transforms our regression problem into a classification one, enabling us to apply binary classification algorithms such as Support Vector Machines (SVM).

The following is the code to generate the Swiss dataset:

Listing 10: Code for generating Swiss dataset (dataGenSwiss.py

```
from sklearn.datasets import make_swiss_roll
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
import numpy as np


def generate_data(test_size, n_samples, noise, random_state):

    X, t = make_swiss_roll(n_samples=n_samples, noise=noise,
        random_state=random_state)

    # threshold based on the mean of t
    threshold = np.mean(t)
    y = np.where(t < threshold, -1, 1)
    X_train, X_test, y_train, y_test = train_test_split(X, y,
        test_size=test_size, random_state=random_state)

    return X_train, X_test, y_train, y_test

X_train, X_test, y_train, y_test = generate_data(test_size=0.5,
    n_samples=1000, noise=0.2, random_state=42)

num_train_points = X_train.shape[0]
num_test_points = X_test.shape[0]

print(f"Number of points in the training data: {num_train_points}")
print(f"Number of points in the test data: {num_test_points}")


#Save training data
np.savetxt('train_data.txt', np.hstack((X_train, y_train.reshape(-1, 1))),
    fmt='%0.8f')

# Save testing data
np.savetxt('test_data.txt', np.hstack((X_test, y_test.reshape(-1, 1))),
    fmt='%0.8f')


clf = make_pipeline(StandardScaler(), SVC(kernel='rbf', gamma=10))
clf.fit(X_train, y_train)

# Evaluate the classifier on the test data
accuracy = clf.score(X_test, y_test)
```

```
print(f"Test set accuracy: {accuracy:.2f}")
```

Once we had the train and test split, we needed to create a .dat file to be able to use it with AMPL. See 11.

```
param m_train := 500;
param m_test := 500;
param n := 3;

param nu := 1;

read {i in 1..m_train}(
A_train[i,1], A_train[i,2], A_train[i,3], y_train[i]
) < swiss_data/train_data.txt;

read {i in 1..m_test}(
A_test[i,1], A_test[i,2], A_test[i,3], y_test[i]
) < swiss_data/test_data.txt;
```

Listing 11: Code for dataset.run of the swiss data

Since we are working with a linearly nonseparable dataset, it is more effective to use a Gaussian kernel (RBF). Thus we created a file called dualRBF.mod which implements the dual with a Gaussian kernel (see 12).

```
param m_train;
param m_test;
param n;
param nu;

param y_train {1..m_train};
param A_train {1..m_train, 1..n};

param y_test {1..m_test};
param A_test {1..m_test, 1..n};

var lambda {1..m_train} >=0 ,  <=nu;

minimize svm_dual_rbf:
        (1/2) * (sum {i in 1..m_train, j in 1..m_train} lambda[i] *
            y_train[i] * lambda[j] * y_train[j] * exp(-(1/n * (sum{k in 1..n}
            (A_train[i,k] - A_train[j,k])^2)))) - (sum {i in 1..m_train}
            lambda[i]);

subject to c1:
        sum {i in 1..m_train} lambda[i] * y_train[i] = 0;
```

Listing 12: Code for dualRBF.mod

Then we created the file dualRBF.run which computes *gamma*, *w* and the accuracy. See 13

```
reset;

print "SOLVING DUAL kernel swiss";

option solver cplex;

model dualRBF.mod;

```

```
data ./swiss_data/dataset.dat;

solve;

display lambda;

display nu;

param tol_accuracy := 1e-6;


param w {1..n};
let {j in 1..n} w[j] := sum{i in 1..m_train} lambda[i] * y_train[i] *
    A_train[i,j];
display w;

var gamma;


for {i in 1..m_train : lambda[i] > tol_accuracy && lambda[i] < nu -
    tol_accuracy} {
    let gamma := 1/y_train[i] - sum {j in 1..n} lambda[j] * y_train[j] *
        exp(-(1/n * (sum{k in 1..n} (A_train[i,k] - A_train[j,k])^2)));
    break;
}


display gamma;

param y_pred_continuous {1..m_test};
var y_pred_binary {1..m_test};


for {t in 1..m_test} {
    let y_pred_continuous[t] := sum {j in 1..m_test} lambda[j] * y_train[j] *
        exp(-(1/n * (sum{k in 1..n} (A_train[j,k] - A_test[t,k])^2))) + gamma;
}

for {t in 1..m_test} {
    let y_pred_binary[t] := if y_pred_continuous[t] <= 0 then -1 else 1;
}

param match {1..m_test};
let {t in 1..m_test} match[t] := if y_pred_binary[t] = y_test[t] then 1 else
    0;

param correct_predictions := sum {t in 1..m_test} match[t];

# Calculate and display accuracy
param accuracy := correct_predictions / m_test;
display accuracy;
```

Listing 13: Code for dualRBF.run

6.2 Results

When applying a support vector classifier with a linear kernel to the dataset, it becomes evident that this model is not suitable. We obtained an accuracy of almost 0.6.
But, when applying a support vector classifier with a gaussian kernel to the dataset an accuracy of almost 0.98 is achieved.

7 Conclusions

Drawing on the theoretical knowledge acquired in this course, we successfully programmed both the primal and dual formulations of Support Vector Machines in AMPL. We evaluated our implementations using synthetic and real-world datasets. The encouraging outcomes demonstrated the effectiveness of our models. Furthermore, the outputs confirmed the consistency between the two approaches, as they both identified the identical hyperplane.

Overall, the findings have illustrated that theoretical models are applicable in practical scenarios, highlighting the critical need for careful selection of kernels and hyperparameters tailored to the data's unique attributes.