Support Vector Machine Applied to the Classification of Iris Species

Alonso Gutierrez
Jessica Quiroz Galvez
MAP 4112

Introduction

For this project, we chose to implement the Support Vector Machine or SVM algorithm using a real data set pertaining to the classification of Iris plant species using Python. SVM is a machine learning algorithm used for binary classification of data, where each class of data is usually denoted by either 1 or -1 by solving the following optimization problem:

$$min \frac{\|\omega\|^2}{2}$$

$$\text{s.t } y_i(\omega \cdot x_i + b) \geq 1 \text{ for all } 1 \leq i \leq N,$$

where N is the number of samples in the dataset.

It works by feeding it a dataset, which contains n-dimensional feature vectors, denoted by x_i, that contain the features of each sample. Each x_i is assigned to a class y_i that could be designated either a 1 or a -1 depending on how the algorithm classifies the sample. The algorithm creates a hyperplane that separates the two classes of data by finding the best fitted boundary in terms of its weights, along with the widest margin, ω, of said boundary (taking into account the bias b), which yields the better separation of the two groups.

Background

The dataset chosen for this project is Iris Species from Kaggle.com. It contains 50 samples of each of the three iris plant species (Iris Virginica, Iris Setosa, and Iris Versicolor) where one species is linearly separable from the other two. The features/columns of this dataset are:

- Id: id of the sample
- SepalLengthCm: length of sepal in centimeters
- SepalWidthCm: width of sepal in centimeters
- PetalLengthCm: length of petal in centimeters
- PetalWidthCm: width of petal in centimeters

- Species: species name

For simplicity, we removed the Iris Virginica species samples, and will only be working with Iris Setosa and Iris Versicolor, which are linearly separable.
Based on these features, the SVM algorithm will be trained to find the two separable Iris plant classes by finding the hyperplane within the 4-dimensional space.

Algorithm

We first begin by importing the necessary libraries to aid with the calculations needed for this algorithm.

```
# Imports
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from sklearn.metrics import accuracy_score, recall_score, precision_score

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import pprint as pp
```

Then, we prep the dataset in order for it to be fed to the algorithm, for instance shuffle rows to ensure fairness and using a scaler to have values that are easier to work with. Additionally, we split the data horizontally in half. The first half will be used to train the algorithm and the second half will be used to test it. Lastly, we inserted 1's in every row to be able to push the bias intercept b into the ω vector: $f(x_i)=\omega \cdot x_i$, where $\omega=(\omega', b)$, $x_i=(x_i', 1)$, where ω' is the original weight vector and x_i is the original dataset.

```
df = pd.read_csv('Iris.csv',index_col='Id')

#Removing third category of Iris two simplify process
df = df[df.Species != 'Iris-virginica']

df = df.replace(['Iris-setosa','Iris-versicolor'],[-1,1]).astype(np.float64)
df = shuffle(df)
Y = df.Species
X = df.drop(['Species'],axis=1)
X.insert(loc=len(X.columns), column = 'intercept', value =1)
XN = MinMaxScaler().fit_transform(X.values)
X = pd.DataFrame(XN)
X_tr,X_te,Y_tr,Y_te = train_test_split(X,Y,
                                       test_size=0.4,
                                       random_state=42)
```

The next step is to call the class svm() which contains the four methods needed for the SVM algorithm. However, before we get to the methods, we had to initialize the following variables:

- learning_rate: learning rate parameter for hyperplane set to 0.0001
- lambda_parameter: set to 0.01
- epochs: number of iterations of data for training set to 5000
- weights: ω will be calculated later
- reg_strength: regularization strength parameter for hyperplane set to 10000

```
class svm:

    import numpy as np
    import matplotlib.pyplot as plt
    import pandas as pd


    def __init__(self, learning_rate = 0.00001, lambda_parameter = 0.01, epochs␣
 ↪= 5000):

        self.learning_rate = learning_rate
        self.lambda_parmeter = lambda_parameter
        self.epochs = epochs
        self.weights = None
        self.reg_strength = 10000
```

The first method in the SVM class is cost_computation. In this method we calculate the Lagrangian or Cost Function.

$$L(\omega)=\frac{\|\omega\|^2}{2}+C\left[\frac{1}{N}\sum_{i=1}^{n} max\left(0,1-y_i * (\omega\cdot x_i+b)\right)\right]$$

Here, we first calculate the distance which is $1-y_i * product\ of\ x_i\ with\ \omega$. Then we calculate ethe hinge loss by multiplying the regularization strength, C, by the sum of all the distances over N. And finally, we calculate the Lagrangian by finding the dot product of omega by itself times ½ plus the hinge loss.

```
    def cost_computation(self, W, X, Y):

        # Hinge loss calculation
        n = X.shape[0]
        distances = 1 - Y * (np.dot(X, W))
        distances[distances < 0] = 0 # make all distances less than 0 equal to␣
↪0.
        hinge_loss = self.reg_strength*(np.nansum(distances)/n)

        # Calculate cost
        cost = 1/2 * np.dot(W,W) + hinge_loss
        return cost
```

The second method is cost_gradient_computation. This is where we coded the gradient or derivatives of the Lagrangian/Cost Function to be able to minimize it and turn it into a dual problem which is done in the next method.

$$\nabla_\omega L(\omega)=\frac{1}{N}\sum_{i=1}^{n}\begin{cases}\omega & if\ max\left(0,1-y_i * (\omega\cdot x_i)\right)=0\\ \omega-C\,y_i x_i & otherwise\end{cases}$$

```
def cost_gradient_computation(self, W, X_ij, Y_ij):

    if type(Y_ij) == np.float64:
        Y_ij = np.array([Y_ij])
```

```
        X_ij = np.array([X_ij])  # gives multidimensional array

    distance = 1 - (Y_ij * np.dot(X_ij, W))
    dw = np.zeros(len(W))

    for i, d in enumerate(distance):
        if max(0,d) == 0:
            di = W
        else:
            di = W - (self.reg_strength * Y_ij[i]* X_ij[i])
        dw += di

    dw = dw/len(Y_ij) # average calculation
    return dw
```

The third method is sgd_fit. In this method, we calculate the stochastic gradient descent to minimize the Lagrangian/Cost Function. The way Gradient Descent works is by first, finding the gradient/derivative of the Lagrangian (done in cost_gradient_computation), then moving the opposite direction of the gradient by a specific rate since the gradient is the fastest increase direction of the Lagrangian (we do this by updating the weight $\omega = \omega - (self\ learning\ rate\ *\ gradient)$), and then repeat the first two steps until we have found the smallest Cost Function.

```
def sgd_fit(self, X, Y):

    # Stochastic gradient descent
    max_epochs = 5000
    self.weights = np.zeros(X.shape[1])
    nth = 0
    prev_cost = 9999999
    cost_threshold = 0.001  # in percent

    # stochastic gradient descent
    for epoch in range(1,max_epochs):

        # shuffle
        X_i, Y_i = shuffle(X, Y)

        for i, x in enumerate(X_i):

            ascent = self.cost_gradient_computation(self.weights, x, Y_i[i])
            self.weights = self.weights - (self.learning_rate * ascent)

        if epoch == 2 ** nth:
            cost = self.cost_computation(self.weights, X, Y)
            print("Epoch is: {} and Cost is: {}".format(epoch, cost))
            nth += 1

    return self.weights
```

The fourth method is the predict method. Here is where we calculate $\omega * x_i$ and we print our prediction of the species compared with the actual species of our samples.

```
def predict(self, X, Y):

    Y_te_predictions = np.array([])
```

```
    for i in range(X.shape[0]):
        Yp = np.sign(np.dot(self.weights, X.to_numpy()[i]))
        Y_te_predictions = np.append(Y_te_predictions, Yp)

    print("\naccuracy: {}".format(accuracy_score(Y, Y_te_predictions)))
    print("recall: {}".format(recall_score(Y, Y_te_predictions)))
    print("precision: {}".format(precision_score(Y, Y_te_predictions)))

    return Y_te_predictions
```

Once the training is completed, the second half of the dataset can be fed for testing.

Results

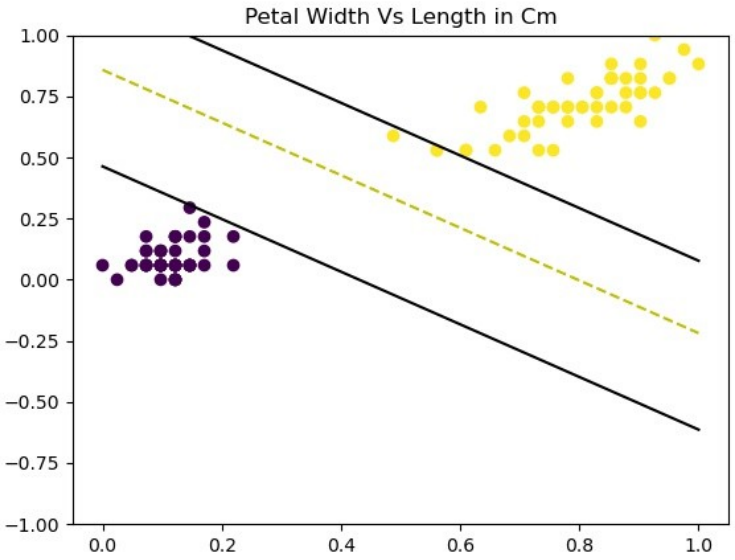
Starting at epoch zero our weights were defaulted at a starting value of zero, but as the algorithm begins it's descending into a better solution we see that our cost computation is in the 3 orders of magnitude (Epoch is: 1 and Cost is: 2449.5960395476714) with a better but not great approximation of what our features' weights should be ([0.22576964 -1.13302155 1.02397005 0.95844238]). The progression through the allowed number of iterations sets us a magnitude lower in our cost computation (Epoch is: 512 and Cost is: 150.08996523861157, weights: [0.53530258 -4.66174945 2.87363584 1.17607862]). Finally, closely to the allowed number of iterations, at epoch 4096 we have almost settled at an answer that is still 2 orders of magnitudes (Epoch is: 4096 and Cost is: 136.42632515296685 , weights: [1.23815623 -7.75490172 5.94308176 -0.25348937]). This is resemblant of a logarithmic decay in our cost calculation. Even thou, we could allow the algorithm to keep on iterating and readjusting our weights it would be best to evaluate the accuracy, precision and recall of it in order to avoid over-fitting and waste of resources. At this limit we see that the classifier has acquired a great (if not amazing) score in each category:

accuracy: 1.0
recall: 1.0
precision: 1.0

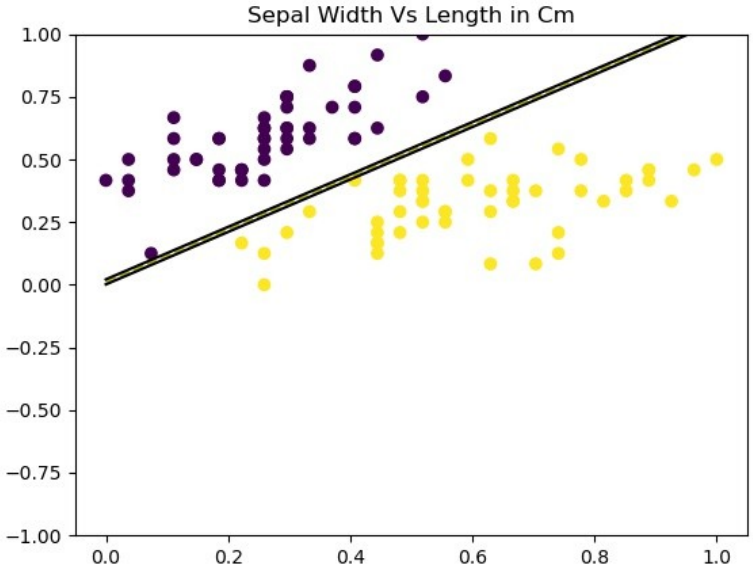
```
reading dataset…

Epoch is: 1 and Cost is: 2748.590251762838
Epoch is: 2 and Cost is: 795.1200104594031
Epoch is: 4 and Cost is: 270.34296610945535
Epoch is: 8 and Cost is: 186.53035437449785
Epoch is: 16 and Cost is: 160.67116493114304
Epoch is: 32 and Cost is: 158.82345300339355
Epoch is: 64 and Cost is: 161.93551892198573
Epoch is: 128 and Cost is: 158.51971124809324
Epoch is: 256 and Cost is: 156.96743619582224
Epoch is: 512 and Cost is: 150.23046520728812
Epoch is: 1024 and Cost is: 145.24665739861337
Epoch is: 2048 and Cost is: 137.82827718297412
Epoch is: 4096 and Cost is: 136.53505481593226

accuracy: 1.0
recall: 1.0
precision: 1.0

weights
[ 1.29658965 -7.90503854  6.14229532 -0.29780744  0.        ]

side by side comparison of outputs vs real solutions:
[(-1.0, -1.0),
 (-1.0, -1.0),
 (1.0, 1.0),
 (-1.0, -1.0),
 (-1.0, -1.0),
 (1.0, 1.0),
 (1.0, 1.0),
 (1.0, 1.0),
 (1.0, 1.0),
 (-1.0, -1.0),
 (1.0, 1.0),
 (-1.0, -1.0),
 (1.0, 1.0),
 (1.0, 1.0),
 (-1.0, -1.0),
```

```
(-1.0, -1.0),
(1.0, 1.0),
(-1.0, -1.0),
(1.0, 1.0),
(-1.0, -1.0),
(1.0, 1.0),
(-1.0, -1.0),
(1.0, 1.0),
(1.0, 1.0),
(1.0, 1.0),
(-1.0, -1.0),
(-1.0, -1.0),
(-1.0, -1.0),
(1.0, 1.0),
(1.0, 1.0),
(-1.0, -1.0),
(-1.0, -1.0),
(1.0, 1.0),
(-1.0, -1.0),
(-1.0, -1.0),
(1.0, 1.0),
(1.0, 1.0),
(1.0, 1.0),
(-1.0, -1.0),
(-1.0, -1.0)]
```

For visualization purposes, we ran our code with only two columns of data at a time, so we could picture the 2-dimensional scenarios where the hyperplane is a line.
The first one is Petal Width and Petal Length, where Iris Setosa is represented as the purple dots and Iris Versicolor is represented as the yellow dots.



The second one is Sepal Width and Sepal Length where the Iris species are depicted as above:



Conclusion

Although the results are good enough it is best to keep in mind that this is drawn from a very simplistic and small data set. Moreover, Although regular gradient descent would have been enough for this project, stochastic gradient descent was employed since originally the classifier was being training on a larger, more complex data set built to train a classifier to predict stroke in patients. These sorts of factors can greatly affect the perceived efficacy of the classifier. Lastly, it

is important to keep in mind that due to the simplicity of the set, combined with the complex stochastic descent embedded in the algorithm, our classifier was trained incredibly well, but may fall to short in performance with other sets depending too on the hardware used to run the code. At the moment this classifier was trained on an intel i7-6700HQ + 16gb ddr4 of ram.

References

Abbassi, Q. (2020, April 1). SVM from scratch - python. Medium. Retrieved December 2, 2021, from https://towardsdatascience.com/svm-implementation-from-scratch-python-2db2fc52e5c2#66a2.

Deisenroth, M. P., Faisal, A. A., & Ong, C. S. (2020). Mathematics for Machine Learning. Cambridge University Press.

Learning, U. C. I. M. (2016, September 27). Iris species. Kaggle. Retrieved December 2, 2021, from https://www.kaggle.com/uciml/iris.
1.

Wikimedia Foundation. (2021, December 2). Support-Vector Machine. Wikipedia. Retrieved December 2, 2021, from https://en.wikipedia.org/wiki/Support-vector_machine.