Sample Selection Methods for SVM

EE236A: Linear Programming - Course Project

Kunal Kishore (UID: 505848811), *Anika Tabassum* (UID: 405790429), *Aakash Sagar Varma* (UID: 705362460)

I. Part 1

A. Problem Statement

In the first part of the project, we have to build a *Support Vector Machine* (SVM) binary classifier by formulating and solving a Linear Program. Furthermore, we have to develop a *Sample Selection* algorithm, which - in an online manner - selects 'useful' samples to be sent to train the SVM classifier. The main challenge here lies in finding out the criteria to select these *useful* samples, and to ensure that the overall performance of the algorithm is better, if not the same, than the case where we use all the data samples at once - be it in terms of the test accuracy obtained, or the processing speed. The assumption here is that the sample selection tool has access to only the current sample and the classifier's outputs.

B. Our Approach

Since we have to devise an *Online Learning* algorithm for our classification, it is understood that we will be 're-training' our classifier model multiple times iteratively. Keeping this in mind, we divide the problem statement into two parts: 1) Designing the sample selection algorithm 2) Designing the SVM classifier that can operate on updating it's weights at every new sample. For the first part, we make use of the knowledge that in an SVM linear classifier, the points that lie close to the optimal hyperplane are the ones that most influence it's position and orientation. In this context, these points are referred to as the *support vectors* of the SVM. Next, for the Online SVM classifier, we solve it's most common Linear Program (LP) formulation - but with an additional modification in order to enable iterative re-training of the classifier with a new sample, without having to re-train over the entire old batch of data.

C. Classifier Formulation

In binary classification, we are given a training dataset of n points: $(\boldsymbol{x_1}, y_1), ..., (\boldsymbol{x_n}, y_n) \in (\mathbb{R}^n, \mathbb{R})$ where $\boldsymbol{x_i}$ is the feature vector and $y_i = 1$ or -1 is the class of the i^{th} sample. A linear SVM finds the maximum-margin hyperplane $f(\boldsymbol{w}, b) = \boldsymbol{w}^T\boldsymbol{x} + b$ that *best* divides the feature space according to the class. The most common LP formulation of a linear SVM is given below:

$$\begin{aligned} \min \quad & \mathbb{1}^T\boldsymbol{t} + \lambda\|\boldsymbol{w}\|_2^2 \\ \text{s.t.} \quad & 1 - y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \leq t_i \quad \forall i = 1, ..., n \\ & \boldsymbol{t} \geq \mathbb{0} \end{aligned} \tag{1}$$

Here $\boldsymbol{w}, \boldsymbol{t} \in \mathbb{R}^n$ and $b \in \mathbb{R}$ are the variables and λ is a regularization parameter (we use $\lambda = 0.1$). We can actually ignore b by pushing it to the 'weight' vector $\boldsymbol{w}$ and appending a constant 1 to the feature vector $\boldsymbol{x}$. In the context of this report, we will mention just the weights vector and not the bias, knowing that it can be generalized by understanding that the bias term is included in the weights itself.

While this formulation works fine given the set of labelled samples $\{(\boldsymbol{x_i}, y_i)\}$, but it works in an *offline* manner - in the sense that it 'learns' over the entire set of points as a batch. So once we have solved this LP, if we get a new labelled sample that we wish to use for re-training in addition to the old set of training data, we will have to solve the full LP once again from scratch. This seems like a redundant task given that we already knew the last optimal hyperplane and just needed to fit the new point alone. In order to avoid this redundancy, we propose the following method for the k^{th} iteration that re-trains over the new sample only and solves a much easier LP thus saving time and processing resources:

$$\begin{aligned} \min \quad & t + \lambda\|\boldsymbol{w}_k - \boldsymbol{w}^*_{k-1}\|_2^2 \\ \text{s.t.} \quad & 1 - y_k(\boldsymbol{w}_k^T\boldsymbol{x}_k) \leq t \\ & t \geq \mathbb{0} \end{aligned} \tag{2}$$

Here, $\boldsymbol{w}_k \in \mathbb{R}^n$ and $t \in \mathbb{R}$ are the variables. Note that there's only a single constraint in this formulation as opposed to n constraints in the original LP. The trick that enables us to do this is the introduction of $\boldsymbol{w}^*_{k-1}$ in our formulation: which is the optimal weight vector solved for the last $k-1$ selected samples. We claim that by having the second difference norm term in the objective, we are making sure that the optimal hyperplane vector $\boldsymbol{w}^*$ of this LP remains as close as possible to the last optimal hyperplane $\boldsymbol{w}^*_{k-1}$ which had minimized the classification error over the previous $k-1$ samples. The constraint in the above LP makes sure that the new hyperplane also minimizes the error on the newly added sample $(\boldsymbol{x}_k, y_k)$. Now we can use this to run the SVM classifier in an iterative way without losing on unnecessary processing delays.

D. Sample Selection

As noted in section B, we need to fetch those samples that lie close to the classification boundary. In the context of SVM, we can state this as fetching those samples which lie close to the optimal hyperplane. In the context of LP, we can restate this as fetching those samples at which the first constraint of (1) are satisfied with equality at the optimum. This essentially corresponds to those points which either fall into the margin of the current SVM or which are misclassified by the current hyperplane. Accordingly, for checking whether a new sample

$(\boldsymbol{x}_k, y_k)$ falls into either of these 2 categories, we can write as a single condition for it to be chosen as follows:

$$1 - y_k(\boldsymbol{w}_{k-1}^{*T}\boldsymbol{x}_k) \geq 0 \tag{3}$$

Here $\boldsymbol{w}_{k-1}^*$ is the current optimal hyperplane that best classifies the last $k-1$ samples so far. If the condition (3) is satisfied by a new sample $(\boldsymbol{x}_k, y_k)$, then we select that sample and retrain the classifier using (2).

Note that as a starting point, we have to select the first (few) sample(s) without checking for the condition (3). In our code we select the first 2 samples to start with, and check for the condition (3) starting from the 3^{rd} sample.

E. The Algorithm

Algorithm 1: Online SVM Training for Part 1

Initialize: $\boldsymbol{w}_0 \leftarrow \mathbb{0} \in \mathbb{R}^n$

$k \leftarrow 1$

for $(\boldsymbol{x}, y)$ *in* $\mathcal{D}$: **do**

  if $1 - y(\boldsymbol{w}_{k-1}^{*T}\boldsymbol{x}) \geq 0$ ***or*** $k \leq 2$ **then**

    Solve for $\boldsymbol{w}_k^*$: $\begin{bmatrix} \min & t + \lambda\|\boldsymbol{w}_k - \boldsymbol{w}_{k-1}^*\|_2^2 \\ \text{s.t.} & 1 - y(\boldsymbol{w}_k^T\boldsymbol{x}) \leq t \\ & t \geq \mathbb{0} \end{bmatrix}$

    $k \leftarrow k + 1$

  end

end

As a further improvement, while our trick in (2) simplifies the re-training of SVM LP by a lot, we can further reduce the number of re-trainings done by the central node by skipping over some s number of samples which would potentially reduce our overall computation time while maintaining a similar accuracy. The drawback of this method could be that it ends up picking more number of samples - including some which were not the final support vectors.

F. Performance Evaluation & Accuracy

Fig. 1. illustrates the synthetic dataset consisting of 2000 points and Fig. 2 shows the 154 samples selected for training using our algorithm which gave final test accuracy of 93.4%.

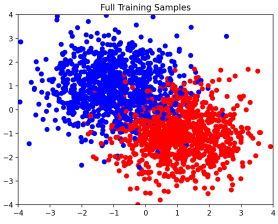

Fig. 1: *Full Training Samples*

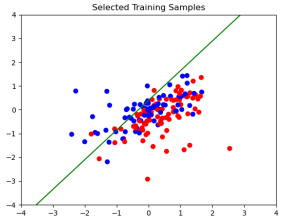

Fig. 2: *Selected Samples*

The results of our algorithm are compared in Table I to the cases when we use the entire dataset and when we use random sample selection with 30% probability of selection. Our proposed method achieved 93.4% testing accuracy compared to 93.7% training accuracy using the full dataset while only using 7.5% of the total training data.

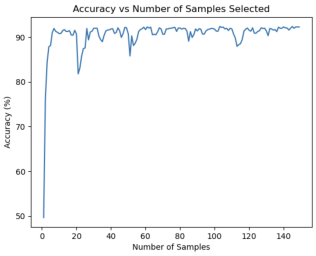

Fig. 3: *Accuracy v/s No. of Samples (Synthetic)*

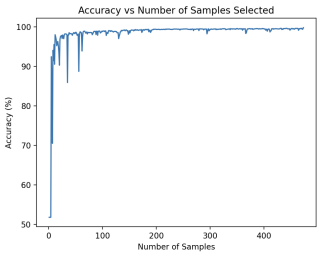

Fig. 4: *Accuracy v/s No. of Samples (MNIST)*

Table II summarizes the results of our solution compared to using 30% random sample selection and full dataset for MNIST dataset. Our proposed method achieved 99.26% testing accuracy compared to 99.306% testing accuracy using the full dataset while only using 3.62% of the total training data. This method reduced the computation time by 98.9%. Compared to randomly selecting 30% of the samples, we achieved 91.83% faster training and slightly(0.24%) better testing accuracy.

Fig. 3 and Fig. 4 illustrate the performance of the classifier as a function of the number of data samples used for training it for synthetic and MNIST dataset respectively. As the number of samples increases, the accuracy increases initially and then saturates. Table III shows the number of data points required to achieve 50%, 65%, 80%, 95% accuracy. Note that for the synthetic data set 95% could not be reached.

TABLE I: *Performance of our algorithm with respect to full dataset and random selection for Synthetic dataset*

	Full Dataset	Random Selector	Sample Selector(1)
Samples(#)	2000	601	**154**
Train Time(s)	3.13	0.85	**0.22**
Train Accu(%)	92.14	90.93	**91.35**
Test Accu(%)	93.71	93.12	**93.40**

TABLE II: *Performance of our algorithm with respect to full dataset and random selection for MNIST dataset*

	Full Dataset	Random Selector	Sample Selector(1)
Samples(#)	13007	3830	**472**
Train Time(s)	103.6579	13.776	**1.125**
Train Accu(%)	99.97	99.538	**99.8**
Test Accu(%)	99.306	99.0291	**99.26**

TABLE III: *Number of data samples required to achieve certain values of accuracy*

% Accuracy	**MNIST**	**Synthetic**
50	1	2
65	5	3
80	8	5
90	10	7
95	12	NA

II. Part 2

A. Problem Statement

In the second part of the project, in continuation with the previous part, we now have to build an *offline* sample selector for training the SVM classifier in (1). So we now have the entire data set at once to select the support vectors to be sent to the classifier. The main challenge here lies in formulating an ILP that can detect all the 'useful' samples by using the given data. We then need to relax the ILP into an LP and use an approximation method to get integer solutions close to the ILP. We expect that the samples selected using this give similar (if not better) performance than our algorithm in part 1 in terms of # of samples and/or accuracy.

B. Our Approach

The overall concept of our approach is based on the distances between points from opposite classes. We will use the visualization from the Synthetic dataset that we'd generated for Part 1 in the context of this report. We observe that given the entire dataset, the points that we deem most useful lie close to points from the opposite class. Therefore, if we just select the points whose *minimum* distance from points of the opposite class is the least, we should be done. But this would also pose a problem if there are lot of outliers in the data, since with this approach alone, all the outliers will be most likely selected. So we therefore propose an algorithm based on the *K-Nearest Neighbors* (KNN) algorithm that would not only choose the points based on the 'minimum opposite class distance' but would also be robust against selecting outliers. In the next section, we first discuss all the pre-processing that would be needed when we move on to formulating the final ILP for the sample selection.

C. Preprocessing

We first compute a cross class distance matrix $D \in \mathbb{R}^{p \times q}$ where p is the number of 'positive' samples (belonging to class 1) and q is the number of 'negative' samples (belonging to class 2), as follows:

$$D = \begin{bmatrix} d_{11} & d_{12} & \cdots & d_{1p} \\ d_{21} & d_{22} & \cdots & d_{2p} \\ \vdots & & \ddots & \vdots \\ d_{q1} & d_{q2} & \cdots & d_{qp} \end{bmatrix} \tag{4}$$

Here d_{ij} represents the distance between the i^{th} negative sample and the j^{th} positive sample.

Based on this, we then construct 2 'KNN' matrices $N_N \in \mathbb{R}^{q \times k}$ and $N_P \in \mathbb{R}^{p \times k}$ where k is our hyper-parameter. These are defined as follows:

$$N_{N,ij} = \begin{cases} \text{Index of the } j^{th} \text{ nearest positive sample to the} \\ i^{th} \text{ negative sample.} \end{cases}$$
$$N_{P,ij} = \begin{cases} \text{Index of the } j^{th} \text{ nearest negative sample to the} \\ i^{th} \text{ positive sample.} \end{cases} \tag{5}$$

Based on these 2 matrices, we further compute 2 matrices $W_P \in \mathbb{R}^{q \times p}$ and $W_N \in \mathbb{R}^{q \times p}$ defined as follows:

$$W_{P,ij} = \begin{cases} 1, & \text{if } j \text{ exists in } N_{N,i:} \\ 0, & \text{otherwise} \end{cases}$$
$$W_{N,ij} = \begin{cases} 1, & \text{if } i \text{ exists in } N_{P,j:} \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

Based on these 2 matrices, we generate 2 vectors $\boldsymbol{f}_P \in \mathbb{R}^p$ and $\boldsymbol{f}_N \in \mathbb{R}^q$ defined as follows:

$$\boldsymbol{f}_{P,i} = \sum_{j=1}^{q} W_{P,ji} \quad , \quad \boldsymbol{f}_{N,i} = \sum_{j=1}^{p} W_{N,ij} \tag{7}$$

We then concatenate these 2 vectors into a single vector $\boldsymbol{f} \in \mathbb{R}^n$ (where n is total number of given samples) corresponding to the indices of the positive and negative samples in the given dataset $(\boldsymbol{x}_i, y_i)$ as follows:

$$\boldsymbol{f}_{i,y_i=+1} = f_{P,:} \quad , \quad \boldsymbol{f}_{i,y_i=-1} = f_{N,:} \tag{8}$$

D. Algorithm Logic and Claim

Intuitively, the i^{th} value in this $\boldsymbol{f}$ vector shows the number of times a particular sample $\boldsymbol{x}_i$ is in the k nearest neighbors of another sample of opposite class. We claim that if we pick the samples with a high value in this corresponding $\boldsymbol{f}$ vector, it will include all the support vectors but mostly exclude all the outliers.

The first part of the claim is self-explanatory by the definition of this matrix, since if a sample is in the k nearest neighbor of samples from opposite class, it is likely that such a sample exists near the optimal classifying hyperplane. But how does this possibly exclude the outliers? We observe that the outliers will be less likely to be nearest neighbors of samples of opposite class, because there will be more population of the opposite class points in that region which will be the nearest neighbors amongst themselves.

We will now use this final vector $\boldsymbol{f} \in \mathbb{R}^n$ to formulate an ILP in the next section that will generate the selection decision for each sample $\{\boldsymbol{x}_i\} : i = 1, ..., n$

E. Formulating the ILP

As discussed in the previous section, our goal is to pick those points which have a high value in the $\boldsymbol{f}$ vector. But note that we also need to minimize the number of total samples selected at the end. So while we have a maximization task of picking the highest $f(i)$ values, we need to combine it with the minimization task of picking minimum number of samples. We therefore propose the following ILP to get the optimal selection of the samples:

$$\begin{aligned} \min \quad & \mathbb{1}^T \boldsymbol{s} - \mu \boldsymbol{f}^T \boldsymbol{s} \\ \text{s.t.} \quad & \boldsymbol{s} \in \{0, 1\} \end{aligned} \tag{9}$$

Here $\boldsymbol{s}$ is the variable and μ is the regularization parameter that controls how much weightage we give to reducing the number of samples vs. picking more samples which are candidate support vectors. After cross-validating with both the synthetic

and MNIST datsets, we found that $\mu = 10$ gives us the best results in terms of accuracy. This parameter can be reduced if there is a lower bound restriction on the number of samples to be selected, while it can be increased if higher accuracy is desired at the expense of more number of samples selected.

Based on the optimal point $\boldsymbol{s}^*$ of this ILP, we will select the i^{th} sample in our training set iff $s_i = 1$.

F. LP relaxation

Since solving an ILP could be time and resource consuming, we intend to relax the aforementioned ILP into an LP. We therefore relax this ILP into the following LP:

$$\begin{aligned} \min \quad & \mathbb{1}^T \boldsymbol{s} - \mu \boldsymbol{f}^T \boldsymbol{s} \\ \text{s.t.} \quad & \boldsymbol{s} \leq 1 \\ & \boldsymbol{s} \geq 0 \end{aligned} \tag{10}$$

Note that since the constraints of this LP form an *integral polyhedron*, the optimal solution generated by solving this LP will be the same as solving the ILP i.e. s_i is going to be 0 or 1 only. So even though we ideally don't need any rounding off for processing the optimal solution, practically the LP solver that we use in python gives a small error in the optimal solution and rather gives a near-optimal solution. This means that the optimal point $\boldsymbol{s}^*$ may consist of values other than 0 and 1. Therefore, we came up with a relaxed criteria for selecting samples based on the optimal point $\boldsymbol{s}^*$ of this ILP: we will select the i^{th} sample in our training set if $s_i \geq 0.5$.

G. Performance Evaluation and Accuracy

Fig. 5 illustrates the synthetic dataset consisting of 2000 points and Fig. 6 shows the 314 samples selected for training using our sample selection algorithm which gave a final test accuracy of 93.68% (using $\mu = 10$).

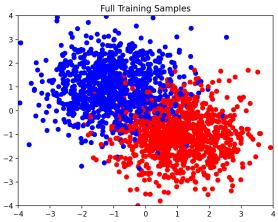

Fig. 5: *Full Training Samples*

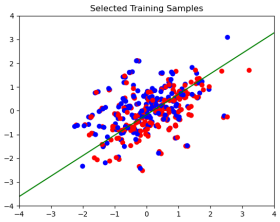

Fig. 6: *Selected Samples*

TABLE IV: *Number of data samples required to achieve certain values of accuracy*

% Accuracy	MNIST	Synthetic
50	1	1
65	6	5
80	9	7
90	13	12
95	19	NA

H. Hyperparameter Tuning

Table V illustrates the results of using different hyperparameters μ on the synthetic dataset:

TABLE V: *μ value vs. # of Samples and Test Accuracy*

μ	# of Samples Selected	Test Accuracy
0.1	52	92.17
1	189	93.12
10	**314**	**93.68**
100	561	93.70

Observe that as discussed in section E, a lower value of μ selects lesser number of samples while trading off with a lesser test accuracy. And similarly, a higher value of μ achieves a higher accuracy while selecting more number of samples.

I. Comparison of Part 1 and Part 2

In this section, we compare the results of the 'online' algorithm of Part 1 and the 'offline' batch algorithm Part 2. We observe that while the results from both the parts are quite comparable to each other, the Part 2 algorithm was able to marginally perform better in case of Synthetic dataset in terms of accuracy. But when we compare the time taken for the sample selection, Part 1 performed way faster due to the iterative light LP that we had proposed, while the single ILP (or relaxed LP) turned out to be a little more time consuming.

Observe that even though the performances of Part 1 and Part 2 in terms of accuracy is similar, our Part 2 algorithm ended up selecting a little more number of samples overall than Part 1. This is because in Part 1, we were able to iteratively solve for the optimal hyperplane at every step, and since it was able to achieve very high accuracy within just the first few 10's of samples, we already knew what the optimal hyperplane almost looked like. This gave Part 1 an huge advantage over Part 2, since whatever estimate Part 1 does at every step to select the candidate support vectors, it's going to be almost accurate. While in Part 2, since we are having to get an estimate of the candidate support vectors in a 'blind' manner, it's difficult to outperform the algorithm of Part 1 with even fewer samples this way.

But having observed that, we note that the Part 1 is not robust enough in the following scenario: When the input dataset is sequentially skewed - in the sense that there are relatively more number of samples of a particular class towards the beginning of the dataset and most of the other class samples lie towards the end of the dataset. In this case, the Part 1 algorithm will not be able to correctly estimate the optimal hyperplane with just the first few samples and could likely take more than half the dataset to start getting a good estimate. This means that it could end up selecting a lot more number of samples that aren't actually the support vectors of the final resulting hyperplane. This kind of problem can never occur in Part 2, since it operates in a batch manner, so clearly the order of samples in the dataset does not influence the operation of the algorithm.