Implementation of a Support Vector Machine Using Sequential Minimal Optimization in C++ and a Two-Dimensional Visualization Package in R

Rui Ma, Shaocheng Wu, Mingyu Du

University of Michigan

Introduction

Machine learning, as the name may suggest, is the study of algorithms and models that computers learn and improve on their own to complete certain tasks that we assign to them. These tasks can be broadly divided into two groups: supervised and unsupervised learning. In supervised learning, users provide training data that contain both inputs and their corresponding labels/desired outputs for computers to build the optimal models. When the desired outputs are continuous, regression algorithms are typically incorporated into the models. Otherwise if the outputs are a set of finite discrete values, a classification algorithm is involved. On the other hand, unsupervised learning is concerned with data without outputs and the objectives normally involve data clustering and structure finding.

SVMs, short for Support Vector Machines, are some of the most well-known supervised machine learning models used for both classification and regression analyses. There are many incredible applications of SVM in the real world. Face detection, where n by n pixels and their corresponding binary facial recognition tags are taken as the training data, and cancer classification, where the training data consist of gene expressions and cancer labels, are some of the prominent examples.

SVM sure is powerful, however solving for the most optimal SVM model may not be straightforward. Successfully training an SVM model requires the solution to a complex quadratic programming (QP) problem. The sequential minimal optimization algorithm (SMO), developed by John Platt in 1998, helps break down the convoluted QP problem into series of much smaller ones, and thus vastly improves computational speed and reduces memory needed. It has since become one of the most popular solutions to the SVM optimization problem. Our project is an implementation of an SVM in C++ that solely focuses on classification, and it employs the SMO algorithm based on the pseudo code provided by John Platt in his original paper. To demonstrate the performance of the SMO algorithm and further visualize the SVM classification results in a two-dimensional space, we also include a self-developed R package (“Smo615”) that enables users to graph the training and test data along with the decision boundaries and margin.

Objectives and Methods

In order to understand the SMO algorithm, we must review how the SVM classifier functions and what the overall goal is. Suppose each training input (p-dimensional) belongs to one of two classes (e.g. "yes" or "no", corresponding to +1 or -1). Then given a new input, we want to classify whether it is a 1 or -1. In the simplest linear form, an SVM is a $(p-1)$-dimensional hyperplane that divides all inputs into two regions, where each region represents one of the classes. The most ideal hyperplane should maximize the distance between the two classes of data points, and thus we aim to maximize the margin – the distance of the hyperplane to the nearest data point on each side. To write the primal form in mathematical formulations, our goal is to

$$\text{Minimize } \frac{1}{2}\left|\left|\vec{w}\right|\right|^2 \text{ subject to } y_i(\vec{w}\cdot\overrightarrow{x_i}-b)\geq 1, \text{ for } i=1,\dots,n$$

where $\vec{w}$ is the normal vector to the hyperplane (i.e. the support vector), b is the bias term, $\overrightarrow{x_i}$ is each input data point, y_i is their corresponding label ($+1$ for the positive class and -1 for the negative class), and n is the size of the given inputs.

To further simplify the problem, we opt to transfer the primal problem above into the Lagrangian dual problem

$$\text{Maximize } \left(\sum \alpha_i - \frac{1}{2}\sum\sum y_i\alpha_i\left(\overrightarrow{x_i}\cdot\overrightarrow{x_j}\right)y_j\alpha_j\right) \text{ subject to } \sum \alpha_i y_i = 0 \text{ and } 0\leq\alpha_i\leq C, \text{ for } i,j=1,\dots,n \quad \textbf{(1)}$$

where α_i is the Lagrange multiplier associated with the i^{th} training input data, and C is the regularization parameter. A smaller C would introduce more regularization. With this duality form, the final SVM decision function becomes

$$\sum \alpha_i y_i \overrightarrow{x_i} - b \qquad \textbf{(2)}$$

where b here is the scalar bias term that will be updated when we train our model.

However, not all data can be separated by a linear decision boundary. Thus, we need to employ the kernel trick – we map a single vector to a vector of higher dimensionality, so that the model can learn a nonlinear classification rule. To be specific, we incorporated the radial basis function kernel, also known as the Gaussian kernel, in our C++ program

$$K(x,z)=\exp\left(-\frac{\left|\left|x-z\right|\right|^2}{2\sigma^2}\right)$$

where $\left|\left|x-z\right|\right|^2$ is the squared Euclidean distance between two vectors. In our project, if the user would like to use a linear kernel ($ind=0$), then the objective and decision function would be **(1)** and **(2)** from above written in C++, respectively. If the user would rather prefer the Gaussian kernel, then the objective and decision function would become

$$\sum \alpha_i - \frac{1}{2}\sum\sum y_i\alpha_i K\left(\overrightarrow{x_i},\overrightarrow{x_j}\right)y_j\alpha_j \text{ and } \sum \alpha_i y_i K(\overrightarrow{x_i},\vec{x}) - b \qquad \textbf{(3)}$$

Before we proceed to the next optimization step, we must normalize the training input data (included in our C++ code). Suppose we are given some input data that have two features and one is significantly greater than the other. Have we not normalized the data, the larger feature would dominate the other completely when we calculate the margin, thus making the SVM far less accurate.

Prior to John Platt's invention of the SMO algorithm, the chunking and Osuna's algorithms were the go-to methods to tackle the QP optimization problem described in **(1)**, **(2)**, and **(3)**. However, while solving for large-scale training problems, the matrices involved sometimes cannot even be stored in the memory, let alone having to deal with the numerical QP solver that introduces precision issues.

SMO works by breaking the SVM QP problem into the smallest possible optimization problem that only involves two Lagrange multipliers. The reason that the smallest possible problem is still concerned with at least two α's is that we still have to obey the linear equality constraint described in **(1)** – updating one α would require adjustment to another to keep the sum at zero. The main advantage that the SMO algorithm brings to the table is that solving for two Lagrange multipliers can be done analytically without utilizing the numerical QP solver. This results in a much shorter inner loop of the algorithm, the part that works out the optimization problem analytically, and thus largely improves the stability and the speed of the calculations compared to chunking. Additionally, SMO does not require any extra matrix storage outside of the kernel computations, making this SVM implementation very feasible on any ordinary personal computer.

The final Lagrange multiplier vector after the training process should be mostly zeros except where the corresponding training inputs are exactly on or near the eventual decision margins. This implies that the final decision boundary is widely dependent on those and the ones near those training examples. Thus, adding more inputs that are far from the decision boundary to the training set should not alter the results much.

In our C++ code (*Smo615_c.cpp*), we have a class named *SMOModel*. A valid *SMOModel* object would have the following values: training inputs *X* as an MatrixXd, training labels *y* as a VectorXd, the regularization parameter *C* as a double, the Lagrange multipliers *alphas* also as a VectorXd, the scalar bias *b* as a double, the error cache *errors* as a VectorXd, and the indicator *ind* as an int (0 or 1). The class also includes the core of the SMO algorithm: a void *train()* function, an *examine_example(int i2)* function that returns an integer, a *take_step(int i1, int i2)* function that also returns an integer, and a *predict(MatrixXd test)* function that returns the prediction results as a VectorXd.

According to Osuna's theorem, as long as SMO alters the two Lagrange multipliers at every step and at least one violates the Karush-Kuhn-Tucker (KKT) conditions, convergence is bound to occur. In order to speed up the process, SMO uses heuristics to select which two multipliers to optimize. The choice of the first Lagrange multiplier is determined by the *train()* function, which is essentially the outer loop of the

SMO algorithm. It iterates over the full training set and picks the inputs that violate the KKT conditions for optimization in the *examine_example()* function. After each full iteration, the outer loop goes over all non-bound examples (multipliers that are neither 0 nor *C*) and once again picks the ones that violate the KKT conditions for optimization. The loop keeps repeating these procedures – going over all training inputs and non-bound examples – until none of the inputs breach the condition, which is when the training process terminates. *Examine_example()* chooses the second Lagrange multiplier that maximizes the difference in error evaluated in the optimization step, the *take_step()* function. Under special circumstances where positive progress cannot be achieved, the algorithm would loop through the error cache at random points in order to proceed.

Inside the *take_step()* function, lower and higher bounds are first calculated for the second Lagrange multiplier before computing the second derivative of the objective function in **(3)**

$$K(\overrightarrow{x_1}, \overrightarrow{x_1}) + K(\overrightarrow{x_2}, \overrightarrow{x_2}) - 2K(\overrightarrow{x_1}, \overrightarrow{x_2})$$

In general, the objective function is positive definite, and the value above would be positive. In other cases, the bounds and the objective function would have slightly different forms ensuring that the SMO algorithm can move along. The threshold *b* and the errors corresponding to the two Lagrange multipliers' indices are also updated in this function.

Since the Lagrange multipliers are embedded in the *SMOModel* class, once the training process is completed, the optimal multipliers can then be accessed from the *alphas* vector. These multipliers are essential to the prediction/classification (handled by the *predict()* function) of the test samples that the user supplies. If the linear kernel is chosen, the vector that determines the predicted labels for the corresponding test inputs is

$$\overrightarrow{x_{test}} \sum y_i \alpha_i \overrightarrow{x_i} - b$$

, and

$$\sum y_i \alpha_i K(\overrightarrow{x_i}, \overrightarrow{x_{test}}) - b$$

if the Gaussian kernel is selected instead. If the i^{th} element of the resultant vector above is greater than 0, then the i^{th} test sample would be classified as a *1*. Conversely, if a vector element is less than 0, the matching test input would be classified as a *-1*. Now suppose the user also provides the actual test labels. Our program would spit out an accuracy rate, the ratio of the number of correctly classified test samples to the size of the entire test dataset.

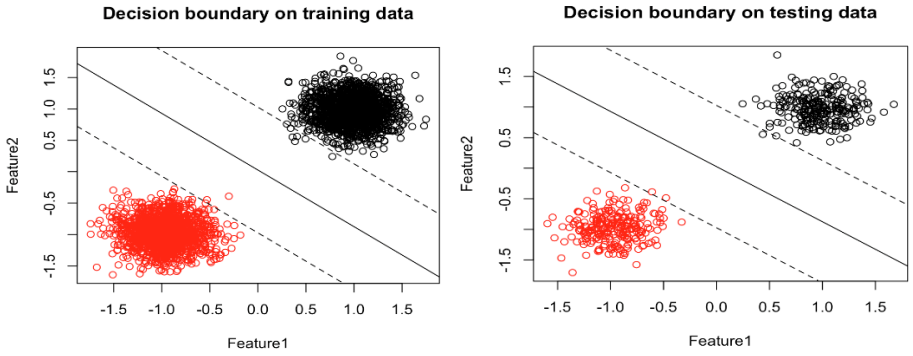
In order to visualize the final SVM classification results in a 2D space in R, without using any dimension reduction techniques, we must select inputs and test data that only contain two features. Furthermore, to interface our C++ code with R, we have a second version of the C++ program that has the *main* function modified to an *extern "C"* function, included in the *smo_package* folder. Using the *dyn.load()* and *.C()* functions, we could then dynamically link the C++ code to R and successfully make a call to C from R.

Results

To test our linear SVM model, we simulated some isotropic Gaussian blobs using scikit-learn's *make_blobs()* function in python, making sure that the data generated can be separated by a linear decision boundary. Specifically, we generated 3000 training samples and 500 test cases with two features and two centers. The prediction results, measured by the accuracy rate, is at exactly 1 using our linear kernel ($C = 10.0, ind = 0$), meaning that none of the 500 test cases were classified incorrectly.

```
luigi% ./Smo615 data/sample_data.txt data/sample_label.txt data/test_data.txt data/test_label.txt
10.0 0 0
Accuracy: 1
```

To visualize the performance of our linear SVM model in a 2D space in R, we first loaded our "Smo615" package in R. The graphs below are the results after the user initializes the corresponding *smo615()* model:

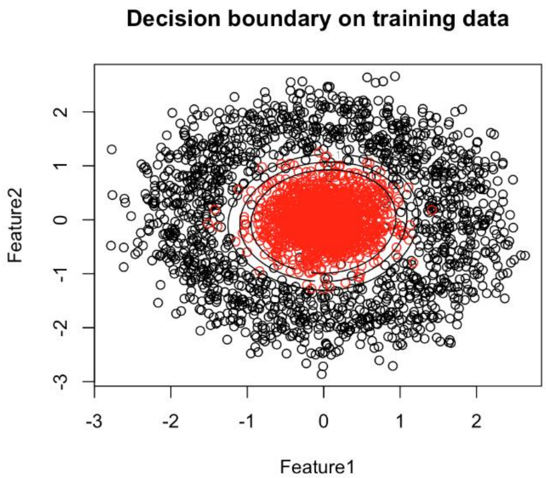


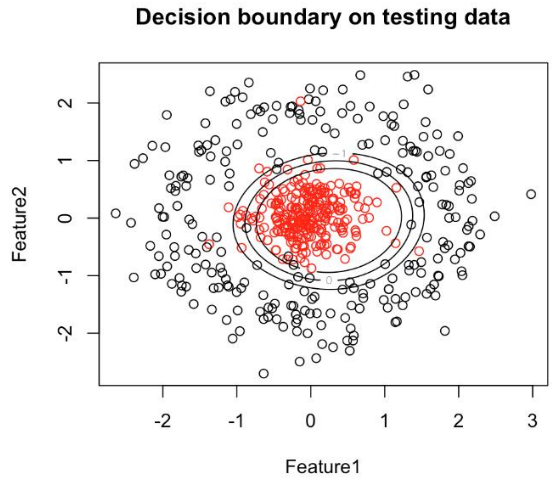
Which agree with our prediction accuracy rate for the linear SVM model mentioned above.

As for the nonlinear SVM case, we generated some toy datasets using the *make_circles()* function from the scikit-learn library in python as well. The training dataset has 3000 samples that are essentially a "large circle containing a smaller circle in 2D" with factor and noise set to 0.1 and 0.2, respectively. The accuracy for this case is 0.97 after opting for the Gaussian kernel ($C = 1.0, ind = 1$), meaning $15/500$ test cases were classified incorrectly.

```
luigi% ./Smo615 data/circle_train_data.txt data/circle_train_label.txt data/circle_test_data.txt data/circle_test_label.txt
1.0 1 0
Accuracy: 0.97
```

The graphs below are the results in R for this case after the user initializes the corresponding *smo615()* model:





Notice that in the command-line screenshots above there is a third argument for the input parameters (*cin*). If the user chooses to enter 1, in addition to the accuracy rate, the output would also include the resultant classified labels for the corresponding test data.

Discussion

There are several existing comprehensive libraries designed specifically for training SVM models. LIBSVM, one of the most well-known SVM libraries, includes an R package "e1071" that supports both linear and kernel SVM. This package is also written in C++ and employs an SMO-type algorithm. Other than "e1071", "caret" is another package that does classification and regression training and is written in R instead. To compare the performance of our program with the two mentioned above, we trained three kernel SVM models with the linearly inseparable data (used above). The results below present the time taken (in milliseconds) to train each model 100 times (in the order of: ours, "e1071", "caret").

```
Unit: milliseconds
                                                                                                                     expr
         smo615("circle_train_data.txt", "circle_train_label.txt", "circle_test_data.txt",      "circle_test_label.txt", 1, 1)
     e615("circle_train_data.txt", "circle_train_label.txt", "circle_test_data.txt",      "circle_test_label.txt", "radial", 1)
 caret615("circle_train_data.txt", "circle_train_label.txt", "circle_test_data.txt",      "circle_test_label.txt", "svmRadial")
        min         lq       mean     median         uq        max neval cld
  9289.2094  9345.6984  9455.1498  9382.0814  9502.6384  9917.9935   100  b
   322.7549   346.9492   350.8369   349.7135   353.4245   392.8215   100 a
 12183.3589 12258.7996 12369.8260 12301.6929 12423.9188 13499.8784   100   c
```

We see that even though it took much longer for our implementation to train the Gaussian SVM model than the "e1071" package, our "Smo615" package still seems more efficient than the "caret" package. These time comparison results were somewhat expected. Due to the structural differences in the two programming languages, it is not surprising that our implementation has the upper hand over the "caret" version. There are a few plausible reasons as to why ours is substantially slower than the "e1071" implementation. Firstly, the time complexity of our kernel computation is $O(n^3)$, which could potentially be further optimized. Secondly, to read in the training and test matrices from text files, our program relies on the "ReadMatrix615.h" header file introduced in class, whereas "e1071" is free of this complication.

References

1. Platt, J.: Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machine. Technical Report MSR-TR-98-14. Microsoft research (1998)
2. Wikipedia contributors. Support Vector Machine. Wikipedia, The Free Encyclopedia. Available at: https://en.wikipedia.org/wiki/Support_vector_machine. Accessed December 1, 2018.
3. Charest, J. (n.d.). Implementing a Support Vector Machine using Sequential Minimal Optimization and Python 3.5. Retrieved from https://jonchar.net/notebooks/SVM/
4. Wikipedia contributors. Machine Learning. Wikipedia, The Free Encyclopedia. Available at: https://en.wikipedia.org/wiki/Machine_learning. Accessed December 1, 2018.
5. Wikipedia contributors. Sequential Minimal Optimization. Wikipedia, The Free Encyclopedia. Available at: https://en.wikipedia.org/wiki/Sequential_minimal_optimization. Accessed December 1, 2018.
6. Scikit-learn Developers. (n.d.). Sklearn.datasets.make_circles. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_circles.html
7. Scikit-learn Developers. (n.d.). Sklearn.datasets.make_blobs. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html
8. C.-C. Chang and C.-J. Lin. LIBSVM: a library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2:27:1—27:27, 2011.