Methodology of Creating SVM Kernels from Scratch Using Python and NumPy

Austin Lackey and Tomy Sabalo Farias

DSCI 320, Colorado State University

December 15, 2023

Abstract

This paper presents a methodology for creating Support Vector Machine (SVM) kernels from scratch using Python and NumPy. We discuss the implementation of linear, sigmoid, polynomial, and radial basis function (RBF) kernels in a binary and multiclass SVM. These kernels are tested on E. coli data and compared to the results of the scikit-learn SVM implementation. The goal of the E. coli dataset is to predict the localization sites of proteins inside the cell. The results show that the implemented kernels often yield better accuracy than the scikit-learn implementation; at the cost of increased training time. Kernel training times are ran multiple times and averaged to provide a more accurate metric since training times are low and have a high variance.

1 Introduction

Support Vector Machines (**SVMs**) are machine learning algorithms that are used to complete classification tasks. When we have a large dimensional data set that we would like to create decision boundaries, we can create a hyperplane that maximizes the margin between the observations' labels. The data points that are closest to the decision boundary influence the position and direction of the hyperplane, and these close data points are called **Support Vectors**. The goal is to optimize the distance between the support vectors and the hyperplane (also known as the **margin**), and so we can draw a line between the points.

With higher dimensional data, sometimes a linear boundary will not be effective, and so we can implement kernel tricks. These kernels allow for the original input to be mapped to a higher-dimensional space where a linear boundary can be drawn. We will demonstrate the usage of SVMs, as well as the effect that different kernels have on the same data.

2 Methodology

When in discussion of SVMs, we need to understand the steps behind the math to know what it is doing. Here, we will discuss how this algorithm is run to calculate our optimal parameters for our hyperplane decision boundary.

Our goal is to find a hyperplane in the form:

$$f(x) = w \cdot x + b = 0$$

We want our decision boundary to have a maximal margin between the hyperplane and the support vectors. These vectors are found with:

$$w \cdot x + b \geq 1$$

$$w \cdot x + b \leq -1$$

Since we want to maximize the margin in between, we can think about finding those inputs that satisfy the support vectors in the right labels. For example, if we find an x_n such that it is the nth observation in the dataset, and it has the label $y_n = 1$, then we want this observation to be found above the positive support vector. Mathematically, we can say that:

$$\begin{cases} w \cdot x + b \geq 1 & \text{if } y = 1 \\ w \cdot x + b \leq -1 & \text{if } y = -1 \end{cases}$$

In other words, we can simply this to be:

$$y_i(w \cdot x_i + b) \geq 1$$

for $i = 1, ..., N$

The support vectors are described, but we want to find the maximum length a unit step takes from the decision boundary to the support vectors to be optimized. Therefore, if we are given a vector x, with an orthogonal vector w to the decision boundary, we can take a unit step from the decision boundary to the support vector by:

$$1 = w \cdot x + b = w(x + \frac{w}{|w|^2}) + b = 1$$

We know that $w \cdot x + b = 0$, so we can simplify:

$$\frac{ww}{|w|^2} = 1$$

$$\frac{1}{|w|}$$

This represents half the margin of the SVM, and so we need to do it to both sides of the boundary:

$$\frac{2}{|w|}$$

Therefore, to maximize the margin of the hyperplane, we need to minimize w.

We can combine this all to formulate our optimization problem: minimize $||w||$ constrained by $y_i(w \cdot x_i + b) \geq 1$ for $i = 1, ..., N$. We can set up a Lagrange dual optimization problem, such that:

$$L = \frac{1}{2}||w||^2 - \sum_{i=1}^{n} \alpha_i \cdot [y_i(w \cdot x + b) - 1]$$

Minimizing for w and b, we get:

$$w = \sum_{i=1}^{n} \alpha_i y_i x_i$$

$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

Our problem can now be in terms of a constrained minimization problem by plugging in the constraints we got. This can be simplified into a quadratic form. The final form of our problem will be

$$L = 1^T \alpha - \frac{1}{2} \alpha^T X^T X \alpha \tag{1}$$

2.1 Kernel Functions

We implemented four different kernel functions which are listed below.

- Linear Kernel
$$K(X, Y) = X^T Y \tag{2}$$
- Sigmoid Kernel
$$K(X, Y) = \tanh(\gamma X^T Y + r) \tag{3}$$
- Polynomial Kernel
$$K(X, Y) = (\gamma X^T Y + r)^d, \gamma > 0 \tag{4}$$
- Radial Basis Function (RBF) Kernel
$$K(X, Y) = \exp(-\gamma ||X - Y||^2), \gamma > 0 \tag{5}$$

The kernel tricks used to form a linear decision boundary on a non-linear data set will be implemented into the optimization problem (1). For the linear kernel, no difference is made because it is the same thing that is done in the equation, just the product of the inputs. However, for the other kernels, the trick is implemented in the transformation of the input:

$$(X^T X)_{ij} = \sum_d y_i x_{id} \cdot y_j x_{jd} = \sum_d y_i y_k \cdot x_{id} x_{jd}$$

$$(k^T k)_{ij} = \sum_{d'} y_i k_{id'} \cdot y_j k_{jd'} = \sum_{d'} y_i y_k \cdot k_{id'} k_{jd'}$$

The k is found using the different kernel equations [(2)-(5)], which allows for versatility in the shape of the boundary line. It is the optimal decision boundary that contains

2.2 Binary SVM

A binary classification on the data is the foundation of the calculations. We want to form a hyperplane that separates two labels effectively, with a margin for error when making predictions on new observations. The steps to complete to find the terms for our hyperplane can be seen below:

- For each training example $(\mathbf{x}_i, y_i)$:
  - Calculate error E_i for $\mathbf{x}_i$ using the current model:

    $$E_i = f(\mathbf{x}_i) - y_i$$

    $$f(x_i) = \sum_{j=1}^{n} \alpha_i y_i K(x_j, x_i) + b$$

  - If $(y_i \cdot E_i < -tol$ and $\alpha_i < C)$ or $(y_i \cdot E_i > tol$ and $\alpha_i > 0)$:
    - ∗ Select a second example $(\mathbf{x}_j, y_j)$ to update alpha (using heuristics to choose $j \neq i$)
    - ∗ Calculate error E_j for $\mathbf{x}_j$
    - ∗ Save old alpha values: $\alpha_i^{old} = \alpha_i$, $\alpha_j^{old} = \alpha_j$
    - ∗ Compute bounds L and H for α_j (satisfying $0 \leq \alpha_j \leq C$)
    - ∗ If L equals H, continue to the next training example
    - ∗ Compute η (second derivative of the SVM objective function)
    - ∗ Update α_j:

      $$\alpha_j = \alpha_j^{old} - \frac{y_j \cdot (E_i - E_j)}{\eta}$$

      (Clip α_j if necessary: $\alpha_j = \text{clip}(\alpha_j, L, H)$)

      (Continue if $|\alpha_j - \alpha_j^{old}| <$ epsilon)

    - ∗ Update α_i:

      $$\alpha_i = \alpha_i^{old} + y_i \cdot y_j \cdot (\alpha_j^{old} - \alpha_j)$$

    - ∗ Update bias term b (compute b_1 and b_2, adjust b based on α_i and α_j, update E for all examples)

Using this algorithm, you are able to find the w's and b that define the hyperplane between the two classes. The $K()$ function represents the kernel that is being used, and so this is where the different transformations will have an effect.

2.3 Multiclass SVM

The multi-classification on a data set using SVM is only a slight variation from the binary classification. In binary, we trained the model on a **1 vs 1** basis. We had two classes, -1 and 1, and wanted to create a decision boundary between them. However, multiple classes means that we have more than two classes, and you can't just draw one line to separate the labels.

Therefore, the meta-algorithm for multi-class is to approach with a **one vs others** mindset. For however many classes you have, you want to create a decision boundary between one class and all the others, and repeating this until you have done so for each label. It follows the same steps as binary classification, but does it s times for the number of labels in the data set to differentiate.

3 Data Overview

The data is composed of seven predictors of type float, with the last column being the localization site class label. There were 336 total observations in the data set, and the class labels were not evenly distributed. As you can see below, the class labels *cp* and *im* make up the majority of the data set. We used these two classes as the positive and negative classes for the binary SVMs as they were the most common. For a test/train split, we used 80% of

Table 1: Class Frequency

Class	**Frequency**
cp	143
im	77
pp	52
imU	35
om	20
omL	5
imS	2
imL	2

the data for training and 20% for testing. This resulted in 268 training observations for the multi-class SVMs and 176 training observations for the binary SVMs.

4 Results and Discussion

4.1 Training Time and Accuracy

After implimenting the SVM kernels for both binary and multiclass classification, we tested the kernels on the E. coli data set. Since training times were low and had a higher variance, we ran each kernel 10 times and averaged the results. This allows for a more accurate representation of the training time for each kernel to prevent outliers from skewing the results. The results of the training times and accuracy can be seen in Table 2.

Table 2: Classifier Performance

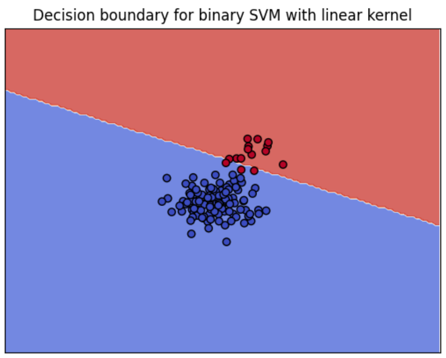
Classifier	**Implementation**	**Kernel**	**Avg Accuracy**	**Avg Runtime**
Binary	sklearn	linear	1.00000	0.00052
Binary	custom	linear	0.99318	0.00795
Binary	sklearn	sigmoid	0.68180	0.00066
Binary	custom	sigmoid	0.99546	0.00757
Binary	sklearn	rbf	1.00000	0.00054
Binary	custom	rbf	0.99546	0.00977
Binary	sklearn	poly	1.00000	0.00040
Binary	custom	poly	0.97498	0.00929
Multi	sklearn	linear	0.77940	0.00096
Multi	custom	linear	0.81468	0.15420
Multi	sklearn	sigmoid	0.47060	0.00183
Multi	custom	sigmoid	0.82497	0.15641
Multi	sklearn	rbf	0.75000	0.00156
Multi	custom	rbf	0.82350	0.19177
Multi	sklearn	poly	0.76470	0.00111
Multi	custom	poly	0.83967	0.36326

Note: These times were calculated with 10 runs for each kernel and averaged.

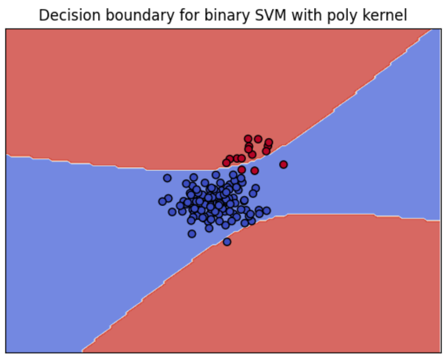
I think that the results are interesting because the custom implementation of the kernels often outperformed the scikit-learn implementation. However, the custom implementation of the kernels took significantly longer to train. Since we used Stochastic Gradient Descent (SGD) to train the SVMs, the convergence can take a long time. In Sklearn, they use LibSVM which is a C++ library that is much faster than our custom implementation. Although the multi-class SVMs had lower accuracy than the binary SVMs, the training times were much higher. This makes sense because the multi-class SVMs are just a combination of binary SVMs and the decision boundaries start to become more complex and harder to fit accurately.

4.2 Decision Boundaries

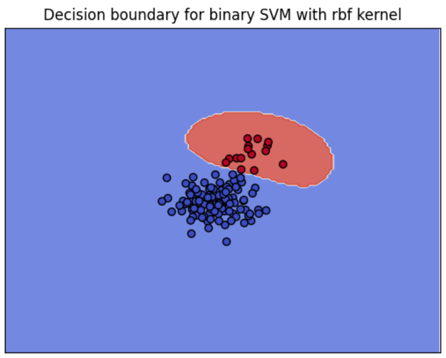
The decision boundaries for the binary and multi-class SVMs can be seen in Figures 1 and 2. In Figure 1, the decision boundaries for the binary SVMs are shown. We can see how each kernel affects the decision boundary. Although these examples are intuitive and easy to understand, the data was reduced to 2 dimensions using Principal Component Analysis (PCA). We decided to use PCA to reduce the dimensionality of the data because we knew that the training data would retain more information than regular feature selection. By implementing feature extraction, we were able to reduce the dimensionality of the data while retaining more information. This allows us to show the decision boundaries in 2 dimensions while still retaining the accuracy of the SVMs.



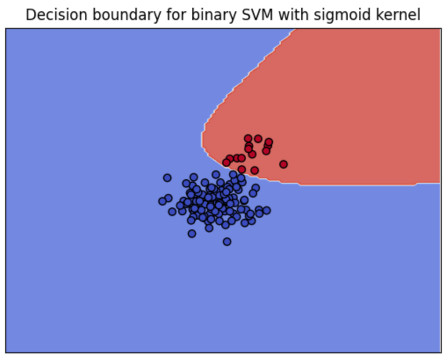
(a) Linear Binary SVM



(b) Polynomial Binary SVM



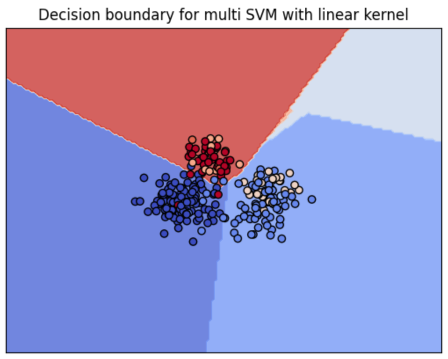
(c) RBF Binary SVM



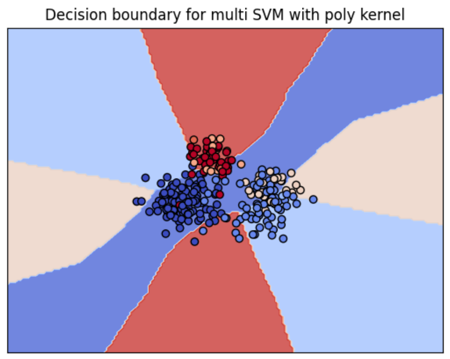
(d) sigmoid Binary SVM

Figure 1: Binary SVM Decision Boundaries

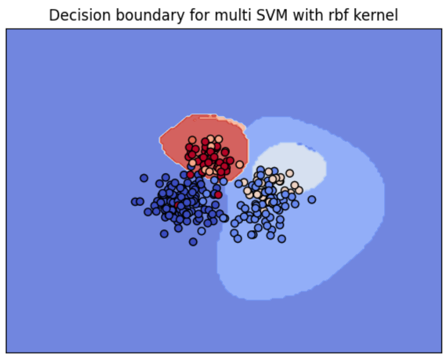
For the binary SVMs, we selected the classes *cp* and *im* to be the positive and negative classes respectively. We can see that the linear kernel has a linear decision boundary which is expected and the rest of the kernels have non-linear decision boundaries.



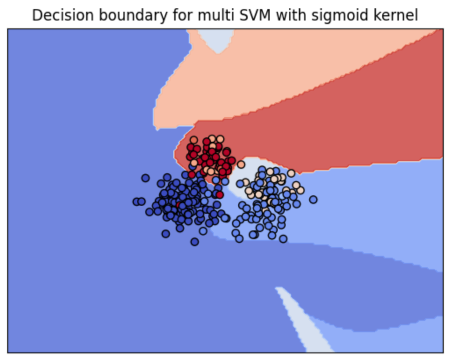
(a) Linear Multi-Class SVM



(b) Polynomial Multi-Class SVM



(c) RBF Multi-Class SVM



(d) sigmoid Multi-Class SVM

Figure 2: Multi-Class SVM Decision Boundaries

5 Conclusion

This paper presented a methodology for creating Support Vector Machine (SVM) kernels from scratch using Python and NumPy. We discussed the implementation of linear, sigmoid, polynomial, and radial basis function (RBF) kernels in a binary and multiclass SVM and how Binary SVMs are used underneath the hood of multiclass SVMs. These kernels were tested on E. coli data and compared to the results of the scikit-learn SVM implementation. The results show that the implemented kernels often yield better accuracy than the scikit-learn implementation; at the cost of increased training time. Kernel training times were ran multiple times and averaged to provide a more accurate metric as training times were low and had a high variance. By harnessing the power of dimensionality reduction, we were able to show the decision boundaries in 2 dimensions while still retaining the accuracy of higher dimensional data. These decision boundaries show how each kernel can affect the shape of the classification boundary and how the kernels can be used to classify non-linear data.