# Programming Assignment 2: Training SVMs with Stochastic Gradient Descent

Daniel Rose 12025566
Roxane Jacob 12030167

15th May 2022

## 1 Introduction

Support-vector machines (SVMs) are a set of supervised learning methods used for classification, regression and outlier detection. They are based on the statistical learning theory proposed by VAPNIK and CHERVONENKIS, and are quite popular due to their effectiveness in high-dimensional feature spaces, relative memory efficiency and versatility when combined with kernel functions.

Given a set of binary, labeled training examples, an SVM training algorithm builds a model that assigns new examples to one category or the other, by mapping training examples to points in space so as to maximise the width of the gap between the two categories. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall. In their original formulation, SVMs are binary, linear classifiers, but they can also perform multiclass classification by using multiclass loss functions, and non-linear classification by implicitly mapping their inputs into high-dimensional feature spaces in what is called the kernel trick.

## 2 Task Definition

The goals of this assignment were to:

- implement a linear version of SVM using Stochastic Gradient Descent (SGD) for optimization,
- implement an approximate Radial Basis Function (RBF) kernelized SVM using Random Fourier Features (RFFs) to approximate the Laplacian kernel function,[1]
- parallelize the obtained algorithms,
- and compare the accuracy and runtime of the different implementations on various data sets.

## 3 Information on Hardware, Data, and How to Reproduce the Results

The code for this assignment was written in PYTHON. The results presented in this report were generated with the hardware reported in Table 1, and can be reproduced by running the `main.py` file in the top directory of the submitted .zip-folder. Running this script generates 12 figures, which are saved as portable network graphics (.png) files in the output folder. Please refer to the `README.md` file for detailed information on how to execute the code.

**Table 1:** Hardware specifications.

| | |
|---|---|
| Processor | Intel(R) Core(TM) i7-1065G7 @ 1.30 GHz<br>1 Processor, 4 Cores |
| L1 Cache | 32.0 KB |
| L2 Cache | 2.0 MB |
| L3 Cache | 8.0 MB |
| RAM | 16.0 GB |

We tested our implementations on 3 data sets, which can all be found in the data sub-folder

of our submission .zip-file. The smallest one, `toydata_tiny.csv`, was mainly used to develop the code and visualize our results. The second one, `toydata_large.csv`, was used to evaluate the effects on runtime and quality when going from serial to parallel learning. The third data-set, MNIST was used to test our implementation of the multi-class hinge loss and see how our SVMs behave on a "real-world" dataset. Table 2 provides an overview of the size and dimensionality of each data set.

**Table 2:** Data sets for this assignment.

| name | number of samples | dimensionality | number of distinct labels |
|---|---|---|---|
| `toydata_tiny.csv` | 200 | 2 | 2 |
| `toydata_large.csv` | 200000 | 8 | 2 |
| MNIST | 60000:10000 (train:test) | 784 | 10 |

## 4 Implementation

The `main.py` file calls two routines, one for the toy data and one for the MNIST data set, in which all required computation is performed. In each of the routines we start by importing the data, scale it with sklearn's standard scaler, perform the train/test split and compute the baseline accuracy with sklearn's stratified dummy classifier.

For the toy data, the routine continues with applying sklearn's svm.SVC (C-Support Vector Classification) classifier, whose results we used as a golden standard for the classification accuracy on each data set. The routine then classifies the data *via* our linear implementation (sequential and parallel), computes the random Fourier features (RFF), and performs the sequential and parallel classification again on the RFF. Note that each time a classifier is called, we first perform a grid-search over a list of parameters to determine the optimal learning rate $\eta$ and regularization parameter $\lambda$, and then perform a five-fold cross-validation with the optimal parameters to obtain the predicated labels, the runtime and the accuracy. The routine also runs the parallel linear SVM classification on an increasing number of machines. The resulting accuracies and runtimes are then returned by the routine, and plotted in the main script. In the case of the tiny toy data set, we also plot the results of each algorithm in two-dimensions.

The routine for the MNIST data set is constructed similarly, with some small differences. After

loading and scaling the data, we compute the baseline accuracy and perform the the classification with our linear algorithm (sequential and parallel), for which we determine the optimal parameters *via* grid-search, but this time we do not use 5-fold cross validation to compute the performances, since it was explicitly not required in the assignment. The routine also runs as before the parallel linear SVM classification on an increasing number of machines. We then only use a subset of the data (3000 samples for the training and 500 samples for the test data set) to compute the RFF and run the sequential and parallel SVM on the transformed features. Again, we determine the optimal learning rate and regularization parameters *via* grid-search, and this time we also perform a grid-search to determine the optimal number of random fourier features and the optimal $\sigma$ parameter for the Laplacian kernel. We finally make plots comparing the performances of our SVM implementations and sklearn's svm.SVC[2] when training on 1000, 2000, and 3000 samples. The routine returns as before the performances of the parallel linear SVM on a number of machines ranging from 1 to 4, which are then plotted in the main script.

We implemented the linear SVM in the primal as indicated in equations 1 and 2 and used SGD for optimization. The regularized hinge loss minimization formulation of the SVM for the binary classification problems is given by:

$$\min_{\mathbf{w}} \lambda \mathbf{w}^T \mathbf{w} + \sum_{i=1}^{n} \max(0, 1 - y_i \mathbf{w}^T \mathbf{x}_i) \tag{1}$$

where $y_i \in \{-1, +1\}$ are the labels, $\mathbf{w}$ is the weight vector, $\mathbf{x_i}$ is the respective sample and $\lambda$ is the regularization parameter.

Because the MNIST data set has 10 classes with labels $y_i \in \{0, \ldots, 9\}$, we used the multi-class hinge loss as given by equation 3. The algorithm for the SGD was taken from *Zinkevich et al.*[3] and is given in Algorithm 1.

$$\min_{\mathbf{w}} \lambda \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^{n} l(\mathbf{w}^0, ..., \mathbf{w}^9); \mathbf{x}, y) \tag{2}$$

$$l(\mathbf{w}^0, ..., \mathbf{w}^9); \mathbf{x}, y) = \max\{0, 1 + \max_{j \in \{0,...9\}, j \neq y} \mathbf{w}^{j,T} \mathbf{x} - \mathbf{w}^{y,T} \mathbf{x}\} \tag{3}$$

where $\mathbf{w}^0, ..., \mathbf{w}^9$ are the weights of the SVM for each of the 10 possible digits, $\mathbf{x}$ are the features of a sample and $y$ is the corresponding label.

**Algorithm 1** SGD($\{c^1, ..., c^m\}, T, \eta, w_0$)

**for** $t = 1$ **to** $T$ **do**
  Draw $j \in 1...m$ uniformly at random.
  $w_t \leftarrow w_{t-1} - \eta \delta_w c^j(w_{t-1})$.
**end for**
**return** $w_T$.

with $c_i(w)$ a loss function indexed by $i$ with parameter $w$, $T$ the number of steps, $\eta$ a fixed learning rate, and $w_0$ the initial guess of $w$.

The NonLinearFeatures class in the `svm.py` script implements an explicit kernel function and projects some input data of dimension $n \times d$ onto a higher dimensional features space of dimension $n \times m$, where $d < m$. The projection uses Random Fourier Features (RFF) to approximate a Laplacian Kernel.

The RFF method draws $m$ random samples $\omega_1, \ldots, \omega_m$ and $b_1, \ldots, b_m$, where $\omega_i$ are vectors of dimension $d$, drawn from a standard Cauchy distribution and $b_i$ are random numbers, drawn from a continuous distribution from the interval $[0, 2\pi]$.

The random features $\mathbf{z}$ are then generated by applying equation 4 to the original features $\mathbf{x}$.

$$\mathbf{z}(\mathbf{x}) = \sqrt{\frac{2}{m}} \left[ \cos(\omega_1^T \mathbf{x} + b_1), \ldots, \cos(\omega_m^T \mathbf{x} + b_m) \right] \tag{4}$$

The SGD was parallelized according to the SimuParallelSGD algorithm given by *Zinkevich et al.*[3]

**Algorithm 2** SimuParallelSGD(Examples $\{c^1, ..., c^m\}$, Learning Rate $\eta$, Machines $k$)

Define $T = \lfloor \frac{m}{k} \rfloor$
Randomly partition the examples, giving $T$ examples to each machine.
**for all** $i \in 1, ..., k$ **parallel do**
  Randomly shuffle the data on machine $i$.
  Initialize $w_{i,0} = 0$.
  **for all** $t \in 1, ..., T$ **do**
    Get the $t$th example on the $i$th machine (this machine), $c^{i,t}$
    $w_{i,t} \leftarrow w_{i,t-1} - \eta \delta_w c^i(w_{i,t-1})$.
  **end for**
**end for**
Aggregate from all machines $v = \frac{1}{k} \sum_{i=1}^{k} w_{i,t}$ and **return** $v$.

We implemented threading *via* the `ray` library.[4] The reported performances for the parallel

implementation of an SVM were generated using 4 threads, unless explicitly stated otherwise.

## 5 Results and Discussion

The learning rate and regularization parameters were selected *via* a grid-search for every single data set and algorithm combination, and are listed in Table 3. Note that for the toy data sets these parameters can change a lot from one run to the next, because almost every tested combination of parameters leads to a very high accuracy. The grid-search then simply randomly returns one of the multiple combinations that led to the best accuracy.

Table 3 also contains the runtime and accuracy of each experiment. Note that these values can change slightly from one run to run, even though 5-fold cross-validation was used to mitigate these variations. The baseline accuracies of all three data sets are reported in Table 4.

Two dimensional visualizations of the classification results of all four versions of the SVM algorithm (sequential linear, parallel linear, sequential RFF, parallel RFF) on the tiny toy data set are provided in the Appendix.

We obtained almost perfect classification results (accuracy $\geq 0.98$) on both toy data sets for all four SVM implementations. For the MNIST data set, we obtained an accuracy of 0.75 on the original features, and 0.43 and 0.36 for the sequential and parallel versions respectively on the random Fourier features.

We observe that for large enough data sets, the runtime of the SVM increases with the size of the dataset, and is of course lower for the parallel implementation than for its sequential counterpart. This is of course not observable for the tiny toy data set, for which the parallelized runtimes are equal to or even greater than the sequential runtimes, because the sequential algorithm is already so fast that parallelizing it does not improve the runtime, and can even add to it due to the cost of communication.

**Table 3:** Data sets for this assignment.

| data set | algorithm | features | learning rate | regularization | runtime | accuracy |
|---|---|---|---|---|---|---|
| `toydata_tiny.csv` | sklearn | linear | - | - | $< 0.0001$ | 1.00 |
| `toydata_tiny.csv` | sequential | linear | $10^2$ | $10^{-4}$ | 0.0031 | 1.00 |
| `toydata_tiny.csv` | parallel | linear | $10^2$ | $10^{-4}$ | 0.0031 | 1.00 |
| `toydata_tiny.csv` | sequential | RFF | 10 | $10^{-4}$ | $< 0.0001$ | 1.00 |
| `toydata_tiny.csv` | parallel | RFF | 1 | $10^{-2}$ | 0.0031 | 0.98 |
| `toydata_large.csv` | sklearn | linear | - | - | 2.2498 | 0.99 |
| `toydata_large.csv` | sequential | linear | 1 | $10^{-3}$ | 0.4375 | 1.00 |
| `toydata_large.csv` | parallel | linear | 1 | $10^{-3}$ | 0.0200 | 1.00 |
| `toydata_large.csv` | sequential | RFF | $10^{-1}$ | $10^{-2}$ | 0.5031 | 1.00 |
| `toydata_large.csv` | parallel | RFF | $10^2$ | $10^{-5}$ | 0.3031 | 1.00 |
| MNIST | sequential | linear | $10^{-3}$ | $10^{-5}$ | 2.0674 | 0.75 |
| MNIST | parallel | linear | $10^{-2}$ | $10^{-2}$ | 1.1386 | 0.75 |
| MNIST | sequential | RFF | 1 | $10^{-5}$ | 0.2812 | 0.43 |
| MNIST | parallel | RFF | $10^{-1}$ | $10^{-2}$ | 0.1094 | 0.36 |

For the MNIST data set, the optimal number of RFF $m$ and the value of the $\sigma$ parameter for the Laplacian kernel were determined *via* grid search and are: $m = 2000$, $\sigma = 300$. Setting the number of features to lower or higher values led to a rapid decrease of the classification accuracy. We are not surprised by the fact that setting the number of RFF to a value that is smaller than the dimensionality of the data set (in our case $d = 784$) leads to a bad approximation of our features, however, we were surprised by the fact that increasing the number of RFF also appears to worsen the classification accuracy. We actually expected that once a certain number of features was attained, the accuracy would remain good because the approximation could

only become better by computing even more features.

Despite having optimized the hyper parameters of the RFF approximation, we could not achieve an improvement of the accuracy on the test data. We conclude that the Laplacian kernel function might not be suited for the MNIST data. There is a range of other radial basis functions available. Further investigation of these might reveal a kernel function that could lead to an increase rather than a loss of the accuracy.

For the toy data sets, we report that setting the number of random Fourier features over 20 is enough to obtain a classification accuracy of 1.0. Nevertheless, we set the number of RFF to $m = 100$ and $\sigma = 1.0$, to be in accordance with the requests of the assignment.

**Table 4:** Baseline accuracies.

| data set | baseline accuracy |
|---|---|
| `toydata_tiny.csv` | 0.50 |
| `toydata_large.csv` | 0.50 |
| MNIST | 0.10 |

The following plots (Figure 1 and 2) illustrate the convergence of the SGD over the number of epochs. The relative training error $\epsilon$ was computed for every epoch according to equation 5, where $w$ is the weight vector at the current epoch, and $w^*$ is the final weight vector.

$$\epsilon = \frac{||w - w^*||_1}{||w^*||_1} \tag{5}$$

For the tiny toy data set (Figure 1), the SVM over the linear features converges after only two epochs, and the SVM trained with the RFF converges in 150 epochs. For the large toy data set (Figure 2), both SVMs converge in approx. 150,000 epochs. The notable difference between the linear features and the RFF is that the relative training error of the RFF changes much less per epoch and results into a smoothed curve when compared to the linear feature case.

We can also observe that the SGD starts by performing significant updates of the weight vectors leading to a strong decrease in the training error, followed by more refined steps, with smaller changes of the training error. This behavior is consistent with our expectations.

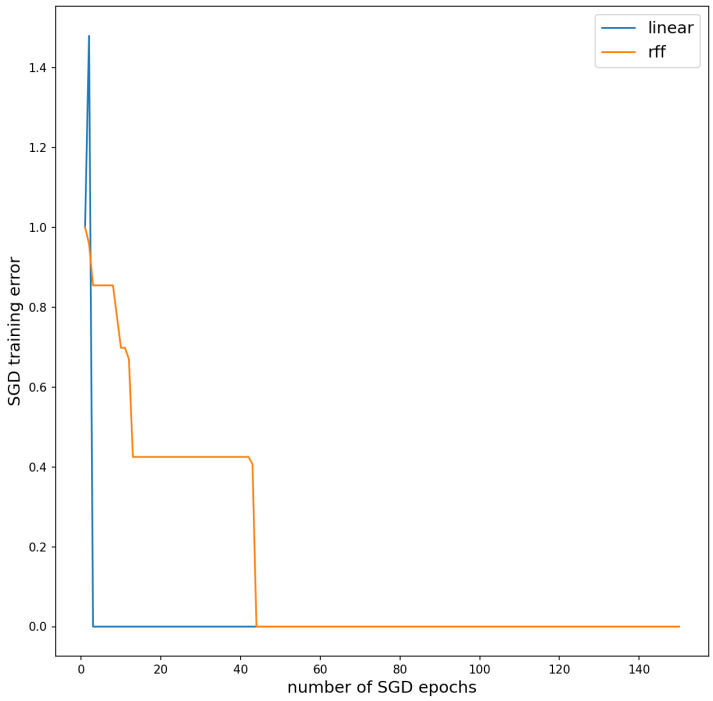


**Figure 1:** Convergence of Stochastic Gradient Descent (dataset: toydata tiny).

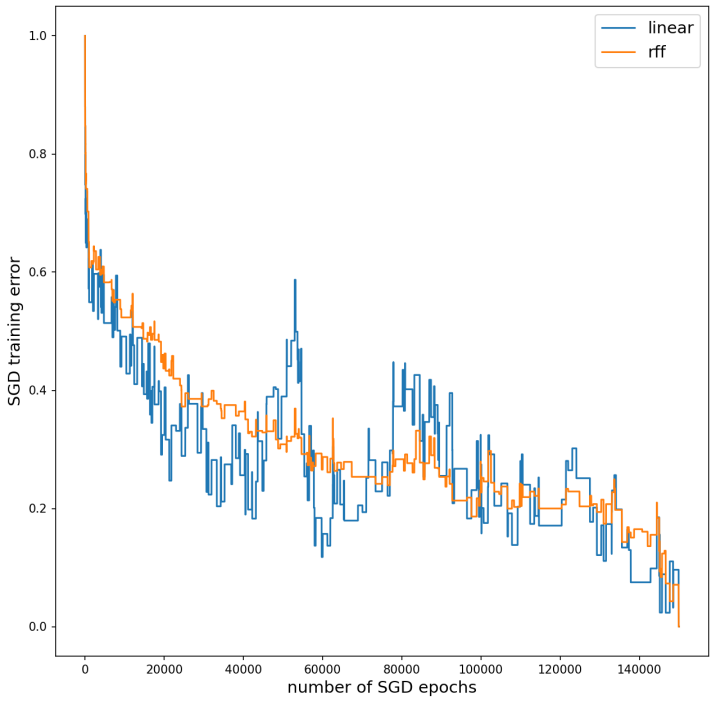


**Figure 2:** Convergence of Stochastic Gradient Descent (dataset: toydata large).

Figure 3 compares the runtime and performance of our sequential RFF SVM algorithm with sklearn's SVM.svc classification algorithm, when training on 1000, 2000, and 3000 samples from the MNIST dataset.

We observe that sklearn's algorithm attains higher classification accuracies than our algorithm, but takes also longer to achieve those results. We also observe that the runtime unsurprisingly increases with the size of the data set, but more interestingly, that the classification accuracy of our algorithm is more dependent on the size of the data than sklearn's algorithm: the more samples we train on, the better the accuracy.

The implementation of sklearn's svm.svc[2] class is based on the library `libsvm`. The multiclass support is handled according to a one-*vs*.-one scheme. They report that the fit time scales at least quadratically with the number of samples and may be impractical beyond tens of thousands of samples. We had the chance to verify this statement by attempting to run sklearn's algorithm on the entire MNIST data set: the algorithm took so long that we decided to inter-

rupt it before it had reached convergence.

We conclude that on very large data sets, our SVM classifier is superior to sklearn's algorithm, but on smaller data sets, we should rather consider using sklearn's implementation.

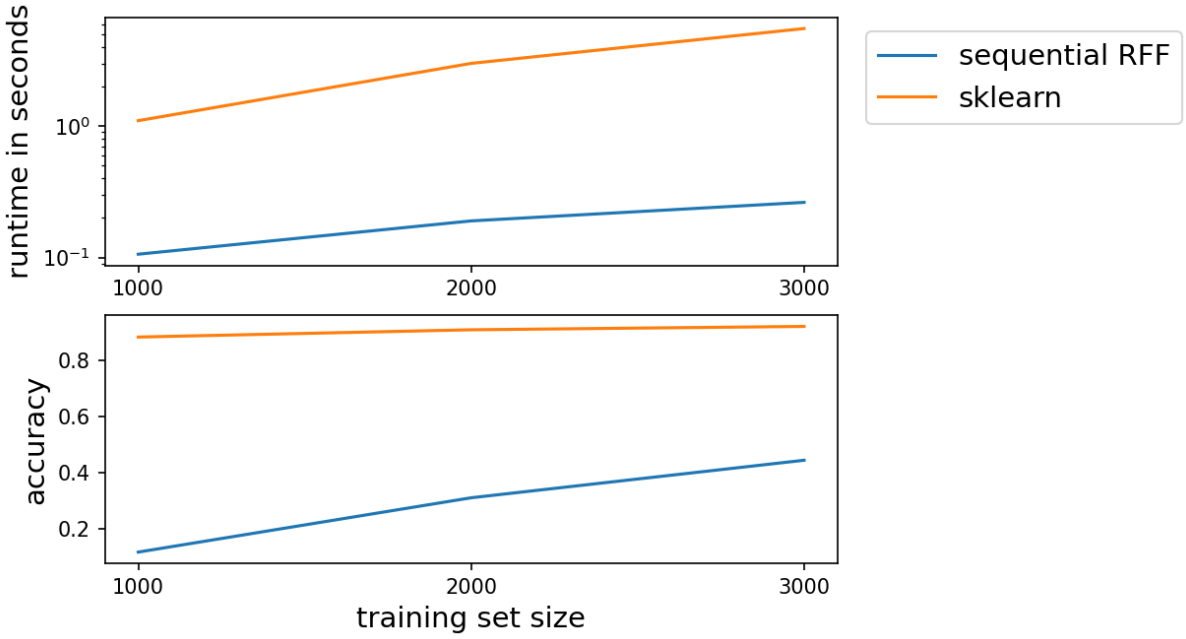


**Figure 3:** Performance comparison of our (sequential) RFF SVM algorithm vs. sklearn's svm.SVC class on subsets of the MNIST data set.

By increasing the number of threads (equiv. machines), we could reduce the runtime of our SVM algorithm. Figure 4 shows how the runtime (right plot) decreases when a thread is added. In theory, doubling the number of threads should halve the runtime, but in practice there exists a lower bound on the achievable runtime of each algorithm, which is independent of the number of machines on which the workload is divided. This lower bound is given by the proportion of code that is not parallelizable.

The left plot in Figure 4 shows that the number of threads does not affect the accuracy of the SVM. The fact that this statement is not true for the tiny toy data set (blue line) can be disregarded due to the size of this dataset. Indeed, it is so small that parallelizing the SVM does not improve the runtime (as shown in the right plot), and sometimes has the adverse effect of lowering the accuracy because the subsets that are send to the various threads become too small to derive a correct separating hyperplane.

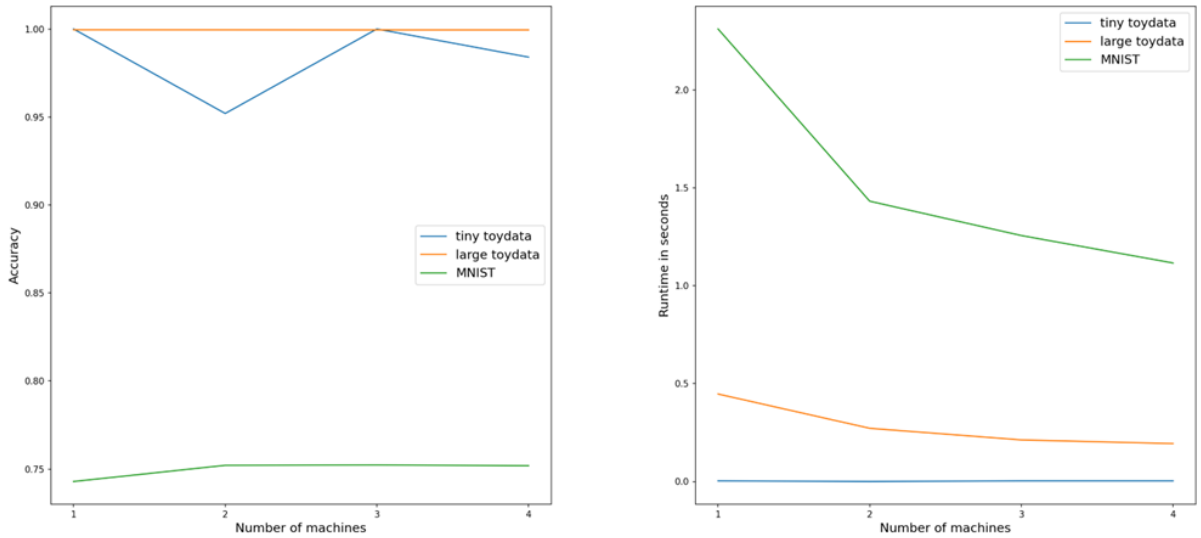


**Figure 4:** Accuracy (left) and Runtime (right) of the Parallel SVM Algorithm on the Tiny Toy Data Set (blue), Large Toy Data Set (orange), and MNIST Data Set (green).

Table 3 showed that the classification accuracies were the same for the parallel and the sequential versions of the SVM algorithm on both toy data sets. However, the classification accuracy on the MNIST data set decreases from 0.43 to 0.36 after parallelizing. Unfortunately, we could not explain why this effect occurs.

# 6 Appendix

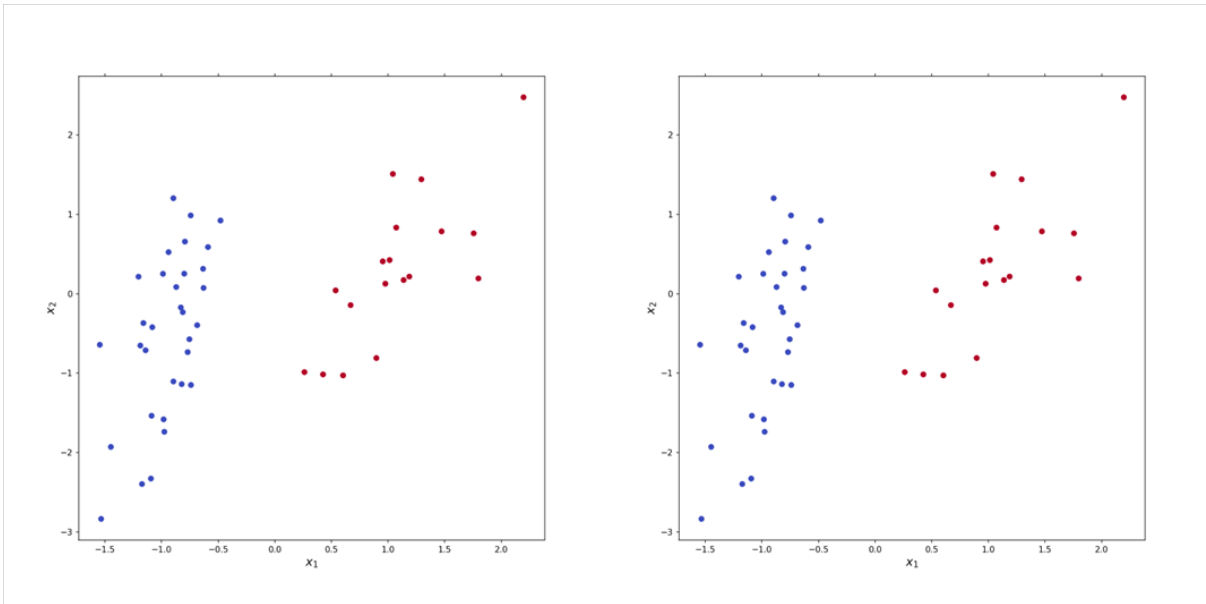


**Figure 5:** Scatter Plot of the Tiny Toy Data Set: True Labels (left) and Labels as Predicted by Sklearn's SVM.svc Algorithm (right).

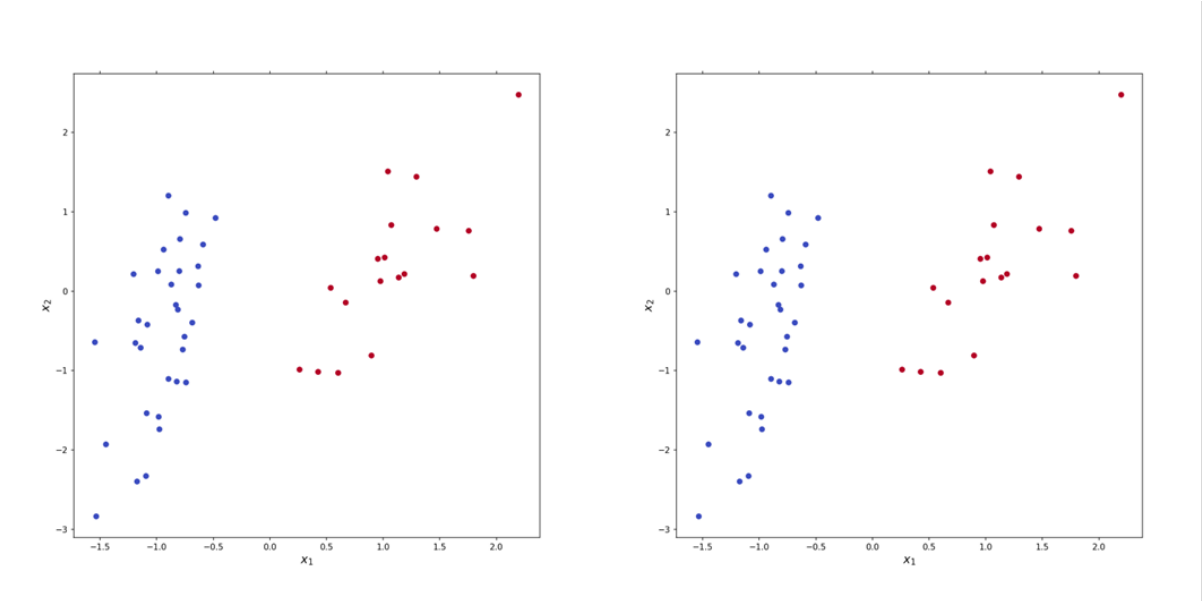


**Figure 6:** Scatter Plot of the Tiny Toy Data Set: Labels as Predicted by the Sequential (left) and parallel (right) SVM Algorithm on the Original Data.

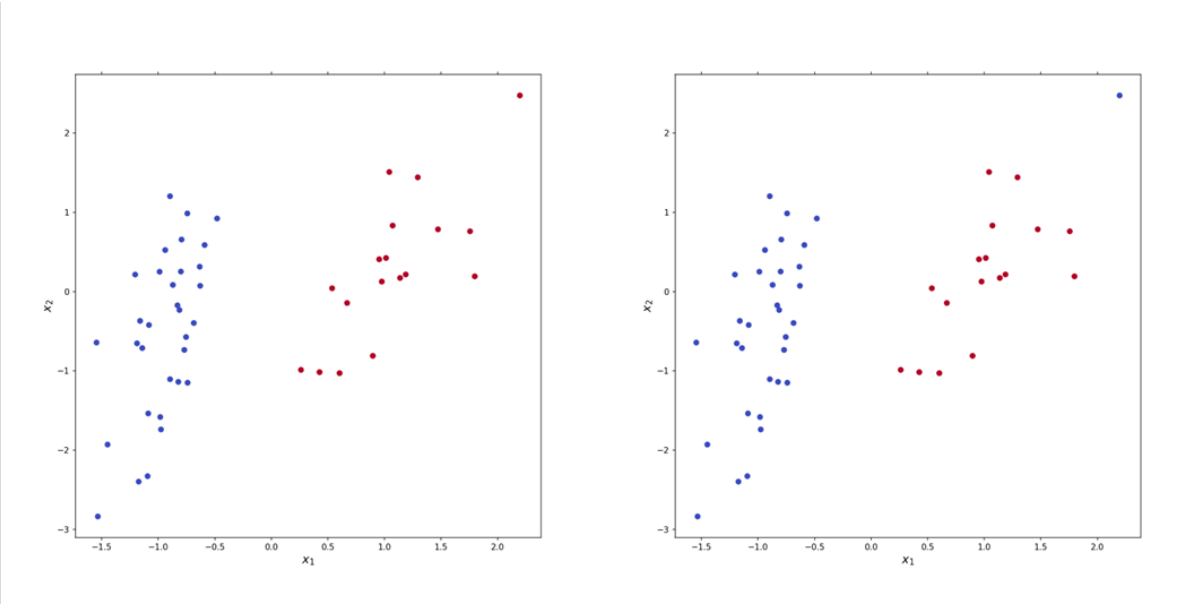


**Figure 7:** Scatter Plot of the Tiny Toy Data Set: Labels as Predicted by the Sequential (left) and parallel (right) SVM Algorithm on the Random Fourier Features.

## References

[1] R. B. Rahimi, A., "Random features for large-scale kernel machines," *Advances in neural information processing systems*, vol. 20, 2007.

[2] https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

[3] W. M. S. A. J. L. L. Zinkevich, M., "Parallelized stochastic gradient descent," in *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010. Proceedings of a meeting held 6-9 December 2010*, (Vancouver, British Columbia, Canada), pp. 2595–2603, 2010.

[4] https://www.ray.io/.