Machine learning, statistical learning, deep learning and artificial intelligence - Machine learning 2022/2023

PEGASOS implementation from scratch

Support Vector Machines algorithm with Stochastic Gradient Descent optimization

Data Science and Economics

Alessandro Mirone

966880

UNIVERSITÀ DEGLI STUDI DI MILANO

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.

Introduction and data description

The present work aims at implementing a Pegasos algorithm in python 3 language to perform multiclass classification of handwritten digits to numerical labels identifying the correct digit, and assess the predictor accuracy. The analysis is organized as follows:

- Introduction and data description
- Analytical description of the algorithm
- Kernel trick: dealing with non-linearly separable data
- Code description and implementation
- Results

The implementation of the algorithm is written in Python base language, that is, no libraries are used apart from `h5py` for data ingestion, `Pandas` , `NumPy` and `copy` for data manipulation and `random` for sampling.

The object of the analysis are images of handwritten digits: each image is described by 256 bytes and they are rendered in greyscale, thus each pixel is represented by just eight bits of information as opposed to 24 bits in case of RGB images. Consequentially, a byte describes the luminous intensity of the corresponding pixel by a number ranging from 0.0000 (black) to 1.0000 (white). An extra vector of numbers, ranging from 0 to 9, is attached to the data, expressing the already assigned right label for each digit. In total, the complete dataframe has dimensions 9298 x 257, i.e. there are 9298 examples of handwritten digits in the data set. First and last five rows of the data set are visualized below as illustration.

In [3]:
```
#dataset
df=pd.DataFrame(np.concatenate((np.column_stack((X_tr,y_tr)),np.column_stack((X_te,y_te))), axis = 0))
df
```

Out[3]:

	0	1	2	3	4	5	6	7	8	9	...	247	248	249	250	251	252	253	254	255	256
0	0.0	0.0	0.0	0.0000	0.0000	0.0000	0.0000	0.1845	0.9310	0.4165	...	0.9115	1.0000	0.7410	0.2630	0.0045	0.0000	0.0000	0.000	0.0	6.0
1	0.0	0.0	0.0	0.0935	0.1645	0.0955	0.0565	0.1645	0.0735	0.0000	...	0.1645	0.4835	0.8805	0.8810	0.5630	0.4525	0.1645	0.086	0.0	5.0
2	0.0	0.0	0.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0020	...	0.0000	0.0000	0.4455	1.0000	0.4105	0.0000	0.0000	0.000	0.0	4.0
3	0.0	0.0	0.0	0.0000	0.0000	0.3635	0.8420	0.9800	0.7250	0.4665	...	1.0000	0.7680	0.0065	0.0000	0.0000	0.0000	0.0000	0.000	0.0	7.0
4	0.0	0.0	0.0	0.0000	0.0000	0.0360	0.3980	0.8755	0.7330	0.6170	...	0.8195	1.0000	1.0000	0.8955	0.7195	0.4005	0.0585	0.000	0.0	3.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
9293	0.0	0.0	0.0	0.0000	0.0000	0.2915	0.9070	1.0000	0.8875	0.1385	...	1.0000	0.7770	0.5920	0.2580	0.0000	0.0000	0.0000	0.000	0.0	3.0
9294	0.0	0.0	0.0	0.0000	0.0000	0.0000	0.0725	0.6045	0.9705	0.9045	...	0.6595	1.0000	0.5280	0.0000	0.0000	0.0000	0.0000	0.000	0.0	9.0
9295	0.0	0.0	0.0	0.0000	0.4845	0.8760	0.2845	0.0000	0.0000	0.0000	...	0.9640	0.3035	0.0000	0.0000	0.0000	0.0000	0.0000	0.000	0.0	4.0
9296	0.0	0.0	0.0	0.0000	0.2330	0.7890	1.0000	0.6505	0.3360	0.0305	...	0.7150	0.6000	0.4695	0.0125	0.0000	0.0000	0.0000	0.000	0.0	0.0
9297	0.0	0.0	0.0	0.0000	0.0000	0.0000	0.0000	0.6995	0.9745	0.3005	...	0.2475	0.9535	0.8275	0.0250	0.0000	0.0000	0.0000	0.000	0.0	1.0

9298 rows × 257 columns

The algorithm will be implemented in a gaussian kernel space to deal with non-linearly separable data. Finally, a 5-fold cross validation will be used to evaluate the multiclass classification risk (with zero-one loss) of the method for 25 different combinations of hyperparameters. Note that, since Pegasos algorithm output binary predictors, the final multiclass predictors are constructed following a one-vs-all approach, involving the creation of 10 binary predictors for each run of cross-validation.

Analytical description of the algorithm

This section describes the algorithm implemented. In order to do so, some arguments need to be presented first; the content is then furtherly divided into four subsections:

1. Linear predictors
2. Convex optimization with Online Gradient Descent
3. Support Vector Machines

4. PEGASOS

Each of the first three subsection will briefly introduce some core concepts necessary to understand the algorithm's description, presented in the fourth one. Information used to write this section are taken from the module "Machine Learning", held by Prof. N. Cesa Bianchi, of the Master degree "Data Science and Economics" ' course: "Machine learning, statistical learning, deep learning and artificial intelligence" (lecture notes 9,10,12; 2020/2021/2022).

1) Linear Predictors:

Pegasos algorithm outputs a linear predictor h which is a function

$$h : \mathbb{R}^{\mathrm{d}} \to \mathbb{R} \; such \; that \; h(\mathbf{x}) = f(\mathbf{w}^T\mathbf{x}) \; for \; some \; \mathbf{w}, \mathbf{x} \in \mathbb{R}^{\mathrm{d}}$$

(hence the name *linear*). The function f, sometimes called *activation function* is, in the case of binary classification, the sign function

$$sign : \mathbb{R} \to \{1, 0\} \; such \; that \; sign(z) = 1 \iff z > 0 \; and \; 0 \; otherwise$$

then, $h(\mathbf{x}) = sign(\mathbf{w}^T\mathbf{x})$. The vector $\mathbf{w}$ is orthogonal to the homogeneous hyperplane defined by $\{\mathbf{x} \in \mathbb{R}^d : \mathbf{w}^T\mathbf{x} = 0\}$ and $\mathbf{w}^T\mathbf{x}$ is the projection of the data point $\mathbf{x} \in \mathbb{R}^d$ onto the vector $\mathbf{w}$. Finally note that, being $\mathbf{w}^T\mathbf{x}$ a dot product between two vectors (for $d > 1$), $\mathbf{w}^T\mathbf{x} \in \mathbb{R}$. This argument allows to write loss functions

$$l : \mathbb{R} \to \{1, 0\} \; such \; that \; l(y, h(\mathbf{x})) = 0 \iff sign(y) = sign(\mathbf{w}^T\mathbf{x}) \; and \; 1 \; otherwise$$

where y is the binary label of the data point $\mathbf{x}$. This *naive loss function* would be suited for linearly separable data, but even in this case finding the $\mathbf{w}$ that minimizes $l(y, h(\mathbf{x}))$ is NP-hard. Instead, the loss function can be tweaked to include slack variables ξ to allow for more flexibility in the non-linearly separable case and at the same time obtain a strongly convex and differentiable function that can be optimized with online gradient descent.

2) Convex optimization with Online Gradient Descent:

Gradient Descent (GD) is an optimization algorithm for finding points of local minima that can be applied to any differentiable - and therefore continuous - multi-variable function f. It works by computing the gradient of the function for a given point $\mathbf{a}_t$, thus identifying the direction along which the function's value increases the most, and then taking a "step" in the opposite direction, obtaining a new point in which the function has the maximum decrease in value: $\mathbf{a}_{t+1} = \mathbf{a}_t - \eta \bigtriangledown f(\mathbf{a}_t)$.

By iteratively computing this operation, the method can converge to a point of local minimum in the domain of f. If certain assumptions on the function f are introduced, GD is guaranteed to identify a point of global minimum, these assumption being the convexity of f (which means that all local minima are also global minima), the Lipschitz continuity of the function's gradient and a suitable choice of the parameter η, known as *learning rate*, which controls the size of each step.
Algorithms that learn sequentially (that is, they process each training point sequentially and optimize the loss function accordingly, generating a sequence of losses), like Support Vector Machines, can be optimized with a modification of the original GD, called **Online gradient Descent (OGD)** which accounts for the sequentiality of the inputs and avoids the need to retrain the learner for each new point added to the training set; in this way, the algorithm needs only to know the current point's coordinates and the previous value of the predictor in order to compute the new predictor. In sequential learning, because at each step a new predictor is generated, the loss function to be minimized changes as well with each iteration (note that the form of the function stays the same, what changes is its value); the performance of the algorithm is evaluated through the *sequential risk*:

$$\frac{1}{T}\sum_{t}^{T} l(y_t, h_t(\mathbf{x}_t))$$

where $h_t(\mathbf{x}_t) = sign(\mathbf{w}_t^T \mathbf{x}_t)$. OGD deals with this situation by adding a projection step to the GD protocol. In the case of linear predictors $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ and for a sequence of convex and differentiable losses $l_1, l_2, \dots$ of the form $l_t(\mathbf{w}_t) = l(y_t, \mathbf{w}_t^T\mathbf{x}_t)$, denoting the loss at iteration t of the predictor h_t based on the current value of $\mathbf{w}$, OGD method works as follows:

OGD with Projection step:

for $t = 1, 2, \dots$; for $\eta > 0$; for $U > 0$
$\mathbf{w}_1^T = [0, \dots, 0]$

1. $\mathbf{w}'_{t+1} = \mathbf{w}_t - \frac{\eta}{\sqrt{t}} \bigtriangledown l_t(\mathbf{w}_t)$
2. $\mathbf{w}_{t+1} = \underset{\mathbf{w}:||\mathbf{w}|| \leq U}{\operatorname{argmin}} ||\mathbf{w} - \mathbf{w}'_{t+1}||$

Let $\eta_t = \frac{\eta}{\sqrt{t}}$ where η is found through experimentation; note then that the first step is the same as the regular GD. In the second step, the candidate vector $\mathbf{w}'_{t+1}$ is projected to an Euclidean sphere of radius U centered at the origin, which contains the best predictor $\mathbf{u}_t^*$ in the class H of predictors; in fact we have that

$$\mathbf{u}_t^* = \underset{\mathbf{u}:||\mathbf{u}||\leq U}{\operatorname{argmin}} \frac{1}{T}\sum_{t=1}^{T} l_t(\mathbf{u})$$

If $||\mathbf{w}'_{t+1}|| \leq U$, then $\mathbf{w}_{t+1} = \mathbf{w}'_{t+1}$. With this method, whatever the case, $H = \{\mathbf{w} \in \mathbb{R}^d : ||\mathbf{w}|| \leq U\}$ for $U > 0$: the predictor will always be contained in the sphere of radius U and it can be shown that the sequential risk of OGD, computed based on the predictor $\mathbf{w}_t$, is converging to the sequential risk computed for the best predictor $\mathbf{u}_t^*$ at the rate $UG\sqrt{\frac{8}{T}}$, where G is a constant that depends on specific assumptions about the examples and the loss function used, for any number of iterations T and for any $\mathbf{u}$ inside the ball of radius U; that is:

$$\frac{1}{T}\sum_{t=1}^{T} l_t(\mathbf{w}_t) \leq \min_{\mathbf{u}:||\mathbf{u}||\leq U} \frac{1}{T}\sum_{t=1}^{T} l_t(\mathbf{u}) + UG\sqrt{\frac{8}{T}} \tag{1}$$
$$\forall \mathbf{u} : ||\mathbf{u}|| \leq U \; and \; \forall T \in \mathbb{R}$$

However, when using **strongly convex** loss functions, the projection step of the OGD is no longer required as it can be shown that

$$\frac{1}{T}\sum_{t=1}^{T} l_t(\mathbf{w}_t) \leq \min_{\mathbf{u}\in\mathbb{R}^d} \frac{1}{T}\sum_{t=1}^{T} l_t(\mathbf{u}) + \frac{G^2}{2\sigma}\frac{\ln(T+1)}{T} \tag{2}$$
$$\forall \mathbf{u} \in \mathbb{R}^d and \; \forall T \in \mathbb{R}; \; with \; \sigma > 0$$

The result in (2) holds for any σ-strongly convex and differentiable function; note that the rate of convergence in this case is improved with respect to (1).

3) Support Vector Machines

Support Vector Machines (SVM) is a type of sequential learning algorithm that outputs binary linear predictors of the type seen above in section 1. Classical SVM is appliable to linearly separable datasets, as, given a (linearly separable) training set with binary labels $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m) \in \mathbb{R}^d\} \times \{(1, -1)\}$, SVM output the linear classifier corresponding to the unique solution $\mathbf{w}^* \in \mathbb{R}^d$ of the following convex optimization problem with linear constraints:

$$\min_{\mathbf{w}\in\mathbb{R}^d} \frac{1}{2}||\mathbf{w}||^2 \tag{3}$$
$$s.t. \; y_t\mathbf{w}^T\mathbf{x}_t \geq 1 \quad for \; t = 1, \ldots, m$$

Geometrically, the solution $\mathbf{w}^*$ corresponds to the **maximum margin separating hyperplane**. For any linearly separable data set $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m) \in \mathbb{R}^d\} \times \{(1, -1)\}$, the maximum margin γ^* is defined by

$$\gamma^* = \max_{\mathbf{u}:||\mathbf{u}||=1} \min_{t=1,\ldots,m} y_t \mathbf{u}^T \mathbf{x}_t \tag{4}$$

In (4) we can see that the margin $\gamma()$, i.e. the distance between the hyperplane and the point closest to it ($\min_{t=1,\ldots,m} y_t \mathbf{u}^T \mathbf{x}_t$) , is maximized for the vector $\mathbf{u}^*$, the maximum margin separator, i.e. the vector orthogonal to the hyperplane that achieves maximum margin. Note also that the norm of $\mathbf{u}$ is fixed; in fact, it can be shown that finding the maximum margin while keeping the norm of the margin separator constant is equivalent to minimize the norm of the separating hyperplane while keeping its margin constant, which is exactly (3), the objective of the SVM.

This result is formalized through the following theorem:

Theorem:

For any $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$ linearly separable , the maximum margin separator $\mathbf{u}^*$ achieving margin γ^* satisfies

$$\mathbf{u}^* = \gamma^* \mathbf{w}^*$$

where $\mathbf{w}^*$ is the unique solution to $\mathbf{w}^* \in \mathbb{R}^d$ such that: $y_t \mathbf{w}^T \mathbf{x}_t \geq 1$ \, for $t = 1, \ldots, m$.

The proof of this theorem, not given here, demonstrates that finding $\mathbf{u}^*$, which can be written as the solution to the problem

$$\max_{\mathbf{u} \in \mathbb{R}^d, \gamma > 1} \gamma^2 \tag{5}$$
$$such\ that\ ||\mathbf{u}||^2 = 1\ and\ y_t \mathbf{u}^T \mathbf{x}_t \geq 1,\ \ for\ t = 1, \ldots, m$$

is equivalent to finding the solution to the problem

$$\min_{\mathbf{w} \in \mathbb{R}^d} \frac{1}{2} ||\mathbf{w}||^2 \tag{6}$$
$$having\ y_t \mathbf{w}^T \mathbf{x}_t \geq 1 \quad for\ t = 1, \ldots, m$$

once we assume $\mathbf{w} = \frac{\mathbf{u}}{\gamma}$. The solution to (6) can be written as $\mathbf{u}^* = \gamma \mathbf{w}^*$, where $\mathbf{w}^*$ is the solution to the constraint in (6). Note also that in (5) we chose to maximize γ^2 instead of γ , since we are looking for $\mathbf{u}^*$ and the solution will be the same for both γ and γ^2. This proves that the

solution that maximizes the margin γ while keeping $||\mathbf{u}^2|| = 1$ is also the solution that minimizes $||\mathbf{w}^2||$ while having $\gamma \geq 1$. In fact, the solution to (5), $\mathbf{u}^*$, and the solution to (6), $\mathbf{w}^*$, are equivalent and related by $\mathbf{w}^* = \frac{\mathbf{u}^*}{\gamma^*}$. Because (6) is exactly equal to (3), this shows that the SVM solution, the minimal $\mathbf{w}$, will also be the $\mathbf{w}$ that maximizes the margin.

Having seen that finding the solution to (3) means that that solution will also be maximizing the margin, we now briefly examine how such solution can be found, i.e. how $\mathbf{w}^*$ depends on the training set. In order to do so, we firstly introduce a standard result in nonlinear optimization, the lemma known as Fritz John conditions:

Fritz John conditions:

Consider the problem

$$\min_{\mathbf{w} \in \mathbb{R}^d} f(\mathbf{w})$$
$$s.t.\, g_t(\mathbf{w}) \leq 0,\ t = 1, \ldots, m$$

where the functions $f, g_1, \ldots, g_m$ are all differentiable. If $\mathbf{w}_0$ is an optimal solution, then there exists a nonegative vector α such that

$$\triangledown f(\mathbf{w}_0) + \sum_{t \in I} \alpha_t \triangledown g_t(\mathbf{w}_0) = \mathbf{0}$$

where $I = \{1 \leq t \leq m : g_t(\mathbf{w}_0) = 0\}$ is the set of active constraints

---

Applying this lemma to (3) and setting the constraints accordingly, we can then develop to write:

$$\mathbf{w}^* + \sum_{t \in I} \alpha_t (-y_t \mathbf{x}_t) = \mathbf{0}$$

that is

$$\mathbf{w}^* = \sum_{t \in I} \alpha_t y_t \mathbf{x}_t$$

where $I = \{t : y_t \mathbf{x}_t^T \mathbf{w}^* = 1\}$ is the subset of training points closest to the separating hyperplane, that is, point in which the margin $\gamma(\mathbf{w})$ is exactly 1; since the margin can be atmost 1, these are the points with the smallest distance from the hyperplane. The points in I are called

support vectors, and characterize the solution: in fact we just saw how $\mathbf{w}^*$ can be represented as a convex combination of the subset of training points with active constraints, i.e. the support vectors.

4) PEGASOS

If the dataset is not linearly separable, standard SVM could fail to output a solution, since the set I could now be empty. A way to adress this problem is introducing ***slack variables*** for relaxing the constraint on the margin, so that if some points would not satisfy the condition of being distant atmost 1 from the hyperplane, we might still classify them correctly. The relaxed constraint will have the form

$$y_t \mathbf{w}^T \mathbf{x}_t \geq 1 - \xi_t$$
$$with\ \xi_t \geq 0,\ t = 1, \dots, m$$

We want to pick the slack variables ξ_t large enough such that the constraint is sufficiently relaxed, but also keep their interaction in the problem as little as possible. Thus they can be introduced in the objective function, minimizing their average. The new objective function will have the form :

$$\min_{(\mathbf{w},\xi) \in \mathbb{R}^{d+m}} \frac{\lambda}{2} ||\mathbf{w}||^2 + \frac{1}{m} \sum_{t=1}^{m} \xi_t \tag{7}$$
$$s.t.\ y_t \mathbf{w}^T \mathbf{x}_t \geq 1 - \xi_t,\ t = 1, \dots, m$$
$$\xi_t > 0,\ t = 1, \dots, m$$

note the addition of the regularization parameter $\lambda > 0$ to balance this trade-off.

To obtain the optimal choice for the slack variables, first we fix $\mathbf{w}$ and then consider:

$$\xi_t^* = \begin{cases} 0 & \text{if } y_t \mathbf{w}^T \mathbf{x}_t \geq 1 \\ 1 - y_t \mathbf{w}^T \mathbf{x}_t & \text{otherwise} \end{cases}$$

In this way, if the constraint is already satisfied for the current point, the slack variable will have no impact on the solution; otherwise, the slack variable for the current point will be set to the amount for which $y_t \mathbf{w}^T \mathbf{x}_t$ falls short of one, i.e. the minimum amount possible to guarantee that the constraint is satisfied.

We now examine the slack variable form for a given $\mathbf{w}$ in the case of impossibility of satisfying the constraint:

$$\xi_t(\mathbf{w}) = \max\{0, 1 - y_t \mathbf{w}^T \mathbf{x}_t\} = [1 - y_t \mathbf{w}^T \mathbf{x}_t]_+ = hin_t(\mathbf{w}) \tag{8}$$

and see that it corresponds to the *hinge loss* convex function $hin_t(\mathbf{w})$. Thanks to the result in (8), it's possible to eliminate the constraints from (7), replacing $\xi_t(\mathbf{w})$ with the optimal choice, $hin_t(\mathbf{w})$. The SVM objective can then be rewritten as:

$$\min_{\mathbf{w}\in\mathbb{R}^d} f(\mathbf{w}) \quad \text{where } f(\mathbf{w}) = \frac{1}{m}\sum_{t=1}^{m} hin_t(\mathbf{w}) + \frac{\lambda}{2}||\mathbf{w}||^2 \tag{9}$$

since hin_t will automatically satisfy $y_t\mathbf{w}^T\mathbf{x}_t \geq 1$ and $\xi_t \geq 1$ by design. Note that we can still express the solution to (9) as $\mathbf{w}^* = \sum_{t=1}^{m} \alpha_t y_t \mathbf{x}_t$, where $\alpha_t > 0 \iff hin_t(\mathbf{w}^*) > 0$ and $\alpha_t = 0 \iff hin_t(\mathbf{w}^*) = 0$

I will now present the last preliminary result before moving to the algorithm for computing $\mathbf{w}^*$. Note that

$$f(\mathbf{w}) = \frac{1}{m}\sum_{t=1}^{m} l_t(\mathbf{w}) \tag{10}$$

where $l_t(\mathbf{w}) = hin_t(\mathbf{w}) + \frac{\lambda}{2}||\mathbf{w}||^2$ is a strongly convex function; indeed, $hin_t(\mathbf{w})$ is convex, while $\frac{\lambda}{2}||\mathbf{w}||^2$ is λ-strongly convex. This implies that their sum is λ-strongly convex. Moreover, the objective function written as in (10) has the same form of the sequential risk presented in subsection 2. Using these two arguments, we can apply OGD with strongly convex losses for optimizing the objective function (9).

This is done in practice by the following algorithm, denominated ***PEGASOS*** (**P**rimal **E**stimated sub-**G**r**A**dient **SO**lver for **S**vm):

PEGASOS algorithm

Parameters : Number T of iterations, regularization coefficient $\lambda > 1$

Input : Training set $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m) \in \mathbb{R}^d\} \times \{(1, -1)\}$

Set $\mathbf{w}_1^T = [0, \ldots, 0]$

For $t = 1, \ldots, T$:

- Draw uniformly at random an element $(\mathbf{x}_{Z_t}, y_{Z_t})$ from the training set
- Set $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla l_{Z_t}(\mathbf{w}_t)$

Output: $\bar{\mathbf{w}} = \frac{1}{T}(\mathbf{w}_1 + \ldots + \mathbf{w}_T)$

In this notation, we use Z_t to denote the realization of the t-th draw. The result $\bar{\mathbf{w}}$ is the minimizer of (10) and the optimal solution given the training set and λ. $\eta_t = \frac{1}{\lambda t}$ is the parameter controlling the learning rate and includes the regularization term λ. Note that, since $l_{Z_t}(\mathbf{w}_t) = hin_{Z_t}(\mathbf{w}_t) + \frac{\lambda}{2}||\mathbf{w}_t||^2$ represents the loss for the current iteration t, computed on the drawn example and the current separating hyperplane $\mathbf{w}_t$, $\nabla l_{Z_t}(\mathbf{w}_t)$ will have form

$$\nabla l_{Z_t}(\mathbf{w}_t) = -y_{z_t}\mathbf{x}_{Z_t}\mathbb{I}\{hin_{z_t}(\mathbf{w}_t) > 0\} + \lambda w_t$$

where $\mathbb{I}()$ is the indicator function. This means that the update $\mathbf{w}_{t+1}$ can have one of two forms depending on whether the point $\mathbf{x}_{Z_t}$ violates the constraint $y_{Z_t}\mathbf{w}_t^T\mathbf{x}_{Z_t} \geq 1$ or not:

$$\mathbf{w}_{t+1} = \begin{cases} \mathbf{w}_t - (\eta_t\lambda\mathbf{w}_t) & \text{if } y_{Z_t}\mathbf{w}_t^T\mathbf{x}_{Z_t} \geq 1 \\ \mathbf{w}_t + \eta_t y_{Z_t}\mathbf{x}_{Z_t} - \eta_t\lambda\mathbf{w}_t & \text{otherwise} \end{cases}$$

Finally, note that, while we apply OGD, we are not dealing with a continuous stream of points, but rather with a continuous *sampling*, as the algorithm draws each point and then proceed to optimize the objective calculated on that point, update the classifier and then move on to the next draw. For this reason, we refer to this implementation of the OGD as **Stochastic Gradient Descent** (SGD).

Kernel trick: dealing with non-linearly separable data

PEGASOS is an algorithm that achieves greater flexibility and computational efficiency with respect to standard SVM thanks to the use of hinge losses in the objective function, allowing to decrease bias in case of non-linearly separable data sets. However standard PEGASOS, as presented in the subsection above, will still learn only linear classifiers: this cause large bias when the optimal decision boundaries have pronounced nonlinearity.

One popular way to solve this issue is **feature expansion**, a technique that augments the data by adding extra features that are nonlinear combinations of the original features to each point. This can be formally presented in terms of a function $\phi : \mathbb{R}^d \to \mathcal{H}$ mapping data points $\mathbf{x}_t \in \mathbb{R}^d$ to a higher dimensional space $\mathcal{H}$. By training a linear predictor on a feature-expanded training set, one actually learns a more complex nonlinear predictor in $\mathbb{R}^d$.

There is, however, a great limitation to the use of feature expansion in machine learning problems: consider the polynomial feature expansion map $\phi : \mathbb{R}^d \to \mathcal{H}$ where $\mathcal{H} \equiv \mathbb{R}^N$, that uses features of the form $\prod_{s=1}^{k} x_{v_s},\ \forall v \in \{1, \ldots, d\}^k\ and\ \forall k = 0, 1, \ldots, n$. If we fix such a ϕ and consider the classifier $h : \mathbb{R}^d \to \{-1, 1\}$ defined by

$$h(\mathbf{x}) = sgn(\mathbf{w}^T\phi(\mathbf{x})) \text{ where } \mathbf{w}^T\phi(\mathbf{x}) = \sum_{i=1}^{N} w_i\phi(\mathbf{x})_i$$

(w_i denotes the i-th component of the vector $\mathbf{w}$, while $\phi(\mathbf{x})_i$ is the i-th component of the expanded vector) we can see that

$$N = \sum_{k=0}^{n} |1, \ldots, d^k| = \sum_{k=0}^{n} d^k = \frac{d^{n+1} - 1}{d - 1}$$

This implies that $N = \Theta(d^n)$ is exponential in the degree n, and computing ϕ becomes infeasible even for moderately large n.

Luckily, this limitation can be completely surpassed thanks to the so called **Kernel trick**: it involves using *kernel functions* to compute the dot product between two vectors $\mathbf{x}_i^T\mathbf{x}_j$ in an expanded feature space without having to explicitly compute their expansions $\phi(\mathbf{x}_i)$, $\phi(\mathbf{x}_j)$. Many algorithms for learning linear classifiers, including PEGASOS, involve computing the dot product between the vector orthogonal to the separating hyperplane, $\mathbf{w}$, and the vectors of data points $\mathbf{x}$; the kernel trick can then be applied to perform such operation in an expanded feature space withouth having to compute the expansion maps. Consider for example the hinge loss (8) evaluated during an update of the PEGASOS algorithm: first note that the hyperplane $\mathbf{w}_t$ is a linear combination of the support vectors and their labels, i.e.

$$\mathbf{w}_t = \frac{1}{\lambda t} \sum_{r=1}^{t} y_{S_r}\mathbf{x}_{S_r}$$
$$where\ S = \{r : y_r\mathbf{x}_r^T\mathbf{w}_r < 1\}\ for\ r = 1, \ldots, t$$

Then, in an expanded feature space, the product in (8) becomes:

$$y_t\left(\frac{1}{\lambda t} \sum_{r=1}^{t} y_{S_r}\phi(\mathbf{x}_{S_r})\right)^T \phi(\mathbf{x}_t)$$

Because the inner product is a bilinear operator, we can write

$$y_t\frac{1}{\lambda t} \sum_{r=1}^{t} y_{S_r}\langle\phi(\mathbf{x}_{S_r}), \phi(\mathbf{x}_t)\rangle$$

and thus, after choosing an appropriate kernel function $K()$,

$$y_t \frac{1}{\lambda t} \sum_{r=1}^{t} y_{S_r} K(\mathbf{x}_{S_r}, \mathbf{x}_t)$$

Using this solution it's possible to learn nonlinear predictors in $\mathbb{R}^d$ efficiently. The objective function (9) of this ***Kernelized PEGASOS*** will become:

$$\min_{\mathbf{g} \in \mathcal{H}} f(\mathbf{g}) \text{ where } f(\mathbf{g}) = \frac{1}{m} \sum_{t=1}^{m} hin_t(\mathbf{g}) + \frac{\lambda}{2} ||\mathbf{g}||^2 \tag{11}$$
$$\text{where } hin_t(\mathbf{g}) = [1 - y_t dotg(\mathbf{x}_t)]_+ \; with \; dotg(\mathbf{x}_t) = \langle \mathbf{g}, \phi(\mathbf{x}_t) \rangle$$

Here $\mathbf{g}$ represents $\phi(\mathbf{w})$, the expansion in $\mathcal{H}$ of the hyperplane in the original feature space. The update, run in the kernel space, will have form:

$$\mathbf{g}_{t+1} = \frac{1}{\lambda t} \sum_{r=1}^{t} y_{S_r} K(\mathbf{x}_{S_r}, .) \mathbb{I}\{hin_r(\mathbf{g}_r) > 0\} \tag{12}$$

There exist many suitable kernel functions $K()$ that can be used for performing the kernel trick; in general, a function is a kernel iff:
$\forall m \in \mathbb{N}$ and $\forall x_1, \ldots, x_m \in \mathcal{X}$, the $m \times m$ matrix A such that $A_{i,j} = K(x_i, x_j)$ is positive semidefinite.

The kernel function used in this paper for the computation of the dot products in the steps of the Kernelized pegasos presented below is the **Gaussian Kernel**, that has form

$$K_\gamma(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2\gamma} ||\mathbf{x} - \mathbf{x}'||^2\right) \tag{13}$$

where γ is a parameter controlling the width of the Gaussians centered on points $\mathbf{x}_s$. Theory suggest a value of γ = 0.25 for this particular problem.

Code description and implementation

The following section illustrates the code used to implement the kernelized version of the PEGASOS algorithm, as presented above: it is furtherly divided into four subsection, each dedicated to different aspects of the implementation. The subsections are

- **Libraries**: load data and packages used for the coding
- **Auxiliary functions**: a set of five functions performing various side tasks inside the main loop
- **PEGASOS**: the algorithm is split into two functions, one for training and one for predicting labels
- **CV**: a 5-fold standard cross validation is used for looping between different values of λ and T

Libraries

In [1]:
```
#load data
import h5py
with h5py.File(r'your directory, 'r') as hf:
        train = hf.get('train')
        X_tr = train.get('data')[:]
        y_tr = train.get('target')[:]
        test = hf.get('test')
        X_te = test.get('data')[:]
        y_te = test.get('target')[:]
```

In [2]:
```
import numpy as np
import pandas as pd
import random
from copy import deepcopy
```

In [ ]:
```
#build dataframe
df=pd.DataFrame(np.concatenate((np.column_stack((X_tr,y_tr)),np.column_stack((X_te,y_te))), axis = 0))
```

Auxiliary functions

In [4]:
```
#Function for changing label into a binary feature
def chg_pred(df,a):
    bin_y = np.where(df.loc[:,256] == a, 1, -1)
    df[257] = bin_y
```

The function `chg_pred` is used inside the innermost loop of CV for changing the labels of the currently evaluated training and test set to binary labels. Because PEGASOS ouputs a binary classifier (the activation function for PEGASOS predictor is $sign()$, so $h_j(g) = sign(g_T)$ where g_T is as in (12) for $t = T$), multiclass labels of the dataframe `df` are recoded with a one-vs-all approach. Note that the subscript j of $h_j(g)$ denotes the j -th class of predictions, i.e. the digit for which the current h is looking for.

In [5]:
```
def sign(x): #sign function
    if x > 0:
        return 1
    elif x == 0:
        return 0
```

```
    else:
        return -1
```

A simple function for detecting the sign of the final prediction and assign the binary label accordingly. Used in `pred_pegasos`.

In [6]:
```
#define the gaussian kernel function
def gauss_k(gamma,a,b):
    return np.exp((-1/2*gamma)*(np.linalg.norm((a-b), ord = 1)**2))
```

Function for defining the Gaussian kernel, as written in (13). The function `np.linalg.norm` from the `NumPy` package is used to calculate the Euclidean norm.

In [7]:
```
def assign_pred(df): #takes as input result arrays for the ten predictors (otp_train|otp_test)
    return pd.DataFrame(df).T.idxmax(axis=1)
```

The last two functions work in tandem inside the loop over the folds of the current run of CV. `assign_pred` takes as input the vectors of binary predictions from all the 10 predictors' output, stacked horizontally in the objects `otp_train` and `otp_test`; it then loops over the rows of the resulting dataframe and returns the index of the column where the first "1" appears (indicating that the predictor for that digit found a match). Because predictors are constructed sequentially starting from 0 to 9, the first column corresponds to the binary prediction of the predictor returning a match for zeros, the second returning a match for ones, and so on. This means that the index of the columns will also represents the predicted digit label.

In [8]:
```
def comp_mc(df1,df2): #takes as input the training/test folds and the result of assign_pred and compute the average mc error
    comp = df1.loc[:,256].reset_index(drop=True) == df2
    errors_count = comp.value_counts().get(False, 0)
    total_count = len(comp)
    return errors_count/total_count #fraction of examples incorrectly classified
```

`comp_mc` takes as inputs the current training or test set and the results of `assign_pred`, comparing the true multiclass labels with the predicted ones; then it calculates the misclassification error (mc) , as

$$mc = \frac{\sum_{i=1}^{M} \mathbb{I}\{h(x)_i \neq y_i\}}{|M|}$$

Note that $h(x)$ is the multiclass predictor as constructed by `assign_pred`, as opposed to $h_j(g)$ that is instead the binary predictor for the j-th class. As stated above, $h(x)$ is simply

$$first(\mathbb{I}\{h_j(g)_i = 1\})$$
$$for\ j = 0, \ldots, 9\ and\ i = 1, \ldots, M$$

where the function $first()$ returns the index, corresponding to the digit, of the column j iff $\mathbb{I}\{h_j(g)_i = 1\} = 1$. $|M|$ is the cardinality of the set currently evaluated.

Pegasos

In [ ]:
```
#train pegasos
def train_pegasos(df_train,T,lam):
    t = 1
    S = [] #list for examples classified incorrectly

    for i in range(T):
        eta = 1/(lam*t) #update eta for the current iteration
        sample = df_train.iloc[np.random.choice(len(df_train), replace=True, size = 1)] #draw an example
        x_t = sample.iloc[0,0:256]
        y_t = sample.iloc[0,257]

        g_list = [y_s * gauss_k(0.25, x_s, x_t) for y_s, x_s in S] #note that we are using bi-linearity of the inner product
        g = eta * sum(g_list) #compute g using current eta factor

        if y_t * g < 1: #check condition
            S.append((y_t,x_t))

        t += 1 #update counter

    return S
#The algorithm returns the support vectors' set , used as input in the prediction
```

The first of the two functions for implementing PEGASOS is dedicated to training: it takes as inputs the training set, the number of iterations T and the parameter λ. The algorithm works exactly as explained in subsection 4 of the analytical description; first it draws a sample from the training set, then set the parameter η and use the support vectors to create the separator g ; if the data point is incorrectly classified, it adds the example to the set of support vectors S that will be used in the next iteration for updating the classifier. Note the use of the gaussian kernel for computing the dot product between the support vectors and the current point, as illustrated in the section *Kernel trick*.

In [ ]:
```
def pred_pegasos(df, S):
    pred_vec = np.repeat(0, len(df[256]))
```

```
    for i in range(len(df)):
        x_t = df.iloc[i,0:256]
        g_list = [y_s * gauss_k(0.25, x_s, x_t) for y_s, x_s in S]
        g = sum(g_list)

        pred_vec[i] = sign(g)

    return pred_vec
```

The second function takes as inputs the set for which it needs to make predictions for and the set of support vectors as output by `train_pegasos`; it returns a vector of binary predictions made by the current classifier $h_j(g_T)$.

CV

In [ ]:

```
lambdas = [1*10**-10, 1*10**-5, 1*10**-3, 1*10**-1, 1*10**5]
iterations = [250, 500, 1000, 3500, 7438]

S_h = list(np.repeat(0, 10)) #container for storing different sets of support vectors (one for each predictor h)
otp_train = list(np.repeat(0, 10)) #lists for storing vectors of predictions for different predictors
otp_test= list(np.repeat(0, 10))
cv_error = list(np.repeat(0, 10)) #lists for storing errors and results of cv
cv_results = list(np.repeat(0, 9))

df = df.sample(frac=1) #reshuffle
folds = np.array_split(df, 5) #create folds

indx = 0

for lam in lambdas:
    for T in iterations:
        for f in range(5):
            df_test = folds.pop(0) #take first element
            df_test = df_test.reset_index(drop=True) #rearrange indexes
            df_train = pd.DataFrame(np.concatenate((folds),axis = 0)) #concatenate other elements

            for h in range(10):
                df_train_copy = deepcopy(df_train)
                df_test_copy = deepcopy(df_test)
                chg_pred(df_train_copy, h)
                chg_pred(df_test_copy, h)

                S_h[h] = (train_pegasos(df_train_copy,T,lam)) #train current predictor
```

```
                otp_train[h] = pred_pegasos(df_train_copy, S_h[h]) #classify train with current predictor
                otp_test[h] = pred_pegasos(df_test_copy, S_h[h]) #classify test with current predictor

            cv_error[f] = comp_mc(df_train_copy,assign_pred(otp_train)) #first five elements of cv_error are avg. fth fold's tra
            cv_error[f+5] = comp_mc(df_test_copy,assign_pred(otp_test)) #second five elements of cv_error are avg. fth fold's te

            folds.append(df_test) #append test fold in last position

        cv_results[indx] = (sum(cv_error[0:5])/5, sum(cv_error[5:10])/5)
        indx += 1
```

This is the 5-fold Cross validation framework applying the functions described until now. After the creation of the lists for storing the results of each phase and the creation of the folds, the code works as follows:

- for a certain value of λ :
  - for a certain value of T :
    - for the current split:
      - Assign the test and training folds.
      - for each class of labels j :
        - create a copy of the train and test folds.
        - apply `chg_pred` to both train and test copies.
        - apply `train_pegasos` for the current value of λ and T ; store the support vectors' set in the list S.
        - Use the precedently stored support vectors' set in `pred_pegasos` to make predictions for the current train and test folds. Store the results in `otp_train` and `otp_test` respectively.
        - exit the loop and reinitialize with the next class j
      - After having created and applied all 10 $h_j(g_T)$ predictors to the current training and test folds, apply `comp_mc` (note the presence of `assign_pred` inside the call) to evaluate the multiclass misclassification error for the current training and test folds. Store the results in `cv_error` , separating the training results from the test results.
      - Initialize the next split and repeat
    - compute the CV misclassification training and test errors for the current pair of values of λ and T. The CV error is the average between all of the five splits' errors
    - Reinitialize for the next value of T
  - Proceed with the next value of λ

After having completed the steps above for all the possible pairs of λ and T, the list `cv_results` will contain 25 tuples (one for each pair), where the first element represents the CV misclassification training error, and the second the CV misclassification test error. Such metrics can be used to assess the experiment results.

Results

CV was used to perform a gridsearch over 5 possible values each of λ and T. The values for lambda were chosen to be relatively distant from one another, in order to explore better the differences in results when using quite different regularization parameters. The number of iterations were selected in order to assess the algorithm's performance when using both suboptimal (250 and 7438 iterations) and sensible T (500,1000,3500 iterations) with respect to the cardinality of the training set (7438).

The following heatmaps display the results of the experiment, presenting the misclassification cross validated errors for both training and test sets and for the 25 possible pairs of parameters.

In [4]:
```
from IPython.display import display_html
import seaborn as sns

train_styler = train.style.set_table_attributes("style='display:inline'").set_caption('Training CV errors').format_index(formatt
test_styler = test.style.set_table_attributes("style='display:inline'").set_caption('Test CV Errors').format_index(formatter = la

display_html(train_styler._repr_html_() + test_styler._repr_html_(), raw=True)
```

Training CV errors

Iterations	250	500	1000	3500	7438
lambda					
1.00e-10	0.194047	0.138874	0.101366	0.050118	0.026672
1.00e-05	0.195041	0.142100	0.103786	0.048371	0.024871
1.00e-03	0.193832	0.137529	0.105856	0.050414	0.024387
1.00e-01	0.187944	0.142020	0.103571	0.051248	0.024252
1.00e+05	0.186115	0.142854	0.104700	0.051490	0.026108

Test CV Errors

Iterations	250	500	1000	3500	7438
lambda					
1.00e-10	0.200040	0.145406	0.110454	0.072596	0.058292
1.00e-05	0.200369	0.147775	0.114756	0.073242	0.054959
1.00e-03	0.201764	0.146483	0.116693	0.074962	0.055928
1.00e-01	0.191869	0.149600	0.115509	0.070338	0.055281
1.00e+05	0.194986	0.153472	0.113465	0.070230	0.056356

Figure 1

The tables in *figure 1* are coloured based on the values inside each column, i.e. the colours identify the best (black) and worst (white) values of cv errors by column, as well as intermediary values (please see *figure 2* for the colour scale).

In [5]:
```
import matplotlib.pyplot as plt
%matplotlib inline
sns.heatmap(pd.DataFrame(test), annot = True, fmt='g', cmap= "hot", )
plt.title('Test CV Errors', fontsize = 12)
```

Out[5]:
```
Text(0.5, 1.0, 'Test CV Errors')
```

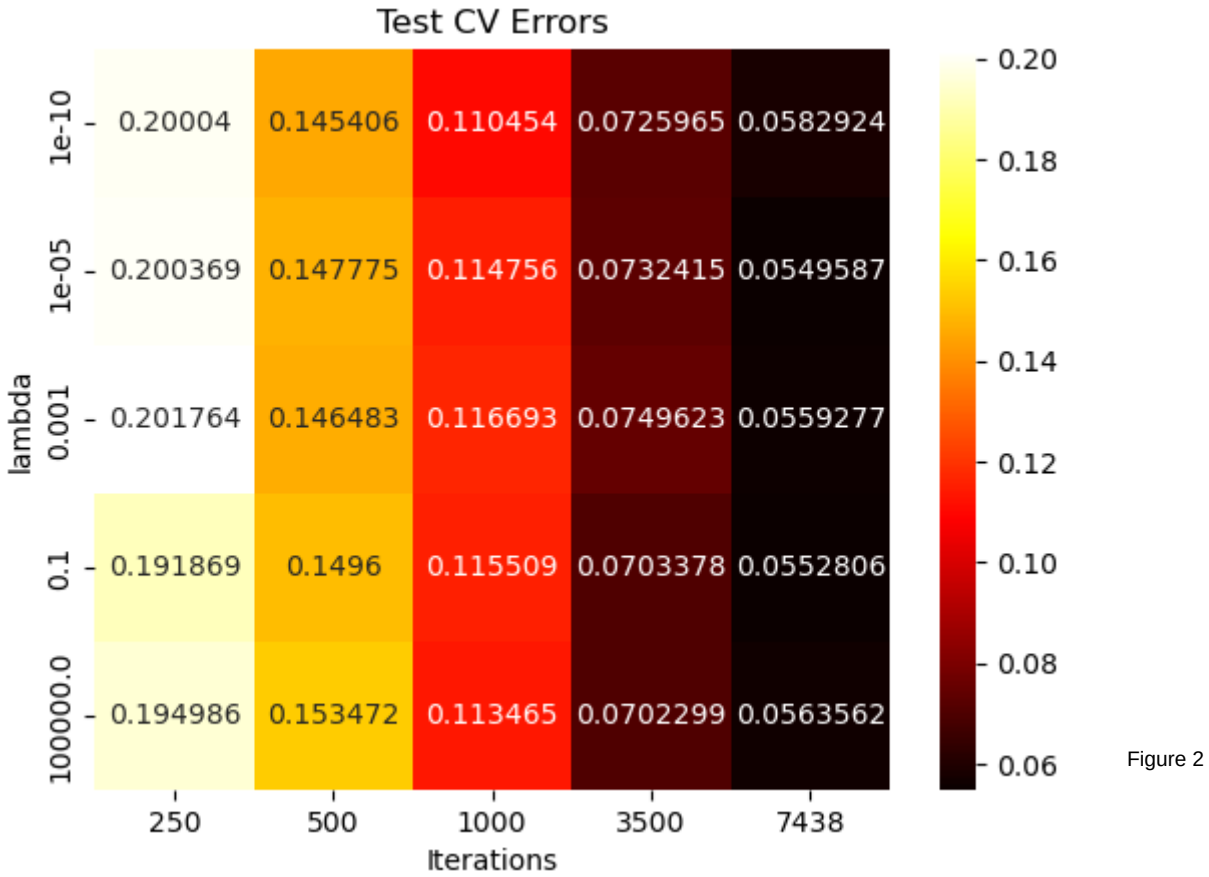

Figure 2

The table in *figure 2* instead focus on the test errors; in it, values are coloured across columns, i.e. not grouped by number of iterations as the tables in *figure 1*.

The first immediately noticeable result is the improving in accuracy when using greater values of T: as PEGASOS build its classifier using the set of support vectors, an higher number of samples has greater probability of increasing the cardinality of said set, resulting in predictors that fit better the data; however, a classifier too complex will incurr in overfitting. This can be clearly inferred looking at the training errors from *figure 1*: For

$T = 250$ the training error decrease the most when injecting as much variance as possible, that is, in correspondence of $lambda = 1e + 5$; for the values in between, the variance required to counterbalance the higher bias coming from the scarcity of examples starts to diminish and then rest on the lowest possible value for T = 1000, i.e. as the increasing in available data points reduce the bias. A similar behaviour is observable also looking at the test errors, albeit they benefit quicker from the increase in datapoints and thus require less variance for optimal results already at $T = 500$. This is due to the fact that unseen data will generally not have the exact same patterns found in the training; thus a predictor not too specific with regard to the training set, which means that it will have less variance and more bias, will imply better results on unseen data, as long as the bias is not too much.

The second observation that can be made revolves around what happens when moving from 1000 to 3500 and finally 7438 samples. The steep decrease in training errors of the last two columns in *figure 1* with resepect to the rest of the table, suggest that the increase availabilty of points has allowed the predictor to pick up some previously noisy pattern in the data and adapt to it, achieving better results also on unforseen data; this can be confirmed looking again at the increase in accuracy for higher values of T noticeable in *figure 2*.

In summary, when T is at its lowest, the bias is high and the predictor needs more variance in order to counterbalance its effect. As the cardinality of the training set starts to increase, bias decreases and the excess variance is no longer needed (columns 2 and 3 of *figure 1*); however, the overall accuracy is not yet optimal (*figure 2*). When $T = 3500$, a change occurs: the predictor now picks up previously unseen patterns in the data and both training and test errors decrease steeply (*figure 1* and *2*). In particular, looking at the group $T = 3500$ in *figure 1*, it can be noticed that the lowest value of MC error can be found in correspondence of the lambda associated with the highest increase in variance ($\lambda = 1e + 5$): this result confirms that the patterns now detected are in fact present in the data generating distribution D and should be taken into consideration for improving the accuracy.

A final consideration completes the picture: when the algorithm uses the highest possible number here considered of examples (equal to the cardinality of the training set - notice however that the data are sampled with replacement, so this does not mean that the training set uses all examples available for $T = 7438$), the predictor oscillates again towards overfitting (*figure 1*); moreover, the increase in accuracy resulting from the higher number of samples is half the one observed when moving from $T = 1000$ to $T = 3500$. This suggests that a good choice of T lies between 3500 and 7438 samples and the best value of lambda would probably be found in the regions near $\lambda = 1e - 3$

In conclusion, considering what has been said so far, the PEGASOS algorithm as implemented here is capable of reaching a misclassification rate in the test set as low as 5.4% in correspondence of $T = 7438$ and $\lambda = 1 \times 10^{-}5$. However, considering the increase in runtime associated with high values of T and the results observed so far, It must be concluded that the optimal choice of T and λ will be found for $3500 \leq T \leq 7438$ and $1e - 10 \leq \lambda \leq 1e - 3$. Further evaluation may confirm these findings.