Signal Processing and Optimization for Big Data - Project Presentation

Course offered by

Prof. Paolo Banelli

Venturi Marco, 346951

Perugia, Academic Year 2023/2024
University of Perugia
Master's Degree in Computer Engineering and Robotics
Data Science Curriculum
Department of Engineering

Contents

1. Project Introduction

This project aims to implement and compare two versions of the Support Vector Machine model. The two models implemented are:

- SVM standard model
- SVM ADMM distributed by examples model

The models are compared by examining their convergence and assessing whether, from a real data perspective, the distributed version can match the centralized one. The project also includes model tuning, with a greater focus on understanding how ADMM parameters influence the convergence of the distributed model.

1.1 Parameters and score metrics

The parameters used for the project purposes are:

- **lambda :** This hyper-parameter indicates the regularization inside the SVM model. It's used in the centralized version for model fine-tuning, and in the distributed version, the best lambda obtained from the centralized model grid search is reused.
- **rho :** This hyper-parameter indicates the convergence within the Distributed version. It's used to check which value of rho gives the fastest convergence to the model.
- **Stopping Criteria (or Early Stopping) :** This is not a parameter but an option that indicates stopping the model when it reaches convergence instead of waiting until the model reaches the predetermined iteration value.

Other considerations involve general metrics used for comparing model performance:

- **Accuracy :** This metric is used to assess how well the model generalizes data.
- **Confusion matrix :** This visual matrix provides a general look at how the model misclassifies the two different classes.

- **ROC and AUC :** These metrics offer a better understanding of how the decision function of the model performs in terms of True Positive Rate (TPR) and False Positive Rate (FPR).
- **Loss Trend :** This visualization is used to observe how the loss of distributed models decreases through iterations.
- **Disagreement :** This visualization is used to monitor the trend of the local variables converging to the same value.

1.2 Dataset

In this project, two datasets are used: a synthetic dataset is employed solely to verify if the code implementation adheres to the mathematical description, and a real dataset is used as the second dataset. The last-mentioned dataset is called Apple Quality; it can be found on the Kaggle website. This dataset comes from an American agriculture company and is already scaled and cleaned. It's important to note that the purpose of the project is not to implement a model for a specific task but to develop a framework for distributed optimization and to compare which implementation can be used with respect to a potential company infrastructure. For this reason, the chosen dataset is not deeply analyzed for better model performance scores.

2. Support Vector Machine

The Support Vector Machine is a well-known model used for classification purposes. Its main concept is the construction of a specific hyperplane as a decision boundary that divides the data space into n regions. Each boundary has two additional margin lanes where only the data points, named support vectors, lie on these additional boundaries and not others. In simple words, the Support Vector Machine model aims to generate a decision boundary as far as possible from the data vectors that are close to the generated hyperplane. Because the model's logic is based on the closest nodes that are perpendicular to the hyperplane, these nodes are used to generate the marginal boundary. For this reason, they are called support vectors, as they are the vectors used to define the margin boundaries and, after training the model, they are used to infer new data.

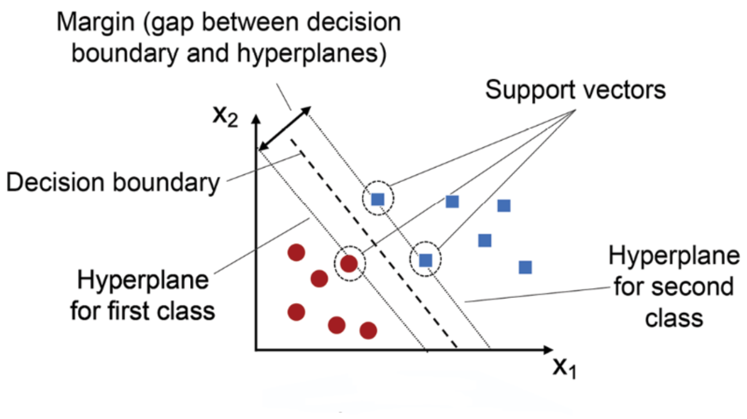

Margin (gap between decision boundary and hyperplanes)
Support vectors
x_2
Decision boundary
Hyperplane for first class
Hyperplane for second class
x_1

Figure 2.1: Support Vector Machine example

The mathematical formulation of SVM is as follows:

$$\min_{(\underline{\beta}, \beta_0)} \frac{1}{N} \sum_{i=1}^{N} \max \left\{0, 1 - y_i \left(\underline{\beta}^T \underline{x}_i + \beta_0\right)\right\} + \underline{\lambda} ||\underline{\beta}||^2$$

This formulation is the basic form used to define the model loss for centralized training and inference. But what could be used to distribute the model over other nodes or CPUs? The answer is the Alternative Directions Multipliers Method (ADMM). ADMM is a mathematical framework that can be used to implement a distributed version of models or distribute the computations over examples or features. In this project, only distribution over examples is demonstrated.

2.1 Alternative Directions Multipliers Method

The Alternative Directions Multipliers method has a generally scaled form as:

$$\underline{x}^{(k+1)} = \underset{\underline{x}_i}{\operatorname{argmin}} \left\{ f(\underline{x}) + \frac{\rho}{2} \left\| A\underline{x} - B\underline{z}^{(k)} - c + \underline{u}^{(k)} \right\|_2^2 \right\}$$
$$\underline{z}^{(k+1)} = \underset{\underline{z}}{\operatorname{argmin}} \left\{ g(\underline{z}) + \frac{\rho}{2} \left\| A\underline{x}^{(k+1)} + B\underline{z} - c + \underline{u}^{(k)} \right\|_2^2 \right\}$$
$$\underline{u}^{(k+1)} = \underline{u}^{(k)} + \left(A\underline{x}^{(k+1)} + B\underline{z} - c \right)$$

This framework provides the opportunity to divide the loss function and the regularization part into separate steps. In this way, it is possible to perform these two steps as separate optimization procedures while maintaining consistency between them. The scaled form, in general, is convenient. The additional part of the objective function ($+\frac{\rho}{2} \left\| A\underline{x}^{(k+1)} + B\underline{z} - c + \underline{u}^{(k)} \right\|_2^2$) is called the Augmented Lagrangian and is used to ensure a strongly convex objective function.

From this framework, it is possible to implement a distributed version of the SVM model that distributes the examples over N agents and uses the first step to calculate the $\underline{x_i}$ local variables and then calculate the global variable z that is used to constrain all the $\underline{x_i}$ to converge to the same value. The Distributed Version over example is:

$$\underline{x}_i^{(k+1)} = \underset{\underline{x}_i}{\operatorname{argmin}} \left\{ \sum_j \max\left\{ \underline{0}, 1 - \underline{a}_{ij}^T \underline{x}_i \right\} + \frac{\rho}{2} \left\| \underline{x}_i - \underline{z}^{(k)} + \underline{u}_i^{(k)} \right\|_2^2 \right\}$$
$$\underline{z}^{(k+1)} = \underset{\underline{z}}{\operatorname{argmin}} \left\{ \lambda \|C\underline{z}\|_2^2 + \frac{N\rho}{2} \left\| \underline{z} - \overline{\underline{x}}^{(k+1)} - \overline{\underline{u}}^{(k)} \right\|_2^2 \right\}$$
$$\underline{u}_i^{(k+1)} = \underline{u}_i^{(k)} + \left(\underline{x}_i^{(k+1)} - \underline{z}^{(k+1)} \right)$$

In the next sections it is possible to observe how the ADMM Framework converge respect to the ρ parameter and how the number of iteration can be reduced before it reach the selected ones

3. SVM Code Implementation

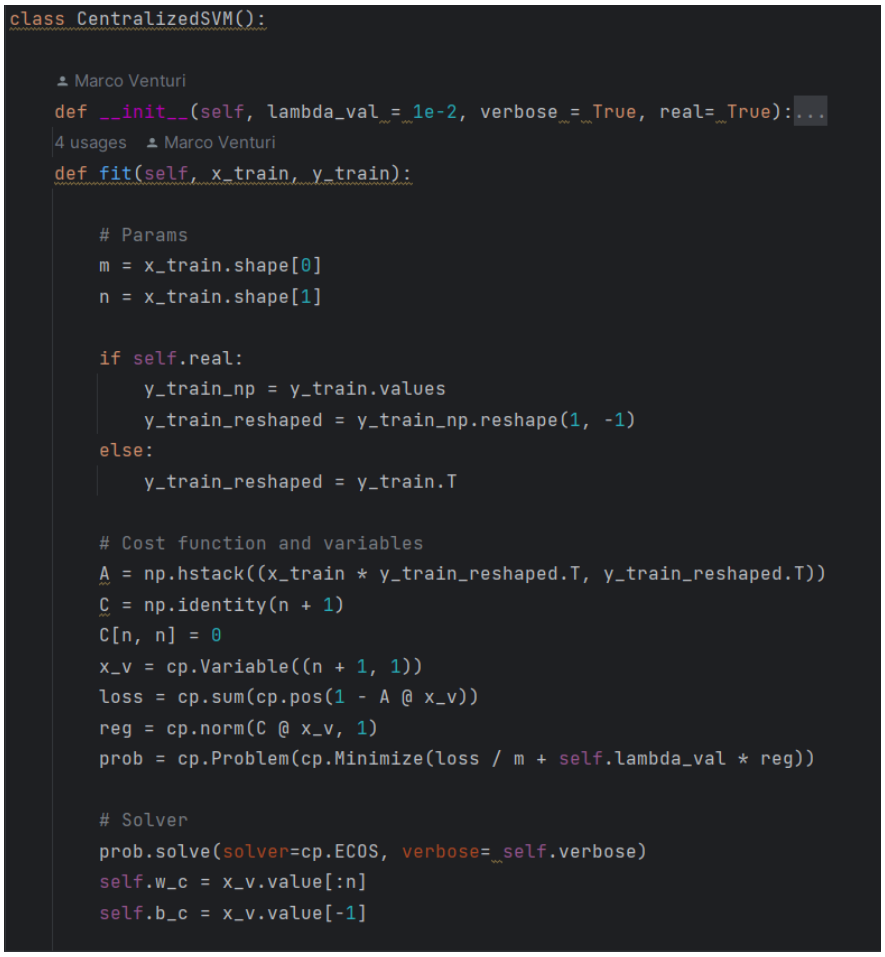
The project is written in the Python language, utilizing CVXPY, a Python-embedded modeling language for convex optimization problems. CVXPY automatically transforms the problem into standard form, calls a solver, and unpacks the results. The solver chosen from the possible CVXPY suggestions is the Embedded Conic Solver (ECOS), a numerical software for solving convex second-order cone programs.

3.1 Centralized SVM

The Centralized version is a simple SVM implementation. I opted not to use scikit-learn's implementation because, for the purpose of the project, it is important to test both models using the same solver to accurately compare their convergence speed.
The Figure 3.1 shows how the training phase of the CentralizedSVM class is implemented. In the fit function, the dimension of the label vector is first managed, followed by the generation of the A matrix of data and then the C matrix. After defining all the cost function variables, the loss function and its regularization part are defined. After the definition of the Loss the fit function call the solver object and then the resulting optimized parameter are saved.

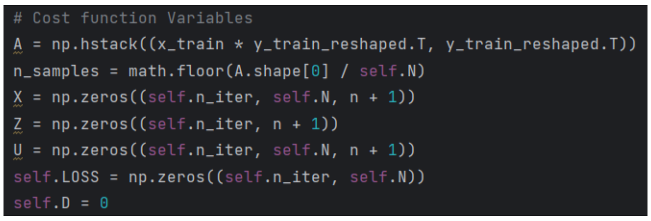
3.2 Distributed SVM (by Samples)

In this section is showed the DistributedSVM class that implement the ADMM framework in Python.
In Figures 3.2 to 3.4, the implementation of the ADMM Framework is depicted. In Figure 3.2, the definition of the cost function parameters is shown, while Figure 3.3 illustrates the loop that runs until the number of iterations exceeds the instance definition and how the first optimization problem is solved. In Figure 3.4, on the other hand, the implementation of the second step (the fusion center) and the third step is shown.
In Figure 3.5, the function that calculates the disagreement between $\underline{x_i}$ agents is shown, along with the stopping criteria if the class instance has been initialized with the parameter 'early_stopping' set to True. In the end the resulting optimized parameter are saved.



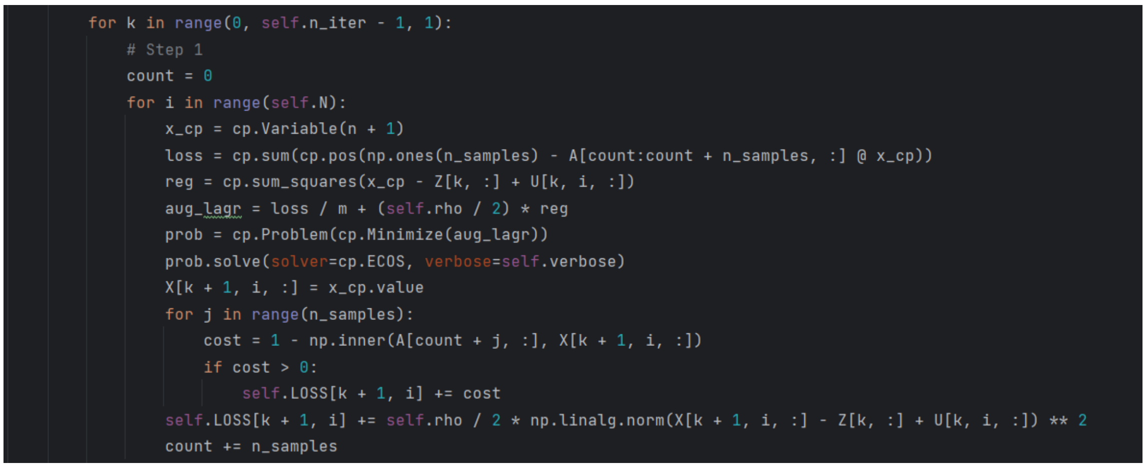
```
class CentralizedSVM():

    Marco Venturi
    def __init__(self, lambda_val = 1e-2, verbose = True, real= True):...
    4 usages   Marco Venturi
    def fit(self, x_train, y_train):

        # Params
        m = x_train.shape[0]
        n = x_train.shape[1]

        if self.real:
            y_train_np = y_train.values
            y_train_reshaped = y_train_np.reshape(1, -1)
        else:
            y_train_reshaped = y_train.T

        # Cost function and variables
        A = np.hstack((x_train * y_train_reshaped.T, y_train_reshaped.T))
        C = np.identity(n + 1)
        C[n, n] = 0
        x_v = cp.Variable((n + 1, 1))
        loss = cp.sum(cp.pos(1 - A @ x_v))
        reg = cp.norm(C @ x_v, 1)
        prob = cp.Problem(cp.Minimize(loss / m + self.lambda_val * reg))

        # Solver
        prob.solve(solver=cp.ECOS, verbose= self.verbose)
        self.w_c = x_v.value[:n]
        self.b_c = x_v.value[-1]
```

Figure 3.1: Support Vector Machine Code



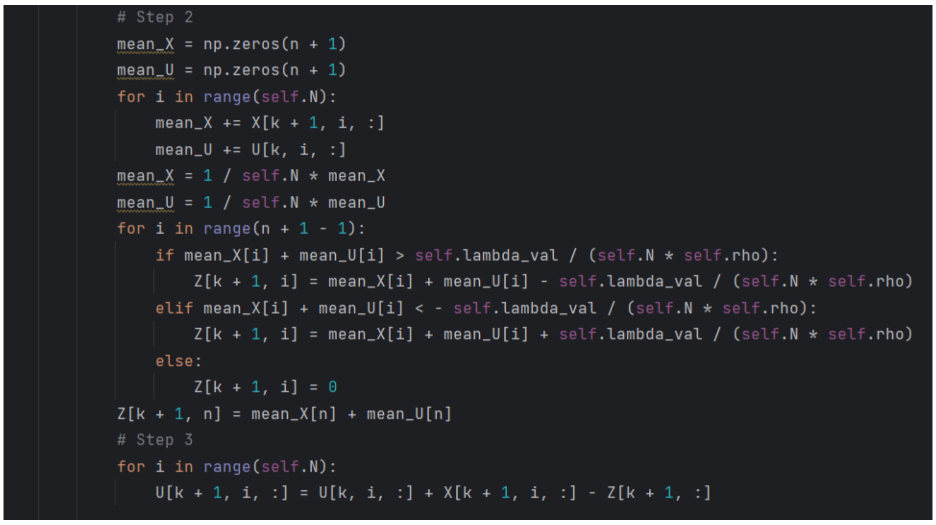
```
# Cost function Variables
A = np.hstack((x_train * y_train_reshaped.T, y_train_reshaped.T))
n_samples = math.floor(A.shape[0] / self.N)
X = np.zeros((self.n_iter, self.N, n + 1))
Z = np.zeros((self.n_iter, n + 1))
U = np.zeros((self.n_iter, self.N, n + 1))
self.LOSS = np.zeros((self.n_iter, self.N))
self.D = 0
```

Figure 3.2: Cost function parameter definition for ADMM



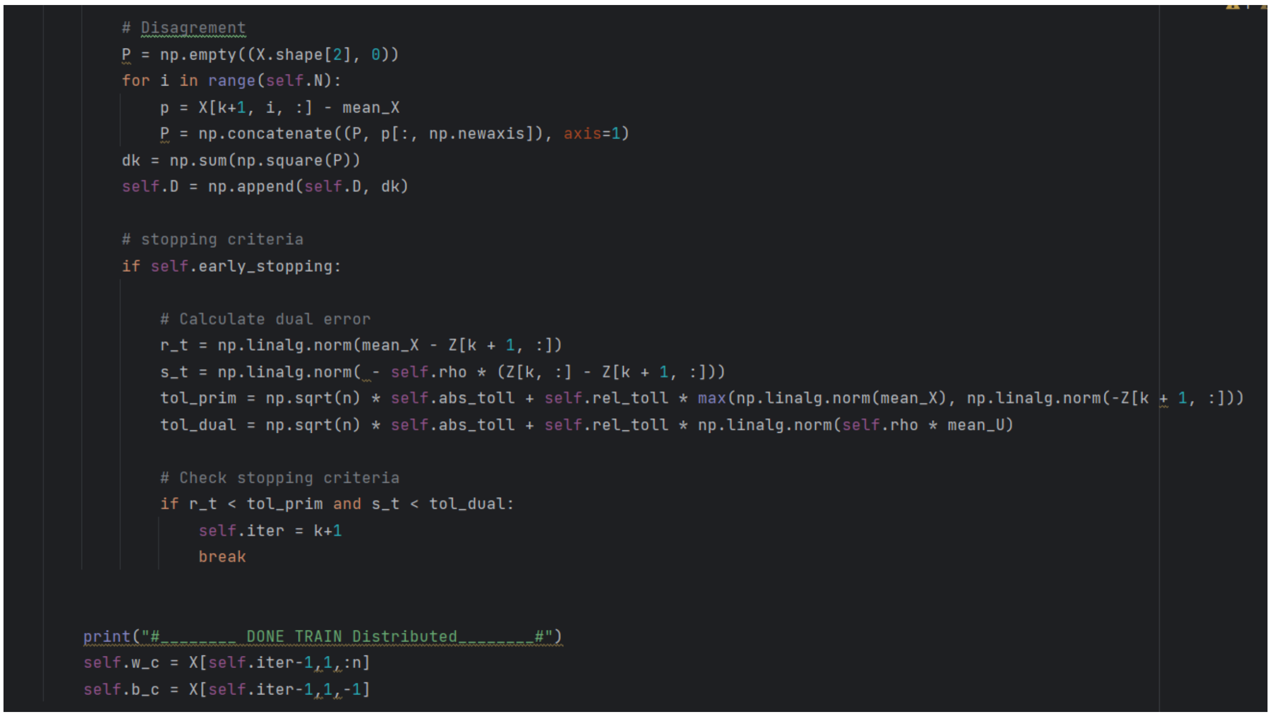
```
for k in range(0, self.n_iter - 1, 1):
    # Step 1
    count = 0
    for i in range(self.N):
        x_cp = cp.Variable(n + 1)
        loss = cp.sum(cp.pos(np.ones(n_samples) - A[count:count + n_samples, :] @ x_cp))
        reg = cp.sum_squares(x_cp - Z[k, :] + U[k, i, :])
        aug_lagr = loss / m + (self.rho / 2) * reg
        prob = cp.Problem(cp.Minimize(aug_lagr))
        prob.solve(solver=cp.ECOS, verbose=self.verbose)
        X[k + 1, i, :] = x_cp.value
        for j in range(n_samples):
            cost = 1 - np.inner(A[count + j, :], X[k + 1, i, :])
            if cost > 0:
                self.LOSS[k + 1, i] += cost
        self.LOSS[k + 1, i] += self.rho / 2 * np.linalg.norm(X[k + 1, i, :] - Z[k, :] + U[k, i, :]) ** 2
        count += n_samples
```

Figure 3.3: Step 1 of ADMM



```
# Step 2
mean_X = np.zeros(n + 1)
mean_U = np.zeros(n + 1)
for i in range(self.N):
    mean_X += X[k + 1, i, :]
    mean_U += U[k, i, :]
mean_X = 1 / self.N * mean_X
mean_U = 1 / self.N * mean_U
for i in range(n + 1 - 1):
    if mean_X[i] + mean_U[i] > self.lambda_val / (self.N * self.rho):
        Z[k + 1, i] = mean_X[i] + mean_U[i] - self.lambda_val / (self.N * self.rho)
    elif mean_X[i] + mean_U[i] < - self.lambda_val / (self.N * self.rho):
        Z[k + 1, i] = mean_X[i] + mean_U[i] + self.lambda_val / (self.N * self.rho)
    else:
        Z[k + 1, i] = 0
Z[k + 1, n] = mean_X[n] + mean_U[n]
# Step 3
for i in range(self.N):
    U[k + 1, i, :] = U[k, i, :] + X[k + 1, i, :] - Z[k + 1, :]
```

Figure 3.4: Step 2 and 3 of ADMM



```
        # Disagreement
        P = np.empty((X.shape[2], 0))
        for i in range(self.N):
            p = X[k+1, i, :] - mean_X
            P = np.concatenate((P, p[:, np.newaxis]), axis=1)
        dk = np.sum(np.square(P))
        self.D = np.append(self.D, dk)

        # stopping criteria
        if self.early_stopping:

            # Calculate dual error
            r_t = np.linalg.norm(mean_X - Z[k + 1, :])
            s_t = np.linalg.norm( - self.rho * (Z[k, :] - Z[k + 1, :]))
            tol_prim = np.sqrt(n) * self.abs_toll + self.rel_toll * max(np.linalg.norm(mean_X), np.linalg.norm(-Z[k + 1, :]))
            tol_dual = np.sqrt(n) * self.abs_toll + self.rel_toll * np.linalg.norm(self.rho * mean_U)

            # Check stopping criteria
            if r_t < tol_prim and s_t < tol_dual:
                self.iter = k+1
                break


    print("#_______ DONE TRAIN Distributed_______#")
    self.w_c = X[self.iter-1,1,:n]
    self.b_c = X[self.iter-1,1,-1]
```

Figure 3.5: Disagreement and Stopping criteria ADMM

3.2.1 Stopping Criteria

This section aim to break throughout the stopping criteria implementation:

- **Dual Error Calculation :**
  - r_t: This variable calculates the dual error based on the norm of the difference between the mean of the data matrix mean_X and the current estimate of the dual variable Z.
  - s_t: This variable calculates another component of the dual error, involving the norm of the difference between the current estimate of the dual variable Z and its previous estimate, scaled by the parameter rho.

  The purpose of calculating these errors is to monitor the convergence of the ADMM algorithm. If both r_t and s_t become sufficiently small, it indicates that the algorithm has converged.

- **Tolerance Criteria :**
  - tol_prim: This variable defines the tolerance for the primal residual error. It is a combination of an absolute tolerance (abs_toll) and a relative tolerance

(rel_toll) scaled by the square root of the number of features (*n*) and the maximum norm of the mean of the data matrix and the current estimate of the dual variable.

  - tol_dual: This variable defines the tolerance for the dual residual error. It is calculated similarly to tol_prim but based on the norm of the scaled mean of the dual update variable mean_U.

- **Stopping Criteria:**
  - The stopping criteria for the ADMM algorithm are checked using the calculated primal and dual errors (r_t and s_t) compared against their respective tolerance thresholds (tol_prim and tol_dual).
  - If both the primal and dual errors fall below their respective tolerance thresholds, the algorithm is considered to have converged, and the iteration count (self.iter) is updated to the current iteration (k+1), and the loop is terminated.

4. Model selection results

The grid search resulting from the Centralized version of SVM outputs a lambda value of $\lambda = 0.01$ from a search space of $(0.0001, 0.001, 0.01, 0.1, 1)$. Figure 4.1 displays the potential performance of the model in terms of true positive rate and false positive rate. In the following sections, we will examine whether the distributed implementation achieves these results.

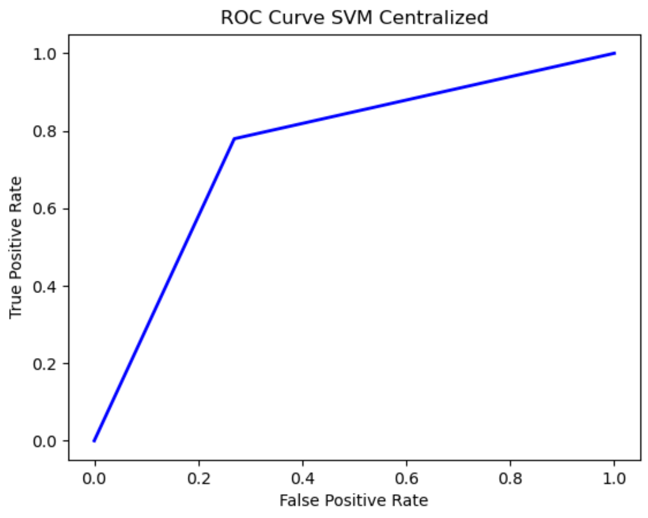


Figure 4.1: ROC curve of classic SVM

4.1 ADMM model selection

Instead of merely observing the resultant scores from distributed Grid Search, it is crucial to examine how the different values of ρ influence loss convergence and how this affects the model's performance. The search space for ρ has been defined as the values $(0.01, 0.1, 0.3, 0.5, 0.7, 0.9)$.

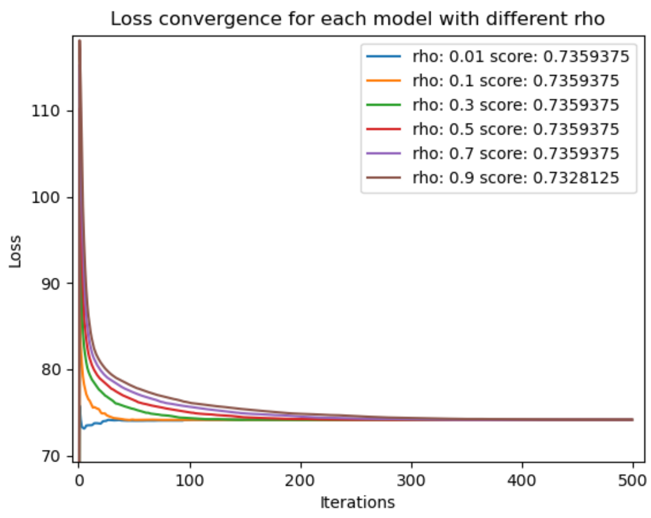


Figure 4.2: Losses convergence respect to different rho values

From Figure 4.2, it appears that all six losses converge to the solution at some point, with a significant difference between $\rho = 0.01$ and the others. However, the scores for all of them converge to the same performance level. Based on these observations, I decided to use $\rho = 0.01$ to analyze the disagreement and the loss of the selected model.

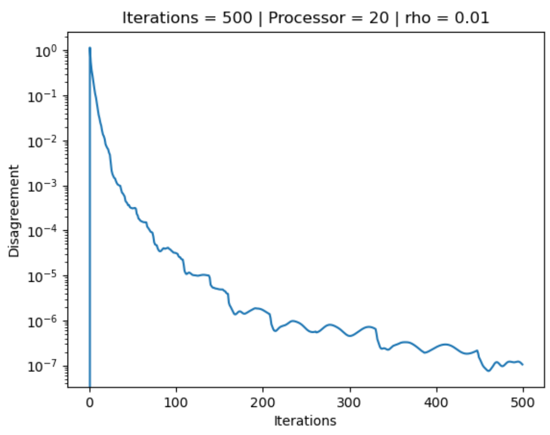


Figure 4.3: Convergence of single local variable to global ones

In Figure 4.3, it is shown how the twenty local variables $\underline{x_i}$ converge to the global variable $\underline{z}$ with a difference of 10^{-7}. While the convergence is respected, it can be noticed how the loss trend, shown in Figure 4.4, converges very rapidly. This observation leads to considering the possibility that perhaps 500 iterations are too many to solve this

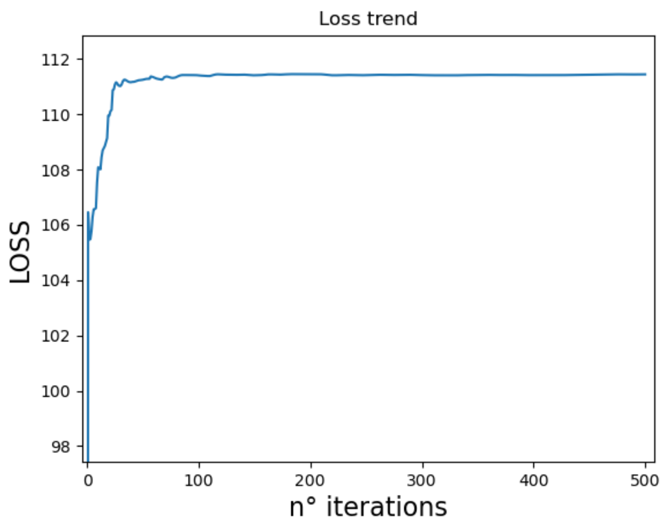


Figure 4.4: Loss convergence respect to $\rho = 0.01$

problem. Therefore, it would be beneficial to observe how the model's behavior changes when stopping criteria techniques are applied.

4.2 ADMM stopping criteria

The implementation of stopping criteria is tested on the previous ρ search space. Figure 4.5 illustrates how all six losses converge to the solution and at which iteration each individual loss stops occurring.

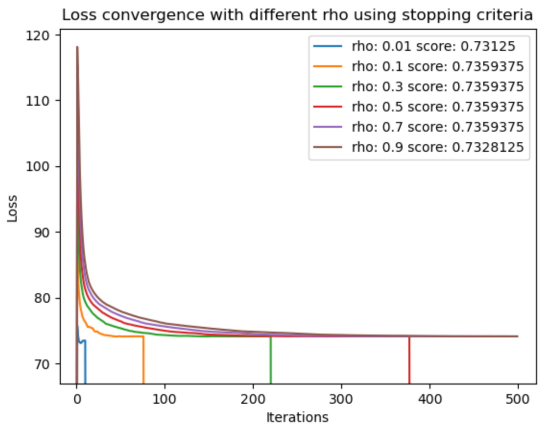


Figure 4.5: Losses convergence respect with stopping criteria application

It appears that the previously selected $\rho = 0.01$, obtained from the ADMM SVM grid search, is no longer the best result as it has a lower score compared to $\rho = 0.1$. Figure 4.6 illustrates the number of iterations reached for each ρ value.

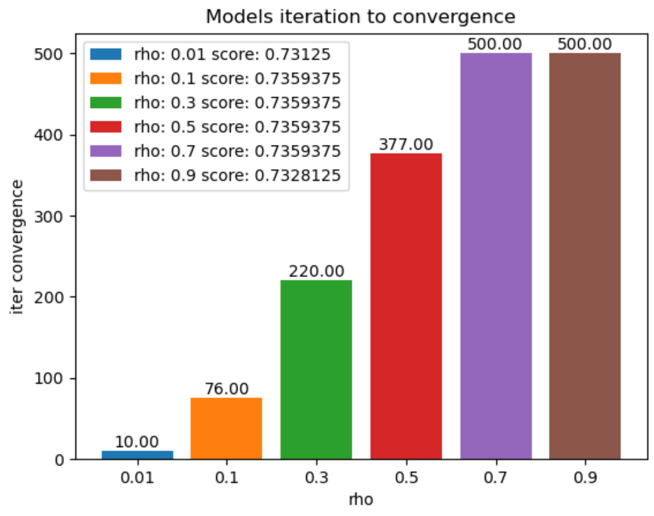


Figure 4.6: Models Iterations respect to stopping criteria

At this moment there are three possible choice to take :

- The model with $\rho = 0.01$ is constrained to have a number of iterations between 10 and 75 to determine if there is a lower number of iterations compared to the model with $\rho = 0.1$. This is because from the last observation, it is clear that $\rho = 0.01$ can achieve a higher score.
- Because the model with $\rho = 0.01$ is capable of convergence, it is possible to adjust the current tolerance values for a better understanding of why the first model is outputted before convergence.
- The model with $\rho = 0.1$ is chosen because it has the best score and requires fewer iterations, without any other assumptions.

The decision tree has been chosen because, in this case, it does not matter to implement a more optimized version, as the results achieved are satisfactory for the project's purpose. The loss trend of the chosen model is depicted in Figure 4.7, while the disagreement is shown in Figure 4.8. The disagreement appears to be greater by an order of magnitude compared to the last disagreement shown.

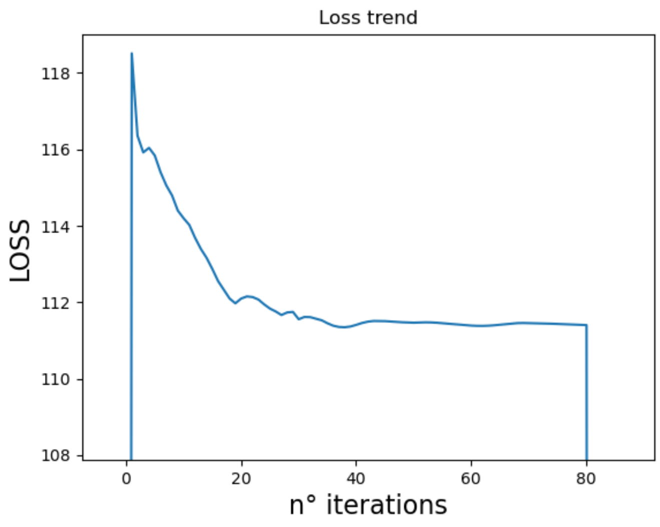


Figure 4.7: Loss from the model with $\rho = 0.1$ with stopping criteria

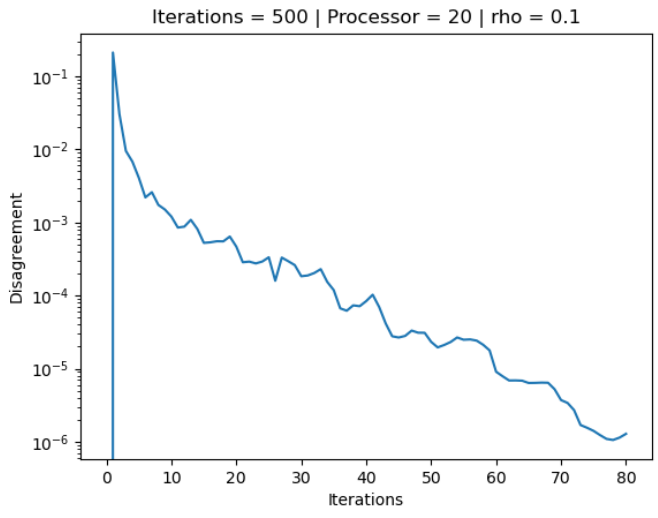


Figure 4.8: Disagreement from the model with $\rho = 0.1$ with stopping criteria

5. Comparison and Conclusions

The two models obtained from the last chapter are used to compare performance against the centralized and distributed versions. In Figure 5.1 and 5.2, it is observed that the two models exhibit the same performance regarding the prediction of the same test set.

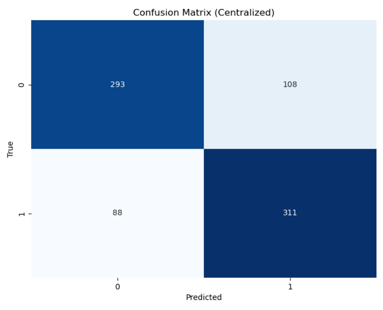


Figure 5.1: Centralized Confusion Matrix

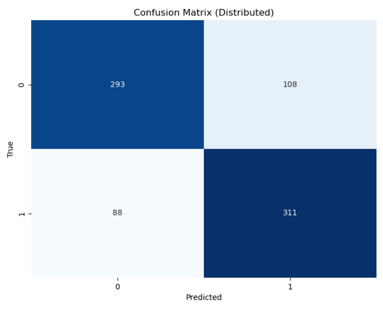


Figure 5.2: Distributed Confusion Matrix

The same performances are also observed with respect to the accuracy score and the ROC and AUC score. Those comparisons are depicted in Figures 5.3 and 5.4.

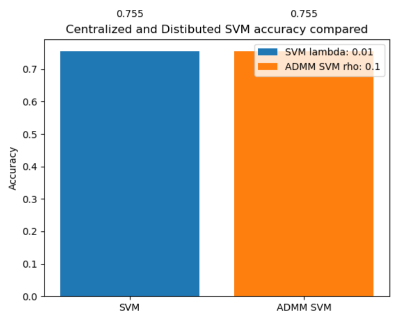


Figure 5.3: Accuracy score comparison

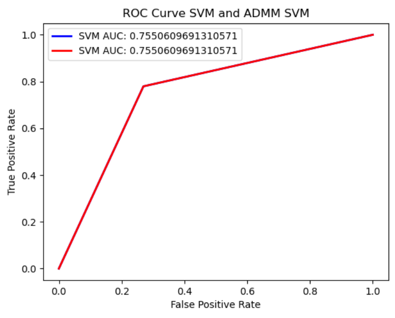


Figure 5.4: ROC and AUC comparison

5.1 Conclusion

The distributed model converges to the same results as the centralized one with this dataset. For further study, it would be interesting to test the performances with datasets containing a larger number of samples, as well as datasets with a large number of samples and features, to observe potential differences in convergence between the two models. Additionally, investigating whether the distributed model achieves convergence in less time than the centralized one and if it outperforms the centralized one would be valuable. Another interesting avenue would be to implement these models using different solvers with various dual ascent algorithms to gain a better understanding of the optimization speed relative to them.