Universitat Politècnica de Catalunya

Facultat d'Informàtica de Barcelona

Master in Data Science

Support Vector Machines

Optimization Techniques for Data Mining

Second laboratory report

Arnau Arqué and Daniel Esquina

Professor: Jordi Castro Pérez

November 27, 2022

Contents

List of Tables

Listings

1 Introduction

This project is part of the subject Optimization Techniques for Data Science of the Master in Data Science coursed in the Facultat d'Informàtica de Barcelona. The main objective of this task is to implement the primal and dual quadratic formulation of the Support Vector Machine (SVM) in AMPL. This modelling language, created for optimization purposes, will be very useful to tackle this task. It allows the formulation of complex models using simple syntax, close to mathematical formulation. Once we have this implementation, we will validate it with some generated data and experiment with an additional dataset. All these steps will be included in the following sections of the report.

In order to implement a SVM, we will need some theoretical background to support our coding decisions. The main idea of a SVM is to find a hyperplane that splits the data in two different classes. Thus making this algorithm a binary classification one. This data split can be done either in the original space or in a commonly higher dimensionality one called feature space. The goal of the SVM is then finding the hyperplane that maximizes its geometric margin over the data. As we have seen in theory class, finding the biggest margin is the same as finding the optimal hyperplane. This search requires us to solve an optimization problem, hence the relationship with this subject. Depending on the used constraints for the SVM, it can be considered a Hard-Margin or a a Soft-Margin SVM. In our case we will use a Soft-Margin SVM.

We have seen two formulations of this model with different objective functions that we are asked to implement for this project. The first one is the **primal**:

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

$$\text{s.t.} \quad y_i(w^\intercal x_i + \gamma) + s_i \geq 1 \quad i = 1, ..., n \tag{2}$$

$$s_i \geq 0 \quad i = 1, ..., n \tag{3}$$

Where n are the dimensions of the data, m the number of instances, w is the normal to the separation hyperplane, and γ its location with respect to the origin. x_i are the data instances and y_i its corresponding labels (-1,1). s_i are the slacks which determine the degree to which the constraint can be violated. In other words, the amount that each data point can be within the margins. The ν parameters weights the trade-off between maximizing the margin and minimizing the training error.

The equation which then yields the correct classification of the data for the primal formulation is:

$$\begin{cases} 1 & \text{if} \quad w^\intercal x + b \geq 1 \\ -1 & \text{otherwise} \end{cases} \tag{4}$$

Where we can see that computing the scalar product is expensive if the number of dimensions d is large.

Now for the **dual** formulation of the Support Vector Machine we have the following function:

$$\max_{\alpha \in \mathbb{R}^n} \quad \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j x_i^\intercal x_j \tag{5}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{6}$$

$$0 \leq \alpha_i \leq \nu \quad i = 1, ..., n \tag{7}$$

Where the only not previously defined parameter are the α which are the Lagrange multipliers. Notice that the inner product $x_i^\intercal x_j$ of the objective function 5 can be exchanged for the resulting value of applying a kernel function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, where $\mathcal{X}$ is the input data space, to the input data $k(x_i, x_j)$.

Departing from the primal classification formula, we can derive an equivalent one to classify data in terms of α.

$$w^\intercal x + b = \left(\sum_{i=1}^{m} \alpha_i y_i x_i \right)^\intercal x + b = \sum_{i=1}^{m} \alpha_i y_i x_i^\intercal x + b \tag{8}$$

The equation which then yields the correct classification of the data for the dual formulation is:

$$\begin{cases} 1 & \sum_{i=1}^{m} \alpha_i y_i x_i^\intercal x + b \geq 1 \\ -1 & \text{otherwise} \end{cases} \tag{9}$$

In this equation it is easy to see that an efficient calculation can take place if there are only a few support vectors, since we don't need to compute the inner products $x_i^\intercal x$ for those $\alpha_i = 0$.

2 Implementation

In this section we will comment on the process we have carried out to implement the procedures and codes associated with this project. We will begin by analyzing the implementation of the primal and dual SVM formulations in AMPL. Next, we will analyze the code generated for the experimentation stage. You can find the code files associated with this project in the `./code/` directory.

2.1 Primal and dual formulations

We will start by dealing with the primal formulation. As we commented in section 1, the primal formulation of an SVM represents, ultimately, the optimization problem that we defined in the equation 1. In order to carry out the implementation in AMPL we have generated the codes that you can find in the directory `./code/primal/`. You can also see the implementation of the primal model in the 1 listing.

```
1  # Parameters declaration
2  param d; # Dimension of input space
3  param n; # Number of instances
4  param x {1..n, 1..d}; # Input data
5  param y {1..n}; # Labels
6  param nu;
7
8  # Variables declaration
```

```
var gamma;
var w {1..d};
var s {1..n};

# Objective function
minimize objective_function:
    0.5 * sum {i in 1..d} w[i]^2 + nu * sum {i in 1..n} s[i];

# Constraints
subject to constraint_1 {i in 1..n}:
    y[i]*(sum {j in 1..d} (w[j]*x[i,j]) + gamma) + s[i] >= 1;

subject to constraint_2 {i in 1..n}:
    s[i] >= 0;
```

Listing 1: Implementation of the primal formulation.

As you can see, the code starts by declaring the parameters needed to perform the optimization, namely the dimensionality d of the input space $\mathcal{X} = \mathbb{R}^d$, the number of data n, the data x, along with the labels of the instances y and the hyperparameter ν. Apart from that, we have also declared the variables $\gamma \in \mathbb{R}$, $w \in \mathcal{X}$ and $s \in \mathbb{R}^n$, whose value will be determined during the process of optimization. Next, we have defined the objective function following what is established by the primal formulation 1. To finish, we have defined two constraints `constraint_1` and `constraint_2` that establish the conditions established in the equations 2 and 3, respectively.

```
# Parameters declaration
param d; # Dimension of input space
param n; # Number of instances
param x {1..n, 1..d}; # Input data
param y {1..n}; # Labels
param nu;

# Variables declaration
var alpha {1..n};

# Objective function
maximize objective_function:
    (sum {i in 1..n} alpha[i]) -
    0.5 * sum {i in 1..n, j in 1..n} alpha[i]*alpha[j]*y[i]*y[j]*
        sum {k in 1..d} x[i,k]*x[j,k];

# Constraints
subject to constraint_1:
    sum {i in 1..n} alpha[i]*y[i] = 0;

subject to constraint_2 {i in 1..n}:
    alpha[i] >= 0;

subject to constraint_3 {i in 1..n}:
    alpha[i] <= nu;
```

Listing 2: Implementation of the dual formulation.

Regarding the dual formulation, you can find its code in the `./code/dual/` directory and, more specifically, you can see the code associated with the dual model in the 2 listing. As you can see, we start the procedure by defining the necessary parameters to perform the optimization, namely, the dimension d of the input space $\mathcal{X} = \mathbb{R}^d$, the number of data n, the input data x together with the respective labels y and finally the hyperparameter nu. Unlike the primal model, we now only need one variable $\alpha \in \mathbb{R}^n$ that ultimately represents the values to be optimized.

Once the parameters and variables have been established, we define the objective function following the formulation established in the equation 5. Next, we define the constraint `constraint_1` associated with the constraint established in the equation 6. It is not difficult to realize that the constraint established in the equation 7 is, in short, two constraints $0 \leq \alpha_i$ and $\alpha_i \leq \nu$. For this reason we have defined the constraints `constraint_2` and `constraint_3` of the model.

In addition to the files with the implementation of the models we have also generated a `./code/data/data.dat` file to read the data. This script reads the data for training (located in file `auxtrain.txt`) and testing (located in file `auxtest.txt`) from the directory `./data/`. The format that the data must have in the case of training data is a header line that contains only two values n and d separated by a white space that represent the number of instances of the dataset and the dimensionality of the data respectively. Next, there must be n lines and, in each, $d+1$ values that correspond to the features of the instance in question. The value $d+1$ of the line will indicate the label $\{-1, 1\}$. Once the training data has been read, the script proceeds to read the testing data. The procedure is identical to that of the training data. However, in the first line it will only be necessary to indicate the number of testing data, since it is assumed that the dimensionality of the data has not changed from that of the training data.

The files that run the model and compute the relevant metrics are `./code/primal/primal.run` and `./code/dual/dual.run` for the primal and dual models, respectively.

2.2 Python script

At this point, we have already presented the implementation in AMPL of the primal and dual SVM models. However, we thought it would be convenient to generate a script that allows us to carry out executions with different parameterizations and datasets without the need to use the AMPL terminal. So, we generated the `run-svm.py` script located in the `./code/` directory of the deliverable. In this section we will comment, analyze and justify the main functionalities we have implemented.

The first functionality we have implemented are different methods for preprocessing the data that we will use during the experimentation stage. You can find the implementation of these methods in the `Preprocessing methods` section of `run-svm.py`. These functions format the training datasets (the first line only contains `n d` where `n` is the number of training data and `d` its dimensionality and then there is `n` lines with `d+1` values `ft-1, ft-2, ..., ft-d, label` that represent the features of each instance and its label. Finally, the preprocessed data is saved in the files `auxtrain.txt` and `auxtest.txt` which will use `data.dat` (see listing 5) to read the data

Note that the current implementation of SVM only allows working with numerical values. In case the dataset contains categorical variables, it converts them to numeric using the One-Hot Encoding technique. In future implementations of the algorithm it would be possible to work with variables of multiple types using kernel functions.

As we discussed in previous sections, the algorithm consists of a hyperparameter ν on which the results obtained with our implementation of the SVM depend. A good practice is to use Cross-Validation (CV) using only the training data to estimate its value and then use the value of ν that gave us the best accuracy to run the algorithm using training data for training and testing data for validation.

We have implemented the aforementioned procedure in the `cv` method of the script. This method mainly receives three parameters k, *start* and *end*. k determines the number of iterations to be carried out in the CV process for each value of ν to be studied. In each iteration, n/k data from the training dataset will be used as testing data, while the rest will be used as training data. The values of ν that we will study will be $\nu = \{2^i \mid i \in [start,\ end]\}$. For each value of ν, the arithmetic mean of the testing accuracies is calculated and, finally, the ν that has obtained a higher accuracy is selected.

The main function of the program is found in the `Main program` section of the script. First, the required arguments are parsed[1]. The `--data` parameter allows us to choose between `{synthetic,sonar}` to set the dataset that the SVM will use in the run. The parameter `--type` allows us to choose between the algorithm that will be used `{primal,dual}`. Finally, the arguments `--nu` and `--cv k start end` allow defining a specific ν value or determining the parameters from which the value of ν will be computed through the CV process stated in previous paragraphs[2].

Once the parameters have been checked and all the data is valid, the preprocessing of this data is carried out. Next, the environment variable `NU` is set with the command `export NU=nu_value`[3] (since that is the manner in which the data file can later read its value) and run the SVM model. Finally, the results are written to standard output, along with the total execution time of the program.

3 Experimentation

At this point we have just commented on the implementation of both formulations of the Support Vector Machine but this was not the only task for this deliverable. Now we will focus on the experimentation part of the project. We will not only use the given data generator but also an additional dataset. We will validate the data set using training and test data splits and even include a cross-validation technique. The results of this section of the project can be found in the `./code/results` folder under the main directory.

3.1 Generated dataset

On the previous section 2.1 we have presented how data is loaded in order to run the created models with AMPL. We have used the given generator to create a training and testing datasets in order to validate our models. The seeds used to generate these datasets are `29121998` and `18091998` respectively. This execution will compute a traing and testing accuracies as a fraction of the number of samples correctly classified divided by the total number of samples. The results found for this execution that can be inspected in files `synthetic-primal` and `synthetic-dual`. All the support vectors and the classification value for each sample can be seen in the `output-synthetic-primal` and `output-synthetic-dual`.

Let's start with the primal model. With a value of $\nu = 30$ we get the following hyperplane with the extracted values of w and γ:

$$\pi_{primal} : 4.05048x_1 + 4.27696x_2 + 5.15461x_3 + 5.26113x_4 - 9.26190 = 0 \tag{10}$$

[1] Run `python3 run-svm.py -h` to see the required arguments.

[2] Note that it is necessary to either establish `--nu` or `--cv`. If both are present, the value computed through the CV process will prevail.

[3] Note that it is necessary to run the script from a bash shell

The obtained training accuracy is 0.94% and the testing accuracy 0.90%, it seems reasonable to assume our models are behaving correctly.

Now for the output of the dual execution, with the same value of $\nu = 30$ we get the hyperplane:

$$\pi_{dual} : 4.05048x_1 + 4.27696x_2 + 5.15461x_3 + 5.26113x_4 - 9.26190 = 0 \tag{11}$$

The training and testing accuracies are evidently the same. We can safely assume so because as it can be seen in equations 10 and 11, the obtained separation hyperplanes coincide.

3.2 Sonar dataset

As we were interested in further testing our implementation we decided to use an additional dataset with real world data. The dataset we have chosen is named Connectionist Bench an it can be found in the UCI Machine Learning repository [4]. The proposed task is to discriminate between sonal signals bounced off metal cylinder and those bounced off a roughly cylindrical rock. This signals are recorded and transformed into a set of sixty normalized numbers that will be used as features to train our model. We found this binary balanced classification task very interesting as it has a significant difference with the generated one in terms of data's dimensionality.

There was a minimal amount of preprocessing concerning this dataset that we have commented on the previous section 2.2. As we have also commented on that section, we have implemented a 10 fold cross-validation to tune the ν hyperparameter of our models. We decided to do this as it feels more appropriate for a real world solution. The results of that cross-validation for the said hyperparameter can be seen in table 1 and in the files `sonar-primal-cv` and `sonar-dual-cv` from the results directory.

Table 1: 10-fold CV to estimate ν hyperparameter.

ν	2^{-5}	2^{-4}	2^{-3}	2^{-2}	2^{-1}	2^{0}	2^{1}	2^{2}	2^{3}	2^{4}	2^{5}
Accuracy (primal)	0.60	0.68	0.71	0.74	0.77	0.77	0.76	0.76	0.78	0.76	0.75
Accuracy (dual)	0.55	0.68	0.71	0.74	0.77	0.77	0.76	0.76	0.78	0.76	0.75

We can see how the best accuracy can be found when $\nu = 8$ in both cases, so we will use that to retrieve the results of our SVM implementations. These results can be seen in the aforementioned files and the raw results of the AMPL implementation in the `output-sonar-primal` and `output-sonar-dual` files of the same repository. In these files, additional information about the classification of each data point can be checked. After running our code we get a training accuracy of 88.0% and a testing accuracy of 81.0%, evidently in both primal and dual versions. These results seem rather promising as the baseline performance for this task was around 53.3%. We have also computed the hyperplane, just like we did for the previous generated dataset, as prompted by the statement, and it can be seen followingly:

[4] `https://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+(Sonar,+Mines+vs.+Rocks)`

$$\pi : -1.99065x_1 - 0.99145x_2 - 0.62084x_3 - 2.66838x_4 - 2.58531x_5 + 0.01345x_6 + 3.67006x_7$$
$$+2.79315x_8 - 3.72650x_9 - 2.02520x_{10} - 3.04797x_{11} - 1.87592x_{12} - 1.28589x_{13} + 1.00855x_{14}$$
$$-0.78239x_{15} + 1.62846x_{16} + 1.27242x_{17} - 0.50752x_{18} - 0.97678x_{19} - 0.70165x_{20} + 1.10769x_{21}$$
$$-1.72017x_{22} - 0.65290x_{23} - 1.02061x_{24} + 0.58838x_{25} + 0.07411x_{26} - 0.10658x_{27} - 0.47351x_{28}$$
$$-0.71915x_{29} - 1.00678x_{30} + 2.89758x_{31} - 0.93597x_{32} + 0.12209x_{33} + 0.42591x_{34} - 1.02182x_{35}$$
$$+2.44277x_{36} + 0.87034x_{37} - 0.99176x_{38} - 0.83542x_{39} + 2.02701x_{40} + 0.11667x_{41} + 0.08464x_{42}$$
$$-0.91545x_{43} - 0.38361x_{44} - 2.66744x_{45} - 1.29246x_{46} - 1.59715x_{47} - 3.59530x_{48} - 2.31778x_{49}$$
$$+1.00110x_{50} - 0.80262x_{51} - 0.90323x_{52} - 0.17370x_{53} - 0.34825x_{54} + 0.64189x_{55} - 0.07480x_{56}$$
$$+0.42801x_{57} - 0.47223x_{58} - 0.52968x_{59} - 0.12739x_{60} + 3.88641 = 0$$

4 Conclusions

Due to our theoretical background learnt on this subject we were able to code the two seen Support Vector Machine formulations in AMPL. We tested these implementations with a given synthetically generated dataset and a real world one. In both cases validation methods were applied and promising results were obtained. The output of the models allowed us to verify that both methods find the same hyperplane.

Developing this project did not only allow us to corroborate the knowledge we have been adquiring in class, but also broaden our scope to further posibilities regarding Support Vector Machines. Future work on this project could entail testing or developing different kernel functions that better solve different real world problems. For instance, when deciding dataset to test, we came across many datasets with a majority of categorical features. Although we are aware that we could use One-Hot encoding or similar techniques to represent these variables, investigating a kernel for categorical data and comparing its results could be a relevant study.

On the experimental part of this study we have put to practice a real application of this technique and promising results were found. Although the accuracy obtained was not perfect, it correctly classified more instances than we were expecting for such noisy data. We need to take into account that no extra data preprocessing steps were done, as that is out of the scope of this course, so we can only imagine how this technique could perform with more tunning.

5 Appendix: AMPL extra files

5.1 primal.run

```
reset;

# ----------------------------------------------------------
# -- Loading model, data and running model
# ----------------------------------------------------------

# Loading model
model './primal/primal.mod';

# Definition of the params and variables
param n_test;
var df {1..n, 1..d+1};
var df_test {1..n_test, 1..d+1};
var x_test {1..n_test, 1..d};
var y_test {1..n_test};

# Loading data
data './data/data.dat';

# Showing model
#show;
#expand;

# Solving
option solver '../../ampl_macos64/cplex';
solve;

# ----------------------------------------------------------
# -- Computing hyperplane: <w,x> + gamma = 0
# ----------------------------------------------------------

# The computation is immediate since the solver provides
# 'w' and 'gamma'.

# ----------------------------------------------------------
# -- Training accuracy
# ----------------------------------------------------------

var pred_train {i in 1..n} =
    if (sum {j in 1..d} w[j]*x[i,j]) + gamma >= 0
    then 1
    else -1;

var tr_counts;
for {i in 1..n} {
    if y[i] == pred_train[i] then {let tr_counts := tr_counts + 1;}
}
var tr_acc = tr_counts/n;

# ----------------------------------------------------------
# -- Testing accuracy
# ----------------------------------------------------------

var pred_test {i in 1..n_test} =
    if (sum {j in 1..d} w[j]*x_test[i,j]) + gamma >= 0
    then 1
    else -1;

var te_counts;
```

```
60  for {i in 1..n_test} {
61      if y_test[i] == pred_test[i] then {let te_counts := te_counts + 1;}
62  }
63  var te_acc = te_counts/n_test;
64
65  # ------------------------------------------------------------
66  # -- Showing results
67  # ------------------------------------------------------------
68
69  # nu used
70  printf "\nnu = %.4f\n", nu;
71
72  # Hyperplane
73  printf "ws = "; printf {i in 1..d} "%.5f ", w[i]; printf "\n";
74  printf "gamma = %.5f\n",  gamma;
75
76  # Training accuracy
77  printf "pred_train = "; printf {i in 1..n} "%d ", pred_train[i]; printf "\n";
78  printf "tr_acc = %.2f\n", tr_acc;
79
80  # Testing accuracy
81  printf "pred_test = "; printf {i in 1..n_test} "%d ", pred_test[i]; printf "\n
        ";
82  printf "te_acc = %.2f\n", te_acc;
83
84  # alphas
85  printf "s = "; printf {i in 1..n} "%.6f ", s[i]; printf "\n";
```

Listing 3: Implementation of the dual run.

5.2 dual.run

```
reset;

# ----------------------------------------------------------
# -- Loading model, data and running model
# ----------------------------------------------------------

# Loading model
model './dual/dual.mod';

# Definition of variables
param n_test;
var df {1..n, 1..d+1};
var df_test {1..n_test, 1..d+1};
var x_test {1..n_test, 1..d};
var y_test {1..n_test};

# Loading data
data './data/data.dat';

# Showing model
#show;
#expand;

# Solving...
option solver '../../ampl_macos64/cplex';
solve;

# ----------------------------------------------------------
# -- Hyperplane computation: <w,x> + b = 0
# ----------------------------------------------------------

var w {i in 1..d} = sum {j in 1..n} alpha[j] * y[j] * x[j,i];

param sv;
let sv := 1;
repeat until sv >= n or (alpha[sv] >= 0.000001 and alpha[sv] + 0.000001 < nu) {
    let sv := sv + 1;
};

var b = 1/y[sv] - sum {j in 1..d} w[j]*x[sv,j];

# ----------------------------------------------------------
# -- Training accuracy
# ----------------------------------------------------------

var pred_train {i in 1..n} =
    if (sum {j in 1..n}
        alpha[j] * y[j] * sum {l in 1..d} x[j,l]*x[i,l] + b) >= 0
    then 1
    else -1;

var tr_counts;
for {i in 1..n} {
    if y[i] == pred_train[i] then {let tr_counts := tr_counts + 1;}
}
var tr_acc = tr_counts/n;

# ----------------------------------------------------------
# -- Testing accuracy
# ----------------------------------------------------------

```

```
62  var pred_test {i in 1..n_test} =
63      if (sum {j in 1..n}
64          alpha[j] * y[j] * sum {l in 1..d} x[j,l]*x_test[i,l] + b) >= 0
65      then 1
66      else -1;
67
68  var te_counts;
69  for {i in 1..n_test} {
70      if y_test[i] == pred_test[i] then {let te_counts := te_counts + 1;}
71  }
72  var te_acc = te_counts/n_test;
73
74  # ------------------------------------------------------------
75  # -- Showing results
76  # ------------------------------------------------------------
77
78  # nu used
79  printf "\nnu = %.4f\n", nu;
80
81  # Hyperplane
82  printf "ws = "; printf {i in 1..d} "%.5f ", w[i]; printf "\n";
83  printf "b = %.5f\n",  b;
84
85  # Training accuracy
86  printf "pred_train = "; printf {i in 1..n} "%d ", pred_train[i]; printf "\n";
87  printf "tr_acc = %.2f\n", tr_acc;
88
89  # Testing accuracy
90  printf "pred_test = "; printf {i in 1..n_test} "%d ", pred_test[i]; printf "\n
        ";
91  printf "te_acc = %.2f\n", te_acc;
92
93  # alphas
94  printf "alpha = "; printf {i in 1..n} "%.6f ", alpha[i]; printf "\n";
```

Listing 4: Implementation of the dual run.

5.3 data.dat

```
data;

# ------------------------------------------------------------
# -- Train
# ------------------------------------------------------------

# Setting numerical parameters
read n, d < './data/auxtrain.txt';
let nu := num($NU);

# Reading dataset
read {i in 1..n, j in 1..d+1} df[i,j] < './data/auxtrain.txt';

# Filling data (x) and labels (y)
for {i in 1..n} {
    for {j in 1..d} {
        let x[i,j] := df[i,j];
    }
    let y[i] := df[i,d+1];
}

# ------------------------------------------------------------
# -- Test
# ------------------------------------------------------------

# Setting numerical parameters
read n_test < './data/auxtest.txt';

# Reading dataset
read {i in 1..n_test, j in 1..d+1} df_test[i,j] < './data/auxtest.txt';

# Filling data (x) and labels (y)
for {i in 1..n_test} {
    for {j in 1..d} {
        let x_test[i,j] := df_test[i,j];
    }
    let y_test[i] := df_test[i,d+1];
}
```

Listing 5: Implementation of the data loading.