SVM for digit recognition

Matteo Cavada

January 23, 2023

Contents

1 Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

2 Abstract

In this project I implemented the kernelized PEGASOS algorithm in Python. The algorithm was then trained for the task of images recognition, specifically *hand-written digit recognition*. A number of combinations of hyperparameters were tested; lowest levels of test error were obtained when using either a Gaussian or a high-degree polynomial kernel.

3 Dataset and pre-processing

3.1 Dataset

The dataset is available at kaggle.com.

The USPS dataset [1] *"is a digit dataset automatically scanned from envelopes by the U.S. Postal Service"*.

The images are all in grayscale, with size 16x16. Each pixel is a continuous value between 0 (black) and 1 (white).

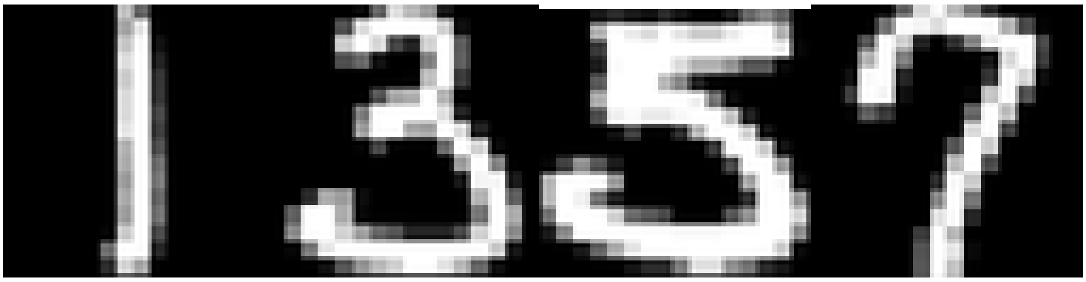

Figure 1: Examples of few selected digits from the dataset

3.2 Preprocessing of data

The data is stored in a single binary file using the HDF5 format; the extraction of data from the file is achieved through the python library h5py.

Once retrieved, each image is stored as a NumPy[1] [2] array; the label of each image is stored as a simple number.

A few, basic sanity checks were implemented to verify that the dataset is well-formed (ie. all labels are numbers from 0 to 9).

4 Training

The algorithm used is **Pegasos** with a Gaussian or polynomial kernel.

4.1 Theoretical background

4.1.1 An overview of SVMs

Support Vector Machines (SVM)[3] are a machine learning technique used to learn linear classifiers. The idea behind them can be explained geometrically: on a linearly-separable training set, there exists a (hyper-)plane which divides the dataset in two, so that every point in a given half-space is associated to the same label. SVMs finds this separating plane by maximizing the distance between the hyperplane and the closest data points to it.

More formally, the optimal hyperplane (also known as *maximum-margin separating hyperplane*) can be found by solving the following convex optimization problem:

$$\begin{array}{cc} \underset{\boldsymbol{w}\in\mathbb{R}^n}{max} & \frac{1}{2}\left\|\boldsymbol{w}\right\|^2 \\ s.t. & y_t\boldsymbol{w}^T\boldsymbol{x}_t \geq 1 \end{array} \tag{1}$$

This formulation of the problem works only if the training data are linearly separable. In the case of a non-linearly separable dataset, we change the constraints above by adding *n slack variables* ξ_i which, informally, encode how serious the violation of the margin constraint is for the datapoint x_i. More formally, the optimization problem now becomes:

$$\begin{array}{cc} \underset{\boldsymbol{w}\in\mathbb{R}^n}{max} & \frac{\lambda}{2}\left\|\boldsymbol{w}\right\|^2 + \frac{1}{m}\sum_{t=1}^{m}\xi_t \\ s.t. & y_t\boldsymbol{w}^T\boldsymbol{x}_t \geq 1 - \xi_t \\ & \xi_t \geq 0 \end{array} \tag{2}$$

Through simple algebraic manipulations, it can be seen that the slack ξ_t is equivalent to $max(0, 1 - y_t\boldsymbol{w}^T\boldsymbol{x}_t)$, i.e. the hinge loss of $\boldsymbol{w}$ on $(\boldsymbol{x}_t, y_t)$. Also note the introduction of the regularization parameter λ, used to balance the two terms in the objective function.

4.1.2 PEGASOS

The PEGASOS algorithm [4], by means of stochastic gradient descent, solves the optimization problem given by the SVM objective function.

It can be proved that the solution to such optimization problem is, in fact, the solution $\boldsymbol{w}^*$ of the following equation:

$$\underset{\boldsymbol{w}}{minarg}\ \frac{\lambda}{2}\left\|\boldsymbol{w}\right\|^2 + \frac{1}{m}\sum_{(\boldsymbol{x},y)\in S} l_{hinge}(y, \boldsymbol{w}^T\boldsymbol{x}) \tag{3}$$

[1] https://numpy.org/

where $l_{hinge}(y, \boldsymbol{w}^T \boldsymbol{x})$ is defined as $max(0, 1 - y\boldsymbol{w}^T \boldsymbol{x})$, m is the number of points in the dataset, and S is the dataset itself. This holds both in case of linearly and non-linearly separable dataset.

Since the above equation can be equivalently rewritten as:

$$\underset{w}{minarg} \ \frac{1}{m} \sum_{t=1}^{m} l_t(\boldsymbol{w}) \tag{4}$$

with $l_t(\boldsymbol{w}) = \frac{\lambda}{2} \|\boldsymbol{w}\|^2 + l_{hinge}(y_t, \boldsymbol{w}^T \boldsymbol{x}_t)$, and since l_t is a convex and differentiable function for each t, we can apply Online Gradient Descent (OGD) to solve the SVM optimization problem.

As the loss functions l_t are also λ -strongly convex, we can perform an OGD without the projection step.

4.1.3 Kernelized PEGASOS

The representer theorem shows that the solution $\boldsymbol{w}^*$ to (4) must be a linear combination of the training datapoints; that allows us to deploy the machinery given by PEGASOS to a Reproducing Kernel Hilbert Space (RKHS) of our choice. In practice, we are able to work in higher-dimensional data spaces without incurring in significant performance losses.

Specifically, we can write the former $\boldsymbol{w}^*$ as:

$$\sum_{s \in S} \alpha_s \, y_s \, K(x_s, \cdot) \tag{5}$$

This core idea allows us to write a kernelized version of PEGASOS; the Python implementation is found in the next section, along with some comments.

4.2 Implementation

4.2.1 Training algorithm

Below is a Python implementation of the kernelized PEGASOS algorithm. Note that this version differs slightly from the actual implementation found in the repository, as few debug instructions were removed and the return type is changed; the core, though, remains the same.

```python
def pegasos(X, Y, for_digit, kernel, lambd, T) -> Callable:
  alpha = list()

  for t in range(T):
      idx   = random.randint(0, len(X)-1)
      x, y = X[idx], Y[idx]

      y = 1 if y == for_digit else -1

      prediction  = 1 / (lambd * t)
      prediction *= sum(ys * kernel(xs, x) for xs,ys in alpha)

      if y * prediction < 1:
        alpha.append((x,y))

  return lambda x: (
      sum(ys * kernel(xs, x) for xs,ys in alpha)
  )
```

This algorithm builds the predictor for a given value from 0 to 9 (`for_digit`).

The `kernel` argument is a Python function that calculates a given kernel: it takes two numpy vectors as arguments and returns a float. Passing such a function as argument to `pegasos` avoids writing boilerplate code for each different kernel I wanted to use.

`lambd` is the λ parameter which influences the learning rate (see An overview of SVMs). `lambd` varies from 10^{-8} to 10^{-5} during during the execution of the *k-fold* (see: On the choice of λ).

`T` is the number of training points that will be used.

The core of the algorithm iterates `T` times. Each iteration:

1. Picks a random training point and its label, `(x,y)`.
2. Transforms `y` in one of `{-1, 1}` based on `for_digit`.
3. For the prediction formula, see Kernelized PEGASOS.
4. When the prediction is incorrect, the wrongfully predicted pair `(x,y)` is put in the list `alpha`

The return value is a function, which takes a data-point and performs a prediction by iterating on all the points in `alpha`.

4.2.2 Multi-class training and predictions

As the algorithm above produces a binary predictor for a *single* digit, there must be a way to choose which is the "best fitting" among ten different prediction.

The approach used in this project is to choose, as the best fitting prediction, the one whose absolute value is greatest among all other predictions. This explains why the algorithm above does not return a binary $\{1, -1\}$ predictor, but instead returns directly a continuous value.

5 Testing

5.1 K-Fold cross-validation

In order to evaluate and compare the performance of PEGASOS on a number of different hyperparameters combinations, the 5-fold external cross validation technique was used.

For each combination:

1. The dataset is randomly shuffled
2. The first fifth of the dataset is used as testset, while the remaining part is used as training set
3. Predictors are generated for the given training set and the actual test error is calculated
4. The dataset is rotated (ie. each index is shifted to the right, so that the last fifth of the dataset is put at the beginning of the dataset);
5. GOTO 3 until you have rotated the dataset in the exact initial position

Code for this is mostly found in the `kfold.py` file.

5.2 Hyperparameters

A number of possible combinations of hyperparameters could be tested. Through a *grid search*, I decided to vary:

- The kernel used (see: Kernels):
  - Gaussian Kernel with parameter $\gamma = 2$
  - Polynomial Kernel with parameters $exp = 1$
  - Polynomial Kernel with parameters $exp = 3$
  - Polynomial Kernel with parameters $exp = 7$
- The training epochs (T is the size of the training dataset):
  - $\frac{T}{10}$
  - $\frac{T}{2}$
  - T
  - $2T$
- The λ parameter:
  - 10^{-8}
  - 10^{-7}
  - 10^{-6}
  - 10^{-5}

5.2.1 On the choice of kernel function

A number of possible kernel functions can be used for kernelized learning algorithms. Besides the suggested Gaussian Kernel, in this project I experimented with polynomial kernels with various exponents.

Formally, the Gaussian kernel has the following definition:

$$K_{gauss}(\mathbf{x}_1, \mathbf{x}_2) = exp(-\frac{1}{2\gamma^2} \parallel \mathbf{x}_1 - \mathbf{x}_2 \parallel^2) \tag{6}$$

γ is an hyperparameter of the kernel; in my code, I set $\gamma = 2$.

The polynomial kernel has the following formulation:

$$K_{poly}(\mathbf{x}_1, \mathbf{x}_2) = (1 + \mathbf{x}_1^T \mathbf{x}_2)^n \tag{7}$$

n is the hyperparameter for this kernel. In the code, I experimented with values $n = 1, 3, 7$. Likely, bigger values for the polynomial degree would have incurred in overfitting.

5.2.2 On the choices of λ

In [4] a number of experiments are run with λ values ranging from 10^{-6} to 10^{-4}; it is also suggested that, precisely for experiments on the USPS dataset, smaller values of λ do not improve the quality of the generated predictor.

In order to test whether this is true or not, I decided to range λ from 10^{-8} to 10^{-5}.

It is important to notice, however, that the experiments and the results shown in [**?**] are related to the task of generating a predictor only for the digit "8", and that only a gaussian kernel was used.

5.3 Results

The best predictors were those which were trained on a bigger number of training points and with bigger values of λ.

Almost each kernel was able to reach a low test error (less than 0.05) when these conditions were met. The only exception is the linear kernel, which underperforms if compared to the other kernels.

The value of λ seems to be inversely proportional to the test error of a generated predictor. However, as shown in appendix 2 and especially for the gaussian kernel, a bigger value of λ resulted in quite longer execution times.

For graphical representations of the results, see appendix 1.

5.4 Training times

As the code was not run on dedicated hardware such as GPUs, training times are somewhat long.

Also, the Kernelized PEGASOS algorithm suffers from performance issues when training on big datasets: as the number of wrongful prediction during training increases, the number of subsequent kernel calculations increases as well.

Empirically, the gaussian kernel was the slowest one to train; at parameters $\lambda = 0.5$ and $2T$ epochs, the predictor for a single digit was calculated in 36 seconds (on average) – it gives the smallest test error at the expense of speed of execution.

Averages for each hyperparameter combination are found in the file `results/results.csv` on the repository. A table, summarizing the average training time for each combination of hyperparameter can be found in appendix 2.

6 Bibliography

References

[1] J. J. Hull *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

[2] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, “Array programming with NumPy,” *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[3] C. Cortes and V. Vapnik, “Support vector networks,” *Machine Learning*, vol. 20, pp. 273–297, 1995.

[4] S. Shalev-Shwartz, Y. Singer, and N. Srebro, “Pegasos: Primal estimated sub-gradient solver for svm,” in *Proceedings of the 24th International Conference on Machine Learning*, ICML ’07, (New York, NY, USA), p. 807–814, Association for Computing Machinery, 2007.

7 Appendix 1: test errors

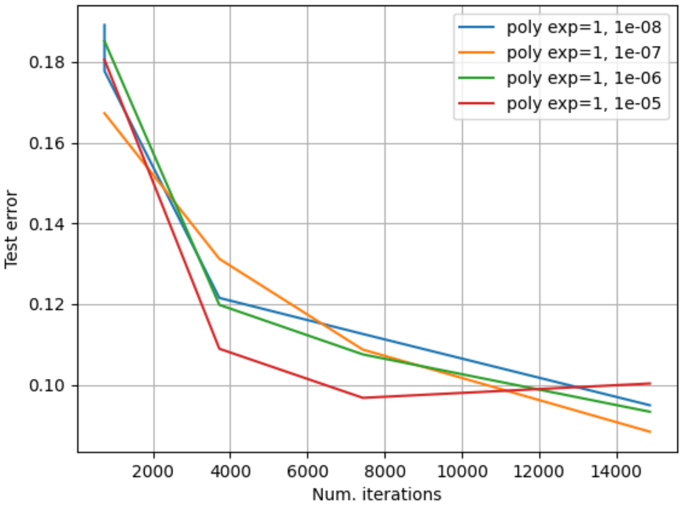

Figure 2: Iterations vs test error on Polynomial Kernel, exp=1

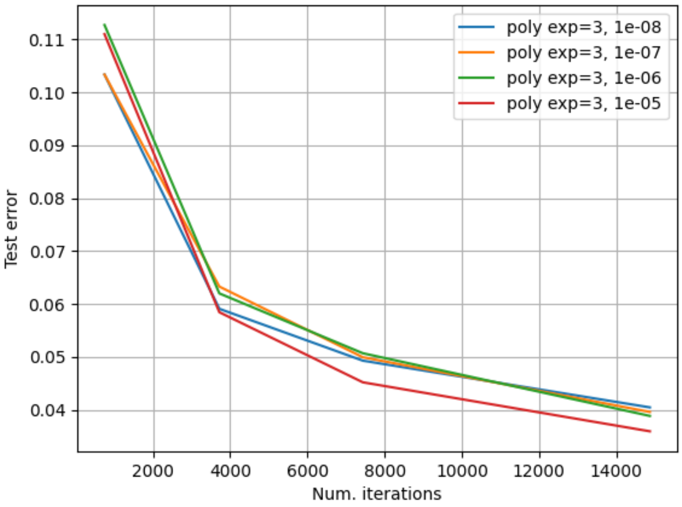

Figure 3: Iterations vs test error on Polynomial Kernel, exp=3

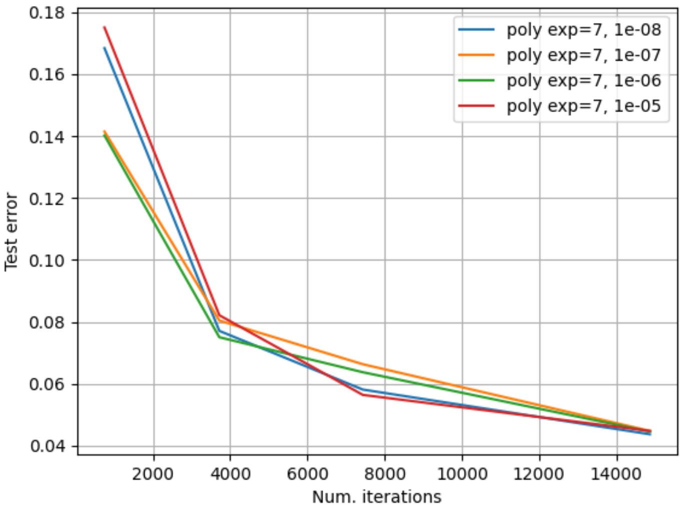

Figure 4: Iterations vs test error on Polynomial Kernel, exp=7

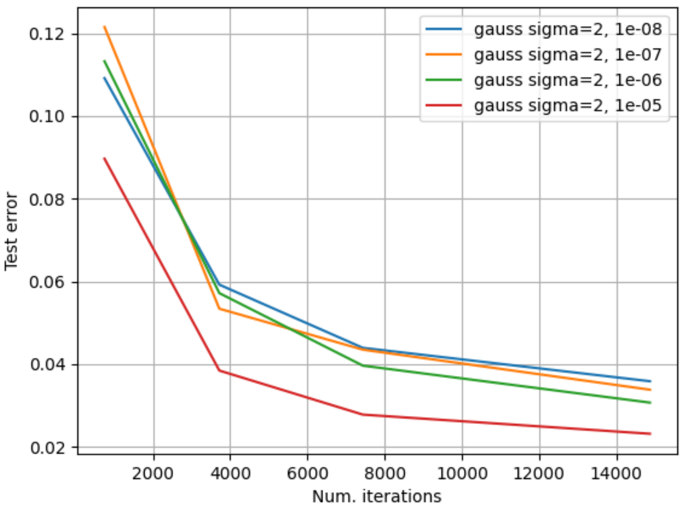

Figure 5: Iterations vs test error on Gaussian Kernel

7.1 Tabular form

Poly, e=1	T/10	T/2	T	2T
1.00E-08	0.1777	0.1215	0.1126	0.0949
1.00E-07	0.1673	0.1312	0.1087	0.0884
1.00E-06	0.1851	0.1198	0.1075	0.0933
1.00E-05	0.1805	0.1089	0.0968	0.1003

Poly, e=3	T/10	T/2	T	2T
1.00E-08	0.1033	0.0591	0.0493	0.0405
1.00E-07	0.1033	0.0633	0.0499	0.0396
1.00E-06	0.1127	0.0620	0.0507	0.0389
1.00E-05	0.1110	0.0584	0.0452	0.0360

Poly, e=7	T/10	T/2	T	2T
1.00E-08	0.1684	0.0771	0.0581	0.0437
1.00E-07	0.1414	0.0804	0.0663	0.0448
1.00E-06	0.1402	0.0750	0.0637	0.0446
1.00E-05	0.1750	0.0821	0.0564	0.0448

Gauss	T/10	T/2	T	2T
1.00E-08	0.1091	0.0592	0.0439	0.0358
1.00E-07	0.1215	0.0534	0.0435	0.0338
1.00E-06	0.1132	0.0572	0.0396	0.0307
1.00E-05	0.0897	0.0384	0.0278	0.0231

Figure 6: Test errors on various kernels; the columns represent the number of epochs, while the rows indicate the λ values

8 Appendix 2: train and test times

Poly, e=1	T/10	T/2	T	2T
1.00E-08	3	19	52	155
1.00E-07	3	19	58	142
1.00E-06	3	19	56	131
1.00E-05	3	19	52	134

Poly, e=3	T/10	T/2	T	2T
1.00E-08	2	12	28	70
1.00E-07	2	12	28	70
1.00E-06	2	12	28	70
1.00E-05	2	12	29	72

Poly, e=7	T/10	T/2	T	2T
1.00E-08	3	14	32	77
1.00E-07	3	14	33	78
1.00E-06	2	14	32	76
1.00E-05	3	14	33	78

Gauss	T/10	T/2	T	2T
1.00E-08	6	27	61	141
1.00E-07	6	28	63	145
1.00E-06	6	32	74	186
1.00E-05	9	59	155	443

Figure 7: Average training + test time (in seconds) of each combination of hyperparameters; the columns represent the number of epochs, while the rows indicate the λ values.