Implementation of SVM with Various Kernels and Comparison with SciKitLearn

Arya Amarnath, Trevor Overton, Lucas Pacheco

Introduction

For this project, we implemented the four most common kernels into our support vector machine and compared our results with the prebuilt package in SciKitLearn. The kernels implemented were linear, polynomial, radial basis function, and sigmoid. These kernels were chosen primarily because they are also the kernels found in the SciKitLearn support vector machine function, so we can compare the efficacy of our code. In our code, we elected to create a binary classifier function that does the actual calculations for the support vector machine, and from here we can do multigroup classification via several all-vs-one binary classifications to determine which option is best. After our testing, we were able to conclude that the Radial Basis Function and Linear kernels both performed the best in comparison to our polynomial and sigmoid kernels. In order to collaborate our coding efforts, we elected to use Google Colab.

For data management, we elected to remove the first column, use columns 1 through 7 as our data, and use our last column as our classifiers. This added the task of converting strings to integers, but this was solved via a built-in label encoder from SciKit. Since a few of the labels have very few entries, this means we will be unable to predict them effectively, however as the spirit of the project is in comparison of our classification and SciKit classification this should lead to interesting comparisons of minimal data prediction. One solution to allow us to predict more effectively would be to combine some labels to increase the sample size of the labels, however, this would lead to lower overall predictive performance. It is also worth it to mention that SciKit does one-vs-one multiclass classification whereas we elected to do one-vs-all

multiclass classification. As such, part of our comparison will also be seeing if one-vs-one classification is worth the extra computation cost compared to one-vs-all classification.

Kernels and Implementation

The four kernels we elected to include were linear, polynomial, radial basis function, and sigmoid. These kernels were chosen for their popularity and because they are what is used in SciKit. The reason behind implementing kernels is to project into a higher dimensional space than our data is actually in so that we can account for nonlinearity in the hyperplane we are using for separation. In code, this corresponds to changing the inner product at the beginning of a binary classifier. Different kernels can achieve this in different ways, and as such different kernels will create different features in our hyperplane.

The linear kernel is the 'basic' kernel to such a degree that they are what we learned in class originally in support vector machines. They technically are not even kernels, as we are not projecting into a higher dimensional space in order to do classification. The advantages are that they are easy to implement, cost the least computationally, and do not risk overfitting your data. The major disadvantage is that your data has to truly be split along a linear hyperplane, and a lot of data is not. The mathematical formulation is simply the inner product:

$$< x, x' >$$

Polynomial kernels create polynomials, and have the unique property of having degree as a hyperparameter. Since we can control how curvy our hyperplane is via the degree of a polynomial, we can fit any split between datasets perfectly eventually as is seen via Taylor's expansions. However, this creates the additional problem of overfitting becoming very possible. As such, we need to be careful to select a polynomial degree that is high enough to fit any eccentricities of the data but is low enough that we will not overfit to the training set. We also

add the hyperparameter of an intercept term, which is what allows for our Taylor Expansion-like fit to work. The mathematical formulation is

$$(\gamma < x, x' > + r)^d$$

where γ is a scaling constant, r is our intercept term, and d is our degree. As the polynomial function has 3 hyperparameters to tune, it can often be difficult to find the best fit

The Radial Basis Function is seen by many as the go-to kernel for any data we are given. It creates radial regions of similar data, leading to a fitting that appears to have large ellipses for classes. The RBF achieves this by creating regions of similarity for the data that we are trying to classify. It has the distinct advantage of projecting into infinitely higher dimensional space, and as such it will always be able to fit the eccentricities of the data. Since it creates regions of similarity, it has the added benefit of being more robust towards overfitting than polynomial kernels, and can even bimodal data. The mathematical formulation is

$$exp(-\gamma||x - x'||^2)$$

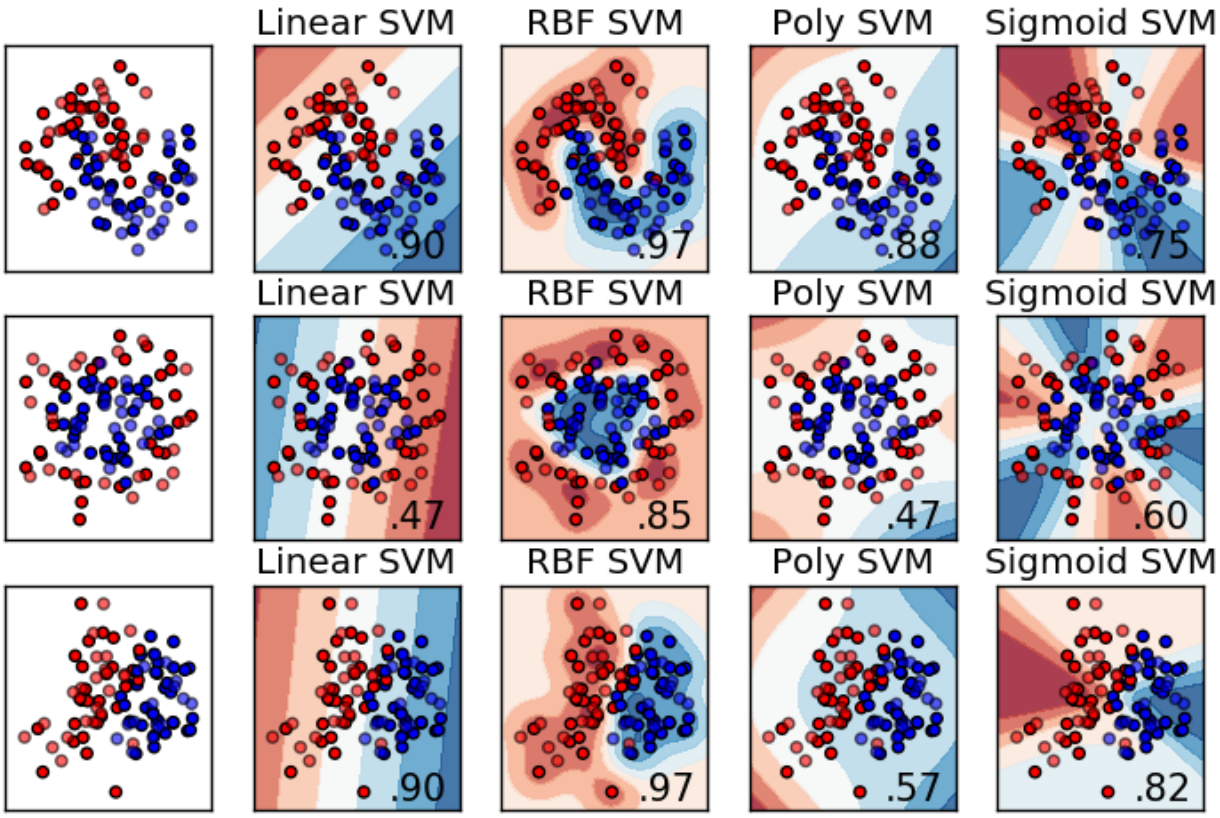
where γ is a constant. Since there is only one hyperparameter to tune, it is easier to find the best-fitting model compared to polynomial or sigmoid kernels.

The Sigmoid kernel has a very specific use case. It creates classification regions that look like hyperbolic tangent functions, and as such performs very well with data distributed that way. However, it does poorly for data of any other distribution. This is the least robust and most specialized of all the kernels we implemented. The mathematical formulation is

$$tanh(\gamma < x, x' > + r)$$

where γ and r are constants. Since there are two hyperparameters, it is not as easy to tune as the RBF, but easier than the polynomial.

There has been lots of work on visualizing these kernels. We are using one such example originally from WittmanF on GitHub



Reference: https://gist.github.com/WittmannF/60680723ed8dd0cb993051a7448f7805

This image demonstrates all 4 kernels on 3 arbitrary datasets, with accuracy in the bottom right corner. Between these 4 kernels, most data should be able to be fit with a high degree of accuracy. We can see that RBF performs very well consistently, whereas other kernels are much more dependent on the distribution. While none of the datasets fit the hyperbolic tangent dataset, it is easy to imagine data where the sigmoid kernel would perform best.

Binary Classification

In traditional support vector machines, we can only separate data into two groups. This is known as binary classification. For binary classification, support vector machines create a hyperplane in the space of our data that separates our two groups. This is done by having one group be above our hyperplane and another group below our hyperplane. Since the hyperplane is determined by an objective function being set to zero, this corresponds to our objective function being positive or negative. We generate our hyperplane using existing data, or a test set, and then we can use new data, or a training set, to check our accuracy. If our new data is positive, we classify it with the group above the hyperplane. If it is negative, we classify it with the group below the hyperplane.

From here, we could also consider the magnitude of our function to be our confidence in our classification. For example, if we return a value of .01, we would be not very confident that our data should be above the plane, whereas if we return a value of 1000 we are very confident our data should be above the plane. There is also the problem that sometimes there is not a clear hyperplane between the data. In order to make SVM more robust, we can then add in 'fuzzy' boundaries. This allows for misclassifications in our test set but penalizes them heavily.

Although we decided to focus on multiclass classification, we decided to run a binary classification for Cytoplasm or 'cp' within our dataset. This would mean that our response variable would have 1 for 'cp' and -1 for anything else. This was tested with all 4 kernels against both our custom SVM and SKLearn's SVC and we got these results for accuracy:

	SKLearn	Custom SVC
Linear	0.94	0.89
Polynomial	0.53	0.85
RBF	0.94	0.87

Sigmoid	0.94	0.88

Multiclass Classification

Unlike binary classification, which restricts data into two categories, multiclass classification in our project allows for categorizing data into more than two classes. Specifically, we segmented our data into eight distinct classes, corresponding to the different localization sites of E. coli. This approach broadens the classification spectrum, enabling the capture of more complex patterns in our dataset. However, it also presents challenges, particularly in datasets where class features significantly overlap or exhibit high variability within class distributions.

In our project, we implemented the "One-vs-All" classification strategy due to its simplicity and scalability. The "One-vs-All" approach involves transforming our multiclass dataset into multiple binary classification problems. Essentially we ended up creating a new binary classifier that distinguishes one unique class from the rest of them. This classifier distinguishes its respective class from all others. In the context of our project, this meant developing eight distinct classifiers, each representing a unique E. coli localization site. The process can be summarized as follows:

Classifier 1: If Class = 1 → label = 1, otherwise label= -1

Classifier 2: If Class = 2 → label = 1, otherwise label= -1

Repeated till Classifier 8...

Each of our eight classifiers is trained on the entire dataset, which allows us to aggregate the results and make a final prediction. In contrast, SciKitLearn and its Support Vector Classifiers end up using “One-vs-One” classification which slightly differs from “One-vs-All” and is much more computationally expensive. Similarly to “One-vs-All”, “One-vs-One” classification also splits our dataset into multiple binary classification problems, however, the

key difference is that a binary classifier is created for every pair of classes instead of each unique class. This approach, while potentially more accurate, significantly increases the number of classifiers, following the formula:

$$\frac{NumClasses * (NumClasses - 1)}{2}$$

This is the reason as to why utilizing "One-vs-One" is much more computationally expensive than the latter. Once we start getting to larger datasets, this method becomes slower and unmanageable due to the number of classifiers growing quadratically with the number of classes leading to a significantly increased computational time.

Overall, when comparing the two, "One-vs-All" is more efficient as it only requires training one classifier per class which provides clear interpretability and allows easy scalability with large datasets. It does have its drawbacks with balancing issues when certain classes are underrepresented which may lead to biased classes which was a concern for us. Certain E. coli localization sites such as omL, imL, and imS all had under 5 observations which were heavily underrepresented when compared to sites like cp and im with 143 and 77 observations respectively. This is where One-vs-One can hopefully alleviate some of the bias through its more complex data processing due to each classifier distinguishing between two classes. Focusing on just two classes, allows the classifier to focus on finding the optimal decision boundary between them. Having these two classes per classifier will likely decrease/reduce the class overlap and the imbalance caused by the uneven observations of our dataset. One-vs-one allows for better relationships between our features and performance with complex datasets while coming at a higher computational cost as highlighted by the formula.

Now that we have explained the difference between OvO and OvA, let's go into how our "One-vs-All" classification structure performed. For our hyperparameters, we didn't play around with them too much however we did end up changing the values for gamma and degree which prompted for better model performance. After we trained and fit our models, we obtained these accuracy metrics:

	SKLearn	Custom SVC
Linear	0.91	0.75
Polynomial	0.74	0.71
RBF	0.93	0.75
Sigmoid	0.91	0.75

We can see that we obtained slightly lower estimates overall when compared to our previous binary classification however these results are much more accurate and are actually tested against all 8 classifiers when compared to simply just one (testing for cp) from before. Our SKLearn RBF model performed the best as expected and the Linear/RBF models were the best from our custom SVC.

Conclusion

In conclusion, after thorough testing of binary and multiclass classification models for both our custom SVM code and SciKitLearn's SVC package, we can say that the multiclass SKLearn RBF kernel had the best accuracy and lowest overall error rate. This is most likely due to the RBF's robustness and flexibility. The only class where our code performed better was the multiclass polynomial, which was slightly unexpected. When we compare our two modes of classification, we can see that our binary classification was much easier to implement than multi-class classification. While our multiclass method performed much lower, it is difficult to

tell if that is due to one-vs-all classification or one-vs-one classification, or if SciKit does better hyperparameter tuning. Overall we were successful in implementing various kernels such as RBF, Polynomial, and sigmoid in our custom support vector machine code which was able to tell us that SciKitLearn is the more efficient SVM.