Manon GEORGET M2 MASH

Optimization for Machine Learning : Project

The goal of this project is to illustrate the theorical results of the course "Optimization for Machine Learning" and to implement the algorithms studied on a real Data Set.

1 Presentation of the Data Set

In this project, I work on a Data Set I have started studying a few months ago and on which I wanted to implement the following algorithms and to go further. The Data Set is downloaded from the website UCI (Machine Learning Repository) and it is a subset of data analyzed by Ahmed MM, Dhanasekaran AR, Block A, Tong S, Costa ACS, Stasko M (2015) in "Protein Dynamics Associated with Failed and Rescued Learning in the Ts65Dn Mouse Model of Down Syndrome". Our Data Set is called "*DataCortexNuclear*" and it is about medicine. Indeed, the data were taken during a medical experiment on mice. They represent the expression levels of proteins measured in those mice' cerebral cortex, which are very useful values because proteins are genes' products, so measurements of proteins' expression levels allow to have measurements of a gene. To be more precise, the website describes the Data Set as "Expression levels of 77 proteins measured in the cerebral cortex of 8 classes of control and Down syndrome mice exposed to context fear conditioning, a task used to assess associative learning".

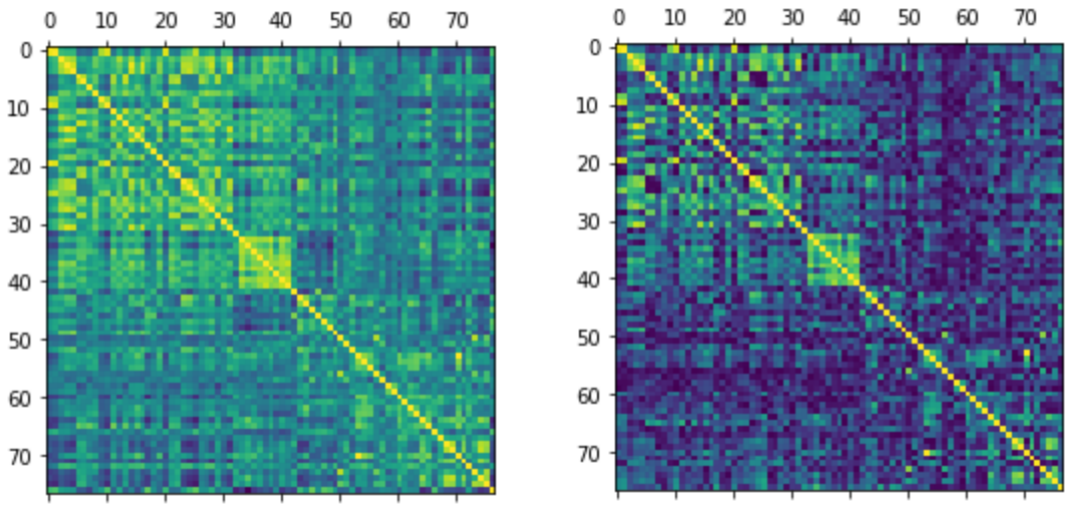
We have 72 mice which are in fact divided in two large groups : the first group contains Control mice (for which genotype is "Control") and the second one contains Trisomic mice which have Down syndrom (for which genotype is "Ts65Dn"). For each mouse, we have 15 measurements of the proteins. So the Data Set is composed of 1080 rows (1080 measurements which are considered as 1080 independant samples) and of 82 attributes in columns (the attributes are mouse's ID, genotype, treatment, behavior, class and the 77 different proteins). In total, we have 88 000 elements in our Data Set.

	MouseID	DYRK1A_N	ITSN1_N	BDNF_N	NR1_N	NR2A_N	pAKT_N
0	309_1	0.503644	0.747193	0.430175	2.816329	5.990152	0.218830
1	309_2	0.514617	0.689064	0.411770	2.789514	5.685038	0.211636
2	309_3	0.509183	0.730247	0.418309	2.687201	5.622059	0.209011
3	309_4	0.442107	0.617076	0.358626	2.466947	4.979503	0.222886
4	309_5	0.434940	0.617430	0.358802	2.365785	4.718679	0.213106
...	...	...	...	...	...	...	...

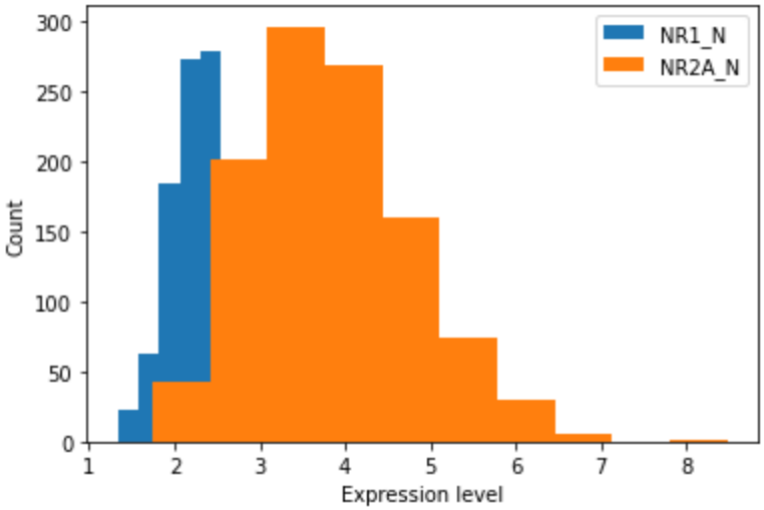
Visualization of a part of the Data Set

Before working on this Data Set, we study it a bit. We plot its covariance matrix and its absolute covariance matrix to see if we have some correlations. The clear squares (see figures below), which represent high correlation, show that some features are more correlated than others. It means that some expression levels of proteins are linked, which seems to be logic. For example, DYRK1-AN and ITSN1-N are really correlated. It is also the case for NR1-N and NR2A-N (which is consistent looking at the two

proteins' names). The particularity of our Data Set (and it is often the case for medical/genetic data) is that we have many features and that they are abstract to us. So it is difficult to have a precise idea of what those correlations represent.



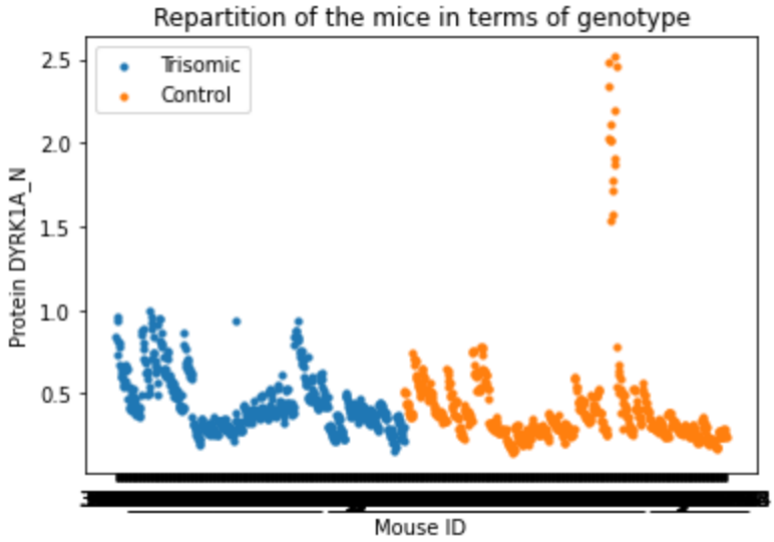
Covariance matrix and absolute covariance matrix of the Data Set



Histogram of two correlated proteins. We can observe a kind of symmetry for the counting even if the expression levels of NR2A-N are higher.

To only have quantitative values, we need to transform covariables which are categorical. For example, for the variable Genotype, Control ("Control") is represented by the number -1 and Down ("Ts65Dn") is represented by the number 1. We also replace the NA values by a value near the median of the corresponding protein's expression level, so it doesn't affect the classification.

We can now count and visualize the repartition of the mice in terms of genotypes. We plot against the samples' id and against the expression of the first protein. We remark that our samples seem to be fairly distributed between the two types of genotypes.



Representation of the two types of Genotype. The expression level of DYRK1-AN may be linked to the Down syndrom, as only Control mice takes some high values.

2 Presentation of the problem

2.1 Logistic regression

Our goal is to predict if a mouse has the Down Syndrom (which means if a mouse is Trisomic) only based on its proteins expression data, to see if proteins are sufficient informations to guess a diagnosis.

The output vector Y (denoted Y_{new} on the notebook) is then « Genotype ». The quantitative covariables are the proteins expression levels. The design matrix X is the matrix of the data associated to these explanatory covariables. We denote n the number of rows of X (ie observations) and we denote d the number of columns of X (ie attributes).

Y takes two values (-1 or 1) so the output space is discrete and we face a classification problem. We only have two classes so we do a logistic regression.

To have an idea of the results we should obtain with the next algorithms and to know the model's accuracy, we first do the logistic regression using sklearn, which returns the prediction vector using the GLM's method (with maximum likelihood estimator for the parameter). We remark that the method doesn't converge : maybe we have too many covariables or maybe the data aren't scaled so it is really slow. So we standardize the data : we center them by removing the mean value of each feature to have zero mean and we devide the features by their standard deviation to have variance one. This standardized design matrix is denoted X_{bis} on the Python notebook. Now, we can work on these data. The logistic regression seems to work well as our prediction Ytrain classifies nearly the same way as the true output Y_{new} and as we have an accuracy score of 0.98. We now use regularization. We first try a L2 penalisation (Ridge estimator) which reduces overfitting. Then, we try a L1 penalisation (Lasso estimator), which also allows to select covariables (indeed, it selects the coefficients associated to the revelant covariables, that's why the Lasso coefficient vector we obtain is sparse).

We test the performances of our models on training and testing sets. We slip our Data Set in two parts (testing set = 20% of the total Data Set), then we compute the estimators on the training set and we predict on the testing test. We evaluate the performance of our classifiers by computing the error, which is in this case the mean of the number of false classifications. After trying different values for the regularization parameter, we obtain an error of 0.09 for Ridge and of 0.08 for Lasso, which isn't bad.

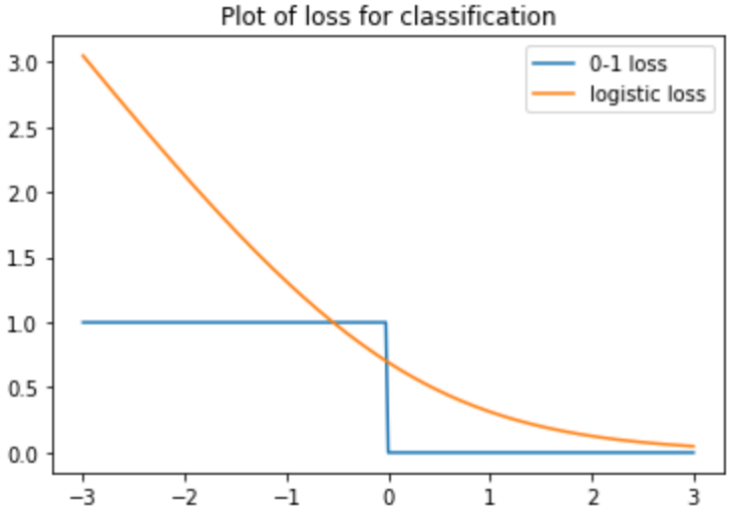
2.2 Theorical results about the objective function

We have seen that our problem is a logistic regression. So we are in the case of an empirical risk minimization method. Indeed, to assess the accuracy of our model in predicting the data, we try to

minimize an objective function. This objective is define thanks to a loss function, which is the 0-1 loss used in classification. The 0-1 loss returns 0 if the sign of the true label corresponds to the sign of the prediction and it returns 1 otherwise. In practice we use a smooth convex approximation of the 0-1 loss, which is the logistic loss. We define the sigmoid function such as : $\sigma(x) = \frac{1}{1+\exp(-x)}$. The loss function for a linear predictor $f(x) = \theta\ ^t x$ is then :

$l(y, f(x))) = log(1 + \exp(-yx\ ^t\theta))$

$= -log(\sigma(yx\ ^t\theta))$



The 0-1 loss and its smooth convex approximation

Our model classifies the data minimizing the empirical risk :

min $F(\theta) = \frac{1}{n}\sum_{i=1}^{n} log(1 + \exp(-yixi\ ^t\theta) + \lambda\frac{\|\theta\|^2}{2}$, with respect to $\theta \in R^d$ (where $\sum_{i=1}^{n} xi = X_{bis}$ in the notebook).

We look at some interesting analytical properties of the objective function F.

— F is continuous and coercive on R^d so there exists a minimum.

— As in the notebook LabSG01 of Clement Royer, we added an l2 penalisation so F is a strongly convex function and the minimizer is global and unique. Indeed, we know that the sum of a convex function with a factor $\lambda\frac{\|\theta\|^2}{2}$ is a λ-strongly convex function (in the Python notebook, we denote μ=λ to follow the course's notation). Having a strongly convex objective will be useful for some theorical results of the following algorithms (Stochastic Gradient for example).

— F is L-smooth (ie $C_L^{1,1}$), where $L = \frac{1}{4n}X\ ^tX + \lambda$. Indeed, F is twice differentiable and we can compute its gradient and its Hessian matrix :

1. $\nabla Fi(\theta) = -\frac{xi\ ^tyi}{1+exp(yixi\ ^t\theta)} + \lambda\theta$
2. $\nabla F(\theta) = -\frac{1}{n}\sum_{i=1}^{n} \frac{xi\ ^tyi}{1+exp(yixi\ ^t\theta)} + \lambda\theta$
3. $\nabla^2 F(\theta) = \frac{1}{n}\sum_{i=1}^{n} xi\ ^txiyi\ ^tyi\frac{exp(yixi\ ^t\theta)}{(1+exp(yixi\ ^t\theta))^2} + \lambda Id.$

   We use the fact that the yi's are smaller than 1 and that $\frac{exp(x)}{(1+exp(x))^2} \leq 1/4$ for all x to obtain that $\nabla^2 F(\theta) \leq \frac{1}{4n}X\ ^tX + \lambda = L$.

3 Gradient Descent

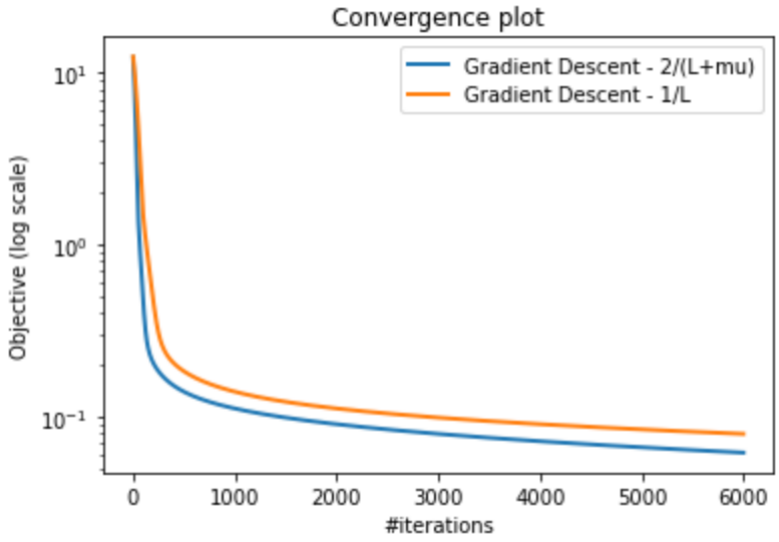
3.1 Convergence

We have an optimization problem for the objective function F that we first want to solve thanks to Gradient Descent algorithm. The Gradient Descent algorithm uses the update rule :

$\theta_{k+1} = \theta_k - \alpha_k\nabla F(\theta_k))$, $\alpha_k > 0$ for all k.

We test the convergence and we compare some step sizes (different values of constant step size, decreasing step size). We note that the use of backtracking line search for the step size isn't useful here as we have a L-smooth function. Indeed in this case we already know the good choices for α_k and backtracking would add an unnecessary cost.

We have a L-smooth function so the algorithm is a descent method for a step size $< \alpha_{max} = 2/L$. For a L-smooth and strongly convex function and for a constant step size $\alpha = 1/L$, the algorithm converges to the minimum value with the linear rate $O((1-\frac{\mu}{L})^k)$=$O((1-\frac{\lambda}{L})^k)$ (where k denotes the k-th iteration). For a L-smooth and convex function and for a constant step size $\alpha = 1/L$, the algorithm converges to the minimum value with the linear rate $O(\frac{1}{k})$. We try Gradient Descent with the objective defined in the last section (ie with a l2 penalization) and without penalization (ie $\lambda = 0$). In both cases, for $\alpha = 1/L$, the results we obtain in the plot are consistent with these theorical results, as we have convergence. We can also verify that the gradient of our minimizer $\hat{\theta}$ is approximately 0 as we know that for a convex function F, $\hat{\theta}$ being a minimizer of F is equivalent to $\nabla F(\hat{\theta}) = 0$. (for the next algorithms, we will always note that this first order condition is verified)

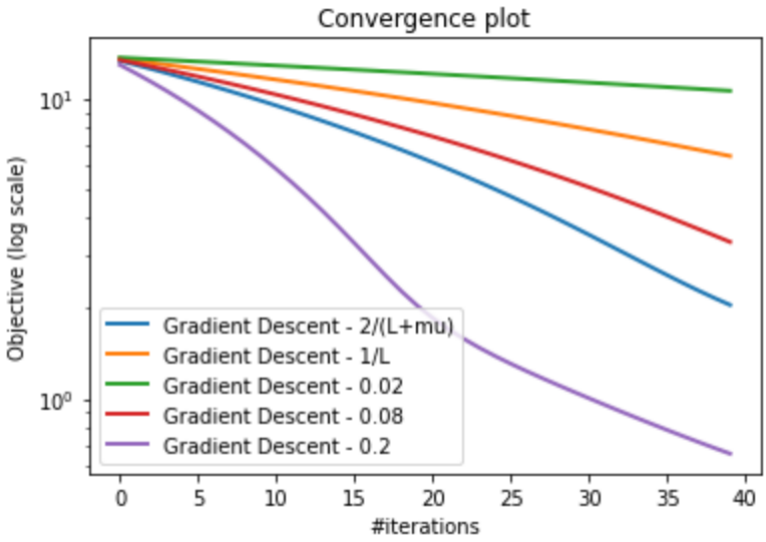


Gradient Descent with theorical constant step size and with optimal step size for quadratic functions (not the case here)

3.2 Constant step size

After trying the step size $\alpha = 1/L$ which gave us the theorical bound of convergence, we now test the impact of other constant step sizes.

- We test $\alpha = 2/(L + \lambda)$ which would be the optimal step size for a quadratic function and which is also a learning rate that guarantees linear convergence in our case (indeed, we have linear convergence for $\alpha \leq 2/(L + \lambda)$). We see that Gradient Descent converges to a little better value.
- We test $\alpha = 1/(L+\lambda)$. The difference between this stepsize and $\alpha = 1/L$ is minimal for our choice of λ. Only here, we choose another (larger) value for λ. It is enough to verify the theorical result we saw in the Advanced Gradient lectures (exercise 2 about convergence rates) : the rate is slower for $\alpha = 1/(L + \lambda)$ than for $\alpha = 1/L$.
- We test $\alpha = 0.02$, $\alpha = 0.08$ and $\alpha = 0.2$. As expected, for a small learning rate, the function decreases slowly because the algorithm takes a small step in the opposite direction of the gradient at each iteration so it doesn't varie a lot. It is the contrary for a large learning rate.



Gradient Descent with different constant step sizes

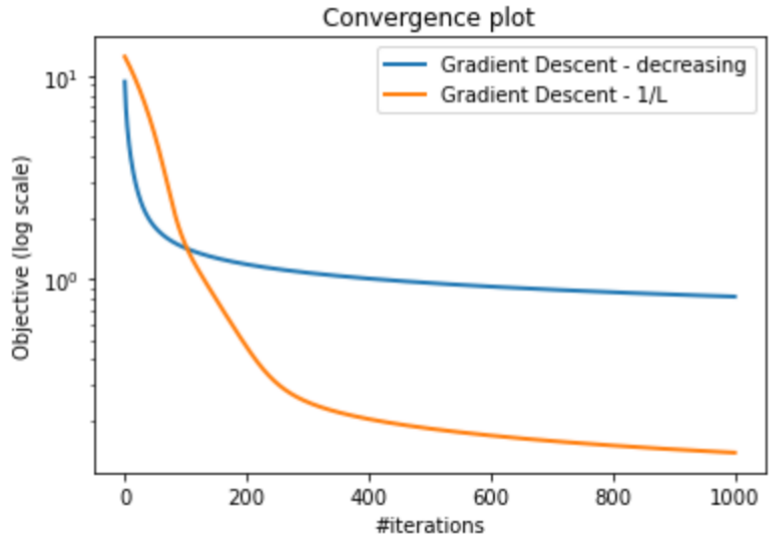
— To confirm this result, we try a really small constant step size $\alpha = 0.001$. The value after 1000 iterations is far from the minimum value we had for $\alpha = 1/L$ and the gradient is far from 0 : the convergence is worst.

— Finally, for $\alpha = \frac{2}{L} + 20 > \frac{2}{L}$, the values don't decrease and we have increasing peaks at some iterations. Indeed, we saw that choosing a too large step size doesn't allow the algorithm to be a descent method and it may diverge in some cases.

3.3 Decreasing step size

We want to test a step size which changes at each iteration. We define the decreasing step size $\alpha_k = \frac{1}{k+1}$ for all k. In this case, the values of F are decreasing at the first iterations, then the convergence slows down and after 1000 iterations, the average minimal value (for 10 runs) is larger than the one of Gradient Descent for constant step size $1/L$. So this stepsize doesn't seem to be a good choice. However, it is independant of the constant L, which is a good point when L is unknown.

```
decreasing : 0.8170689123809332 constant 1/L: 0.13917011851018815
```



Gradient Descent with a decreasing step size and with a constant step size

4 Accelerated Gradient Descent

In this section, we try to improve our convergence results. We saw in the course that there exists methods which can achieve better convergence rates and better complexity bounds thanks to an acceleration process. Indeed, Gradient Descent doesn't take into account the information from the precedent steps. The accelerated methods are first order algorithms which try to fix this problem, using a momentum term $\theta_k - \theta_{k-1}$.

As the Heavy Ball method is dedicated to quadratic functions, we implement the Nesterov method in the case of a strongly convex function. We compute the two loops version of the algorithm such that :

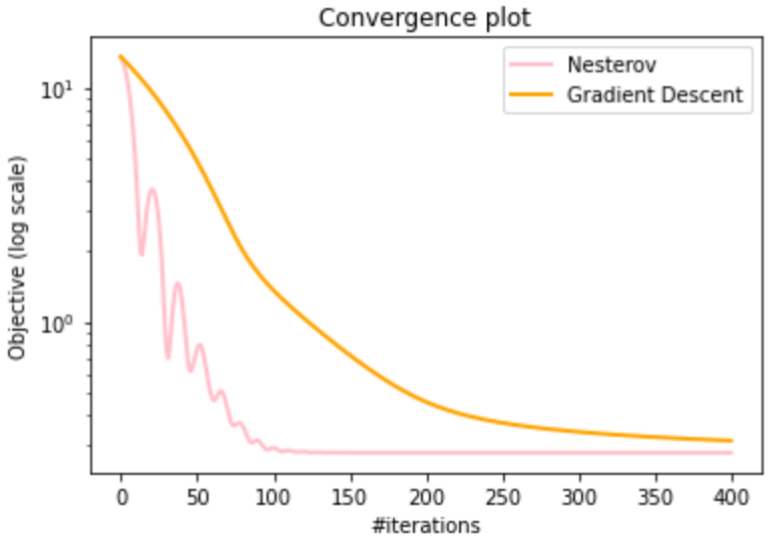
$\theta_{k+1} = zk - \alpha_k \nabla F(z_k))$

$z_{k+1} = \theta_{k+1} + \beta_{k+1}(\theta_{k+1} - \theta_k)$

This version allows to decouple the steps : we have a gradient step on z_k and a momentum step on θ_{k+1}.

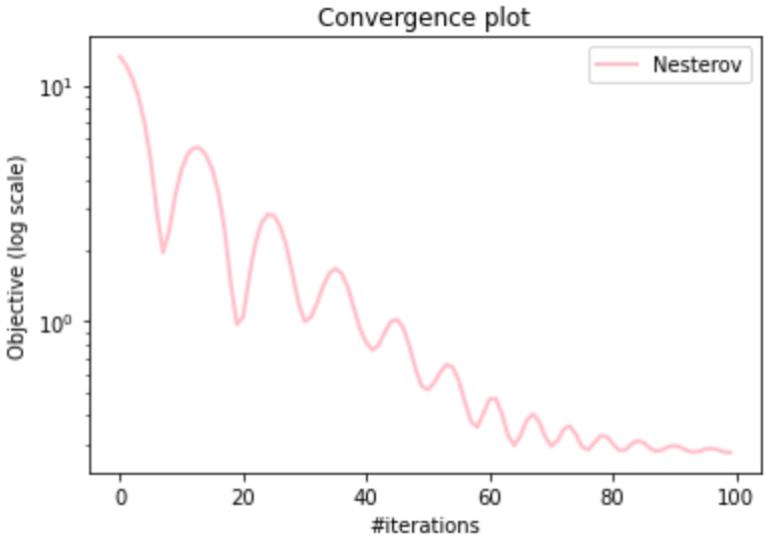
We plot the results for $\alpha = 1/L$ and for $\beta = \frac{\sqrt{L}-\sqrt{\lambda}}{\sqrt{L}+\sqrt{\lambda}}$ so we can compare them to the results obtained for Gradient Descent (with step size $\alpha = 1/L$). We see that Nesterov converges faster than a simple Gradient Descent. For instance, after 40 iterations and for 10 runs, Nesterov reaches the average minimal value of 1.37 whereas Gradient Descent only reaches 6.43. During the course, we saw in fact that the convergence rate for Nesterov is always better than the one of Gradient Descent, as it is a linear rate in $O((1-\sqrt{\frac{\lambda}{L}})^k)$.

```
after 40 iterations Nesterov : 1.3733498930399783 GD : 6.432931
after 400 iterations Nesterov : 0.2769136143019635 GD : 0.31160
```



Comparison of Accelerated Gradient Descent (Nesterov) and Gradient Descent with constant step sizes

Moreover, when we focus on the plot of Nesterov, we remark that this algorithm isn't a descent method as the function value increases at some iterations. It is due to the use of the momentum term. When it happens, the step improves at the next iteration so it is still faster than Gradient Descent.



Nesterov with constant step size

5 Stochastic Gradient

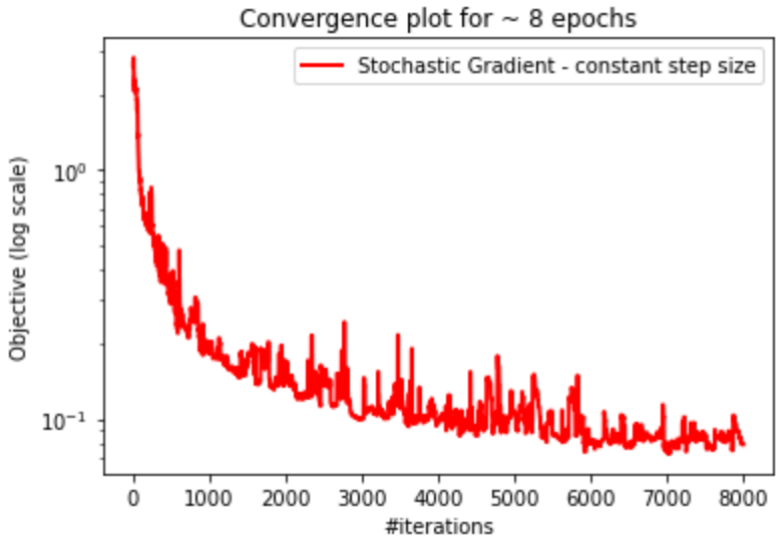
In this section, we try a stochastic method. Indeed, our empirical logistic risk is a finite sum, where every term in the sum only involves one example. When we implemented Gradient Descent, the gradient was then the average of all partial gradients. It means that for one iteration, the algorithm had to go over the entire Data Set. For a large Data Set, this has a big cost. To reduce this cost, we implement the Stochastic Gradient algorithm which chooses a random index $i \in \{1, ..., n\}$ and does the step in the opposite direction of the partial gradient of the component F_i. The update rule of Stochastic Gradient is then :

$\theta_{k+1} = \theta_k - \alpha_k \nabla F_{ik}(\theta_k)$, $\alpha_k > 0$ for all k, with $ik \in \{1, ..., n\}$ random.

5.1 Theorical and pratical results

We are in the case of a L-smooth and λ-strongly convex function. We suppose that the tree hypothesis seen in class are satisfied which means that the indices are independant, that the stochastic gradient component is an unbiased estimate of the full gradiant and that we can control the variance of the norm of this stochastic gradient. For example, these hypothesis are true for the Uniform$\{1, ..., n\}$ law, except maybe the last one as we don't know the bound in practice. Then for a constant step size $\alpha \leq 1/L$, the algorithm converges in a neighborhood of the optimal value. In fact we have convergence in expectation with the linear rate $O((1 - \alpha\lambda)^k)$ and some noise.

We plot the results for $\alpha = 1/L$. It looks consistent with these theorical results : we have convergence but an irregular behavior, which suggests the use of randomness and the convergence in a neighborhood. The plot also reminds us that stochastic gradient isn't a descent method.

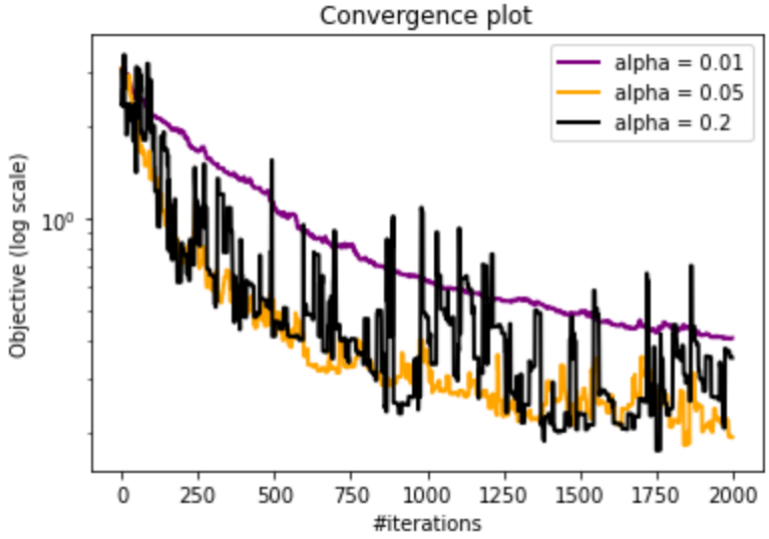


Stochastic Gradient with theorical constant step size

5.2 Test of step sizes

We try different values of step sizes and plot the results.

— We begin with fixed step sizes.

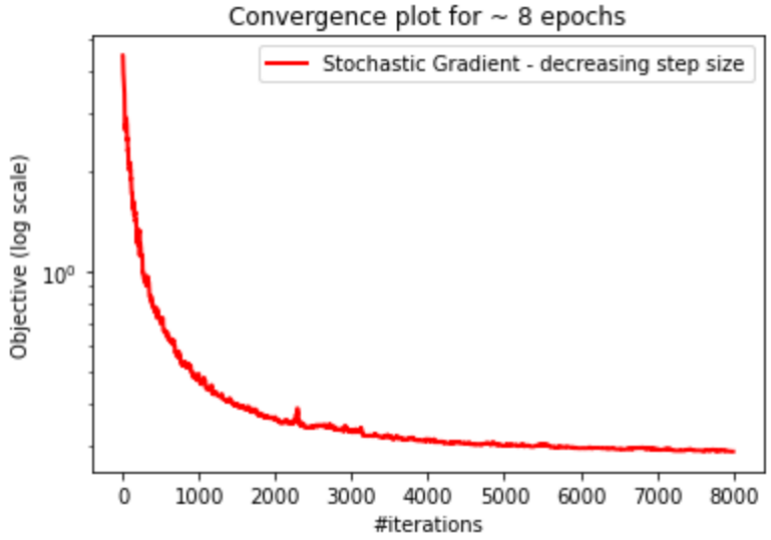


Stochastic Gradient with different constant step sizes

We see that Stochastic Gradient decreases quickly with a large step size (for example it is faster for $\alpha = 0.2$ than for $\alpha = 0.01$) but in return the behavior is really bumpy. The optimal step size seems to be $\alpha = 0.05$ as it has the best trade off between speed and smoothness.

— We test a decreasing step size $\alpha_k = \frac{\beta}{k+\phi} = \frac{24}{k+600}$, where k is the k-th iteration. For the value λ chosen in the Python notebook, we have $\alpha_0 = \frac{24}{600} \leq \frac{1}{L} = 0.05$ and $\beta = 24 > \frac{1}{\lambda} = 23.5$. By the course, we know that using a decreasing step size with these assumptions results in a sublinear convergence rate without noise. It is worse than the linear rate we had with a constant stepsize but it allows to reach any neighborhood of the optimal value which means that the difference between the minimal value found by the algorithm and the optimal value tends to 0 when the number of iterations tends to infinity. Indeed, we see on the plot that the curve is much smoother, especially for the last iterations, which suggests the absence of noise in the convergence. Moreover, when we look at the values, we note that the minimal average value (for 10 runs) of Stochastic Gradient

with decreasing step size is larger than the one of Stochastic Gradient with constant step size for 1000 iterations but most of the times it is smaller after 8000 iterations, which is consistent with the sublinear rate.



Stochastic Gradient with a decreasing step size

5.3 Mini Batch

In this part, we implement Batch Stochastic Gradient. This method is a variant of Stochastic Gradient which uses a gradient estimate built using several samples at once thanks to a Batch set S_k drawn at random with replace. The update rule for Batch is then :

$\theta_{k+1} = \theta_k - \alpha_k \frac{1}{|S_k|} \sum_{i \in S_k} \nabla F_i(\theta_k)$ for all k, with $\alpha_k > 0$ and $S_k \in \{1, ..., n\}$ a random set.

As we can see in the code below, Batch algorithm for a Batch size $|S_k| = 1$ (denoted nb in the notebook) corresponds to Stochastic Gradient. We can also verify that Batch algorithm for $|S_k| = n$ without replace coincides with Gradient Descent. So the choice of the Batch size can be seen as a trade-off between Gradient Descent and Stochastic Gradient.

```
#Batch Stochastic
def batch(rate,niter,theta,nb,f_objective):
    objvals = []
    n=Xnew.shape[0]
    for i in range(niter):
        indice = np.random.choice(n,nb,replace=True)
        sg = np.zeros(d)
        for j in range(nb):
            gradienti = gradientp(theta,Xnew,Ynew,lbda,indice[j])
            sg = sg + gradienti
        sg = (1/nb)*sg
        thetanew=theta-rate*sg
        theta=thetanew
        obj=f_objective(theta,Xnew,Ynew,lbda)
        objvals.append(obj)
    return(theta,np.array(objvals))
```

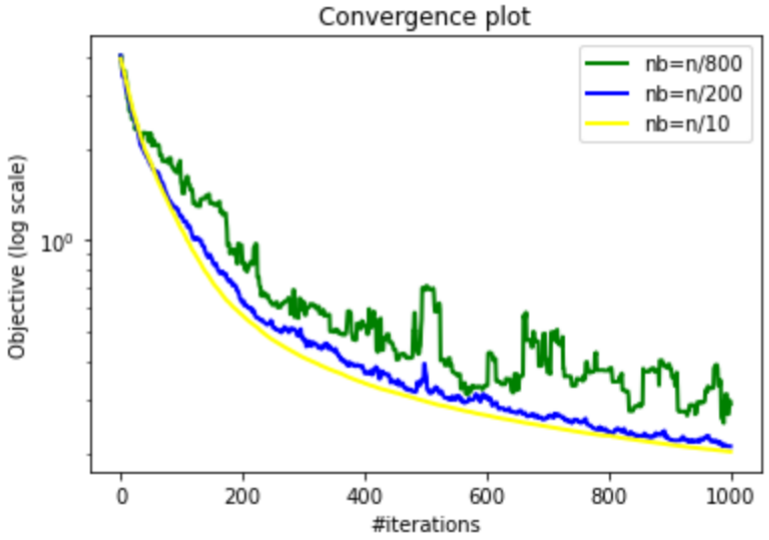
General Batch Algorithm which allows to choose the batch size nb

— We first do Mini Batch (that is Batch with $|S_k| \gg 1$ and $|S_k| \ll n$) with $|S_k| = 32$. Once again, we can link our practical results to the theorical results we saw in class. We know that the use

of a single sample is responsible of the importance of the variance term in Stochastic Gradient. With Mini Batch, we use 32 samples at once and it seems to reduce the variance of the component gradients as the plot is now smooth.

— We compare the performances of Mini Batch for three values of the batch size. We know that improving the Batch size reduces the variance and reduces the neighborhood.The convergence bound depends on the variance so it leads to a better performance. But when the Batch size is close to n, its cost is close to the cost of a full gradient so it is expensive and the algorithm can suffer from redundancies in the date : the behavior is worse. We test those results with $|S_k| = n/800$, $|S_k| = n/200$ and $|S_k| = n/10$. We obtain that the curve is smoother when $|S_k|$ increases, which means we have less variability. About the results, in our case $|S_k| = n/200$ and $|S_k| = n/10$ give (approximately) the same result at the last iterations, but $|S_k| = n/800$ (smaller Batch size) returns a larger value. So our practical results are mostly consistent with the theorical ones.

```
nb=n/800 : 0.28935174389277385 nb=n/200 : 0.2103582903595577 nb=n/10 : 0.2020
```



Batch algorithms for different Batch sizes

5.4 Comparison between the methods

In the last sections, we saw three methods (Gradient Descent, Stochastic Gradient and Mini Batch) which use different processes to minimize the objective function. We want to compare their performances.

— First, we compare Mini Batch and Stochastic Gradient.
We saw in the course that for a given number of iterations and for a given step size, Mini Batch method reaches a closer neighborhood of the objective's optimal value than Stochastic Gradient does. In fact, Stochastic Gradient needs $|S_k|$ times more iterations than Mini Batch to reach an equivalent value and Stochastic Gradient needs to choose the step size $\frac{\alpha}{|S_k|}$ to have a similar convergence result as Batch with step size α. But then the rate of Stochastic gradient becomes $O((1-\frac{\alpha\lambda}{|S_k|})^k)$ (plus noise), whereas the rate of Batch is $O((1-\alpha\lambda)^k)$ (plus noise), which is better. To compare each method with a consistent number of iterations, we use the notion of epoch. One epoch is a metric equivalent to n accesses to a data point ie n computations of a partial gradient ∇F_i, which means the entire Data Set is used in one epoch. By definition, n iterations of Stochastic Gradient is equivalent to one epoch and $\frac{n}{|S_k|}$ iterations of Batch is equivalent to one epoch. We try for 60 epochs, which is equivalent to $60n$ iterations of Stochastic Gradient and to $60\frac{n}{|S_k|}$ iterations of Batch. We obtain that Stochastic Gradient has a better minimal value than Mini Batch (for 10 runs, we obtained the average minimal value of 0.03 for Stochastic Gradient and of 0.12 for Mini Batch). So using epochs as a metric, Mini Batch isn't always better than Stochastic Gradient, but it always has the advantage of reducing the variance.

— Then, we compare Stochastic Gradient and Gradient Descent.

We saw in the course that Stochastic Gradient is n times cheaper than Gradient Descent as it uses only one data point per iteration. So Stochastic Gradient reaches a better value than Gradient Descent for the same number of iterations.
Once again, we have to use the notion of epoch to do the comparison, as the cost of one iteration of Gradient Descent is different from the cost of one iteration of Stochastic Gradient. Indeed, one epoch is equivalent to one iteration of Gradient Descent. So 60 epochs is equivalent to 60 iterations of Gradient Descent and to $60n$ iterations of Stochastic Gradient. For 60 epochs, we note that Stochastic Gradient Algorithm has a better value than Gradient Descent Algorithm (for 10 runs, we obtained the average minimal value of 0.028 for Stochastic Gradient and of 1.679 for Gradient Descent). It confirms the fact that Stochastic Gradient is very useful when the Data Set contains many observations.

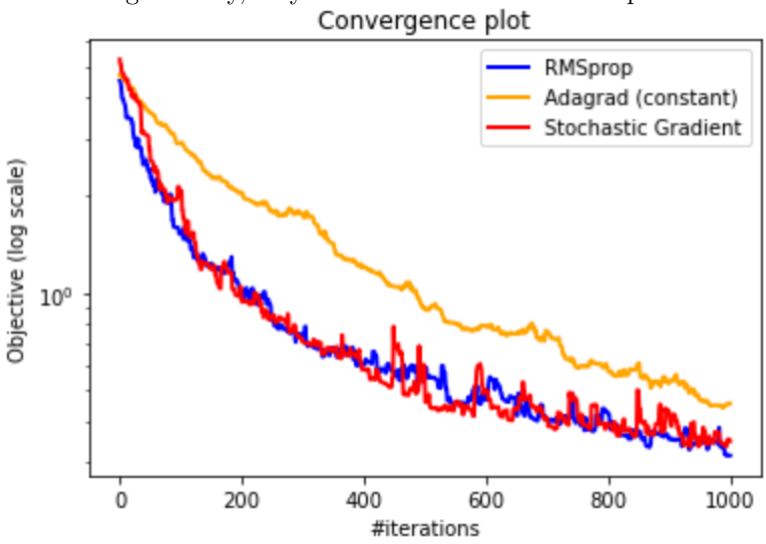
5.5 Advanced Stochastic Gradient methods

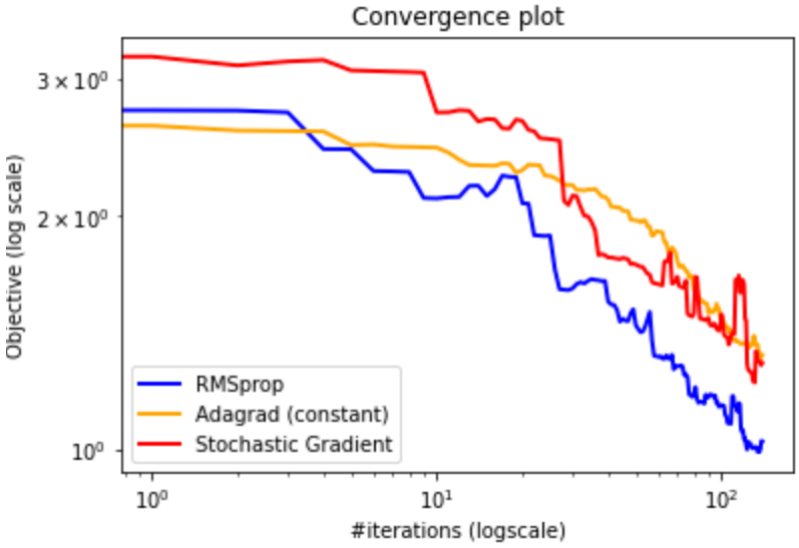
To finish about this part, we want to implement some useful variants of Stochastic Gradient.

5.5.1 Diagonal scaling

Stochastic Gradient is sensitive to conditioning as its convergence results depend on the lipschitz constant L. So it is not invariant to transformation of the data (linear transformation for example). We want to have scale invariant algorithms, which aren't affected to these transformations. We try diagonal scaling methods, which use the second order information to rescale the Stochastic Gradient step componentwise. We implement RMSprop and Adagrad with constant step size $\alpha = 1/L$. For both methods, we choose $\lambda = \frac{1}{2\sqrt{n}}$ and for RMSprop, we choose $\beta = 0.8$ in the recursion of $[R_k]_i$ (as suggested in the notebook of Clément Royer). We also add a regularization parameter $\mu > 0$ which allows to escape division by 0

As we can see on the plots below, the curves of RMSprop and Adagrad are a bit smoother than the one of Stochastic Gradient. Moreover, RMSprop seems to reach the better minimal value after 1000 iterations (RMSprop : 0.35, Adagrad : 0.49, Stochastic Gradient : 0.37), whereas Adagrad is decreasing too but converges slowly, maybe because it does small steps.





Stochastic Gradient VS Stochastic Gradient with diagonal scaling

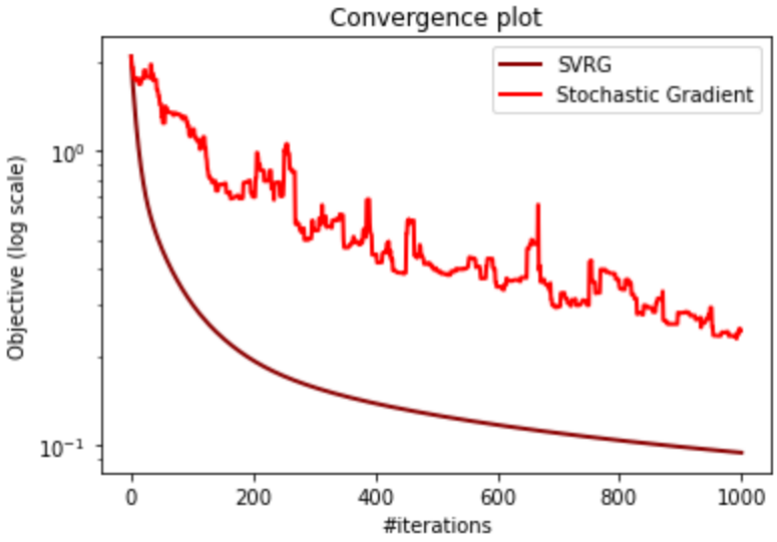
5.5.2 Variance reduction

Mini Batch allowed us to reduce the variance and to have a better convergence bound. We now want to try an aggregation method, which should also reduce the variance of Stochastic Gradient. We implement the SVRG method (Stochastic Variance Reduced Gradient). This algorithm computes a full gradient at the beginning of some major iterations, then it adds a partial gradient during m sub-iterations, as we can see in the code below in the loop "for j in range(m)".

```
def SVRG(rate,niter,theta0,f_objective,x,y,m):
    objvals = []
    n=x.shape[0]
    theta=theta0
    obj=f_objective(theta,x,y,lbda)
    objvals.append(obj)
    for i in range(niter):
        g=gradient(theta,x,y,lbda)
        wtilda=theta
        for j in range(m):
            ind=np.random.choice(n,1,replace=True)
            sg=gradientp(wtilda,x,y,lbda,ind[0])-gradientp(theta,x,y,lbda,ind[0])+g
            wtilda = wtilda-rate*sg
            if (i+n+j)//n>(i+n)//n:
                objvals.append(obj)
        theta=wtilda
        obj=f_objective(theta,x,y,lbda)
        objvals.append(obj)
    return(theta,np.array(objvals))
```

SVRG Algorithm with m sub-iterations

We test this method on our data for $m = 5$ sub-iterations. We can observe that the curve is much smoother than the one of Stochastic Gradient as we don't have oscillations at all, which shows the decrease of variability.



Stochastic Gradient VS Stochastic Gradient with SVRG ($m = 5$)

About SVRG convergence properties, we can obtain a linear convergence without additional noise under some assumptions for the step size. We also know that, using epoch as a metric, Stochastic Gradient is in general more efficient than SVRG during the first epochs. SVRG needs to compute $n+2m$ gradients at each iteration so one epoch is equivalent to $\frac{n}{n+2m}$ iterations of SVRG. For 40 epochs, Stochatic Gradient reaches indeed a smaller value than SVRG (value of 1.01 for SVRG and value of 0.38 for Stochastic Gradient). But for more epochs (200), SVRG is better in most of the cases (value of 0.26 for SVRG and value of 0.30 for Stochastic Gradient).

6 Regularization

Since the beginning, the objective function to minimize was penalized with an l2 norm and a regularization parameter $\lambda > 0$, which was useful for the convergence of stochastic methods (function strongly convex). We also tried Gradient Descent without regularization and observed that the l2 penalization didn't had a real impact on the minimization in our case.

In this part, we test the impact of another regularization such that we want to minimize :

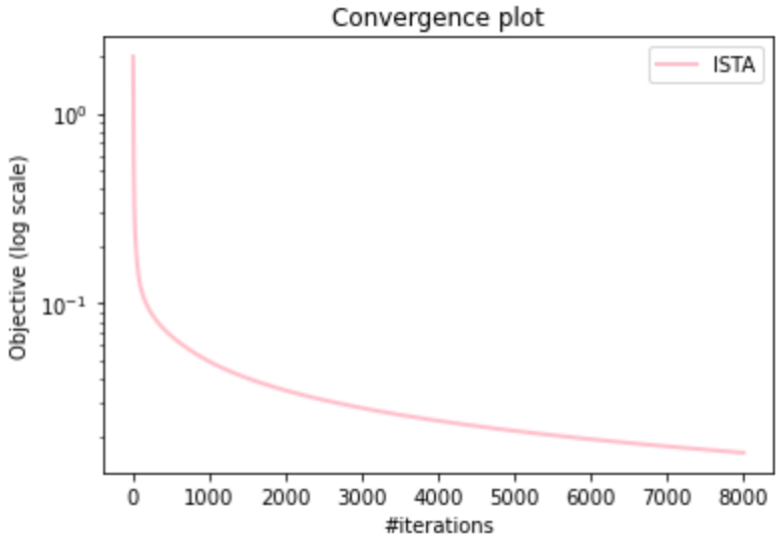
min $F(\theta)+\lambda R(\theta)$ with respect to $\theta \in R^d$, where $R(\theta)$ is the regularizer and $\lambda > 0$ is the regularization parameter.

Here we test a sparse regularization which uses the l1 norm to performs features selection. The new objective function to minimize is then :

min $f(\theta_k) = \frac{1}{n}\sum_{i=1}^{n} log(1+\exp(-yixi\ ^t\theta) + \lambda\|\theta\|_1$ with respect to $\theta \in R^d$ (where $X_{bis} = \sum_{i=1}^{n} xi$ in the notebook).

To perform the minimization, we implement the ISTA algorithm (iterative soft thresholding), which can be viewed as an extension of Gradient Descent as it does a gradient step on the smooth part of F (forward step) and then a proximal step (backward step) thanks to the l1 penalty. The proximal step is linked to the subdifferential of the l1 norm and it is defined using : $s_\lambda(x) = max(|x| - \lambda, 0) * sign(x)$, where $s_\lambda(x)$ is called the soft thresholding operator.

The update rule for the ISTA algorithm is then : $\theta_{k+1} = s_{\lambda\alpha}(\theta_k - \alpha\nabla f(\theta_k)))$, with $\alpha > 0$.



ISTA algorithm with regularization parameter $\lambda = \frac{1}{n^2} > 0$

For a well-chosen regularization parameter λ and for $\alpha = 1/L < 2/L$, we obtain that the ISTA algorithm converges to a little smaller value than Gradient Descent without regularization and than Gradient Descent with l2 regularization.

Finally, we can plot the weights to see which features are selected by Lasso (ie which proteins are linked to our problem). We note that some weights are near zero, which means that the related proteins aren't really relevant for the classification.

7 Classification

In the second part, we did a logistic regression with sklearn which predicted the classification between the two genotypes on a testing set. We now want to try the logistic regression using our empirical risk minimization algorithms.

We denote $\hat{\theta}$ a minimizer found by each algorithm. The classifier is then : $\hat{Y} = sign(X_{bis}\ {}^t\hat{\theta})$. For each method, we compute the proportion of classification errors : $\frac{1}{n}\sum_{i=1}^{n} 1\hat{y_i} \neq y_i$, where y_i is the true class.

First, we evaluate the performance of our predictors on the entire Data Set or on the training test to see if it works. We obtain an error rate of 0.03 for Gradient Descent (accuracy = 0.97), of 0.07 for Stochastic Gradient (accuracy = 0.93) and of 0.05 for Gradient Descent with Lasso regularization (ISTA) (accuracy = 0.95). Using sklearn, we had a model accuracy of 0.98, so the results are more or less similar.

Finally, we split our Data Set in two parts. We compute our minimizers $\hat{\theta}$ on the training set and we evaluate the performance of their associated classifiers on the training test. We obtain an error of 0.08 for Gradient Descent, of 0.18 for Stochastic Gradient and of 0.07 for Gradient Descent with Lasso regularization (ISTA). So on the testing set, the l1 penalisation seems to be the best. We saw that the algorithm using Soft Thresholding returns the minimal average value of the objective function, which means it is the method which minimizes the empirical risk. So it seems consistent that it has the best classifier. With sklearn, for the Ridge estimator (such as Gradient Descent and Stochastic Gradient with our l2-penalised objective), we had an error rate of 0.09. For the Lasso estimator with l1 penalisation (such as ISTA), we had an error rate of 0.08. The performances of our algorithms with L2 penalisation are a litte worse than they should on the testing test.

8 Conclusion

This project presented and compared different methods to solve a logistic regression problem thanks to the empirical risk minimization. We can note that most of the theorical results we studied in class

can be observed in practice (and maybe the stochastic properties would be even more obvious if we had a larger Data Set). Moreover, this project give us a first good approach of algorithms such as Gradient Descent, Stochastic Gradient, Mini Batch or ISTA, which are very useful in many types of problems.

Note : The codes implemented on this project are based on the Jupyter Notebooks of Gabriel Peyré and of Clément Royer.