MMD: Programming Assignment #2

Group 1: E. Guliev, J. Rass, C. Wiskott

University of Vienna — May 23, 2021

Abstract

All tasks were completed and instructions on executing the program can be found in the attached `README.md`. Furthermore they are uploaded to the GitHub repository: `https://github.com/glarkstoat/SVM-using-SGD`. The files require no input parameters and can simply be run with a standard IDE or from the command-line. The results of the hyperparameter search as well as the classification accuracies and convergence plots can be found in the provided Jupyther notebooks in the root folder.

The *LinearSVM* achieved a perfect linear seperation of both toydata test-sets and 92.65 % classification accuracy on the MNIST test-set. Those figures and their corresponding runtimes are summarized in table 2.

Random Fourier Features have very comparable performance in terms of runtime and accuracy compared to the regular features using a linear SVM.

Futhermore, SimuParallelSGD improved accuracy and runtime (with up to 3.3x speedup) compared to values obtained prior with LinearSVM.

Introduction

The task of this assignment was to handle scaled-up supervised learning problems using different approaches for three dataset with different sizes and number of classes. The first third of this report will deal with the implementation of the *linear SVM model* which uses mini-batch gradient descent for optimization, as well as the *MultiClass SVM*, which was designed to handle the MNIST dataset. The second section will deal with the computation of the *Random Fourier Features* and will compare the results with the optimization using the original features. The last part will present the results of the parallel implementation and will showcase the potential speed-up compared to the serial version.

Data Loading and Data Preparation

For loading the data the class *DataLoader* was created, which has the functions *get_toydata_tiny*, *get_toydata_large* & *get_mnist*. Each of the functions reads the corresponding file, performs a train- test split with train size = 0.7 and applies normalization (*StandardScalar*) to the features, in order to increase classification accuracy during the training-phase. Fig. 1 shows the distribution of the class labels for each class.

Linear SVM Model

The *LinearSVM* class can calculate the optimization either in a serial- or a parallel fashion. The serial *SGD* performs optimization via mini-batch gradient descent with an adaptive learning rate of $\eta_{t+1} = \eta_t/\sqrt{t}$. Each epoch consists of shuffling the training data and iterating over the entire set in batches of size *batch_size*. For each (sample, label) - pair in a given batch, the prediction

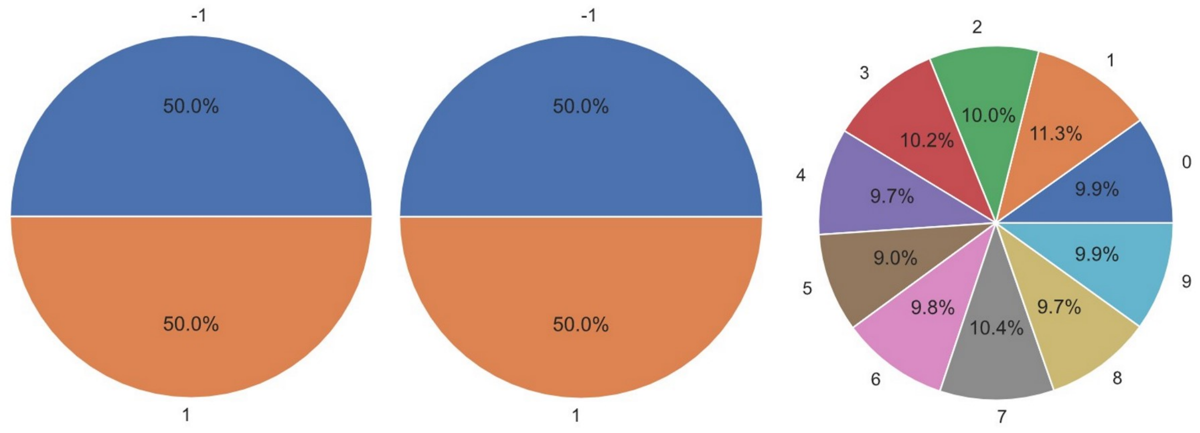

Figure 1: Distribution of class labels for all three datasets. From left to right: toydata_small, toydata_large and MNIST. It is apparent that all datasets are roughly balanced and thus optimization should not be impeded by the inherent problems of unbalanced data-sets.

based on the current weights is made, then if the sample was incorrectly classified or it's located within the margin, the corresponding gradient is calculated. These gradients are summed up for every batch and the weights are updated with the average gradients after a batch has been completed. After all epochs have been completed the sequences of hinge losses and accuracy rates for every epoch are returned by the *train* function.

The *MultiClassSVM* class has the same functionality but can handle multiple classes by minimizing the multi-class hinge loss described in the assignment instructions. Here every class label has its own set of weights which are updated during the optimization process. The central condition that we want to enforce is described in eq. 1. It states that for a given label y, the confidence in the label using the corresponding weights $w^{(y)T}x$ is greater or equal than the maximum confidence with any other set of weights $w^{(i)T}x + 1$. With 1 being the typical SVM margin that allows for some error.

$$w^{(y)T}x \geq \max_{i\in(1,..,c),\ i\neq y} (w^{(i)T}x) + 1 \tag{1}$$

This ensures that the weights $w^{(y)}$ return the highest confidence scores for labels of class y while incorporating the error margin. If this condition is fulfilled the weights do not have to be updated and the next (sample, label) - pair is considered. Equations 2b and 2c show the cases in which the weights are updated via GD. The relevant code can be inspected in *MultiClassSVM.py* on lines 80-95.

$$\Delta_{w^{(i)}}\, l_{mc-h}(w^{(1)},...,w^{(c)}; x, y) = \begin{cases} 0, & \text{if (1) satisfied or } (i \neq y) \text{ and i } \notin argmax_j\ w^{(i)T}x \quad (2a) \\ -x, & \text{if (1) not satisfied and } i = y \quad (2b) \\ x, & \text{otherwise} \quad (2c) \end{cases}$$

The theoretic framework was taken from [1].

Hyperparameter Search

In order to obtain the optimal hyper-parameters, cross-validation was employed following the guidelines of the assignment instructions. For this purpose the file *CV.py* was used, which is recycled code from an IML assignment. The hyper-parameter search can be inspected in the *test.ipynp* file for all three datasets. Both toydata sets returned perfect CV-scores using the parameters of table 1, while the MNIST dataset returned 91.63 % average classification score on the validation sets.

Dataset	η_0	C	$batch\_size$	CV-score [%]
Toydata Tiny	1	0.001	10	100.0
Toydata Large	0.1	0.001	50	100.0
MNIST	0.01	0.01	50	91.63

Table 1: Optimal hyper-parameters for all three data-sets, evaluated based on the corresponding cross-validation score on the validation set. η_0 refers to the initial learning rate, C to the regularization parameter and $batch\_size$ to the batch size during optimization with the mini-batch GD.

Results

The obtained hyper-parameters were used to **(*1*)** train the SVMs for each data set and to **(*2*)** compute the classification accuracies for the respective test sets.

(*1*) Fig. 2 showcases the convergence behaviour during training of the data-sets. We can see that the optimization of the tiny training set converged very quickly to a point where improvements to the loss stagnated and the accuracy stayed constant at around 99.3 %. Upon further inspection this was due to a single training sample that could not be classified correctly. The results for the large data-set show very similar convergence behaviour but with the classification accuracy being at a constant 100 %. This shows that both toydata sets are very easy to optimize using both the original features and the RFFs.

Notice: In order to decrease the run-time, the convergence plots show the results per epoch and not per mini-batch during optimization. In the case of the larger datasets, the algorithm will already have made (depending on the batch size) several thousand updates to the weights, before the first accuracy is calculated at the end of the first epoch. Thus it can happen that the first accuracy is already at 100 %.

The convergence plot of the MNIST training-set reveals that the convergence was fairly slow and only improved the initial accuracy by +1.5 % after the final epoch to around 93.4 %. Although this is a fairly high accuracy score, the addition of more epochs only further increases run-time while only providing slightly better results.

(*2*) The results are presented in table 2 and show that the test sets of the toydata-sets are both perfectly linearly separable due to the 100 % classification accuracy in both cases. Due to the small size of the tiny-set the execution time was only 0.03 seconds, which was much faster than the 8.64 seconds for the large data-set. This difference stems from the fact that the large toydata set contains 4000-times the amount of features. The classification accuracy for the MNIST data-set reached 92.65 % with an execution time of 175.32 seconds. Based on the exceptionally high dimensionality of the data-set this was expected.

Random Fourier Features

The Random Fourier Features generated for this task (can be seen in $RFF.py$) follow the principles presented in the lecture, but with a slightly different implementation that leans more on efficiency. The random weights and biases are therefore generated the following way

Dataset	Accuracy [%]	Execution Time [s]
Toydata Tiny	100.0	0.03
Toydata Large	100.0	8.64
MNIST	92.65	175.32

Table 2: Classification accuracies using LinearSVM for all three data-sets, evaluated on the corresponding test set. Both toy data test-sets appear to be perfectly linearly separable, with the mini-batch gradient descent converging very quickly to the optimal solution. The execution times were measured during training, using 30, 10 and 3 epochs for toydata_tiny, toydata_large and MNIST, respectively. Figure 3 shows the visual representation of the computed margins and the classification accuracy for the tiny toydata set.

$$\text{weights} = 1/\sigma \cdot \mathcal{N}(n\_features,\ n\_components)$$
$$\text{biases} = \mathcal{U}(0, 2\pi,\ n\_components)$$

with σ being a regularization factor, $\mathcal{N}$ drawing from a normal distribution and $\mathcal{U}$ drawing from a uniform distribution. The parameters σ and $n\_components$ are chosen by the user, while the rest is fixed to ensure the right dimensions for the transformation of the data and computation of the kernel matrix. The transformation and kernel are calculated with

$$\text{Projection}(X, \mathbf{w}, b) = \frac{\sqrt{2}}{\sqrt{n\_components}} \cos\left(X \cdot \mathbf{w} + b\right)$$
$$\text{Kernel}(Z) = Z \cdot Z^T,\ Z \text{ being the Projection calculated above}$$

Testing these steps on random data with dimensions (1000, 200) against the explicit RBF kernel used by $sklearn$, shows that with higher RFF components the difference between the approximated kernel and the true RBF kernel gets negligible. With this confirmation that the implementation of the RFF is sufficiently good at approximating the RBF kernel, the same tests as with the linear SVM can be done with the RFF transformed data.

The hyperparameter search is done in a similar fashion as with the regular linear SVM but with the RFF transformed features instead, using $n\_components \in \{100, 250, 400\}$ and $\sigma \in \{0.33, 0.66, 1\}$ as additional parameters for the cross validation. Contrary to the regular approach of choosing $\sigma = 1/n\_labels$, the $MNIST$ dataset is a special case where higher numbers of components and values for sigma are required to achieve comparable results. In this case $n\_components \in \{200, 400, 600\}$ and $\sigma \in \{15, 20, 25\}$ is used instead. Table 3 shows the optimal parameters using 5-fold cross validation with the full results available in *test rff.ipynb*.

Dataset	η_0	C	$batch\_size$	$n\_components$	σ	CV-score [%]
Toydata Tiny	3	0.01	10	100	1	100.0
Toydata Large	0.1	0.001	50	250	1	100.0
MNIST	3	1	100	600	20	87.50

Table 3: Optimal hyper-parameters for all three data-sets, evaluated based on the corresponding cross-validation score on the validation set. η_0 refers to the initial learning rate, C to the regularization parameter, $batch\_size$ to the batch size, $n\_components$ to the number of RFF components, and σ to the regularization parameter for the RFF during optimization with the mini-batch GD.

Compared to the regular linear SVM, the RFFs tend to favour smaller batch sizes with 100 - 250 components depending on the size of the Toydata set. MNIST prefers a higher initial learning rate and SVM regularization factor of $C = 1$, along with generally higher number of RFF components and large sigma. Although the CV-score for the MNIST data set is around 4% lower with RFF, this doesn't seem to affect the accuracy on the test set that much. For both the Toydata sets the amount of RFF components isn't that much of a factor for the performance if sigma is kept at 1, since they only have two features to begin with and projecting them into multiple hundreds of RFFs doesn't affect their results too much. For MNIST on the other hand, choosing too little components has adverse effects on the accuracy, since projecting the nearly 800 features down to am small amount of RFFs can lead to inaccuracies. From testing, it appears that around 400 - 600 RFFs with a high value for sigma seems to be ideal for this particular data set, which is reflected in the results.

Dataset	Accuracy [%]	Execution Time [s]
Toydata Tiny	100.0	0.04
Toydata Large	100.0	9.57
MNIST	90.55	182.19
MNIST (sklearn)	91.25	~ 12

Table 4: Classification accuracies for all three data-sets with Random Fourier Features, evaluated on the corresponding test set.

As seen in Table 7, the results are basically the same as for the regular linear SVM. Due to some initial inconsistencies with the home brew multi class SVM, a One Vs Rest Linear Classifier from $sklearn$ is also used as a sanity check. Apart from the not even comparable execution time, the results between our own implementation of multi class SVM and sklearn agree to a very high degree.

Parallel Implementation

In order to complete this part of the assignment, instead of running obtained model sequentially and calculating runtime we rather opted-in for developing **SimuParallelSGD** with real parallel implementation.

To accomplish the task, we have used Python *threading* library. Our model accepts *thread_count* as parameter, under-the-hood train data is split into equal chunks and each thread trains its own chunk in parallel way.

As defined in the algorithm, resulting weights are obtained from each thread, final value is computed as average.

Intending to have one flexible model, we have decided to extract *Stochastic Gradient Descent*, thus our LinearSVM model could work with different implementations of it. By default, if *thread_count=1* is passed to our model, Mini-batch SGD discussed earlier in the report is applied, otherwise - SimuParallelSGD.

To afford even more flexibility, each SGD model accepts loss and accuracy functions as arguments.

Optimization

Parallel implementation required us to think about excluding all probable data-race/syncronization issues. To prevent those, we have optimized threads to cache/copy values from main thread and made sure global variables are accessed as less as possible.

Each thread has it's own copy of training data, resulting weights are written to global variable towards the end of thread execution.

Testing

During testing on *toydata_large* dataset it was clear that the biggest bottleneck was accuracy and loss computation. We have introduced some optimizations to reduce amount of those, as well as an option to disable them in the training process

Models comparison

There are different possible approaches to compare obtained models, we decided map parameters with each other:

Mini-batch SGD parameter	SimuParallelSGD parameter
epoch_count	thread_count
batch_size	examples_per_thread

Table 5: Mapping between Mini-batch SGD and SimuParallelSGD parameters

while rest parameters are the same, including learning_rate and regularization.

Runtime

For runtime tests *toydata_large* was used, thread_count varying from 1 to 8. All calculations were done using optimal parameters from Part 1.

	2 epochs	4 epochs	8 epochs	2-threads	4-threads	8-threads
Runtime (sec)	01.819	02.785	04.840	00.967	01.192	01.537
Accuracy	0.999	0.999	0.999	1.0	1.0	1.0

Table 6: Comparison between Mini-batch SGD and SimuParallelSGD using toydata_large

Averagely, accuracy with SimuParallelSGD is a little better that regular SGD. Considering runtime, as model comparison section suggests, we can see 2x speedup for 2 epochs vs 2 threads, 2.5x for 4 and 3.3x for 8.

Given this, we can also calculate runtime for MNIST dataset

	2 epochs	4 epochs	8 epochs	2-threads	4-threads	8-threads
Runtime (sec)	103	201	395	$\sim$51	$\sim$80	$\sim$119

Table 7: Comparison between Mini-batch SGD and SimuParallelSGD using MNIST

References

[1] Prof. Sebastian Tschiatschek. *Introduction to Machine Learning. Multi-class problems*. Slides 18-19, November 16, 2020.

Contributions

Task	E. Guliev	J. Rass	C. Wiskott
Initial Project Setup	✓	✓	✓
Task distribution planning	✓	✓	✓
Linear SVM Model			✓
Random Fourier Features		✓	
Parallel Implementation	✓		
Testing, Evaluating accuracy	✓	✓	✓
Report	✓	✓	✓

As a side note for the contributions, the task responsibility was better communicated and carried out compared to the last assignment, with one group member developing one sub task and a combined merging and testing effort.

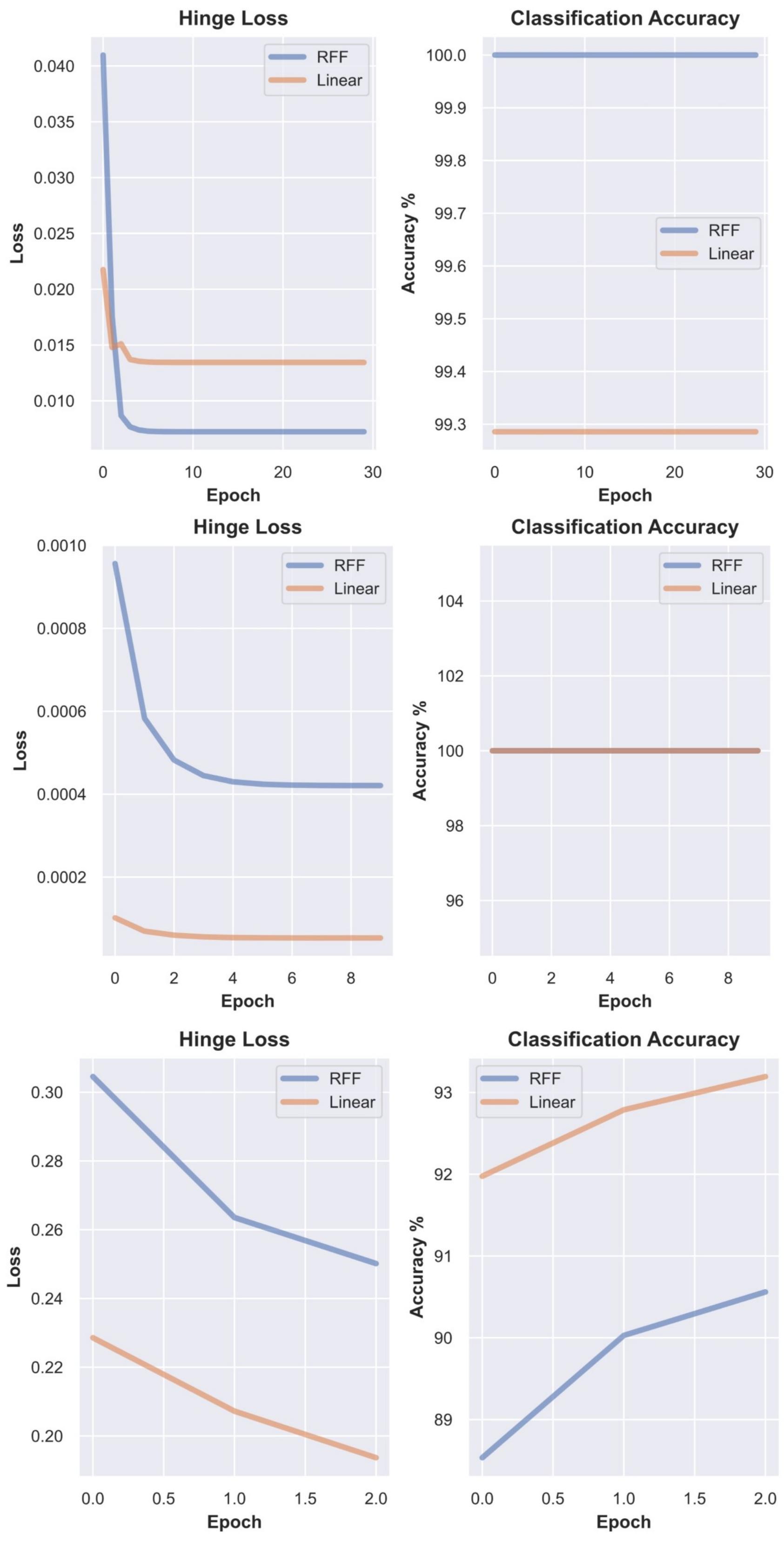

Figure 2: Convergence behaviour for all three datasets showing the hinge loss and the classification accuracy for each epoch during optimization. From top to bottom: *Tiny, large, MNIST*. Compares the performances of using RFF vs. original features. In the case of the tiny dataset the RFFs seem to provide a better classification accuracy although with a small margin. The results for the large dataset are comparable. For the MNIST dataset however, the loss as well as the classification accuracy seem to benefit more from using the original features rather than the RFFs.

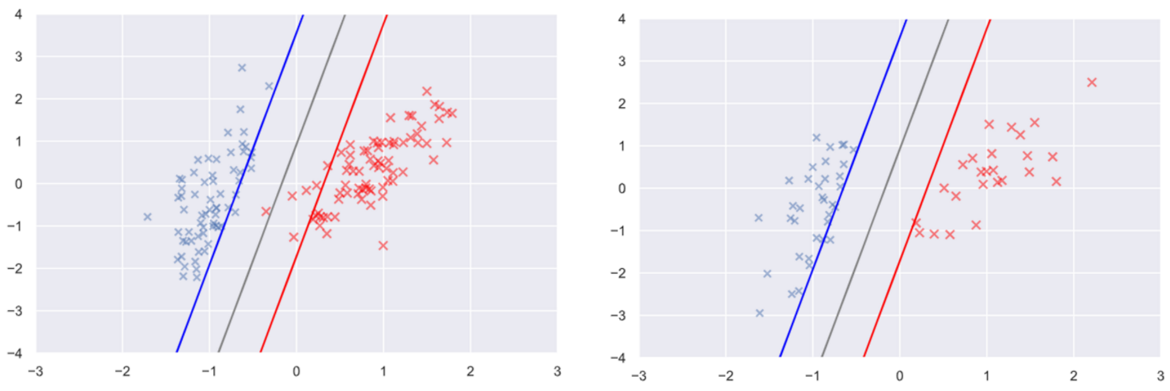
Figure 3: Computed margins for the tiny data set. The left figure shows the training data and its respective margin that were computed during training. The right figure shows the test set with the superimposed margins. This illustrates the perfect linear separation of the tiny test-set.

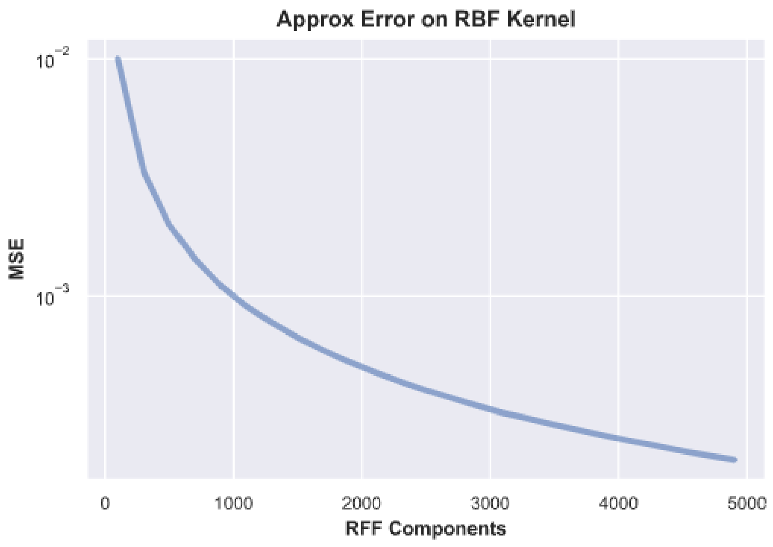

Figure 4: Error between the RFF approximated kernel and the true RBF kernel.