University of Colorado Boulder

APPM 5720

Convex Optimization

A Review of Support Vector Clustering

Grant Baker
J. Matthew Maierhofer

December 10, 2018

Abstract

In this project, we explore a clustering method motivated by the ubiquitous Support Vector Machine, called Support Vector Clustering. We examine the original technique and the reasons why it works and why it fits into the context of optimization. We also discuss the technique's shortcomings and explore two additional algorithms that provide a speed increase when used in practice. Additionally, we develop a Python implementation of Support Vector Clustering, and run it on classic clustering datasets such as two moons and distorted blobs.

Contents

1 Introduction **2**

2 Support Vector Clustering **2**
2.1 Hypersphere Determination 2
2.2 Cluster Determination 6

3 Speed Improvements **6**
3.1 Efficient Cluster Determination 6
3.2 Efficient Boundary Computation 7

4 Results **8**

5 Conclusion **12**

A Code Appendix **12**

1 Introduction

Support Vector Clustering, or SVC, is a clustering algorithm created by Ben-Hur et al. in 2001[BHHSV01]. The algorithm was developed as an extension of Support Vector Machine algorithms, which are used extensively in statistics, data science, and machine learning. SVC relies on Support Vector Machine concepts, but operates in an unsupervised setting whereas Support Vector Machines rely on labelled data.

The vanilla algorithm maps the input data to a high-dimensional feature space and finds the hypersphere of smallest radius that contains all of these points. There is also the ability to introduce slack into this hypersphere's boundary by allowing points to exist outside. Note that this is similar, conceptually, to a Support Vector Machine with all points labelled the same. This hypersphere becomes a decision boundary.

The boundary is then mapped back into our data space. Clusters are determined by the following schema: if two data points have a line between them that is entirely contained within the hypersphere in the feature space, they are determined to be part of the same cluster.

In practice, we employ the *kernel trick*, which allows the use of an inner product instead of a mapping. The most practical example of this is the use of the Gaussian kernel, whose feature space is infinite dimensional but the inner product is easily computed.

We will go over the mathematics of this a bit more in the next section.

2 Support Vector Clustering

The SVC algorithm can be broken down into two major parts: hypersphere determination and cluter determination. In the first phase, given a dataset, $X \in \mathbb{R}^{n \times m}$ with n data points in m dimensions, we seek to find the smallest hypersphere containing the data points, possibly not including some boundary points. In the second phase, we determine which of these points must belong to the same cluster.

2.1 Hypersphere Determination

First, we will introduce the problem we need to solve to find our radius.

$$\begin{aligned} &\min_{R,a} R^2 \\ ||\phi(X_i) - a||^2 \leq &R^2 \ \forall i = 1, 2, .., n \end{aligned} \tag{1}$$

In this problem, R is the radius and a is the center of the hypersphere in the feature space. Conveniently, this problem is a convex minimization!

We do note, however, that this formulation of the problem does not allow for points outside the boundary. In many situations, we want to allow points outside of the boundary in the data, so we must expand our approach.

We introduce slack variables ξ in order to allow for the softening of the constraint. We then linearly penalize this constraint with constant C.

$$\begin{aligned} &\min_{R,a,\xi} \ R^2 + C\sum_j \xi_j \\ ||\phi(X_i) - a||^2 \leq &R^2 + \xi_i \ \forall i = 1, 2, .., n \\ &\xi_i \geq 0 \ \forall i = 1, 2, ..., n \end{aligned} \tag{2}$$

We also note that equation 2 becomes equation 1 as $C \to \infty$, and that this problem is also convex.

The primal problem (2) yields the Lagrangian

$$\mathcal{L}(R, a, \xi, \beta, \mu) = R^2 + C\sum_j \xi_j - \sum_j \beta_j(R^2 + \xi_j - ||\phi(X_j) - a||^2) - \sum_j \xi_j\mu_j \tag{3}$$

where the dual variables $\beta, \mu \geq 0$. The primal problem also satisfies Slater's conditions, since for any a and R, we can choose a non-negative ξ such that our constraints are strictly satisfied. As a result we can use the KKT conditions to gather information about our solution.

First, via stationarity, we know $\nabla_R \mathcal{L} = 0$. Therefore,

$$\begin{aligned} 2R &= \sum_j 2\beta_j R \\ \sum_j \beta_j &= 1 \end{aligned} \tag{4}$$

Similarly, $\nabla_a \mathcal{L} = 0$, so

$$\begin{aligned} \nabla_a \mathcal{L} &= \nabla_a \sum_j \beta_j(-||\phi(X_j) - a||^2) = 0 \\ 2a \sum_j \beta_j &= 2 \sum_j \beta_j \phi(X_j) \\ a &= \sum_j \beta_j \phi(X_j) \end{aligned} \tag{5}$$

Finally, stationarity on ξ yields

$$\begin{aligned} \nabla_\xi \mathcal{L} &= C\mathbb{1} - \beta - \mu \\ \beta_j &= C - \mu_j \end{aligned} \tag{6}$$

We also consider the complementary slackness conditions from KKT

$$\begin{aligned} \beta_j(R^2 + \xi_j - ||\phi(X_j) - a||^2) &= 0 \\ \xi_j \mu_j &= 0 \end{aligned} \tag{7}$$

At this point, we define Support Vectors (SVs) as data points that lie on the boundary in the feature space, Bounded Support Vectors (BSVs) as data points that are outside of the boundary, and Interior Data Points (IDPs) for those strictly inside the boundary.

For BSVs, $\xi_j > 0$, which gives $\mu_j = 0$ and $\beta_j = C$ from the above conditions. For IDPs, $\xi_j = 0$, and it is interior to the radius, so it must be that $\beta_j = 0$. Therefore, if $0 < \beta_j < C$, X_j must be a SV.

Because the problem is continuously differentiable and convex,

$$\max_{\mu,\beta} \left(\min_{R,a,\xi} \left(\mathcal{L}(R, a, \xi, \beta, \mu) \right) \right)$$

is equivalent to

$$\begin{aligned} \max_{\mu,\beta,R,a,\xi} \mathcal{L}(R, a, \xi, \beta, \mu) \\ \nabla_{R,a,\xi} \mathcal{L} = 0 \end{aligned} \tag{8}$$

Equations 4, 5, and 6 are used to reduce the new problem. Let $\hat{L}(\beta) = \mathcal{L}$ under our stationarity conditions.

$$\begin{aligned}
\hat{L}(\beta) &= R^2 + C\sum_j \xi_j - \sum_j \beta_j(R^2 + \xi_j - ||\phi(X_j) - a||^2) - \sum_j \xi_j\mu_j \\
&= R^2(1 - \sum_j \beta_j) + \sum_j \xi_j(C - \beta_j - \mu_j) + \sum_j \beta_j(||\phi(X_j) - a||^2) \\
&= \sum_j \beta_j||\phi(X_j) - a||^2 \\
&= \sum_j \beta_j(||\phi(X_j)||^2 - 2a\cdot\phi(X_j) + ||a||^2) \\
&= \sum_j \beta_j\Big(||\phi(X_j)||^2 - 2\sum_i \beta_i\phi(X_i)\cdot\phi(X_j)\Big) + ||\sum_i \beta_i\phi(X_i)||^2 \\
&= \sum_j \beta_j\phi(X_j)\cdot\phi(X_j) - \sum_{i,j}\beta_i\beta_j\phi(X_i)\cdot\phi(X_j) \qquad (9)
\end{aligned}$$

It is now important to address $\phi(x)$. This function maps the data to a high dimensional feature space, potentially even infinite-dimensional. We make use of the *kernel trick*. While it is clear that sometimes we cannot actually compute this mapping, we don't have to! $\hat{L}(\beta)$ only depends on the inner products of the elements of the feature space. If we define this inner product, usually called the kernel function, we don't actually need to know $\phi(x)$! The kernel function $K(x_1, x_2)$ is defined as

$$K(x_1, x_2) = \phi(x_1)\cdot\phi(x_2)$$

For our project, we use the ubiquitous Gaussian kernel,

$$K(x_1, x_2) = e^{-q||x_1 - x_2||_2^2}$$

with some hyperparameter q. This gives us the dual problem

$$\max_\beta \sum_j \beta_j K(X_j, X_j) - \sum_{i,j}\beta_i\beta_j K(X_i, X_j) \qquad (10)$$

which fits into the known quadratic programming framework, and can be solved by a number of commonly-used methods.

The solution to (10) is used to develop a map from any point x in our data space to a scalar radius from the center of our hypersphere. We denote the map as $R^2(x)$, and it can be written as

$$R^2(x) = ||\phi(x) - a||^2 = K(x,x) - 2\sum_j \beta_j K(x, X_j) + \sum_{i,j} \beta_i \beta_j K(X_i, X_j) \qquad (11)$$

We use this to determine the radius of our sphere by plugging in an SV.

2.2 Cluster Determination

In the second phase of the algorithm, we assign cluster labels to the points. In the vanilla algorithm, it is necessary to check all pairs of data points to determine the adjacency matrix A, which defines a graph in which the nodes are data points. Two data points are considered adjacent if the line between them in the data space is fully contained in the sphere in the feature space. Formally, points X_i and X_j are connected if

$$||\phi(tX_1 + (1-t)X_2) - a||^2 \leq R^2, \ \forall \ t \in (0,1)$$

Practically, we check a sample of the points on the line.

Finally, to assign clusters, each connected set in the graph determined by the adjacency matrix A is a cluster.

3 Speed Improvements

While the vanilla algorithm works, it is not fast. Both phases of the algorithm allow opportunities for speed increases.

3.1 Efficient Cluster Determination

One immediately apparent issue with this is that the cost of the algorithm is extreme for large datasets. Just the adjacency matrix alone is $n \times n$, which for a 1000 element dataset has 1,000,000 elements and might test memory constraints.

In particular, this problem is addressed by Pham, Dang, Le, et al. in [PDLL17]. The most significant speed penalty comes from forming the adjacency matrix.

It is most useful to know about nodes on the boundaries of the clusters. These points, called epsilon points, are within ϵ of the boundary of the hypersphere in feature space, representative of the boundaries of the clusters. Without loss of generality, we can assume the radius is 1 (otherwise we can adjust the weights appropriately). Then, the decision function becomes

$$f(x) = \sum_j \alpha_j K(X_j, x) - 1$$

for weights α_j. We want to look for points where the gradient of f is 0, so for the Gaussian kernel, $\nabla f(x) = 0$ becomes

$$0 = 2\sum_j \alpha_j (X_j - x) e^{-q||X_j - x||^2}$$

$$x = \frac{\sum_j \alpha_j e^{-q||X_j - x||^2} X_j}{\sum_j \alpha_j e^{-q||X_j - x||^2}} = P(x) \tag{12}$$

Equation 12 yields a fixed-point iteration $x_{k+1} = P(x_k)$ to find the zeros of the decision function gradient, which are called equilibrium points. We begin the iteration at each epsilon point to compute a map from each epsilon point to an equilibrium point. These points tend to be near centers of clusters.

Once the equilibrium points are found, it is sufficient to compute the adjacency graph A only on the equilibrium points rather than on the whole dataset.

The clusters are determined by the connected components of the graph A of equilibrium points. Each epsilon point is assigned to the same cluster as its equilibrium point. Each other point is assigned to the same cluster as the nearest epsilon point.

The advantage of this method over the original is that the number of equilibrium points tend to be much smaller than the number of data points, so the adjacency matrix is much smaller, can fit in memory, and it much quicker to search for connected components.

3.2 Efficient Boundary Computation

Computing the boundary function

$$f(x) = \sum_j \alpha_j K(X_j, x) - 1$$

requires knowledge of all of the α_j. Moreover, there are the same number of α_j as data points, and most points don't influence the model substantially.

We fit the model into the Stochastic Gradient Descent (SGD) framework, and limit the number of nonzero α_j to some budget B [PLD17].

For each SGD step, sample one data point X_i at random and set

$$\alpha_j^{(t+1)} = \frac{t-1}{t}\alpha_j^{(t)} \ \forall j$$

We then compute for α_i, the weight corresponding to the sampled point X_i, if X_i lies outside the boundary, i.e. if

$$f(X_i) = \sum_j \alpha_j K(X_j, X_i) - 1 < 0$$

then we update α_i according to

$$\alpha_i^{(t+1)} = \alpha_i^{(t)} + C\eta_t$$

where η_t is an appropriate learning rate.

To employ a budgeting strategy, at each step if there are more nonzero α_j than the budget B, we find the index p_t such that α_{p_t} is the smallest nonzero α_j. Then, set $\alpha_{p_t} = 0$ and continue.

The advantages of this method are clear when the amount of data is large, so it might not fit into memory. Further, each nonzero α_j increases computation time so limiting this number speeds up the algorithm with little impact on results.

4 Results

In attempting to test the validity of our model, we ran the algorithm on several classic datasets for testing clustering. Two classic datasets on which we tested the algorithm were Two Moons and transformed blob datasets.

First, we run the k-means classical clustering algorithm, and the results in Figure 1 indicate that k-means fails to solve the clustering problem on Two Moons.

Overall, the clustering is usually fairly successful with some careful tuning of hyperparameters. Models tend to be very sensitive to changes in hyperparameters, with some small changes producing extreme differences in behavior.

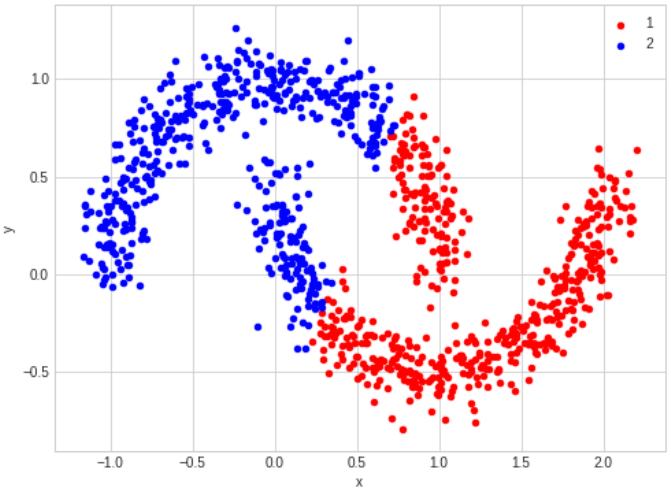

Figure 1: k-means results. The classical algorithm entirely fails at the problem.

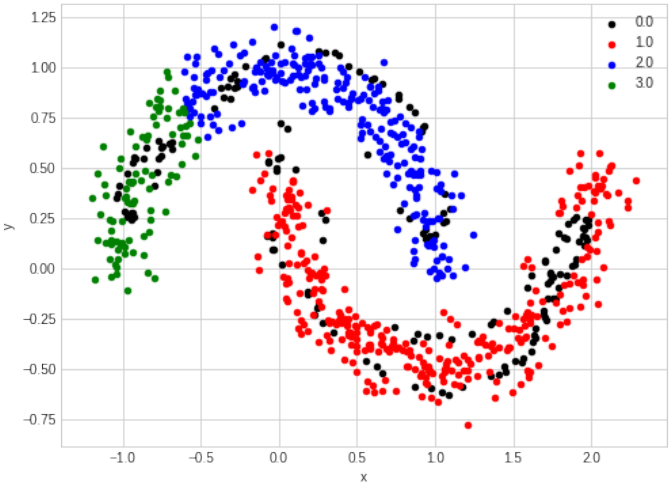

Figure 2: Two Moons example clustering. We see that the SVC algorithm produces three clusters, one of which seems correct and the other two could be combined to be correct. The black points here are the epsilon points, which indicate the boundary of the hypersphere in data space.

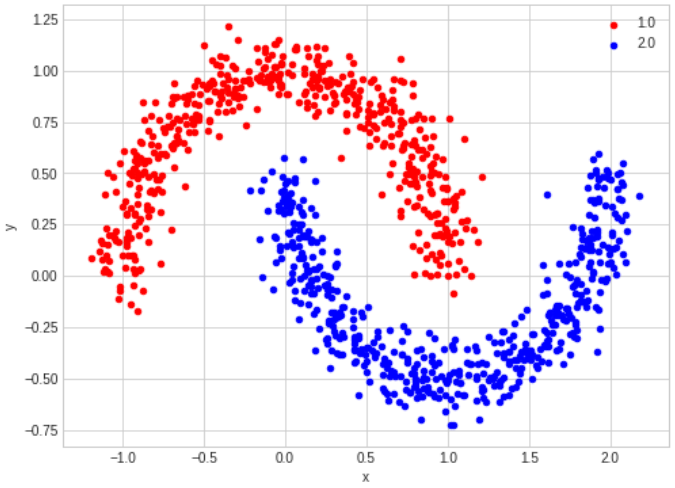

Figure 3: Two Moons optimal clustering. SVC takes careful tuning to produce correct results.

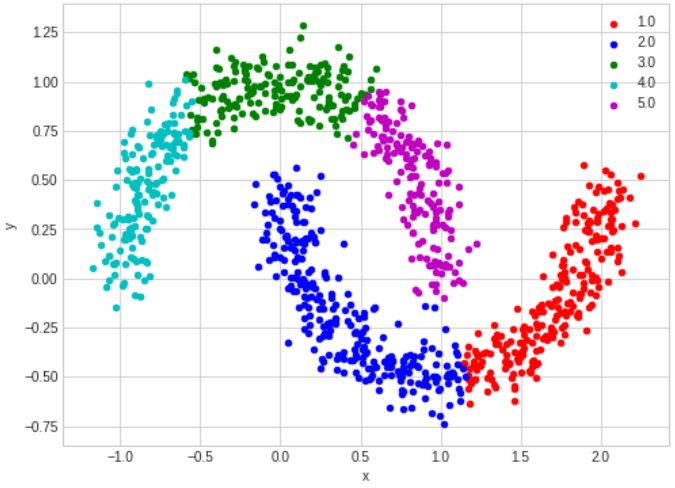

Figure 4: Two Moons suboptimal clustering. In this case, the number of clusters is too high, which suggests that the parameter q is too high.

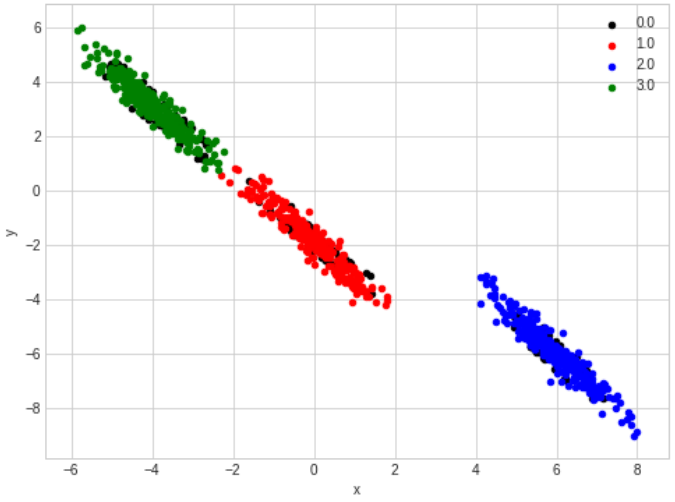

Figure 5: Distorted Blobs. SVC produced good results here.

A successful clustering is seen in Figure 3. One failure we frequently noticed involved the generated equilibrium points. Because there were frequently only a handful of equilibrium points, in a dataset with a non-convex cluster, there was a decent chance that line between some equilibrium points within the same cluster was not contained within the cluster.

An example of this can be seen in Figure 2. The black points are our labelled boundary points. Note that these exclude points farther than epsilon outside the boundary. While the SVC does fail to correctly cluster the upper moon entirely together due to the prior mentioned issue, it still deals well with the difficult task of correctly separating the region where they overlap. This task is nearly impossible for a more naive clustering algorithm, like k-Means, that only relies on a single point to define distance for clusters. Another common failure creates too small or too large of a hypersphere. The latter results in every point being classified as the same cluster, and the former results in many different clusters, as seen in Figure 4.

We then tested on the blob dataset. In this set, clusters tend to be more convex. As a result, the SVC algorithm doesn't frequently struggle with this dataset, unless clusters particularly overlap. An example of the algorithm run on this dataset can be seen in Figure 5.

5 Conclusion

Overall, the SVC algorithm is a powerful clustering tool with careful hyperparameter selection and not useful without. It produces better results than the classic k-means, and sometimes produces better results than other methods like spectral clustering. Its fit into standard convex optimization frameworks and stochastic gradient frameworks makes it easy to understand what it is doing and why. Finally, it can be run relatively quickly (when compared to other clustering algorithms) with specific details in implementation.

A Code Appendix

All of the algorithms in this paper were implemented by the authors in Python 3, and are available at `https://github.com/grantbaker/support-vector-clustering`.

References

[BHHSV01] Asa Ben-Hur, David Horn, Hava Siegelmann, and Vladimir Vapnik. Support vector clustering, 2001.

[PDLL17] Tung Pham, Hang Dang, Trung Le, and Thai Hoang Le. Fast support vector clustering. *Vietnam Journal of Computer Science*, 4(1):13–21, Feb 2017.

[PLD17] Tung Pham, Trung Le, and Hang Dang. Scalable support vector clustering using budget. *CoRR*, abs/1709.06444, 2017.