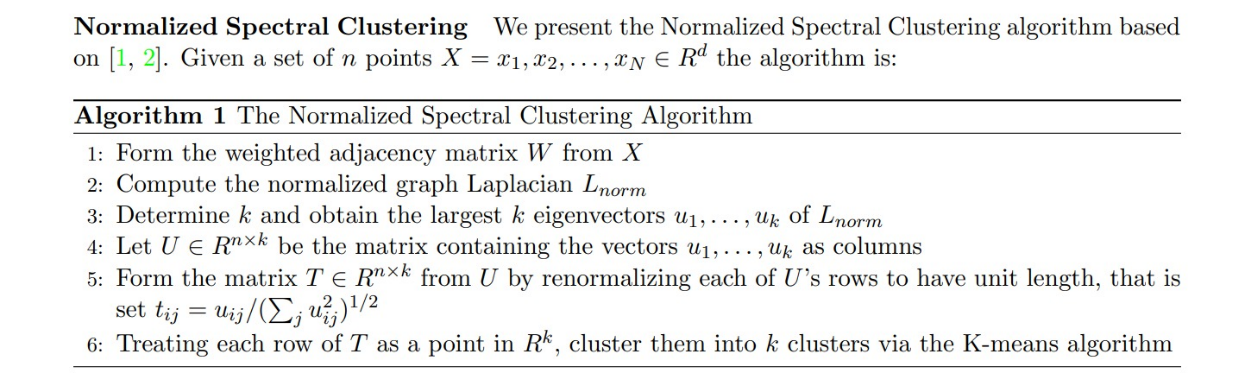
הפרוייקט בסדר כרונולוגי מהרגע שמריצים את קוד הmain שלו:

שלב ראשון: מn דאטה פוינטס בתור קלט (נקודות ממימד n גם כן) לקלטT לKmeans++ ממימד nxk.





מקבלים n דאטה פוינטס בתור קלט (וקטורים), כל דאטה פוינט בגודל n.

* 1. מוצג כשלב 1 בתמונה: מרכיבים מטריצת שכנויות W בין הנקודות (סימטרית)- נותנים משקל לכל זוג נקודות. במקום הi,j שמים משקל שהוא אינדיקציה לכמה הן רחוקות. קרוב = משקל קטן.



* 1. (שלב ביניים)יוצרים מטריצה אלכסונית D. בתא הi,i שמים את סכום השורה הi ממטריצת השכנויות.
  2. מפעילים על המטריצה האלכסונית העלאה בחזרת -0.5, כל איבר בה הופך להיות אחד חלקי השורש של עצמו.
  3. מוצג כשלב 2 בתמונה: מוציאים מהמטריצה האלכסונית וממטריצת השכוניות את מטריצת הלפלסיאן (Lnorm). (שלב ארוך מאוד) שימוש בjacobi)



למה מעוניינים בה? בגלל שכל הערכים העצמיים שלה יהיו אי שליליים ולא מרוכבים (ממשיים).



* 1. (שלב ביניים)מוצאים ערכים ווקטורים עצמיים: לוקחים את Lnorm ומריצים עליה את אלג' jacobi. אלג' יעקובי מקבל את מטריצה A בתור קלט ההתחלתי (ז"א A= מטריצת הלפסיאן בתור הקלט שנשלח לו). על מטריצה A הזו: \*להבין איך עשינו יעקובי ולחזור
  2. (שלב ביניים)משיגים מהפעלת יעקובי את המטריצה A הסופית וגם את המטריצה V, שזו הכפלה של מטריצות הסיבוב (לראות את פרטי יעקובי). האלכסון של A היא הערכים העצמיים של Lnorm ומטריצה V תהיה בעלת כל הוקטורים העצמיים של Lnorm. כל התכלית של יעקובי זה למצוא את הערכים העצמיים והוקטורים העצמיים של מטריצה.

התוצאה: n ערכים עצמיים וn וקטורים עצמיים (בהנחה שיש n דאטה פוינטס)

1.7שלב 3 בתמונה (אייגןגאפ יוריסטיק) ממיינים את הערכים העצמיים בסדר עולה ועוברים עליהם כדי למצוא את הקפיצה הכי גבוהה בין שני ערכים עצמיים בהפרש. ברגע שמוצאים אותה (ההפרש המקסימלי בין שני ערכים עצמיים גבוהים) אז איפה שהקפיצה התבצעה בסדר הממויין של הערכים – זה k (כלומר האינדקס של האיבר הראשון מבינהם). המטרה כאן הייתה למצוא את K, שאיתו נעבור לשלב הבא.



1.8 שלב 4 בתמונה: לוקחים את k האייגן וקטורס הכי קטנים (אצל אביב שינו את זה לגדלים) ולוקחים את ה k וקטורים עצמיים התואמים אליהם. משבצים בעמודות את k הוקטורים העצמיים האלה במטריצה חדשה – U, ממימד n על k.



1.9 שלב 5 בתמונה: הופכים את U למטריצה חדשה T- שזה פשוט U מנורמלת לפי ההסבר שבפסאודו קוד.

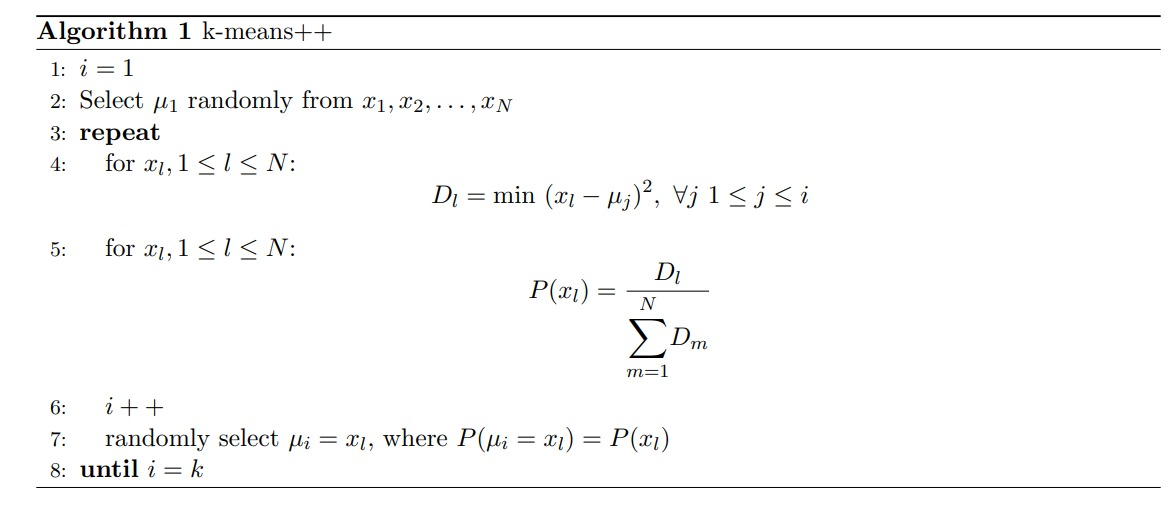


1.10 לוקחים את T שהיא מטריצה עם n שורות וk עמודות ומסתכלים על השורות. כל שורה היא מבחינתו דאטה פוינט, אז בעצם יש עכשיו n דאטה פוינטס וכל דאטה פוינט הוא ממימד k, והוא הולך להיות הקלט ההתחלתי בתור דאטה פוינטס של kmeans++ בשלב הבא. זה לא אותן דאטה פוינטס שהוא לנו בהתחלה, אלא הK וקטורים העצמיים הראשונים לפי סדר עולה של הערכים העצמיים שלהם, שהתייחסנו אליהם במקום כK וקטורים כל וקטור במימד n כn נקודות כל נקודה במימד k. זה אחד החלקים הכי מבלבלים באלגוריתם- אבל מה שתיארתי עכשיו זו T.



1.11 הפעלת Kmeans ++ על T, מטריצה של n נקודות, כל נקודה במימד k במטרה לקבל את הצנטרואידים ההתחלתיים החדשים.





תיאור Kmeans++:

תפקידו: לבחור מתוך n נקודות קיימות במימד k כל אחת(המטריצה T) k נקודות שיהיוו את הצנטרואידים ההתחלתיים לחלק הבא- Kmeans הרגיל שמאמן את המודל ומדייק את הצנטרואידים עוד יותר:

תיאור השלבים בפירוט:

שלב 1: לולאת for שרצה k פעמים. בכל איטרציה i:

1.1אם זו האיטרציה הראשונה: בוחרים את הצטנרוראיד רנדומלית מתוך n הדאטה פוינטס, בהסתברות שווה לכל צנטאוריד להבחר.

אחרת (i>=2) בוחרים את הצנטרואיד הבא אקראית מתוך n הדאטה פוינטס (אם הוא עדיין לא נבחר קודם, צנטרואיד יכול להבחר פעם אחת בלבד) לפי רשימת ההסתברויות שבנינו באיטרציה הקודמת.

1.2 מעדכנים את רשימת ההסתברויות ההסתברויות לקראת האיטרציה הבאה: לכל דאטה פוינט קובעים הסתברות בחירה להיות הצנטרואיד הבא שנבחר- probability שקובעים אותה לפי הdl שלו. dl של נקודה, הוא המרחק המינימלי שלה מבין כל המרחקים שלה מכל אחד מהצנטראודים שנאספו עד כה. נרצה לפעול לפי ההגיון הבא- שככל שלנקודה יש מרחק קטן יותר מצנטרואיד כלשהו, כך היא תהיה פחות אפקטיבית בתור צנטרואיד בעצמה ותנמיך את הפיזור של הצנטראודים על המסך. לכן נרצה שככל שנקודה קרובה יותר לצנטרואיד כלשהו, היא מאוד דומה לו, וכמו שלא היינו לוקחים את אותו הצנטרואיד פעמיים לרשימת הצנטרואידים שלנו, אין טעם שניקח אותה. לכן ננמיך את הסיכוי (p)של נקודה זו להבחר באיטרציה הבאה בתור צנטרואיד חדש. בכל איטרציה בלולאה אמנם נבחר רנדומית צנטרואיד חדש עד שנאסוף k צנטרואידים (אחרי k איטרציות), אבל נרצה שהבחירה תיעשה לפי משקל חכם שמגדיל את הסיכו של נקודות שרחוקות מכל יתר הצנטרואידים האחרים שנבחרו עד כה להבחר. ברגע שנקודה תתקרב מספיק לצנטרואיד שכבר נבחר בעבר, אז הdl שלה יהיה קטן וזה ביחס ישר לסיכוי שלה להבחר.

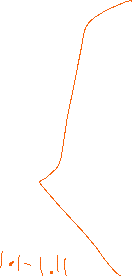
1. אחרי k איטרציות, כלומר אחרי שאספנו k צנטראוידים שונים מתוך הדאטה פוינטס, אנחנו מחזירים אותם בתור k הצנטראודיים ההתחלתיים (הם עדיין נקודות אמיתיות מתוך הדאטה פוינטס) לאלגוריתם שהולך לקבל אותם- kmeans הרגיל (כלומר מגיעים לפה אם goal = spkmeans)

כל זה(kmeans++) היה בעצם שלב 1 פה:

מה כולל שלב הInitialize:

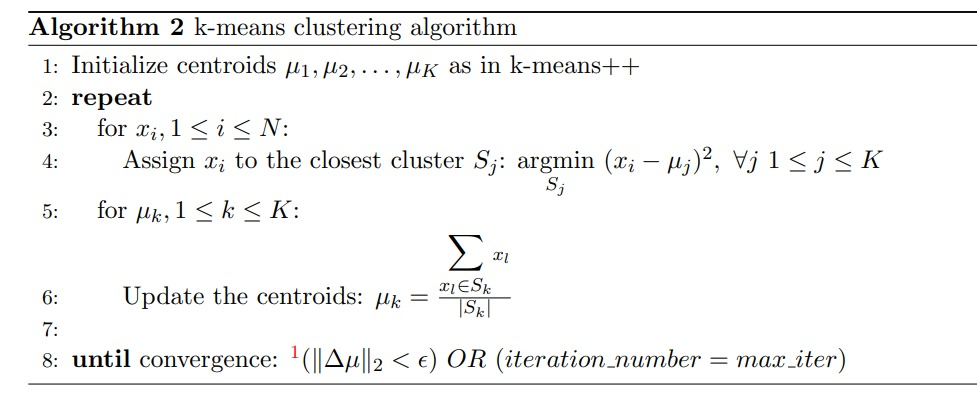


Initialize = Datapoints(input) nxn ->Wnxn ->…->Lnorm nxn-> Jacobi(Lnorm)-returns n eigenvalues and n eigenvectors of lnorm ->sort(eiganvalues) + find k with running eigengap\_heuristic(eiganvalues)->take first k eigenvectors of Lnorm, remember: you found eigenvectore and eigenvalues in jacobi ->now you have k points in length n each-> rotate them 90 degress to have n new points(the new data points) in length k each-> call them 'matrix Unxk'-> normalized U and call it Tnxk->Pass T as an input to Kmeans++(means T now represents new n data points in dimention k each)-> run kmeans++ on k and on T (T is n data points on dimension k each) to pick the best k points from those n points. You will end up with k points in dimension k each, which are the first input centroids for the real kmeans.



\*jacobi לוקח מטריצה סימטרית(Lnorm) ומחזיר את הוקטורים העצמיים שלה ואת הערכים העצמיים שלה. אם המטריצה הסימטרית היא nxn כמו שיש לנו, אז כל וקטור עצמי ממימד n.







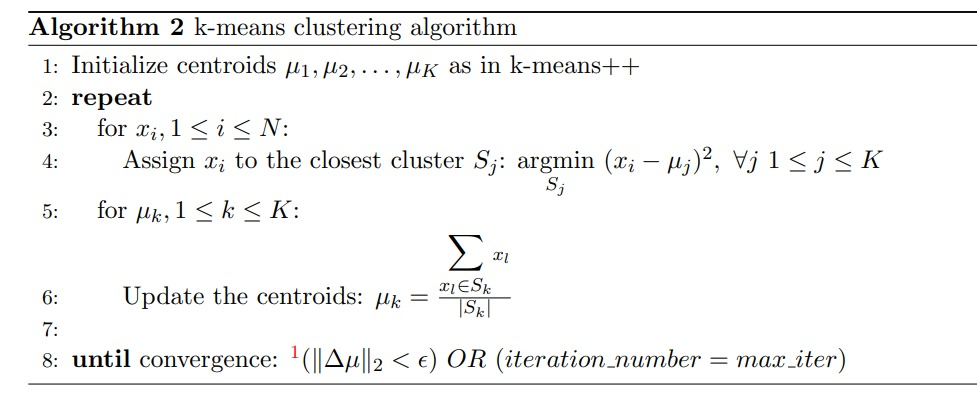
שלב 3: KMEANS הרגיל (לא++)

מטרתו- לעשות את הfit- הפעלתו היא בעצם אימון המודל על n דאטה פוינטס(T במקרה שלנו) יחד עם k צנטרואידים ראשונים מתוכן (שחישבנו בשלבים הקודמים) תפלוט לנו k צנטרואידים הכי טובים שיכולים להיות, כך שאם נעשה אחרכך Predict לנקודה כלשהי (שורה בT) לקטגוריה, נקבל את ההתאמה הכי טובה שיכולה להיות כי הקטגוריות שייקבעו, ייקבעו לפי המרחק המינימלי של הנקודה שבחרנו מT מצנטרואיד כלשהו מהצנטרואידים הסופיים ובגלל שהאלגוריתם Kmeans מוצלח אז הצנטרואידים האלה יהיו במיקומים הכי טובים לחיזוי שיבוא אחרכך.

בשלב זה עושים fit- 'מאמנים את המודל'. שולחים לאלגוריתם kmeans בתור קלט את T בתור הדאטה פוינטס שהוא מקבל (קלט, nxk), ואת k הצנטרואידים ההתחלתיים שמצאנו מתוך T בkmeans++.

מקבל כקלט (שאותו אנחנו בפרוייקט מקבלים במקרה מהשלב הקודם) k צנטרואידים התחלתיים שכל אחד במימד k.  
מקבל גםn דאטה פוינטס באותו מימד של הצנטרואידים שהk צנטרואידים של הקלט הם חלק מהם (זה לא יישמר ככה לאורך כל האלגוריתם. במקרה שלנו בפרוייקט, הדאטה פוינטס שמעבירים כאן לKMEANS (לא ה++) זו T.

האלגוריתם עצמו:





Kmeans(Data points nXk, k initial centroids from them kXk) פסאודו קוד:

1.לולאת while שעושה בכל איטרציה את הדברים הבאים ומפסיקה רק כשאף אחד מהצנטרואידים לא זז יותר (בפועל כשמרחק התזוזה של כל אחד ביחס למיקומו הקודם הוא פחות מאפסילון): (די פשוט)

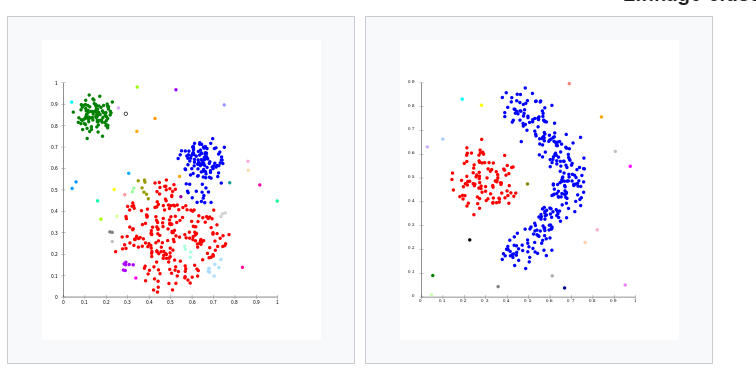


* 1. שיוך נקודות לקלאסטרים: לוקחת כל נקודה בדאטה פוינטס (T) במימד K ומשייכת אותה לקבוצה כלשהי (קלאסטר) לפי הצנטרואיד שהיא הכי קרובה אליו על המישור. לכל קלאסטר יש צנטרואיד שהוא אחראי על הבאת הנקודות לקלאסטר הזה, בגלל שכל הנקודות בו היו הכי קרובות אליו מבין כל הצנטרואידים האחרים.
  2. חישוב ממוצע כל קלאסטר בתור צנטרואיד חדש.

2. להחזיר את הצנטרואידים הסופיים שהתקבלו.הם ישמשו לpredict, ננבא נקודה לקלאסטר מסויים לפי המרחק המינימלי שלה מצנטרואיד כלשהו.

מחזירה את K הצנטרואידים שהתקבלו לאחר שלראשונה לא נראיתה תזוזה של אף אחד מהם לעומת האיטרציה הקודמת, במצב כזה כל נקודה בדאטה פוינטס מופתה לצנטרואיד שהכי מתאים לה ואנחנו בעצם לא רואים מעבר של נקודות לקלאסטרים אחרים ובגלל זה האלגוריתם מסיים לרוץ. שיוך כל נקודה לקלאסטר זה בעצם המאשין לרנינג- להתאים נקודות שמאופיינות בפיצ'רים מסויימים לקטגוריה (קלאסטר) שהוא אאל"ט un labled.





הדגמה: K means הרגיל(מומש בתרגיל בית 1):



כל אחת מהנקודות האלה- זה דאטה פוינט

כל צבע זה קלאסטר. בהתחלה מתחילים עם k קלאסטרים (3) . בוחרים בהתחלה 3 נקודות אקראיות. הקלאסטרים שמתחילים איתם את האלגוריתם הם אקראיים לגמרי. בוחרים k נקודות מתוך כל הn דאטה פוינטס (הקלט) שזה היה במטלה הראשונה.

ואז בככל איטרציה (מהשנייה ואילך):

מוצאים לכל קלאסטר צנטרואיד- נקודה ממוצעת של כל הקלאסטר.

'מנתקים' את הצבעים מכל הנקודות שוב, ואז עושים מיון מחדש לקלאסטרים אבל הפעם לפי k הצנטראודים.

חוזרים על התהליך שוב ושוב ברגע שיש convergance (כשהנורמה האוקלידית של כל אחד מהצנטרואידים לא זזה ביותר מאפסילון כלומר כשבפועל הצנטרואידים נשארו בין שתי איטרציות באותו המקום). כלומר הצנטרואידים נשארו במקום כי לא הזזנו שום דאטה פוינט מקלאסטר אחד לשני, כלומר הנקודות הגיעו לרוויה מבחינת כמה שאפשר לדייק את המיפוי שלהן לקלאסטרים.

בסופו של דבר אתה מחזיר k צנטרואידים שבאמצעותם נעשה ניבוי.

סיכום (אולי עדיף גם לקרוא לפני הקריאה של כל הקובץ הזה):

בפרוייקט בונים אלגוריתם שמקבל n נקודות במימד n, מעביר אותן עיבוד ארוך שכולל גם הורדת מימדים ובונה עבורן מודל בשם kmeans שכל מטרתו היא למפות את הנקודות האלה למספר לא ידוע מראש של קלאסטרים k על ידי פיצ'רים שהוא לא יודע מראש את מספרם (זה הk שמוצאים).

בעצם האלגוריתם לא רק מאמן את המודל על ידי זה שהוא מוצא צנטרואידים שאליהם נמפה לפי מרחק מינימלי מאחד מהם נקודות ובכך נייצר קלאסטרים של נקודות קרובות מספיק סביב צנטרואיד מסויים בצורה אופטימלית, הוא גם מחלץ מתוך הנקודות מה מספר הקלאסטרים האופטימלי עבור אותן הנקודות, זהו k.

הפרוייק ברמה הטכנית:

1. שחרור זכרון מלא, עבר את כל הטסטים
2. קובץ פייתון שמקבל קלט מהמשתמש (קובץ של הדאטה פוינטס וקלט משורת הפקודה על איזו פעולה רוצים לבצע). גם בC אפשר להריץ ריצה דומה.
3. כתיבת module בפייתון בשם setup.py שמאפשר ייצוא של קבצי C לקוד הראשי בפייתון. הוא מייצג לקובץ הפייתון גם את spkmeansmodule.c וגם את spkmeans.c
4. כתיבת הקובץ spkmeansmodule.c שמהווה ממשק בין C לפייתון. קוראים בפייתון לפונקציות שבתוכו בעזרת קריאה לcd1 שזה הקובץ של הסטאפ שהוא מתוכו מכיל חתימות של קבצי הspkmeansmodule.c ושל הspkmeans.c ועל די ייבוא cd1 אפשר ממש לקרוא לפונקציות מתוך שני הקבצים האלה, כמו fit וgoalroutingfunc שמוגדת בתוך spkmeansmodule.c ומחזירות Pyobject חזרה לפונקציות בפייתון. לspkmeansmodule.c ייבאנו את קובץ הheadrs של spkmeans.c ולכן אפשר מתוכו לקרוא להפעלת פונקציות שם.
5. המרה של אוייבקטים שהם Pyobject לאובייקטים בC על ידי פונקציות כמו PyFloat\_AsDouble וכדומה שמאפשרת תרגום אובייקטים מפייתון שנשלחים בתור pyobect ושלנו יש ידע מוקדם על מה הם היו בפייתון, לטיפוס הרלוונטי בC.
6. עבודה עם קבצים גם בC וגם בפייתון.
7. שחרור זכרון בכל התכנית שמריצה במקביל מספר קבצים. לדוגמה ייתכן שקראנו לפונקצייה בקובץ הראשי בC (spkmeans.c) מתוך קובץ המודל של c. אם יש שגיאה בקובץ הראשי, נצטרך לשחרר זכרון גם בקובץ המודול שקרא לו. כתבנו קוד חכם שיודע גם במקרה של שגיאה בהקצאת זכרון לשחרר את כל הזכרון שנצבר בקובץ הנוכחי ולחזור לקובץ שקרא ל ולשחרר גם שם ורק אז לצאת מהתכנית כולה.