MSc in Computer Science
at University of Milan

Statistical Methods for Machine Learning
Kernelized Linear Predictor

course held by **Nicoló Cesa-Bianchi**

Email:
davide.cologni@studenti.unimi.it

Created by:
Davide Cologni
mat. 09732A

Academic year of 2023/2024

Contents

1 introduction

1.1 Project Description

The goal of this project is to train different learning algorithms to solve the binary classification problem. Specifically, we need to predict the value of the label in the y column based on the numerical features $x1$ through $x10$.
As mentioned in the description of the project, I used the zero-one loss as a metric to evaluate the performance of the algorithms.
In the following chapter I describe how I analyzed and preprocessed the dataset to improve the performance of the modules.
In the following chapters I describe how I implemented each algorithm and how I chose the hyperparameters for them.
I have also analyzed how varying the hyperparameter affects the training and testing error and explained these empirical results with the theoretical background provided by the lectures.

I implemented the following algorithms:

- Perceptron
- Pegasos
- Logistic Pegasos
- Feature expanded Perceptron (with 2nd degree polynomial expansion)
- Feature expanded Pegasos (with 2nd degree polynomial expansion)
- Feature expanded Pegasos with logistic loss (with 2nd degree polynomial expansion)
- Kernel Perceptron
- Kernel Pegasos

1.2 Project Structure

The project is divided into the following folders:

- datasets: contains the provided dataset
- models: a set of pre-trained models, created using the train subcommand
- src: the source code of the project, the entry point is main.py and can be used both to train the models from the dataset and to run them
- report: this folder contains the source for this report

1.3 Usage

The entry point to the project is the **main.py** file. It can be called with the command line argument and provides two subcommands 'train' and 'run' The first subcommand requires the name of the algorithm to be trained and stores a predictor in the path provided in the output argument, serialized using the Python pickle module.

```
$ python src/main.py train pegasos models/pegasos.pkl
```

The run subcommand takes a serialized model and then prints its training and test errors

```
$ python src/main.py run models/pegasos.pkl
```

There are other options available that are described using the '–help' option, they will be described in the next sections of this report as they come up.

2 Dataset analysis and preprocessing

2.1 Dataset description

The provided dataset contains 10000 points with 10 features named from $x1$ to $x10$ and a label column named y.
All the feature are floating point values and the dataset is well formed (in the sense that there are no missing values).
The label column contains values that are either -1 or $+1$.
There isn't duplicated data in the training set.
I collect the major statistics from each feature in the dataset: mean, standard deviation, min and max

	x1	x2	x3	x4	x5
min	2.44342055e-03	-7.52493399e+00	9.85724553e+01	-7.07893888e+00	-9.99999717e-01
max	9.38422309e+00	8.30237476e+00	1.01260768e+02	-2.92150729e-06	9.99999998e-01
mean	1.59129826e+00	5.15879411e-01	9.98489361e+01	-1.50413876e+00	7.76447773e-02
std	1.32111881	2.05438485	0.71091203	1.13354878	0.70723419

	x6	x7	x8	x9	x10
min	-6.90697075e+00	-7.14075517e+00	-7.15188951e+00	-5.67739307e+01	-1.00000000e+00
max	8.76030588e+00	9.28726632e+00	6.21145227e+00	-5.42088897e+01	1.00000000e+00
mean	5.18228648e-02	9.75207134e-01	6.35194433e-01	5.19260973e-02	-5.54476783e+01
std	0.70471943	2.16212877	2.21259701	1.76955726	0.71004639

It's clear that the different features are not normalized and follow different probability distributions.

2.2 Feature Scaling

Because the dataset is already well formed and there are no missing values The first thing I do is scale the features.
This step should ensure that the values of the features are in a comparable range.
I tried two approaches: normalization and standardisation.
Scaling sets each feature to have a mean of 0 and a standard deviation of 1. This is achieved by subtracting the feature mean from each value and dividing by its standard deviation:

$$x' = \frac{x - \mu}{\sigma}$$

(where μ is the mean and σ is the standard deviation).
In the case of **normalization**, the features are rescaled in a fixed range between 0 and 1 in the following way:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

.
It's important to note that we should perform both approaches using a sound procedure, more specifically, we should avoid data leakage: When calculating the minimum and maximum of the feature, we should only consider the training set and scale the test set only according to these values, without deriving any information from it.
In the same way, we should calculate the mean and standard deviation for the standardisation process.
Both of these approaches lead to important improvements and can be demonstrated both from an empirical point of view (see table TODO) and from a theoretical perspective.

2.3 Outliers removal

Another approach I tried is removing the outliers from the dataset using the Z-score methods.
I calculated the score $Z = (x - \mu)/\sigma$ for each value where μ is the mean and σ is the variance of the feature.
I then removed all the points with a Z-score greater or equals than 3 in absolute value.
I found (and removed) 265 outliers (recall that the dataset has size 10000).
I tried training non-kernelized Perceptron and Pegasos over the modified dataset but the results shows that the dataset is already sufficently cleaned: in fact it affects the performance of the models in a minimum way with no significative changes, and even in some cases it is (even if only slightly) worsening

	with outliers	without outliers
Perceptron	0.326	0.326
Pegasos	0.2865	0.29
Feature expanded Perceptron	0.087	0.087
Feature expanded Pegasos	0.0555	0.0565

Comparison of test error (using zero one loss) with and without outliers.
Note that when I refere to feature expansion I intend polynomial feature expansion of degree 2.

2.4 Feature correlation

I sorted all the point according to each axis and plot on the other axis all the other features to spot eventual correlation and I observed that the feature 2 and 5 have a linear correlation (with a negative coefficent, see Fig. 1) as the feature 5 and 9 (with positive coefficent, see Fig. 2).

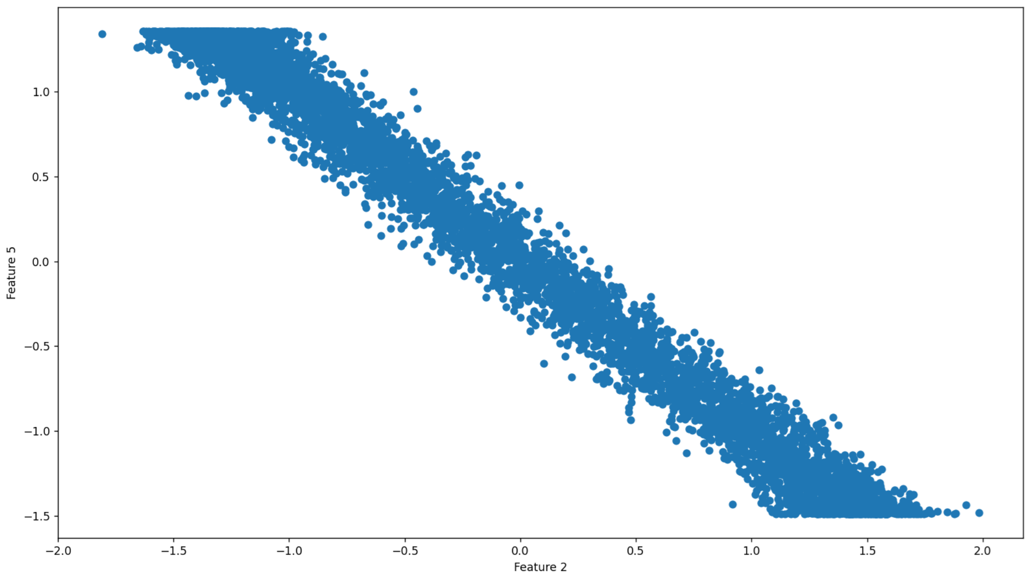

Figure 1: Here there is a clear linear correlation between feature 2 and 5, but very noisy

One possibility in this case during the preprocessing of the data is to remove the correlated features and leave only one of them to avoid redundancy of the data.
I don't follow this approach because there is a sensibile noise in the correlation and removing some

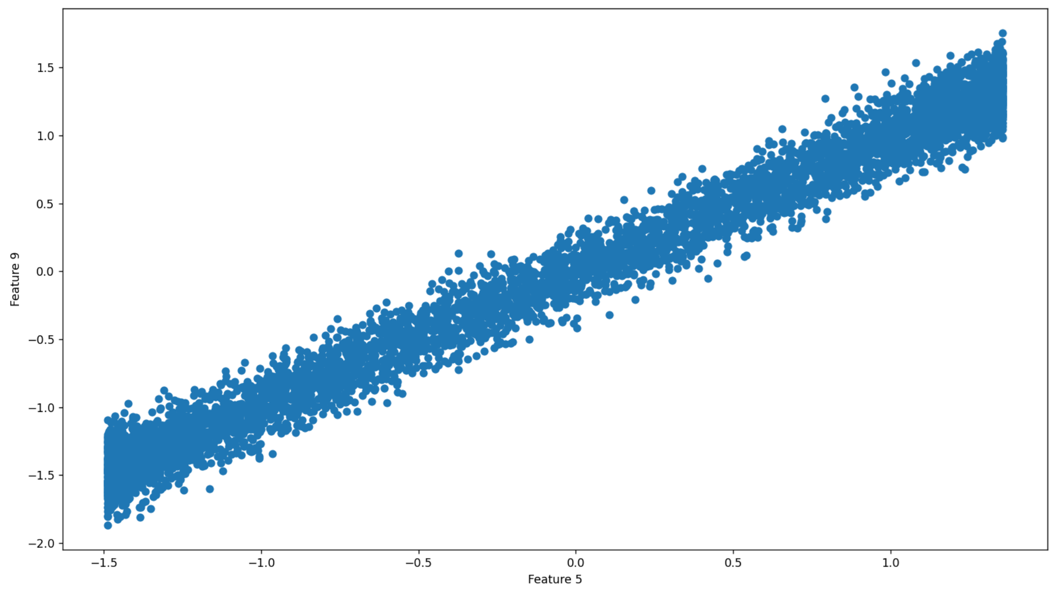

Figure 2: Another correlation is also present between feature 5 and 9 (with a negative coefficent)

features can lead to also removing this noise that can encode important information on the model.

2.5 Feature expansion

To being able to express non-homogeneous linear separators (hyperplane that don't pass through the origin) we add a constant feature of value 1 to each point in the dataset.
Let $(\mathbf{x})$ be any point in the dataset and $(\mathbf{w})$ be the linear separator, if we define $x' = (\mathbf{x}, 1)$ we can define $w' = (w, c)$, in that way:

$$w'^T x' = (\mathbf{w}^T \mathbf{x} + c)$$

3 Perceptron

The first algorithm that I implemented is the perceptron algorithm.
The perceptron algorithm is used to learn linear classifiers.
linear classifiers are identified by an hyperplane that separe the input space into two halfspaces, one positive and one negative.
The positive halfspace is called so because the dot product with the normal vector that identify the hyperplane and any point in that space is positive, similary the negative halfspace has always negative dot products.
One important property of the Perceptron is the convergence (to the ERM) in a finite number of step if the dataset is lineary separable.
This properties is stated by the **Perceptron Convergence Theorem**:

Let $(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_m, y_m)$ be a linearly separable training set. Then the Perceptron algorithm returns a linear classifier with zero training error in a finite number of updates

$$M \leq \left(\min_{\boldsymbol{u}:\gamma(\boldsymbol{u})\geq 1} \|\boldsymbol{u}\|^2 \right) \left(\max_{t=1,\dots,m} \|\boldsymbol{x}_t\|^2 \right)$$

where $\gamma(\boldsymbol{u})$ is the margin obtained by the linear separator $\boldsymbol{u}$

Is possible to show also a bound for non lineary separable cases:

$$M \leq \sum_{t=1}^{T} h_t(\boldsymbol{u}) + (\|\boldsymbol{u}\|X)^2 + \|\boldsymbol{u}\|X \sqrt{\sum_{t=1}^{T} h_t(\boldsymbol{u})} \quad \text{for all } \boldsymbol{u} \in \mathbb{R}^d$$

This shows a bound on the number of mistakes made by the Perceptron algorithm on any data sequence of arbitrary length T.
$h_t(\boldsymbol{u})$ is the hinge loss for the t-example.

3.1 Naive version

Algorithm 1: The Perceptron algorithm

```
   Input: Training set $(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_m, y_m)$
1  $\boldsymbol{w} = (0, \dots, 0)$
2  while $true$ do
3      for $i = 1, \dots, m$ do                          // (epoch)
4          if $y_i \boldsymbol{w}^\top \boldsymbol{x}_i \leq 0$ then
5              $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_i \boldsymbol{x}_i$           // (update)
6      end
7      if no update in last epoch then
8          break
9  end
   Output: $\boldsymbol{w}$
```

My implementation slightly varies from the presented pseudocode: While the above on keep running until convergence if the training set is not lineary separable (as in this case) we never converge and so the algorithm never terminates.
To avoid this I use an additional parameter 'max_epoch' that limits the number of epoch which tha

algorithm can run.
I choose to use a fixed value of 20 for this parameter both for the naive case and over the feature expanded dataset presented in the next section.
Training the perceptron algorithm with the preprocessing methodology described in the previous chapter (For all the algorihtms I describe I trained it with the default command line options), I have obtained a training error of 0.322625 and a test error of 0.326.
The linear separator founded by the perceptron algorihtm has the following features:

(0.55604295, 1.97486085, -2.58700382, -1.74490783, 1.91766823, -3.89876104,
-0.02419966, 3.06371997, 0.21648717, -0.79054099, 1) (Recall the features are 11 because we add a constant feature of 1 to the dataset to being able to express non-homogeneous linear hyperplane).

3.2 Feature Expansion of 2nd degree

I also trained the perceptron algorihtm on a second degree polynomial feature expanded dataset.
In this version of the algorithm is possible to express hyperplane in a high dimensional feature space, and this can also be interpreted as a polynomial curve (of second degree in this case) in the original space.
For this reason the training and test error of the resulting predictor are significantly better than the previous version.
Specifically I obtained a training error of 0.085375, and a test error of 0.087.
This are the features obtained:

(1.87559817e+01 1.13401244e+00 7.68047788e+00 -1.32276315e+01
1.73791816e+01 -5.35283741e+00 1.04738036e+01 6.65533633e+01
3.38354273e+01 5.20890557e+00 -1.40000000e+01 -2.37767269e+00
-1.15700558e+01 3.58887386e+00 -7.70998879e+00 -3.37225323e+00
8.62691037e+00 5.29455094e+00 7.16899813e+01 -2.75843723e+00
-1.44740075e+01 1.87559817e+01 1.58098829e+00 -2.16046998e+01
4.71738784e+00 -9.24568568e-01 2.31787163e+00 -6.88714236e-01
6.36558311e+00 2.24640419e+02 -3.01817078e+01 1.13401244e+00
-1.01581267e+01 -5.54650087e+00 -1.37119017e+01 5.92267047e+00
6.13176127e+00 -1.91940656e+01 1.20197686e+01 -3.78679497e+00
7.68047788e+00 -4.32369339e+00 -2.30745221e-01 -8.05450142e+00
1.43389046e+00 -7.36485907e+01 4.14731220e+00 9.17178493e-01
-1.32276315e+01 9.31783828e-01 -2.90129736e+01 1.10054038e+01
9.29221437e+00 5.16458889e-02 5.24165775e+00 1.73791816e+01
-3.39061351e+00 1.79056771e+01 1.68705706e+01 1.02044524e+01
-7.88104212e+00 -5.35283741e+00 4.04206126e+00 1.28786306e+01
1.01040343e+01 9.69582051e+00 1.04738036e+01 1.03988287e+01
2.78928092e+00 -2.37491314e+01 6.65533633e+01 8.51087821e-01
6.32907160e+00 3.38354273e+01 -2.72097857e+00 5.20890557e+00
-1.40000000e+01)

4 Support Vector Machine

The Support Vector Machine (SVM) algorithm learn linear classifiers, finding a linear classifier that is the **maximum margin separator hyperplane** and so achive the maximum margin from all the point in the training set.

Given a linearly separable training set $S = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_n, y_n)\} \in \mathbb{R}^d \times \{-1, 1\}$ it's possible to find this hyperplane solving the following convex optimization problem with linear constraints.

$$\begin{aligned} &\min_{\boldsymbol{w} \in \mathbb{R}^d} \frac{1}{2}\|\boldsymbol{w}\|^2 \\ &\text{s.t. } y_t \boldsymbol{w}^\top \boldsymbol{x}_t \geq 1 \text{ for } t = 1, \dots, m \end{aligned}$$

The optimization problem that describe the Support Vector Machine is optimized using the Pegasos algorithm.
The Pegasos algorithm is a variant of the Stocastic Gradient Descent algorithm, where at each step a point (or a set of points in the mini-batch variant) is sampled randomly from the training set and the current predictor is updated with the negative gradient of the loss of that training example weighted by a learning rate factor η_t.
In case of Pegasos the learning rate factor η_t is choose at each step as $\frac{1}{\lambda t}$.
I implement the Pegasos algorithm using the standard variant with the hinge loss, and with the logistic loss (Described in the Logistic regression parameter).
Both this functions are convex upper bounds of the zero-one loss, and the λ regularization parameter allow to have a λ-strongly convex function to minimize with the gradient descent.

4.1 Naive

As I previously said I implemented Pegasos using two surrogate losses: hinge loss and logistic loss.
Now I describe the implementation with hinge loss, that differs from the logistic one only by the update step.
Recall that the hinge loss is defined as $l(y, \hat{y}) = \max\{0, 1 - y_t \hat{y}\}$
Given $Z_t = (X_t, Y_t)$ a random sample from the training set, the update rule for Pegasos is:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta_t \nabla \ell_{Z_t}(w_t)$$

Let be s_t the realization for the random variable Z_t
Where $\ell_{s_t}(w) = \left[1 - y_{s_t} \boldsymbol{w}^T x_{s_t}\right]_+ + \frac{\lambda}{2}||w||^2$ so

$$\nabla \ell_{s_t}(w) = -y_{s_t} x_{s_t} \mathbb{I}\{h_{s_t}(\boldsymbol{w}) > 0\} + \lambda w$$

Let $\boldsymbol{v_t} = y_t x_t I\{h_t(\boldsymbol{w_t}) > 0\}$ and choosing $\eta_t = \frac{1}{\lambda t}$ we have

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t(1 - \frac{1}{t}) + \frac{1}{\lambda t}\boldsymbol{v_t}$$

4.1.1 implementation details

I adapted this version of the algorithm from *Pegasos: Primal Estimated sub-GrAdient SOlver for SVM*[1]

Algorithm 2: Pegasos Algorithm

```
Input: $S$, $\lambda$, $T$
for $t = 1, 2, \dots, T$ do
    Choose $i_t$ uniformly at random.
    Set $\eta_t = \frac{1}{\lambda t}$. if $y_{it}\boldsymbol{w}_t^T x_{i_t} < 1$ then
        Set $(w)_{t+1} \leftarrow (1 - \eta_t\lambda)\boldsymbol{w}_t + \eta_t y_{it}\boldsymbol{x}_{i_t}$
    end
    else
        Set $(w)_{t+1} \leftarrow (1 - \eta_t\lambda)\boldsymbol{w}_t$
    end
end
Output $\boldsymbol{w}_{T+1}$
```

The name of the variables are adapted to be consistent with the pseudo code reported
Between the code presented in the lecture and this one presented in the paper there are some differences with the pseudocode presented during the lectures:

- The gradient descent update is written in a slightly different way using a conditional statement instead of the classical indicator function, I choose to remain consistent also with this stilistic choice.

- Instead of return the average of all the weight vector calculated at each step, the paper returns only the last one.
  The authors indicates that they notate an improvment in performance returning the last vector instead of the average.
  I embrace also this variation.

- In the pseudo code of Pegasos they also describe an projection step to clamp the magnitude of the linear predictor, but I don't incorporate it.

- The author also provide a mini-batch version of the Pegasos algorithm, with another hyperparameter (the mini-batch has size k).

Another approach proposed by the paper is **sampling without replacement**: so a random permutation of the training set is choosen and the updates are performed in order on the new sequence of data.
In this way, in one epoch, a training point is sampled only once.
After each epoch we can choose if we restart to sample data sequentially according to the same permutation or create a new one and sampling according that new order.
Although the authors report that this approaches gives better results than uniform sampling as I did, I haven't experiment this variant of the algorithm.

4.1.2 Hyperparameter tuning

Differently from the perceptron the Pegasos algorithm has an hyperparameter to choose: The regularization coefficent λ.
To choose the best hyperparameter for this algorithm (and the other implement) I choose to use the grid search method.
I divide the dataset into 3 different subset: training, test and validation set.
The validation set is used as a surrogate test set to obtain an estimate of the risk.

After splitting the data I choose a finite subset of the possible hyperparameter values $\Theta_0 \subseteq \Theta$ and for each of it create a predictor h_θ trained with the choosen hyperparameter on the training set.
The risk is then estimate using the validation error on each predictor, and the one with the lower risk is then choosen.

4.2 Logistic regression

4.3 Feature Expansion

4.3.1 Pegasos

4.3.2 Logistic Regression

5 Kernel

References

[1] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, Andrew Cotter *Pegasos: Primal Estimated sub-GrAdient SOlver for SVM.* https://home.ttic.edu/ nati/Publications/PegasosMPB.pdf