Newton's Method with Exact Line Search

Jay Senoner

April 11, 2024

Abstract

It is well known that, for Newton's method to have global convergence properties, the introduction in the algorithmic of a line search procedure to select a suitable stepsize is needed. Since the computation of the Hessian matrix dominates the computational cost of each iteration, one might think of carrying out an (almost) exact line search along Newton's direction instead of an inexact backtracking procedure like Armijo's

1 Outline of the work

This lab report discusses the project assignment "Newton's Method with Exact Line Search," assigned by Professor Matteo Lapucci for the "Optimization Techniques for Machine Learning" course in the second cycle degree in Artificial Intelligence, held at the Univeristy of Florence. The source code is available at this GitHub public repository.

The main objective is to reproduce the empirical results outlined in [1]. The report provides a brief theoretical introduction to the logistic regression problem and the techniques employed for its solution, followed by an overview of the implementation details, including algorithms and exact line search techniques used, as well as the dataset and problem description. Finally, the report presents the comparative results of all algorithms in terms of error evolution with respect to the number of iterations and runtime.

2 Theoretical introduction

2.1 Logistic regression

The problem of fitting a logistic regression model is equivalent to the following optimization problem:

$$\min_{w \in \mathbb{R}^p} \mathcal{L}(w) + \lambda \Omega(w)$$

where $w \in \mathbb{R}^p$ are the *weights*, $\mathcal{L}(w)$ is the negative log-likelihood of the logistic model, λ is the *regularization strength* and $\Omega(w)$ is a convex regularizer. As it is shown in Appendix. A of [2], if the set in which the labels are taken changes from $\{0, 1\}$ to $\{-1, 1\}$, the negative log-likelihood of the logistic model $\mathcal{L}(w)$ can be written as

$$\mathcal{L}(w) = \sum_{i=1}^{m} \log(1 + \exp(-y_i w^T x_i))$$

where $x_i \in \mathbb{R}^n$ is the $i-$th example in the dataset and $y_i \in \{-1, 1\}$ is the *label* of the corresponding example. A typical example of convex regularizer added to the problem to reduce overfitting and assure the strongly-convexity of the objective function is the L2-regularizer, which is a convex regularizer of the form

$$\Omega(w) = ||w||^2$$

With that being said, the formal expression of the problem that is being solved in this project work is the following:

$$\min_{w\in\mathbb{R}^p}\sum_{i=1}^{m}\log(1+\exp(-y_i w^T x_i))+\lambda||w||^2$$

Since there is no closed form solution to this kind of problem, iterative algorithms such as Newton's method or gradient descent are required to train the model.

2.2 Hybrid and Greedy Newton

For minimizing a twice differentiable objective function $f:\mathbb{R}^n\to\mathbb{R}$ the pure Newton iteration x_N^{k+1} at the point x^k takes the form

$$x_N^{k+1}=x^k-\nabla^2 f(x^k)^{-1}\nabla f(x^k)$$

In optimization, Newton's method is a powerful tool for minimizing non-linear objectives mainly due to its remarkable property of superlinear (quadratic) convergence in a neighborhood of a strict local minimum (under appropriate conditions)[1]. However, the algorithm is not guaranteed to converge in all possible scenarios, even if the objective function f is strictly decreasing in its domain. One of the proposed fixes that adresses this problem is to add a stepsize $\alpha_k\in\mathbb{R}$, so the Newton iteration becomes:

$$x_N^{k+1}=x^k-\alpha_k\nabla^2 f(x^k)^{-1}\nabla f(x^k)$$

With $\alpha_k=1$ we get back the pure Newton iterations. Another common option to select the value of α_k is to use a backtracking line search procedure (like the Armijo line search) to select a suitable(relative to some fixed condition) α_k at each iteration k. The paper [1] investigates Newton's method where the stepsize α_k is chosen to be the exact minimizer of the objective function evalued at the point x_N^{k+1}:

$$\alpha_k\in\arg\min_{\alpha} f(x^k-\alpha\nabla^2 f(x^k)^{-1}\nabla f(x^k))$$

This variation of Newton method is called **Greedy Newton**. Another possible variant that is studied in the Greedy Newton paper [1] considers the possibility to select the best step between two different methods. The **Hybrid Newton** method proposed in [1] acts like the following:

- Let x_N^{k+1} be the pure Newton step and x_G^{k+1} the gradient descent with exact line search step:
$$x_G^{k+1}=x^k-\alpha_k\nabla f(x^k),\;\;\alpha_k\in\arg\min_{\alpha} f(x^k-\alpha\nabla f(x^k))$$
- Between the two computed steps x_N^{k+1} and x_G^{k+1}, choose the one that minimizes the objective function the most, i.e.
$$x^{k+1}=\begin{cases}x_N^{k+1} & f(x_N^{k+1})<f(x_G^{k+1})\\ x_G^{k+1} & \text{otherwise}\end{cases}$$

In the subsequent section, we will provide a detailed explanation of the implementation of these techniques, and how the exact line search is realized in practice.

3 Implementation Details

The project was realized in Python language, using the NumPy library. The project is structured in 3 modules:

- dataset.py
- solvers.py
- main.py

3.1 Dataset module

The dataset module contains all the functions that are necessary to initialize the dataset using a synthetic data generating process as described in the experiments section of [1]. Additionally, this module implements classes and functions facilitating the construction of regularized and non-regularized instances of a logistic regression problem. It also enables the computation of the loss function $\mathcal{L}(w)$ and its corresponding gradient and hessian functions. Moreover, this module offers a simple method that can test a single solver by passing the name of a callable function in the solvers.py module. The provided testing method, given a solver, executes the solver on the problem defined by the dataset object. Subsequently, it logs various solver-specific results, including the number of iterations required, the absolute error between the optimal value (computed with the SciPy.minimize routine) and the obtained solution, the loss value at the solution point and two arrays: one containing all the absolute errors evaluated at each iteration between the current loss point and the loss at the optimal point, and another containing the time elapsed for each iteration. All logistic regression problems generated by this module are formulated in an unbiased manner, which aims to further simplify the implementation of loss-related functions.

3.2 Solver module

The main purpose of this module is to implement iterative optimization algorithms to solve the logistic regression problem and to provide efficient line search routines to be used by the algorithms. Moreover, all the algorithms implemented provide a way to measure time elapsed and absolute error between the optimal value of the objective function and the current function value at each iteration. In this module a total of 6 different solvers were implemented:

- Gradient descent with Armijo line search
- Gradient descent with Greedy (exact) line search [1]
- Standard Newtons' Method ($\alpha = 1$)
- Newton's Method with Armijo line search
- Greedy Newton(Newton's method with exact line search) [1]
- Hybrid Newton [1]

Every solver in the solvers.py module is a callable method of the Solver class, which takes two main parameters as inputs: the dataset object, representing the specific instance of the logistic regression problem to be solved, and the initial weights vector, representing the starting point of the solver. For solvers requiring an "exact" line search routine, two options are available:

- The exact line search routine from SciPy, specifically "minimize_scalar," which is a Python implementation of the unbounded Brent method.
- An "Approximately Exact Line Search" routine referred to as AELS, as presented in [3].

In specific cases where solvers need to call an exact line search routine, an additional boolean parameter named "aels" enables the selection of one of the implemented exact line search techniques. If "aels" is set to True, the solver will use AELS; otherwise, it will use minimize_scalar. The default value for the parameter "aels" is True.

3.3 Main module

The main module is tasked with providing all the functions necessary to conduct the experiments outlined in [1]. It includes a function to initialize a dataset object with a custom number of examples and features. Additionally, it offers functions designed to perform specific tests on a given set of solvers. There are 2 principal methods for testing:

- **test_all_and_plot**: This method tests all the solvers by creating a custom instance of the logistic regression problem. It then calls the testing method from the dataset module to test each solver individually. With the results obtained after the testing, two plots are generated for each solver: one showing how the error $f(x^k) - f^*$ evolves with respect to the number of iterations, and one showing how the same error evolves with respect to runtime.
- **test_compare**: This method aims to compare the performance between the two different exact line search techniques presented. Specifically, it tests all the exact line search solvers with both the minimize_scalar and the AELS routine. Then, it generates two plots for each solver: one containing the algorithms that performed better in terms of runtime and the other containing the methods that performed better in terms of number of iterations required to reach the solution.

4 Experiments

To conduct the experiments outlined in [1], we first set the parameter *lambda_reg* in the dataset.py module to 1. Then, we execute the test_all_and_plot method four times, each time adjusting the values of the input parameters to generate instances with the required dimensions for the specific test. In addiction to what has been done in [1], we utilize the test_compare method to determine which exact line search technique performs better for each solver, considering both runtime and the number of iterations. By changing the value of the lambda_reg parameter to zero, unregularized instances can be tested using the same code. All instances generated for testing the solvers have the same number of examples set to 500, while the number of features is gradually increased. This variation in the number of features is intended to demonstrate how the performance of different solvers varies as the number of variables increases.

4.1 Results for L2-regularized instances

The following results are obtained by executing the described testing methods with the specified list of solver parameters:

- Parameters for Armijo line search:

  - $\gamma = 0.3$
  - $\delta = 0.25$
  - max_iter_armijo = 1000
  - initial_stepsize = 1

- Parameters for AELS:
  - $\beta = \frac{2}{1+\sqrt{5}}$
  - max_iter_AELS = 1000
  - initial_stepsizeAELS = 1.7

- Stopping criterion parameters:
  - max_iter = 1000
  - eps = 10^{-6}

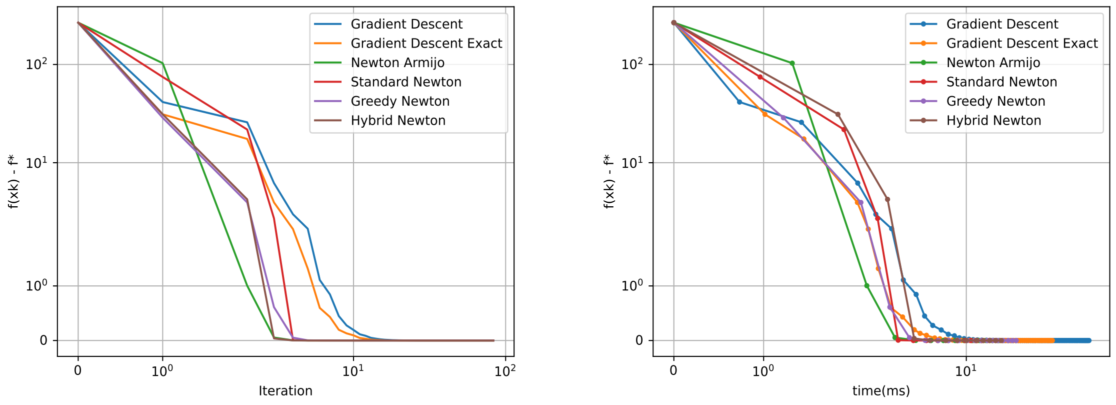

Figure 1: 20 features

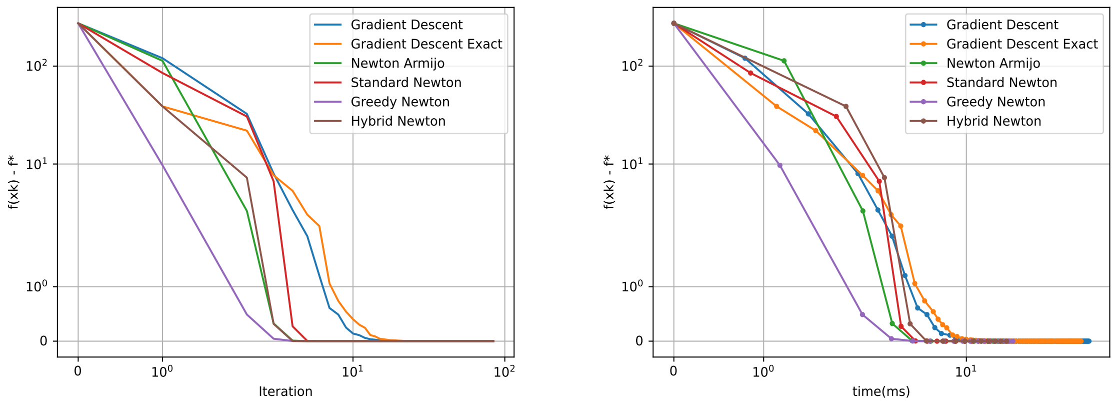

Figure 2: 20 features,10 repeated

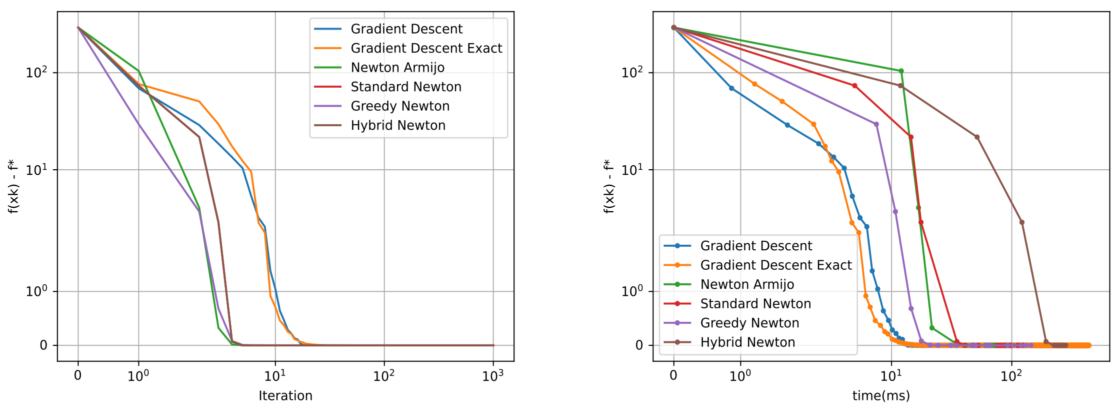

Figure 3: 200 features

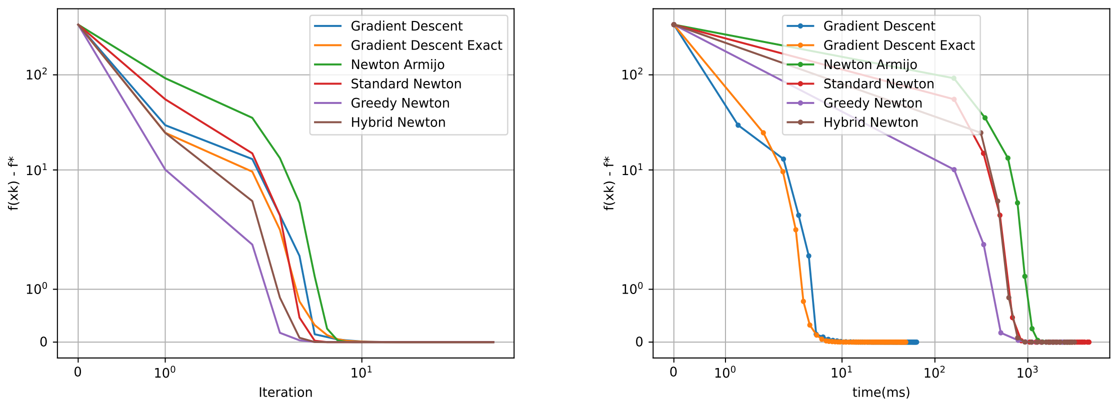

Figure 4: 2000 features

4.2 Results for unregularized instances

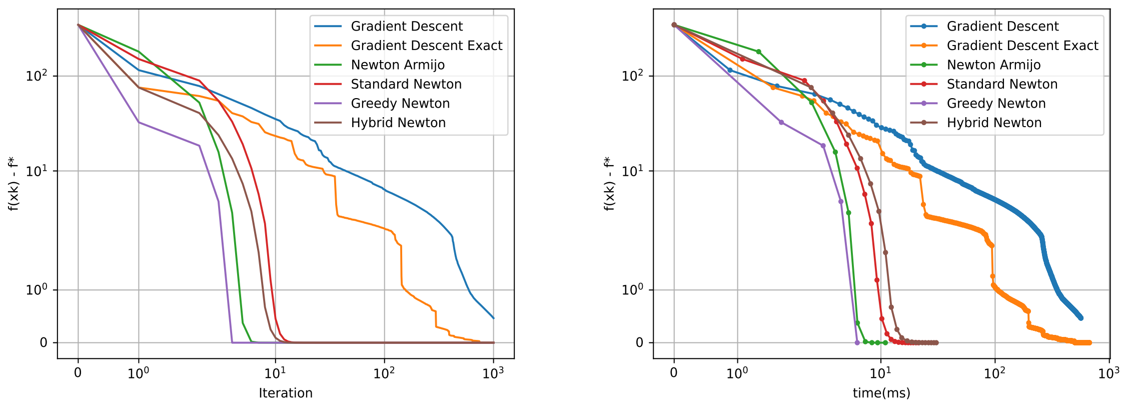

Figure 5: 20 features

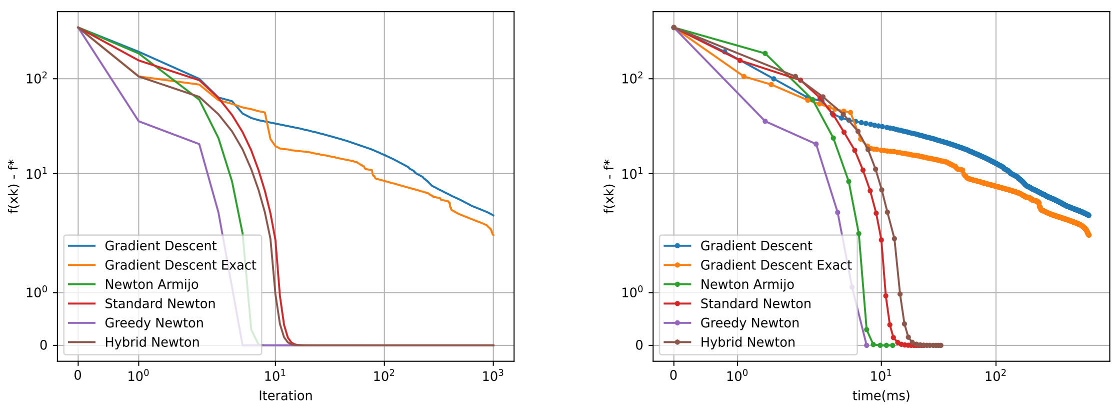

Figure 6: 20 features,10 repeated

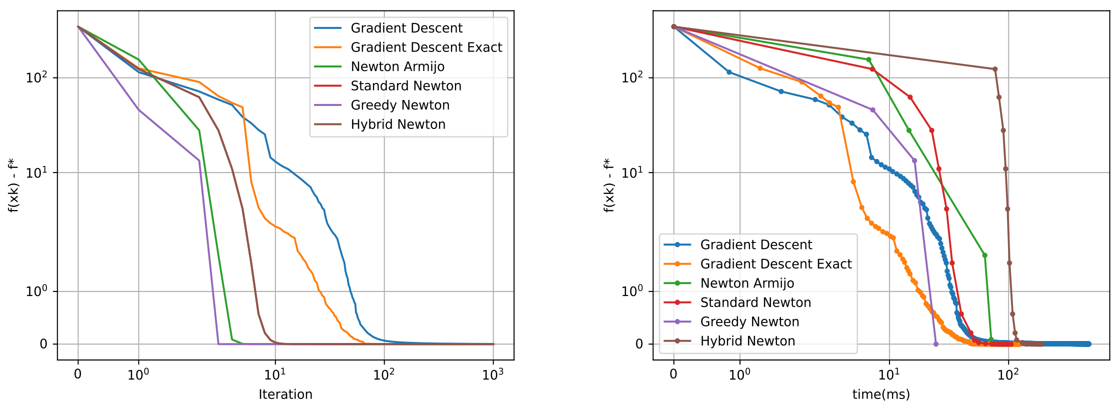

Figure 7: 200 features

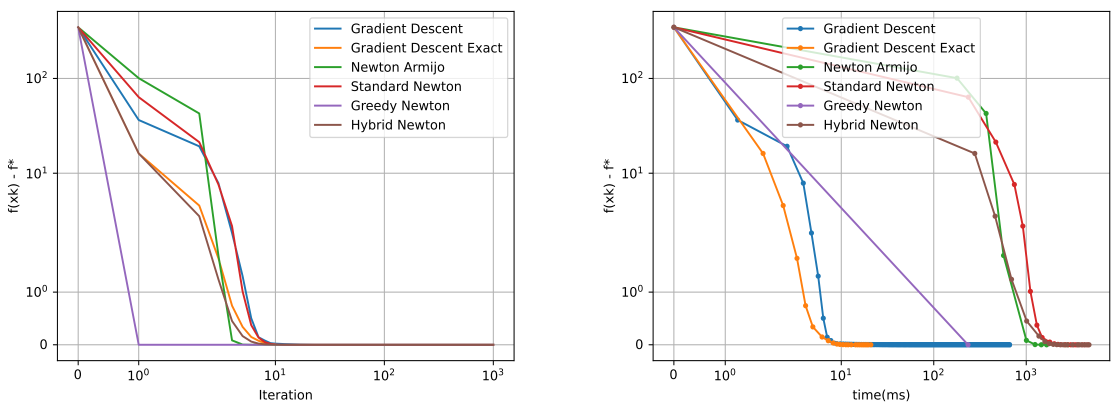

Figure 8: 2000 features

4.3 Comparison between AELS and minimize_scalar

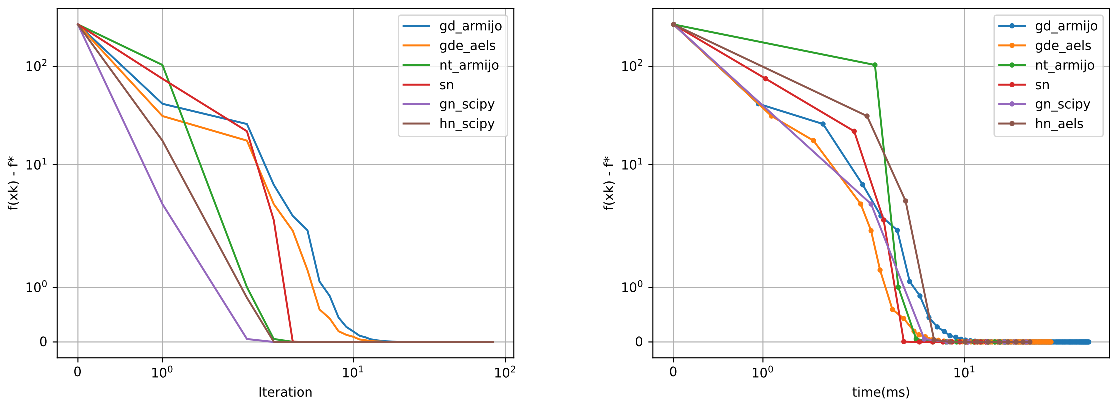

Figure 9: 20 features, regularized

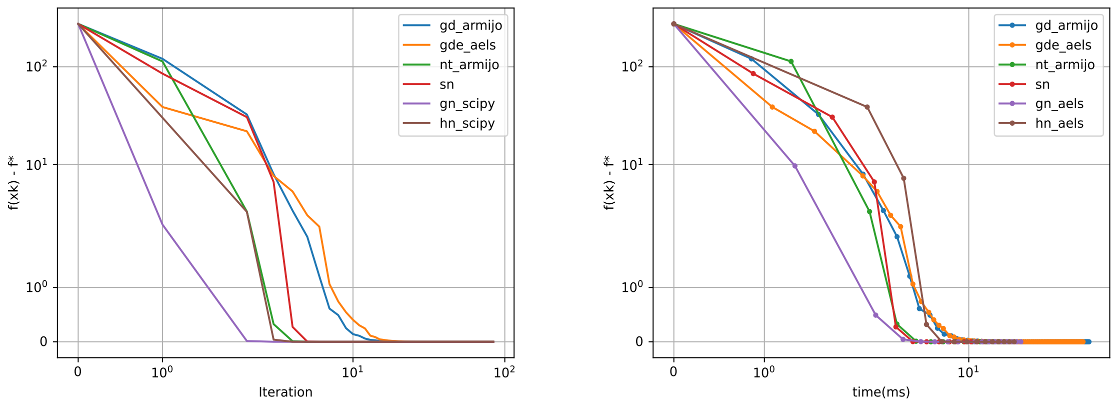

Figure 10: 20 features, 10 repeated, regularized

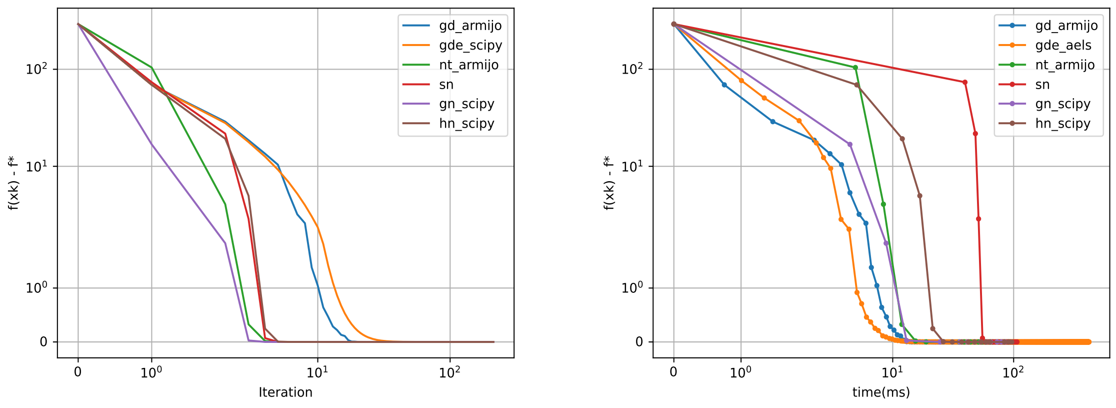

Figure 11: 200 features, regularized

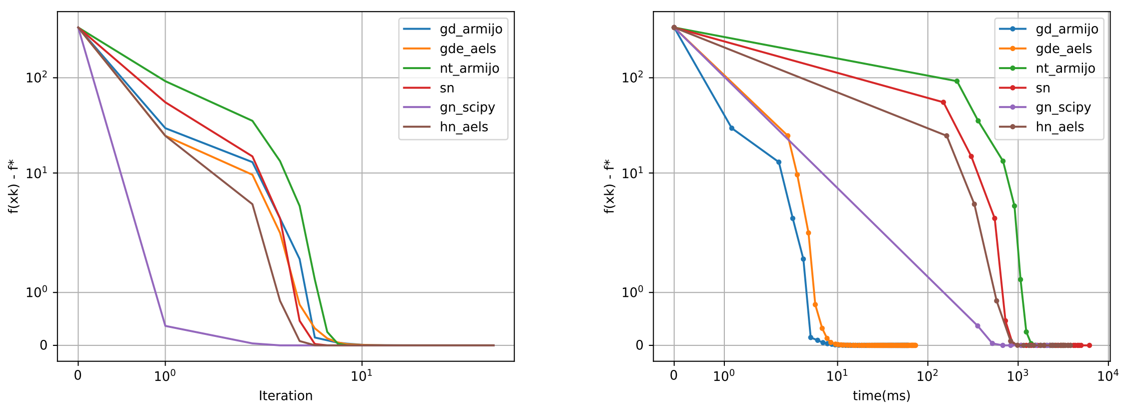

Figure 12: 2000 features, regularized

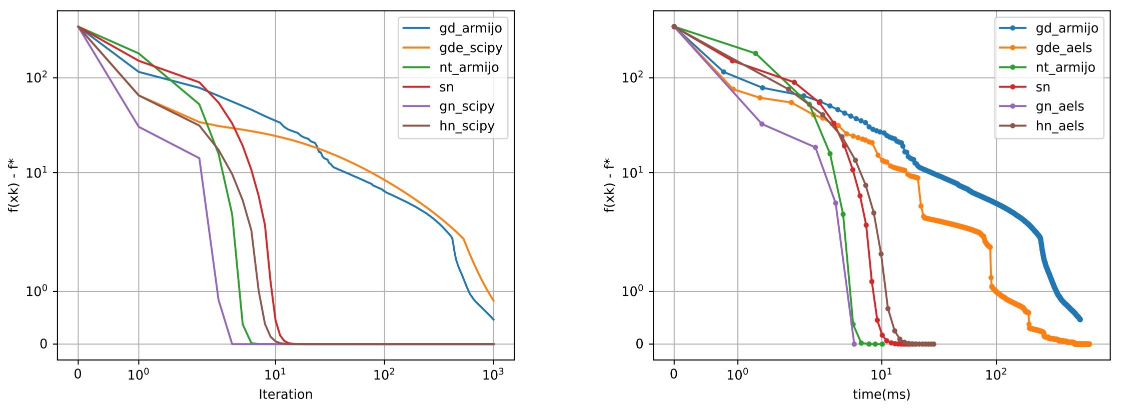

Figure 13: 20 features, unregularized

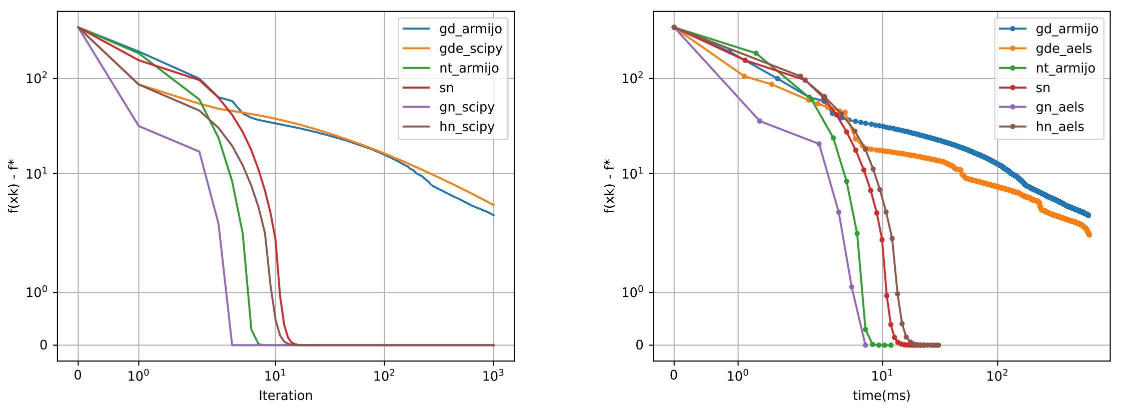

Figure 14: 20 features, 10 repeated, unregularized

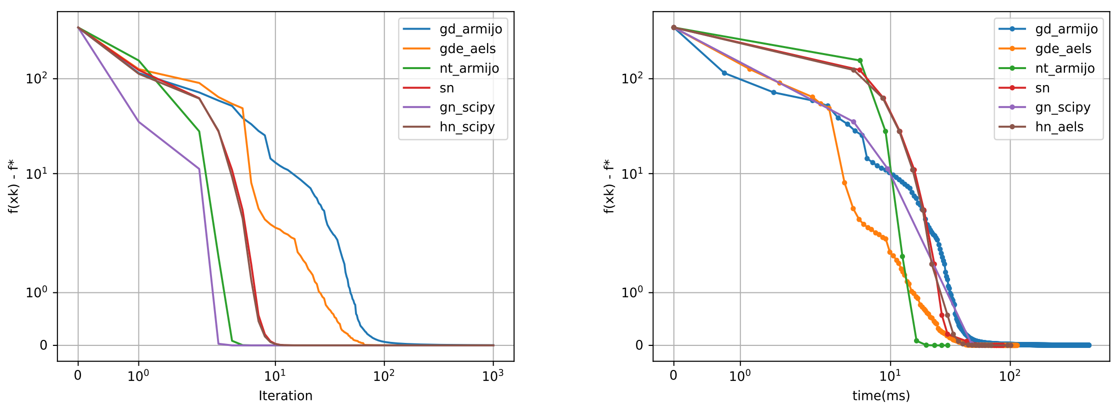

Figure 15: 200 features, unregularized

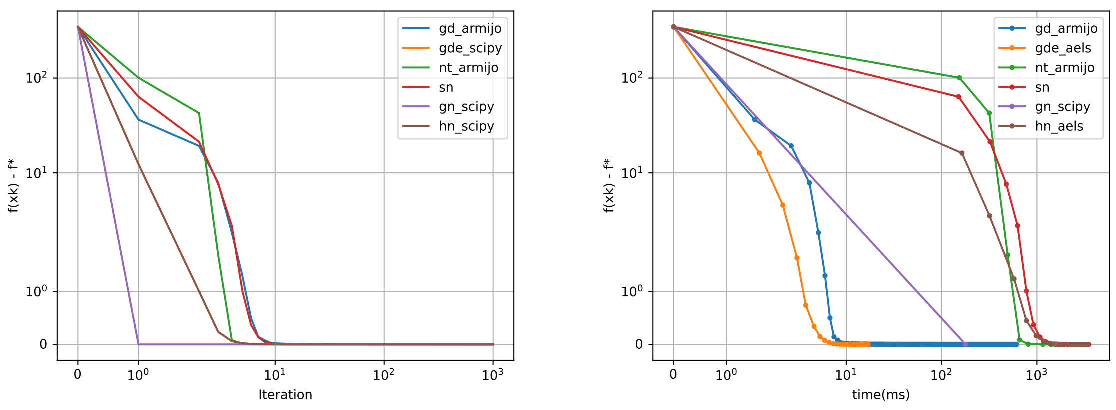

Figure 16: 2000 features, unregularized

5 Final observations and conclusions

As observed in Figure 1, when the problem is regularized and the number of features is small, all algorithms tend to converge to the solution at approximately the same time. However, gradient-based techniques require more iterations to converge to a solution with the same precision as second-order techniques. Specifically, gradient descent and gradient descent with exact line search require 83 and 63 iterations, respectively, while all Newton-based methods converge in less than 20 iterations.

When 10 of the features are repeated(Figure 2), the Greedy Newton method tends to outperform other methods both in terms of the number of iterations and runtime. Similar to the first case, the same difference in the needed iterations to reach the solution is observed.

In the regularized case with 200 features (Figure 3), all Newton-based methods still converge in less than 20 iterations, while the gradient-based methods need significantly more iterations (around 150 for both methods). This increase in the number of features leads to longer runtimes for Newton-based methods, mainly because the computation of the Hessian matrix begins to dominate the cost of the algorithm execution. However, even in the case with 2000 features (Figure 4), the Greedy Newton method outperforms all other Newton-based methods in terms of runtime and outperforms all tested algorithms when comparing the number of iterations needed to converge.

When testing the methods on the unregularized instances, it was observed that in all cases where the number of features is set to be equal to 20 (Figure 5 and 6), the gradient-based methods fail to converge to a solution within the maximum number of iterations (1000). In these cases, the Greedy Newton algorithm still outperforms all other solvers in terms of both runtime and the number of iterations needed to converge. When the number of features of the problem increases to 200 (Figure 7), the gradient-based solvers now reach convergence before the maximum limit of 1000 iterations, but the Greedy Newton method still outperforms all other solvers in all aspects (convergence is obtained in 4 iterations, 3 if using AELS). In the case with 2000 features (Figure 8), the gradient-based solvers have better runtime performance than the Greedy Newton method, but they need hundreds of iterations to converge, while the Greedy Newton method achieves convergence in only 1 iteration. Up to this point, the Greedy Newton method outperforms all other Newton-based methods in terms of both runtime and the number of iterations.

When comparing the performance of the different line search techniques employed, it can be seen that there is not a clear winner. This is likely caused by differences between the tested methods, but also probably by differences between the specific instances of problems that were solved. For instance, comparing the case of a regularized problem with 20 features and the case with the same number of features but where 10 of them are repeated(Figure 9 and 10), observing the results reported for the Greedy Newton method, it can be noted that in the first case, the technique of exact line search that makes the algorithm more efficient in terms of execution time is 'minimize_scalar', whereas in the second case, it turns out to be AELS. In general, we can see that the gradient method with exact line search tends to perform better when AELS is employed as the exact line search routine, while, excluding the 20 feature cases, the Greedy Newton method tends to perform better with the SciPy routine.

As a final note, it should be considered that the routine implementing AELS was implemented from scratch, and therefore, it likely requires further adjustments to optimize its performance. The exact line search routine of SciPy is highly optimized; hence, the results obtained in this comparative phase may not be entirely reliable. In conclusion, we can affirm that the same results as those presented in [1] have been observed.

References

[1] Betty Shea and Mark Schmidt. Greedy newton: Newton's method with exact line search. 2024.

[2] Hung Nghiep Tran and Atsuhiro Takasu. Analyzing knowledge graph embedding methods from a multi-embedding interaction perspective, 2023.

[3] Sara Fridovich-Keil and Benjamin Recht. Approximately exact line search, 2022.