CAP6610 - Machine Learning Course Project

Written by **Jennifer Cheung**, **Joshua Kirstein**, **Abhishek Kumar**, **Abhishek Mohanty**, and **Abhinav Rathi**.

Installation

To run the code samples in this project, python and some extra libraries are required. To install the libraries run the following commands:

```
sudo pip install pandas
sudo pip install keras
sudo pip install cvxopt
sudo easy_install --upgrade six
```

Usage

Simply call

```
python main.py
```

or

```
python kernel_svm_test.py
```

NOTE: A sequence of plots will be shown. In order to progress through this sequence, you must close the plot that's currently being displayed.

Multi-Class Kernel SVM

A multi-class kernel support-vector machine was implemented. We build a multi-class kernel SVM as K two-class kernel SVMs, using the one-against-all schema to classify an incoming pattern x (that is, choose the class k who's two-class kernel prediction gives the highest value). A two-class kernel SVM trained with N patterns is constructed by solving the following convex quadratic programming optimization task (using CVXOPT):

$$\begin{aligned}
\underset{\{\lambda_n\}}{\text{maximize}} \quad & \sum_{i=1}^{N} \lambda_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \lambda_i\lambda_j y_i y_j \kappa(x_i, x_j) \\
\text{subject to} \quad & 0 \le \lambda_i \le C,\ i = 1, \ldots, N. \\
& \sum_{i=1}^{N} \lambda_i y_i = 0
\end{aligned}$$

where C is a constant that helps with outlier rejection and $\kappa(x, y)$ is the kernel function. Once we have $\{\lambda_n\}$ we compute θ_0 as the average of the following values for each support vector i :

$$\theta_{0,i} = \frac{1}{y_i} - \sum_{j=1}^{N} \lambda_j y_j \kappa(x_j, x_i)$$

Finally, to classify an incoming pattern x we simply compute

$$y(x) = \sum_{i=1}^{N} \lambda_i y_i \kappa(x_i, x) + \theta_0$$

Limitations

There are several limitations to this implementation. First of all, the kernel implementation of an SVM is inherently slower due to the kernel computations. Specifically in our implementation, we must compute the Gram matrix for the kernel over the samples — a computation which is quadratic in the number of patterns. Classifying patterns is similarly deficient; looking at the formula for prediction above, we must compute N kernel values. Varying the kernel changes the performance of the algorithm. Our choice of using python (an interpreted language) also slows down these computations greatly.

Accuracy does not seem to be hindered from this implementation. We use an industrial strength convex optimizer (CVXOPT), which gives much better results than the suggested majorized algorithm. Compared to *libsvm*, our implementation of a multi-class kernel SVM is more accurate (albeit *significantly* slower). Using a Gaussian kernel provided the most accurate classification whilst being the slowest. Using a Linear kernel provided mildly accurate classification whilst being the fastest.

Testing

We ran the multi-class kernel SVM implementation on multiple data sets. On the handwritten digit data set, our implementation takes a very long time to run (for train percentage less than 0.7 it runs in a moderate amount of time). To see the results from the other data sets, run the code.

To show that our implementation was indeed kernalized, we ran a visual experiment using two-class Gaussian data. The blue and green points are the data colored by their classes. The points in red describe points that are very close to the splitting surface generated by the two-class kernel SVM (using a Gaussian kernel with $\sigma = 20$), thus effectively plotting the splitting surface.

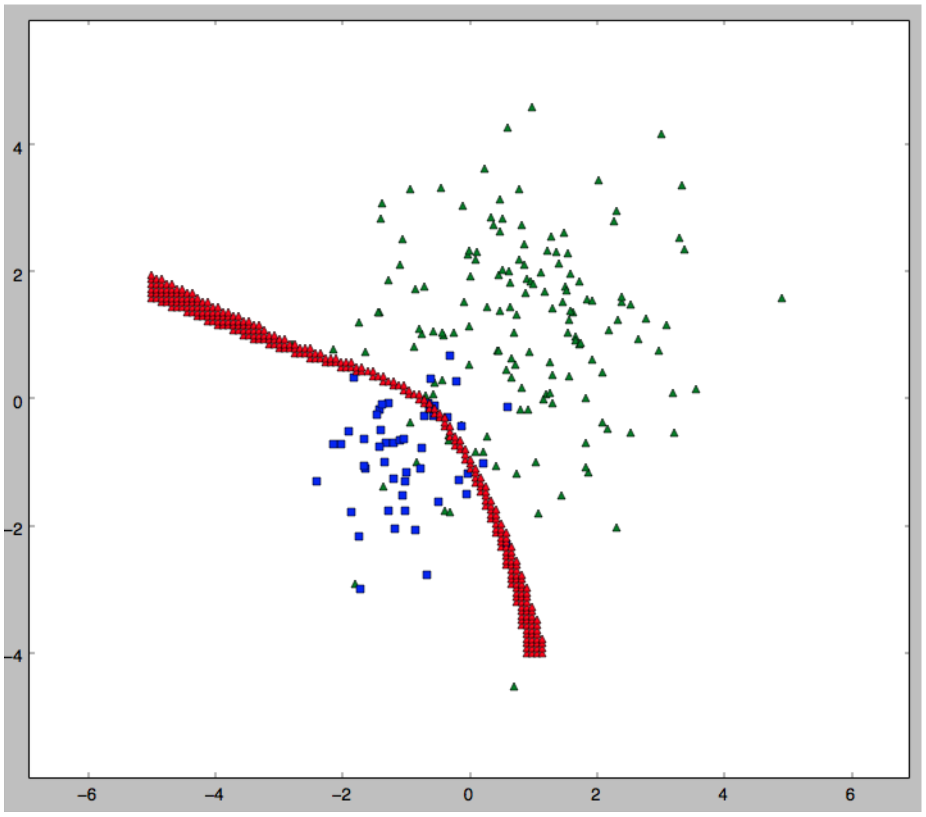

Figure 1: kernel visualization

Algorithms Demonstrated

Neural Network

Random Forests

Support Vector Machine

Testing Methodology