Implementing an SVM: A Shot in the Dark

Daniel Hanson, Sam Kreter, Brendan Marsh, and Christina R.S. Mosnick

***Abstract*—Handwriting recognition allows for handwritten documents to be stored in electronic form. Some benefits of having documents in electronic form include the reduction of space needed to store the documents and the ability to process the handwritten data within these documents. This paper proposes a method for processing the handwritten data using support vector machines (SVMs). The implementation uses 2-class classification SVMs with a multiclass method to provide a robust method for classifying handwritten characters.**

I. Introduction

OUR introduction paragraph goes here. This is just some sample text to fill up the space.

II. Implementation

Our SVM implementation first used a parser class to transform the input data. The parser is able to load and write numpy arrays of the testing and label data. To transform the label sets for training individual class dividers, the parser sets all labels that are of the current class being trained to 1, and to -1 for the rest of the classes. Next, we created a set of kernel functions to be passed into the svm instance dynamically, depending on the particular kernel needed. We packaged the training and prediction functionalities into instances of svm classes, which additionally store trained variables such as the support vectors and the bias. This allowed us to train and store multiple svms.

The SVM class is instantiated with the pre-processed sample (feature) vectors, their corresponding labels (1 or -1), the appropriate kernel function, and its necessary parameters (such as σ for an RBF function). In the training function, the support vectors are computed by finding the Gram Matrix, which stores all possible combinations of data points being passed into the kernel. We then solved the dual form found by lagrangian optimization (below) using cvxopt for our quadratic programming solver.

A. Dual Representation (from Eqn. 7.2 [1])

$$\mathbf{L} = \sum_N a_n - \frac{1}{2}\sum_n \sum_m a_n a_m t_n t_m \mathbf{K} \tag{1}$$

In order to use the CVXOPT solver we need to convert the Dual Representation into the standard quadratic programming form shown in equation 2.

B. Quadratic Programming Problem [3]

$$\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T\mathbf{P}\mathbf{x} - \mathbf{Q}^T\mathbf{x} \tag{2}$$

C. Parameters for Quadratic Programming

By transforming the dual respresentation we derived the parameters shown in equations 3 and 4.

$$P = \sum_n \sum_m t_n t_m \mathbf{K} \tag{3}$$

$$Q = \begin{pmatrix} -1 & -1 & -1 \\ -1 & -1 & -1 \\ -1 & -1 & -1 \end{pmatrix} \tag{4}$$

- Where Q's dimensions are determined by number of samples

D. Constraints

Next we changed the constraints into a form that is necessary for the quadratic programming solver as shown in equations 5, 6.

$$\mathbf{G}\mathbf{x} \preceq \mathbf{h} \tag{5}$$

$$\mathbf{A}\mathbf{x} \leq \mathbf{b} \tag{6}$$

Equation 7 is referencing the transformation for the standard constraints.

$$a_n \geq 0 \rightarrow -a_n \leq 0 \tag{7}$$

$$std\mathbf{G}\begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \tag{8}$$

$$std\mathbf{H}\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{9}$$

Equation 10 is referencing the transformation for the slack constraints.

$$a_n \leq \mathbf{C} \tag{10}$$

$$slack\mathbf{G}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{11}$$

$$slack\mathbf{H}\begin{pmatrix} \mathbf{C} & \mathbf{C} & \mathbf{C} \\ \mathbf{C} & \mathbf{C} & \mathbf{C} \\ \mathbf{C} & \mathbf{C} & \mathbf{C} \end{pmatrix} \tag{12}$$

Equation 13 is referencing the transformation for the equality constraints.

$$\sum_{n=1}^{N} a_n t_n = 0 \tag{13}$$

$$\mathbf{A} = \mathbf{y} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \tag{14}$$

$$\mathbf{B} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{15}$$

E. Bias

Solving for a gives us the indices for the support vectors. These along with the passed in data we now have the vectors, labels and weights of the support vectors we can use to find the bias and start computing predictions on the functions. The bias is simply computed from the form in equation 16.

$$b = \frac{1}{N_M} \sum_{n \in M} (t_n - \sum_{m \in S} a_m t_m k(\mathbf{x}_n, \mathbf{x}_m)) \tag{16}$$

F. Prediction

With this we have everything we need to implement the predict function from equation 17.

$$y(\mathbf{x}) = \sum_{n=1}^{N} a_n t_n k(\mathbf{x}, \mathbf{x}_n) + b \tag{17}$$

We also use this equation in finding the bias since they are essential the same equation if b is zero for equation 17.

After calling the training function with the data set, labels and kernel function we can now call the predict function passing in a test point and will return some value between 1 and -1 which we can use to classify the point.

Next we tackled the problem of classifying many classes with an SVM that only supports two class classification. To do this we use two different methods for comparison. In both cases we use a 1 vs the rest model in a bootstrapping method. We train a matrix of SVMs where the columns denote SVMs trained on different classes and the rows denote SVMs of the same class but different subsets of the training data. In the first case we take the mean of each set of SVMs for a particular class. The class with the highest overall mean will be used for the final label class. In the second case, we choose the classes with the highest results and find the variances of those results. The class with the least variance is chosen for the final label since the smaller variance shows a stronger confidence in the result.

Finally we have implemented pickling function to write and load the trained SVMs to a file in order to persist the state of the trained SVMs instead of having to train the SVMs on the same training data over and over again. This can be very beneficial for using larger bootstrapping samples or having more classes where it would take a large amount of time to train.

III. Experiments

After the SVMs were all trained using the bootstrapping method, we used a committee-waterfall approach to determine the best class for each test point. In order to do this, the SVMs are grouped by classifier, with 7 independently trained SVMs per each of the 8 classifiers. Each test point is run through each of the 7*8=56 SVMs. When committee results are gathered, if the point has less than 4 committee votes for each classifier, it is unclassified. If the point has 4 or more votes from just one classifier group, it is classified to that group. If the point has 4 or more votes from multiple classification committees, it is classified to the committee with the most votes, or in the event of a tie, to a random choice between the tie.

IV. Conclusion

Conclusion paragraph text goes here.

Acknowledgment

Christina would like to thank her mom for her support.

References

[1] Bishop, Christopher M. *Pattern Recognition And Machine Learning.* New York: Springer, 2006. Print.

[2] Tulloch, Andrew. *A Basic Soft-Margin Kernel SVM Implementation In Python* Tullo.ch. N.p., 2013. Web. 24 Mar. 2016.

[3] *Quadratic Programming With Python And CVXOPT. N.p., 2016. Web. 24 Mar. 2016.*

[4] *How To Calculate A Gaussian Kernel Effectively In Numpy.* Stats.stackexchange.com. N.p., 2016. Web. 24 Mar. 2016.

[5] *Scipy.Spatial.Distance.Pdist Scipy V0.17.0 Reference Guide.* Docs.scipy.org. N.p., 2016. Web. 24 Mar. 2016.