Universidad de Oviedo

Escuela de Ingeniería Informática
Universidad de Oviedo

Python implementation of an Unsupervised Learning algorithm: leveraged affinity propagation

Bachelor thesis in Software Engineering

Héctor Díaz Beltrán

Tutors

Juan Luis Mateo Cerdán

Antonio Martínez Sánchez

Oviedo, July 2023

School of Computer Engineering

Universiy of Oviedo

Acknowledgements

I would like to thank my two supervisors, Juan Luis Mateo Cerdán and Antonio Martínez Sánchez, for their guidance and support throughout this bachelor's thesis. I also extend my thanks to the faculty and staff of the University of Oviedo and the School of Computer Engineering for providing an enabling academic environment.

To my family, thanks for always being there for me, supporting and understanding my academic journey. To my friends, thank you for your support and for always being there to lend an ear or offer a helping hand. Your positive energy and shared enthusiasm have made this journey all the more enjoyable.

Abstract

This work presents a new Python library for Leveraged Affinity Propagation (LAP), a clustering technique used in machine learning and data analysis. LAP is designed to reduce memory usage compared to existing Affinity Propagation (AP) implementations, including those in scikit-learn and the original R library. The library's performance and memory usage are evaluated against these existing methods through various experiments on benchmark datasets, both synthetic and real-world. The results demonstrate that the proposed library achieves competitive performance while significantly reducing memory usage. The library is a valuable addition to the Python data science ecosystem and offers an efficient and effective tool for AP-based clustering analysis.

Keywords

Affinity Propagation, Unsupervised learning, Clustering, Python, Cython

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Clustering algorithms play a pivotal role in data analysis, enabling the identification of inherent patterns and structures within datasets, without the need for labels. The result of a clustering algorithm is the division of the dataset into groups or clusters.

Clustering algorithms are a type of unsupervised learning technique used in machine learning and data analysis to discover patterns or groupings in a dataset. The goal of clustering is to fit the data points into distinct groups, also called clusters, based on their similarities or proximity to each other.

Unsupervised learning refers to the process of finding patterns or structures in data without any labels or target variables present. In contrast with supervised learning, where the algorithm is provided with labeled data to learn from, unsupervised learning relies only on the given input data.

There are multiple types of techniques available to perform clustering. Between them, there are some groups that are heavily differentiated and more commonly available in machine learning libraries [1].

Based on partition: This kind of algorithm set the central element of the data points as the center of the cluster. One of the most recognizable algorithms in this category is K-means. The K-means algorithm iterates over the cluster centers until it converges, meaning that K-means has reached a stable and desired solution. K-means requires specifying the number of clusters K, which can be a detriment when the number of clusters to generate is not known *a priori*, requiring experimentation and multiple runs to find an optimal number of clusters.

Based on hierarchy: This kind of clustering algorithm organizes data points into a hierarchical structure based on their similarity or dissimilarity. There exist two methods to create the hierarchy of clusters, depending on the "direction" that the algorithm takes. Agglomerative clustering starts with each data point as a separate cluster and progressively merges the most similar clusters, while divisive clustering begins with a single cluster and recursively splits it. These hierarchical techniques also require specifying the number of clusters to generate.

Based on density: This kind of clustering algorithm group data points based on their density within the dataset. These algorithms are particularly effective in identifying clusters of arbitrary shapes and handling datasets with varying density or containing noise or outliers. The most well-known density-based clustering algorithm is DBSCAN and its variations [1]. DBSCAN does not require specifying the number of clusters to identify.

Affinity Propagation (AP)

AP has emerged as a powerful and versatile clustering algorithm, demonstrating impressive performance across various applications, particularly in bioinformatics [2]. It can be classified as a partitioning-based method [1], and unlike others of the same kind, it does not require specifying the number of clusters to find. However, AP's scalability and memory efficiency become challenging when confronted with large datasets, due to its quadratic scaling memory and computation requirements. To address this limitation, this work investigates a variation of AP, known as Leveraged Affinity Propagation (LAP), specifically designed for handling large datasets.

LAP, implemented as a new Python library, fills a gap in the current landscape of clustering tools by providing a scalable and memory-efficient solution for Python. While similar libraries exist for R (*apcluster* library) [3] and MATLAB, the availability of LAP in Python opens up opportunities for a wider community of data scientists. This work explores the implementation details of the LAP library, emphasizing its compatibility with the *scikit-learn* [4] interface to ensure ease of integration into existing workflows and benchmarks it.

The primary objective of this investigation is to evaluate the performance of LAP in comparison to traditional AP, particularly concerning its ability to handle large datasets without exhausting available memory. The benchmarking process involves subjecting the LAP library to various small and large datasets and assessing its memory consumption, computational efficiency, and clustering performance. The comparative analysis highlights the advantages of LAP over conventional AP when it comes to large datasets.

To ensure broader accessibility, the LAP library will be made available on PyPI, the Python Package Index, allowing Python programmers to easily install and utilize it in their projects. The library's compatibility with the scikit-learn interface ensures seamless integration with existing machine learning pipelines, providing a common environment.

In summary, this work introduces a new Python library for Leveraged Affinity Propagation (LAP), designed to address the challenges of handling large datasets. The evaluation benchmarks LAP against traditional AP, demonstrating its scalability and memory efficiency. The availability of LAP on PyPI and its compatibility with the scikit-learn interface open doors for Python programmers to leverage this clustering technique in their data analysis workflows.

The next chapters in the thesis will review AP and introduce its lightweight variant LAP, along with the design and development of the Python package for the LAP algorithm. Next, the proposed LAP implementation in Python will be tested against other available implementations of LAP in other programming languages, mainly R, and it will also be compared with *scikit-learn*'s AP algorithm in order to ensure the proposed LAP implementation achieves similar clustering performance to AP while reducing peak memory consumption as the size of the dataset grows.

Chapter 2

Affinity Propagation (AP)

Affinity Propagation (AP) is a clustering algorithm that works by exchanging real-valued messages between data points, generating clusters based on similarities between data points [2]. It finds cluster centers, called exemplars, that are themselves points in the dataset. Furthermore, it is a technique that does not need to explicitly configure the number of clusters to identify, as it can automatically determine it from a configurable parameter (*preference*), which can also be used to make specific points more likely to end up being exemplars.

2.1 Algorithm

AP core is based on two key matrices. These two matrices are known as responsibility and availability matrices. The responsibility matrix R contains a quantifier of how well-suited a data point k is to serve as an exemplar of another data point i, relative to other candidates for i. In contrast, the availability matrix A explains how proper would it be for a data point i to take k as an exemplar, considering other data points' preference for k.

The responsibility updates depend on the similarity between data points, and it is what makes the algorithm extremely flexible, as it can be used with any similarity measure. Another important aspect of the similarity matrix is the self-similarity or *preference*. AP uses a similarity matrix, where the diagonal serves an important role: it controls how likely a data point is to be chosen as an exemplar. If the entirety of the diagonal is set to the same value, it then controls how many clusters the algorithm generates. Values closer to the minimum similarity will generate fewer clusters, while bigger values will generate many. In most implementations, the default value of the diagonal is the median similarity of all pairs.

As can be seen, AP has quadratic memory scaling, as the algorithm needs to hold several square matrices that contain the whole dataset. In the following sections, the algorithm will be detailed, allowing us to infer its runtime complexity to be quadratic based on its inner workings. That's why AP viability is reduced when working with large datasets or Big Data.

Initially, both matrices are initialized to zero. Afterward, R and A are updated iteratively until the algorithm converges. When updating the messages, it is important that they be damped to avoid numerical oscillations that arise in some circumstances. Each message is set to times its value from the previous iteration plus (1 – λ) times its prescribed updated value, where the damping factor λ is between 0

and 1 [2]. The damping value λ is a configurable parameter of the algorithm. This damping will not be shown in the equations that follow in this section.

Firstly, responsibilities are updated *per* equation 2.1.

$$r(i,k) \leftarrow s(i,k) - \max_{k' \neq k} \{a(i,k') + s(i,k')\} \tag{2.1}$$

where s represents the similarity matrix, a represents the availability matrix, and r represents the responsibility matrix. That means that the responsibility between two elements i and k is given by the similarity between i and k, minus the maximum of the sum of the availability and similarity of the data point i to other candidate exemplars (k') different from k.

Once the responsibilities are computed, the availability is calculated for elements different than the data point per equation 2.2.

$$a(i,k) = \min \left(0, r(k,k) + \sum_{i' \notin \{i,k\}} \max(0, r(i',k)) \right) \tag{2.2}$$

Availability for each data point to itself, also called self-availability is updated per 2.3

$$a(k,k) = \sum_{i' \neq k} \max(0, r(i',k)) \tag{2.3}$$

Data points are considered exemplars if the sum of their self-responsibility and self-availability is positive.

2.2 Usage

From its inception, AP has been successfully used in genetics and biomedical research. It continues to be used in tasks like clustering genome sequences [5], exemplar gene sets [6], or finding spatially connected and coherent regions of tumor-infiltrating lymphocytes image patches [7]. The has also been additional research on AP variations applied to specific areas like SAP [8] used for text clustering or a *MapReduce* version of AP [9], an approach with linear run-time complexity.

2.3 Available implementations

This section describes currently available implementations for several programming languages and their current state. There are more implementations available and published online, although they may not be as complete or have as much recognition as the ones selected. There are several standalone implementations that have not

been updated or received any attention for years, like [10] an implementation of AP for the JavaScript programming language.

2.3.1 apcluster

The R package *apcluster* [3] is the most complete implementation of AP currently available. It implements both AP and other derivations of it, including sparse versions and leveraged affinity propagation. It is freely available in the CRAN repository [11] licensed under the GPL2 and GPL3 open source licenses.

The sparse version has lower memory requirements when used against a dataset where the similarity matrix is sparse. However, the memory gains from using the sparse version are not always applicable.

2.3.2 matlab

The Probabilistic and Statistical Inference Group from the University of Toronto has published Matlab code for AP and some variants, including Leveraged Affinity Propagation. However, the main website for the tooling is down, and it may only be accessed by using internet archival tools, like the Internet Archive. It is also not part of a documented library *per* se. Only the FAQ on AP is available, and it may only be accessed using an older version of the site (`http://genes.toronto.edu/affinitypropagation/faq.html`).

2.3.3 scikit-learn

The scikit-learn [4] Python package implements AP in such a way that it is possible to work with both dense and sparse similarity matrices. The current version of scikit-learn does not implement leveraged affinity propagation and relies on auxiliary matrices during execution, further exacerbating AP memory constraints. Despite the usage of auxiliary matrices for the inner computations, the memory complexity of the algorithm remains quadratic.

The results can also be plotted easily thanks to how the library implements the clustering results. An example of the plots produced for both AP and LAP can be seen in Figure 2.1.

2.3.4 ELKI

There exists an implementation for the Java programming language in the ELKI toolkit [12]. It implements AP with support for dense matrices, and it belongs to a big machine-learning framework, much like the *scikit-learn* implementation.

2.3.5 Clustering.jl

Available for the Julia programming language, Clustering.jl [13] is a library that implements many different types of clustering algorithms, AP between them. It is an implementation of the classic algorithm described in the original paper, without

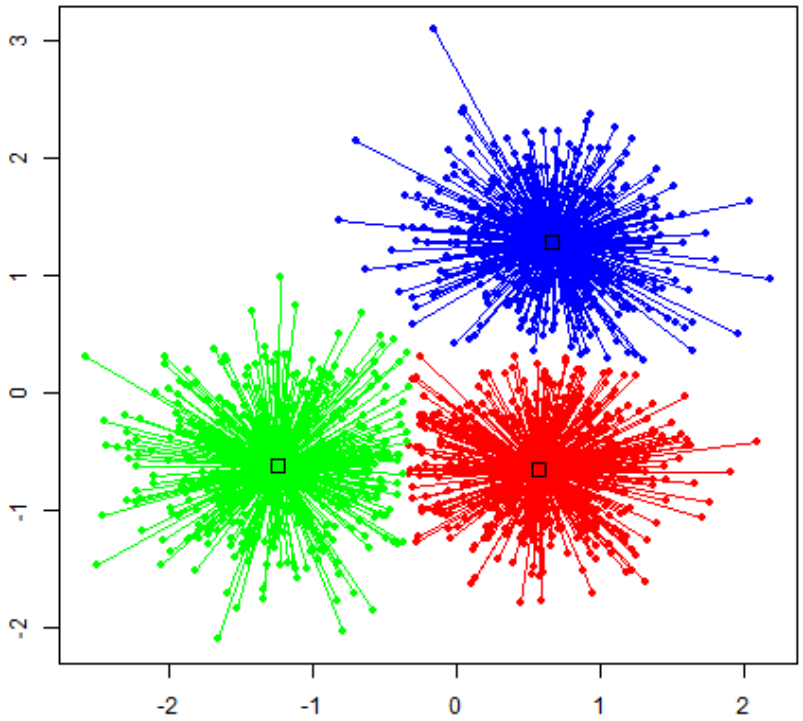


Figure 2.1: Example of clustering results produced by the apclusterL function, which implements Leveraged Affinity Propagation. Points in the figure represent data points of the original dataset, the squares are the centers or exemplars of each cluster generated by Leveraged Affinity Propagation, and the lines join each data point with its exemplar. A different color is used for every cluster found.

any variation or special considerations and functions like those from more complete libraries like *apcluster*.

2.4 Limitations

The primary memory-consuming factor in AP is the similarity matrix, which represents the pairwise similarities among data points. The size of this matrix is proportional to the square of the number of data points.

The memory complexity of AP can be inferred from the maximum space complexity of the data structures that are needed in order to run the algorithm. For AP this means storing at least the similarity matrix [2]. The space complexity of AP can be expressed mathematically as:

$$O(n^2) \tag{2.4}$$

where n represents the size of the input data. This means that the memory usage of the algorithm grows quadratically as the input size increases.

Additionally, AP maintains two other matrices during the iterative process: the responsibility matrix and the availability matrix. Both matrices have the same dimensions as the similarity matrix ($n \times n$). Therefore, they also contribute to the memory requirements, adding to the overall space complexity of the algorithm.

Furthermore, AP implementations need to keep track of the current exemplars for each element, generally in a vector of size n, and use some data structure to hold the necessary information to check for convergence. In *apcluster* and *scikit-learn* implementations of AP, this is achieved through an extra matrix, of size $n \times c$, where n represents the size of the input data, and c represents the number of iterations of AP to keep track of. In both libraries, this parameter is referred to as the convergence iterations and is configurable by the user. In the *apcluster* package it defaults to 100 iterations [3] while in *scikit-learn* it defaults to 15 [4].

The minimum memory requirements of the most common implementations of AP can then be given by:

$$\text{Memory (in bytes)} = 3n^2 w_1 + (n + nc)w_2 \tag{2.5}$$

where n is the size of the input data, c is the number of iterations to check for convergence and w_1 represents the number of bytes needed for each element in the similarity, responsibility, and availability matrices, most often 4 or 8 bytes, used for single and double precision floating points and w_2 the number of bytes used for array indexing, which is an integer type that depends on the platform that the code is executing in.

Chapter 3

Leveraged Affinity Propagation

3.1 Introduction

Leveraged Affinity Propagation (LAP) is a variant of the Affinity Propagation (AP) clustering algorithm that incorporates a subsampling technique during the exemplar selection process.

By utilizing a smaller subset of the data, LAP reduces the computational complexity and memory requirements of the AP algorithm. This subsampling step allows LAP to handle larger datasets that might be impractical to process with the original AP algorithm.

The key idea behind LAP is that the exemplars selected from the leverage set should be representative of the overall data distribution, enabling efficient and effective clustering. LAP aims to strike a balance between computational efficiency and preserving the clustering quality.

Leveraged Affinity Propagation is a useful approach when dealing with large datasets where the computational requirements of the original AP algorithm are prohibitive, making it easier to uncover meaningful patterns and extract valuable knowledge from even the most massive datasets. By leveraging a subsample of the data, LAP offers a scalable solution for clustering tasks while still capturing the essential patterns and structure within the dataset.

3.2 Available implementations

3.2.1 apcluster

As described in previous sections, the *apcluster* R package [3] is one of the most complete implementations of Affinity Propagation algorithms, and it does not fail to integrate Leveraged Affinity Propagation on its repertoire of functions. It has all the expected functionality of LAP, with a lot of versatility in the implementation, for example, allowing precomputed subsamples of the similarity matrix for a given selection if one wishes to run the algorithm step by step.

3.2.2 matlab

The Probabilistic and Statistical Inference Group from the University of Toronto has published MATLAB code for Affinity Propagation and some variants, including

Leveraged Affinity Propagation. However, the main website for the tooling is partially down, and it may only be accessed by using internet archival tools, like the Internet Archive. It is also not part of a documented library *per* se. Only the FAQ on Affinity Propagation is currently available, and it may only be accessed using an older version of the site. For example, while `https://psi.toronto.edu/research/affinity-propagation-clustering-by-message-passing/` is accessible, most of the links included in the page link to URLs that give 404 errors.

3.3 Implementation

The fundamentals of Affinity Propagation are still present in LAP, it keeps the message passing between data points and the concepts of responsibility and availability. However, rather than computing the responsibility and availability between all data points, only a subsample of the whole is considered as potential exemplars.

With a sufficiently large dataset, a random subselection of data points should have enough information to generate appropriate clusters for most entries, only losing significant performance against AP for very small outlier clusters in the dataset where none of the data points were selected as candidates. However, if these outliers are detected, it would be possible to run Leveraged Affinity Propagation with a manual selection of candidates to ensure that at least one data point for the outlier group is considered as a candidate exemplar.

The number of elements that are considered candidate exemplars is defined by the *fraction* parameter, which stands for the fraction of the total number of points n to be subsampled. The selection will have a total of αn candidates. Where α is the value of *fraction*, which must always take a value in the range $(0, 1)$.

Another parameter available to avoid the problem of not having good enough candidates in the selected subsample is *sweeps*. It is used to increase the number of runs or sweeps that LAP makes over the dataset. In each sweep new candidate exemplars are considered, and if the resulting clusters from running the algorithm have greater net similarity than the previous sweep they are added to the new selection. The selection for the new sweep is then the old exemplars and enough randomly selected points to fill the selection until it reaches αn candidates.

LAP always performs the number of sweeps (k) that it is configured to execute, regardless of whether the exemplars remain unchanged after each sweep or not.

3.3.1 Performance

3.3.1.1 Clustering

One key issue that may arise on LAP and is not present in AP is that the number of clusters that the algorithm can find is limited to the size of the subsample. LAP will not find more clusters than the number of candidates that exist in a single sweep. What is more, the number of data points to subsample should be higher than that of the possible desired clusters.

This limitation arises from the previously explained candidate selection mechanism. Since all the resulting exemplars from each sweep are included in the new selection, whenever all the subsample candidates are selected as exemplars no new candidates will be explored on the new sweep.

3.3.1.2 Memory and runtime considerations

As explained in the implementation section, instead of requiring multiple $n \times n$ matrices, LAP uses $n \times \alpha n$ matrices. The space complexity of LAP is then:

$$O(\alpha n^2) \tag{3.1}$$

In the AP section, a minimum memory requirement of AP was given, the adapted formula for LAP is:

$$\text{Memory (in bytes)} = 3\alpha n^2 w_1 + (n + nc)w_2 \tag{3.2}$$

where c is the number of iterations to check for convergence and w_1 represents the number of bytes needed for each element in the similarity, responsibility, and availability matrices, most often 4 or 8 bytes, used for single and double precision floating points and w_2 the number of bytes used for array indexing, which is an integer type that depends on the platform that the code is executing in.

This makes the performance of LAP, particularly for large datasets, much better when it comes to both peak memory usage and might result in less computing time. The memory gains over AP are configurable by the user and are always expressed as α of the total memory consumption of AP.

However, the compute time gains may be lost if the *sweeps* (k) and *fraction* (α) parameters are too high, as the total number of data points evaluated could exceed those of AP for combinations of *sweeps* and *fraction*.

Empirically (see Appendix A), a good estimation of the upper bound for the time taken by LAP in different configurations 1 and 2 can be given by:

$$t_2 = t_1 \times \frac{\alpha_2}{\alpha_1} \times \frac{k_2}{k_1} \tag{3.3}$$

where t_x, α_x and k_x indicate the time taken by configuration x, with parameters *fraction*=α and *sweeps*=k.

The peak memory usage gains are better than the compute time, as it does not depend on the *sweeps*. There exist variations of LAP that attempt to mitigate the algorithm's runtime, for example, using a parallel ensemble variant of LAP [14].

Chapter 4

Python package

4.1 Requirements

This section lists high-level requirements for different areas that the package should cover.

4.1.1 Functional requirements

Algorithmic implementation of LAP, offering both the complete algorithm and the inner affinity propagation functions. Exposing the inner affinity propagation functions allows the users with niche use cases, like very specific candidate exemplar selection, to still have a use for the library using its inner functions.

4.1.2 Non-functional requirements

Compatibility with *scikit-learn*

The Python package should be designed in a way that it seamlessly integrates and is compatible with the scikit-learn library. It should follow the conventions of the scikit-learn API, ensuring good interoperability. This compatibility will allow users to easily incorporate the package into their existing scikit-learn workflows and pipelines without encountering any conflicts or compatibility issues.

Packaging

The package should have the least dependencies possible while retaining proper performance characteristics.

PyPI (Python Package Index) is an integral component of the Python ecosystem and plays a crucial role in facilitating the distribution, discovery, and installation of software packages. As a centralized repository, PyPI serves as a hub where developers can publish their packages, making them accessible to the Python community worldwide. The resulting package should be available to install from PyPI.

Documentation

The Python package should provide comprehensive documentation that includes clear explanations of its functionality, instructions on how to use it, and code examples. The documentation should cover the package's compatibility with scikit-learn, demonstrating how it can be used alongside scikit-learn's pipelines. Furthermore,

the package should provide practical examples and use cases to illustrate its capabilities, helping users understand how it can be integrated effectively with scikit-learn.

4.2 Design

Math-heavy Python libraries, such as those used for numerical computations, simulations, or scientific calculations, often involve repetitive mathematical operations that can benefit from the performance enhancements provided by low-level routines that use optimized machine code rather than the Python interpreter. This is the case of affinity propagation.

There are several options to write performant code in Python, including but not limited to:

- *NumPy* routines, which are implemented mainly in heavily optimized C code under the hood, and are completely transparent to the programmer.
- *numba* methods, which use a Just In Time (JIT) compiler to optimize the code at runtime.
- C or C++ extensions, which offer the most flexibility, but require writing Python extension module code, which can become a complex task.
- Cython, which is a programming language that combines the ease and simplicity of Python with the speed and efficiency of C, simplifying the creation process of Python modules and offering extra flexibility with Python types.

NumPy offers many methods that would help to write performant code, but it falls short of extension modules (C, C++, Cython ...) when there is no matching operation defined that covers a needed use case. The next best option that allows for performant code for arbitrary iteration over matrices is Cython, because while *numba* may be easier to write code for, it is still JIT compiled, with all the drawbacks that it may bring, like the need to perform a function call several times so that it gets properly optimized.

Cython also natively supports NumPy arrays as it can directly access and manipulate NumPy arrays without any significant overhead. This allows for efficient data processing and computation, leveraging the optimized routines provided by NumPy.

By writing critical sections of code in Cython, the library can harness the power of low-level optimizations and achieve faster execution times. Furthermore, memory management is explicit in Cython, allowing much finer control over allocations and memory usage.

The advantage of using Cython over C or C++ directly is the ease of use when it comes to later calling the code from Python, and integration with libraries that would be used, mainly *NumPy*. If the extension code was written in the aforementioned languages, the glue code to call it from Python would have been developed in Cython.

Using any solution that relies on compiled extensions can present certain challenges due to the nature of the language and the need to compile the code, and Cython is no exception. However, if pre-compiled binaries are provided, these hurdles do not significantly affect users. By offering pre-compiled binaries developers can save consumers from compiling the Cython code themselves. This ensures that users can simply install the package using the provided binary distribution, regardless of their system configuration. Said binary distribution can be hosted on PyPI, the Python Package Index.

4.3 Development

First, the package implements all the components of LAP excluding the inner affinity propagation function in Python. These components are what makes the library compatible with the *scikit-learn* library, via the implementation of a Python class LeveragedAffinityPropagation that extends *scikit-learn BaseEstimator* and following naming conventions for the attributes and functions available.

The similarity calculation, the sampling process for the exemplar candidate selection, and the final clusters processing are implemented in Python, using *NumPy* when applicable.

Regarding the inner affinity propagation function, the core availability and responsibility updates were implemented as a Cython function. This Cython function is compiled by Cython during the build process to C++ code that includes Python glue, later compiled by the platform's C++ compiler.

The resulting code was benchmarked using *line_profiler* to check that most of the time spent was running the compiled Cython code. In Figure 4.1 the overall LAP function that runs selection mechanisms and runs new sweeps spends most of its time in the *affinity_propagation* function.

```
5         27.0      5.4      0.0          (
5         25.0      5.0      0.0              cluster_centers,
5         25.0      5.0      0.0              labels,
5         26.0      5.2      0.0              dpsim,
5         26.0      5.2      0.0              expref,
5         22.0      4.4      0.0              net_similarity,
5         23.0      4.6      0.0              pv,
5         23.0      4.6      0.0              it,
5   79946855.0 15989371.0     99.9            ) = affinity_propagation(
5         20.0      4.0      0.0              S,
5         20.0      4.0      0.0              sel_idx,
5         18.0      3.6      0.0              convergence_iter=conv_iter,
5         17.0      3.4      0.0              p=p,
5         18.0      3.6      0.0              q=q,
5         20.0      4.0      0.0              max_iter=max_iter,
5         19.0      3.8      0.0              damping=damping,
5         19.0      3.8      0.0              copy=False,
5         19.0      3.8      0.0              verbose=True,
5         25.0      5.0      0.0              random_state=random_state,
                                           )
5         46.0      9.2      0.0          sweep_net_similarity[i] = net_similarity
```

Figure 4.1: Detail of the line_profiler for the *lap* function. From left to right the columns indicate the number of lines in the code: number of hits or times the line was executed, total time taken in a line, time taken per hit, and percentage of total run-time. The calls to the *affinity_propagation* function take 99.9% of the total time spent in the *lap* function.

In turn, as shown in Figure 4.2, it is shown how the *affinity_propagation* function spends most of its time on the *inner_propagation* call which is the Cython function, which compiles to heavily optimized machine code.

```
164       5       596.0    119.2      0.0      e = np.zeros((n_samples, convergence_iter), dtype=int, order="C")
165       5       583.0    116.6      0.0      I = np.full(n_samples, -1, dtype=int, order="C")
166
167       5   79412584.0 15882516.8     99.3      K, it, never_converged = inner_propagation(
168       5        30.0      6.0      0.0          S, sel, damping, n_samples, m, e, I, max_iter, convergence_iter
169                                             )
170
171       5        82.0     16.4      0.0      if K > 0:
172       5        31.0      6.2      0.0          if never_converged:
```

Figure 4.2: Detail of the *line_profiler* for the *affinity_propagation* function. From left to right the columns indicate the number of lines in the code: number of hits or times the line was executed, total time taken in a line, time taken per hit, and percentage of total run-time.

4.4 Packaging

The Python code and the Cython extension module are packaged using *setuptools* and using a modern, declarative approach with the *pyproject.toml* file, which covers package metadata, build system and dependencies, and core project dependencies.

4.4.1 Installation

Installation of the package from source requires calling *pip install*. The installation should be straightforward, if problems during installation arise, particularly due to compilation, it is most likely to be either the lack of Cython in the current environment, even if it is listed as a build dependency, or the lack of a C++ compiler.

Installation from PyPI should be even easier, just running *pip install* in an environment supported by the main dependencies *NumPy* and *scikit-learn* should suffice.

4.5 Challenges

4.5.1 Algorithm implementation

An initial "pure" Python version of the algorithm was developed, though working with *NumPy* required the use of additional auxiliary matrices. As this was against the spirit of the project, which aimed to implement a leveraged variant of AP precisely to reduce memory usage, and using loops in pure Python proved to be extraordinarily slow, Cython was introduced to optimize both memory usage and time used by the algorithm.

4.5.2 Platform agnostic indexing integers

Writing Cython code that compiles and interoperates with Python in a platform-agnostic manner proved to be a challenge, in particular, minor differences between integer widths between Python and Cython for Linux and Windows were detected, which required an investigation on which types were adequate for the supporting index vector and matrix that LAP were using, as the "int" Python type was not sufficient.

The solution involved creating the index vector and matrix using *numpy* integer pointer types in Python and using the special Cython type "Py_ssize_t" on the Cython function signature.

4.5.3 Packaging

Initially, the package used modern tooling for its dependency management and overall building, namely Poetry (`https://python-poetry.org/`). Unfortunately, the inclusion of Cython and compiled code meant that most of the functionality needed for the build system and specification was not as seamless as it could have been for a pure Python package under the same scenario. The initial build system with Poetry and modern Python Enhancement Proposals (PEP) like PEP621 and PEP631 was scrapped. In the end, a traditional *setuptools* setup and installation was implemented, attempting to keep as much of the modern solution as possible, inspired by other big libraries that use the same tooling and have similar requirements and library usage, such as *scikit-learn*.

This means that the declarative style packaging was kept, with the addition of the *pyproject.toml* file, rather than using only the tradition *setup.py* with configuration options spread out, most likely between the build script and a *setup.cfg* file.

Chapter 5

Validation

5.1 Design

5.1.1 Hardware

The benchmarking machine is equipped with an AMD processor, specifically the R7 1700 model. The AMD R7 1700 is a high-performance processor known for its multi-threading capabilities, offering 8 cores (16 threads with multithreading) allowing for efficient parallel processing. However, it does not offer the best single-core performance. The machine is equipped with 32GB of RAM, providing enough memory to test Affinity Propagation algorithms, as one of the objectives of the benchmarking is to test when AP runs out of memory and how LAP can avoid it when using limited resources.

5.1.2 Software

The benchmarking machine operates on the Windows 10 operating system. The primary programming language used for testing and benchmarking on this machine is Python 3.9, though the inner workings of the proposed LAP algorithm are written in Cython and compiled into C++.

Python Jupyter Notebooks were also used, particularly during the initial validation phase and they were also used for dataset exploration and cleanup tasks. Jupyter notebooks are widely used in the field of machine learning. They offer a flexible and interactive environment for data exploration, model development, and algorithm evaluation. However, this interactivity means they are not suited for accurate benchmarking tasks due to the overhead that they could introduce. The benchmarks implemented in the package use *scikit-learn* common base classes as a base to build the AP and LAP benchmarks.

In order to have more robust benchmarks associated with the Python package, *asv-benchmark* was introduced. The *asv-benchmark* (Airspeed Velocity Benchmark) Python package is a tool designed to perform benchmarking and performance tracking of code across different versions, configurations, and machines. It sees use in many of the big Python machine learning and mathematical packages, including most of the biggest libraries used in data science: *numpy*, *scipy*, *pandas* and *scikit-learn*.

The main objective of *asv-benchmark* is to measure and compare the execution time and resource utilization of different implementations. It allows developers to define

a benchmarking suite, which consists of specific code snippets or functions to be evaluated. These benchmarks offer a lot of flexibility in their configuration.

In the context of machine learning, and more clustering, the *asv-benchmark* package is valuable for assessing the performance and efficiency of algorithms and models. It can be used to evaluate the execution time and memory usage of different machine learning implementations, compare the performance of various libraries or frameworks, and track performance improvements or regressions over time and commits. Introducing these benchmarks promotes the development of high-performance clustering algorithms by providing a systematic and quantitative approach to benchmarking and performance evaluation.

5.1.3 Metrics

Comparing clustering results requires metrics to objectively evaluate the quality and performance of different clustering techniques or variations. In our case, we want to explore LAP clustering performance and compare it with AP.

Employing appropriate metrics for comparing variations of the same clustering algorithm across multiple datasets is crucial for an unbiased evaluation and effective selection of the most suitable variation. This is even more pronounced in AP and LAP, which are not deterministic and rely on random state seeds in order to choose the selected items to be evaluated as exemplars. These metrics enable us to objectively assess the performance of each algorithm variation, including different LAP parameters, and understand their adaptability to different datasets.

- Homogeneity Score: Measures the extent to which each cluster contains only samples from a single class. A score of 1.0 indicates perfect homogeneity.
- V-measure Score: Combines homogeneity and completeness scores into a single metric. It provides a balanced evaluation of clustering results. A score of 1.0 indicates the best clustering result.
- Silhouette Coefficient: Measures how close each sample in one cluster is to samples in neighboring clusters. A higher coefficient indicates better-defined clusters. The range is from -1 to 1, where values close to 1 indicate well-separated clusters.
- Mutual information Score: Measures the similarity between two labels of the same data. It quantifies the shared information and dependence between the sets, indicating how well the clusters align with the true class labels.
- Rand Index: Measures the agreement between the clustering result and the ground truth labels. It calculates the proportion of pairs of data points that are either correctly assigned to the same cluster or correctly assigned to different clusters, relative to all possible pairs. It ranges from 0 to 1, with 1 indicating a perfect match between the clustering and the ground truth. It does not ensure obtaining a value close to 0.0 for random labeling, though its Adjusted version (ARI) does.

- Fowlkes-Mallows: Geometric mean of the pairwise precision and recall. The score ranges from 0 to 1, and higher values indicate good similarity between two clusters. Random label assignments have values closer to 0 regardless of sample size or number of clusters.

5.1.4 LAP Parameters

LAP has two extremely important parameters that dictate performance characteristics from a memory and computing point of view: fraction and sweeps. Fraction controls the percentage of the total dataset that is used as exemplar candidates while sweeps controls how many times does the algorithm run in sequence.

Differences in these two parameters should also be reflected in the quality of the produced clustering for small or imbalanced datasets. As the fraction and sweeps increase, LAP should explore more possible exemplars. If the fraction is too small for the size of the dataset or there is an extreme class imbalance, LAP may never evaluate an exemplar of a certain combination of classes or characteristics, reducing clustering performance.

Due to these reasons, each test is performed for different fraction and sweeps values. It should be noted that the fraction parameter ultimately controls the memory advantage of LAP over AP, along with the maximum number of clusters possible. Moreover, the exemplar candidate selection process is random, so each run is performed multiple times using different random seeds.

As explained in previous sections, an important parameter that controls the number of clusters generated by both AP and LAP is the *preference*, which unless it is explicitly mentioned will be set to the library default, most often the median of the similarities.

Note that for many figures presented in the following sections, references to LAP parameters will be given in the form of α*k. For example, a figure or table that indicates LAP 0.1*5 means that LAP was configured to use α=*fraction*=0.1 and k=*sweeps*=5.

5.2 Datasets

In order to test the proposed Leveraged Affinity Propagation implementation, several datasets, both synthetic and real-world, will be used to assess LAP performance. Both memory and compute time will be measured, along with clustering performance in datasets where it is possible. The proposed implementation will also be tested against AP and R apclusterL.

5.2.1 Synthetic

Some kind of synthetic dataset had to be chosen due to the need for an easily configurable and adaptable dataset. The synthetic datasets used have a known working similarity measure which can be computed fast and can be interpreted and

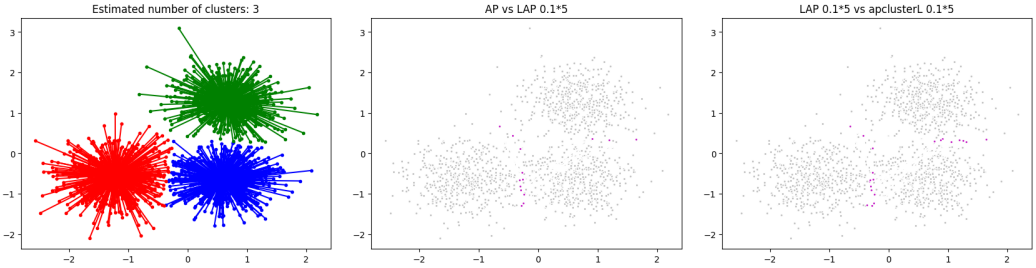
visualized easily. With this in mind, two-dimensional datasets with a predefined number of clusters in different configurations were chosen to test LAP against.

These datasets can be generated using utilities provided by standard data science Python tooling and are present in many documentation examples, making result interpretation much more accessible.

Most tests performed in this section use variations of the *blobs* dataset. The blobs datasets are generated using *scitkit-learn make_blobs* function, which generates isotropic Gaussian blobs used for clustering. Isotropic means that the covariance matrix of the features is diagonal, meaning all the features are uncorrelated.

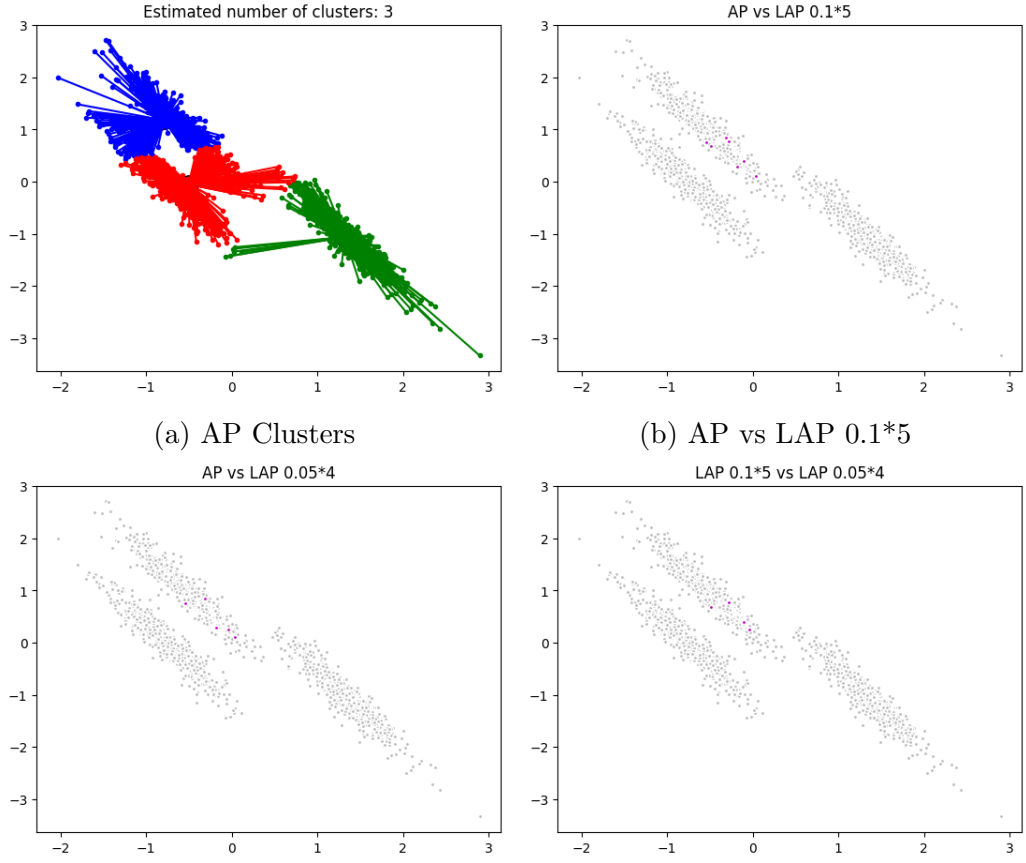
Initially, the proposed LAP implementation and scikit-learn's AP were compared against the R package apcluster implementation: apclusterL. The results of the clustering indicated very similar clusters generated by all algorithms and implementations in the toy synthetic datasets. The results are shown in Figures 5.1 and 5.2.

Most of the work in this section used Jupyter Notebooks along with the *rpy2* Python package [15], used to call R code from Python and share data.



(a) AP Clusters (b) AP vs LAP 0.1*5 (c) 0.1*5: Python vs R

Figure 5.1: Graphical comparison between scikit-learn Affinity Propagation, apclusterL Leveraged Affinity Propagation implementation, and the proposed implementation. The dataset is a blobs dataset with 1500 points and three clusters. Purple points indicate disagreement in the clustering group. All affinity propagation implementations used -200 as preference and 0.9 as damping.



(a) AP Clusters

(b) AP vs LAP 0.1*5

(c) AP vs LAP 0.05*4

(d) LAP 0.1*5 vs LAP 0.05*4

Figure 5.2: Graphical comparison between scikit-learn Affinity Propagation and the Leveraged Affinity Propagation implementation in two different configurations. The dataset is a rotated blobs dataset with 1500 points and three clusters. Purple points indicate disagreement in the clustering group. All affinity propagation implementations used -200 as preference and 0.9 as damping.

Afterward, LAP is tested against additional synthetic datasets. Before delving into the specifics of LAP, it is useful to know how different clustering algorithms perform on the same dataset and compare them.

For this purpose, the clustering example available on the scikit-learn Python package was replicated with the inclusion of multiple configurations of Python LAP. The results can be seen in Figure 5.3. This first introduction to LAP can already show the advantages in processing time between algorithms, and a preview of clustering performance for simple scenarios, even if it is only graphically without employing any robust metric. As shown in Figure 5.3 LAP takes less time to execute for all datasets when compared to AP, offering results in times closer to other alternatives. However, even only the faster LAP configurations can offer results in a comparable time to other algorithms, though it is still far ahead of AP, which takes at least an order of magnitude more than the rest to finish its execution.

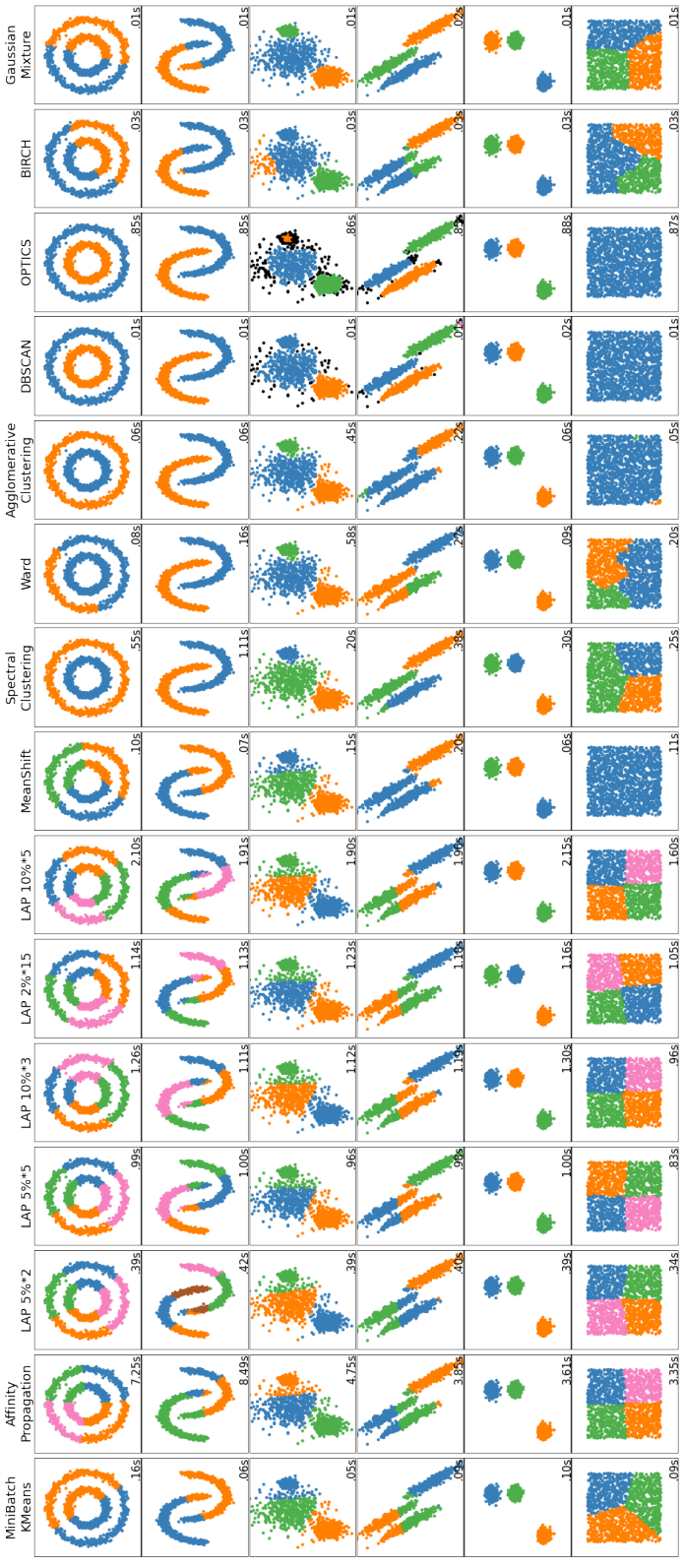


Figure 5.3: Comparison of clustering algorithms with Leveraged Affinity Propagation in different configurations. The comparison is that of [4] clustering documentation extended with the LAP runs. It also includes the time taken by the different algorithms for each dataset in the bottom right corner of each image in seconds. The percentage indicates the fraction parameter and it is multiplied by the sweeps parameter.

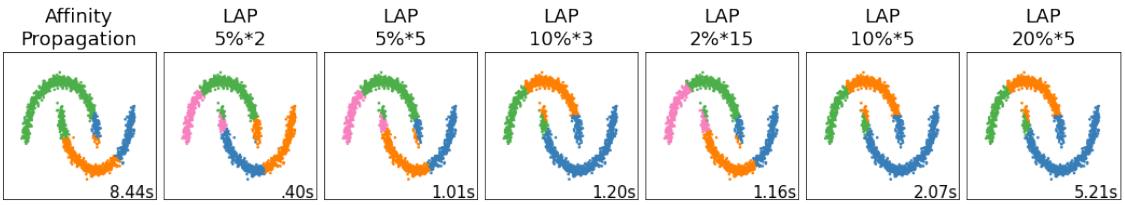
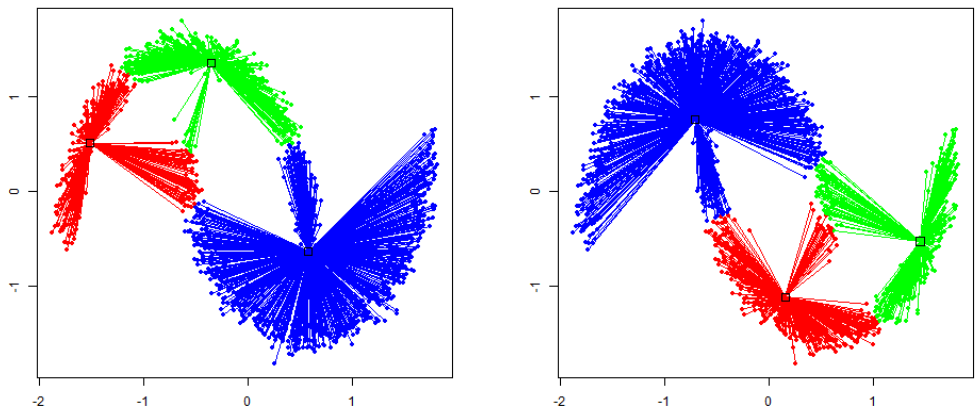


Figure 5.4: Detail of the comparison between AP with Leveraged Affinity Propagation in different configurations to achieve the same number of clusters. The comparison is that of scikit-learn [4] clustering documentation extended with the LAP runs. The percentage indicates the fraction parameter and it is multiplied by the sweeps parameter.

There is another difference that can be appreciated in the figure. LAP generates more clusters in the noisy moons dataset. Lower preference values for LAP and higher sweeps and fractions allowed LAP to also achieve three clusters, as seen in Figure 5.4. The preference values for the LAP runs were the ones used for the tests in Figure 5.3, -220 for all LAP configurations, reduced by 100 except for the LAP 20% * 5 configuration, which reduced the original preference by 90.

Figure 5.5 shows the clustering results over the same noisy moons dataset using the apclusterL R function. Different preferences were used, and it shows in the "mirrored" clusters how it can affect behavior along with the random seed.



(a) preference=−340 seed=10 (b) preference=−320 seed=0

Figure 5.5: R apclusterL over the noisy moons dataset, using 5% as fraction and 6 sweeps, with 0.9 as damping factor.

Now, AP performance will be compared to that of the proposed LAP implementation using the *asv-benchmark* framework. The dataset is a five-cluster, ten features blobs dataset generated using *scikit-learn* dataset functions, with varying sample sizes.

First, the results from the AP benchmarks to be used as a baseline are available in

Table 5.1. Here we can already see the jump in memory consumption from $n = 2000$ and $n = 5000$, where the peak memory consumption more than triples. This increase is even more pronounced when taking into consideration the fact that $n = 10$ has a peak memory consumption of 91MB.

The $n = 10$ scenario is included specifically to understand what could be the baseline memory usage of the benchmarking tooling and the Python process itself. Taking this basic minimum consumption for Python and the benchmarking tools, $n = 1000$ consumes 22MB, $n = 2000$ consumes 88MB and $n = 5000$ consumes 540MB. This is expected, as AP space complexity is quadratic. It is not by chance that a doubling of dataset size results in a 2^2 increase in memory ($22 \times 2^2 = 88$), or that a five-fold increase results in a 5^2 increase in memory consumption ($22 \times 5^2 = 550$).

n_samples	time	peakmem	memory	silhouette	homogeneity	AMI
10	0.0054	91.07MB	0MB	0.6975	0.8658	0.8288
1000	0.4104	113.27MB	22MB	0.7412	1.0000	1.0000
2000	3.2570	178.66MB	88MB	0.0721	1.0000	0.8279
5000	30.3699	634.10MB	443MB	0.0552	1.0000	0.7022

Table 5.1: Synthetic baseline AP results. memory refers to the approximate memory consumed by AP alone, excluding the base consumption of the Python process and benchmarking software. At $n = 10$ the impact of AP on peak memory consumption should be approximately 0MB, this can be tested using Equation 2.5: we can expect AP to consume $3 \times 10^2 \times 4 + (10 + 10 \times 15) \times 4 = 1840\text{B} \approx 1.8\text{KB}$.

These results can also be seen graphically in Figure 5.6 for the peak memory consumption, and Figure 5.7 for the time taken by AP. Note that for Figure 5.7 the y-axis is logarithmic.

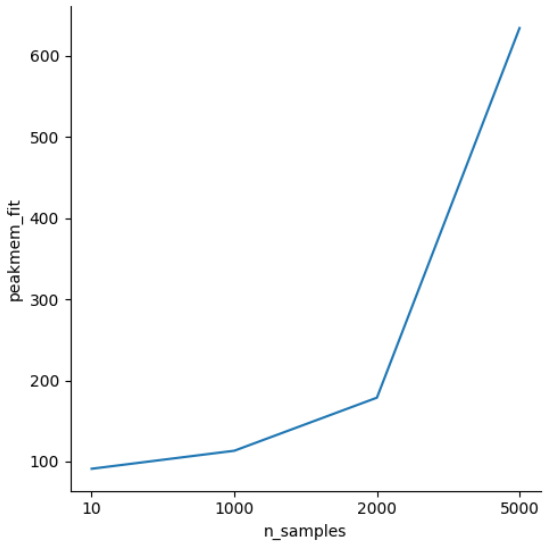


Figure 5.6: Synthetic baseline AP peak memory usage

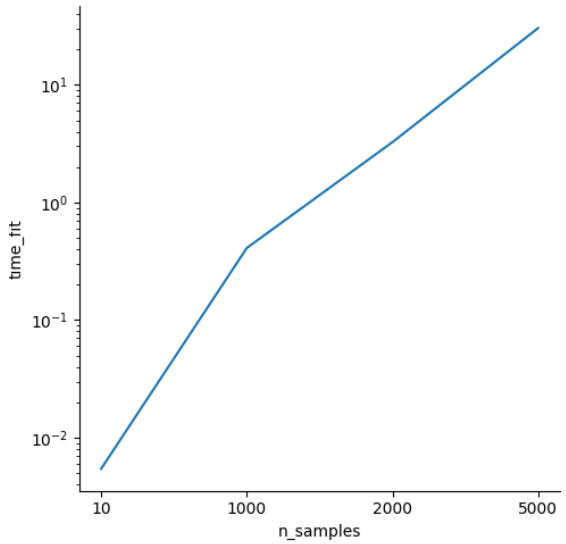
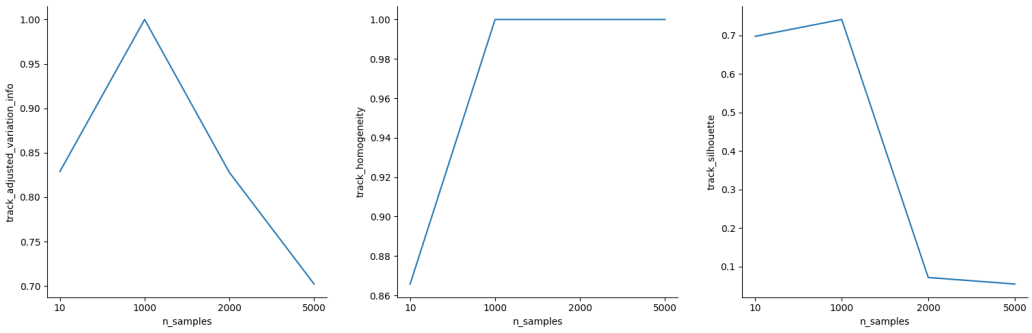


Figure 5.7: Synthetic Baseline AP time taken for different configurations. Note the logarithmic scale on the y-axis

The clustering performance of AP in this five-cluster blobs dataset is evaluated using the Adjusted Mutual Information (AMI), homogeneity, and silhouette scores. The results are shown in Figure 5.8. Notice the decrease in both AMI and silhouette scores when n increases. Both of these metrics penalize clusterings where not all of the data points of a single class are in the same cluster. This observation, coupled with a homogeneity score of 1 is a clear indicator that AP is generating many clusters that contain only data points from one of the five true clusters. Better AMI and silhouette scores could be achieved by reducing the *preference* parameter, which would in turn decrease the number of generated clusters, reducing the penalization for the scores.

α	n	time	peakmemory	memory	silhouette	homo	AMI
0.005	10	0.0023	91.11MB	0MB	0.5006	0.3350	0.3131
0.005	1000	0.0140	91.68MB	0.5MB	0.3744	0.5867	0.7337
0.005	2000	0.0449	92.65MB	1.5MB	0.4668	1.0000	0.9253
0.005	5000	0.1836	96.12MB	5MB	0.0514	1.0000	0.7542
0.1	10	0.0023	90.74MB	0MB	0.5006	0.3350	0.3131
0.1	1000	0.1447	94.91MB	3MB	0.6112	1.0000	0.9595
0.1	2000	0.4786	104.15MB	13MB	0.0666	1.0000	0.8240
0.1	5000	3.8689	167.25MB	76MB	0.0546	1.0000	0.7012

Table 5.2: Synthetic LAP result excerpt. All LAP runs use *sweeps* $= k = 1$. memory refers to the approximate memory consumed by LAP alone, excluding the base consumption of the Python process and benchmarking software. At $n = 10$ the impact of AP on peak memory consumption should be approximately 0MB, this can be tested using Equation 3.2: we can expect AP to consume $3 \times \alpha \times 10^2 \times 4 + (10 + 10 \times 15) \times 4 = 184\text{B} \approx 0.18\text{KB}$.



(a) AMI (b) homogeneity (c) silhouette

Figure 5.8: Synthetic baseline AP clustering performance

Then, the results from LAP clustering and runtime performance for the synthetic dataset in the *asv-benchmark* are shown fully in A.1, an excerpt with relevant scenarios is included in Table 5.2. As it can be seen in the table, LAP, particularly in higher *fraction* (α) configurations, behaves extremely similar to AP, obtaining 0.8240 AMI score to AP's 0.8279 for $n = 2000$ and AMI score of 0.7012 to AP's 0.7022.

Memory consumption of LAP is clearly lower than AP, as shown in Figure 5.9. The Figure includes a polynomial regression of order two and different configurations of *fraction* (α) for LAP. Runtime performance is also better for most all combinations of α and k as shown in Figure 5.10.

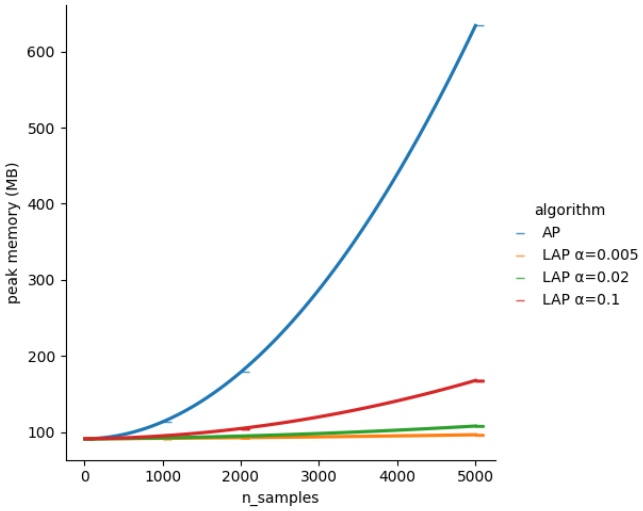


Figure 5.9: AP vs LAP memory consumption in the asv benchmarks

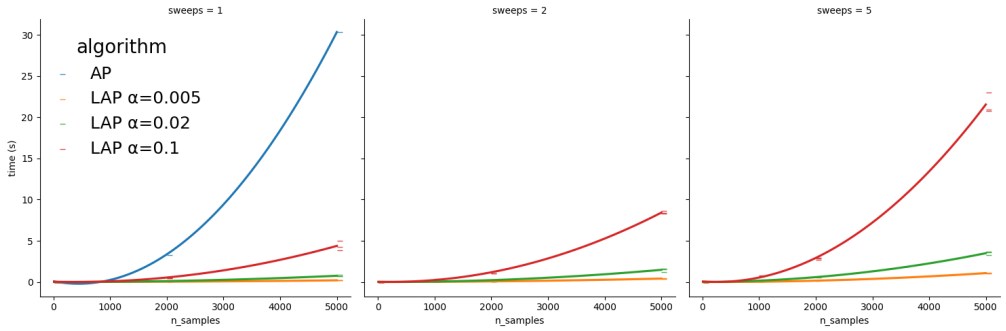


Figure 5.10: AP vs LAP time in the asv benchmarks

The behavior of LAP peak memory usage, time, and clustering performance was also studied for different LAP parameters in this synthetic asv benchmark.

Figure 5.11 shows LAP peak memory consumption for different parameters. As can be seen in the graphs, the number of sweeps that LAP makes over a dataset to refine the clustering results does not have an impact on the peak memory consumption. However, the time taken by LAP does depend on the number of sweeps as shown in Figure 5.12.

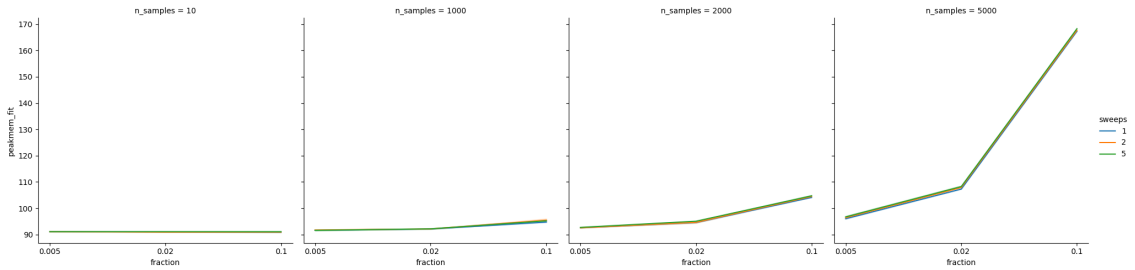


Figure 5.11: Synthetic LAP peak memory usage for different configurations

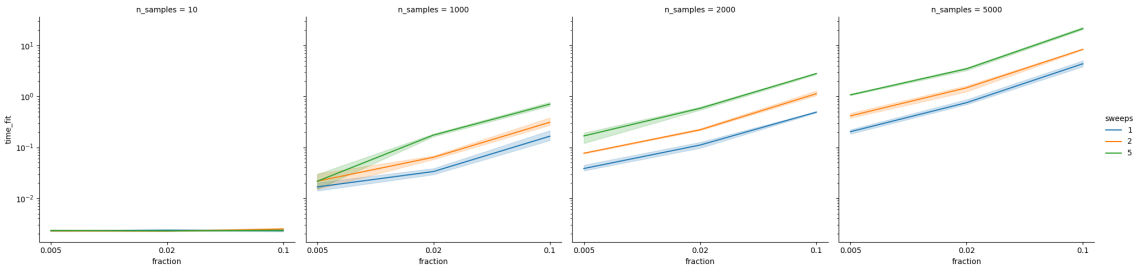


Figure 5.12: Synthetic LAP time taken for different configurations. Note the logarithmic scale on the y-axis.

Next, the clustering performance of LAP in relation to its parameters is evaluated. Do note that these results are not generalizable for all datasets, as it is likely that datasets with more than the five true underlying clusters of the benchmarked dataset would benefit more from the additional exploration from a greater amount of sweeps. Figure 5.15 shows silhouette scores close to zero for $n > 1000$. These scores combined with extremely high homogeneity as can be seen in Figure 5.14 indicate overlapping clusters. Figure 5.13 includes graphs that show the AMI for multiple configurations of LAP. Here we can see how increases in the *sweeps* parameter cause a very slight increase in clustering performance.

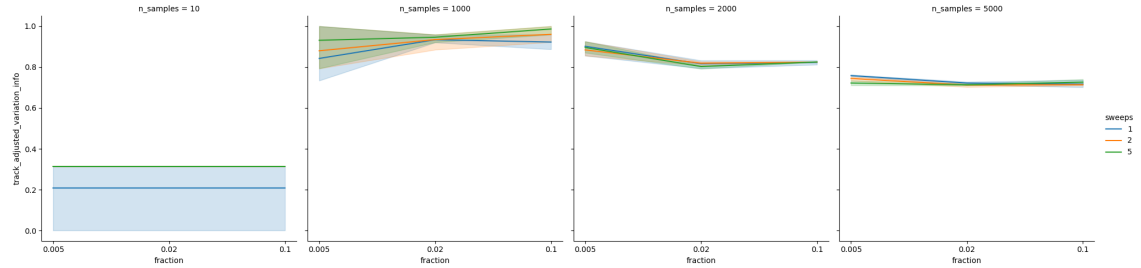


Figure 5.13: Synthetic LAP peak adjusted mutual information for different configurations

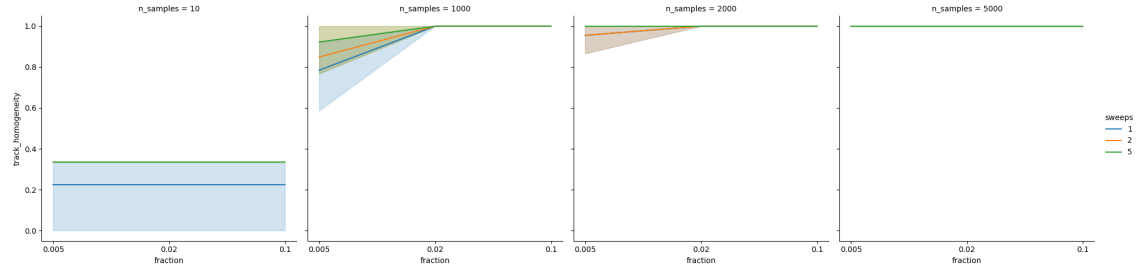


Figure 5.14: Synthetic LAP homogeneity for different configurations

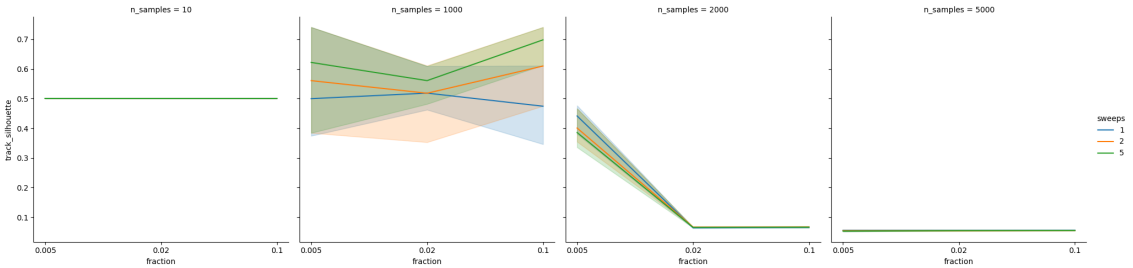


Figure 5.15: Synthetic LAP silhouette score for different configurations

5.2.2 Olivetti Faces

The Olivetti Faces dataset is a widely used benchmark dataset in the field of computer vision and facial recognition. It consists of a collection of grayscale facial images captured under controlled conditions. The dataset was created by AT&T Laboratories Cambridge and contains face images of 40 different subjects, each with 10 different images.

It was included in the benchmarks due to it being one of the original datasets showcased in the original Affinity Propagation paper [2], though they used the dataset to build its own using only the first 100 pictures and then applied transformations. This section covers the original 400 Olivetti faces dataset, rather than the modified version from the paper.

The asv benchmarks were executed for the whole dataset of 400 images, rather than a subsample, damping values of 0.5 and 0.8 were tested for the baseline AP benchmarks. Table 5.3 shows the results of AP for the Olivetti face dataset. Note that the memory consumption was extremely close to that of the previous benchmarks baseline for the Python process, and as shown before $n = 400$ is a relatively low number of samples to hit AP memory constraints.

damping	time	peakmem	silhouette	homogeneity	AMI
0.5	0.0751	114.05MB	0.1516	0.7970	0.6085
0.8	0.1127	114.06MB	0.1478	0.8006	0.6122

Table 5.3: Results from the runs of asv-benchmark on the Olivetti faces dataset for the baseline AP clustering algorithm

The full results from all LAP experiments can be found in Appendix A, in Table A.3. The peak memory consumption of any LAP run never went over 116MB, though some configurations managed to stay near 110MB, consuming a slightly lower amount of memory than AP. The Olivetti dataset shows how in smaller datasets, the difference between AP and LAP can be negligible when it comes to memory consumption while affecting clustering performance and runtime negatively.

Figure 5.16 shows the peak memory usage of LAP for different configurations. Here *sweeps* $= 1$ consumes approximately 1MB less of memory than other configurations. Interestingly, the difference between the rest of *sweeps* values is less pronounced and the same behavior was observed in other datasets, see Figure 5.25.

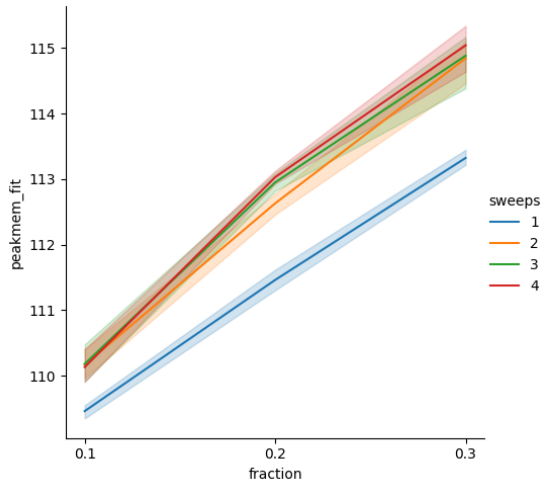


Figure 5.16: Olivetti faces LAP peak memory usage for different configurations

Figure 5.17 shows LAP runtime, which increases with sweeps as it already happened with the synthetic dataset tested in the previous section.

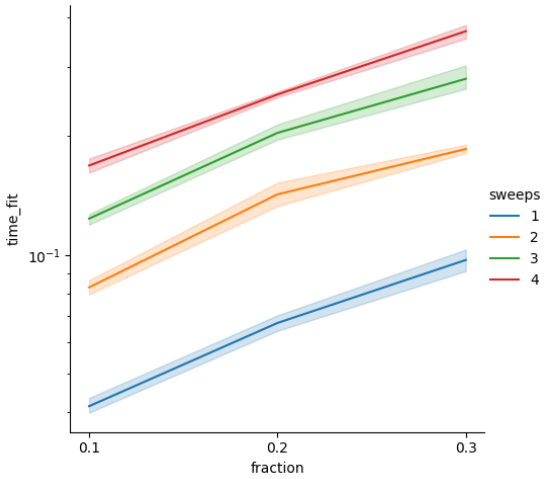
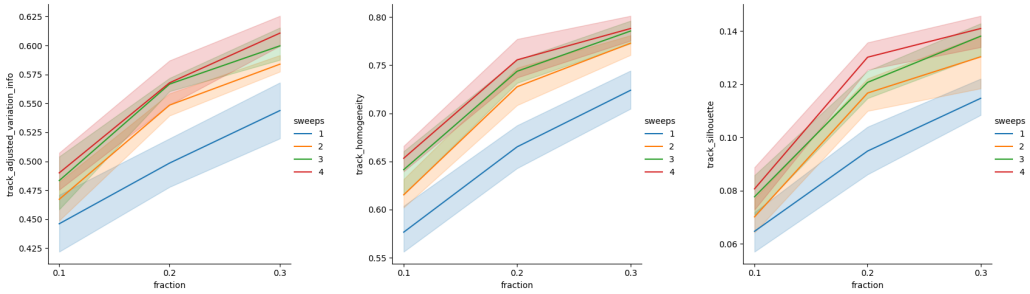


Figure 5.17: Olivetti faces LAP time taken for different configurations. Note the logarithmic scale on the y-axis.

Finally, Figure 5.18 shows clustering performance. As expected, using a real-world dataset makes the additional exploration given by the higher *sweeps* values increase performance in all metrics. This means that the generated clusters were more differentiated and the elements of each cluster were more homogeneous for higher values of *sweeps*.



(a) AMI (b) homogeneity (c) silhouette

Figure 5.18: Olivetti faces LAP clustering performance

LAP still manages to generate clusters of faces that contain images from the same subjects, like in Figure 5.19. However, some clusters include faces from multiple subjects, like in the seventh cluster identified by LAP shown in Figure 5.20.

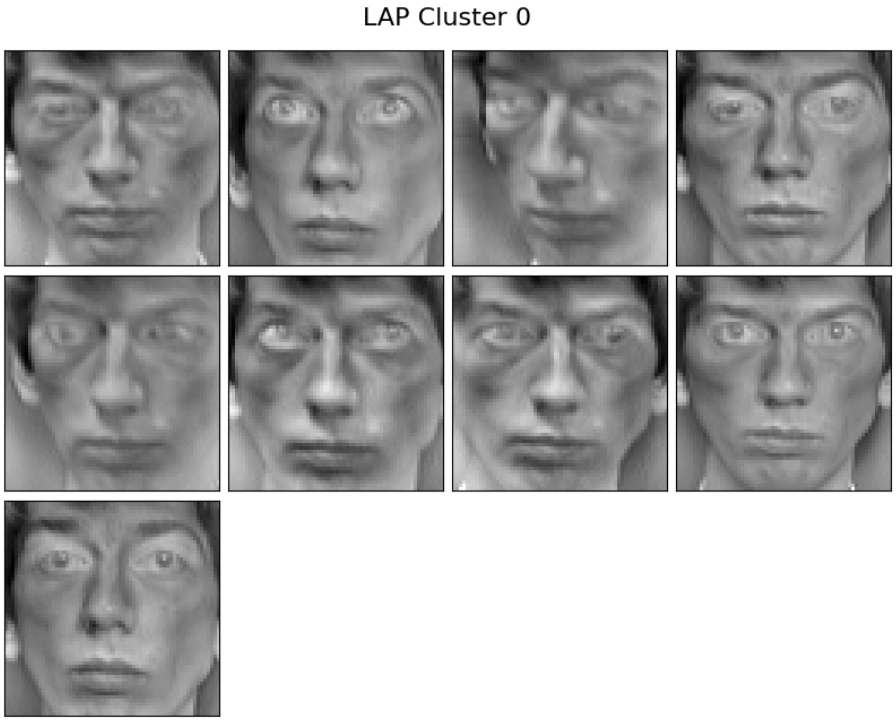


Figure 5.19: LAP results for the Olivetti faces dataset, showing a gallery with the faces from the first cluster identified by LAP. LAP properly identified 9 out of 10 faces from this subject in the same cluster.

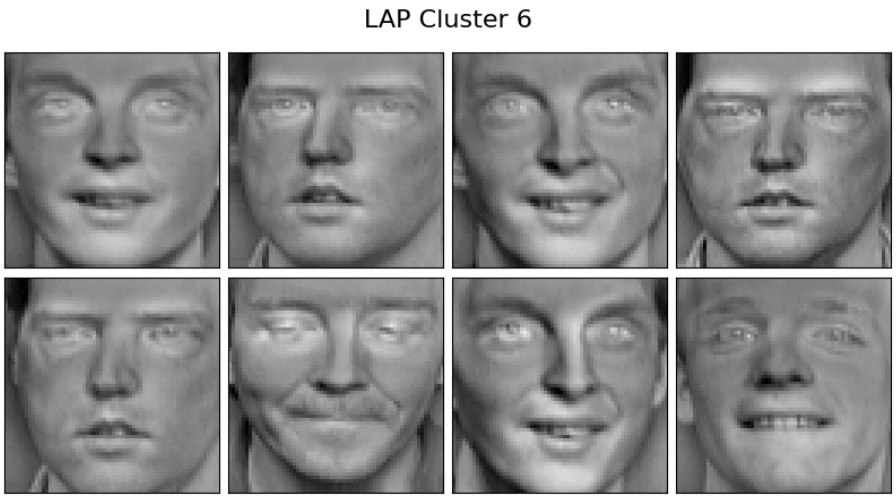


Figure 5.20: LAP results for the Olivetti faces dataset: cluster 6

5.2.3 Fashion-MNIST

The dataset consists of 70,000 grayscale images, each with a resolution of 28x28 pixels. The Fashion-MNIST dataset contains a total of 10 classes or categories, representing different fashion items. It serves as a replacement for the original MNIST handwritten digit dataset, oftentimes used as an introduction to large datasets used for training image processing systems.

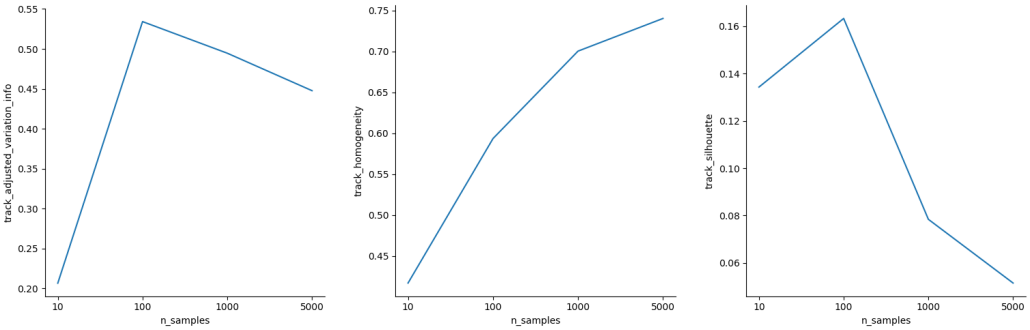
It was chosen as a benchmark to act as a scalable continuation of the Olivetti face dataset. It also offers a more challenging task of classifying fashion-related images when compared to the original MNIST dataset [16].

Moreover, the Fashion-MNIST dataset can be processed by LAP using fast similarity measures, in this case, pairwise Euclidean distances, reducing overhead in the similarity computation in such a way that LAP run-time should be mostly message passing and the algorithm itself rather than similarity computations.

First, AP was executed for varying subsample sizes of the dataset. The results for AP are found in Table 5.4. And the clustering performance can be visualized in Figure 5.21.

n_samples	time	peakmem	memory	silhouette	homogeneity	AMI
10	0.0033	136.08MB	0MB	0.1343	0.4169	0.2067
100	0.0074	137.04MB	1MB	0.1633	0.5935	0.5342
1000	0.4857	169.75MB	34MB	0.0784	0.7003	0.4946
5000	13.8254	922.15MB	786MB	0.0515	0.7402	0.4477

Table 5.4: Results from the runs of asv-benchmark on the Fashion-MNIST dataset for the baseline Affinity Propagation. memory refers to the approximate memory consumed by AP alone, excluding the base consumption of the Python process and benchmarking software. At $n = 10$ the impact of AP on peak memory consumption should be approximately 0MB. As each data point in the Fashion-MNIST dataset is a 28×28 matrix, for values of $n < 1000$ the memory column might be a misleading approximation of memory consumption due to the memory allocated in the process to hold the dataset itself not being negligible.



(a) AMI (b) homogeneity (c) silhouette

Figure 5.21: Fashion-MNIST AP clustering performance

For this dataset, the following parameter grid for LAP was tested. For the asv-benchmarks, fraction values in [0.005, 0.02, 0.01], sweeps in [1, 2, 3], and seeds in [0, 1, 2]. The results of this benchmark are available in the appendix Table A.2. An excerpt of relevant runs is available in Table 5.5. A deeper benchmark that tests against a greater number of LAP parameters can be found in appendix Table B.1.

fraction	n	time	peakmem	memory	silhouette	homo	AMI
0.005	100	0.0033	136.48MB	0.5MB	0.1211	0.1269	0.1668
0.005	1000	0.0199	143.04MB	7MB	0.0951	0.2981	0.3600
0.005	5000	0.2592	173.02MB	37MB	0.0423	0.5141	0.4625
0.02	100	0.0030	136.49MB	0.5MB	0.1234	0.0981	0.1214
0.02	1000	0.0400	143.47MB	7MB	0.0429	0.4475	0.3916
0.02	5000	0.8039	179.41MB	43MB	0.0361	0.6668	0.4615
0.1	100	0.0056	137.12MB	1MB	0.0789	0.4935	0.4483
0.1	1000	0.1973	145.08MB	9MB	0.0631	0.6459	0.4746
0.1	5000	5.8787	256.62MB	120MB	0.0390	0.7156	0.4425

Table 5.5: Results from the runs of asv-benchmark on the Fashion-MNIST dataset for LAP. All runs shown in this table used $sweeps = k = 1$. memory refers to the approximate memory consumed by LAP alone, excluding the base consumption of the Python process and benchmarking software. As each data point in the Fashion-MNIST dataset is a 28×28 matrix, for values of $n < 1000$ the memory column might be a misleading approximation of memory consumption due to the memory allocated in the process to hold the dataset itself not being negligible.

Interestingly, Figure 5.22 shows how increasing the *sweeps* parameter can increase computational time beyond that of AP. This cost may not even come with the clustering performance gains expected from allowing the algorithm to perform more exploration on the dataset, as shown in Figures 5.24, 5.27, 5.28 and 5.29 where the

clustering performance improvements are hard to appreciate, with the most notable difference between *sweeps* values being that the worst results of *sweeps* = 3 are better than the worst results of *sweeps* = 1.

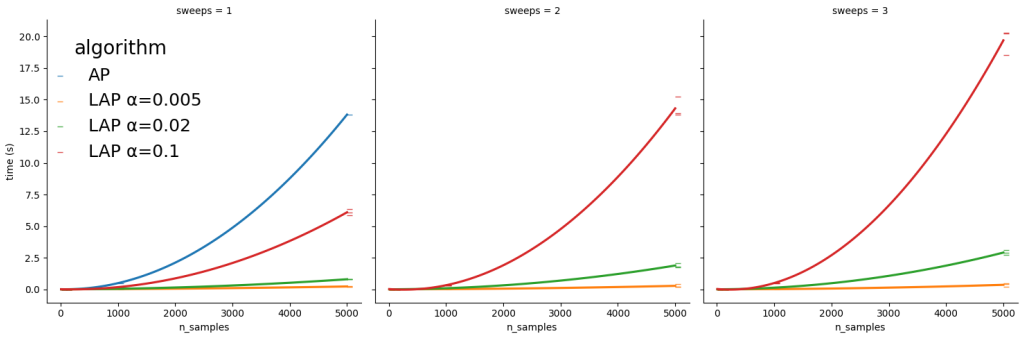


Figure 5.22: Fashion-MNIST AP vs LAP times

In line with other datasets benchmarked, Figure 5.23 shows how peak memory consumption remains better in LAP over AP regardless of the parameters chosen.

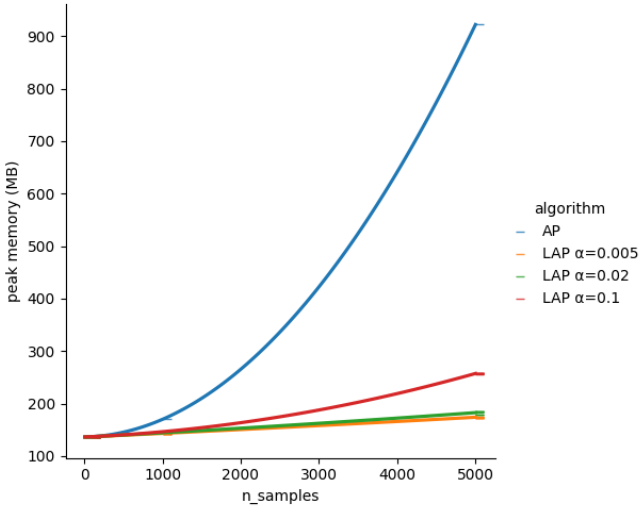


Figure 5.23: Fashion-MNIST AP vs LAP peak memory usage

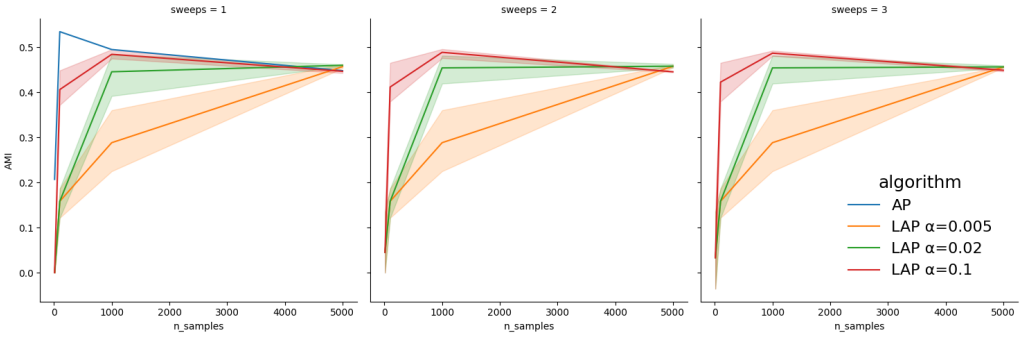


Figure 5.24: Fashion-MNIST AP vs LAP AMI

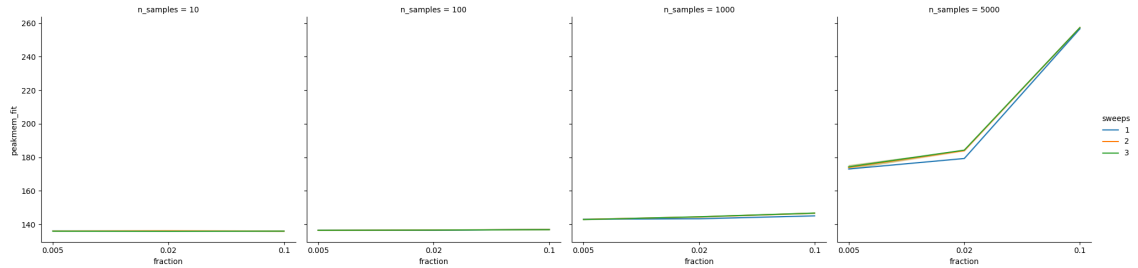


Figure 5.25: Fashion-MNIST LAP peak memory usage for different configurations

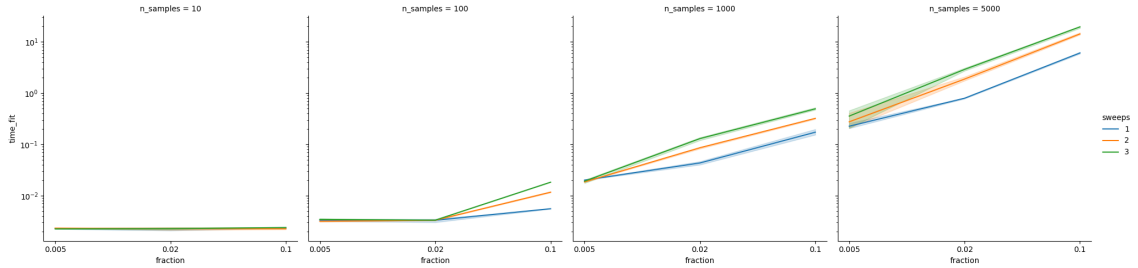


Figure 5.26: Fashion-MNIST LAP time taken for different configurations. Note the logarithmic scale on the y-axis.

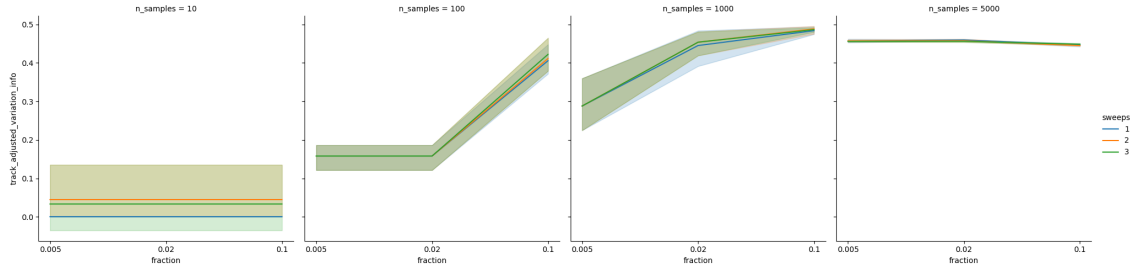


Figure 5.27: Fashion-MNIST LAP peak adjusted mutual information for different configurations

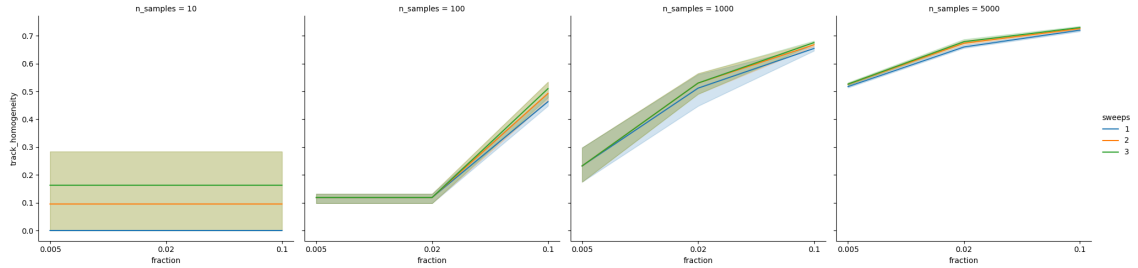


Figure 5.28: Fashion-MNIST LAP homogeneity for different configurations

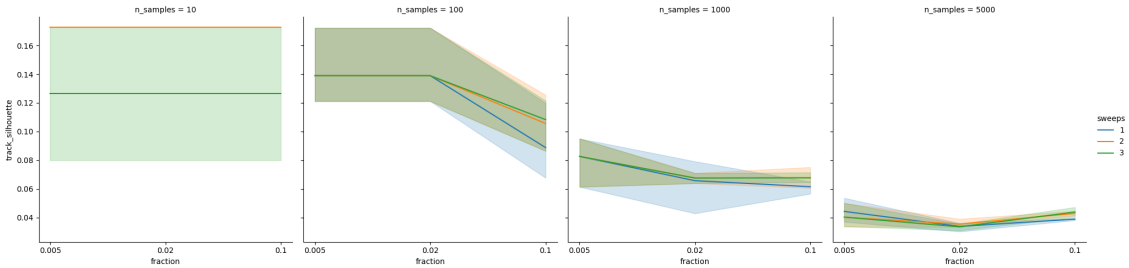


Figure 5.29: Fashion-MNIST LAP silhouette score for different configurations

As to exemplify how Fashion MNIST provides challenging image classification tasks, Figure 5.30 shows the results of the best performing LAP run for the largest subsample tested (fraction=0.1, sweeps=3, seed=2, n_samples=5000).

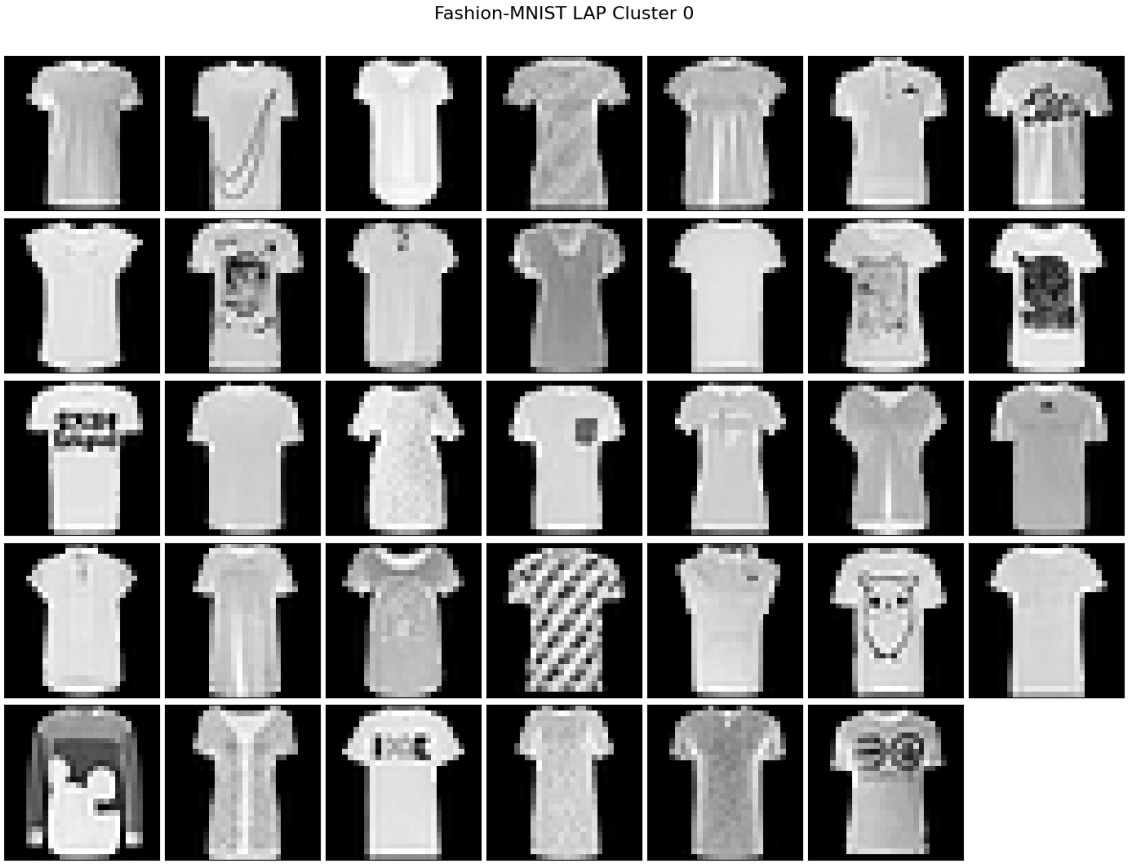


Figure 5.30: Fashion-MNIST LAP first identified cluster. This cluster identified by LAP contains four true categories: 0 (T-shirt/top), 2 (Pullover), 3 (Dress), and 6 (Shirt). Additionally, the long-sleeved pullover is not the only pullover in the cluster, as there is another one.

5.2.4 Amazon

The Amazon Berkeley Objects (ABO) Dataset [17], also known as the Amazon EC2/S3 Berkeley Segmentation Dataset, is a dataset designed for image segmentation and object recognition tasks. It was created in collaboration between Amazon and the University of California, Berkeley.

The metadata associated with each image can also be useful for various NLP (Natural Language Processing) and unsupervised learning tasks by providing additional contextual information. It can also be used as a large collection of real-world data when a very large sample is needed. A large open dataset like ABO is extremely useful when benchmarking memory or CPU-intensive unsupervised learning algorithms. It facilitates thorough testing, performance evaluation, stress testing, algorithm comparison, and resource optimization.

This dataset was chosen to showcase Leveraged Affinity Propagation flexibility and scalability, going from image processing in the two previous benchmarks to Natural Language Processing with more than 120k samples. While it is also a dataset where a target variable could be considered, mainly *type*, extracting new patterns of related products is much more interesting, and benefits from Affinity Propagation not needing to specify the number of clusters to find.

5.2.4.1 Methodology

The ABO metadata was used to run all the testing in this section. First, all the separate metadata files that compose the original dataset were merged. Then the data was cleaned and only items with a name in English were kept. Afterward, a TFIDF vectorizer was trained on the keywords, using at most 5000 features and removing all English stopwords.

The TFIDF values are then used as input to the clustering algorithms. Unless specified otherwise, Affinity Propagation and Leveraged Affinity Propagation will both use cosine similarity as their similarity function. Due to a large number of data points and failure to converge during initial exploration, the maximum number of iterations was set higher than default along with the damping value.

Table 5.6 shows five different metadata entries for the ABO dataset, only including the relevant columns. Note that what is being tested is not the ability of the algorithms to classify any given object to a *type*, but rather their capability to extract new patterns that were not encoded by the type column. In this particular dataset, the patterns extracted could be used to build product recommendation systems, for example.

id	name	keywords	type
B01MTEI8M6	Brand Fix US The 6.5 Floral Slide French Embroidery Havana Tan, Amazon Slipon Loafer, Women's B	gifts spring her zapatos mocasines fashion for wear designer business cocktail office para work fall shoe de ladies moda sexy mujer womans	SHOES
B0853X2F4M	Brand Hard Back for 3D Redmi Designer Case Cover Mi Go Printed Autumn Mobile Amazon Solimo Girl	Back phonecase Redmi Designer mobile phonecover Go cover fashion Cover case backcover cellphonecover cellphonecase mobileguard covers phoneguard Hard mobilecover Case protectivecase polycarbonate backcase Mi Autumn cases and mobilecase Printed Mobile protectivecover Girl	CELLULAR PHONE CASE
B07R91S92W	Brand Hard Back for Redmi (D187) Designer Butterflies Case Cover Y2 Printed Mobile Xiaomi Amazon Solimo	printed new Redmi slim back case girls designer Xiaomi cover Y2 transparent stylish Mobile boys	CELLULAR PHONE CASE
B07CTPR73M	Swatch, & Stone 2502003901 Beam Brown	living windmill seat farmhouse tufted vanity outdoor loveseat sofa savonburg arm size couches sets silver daybed rolled upholstered fountain leather red reclining and room with button set love fabric wind trundle sofas chesterfield homelegance spinners couch loveseats queen a for power	SOFA
B07H9GMYXS	1.75mm, 3D 1.75mm PETG Filament, AMG1052851610 AmazonBasics Printer 1 Spool kg	printer 3D 3d yellow 1.75mm 1kg filament spool translucent petg	MECHANICAL COMPONENTS

Table 5.6: Amazon Berkeley Objects (ABO) metadata example, limited to the id, name, keywords, and type columns. Minimal processing was done to show multiple name or keyword English entries in a single cell.

5.2.4.2 Evaluation

As shown in table 5.7, Affinity Propagation is extremely expensive when compared to LAP, it is so expensive, that the benchmarking machine runs out of memory when attempting to run AP with the entirety of the ABO dataset.

n_samples	algorithm	peak memory	time
20000	AP	15.19GB	37min
20000	LAP 0.02×10	458.84MB	3m 6s
122734	LAP 0.01×10	5.78GB	216m 26s

Table 5.7: Performance profile of AP and LAP for a subsample of the ABO Dataset and the performance profile of LAP for the whole dataset.

It should be mentioned that the similarity matrix for 20000 data points accounts for 2.98GB of the process' total memory usage. Also, the whole ABO Dataset's TF-IDF values consume a total of 10.72MB due to being stored as a sparse matrix, even if it is a 122734 by 5000 matrix.

The subsample was needed to compare clustering results between AP and LAP. The exact number of data points (20000) to consider was chosen due to Affinity Propagation's memory consumption being very close to 16GB of RAM, the maximum memory commonly found in modern commodity hardware. Just the similarity matrix for 40000 datapoints took a total of 11.92GB of memory and what's more, peak process memory consumption for the cosine similarity computation surpassed 16GB, reaching 17.90GB of memory.

The clustering results from LAP were compared to those of the 20000-element AP subsample. For the LAP run that only used the first 20000 elements (LAP 20k) there is no additional processing required, for the LAP run that used the entire ABO dataset (LAP 122k), only the labels of the first 20000 elements are taken into account. The results are available in Table 5.8.

From the selected parameters for LAP, we know that the maximum number of clusters that LAP 20k can find is $n \times \alpha = 20000 * 0.02 = 400$ and the maximum number of clusters for LAP 122k is $n \times \alpha = 122734 * 0.01 = 1223$. Neither LAP run reached the maximum number of exemplars in the selection, meaning that both algorithms explored new exemplar candidates each sweep. AP found 473 clusters in the dataset subsample.

metric	LAP 20k	LAP 122k
Clusters found	333	1048
V-measure	0.819	0.837
AMI	0.738	0.719
Rand Index	0.997	0.998
Adjusted Rand Index	0.687	0.629
Fowlkes-Mallows	0.693	0.652

Table 5.8: Clustering performance of LAP compared to the clustering results of AP as if those were the true labels.

The clustering results from both LAP runs were close to those of traditional AP, with very high Rand Index scores. The V-measure score and AMI can be used to measure agreement between different clustering strategies on the same dataset. Both scores have an upper limit of 1 and they are close to it. This means that the clusters generated by the LAP runs are similar to those produced by AP.

Rand Index has a value between 0 and 1, with 0 indicating that the two data clusterings do not agree on any pair of points and 1 indicating that the data clusterings are exactly the same. The Adjusted Rand Index takes the raw value from the Rand Index and adjusts it for chance. Both scores are high for the two LAP runs, so the results from the clustering of AP and LAP are similar.

From these results, it can be assumed that LAP 22k and LAP 122k mostly agree with the clustering from AP. In LAP 20k case, it shows how we can use LAP to obtain clusters close to those that would be generated by AP at a much lower peak memory consumption. In this particular example which also had run-time gains, it does so in less than a thirtieth of the peak memory consumption and a tenth of the time that AP took.

Chapter 6

Conclusions

This work successfully implemented a lightweight variant of the Affinity Propagation (AP) clustering algorithm in Python, known as Leveraged Affinity Propagation, achieving the objective of making LAP available to Python developers. The implementation of the lightweight variant of AP contributes to the Python ecosystem by addressing the critical challenge of memory consumption, particularly in scenarios with limited resources or large datasets.

The implemented LAP algorithm demonstrated notable improvements in memory efficiency while maintaining comparable clustering quality to the standard AP algorithm for large sample sizes. Through experimental evaluation and benchmarking, it was confirmed that the lightweight variant consistently exhibited reduced memory usage, and different dataset sample sizes were tested, identifying situations where the memory gains of applying LAP were minimal, while runtime and clustering performance suffered when compared to AP.

The Python package is also available to the larger Python community as it has been published on PyPI, the Python Package Index, the most commonly used and available repository to distribute and download Python packages.

6.1 Future work

The package currently implements LAP as its only AP variant, but it may be possible to provide further optimizations for LAP when it comes to runtime, as they were already introduced in [14], or explore other memory optimizations not currently available for AP in Python.

Glossary

availability A value indicating the accumulated evidence for a data point to choose another data point as its exemplar

cluster A group of data points that are similar to each other and dissimilar to points in other clusters

convergence The state where the responsibility and availability matrices reach a stable configuration, indicating the end of the AP algorithm

exemplar A data point selected as a representative of a cluster in AP

outlier A data point that significantly deviates from the patterns or characteristics of other points

responsibility A value indicating the suitability of a data point to be an exemplar for another data point

similarity A measure indicating how alike two data points are in terms of their features

similarity matrix A square matrix that represents the pairwise similarities between data points in a dataset

Acronyms

AMI Adjusted Mutual Information

AP Affinity Propagation

ARI Adjusted Rand Index

IDF Inverse Document Frequency

LAP Leveraged Affinity Propagation

NMI Normalized Mutual Information

RI Rand Index

TF Term Frequency

Bibliography

[1] D. Xu and Y. Tian, "A Comprehensive Survey of Clustering Algorithms," *Annals of Data Science*, vol. 2, no. 2, pp. 165–193, Jun. 2015, ISSN: 2198-5804, 2198-5812. DOI: 10.1007/s40745-015-0040-1. [Online]. Available: http://link.springer.com/10.1007/s40745-015-0040-1.

[2] B. J. Frey and D. Dueck, "Clustering by Passing Messages Between Data Points," *Science*, vol. 315, no. 5814, pp. 972–976, Feb. 16, 2007, ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.1136800. [Online]. Available: https://www.science.org/doi/10.1126/science.1136800.

[3] U. Bodenhofer, A. Kothmeier, and S. Hochreiter, "APCluster: An R package for affinity propagation clustering," *Bioinformatics*, vol. 27, no. 17, pp. 2463–2464, Sep. 1, 2011, ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btr406. [Online]. Available: https://doi.org/10.1093/bioinformatics/btr406.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, no. 85, pp. 2825–2830, 2011, ISSN: 1533-7928. [Online]. Available: http://jmlr.org/papers/v12/pedregosa11a.html.

[5] S. Weng, J. Shang, Y. Cheng, *et al.*, "Genetic differentiation and diversity of SARS-CoV-2 Omicron variant in its early outbreak," *Biosafety and Health*, vol. 04, no. 03, pp. 171–178, Jun. 25, 2022. DOI: 10.1016/j.bsheal.2022.04.004. [Online]. Available: https://mednexus.org/doi/full/10.1016/j.bsheal.2022.04.004.

[6] S. Ligthart, A. Vaez, U. Võsa, *et al.*, "Genome Analyses of >200,000 Individuals Identify 58 Loci for Chronic Inflammation and Highlight Pathways that Link Inflammation and Complex Disorders," *The American Journal of Human Genetics*, vol. 103, no. 5, pp. 691–706, Nov. 1, 2018, ISSN: 0002-9297. DOI: 10.1016/j.ajhg.2018.09.009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0002929718303203.

[7] J. Saltz, R. Gupta, L. Hou, *et al.*, "Spatial Organization and Molecular Correlation of Tumor-Infiltrating Lymphocytes Using Deep Learning on Pathology Images," *Cell Reports*, vol. 23, no. 1, 181–193.e7, Apr. 2018, ISSN: 22111247. DOI: 10.1016/j.celrep.2018.03.086. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2211124718304479.

[8] R. Guan, X. Shi, M. Marchese, C. Yang, and Y. Liang, "Text Clustering with Seeds Affinity Propagation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 4, pp. 627–637, Apr. 2011, ISSN: 1558-2191. DOI: 10.1109/TKDE.2010.144.

[9] D. M. Rose, J. M. Rouly, R. Haber, N. Mijatovic, and A. M. Peter. "Parallel Hierarchical Affinity Propagation with MapReduce." arXiv: 1403.7394 [cs]. (Mar. 28, 2014), [Online]. Available: http://arxiv.org/abs/1403.7394, preprint.

[10] juhis, *Affinity-propagation*, Nov. 27, 2017. [Online]. Available: https://github.com/juhis/affinity-propagation.

[11] "The Comprehensive R Archive Network." (), [Online]. Available: https://cran.r-project.org/.

[12] E. Schubert, "Automatic Indexing for Similarity Search in ELKI," in *Similarity Search and Applications*, T. Skopal, F. Falchi, J. Lokoč, M. L. Sapino, I. Bartolini, and M. Patella, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2022, pp. 205–213, ISBN: 978-3-031-17849-8. DOI: 10.1007/978-3-031-17849-8_16.

[13] *Clustering.jl*, Julia Statistics, Jun. 2, 2023. [Online]. Available: https://github.com/JuliaStats/Clustering.jl.

[14] C. Melkonian, "Similarity-based clustering of big data: A parallel ensemble variant of leveraged affinity propagation,"

[15] *Python -> R bridge*, rpy2, Jun. 10, 2023. [Online]. Available: https://github.com/rpy2/rpy2.

[16] H. Xiao, K. Rasul, and R. Vollgraf. "Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms." arXiv: 1708.07747 [cs, stat]. (Sep. 15, 2017), [Online]. Available: http://arxiv.org/abs/1708.07747, preprint.

[17] J. Collins, S. Goel, K. Deng, *et al.* "ABO: Dataset and Benchmarks for Real-World 3D Object Understanding." version 2. arXiv: 2110.06199 [cs]. (Jun. 24, 2022), [Online]. Available: http://arxiv.org/abs/2110.06199, preprint.

Appendix A

ASV Benchmarks

fraction	sweeps	seed	n	time_fit	fraction	sweeps	seed	n	time_fit
0.005	1	0	10	0.0023	.	.	.	.	
0.005	1	0	1000	0.0140	⋮	⋮	⋮	⋮	
0.005	1	0	2000	0.0449	0.02	2	1	2000	0.2159
0.005	1	0	5000	0.1836	0.02	2	1	5000	1.5700
0.005	1	1	10	0.0023	0.02	2	3	10	0.0022
0.005	1	1	1000	0.0159	0.02	2	3	1000	0.0581
0.005	1	1	2000	0.0348	0.02	2	3	2000	0.2150
0.005	1	1	5000	0.2234	0.02	2	3	5000	1.2414
0.005	1	3	10	0.0023	0.02	5	0	10	0.0023
0.005	1	3	1000	0.0201	0.02	5	0	1000	0.1840
0.005	1	3	2000	0.0355	0.02	5	0	2000	0.5888
0.005	1	3	5000	0.2028	0.02	5	0	5000	3.6802
0.005	2	0	10	0.0022	0.02	5	1	10	0.0023
0.005	2	0	1000	0.0150	0.02	5	1	1000	0.1635
0.005	2	0	2000	0.0781	0.02	5	1	2000	0.6223
0.005	2	0	5000	0.4692	0.02	5	1	5000	3.2479
0.005	2	1	10	0.0023	0.02	5	3	10	0.0023
0.005	2	1	1000	0.0302	0.02	5	3	1000	0.1762
0.005	2	1	2000	0.0749	0.02	5	3	2000	0.5429
0.005	2	1	5000	0.3794	0.02	5	3	5000	3.5708
0.005	2	3	10	0.0023	0.1	1	0	10	0.0023
0.005	2	3	1000	0.0198	0.1	1	0	1000	0.1447
0.005	2	3	2000	0.0774	0.1	1	0	2000	0.4786
0.005	2	3	5000	0.3975	0.1	1	0	5000	3.8689
0.005	5	0	10	0.0023	0.1	1	1	10	0.0024
0.005	5	0	1000	0.0152	0.1	1	1	1000	0.1374
0.005	5	0	2000	0.1900	0.1	1	1	2000	0.5148
0.005	5	0	5000	1.0479	0.1	1	1	5000	5.0410
0.005	5	1	10	0.0023	0.1	1	3	10	0.0022
0.005	5	1	1000	0.0297	0.1	1	3	1000	0.2143
0.005	5	1	2000	0.1207	0.1	1	3	2000	0.4848
0.005	5	1	5000	1.1103	0.1	1	3	5000	4.2602
0.005	5	3	10	0.0024	0.1	2	0	10	0.0024
0.005	5	3	1000	0.0199	0.1	2	0	1000	0.2725
0.005	5	3	2000	0.1925	0.1	2	0	2000	1.2903
0.005	5	3	5000	1.0764	0.1	2	0	5000	8.6382
0.02	1	0	10	0.0024	0.1	2	1	10	0.0024
0.02	1	0	1000	0.0293	0.1	2	1	1000	0.3839
0.02	1	0	2000	0.0970	0.1	2	1	2000	1.0479
0.02	1	0	5000	0.7035	0.1	2	1	5000	8.3682
0.02	1	1	10	0.0024	0.1	2	3	10	0.0026
0.02	1	1	1000	0.0327	0.1	2	3	1000	0.2764
0.02	1	1	2000	0.1101	0.1	2	3	2000	1.0658
0.02	1	1	5000	0.6982	0.1	2	3	5000	8.2853
0.02	1	3	10	0.0023	0.1	5	0	10	0.0024
0.02	1	3	1000	0.0385	0.1	5	0	1000	0.6919
0.02	1	3	2000	0.1260	0.1	5	0	2000	2.7097
0.02	1	3	5000	0.8684	0.1	5	0	5000	20.9603
0.02	2	0	10	0.0023	0.1	5	1	10	0.0024
0.02	2	0	1000	0.0683	0.1	5	1	1000	0.6491
0.02	2	0	2000	0.2329	0.1	5	1	2000	2.9177
0.02	2	0	5000	1.6280	0.1	5	1	5000	20.7587
0.02	2	1	10	0.0023	0.1	5	3	10	0.0022
0.02	2	1	1000	0.0653	0.1	5	3	1000	0.7756
.	.	.	.		0.1	5	3	2000	2.8333
⋮	⋮	⋮	⋮		0.1	5	3	5000	22.9864

Table A.1: Results from the runs of asv-benchmark on the synthetic dataset

fraction	sweeps	seed	n_samples	time	peakmem	peakmem	silhouette	homogeneity	AMI
0.005	1	0	10	0.0023	142606336	136.00MB	NaN	0.0000	0.0000
0.005	1	0	100	0.0033	143110144	136.48MB	0.1211	0.1269	0.1668
0.005	1	0	1000	0.0199	149991424	143.04MB	0.0951	0.2981	0.3600
0.005	1	0	5000	0.2592	181420032	173.02MB	0.0423	0.5141	0.4625
0.005	1	1	10	0.0023	142979072	136.36MB	NaN	0.0000	0.0000
0.005	1	1	100	0.0031	143298560	136.66MB	0.1723	0.1321	0.1864
0.005	1	1	1000	0.0199	150155264	143.20MB	0.0915	0.2241	0.2795
0.005	1	1	5000	0.2136	181460992	173.05MB	0.0370	0.5248	0.4544
0.005	1	2	10	0.0022	142733312	136.12MB	NaN	0.0000	0.0000
0.005	1	2	100	0.0035	143101952	136.47MB	0.1234	0.0981	0.1214
0.005	1	2	1000	0.0204	150151168	143.20MB	0.0614	0.1748	0.2248
0.005	1	2	5000	0.2057	181583872	173.17MB	0.0536	0.5135	0.4530
0.005	2	0	10	0.0023	142635008	136.03MB	0.1728	0.2843	0.1351
0.005	2	0	100	0.0031	143081472	136.45MB	0.1211	0.1269	0.1668
0.005	2	0	1000	0.0190	150200320	143.24MB	0.0951	0.2981	0.3600
0.005	2	0	5000	0.2090	181854208	173.43MB	0.0336	0.5300	0.4608
0.005	2	1	10	0.0023	142921728	136.30MB	NaN	0.0000	0.0000
0.005	2	1	100	0.0031	143114240	136.48MB	0.1723	0.1321	0.1864
0.005	2	1	1000	0.0182	149872640	142.93MB	0.0915	0.2241	0.2795
0.005	2	1	5000	0.2085	181387264	172.98MB	0.0370	0.5248	0.4544
0.005	2	2	10	0.0023	142569472	135.96MB	NaN	0.0000	0.0000
0.005	2	2	100	0.0032	143200256	136.57MB	0.1234	0.0981	0.1214
0.005	2	2	1000	0.0182	149864448	142.92MB	0.0614	0.1748	0.2248
0.005	2	2	5000	0.4097	183955456	175.43MB	0.0501	0.5225	0.4569
0.005	3	0	10	0.0022	142798848	136.18MB	0.1728	0.2843	0.1351
0.005	3	0	100	0.0035	143265792	136.63MB	0.1211	0.1269	0.1668
0.005	3	0	1000	0.0175	149848064	142.91MB	0.0951	0.2981	0.3600
0.005	3	0	5000	0.4578	183828480	175.31MB	0.0339	0.5321	0.4566
0.005	3	1	10	0.0023	142589952	135.98MB	NaN	0.0000	0.0000
0.005	3	1	100	0.0036	143089664	136.46MB	0.1723	0.1321	0.1864
0.005	3	1	1000	0.0198	150028288	143.08MB	0.0915	0.2241	0.2795
0.005	3	1	5000	0.2083	181444608	173.04MB	0.0370	0.5248	0.4544
0.005	3	2	10	0.0023	142479360	135.88MB	0.0800	0.2025	-0.0357
0.005	3	2	100	0.0032	143159296	136.53MB	0.1234	0.0981	0.1214
0.005	3	2	1000	0.0202	149762048	142.82MB	0.0614	0.1748	0.2248
0.005	3	2	5000	0.4046	183287808	174.80MB	0.0501	0.5225	0.4569
0.02	1	0	10	0.0021	142819328	136.20MB	NaN	0.0000	0.0000
0.02	1	0	100	0.0035	143212544	136.58MB	0.1211	0.1269	0.1668
0.02	1	0	1000	0.0465	150462464	143.49MB	0.0752	0.5612	0.4838
0.02	1	0	5000	0.8075	188076032	179.36MB	0.0353	0.6567	0.4599
0.02	1	1	10	0.0023	142536704	135.93MB	NaN	0.0000	0.0000
0.02	1	1	100	0.0034	143220736	136.59MB	0.1723	0.1321	0.1864
0.02	1	1	1000	0.0435	150253568	143.29MB	0.0792	0.5264	0.4604
0.02	1	1	5000	0.7780	187973632	179.27MB	0.0302	0.6556	0.4576
0.02	1	2	10	0.0023	142827520	136.21MB	NaN	0.0000	0.0000
0.02	1	2	100	0.0030	143122432	136.49MB	0.1234	0.0981	0.1214
0.02	1	2	1000	0.0400	150437888	143.47MB	0.0429	0.4475	0.3916
0.02	1	2	5000	0.8039	188121088	179.41MB	0.0361	0.6668	0.4615
0.02	2	0	10	0.0021	142741504	136.13MB	0.1728	0.2843	0.1351
0.02	2	0	100	0.0034	143454208	136.81MB	0.1211	0.1269	0.1668
0.02	2	0	1000	0.0885	151584768	144.56MB	0.0678	0.5647	0.4811
0.02	2	0	5000	2.0903	193159168	184.21MB	0.0333	0.6680	0.4558
0.02	2	1	10	0.0024	142839808	136.22MB	NaN	0.0000	0.0000
0.02	2	1	100	0.0033	143200256	136.57MB	0.1723	0.1321	0.1864
0.02	2	1	1000	0.0820	151298048	144.29MB	0.0711	0.5348	0.4615
0.02	2	1	5000	1.7501	192667648	183.74MB	0.0341	0.6687	0.4544
0.02	2	2	10	0.0023	142966784	136.34MB	NaN	0.0000	0.0000
0.02	2	2	100	0.0033	143364096	136.72MB	0.1234	0.0981	0.1214
0.02	2	2	1000	0.0872	151715840	144.69MB	0.0639	0.4904	0.4191
0.02	2	2	5000	1.8096	193093632	184.15MB	0.0390	0.6829	0.4626
0.02	3	0	10	0.0022	142512128	135.91MB	0.1728	0.2843	0.1351
0.02	3	0	100	0.0033	143208448	136.57MB	0.1211	0.1269	0.1668
0.02	3	0	1000	0.1406	151691264	144.66MB	0.0678	0.5647	0.4811
0.02	3	0	5000	3.0979	193417216	184.46MB	0.0310	0.6764	0.4555
0.02	3	1	10	0.0024	142667776	136.06MB	NaN	0.0000	0.0000
0.02	3	1	100	0.0033	143347712	136.71MB	0.1723	0.1321	0.1864
0.02	3	1	1000	0.1287	151523328	144.50MB	0.0711	0.5348	0.4615
0.02	3	1	5000	2.7269	193150976	184.20MB	0.0341	0.6722	0.4537
0.02	3	2	10	0.0023	142434304	135.84MB	0.0800	0.2025	-0.0357
0.02	3	2	100	0.0033	143380480	136.74MB	0.1234	0.0981	0.1214
0.02	3	2	1000	0.1198	151617536	144.59MB	0.0639	0.4904	0.4191
0.02	3	2	5000	2.8999	193277952	184.32MB	0.0356	0.6873	0.4592

Continued on next page

fraction	sweeps	seed	n_samples	time	peakmem	peakmem	silhouette	homogeneity	AMI
0.1	1	0	10	0.0023	142856192	136.24MB	NaN	0.0000	0.0000
0.1	1	0	100	0.0056	143781888	137.12MB	0.0789	0.4935	0.4483
0.1	1	0	1000	0.1973	152129536	145.08MB	0.0631	0.6459	0.4746
0.1	1	0	5000	5.8787	269086720	256.62MB	0.0390	0.7156	0.4425
0.1	1	1	10	0.0023	142741504	136.13MB	NaN	0.0000	0.0000
0.1	1	1	100	0.0055	143618048	136.96MB	0.1198	0.4482	0.3977
0.1	1	1	1000	0.1539	152203264	145.15MB	0.0650	0.6545	0.4949
0.1	1	1	5000	6.0741	269299712	256.82MB	0.0381	0.7209	0.4465
0.1	1	2	10	0.0024	142573568	135.97MB	NaN	0.0000	0.0000
0.1	1	2	100	0.0056	143765504	137.11MB	0.0677	0.4484	0.3716
0.1	1	2	1000	0.1672	152211456	145.16MB	0.0566	0.6629	0.4825
0.1	1	2	5000	6.3318	269139968	256.67MB	0.0396	0.7235	0.4501
0.1	2	0	10	0.0022	142929920	136.31MB	0.1728	0.2843	0.1351
0.1	2	0	100	0.0115	143532032	136.88MB	0.1256	0.5343	0.4651
0.1	2	0	1000	0.3236	154058752	146.92MB	0.0606	0.6623	0.4756
0.1	2	0	5000	13.9018	269664256	257.17MB	0.0446	0.7242	0.4466
0.1	2	1	10	0.0022	142745600	136.13MB	NaN	0.0000	0.0000
0.1	2	1	100	0.0114	143704064	137.05MB	0.1049	0.4668	0.3900
0.1	2	1	1000	0.3299	153812992	146.69MB	0.0751	0.6632	0.4941
0.1	2	1	5000	15.2352	270114816	257.60MB	0.0428	0.7285	0.4440
0.1	2	2	10	0.0024	142327808	135.73MB	NaN	0.0000	0.0000
0.1	2	2	100	0.0120	143716352	137.06MB	0.0864	0.4781	0.3794
0.1	2	2	1000	0.3119	153735168	146.61MB	0.0675	0.6770	0.4956
0.1	2	2	5000	13.8068	270143488	257.63MB	0.0414	0.7241	0.4452
0.1	3	0	10	0.0024	142516224	135.91MB	0.1728	0.2843	0.1351
0.1	3	0	100	0.0182	143876096	137.21MB	0.1256	0.5343	0.4651
0.1	3	0	1000	0.4731	153878528	146.75MB	0.0646	0.6756	0.4804
0.1	3	0	5000	18.5410	270000128	257.49MB	0.0472	0.7344	0.4512
0.1	3	1	10	0.0024	142880768	136.26MB	NaN	0.0000	0.0000
0.1	3	1	100	0.0183	143515648	136.87MB	0.1130	0.5180	0.4225
0.1	3	1	1000	0.4832	154238976	147.09MB	0.0715	0.6693	0.4872
0.1	3	1	5000	20.2452	269885440	257.38MB	0.0442	0.7277	0.4489
0.1	3	2	10	0.0024	142589952	135.98MB	0.0800	0.2025	-0.0357
0.1	3	2	100	0.0183	143339520	136.70MB	0.0864	0.4781	0.3794
0.1	3	2	1000	0.5251	153649152	146.53MB	0.0671	0.6816	0.4921
0.1	3	2	5000	20.3047	269799424	257.30MB	0.0403	0.7266	0.4460

Table A.2: Results from the runs of asv-benchmark on the Fashion-MNIST dataset

fraction	sweeps	seed	time	peakmem	peakmem	silhouette	homogeneity	AMI
0.1	1	0	0.0390	114712576	109.40MB	0.0760	0.6266	0.4864
0.1	1	1	0.0424	114864128	109.54MB	0.0582	0.5515	0.4184
0.1	1	2	0.0407	114900992	109.58MB	0.0636	0.5562	0.4298
0.1	1	3	0.0400	114823168	109.50MB	0.0732	0.5875	0.4744
0.1	1	4	0.0450	114581504	109.27MB	0.0526	0.5608	0.4214
0.1	2	0	0.0881	115523584	110.17MB	0.0822	0.6453	0.4881
0.1	2	1	0.0872	115499008	110.15MB	0.0612	0.5925	0.4365
0.1	2	2	0.0789	116002816	110.63MB	0.0735	0.6170	0.4767
0.1	2	3	0.0798	115343360	110.00MB	0.0688	0.6059	0.4791
0.1	2	4	0.0803	115220480	109.88MB	0.0649	0.6160	0.4554
0.1	3	0	0.1250	115228672	109.89MB	0.0845	0.6608	0.5019
0.1	3	1	0.1254	116105216	110.73MB	0.0717	0.6583	0.4796
0.1	3	2	0.1228	115150848	109.82MB	0.0907	0.6614	0.5175
0.1	3	3	0.1165	115494912	110.14MB	0.0723	0.6322	0.4816
0.1	3	4	0.1291	115675136	110.32MB	0.0692	0.5945	0.4362
0.1	4	0	0.1791	115515392	110.16MB	0.0899	0.6643	0.5003
0.1	4	1	0.1769	116015104	110.64MB	0.0717	0.6583	0.4796
0.1	4	2	0.1589	115355648	110.01MB	0.0937	0.6717	0.5223
0.1	4	3	0.1623	115429376	110.08MB	0.0752	0.6418	0.4782
0.1	4	4	0.1661	115089408	109.76MB	0.0727	0.6306	0.4697
0.2	1	0	0.0666	117153792	111.73MB	0.1068	0.7057	0.5318
0.2	1	1	0.0685	116674560	111.27MB	0.0845	0.6372	0.4620
0.2	1	2	0.0673	117035008	111.61MB	0.1062	0.6776	0.5155
0.2	1	3	0.0726	116695040	111.29MB	0.0937	0.6346	0.4876
0.2	1	4	0.0615	116817920	111.41MB	0.0831	0.6707	0.4956
0.2	2	0	0.1569	118177792	112.70MB	0.1229	0.7509	0.5649
0.2	2	1	0.1399	118149120	112.68MB	0.1040	0.7024	0.5331
0.2	2	2	0.1292	117825536	112.37MB	0.1229	0.7321	0.5464
0.2	2	3	0.1321	118394880	112.91MB	0.1145	0.7013	0.5400

Continued on next page

fraction	sweeps	seed	time	peakmem	peakmem	silhouette	homogeneity	AMI
0.2	2	4	0.1547	117956608	112.49MB	0.1190	0.7512	0.5583
0.2	3	0	0.2057	118267904	112.79MB	0.1224	0.7549	0.5683
0.2	3	1	0.1930	118288384	112.81MB	0.1092	0.7224	0.5630
0.2	3	2	0.2245	118390784	112.91MB	0.1214	0.7441	0.5728
0.2	3	3	0.1979	118562816	113.07MB	0.1270	0.7353	0.5566
0.2	3	4	0.1991	118669312	113.17MB	0.1233	0.7617	0.5728
0.2	4	0	0.2502	118349824	112.87MB	0.1250	0.7396	0.5451
0.2	4	1	0.2602	118661120	113.16MB	0.1243	0.7278	0.5456
0.2	4	2	0.2496	118456320	112.97MB	0.1336	0.7664	0.5864
0.2	4	3	0.2553	118464512	112.98MB	0.1276	0.7507	0.5617
0.2	4	4	0.2610	118665216	113.17MB	0.1401	0.7932	0.6003
0.3	1	0	0.1030	118759424	113.26MB	0.1126	0.7483	0.5455
0.3	1	1	0.1038	118685696	113.19MB	0.1058	0.6949	0.5119
0.3	1	2	0.0885	118730752	113.23MB	0.1283	0.7161	0.5732
0.3	1	3	0.0879	119091200	113.57MB	0.1178	0.7041	0.5146
0.3	1	4	0.1029	118870016	113.36MB	0.1088	0.7563	0.5740
0.3	2	0	0.1872	120717312	115.12MB	0.1336	0.7810	0.5784
0.3	2	1	0.1945	120532992	114.95MB	0.1365	0.7737	0.5730
0.3	2	2	0.1809	120709120	115.12MB	0.1327	0.7488	0.5865
0.3	2	3	0.1789	119623680	114.08MB	0.1410	0.7825	0.5840
0.3	2	4	0.1870	120524800	114.94MB	0.1075	0.7781	0.5977
0.3	3	0	0.2797	120696832	115.11MB	0.1395	0.7924	0.5938
0.3	3	1	0.2721	120823808	115.23MB	0.1439	0.8039	0.6280
0.3	3	2	0.2517	120717312	115.12MB	0.1401	0.7634	0.5881
0.3	3	3	0.3222	119455744	113.92MB	0.1423	0.7800	0.5839
0.3	3	4	0.2731	120623104	115.04MB	0.1244	0.7878	0.6048
0.3	4	0	0.3841	120844288	115.25MB	0.1449	0.7936	0.6061
0.3	4	1	0.3622	120672256	115.08MB	0.1465	0.8119	0.6385
0.3	4	2	0.3479	119844864	114.29MB	0.1399	0.7646	0.5935
0.3	4	3	0.3623	120655872	115.07MB	0.1451	0.7808	0.6041
0.3	4	4	0.3896	121126912	115.52MB	0.1281	0.7896	0.6115

Table A.3: Results from the runs of asv-benchmark on the Olivetti faces dataset

Appendix B

Fashion-MNIST Benchmarks

The following appendix details the results of an additional run of LAP for a bigger set of fractions and sweeps than that used in the asv-benchmarks. In particular, fraction values of [0.02, 0.05, 0.1, 0.2, 0.3, 0.5] are tested with sweeps [1, 2, 3, 4, 5, 6, 8, 10]. The test was performed several times, using different random seeds [1, 10, 100, 123, 999], as it affects the candidate exemplar selection process. Runs were omitted if the value of fraction * sweeps was greater than 1.

The test was made in order to compare different configurations of LAP parameters, and how it affects the results on a subset of the Fashion-MNIST dataset. Standard clustering metrics were recorded, treating the original classification of images as the true values, along with the number of clusters generated by each configuration.

		V-measure		Rand index		MIS		AMIS		NMIS		Fowlkes Mallows		n_clusters	
		mean	std	mean	std	mean	std	mean	std	mean	std	mean	std	mean	std
frac	sweeps														
0.02	1	0.282	0.059	0.712	0.124	0.562	0.155	0.275	0.058	0.282	0.059	0.283	0.033	8.800	3.271
	2	0.334	0.077	0.754	0.090	0.686	0.183	0.326	0.077	0.334	0.077	0.316	0.056	10.400	1.517
	3	0.335	0.069	0.765	0.081	0.692	0.162	0.328	0.070	0.335	0.069	0.310	0.056	9.600	1.140
	4	0.329	0.068	0.770	0.085	0.684	0.174	0.323	0.068	0.329	0.068	0.306	0.045	9.400	1.949
	5	0.304	0.038	0.749	0.072	0.619	0.111	0.298	0.037	0.304	0.038	0.288	0.032	9.200	1.924
	6	0.310	0.037	0.746	0.067	0.622	0.098	0.304	0.037	0.310	0.037	0.304	0.034	8.800	1.789
	8	0.320	0.050	0.746	0.066	0.641	0.112	0.314	0.050	0.320	0.050	0.312	0.043	8.800	1.789
	10	0.311	0.049	0.747	0.066	0.626	0.114	0.304	0.049	0.311	0.049	0.303	0.039	9.000	0.707
0.05	1	0.288	0.044	0.732	0.068	0.578	0.127	0.281	0.044	0.288	0.044	0.284	0.025	9.200	2.387
	2	0.297	0.037	0.749	0.077	0.615	0.123	0.290	0.037	0.297	0.037	0.280	0.024	10.000	2.121
	3	0.271	0.035	0.714	0.085	0.549	0.105	0.263	0.035	0.271	0.035	0.271	0.022	10.200	1.483
	4	0.272	0.039	0.717	0.084	0.550	0.118	0.264	0.039	0.272	0.039	0.275	0.021	10.000	1.581
	5	0.288	0.058	0.706	0.058	0.572	0.134	0.280	0.059	0.288	0.058	0.289	0.041	10.200	0.447
	6	0.285	0.057	0.708	0.060	0.572	0.138	0.277	0.057	0.285	0.057	0.285	0.039	10.600	0.548
	8	0.319	0.061	0.745	0.061	0.655	0.143	0.311	0.062	0.319	0.061	0.308	0.048	11.000	1.225
	10	0.324	0.059	0.751	0.058	0.665	0.135	0.316	0.060	0.324	0.059	0.308	0.049	10.400	1.517
0.10	1	0.308	0.041	0.762	0.072	0.633	0.120	0.302	0.041	0.308	0.041	0.299	0.034	9.400	1.673
	2	0.333	0.044	0.781	0.046	0.695	0.088	0.326	0.045	0.333	0.044	0.315	0.042	10.600	1.517
	3	0.323	0.057	0.763	0.058	0.663	0.130	0.316	0.058	0.323	0.057	0.308	0.047	9.800	1.304

Continued on next page

		V-measure		Rand index		MIS		AMIS		NMIS		Fowlkes Mallows		n_clusters	
		mean	std	mean	std	mean	std	mean	std	mean	std	mean	std	mean	std
frac	sweeps														
	4	0.323	0.054	0.755	0.051	0.660	0.126	0.316	0.055	0.323	0.054	0.312	0.046	9.800	1.643
	5	0.309	0.043	0.738	0.045	0.625	0.096	0.302	0.044	0.309	0.043	0.299	0.041	9.800	1.643
	6	0.308	0.042	0.746	0.042	0.631	0.095	0.300	0.043	0.308	0.042	0.297	0.042	10.400	1.817
	8	0.322	0.034	0.774	0.030	0.680	0.068	0.315	0.035	0.322	0.034	0.299	0.040	11.000	1.225
	10	0.348	0.044	0.799	0.029	0.753	0.107	0.340	0.044	0.348	0.044	0.318	0.041	11.800	1.095
0.20	1	0.322	0.048	0.768	0.066	0.664	0.125	0.316	0.048	0.322	0.048	0.311	0.040	9.400	1.342
	2	0.314	0.047	0.762	0.056	0.665	0.128	0.306	0.048	0.314	0.047	0.289	0.036	11.600	0.548
	3	0.316	0.045	0.759	0.051	0.669	0.129	0.307	0.045	0.316	0.045	0.285	0.030	11.600	1.673
	4	0.310	0.048	0.758	0.043	0.658	0.122	0.302	0.048	0.310	0.048	0.283	0.037	12.000	1.581
	5	0.358	0.048	0.803	0.048	0.778	0.122	0.350	0.048	0.358	0.048	0.330	0.038	12.200	1.304
0.30	1	0.311	0.065	0.739	0.072	0.637	0.152	0.303	0.066	0.311	0.065	0.299	0.052	10.600	2.510
	2	0.300	0.041	0.733	0.039	0.621	0.093	0.291	0.042	0.300	0.041	0.282	0.036	11.800	0.837
	3	0.328	0.053	0.770	0.045	0.704	0.125	0.319	0.055	0.328	0.053	0.299	0.048	13.000	1.581
0.50	1	0.349	0.044	0.786	0.052	0.743	0.111	0.341	0.045	0.349	0.044	0.319	0.040	11.400	0.548
	2	0.349	0.048	0.802	0.054	0.770	0.131	0.341	0.049	0.349	0.048	0.313	0.040	12.600	0.548

Table B.1: The table presents the performance comparison of Leveraged Affinity Propagation (LAP) algorithm using different fraction and sweep parameters. The results are averaged over 5 random seeds, and the mean and standard deviation are reported for multiple evaluation measures, including V-measure, Rand Index, Info Score, Adjusted Mutual Info Score, Normalized Mutual Info Score, Fowlkes Mallows Score and the number of clusters.