Università degli Studi di Padova
Master degrees: Computer engineering and Data Science
Course: Optimization for Data Science
Academic year: 2023/2024

Frank Wolfe solution for LASSO problem

Agostini Flavio, flavio.agostini.1@studenti.unipd.it

1 Introduction

Originally introduced in 1956 by Frank and Wolfe, Frank-Wolfe methods (also known as *conditional gradient methods*) are a class of first-order algorithms designed for the minimization of convex quadratic objectives over polytopes. The core idea is simple: at each iteration, the algorithm generates a new iterate by moving towards the minimizer of a linearized objective. The original version of the method achieves a sublinear convergence rate of $O\left(\frac{1}{k}\right)$. However, linear convergence can be achieved under certain conditions, such as when there is a lower bound on the gradient norm or when the minimum of the objective function lies in the interior of the feasible set. The attractive properties of this algorithm have sparked renewed interest in recent years, leading to its application in machine learning and data science. Among these applications, we will focus on the LASSO problem, which can be effectively handled by Frank-Wolfe methods due to their sparsity-inducing properties. Variants of the original algorithm have been proposed over the years, offering improved performance and addressing some of the method's weaknesses. In this work, we will examine the away-step variant, which addresses one of the main issues of the original method by improving the convergence rate when the solution lies on the boundary of the feasible set.
The objective of this document is to present an overview of the literature on Frank-Wolfe algorithms, particularly their application to LASSO problems, document the code developed to implement two variants of the Frank-Wolfe algorithm, and illustrate and discuss empirical results obtained on real-world datasets.

2 Literature overview

2.1 Problem outline

We will address general constrained optimization problems in the form:

$$\min_{x \in C} f(x)$$

Where C is a convex and compact (bounded and closed) subset of $\mathbb{R}^n$ and the objective function $f(x)$ is convex and continuosly differentiable, with Lipschitz continuous gradient with constant L > 0. C in the convex hull of a finite set A of which elements we will call atoms.

2.2 Classical Frank-Wolfe method

We present the classical Frank-Wolfe method. As described in Algorithm 1, the method generates a sequence of iterates $x^{(k)}$ by moving, at each iteration k, toward the vertex of the feasible set (referred to as "atoms") that minimizes the scalar product with the gradient at $x^{(k)}$. We select the appropriate atom s_k using a Linear Minimization Oracle (LMO) for the feasible set C, and then compute the descent direction d_k^{FW} by subtracting $x^{(k)}$ from s_k (line 3). If sufficient progress has been made, the algorithm terminates (line 4). Progress is measured in terms of the duality gap g_k, which is proven to be a reliable certificate of the algorithm's convergence [4]. Finally, the new iterate is generated by adding the descent direction computed in line 3 to the previous iterate, scaled by a carefully chosen step size factor $\alpha_k \in (0, 1]$.
An interesting property that emerges from the structure of this algorithm is the sparsity of its iterates: at each iteration k, the iterate can be represented as a convex combination of at most $k + 1$ atoms, $S^{(k)} \subseteq A$. As we will see, this set, called the "active set", is particularly important for the away-step variant of the algorithm.

2.3 Zig-zagging phenomenon and Away-step Frank-Wolfe variant

When the optimal solution lies on the boundary of the feasible set C, the convergence rate of the classical algorithm becomes sublinear. This phenomenon is known in the literature as the "zig-zagging phenomenon" [5] and occurs because the iterates begin bouncing back and forth between the vertices that define the face on which the solution lies. To address this issue, Wolfe designed a variant of the classical algorithm called the *away-steps variant.* The name comes from the fact that, at each iteration, the iterate is now given the option to move *away* from an active atom in $S^{(k)}$.

Algorithm 1 Frank-Wolfe

```
1: Let $x^{(0)} \in C$
2: for k=0 to K do
3:     Set $s_k := \mathrm{LMO}_C(\nabla f(x^{(k)}))$ and $d_k^{FW} := s_k - x^{(k)}$
4:     If $g_k^{FW} := \langle -\nabla f(x^{(k)}), d_k^{FW} \rangle \leq \epsilon$  Return  $x^{(k)}$
5:     Set $x^{(k+1)} := x^{(k)} + \alpha_k d_k^{FW}$ with $\alpha_k \in (0,1]$ suitably chosen stepsize
6: End for
```

We will now highlight the differences between the two algorithms: In line 4, we calculate the away direction by searching the active set $S^{(k)}$ for the vertex v_k that maximizes the dot product with the current gradient and then subtract this from $x^{(k)}$. In lines 6 to 10, we evaluate which direction, Frank-Wolfe (FW) or Away (A), offers the best potential for descent. Line 9 ensures that, in the case of an away direction, the step size is small enough to keep the iterate within C. This step makes it necessary for the algorithm to maintain the active set S, a requirement that was not present in the classic Frank-Wolfe method. In line 11, we update the active set, making sure to adjust the weight assigned to each atom. This line also implements the *drop step*, which occurs when $\alpha = \alpha_{\max}$, resulting in the removal of atom v_k from the active set.

Algorithm 2 Away-steps Frank-Wolfe

```
1: Let $x^{(0)} \in C$, and $S^{(0)} := x^{(0)}$
2: for k=0 to K do
3:     Set $s_k := \mathrm{LMO}_C(\nabla f(x^{(k)}))$ and $d_k^{FW} := s_k - x^{(k)}$
4:     Set $v_k \in \arg\max_{v \in S(k)} \langle \nabla f(x^{(k)}), v \rangle$ and $d_k^A := x^{(k)} - v_t$
5:     If $g_k^{FW} := \langle -\nabla f(x^{(k)}), d_k^{FW} \rangle \leq \epsilon$  Return  $x^{(k)}$
6:     If $g_k^{FW} \geq \langle -\nabla f(x^{(k)}), d_k^A \rangle$:
7:         $d_k := d_k^{FW}$ and $\alpha_{\max}$
8:     else
9:         $d_k := d_k^A$ and $\alpha_{\max} := \alpha_{\mathrm{vt}}/(1 - \alpha_{\mathrm{vt}})$
10:    Set $x^{(k+1)} := x^{(k)} + \alpha_k d_k$ with $\alpha_k \in (0,1]$ suitably chosen stepsize
11:    Update $S^{(t+1)}$
12: End for
```

2.4 The FW gap

The FW gap, also indicated as Duality Gap, is a key parameter often used as a measure of convergence for Frank-Wolfe methods. It is defines as:

$$g(x) := \max_{s \in C} \left\langle -\nabla f(x^{(k)}), s - x \right\rangle$$

A crucial property of the FW gap is that it serves as a certificate for the current approximation quality, i.e., $g(x) \geq f(x) - f(x*)$. In other words, it provides an upper bound on the quality of the approximation. This is particularly useful because, while $f(x*)$ is unknown in real-world cases, $g(x)$ is relatively easy to compute and is naturally obtained during the FW algorithm. These characteristics make it an ideal candidate for use as a stopping criterion in the algorithm. [4] shows a sublinear convergence rate for this quantity, which is valid for all FW variants, confirming that it can be used in practice as a surrogate for the primal error.

2.5 Convergence rates

For classic Frank-Wolfe a sublinear convergence rate for the primal error is demonstrated by [4]. As previously stated, this rate under some strong assumptions can be shown to be linear. These assumptions are: strong convexity and a lower bound on the gradient norm, or strong convexity with objective function having its minimum in the relative interior of the feasible set. The various variants of the algorithm aim to improve this rate to linear in the worst-case scenarios where the minimum lies on the boundary, or close to the boundary, of the feasible set. [5] show a linear convergence rate for Lipschitz continuous and strongly convex functions over C regulated by a constant ρ:

$$h_{\mathrm{t}+1} \leq (1-\rho)h_t, \qquad \rho := \frac{\mu}{4L}(\frac{\delta}{M})^2$$

Where L is the Lipschitz constant, μ is the strong convexity constant, M is the diameter of the set C and δ is the pyramidal width of the set A. We can see that this constant fundamentally depends on the conditioning number of the function, given by $\frac{\mu}{L}$, and by the ratio between the pyramidal width and the diameter of the feasible set, that are parameters that depend on the geometry of the set and can be interpreted as conditioning value for the set C.

2.6 LASSO problem

LASSO is a popular tool to obtain sparse solutions to linear regression problems [1]. Given a dataset with feature matrix A and target vector b, the objective is to find a sparse linear model that describes the data. Formally the problem can be formulated as:

$$\min_{x\in\mathbb{R}} f(x) := ||Ax-b||_2^2$$

$$s.t. ||x||_1 \leq \tau$$

Here, the parameter τ its an hyperparameter that regulates the amount of shrinkage applied by the model. A lower τ corresponds to higher shrinkage, i.e. a more sparse solution is enforced. The feasible set C will be a convex set with the canonical bases e_i as vertexes. The LMO will thus take the form:

$$\mathrm{LMO}_C(\nabla f(x^{(k)})) = \mathrm{sign}(-\nabla_{\mathrm{ik}} f(x_k)) \cdot \tau e_{\mathrm{ik}}$$

Where i_k is the index of the element in the gradient $\nabla_i f(x_k)$ with highest value.

3 Implementation notes

3.1 Notebook structure

The code is available as a Jupyter Notebook but the algorithmic part has been implemented as a Class with the idea of a possible future integration in a package.
The Notebook is structured in this way: in the first part we have the definition of the Frank_Wolfe() class, we then have the generation of a synthetic dataset used for testing as well as to demonstrate theoretical results, and finally we have the application of our methods in 3 real-world datasets.

3.2 Frank_Wolfe() Class

As mentioned above the algorithms have been implemented via a python custom class. The idea was to build reusable, maintainable and well structured code as well as to provide the user with a single interface offering access to all the algorithms implemented with the ability to easily tweak the various parameters. This approach allows also to open the possibility to implement other FW variants in the future.

3.2.1 Constructor

The constructor method takes as arguments the parameters indicated in Table 1 and uses them to create an instance of the model.

Argument	Explanation
`tau`	This is the τ hyperparameter related to the LASSO problem. It regulates the shrinkage applied by the model
`iter`	Maximum number of iterations
`verbose`	Whether or not to display iteration-by-iteration information during fitting
`plots`	Whether or not to display plots after fitting
`delta`	δ parameter of armijo line search
`gamma`	γ parameter of armijo line search
`max_alpha`	α parameter of armijo line search
`min_alpha`	Minimum stepsize for armijo line search
`alpha_search`	Type of stepsize. Choose between "armijo" and "diminishing"
`alg`	Type of algorithm. Choose between "original" and "away-steps"
`e`	ϵ parameter for FW algorithm. It's a stopping criterion based on duality gap

Table 1: Arguments of Frank_wolfe class constructor

3.2.2 fit() method

The fit() method is used, after instantiating an object of the class, to fit the model to a dataset. It takes as input a feature matrix A and a vector with the targets of the regression b. In this method different parameters are initialized:

- m and n are, respectively, number of training examples and number of features of the problem
- $x^{(o)}$ is initialized as a n x 1 zeroes vector
- n x n matrix "E", containing the canonical bases, is created
- Vectors "objf" and "g", containing per iteration value of objective function and duality gap, are created

Based on the value of "alg" parameter one of the supported algorithms is then launched calling "FW()" function, for the classical method, or "AFW()" for the aways-steps variation. A key performance index, CPU time of execution, is stored in a variable called "time". Finally plots with the results of the fitting procedure are shown.

3.2.3 FW() and AWF() methods

These two methods implement Algortim 1 and Algorithm 2. Both contain a cycle with at most "iter" iterations, that breaks whenever a duality gap lesser than ϵ is achieved. All numerical operations are done leveraging efficient numpy matrix operations. Line search is implemented with armijo rule. This allows the algorithm to get a certified good improvement in objective function at each iteration but without the computational cost of conducting an exact line search. The FW() method also allows to use the diminishing stepsize $\frac{2}{k+2}$ proposed in the theoretical demonstrations found in [4]. AWF() implements the management of the active set with a list of tuples S, where each element is composed by the atom s_k and its weight α_{sk}. This set is updated at each iteration to account for drop steps and changes in the weights the atoms with the following rationale, as in [5]:
For FW step: if $\alpha_k = 1$, $S^{(k+1)} = s_k$; else $S^{(k+1)} = S^{(k)} \cup s_k$ and we update the weights: $(\alpha_{\text{sk}}^{(k+1)} = (1 - \alpha_k)\alpha_{\text{sk}}^{(k)} + \alpha_k$; $\alpha_v^{(k+1)} = (1-\alpha_k)\alpha_v^{(k)})$ for $v \in S^{(k)} \backslash \{s_k\}$
For an away step: $S^{(k+1)} = S^{(k)} \backslash \{v_k\}$ if $\alpha_k = \alpha_{\max}$. This is considered a *drop step.* Else $S^{(k+1)} = S^{(k)}$. We also adjust the weights: $\alpha_{\text{vk}}^{(k+1)} = (1-\alpha_k)\alpha_{\text{vk}}^{(k)} - \alpha_k$ and $\alpha_{\text{v}}^{(k+1)} = (1+\alpha_k)\alpha_{\text{v}}^{(k)} - \alpha_k$ for $v \in S^{(k)} \backslash \{s_k\}$

3.2.4 Auxiliary functions

FW() and AWF() use some additional functions to increase the readability of the code. All functions rely on numpy functions to carry out essential calculations efficiently:

- f(x): Computes the value of the function at x as in: $f(x) := \frac{1}{2}||Ax - b||_2^2$
  We opted for the halved version of the objective function for mathematical convenience.
- compute_gradient(): computes the gradient $\nabla f(x) = A^T(Ax - b)$
- LMO(): first finds $i_k = \arg\max_i |\nabla_i f(x_k)|$ and then computes the LMO as in 2.6. Here we use the matrix of canonical bases "E" previously constructed
- predict(A): computes predictions pred = Ax

Finally we have plot_results() and compare() that generate prots and show the numerical results of the fitting procedure.

3.3 Synthetic dataset

In this section of the notebook we create a toy dataset with a 1000 x 25 feature matrix A, and a 25 x 1 target vector b obtained by $b = Ax^*$, with x^* being a sparse random normal 25 x 1 vector with 10% additive noise. We then use this dataset to test our implementation and to obtain some important insights, without having to worry about noise and other problems that could be present in the real-world datasets.

3.4 Real-World datasets

In this section Real-World datasets are imported and some basic data cleaning and pre-processing is applied as some datasets needed outlier handling, na removal and label encoding. These operations have been implemented using pandas python library. The datasets are then standardized in order to make comparison of the results possible, and to improve numerical stability. The goal of this section is to compare the results of the two algorithm variants on real-world problems as well as gathering performance metrics.
The results are obtained by applying our custom implementation of FW methods to three datasets. All three are regression problems suitably chosen to be solved with a LASSO approach. The datasets are chosen to have a diverse number of observations and features, providing a solid proving ground for our implementations. The first one [3] is composed of 4898 examples of white wine samples, with 11 features involving biochemical and physical measurements of various properties of the wine and with objective inferring its quality. The second dataset [2] is composed of 649 observations of 29 metrics that could be correlated with a student's performance, taken from students attending a portuguese class. The target is the final grade at the end of the year. The third [6] involves using 20433 observations of 8 features related to housing in the California bay area to estimate the median value of the properties.

4 Results

4.1 Synthetic dataset

We first present an initial set of results obtained in the synthetic dataset. Looking at the first picture in Figure 1 we can see that the convergence of away-steps Frank Wolfe is much faster if compared with the original Frank Wolfe for this particular set of parameters, reaching the desired duality gap $\epsilon = 0.1$ in 205 iterations against the 484 needed by the original algorithm. In the second picture we can see how both algorithms obtain a steep reduction in cost function right in the first 50 to 100 iterations, even if around the 30th away-steps demonstrates a better performance. The third plot shows how sparsity (here calculated as the fraction of zero elements in x over the total) develops during fitting. In this example a sparsity of 0 is reached by both algorithms, indicating that there are not zero elements in the solution vector. This fact is caused by the parameter τ, that in this case is permissive enough to not enforce strict sparsity of the solution. Looking at Figure 2 we can observe how the solution vector still looks sparse overall, with most entries being very close to zero. From the sparsity graph we can see that away-steps moves more aggressively towards the solution, and finds the set of atoms containing the solution before the original method does.

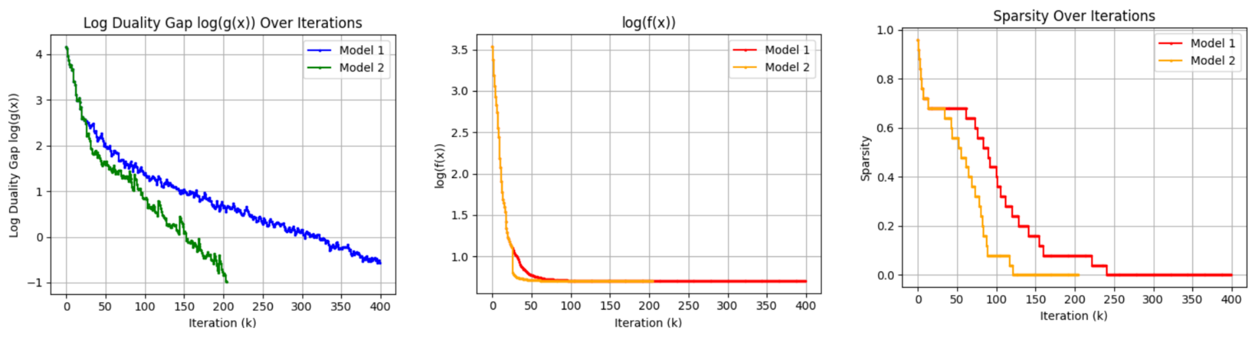

Figure 1: Plots for syhtetic dataset. $\tau = 10$ and $\epsilon = 0.1$

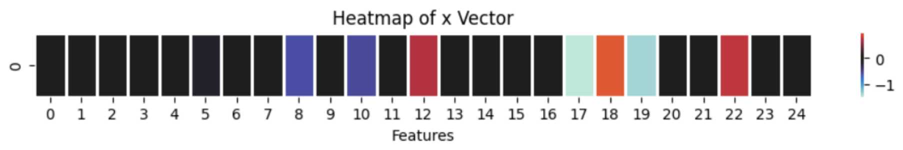

Figure 2: Heatmap of solution vector x

Table 2 presents the results of an experiment designed to study the behaviour of the two methods with different values of τ. We can think of this parameter as an indication of how strictly sparsity is enforced, with lower values representing a stronger enforcement. In practice it is a scaling factor applied to the canonical bases that compose the l-1 norm ball, so when the value is higher our feasible set gets wider. If we compare the results obtained with classic FW and AFW we can see that the level of sparsity achieved is the same in both methods for all τ values. This seems to indicate that both methods are able to find the correct set of atoms that define the solution for any given τ, even if, as seen in Figure 1, AFW is much faster at reaching this conclusion. Looking at the iterations it becomes clear that in most cases the AFW variant converges much faster to the solution, confirming the theoretical results on linear convergence. After a critical value of τ, that seems to be between 5 and 10, the performance of classic FW improves drastically, becoming virtually the same as AFW. I hypothesize that this is because the constraint is so weak that the actual minimum of the objective function resides inside the feasible set, and is therefore readily found by both algorithms. We can see that, while in most cases AFW has a way lower computational time, when the number of iterations is similar this advantage gets lost, probably due to the added cost of managing the active set. From this experiment we can confirm that the advantages of AFW variant come into play when the solution lies in the boundary of the feasible set (i.e. when the solution is sparse), in which case the variant is able to solve the problem using less computational resources.

τ	Iterations	CPU time	Sparsity
0.5	5	0.00708	0.92
0.75	6	0.00403	0.92
1	119	0.08254	0.88
1.5	108	0.07045	0.84
2	54	0.03280	0.8
2.5	*	1.16591	0.76
3	*	1.23666	0.72
5	*	1.18941	0.72
10	484	0.37046	0
50	116	0.10140	0
100	137	0.13492	0

τ	Iterations	CPU time	Sparsity
0.5	5	0.00738	0.92
0.75	6	0.00591	0.92
1	10	0.01040	0.88
1.5	11	0.00736	0.84
2	19	0.01697	0.8
2.5	24	0.01483	0.76
3	26	0.01451	0.72
5	40	0.02443	0.72
10	205	0.18204	0
50	119	0.10285	0
100	131	0.12518	0

Table 2: In the left classic FW, in the right AFW. (*) indicates maximum number of iterations (2000)

4.2 Wine dataset

In figure 3 we can see that for $\tau = 1$, AFW performs much better than classic FW on this dataset. The speed of convergence in duality gap is, after the first 10-15 iterations, markedly in AFWs favour. Here we can see how the away variant is able to push incredibly fast toward the solution, while the classic variant plateaus almost immediately, probably due to the zig-zagging behaviour we've seen in the theoretical analysis. Even if the duality gap shows this behaviour, in objective function we see that a value comparable to the one achieved with AFW is reached not much later, indicating that the problem is still solved efficiently enough. As seen with the synthetic dataset we confirm that AFW is able to find the optimal set of atoms that describe the solution faster. In figure 4 we can see the heatmap of the solution, again appreciating how only few elements contribute most of its weight, even if just two of the elements are precisely equal to zero.

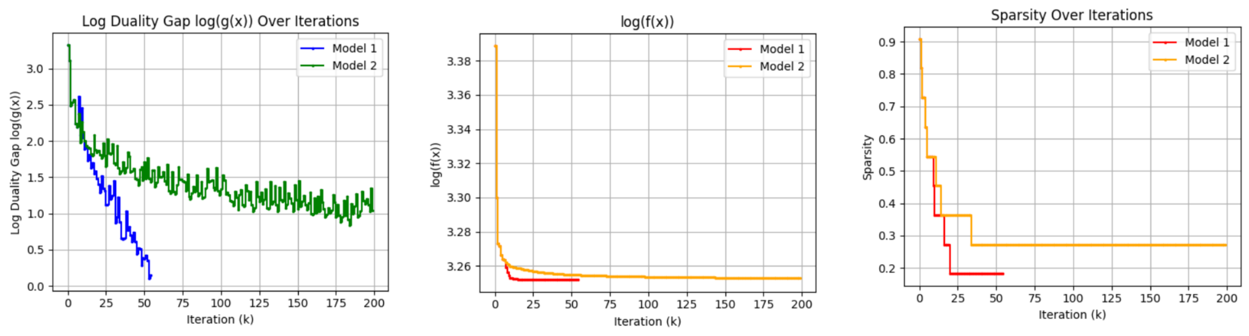

Figure 3: Plots for wine dataset. $\tau = 1$ and $\epsilon = 1$

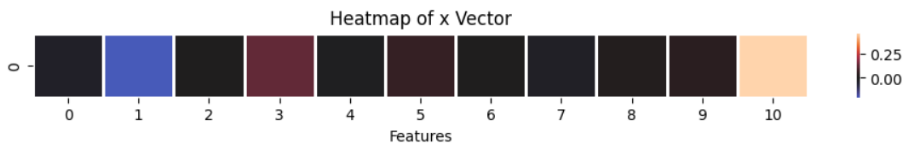

Figure 4: Heatmap of solution vector x

4.3 Portuguese dataset

Portuguese dataset, as in figure 5, shows marked similarities with respect to the wine dataset when using the same parameters. This shows that the methods are consistent even with very different data. Here AFW finds a solution in 40 iterations, against 331 needed by classic FW with an improvement in CPU time going from 0.17790 seconds to 0.02758.

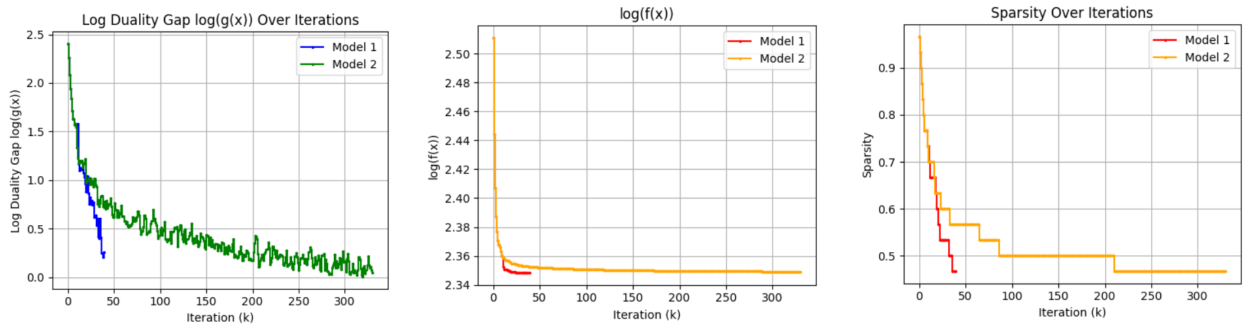

Figure 5: Plots for Portuguese dataset. $\tau = 1$ and $\epsilon = 1$

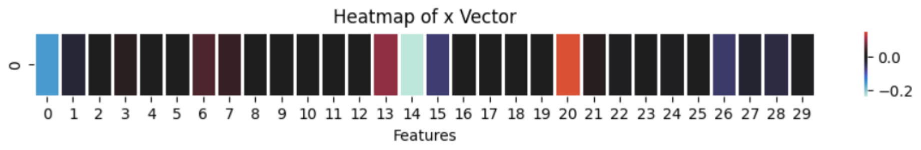

Figure 6: Heatmap of solution vector x

4.4 House prices dataset

In Figure 7 we can see that, once again, with the same parameters, the results are comparable to the ones obtained with the other datasets. For this dataset we also present results using a different τ of 5. This tau is weak enough to not enforce sparsity anymore. As we can see in Figure 8, AFW still converges faster when it comes to duality gap, but in objective function the behaviour of the two methods seems to be the same, indicating that in this practical application AFW does not have a particular advantage over classic FW when it comes to optimizing the function. The biggest advantage is again CPU time, that happens because we use duality gap as stopping criterion for the algorithms.

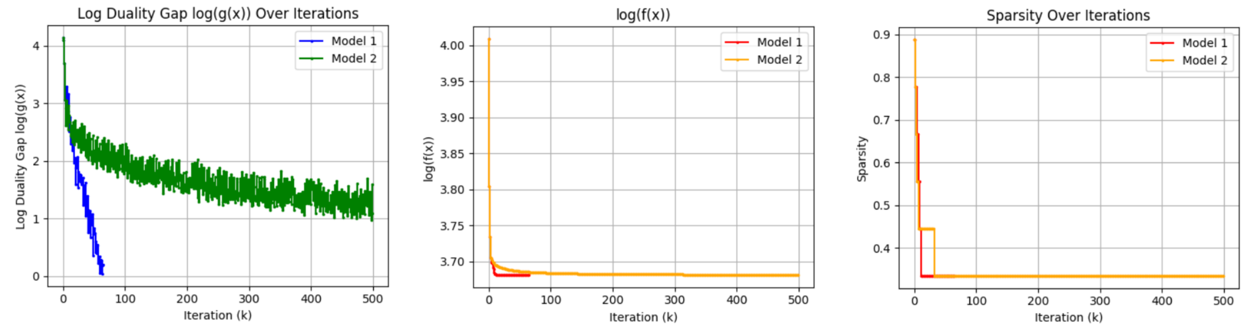

Figure 7: Plots for Housing prices dataset. $\tau = 1$ and $\epsilon = 1$

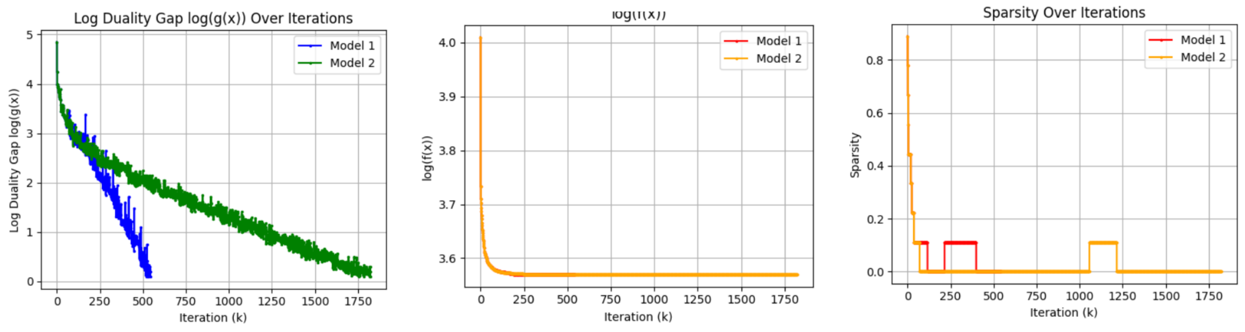

Figure 8: Plots for Housing prices dataset. $\tau = 5$ and $\epsilon = 1$

5 Conclusion

In this project two variants of the Frank Wolfe algorithm, the classical version and the away-steps variation, were successfully implemented in a custom python class, open to further implementations and enhancements. Using the framework of this class and four datasets (three real-world datasets and a synthetic one) theoretical results regarding convergence and sparsity were empirically validated, demonstrating the robustness of these findings. The away-steps variant exhibits a faster convergence when the parameter τ was restrictive enough to enforce a sparse solution. This happens because AFW is able, thanks to its ability to follow away directions, to move more aggressively toward the boundary of the feasible set, where the solution lies if sparsity is enforced. When sparsity is not strongly enforced (i.e. τ is sufficiently large), the performances of the two algorithms gradually, as τ increases, become comparable, with classic FW sometimes being slightly cheaper in terms of computational demands if compared to AFW. This is because AFW requires the active management of the active set, while classic FW does not.
From these experiments it can be concluded that in practical applications a correct choice of τ is crucial, as it directly determines the amount of sparsity obtained in the solution, the speed of convergence, and thus, also computational cost.
In conclusion, the primary objective of the project, which was to provide an effective solution for LASSO problems using Frank-Wolfe methods, has been achieved, along with gaining valuable insights on the behaviour of the algorithms in real-world applications.

References

[1] Immanuel M Bomze, Francesco Rinaldi, and Damiano Zeffiro. "Frank–Wolfe and friends: a journey into projection-free first-order optimization methods". In: *4OR* 19 (2021), pp. 313–345.

[2] Paulo Cortez. *Student Performance*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5TG7T. 2008.

[3] Paulo Cortez. *Wine Quality*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C56S3T. 2009.

[4] Martin Jaggi. "Revisiting Frank-Wolfe: Projection-free sparse convex optimization". In: *International conference on machine learning*. PMLR. 2013, pp. 427–435.

[5] Simon Lacoste-Julien and Martin Jaggi. "On the global linear convergence of Frank-Wolfe optimization variants". In: *Advances in neural information processing systems* 28 (2015).

[6] Cam Nugent. *Housing Prices*. Kaggle. 2017.