UNIVERSITÉ DE NANTES

MÉMOIRE STAGE

Master 2 : Mathématiques et applications (Algébre et géométrie)

Faculté de Sciences - Université de Nantes

Introduction à l'Analyse topologique des données et étude de l'algorithme ToMATo

Auteur : Nil Garcés de Marcilla Escubedo

Tuteur : Bertrand Michel
Écrit à : Laboratoire de Mathématiques Jean Leray

Nantes, 28 juin 2020

Remerciements

Ce mémoire n'aurait pu se terminer sans l'aide des personnes qui m'ont accompagné pendant ces derniers mois si particuliers.
Je voudrais tout d'abord adresser toute ma gratitude à M. Bertrand MICHEL, professeur et chercheur à l'Université de Nantes, qui, en tant que tuteur de mémoire, m'a guidé pendant tout le travail et m'a aidé en tout moment pour continuer à avancer. Grâce toutes nos visioconférences et sa confiance et patience, surtout quand ma motivation était un peu plus faible, j'ai finalement appris plus de choses que je ne le pensais au début de ce travail.
J'aimerais aussi remercier M. Marc GLISSE, chercheur à INRIA-Saclay, qui a été disponible à tout moment pour répondre à mes interrogations sur le code de façon très précise et efficace.
Je désire aussi remercier les professeurs de l'Université de Nantes en général, qui m'ont fourni les outils nécessaires à la réussite de mes études universitaires en France.
Finalement, je voudrais exprimer ma reconnaissance envers ma famille, mes amis et collègues. Ces derniers m'ont apporté leur soutien moral et intellectuel tout au long de ma démarche. Plus spécialement, je remercie les résidents de la cité universitaire et alentours. Durant la période de confinement ils sont devenus comme une partie à part entière de ma famille.

Abstract

The goal of this memoir is to expose and manipulate some modern concepts and tools in the Data Science domain.
In the central part of the work, some basic notions and results of the emerging field of Topological Data Analysis (TDA) are explored, notably persistent homology and persistence diagrams, together with some stability results. Several effective algorithms to compute the homology groups and the persistent homology of a (filtration of a) simplicial complex are also given.
Together with that, following a more general approach, a brief survey of the Machine learning paradigm and some clustering algorithms are exposed in the first two chapters.
In the last chapter, the recently developed clustering method ToMATo is studied. This algorithm relies heavily on some of the concepts explained in the previous chapters. The theoretical study of this method is then followed by a more practical section in which programming takes the leading role : a (rather visual) exploration (in Python) of the implementation of this algorithm in the GUDHI library is carried out, as well as a little guide to understand its parameters and functionalities.

2010 Mathematics Subject Classification : 55Uxx, 62H30, 68T01

Table des matières

0. Introduction

L'importance des domaines de la science des données (Data Science) et l'apprentissage automatique (Machine Learning) continue à croître dans le monde avec l'évolution technologique de notre époque. Dans ce contexte, de nouvelles idées et méthodes sont constamment développées pour traiter, analyser et exploiter la grande quantité de données qui nous entourent. Seule une bonne formulation mathématique peut justifier la pertinence des nouveaux algorithmes et son implémentation.

Parmi le grand nombre de méthodes existantes dans la science des données, nous trouvons les méthodes de *clustering*, ou segmentation des données. L'objectif de ces dernières est de trouver des sous-groupes "naturels" dans notre information de départ. Dans chaque groupe, les données seraient, sous une définition à préciser, plus "similaires" entre elles. Les problèmes de clustering sont particulièrement difficiles à traiter en raison de leur nature exploratoire et non-supervisée. Ainsi, la convenance d'un algorithme de clustering ou d'un autre dépend en grande partie des caractéristiques des données d'entrée.

En parallèle à l'apparition de nouvelles techniques pour réaliser des tâches spécifiques, différentes approches générales à la science des données sont aussi développées. Le domaine émergent de l'Analyse topologique des données (TDA en anglais) étudie les ensembles de données en utilisant des idées de la topologie et de la géométrie. Ce domaine illustre pleinement ce phénomène. L'intérêt pour ce champ d'étude, avec plein de nouveaux concepts et résultats, augmente de plus en plus, et actuellement de nombreux algorithmes très efficaces s'appuient sur la base théorique de la TDA.

Le récemment développé algorithme de clustering ToMATo (*Topological Mode Analysis Tool*) fait partie de ces nouvelles méthodes. Au coeur de son fonctionnement apparaissent les notions d'homologie persistante et les diagrammes de persistance, très habituels dans la TDA. De plus, une implémentation de cet algorithme a été récemment ajouté dans la librairie GUDHI, l'un des outils de programmation de référence dans ce nouveau domaine. Il semble donc raisonnable de faire une exploration théorique mais aussi pratique / informatique de cette technique qui vient d'être conçue.

Structure de la mémoire

Tout d'abord, dans le premier chapitre, nous verrons une exposition générale du domaine de l'apprentissage automatique ; plusieurs concepts de base récurrents sont introduits dans cette partie. Nous essaierons aussi de répondre à trois questions significatives : qu'est-ce que le Machine Learning, pourquoi est-il utile, et comment les ordinateurs arrivent à "apprendre" et à améliorer sa performance. Les références principales de cette partie sont [3], [7] et [9].

Puis, au chapitre 2, nous développerons qu'est-ce que le clustering, ainsi que des notions clés dans l'étude de cette technique. Le contenu de ce chapitre est important afin de mieux comprendre l'algorithme ToMATo, ses innovations et ses particularités. Une partie des algorithmes les plus communs seront expliqués, et nous verrons aussi comment traiter les données pour appliquer au mieux ces méthodes. Les références les plus importantes de cette section sont principalement [9] et la documentation en ligne de la librairie Scikit-Learn [11], [12].

Le troisième chapitre constitue la partie la plus dense et mathématique du mémoire. Certains des concepts les plus fondamentaux de l'Analyse topologique des données y sont exposés. Nous

verrons d'abord comment construire un complexe simplicial sur un nuage de points, et pourquoi cette construction est intéressante. Ensuite, nous ferons l'inférence homologique de ces complexes simpliciaux. Cela implique une connaissance des notions d'homologie persistante ainsi que des diagrammes de persistance. Plusieurs algorithmes sont détaillés pour étudier ces informations dans la pratique. Nous terminerons le chapitre en étudiant la stabilité persistante des fonctions, qui est fondamentale pour justifier de façon théorique la performance de l'algorithme ToMATo. Plusieurs références ont été utilisés dans cette partie, dont : [2], [3], [6], [10] et [5].

Finalement, au chapitre 4 nous explorerons l'algorithme ToMATo. Nous nous appuierons sur les idées exposées aux chapitres précédents. D'un point de vue théorique, il convient d'expliquer certaines constructions de graphes sur des nuages de points, et comment estimer une hypothétique fonction de densité f à partir d'un échantillonnage. Les sources d'informations les plus importantes dans cette section sont [4] et sa version simplifié, ainsi que [1] et [9].
Pour la pratique, nous avons produit un notebook de référence (en anglais) de l'implémentation de l'algorithme, qui vient d'être ajouté à la librairie GUDHI. L'objectif de cette partie était de tester la performance de cette implémentation, ainsi que mieux connaître le langage de programmation Python et certains outils habituels pour réaliser l'analyse de données.

1. Introduction à l'apprentissage automatique

1.1. Machine Learning : idée générale et premiers exemples

Même si le concept de *Machine Learning* (en français, *apprentissage statistique*, ou *apprentissage automatique*) a explosé en popularité ces dernières années, souvent on perçoit un peu de confusion en ce qui concerne la signification authentique de ce terme. Cette confusion est davantage aggravée lorsque d'autres expressions comme *data science*, *artificial intelligence* ou *data mining*, étroitement liées à la première, apparaissent sur la table. Inévitablement, avec le développement intensif de tous ces nouveaux domaines, un nouveau jargon est apparu, et il est indispensable de bien connaître les subtiles différences entre les mots pour encadrer convenablement les problèmes et les explications .

En termes généraux, le Machine Learning (ML) est le domaine d'étude qui cherche à donner aux ordinateurs la capacité d'apprendre sans être explicitement programmés, en utilisant des données (d'ici son interaction avec la *data science*) et/ ou des expériences antérieures. En voyant cette définition, qui d'ailleurs n'est pas très concrète, deux questions émergent naturellement par rapport au terme "apprendre" : qu'est-ce que cela veut dire, exactement, et comment obtient-on cet apprentissage ? De plus, il est naturel de se demander dans quelles situations ou pourquoi le Machine Learning peut être la meilleure option à considérer. Ce sont précisément ces trois questions que nous nous proposons de répondre tout de suite.

La première des trois est possiblement la plus générale : en effet, cette apprentissage peut prendre plusieurs formes, qui peuvent varier énormément en fonction du problème de départ. Ainsi, la manière la plus rapide de se faire une idée de quoi "apprendre" signifie véritablement est de regarder quelques exemples de situations où le Machine Learning s'est avéré être très efficace. Ces exemples vont apparaître plusieurs fois toute au long du chapitre :

— La classification du mail dans *spam* et *no-spam*. Dans ce cas, l'idée est de développer un algorithme pour choisir, en considérant plusieurs aspects (fréquence de quelques mots spécifiques, longueur, structure générale,...), si un courriel contient des informations qui nous intéressent ou pas. Donc, en somme, nous voulons que l'ordinateur apprenne à *classer* une série d'éléments.

— La prédiction de la valeur d'une maison, en sachant quelques aspects comme sa taille, emplacement et d'autres caractéristiques, ainsi comme celles des immeubles à proximité, y compris leur valeur. Dans cet exemple, on assume que tous ces facteurs peuvent être utilisés pour construire un modèle "réaliste" qui donne notre prix approximatif. Le résultat final du processus est une quantité, qui peut donc varier continuellement. Nous avons ici un problème typique de *régression*.

— Dans un magasin, on peut essayer de détecter des groupes de clients similaires selon leurs achats, ou selon leur genre, par exemple. En sachant cela, on peut élaborer des offres ou politiques commerciales plus dirigées vers ces groupes pour augmenter les ventes. Ici, nous avons de nouveau un problème de classification, mais d'une nature assez différente, car les groupes ne sont pas connus à priori, et ils pourraient même ne pas exister d'une façon évidente. Nous parlerons plus de ce type de procédure, appelé *clustering*, en peu plus tard.

— Le développement d'une application digitale de reconnaissance vocale. Par exemple, un programme de smartphone capable d'écrire et chercher sur Internet toute combinaison de mots

qu'on lui dicte. Dans ce cas, il y a aussi de quelque sorte un problème de classification (après tout, l'objectif du programme est de bien identifier chaque mot prononcé), mais les nuances du langage et la complexité de la prononciation humaine situent le défi beaucoup plus loin que d'autres problèmes de classification standards. Ce type de programmes qui cherchent à imiter (et dépasser) le comportement humain et réaliser des tâches plus complexes font partie de ce qu'on appelle *intelligence artificielle*. Ce domaine, de plus en plus actif et prometteur, a ses propres algorithmes et mécanismes, comme par exemple l'utilisation de réseaux neuronaux (neural networks) ou l'apprentissage par renforcement (reinforcement learning).

Donc, nous voyons que la variété de ce qu'on a appelé "apprentissage" est riche et considérable et, en fait, il y a beaucoup plus de situations et applications possibles : diagnostic médical guidé par ordinateur, séquençage d'ADN, vision par ordinateur,...
En tout cas, la plupart des méthodes et algorithmes ont pour objectif de classer des éléments, de prévoir ou d'estimer des résultats ou des valeurs pour prendre des décisions, trouver des relations entre variables, ou une combinaison de ces options.

1.2. Motivations

Mais pourquoi appeler tout cela apprentissage ? Après tout, tous les programmes informatiques visent la simplification des tâches et aider avec les calculs et la prise de décisions...
La différence essentielle avec le Machine Learning est la manière dans laquelle ces programmes arrivent à effectuer ces tâches : rappelons qu'un élément clé de notre brève définition est "sans être explicitement programmés, en utilisant des données et/ ou expériences antérieures". Avant d'expliquer, dans la section suivant, les idées générales qui présentent comment nous pouvons arriver à faire cela, il est naturel de se demander en premier lieu quelles sont les motivations de le faire.

Prenons-nous le premier exemple du mail. Si nous devions programmer nous-mêmes un détecteur de mail *spam* (pour bien le distinguer et séparer du mail "bon"), la manière la plus naturel d'agir serait, d'abord, d'étudier un peu ses caractéristiques générales : quels types de mots ou d'expressions apparaissent le plus souvent dans ce type de courriels et ses fréquences en comparaison avec le mail ordinaire, sa longueur approximative, des régularités dans le nom ou dans l'adresse de l'émetteur, etc. Finalement, avec toute cette information, il faudrait programmer une par une les conditions ou les seuils à dépasser pour le considérer comme un courrier indésirable.

Ce n'est pas une chose facile ni rapide à faire ! Même si nous réussissons à trouver de bonnes conditions pour distinguer les deux types de mail, nous obtiendrions une liste énorme de règles à considérer. Ainsi, le résultat final serait un code très long et complexe : pas pratique à programmer ni facile à maintenir, modifier ou mettre à jour. Un algorithme plus "machine learning" chercherait lui-même les caractéristiques clés en comparant des exemples des deux types de courrier et associerait les poids convenables pour bien les classer.

Dans le dernier exemple de la reconnaissance vocale, la complexité d'un hypothétique programme codé à la main devient encore plus évidente : la quantité d'information et la variabilité dans un fichier audio est tellement énorme qu'il est simplement impossible d'analyser explicitement tous les cas où il sonne une "s" ou une "u". Seulement après avoir exposé à un bon algorithme milliers d'enregistrements des différents mots, nous pouvons espérer qu'il arrivera à les distinguer correctement.

Un autre avantage des algorithmes de Machine Learning est que souvent on peut les concevoir pour qu'ils soient adaptables à de possibles actualisations ou accroissements des données. Cette caractéristique peut être très utile dans toutes les situations, y comprises celles des exemples antérieurs.

Finalement, nous pouvons nous servir de toutes ces techniques et procédures pour améliorer l'apprentissage humaine même. En effet, quelques algorithmes ML peuvent être inspectés pour voir ce qu'ils ont appris, et ainsi mieux comprendre des corrélations et tendances non reconnues précédemment. Utiliser les techniques ML avec cet objectif s'appelle *data mining*.

1.3. Méthodologie et différents types de systèmes d'apprentissage

Alors, comment obtenir cet apprentissage ? Le principe de tous les systèmes de Machine Learning consiste dans le fait que la majorité des paramètres sont trouvés en utilisant des données et des exemples déjà existants, qu'on appelle "données d'entraînement" (*training data*). En tout cas, c'est pratique de classer ces systèmes de plusieurs manières en considérant quelques-unes de ses différences méthodologiques fondamentales.

Possiblement la distinction la plus important au niveau méthodologique, car il affecte notamment les possibles algorithmes à appliquer, est celle d'apprentissage *supervisé* et *non-supervisé* (et quelques types "intermédiaires"). Cette classification prend en compte dans quelle mesure les données sur lesquelles on construit l'algorithme contiennent déjà des informations certaines, i.e. on a une connaissance préalable des types de solutions qu'on devrait obtenir.

Dans l'apprentissage supervisé, possiblement le plus naturel et intuitif, les données d'entraînement incluent les solutions souhaitées ; elles sont "étiquetées" (*labelled*). Par exemple, dans les deux premières situations expliquées précédemment, nous construirions le classificateur de mail à partir d'exemples de courriels "bons" et "spam" ; pareillement, on estimerait le prix de la maison en utilisant un modèle qui prend en compte les caractéristiques, mais aussi les prix (i.e. la "solution", ils sont donc étiquetées) des différentes maisons à proximité. Ces caractéristiques utilisées pour construire le modèle s'appellent *features*, ou *predictors*.
En résumé, les systèmes d'apprentissage supervisé sont conçus pour donner les résultats attendus sur les données d'entraînement, que nous connaissons. Les problèmes de régression et de classification en groupes spécifiques sont des exemples de ce type d'apprentissage.

Dans l'apprentissage non-supervisé, les données sur lesquelles nous travaillons ne sont pas étiquetées, et il n'y a pas une façon directe de vérifier ou mesurer la performance du système. Ce type d'apprentissage est plutôt lié à la visualisation des données et son exploration : corrélations inattendues, groupes avec des similitudes, détection des données mauvaises ou bizarres (*outliers*),... Par exemple, les méthodes de clustering sont de nature non-supervisée, y compris notre algorithme ToMATo, dont nous parlerons plus tard. Dans ce type d'apprentissage il y aurait aussi ces algorithmes de visualisation qui essaient de représenter les donnés en 2D et 3D en préservant au maximum sa structure. Finalement, nous y ajouterions aussi toutes les procédures de réduction de la dimensionnalité, qui ont pour objectif simplifier les données sans perdre trop d'information, par exemple en combinant plusieurs features corrélées entre elles.

D'autres types d'apprentissage sous ce critère seraient l'apprentissage semisupervisé, qui combine les deux types antérieurs, ou l'apprentissage par renforcement. Dans ce dernier, assez lié au

domaine de l'intelligence artificielle, l'algorithme observe continuellement les données et l'environnement, et sélectionne et réalise des actions qui peuvent être récompensées ou pénalisées ; au fil du temps, il apprend lui-même les stratégies le plus efficaces pour obtenir les meilleures récompenses.

Une autre manière de classer les systèmes ML est selon sa capacité d'adaptation aux nouvelles données. Les algorithmes qui ont besoin de tout l'ensemble de données pour être construits correctement font partie de ce qu'on appelle apprentissage *offline* ; ceux qui peuvent incorporer de nouvelles données et apprendre progressivement, une propriété en général désirable pour sa flexibilité et réduction du coût de calcul, sont de type *online*.

Finalement, une autre classification décisive au niveau méthodologique est celle qui prend en compte comment le système ML se généralise aux nouveaux cas ; c'est-à-dire de quelle façon on mesure sa performance en tant que prédicteur, avec de nouvelles observations.
Dans l'apprentissage *basé sur des instances*, l'algorithme apprend les exemples par coeur et étudie les nouveaux cas en utilisant une "mesure de similitude". Cette dernière compare quantitativement les nouveaux cas avec les données d'entraînement, afin de les étudier. En revanche, dans l'apprentissage *basé sur des modèles*, on essaie de construire un bon modèle ou formule à partir des exemples pour faire des prédictions. Normalement, dans le design de ce modèle, on utilise une fonction d'"aptitude" (*fitness function*, ou *cost function*) pour étudier quantitativement sa convenance sur les données d'entraînement.
Dans les deux cas, il faut toujours garder à l'esprit que tout ensemble de données d'entrée contient inévitablement du *bruit* : elles sont partiellement aléatoires, et l'information n'est jamais transparent. Donc, ajuster la flexibilité du modèle en fonction de chaque cas est toujours essentiel.

1.4. Problèmes et challenges du Machine Learning

En somme, dans tout processus d'apprentissage statistique nous trouvons deux étapes : la sélection d'un algorithme convenable et l'entraînement postérieur avec des données. Naturellement, il faut faire attention à ces deux choses si nous voulons obtenir un apprentissage effectif. Certains défis ou aspects à prendre en compte en ce qui concerne cela seraient :

- *Quantité insuffisante de données :* Dans la majorité des algorithmes, il faut disposer de beaucoup de données pour entraîner correctement le modèle et le faire fonctionner. En général, on a besoin de milliers d'exemples, ou des millions dans les problèmes les plus complexes. Dans certaines situations, il est possible de combiner ou extraire des nouvelles données à partir de celles déjà existantes, pour en avoir plus. Plusieurs études montrent que des algorithmes très différents peuvent accomplir des niveaux de succès similaires en utilisant suffisamment de données.

- *Données d'entraînement non représentatives :* Afin d'obtenir de bonnes généralisations, les données d'entraînement doivent être représentatives des nouveaux cas qu'on cherche à généraliser ; sinon, les prédictions du modèle difficilement s'ajusteront aux valeurs réelles. Par exemple, le caractère d'un modèle pour calculer quelque spécificité d'un pays peut changer largement en fonction de la richesse des pays utilisés pour le concevoir ; il faudrait se servir des pays avec un niveau économique similaire. Le même principe s'applique pour prédire les résultats d'une élection à partir des sondages.
  Quand les données utilisées ne sont pas représentatives, même si nous en avons une grande quantité, il s'agirait ici d'un "biais d'échantillonnage" (*sampling bias*).

- *Données de mauvaise qualité :* Naturellement, si les données d'entraînement contiennent beaucoup d'erreurs, outliers et bruit, les algorithmes auront plus de problèmes pour trouver des "patterns" et atteindre ses objectifs. Donc, en général, c'est recommandable d'investir du temps à détecter et écarter les outliers et traiter les valeurs manquantes ou incomplètes (*data cleaning).*

- *Features non pertinents :* Indépendamment de l'algorithme, celui-ci seulement apprendra si les données utilisées pendant l'étape d'entraînement ont un véritable lien avec ce que nous voulons estimer. Le complexe processus d'obtenir un ensemble de features pertinents s'appelle *feature engineering.* Il comprend, entre autres : sélectionner les features les plus utiles et écarter les autres, les combiner pour en obtenir des nouvelles d'une façon plus compacte (étroitement lié à la réduction de la dimensionnalité), en créer d'autres à partir de nouvelles données,...

- *Overfitting et underfitting :* Ces deux phénomènes, plus liés à l'algorithme lui-même qu'à la nature des données, se produisent quand le modèle obtenu se base trop ou respectivement trop peu sur les données d'entraînement.
  Tout algorithme de Machine Learning essaie de trouver des régularités dans les données, mais celles-ci possèdent aussi de manière naturelle une variabilité qui peut empêcher l'algorithme de bien se généraliser à de nouveaux cas si nous nous y basons trop. Par exemple, il est presque toujours possible de trouver une fonction polynomiale qui passe pour n'importe quelle quantité de points dans $\mathbb{R}^2$ si son dégrée est suffisamment élevé (i.e. si nous augmentons suffisamment les dégrées de liberté), mais un modèle si "courbé" ne sera possiblement pas le meilleur à prédire de futures observations. En somme, l'overfitting se produit quand le modèle est trop complexe par rapport au bruit et à la quantité de données d'entraînement.
  Quelques possibles solutions dans ce cas seraient : recueillir plus de données, réduire le bruit des données (i.e. réparer les erreurs dans les données et écarter les outliers) ou simplifier le modèle, chose que nous pouvons faire en utilisant moins de paramètres, en considérant moins de features ou en "contraignant" le modèle. Ce dernier approche, appelé *regularization*, contient plein de méthodes et techniques : l'idée de base est d'utiliser des paramètres supplémentaires dans l'algorithme (les *hyperparamètres*), indépendants du modèle, fixés d'abord et constants pendant l'entraînement, qui "limitent" en quelque sorte la liberté des paramètres du modèle. Trouver de bons hyperparamètres est l'une des parties importantes de construire un bon système de Machine Learning.

  Le underfitting est le problème contraire : il se produit quand le modèle est trop simple pour bien apprendre la structure sous-jacente des données. Trois stratégies pour améliorer rapidement cette situation sont : admettre plus de paramètres dans le modèle, réduire les contraintes s'il y en a, ou augmenter la pertinence des features.

- *Essai et validation :* Pour étudier l'efficacité du modèle, une bonne pratique consiste à diviser les données disponibles en plusieurs sous-groupes complémentaires et les entraîner, mesurer et vérifier les uns contre les autres. En somme, nous trouvons trois types de ces groupes : les données d'entraînement (*training set*), à partir duquel on construit le modèle ou mesure de similitude ; les données de validation (*validation set*), qui servent pour modifier le modèle ou les hyperparamètres jusqu'à obtenir une performance désirable ; et les données de vérification (*test set*), pour se faire une idée de l'erreur de généralisation (i.e. sa performance avec de nouveaux cas).
  On appelle *cross-validation* la méthode, très commune à pratiquer, qui consiste à faire cette procédure plusieurs fois avec tout l'ensemble des données pour mieux choisir le modèle et les hyperparamètres.

2. Plus en détail : algorithmes de clustering habituels

2.1. Considérations générales

Le *cluster analysis*, ou segmentation des données, a pour objectif le regroupement d'un ensemble d'éléments en sous-groups ou *clusters*. Ainsi, dans chaque cluster, les éléments sont plus "proches" entre eux à la différence des éléments classés dans des clusters différents. Dans cette branche du Machine Learning, à caractère non supervisé et exploratoire, les algorithmes cherchent à établir si les données peuvent être divisées dans des groupes différents avec des propriétés suffisamment distinctes. L'algorithme ToMATo, récemment développé, fait aussi partie de ces méthodes. Nous aborderons plus en détail cet algorithme dans le chapitre quatre.

La question fondamentale dans le cluster analysis est comment nous mesurons ce "degrée de similarité" (ou dissimilarité) entre les données, donc c'est la définition sur laquelle les algorithmes se basent.
Un approche assez flexible consiste à utiliser ce qu'on appelle une *matrice de proximité*. Avec un ensemble de N éléments (ordonnées), $\{x_1, \ldots, x_N\}$, on construit une matrice D de dimension $N \times N$, où le coefficient d_{ij} mesure quantitativement la proximité ou similarité de l'élément i à l'élément j. En général, plus le numéro est faible, plus des similitudes sont remarqués. De ce fait, la plupart des algorithmes assument $d_{ii} = 0,\ \forall i \in [1, N]$. De plus, certains algorithmes imposent notamment que la matrice soit symétrique ; sinon, D peut toujours être remplacée par $(D + D^T)/2$. Pour travailler avec "dissimilitudes", on peut toujours convertir tous les valeurs avec une fonction monotone décroissante convenable.

Une des situations le plus habituelles est celle où chaque élément x_i consiste en p attributs de nature quantitative. Si ces attributs sont de nature qualitative (ou catégorique), on peut parfois les convertir facilement en numéros : par exemple, si nous avons une variable qualitative ordonnée avec M options, nous pouvons utiliser les valeurs $\dfrac{i - \frac{1}{2}}{M},\ i = 1, \ldots, M$, toutes entre 0 et 1. Si la variable n'est pas ordonnée et peut prendre M différentes valeurs, il faut préciser le "niveau de différence" entre les paires de valeurs en utilisant une matrice (comme évoqué dans le paragraphe précédent) : ses entrées, normalement 1s sauf 0s à la diagonale, jouent le rôle des d_j que nous expliquerons tout de suite.

Supposons que nous disposons de plusieurs données numériques $x_{ij},\ i \in [1, N], j \in [1, p]$ (les cas catégoriques ont déjà été traités). En s'appuyant sur ces données, on construit une notion de "dissimilarité" entre les valeurs du j-ème attribut de deux éléments différents, $d_j(x_{ij}, x_{i'j})$. Le choix le plus commun pour d_j est la distance au carré,

$$d_j(x_{ij}, x_{i'j}) = (x_{ij} - x_{i'j})^2.$$

D'autres options existent aussi, comme par exemple la différence absolue $|x_{ij} - x_{i'j}|$, qui pénalise moins les grandes différences. Les résultats peuvent varier considérablement en fonction de la distance choisie.

Puis, nous définissons la "mesure de dissimilarité totale" $d(x_i, x_{i'})$ entre deux éléments en combinant ces p dissimilarités individuelles. Bien que la somme est l'option la plus naturelle à considérer, nous gagnons en flexibilité en travaillant avec une moyenne pondéré

$$d(x_i, x_{i'}) = \sum_{j=1}^{p} w_j \cdot d_j(x_{ij}, x_{i'j}) \ , \quad \sum_{j=1}^{p} w_j = 1.$$

Cette dernière permet d'ajuster un poids convenable à chaque attribut (ces poids dépendent nécessairement de la nature du problème et des données concrètes). Pour bien adapter ces poids, il est important de remarquer que l'influence du j-ème attribut sur la dissimilarité totale $D(x_i, x_{i'})$ dépend de sa contribution relative à la moyenne des dissimilarités totales entre toutes les paires d'éléments de l'ensemble,

$$\bar{D} = \frac{1}{N^2} \sum_{i=1}^{N} \sum_{i'=1}^{N} d(x_i, x_{i'}) = \sum_{j=1}^{p} w_j \cdot \bar{d}_j,$$

où

$$\bar{d}_j = \frac{1}{N^2} \sum_{i=1}^{N} \sum_{i'=1}^{N} d_j(x_{ij}, x_{i'j})$$

est la dissimilarité moyenne du j-ème attribut. Ainsi, l'influence relative de la j-ème variable est $w_j \cdot \bar{d}_j$, et fixer $w_j \sim 1/\bar{d}_j$ (ou directement $w_j = 1/\bar{d}_j$, standardisé plus tard) donne à chaque attribut la même influence sur la dissimilarité totale.

Bien que cette dernière option semble appropriée, elle peut aussi être contre-productive. En effet, souvent les attributs ne contribuent pas de la même manière à la notion de similitude : certaines différences entre les valeurs peuvent refléter plus de dissimilarité que d'autres dans le contexte du problème, et devraient donc avoir plus de poids. Pour cela, il est important de préciser correctement toutes ces variables, ainsi que la fonction de similitude, chose qui dépend dans une large mesure du problème spécifique. En fait, tous ces paramètres peuvent avoir plus d'importance que l'algorithme lui-même pour réussir avec le clustering.
Finalement, il est aussi important de prêter attention à bien traiter les données manquantes (*missing values* en anglais) dans un ou plus des attributs. On peut faire cela en utilisant une moyenne (ou quelque autre valeur, processus appelé "imputation statistique"), en utilisant une nouvelle catégorie qualitative "missing", en omettant quelques dissimilarités concrètes ou en écartant directement ces éléments.

2.2. Algorithmes de clustering combinatoires

Pour résumer, nous trouvons trois types d'algorithmes de clustering :

- Les *algorithmes combinatoires* travaillent directement sur les données, sans avoir aucun type de modèle probabiliste sous-jacent, et assignent directement chaque élément à un group.
- Les *modèles de mélange* supposent que les données constituent un échantillon *i.i.d* d'une population décrite par une fonction de densité. Cette fonction de densité est caractérisée par un modèle paramétrique formé par un mélange/ somme de plusieurs fonctions de densité (habituellement gaussiennes) : chacune de ses fonctions décrirait un cluster.
- Les algorithmes *mode-seeking* ("chercheurs de modes"), aussi appelés *bump hunters*, ont une approche non paramétrique et tentent d'estimer directement les différentes modes (i.e. maximums locaux) d'une hypothétique fonction de densité de base. Les éléments les plus proches de chaque mode définissent ainsi les clusters individuels.

Ceux du premier type sont spécialement employés pour leur simplicité. Avec les données $\{x_1, \ldots, x_N\}$, un numéro préspécifié de clusters $K < N$ est choisi, chacun étiqueté par un numéro $k \in \{1, \ldots, K\}$. On assigne après à chaque élément i de l'ensemble un cluster $C : \{1, \ldots, N\} \to \{1, \ldots, k\}$, $i \mapsto C(i) = C_k$, en essayant de minimiser une fonction "de perte" qui prend en compte les dissimilarités $d(x_i, x_{i'})$ entre les données. Une fonction de perte naturelle à considérer serait

$$W(C) = \sum_{k=1}^{K} \sum_{i,i' \in C_k} d(x_i, x_{i'}), \tag{1}$$

qui quantifie de quelle manière les observations mises dans le même cluster sont proches entre elles. Il est facile de voir que minimiser $W(C)$ est équivalent a maximiser

$$B(C) = \sum_{k=1}^{K} \sum_{i \in C_k} \sum_{i' \notin C_k} d(x_i, x_{i'}),$$

car $T = \sum_{i,i'}^{N} d(x_i, x_{i'}) = W(C) + B(C)$ est constant.

Nous pourrions penser que cela réduit le problème au calcul de la valeur de la fonction de perte sur toutes les possibles combinaisons, mais dans la pratique le nombre d'allocations possibles pour tous les éléments augmente très rapidement avec N et k. De ce fait, tout algorithme de clustering efficace étudie seulement une fraction très petite des attributions $k = C(i)$ possibles, avec l'objectif d'identifier un sous-ensemble susceptible de contenir l'optimale, ou au moins une correspondance assez bonne.

La stratégie se base généralement sur ce qu'on appelle un "greedy descent" itérative : une partition initiale est choisie et, à chaque pas, les attributions sont changées de sorte que la valeur du critère est améliorée par rapport à l'antérieure. L'algorithme se termine par une partition lorsque aucune amélioration est possible.
Ces algorithmes, travaillant avec un sous-ensemble très petit des combinaisons possibles, convergent toujours à un maximum local, qui peut être très sub-optimal en comparaison avec le maximum global.

2.2.1. K-means clustering

Il est un des algorithmes les plus populaires en raison de sa vitesse et sa simplicité. Il a aussi des applications importantes dans la compression des images et signaux (*vector quantization*).

Cet algorithme suppose que toutes les variables sont de type quantitative, et il prend la distance euclidienne habituelle au carré, $d(x_i, x_{i'}) = \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2 = ||x_i - x_{i'}||^2$, pour mesurer la dissimilarité entre les observations. Avec ces conditions, nous remarquons que (1) est égal à

$$W(C) = \sum_{k=1}^{K} N_k \sum_{i \in C_k} ||x_i - \bar{x}_k||^2, \tag{2}$$

où $\bar{x}_k = (\bar{x}_{1k}, \ldots, \bar{x}_{pk})$ est le vecteur moyen associé aux observations du cluster k, et N_k est son nombre d'éléments.

Du fait que la moyenne des $\{y_1, \ldots, y_m\}$ minimise la fonction $f(y) = \sum_{i=1}^{m} (y_i - y)^2$, nous

pouvons obtenir une méthode itérative descendant pour résoudre

$$C^* = \min_C \sum_{k=1}^{K} N_k \sum_{i \in C_k} ||x_i - \bar{x}_k||^2, \tag{3}$$

notre problème original, en considérant le problème d'optimisation plus général

$$C^* = \min_{C, \{m_k\}_1^K} \sum_{k=1}^{K} N_k \sum_{i \in C_k} ||x_i - m_k||^2. \tag{4}$$

L'algorithme est le suivant :

Algorithm 1: *K-means clustering*

Input: $\{x_1, \ldots, x_N\}$ observations quantitatives (p features chacune)
K numéro de clusters souhaité
Output: Pour chaque observation, une étiquette $k \in [1, K]$ (cluster assigné)

1 On fait une première attribution $C(i)$ pour chaque observation, aléatoire ou avec unes moyennes déjà établies.
2 Avec notre partition C, on minimise la variance totale du clustering (4), obtenant ainsi les moyennes $\{m_1, \ldots, , m_k\}$ associées à chaque cluster.
3 Avec ces valeurs $\{m_1, \ldots, , m_k\}$, on minimise davantage (4) en assignant à chaque observation le cluster avec la moyenne la plus proche : $C(i) = \arg\min_{1 \leq k \leq K} ||x_i - m_k||^2$
4 On répète 2 et 3 jusqu'à ce que les attributions C ne changent plus.

Étant donné que à les étapes 2 et 3 la quantité (4) diminue, la convergence de la méthode est assurée. Néanmoins, normalement on atteint un maximum local sub-optimal. De ce fait, c'est une bonne idée de courir l'algorithme avec différentes partitions initiales et prendre le meilleur résultat final.

Nous pouvons généraliser l'idée du clustering K-means à distances différentes à l'euclidienne et features pas nécessairement quantitatives si nous travaillons directement avec les dissimilarités $d(x_i, x_{i'})$. Pour cela, nous pouvons utiliser l'algorithme décrit avant en changeant le m_k : au lieu de la moyenne des éléments du cluster k, nous prenons un de ces éléments ; en particulier, l'élément x_k qui minimise $\sum_{i \in C_k} d(x_k, x_i)$. Cette nouvelle méthode, qui s'appelle *clustering K-medoids*, a aussi un coût informatique considérable, et n'est souvent pas réalisable exhaustivement.

2.2.2. Clustering hiérarchique

Contrairement au clustering K-means/ K-medoids, qui part d'un nombre de clusters K préréglé et les cherchent, les méthodes de clustering hiérarchiques produisent une représentation "en échelle" qui passe pour tous les nombres possibles, et où les clusters à chaque niveau sont crées en unifiant ou divisant ceux du niveau inférieur. De cette façon, il est possible de voir plus facilement quel est le "bon" numéro de clusters de l'ensemble. Naturellement, il est encore nécessaire d'établir une "mesure de similitude" entre groupes, basée sur les dissimilarités entre paires d'éléments.

Il y a deux stratégies principales pour ce type de clustering : l'agglomérative (*bottom-up*), où nous commençons avec un cluster pour chaque observation et nous les unifions par paires à mesure que l'algorithme court ; et la divisive (*up-bottom*), qui part par un seul cluster et ensuite les divise

en deux peu à peu. Dans le deux cas, chaque niveau de la hiérarchie représente un regroupement spécifique des données en clusters disjoints, et la hiérarchie entière les différents "seuils" où ils apparaissent.

Ces agglomérations/ divisions binaires récursives peuvent être représentées sous forme d'arbre, qui commence avec une seule racine (le cluster avec toutes les données) et, à la fin, a une feuille pour chaque élément. De plus, une partie important de ces méthodes ont la propriété de la "monotonie", c'est à dire, la dissimilarité entre clusters (qui se mesure quantitativement) augmente de manière monotone à mesure qu'on les unifie. Ainsi, l'arbre peut être dessiné de sorte que les bifurcations entre les branches se produisent à des hauteurs qui reflètent la durée de tous les clusters de manière proportionnelle. Ce type de représentation graphique, assez complète et informative sur les données, s'appelle *dendrogram*.

Néanmoins, ces dendrograms sont assez sensibles aux données et à les particularités de la méthode choisie, et ils imposent sur les données une structure hiérarchique qui pourrait ne pas exister. Donc, plus qu'une "carte" infaillible de la structure des données elles-mêmes, le dendrogram devrait être vu plutôt comme une carte de la structure du clustering des ces données, obtenues avec un algorithme et une métrique spécifiques.

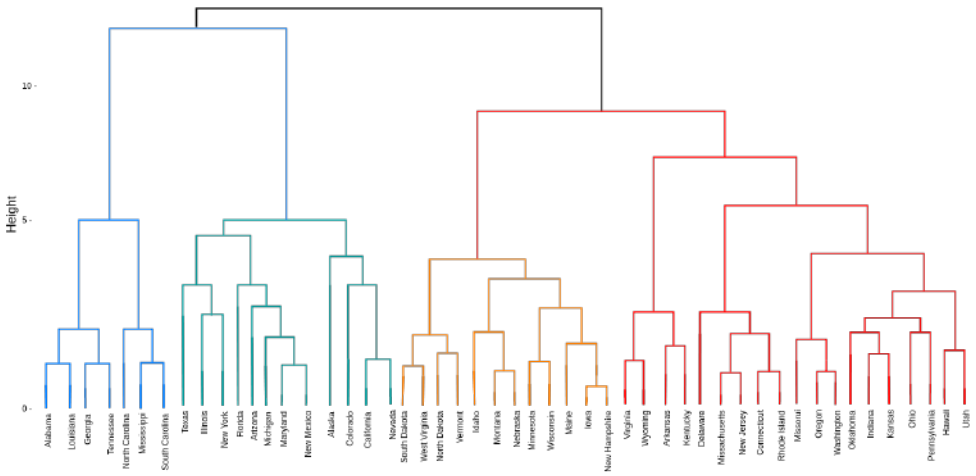


FIGURE 1 – Exemple de dendrogram (où on regroupe les états des États-Unis, critère inconnu)

Stratégies agglomératives

Ces méthodes commencent avec un singleton cluster pour chaque élément. Puis, à chacun des $N-1$ pas, les deux groupes les plus "proches" sont fusionnés, et on perd un cluster au niveau suivant.

Naturellement, la notion de "proximité" entre les groupes doit être définie en considérant les dissimilarités entre les paires d'observations. Soient G et H deux de ces groupes. Nous remarquons plusieurs options pour définir la dissimilarité $d(G,H)$:

— Le *saut minimum* (*single linkage* en anglais) est défini pour $d_{SL}(G,H) = \min_{i\in G, i'\in H} d_{ii'}$.

— Le *saut maximum* (*complete linkage* en anglais) se définie comme $d_{CL}(G,H) = \max_{i\in G, i'\in H} d_{ii'}$.

— Le *lien moyen* (*group average* en anglais) est défini pour $d_{GA}(G,H) = \dfrac{1}{N_G N_H}\sum_{i\in G}\sum_{i'\in H} d_{ii'}$.

En général, si les données montrent une structure claire, avec des clusters bien séparés les uns des autres et suffisamment compacts (i.e. ses éléments sont proches entre eux en comparaison avec ceux de clusters différents), les trois méthodes produisent des résultats similaires. Développons maintenant les particularités de chaque méthode :

Le type *saut minimum* a seulement besoin que deux éléments de groupes différents soient proches pour les fusionner, indépendemment des autres éléments ; cela résulte souvent en clusters peu compacts.
Le type *saut maximum* est le pôle opposé ; deux groupes G et H sont proches seulement si tous les éléments dans son union sont relativement similaires, et les clusters sont plus compactes. Pourtant, cela peut causer aussi une relaxation du "principe de proximité" : un élément assigné dans un cluster peut être beaucoup plus proche des éléments d'autres clusters que ceux de son propre groupe.
Le type *lien moyen* permet un compromis entre ces deux extrêmes, mais il est aussi plus dépendant des valeurs spécifiques des $d_{ii'}$, à la différence des deux autres, qui dépendent uniquement de son ordre.

Stratégies divisives

Ces méthodes commencent avec toutes les données dans un unique groupe, et divisent à chaque itération un cluster existant en deux clusters plus petits.

Même si elles sont moins étudiées que les méthodes agglomératives, on peut toujours diviser n'importe quel cluster en appliquant une méthode combinatoire, comme K-means avec $K = 2$, à chaque itération. Cependant, en général ce processus ne produit pas une séquence de clusters avec la propriété de la monotonie nécessaire pour la représenter correctement en forme de dendrogram.

Un algorithme qui satisfait cela serait celui-ci :

Algorithm 2: *Clustering hiérarchique divisive monotone*

Input: $\{x_1, \ldots, x_N\}$ observations quantitatives
Les dissimilarités $d_{ii'}$ entre toutes les paires d'observations
Output: Une séquence hiérarchique de clusters

1 On met toutes les observations dans un unique cluster, G.
2 On trouve l'élément i dans G avec la dissimilarité moyenne avec les autres éléments de G, $\frac{1}{N_G}\sum_{j\in G} d_{ij}$, la plus élevée. Cet élément sera le premier membre d'un deuxième cluster H.
3 On prend l'élément de G qui a la distance moyenne avec les éléments de G moins la distance moyenne avec les éléments de H la plus grande et le transfère à H.
4 On continue à faire cela jusqu'à ce que cette différence devienne négative. En ce moment, il n'y a plus d'observations dans G qui sont, en moyenne, plus proches à celles du H qu'à celles de son groupe G. Nous avons alors deux nouveaux clusters.
5 Nous continuons de répéter 2, 3 et 4 avec un cluster présent, nouveau ou pas, jusqu'à obtenir N singleton clusters. Pour choisir le groupe suivant a diviser, deux critères utiles seraient :
   — Le cluster C avec le diametre $D_C = \max_{i,i'\in C} d_{ii'}$ le plus grand.
   — Celui avec la dissimilarité entre éléments moyenne, $\bar{d}_C = \frac{1}{N_C^2}\sum_{i,i'\in C} d_{ii'}$, la plus grande.

3. Introduction à l'Analyse topologique des données

3.1. Idée générale et motivation

L'Analyse topologique des données (*Topological Data Analysis* en anglais, souvent nommée *TDA*), commence à se développer dans les années 2000 à partir de quelques travails dans la topologie appliquée et la géométrie algorithmique. Ce champ d'étude cherche à explorer et étudier les bases de données en utilisant des techniques et idées typiques du domaine de la topologie. Cette nouvelle approche de la science de données, qui s'est déjà avérée très utile dans plusieurs contextes, a pour objectif mieux comprendre la "forme" d'un ensemble de données. Cette question peut être spécialement compliquée quand on travaille en dimensions élevées, et avec des donnés incomplètes ou avec une forte présence de bruit.

En résumé, la TDA essaie de fournir des méthodes mathématiques, statistiques et algorithmiques pour révéler, analyser et utiliser des structures géométriques et topologiques non évidents dans un ensemble des données. Notamment, un de ses outils principales est celui de l'*homologie persistante*, une adaptation de l'homologie pour nuages de points, qui a besoin d'une solide formulation théorique et mathématique.

Le schéma de déroulement habituel en TDA est :

1. L'input est généralement un ensemble fini de points avec quelque type de similarité ou distance définie entre eux. Cette distance peut venir induite pour un hypothétique espace ambiant (par exemple, $\mathbb{R}^d$) ou être définie intrinsèquement entre paires de points, en fonction du cas.
2. Quelque type de structure géométrique de nature traitable et algorithmique est construite sur ces points, avec l'objectif de faire plus évidents quelques de ses caractéristiques. Souvent, nous faisons cela en utilisant un ou plusieurs *complexes simplicials*, qui peuvent être vus comme une généralisation des graphes en dimensions plus élevées.
3. Nous extrayons cette information géométrique et topologique en utilisant différents méthodes, et nous étudions sa pertinence et stabilité par rapport à possibles perturbations des données ou présence de bruit. Cette information est après souvent visualisée et combinée avec d'autres descripteurs pour guider les prochaines étapes de l'analyse des données ou tâches de ML.

Notre algorithme ToMATo fait usage de certains des concepts de ce nouveau champ d'étude, notamment de l'homologie persistante et les diagrammes de persistance. Donc, l'objectif de cette partie du travail est d'introduire avec rigueur et généralité les fondements de la TDA et les bases mathématiques de l'homologie persistante.

3.2. Complexes simpliciales, recouvrements et le Théorème du Nerf

Étant donné que la plupart des concepts de la topologie et la géométrie sont associés à des espaces continus, une pratique habituelle dans le TDA est de "connecter" de quelque sorte les données (représentées comme points) qui sont proches les unes des autres. On formalise souvent cette notion de proximité en utilisant une distance entre points, qui peut être définie entre paires directement (espace métrique discrète) ou en plongeant les données dans un espace métrique plus grand (typiquement, $\mathbb{R}^d$).

En tout cas, après avoir connecté les données proches, nous obtenons un graphe de voisinage, qui permet déjà appliquer plusieurs méthodes d'analyse. Pour aller au-delà de la connectivité,

nous pouvons associer pas seulement paires mais aussi $(k+1)$-tuples de points proches entre eux. Nous obtenons ainsi un *complexe simplicial*, qui permet identifier de nouvelles caractéristiques topologiques, comme cycles, "trous" et leurs généralisations en haute dimension.

Définition 3.1. *Soit $\mathbb{X} = \{x_0, \ldots, x_k\} \subset \mathbb{R}^d$ $(k+1)$ points affines linéairement indépendants. Le simplexe k-dimensionnel $\sigma = [x_0, \ldots, x_k]$ généré pour $\mathbb{X}$ est l'enveloppe convexe de $\mathbb{X}$. Les points originales sont ses* sommets*, et les simplexes générés pour les sous-ensembles de ces points sont les* faces *de σ.*

Remarque 3.2. Formellement, les sous-ensembles d'un simplexe sont appelés ses n-faces, où n est sa cardinalité moins 1. Cependant, pour les cas 0 et 1 normalement on utilise les mots *sommets* et *arêtes* respectivement, et le mot *face* pour le cas 2 et en général.

Définition 3.3. *Un* complexe simplicial géométrique *$K \subset \mathbb{R}^d$ est une collection de simplexes telle que :*

1. *Toute face d'un simplexe de K est aussi un simplexe de K.*
2. *Toute intersection de deux simplexes de K est vide ou une face commune aux deux.*

Encore plus généralement, un complexe simplicial abstrait *avec des sommets V est une collection K de sous-ensembles finis de V telle que que les éléments de V appartient à K et, pour tout élément σ de K, tout sous-ensemble de σ appartient aussi à K.*

Clairement, on peut utiliser la dernière définition, de nature plus combinatoire, pour étudier un complexe simplicial géométrique, mais la direction inverse fonctionne aussi : on peut mettre tout complexe simplicial abstrait dans $\mathbb{R}^d$ pour quelque d, et le considérer comme un subespace avec la topologie induite. C'est cette structure, appelée *réalisation géométrique* de K, qui permet utiliser sans problèmes plein de notions géométriques et topologiques sur K.

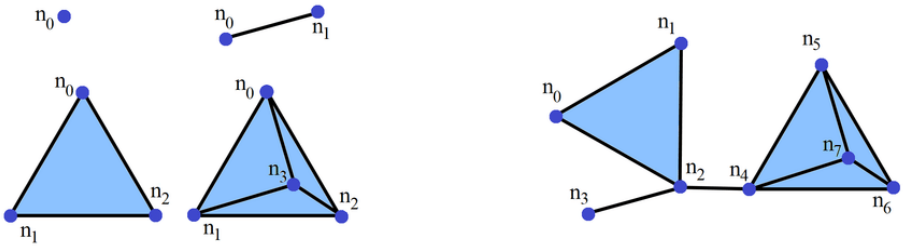


FIGURE 2 – Simplexes de dimension 0,1,2 et 3, et exemple d'un complexe simplicial géométrique

Définition 3.4. *La dimension d'un simplexe est simplement sa cardinalité moins 1. La dimension d'un complexe simplicial est la dimension plus grande parmi les simplexes qui le constituent.*

Remarque 3.5. Un graphe est un complexe simplicial de dimension 1.

Étant donnée un ensemble de points X (imaginons que dans un espace métrique (M, d)), nous pouvons construire des complexes simplicials au-dessus de plusieurs manières. Deux des constructions les plus habituelles seraient :

1. *Complexe de Vietoris-Rips, $Rips_\alpha(X)$:* La généralisation immédiate de la notion de graphe de voisinage. C'est le complexe simplicial qui a pour ensemble de faces les simplexes $[x_0, \ldots, x_k]$ qui satisfont $d(x_i, x_j) \leq \alpha$ pour tout $0 \leq i, j \leq k$.

2. *Complexe de Čech, $Cech_\alpha(X)$:* Étroitement lié au Vietoris-Rips complexe, c'est le complexe simplicial formé pour les simplexes $[x_0, \ldots, x_k]$ qui satisfont que l'intersection des $k+1$ boules $\overline{B}(x_i, \alpha)$ n'est pas vide.

Remarque 3.6. Même si X est un ensemble fini de points dans R^d, $Rips_\alpha(X)$ et $Cech_\alpha(X)$ n'admettent pas toujours une réalisation géométrique dans R^d, donc ses dimensions peuvent être plus élevées.

Remarque 3.7. C'est facile de voir qu'on a toujours $Rips_\alpha(X) \subseteq Cech_\alpha(X) \subseteq Rips_{2\alpha}(X)$, où les inclusions peuvent être strictes. Si $X \subset \mathbb{R}^d$, $Cech_\alpha(X)$ et $Rips_{2\alpha}(X)$ ont le même squelette 1-dimensionnel, i.e. le même ensemble de sommets et arêtes.

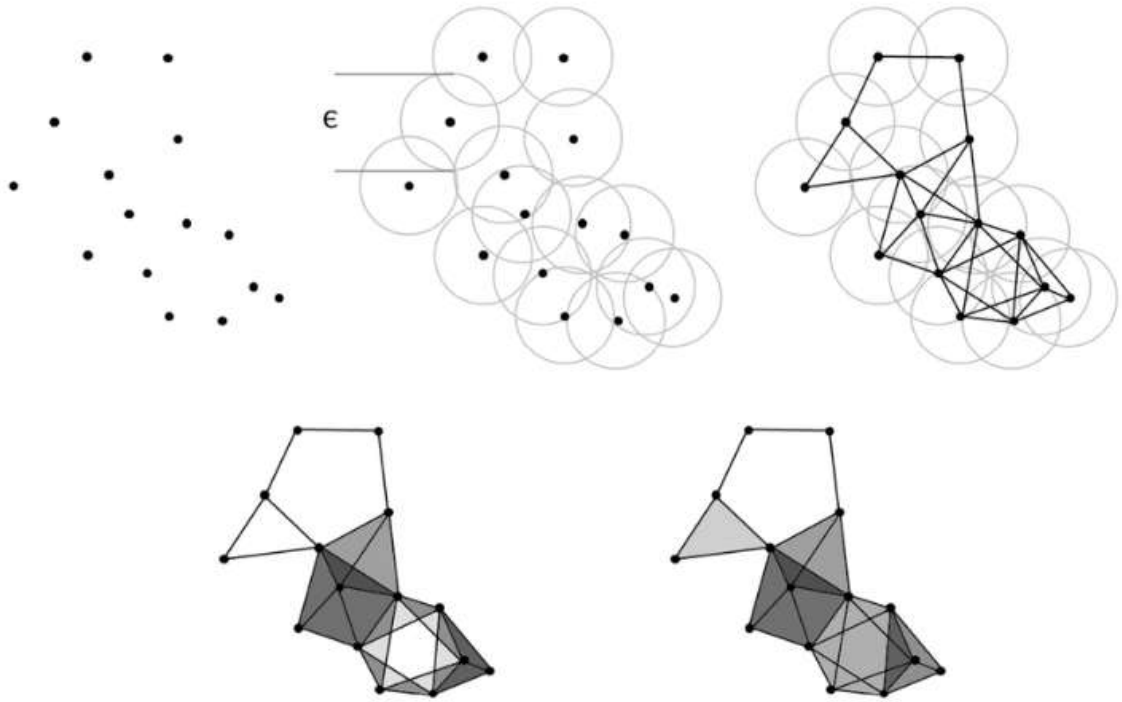


FIGURE 3 – Construction des complexes de $Cech_{\frac{\varepsilon}{2}}$ (en bas à gauche) et de $Rips_e$ (en bas à droite). La troisième image montre comme les deux complexes ont le même squelette 1-dimensionnel.

En fait, le complexe de Čech est un cas particulier d'une construction de complexes plus générale en utilisant des recouvrements :

Définition 3.8. *Soit M un espace topologique (ou un ensemble, en général). Un* recouvrement U *de M est une famille de sous-ensembles de M, $U = (U_i)_{i \in I}$, qui satisfont $\cup_{i \in I} U_i = M$. Le* nerf *d'un recouvrement U de M est le complexe simplicial abstrait $C(U)$ qui a U_i comme sommets et les faces*

$$\sigma = [U_{i_0}, \ldots, U_{i_k}] \in C(U) \iff \cap_{j=0}^{k} U_{ij} \neq \emptyset$$

De cette façon, $Cech_\alpha(X)$ est le nerf du recouvrement $U = (\overline{B}(x_i, \alpha))_{x_i \in X}$ de l'ensemble $M = \cup_{x_i \in X} \overline{B}(x_i, \alpha)$, qui contient évidemment l'ensemble de points original. Mais un recouvrement d'un ensemble de données ne doit pas forcement être basé sur des boules centrées sur elles ;

par exemple, nous pourrions aussi en avoir un en faisant une subdivision des données en groupes de points (non nécessairement disjoints) avec des propriétés similaires.

En tout cas, le nerf d'un recouvrement offre une description de nature combinatoire assez globale et compacte de la relation entre les sous-ensembles du recouvrement en considérant ses plusieurs intersections. Le Théorème du Nerf (Nerve Theorem) est un résultat de topologie algébrique qui lie, avec quelques assomptions, la topologie du nerf d'un recouvrement avec la topologie du recouvrement lui-même. Ses implications dans l'Analyse topologique des données sont remarquables, et même si nous ne le démontrerons pas, le but de la dernière partie de cette section est introduire les notions nécessaires pour bien comprendre ce qu'il dit.

Dans la topologie, normalement nous considèrons deux espaces topologiques X et Y comme égales quand ils sont *homéomorphes*, i.e. nous pouvons trouver deux applications continues et bijectives $f : X \to Y$ et $g : Y \to X$ qui satisfont $g \circ f = id_X$ et $f \circ g = id_Y$. Cependant, dans plusieurs situations, aussi dans la TDA, la notion d'homémorphisme est trop rigide, et souvent il est convenable d'étudier des similitudes entre espaces topologiques un peu plus faibles. C'est ici où l'idée de l'homotopie apparaît :

Définition 3.9. *Soit X et Y deux espaces topologiques. Deux applications continues $f_0, f_1 : X \to Y$ sont* homotopiques *s'il existe une application continue $H : X \times [0,1] \longrightarrow Y$ telle que, $\forall x \in X$, $H(x,0) = f_0(x)$ et $H(x,1) = f_1(x)$. Dans ce cas, on écrit $f_0 \simeq f_1$.*
On dit que X et Y sont des espaces topologiques homotopiquement équivalents *si on peut trouver deux applications $f : X \to Y$ et $g : Y \to X$ tels que $g \circ f \simeq id_X$ et $f \circ g \simeq id_Y$. Dans ce cas, on écrit $X \simeq Y$.*

La notion d'équivalence homotopique est plus faible que celle d'homéomorphisme, donc deux espaces homéomorphes sont toujours homotopiquement équivalents, mais le réciproque n'est pas vrai. En tout cas, l'intérêt principal derrière l'homotopie est que nous pouvons définir des objets (souvent de nature algébrique) sur les espaces topologiques qui sont effectivement des invariants homotopiques, c'est-à-dire qui sont conservés entre des espaces topologiques homotopiquement équivalents. Les exemples les plus notables seraient les *groupes d'homotopie* et les *groupes d'homologie* (singulaire, simpliciale). On parlera plus en détail de l'homologie dans la section suivante.

Définition 3.10. *Un espace X est* contractile *s'il est homotopiquement équivalent à un point.*

Exemple 3.11. Tout boule dans $\mathbb{R}^d$, ouverte ou fermée, est contractile. Plus généralement, tout sous-ensemble convexe X dans $\mathbb{R}^d$ est contractile. En effet, si on suppose $0 \in X$, il y a les applications $f : X \to \{0\}$, $x \mapsto 0$, et $g : \{0\} \to X$, $0 \mapsto 0$. Clairement $f \circ g \simeq id_{\{0\}}$ (en fait, $f \circ g = id_{\{0\}}$), et $g \circ f \simeq id_X$, avec l'application continue $H : X \times [0,1] \longrightarrow X$, $H(x,t) = t \cdot x$.

Un recouvrement ouvert est celui où tous les éléments de la famille sont ouverts. Un recouvrement ouvert fini où tous les éléments et intersections entre éléments sont contractiles satisfait le résultat suivant, souvent nommé le Théorème du Nerf :

Théorème 3.12. (Théorème du Nerf) *Soit $U = (U_i)_{i \in I}$ un recouvrement ouvert fini d'un sous-ensemble $X \subseteq \mathbb{R}^d$ tel que toute intersection des U_i's est vide ou contractile. Alors X et $C(U)$ sont homotopiquement équivalents.*

Ainsi, on a que le nerf défini par un "bon" recouvrement de X est homotopiquement équivalent à X, ce qui est remarquable pour des applications ; en effet, normalement un complexe simplicial possède une nature beaucoup plus traitable algoritmiquement qu'un espace topologique général.

En tout cas, le complexe de Čech se construit avec des boules fermés centrées sur chaque donnée, donc le recouvrement n'est pas ouvert dans ce cas. Heureusement, la version suivante du théorème est aussi vraie :

Théorème 3.13. (Théorème du Nerf pour un recouvrement convexe) *Soit $X \subseteq \mathbb{R}^d$ une union finie d'ensembles fermés convexes $F = (F_i)_{i \in I}$ dans $\mathbb{R}^d$. Alors X et $C(F)$ sont homotopiquement équivalents.*

De ce fait, on obtient que, en effet, si X est un nuage de points dans $\mathbb{R}^d$, alors $Cech_\alpha(X)$ est homotopiquement équivalent à l'union des boules $\bigcup_{x \in X} \overline{B}(x, \alpha)$.

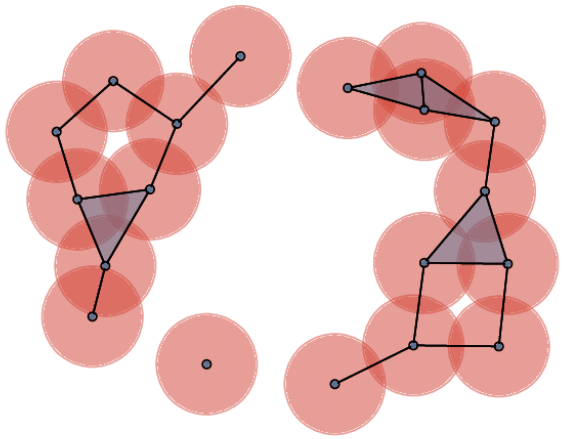


FIGURE 4 – Représentation de comment l'union des boules et le nerf associé (i.e. le complexe de Čech des données) sont homotopiquement équivalents.

3.3. Inférence homologique

Résumons la situation jusqu'à ici : pour aller plus loin dans notre étude de nos ensembles de données/ nuages de points, nous avons vu comment construire quelques structures géométriques au-dessus, appelées complexes simplicials, de nature plus algorithmique. Après, nous avons exposé le Théorème du Nerf, donc nous avons vu que, quand $\mathbb{X}$ est un ensemble de points dans $\mathbb{R}^d$, $Cech_\alpha(\mathbb{X})$ est homotopiquement équivalent à l'union de boules $\cup_{x \in \mathbb{X}} \overline{B}(x, \alpha)$.

Dans toute situation avec des données numériques (supposons dans R^d), et d'un point de vue statistique, il y a fondamentalement deux questions qui nous intéressent : d'une part, il y a l'"espace d'échantillonnage" de nos données, i.e. dans quelle région $M \subseteq \mathbb{R}^d$ toutes les possibles données "vivent" ; de l'autre part, il y a la mesure de probabilité μ sur cette région M, qui encode quelles zones de M sont plus probables d'avoir plus de points, et de quelle manière les données se regroupent. Normalement, nous supposons que M, le *support* de μ, est compact, et que nos données $\mathbb{X} = \{x_1, \ldots, x_n\}$ ont été échantillonnées i.i.d. en suivant μ.

Évidemment, pour mieux comprendre nos données, faire des prédictions, etc., nous sommes intéressés à connaître μ et la "forme" de son support M. Le processus qui essaie de mieux caractériser

M s'appelle "reconstruction géométrique", et un schéma habituel pour inférer cette information géométrique et topologique est :

1. Nous recouvrons $\mathbb{X}$ avec des boules $B(x, \alpha)$; sous certaines conditions de régularité sur M, nous pouvons lier la topologie de cette union avec celle de M.
2. Certaines propriétés topologiques de M sont inférées à partir du nerf de l'union de ces boules, en utilisant le Théorème du Nerf.

Des résultats mathématiquement rigoureux et importants existent avec cette approche de reconstruction. Néanmoins, ce n'est pas toujours possible, ni souhaitable, d'essayer de reconstruire complètement la forme de base à partir de nos données. De plus, dans le schéma que nous venons d'exposer, nous voyons que le choix du rayon des boules, qui souvent n'est pas du tout évident, joue un rôle clé dans les résultats obtenus.

Une autre manière de travailler les données est d'essayer de trouver des invariants topologiques plus faibles, mais plus faciles d'inférer. C'est ici que le concept d'*homologie*, un outil déjà classique dans la topologie algébrique, entre en scène. Plus notamment, nous pouvons faire usage de l'*homologie simplicial* sur nos complexes simplicials pour mieux les comprendre et, finalement, élaborer davantage cette information homologique pour développer ce qu'on appelle *homologie persistante*, qui garde une trace de comme l'homologie des complexes simplicials obtenues évolue en variant le rayon. Une manière de représenter visuellement une bonne partie de toute cette information est avec ce qu'on appelle un *diagramme de persistance*.

3.3.1. Homologie simpliciale et nombres de Betti

L'idée intuitive derrière de l'homologie en général est de traiter et formaliser algébriquement la notion de "trou", ou "boucle", dans de différentes contextes mathématiques, notamment dans les espaces topologiques. Pour toute dimension n, les "trous" n-dimensionnels sont représentés par un espace vectoriel H_n, et sa dimension serait le numéro de trous "indépendants" de ce type. Par exemple, H_0 représente les composantes connexes de notre espace, H_1 les "boucles unidimensionnelles", H_2 les "cavités 2-dimensionnelles", etc.

Le premier type de théorie d'homologie qui a été développé, il y a environ un siècle, est l'homologie simpliciale, qui se construit sur les complexes simplicials. Sur ces objets, c'est relativement simple d'imaginer la notion de trou k-dimensionnel. Même si les concepts que nous exposerons ensuite sont sensés avec tout corps k, nous travaillerons désormais avec $k = \mathbb{Z}/2\mathbb{Z} = \mathbb{Z}_2$, plus intuitif à niveau géométrique, et qui simplifie les arguments ; sinon, il faudrait considérer une orientation sur les sommets/ faces de notre complexe, et les formules deviendraient plus compliquées.

Soit K un complexe simplicial de dimension d :

Définition 3.14. *Une* n-chaîne *est une somme formelle de simplexes n-dimensionnelles de K ; c'est à dire, si $\{\sigma_1, \ldots, \sigma_p\}$ sont les n-faces de K, une n-chaîne c est une expression du type*

$$c = \sum_{i=0}^{p} \lambda_i \sigma_i, \ \ \textit{avec } \lambda_i \in \mathbb{Z}_2$$

Pour chaque n, l'ensemble des n-chaînes $C_n(K)$ a une structure évidente de $\mathbb{Z}_2$-espace vectoriel, où l'ensemble des n-faces de K est une base de $C_n(K)$. Les chaînes avec des coefficients dans $\mathbb{Z}_2$ ont une interprétation géométrique simple : du fait que toute n-chaîne peut être uniquement écrite comme $c = \sigma_{i_1} + \cdots + \sigma_{i_m}$, c représente simplement l'union des n-simplexes σ_{i_j}.

Définition 3.15. *Le* bord $\partial(\sigma)$ *d'un* n*-simplexe* σ *est la somme de ses* $(n-1)$*-faces. Donc, pour le* n*-simplexe* $\sigma = [v_0, \dots, v_n]$*, on obtient la* $(n-1)$*-chaîne*

$$\partial(\sigma) = \sum_{i=0}^{n} [v_0, \dots, \hat{v_i}, \dots, v_n]$$

où $[v_0, \dots, \hat{v_i}, \dots, v_n]$ *est le* $(n-1)$*-simplexe formé pour les sommets originels sauf* v_i*.*

Le bord d'un n-simplexe nous donne les $(n-1)$-faces qui le constituent. Le bord ainsi défini sur les simplexes de K peut être étendue de manière naturelle à une (plusieurs) fonction entre les $C_i(K)$. Même si on devrait les distinguer ∂_i, souvent on écrit simplement ∂ pour éclaircir le texte :

Définition 3.16. *La* fonction bord *est l'application linéaire définie par*

$$\begin{aligned} \partial : C_n(K) &\longrightarrow C_{n-1}(K) \\ c \quad &\mapsto \partial(c) = \sum_{\sigma \in c} \partial(\sigma) \end{aligned}$$

La propriété plus fondamentale de ∂ est celle-ci :

Proposition 3.17. $\partial\partial = \partial \circ \partial = 0$

Démonstration. Puisque la fonction bord est linéaire, il suffit de le vérifier simplement pour un seul simplexe $\sigma = [v_0, \dots, v_n]$, de dimension n :

$$\partial\partial\sigma = \partial\Big(\sum_{i=0}^{n} [v_0, \dots, \hat{v_i}, \dots, v_n]\Big) = \sum_{i=0}^{n} \partial[v_0, \dots, \hat{v_i}, \dots, v_n] =$$

$$= \sum_{j<i} [v_0, \dots, \hat{v_j}, \dots, \hat{v_i}, \dots, v_n] + \sum_{j>i} [v_0, \dots, \hat{v_i}, \dots, \hat{v_j}, \dots, v_n] = \sum_{\substack{j,i=0 \\ j \neq i}}^{n} 2[v_0, \dots, \hat{v_i}, \dots, \hat{v_j}, \dots, v_n] = 0$$

□

La fonction bord définit une séquence d'applications linéaires entre les $C_i(K)$:

Définition 3.18. *Le* complexe de chaînes *associé au complexe simplicial* K *est la séquence d'espaces vectoriels et applications linéaires :*

$$\{0\} \xrightarrow{\partial} C_d(K) \xrightarrow{\partial} C_{d-1}(K) \xrightarrow{\partial} \dots \xrightarrow{\partial} C_1(K) \xrightarrow{\partial} C_0(K) \xrightarrow{\partial} \{0\}$$

Pour $k \in \{0, \dots, d\}$*, l'ensemble* $Z_k(K)$ *de* k-cycles de K *est le noyau de* $\partial : C_k(K) \to C_{k-1}(K)$ *:*

$$Z_k(K) = \{c \in C_k(K) |\ \partial(c) = 0\},$$

et l'ensemble $B_k(K)$ *de* k-bords de K *sont les chaînes qui appartient a l'image de l'application* ∂ *:*

$$B_k(K) = \{c \in C_k(K) |\ \exists\ b \in C_{k+1}(K) \text{ tel que } \partial(b) = c\},$$

De quelque sorte, Z_k encode quelles k-chaînes sont "fermées" (d'ici le nom "cycles"), et B_k quels ensembles de k-faces sont le bord d'une $(k+1)$-chaîne.
Z_k et B_k sont évidemment des sous-espaces de C_k, et en vue de la Proposition 3.17, on a toujours

$B_k \subseteq Z_k$, où la inclusion peut être stricte. Ce dernier fait motive la définition des groups d'homologie, qui essaient de trouver des "trous" dans notre complexe simplicial, i.e. des k-chaines fermées qui ne sont la frontière d'aucune $(k+1)$-chaîne du complexe :

Définition 3.19. *Le* k-ème groupe d'homologie *de* K *est l'espace vectoriel quotient*

$$H_k(K) = Z_k(K)/B_k(K);$$

ses éléments s'appellent les classes d'homologie *de* K*. Deux cycles qui appartient à la même classe d'homologie sont appelés* homologues.
La dimension $\beta_k(K)$ *de* $H_k(K)$ *s'appelle le* k-ème nombre de Betti *de* K*.*

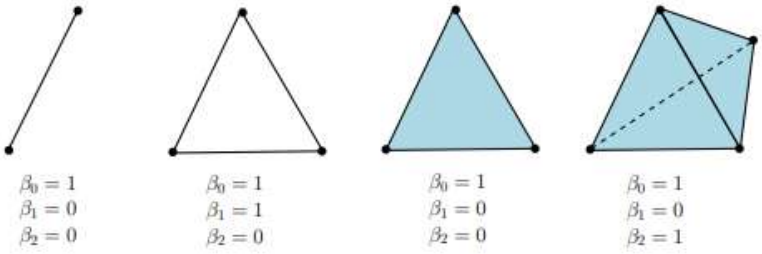


FIGURE 5 – Les nombres de Betti associés à quelques complexes simplicials simples : une arête (avec ses sommets), le périmètre d'un triangle, un triangle, et un tétraèdre.

On peut démontrer (ce qui n'est pas immédiat) que les groupes d'homologie et les nombres de Betti sont des invariants topologiques : si K_0 et K_1 sont deux complexes simplicials avec des réalisations géométriques homéomorphes, alors ses groups d'homologie sont isomorphes et ses nombres de Betti sont égales. De plus, ces résultats sont aussi vraies si les réalisations géométriques sont seulement homotopiquement équivalents.

Ces résultats sont une conséquence de l'étroite relation entre l'homologie simpliciale et une autre type d'homologie, l'*homologie singulière*, beaucoup plus générale et qui peut être définie pour tout espace topologique. En fait, on peut démontrer que les groupes d'homologie simpliciales et singulières d'un complexe simplicial sont toujours isomorphes, et le résultat est une conséquence de l'invariance homotopique de l'homologie singulière. Bien que notre intention dans ce mémoire n'est pas d'introduire l'homologie singulière, le résultat suivant, pas difficile mais sans démonstration, nous sera utile dans les pages qui viennent :

Proposition 3.20. *Soit* X *un espace topologique (resp. un complexe simplicial). Alors, la dimension du premier groupe d'homologie singuilière* $H_0(X)$ *(resp. homologie simpliciale) est égale au nombre de composantes connexes (par arcs) de* X*.*

3.3.2. Filtrations

Définition 3.21. *Une filtration d'un complexe simplicial* K *est une suite de subcomplexes* $(K_r)_{r\in T}$*, où* $T \subseteq \mathbb{R}$ *fini ou infini, telle que* $\forall r_0, r_1 \in T$*,* $r_0 \leq r_1 \implies K_{r_0} \subseteq K_{r_1}$*, et* $K = \cup_{r\in T} K_r$*. La définition peut être généralisée de la manière évidente à tout espace topologique.*

Dans des situations pratiques, les valeurs $r \in T$ souvent jouent le rôle de "paramètres d'échelle", qui ajustent la résolution du complexe. Deux filtrations habituelles dans le TDA seraient :

— *Filtrations construites sur des nuages de points :* Étant donné un sous-ensemble fini X d'un espace métrique compact (M, d), les familles de complexes $(Rips_r(X))_{r\in\mathbb{R}}$ et $(Cech_r(X))_{r\in\mathbb{R}}$ sont des filtrations. Dans ces dernières, r peut être vu comme le paramètre de résolution, où, pour $r \leq 0$, nous considérons seulement les points. Par exemple, quand X est un nuage de points à $\mathbb{R}^d$, grâce au Théorème du Nerf, $(Cech_r(X))_{r\in\mathbb{R}}$ encode la topologie de la famille d'unions de boules $X_r = \cup_{x\in X} B(x, r)$ lorsque r varie de zéro à infini.

— *Filtrations associées aux ensembles de niveau :* Étant donné un espace topologique M et une fonction $f : M \to \mathbb{R}$, la famille $M_r = f^{-1}((-\infty, r])$, $r \in \mathbb{R}$ définit une filtration. On appelle les ensembles $M_r \subseteq M$ les *ensembles de sous-niveau de* f. On peut définir également les ensembles de super-niveau de f et sa filtration associé.
Dans les cas où nous travaillons avec un complexe simplicial K, normalement la fonction est définie seulement sur son ensemble de sommets V. Nous pouvons étendre f à tout simplexe de K en prenant $f([v_0, \ldots, v_k]) = \max_{0\leq i\leq k} f(v_i)$ pour tout $\sigma = [v_0, \ldots, v_k] \in K$. Ainsi, la famille de sous-complexes $K_r = \{\sigma \in K | f(\sigma) \leq r\}$ définit la filtration associée aux ensembles de sous-niveau de f.

Avec ces deux filtrations, dans des cas réels, même si T est infini, toutes les filtrations sont construites sur des nuages de points, qui sont des ensembles finis, donc elles sont aussi finies. Par conséquent, le complexe obtenu change seulement un numéro fini de fois, ce qui facilite son étude d'un point de vue algorithmique.

Nous exposons finalement un autre type de filtration sur les complexes simpliciales, facile de calculer et très pratique au niveau algorithmique, comme nous verrons toute de suite :

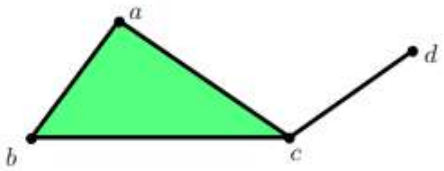
Définition 3.22. *Une* filtration de décomposition *d'un complexe simplicial K est une suite de subcomplexes*

$$\emptyset = K_0 \subset K_1 \subset K_2 \subset \cdots \subset K_{m-1} \subset K_m = K$$

qui satisfait $K_i = K_{i-1} \cup \sigma_i$, où σ_i est un simplex.

Intuitivement, dans une filtration de décomposition, nous ajoutons seulement un simplexe à chaque fois, et il faut que toutes ses faces appartiennent déjà au sous-complexe quand il est introduit.

Exemple 3.23. Avec le complexe simplicial de l'image suivant, une filtration de décomposition pourrait être $a, b, c, ab, ac, d, bc, abc, cd$.



Remarque 3.24. C'est facile de voir que toute filtration d'un complexe simplicial (y comprises les deux filtrations précédentes) peut être affinée à une filtration de décomposition : il faut seulement décomposer les nouveaux simplexes de K_i respect à K_{i-1} en sommets, arêtes, 2-faces,... et les ajouter un par un à chaque fois.

Cette dernière remarque ouvre tous les algorithmes typiques des filtrations de décomposition, comme ceux dans les sections suivants, à toute filtration. C'est à cause de ce fait que désormais nous travaillerons plutôt avec ce type de filtrations.

3.3.3. Un algorithme pour calculer les nombres de Betti

Supposons un complexe simplicial K avec une filtration de décomposition. Dans cette situation, il existe un algorithme assez simple pour calculer les nombres de Betti associés à K, seulement en gardant une trace des nombres de Betti tout au long de la filtration ; en fait, l'algorithme calcule les nombres de Betti de chaque sous-complexe de la filtration, ce qui sera important dans l'homologie persistante.

Pour bien effectuer cette méthode, c'est indispensable de détecter quand le nouveau simplexe σ_i ajouté, de dimension k, appartient à quelque k-cycle ou non, ce qui motive la définition suivante :

Définition 3.25. *Si σ_i appartient à quelque k-cycle, on dit qu'il est un simplexe* positif *; dans le cas contraire, c'est un simplexe* négatif.

Dans les sections suivantes, où plus d'algorithmes seront détaillés, nous expliquerons comment savoir si nous ajoutons un simplexe positif ou négatif. Pour l'instant, et pour expliquer l'algorithme, nous pouvons supposer que nous savons détecter quand σ_i est positif ou négatif :

Proposition 3.26. (Algorithme) : *Les nombres de Betti de K peuvent être calculés de manière inductive en faisant usage d'une filtration de décomposition.*

Démonstration. Évidemment, tous les nombres de Betti de $K_0 = \emptyset$ sont zéro.
Pour calculer les nombres de Betti de K_i, supposons que les nombres de Betti de K_{i-1} sont déjà calculés, et ajoutons le simplexe σ_i, de dimension k, pour obtenir K_i. Observons que, par définition de filtration de décomposition, σ_i ne peut pas faire partie du bord d'aucun $(k+1)$-simplexe de K_i. Par conséquent, si σ_i est contenu dans un k-cycle de K_i (i.e. positif), ce cycle n'est pas le bord d'une $(k+1)$-chaîne de K_i.

Il y a deux situations possibles :

Cas 1 : Si σ_i est positif et appartient à un k-cycle c de K_i, alors c ne peut pas être homologue à un cycle c' de K_{i-1}. En effet, dans ce cas $c + c'$ serait le bord d'une $(k+1)$-chaîne d de K_i, et comme σ_i ne peut pas appartenir à c' (donc nous venons d'introduire cet nouveau simplexe à K_i), σ_i appartient à $c + c' = \partial d$, ce qui n'est pas possible comme nous avons déjà remarqué au début de la démonstration. Par conséquent, c crée une nouvelle classe d'homologie, qui est linéairement indépendant des classes générées par les cycles de K_{i-1}, donc $\beta_k(K_i) \geq \beta_k(K_{i-1}) + 1$.
Nous pouvons voir aussi que la dimension du k-ème groupe d'homologie ne peut pas augmenter plus que 1 : si c et c' sont deux k-cycles qui contient σ_i, $c + c'$ est un k-cycle de K_{i-1}, donc c' est inclus au sous-espace linéaire généré pour $Z_k(K_{i-1})$ et c. D'ici on a que $\dim Z_k(K_i) \leq \dim Z_k(K_{i-1}) + 1$ et, comme $B_k(K_{i-1}) = B_k(K_i)$, on a $\beta_k(K_i) \leq \beta_k(K_{i-1}) + 1$.
Il reste seulement pour montrer que $B_{k-1}(K_i) = B_{k-1}(K_{i-1})$, donc $H_{k-1}(K_i)$ est le seul autre group d'homologie de K_i qui peut changer en ajoutant σ_i, et clairement $Z_{k-1}(K_i) = Z_{k-1}(K_{i-1})$. Le résultat est une conséquence directe du fait que σ_i est positif, et il appartient donc à un k-cycle c de K_i : en effet, $0 = \partial c = \partial\sigma_i + \sum \partial(\text{autres } k\text{-simplexes de } K \text{ déjà ajoutés})$, et $\partial\sigma_i$ peut être écrit comme une combinaison linéaire de bords de k-chaînes de K_{i-1}.

Cas 2 : Si σ_i est négatif et n'appartient à aucun k-cycle de K_i, alors le $(k-1)$-cycle $\partial\sigma_i$ n'est pas un bord à K_{i-1}. En effet, dans ce cas nous pourrions trouver une k-chaîne c' à K_{i-1} tel que $\partial c = \partial\sigma_i$, ou de façon équivalente, $\partial(c + \sigma_i) = 0$, ce qui implique que $c + \sigma_i$ est un k-cycle de K_i qui contient σ_i : contradiction. Par conséquent, comme le $(k-1)$-cycle $\partial\sigma_i$, qui n'était pas un bord à K_{i-1}, dévient un bord à K_i, nous avons $\beta_{k-1}(K_i) \leq \beta_{k-1}(K_{i-1}) - 1$. Avec un argument similaire

à celui de la fin du Cas 1, nous pouvons démontrer l'égalité.
Du fait que σ_i est négatif, c'est aussi évident que le groupe d'homologie $H_k(K_i)$ reste inaltéré. □

Nous voyons donc que, de quelque sorte, le processus se limite à trouver la différence entre les simplexes positifs et négatifs de la filtration ; les positifs créent de nouvelles k-classes d'homologie tandis que les négatifs effacent des $(k-1)$-classes. Voici un résumé de l'algorithme :

Algorithm 3: *Calcul des nombres de Betti d'un complexe simplicial K*

Input: Une filtration de décomposition de K, complexe simplicial d-dimensionnel avec m simplexes
$\beta_0, \beta_1, \ldots, \beta_d = 0$
Output: Les nombres de Betti $\beta_0, \beta_1, \ldots, \beta_d$ de K

1 **for** $i = 1$ jusqu'à m :
k= $\dim \sigma_i$
Si σ_i est positif :
$\beta_k = \beta_k + 1$
Si σ_i est négatif :
$\beta_{k-1} = \beta_{k-1} - 1$

3.3.4. Homologie persistante : définitions et algorithmes

Nous avons vu que l'algorithme précédant ne compute pas seulement les nombres de Betti d'un complexe simplicial, mais de tous les sous-complexes de la filtration (de décomposition). Intuitivement, l'objectif de l'homologie persistante est de garder une trace de toute cette information et enregistrer à quels moments chaque classe d'homologie est crée et détruite pendant le processus.

Avant d'expliquer les formalismes, montrons un petit exemple, en utilisant l'homologie singulière et la Proposition 3.20 :

Exemple 3.27. Soit $f : (0,1) \to \mathbb{R}$ la fonction représentée dans l'image suivant :

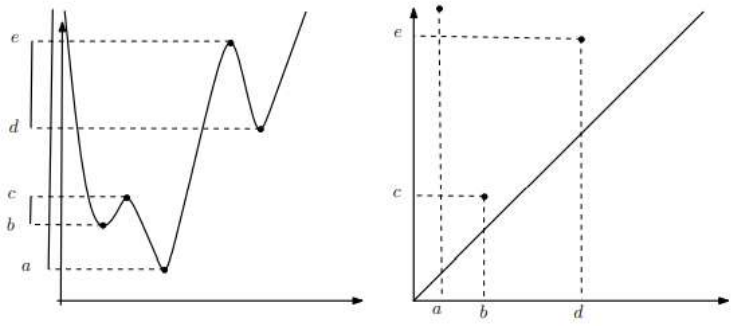


FIGURE 6 – Diagramme de persistance d'une fonction réelle, où seulement les composantes connexes (i.e. H_0) sont enregistrées.

Nous sommes intéressés à étudier l'évolution de la topologie de la filtration associé aux ensembles de sous-niveau de f, $f^{-1}((-\infty, t])$, à mesure que t augmente. La topologie de ces sous-ensembles change quand t atteint les valeurs critiques a, b, c, d et e :

Quand $t = a$, une nouvelle composante connexe apparaît, et pour $a \leq t \leq b$, $f^{-1}((-\infty, t))$ est un intervalle. Quand t atteint la valeur b, une deuxième composante connexe apparaît, et pour $b \leq t \leq c$, $f^{-1}((-\infty, t))$ a deux composantes connexes. Pour $t = c$, ces deux composantes connexes sont fusionnées : celle qui a été crée plus récemment, quand $t = b$, est unifiée à la première. Ainsi, on enregistre la paire (b, c) comme les temps de création et destruction de la composante ; cette paire est après représentée avec les cordonnées (b, c) au plan à droite. Intuitivement, le plus éloignée un point est de la diagonale, le plus relevant est la composante.
Si nous continuons à augmenter t, encore une nouvelle composante est crée à $t = d$, qui est finalement unifiée à la première quand t atteint la valeur e ; ainsi, un deuxième point est enregistré à droite, avec cordonnées (d, e). La première valeur a ne peut pas être associée à aucune autre valeur finie, donc la composante connexe crée pour cette t ne meurt jamais ; par conséquent, elle est associée à $+\infty$.
À la fin, toutes ces paires peuvent être représentées comme une famille d'intervalles (*barcode*) ou comme un diagramme au plan, appelé *diagramme de persistance*. Pour des raisons qui deviendront claires plus tard, c'est aussi naturel d'ajouter la diagonale $\{y = x\}$ au diagramme.

Quand nous considérons des fonctions définies dans des espaces topologiques générales, atteindre certaines valeurs critiques peut changer ne pas seulement les composantes connexes des ensembles de sous-niveau, mais d'autres propriétés topologiques encodées dans les autres groups d'homologie (i.e. les "trous" n-dimensionnels). De ce fait, il est aussi raisonnable de créer des paires de création/ destruction pour chaque dimension.

Finalement, supposons une fonction g "proche" à f comme celle de l'image d'en bas. Nous pouvons observer que, même si g a plus de paires dans son diagramme de persistance, la majorité sont très proches à la diagonale, donc une durée de vie assez courte. En revanche, les paires associées à un intervalle plus long sont proches à celles de f. En d'autres termes, les propriétés topologiques qui ont une persistance élevée sont préservées, tandis que celles qui son crées à cause de perturbations sur la fonction ont une persistance plus petite. Nous verrons que, en effet, deux fonctions "proches" ont toujours des diagrammes de persistance "proches". Cette notion de proximité est essentielle pour bien distinguer et traiter le bruit topologique dans nos données.

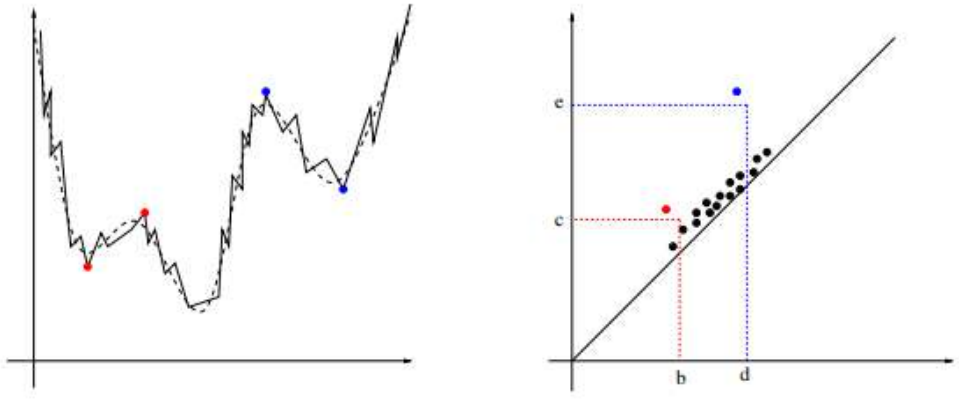


FIGURE 7 – Une approximation g de f, et les diagrammes de persistance respectifs.

Homologie persistante d'une filtration

Nous définissons d'abord la notion de persistance pour la filtration d'un complexe simplicial, qui a pour objectif étudier l'évolution de l'homologie des sous-complexes de la filtration.

Soit K un complexe simplicial d-dimensionnel avec une filtration de décomposition

$$\emptyset = K^0 \subset K^1 \subset \cdots \subset K^m = K.$$

Pour tout $0 \leq n \leq m$, on dénote C_k^n l'ensemble de k-chaînes de K^n (avec des coefficients à $\mathbb{Z}_2$). Notons que, puisque $\partial^2 = 0$ et nous avons une filtration de décomposition, la restriction de la fonction bord sur C_k^n a toujours sa image contenue dans C_{k-1}^{n-1}. Dénotons Z_k^n et B_k^n les cycles et bords k-dimensionnels de K^n respectivement, donc le k-ème groupe d'homologie de K^n est $H_k^n = Z_k^n / B_k^n$. Avec ces notations nous avons aussi

$$Z_k^0 \subset Z_k^1 \subset \cdots \subset Z_k^m = Z_k(K), \quad B_k^0 \subset B_k^1 \subset \cdots \subset B_k^m = B_k(K).$$

Définition 3.28. (Nombres de Betti persistants) : *Pour $p \in \{0, \ldots, m\}$ et $l \in \{0, \ldots, m-p\}$, le* k-ème nombre de Betti persistant de K^l *est la dimension de l'espace vectoriel $H_k^{l,p} = Z_k^l / (B_k^{l+p} \cap Z_k^l)$.*

Intuitivement, le k-ème nombre de Betti persistant de K^l représente le numéro de classes d'homologie indépendantes de k-cycles dans K^l qui ne sont pas de bords dans K^{l+p} ; par conséquente, de manière informelle, sa durée de vie est supérieure à p.
Nous avons vu, dans la section précédente, qu'une classe d'homologie est crée quand un simplexe positif est ajouté à la filtration, et qu'une classe d'homologie est détruite quand le simplexe est négatif. L'homologie persistante offre un cadre théorique pour associer des simplexes positifs et négatifs : quand un simplexe positif est ajouté à la filtration, il crée une classe d'homologie, qui disparaît quand on ajoute son simplexe négatif associé (s'il existe).

Nous sommes intéressés à trouver un algorithme pour calculer ces nombres de Betti persistantes. Pour l'obtenir, il faut mieux caractériser les classes d'homologie crées à chaque fois que nous ajoutons un simplexe positif à la filtration. Rappelons que, dans la définition 3.25 de simplexe positif et négatif, il faut seulement que σ_i appartient à un k-cycle, mais en général ce k-cycle n'est pas unique. Heureusement, pour chaque k-simplexe positif σ_i que nous ajoutons ajoute à la filtration, il y a un k-cycle associé "minimal", qui facilitera, à la fois, le calcul des nombres de Betti persistantes :

Lemme 3.29. *Soit σ_i un k-simplexe positif ajouté à la filtration de K au pas i. Or, il n'y a qu'un seul k-cycle c qui n'est pas un bord dans K^i, qui contient σ_i et qui ne contient aucun autre k-simplexe positif.*

Démonstration. Nous travaillons par induction sur l'ordre avec lequel les k-simplexes positifs sont ajoutés à la filtration. Pour le premier k-simplexe positif σ ajouté, ce k-cycle c existe par définition, est il est nécessairement unique parce que s'il y en avait un autre c' de différent, $c + c' \neq 0$, qui ne contient pas σ, serait aussi un k-cycle et ses éléments seraient des k-simplexes positifs, contradiction.

Supposons maintenant que le résultat est vrai pour tous les k-simplexes positifs déjà ajoutés, et ajoutons σ_i. Comme σ_i est positif, il existe un k-cycle d qui n'est pas un bord dans K_i et qui contient σ_i. Soit $\sigma_{i_j},\ j = 1, \ldots, p$ les k-simplexes positifs différents de σ_i contenus dans d, et c_{i_j} ses

k-cycles respectifs associés, qui existent par hypothèse d'induction. Alors

$$c = d + c_{i_1} + \cdots + c_{i_p} \tag{5}$$

est un k-cycle où σ_i est le seul k-simplexe positif. Du fait que σ_i est le dernier simplexe ajouté à K_i, il n'existe aucun $(k+1)$-simplexe dans K_i où σ_i est une face. Par conséquent, c n'est pas un bord, et l'existence est démontrée.

Pour démontrer l'unicité de c, supposons deux k-cycles $\hat{d}_1$ et $\hat{d}_2$ qui ne sont pas un bord dans K_i et qui contient σ_i, et répétons la construction précédente pour obtenir $\hat{c}_1 \neq \hat{c}_2$. Alors $\hat{c}_1 - \hat{c}_2 \neq 0$ est un k-cycle qui ne contient aucun k-simplexe positif, et nous pourrions toujours le combiner avec un c_i antérieur pour obtenir un k-cycle avec les mêmes propriétés du lemme, ce qui contredit sa unicité et l'hypothèse d'induction. Donc, nous concluons que $\hat{c}_1 - \hat{c}_2 = 0$, et l'unicité est démontrée. □

Bases des groupes d'homologie persistante et paires de persistance

Proposition 3.30. *Les k-cycles associés aux k-simplexes positifs décrits au Lemme 3.29 peuvent être utilisés pour calculer une base des différents groupes d'homologie k-dimensionnels de tous les sous-complexes de la filtration.*

Démonstration. Évidemment, au début toutes les bases de $H_n^0(K) = H_k(K_0)$ sont vides pour $k = 0, \ldots, d$. Les bases des successives H_k^i sont construites de manière inductive :

- Supposons que nous avons déjà une base de H_k^{i-1} et que σ_i est positif et de dimension k. Si nous ajoutons à notre base la classe d'homologie définie par le cycle c_i associé à σ_i, nous obtenons une base de H_k^i grâce au Lemme précédent. En effet, du fait que c_i est une somme de σ_i et k-simplexes négatifs, il n'est homologue à aucune combinaison linéaire des cycles qui définissent la base de H_k^{i-1}. Du fait que $\dim H_k^i = \dim H_k^{i-1} + 1$, vu dans la Proposition 3.26, nous obtenons une base de H_k^i.

- Supposons maintenant qu'une base de H_k^{j-1} est déjà construite et que le simplexe σ_j est négatif et de dimension $k+1$. Soient $c_{i_1}, \ldots, c_{i_p}$ les k-cycles associés aux simplexes positifs déjà ajoutés, qui définissent les classes d'homologie qui forment notre base de H_k^{j-1}. Comme nous l'avons déjà expliqué, le bord $\partial\sigma_j$ est un k-cycle de K_{j-1} qui n'est pas un bord dans K_{j-1}, mais qui devient un bord dans K_j. Par conséquent, il peut être écrit de manière unique comme

$$\partial\sigma_j = \sum_{k=1}^{p} \varepsilon_k c_{i_k} + b, \tag{6}$$

où $\varepsilon_k \in \{0, 1\}$ et b est un bord. Soit $l(j) = \max\{i_k |\ \varepsilon_k = 1\}$.

Claim : Si on enlève la classe d'homologie associé à $c_{l(j)}$ de la base de H_k^{j-1}, on obtient une base de $H^j k$.

En effet, comme $\dim H_k^{j-1} = \dim H_k^j + 1$ par la Propositon 3.26, il suffit de montrer que $c_{l(j)}$ est une combinaison linéaire d'un bord avec les autres cycles c_{ik} dans Z_k^j. L'équation (6) antérieure montre une telle décomposition, ce qui finis la démonstration. □

Définition 3.31. (Paires de persistance) *Les paires de simplexes $(\sigma_{l(j)}, \sigma_j)$ s'appellent les* paires de persistance *de la filtration de K.*

Intuitivement, la classe d'homologie crée pour $\sigma_{l(j)}$ dans $K_{l(j)}$ est détruite pour σ_j dans K_j. La persistance de cette paire est $j - l(j)$.

Le problème avec l'algorithme antérieur est de déterminer $l(j)$. N'oublions pas non plus qu'il faut aussi encore expliquer, en vue de l'algorithme du calcul des nombres de Betti de la Proposition 3.26, comment détecter si un nouveau k-simplexe est positif ou négatif. Toutes ces questions peuvent être répondues à la fois en étudiant la filtration sous une forme matricielle. D'idée derrière cette matrice est d'encoder le résultat de la fonction bord sur tous les simplexes de la filtration, ordonnés. Avec cette matrice, nous pouvons réélaborer la proposition antérieur pour obtenir un algorithme effectif pour calculer les paires de persistance, ce qui permet trouver les nombres de Betti persistantes. Elle proportionne aussi une manière de détecter quand le k-simplexe ajouté est positif ou négatif.

Soit K un complexe simplicial avec une filtration de décomposition. Soit $M = (m_{ij})_{i,j=1,\dots,m}$ la matrice associé au pas m de la filtration, où

$$m_{i,j} = 1 \text{ si } \sigma_i \text{ est une face de } \sigma_j, \text{ et } m_{i,j} = 0 \text{ autrefois.}$$

Cette matrice augmente "à droite et en bas" à mesure que la filtration avance, et elle est toujours triangulaire supérieure puisqu'on a une filtration de décomposition.

Pour une colonne C_j, soit $l(j) = \max\{i|\ m_{i,j} = 1\}$, et non-assigné si la colonne contient seulement des zéros. Nous pouvons alors considérer l'algorithme suivant :

Algorithm 4: *Calcul des paires de persistance, version matricielle*

Input: Une filtration de décomposition de K, le sous-complexe K^m (qui contient m simplexes) et la matrice M associé au pas m

for $j = 0$ jusqu'à m :
  while qu'il existe $j' < j$ avec $l(j') == l(j)$
    $C_j = C_j + C_{j'}$ (mod 2)

Output: Les paires $(l(j), j)$

Proposition 3.32. *L'algorithme antérieur calcule les paires de persistance de la filtration de décomposition de K jusqu'au pas m, ainsi comme quels simplexes sont positifs et quels sont négatifs.*

Démonstration. Remarquons que, à chaque pas de l'algorithme, la colonne C_j représente une chaîne de la forme $\partial\Big(\sigma_j + \sum_{i<j} \varepsilon_i \sigma_i\Big)$, où $\varepsilon_i \in \{0, 1\}$.

- Si à la fin de l'algorithme j satisfait que $l(j)$ est assigné, alors $\sigma_{l(j)}$ est un simplexe positif. En effet, on a $\partial\Big(\sigma_j + \sum_{i<j} \varepsilon_i \sigma_i\Big) = \sigma_{l(j)} + \sum_{p<l(j)} \lambda_p \sigma_p$, où $\lambda_p \in \{0, 1\}$. Du fait que $\partial^2 = 0$, on a que $\sigma_{l(j)}$ appartient à un cycle et il est donc positif.

- Si à la fin de l'algorithme C_j contient seulement des zéros, σ_j est positif. Effectivement, $\partial\Big(\sigma_j + \sum_{i<j} \varepsilon_i \sigma_i\Big) = 0$, et σ_j appartient donc à un cycle.

-Finalement, si à la fin de l'algorithme la colonne C_j contient des termes non nuls, $(\sigma_{l(j)}, \sigma_j)$ est une paire de persistance, et σ_j est donc négatif. En effet, le bord de σ_j peut alors être écrit de la forme $\sigma_{l(j)} + \sum_{p<l(j)} \lambda_p \sigma_p + \partial\Big(\sum_{i<j} \varepsilon_i \sigma_i\Big)$. Or, $\sigma_{l(j)}$ est positif, donc il a crée une classe d'homologie au moment $l(j)$, et il reste non associé. Du fait que le dernier index non nul d'une colonne est unique et par la Proposition 3.30, on peut déduire que $(\sigma_{l(j)}, \sigma_j)$ est une paire de persistance. □

3.3.5. Diagrammes de persistance et stabilité

Rappelons que, d'après le Remarque 3.24, tous les algorithmes que nous venons d'expliquer sont aussi applicables à des filtrations pas nécessairement de décomposition. En effet, nous pouvons toujours faire une affinage, et après prendre les coefficients de la filtration original. Également, avec des algorithmes qui facilitent les calculs ou pas, les notions de l'homologie persistante introduites jusqu'à ici, notamment les paires de persistance, peuvent aussi être appliquées avec des filtrations d'espaces topologiques plus générales, en prenant l'homologie singulière. Nous avons déjà observé ce fait informellement dans l'exemple 3.26, et nous en verrons aussi dans cette section.

Dans tous les cas, beaucoup des informations de l'homologie persistante, notamment la durée de vie des différentes classes d'homologie, peuvent être facilement représentées en forme de diagramme :

Avec k fixée, soient $(\sigma_{l(j)}, \sigma_j)$ les paires de persistance (calculées avec les algorithmes précédentes, par exemple), où $\sigma_{l(j)}$ et σ_j ont dimension k et $k+1$ respectivement. Nous représentons chacune de ces paires dans $\mathbb{R}^2$ avec le point de coordonnés $(l(j), j)$; pour les simplexes positifs σ_i qui restent non associés, nous créons les points $(i, +\infty)$.

Définition 3.33. *Nous appellons la représentation de cet ensemble de points dans $\mathbb{R}^2$ avec la diagonale $\Delta = \{x = y\}$ le* diagramme de persistance k-dimensionnel *de la filtration.*

Plus généralement, si la filtration est indexée par une suite croissante de numéros réels, comme par exemple avec les filtrations introduites dans la section 3.3.2,

$$\emptyset = K_{\alpha_0} \subset K_{\alpha_1} \subset \cdots \subset K_{\alpha_{n-1}} \subset K_{\alpha_n} = K,$$

une paire de persistance de simplexes (σ_i, σ_j) est représentée par le point (α_i, α_j), avec les indices d'apparition et mort ; si le simplexe σ_i reste non associé, nous ajoutons ajoute le point $(\alpha_i, +\infty)$.

Le même type de points peuvent être crées pour toute filtration d'un espace topologique et avec l'homologie singulière, où la coordonné x enregistre l'apparition d'une classe d'homologie et la coordonné y sa mort. En tout cas, dans ces cas plus générales, il faut faire attention au fait que plusieurs paires peuvent être associées au même point dans le plan. Donc, dans ces diagrammes de persistance il faut aussi considérer une multiplicité pour chaque point. Par convention, les points de la diagonale ont tous multiplicité infinie. Désormais, nous considérerons aussi une multiplicité pour chaque point dans la définition de diagramme de persistance.

Nous pouvons définir une distance entre diagrammes de persistance pour mieux les comparer :

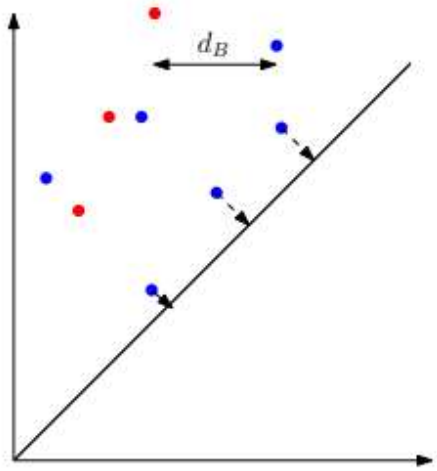
Définition 3.34. (Distance "bottleneck") *Soient D_1 et D_2 deux diagrammes de persistance. La* distance goulot *("bottleneck" en anglais) entre D_1 et D_2 est définie comme*

$$d_B(D_1, D_2) = \inf_{\gamma} \sup_{p \in D_1} ||p - \gamma(p)||_\infty,$$

où γ est l'ensemble de bijections entre les points de D_1 et D_2 ; on prend m copies disjointes si un point a multiplicité $m > 1$. Par convention, si $p = (x_p, +\infty)$, $q = (x_q, +\infty)$, $||p - q||_\infty = |x_p - x_q|$.

Remarque 3.35. C'est précisément cette distance qui motive ajouter la diagonale aux diagrammes de persistance : elle permet de comparer des diagrammes qui n'ont pas le même nombre de points dehors la diagonale en les associant avec des points de la diagonale.

Nous omettrons la démonstration qu'il s'agit vraiment d'une distance. Dans l'image suivante il apparaît une représentation de la distance bottleneck entre deux diagrammes de persistance :



Stabilité persistante des fonctions

Dans cette dernière partie de la section, inévitablement plus avancée et sans toutes les démonstrations, nous allons lier la proximité de deux fonctions f et g à la proximité de ses diagrammes de persistance. Cette relation est fondamentale pour justifier la convenance des diagrammes de persistance associées à une fonction, ainsi comme pour identifier ses propriétés "proéminentes" et les distinguer du "bruit topologique", i.e. des caractéristiques topologiques de courte durée crées pour de petites perturbations. Ces résultats deviennent aussi importants pour justifier la procédure de l'algorithme ToMATo du prochain chapitre, où nous travaillons avec une estimation $\hat{f}$ d'une densité f, et nous regardons son diagramme de persistance (0-dimensionnel).

Expliquons brièvement la situation : soit $f : X \to \mathbb{R}$ une fonction réelle continue définie sur un espace topologique X. Nous voulons étudier le diagramme de persistance k-dimensionnel associé à ses ensembles de sous-niveau $\{F_\alpha\}_{\alpha\in\mathbb{R}}$, où $F_\alpha = f^{-1}((-\infty, \alpha])$, avec l'homologie singulière.
Une propriété fondamentale de l'homologie singulière est que toute application continue entre espaces topologiques $h : X \to Y$ induit un morphisme entre les respectives (k-èmes) groupes d'homologie, $h_* : H_*(X) \to H_*(Y)$. Plus concrètement, la (k-ème) homologie singulière est un foncteur (covariant) de la catégorie des espaces topologiques à la catégorie des groupes. Donc, il est toujours vrai que $(id_X)_* = id_{H_*(X)}$ et $(h \circ g)_* = h_* \circ g_*$.

Dans notre cas, on peut étudier les applications induites par les inclusions $F_a \subset F_b$ quand $a < b$, $f_a^b : H_*(F_a) \to H_*(F_b)$. Ces groupes et morphismes encodent toute l'information de l'homologie persistante : quelques références appellent *groupes d'homologie persistante* aux groupes $\mathrm{Im}\, f_a^b$, qui suivent la même idée que les groupes qui apparaissent dans la Définition 3.28, dans le cas simplicial. Intuitivement, avec deux fonctions "proches" (avec la distance $||f-g||_\infty = \sup_{x\in X} |f(x)-g(x)|$), les moments de création et de mort de certaines caractéristiques topologiques (gardés dans les groupes d'homologie, et représentés graphiquement dans les diagrammes de persistance respectifs $D(f)$ et $D(g)$) devraient être similaires. Cette idée est formalisée dans le théorème suivant :

Théorème 3.36. (Théorème de la stabilité bottleneck des diagrammes de persistance) : *Soit X un espace topologique triangulable avec des fonctions tame $f, g : X \to \mathbb{R}$. Alors, les diagrammes de persistance satisfont $d_B(D(f), D(g)) < ||f - g||_\infty$.*

Nous démontrerons le théorème à la fin de la section, mais sans tous les pas intermédiaires. Avant, plusieurs définitions et résultats sont nécessaires :

Définition 3.37. *Soit X un espace topologique, $f : X \to \mathbb{R}$. Une* valeur critique homologique *de f est un numéro réel a pour lequel il existe un entier k tel que, $\forall \varepsilon > 0$, l'application induite $f_{a-\varepsilon}^{a+\varepsilon} : H_k(F_{a-\varepsilon}) \to H_k(F_{a+\varepsilon})$ n'est pas un isomorphisme.*

Lemme 3.38. (Lemme de la valeur critique) : *Si $[a, b]$ ne contient aucune valeur critique homologique de f, f_a^b est un isomorphisme pour tout $k \in \mathbb{Z}$.*

Démonstration. Soit $m = \frac{x+y}{2}$, or $f_a^b = f_m^b \circ f_a^m$. Si f_a^b n'est pas un isomorphisme, au moins une des deux fonctions antérieures ne l'est pas non plus. Ainsi, on peut construire de manière inductive une suite d'intervalles fermés décroissants, l'intersection desquels est une valeur critique homologique dans $[x, y]$, ce qui est une contradiction. □

Le lemme antérieur formalise l'idée que c'est seulement quand on atteint des valeurs critiques homologiques que nouvelles caractéristiques topologiques sont crées ou destruites. Notre résultat requiert une condition de finitude sur notre fonction, qui se satisfait dans la plupart des cas d'étude :

Définition 3.39. *Une fonction $f : X \to \mathbb{R}$ s'appelle* tame *si elle a un numéro fini de valeurs critiques homologiques et tous les groupes d'homologie $H_k(F_a)$ ont dimension finie $\forall k \in \mathbb{Z}$, $a \in \mathbb{R}$.*

Rappelons maintenant la définition de la distance de Hausdorff, très habituelle dans la TDA, pour des nuages de points :

Définition 3.40. *Soient X et Y des ensembles de points (avec multiplicité). Alors la distance Hausdorff est $d_H(X, Y) = \max\{\sup_x \inf_y ||x - y||_\infty, \sup_y \inf_x ||y - x||_\infty\}$.*

Un résultat important, mais sans démonstration, que nous utiliserons plus tard est celui-ci :

Proposition 3.41. *Soit X un espace topologique triangulable avec des fonctions tame $f, g : X \to \mathbb{R}$. Alors $d_H(D(f), D(g)) < ||f - g||_\infty$.*

Remarque 3.42. La distance Hausdorff entre deux diagrammes de persistance n'excède jamais la distance bottleneck, car elle ne fait pas attention aux multiplicités, ou regroupements de points. Le résultat pour la distance bottleneck est plus fort, et clé, pour quelques applications.

Voyons avant le résultat du Théorème 3.36 pour un cas spécial, et plus simple. Nous nous en servirons plus tard pour prouver le cas général.
Étant donnée une fonction tame $f : X \to \mathbb{R}$, soit δ_f la distance minimale entre deux points dehors la diagonal, ou entre un point dehors la diagonale et un autre dedans :

$$\delta_f = \min\{||p - q||_\infty |\ D(f)\backslash\Delta \ni p \neq q \in D(f)\}.$$

Si on dessine des carrés de rayon $\varepsilon = \delta_f/2$ centrés sur les points de $D(f)$, on obtient une diagonale plus "grosse", et une collection finie de carrés disjoints entre eux et avec la diagonale.

Définition 3.43. *Une autre fonction tame $g : X \to \mathbb{R}$ est appelée* très proche *à f si $||f-g||_\infty < \frac{\delta_f}{2}$.*

Ici un autre lemme nécessaire mais sans démonstration, de nature plus technique :

Lemme 3.44. *Soient $f, g : X \to \mathbb{R}$ des fonctions tames, g très proche à f. Soient $p \in D(f)\backslash\Delta$, m_p sa multiplicité et $\square_\varepsilon$ le carré centré en p de rayon $\varepsilon = ||f - g||_\infty$. Alors $|D(g) \cap \square_\varepsilon| = m_p$.*

Lemme 3.45. (Lemme de la bijection) : *Soit X un espace topologique triangulable, $f, g : X \to \mathbb{R}$ des fonctions tames, g très proche à f. Alors $d_B(D(f), D(g)) \leq ||f - g||_\infty$.*

Démonstration. Soient $p \in D(f)\backslash\Delta$ et $\square_\varepsilon$ le carré centré en p de rayon $\varepsilon = ||f-g||_\infty$, comme avant. Du lemme précédant, tous les points de $D(g) \cap \square_\varepsilon$ peuvent être associés à p. Nous pouvons suivre cette procédure pour tous les points dehors la diagonal de $D(f)$. Après, les seuls points de $D(g)$ qui restent sans image ont une distance supérieure à ε de $D(f)\backslash\Delta$. Du fait que $d_H(D(f), D(g)) \leq \varepsilon$ (Proposition 3.41), ces points de $D(g)$ sont au plus à distance ε de la diagonale. Si nous leur associons respectivement les points les plus proches de la diagonale, nous obtenons une bijection entre $D(f)$ et $D(g)$ (rappelons que les points de la diagonale ont multiplicité infinie). Cette bijection déplace les points au plus ε, ce qui finit la démonstration. □

Rappelons qu'un espace topologique est appelé *triangulable* s'il existe un complexe simplicial fini avec une réalisation géométrique homéomorphe. Nous pouvons montrer notre théorème dans toute sa généralité en faisant des approximations successives de notre fonction originelle $f : X \to \mathbb{R}$ par des fonctions linéaires par morceaux définies sur un complexe simplicial :

Définition 3.46. *Soit K un complexe simplicial avec des valeurs réels spécifiées sur chaque sommet x_i, $f(x_i)$. Sa fonction linéaire par morceaux (LPM) associée est $\hat{f} : K \to \mathbb{R}$, $\hat{x} = \sum_i b_i(x) f(x_i)$, où $b_i(x)$ sont les coordonnés barycentriques de x. On obtient une fonction linéaire sur chaque simplexe du complexe.*

Remarquons que, à cause de sa nature finie et linéaire, une fonction LPM sur un complexe simpliciale fini est toujours tame. Ce fait permet de démontrer le Théorème 3.36 pour deux fonctions LPM $\hat{f}, \hat{g}$ définies sur un complexe simplicial K fini. Avant, une dernière définition :

Définition 3.47. *Une* combinaison convexe *de $\hat{f}$ et $\hat{g}$ est une fonction du type $h_\lambda = (1-\lambda)\hat{f} + \lambda\hat{g}$, avec $\lambda \in [0, 1]$. Cette famille de combinaisons convexes entre les deux fonctions, où $h_0 = \hat{f}$ et $h_1 = \hat{g}$, s'appelle* interpolation linéaire de $\hat{f}$ à $\hat{g}$.

Lemme 3.48. (Lemme d'interpolation) : *Soient $\hat{f}, \hat{g}$ deux fonctions LPM définies sur un complexe simplicial K fini. Alors $d_B(D(\hat{f}), D(\hat{g})) \leq ||\hat{f} - \hat{g}||_\infty$*

Démonstration. L'idée de base de la démonstration est de décomposer l'interpolation linéaire de $\hat{f}$ à $\hat{g}$ en petites sections pour utiliser le Lemme de la bijection, et ainsi obtenir une bijection dans chaque section.

Soit $\varepsilon = ||\hat{f} - \hat{g}||_\infty$, et observons que, pour tout $\lambda \in [0, 1]$, h_λ est tame (car elle est aussi une fonction LPM) et $\delta(\lambda) = \delta_{h_\lambda}$ est strictement positif quand au moins $\hat{f}$ ou $\hat{g}$ ont un point dehors la diagonale (sinon, l'inégalité du lemme est triviale).
Donc, la famille $C = \{J_\lambda\}_{\lambda \in \mathbb{Q} \cap [0,1]}$ d'intervalles ouverts $J_\lambda = (\lambda - \frac{\delta(\lambda)}{4\varepsilon}, \lambda + \frac{\delta(\lambda)}{4\varepsilon})$ forme un recouvrement ouvert de l'intervalle $[0, 1]$. Prenons un sous-recouvrement fini C' de C, qui existe pour être $[0, 1]$ compact, et minimal. Soient $\lambda_1 < \cdots < \lambda_n$ les points médians des intervalles de C'. Du fait que C' est minimale, deux intervalles consécutifs J_{λ_i} et $J_{\lambda_{i+1}}$ ont toujours intersection non-vide, et

$$\lambda_{i+1} - \lambda_i \leq \frac{\delta(\lambda_i) + \delta(\lambda_{i+1})}{4\varepsilon} \leq \frac{\max\{\delta(\lambda_i), \delta(\lambda_{i+1})\}}{2\varepsilon}$$

Par définition de ε, on a aussi $||h_{\lambda_i} - h_{\lambda_{i+1}}||_\infty = ||(\lambda_{i+1} - \lambda_i)(\hat{g} - \hat{f})||_\infty = \varepsilon(\lambda_{i+1} - \lambda_i)$. Par conséquent,

$$||h_{\lambda_i} - h_{\lambda_{i+1}}||_\infty \leq \frac{\max\{\delta(\lambda_i), \delta(\lambda_{i+1})\}}{2},$$

ce qui implique que h_{λ_i} est très proche à $h_{\lambda_{i+1}}$, ou à l'inverse. Nous pouvons alors appliquer le Lemme de la bijection, qui dit que $d_B(D(h_{\lambda_{i+1}}), D(h_{\lambda_i})) \leq ||h_{\lambda_{i+1}} - h_{\lambda_i}||_\infty$ pour $1 \leq i \leq n-1$. Observons que, si nous ajoutons $\lambda_0 = 0$ et $\lambda_{n+1} = 1$ (donc $h_0 = \hat{f}$ et $h_1 = \hat{g}$), ces derniers raisonnements sont encore vraies, car 0 et 1 font aussi partie du recouvrement, et h_λ varie continuellement avec λ. Donc $\hat{f}$ est très proche à h_{λ_1} (ou à l'inverse), et $\hat{g}$ est très proche à h_{λ_n} (ou à l'inverse).

Maintenant, l'inégalité triangulaire donne

$$d_B(D(\hat{f}), D(\hat{g})) \leq \sum_{i=0}^{n} d_B(D(h_{\lambda_i}), D(h_{\lambda_{i+1}})) \leq \sum_{i=0}^{n} ||h_{\lambda_i} - h_{\lambda_{i+1}}||_\infty.$$

Du fait que les h_{λ_i} forment une interpolation linéaire de $\hat{f}$ à $\hat{g}$ et leurs valeurs changent linéairement entre les deux, la dernière somme est égale à $||\hat{f} - \hat{g}||_\infty$, ce qui finit la démonstration. □

Avec ce dernier résultat, nous pouvons démontrer le Théorème 3.36 :

Théorème de la stabilité bottleneck des diagrammes de persistance : *Soit X un espace topologique triangulable avec des fonctions tame $f, g : X \to \mathbb{R}$. Alors, les diagrammes de persistance satisfont $d_B(D(f), D(g)) < ||f - g||_\infty$.*

Démonstration. (du Théorème 3.36 :) Par définition de triangulabilité, il existe un complexe simplicial fini L et un homéomorphisme $\phi : L \to X$. Notons que, du fait que ϕ est un homéomorphisme, $\phi^{-1}(f^{-1}((-\infty, a])) \cong f^{-1}((-\infty, a])\ \forall a \in \mathbb{R}$, et les groupes d'homologie singulière sont aussi tous isomorphes à cause de sa fonctorialité. Par conséquent, le diagramme de persistance reste non altéré par ce changement de variables : $f \circ \phi : L \to \mathbb{R}$ est aussi tame et a le même diagramme de persistance que f.

Soit $\delta > 0$ suffisamment petit. Du fait que f et g sont continues et L est compact, il existe une sous-division K de L telle que

$$|f \circ \phi(x) - f \circ \phi(y)| \leq \delta, \quad |g \circ \phi(x) - g \circ \phi(y)| \leq \delta \tag{7}$$

pour x, y dans le même simplexe de K.
Soient $\hat{f}, \hat{g} : \mathrm{Sd}K \to \mathbb{R}$ les fonctions linéaires par morceaux qu'on obtient à partir de $f \circ \phi$ et $g \circ \phi$ sur les sommets de $\mathrm{Sd}K$, où $\mathrm{Sd}K$ dénote la sous-division barycentrique de K. Par construction de K, ces fonctions satisfont $||\hat{f} - f \circ \phi||_\infty \leq \delta$ et $||\hat{g} - g \circ \phi||_\infty \leq \delta$. En faisant usage du Lemme d'Interpolation, le fait que $\hat{f}$ et $\hat{g}$ diffèrent au maximum δ de $f \circ \phi$ et $g \circ \phi$ respectivement, et $||f - g||_\infty = ||f \circ \phi - g \circ \phi||_\infty$, on obtient

$$d_B(D(\hat{f}), D(\hat{g})) \leq ||\hat{f} - \hat{g}||_\infty \leq ||f \circ \phi - g \circ \phi||_\infty + 2\delta = ||f - g||_\infty + 2\delta. \tag{8}$$

Si nous supposons de plus que $\delta < \frac{\delta_f}{2}$, nous obtenons une bijection du Lemme de la Bijection. Du fait que le changement de variables n'affecte pas le diagramme de persistance, on a

$$d_B(D(f), D(\hat{f})) = d_B(D(f \circ \phi), D(\hat{f})) \leq \delta. \tag{9}$$

Si nous supposons pareillement que $\delta < \frac{\delta_g}{2}$, l'inégalité triangulaire appliqué plusieurs fois avec (8) et (9) donne

$$d_B(D(f), D(g)) \leq ||f - g||_\infty + 4\delta,$$

ce qui montre le résultat, donc δ peut être arbitrairement petit. □

4. L'algorithme ToMATo

4.1. Introduction

L'exposition et exploration que nous ferons maintenant de l'algorithme ToMATo (*Topological Mode Analysis Tool*) et son implementation dans la librairie GUDHI constituent la partie la plus innovante de notre travail. Cette méthode, récemment conçue, se situe dans les techniques de clustering, donc dans l'apprentissage non supervisé. Bien que le fonctionnement ne soit pas spécialement complexe, il se base sur des idées de l'analyse topologique de données exposées antérieurement qu'il faut bien comprendre, notamment les complexes simpliciales ("graphes" désormais) et les diagrammes de persistance.

Un autre principe important de notre algorithme est qu'il est sensé fonctionner avec des sous-variétés de $\mathbb{R}^d$ (ou variétés riemanniennes en général), indépendemment de sa "forme". Cela est remarquable puisqu'une bonne partie des algorithmes de clustering existants (par exemple, la méthode K-means, déjà exposée) ne parviennent pas à identifier les clusters lorsque ces derniers s'éloignent d'une structure convexe. Par exemple, certains algorithmes sont incapables de bien regrouper un ensemble de données échantillonnées à partir de deux anneaux concentriques dans $\mathbb{R}^2$. Du ce fait, dans la section 4.1.1., nous exposerons les constructions et les arguments en prenant une variété riemannienne X, le cas le plus général. Cependant, dans la pratique nos données sont presque toujours dans $\mathbb{R}^d$, et seulement dans certains cas particuliers ils présentent une forme clairement semblable a une sous-variété de $\mathbb{R}^d$.

En nous appuyant sur la classification des techniques de clustering faite au début de la section "Algorithmes de clustering combinatoires", au deuxième chapitre, on pourrait affirmer que l'algorithme ToMATo combine une partie "mode-seeking" et une partie de nature plus combinatoire. En plus de cela, son innovation principale est que, pour guider la fusion des différents mini-clusters tout au long de la méthode, il utilise la notion de "persistance topologique", introduite au chapitre précèdent. En plus d'étiqueter les données dans de différents groups, l'algorithme produit aussi un diagramme de persistance, qui permet de choisir des paramètres précis afin d'obtenir le nombre de clusters souhaité.

4.1.1. L'intuition derrière l'algorithme : le cas continu

L'idée de base de la méthode est que, si les données sont obtenues en suivant une fonction de densité f, les clusters le plus logiques sont ces régions où la fonction fait des "bosses significatives". C'est dans ces dernières où les points seront plus probablement situés et regroupés.

Soit X une variété riemanniene de dimension m et f une fonction $f : X \to \mathbb{R}$ $\mathcal{C}^2$-continue sans points critiques dégénérés. Supposons aussi que f a un nombre fini de points critiques. La *région ascendante* d'un point critique m est le sous-ensemble de points $A(m) \subseteq X$ qui parviennent finalement à m en suivant le flux induit pour le champ de vecteurs gradient de f. On appelle m la *racine* de $x \in A(m)$.
On peut démontrer que les régions ascendantes des pics de f forment des sous-ensembles de X disjoints et homéomorphes à $\mathbb{R}^m$. De plus, si f est bornée et propre, les régions ascendantes de ces pics couvrent X sauf un sous-ensemble de mesure de Lebesgue zéro. Il est donc logique d'utiliser ces régions pour découper X p.p. en régions d'influence.

Considérons maintenant la famille de sous-espaces $\{F_\alpha\}_{\alpha\in\mathbb{R}}$, où $F_\alpha = f^{-1}([\alpha, +\infty))$ et α varie de $+\infty$ à $-\infty$ (i.e. la filtration de X associée aux ensembles de super-niveau de f). Pour $\alpha \in \mathbb{R}$

et $x \in F_\alpha$, appelons $C(x, \alpha) \subseteq F_\alpha$ la composante connexe par arcs de F_α contenant x. Selon la Théorie de Morse, lorsqu'un maximum local m_p de f entre dans la filtration au moment $\alpha = b(m_p)$ (b de "birth"), une nouvelle composante connexe par arcs $C(m_p, \alpha)$ apparaît dans F_α. Puis, cette dernière cesse d'être indépendante quand elle se connecte avec une autre composante générée pour un pic m_q plus haut, pour quelque autre $\alpha = d(m_p)$ (d de "death"). Dans ce cas, on nomem m_q la racine de m_p, et on écrit $m_q = r(m_p)$. Dans le diagramme de persistance 0-dimensionnel de f, la durée de vie de m_p comme racine est encodée pour le point $p = (b(m_p), d(m_p))$, et on appelle la différence $dp = p_x - p_y$ la *proéminence* de m_p, ou que m_p est *dp-proéminent*.

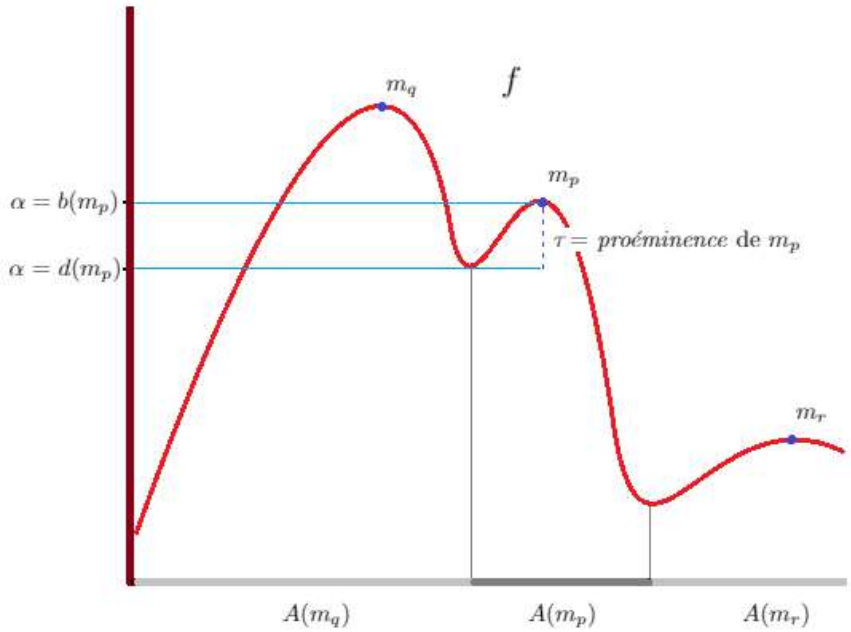


FIGURE 8 – Représentation graphique, avec f une fonction réelle d'une variable, de toutes les idées exposées jusqu'à ici : pics de f (points critiques/ maximums locaux), régions ascendantes de ces pics et proéminence du pic m_p.

En nous appuyant sur un "paramètre de fusion" $\tau \geq 0$, on peut seulement considérer les pics de f de proéminence au moins τ. Pour tout pic m_q de f, on itère l'"application racine" $m_q \mapsto r(m_q)$ jusqu'à ce qu'un pic de proéminence τ soit obtenu. Ce processus finit toujours, donc f a un nombre fini de points critiques, et on a toujours $f(m_q) \leq f(r(m_q))$. Appelons cette fonction itérée r_τ^*. Observons que tout pic de proéminence au moins τ est un point fixe de r_τ^*.
L'union des régions ascendantes de tous les pics qui arrivent finalement à m_p avec r_τ^* est appelée le *bassin d'attraction* de m_p (de paramètre τ), $B_\tau(m_p)$:

$$\forall m_p \text{ tel que } p_x - p_y \geq \tau,\ B_\tau(m_p) = \bigcup_{r_\tau^*(m) = m_p} A(m).$$

Clairement, $B_\tau(m_p)$ contient $A(m_p)$, donc m_p est un point fixe de r_τ^*. De plus, Ces bassins d'attraction forment une partition de l'union de toutes les régions ascendantes. Ce sont précisément ces bassins d'attraction qui constituent nos candidats à clusters.

4.2. Les données d'entrée (input data)

Dans tous les cas, pour fonctionner, la méthode ToMATo a besoin de deux informations sur nos données. Tout d'abord, l'algorithme requiert un poids pour chaque point, c'est à dire, une valeur $\hat{f}(i)$ associée à chaque donnée i, qui représente une estimation d'une hypothétique fonction de densité f, sur laquelle les données ont été obtenues. Puis, il est aussi nécessaire d'avoir un graphe de voisinage, qui "connecte" de quelque sorte ces données entre elles, et qui encode la proximité des points.

De plus, une autre information clé dois être aussi transmise à l'algorithme pour guider la fusion des clusters intermédiaires : un *paramètre de fusion* τ. En somme, ce paramètre détermine à partir de quelle persistance un point ou région de points se mélange avec d'autres ou pas. Sa valeur "idéale" change selon le cas, donc elle dépend de $\hat{f}$ et le numéro de clusters que l'on souhaite obtenir. Le fonctionnement et la détermination de ce paramètre, très important dans la méthode, deviendront beaucoup plus claires après avoir étudié la procédure de l'algorithme et le diagramme qu'il produit.

Développons à présent les deux premières informations nécessaires pour l'algorithme. Nous remarquons que, dans la pratique, nous ne les avons presque jamais directement. En effet, dans le cas le plus simple, nous avons seulement un nuage de points dans $\mathbb{R}^d$ avec n observations, ou, dans des cas plus élaborés et "théoriques", un ensemble de points dans une variété riemmanienne, qui permet également de définir des distances entre eux. Avec une base de données réelle, nous avons généralement une quantité n de données avec p attributs quantitatives et/ou catégoriques, où nous pouvons définir distances entre paires, ou les plonger dans $\mathbb{R}^p$ "convenablement" avec un métrique (voir l'introduction de la Section 2).

Indépendemment de la façon dont les calculs sont réalisés ou si on utilise un espace métrique ambiant (normalement $\mathbb{R}^d$), utiliser des distances entre paires de données est très pratique : elles permettent de construire assez rapidement les graphes de voisinage les plus naturels, et notre algorithme a besoin d'un graphe entre les données pour bien fonctionner. De plus, elles sont aussi pratiques pour calculer certaines estimations de la fonction de densité de base f.

4.2.1. Quelques constructions de graphes habituelles

Développons à présent certaines constructions de graphes habituelles sur des nuages de points, qui peuvent naturellement être utilisées dans notre situation. Nous assumons qu'il n'y a jamais la même distance entre toutes les paires de points. Si ce n'est pas le cas, nous pouvons adapter notre démarche en fonction de la situation :

- *Graphe α-Rips :* Il unit toutes les paires de points x, y qui satisfont $d(x, y) \leq \alpha$. Il est donc le squelette 1-dimensionnel de $Rips_\alpha(X)$, ou $Cech_{\frac{\alpha}{2}}(X)$.
  Il constitue, en quelque sorte, le graphe le plus naturel pour connecter les points proches entre eux, et il est aussi très facile à construire. Néanmoins, le nombre d'arêtes peut beaucoup varier entre sommets différents, et le paramètre α n'est pas toujours évident pour obtenir les résultats souhaités : si c'est trop petit, il peut y avoir un numéro excessif de composantes connexes ; cependant, s'il est trop grand, la structure de proximité se dilue aussi.

- *Graphe des k plus proches voisins (k-NN) :* Il connecte chaque sommet avec ses k autres sommets les plus proches. De cette façon, chaque sommet est l'extrémité d'au moins k arêtes. C'est à priori un graphe orienté, donc cette relation de proximité n'est pas symétrique : par exemple, avec $k = 1$, un sommet 1 peut avoir le sommet 2 comme le sommet le plus proche,

mais ce dernier avoir un sommet 3 plus proche que le sommet 1. Parfois, dans la pratique, on ignore cette directionnalité et on accepte que quelques sommets aient plus d'arêtes incidentes. Ce graphe est intéressant et utile puisque, en général, le numéro d'arêtes incidentes à chaque sommet reste assez similaire, et il n'y a jamais des points isolés. Il est aussi un peu plus exigeant a niveau de calcul, donc il faut ordonner à chaque pas les distances d'un sommet aux autres, mais certains algorithmes pour trouver des approximations du graphe k-NN existent qui sont beaucoup plus rapides. Son désavantage principal est que parfois il connecte de points qui ne sont pas spécialement proches.

- *Graphe de Delaunay :* C'est le graphe qu'on obtient si on triangule les points de façon à ce qu'aucun des points reste a l'intérieur du circumcercle d'aucun des triangles. Normalement, on obtient ainsi une triangulation sans beaucoup d'angles pointus. Il y a des algorithmes assez rapides pour le calculer, et il est aussi généralisable aux dimensions supérieures.
  Son principal avantage est que, à la différence des deux algorithmes précédents, ce dernier n'a pas besoin d'un paramètre pour être défini. Pareillement au graphe k-NN, le numéro d'arêtes incidentes à chaque sommet est souvent similaire, mais parfois il unit des points qui ne sont pas spécialement proches entre eux.

Ci-dessous, un exemple de chacun de ces trois graphes.

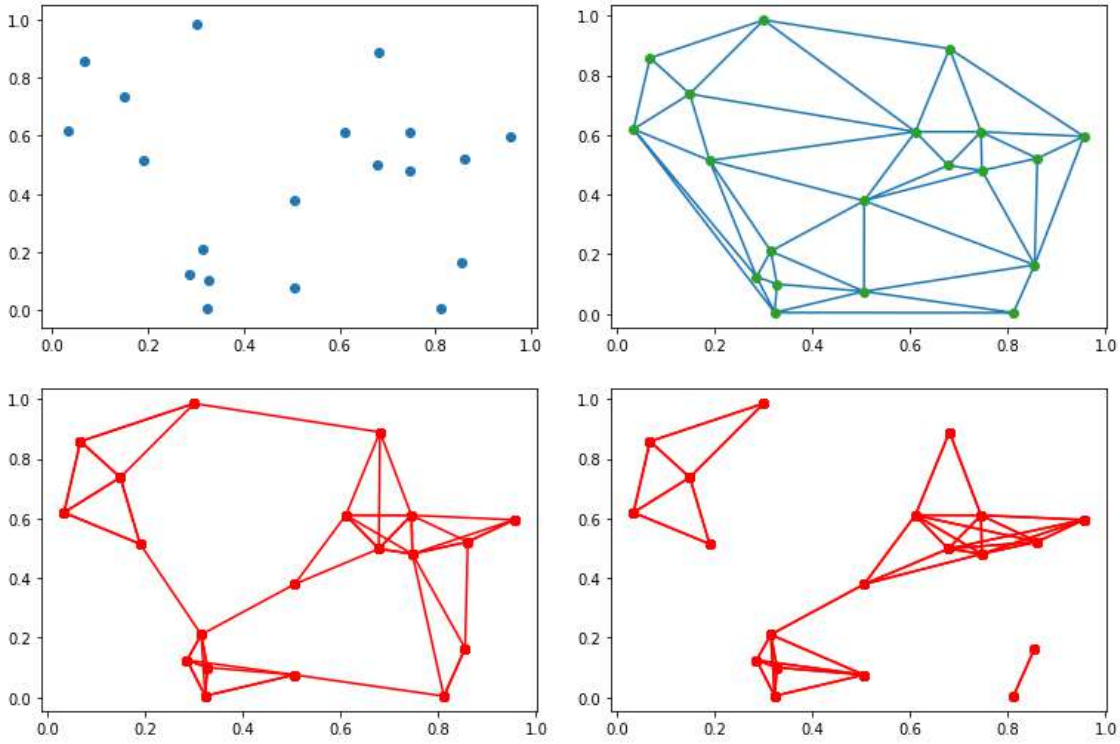


FIGURE 9 – Représentation des trois constructions exposés antérieurement avec un ensemble de 20 points échantillonnés dans le carré 1x1 (distribution uniforme). En haut à droite, le graph de Delaunay. En bas à gauche : le graphe k-NN avec $k = 4$ (chaque point est aussi son propre voisin le plus proche) ; nous observons qu'une partie importante des sommets a plus de trois arêtes incidentes. En bas à droite ; le graphe α-rayon avec $\alpha = 0.3$.

4.2.2. Quelques estimateurs classiques de la fonction de densité

Exposons à présent deux manières (non-paramétriques) d'estimer la fonction de densité f sur laquelle on suppose que les données ont été obtenues. L'idée est toujours de construire une fonction $\hat{f}$, estimation de la "véritable" fonction de densité f, en utilisant la disposition des points, qui nous donne des informations sur f. Nous ne détaillerons pas les arguments théoriques qui justifient la justesse (asymptotique) vers f de ces méthodes, et nous ignorerons aussi les possibles généralisations sur des sous-variétés : nous supposerons que f est simplement définie sur $\mathbb{R}^d$ (et souvent, seulement $\mathbb{R}$).

- *Estimation par les k plus proches voisins :* Rappelons avant que, par définition de fonction de densité, un vecteur aléatoire X dans $\mathbb{R}^d$ satisfait, pour tout borélien $A \subseteq \mathbb{R}^d$, $\mathbb{P}(X \in A) = \int_A f$. Donc, si A est une boule petite centrée sur x, la probabilité que X tombe dans A est approximativement $f(x)$ multiplié par le volume de A. En fait, avec des hypothèses assez faibles sur f continue, on a

$$\lim_{\alpha \to 0} \frac{\int_{B(x_0,\alpha)} f(x)dx}{|B(x_0,\alpha)|} = f(x_0), \tag{10}$$

  où $|\cdot|$ dénote le volume dans $\mathbb{R}^d$. Rappelons aussi que le volume de la boule unité dans $\mathbb{R}^d$ satisfait la formule

$$V_d = \frac{\pi^{\frac{d}{2}}}{\Gamma(\frac{d}{2}+1)},$$

  où Γ est la fonction Gamma $\Gamma(m) = \int_0^{+\infty} x^{m-1}e^{-x}dx$, et si on varie le rayon r de la boule, le volume change en suivant la formule $V_d \cdot r^d$.

  En vue de l'équation (10), avec les données $\{x_1, \ldots, x_n\}$, on peut estimer $f(x)$ de la manière "naturelle" suivante : soit k un entier avec $1 \leq k \leq n$, $R_{(k)}(x) = ||x_{(k)}(x) - x||$ la distance de x à son k plus proche voisin, et μ_n la fonction de répartition empirique, où pour tout borélien $A \subseteq \mathbb{R}^d$, $\mu_n(A) = \frac{1}{n}\sum_{i=1}^n \mathbb{1}_{\{x_i \in A\}}$. Alors, l'estimateur par les k plus proches voisins est défini comme

$$f_n(x) = \frac{\mu_n(B(x, R_{(k)}))}{|B(x, R_{(k)})|} = \frac{k}{nV_d||x_{(k)}(x) - x||^d}, \quad x \in \mathbb{R}^d. \tag{11}$$

- *Estimation par noyau (Kernel density estimation) :* C'est possiblement la méthode d'estimation la plus habituelle et étudiée. En résumé, c'est une généralisation de la notion d'histogramme, mais facilement réalisable en dimensions plus élevées, et (souvent) aussi continue et différenciable.
  L'idée est de construire $\hat{f}$ en additionnant plusieurs petites fonctions centrées chacune sur une donnée. On appelle ces petites fonctions *noyaux*, qui sont toujours réelles, non-négatives et intégrables. De plus, en général on assume aussi, pour notre fonction noyau $K(x)$, que $\int_{\mathbb{R}^d} K(x)dx = 1$ (i.e. $K(x)$ est une fonction de densité) et que K est radiale ($K(-x) = K(x)$ quand $d = 1$, K constant sur $S_r = \{x \in \mathbb{R}^d | \; ||x|| = r\}$ en général).
  Prenons maintenant nos données $(x_1, \ldots, x_n)$ (indépendantes et identiquement distribuées, obtenues à partir de f). Nous supposerons désormais que $d = 1$ pour simplifier les notations, bien que pour d général les constructions suivants sont aussi valides avec quelques légères mo-

difications. En choisissant une fonction noyau $K(x)$, nous construisons la fonction $\hat{f}$ comme :

$$\hat{f}_h(x) = \frac{1}{n}\sum_{i=1}^{n} K_h(x - x_i) = \frac{1}{nh}\sum_{i=1}^{n} K\big(\frac{x - x_i}{h}\big). \tag{12}$$

Ici, h est un *paramètre d'échelle* à déterminer, mais qui a une grand influence sur l'estimation finale. Ce paramètre détermine de quelque sorte la largeur de la fonction noyau autour de chaque donnée. C'est aussi immédiat de vérifier que quand $\int_{\mathbb{R}} K(x)dx = 1$, on a $\int_{\mathbb{R}} f_h(x)dx = 1$.
Dans la pratique, la meilleure valeur de ce paramètre est difficile à déterminer, donc il y a toujours un compromis entre biais et variance. Différents travaux essaient d'étudier les meilleures valeurs de h en fonction de chaque situation. En tout cas, bien que certaines indications existent (par exemple, avec un noyau gaussienne, il est habituel de prendre $h \approx 1.06 \cdot \min\big(\hat{\sigma}, \frac{EIQ}{1.34}\big) \cdot n^{-\frac{1}{5}}$, où $\hat{\sigma}$ est l'estimateur de l'écart-type habituel et EIQ est l'écart interquartile), normalement l'estimation est faite avec plusieurs valeurs de h et on prend celle qui donne le meilleur résultat.

En ce qui concerne les fonctions noyau, nous remarquons différentes options. Nous montrons, pour $d = 1$ (mais facilement généralisables à d supérieure en prenant $||x||$ au lieu de x, et en changeant légèrement quelques coefficients en fonction de la dimension), certaines des plus utilisées, mais sans entrer dans les détails et particularités de chacune :

1. Noyau gaussienne : $K(x) = \dfrac{1}{\sqrt{2\pi}}\, e^{-\frac{1}{2}x^2}$.
2. Noyau uniforme : $K(x) = \dfrac{1}{2}\, \mathbb{1}(x)_{\{-1\leq\, x\, \leq 1\}}$.
3. Noyau triangulaire : $K(x) = (1 - |x|)\ \mathbb{1}(x)_{\{-1\leq\, x\, \leq 1\}}$.
4. Noyau de Epanechnikov (parabolique) : $K(x) = \dfrac{3}{4}(1 - x^2)\ \mathbb{1}(x)_{\{-1\leq\, x\, \leq 1\}}$.
5. Noyau tricubique : $K(x) = \dfrac{70}{81}(1 - |x|^3)^3\ \mathbb{1}(x)_{\{-1\leq\, x\, \leq 1\}}$.

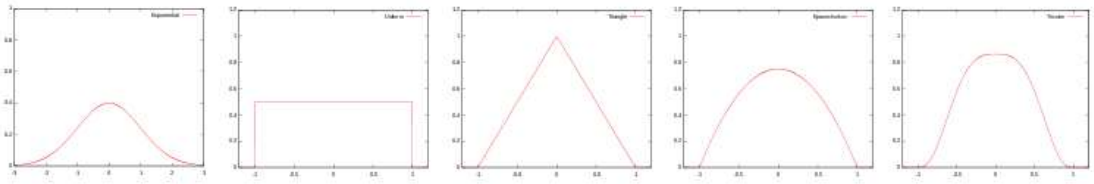


FIGURE 10 – Toutes les fonctions noyaux mentionnées, dans le même ordre.

4.3. La procédure de l'algorithme

Expliquons maintenant comment l'algorithme obtient les différents clusters. Supposons que nous avons un graphe de voisinage G entre les points, des valeurs $\hat{f}(i)$ pour chaque sommet i, et le paramètre de fusion τ. L'algorithme de base se compose de deux parties :

1. (Recherche de modes) Pour calculer les clusters de départ, ToMATo ordonne d'abord tous les sommets de manière décroissante en fonction de sa valeur $\hat{f}$. Avec cet ordre, il passe par

chaque sommet i et il simule un effet de gradient de la fonction de densité de base : il connecte i avec son voisin dans G avec la $\hat{f}$-valeur la plus haute. Si tous ses voisins ont des valeurs inférieures, i est étiqueté comme un "pic" (ou mode) de $\hat{f}$.
En regardant les arêtes "de gradient" à la fin de ce processus, on obtient ce qu'on appelle une "forêt couvrante" du graphe G, une notion similaire à celle de "arbre couvrant" (i.e. un sub-graphe de G connexe et acyclique qui passe pour tous les sommets de G) ; cependant, dans ce cas, la connectivité n'est pas exigée. Chaque composante connexe dans une forêt est un arbre avec la définition usuelle dans la théorie de graphes ; d'ici provient le nom.
Chaque arbre dans cette forêt couvrant peut être vu comme l'équivalent discret d'une région ascendante de f dans le cas continu, expliqué en 4.1.1, donc un cluster initial de notre nuage de points. Pour conserver toute cette information, on peut numéroter les q arbres obtenus et étiqueter chaque sommet avec son arbre correspondant. Dans un arbre i, on appelle $r(i)$ (r de "racine") son sommet avec la $\hat{f}$-valeur la plus haute, et on appelle aussi $a(j)$ l'arbre associé à un sommet j.

2. (Fusion des arbres) Bien que l'idée de la première étape soit logique, donc elle regroupe des données dans des "bosses" de $\hat{f}$, elle est aussi un peu aléatoire et inévitablement insuffisant. Dans cette deuxième étape, l'objectif es de fusionner les arbres "similaires", en adaptant la notion de persistance topologique introduite à 4.1.1.
Pour cela, ToMATo passe plusieurs fois sur les sommets de G dans le même ordre. Ici, tous les sommets sont déjà étiquetés dans un arbre. Dans cette itération, étant donné un sommet i, deux cas sont possibles :
   (a) i est déjà un pic d'un arbre, et donc aussi sa racine, et tous les voisins de i ont des $\hat{f}$-valeurs inférieures. Dans ce cas, les correspondances entre arbres et sommets restent inaltérées.
   (b) i n'est pas le pic de $a(i)$, et on cherche des fusions potentielles entre $a(i)$ et d'autres arbres "à côté".
   Pour cela, on prend les voisins k de i dans G (aussi de manière ordonnée) qui satisfont $\hat{f}(k) \geq \hat{f}(i)$, et on regarde si $\hat{f}(r(a(k)) \leq \min\{\hat{f}(r(a(i)), \hat{f}(i) + \tau\}$; ainsi, on étude si le pic de $a(k)$ est inférieur à celui de $a(i)$ et si sa proéminence est inférieure à τ. Si c'est le cas, toutes les sommets appartenant à $a(k)$ sont réétiquetés à $a(i)$. De la même manière, nous pouvons vérifier si $\hat{f}(r(a(i)) \leq \min\{\hat{f}(r(a(k)), \hat{f}(k) + \tau\}$, et réétiqueter les sommets de $a(i)$ à $a(k)$ si c'est le cas.

   À la fin de cette deuxième étape, tous les arbres (mini-clusters) de départ avec des pics de proéminence moins de τ et avec des sommets "connectés" à d'autres arbres ont été unifiés les uns avec les autres pour créer des arbres plus grands, et avec une proéminence d'au moins τ (nos clusters finaux). De plus, on a enregistré dans quel arbre/ cluster chaque donnée appartient.

4.4. Information finale obtenue

Avec le processus expliqué précédemment, l'information finale obtenue semble claire : pour chaque donnée i, une étiquette $a(i)$, son cluster final associé. Néanmoins, la méthode précédente n'est pas la plus utile pour travailler avec le type de problèmes que l'on retrouve avec des données réelles. On peut donc utiliser les notions expliquées au troisième chapitre pour obtenir un algorithme plus flexible et informatif.

En effet, en reprenant la deuxième étape exposée précédemment, il est évident que la valeur τ joue un rôle essentiel dans l'algorithme ; c'est ce numéro qui décide quelle doit être la proéminence

minimale d'un pic-cluster pour ne pas être fusionné avec d'autres pics-clusters "proches". Cependant, dans la pratique, normalement nous n'avons pas connaissance de la valeur τ "idéale" pour obtenir le meilleur résultat de clustering. Par exemple, nous vons déjà remarqué que les valeurs $\hat{f}$ associées à chaque donnée dépendent de l'estimation choisie et, par conséquence, un "bon" paramètre τ, s'il existe, possiblement aussi.

C'est au moment de choisir une valeur de τ convenable que les diagrammes de persistance introduits au troisième chapitre deviennent utiles. L'idée est de créer une représentation graphique de la proéminence de tous les différents clusters pour mieux détecter quels sont spécialement proéminents. Avec cette information, nous pouvons ajuster τ pour obtenir un nombre de clusters plus naturel, avec les étiquettes correspondantes.

Notre diagramme de persistance peut être obtenu de la manière suivant :

— Au début, on crée un point (x, y) pour chaque arbre-cluster initial, qui a toujours un pic associé : son sommet avec l'estimation de f la plus élevée, un mode de f. La cordonné x stocke cette valeur, tandis que la cordonné y reste non-assignée.

— Puis, on commence à fusionner ces clusters initiaux, en suivant la deuxième étape expliquée dans la section précédente et en gardant une trace de ces fusions. Intuitivement, on peut imaginer le paramètre de fusion τ qui vaut 0 au début, et qui augmente progressivement. Chaque fois que deux clusters sont fusionnés, on enregistre la mort du plus "petit" (i.e. moins proéminent, i.e. avec un pic associé moins haut) dans la cordonnée y, qui prend la valeur $y = x - \tau$, tandis que le plus "grand" continue d'exister.

— Ce processus continue jusqu'à ce que toutes les fusions possibles aient lieu. À ce moment, seulement les clusters associés aux composantes connexes du graphe de voisinage restent en vie, et on leur assigne la cordonnée $y = -\infty$.

Enfin, on obtient un ensemble de points qui encode d'une manière assez complète les proéminences relatives de tous les clusters/basins de f, où les distances (verticales) entre les (x, y) et la diagonale sont leur proéminence. Il est recommandable de dessiner les points avec $y = -\infty$ avec une couleur différente, pour mieux identifier dans le diagramme le nombre de composantes connexes existantes.

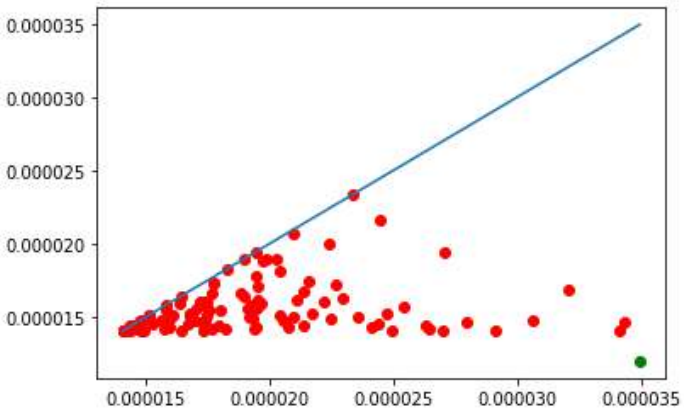


FIGURE 11 – Exemple de diagramme de persistance du type que nous venons d'exposer.

On voit rapidement que le diagramme de persistance obtenu n'est qu'une variation du diagramme de persistance 0-dimensionnel associé aux ensembles de super-niveau d'une fonction f,

comme celui décrit à l'exemple 3.26 (où on prenait les ensembles de sous-niveau). Cependant, la "connectivité" est encodée dans un graphe de voisinage, et les points habitent la moitié inférieure de la diagonal. Néanmoins, les différents résultats sur les diagrammes de persistance, notamment ceux liés à sa stabilité (section 3.5), peuvent être appliqués avec de petites variations dans notre cas discret.

Nous remarquons que, même si nous avons exposé la méthode en imaginant que la valeur τ augmente progressivement, au niveau algorithmique tous les points du diagramme de persistance peuvent être calculés d'une façon directe : au début, et aussi après chaque fusion, on parcourt tous les sommets ordonnés en fonction de $\hat{f}$ en cherchant, pour un sommet i avec son pic correspondant p_i, un voisin k dans un cluster différent qui satisfait $\hat{f}(k) \geq \hat{f}(i)$ et $\hat{f}(p_k) \leq \hat{f}(p_i)$ (resp. $\hat{f}(p_k) \leq \hat{f}(p_i)$). Dans ce cas, on assigne la valeur $y = \hat{f}(p_k) - \hat{f}(i)$ au cluster du sommet k, et tous ses sommets sont étiquetés avec le cluster du sommet i, et p_i devient son nouveau pic (resp. on assigne la valeur $y = \hat{f}(p_i) - \hat{f}(i)$ au cluster du sommet i, et tous ses sommets sont étiquetés au cluster du sommet k, et p_k devient son nouveau pic).

Dans tous les cas, ce diagramme de persistance devient très utile pour choisir une bonne valeur de τ pour l'algorithme original, exposé à la section précédant : il convient de regarder quels points sont de manière naturel plus éloignés de la diagonale (et combien il y en a) ; puis, nous choisissons une valeur τ inférieure a sa proéminence, les laissant intacts à la fin. En fait, après avoir calculé toutes les proéminences relatives pour dessiner le diagramme de persistance, on peut coder l'algorithme d'une manière encore plus intuitive : au lieu de donner une valeur τ d'entrée, on donne le numéro de clusters final souhaité m, et les fusions continuent de se produire jusqu'à ce que seulement les m clusters les plus proéminents restent. Cependant, il faut prendre en compte que l'algorithme ne peut pas fusionner des composantes connexes différentes (qui ont une proéminence "infinie").

4.5. Mise en œuvre de l'algorithme et exploration

L'algorithme ToMATo exposé à ce chapitre vient d'être implémenté à Python/ C++ et ajouté à la libraire GUDHI [8], une des librairies de référence de la TDA. Cette librairie open-source, codée en C++ mais avec une interface Python, offre des méthodes et ressources pour construire des complexes simpliciales et d'autres structures sur des nuages de points, et calculer les différents types d'homologie persistante.

La partie la plus pratique de ce travail a été de bien comprendre cette implémentation, réalisée par le chercheur Marc Glisse. Puis, nous avons essayé de tester ses limites et possibles erreurs. Cela a été fait par correspondance virtuelle avec plusieurs Jupyter notebooks. Cela a impliqué un apprentissage continue de Python et d'autres outils de programmation qui sont très pratiques et habituels dans le monde de la science des données et sur le marché du travail en général.

Finalement, avec le code déjà définitif, il paraissait approprié de préparer aussi un tutoriel de référence (en anglais) montrant toutes les options du code. Dans ce dernier, plusieurs exemples illustratifs aideraient et guideraient les utilisateurs potentiels. Le tutoriel final est annexé en PDF à la fin de ce travail. Il peut aussi être consulté en version HTML (de façon temporaire) avec le lien nilgarces.com/tomato.html.

Références

[1] Gérard Biau et Luc Devroye. *Lectures on the Nearest Neighbor Method*. Springer Series in Data Sciences. Springer New York Inc.

[2] Jean-Daniel Boissonant, Frédéric Chazal et Mariette Yvinec. « Geometric and Topological Inference ». In : (2018).

[3] Frédéric Chazal et Bertrand Michel. « An introduction to Topological Data Analysis : fundamental and practical aspects for data scientists ». In : (oct. 2017).

[4] Frédéric Chazal et al. « Persistence-Based Clustering in Riemannian Manifolds ». In : *Journal of the ACM* 60 (juin 2011). doi : 10.1145/1998196.1998212.

[5] David Cohen-Steiner, Herbert Edelsbrunner et John Harer. « Stability of Persistence Diagrams ». In : *Discrete Computational Geometry* 37 (2007), p. 103-120.

[6] Herbert Edelsbrunner, David Letscher et Afra Zomorodian. « Topological Persistance and Simplification ». In : *Discrete Computational Geometry* 28 (2002), p. 511-533.

[7] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, Inc., 2017. isbn : 9781492032632.

[8] *GUDHI Library*. url : https://gudhi.inria.fr/. (accessed : 20.06.2020).

[9] Trevor Hastie, Robert Tibshirani et Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., 2001.

[10] Allen Hatcher. *Algebraic topology*. Cambridge : Cambridge Univ. Press, 2000. url : https://cds.cern.ch/record/478079.

[11] F. Pedregosa et al. « Scikit-learn : Machine Learning in Python ». In : *Journal of Machine Learning Research* 12 (2011), p. 2825-2830.

[12] *Scikit-Learn documentation : Clustering*. url : https://scikit-learn.org/stable/modules/clustering.html. (accessed : 20.05.2020).

A handy guide to using the ToMATo algorithm

Introduction

This code is an implemention of the ToMATo algorithm exposed in [1], a clustering method based on the idea of topological persistance. In short, the algorithm needs a density estimation (so to each point x we associate a value $\hat{f}(x)$) and a neighborhood graph. First, it starts with a mode-seeking phase (naive hill-climbing) to build the initial clusters (each with its own mode), following the connected points in the neighborhood graph. Finally, it merges these initial clusters based on their prominence. This merging has a hierarchical nature, i.e. we always obtain the successive new clusters by merging two existing ones.

The merging phase depends on a parameter, which is the minimum prominence a cluster needs to avoid getting merged into another, adjacent, bigger cluster (i.e. with a higher associated mode); thus, it determines to a great extent the obtained number of clusters. In practice, the convenience of this parameter depends on the input graph and the density estimation, and it can be hard to choose it properly. This is why, in our implemention, we allow instead the option to choose the "desired" final number of clusters n, and the algorithm itself, after computing the initial clusters as well as their prominences, keeps merging them "parameterless-ly" until only the n clusters with highest prominence remain (if possible).

Along with the clustering itself, the algorithm also produces the *persistence diagram* of the merge tree of the initial clusters. This is a really convenient graphical tool to help decide the "natural" number of clusters in our input data. We explain its interpretation briefly in the section "Output information".

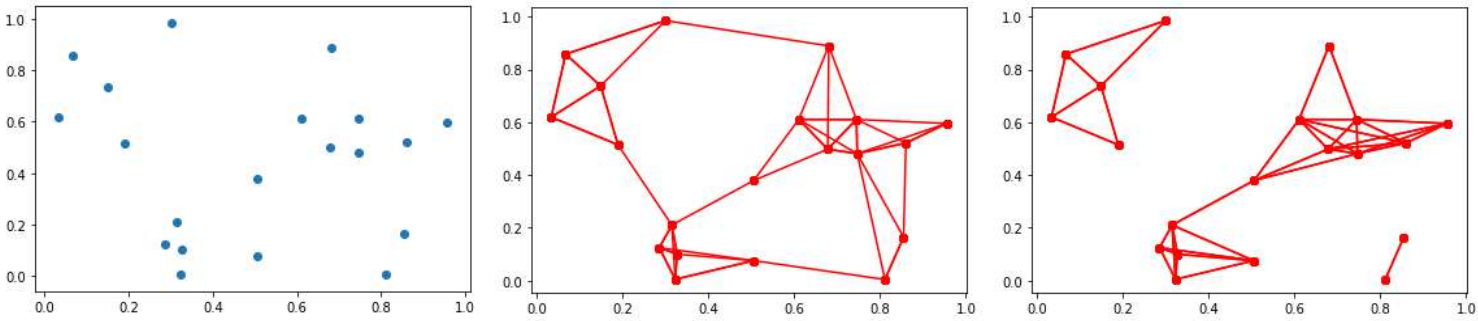
Input data format

As mentioned, the algorithm needs a neighborhood graph of the data and a value associated each entry (an estimation of f over it). Given that, in many situations, the input data is a point cloud (i.e. a set of n observations each with p numerical features), the code provides a few density estimators and graph constructions over them for convenience, but advanced users may provide their own graph and density estimates instead of point coordinates.

Since the algorithm essentially computes basins of attraction, it is also encouraged to use it on functions that do not represent densities at all.

For an input point cloud, the density estimation and graph construction methods that have been implemented are:

- For density estimation, the ubiquitous Kernel Density Estimation (KDE for short) can be used (using the scikit-learn library), and also the Distance-to-a-Measure method (DTM), a bit more experimental and recently developed to face more efficiently the potential presence of outliers; more information about it can be found in the tutorial [2] and the paper [3]. The logarithmic versions of both estimation methods are also implemented.
- Regarding the building of the graph, there is the option to construct the k-NN graph (where, for each vertex, an edge is created between it and its k nearest neighbors), and the r-radius graph (where an edge is created whenever two vertices lay in a distance less than r). Obviously, both parameters can (and should) be properly chosen. In the following image we can see both constructions over a point cloud in the square 1x1 (first image); in the second one, we have the k-NN graph (with k=4), while in the third we have the r-radius graph (with r=0.3):



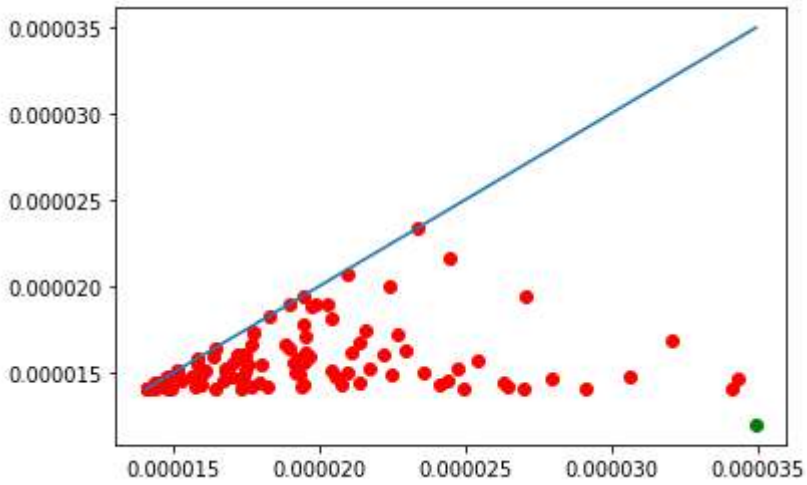
Output information

At the end, the algorithm outputs basically two informations of interest:

In all cases, it produces the (0-dimensional) *persistance diagram* of the merging process of the initial clusters. In short, this is a graphical representation of the lifespan of the different clusters as we keep diminishing the prominence threshold.

At the beginning, we have a point for each initial cluster, which also has an associated peak (the vertex with the highest estimate of f, a "mode" of f). Then, we start looking for merges of these clusters, by melding them with neighboring clusters with higher associated peaks. To do so, we basically keep checking, for the different vertices i (in decreasing order), which "neighboring" peaks p_j lower than p_i satisfy $\hat{f}(p_j) < \hat{f}(i) + \tau$, where τ is our prominence value. When this happens, we merge the whole cluster associated to that peak p_j to the one in which i belongs, forming a new, bigger cluster, still with peak p_i. The higher τ needs to be before this happens, the more prominent is p_j and its associated cluster.

In a persistance diagram, all this information is encoded in the following way: there is a point (x, y) for each initial cluster. The x coordinate is the value of its associated peak p. The y coordinate is the value $\hat{f}(p) - \tau$ from which we can find a "neighboring point" of that peak, but belonging to a different cluster, with equal or greater $\hat{f}$; equivalently, it is the highest neighbor of p not belonging to the cluster it defines. Thus, the length of vertical line connecting (x, y) with the diagonal, or equivalently $x - y$, is the prominence of the peak. In consequence, to get an idea of the real number of clusters, it is natural to look for the number of points in the persistance diagram further away from the diagonal. The points associated to a peak of a cluster which never dies (i.e. it never gets merged, so it forms a connected component at the end) are colored in green.



In view of the persistance diagram obtained, it is then natural to ask for a specific number of clusters at the end, or to specify a certain persistance threshold. After this has been stipulated, the algorithm also outputs a numerical "label" for each entry in the input data (in the same order they have been introduced, whatever the format): the cluster it has been assigned to. This labelling is saved in the attribute "labels_" as an ordered vector, so it can be easily used to plot the data in different colors or formats depending on their assigned cluster.

THE TOMATO CLASS

The code now

This is the current version of the code in the Gudhi Library:

In [63]:
```
# This file is part of the Gudhi Library - https://gudhi.inria.fr/ - which is released under MIT.
# See file LICENSE or go to https://gudhi.inria.fr/licensing/ for full license details.
# Author(s):       Marc Glisse
#
# Copyright (C) 2020 Inria
#
# Modification(s):
#   - YYYY/MM Author: Description of the modification

import numpy
from ..point_cloud.knn import KNearestNeighbors
from ..point_cloud.dtm import DTMDensity
from ._tomato import *

# The fit/predict interface is not so well suited...


class Tomato:
    """
    This clustering algorithm needs a neighborhood graph on the points, and an estimation of the density at each point.
    A few possible graph constructions and density estimators are provided for convenience, but it is perfectly natural
    to provide your own.

    :Requires: `SciPy <installation.html#scipy>`_, `Scikit-learn <installation.html#scikit-learn>`_ or others
        (see :class:`~gudhi.point_cloud.knn.KNearestNeighbors`) in function of the options.

    Attributes
    ----------
    n_clusters_: int
        The number of clusters. Writing to it automatically adjusts `labels_`.
    merge_threshold_: float
        minimum prominence of a cluster so it doesn't get merged. Writing to it automatically adjusts `labels_`.
    n_leaves_: int
        number of leaves (unstable clusters) in the hierarchical tree
    leaf_labels_: ndarray of shape (n_samples,)
        cluster labels for each point, at the very bottom of the hierarchy
    labels_: ndarray of shape (n_samples,)
        cluster labels for each point, after merging
    diagram_: ndarray of shape (`n_leaves_`, 2)
        persistence diagram (only the finite points)
    max_weight_per_cc_: ndarray of shape (n_connected_components,)
        maximum of the density function on each connected component. This corresponds to the abscissa of infinite
        points in the diagram
    children_: ndarray of shape (`n_leaves_`-n_connected_components, 2)
        The children of each non-leaf node. Values less than `n_leaves_` correspond to leaves of the tree.
        A node i greater than or equal to `n_leaves_` is a non-leaf node and has children children_[i - `n_leaves_`].
        Alternatively at the i-th iteration, children[i][0] and children[i][1] are merged to form node `n_leaves_` + i
    weights_: ndarray of shape (n_samples,)
        weights of the points, as computed by the density estimator or provided by the user
    params_: dict
```

```
        Parameters like metric, etc
    """

    def __init__(
        self,
        graph_type="knn",
        density_type="logDTM",
        n_clusters=None,
        merge_threshold=None,
        #       eliminate_threshold=None,
        #           eliminate_threshold (float): minimum max weight of a cluster so i
t doesn't get eliminated
        **params
    ):
        """
        Args:
            graph_type (str): 'manual', 'knn' or 'radius'. Default is 'knn'.
            density_type (str): 'manual', 'DTM', 'logDTM', 'KDE' or 'logKDE'. When yo
u have many points,
                'KDE' and 'logKDE' tend to be slower. Default is 'logDTM'.
            metric (str|Callable): metric used when calculating the distance between
 instances in a feature array.
                Defaults to Minkowski of parameter p.
            kde_params (dict): if density_type is 'KDE' or 'logKDE', additional param
eters passed directly to
                sklearn.neighbors.KernelDensity.
            k (int): number of neighbors for a knn graph (including the vertex itsel
f). Defaults to 10.
            k_DTM (int): number of neighbors for the DTM density estimation (includin
g the vertex itself).
                Defaults to k.
            r (float): size of a neighborhood if graph_type is 'radius'. Also used as
default bandwidth in kde_params.
            eps (float): (1+eps) approximation factor when computing distances (ignor
ed in many cases).
            n_clusters (int): number of clusters requested. Defaults to None, i.e. no
merging occurs and we get
                the maximal number of clusters.
            merge_threshold (float): minimum prominence of a cluster so it doesn't ge
t merged.
            symmetrize_graph (bool): whether we should add edges to make the neighbor
hood graph symmetric.
                This can be useful with k-NN for small k. Defaults to false.
            p (float): norm L^p on input points. Defaults to 2.
            q (float): order used to compute the distance to measure. Defaults to di
m.
                Beware that when the dimension is large, this can easily cause overfl
ows.
            dim (float): final exponent in DTM density estimation, representing the d
imension. Defaults to the
                dimension, or 2 when the dimension cannot be read from the input (met
ric is "precomputed").
            n_jobs (int): Number of jobs to schedule for parallel processing on the C
PU.
                If -1 is given all processors are used. Default: 1.
            params: extra parameters are passed to :class:`~gudhi.point_cloud.knn.KNe
arestNeighbors` and
                :class:`~gudhi.point_cloud.dtm.DTMDensity`.
        """
        # Should metric='precomputed' mean input_type='distance_matrix'?
        # Should we be able to pass metric='minkowski' (what None does currently)?
        self.graph_type_ = graph_type
        self.density_type_ = density_type
```

```
        self.params_ = params
        self.__n_clusters = n_clusters
        self.__merge_threshold = merge_threshold
        # self.eliminate_threshold_ = eliminate_threshold
        if n_clusters and merge_threshold:
            raise ValueError("Cannot specify both a merge threshold and a number of c
lusters")

    def fit(self, X, y=None, weights=None):
        """
        Args:
            X ((n,d)-array of float|(n,n)-array of float|Sequence[Iterable[int]]): co
ordinates of the points,
                or distance matrix (full, not just a triangle) if metric is "precompu
ted", or list of neighbors
                for each point (points are represented by their index, starting from
 0) if graph_type is "manual".
            weights (ndarray of shape (n_samples)): if density_type is 'manual', a de
nsity estimate at each point
            y: Not used, present here for API consistency with scikit-learn by conven
tion.
        """
        # TODO: First detect if this is a new call with the same data (only threshold
changed?)
        # TODO: less code duplication (subroutines?), less spaghetti, but don't compu
te neighbors twice if not needed. Clear error message for missing or contradictory pa
rameters.
        if weights is not None:
            density_type = "manual"
        else:
            density_type = self.density_type_
            if density_type == "manual":
                raise ValueError("If density_type is 'manual', you must provide weigh
ts to fit()")

        if self.graph_type_ == "manual":
            self.neighbors_ = X
            # FIXME: uniformize "message 'option'" vs 'message "option"'
            assert density_type == "manual", 'If graph_type is "manual", density_type
must be as well'
        else:
            metric = self.params_.get("metric", "minkowski")
            if metric != "precomputed":
                self.points_ = X

        # Slight complication to avoid computing knn twice.
        need_knn = 0
        need_knn_ngb = False
        need_knn_dist = False
        if self.graph_type_ == "knn":
            k_graph = self.params_.get("k", 10)
            # If X has fewer than k points...
            if k_graph > len(X):
                k_graph = len(X)
            need_knn = k_graph
            need_knn_ngb = True
        if self.density_type_ in ["DTM", "logDTM"]:
            k = self.params_.get("k", 10)
            k_DTM = self.params_.get("k_DTM", k)
            # If X has fewer than k points...
            if k_DTM > len(X):
                k_DTM = len(X)
            need_knn = max(need_knn, k_DTM)
```

```
            need_knn_dist = True
            # if we ask for more neighbors for the graph than the DTM, getting the di
stances is a slight waste,
            # but it looks negligible
        if need_knn > 0:
            knn_args = dict(self.params_)
            knn_args["k"] = need_knn
            knn = KNearestNeighbors(return_index=need_knn_ngb, return_distance=need_k
nn_dist, **knn_args).fit_transform(
                X
            )
            if need_knn_ngb:
                if need_knn_dist:
                    self.neighbors_ = knn[0][:, 0:k_graph]
                    knn_dist = knn[1]
                else:
                    self.neighbors_ = knn
            elif need_knn_dist:
                knn_dist = knn
        if self.density_type_ in ["DTM", "logDTM"]:
            dim = self.params_.get("dim")
            if dim is None:
                dim = len(X[0]) if metric != "precomputed" else 2
            q = self.params_.get("q", dim)
            weights = DTMDensity(k=k_DTM, metric="neighbors", dim=dim, q=q).fit_trans
form(knn_dist)
            if self.density_type_ == "logDTM":
                weights = numpy.log(weights)

        if self.graph_type_ == "radius":
            if metric in ["minkowski", "euclidean", "manhattan", "chebyshev"]:
                from scipy.spatial import cKDTree

                tree = cKDTree(X)
                # TODO: handle "l1" and "l2" aliases?
                p = self.params_.get("p")
                if metric == "euclidean":
                    assert p is None or p == 2, "p=" + str(p) + " is not consistent w
ith metric='euclidean'"
                    p = 2
                elif metric == "manhattan":
                    assert p is None or p == 1, "p=" + str(p) + " is not consistent w
ith metric='manhattan'"
                    p = 1
                elif metric == "chebyshev":
                    assert p is None or p == numpy.inf, "p=" + str(p) + " is not cons
istent with metric='chebyshev'"
                    p = numpy.inf
                elif p is None:
                    p = 2  # the default
                eps = self.params_.get("eps", 0)
                self.neighbors_ = tree.query_ball_tree(tree, r=self.params_["r"], p=p
, eps=eps)

            # TODO: sklearn's NearestNeighbors.radius_neighbors can handle more metri
cs efficiently via its BallTree
            # (don't bother with the _graph variant, it just calls radius_neighbors).
            elif metric != "precomputed":
                from sklearn.metrics import pairwise_distances

                X = pairwise_distances(X, metric=metric, n_jobs=self.params_.get("n_j
obs"))
                metric = "precomputed"
```

```
            if metric == "precomputed":
                # TODO: parallelize? May not be worth it.
                X = numpy.asarray(X)
                r = self.params_["r"]
                self.neighbors_ = [numpy.flatnonzero(l <= r) for l in X]

        if self.density_type_ in {"KDE", "logKDE"}:
            # Slow...
            assert (
                self.graph_type_ != "manual" and metric != "precomputed"
            ), "Scikit-learn's KernelDensity requires point coordinates"
            kde_params = dict(self.params_.get("kde_params", dict()))
            kde_params.setdefault("metric", metric)
            r = self.params_.get("r")
            if r is not None:
                kde_params.setdefault("bandwidth", r)
            # Should we default rtol to eps?
            from sklearn.neighbors import KernelDensity

            weights = KernelDensity(**kde_params).fit(self.points_).score_samples(sel
f.points_)
            if self.density_type_ == "KDE":
                weights = numpy.exp(weights)

        # TODO: do it at the C++ level and/or in parallel if this is too slow?
        if self.params_.get("symmetrize_graph"):
            self.neighbors_ = [set(line) for line in self.neighbors_]
            for i, line in enumerate(self.neighbors_):
                line.discard(i)
                for j in line:
                    self.neighbors_[j].add(i)

        self.weights_ = weights
        # This is where the main computation happens
        self.leaf_labels_, self.children_, self.diagram_, self.max_weight_per_cc_ = h
ierarchy(self.neighbors_, weights)
        self.n_leaves_ = len(self.max_weight_per_cc_) + len(self.children_)
        assert self.leaf_labels_.max() + 1 == len(self.max_weight_per_cc_) + len(self
.children_)
        # TODO: deduplicate this code with the setters below
        if self.__merge_threshold:
            assert not self.__n_clusters
            self.__n_clusters = numpy.count_nonzero(
                self.diagram_[:, 0] - self.diagram_[:, 1] > self.__merge_threshold
            ) + len(self.max_weight_per_cc_)
        if self.__n_clusters:
            # TODO: set corresponding merge_threshold?
            renaming = merge(self.children_, self.n_leaves_, self.__n_clusters)
            self.labels_ = renaming[self.leaf_labels_]
            # In case the user asked for something impossible.
            # TODO: check for impossible situations before calling merge.
            self.__n_clusters = self.labels_.max() + 1
        else:
            self.labels_ = self.leaf_labels_
            self.__n_clusters = self.n_leaves_
        return self

    def fit_predict(self, X, y=None, weights=None):
        """
        Equivalent to fit(), and returns the `labels_`.
        """
        return self.fit(X, y, weights).labels_
```

```
    # TODO: add argument k or threshold? Have a version where you can click and it sh
ows the line and the corresponding k?
    def plot_diagram(self):
        """
        """
        import matplotlib.pyplot as plt

        l = self.max_weight_per_cc_.min()
        r = self.max_weight_per_cc_.max()
        if self.diagram_.size > 0:
            plt.plot(self.diagram_[:, 0], self.diagram_[:, 1], "ro")
            l = min(l, self.diagram_[:, 1].min())
            r = max(r, self.diagram_[:, 0].max())
        if l == r:
            if l > 0:
                l, r = 0.9 * l, 1.1 * r
            elif l < 0:
                l, r = 1.1 * l, 0.9 * r
            else:
                l, r = -1.0, 1.0
        plt.plot([l, r], [l, r])
        plt.plot(
            self.max_weight_per_cc_, numpy.full(self.max_weight_per_cc_.shape, 1.1 *
l - 0.1 * r), "ro", color="green"
        )
        plt.show()

    # Use set_params instead?
    @property
    def n_clusters_(self):
        return self.__n_clusters

    @n_clusters_.setter
    def n_clusters_(self, n_clusters):
        if n_clusters == self.__n_clusters:
            return
        self.__n_clusters = n_clusters
        self.__merge_threshold = None
        if hasattr(self, "leaf_labels_"):
            renaming = merge(self.children_, self.n_leaves_, self.__n_clusters)
            self.labels_ = renaming[self.leaf_labels_]
            # In case the user asked for something impossible
            self.__n_clusters = self.labels_.max() + 1

    @property
    def merge_threshold_(self):
        return self.__merge_threshold

    @merge_threshold_.setter
    def merge_threshold_(self, merge_threshold):
        if merge_threshold == self.__merge_threshold:
            return
        if hasattr(self, "leaf_labels_"):
            self.n_clusters_ = numpy.count_nonzero(self.diagram_[:, 0] - self.diagram
_[:, 1] > merge_threshold) + len(
                self.max_weight_per_cc_
            )
        else:
            self.__n_clusters = None
        self.__merge_threshold = merge_threshold
```

Description

Parameters

By "parameters" we mean the information we (must) provide to construct a specific instance of the class. They are given as arguments in the constructor function "__init__":

- **graph_type** (str): 'manual', 'knn' (default) or 'radius'.
- **density_type** (str): 'manual', 'DTM', 'logDTM' (default), 'KDE' or 'logKDE'. With many points, 'KDE' and 'logKDE' tend to be slower.
- **n_clusters** (int): number of clusters requested. Defaults to None, i.e. no merging occurs and we get the maximal number of clusters.
- **merge_threshold** (float): minimum prominence of a cluster so it doesn't get merged.

(Naturally, both n_clusters and merge_threshold cannot be provided simultaneously, as it can be deduced from the explanation of the algorithm)

- **metric** (str|Callable): metric used to compute the pairwase distances between points (if we don't input them). If None, use Minkowski of parameter p.
- **kde_params** (dict): if density_type is 'KDE' or 'logKDE', additional parameters passed directly to sklearn.neighbors.KernelDensity.
- **k** (int): number of neighbors for a k-NN graph (including the vertex itself). Defaults to 10.
- **k_DTM** (int): number of neighbors for the DTM density estimation (including the vertex itself). Defaults to k.
- **r** (float): size of a neighborhood if graph_type is 'radius'. Also used as default bandwidth in kde_params.
- **eps** (float): approximation factor when computing nearest neighbors (ignored in many cases).
- **symmetrize_graph** (bool): whether we should add edges to make the neighborhood graph symmetric. This can be useful with k-NN for small k. Defaults to false.
- **p** (float): norm L^p on input points (numpy.inf is supported without gpu). Defaults to 2.
- **dim** (float): final exponent in DTM density estimation, representing the dimension. Defaults to the dimension, or 2 when the dimension cannot be read from the input (metric is "precomputed").
- **q** (float): order used to compute the distance to measure. Defaults to dim. Beware that when the dimension is large, this can easily cause overflows.
- **n_jobs** (int): Number of jobs to schedule for parallel processing on the CPU. If -1 is given all processors are used. Default: 1.
- **params**: extra parameters are passed to the classes `gudhi.point_cloud.knn.KNearestNeighbors` and `gudhi.point_cloud.dtm.DTMDensity` , for example 'implementation="keops"' for the first one.

Attributes

By "attributes" we mean the properties, or variables, created within a class: they store its information, allow it to run some of its methods and functionalities, etc... We recall also that, as a common practice, the attributes of a class (those defined with self.) usually have some "_" in its name to make them more distinguishable within the code.

Naturally, the values of most of the attributes depend on the instance itself, and, depending on it, some of them will be present or not. Actually, many of the previous parameters have their corresponding attribute, as for example n_clusters_ and merge_threshold_ (which, when modified, can alter the values of other attributes, as the .setter propery shows), or they are stored inside the "params_" dictionary; input_type, metric,...

Other important attributes which are created specifically to run the desired methods and are not given as parameters are:

- n_leaves_ (int): Number of leaves (unstable clusters) in the hierarchical tree. Basically, the number of "temporary" clusters (or mini-clusters) we have along the way.
- leaf_labels_ (ndarray of shape (n_samples)): Cluster labels for each point, at the very bottom of the hierarchy.
- labels_ (ndarray of shape (n_samples)): Cluster labels for each point, after merging. Writing to n_clusters_ and merge_threshold_ automatically adjusts it.
- diagram_ (ndarray of shape (n_leaves_, 2)): Persistence diagram (only the finite points).
- weights_: (ndarray of shape (n_samples,)): Weights of the points, as computed by the density estimator or provided by the user.
- max_weight_per_cc_: (ndarray of shape (n_connected_components,)): Maximum of the density function on each connected component. This corresponds to the abscissa of infinite points in the diagram.

Methods

The Tomato class contains, in essence, two methods:

- The first one is the **.fit** method, which does basically everything: it processes the input data taking into account its format and the given arguments, it does the merging process depending on them, does the labelling of the entries and stores the points that will eventually form the persistance diagram. The method **.fit_predict** is identical, but it returns the labels vector. Both of them take as the input the coordinates of the points/ distance matrix/ neighborhood matrix, and possibly a "weights" vector, the estimate of f on each entry.

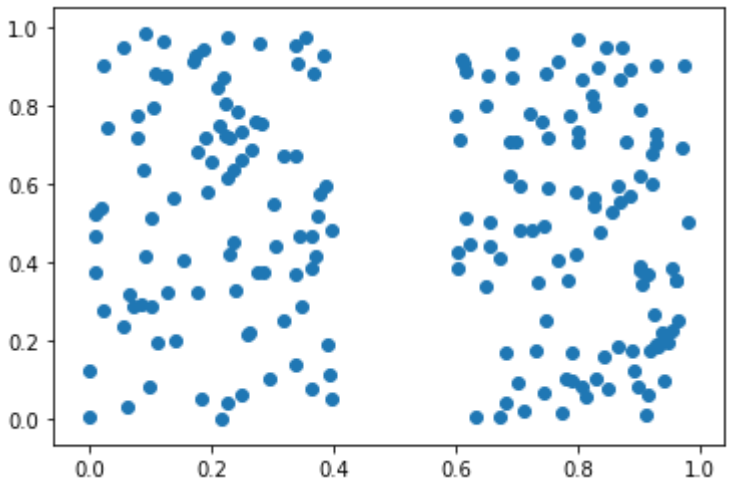
- The second one is the **.plot_diagram** method, without arguments, that plots the persistance diagram (after the fit method).

EXAMPLES AND TESTS

Example 1

We start with a really simple example with a few hundreds points to get used to manipulating the Tomato class.

In [32]:
```
import matplotlib.pyplot as plt

cmap = plt.cm.Spectral;
fig, ax = plt.subplots();

import random as rd
import numpy as np


# Simple function to get random values for x uniformly but within intervals (0,a) U
 (b, 1)
def x_var(x):
    if x > 0.5:
        return rd.uniform(0.6, 1)
    else:
        return rd.uniform(0, 0.4)


p1 = np.zeros((200,2))
for i in range(200):
    p1[i,0] = x_var(rd.uniform(0,1))
    p1[i,1] = rd.uniform(0,1)

ax.cla()
ax.scatter(*zip(*p1));
```

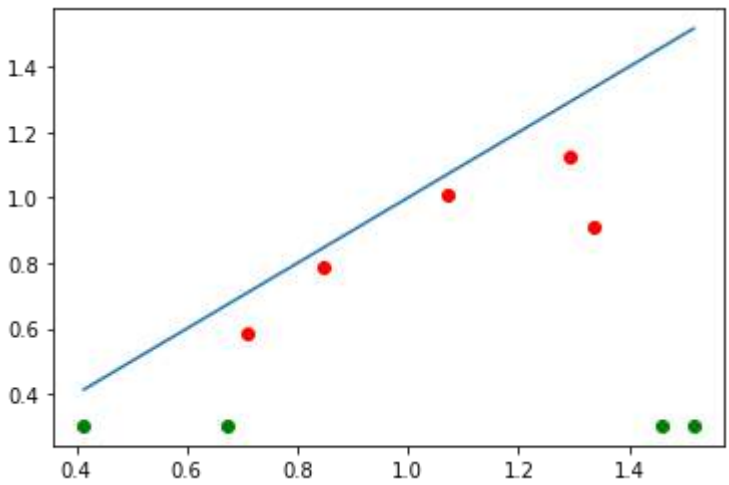


There are "clearly" two main groups of points.

Let's suppose we don't know that, so we run the Tomato algorithm blindly. We use the KDE (without specifying extra parameters, thus using the default parameters in Scikit-Learn) and the radius graph with r= 0.1. We want to take a look at the persistance diagram:

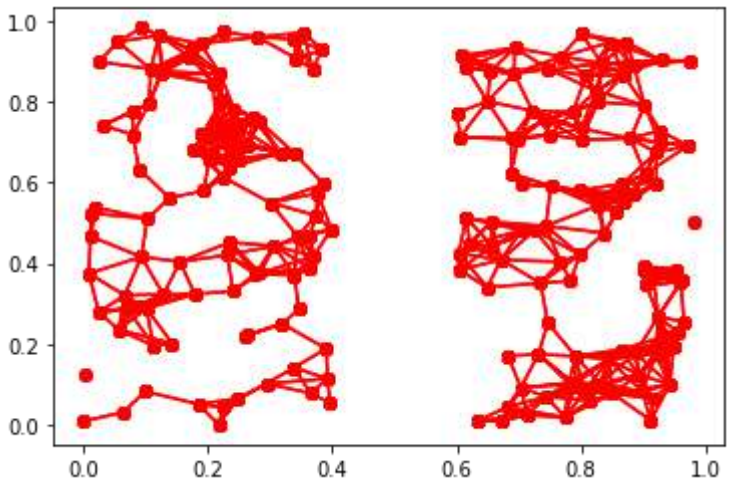
In [57]:
```
import gudhi

from gudhi.clustering.tomato import Tomato

ex1 = Tomato(
        input_type="points",
        metric="euclidean",
        graph_type="radius",
        density_type="KDE",
        #n_clusters=2,
        r=0.1,
    )

labels = ex1.fit_predict(p1)
print(labels)

print("\nThere are " + str(ex1.n_clusters_) + " initial clusters")
ex1.plot_diagram()
```

```
[1 4 1 6 3 1 0 2 3 1 3 0 3 0 0 4 1 4 1 4 1 2 0 3 1 4 3 1 4 1 2 1 3 3 1 6 0
 3 2 0 0 0 6 1 0 2 2 6 0 1 8 2 7 4 2 6 1 1 1 4 4 5 2 1 0 0 1 6 2 3 4 3 2 2
 3 1 1 2 1 0 6 2 4 1 0 3 2 1 1 3 2 0 4 3 2 3 0 3 3 0 1 4 0 0 3 0 3 6 3 4 5
 1 0 0 0 6 1 0 1 0 0 2 3 1 1 2 4 0 2 0 4 3 2 1 4 2 2 0 2 1 6 2 0 2 4 4 0 2
 0 1 3 5 2 4 4 1 1 1 2 4 1 1 3 3 0 5 1 0 1 4 3 1 4 2 1 1 4 1 3 1 6 1 2 2 0
 2 0 1 0 2 2 3 0 1 0 4 1 0 6 4]

There are 9 initial clusters
```



Even if n_clusters_ gives us 9 initial clusters (when we don't specify the parameter n_clusters in Tomato no merging occurs), we can see from the bottom-right that there are clearly two more prominent groups, but four connected components. Indeed, let's output the graph built on top of our data:

In [58]:
```
from gudhi.point_cloud.knn import KNearestNeighbors
X = np.array(p1)
nbrs = KNearestNeighbors(k=30, return_distance= True)
indices, distances = nbrs.fit_transform(X)
plt.plot(X[:,0], X[:,1], 'o')
for i in indices:
    Y = np.zeros((2,2))
    for j in range(len(i)):
        if distances[int(i[0]), j] < 0.1:
            Y[0][0]= X[int(i[0])][0]
            Y[1][0]= X[int(i[0])][1]
            Y[0][1]= X[int(i[j])][0]
            Y[1][1]= X[int(i[j])][1]
            plt.plot(Y[0], Y[1], 'ro-')

plt.show()
```



Even if we know that "there are" two main clusters, we cannot force the algorithm to output them, because there is no way the algorithm can merge disconnected components. We don't have problems if we ask for a bigger number of clusters:

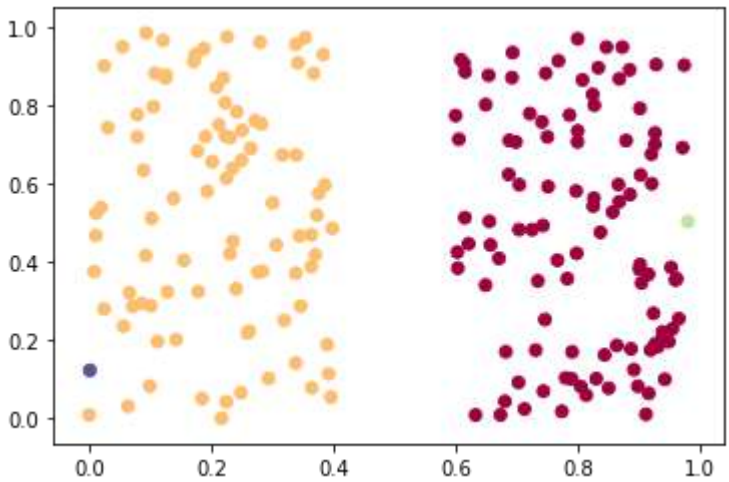
In [61]:
```
ex1.n_clusters_ = 6
print(ex1.n_clusters_)
print(ex1.labels_)

ex1.n_clusters_ = 2
print(ex1.n_clusters_)
print(ex1.labels_)
```

```
6
[1 1 1 1 3 1 0 2 3 1 3 0 3 0 0 1 1 1 1 1 1 2 0 3 1 1 3 1 1 1 2 1 3 3 1 1 0
 3 2 0 0 0 1 1 0 2 2 1 0 1 5 2 4 1 2 1 1 1 1 1 1 1 2 1 0 0 1 1 2 3 1 3 2 2
 3 1 1 2 1 0 1 2 1 1 0 3 2 1 1 3 2 0 1 3 2 3 0 3 3 0 1 1 0 0 3 0 3 1 3 1 1
 1 0 0 0 1 1 0 1 0 0 2 3 1 1 2 1 0 2 0 1 3 2 1 1 2 2 0 2 1 1 2 0 2 1 1 0 2
 0 1 3 1 2 1 1 1 1 1 1 2 1 1 1 3 3 0 1 1 0 1 1 3 1 1 2 1 1 1 1 3 1 1 1 2 2 0
 2 0 1 0 2 2 3 0 1 0 1 1 0 1 1]
4
[1 1 1 1 0 1 0 0 0 1 0 0 0 0 0 1 1 1 1 1 1 0 0 0 1 1 0 1 1 1 0 1 0 0 1 1 0
 0 0 0 0 0 1 1 0 0 0 1 0 1 3 0 2 1 0 1 1 1 1 1 1 1 0 1 0 0 1 1 0 0 1 0 0 0
 0 1 1 0 1 0 1 0 1 1 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 0 1 1
 1 0 0 0 1 1 0 1 0 0 0 0 1 1 0 1 0 0 0 1 0 0 1 1 0 0 0 0 1 1 0 0 0 1 1 0 0
 0 1 0 1 0 1 1 1 1 1 0 1 1 1 0 0 0 1 1 0 1 1 0 1 1 0 1 1 1 1 0 1 1 1 0 0 0
 0 0 1 0 0 0 0 0 1 0 1 1 0 1 1]
```

Unsurprisingly, if we plot the points with different colors according to their labels, we don't get a very satisfying result:

In [62]:
```
n = ex1.n_clusters_
labels = ex1.labels_

norm = plt.Normalize(vmin=0, vmax=n-1)

ax.cla()
ax.scatter(*zip(*p1), c=cmap(norm(labels)))
fig
```

Out[62]:



This is the reason why running the algorithm for different values of the parameters is a good idea, specially if the algorithm produces persistance diagrams with several green dots (i.e. connected components) near the bottom-left part (i.e. low, isolated peaks).

Here is the situation when we increase r to 0.15:

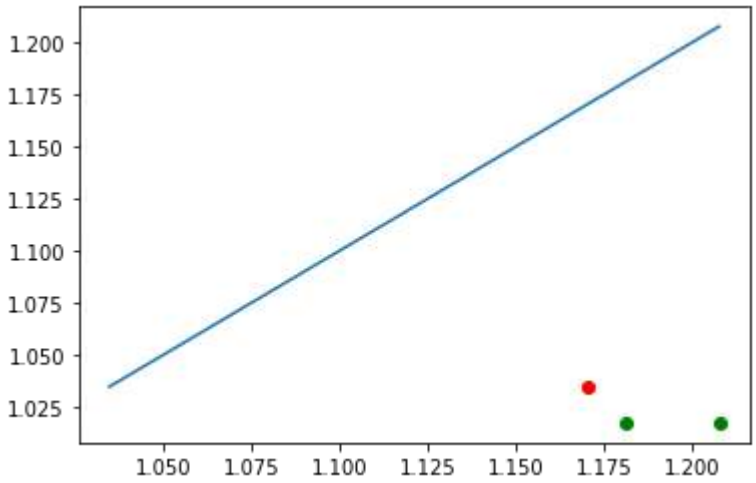
In [63]:

```
ex1 = Tomato(
        input_type="points",
        metric="euclidean",
        graph_type="radius",
        density_type="KDE",
        n_clusters=2,
        r=0.13,
    )

n = ex1.n_clusters_
print("We obtain " + str(n) + " clusters.")
labels = ex1.fit_predict(p1)
print(ex1.labels_)

print("\nThe persistance diagram looks better, with just two connected components, an
d two prominent regions:")
ex1.plot_diagram()

print("\nThe graph over which the algorithm runs is:")

plt.plot(X[:,0], X[:,1], 'o')
for i in indices:
    Y = np.zeros((2,2))
    for j in range(len(i)):
        if distances[i[0]][j] < 0.15:
            Y[0][0]= X[i[0]][0]
            Y[1][0]= X[i[0]][1]
            Y[0][1]= X[i[j]][0]
            Y[1][1]= X[i[j]][1]
            plt.plot(Y[0], Y[1], 'ro-')

plt.show()

print("\nAnd the plot of the points according to their label is:")

norm = plt.Normalize(vmin=0, vmax=n-1)

ax.cla()
ax.scatter(*zip(*p1), c=cmap(norm(labels)))
fig
```

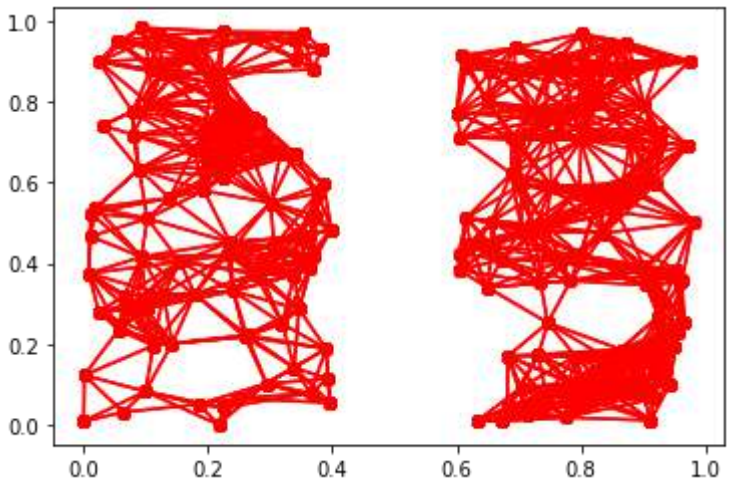
We obtain 2 clusters.

```
[1 1 1 1 0 1 0 0 0 1 0 0 0 0 0 1 1 1 1 1 1 0 0 0 1 1 0 1 1 1 0 1 0 0 1 1 0
 0 0 0 0 0 1 1 0 0 0 1 0 1 1 0 0 1 0 1 1 1 1 1 1 1 0 1 0 0 1 1 0 0 1 0 0 0
 0 1 1 0 1 0 1 0 1 1 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 0 1 1
 1 0 0 0 1 1 0 1 0 0 0 0 1 1 0 1 0 0 0 1 0 0 1 1 0 0 0 0 1 1 0 0 0 1 1 0 0
 0 1 0 1 0 1 1 1 1 1 0 1 1 1 0 0 0 1 1 0 1 1 0 1 1 0 1 1 1 1 0 1 1 1 0 0 0
 0 0 1 0 0 0 0 0 1 0 1 1 0 1 1]
```

The persistance diagram looks better, with just two connected components, and two pr
ominent regions:

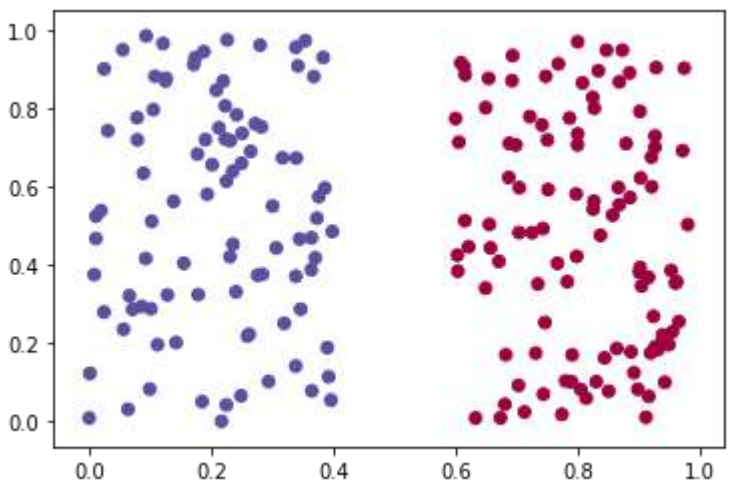


The graph over which the algorithm runs is:



And the plot of the points according to their label is:

Out[63]:

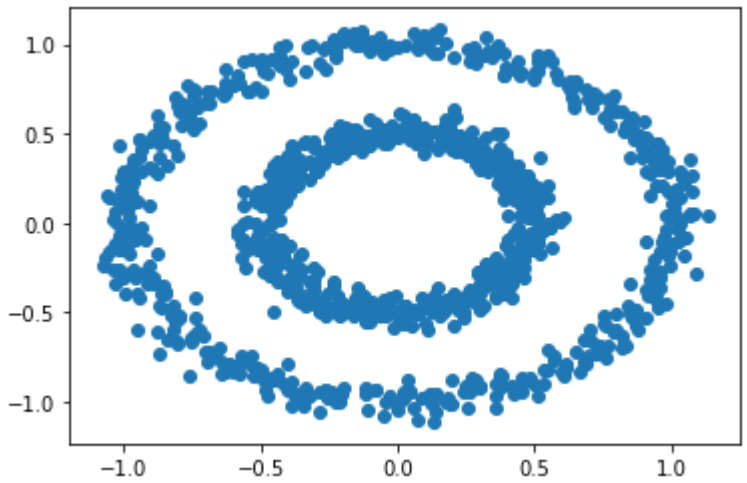


Example 2

We use now a rather typical example to test clustering algorithms: a point cloud sampled from two concentric circles:

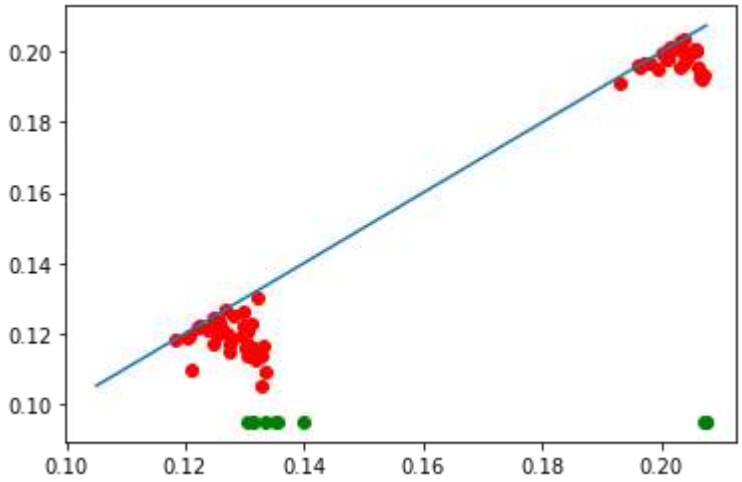
In [97]:
```
from sklearn import manifold, datasets
p2, y = datasets.make_circles(n_samples=1000, factor=.5, noise=.05)

ax.cla()
ax.scatter(*zip(*p2))
fig
```

Out[97]:



It is well known that many clustering methods perform poorly with non-convex groupings of data, as the one above. This is not the case with the Tomato algorithm, which relies just on looking for "nearby" modes. We use now the k-NN graph construction, with k=7, and the KDE again, specifying some of it parameters now (for more information, check the Scikit-learn documentation):

In [98]:
```
ex2 = Tomato(
        input_type="points",
        metric="euclidean",
        graph_type="knn",
        density_type="KDE",
        kde_params = {"bandwidth": 1.3, "kernel": "epanechnikov"},
        #n_clusters=2,
        k=7,
        eps=0.05,
    )

ex2.fit_predict(p2)
ex2.plot_diagram()
```



The diagram is not specially obvious; if this happens, it is in general a good idea tu run the algorithm with different values in the parameters.

We also see that there are several connected components, more specifically 9; a quick way to know how many of them we have is check the size of the attribute "max_weight_per_cc_":

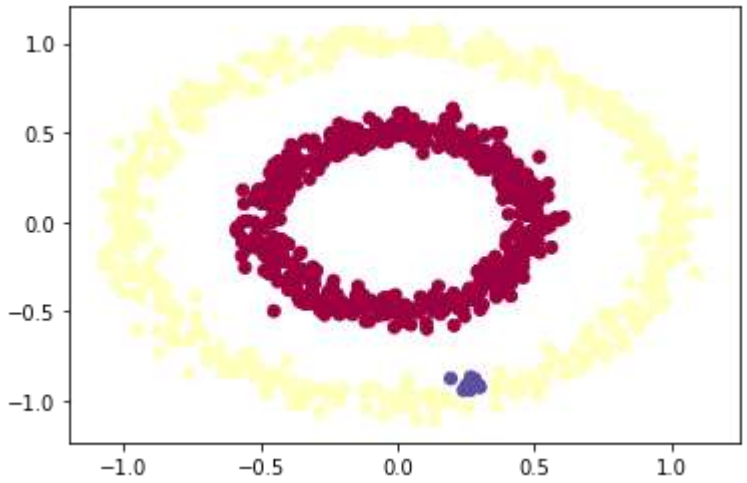
In [99]:
```
n = len(ex2.max_weight_per_cc_)
print("There are " + str(n) + " connected components")
```

```
There are 9 connected components
```

Let's plot these components:

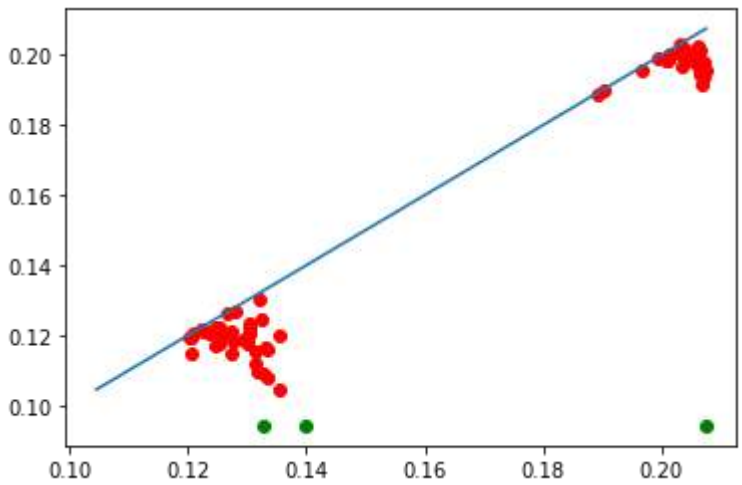
In [103]:
```
ex2.n_clusters_ = n
labels = ex2.labels_

norm = plt.Normalize(vmin=0, vmax=n-1)

ax.cla()
ax.scatter(*zip(*p2), c=cmap(norm(labels)))
fig
```

Out[103]:



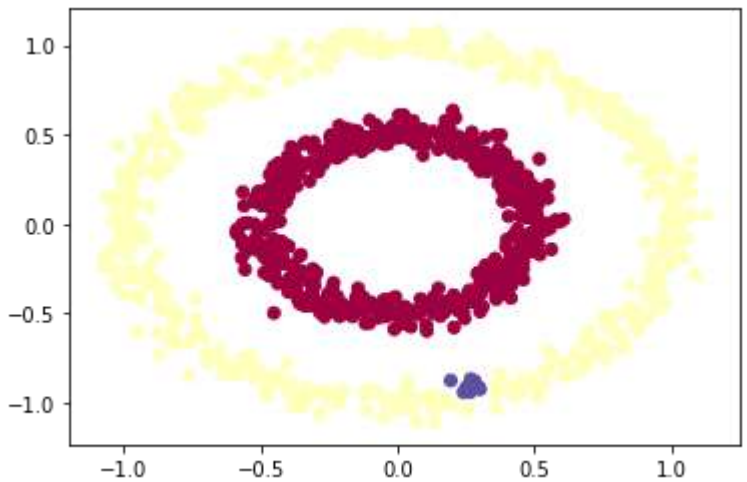
A bit frustrating; this is "natural" consequence of the the k-NN graph being directed. We can "solve" this by symmetrizing the graph, although its effectiveness is uncertain. In this case it also makes sense to reduce k, as we add more edges:

In [102]:
```
ex2 = Tomato(
        input_type="points",
        metric="euclidean",
        graph_type="knn",
        density_type="KDE",
        kde_params = {"bandwidth": 1.3, "kernel": "epanechnikov"},
        #n_clusters=2,
        k=5,
        symmetrize_graph = True,
        eps=0.05,
    )

ex2.fit_predict(p2)
ex2.plot_diagram()

n = len(ex2.max_weight_per_cc_)
print("There are " + str(n) + " connected components")

ex2.n_clusters_ = n
labels = ex2.labels_

norm = plt.Normalize(vmin=0, vmax=n-1)

ax.cla()
ax.scatter(*zip(*p2), c=cmap(norm(labels)))
fig
```




```
There are 3 connected components
```

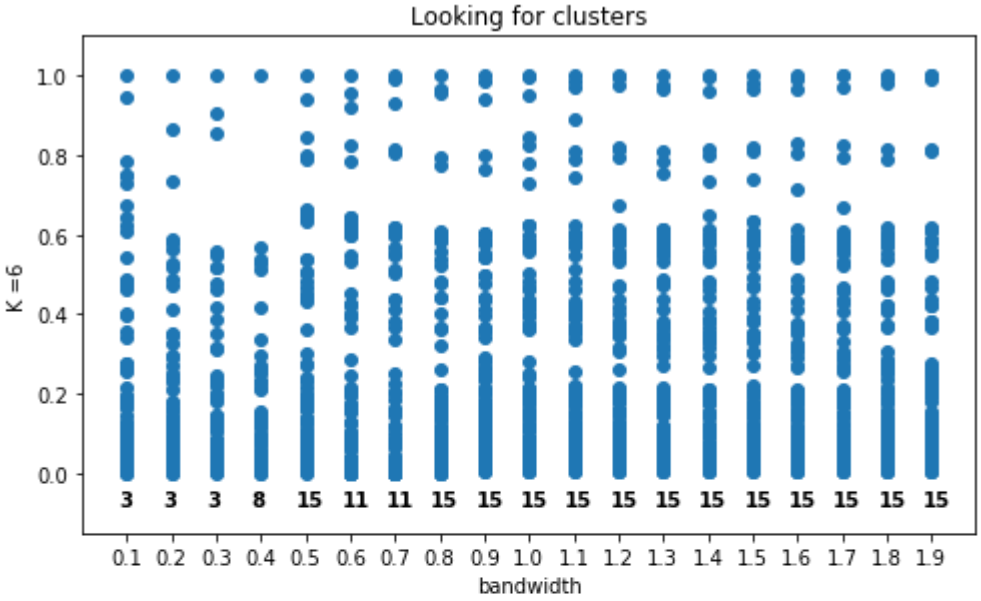
Out[102]:

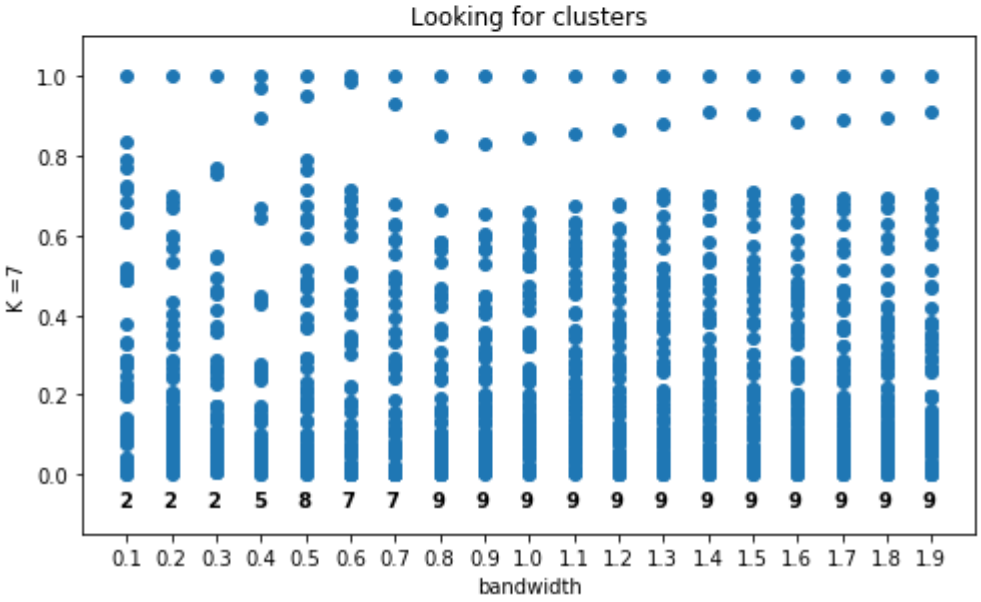


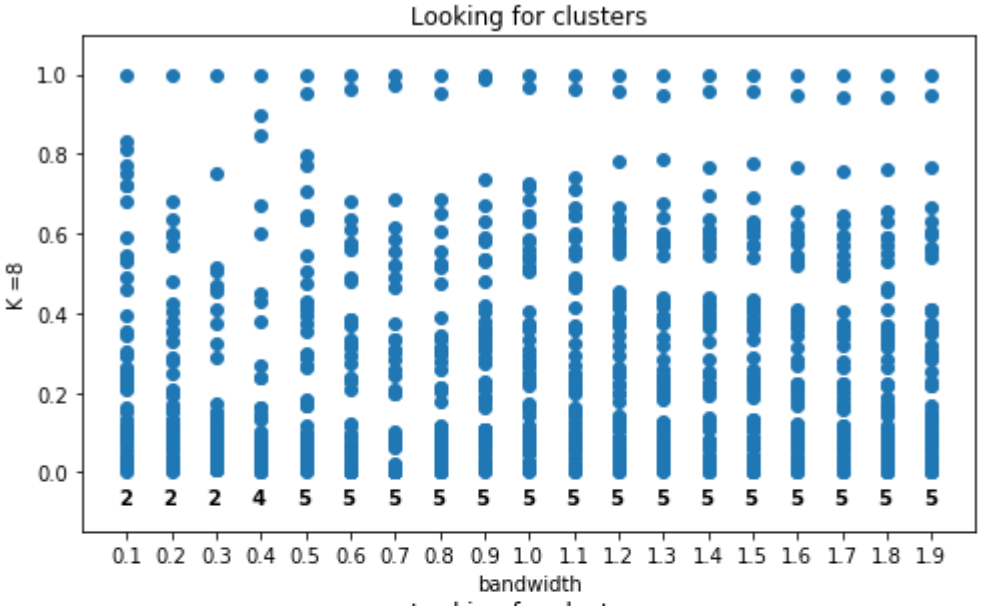
In general, and intelligent way to proceed would be to run the algorithm for different values of k and the bandwidth λ, and see for which values we obtain "good" persistance diagrams, with "clearly prominent clusters". This is what we do below, where, for a fixed k and different values of λ, we compute the prominence of each point of the persistance diagram ($x - y$), and we plot the information, as well as the number of connected components (number under every vertical bar) in each case:

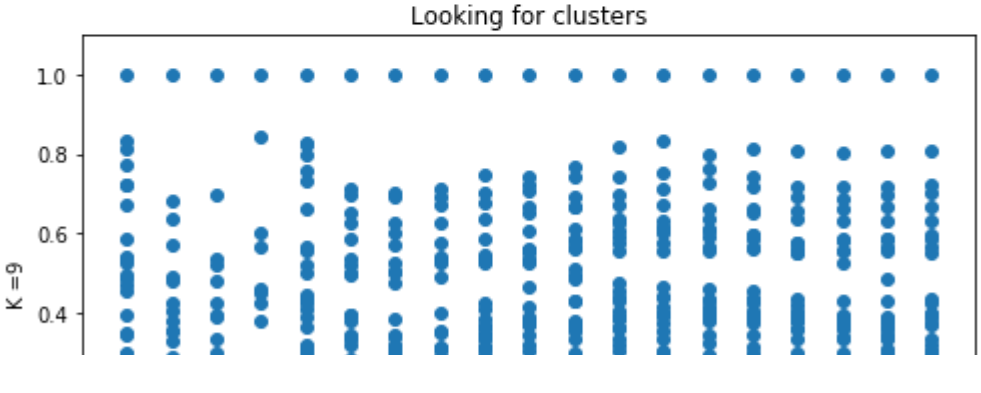
In [148]:

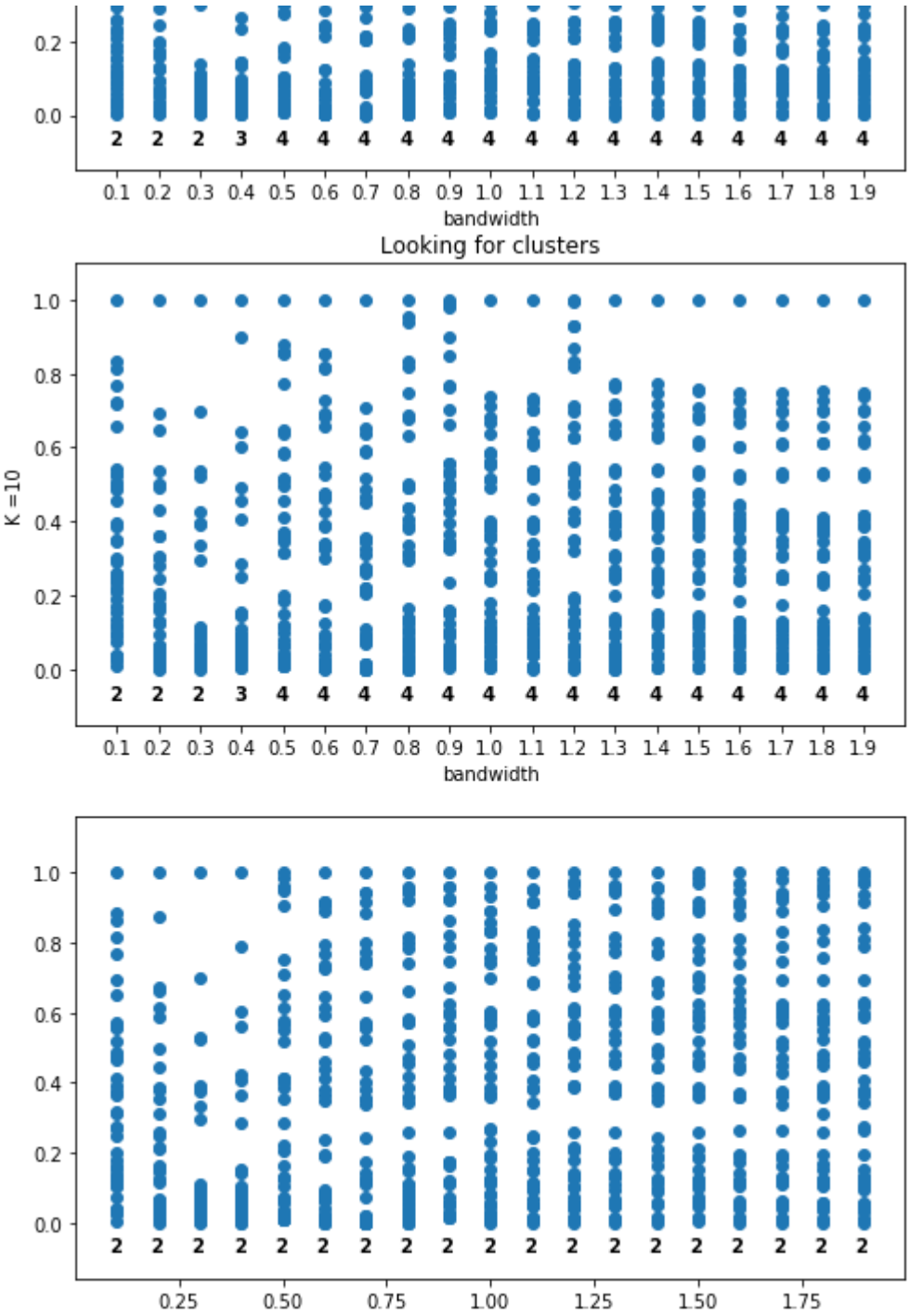
```
for n_neigh in range(6,12):
    n_diagram = []
    x_diagram = []
    y_diagram = []
    cc = []
    y_cc = []
    bandwidth_values = [0.1, 2, 0.1]
    bandwidth = bandwidth_values[0]

    while bandwidth < bandwidth_values[1]:
        ex2 = Tomato(
        input_type="points",
        metric="euclidean",
        graph_type="knn",
        density_type="KDE",
        kde_params = {"bandwidth": bandwidth, "kernel": "epanechnikov"},
        #n_clusters=2,
        k=n_neigh,
        eps=0.05,
    )
        ex2.fit(p2)
        cc.append(str(len(ex2.max_weight_per_cc_)))
        init_clusters = len(ex2.diagram_)
        prominences = np.zeros(init_clusters)
        for i in range(init_clusters):
            prominences[i] = ex2.diagram_[i,0] - ex2.diagram_[i,1]

        ##"Normalizing" prominences
        max_prom = np.max(prominences)
        for i in range(init_clusters):
            prominences[i] /= max_prom

        n_diagram.append(prominences)
        bandwidth += bandwidth_values[2]


    for i in range(len(n_diagram)):
        for j in range(len(n_diagram[i])):
            x_diagram.append(bandwidth_values[0] + i*bandwidth_values[2])
            y_diagram.append(n_diagram[i][j])
            y_cc.append(-0.08)


    plt.title('Looking for clusters')
    plt.axis('tight')
    plt.ylabel('K =' + str(n_neigh-1))
    plt.ylim((-0.15, 1.1))
    plt.xlabel('bandwidth')
    plt.xticks(np.arange(bandwidth_values[0], bandwidth_values[1], bandwidth_values[2
]))
    plt.subplot(6, 1, n_neigh-5)
    n = int((bandwidth_values[1]-bandwidth_values[0])/bandwidth_values[2]) + 1
    for i in range(n):
        plt.text(-0.02 + bandwidth_values[0] + i*bandwidth_values[2], y_cc[i], cc[i],
fontdict={'weight': 'bold', 'size': 10})

    plt.scatter(x_diagram, y_diagram)

fig = plt.gcf()
fig.set_size_inches(8, 32)
plt.show()
```


Looking for clusters
K =6
1.0
0.8
0.6
0.4
0.2
0.0
3 3 3 8 15 11 11 15 15 15 15 15 15 15 15 15 15 15 15
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9
bandwidth


Looking for clusters
K =7
1.0
0.8
0.6
0.4
0.2
0.0
2 2 2 5 8 7 7 9 9 9 9 9 9 9 9 9 9 9 9
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9
bandwidth

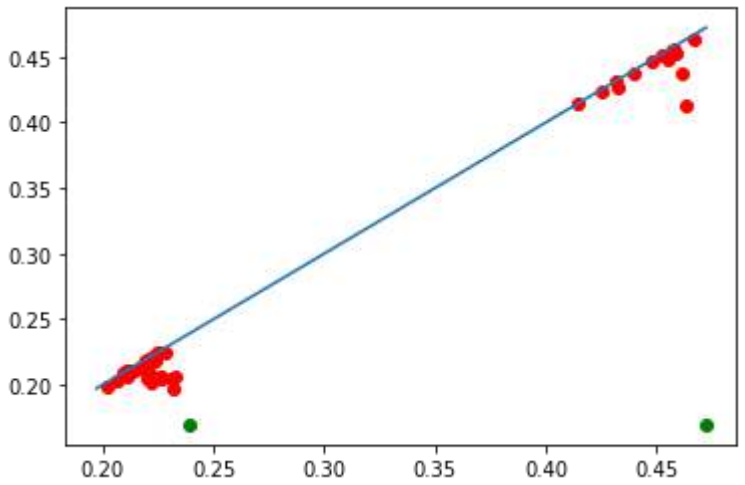

Looking for clusters
K =8
1.0
0.8
0.6
0.4
0.2
0.0
2 2 2 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9
bandwidth

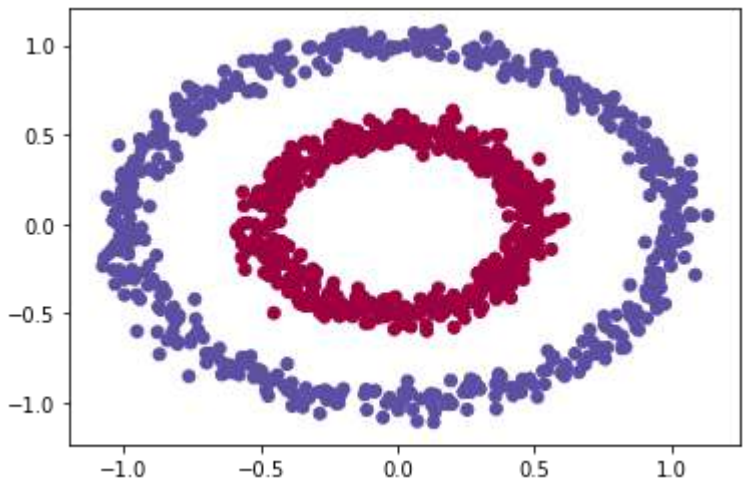

Looking for clusters
K =9
1.0
0.8
0.6
0.4



One can see, for example, that when the bandwidth is λ = 0.3, two more prominent clusters appear consistently, for all the last values of k, and we always get two connected components. If we run Tomato with these parameters, we obtain the "desired" result:

In [185]:
```
ex2 = Tomato(
        input_type="points",
        metric="euclidean",
        graph_type="knn",
        density_type="KDE",
        kde_params = {"bandwidth": 0.3, "kernel": "epanechnikov"},
        n_clusters=2,
        k=9,
        eps=0.05,
    )

labels = ex2.fit_predict(p2)
ex2.plot_diagram()

norm = plt.Normalize(vmin=0, vmax=1)

fig, ax = plt.subplots()

ax.cla()
ax.scatter(*zip(*p2), c=cmap(norm(labels)));
```



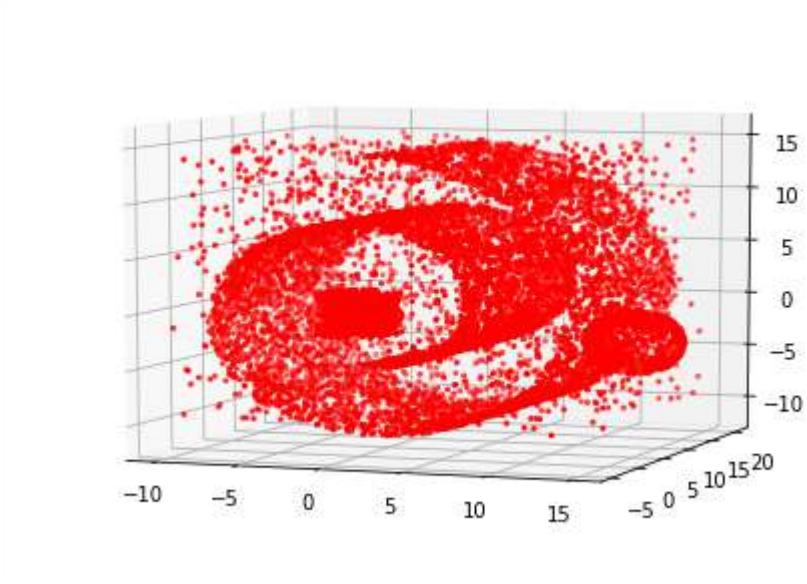


Example 3

We do now a rather spectacular example in 3D just to show the effectiveness of the algorithm to separate clusters with different shapes. We will generate, using points, a cube, a sphere, and a "swiss roll", together with some noise:

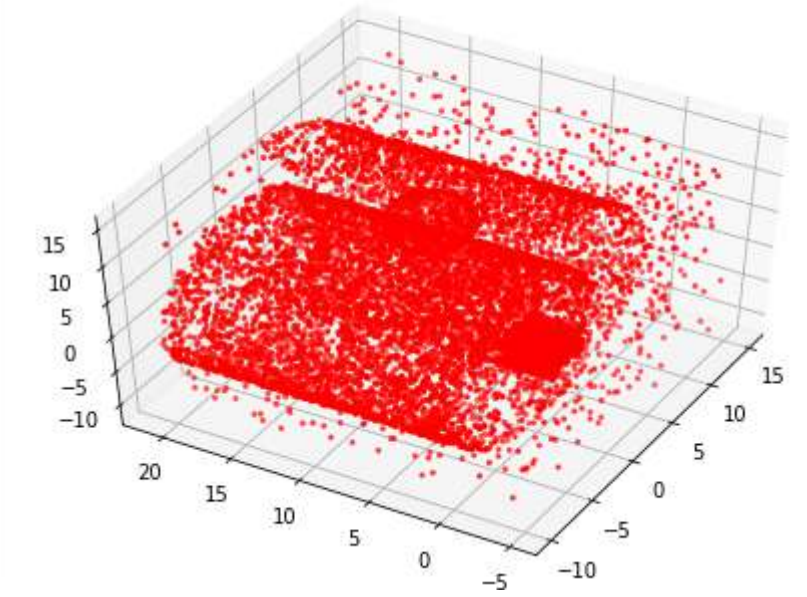
In [157]:

```
import mpl_toolkits.mplot3d.axes3d as plt3
from sklearn.datasets import make_swiss_roll

fig3 = plt.figure()
ax = plt3.Axes3D(fig3)
ax.view_init(7, -70)

points_cube = 1000
points_sphere = 800
#points_line = 700
points_sr = 8000
points_noise = 2000

X1 = np.zeros((points_cube, 3))
for i in range(points_cube):
    X1[i,0], X1[i,1], X1[i,2] = rd.uniform(-2,2), rd.uniform(-2,2), rd.uniform(-2,2)

X2 = np.zeros((points_sphere, 3))
for i in range(points_sphere):
    X2[i,0], X2[i,1], X2[i,2] = rd.uniform(-1,1), rd.uniform(-1,1), rd.uniform(-1,1)
    X2[i,0], X2[i,1], X2[i,2] = 12 + 3*X2[i,0]/np.sqrt(X2[i,0]**2 + X2[i,1]**2 + X2[i
,2]**2), 15 + 3*X2[i,1]/np.sqrt(X2[i,0]**2 + X2[i,1]**2 + X2[i,2]**2), -4 + 3*X2[i,2]
/np.sqrt(X2[i,0]**2 + X2[i,1]**2 + X2[i,2]**2)

"""
X3 = np.zeros((points_line, 3))
for i in range(points_line):
    param = rd.uniform(-15, 15)
    X3[i,0], X3[i,1], X3[i,2] = 2 - param*0.7, 4 + param*0.7, 2 - param*0.6
    X3[:,0] += 0.02*np.random.randn(points_line)
    X3[:,1] += 0.02*np.random.randn(points_line)
    X3[:,2] += 0.02*np.random.randn(points_line)
"""

X4, _ = make_swiss_roll(n_samples=points_sr, noise=.05)

X5 = np.zeros((points_noise, 3))
for i in range(points_noise):
    X5[i,0], X5[i,1], X5[i,2] = rd.uniform(-10,15), rd.uniform(-5,20), rd.uniform(-10
,15)

X = np.concatenate((X1,X2,X4,X5))
X = np.array(X)

ax.scatter(X[:, 0], X[:, 1], X[:, 2], color="red", s=4);
```



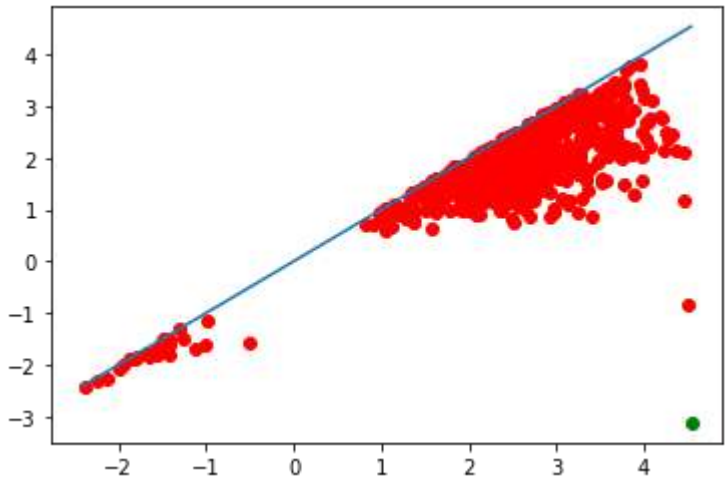
In [158]:
```
ax.view_init(50, -150)
fig3
```

Out[158]:



Let's run the algorithm with k-NN and the logDTM estimation. We also use the parameter n_jobs=-1, which becomes useful to increase the computational power when the size of our dataset becomes large, even though in our case we don't have an specially high number of points:

In [159]:
```
ex3 = Tomato(
        input_type="points",
        metric="euclidean",
        graph_type="knn",
        density_type="logDTM",
        #n_clusters=2,
        #symmetrize_graph= True,
        k=9,
        n_jobs=-1,
    )

ex3.fit(X)
ex3.plot_diagram()
print(ex3.labels_)
```

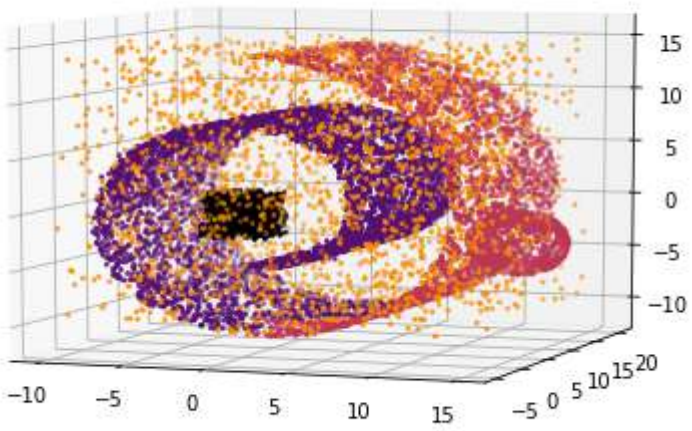



```
[158  88 158 ... 436 382 174]
```

We see 2-3 prominent clusters in the persistance diagram. We can "identify" the noise by checking which points have a low estimate, and creating a new label. We plot the result at the end:

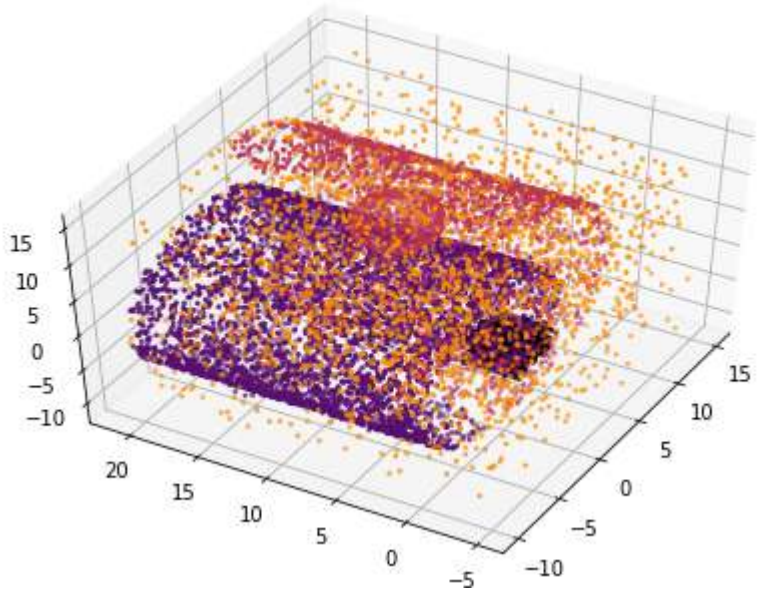
In [180]:
```
ex3.n_clusters_ = 3
label = ex3.labels_

for i in range(len(X)):
    if ex3.weights_[i] < 0.5:
        label[i] = 3

print(label)

fig3 = plt.figure()
ax = plt3.Axes3D(fig3)
ax.view_init(7, -70)

for l in np.unique(label):
    ax.scatter(X[label == l, 0], X[label == l, 1], X[label == l, 2],
               color=plt.cm.inferno(np.float(l) / np.max(label + 1)),
               s=3)
```

```
[0 0 0 ... 3 3 3]
```



In [181]:
```
ax.view_init(50, -150)
fig3
```

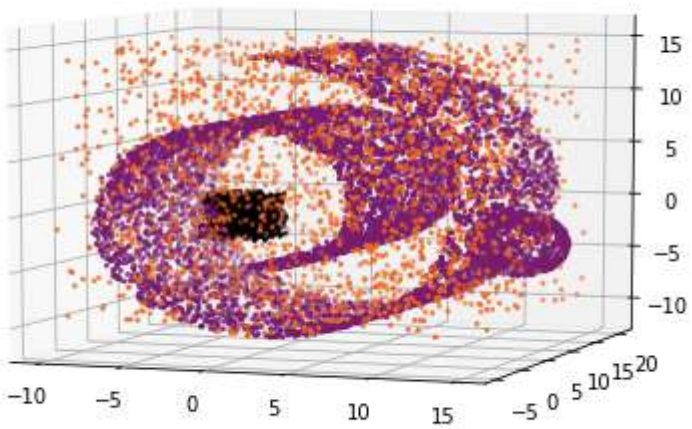
Out[181]:



The swiss roll is not completely clustered and it gets separated into two regions due to the presence of the sphere; we cannot expect our algorithm to distinguish them properly with an intersection so noticeable. The result with two clusters is also quite satisfactory and more realistic, with the whole spiral and the sphere clustered together. We also see that, in both cases, the noise is quite properly identified:

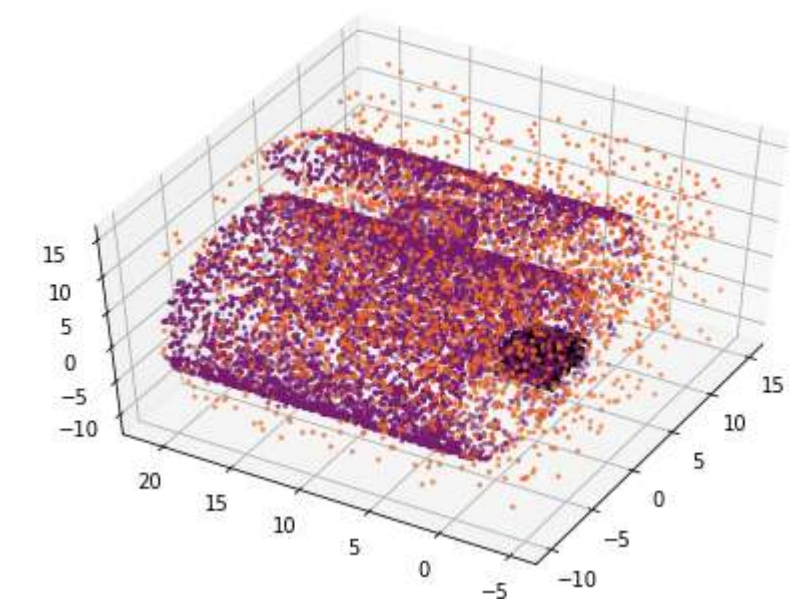
In [182]:
```
ex3.n_clusters_ = 2
label = ex3.labels_

for i in range(len(X)):
    if ex3.weights_[i] < 0.5:
        label[i] = 2

print(label)

fig3 = plt.figure()
ax = plt3.Axes3D(fig3)
ax.view_init(7, -70)

for l in np.unique(label):
    ax.scatter(X[label == l, 0], X[label == l, 1], X[label == l, 2],
               color=plt.cm.inferno(np.float(l) / np.max(label + 1)),
               s=3)
```

```
[0 0 0 ... 2 2 2]
```



In [183]:
```
ax.view_init(50, -150)
fig3
```

Out[183]:

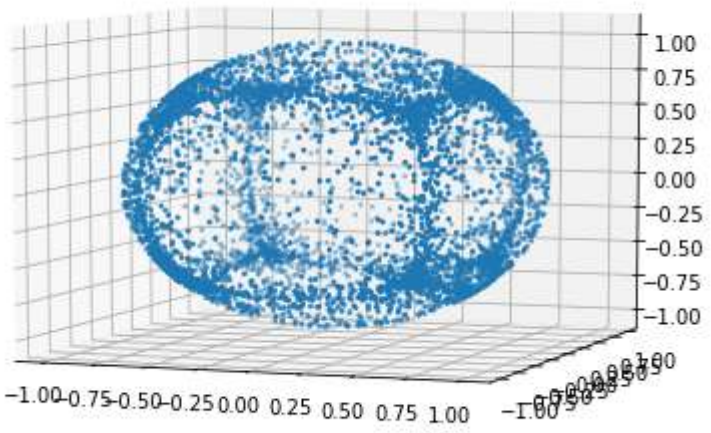


Example 4

In this example we explore the case in which we don't give the coordinates of the points directly, but the distances between them.

To do so, we sample a set of points over the unit sphere, but not uniformly: we sample them first in the cube 1x1x1 using a sigmoid function in each variable to concentrate them near the vertices and edges of the cube, and then we normalize them. This creates naturally regions of the sphere with more points, more specifically the directions pointing towards the vertices and edges of the cube:

In [188]:
```
def sample_spherical(npoints):
    vec = []
    vec.append(-0.5 + 1/(1 + np.exp(-5*np.random.uniform(-1,1, npoints))))
    vec.append(-0.5 + 1/(1 + np.exp(-5*np.random.uniform(-1,1, npoints))))
    vec.append(-0.5 + 1/(1 + np.exp(-5*np.random.uniform(-1,1, npoints))))
    vec /= np.linalg.norm(vec, axis=0)
    return vec

npoints = 6000
points = sample_spherical(npoints)

fig3 = plt.figure()
ax = plt3.Axes3D(fig3)
ax.view_init(7, -70)

ax.scatter(points[0,:], points[1,:], points[2,:], s=3);
```



We compute now the pairwise distances between all the points, using the "spherical" distance d_S: the distance between two points on the surface of a unit sphere with coordinates $a = (a_1, a_2, a_3)$ and $b = (b_1, b_2, b_3)$ is given by the formula:

$$d_S(a, b) = \arccos(a_1 b_1 + a_2 b2 + a_3 b_3)$$

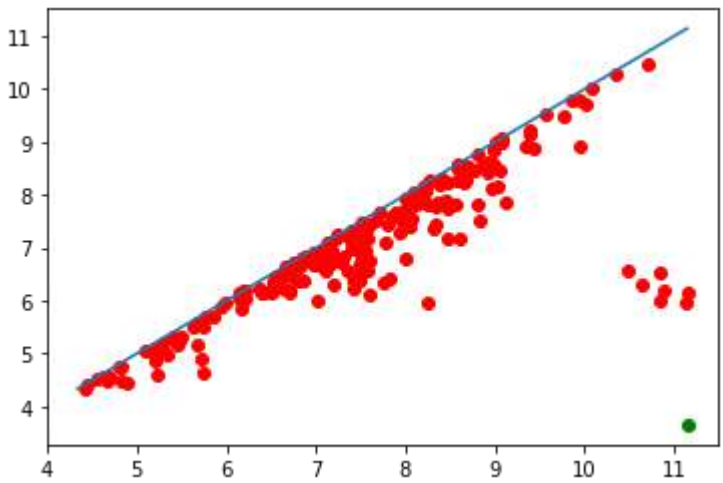
As we don't have many points, we can compute all pairwise distances without much problem:

In [189]:
```
distance_matrix = np.zeros((npoints, npoints))

for i in range(npoints):
    distance_matrix[i,i]= 0
    for j in range(i+1, npoints):
        distance_matrix[i,j] = np.arccos(points[0,i]*points[0,j] + points[1,i]*points
[1,j] + points[2,i]*points[2,j])
        distance_matrix[j,i] = distance_matrix[i,j]

print(distance_matrix)
```

```
[[0.         1.1558255  1.47491536 ... 2.68105662 0.94665096 0.8865952 ]
 [1.1558255  0.         1.83518165 ... 1.77024725 1.71115835 1.14674659]
 [1.47491536 1.83518165 0.         ... 2.12567266 2.05734047 0.7744797 ]
 ...
 [2.68105662 1.77024725 2.12567266 ... 0.         1.88764992 2.57444621]
 [0.94665096 1.71115835 2.05734047 ... 1.88764992 0.         1.80283258]
 [0.8865952  1.14674659 0.7744797  ... 2.57444621 1.80283258 0.        ]]
```

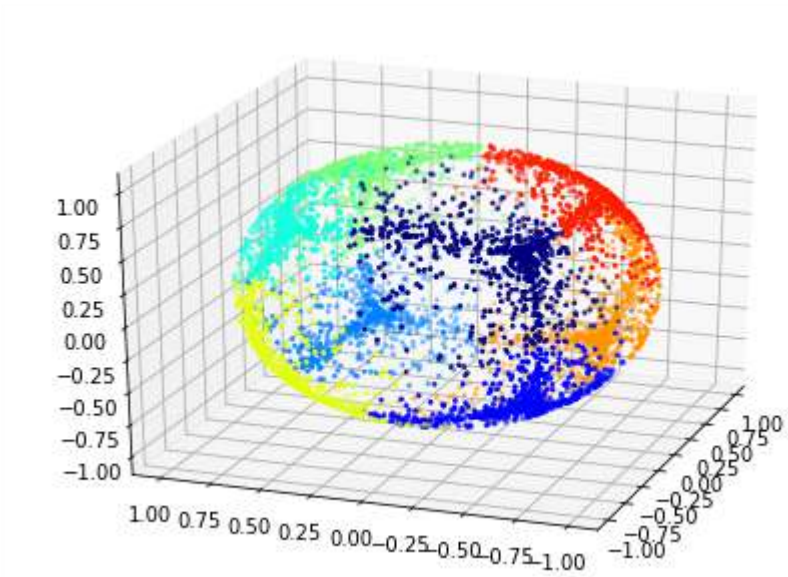
KDE and logKDE use the already-built Scikit-learn library and we cannot use them for a precomputed distance matrix. We use logDTM insted of DTM to make the persistance diagram look more clear:

In [190]:
```
ex4 = Tomato(
        input_type="points",
        metric="precomputed",
        graph_type="knn",
        density_type="logDTM",
        #n_clusters=2,
        k=10,
    )

ex4.fit(distance_matrix)
ex4.plot_diagram()
```



There are 8 clear clusters, a quite expected result:

In [191]:
```
ex4.n_clusters_ = 8
label = ex4.labels_

fig3 = plt.figure()
ax = plt3.Axes3D(fig3)
ax.view_init(25, -160)

for l in np.unique(label):
    ax.scatter(points[0, label == l], points[1, label == l], points[2, label == l],
               color=plt.cm.jet(np.float(l) / np.max(label + 1)),
               s=3)
```



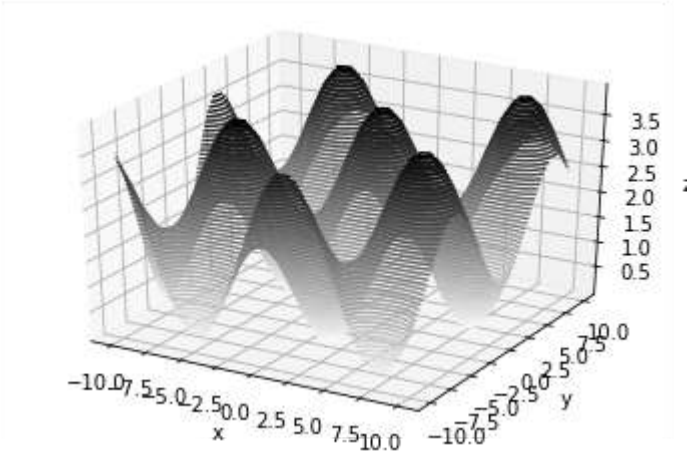
Example 5

We do another easy example just to get used to other input formats to our algorithm. In this one we will input ourselves the weights of the points as well as a neighboring graph, which will just be a rectangular mesh in the square 10x10. For the weights, we will be using the function:

$$f(x,y) = \sin\left(\frac{x+y}{2}\right) + \cos\left(\frac{x-y}{2}\right),$$

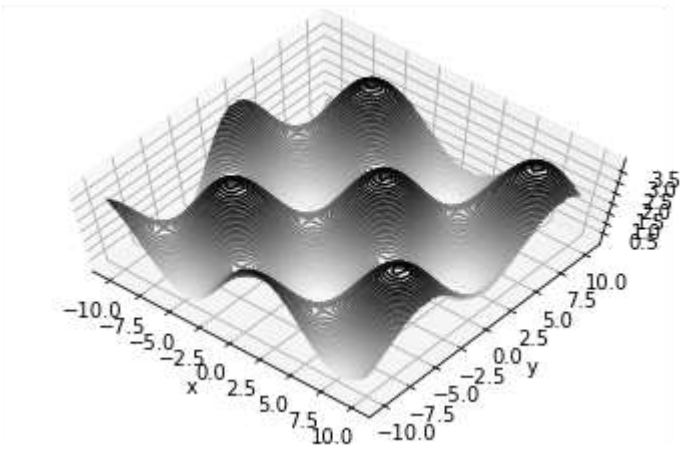
plotted below. In this setting, our algorithm will be just looking for basins of attraction of our function.

In [208]:
```
def f(x, y):
    return 2+ np.sin(0.5*(x+y)) + np.cos(0.5*(x-y))

x = np.linspace(-10, 10, 30)
y = np.linspace(-10, 10, 30)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, Z, 50, cmap='binary')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');
```



In [198]:
```
ax.view_init(70, -50)
fig
```

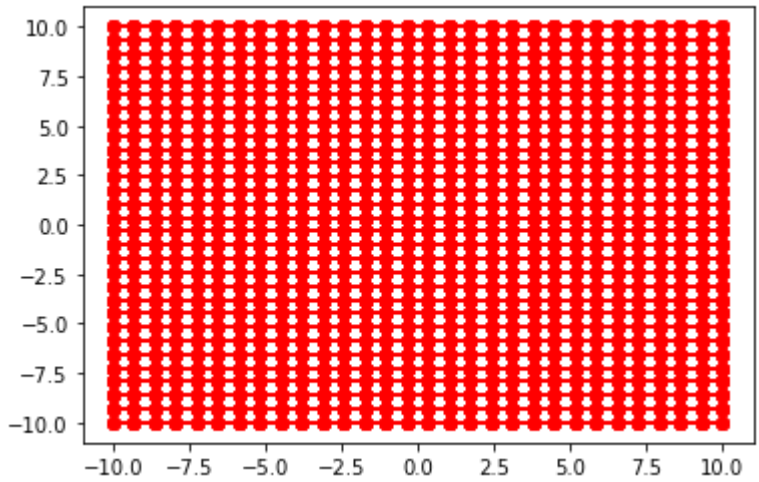
Out[198]:



And now the points, with the neighboring graph:

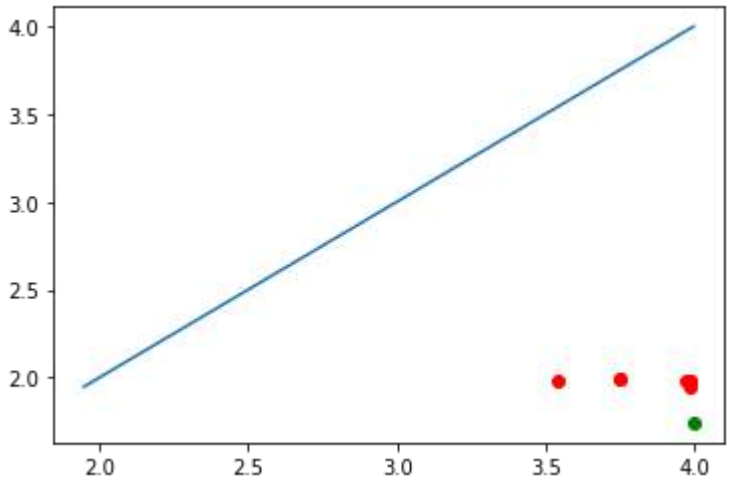
In [200]:
```
size_mesh = 30
points = np.zeros((2, size_mesh**2))
arange = np.linspace(-10., 10., size_mesh)

#Coordinates of the points
for i in range(size_mesh):
    for j in range(size_mesh):
        points[0][i*size_mesh + j] = arange[i]
        points[1][i*size_mesh + j] = arange[j]

#Neighboring graph
neigh_graph = []
for i in range(size_mesh):
    for j in range(size_mesh):
        neigh = []
        if i > 0:
            neigh.append((i-1)*size_mesh + j)
        if i < size_mesh -1:
            neigh.append((i+1)*size_mesh + j)
        if j > 0:
            neigh.append(i*size_mesh + j-1)
        if j < size_mesh -1:
            neigh.append(i*size_mesh + j+1)
        neigh_graph.append(neigh)
```

In [201]:
```
#Drawing the graph
plt.plot(points[0,:], points[1,:], 'o', markersize=2)

for i in range(len(neigh_graph)):
    Y = np.zeros((2,2))
    for j in neigh_graph[i]:
        Y[0][0]= points[0][i]
        Y[1][0]= points[1][i]
        Y[0][1]= points[0][j]
        Y[1][1]= points[1][j]
        plt.plot(Y[0], Y[1], 'ro-', linewidth=2)

plt.show()
```



We now associate the weights to the different points according to f, and run the Tomato algorithm to compute the basins of attraction:

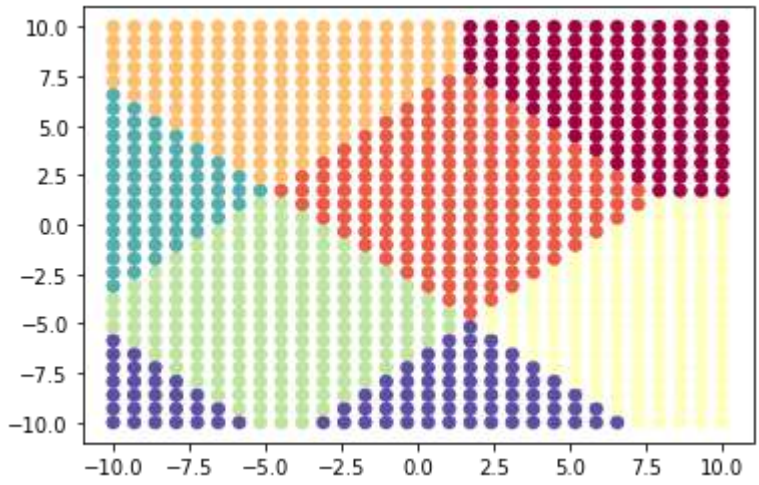
In [202]:
```
#We associate the weights
weights = np.zeros(size_mesh**2)
for i in range(size_mesh**2):
    weights[i] = f(points[0][i], points[1][i])

#We run Tomato
ex5 = Tomato(
        graph_type = "manual",
        density_type = "manual"
    )

ex5.fit(neigh_graph, weights= weights)
ex5.plot_diagram()
print(ex5.diagram_)
```



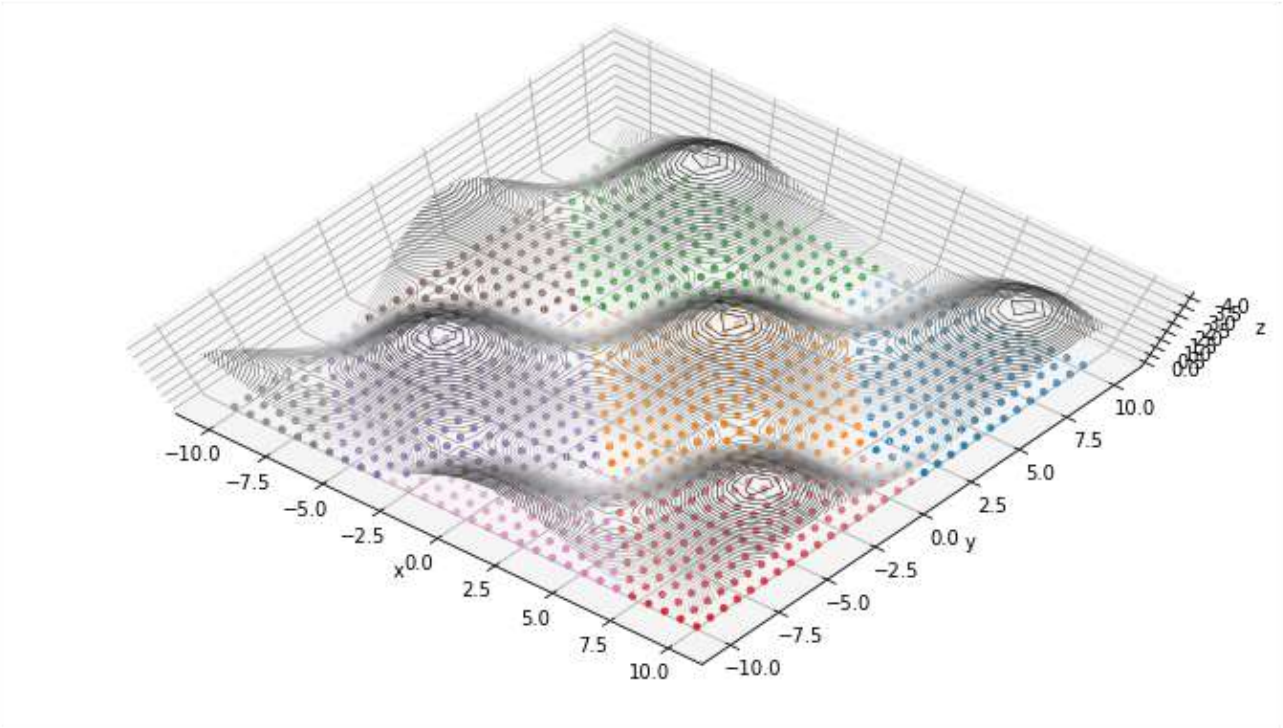

```
[[3.75212014 1.99409954]
 [3.75212014 1.99409954]
 [3.98535762 1.98244992]
 [3.98535762 1.98244992]
 [3.9737506  1.9824371 ]
 [3.54402111 1.97677169]
 [3.9882662  1.94776161]]
```

In [206]:

```
ex5.n_clusters = 7
labels = ex5.fit_predict(neigh_graph, weights= weights)

norm = plt.Normalize(vmin=0, vmax=6)

fig, ax = plt.subplots();

ax.cla()
ax.scatter(points[0,:], points[1,:], c=cmap(norm(labels)));
```



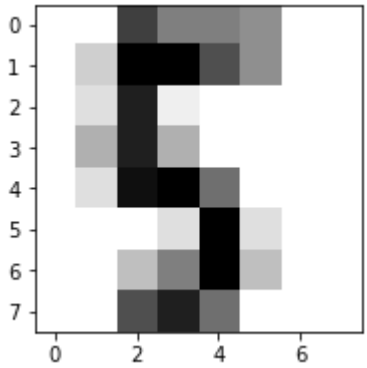
In [209]:
```
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, Z, 50, cmap='binary', linewidths=0.5);
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');

points3d = np.zeros((3, size_mesh**2))
for i in range(size_mesh**2):
    points3d[0, i] = points[0,i]
    points3d[1, i] = points[1,i]

z_coord = np.zeros(size_mesh**2)
for l in np.unique(labels):
    ax.scatter(points3d[0, labels == l], points3d[1, labels == l], points3d[2, labels
== l], s=10)

ax.view_init(80, -50)
fig = plt.gcf()
fig.set_size_inches(12,7)
plt.show()
```



Example 6

In this last example, closer to the kind of datasets we could find in real life, we will work with the famous "Digits dataset", containing 1797 observations each with 64 features: each entry represents a (highly compressed) hand-written digit in a 8x8 grid, where each cell can vary from 0 to 16, representing its opacity. Naturally, the dataset also contains the correct labels of each instance: a number from 0 to 9, the one written in the grid.

In [987]:
```
#Load the digits dataset
digits = datasets.load_digits()

#Display the 25th digit
plt.figure(1, figsize=(3, 3))
plt.imshow(digits.images[25], cmap=plt.cm.gray_r, interpolation='nearest')
plt.show()
```



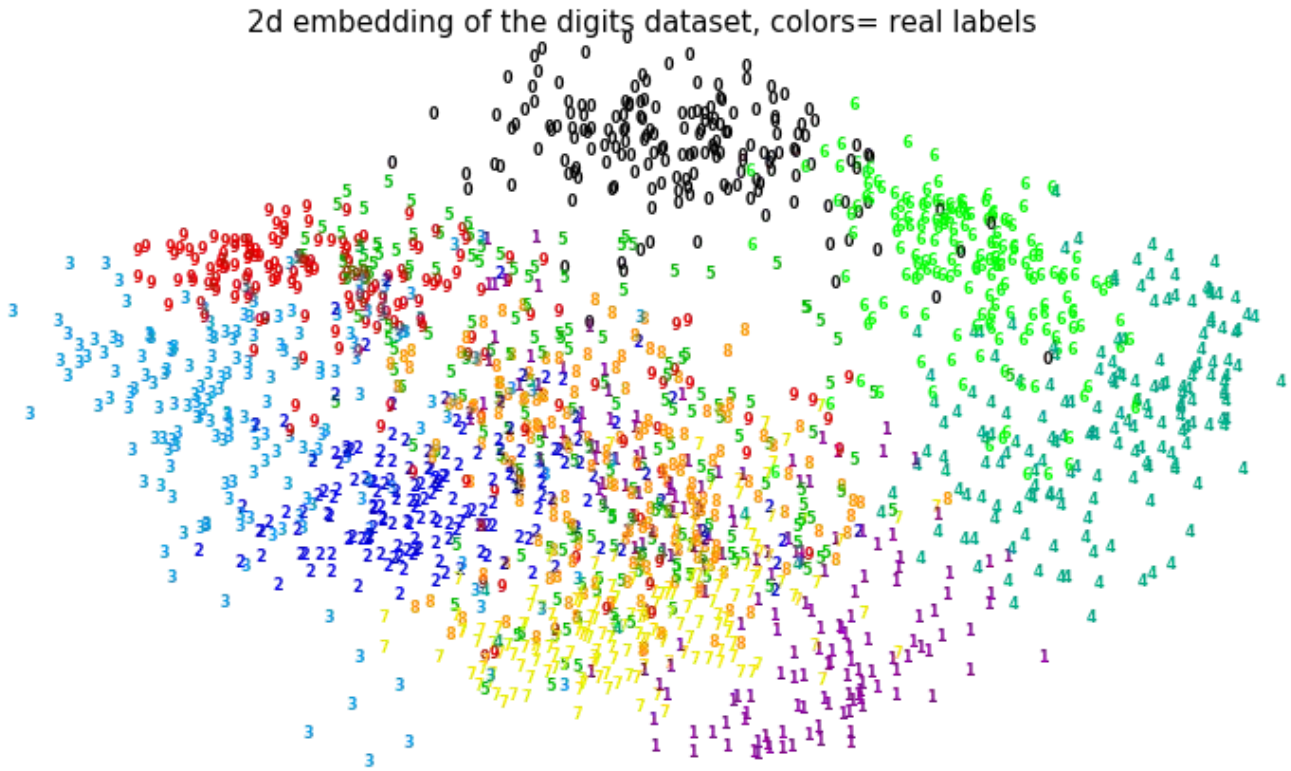
It is well known the difficulty of performing data science algorithms in high dimension, and clustering is not an exception; in fact, it is a process particularly sensitive to numerical data being sparse. Thus, even with dimensionality reduction techniques, it's not a good idea to expect a brilliant performance of our algorithm in this setting. In any case, it is interesting to see what kind of results we get. The results of other clustering methods over this dataset can be found in [4].

In [1056]:
```
digits, real_label = datasets.load_digits(return_X_y=True)

print(digits)
print(real_label)
```

```
[[ 0.  0.  5. ...  0.  0.  0.]
 [ 0.  0.  0. ... 10.  0.  0.]
 [ 0.  0.  0. ... 16.  9.  0.]
 ...
 [ 0.  0.  1. ...  6.  0.  0.]
 [ 0.  0.  2. ... 12.  0.  0.]
 [ 0.  0. 10. ... 12.  1.  0.]]
[0 1 2 ... 8 9 8]
```

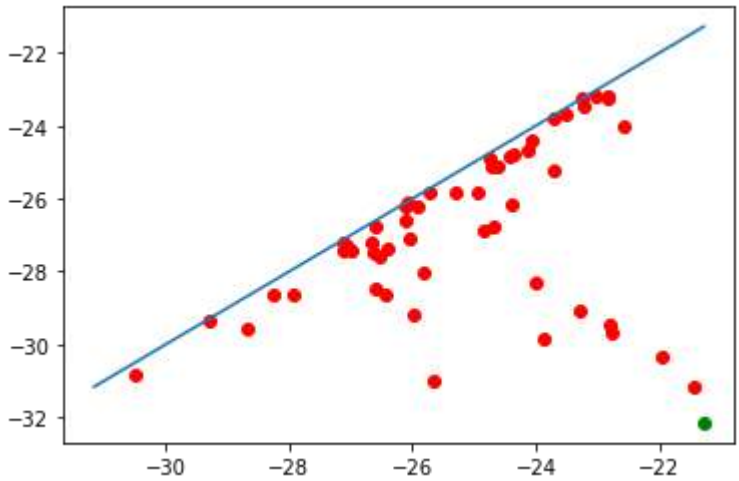
We can embed the dataset in the plane by using PCA dimensionality reduction. We observe that, with that reduction level, the different clusters of numbers are somewhat distinguishable, but there is also considerable overlapping:

In [1009]:
```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
digits_red = pca.fit_transform(digits)

def plot_clustering(X_red, labels, title=None):
    x_min, x_max = np.min(X_red, axis=0), np.max(X_red, axis=0)
    X_red = (X_red - x_min) / (x_max - x_min)

    plt.figure(figsize=(6, 4))
    for i in range(X_red.shape[0]):
        plt.text(X_red[i, 0], X_red[i, 1], str(y[i]),
                 color=plt.cm.nipy_spectral(labels[i] / 10.),
                 fontdict={'weight': 'bold', 'size': 8})

    plt.xticks([])
    plt.yticks([])
    if title is not None:
        plt.title(title, size=15)
    plt.axis('off')
    fig = plt.gcf()
    fig.set_size_inches(12,7)

plot_clustering(digits_red, real_label, title = "2d embedding of the digits dataset,
 colors= real labels")
```



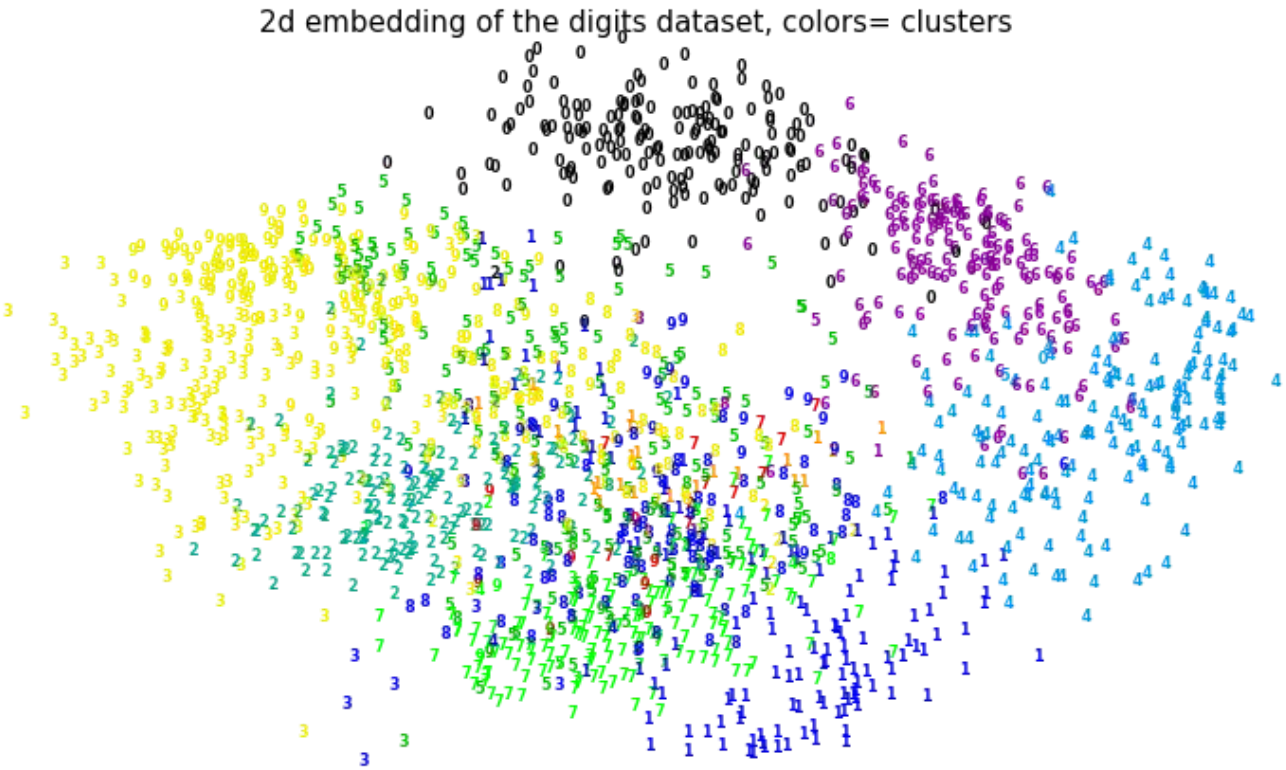
It's useless to try to run the algorithm without doing any kind of dimensionality reduction first: accurate density estimation is almost always unsucessful with highly sparse data. We can try to use our algorithm after killing some dimensions first. Wit our dataset, after some experimentation, when there are 11 dimensions left DTM density estimation looks quite well:

In [1110]:
```
pca = PCA(n_components=11)
digits_red = pca.fit_transform(digits)

ex6 = Tomato(
        input_type="points",
        metric="euclidean",
        graph_type="knn",
        density_type="logDTM",
        n_clusters=10,
        k=9,
    )

ex6.fit(digits_red)
ex6.plot_diagram()
```



It looks like the algorithm found "naturally" 9-10 clusters, let's plot these 10 groups in 2D:

In [1100]:
```
labels = ex6.fit_predict(digits_red)

pca = PCA(n_components=2)
digits_red = pca.fit_transform(digits_red)

plot_clustering(digits_red, labels, title = "2d embedding of the digits dataset, colo
rs= clusters")
```

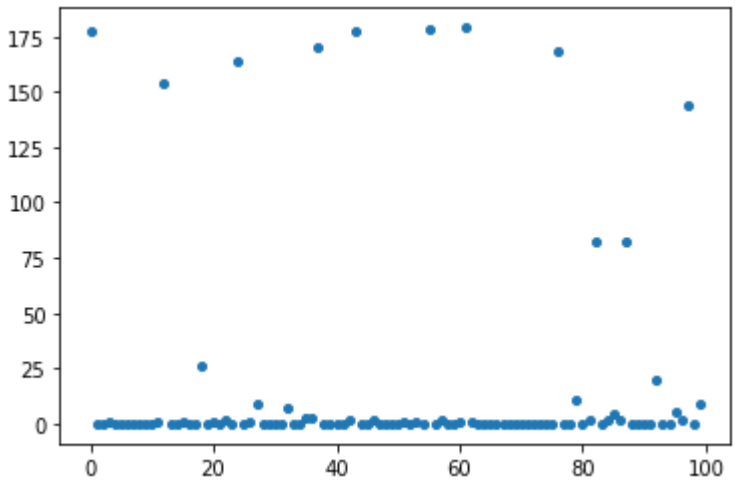


The result looks suprisingly good, actually.

A way to measure the matching degree consists in computing the vector $10 \cdot \text{real\_labels} + \text{clustering\_labels}$, which takes values $\in \{0, \ldots, 99\}$, and then counting the number of times each number appears. In a perfect classification, only 10 values would appear, more specifically $10 \cdot i + label_i$, with $i \in \{0, \ldots, 10\}$; in a decent clustering, we should at least see some clearly more prominent values, which is indeed what happens in our case!

In [1111]:
```
vect_count = 10*real_label + labels
print(vect_count)

count = np.zeros((2,100))
for i in range(100):
    count[0,i]= i
for i in vect_count:
    count[1,i] += 1

fig, ax = plt.subplots();
ax.cla()
ax.scatter(count[0,:], count[1,:], s=16);
```

```
[ 0 12 27 ... 82 97 87]
```



From this graph it looks like most of the values have been properly grouped. Only the value 8 looks more mismatched. We can also get an idea about how the numbers have been grouped or labeled using a table, and checking the columns and rows: each of them should only contain one "big" value:

In [1102]:
```
import pandas as pd
data = [[1, 2], [3, 4]]
pd.DataFrame(data, columns=["Foo", "Bar"])

table = []
for i in range(10):
    row = []
    #row.append(i)
    for i in range(i*10, i*10 +10):
        row.append(count[1, i])
    table.append(row)

pd.DataFrame(table, columns=[ 'Label 0', 'Label 1', 'Label 2', 'Label 3', 'Label 4',
'Label 5', 'Label 6', 'Label 7', 'Label 8', 'Label 9'])
```

Out[1102]:

	Label 0	Label 1	Label 2	Label 3	Label 4	Label 5	Label 6	Label 7	Label 8	Label 9
0	177.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	1.0	154.0	0.0	0.0	1.0	0.0	0.0	26.0	0.0
2	1.0	0.0	2.0	0.0	164.0	0.0	1.0	9.0	0.0	0.0
3	0.0	0.0	7.0	0.0	0.0	3.0	3.0	170.0	0.0	0.0
4	0.0	0.0	2.0	177.0	0.0	0.0	2.0	0.0	0.0	0.0
5	0.0	1.0	0.0	1.0	0.0	178.0	0.0	2.0	0.0	0.0
6	1.0	179.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
7	0.0	0.0	0.0	0.0	0.0	0.0	168.0	0.0	0.0	11.0
8	0.0	2.0	82.0	0.0	2.0	4.0	2.0	82.0	0.0	0.0
9	0.0	0.0	20.0	0.0	0.0	5.0	2.0	144.0	0.0	9.0

The clustering has been quite successful. In any case, the numbers 3's and 9's have been almost completely clustered together (which is not that suprising, given the low resolution of the dataset), and the number 8 is almost evenly divided between labels 2 (with the 1's) and 7 (with the 3's and 9's), again not very surprising.

References

[1]. Frédéric CHAZAL et al., *Persistence-Based Clustering in Riemannian Manifolds* . In :Journal of the ACM 60 (juin 2011).doi:10.1145/1998196.1998212 : URL: https://hal.inria.fr/inria-00389390/document (https://hal.inria.fr/inria-00389390/document)

[2]. Tutorial for DTM: https://github.com/GUDHI/TDA-tutorial/blob/master/Tuto-GUDHI-DTM-filtrations.ipynb (https://github.com/GUDHI/TDA-tutorial/blob/master/Tuto-GUDHI-DTM-filtrations.ipynb)

[3]. Gérard BIAU, Frédéric CHAZAL, David COHEN-STEINER, Luc DEVROYE, Carlos RODRIGUEZ. *A Weighted k-Nearest Neighbor Density Estimate for Geometric Inference* . 2011. ffinria-00560623v1 : URL: http://luc.devroye.org/BiauChazalCohenDevroyeRodriguez-kNN-2EJS-2011.pdf (http://luc.devroye.org/BiauChazalCohenDevroyeRodriguez-kNN-2EJS-2011.pdf)

[4] Various Agglomerative Clustering on a 2D embedding of digits: https://scikit-learn.org/stable/auto_examples/cluster/plot_digits_linkage.html#sphx-glr-auto-examples-cluster-plot-digits-linkage-py (https://scikit-learn.org/stable/auto_examples/cluster/plot_digits_linkage.html#sphx-glr-auto-examples-cluster-plot-digits-linkage-py)

/