IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Gabriel GROS

Jack WHITE

v.2.5

CODEV Recherche

Classification de données pour de l'apprentissage avec peu d'exemples grâce au Traitement du Signal sur Graphes

Sous la supervision de

Bastien Pasdeloup et Mounia Hamidouche

Ce rapport contient 18 pages et a été rédigé par Gabriel Gros et Jack White.

Le *Machine Learning* a connu de nombreux changements et glissements ces dernières années. Si dans les imaginaires collectifs l'exemple le plus connu reste celui de l'algorithme en capacité de reconnaître des chiffres manuscrits, les intelligences artificielles modernes sont désormais aptes, pour ne citer que ces exemples, à détecter des objets ou bien à classer des images. C'est pour cette raison que le *Machine Learning* est aujourd'hui massivement utilisé dans le domaine de la *Data Science* et de l'analyse de données pour développer, tester et appliquer des algorithmes d'analyses prédictives. Aussi bien, la recherche de nouvelles techniques d'optimisation de ces algorithmes est-elle un enjeu contemporain majeur auquel essaie de répondre une communauté scientifique en développement.

Le *Deep Learning*, en particulier, s'est largement répandu ces dernières années en raison de son efficacité comparée aux autres familles de *Machine Learning*. Il se définit comme la concaténation de deux ensembles qui peuvent, chacun, être améliorés : un extracteur et un classifieur. Le rôle de l'extracteur est de dessiner un réseau de neurones capable de transformer les données en entrée du modèle si elles ne sont pas exploitables, quand celui du classifieur consiste à trier et classer les données de manière efficace.

I. Contextualisation

Pour ce projet, la tâche nous revient de traiter un problème dit de classification ou de *clustering* parce qu'il s'agit de déterminer la classe d'un objet en entrée du classifieur. C'est pourquoi notre étude consiste à améliorer le classifieur, et non l'extracteur **[1]**.

Par ailleurs, afin d'assurer leur bon fonctionnement, les algorithmes de *Deep Learning* ont besoin d'être entraînés pour fonctionner de manière satisfaisante. Il s'agit d'un apprentissage dit « supervisé », ce qui revient à intégrer uniquement des données labellisées dans l'algorithme. En réalité, l'ordinateur connait la classe de chacun des attributs en entrée et se charge de comparer avec la classe attribuée en sortie de l'algorithme pour mesurer l'efficacité de ce dernier. Cependant, lorsque les jeux de données deviennent très importants – soit des millions de données différentes destinées à entraîner l'algorithme – ou lorsqu'on ne dispose pas de suffisamment de données, cette opération s'avère parfois très fastidieuse. C'est la raison pour laquelle l'apprentissage dit « semi-supervisé », lequel utilise à la fois des données étiquetées et des données non étiquetées – respectivement appelés *shots* et *queries* -, peut être particulièrement intéressant. En effet, contrairement à l'apprentissage supervisé, le semi-supervisé requiert une quantité de données moindre et est en conséquence beaucoup moins coûteux. Plus précisément, il nous revient de répondre au problème dit du *Few shot* qui consiste à classer des données avec très peu d'exemples.

En un mot, et pour problématiser en synthèse, l'objet de cette étude est de développer un classifieur de données dans le cadre de ce problème du *Few Shot*.

Pour ce faire, nous allons faire appel au Traitement du Signal sur Graphes (TSG). En effet, passer par des graphes permet de mettre en valeur les relations entre les entités, ce qui est au cœur du projet de classification des données. En particulier, dans le cadre du *Few shot,* nous disposons de peu de données et par conséquent, le classifieur sera particulièrement sensible à chacune d'entre elles. C'est pour cette raison que « lisser » les données les rendraient ainsi plus cohérentes aux yeux du classifieur. Dans un premier mouvement, on pourrait penser qu'il suffit de filtrer les données en entrée de l'extracteur, mais cette possibilité est vite écartée car la tâche est complexe dans le domaine de représentation initial des données : cela serait susceptible de limiter les performances du programme. En revanche, filtrer les données en sortie de l'extracteur s'avère utile pour le « débruitage » car l'algorithme pourra considérer des données plus abstraites. Dès lors, la force du TSG est évidente : elle a le pouvoir de représenter, d'une

part, les données labellisées de chaque classe par les sommets d'un graphe et d'autre part, la similarité entre ces données par les poids dudit graphe. Il suffit enfin de filtrer chaque graphe.

Pour finir, notre projet a été encadré par Bastien Pasdeloup et Mounia Hamidouche, tous les deux chercheurs à l'IMT Atlantique sur le campus de Brest. Ils nous ont suggéré d'implémenter, en plus du TSG décrit dans leur article *Improving Classification Accuracy with Graph Filtering* **[2]**, un algorithme dit de *Mean Shift*. Ce dernier consiste à déplacer le centroïde de chacun des *clusters* de manière itérative vers une région de densité élevée jusqu'à obtenir une convergence. Dans le cas de notre étude, cela revient à évaluer un centroïde pour chaque classe – soit la « moyenne » des données labellisées de chacune des classes – puis à le déplacer en fonction des données non labellisées.

Notre conclusion se propose de questionner la compatibilité du TSG et du *Mean Shift,* cela en comparant nos résultats avec ceux obtenus par nos tuteurs.

II. Conception

Pour le projet, nous disposons d'un jeu de 20 classes différentes contenant chacune 600 données. Afin de nous placer dans le cadre du problème du *Few shot*, sur les conseils de nos tuteurs, nous considérerons 5 classes choisies de manière aléatoire contenant chacune 5 données labellisées et 15 données non labellisées, en sorte que nous disposons de 100 données qui seront tirées aléatoirement. En outre, ces tirages seront réitérés à chaque fois que l'algorithme sera lancé.

Ainsi, l'objectif premier est de construire un centroïde pour chaque classe grâce aux données labellisées. Nous disposerons de cette façon d'un jeu de données appartenant de manière certaine à un *cluster* et les données non étiquetées serviront de mesure pour l'efficacité du programme. En effet, puisque l'algorithme devra affecter chaque donnée non labellisée à une classe, il suffira ensuite de vérifier s'il l'a affectée à la classe dans laquelle cette donnée avait été tirée aléatoirement. En l'occurrence, il est presque impossible d'arriver à une efficacité de 100%, mais nous misons sur une efficacité de 90%, dans le meilleur des cas.

A. Principes du Traitement du Signal sur Graphes

Comme cela a été évoqué dans l'introduction, l'étude des données labellisées doit se faire après filtrage de ces dernières. Pour toute cette partie, nous avons principalement été guidés par **[2]** et **[3]**. Pour ce faire, nous devons étudier le formalisme du TSG. Prenons un graphe $G = \{V, E\}$ avec V l'ensemble des sommets et E l'ensemble des arrêtes. Mathématiquement, un graphe est représenté par sa matrice d'adjacence W avec :

$$W[i,j] = \begin{cases} wij, & si\ l'arrête(i,j) \in E \\ 0, & sinon \end{cases}$$

Avec : wij la valeur de l'arrête reliant les sommets i et j tel que $wij \in \mathbb{R}_+^*$

À partir de la matrice d'adjacence, il est possible d'établir la matrice de degré du graphe définie comme tel :

$$D = diag\left(\left[\sum_{j=1}^{|V|} W[i,j]\right] avec\ i \in \{1, \dots, |V|\}\right) \quad (1)$$

On peut alors définir le Laplacien du graphe G : $L = D - W$ (2)

Si tous les sommets du graphe G sont reliés, il est alors possible d'écrire le Laplacien normalisé de G :

$$Ł = I - D^{-1/2} W D^{-1/2} \quad (3)$$

Avec : I la matrice identité

La matrice W étant symétrique à valeurs réelles et les matrices I et D étant diagonales, on en déduit que Ł est elle-même symétrique à valeurs réelles. Ainsi, d'après le théorème spectral, elle peut être diagonalisée au moyen d'une matrice de passage orthogonale que l'on notera U. Cette dernière est composée des vecteurs propres de W et possède $|V|$ valeurs propres non nulles notées telles que :

$$\lambda_1 \leq \cdots \leq \lambda_{|V|}$$

En notant Λ l'ensemble de ces valeurs propres, dès lors, on a :

$$Ł = U diag(\Lambda) U^T \quad (4)$$

On a diagonalisé Ł . On cherche désormais à filtrer les données étiquetées. Pour ce faire, il faut considérer chaque donnée labellisée comme un vecteur $x \epsilon \mathbb{R}^{|V|}$. L'opération $\hat{x} = U^T x$ est appelée *Graph Fourier Transform* (GFT). Son inverse est donné par $x = U\hat{x}$ (iGFT).

Par analogie avec la transformée de Fourier classique qui consiste à passer un signal dans l'espace des fréquences, on peut considérer les valeurs propres du Laplacien comme des fréquences de ce dernier. On pourrait ainsi filtrer les « hautes fréquences » du Laplacien, ce qui reviendrait à réduire la variance des données, comme le montre le papier *Improving Classification Accuracy with Graph Filtering* **[3]** livré par nos tuteurs (cf. partie 3 : *Effect of low-pass graph filters on centroids*). En effet, retirer les hautes fréquences équivaut à supprimer le « bruit » et de fait, à conserver uniquement les fréquences d'intérêt du signal étudié. Nous devons par conséquent appliquer un filtre passe-bas sur le Laplacien. Pour ce faire, on note le filtre $h : \lambda \rightarrow h(\lambda)$ ce qui nous permet d'écrire la matrice de filtrage $H = diag(h(\lambda))$. Dès lors, un graphe dans le domaine de Fourier est donné par :

$$\hat{x} = U^T x \quad (5)$$

Par analogie avec la théorie de Fourier, on peut donc filtrer ce graphe grâce à la matrice H :

$$H\hat{x} = HU^T x \quad (6)$$

Il suffit enfin de multiplier (6) par U pour revenir dans le domaine des graphes d'après l'IGFT et ainsi obtenir x^{filter} :

$$x^{filter} = UHU^T x \quad (7)$$

Il est désormais nécessaire de transposer nos données labellisées en un graphe afin d'appliquer ce filtrage. Pour cela, on récupère les *queries* $F = \phi(X)$ en sortie de l'extracteur, où X désigne les données en entrée de l'extracteur. On calcule ensuite la similarité entre chaque donnée labellisée d'une classe donnée puis on stocke chaque similarité dans une matrice. On a donc :

$$S_{i,j} = \begin{cases} s(F_{i,:}, F_{j,:}), & si\ i \neq j \\ 0, & sinon \end{cases}$$

Avec : $s : (x, y) \rightarrow s(x, y)$ la similarité entre x et y – typiquement une exponentielle décroissante ou un cosinus, tous deux fonctions de la norme entre les deux vecteurs.

On pourrait alors considérer la matrice de similarité comme la matrice d'adjacence du graphe d'une classe. Cependant, nous aurions un graphe complet et nous conserverions des similarités bien moins importantes entre certaines données étiquetées.

Aussi est-il pertinent de sélectionner seulement une partie des facteurs dans la matrice de similarité. On obtiendra de la sorte un graphe non nécessairement complet qui conservera les similarités les plus importantes. Ces dernières seront les données les plus « représentatives » de chaque classe.

Par conséquent, on décide de conserver les k plus grand facteurs de chaque ligne de la matrice de similarité et de nullifier les autres valeurs. Cette matrice sera considérée comme la matrice d'adjacence d'un graphe correspondant à une classe donnée. Il suffira ensuite d'appliquer le filtre comme en (7) :

$$F^{filter} = UHU^T F \quad (8)$$

En l'occurrence, comme nous le montrerons au cours de ce rapport, nous avons pensé à plusieurs types de filtrages qui différencieraient en fonction des données auxquels ils s'appliqueraient. Cela permettrait ainsi de ne pas se limiter au traitement classe par classe des données labellisées.

Il revient maintenant d'expliquer le processus de l'algorithme du *Mean Shift* qui est essentiel dans la classification des données non étiquetées.

B. Algorithme du *Mean Shift*

Les données labellisées ayant été traitées, il reste désormais à classer les données non labellisées grâce à l'algorithme du *Mean Shift*. D'après le processus décrit dans l'introduction, ce dernier doit affecter temporairement les *queries* à une classe et permettre de réévaluer le centroïde de cette classe. En effet, il est nécessaire que l'algorithme puisse discuter de l'affectation d'un *query* à une classe plutôt qu'à une autre. C'est pour cette raison qu'une donnée non étiquetée ne sera ajoutée de manière définitive à une classe – devenant ainsi étiquetée – que si elle est apparue plusieurs fois comme étant la plus proche d'un des centroïdes. Sans cette condition, elle ne se verra donc affectée que temporairement à la classe correspondante.

Cette partie de l'algorithme est fondamentale car elle différencie le *Mean Shift* de son homologue, l'algorithme des *K-means*, qui se satisfait d'ajouter des *queries* aux *clusters* les plus proches. Le *Mean Shift* possède de ce fait un traitement dynamique dans la classification des données non labellisées. En revanche, ce n'est pas le cas pour les centroïdes : le calcul de ces derniers ne sera pas remis en cause d'une itération à une autre.

Par ailleurs, il est essentiel que l'algorithme tienne compte de tous les points qui appartiennent déjà à une classe afin de réaliser une moyenne pondérée lors de la réévaluation des centroïdes et non pas une moyenne simple entre le centroïde et le *query* ajouté temporairement.

Enfin, nous proposons de réaliser un filtrage des données lorsqu'une donnée non étiquetée deviendra étiquetée. Ce filtrage « dynamique » pourrait permettre de lisser les données de manière continue et ainsi améliorer l'efficacité globale de l'algorithme. En l'occurrence, le programme permettra de lancer ou non ce filtrage dynamique.

Maintenant les fondations posées, venons-en à notre programme, point cardinal du projet.

III. Réalisation

A. Extraction et tirage des données

Nous devons commencer par extraire les données. Nos tuteurs avaient mis à notre disposition un jeu de données que nous pourrions utiliser pour nos tests. Ainsi, chaque vecteur des 20 classes dont nous disposons représente une donnée sortie de l'extracteur : c'est pour cette raison que les coordonnées de ces vecteurs n'ont pas de réalité physique.

Ensuite, il faut tirer de manière aléatoire 5 classes pour ensuite piocher aléatoirement 20 données par classe : les 5 premières seront labellisées et les 15 autres, non labellisées. Procéder de cette façon évite le croisement des données tout en conservant le caractère aléatoire du tirage au sort.

```
#tirage au sort des classes et on évite les doublons
random_classes = rd.sample(range(80,99),nb_classes)

for rand in random_classes:
    #tirage au sort des données dans chaque classe
    data_class = rd.sample(data[rand], nb_labelled + nb_non_labelled)

    #affectation des DL et des DNL
    classes.append(data_class[0 : nb_labelled])
    non_labelled_data += data_class[nb_labelled : nb_labelled + nb_non_labelled]
```

B. Codage du TSG

Nous pouvons ensuite en venir au développement sur le TSG. Tout d'abord, nous devons établir la matrice de similarité. Pour ce faire, on crée une matrice carrée – comme toutes les autres matrices que nous serons amenés à traiter- nulle de dimension égale au nombre de *queries* dont on doit évaluer la similarité. En l'occurrence, pour l'initialisation, étant donné que nous disposons de 5 données labellisées par classe, notre matrice de similarité sera de dimension 5x5. Il convient ensuite de parcourir cette matrice pour affecter la similarité entre une donnée *i* et une donnée *j*. Enfin, il est nécessaire de diviser par la norme L1 pour ainsi travailler avec des vecteurs de probabilités. Cela revient à diviser par la somme des coordonnées de la matrice.

```
def s(x,y):
    return np.exp(-np.linalg.norm(x - y)**2/10)

def similarity(F):
    n = len(F)
    S = np.zeros((n,n))

    for i in range(n):
        for j in range(n):
            S[i,j] = s(F[i],F[j]) #calcul de la similarité entre chaque feature

    somme = np.sum(S)
    S /= somme #division par la norme L1

    return S
```

On remarquera que la similarité est une exponentielle décroissante de la norme écartant deux données labellisées, à un facteur près. Nous avions pensé mettre cette distance au carré mais les résultats étaient toutefois moins concluants.

Une fois la matrice de similarité obtenue, nous devons extraire les k plus grands facteurs de chaque ligne. On intègre donc dans la future matrice d'adjacence les k premiers éléments de chaque ligne qui sont sélectionnés en triant chaque ligne de la matrice de similarité. En

l'occurrence, cette affectation se fait de manière symétrique puisque toute matrice d'adjacence est symétrique.

```
def W_new(S,k=3):
    n = np.shape(S)[0]
    W = np.zeros((n,n))

    #On sélectionne les k plus grands facteurs d'une ligne
    for i in range(n):
        L_sort = sorted(list(S[i,:]))[::-1] #liste décroissante des facteurs d'une ligne
        L_new = []

        #on prend les k plus grand facteurs
        for m in range(k):
            L_new.append(L_sort[m])

        for j in range(n):
            if S[i,j] in L_new:
                W[i,j],W[j,i] = S[i,j],S[j,i] #par symétrie
    return W #W est bien symétrique
```

La matrice d'adjacence une fois créée, il suffit de filtrer le graphe correspondant. La première étape est d'obtenir le Laplacien normalisé Ł. Ce dernier se calcule, d'après (3), à-partir de trois matrices : la matrice d'adjacence W, la matrice identité I et la matrice de degré D à la puissance $-1/2$. Puisque nous disposons déjà de la matrice d'adjacence, il faut calculer I et D. En particulier, la seconde est définie en (1). Elle correspond à la somme des poids des arrêtes voisines de tous les sommets du graphe. D'un point de vue matriciel, cela revient à sommer les coordonnées de chaque ligne et à les placer sur la diagonale de la matrice.

Une fois ces deux matrices obtenues, il est nécessaire de diagonaliser Ł. Cette opération peut se faire, une nouvelle fois, grâce au module *numpy* qui renseigne : un vecteur contenant les valeurs propres Λ et la matrice de passage orthogonale U permettant le passage dans le domaine des « fréquences ». Une fois que l'on a obtenu la matrice de passage orthogonale contenant les vecteurs propres (cf. I – Conception. Théorème spectral), il est nécessaire de transposer cette matrice afin de faire apparaître la diagonalisation. Dès lors, on vérifie précisément l'équation (4).

Pour finir, le Laplacien étant dans une base adaptée aux valeurs propres – soit aux « fréquences » -, on peut alors appliquer le filtre comme décrit en (7). Il suffit pour cela de créer la matrice H et une fois cela effectué, on multiplie U, H, U^T et F afin d'obtenir le graphe filtré.

```python
def filtrage_graphe(F,W):
    n = np.shape(W)[0]
    D = np.zeros((n,n))

    for i in range(n):
        for j in range(n):
            D[i,i] += W[i,j] #matrice de degré
        D[i,i] = 1/np.sqrt(D[i,i]) #D^-(1/2)

    L = np.eye(n) - np.linalg.multi_dot((D,W,D)) #laplacien normalisé
    U = np.linalg.eig(L)[1] #matrice de passage orthogonale
    U_trans = np.transpose(U)

    H = np.zeros((n,n))
    vp = np.linalg.eig(L)[0] #vecteur des valeurs propres du laplacien normalisé
    for i in range(n):
        H[i,i] = filtre(vp[i]) #filtre matriciel

    return np.linalg.multi_dot((U,H,U_trans,F)) #features filtrés
```

C. Programmation du *Mean Shift*

Maintenant que le filtrage a été expliqué au plan de la programmation, il convient de mettre en place la fonction *predict* qui prédira la classe des données non labellisées. En outre, cette fonction se divisera en deux étapes : l'initialisation des données labellisées et le *Mean Shift*. La première étape se charge de filtrer ou non les classes et d'établir les centroïdes. Une fois que cela est fait, on doit compter le nombre de données ayant servi au calcul de chaque centroïde. On pourra alors réaliser des moyennes pondérées lors de la réévaluation des centroïdes et non pas une simple moyenne entre un *query* et un centroïde.

Une fois l'initialisation effectuée, on peut créer la matrice des distances entre chaque *query* et chaque centroïde. On initialise par ailleurs une matrice de booléens qui permettra de gérer l'affectation temporaire d'un *query* à un centroïde et qui sera réévaluée à chaque itération. En effet, comme cela est expliqué dans la première partie, un centroïde ne sera affecté de manière définitive à un *cluster* que s'il apparaît deux fois comme étant le plus proche du centroïde correspondant.

```python
    distances = np.zeros((nb_features,nb_classes))
    deja_vu = np.zeros((nb_classes,nb_features),dtype=bool)

    #initialisation d'une matrice des distances entre les features et les centroïdes
    for i, centroid in enumerate(centroids):
        for j, feature in enumerate(features_copy):
            distances[j,i] = np.linalg.norm(feature - centroid)
```

D'autre part, il est évident qu'il faut programmer une façon de retirer les *queries* affectés à une classe afin d'éviter de les reprendre en compte lors de l'actualisation des centroïdes. Pour cela, à chaque fois qu'un *query* est affecté à un *cluster*, on le remplace par une liste vide et lorsqu'on réévalue la matrice des distances, on considère que tout élément vide de la liste des *queries* se trouve à une distance infinie des centroïdes. A cela, on ajoute un critère d'arrêt cohérent avec ce remplacement de *queries* par une liste vide. Enfin, on indente une boucle *while* qui s'arrêtera lorsque la liste des *queries* sera vide, soit une fois que toutes les données non étiquetées seront devenues étiquetées.

L'ensemble de ces critères ayant été explicités, il est possible d'en venir désormais au principe central du *Mean Shift* : l'affectation des données. Le principe est simple. On commence par parcourir les *queries* un à un. Prenons un *query j*. Si celui-ci est « vide », cela signifie qu'il a déjà été affecté à une classe ; on passe donc au *query* suivant. On récupère alors l'indice du centroïde le plus proche. S'il a déjà été affecté temporairement à ce centroïde, cela suggère qu'il faut ajouter *j* de manière définitive à la classe correspondante. On réalise ensuite – ou non - un filtrage de la classe ainsi mise à jour. Dans le cas contraire, autrement dit, si le *query* n'a pas encore été affecté temporairement au centroïde, on l'utilise uniquement pour la réévaluation des centroïdes. On actualise donc pour finir les centroïdes, puis la matrice des distances et on recommence le processus tant que tous les *queries* n'ont pas tous été affectés. Le code *in extenso* est donné ci-après.

```
temp_centroids=classes
while false(features_copy):  # = tant que la liste des features ne contient pas que des éléments vides

    for jind,feature in enumerate(features_copy):

        if len(feature)==0:
            pass

        else:
            #récupération de l'indice du centroïde le plus proche de chaque feature
            closest_centroid_distance = np.min(distances[jind])
            closest_centroid_index = int(np.where(distances[jind] == closest_centroid_distance)[0])

            #si le feature avait déjà été affecté temporairement à la classe, on l'ajoute définitivement
            if  deja_vu[closest_centroid_index,jind] :
                classifications[jind] = closest_centroid_index

                #filtrage dynamique
                if filtered:
                    c = classes[closest_centroid_index]
                    c.append(features_copy[jind])
                    cluster_filtered = filter(c)
                    centroids[closest_centroid_index] = np.average(cluster_filtered, axis = 0)
```

```
                #pour ne plus prendre en compte le feature qui est devenu labellisé, on le remplace par une liste vide
                features_copy[jind]=[]

            #si le feature n'a jamais encore été détecté comme le plus proche de la classe, on l'affecte temporairement
            #en s'en servant uniquement pour mettre à jour le centroïde correspondant
            else:
                deja_vu[closest_centroid_index,jind] = True
                temp_centroids[closest_centroid_index].append(feature)

    #mise à jour des centroïdes
    for i, centroid in enumerate(centroids):
        centroids[i]=np.average(temp_centroids[i],axis=0)

        #mise à jour de la matrice des distances
        for j, feature in enumerate(features_copy):
            if len(feature)==0:
                    distances[j,i] = Inf
            else :
                distances[j,i] = np.linalg.norm(feature - centroid)
```

D. Mesure de l'efficacité moyenne et de la variance des données

Il reste enfin à implémenter le code permettant de comparer les résultats donnés par la fonction *predict* avec la classe dans laquelle les *queries* avaient été piochés. Au vrai, la fonction *predict* servait de test et il convient désormais de mesurer son efficacité. Pour cela, après avoir généré les données de manière aléatoires (cf. II.A), on fait appel à *predict*, une fois en filtrant et une fois sans filtrer. On génère une liste avec les classes attendues puis on fait la différence entre cette liste et le résultat retourné par *predict*. Ainsi, si la différence est nulle, *predict* a affecté le *query* à la bonne classe et sinon, ce n'est pas le cas. Pour finir, on réitère ces étapes une centaine de fois dans le but d'obtenir la moyenne. On récupère ensuite l'efficacité moyenne de l'algorithme ainsi que la variance des données.

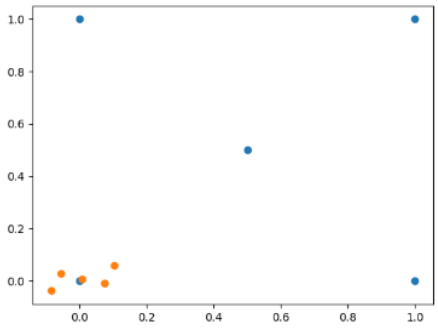
IV. Comparaison et analyse des résultats

A. Nombre de tests

Avec 5 classes, 5 données labellisés par classe et 15 données non labellisés par classe, la totalité des combinaisons est égale à $\binom{20}{5}\binom{600}{20} \sim 10^{42}$, il est donc difficile d'évaluer le nombre de tests nécessaires pour déterminer l'efficacité. L'efficacité peut varier de 6% entre deux essais de 100 tests. Par conséquent, on a fixé le nombre de tests à 2000 de sorte que les variations de l'efficacité soient de l'ordre de 10^{-2} entre plusieurs essais.

B. Choix des filtres

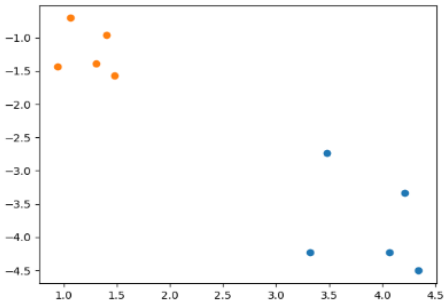
On travaille en dimension 2 dans un premier temps, le but étant de visualiser l'action d'un filtre sur un amas de points. Cette distribution des points est donnée par la bibliothèque python *sklearn.datasets.make_blobs.*



Pour tous les tests, qu'ils soient en dimension 2 ou non, nous avons choisi la fonction de similarité avec laquelle les résultats ont été les plus concluants.

$$s(x,y) = e^{-\|x-y\|^2 \cdot \beta} \quad (9)$$

Pour les essais plans, on a $\beta = 10$. Les points bleus correspondent respectivement aux points avant et après filtrages.

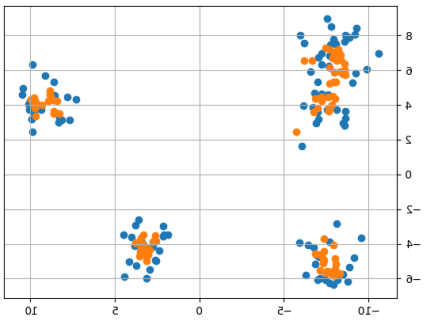


Sur ces deux premiers exemples, on a $h(\lambda) = e^{-\lambda/2}$ on constate que les amas de points se déplacent.

$$\|f_{filtered}\| = \|UHU^T f\| = \|Hf\| = \sqrt{\sum_i h(\lambda)^2 f_i^2},$$

car U est une matrice de passage orthogonale.

Le filtrage a une conséquence sur la norme de notre vecteur ce qui explique le déplacement sur les deux exemples. De manière générale, les filtres « passes bas » sont de la forme $h(\lambda) = e^{-\lambda * \alpha}$. Nous avons testé des fonctions de filtrages polynomiales et des fonctions en escaliers sans succès.



En l'occurrence, ce dernier exemple montre un filtrage sur plus de points avec $h(\lambda) = e^{-\lambda/3}$. Nous pouvons constater que tous les points se resserrent pour former un amas plus condensé : le filtre fonctionne de manière satisfaisante.

C. Une proposition alternative

Tous les filtrages présentés ci-dessus correspondent à des « filtres passes bas ». C'est-à-dire que $h(\lambda) \to 0$ quand $\lambda \to +\infty$. Les grandes fréquences sont par conséquent atténuées, comme nous l'avons déjà évoqué.

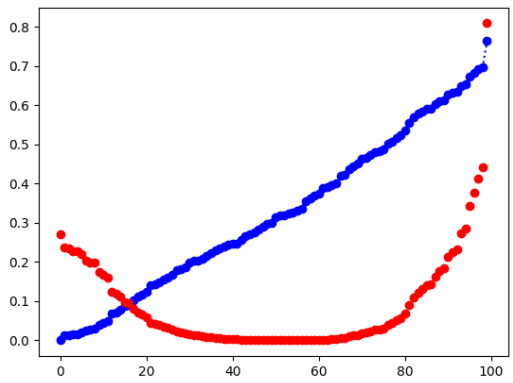
Un autre filtrage serait d'atténuer les valeurs propres proches de la moyenne et d'amplifier celles qui s'en éloignent. Les basses et hautes fréquences seraient, de fait, amplifiées, ce qui aurait pour effet de diminuer la variance dans les zones denses et de l'augmenter dans les zones qui le sont moins. En d'autres termes, les classes se « refermeraient » sur elles-mêmes et s'éloigneraient des autres classes. On pourrait alors penser que la classification des données en serait facilitée. Un exemple de filtre (nous avons choisi d'exhiber celui pour lequel les résultats obtenus étaient les plus concluants) serait :

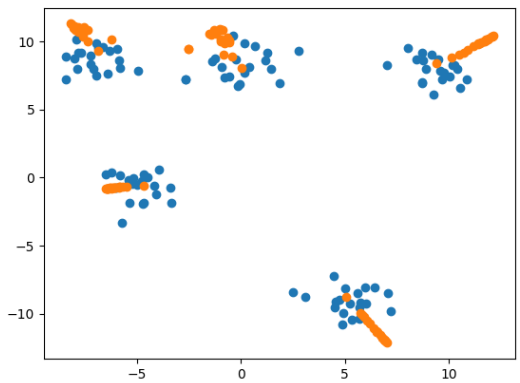
$$h(\lambda) = e^{|\lambda - \lambda_m|^3 * \alpha} - 1 \qquad (10)$$

Avec : λ_m la moyenne des valeurs propres

Dans la pratique, on prendra $\lambda_m = \lambda_{moy} + \frac{\lambda_{max} - \lambda_{moy}}{\beta}$ avec $\beta \in \mathbb{N}^*$ (11)

Le graphe ci-dessous à gauche correspond au numéro des valeurs propres dans l'ordre croissant en fonction de leur valeur. Plus précisément, la courbe bleue correspond aux valeurs propres avant et la rouge, aux valeurs propres après filtrage. On retrouve ainsi le principe de ne conserver que les valeurs se trouvant « loin » de la moyenne.





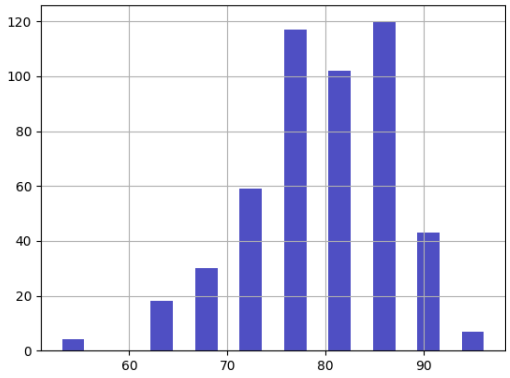
En particulier, pour $\alpha = 1$, on obtient l'exemple figurant en haut à droite où l'on constate que la densité des points orange est plus élevée que celle des groupements bleus.

De plus, on remarque que certains amas semblent s'aligner le long d'une droite passant par l'origine, comme si les vecteurs étaient colinéaires. Par conséquent, on pourrait penser que décider l'appartenance d'un point à un *cluster* reviendrait à trouver le coefficient directeur de la droite passant par l'origine et ce point.

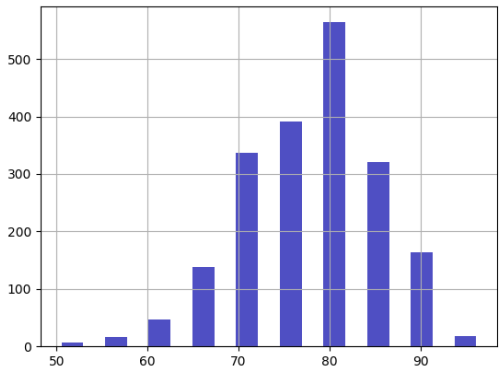
Une utilité à ce filtre serait donc de filtrer la totalité des données, labellisées et non labellisées. Mais malgré de nombreux essais et changements dans le choix des filtres, nous ne sommes jamais parvenus à plus de 60% d'efficacité. Effectivement, suite à cette opération, l'ensemble des *features* est très dense, les classes sont beaucoup plus serrées et l'action d'écarter les groupes les fait « exploser », déplaçant des données non labellisées loin de leur classe.

D. Comparaison des résultats obtenus

Il reste enfin à comparer et à analyser les résultats obtenus en lançant l'algorithme dans son ensemble sur le jeu de données. Sauf pour le filtrage dynamique qui était trop coûteux en ressources – nous avons réalisé dans ce cas 500 tests -, nous avons lancé les 2 000 tests afin de mesurer, puis de comparer l'efficacité moyenne de ces différents filtrages. Dès lors, on obtient le tableau donné sur la page suivante.



Histogramme de l'efficacité moyenne avec filtrage dynamique et avec initialisation de toutes les données en même temps



Histogramme de l'efficacité moyenne sans filtrage dynamique et avec initialisation de toutes les données en même temps

Comparaison de l'efficacité moyenne de l'algorithme en fonction du filtrage	
Sans filtrage	77,6 %
Filtrage classe par classe	77,9 %
Filtrage dynamique avec initialisation classe par classe	77,7 %
Filtrage de toutes les données	73,2 %
Filtrage de toutes les classes en même temps	79,4 %
Filtrage dynamique avec initialisation de toutes les classes en même temps	79,1 %

Comme on peut le constater, le résultat est sans équivoque : c'est lorsque l'on filtre toutes les classes en même temps lors de l'initialisation que le résultat est le plus pertinent. En effet, dans ce cas, 79,4 % du temps, l'algorithme parvient à déterminer le bon *cluster* dans lequel doivent être affectés les *queries*. Si l'on pensait que le filtrage dynamique pouvait apporter de meilleures performances, on peut constater que, en plus d'un temps de calcul considérable, il est moins efficace. En comparant les histogrammes des résultats entre le filtrage de toutes les classes en même temps avec son homologue dynamique (cf. les histogrammes *supra*), on peut constater que la variance des résultats est plus élevée dans le cas dynamique. Ainsi, le filtrage dynamique est plus sensible aux données en entrée alors même que l'on aurait pu penser que filtrer en continu aurait eu tendance à lisser ces disparités obtenues dans les résultats. Cependant, cette conclusion pourrait être remise en cause car, on le rappelle, les tests réalisés pour le filtrage dynamique ont été moins nombreux en raison du temps de calcul qu'ils nécessitaient.

Par ailleurs, on remarque que le filtrage classe par classe uniquement à l'initialisation affiche de meilleurs résultats que lorsque l'on n'applique aucun filtrage. Cette conclusion est

essentielle dans la mesure où elle vérifie ce que nos tuteurs avaient eux-mêmes vérifiés dans leur étude.

Néanmoins, on constate que ce n'est pas le cas pour tous les types de filtrages. Effectivement, le filtrage de toutes les données – qu'elles soient étiquetées ou non -, affiche des résultats bien en-dessous de tous les autres cas. Ce résultat peu satisfaisant est lui aussi mis en avant par une hypersensibilisation aux données en entrée de l'algorithme. Bien que l'on ne puisse pas conclure de manière certaine sur le filtrage dynamique, son gain d'efficacité par rapport à des données labellisées non filtrées semble donc relativement faible.

En un mot, on constate que l'algorithme du *Mean Shift* est compatible avec le TSG mais seulement dans certains cas, le meilleur étant où l'on filtre en même temps toutes les données labellisées uniquement lors de l'initialisation.

Conclusion

Pour conclure, nous avons utilisé la théorie du Traitement du Signal sur Graphe couplée à un algorithme de *Mean shift* pour tenter d'améliorer la classification des données dans le cadre de l'apprentissage avec peu d'exemples. Nous avons réussi à démontrer que le TSG avait un impact généralement positif sur la classification des données et qu'il était compatible avec notre algorithme de *Mean Shift*. De plus, nous avons testé plusieurs types de filtrages et nous avons démontré, en particulier, que le filtrage sur l'ensemble des classes lors de l'initialisation permet d'améliorer l'efficacité de l'algorithme de presque 3%.

Néanmoins, comme le montre **[3]**, nous sommes encore loin des résultats obtenus pas nos tuteurs, quand ces derniers parviennent à une efficacité de plus de 80 % avec notre jeu de données, mais pour une différence d'efficacité moins importante entre les données filtrées et les données non filtrées. C'est pourquoi, à l'avenir, nous pourrions mener davantage de recherches sur le choix des filtres, enquêter sur d'autres algorithmes que le *Mean Shift* – comme celui des *K-means* - et creuser l'idée d'un filtre passe-bande (cf. IV.C).

Nous exprimons toute notre gratitude à Bastien Pasdeloup et à Mounia Hamidouche pour nous avoir initiés à ce domaine et pour l'aide qu'ils nous ont apportée pendant toute la durée du projet.

Bibliographie

- **[1]**, Mounia Hamidouche, Bastien Pasdeloup, Lucas Drumetz, Vincent Gripon, « Rapport final sur le projet – DeepGraphs », November 2020.

- **[2]**, David I Shuman, Sunil K. Narang, Pascal Frossard, Antonio Ortega, Pierre Vandergheynst "The Emerging Field of Signal Processing on Graphs : Extending High-Dimensional Data Analysis to Networks and Other Irregular Domains", https://arxiv.org/abs/1211.0053, 31 Oct 2012.

- **[3]**, Mounia Hamidouche, Carlos Lassance, Yuqing Hu, Lucas Drumetz, Bastien Pasdeloup, Vincent Gripon , "Improving Classification Accuracy with Graph Filtering", https://arxiv.org/abs/2101.04789, 12 Jan 2021.