

**MEB and Anomaly Detection using variants of Frank-Wolfe**

Optimization for Data Science

Project report

Authors:

Marija Cveevska

Dejan Dichoski

Suleyman Erim

Supervisor:

Francesco Rinaldi

September 22, 2023

# Introduction

*this is just copy-paste – NEEDS to be rewritten after finishing other sections*

Minimum enclosing balls (MEBs) provide a very versatile data representation for a wide range of learning and analysis tasks. Use cases include (but are not limited to) accelerated training of support vector machines or other classifiers, symbolic learning, identification of landmarks or prototypes, analysis of data streams, and anomaly detection. (**TODO**: ADD HERE THAT IN THIS PROJECT WE FOCUS ON THE LAST PART). MEBs are also at the heart of one class support vector machines, support vector clustering, or support vector data description. While there exists a whole spectrum of algorithms for estimating the MEB of a given data set, a particularly general approach is to formalize the problem in terms of constrained quadratic optimization and we, too, will adhere to this strategy.

Minimum enclosing balls have numerous important applications in clustering, nearest neighbor search, data classification, support vector machines, machine learning, facility location, collision detection, computer graphics, and military operations. In particular, many of these applications give rise to large-scale instances of the MEB problem, and a reasonably small accuracy suffices for such applications.

While the primal problem of estimating an MEB is rather unwieldy, we will see that the corresponding dual problem can be solved rather easily. Indeed, the dual MEB problem is of a form that allows for using simple solvers such as the basic Frank-Wolfe algorithm and its variations. We will discuss this in detail in this project. (**TODO**: change last sentence)

Notation

We use bold letters to denote vectors and matrices, while non-bold letters represent constants. For a vector , denotes its i-th component at iteration t. Inequality statements involving vectors apply to each of their components. Functions and operators are represented using uppercase letters. Uppercase script letters are employed to denote various objects, including sets and balls.

I think the above notation is enough. I started writing down bellow each element meaning, but it would be too much to cover everything, and we will probably forget something.

bold – vector/matrix

– set of points ( – number of points, – dimension of each point)

( - i-th point of the set of points (with dimension )

– ball with center (with dimension = n) and radius (constant)

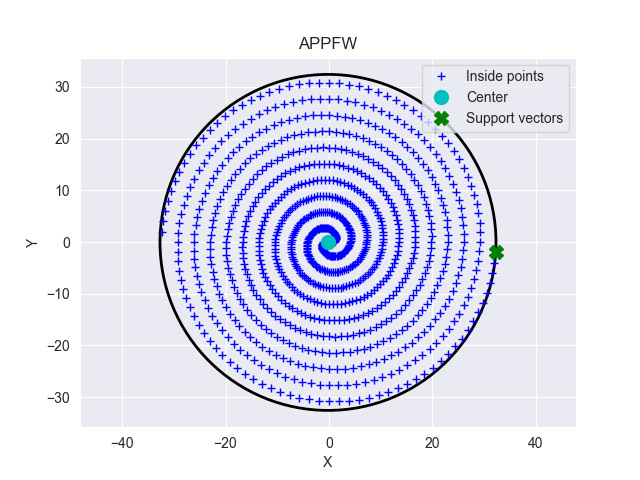
is a vector of Lagrange multipliers

– Lagrangian dual function

**The TODOs in brackets will be filled at the end, after adding equation numbers.**

# Problem definition

## The primal MEB problem

In the minimum enclosing ball problem, given a finite set of vectors , the objective is to determine the smallest n-dimensional ball that contains all the points in :

Since Euclidean balls are uniquely defined in terms of their center and radius , the basic problem is therefore to determine the optimal choices and for these two parameters. In this report, we approach this problem from the point of view of constrained convex optimization. Indeed, noting the equivalences:

Figure 1: Data points forming a Fermat’s spiral, and their minimum enclosing ball (which in 2D is a circle).

we can immediately formalize the problem of finding and as an inequality constrained quadratic minimization problem, namely

Which we can rewrite as:

In simpler terms, this means that we are simultaneously searching for an appropriate center and radius. The (squared) radius should be as small as possible but not smaller. In other words, the (squared) distance between each of the given data points and the center of the ball must be less than or equal to the (squared) radius of the ball.

While there are algorithms that can solve (TODO) directly, a more general approach consists of working with the corresponding dual problem which we will derive next.

By setting , we obtain:

Our goal is to show that the above primal (minimization) problem has the following dual (maximization) problem:

where is a vector of Lagrange multipliers.

## Deriving the dual problem

To begin with, we write down the Lagrangian of the problem in (TODO). Letting denote the Lagrange multiplier for the i-th inequality in (TODO), we have:

Notice that, when expanding the expression in (TODO) to the one in (TODO), we introduced the shorthand just to reduce notational clutter.

In fact, we can write our Lagrangian even more compactly. To do this, we first note that:

where is a data matrix whose columns correspond to the data points in . Hence, the Lagrangian in (7) can also be written as:

Given this Lagrangian, we will now evaluate the Karush-Kuhn-Tucker conditions. Focusing on the KKT 1 conditions (stationarity), we have:

When we substitute the result from (TODO) into (TODO), the latter simplifies to:

Furthermore, by plugging the expressions from (TODO) and (TODO) back into the Lagrangian in (TODO), we obtain the Lagrangian dual, since:

is a function that depends solely on the Lagrange multipliers .

Now, if we further note that the KKT 3 conditions (dual feasibility) require , all our (intermediate) results and considerations so far indicate that the dual of (TODO) amounts to

Finally, it's worth noting that is concave in , making is convex. Therefore, the solution to the maximization problem in (TODO) can also be found by considering an equivalent minimization problem, namely:

## From Lagrange multipliers to MEB parameters

If we could solve the dual problem in (TODO), we would obtain an optimal Lagrange multiplier vector, denoted as . Assuming we could determine , it would immediately enable us to compute the center of the sought-after MEB, as per equation (TODO):

If we could solve the dual problem in (TODO) we would obtain a vector of optimal Lagrange multipliers. Assuming we could determine , it would immediately enable allow us to compute the center of the sought-after MEB because, according to (TODO), we have:

But what about the corresponding radius ?

The radius of the minimum enclosing ball can be determined by considering the KKT 4 conditions (complementary slackness). These conditions dictate that at a solution, we must have:

Summing these equations over all i then yields

With additional algebraic manipulations based on our previous calculations, we arrive at:

In other words, we find that the radius of the minimum enclosing ball for the given data is:

# Frank-Wolfe for Minimum enclosing balls

Reiterating what we discussed above, we emphasize that the dual MEB problem in (TODO) seeks a minimizer of the convex function (TODO). Additionally, the non-negativity () and sum-to-one () constraints in (TODO) imply that any feasibly solution must reside within the standard simplex:

This insight allows us to express our problem in an even more compact manner, namely:

Written like this, the dual MEB problem is now clearly recognizable as an instance of a convex minimization problem over a compact convex set. This is significant because the Frank Wolfe algorithm provides a straightforward approach for solving this particular kind of problem. Next, we show how to specialize the general form of the Frank-Wolfe to the minimum enclosing ball problem in (TODO equation above).

The Frank-Wolfe algorithm is an iterative solver. First, it makes an initial feasible guess for the solution. The midpoint of is contained within and is therefore feasible. This means that we can use as a starting point. However, for simplicity in our implementation, we decided to use the first row of the basis matrix.

In each iteration, the algorithm determines that minimizes the inner product and applies a conditional gradient update , where the step size [0, 1] decreases over time. Updates will thus never leave the feasible set and the efficiency of the algorithm is due to its ability to transform a quadratic problem into a series of linear ones.

Calculating the step direction involves determining the gradient of the objective function at the current estimate. In our specific case, this gradient is given by:

Therefore, each Frank-Wolfe iteration has to solve:

Note that the expression on the right-hand side is linear in and needs to be minimized over , which is a compact convex set. The minimum of a linear function over a convex set is necessarily achieved at a vertex of said set. Since the vertices of correspond to the standard basis vectors of , we only need to determine which of these minimizes the inner product mentioned above. Hence, the problem in (TODO) simplifies to:

To make it explicit that the resulting will be a standard basis vector , we can also express it as:

We can solve this without computing inner products; we only need to identify the index corresponding to the smallest component of the gradient, which is:

Given these considerations, the general update step for the Frank-Wolfe algorithm thus specializes to:

In conclusion, solving the dual MEB problem can be fairly straightforward by applying the original Frank-Wolfe algorithm and its variations. In this project, we examined, implemented, and compared three different versions of the Frank-Wolfe algorithm. Next, we will introduce each of these algorithms and present our results by applying them to address the MEB problem in the context of anomaly detection.

## Away-steps Frank-Wolfe algorithm

The convergence rate of the original Frank Wolfe algorithm is known to be slow (sublinear) when the solution lies on the boundary. A simple improvement is to introduce the possibility of taking ‘away steps’ during optimization, an operation that importantly does not require a feasibility oracle. The away-steps variant of the Frank-Wolfe algorithm, which can also remove weight from ‘bad’ atoms in the current active set, was proposed by P. Wolfe. The precise method is outlined below in Algorithm 1.

When the optimal solution lies on the boundary of , the iterates of the classical FW algorithm start to zig-zag between the vertices defining the face containing the optimal solution. This results in a slow convergence rate (sublinear).

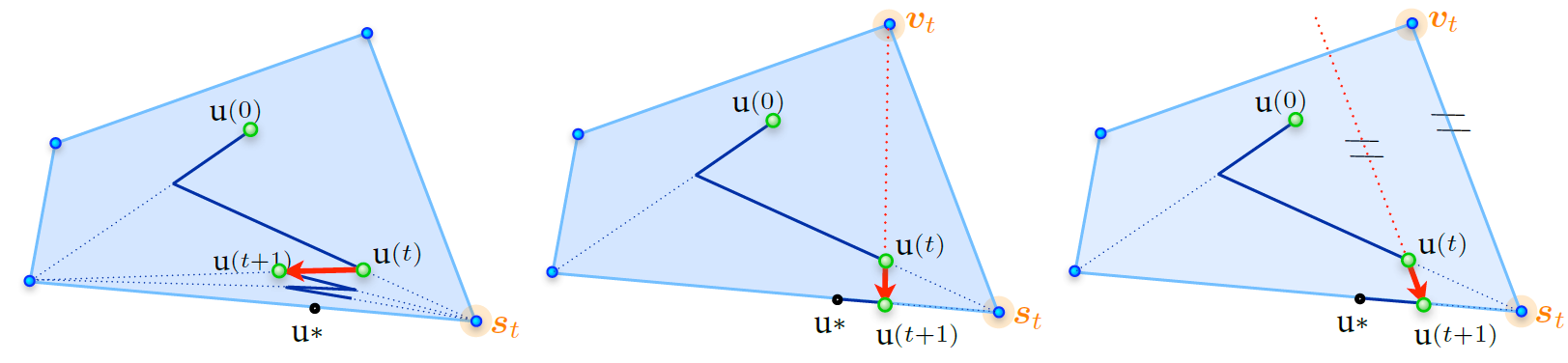


Figure 2: (left) The FW algorithm zig-zags when the solution lies on the boundary. (middle) Adding the possibility of an away step mitigates this problem. (right) As an alternative, a pairwise FW step.

To address the zig-zagging issue in the FW algorithm, Wolfe proposed adding the possibility of moving away from an active atom in (as shown in the middle of Figure 1). This simple modification is sufficient to achieve linear convergence for strongly convex functions. We describe the away-steps variant of Frank-Wolfe algorithm in Algorithm 1.

**Algorithm 1: Away-steps Frank-Wolfe algorithm**

1. ,  **//** *so that for* *and 0 otherwise*
2. **For** **do**
3. and // *the FW direction*
4. and // *the away direction*
5. **if** **then** **return**
6. **if** **then**
7. and // *choose the FW direction*
8. **else**
9. and // *choose away direction; maximum feasible step-size*
10. **end if**
11. Line-search:
12. **//** *and accordingly for the weights*
14. **end for**

The away direction, denoted as **,** is defined in line 4 by finding the atom in that maximizes the potential descent given, by . It's important to note that this search is performed over the (typically small) active set , making it fundamentally easier than the linear oracle . The maximum step-size ,defined in line 9, ensures that the new iterate remains within conv() ⊆ , which guarantees that the convex representation is maintained. Computing the true maximum feasible step-size would require the ability to know when we cross the boundary of along a specific line, which is generally possible for arbitrary . By using the conservative maximum step-size outlined in line 9, we eliminate the need for this more powerful oracle. This is why Algorithm 1 requires maintaining , unlike the standard FW algorithm. Finally, as in classical FW, the FW gap serves as an upper bound on the unknown suboptimality and can be used as a stopping criterion:

(by convexity)

If , then we refer to this step as a drop step, as it completely removes the atom from the currently active set of atoms, , (by settings its weight to zero). We cannot guarantee sufficient progress in this case, but the drop step decreases the active set size by one, and thus, they cannot occur too frequently (not more than half the time).

The weight updates for lines 12 and 13 take the following form: For a FW step, we have if ; otherwise . Additionally, we have and for . For an away step, we have if (a drop step); otherwise .Also, we have and for .

## Blended Pairwise Conditional Gradients (BPCG)

The Pairwise Conditional Gradients algorithm is a natural generalization of the Away-step Frank-Wolfe algorithm. Moreover, it uses the blending criterion introduced in (Braun et al., 2019a) to efficiently combine local PCG steps with global Frank-Wolfe steps. Unfortunately, PCG exhibits so-called swap steps that might not provide sufficient primal progress. The number of these suboptimal steps is bounded by a function of the dimension, and therefore, known guarantees do not generalize to the infinite-dimensional case.

A simpler modification of the PCG algorithm can be achieved by combining it with the blending criterion from the Blended Conditional Gradients Algorithm. This new algorithm is called the Blended Pairwise Conditional Gradients. BPCG eliminates the occurrence of swap steps. It is straightforward to implement, and does not require any internal gradient alignment procedures. The convergence rate of BPCG is basically that of PCG, if no drop steps would occur. As such, it improves the convergence rates of PCG. Additionally, numerical experiments show that BPCG’s solutions are much sparser than those of PCG.

The Blended Pairwise Conditional Gradients algorithm (BPCG) is presented in Algorithm 2. As mentioned earlier, our focus is on the general smooth convex case and the strongly convex case over polytopes.Top of Form

**Algorithm 2: Blended Pairwise Conditional Gradients (BPCG)**

1. ,
2. **for**  **do**:
3. // *away vertex*
4. // *local FW*
5. // *global FW*
6. **if** **then**


10. **if** **then**
11. // *descent step*
12. **else**
13. // *drop step*
14. **end if**
15. **else**

18. (or **if )** // *FW step*
19. **end if**
21. **end for**

In Algorithm 2, if the local pairwise gap is smaller than the Frank-Wolfe gap , a FW step (line 16-18) is taken. Otherwise, the weights of the active atoms in are locally optimized by the Pairwise Conditional Gradients (PCG). If the step size is larger than , the away vertex is removed from the active set , and we refer to this step as a drop step (line 13). Otherwise, it is called a descent step (line 11). Collectively, the descent step and drop step are referred to as pairwise steps.

Due to the structure of the BPCG algorithm, the sparsity of the solutions is expected since the new atoms are not added to until the local pairwise gap decreases sufficiently. Additionally, since the PCG is implemented locally, BPCG does not exhibit swap steps in which weight shifts from the away atom to the Frank-Wolfe atom.

Note that in Algorithm 2, we use a line search for simplicity in presentation. However, it can alternatively be executed with a simpler decaying strategy, such as .

Advantages of the BPCG algorithm compared to other CG (FW) variants:

* **Compared to the vanilla CG**: The BPCG algorithm ensures linear convergence for the strongly convex polyhedral case, while the vanilla CG does not. Although both algorithms guarantee a convergence for infinite-dimensional cases, we expect the BPCG algorithm to produce much sparser solutions due to its aggressive removal of unnecessary vertices from the convex combination and the addition of new vertices only when needed for primal progress.
* **Compared to the Away CG algorithm**: BPCG and Away CG have the same convergence guarantee for the general smooth convex case and strongly convex polyhedral case. However, BPCG empirically generates significantly sparser solutions.
* **Compared to the Pairwise CG algorithm**: BPCG’s convergence rate aligns more closely with that of the Away-steps Frank-Wolfe algorithm than with PCG. Moreover, since the iterations of BPCG include many local updates in which no new atoms are added, it is expected that the BPCG algorithm outputs sparser solutions than the PCG algorithm in terms of the support size of the supporting convex combination.

BPCG also extends to the infinite-dimensional setting, and its (empirical) sparsity is very high, making it a prime candidate for problems such as sparse signal recovery, matrix completion, and numerical integration.

## -approximation to MEB

Given a finite set of points , our goal is to compute an approximation to the minimum enclosing ball of , denoted by MEB(). Given a given ε > 0, we define a ball to be a -approximation to MEB() if:

A subset is said to be an -core set (or a core set) of if:

Small core sets play an important role in designing efficient algorithms for large-scale problems because they provide a compact representation of the input set . If a small -core set is available, then solving the problem on already yields a good approximation to MEB(). Since the center of MEB() lies within the convex hull of , Carathéodory’s theorem guarantees the existence of a 0-core set with a size of at most .

Algorithm 3 is closely related to the Frank–Wolfe algorithm with proper initialization, applied to the dual formulation of the minimum enclosing ball problem. In each iteration, the algorithm can only add points to the working core set. This algorithm computes a -approximation to MEB() in operations and converges in iterations, matching the currently best-known dependence on epsilon. In addition, it explicitly computes a ‘core set’ of size , which is independent of both and .

Algorithm 3 constructs a sequence of balls with strictly increasing radii in each iteration, and it updates both the radius and the error bound at each iteration. In each iteration, the center is shifted towards the furthest point from the center of the current ball, but the center's movement is limited to only a fraction of this distance. Also, a termination criterion is checked in each iteration, which has the potential advantage of earlier termination than that what is predicted by the theoretical worst-case estimate.

**Algorithm 3: *-*approximation to MEB**

1. **While** and **do**
2. **loop**







11. **end loop**

We will now provide a more detailed description of Algorithm 3. In step 1, the algorithm calculates the furthest point from and then computes the furthest point from . Steps 2 and 3 initialize the vector . It’s worth noting that is a feasible solution of the dual problem (TODO). The core set is initialized at step 4. In each iteration, the algorithm implicitly constructs a ‘trial’ ball with center and radius . This ball coincides with MEB() if and only if is an optimal solution of (TODO). Otherwise, it means that at least one point in lies outside of this ball. Here, is such that , where is the furthest point from . Consequently, the trial ball encloses if its radius is expanded by a factor of , i.e., . Unless the termination criterion is satisfied, the new center is computed by moving toward the furthest point , which is then added to the working core set . is updated accordingly to ensure that dual feasibility is maintained. The algorithm continues iteratively, computing a new trial ball corresponding to .

At each iteration of Algorithm 3, the quadratic objective function of (TODO) is linearized at the current feasible solution . Since the feasible region of (TODO) is the unit simplex, the unit vector , where is the index of the furthest point in from , solves the linearized subproblem. It is easy to verify that:

We remark that Algorithm 3 uses only the first-order approximation to the objective function . As such, each iteration is fairly cheap, but the number of iterations is usually significantly higher than other algorithms that use second-order information such as interior-point methods. However, such general-purpose algorithms become computationally infeasible for larger problems, since each iteration is usually much more expensive.

Lemma 3.1. satisfies , whereand are the optimal solution and the optimal value of (TODO), respectively. Furthermore, .

Lemma 3.2: For each the following relationship is satisfied:

The main idea behind Algorithm 3 is to approximate the given input set using only a carefully selected finite subset of points and then to refine this approximation by adding more points if necessary. This leads to an approximation of the primal formulation with only a finite number of constraints, and this approximation is refined by adding more constraints. In the dual formulation, we therefore start with a finite number of variables and add more variables if necessary.

The initial working core set provides the first approximation to the given input set with only two points. Let denote the objective function of the dual formulation of the minimum enclosing ball problem for and let denote the optimal value of the aforementioned semi-infinite primal formulation.

Similarly, let denote the objective function of the dual formulation of the minimum enclosing ball problem for . At iteration t in each algorithm, provides the current finite approximation to . Let denote the current center. Algorithm 3 computes the furthest point in from . In Algorithm 3, is obtained by adding this point to . Unless the furthest point in already belongs to , the dual formulation for differs from that for in only one variable. Therefore, is a feasible solution for the new dual formulation that satisfies which implies that the improvement in each iteration still obeys the relation given by Lemma 3.2, with replaced by and by . Note that the dimension of is one more than that of uk in this case.

## Line search strategies

Algorithms 1 and 2 need to perform a line search in each iteration t to find a suitable learning rate that is within the range . There are several options available for performing this line search.

### **Inverse time decay learning rate**

In the initial stages of our work, we began with a relatively straightforward strategy. We employed a basic formula to establish a diminishing learning rate, one that relied solely on the current iteration - represented as . While this approach did yield satisfactory results in terms of generating reasonable parameters (center and radius) for the MEB and kept CPU time to a minimum, both algorithms struggled to converge.

Increasing the number of iterations would lead to a significant increase in CPU time, which was against our objective of being efficient. This emphasized the necessity for a more sophisticated and refined strategy.

### Golden section search

The golden section line search is a technique for finding the minimum of a unimodal function within a specified interval, such as in our case. The method works by iteratively shrinking the search interval while ensuring that the ratio of the smaller subinterval to the larger subinterval remains the golden ratio, approximately 1.618. By doing this, it narrows down the search space efficiently, converging towards the optimal solution.

The golden section line search is relatively simple and relies solely on (dual) function evaluations. Our experiments demonstrated its high efficiency when used in conjunction with Algorithm 1. It consistently converged well before reaching the maximum iteration limit (1000). However, in combination with Algorithm 2, the performance was poor. It failed to converge, and the returned radius was much smaller than the optimal value.

### Armijo’s rule

We proceeded by implementing an inexact backtracking line search, also known as Armijo’s rule. In this case, the learning rate is determined through an iterative search process until a sufficient decrease of the objective function is achieved. We fixed the parameters as follows: , , and initiated with a large starting step-size . We then iteratively (and geometrically) decreased by multiplying its current value by , represented as , where until inequality

is satisfied, and then we return

Even though the obtained results for the center and radius using Armijo’s rule were close to the optimal ones, it was evident that this line search strategy is very slow, and often failed to converge. This issue can be attributed to the problem of ill-conditioning. A high condition number typically leads to slow convergence, and given the randomness of data sampling, such a scenario was likely to occur.

An illustrative example of this situation arises when data points are nearly collinear or closely aligned on the same hyperplane. In such cases, it becomes challenging for the algorithm to identify a unique and well-defined minimum enclosing ball. For instance, in a 2D context, one feature might have small values, while another feature may exhibit significantly larger values.

Given that we had no control over the ill-conditioning, which depended entirely on the feature values of the data points in the datasets, and considering our intention to evaluate the algorithms on various benchmark datasets, we made the decision to proceed with the best option: an exact line search.

### Exact line search

In this approach, instead of relying on iterative approximations like in some other line search methods (e.g., Armijo's rule or golden section search), the exact value of the learning rate that would guide us to the minimum can be computed analytically, as the objective function is quadratic and convex.

When examining the minimization version of our dual function, i.e., **,** we can represent it in the form of , where is a symmetric matrix, by substituting: and **.**

Suppose that we have selected a step direction . In this case, to find the exact step-size, we need to minimize , which can be expressed as:

This is a quadratic function of , and thus we can compute the optimal step-size by finding the , such that . By expanding out the quadratic term, it is easy to show that:

Setting this equal to zero and solving for yields the step-size:

or equivalently:

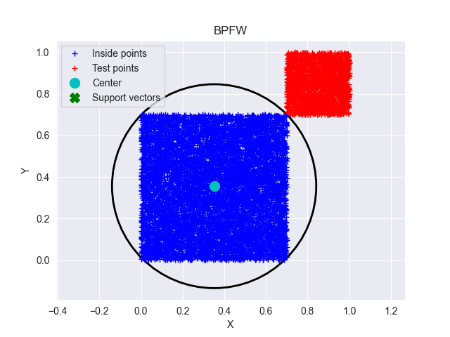
where the gradient at is **,** or equivalently **.**

## Experiments

The experiments aim to assess the quality of three given algorithms on both synthetic and real-world datasets. The experimental section consists of two parts. The first part involves training and testing the algorithms on synthetic datasets. In the second part, we apply a similar approach to real-world datasets, also incorporating epsilon (ε) as a variable hyperparameter. All experimental results were obtained using a Windows workstation with a 2.4GHz Intel i5-1135G7 CPU and 8GB DDR4 3200MHz Memory. The algorithms were implemented in Python 3.10.

### I. Random datasets

#### 1. Uniform distribution

This initial experiment aimed to assess the quality of the MEB returned by the three algorithms. To achieve this, we created two closely spaced yet separable clusters: one for training with 8000 data points sampled from U[0.0, 0.7), and the other for testing (containing anomalies) with 2000 points sampled from U[0.7, 1.0).

In order to improve visualization and comprehension of the results, we began with only two features. This allowed us to display the data in 2D alongside the generated circle (using the center and radius obtained), as depicted in Figure TODO. With this configuration, all three algorithms achieved perfect accuracy. This was expected since the datasets were sampled using different ranges, ensuring cluster separability.

Figure 3: The blue cluster represents nominal points, while the red one represents anomaly points. The circle shows the MEB constructed by Algorithm 1.

We then proceeded to increase the number of features. For every dimension we explored, we consistently obtained perfect results. Table TODO presents our findings for a dataset with an arbitrarily chosen dimension of 15, while Figure TODO displays various plots that comprehensively depict the training process (the construction of the MEB).

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Training** | | | | | **Test** | | |
|  | *8000 MEB samples from U[0,7)* | **Iterations** | **CPU time (s)** | **Radius** | **Active set size** | *2000 MEB samples from U[0.7,1)* | **Recall (%)** | **F1 score (%)** |
| **ASFW** | 379 | 23.632 | 1.019375 | 15 | 100 | 100 |
| **BPFW** | 272 | 16.606 | 1.019365 | 14 | 100 | 100 |
| **(1+ε)-approx.** | 747 | 22.108 | 1.019697 | 18 | 100 | 100 |

Table 1: TODO

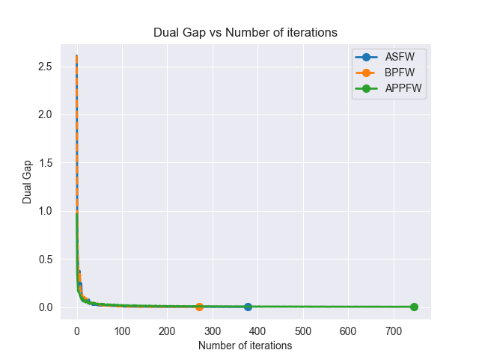
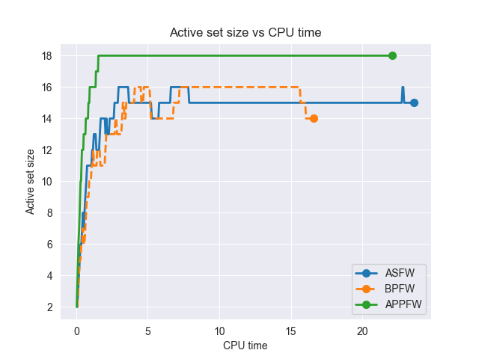
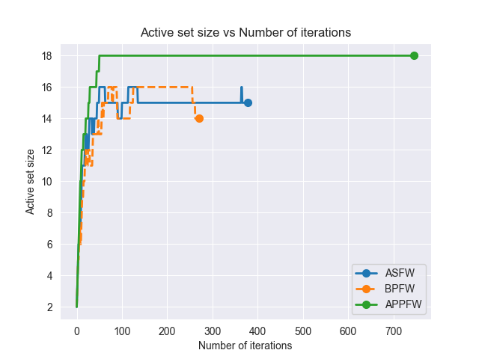
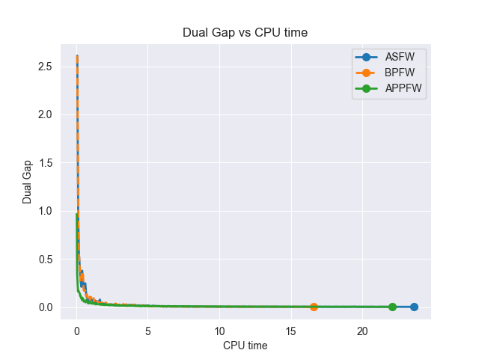
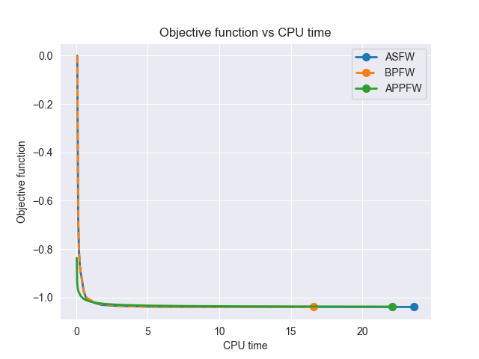
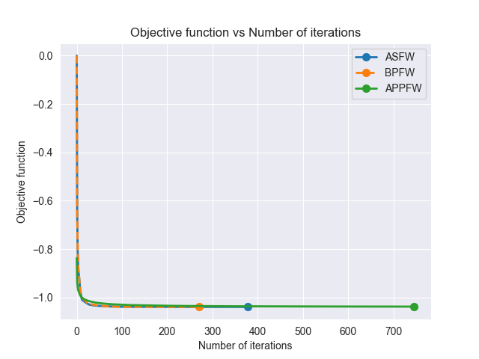


Figure 4: Results obtained from training (the construction of the minimum enclosing ball) using a training set consisting of 8000 data points with a dimension of 15. These data points were drawn from U[0, 0.7). The plots on the left depict how the objective function, dual gap, and size of the active set change over each iteration. Meanwhile, the plots on the right showcase the corresponding changes in CPU time. Each algorithm is represented by a distinct color on the graphs.

Discussion (roughly written, no chatGPT 😊):

BPFW (Algorithm 2) takes the least number of iterations (272) and also the lowest total CPU time to converge. This means that this algorithm makes the most amount of progress per iteration. On the other hand, ASFW (Algorithm 2) needs less iterations than APPFW (Algorithm 3), but slightly more CPU time to converge.

Since the primal error is not computable (the optimal solution (c\*, r\*) is unknown), the duality gap represents a good optimality measure (stopping criterion) that we obtain for free from our algorithms. From the second row of graphs, we can observe how the dual gap/delta value decreases per iteration/CPU time. It’s evident that all of the algorithms are very far from linear convergence rate. Closer inspection shows that the convergence rate is the best for BPFW, then for ASFW, and lastly for APPFW.

Finally, from the last row of plots we can observe that BPFW is also better than the other two algorithms in terms of the size of the active set . We consider this to be very beneficial, because in many instances (especially for big data), storing the active set can be very costly. Whenever the line (function) decreases by one, that means that an atom was removed from the active step, i.e., the algorithm took a drop step. We cannot guarantee sufficient progress in this case, but luckily these cannot happen too often. The ASFW took 8 drop steps (out of 379 iterations), while BPFW took 10 (out of 239 iterations). The APPFW doesn’t have drop steps, which can be seen on the plots: the green line only increases in each iteration. In Algorithm 3, when a new center is computed the current center is shifted towards the furthest point, which is added to the working core set.

From the paper (Yildirim):

The main idea is to approximate the given input set using only a carefully selected finite subset of points and then to refine this approximation by adding more points if necessary. This leads to an approximation of the primal formulation with only a finite number of constraints, and this approximation is refined by adding more constraints. In the dual formulation, we therefore start with a finite number of variables and add more variables if necessary.

#### 2. Gaussian distribution

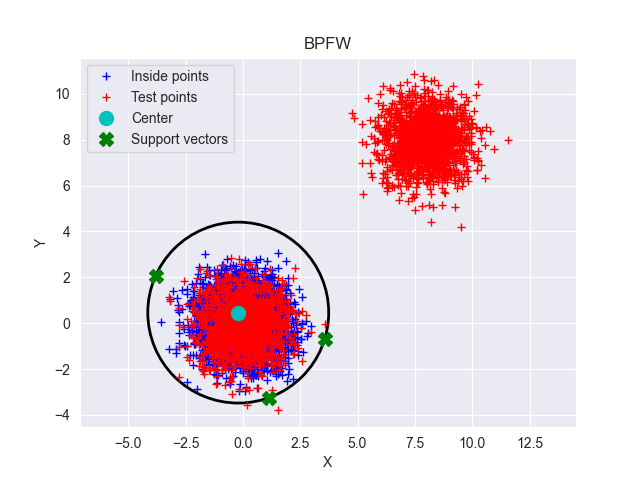
Next, another synthetic dataset was generated, consisting of 8000 samples for training and 2000 samples for testing, with each sample having 10 features. The training dataset was created by randomly sampling from normal distribution N(0,1). For the testing dataset, half of the samples were drawn from the same Gaussian distribution as the training dataset, and these are referred to as normal data. The other half of the testing dataset was generated by sampling from a different Gaussian distribution N(7,1) deliberately chosen to create well-separated clusters. These samples are referred to as novel or anomaly data.

Figure 5: The blue cluster represents training points, while the red one represents test points. The circle shows the MEB constructed by Algorithm 2. Support vectors are training points that lie on the boundary (MEB). The red points around the center of the MEB are test points belonging to the nominal class, while the red cluster on the top right represents the anomaly points in the test set.

Same as above, we initially conducted the experiment using only 2 features to enhance visual comprehension. The outcomes for this setup are presented in Figure TODO. Following that, we performed training and testing using the 10-feature dataset. The training results per iteration and per CPU time are presented in the figure below.

Discussion:

Both from the graphs and from the table, we can see that the APPFW takes the most iterations to converge, and also most CPU time. This can be attributed due to the high epsilon value being used in this experiment (0.001). To test this claim, we reduced the epsilon value to 0.1, and obtained the following results: 0.201s and 7 iterations for APPFW, 2.884s and 51 iterations for BPFW and 3.454s and 62 iterations for ASFW. This proves our claim, and suggests that the epsilon value allows us to make a trade-off between correctness and speed. The APPFW algorithm, as it is an approximation algorithm, can be very efficient in scenarios when we need a very fast solution, for the cost of increasing the error slightly. While increasing the epsilon value decreased the CPU time for BPFW and ASFW approximately by a factor of 3, for the APPFW this decrease is significantly higher (100 times faster).

The analysis of the dual gap/delta value per iteration gives the same finding as before: the BPFW has best convergence rate, which means it makes the highest progress per iteration.

The active set per iteration/CPU time for ASFW is almost identical to BPFW. For this experiment, both of these algorithms make just two drop steps. On the other hand, the third algorithm just adds atoms to the active set up until iteration 80, after which the size of the active set (core set) remains the same, i.e., 12.

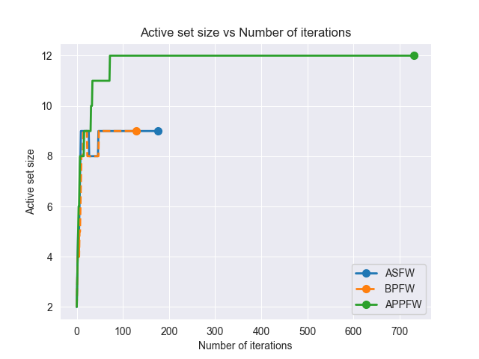
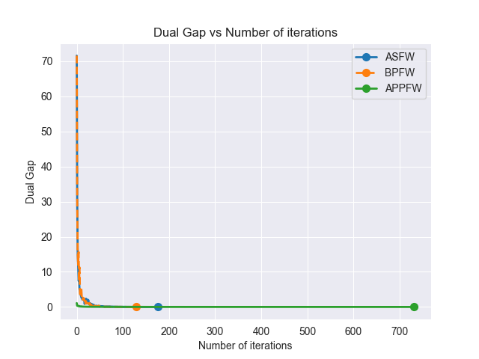
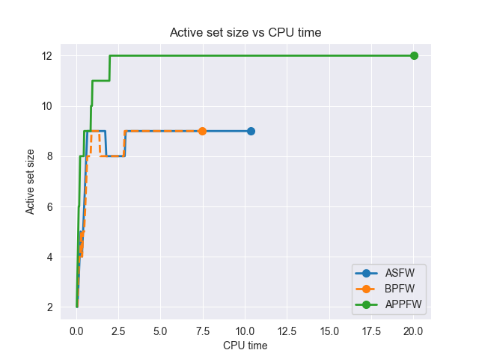
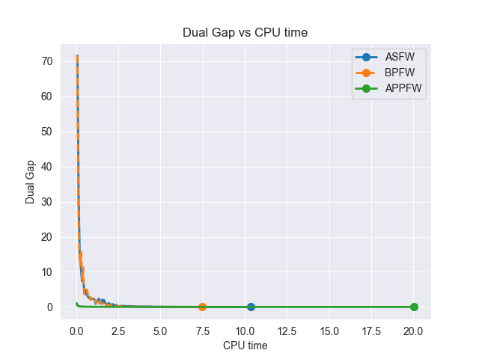
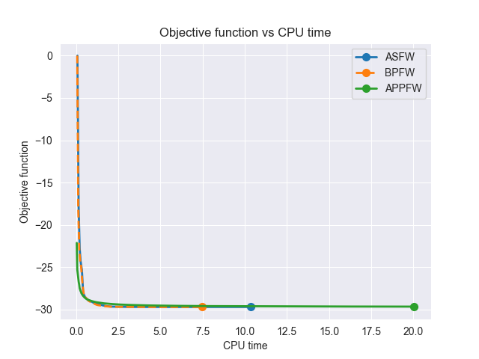
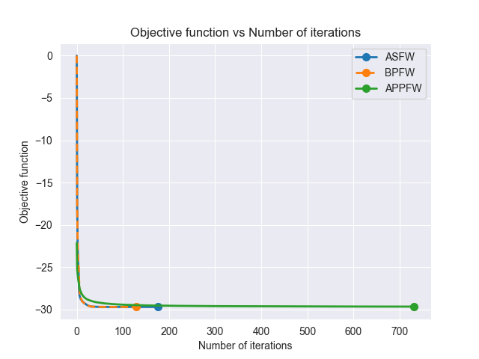


Figure 6: Results obtained from training (the construction of the MEB) using a training set consisting of 8000 data points with a dimension of 10. These data points were drawn from N(0, 1). The plots on the left depict how the objective function, dual gap, and size of the active set change over each iteration. Meanwhile, the plots on the right showcase the corresponding changes in CPU time. Each algorithm is represented by a distinct color on the graphs.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Training** | | | | | **Testing** | | |
|  | *8000 MEB samples from N(0,1)* | **Iterations** | **CPU Time (s)** | **Radius** | **Active set size** | *1000 MEB samples from N(0,1) + 1000 anomalies from N(7,1)* | **Recall (%)** | **F1 Score (%)** |
| **ASFW** | 177 | 9.614 | 5.45032 | 9 | 100 | 99.85 |
| **BPFW** | 130 | 7.219 | 5.45032 | 9 | 100 | 99.85 |
| **(1+ε)-approx.** | 733 | 21.904 | 5.45191 | 12 | 100 | 99.85 |

Table 2: TODO

A summary of the results from training and testing with the three different algorithms can be seen in the table above. As observed from the table, in the second experiment, the F1 score is not perfect. We conducted an investigation to understand why this could happen, and here we present our findings.

The test dataset contains 2000 points: 1000 anomalies and 1000 nominal. All anomaly points are correctly classified as anomalies (true positives). We can gain a better understanding of these anomalies by referring to Figure 5, where they are represented by the red cluster in the top right of each plot.

However, there are instances of false positives: 3 out of 1000 points that were intended to be classified as nominal have been mistakenly classified as anomalies. These points can also be observed in Figure TODO as the red points positioned below the MEB. The reason for these false positives is that some nominal points fall just slightly outside the MEB. We do not consider this to be a significant issue.

One potential solution could be to classify as anomalies only those points that are farther than δ\*r from the center, where δ is a constant greater than 1, and r represents the radius of the MEB. However, we have chosen not to pursue this approach because we anticipated that real-world datasets may exhibit overlapping characteristics, and selecting an appropriate value for δ would require additional fine-tuning.

#### II. Benchmarking on real-world datasets

In this section, we evaluate all three algorithms using two distinct real-world datasets and conduct a comparative analysis of the results. We also provide comprehensive descriptions of the datasets and explain the preprocessing procedures applied. Different ε (epsilon) values were used for each experiment to vary the tightness of the minimum enclosing ball.

##### Preprocessing

For both datasets, as described in the sections below, we initially ensured the data's validity. After a brief exploratory data analysis (EDA), it became evident that feature scaling was necessary due to the features being on different scales. This step is essential to prevent one feature from dominating the radius computation when we are dealing with Euclidean distances. We considered two options for scaling: standard scaling and min-max scaling. After experimentation, we opted for standard scaling since it yielded better results.

During the preprocessing phase, we converted binary labels from strings to numeric values (0 for nominal data, 1 for anomaly data) and removed irrelevant features. Subsequently, the data was divided into two subsets: one containing normal cases (*nominal\_features* and *nominal\_labels*) and the other anomaly cases (*anomaly\_features* and *anomaly\_labels*). Importantly, a portion of the nominal data was allocated for training, while the remaining nominal data and all anomaly data comprised the test set. Finally, the data underwent standardization for feature scaling. To fit the scaler, we exclusively used the training data. Subsequently, we separately transformed the training and test data using this scaler, ensuring that we used the mean and standard deviation from the training set's portion of the nominal data.

As only the training data is used by the algorithms, they will only learn the distribution of the normal data, including the center and the enclosing radius. In the end, we hope that this will enable us to differentiate between normal and anomaly data.

##### 1. Breast Cancer Wisconsin Dataset

The “Breast Cancer Wisconsin (Diagnostic)”, sourced from Kaggle, provides a comprehensive collection of information derived from digitized images of fine needle aspirates (FNA) of breast masses. These characteristics and features describe the cell nuclei found within the images, with the primary aim of distinguishing between malignant and benign cases of breast cancer. This dataset consists of 569 rows and 32 columns.

The dataset encompasses the following attributes:

* *ID number*: A unique identifier for each data point.
* *Radius*: The mean distance from the center of the nucleus to points on its perimeter.
* *Texture*: The standard deviation of gray-scale values within the nucleus.
* *Perimeter*: The perimeter of the nucleus.
* *Area*: The area enclosed by the nucleus.
* *Smoothness*: A measure of local variation in the lengths of the nucleus radii.
* *Compactness*: A derived measure, computed as (perimeter^2 / area - 1.0).
* *Concavity*: A measure of the severity of concave portions within the contour of the nucleus.
* *Concave points*: The count of concave portions present in the contour of the nucleus.
* *Symmetry*: A measure of symmetry in the nucleus.
* *Fractal dimension*: A measure based on the coastline approximation," reflecting the complexity of the contour.
* *Diagnosis*: The outcome of interest, categorized into two classes: 'M' for malignant and 'B' for benign.

Our model included all features except the ID number, which was omitted because IDs should not be used in classification models. For all the other features, three statistics are provided: the mean, standard error, and "worst" or largest (mean of the three largest values). Notably, this dataset does not contain any missing values, ensuring data completeness. In terms of class distribution, it comprises 357 benign cases and 212 malignant cases, making it a valuable resource for developing predictive models to aid in the diagnosis of breast cancer and use it for anomaly detection as we are doing.

**TODO: Talk about experiments, and the train-test split distribution**

To create training and testing datasets, we first separated the nominal data and the anomaly data. Half of the nominal data was used for training (178 samples), and the other half for testing. On the other hand, the anomaly data was used just for testing. Therefore the testing dataset comprised of 391 samples (179 nominal and 212 anomalies). All of the samples were 33-dimensional.

Varying thresholds were used for the stopping criterion, and all of these are reported in the Table TODO.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Training** | | | | | **Testing** | | | |
| **ε** | **Iterations** | **CPU Time (in ms)** | **Radius** | **Active set size** | **FN** | **FP** | **Recall (%)** | **F1 Score (%)** |
| **ASFW** | 0.1 | 64 | 26.38 | 11.077366 | 6 | 50 | 5 | 76.415 | 85.488 |
| 0.01 | 92 | 36.97 | 11.077386 | 6 | 50 | 5 | 76.415 | 85.488 |
| 0.001 | 118 | 98.634 | 11.077386 | 6 | 50 | 5 | 76.415 | 85.488 |
| **BPFW** | 0.1 | 32 | 18.165 | 11.077378 | 6 | 50 | 5 | 76.415 | 85.488 |
| 0.01 | 50 | 24.112 | 11.077386 | 6 | 50 | 5 | 76.415 | 85.488 |
| 0.001 | 66 | 31.402 | 11.077386 | 6 | 50 | 5 | 76.415 | 85.488 |
| **(1+ε)-approx.** | 0.1 | 4 | 2.004 | 11.758557 | 5 | 59 | 3 | 72.17 | 83.152 |
| 0.01 | 35 | 11.0 | 11.130932 | 7 | 50 | 5 | 76.415 | 85.488 |
| 0.001 | 608 | 146.278 | 11.080393 | 7 | 50 | 5 | 76.415 | 85.488 |

Table 3: TODO

Discussion:

On the right part of Table TODO, we can see the results from testing. It’s evident that changing the epsilon value didn’t change the outcome much. The only deviation that occurred was for epsilon=0.1 and the third algorithm. However, the decrease in F1 score is very small, so we can disregard this.

The testing results allow us to come to a conclusion that the classes in the Breast Cancer dataset are well separated, and even a naïve approach as this one is able to do well. Having this thought in mind, we aimed to find an opposing scenario, i.e., a dataset for which the classes are overlapping, and see a different perspective of whether this approach is suitable for more complex scenarios.

(the next can be placed bellow the graphs)

The graphs on Figure TODO show the results obtained using epsilon=0.001. We can see that for this desired correctness of the MEB, the BPFW algorithm converges the quickest both in regards to iterations and CPU time. On the other hand ASFW needs drastically less iterations compared to APPFW (118 vs 608), and also it’s CPU times is much less (60% faster)

The active set size graphs are a little crowded, however they show a generally increasing trend, which means that the number of drop steps is low. Further investigation shows that this is correct: 1 drop step for ASFW and 2 for BPFW. This means that most of the iterations are influential. However, we still take a lot of away/descent steps, which can be observed by the zig-zagging line in the dual gap per iteration/CPU time plots. ***(I am not sure whether this sentence is correct.)***

The value of epsilon has the highest influence for the third algorithm. Higher values drastically reduce the CPU time: from 146ms to 2ms when increasing epsilon from 0.001 to 0.1. The number of iterations also decreases greatly: from 608 to 4. However, as a consequence the F1 score and recall slightly drop. In this case the trade-off is good, because the decrease in the testing metrics is minimal, while the decrease during training is substantial.

Figure 8:

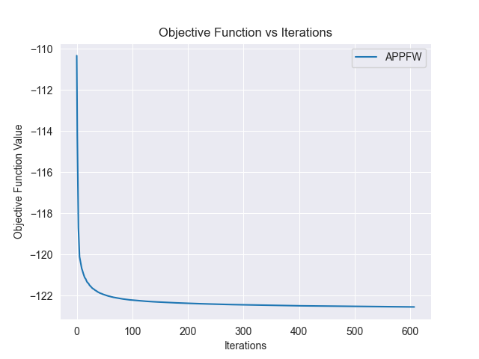
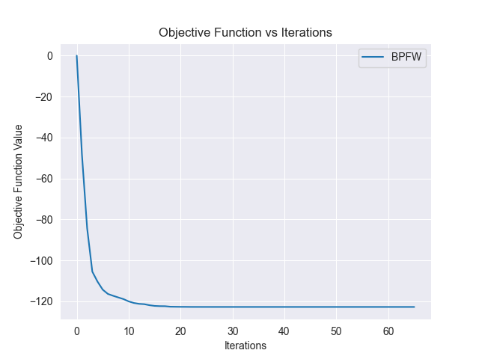
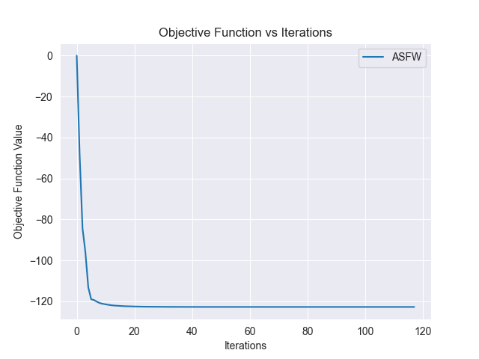
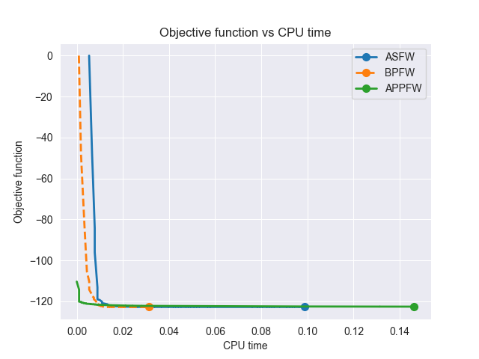
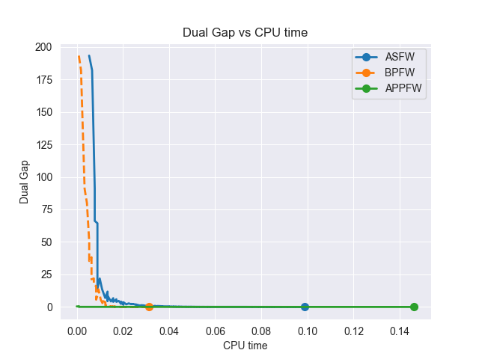
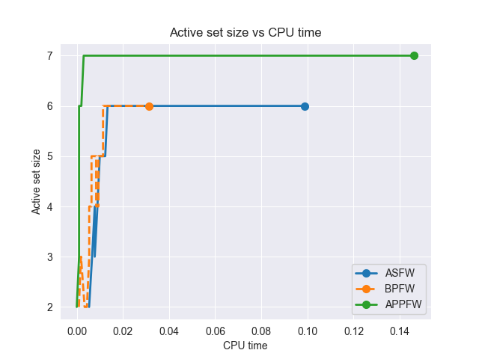
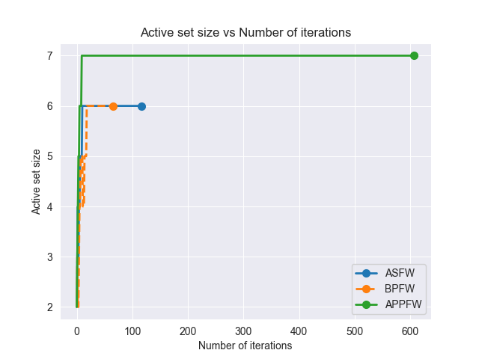
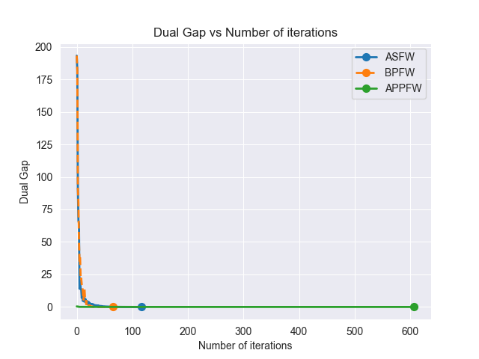
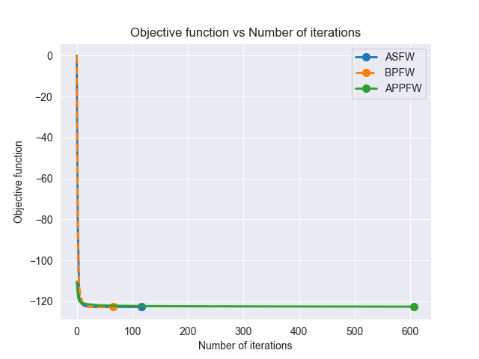


Figure 7: TODO Change these, use Delta/Dual gap vs iterations, since we use that for the convergence analysis

##### 2. Iranian Churn Dataset

The "Iranian Churn Dataset" from the UCI Machine Learning Repository is a collection of data randomly gathered from a telecommunications company in Iran, spanning a period of 12 months. This dataset contains a total of 3,150 rows, with each row representing a unique customer. The information is organized into 14 columns, each providing insights into customer behavior and characteristics.

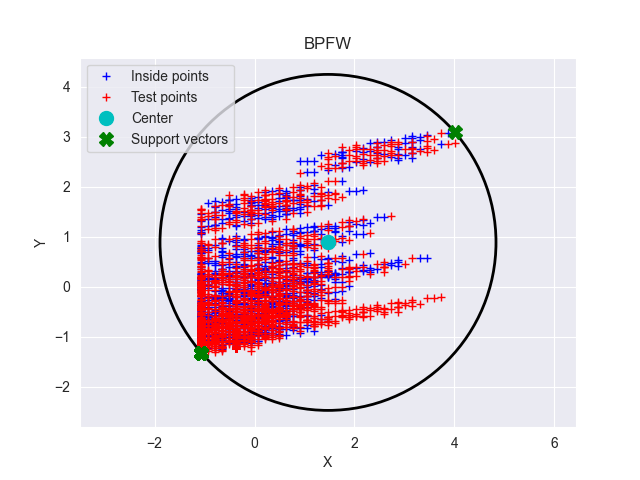
The key attributes included in this dataset are as follows:

* *Anonymous customer ID*: A unique identifier assigned to each customer, ensuring data privacy.
* *Call failures*: This attribute indicates the number of call failures experienced by each customer.
* *Complaints*: A binary attribute with values 0 (no complaint) and 1 (complaint), denoting whether a customer registered a complaint.
* *Subscription length*: The total number of months for which a customer has maintained a subscription.
* *Charge amount*: An ordinal attribute representing the charge amount, with values ranging from 0 (lowest amount) to 9 (highest amount).
* *Seconds of use*: The total number of seconds a customer has spent making calls.
* *Frequency of use*: This attribute reflects the total number of calls made by a customer.
* *Frequency of SMS*: The total number of text messages sent by a customer.
* *Distinct called numbers*: The total count of distinct phone numbers a customer has called.
* *Age group*: An ordinal attribute categorizing customers into age groups, with values ranging from 1 (younger age) to 5 (older age).
* *Tariff Plan*: A binary attribute indicating the type of service plan a customer is on, with values 1 (pay as you go) and 2 (contractual).
* *Status*: Another binary attribute that indicates whether a customer is currently active (1) or non-active (2).
* *Customer value*: This attribute represents a calculated value that likely encapsulates a customer's overall worth or contribution to the telecom company.
* *Churn*: This binary attribute serves as the class label, distinguishing between customers who have churned (1) and those who have not (0) at the end of the 12-month period. Churn signifies customers who have discontinued their services.

**TODO: Talk about experiments, and the train-test split distribution**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Training** | | | | | **Testing** | | | |
| **ε** | **Iterations** | **CPU time (in ms)** | **Radius** | **Active set size** | **FN** | **FP** | **Recall (%)** | **F1 Score (%)** |
| **ASFW** | 0.1 | 55 | 131.633 | 6.708818 | 6 | 445 | 2 | 10.101 | 18.282 |
| 0.01 | 80 | 154.788 | 6.709065 | 6 | 447 | 2 | 9.697 | 17.615 |
| 0.001 | 100 | 217.184 | 6.709067 | 6 | 447 | 2 | 9.697 | 17.615 |
| **BPFW** | 0.1 | 32 | 74.47 | 6.709004 | 6 | 448 | 3 | 9.495 | 17.248 |
| 0.01 | 40 | 97.306 | 6.709067 | 6 | 447 | 2 | 9.697 | 17.615 |
| 0.001 | 54 | 116.559 | 6.709067 | 6 | 447 | 2 | 9.697 | 17.615 |
| **(1+ε)-approx.** | 0.1 | 3 | 5.293 | 7.184624 | 4 | 491 | 1 | 0.808 | 1.6 |
| 0.01 | 13 | 14.184 | 6.753838 | 6 | 454 | 1 | 8.283 | 15.27 |
| 0.001 | 178 | 178.887 | 6.711496 | 7 | 447 | 2 | 9.697 | 17.615 |

Table 4: TODO

-> explain the bad test results using this photo

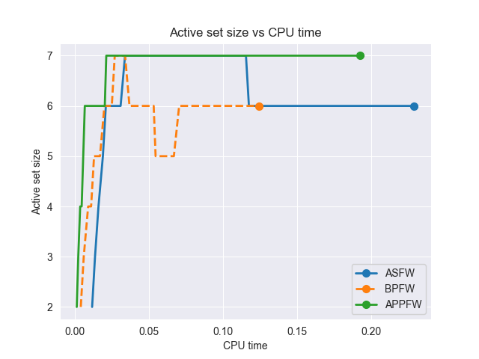
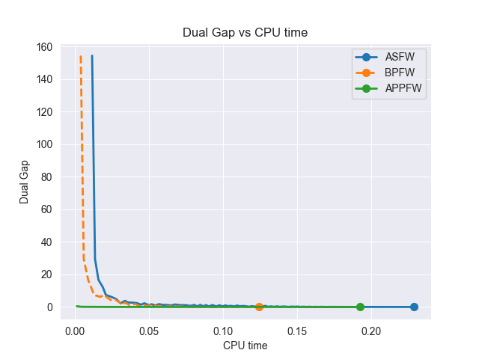
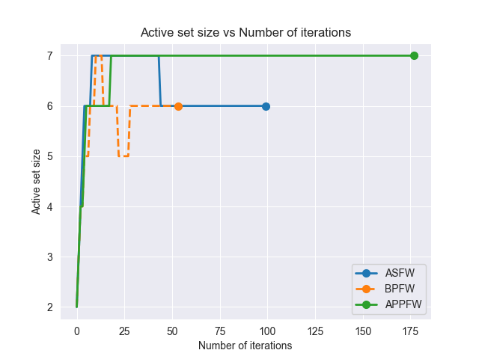
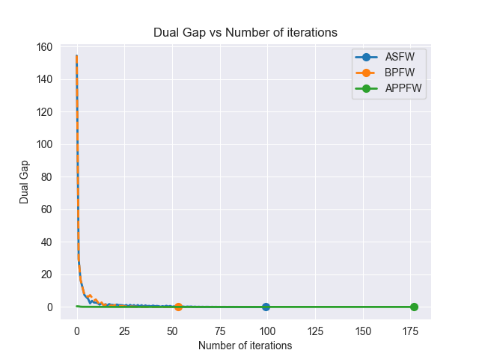
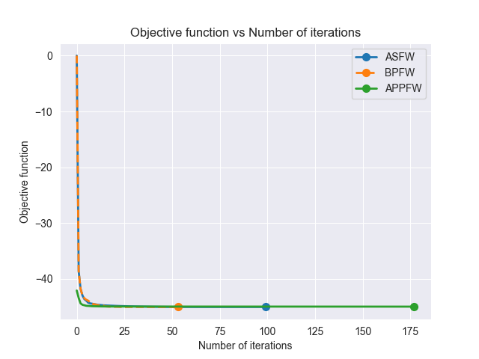
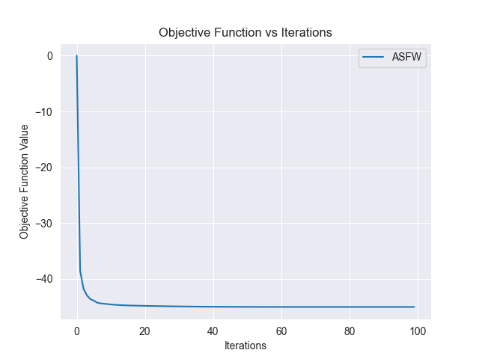
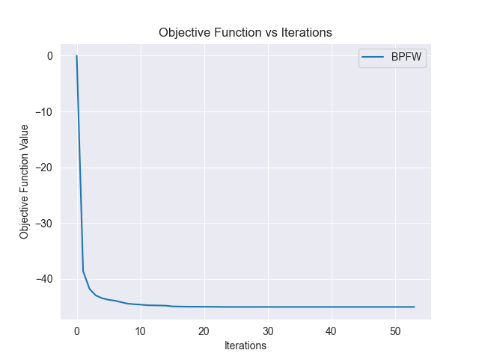
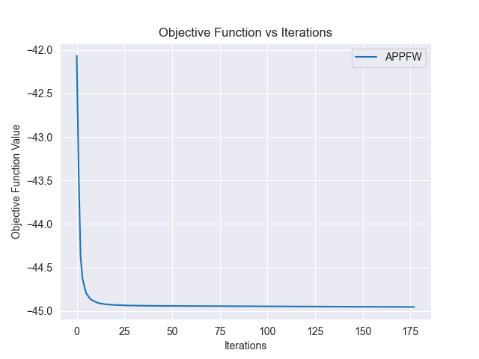


Figure 9: TODO

Figure 10: Objective function values per iteration for the three algorithms on the Iranian Churn Dataset – Change to Dual value



# References

Bauckhage, C. (2020). *NumPy / SciPy Recipes for Data Science: Frank-Wolfe for Minimum Enclosing Balls.* Retrieved from https://www.researchgate.net/publication/344138310\_NumPy\_SciPy\_Recipes\_for\_Data\_Science\_Frank-Wolfe\_for\_Minimum\_Enclosing\_Balls

Bauckhage, C. a. (2019). *Lecture Notes on Machine Learning: Frank-Wolfe for Minimum Enclosing Balls.* Retrieved from https://www.researchgate.net/publication/331315423\_Lecture\_Notes\_on\_Machine\_Learning\_Frank-Wolfe\_for\_Minimum\_Enclosing\_Balls

Bauckhage, C. a. (2019). *Lecture Notes on Machine Learning: Minimum Enclosing Balls.* Retrieved from https://www.researchgate.net/publication/331152293\_Lecture\_Notes\_on\_Machine\_Learning\_Minimum\_Enclosing\_Balls

Davenport, M. A. (2021, Fall). *Line search methods and convergence of gradient descent revisited.* Retrieved from ECE 3803: Optimization for Information Systems: https://mdav.ece.gatech.edu/ece-3803-fall2021/notes/11-notes-3803-f21.pdf

Jaggi, S. L.-J. (2015). On the Global Linear Convergence of Frank-Wolfe Optimization Variants. Neural Information Processing Systems 2015.

Jiang, B. (2020, November 9). *John Hopcroft Center for Computer Science.* Retrieved from CS257 Linear and Convex Optimization, Lecture 10: https://jhc.sjtu.edu.cn/public/courses/CS257/2020/slides/lec10.pdf

Kazuma Tsuji, K. T. (2022). Pairwise Conditional Gradients without Swap Steps and Sparser Kernel Herding. Proceedings of the International Conference on Machine Learning.

Yildirim, E. A. (2008, 01). Two Algorithms for the Minimum Enclosing Ball Problem. *SIAM Journal on Optimization, 19*, 1368-1391. doi:10.1137/070690419