אופטימיזציה ללמידת מכונה

מימוש אלגוריתם

Good Subspace (Shyamalkumar & Varadarajan, 2012)

דו"ח הפרויקט

שמות מגישים

עויידה חטיב, 201278066

אסאל חדאד, 207718701

מטרת הפרויקט

מטרת הפרויקט היא **יישום אלגוריתם Good Subspace** (Shyamalkumar & Varadarajan, 2012), שהוא אלגוריתם רנדומלי ליצירת משטח (Flat) k-ממדי (תת מרחב) בעל מרחק (אוקלידי) הקטן ביותר האפשרי לקבוצת נקודות נתונה במרחב d-ממדי, כאשר $k < d$. האלגוריתם רץ בזמן $O\left(\frac{ndk}{\varepsilon}\log\left(\frac{1}{\varepsilon}\right)\right)$, כאשר n הוא מספר הנקודות, ומבטיח פתרון בעל קירוב $O(1+\varepsilon)$ מהפתרון האופטימלי בהסתברות של $2^{-O\left(\frac{k}{\varepsilon}\log^2\frac{1}{\varepsilon}\right)}$. האלגוריתם מוצג כשיפור משמעותי לעומת אלגוריתמים קיימים עד אז שרצים בזמנים אקספוננציאליים במרחבים ממדים גבוהים.

אתגרים שפגשנו במהלך הפיתוח

במהלך יישום האלגוריתם כקוד נתקלנו בכמה אתגרים, נזכיר את המשמעותיים מביניהם:

- **בחירת הקבוע c** ב- $i = \frac{c}{\varepsilon} * \log\left(\frac{1}{\varepsilon}\right)$, שהוא מספר הקווים (כיוונים) שצריך לחשב. ערך קטן יצר מעט קווים, דבר שהשתקף בזמן ריצה מהר, אולם על חשבון הסיכוי להצלחה (העמידה בערבות הקירוב). ערכי c גבוהים מדי השיגו בדיוק ההפך (זמן ריצה איטי, סיכוי הצלחה גדול יותר). הערך של 100 יצר איזון בין דיוק לביצוע.
- **הטיפול במקרה הקצה בו כל הנקודות הם עם נורמה אפס**: המקרה יכול להופיע בפונקציה *sample_point_norm_weighted*, אשר נקראת במקומות שונים במהלך

הקוד שלנו, ודוגמת נקודה מבין כלל הנקודות בהסתברות יחסית לנורמה שלה. על כן, ההסתברות של נקודה כלשהי להידגם שווה לנורמה שלה מחולק בסכום הנורמות של כלל הנקודות, מה שיכול להוביל לחלוקה באפס במקרה קצה בו כל הווקטורים הם בעלי נורמה אפס. המאמר לא מתייחס למקרה הזה ולאופן הטיפול בו. המרצה ביקש להחזיר שגיאה (*ValueError*) במקרים כאלה במקום פתרון אחר שהצענו אז שלוקח אחת הנקודות בהסתברות שווה.

- **הטיפול במקרה בו הנורמה של הוקטור היא אפס או קטנה מדי**: זה יכול לקרות בפונקציה *unit_vector* שגם היא נקראת בהרבה מקומות במהלך הקוד, במטרה לנרמל את הוקטור ע"י החלוקה בנורמה. כאשר הוקטור הוא וקטור האפס, החלוקה תהיה באפס ועל כן הטיפול במקרה הזה היה דומה למקרה הקודם.

הרצת האלגוריתם

הרצת הקוד מתבצעת ע"י אחת מבין שתי הפקודות הבאות:

```
python good_subspace.py --n n_val --d d_val --k k_val --epsilon eps_val
python good_subspace.py --points npy_file --k k_val --epsilon eps_val
```

כאשר:

- n: מספר הנקודות.
- d: ממד המרחב הכללי.
- k: ממד המרחב של ה- Flat.
- epsilon: רמת הקירוב\דיוק.
- points: קובץ txt. המכיל קואורדינטות של נקודות.

ההבדל בין שתי הפקודות הוא שבשנייה הנקודות נתונות כארגומנט, ובכך ערכיהם של n,d מאותחלות כמספר הנקודות ומימד הנקודות בהתאמה (מצורף קובץ txt. המכיל נקודות כדוגמה). בפקודה הראשונה, מנגד, הנקודות לא נתונות והאלגוריתם מייצר אותם באופן רנדומלי, כך שיהיו n במספרם ובעלות ממד d. יצירת הנקודות נעשית ע"י בחירת תת מרחב רנדומלי מממד k, בחירת נקודות על המרחב הזה והוספת רעש כלשהו להם.

הפלט של האלגוריתם

הפלט כולל את:

- הנוסחה הפרמטרית של תת המרחב (Flat) שנמצא.
- ה- RD cost, העלות האופטימלית לפי ה- PCA, והודעה של האם האלגוריתם עמד בדרישת הקירוב (העלות האופטימלית * (1+ ε)).
- אם המרחב הוא דו-ממדי או תלת-ממדי, תוצג גם ויזואליזציה של ההתאמה ובה מוצגים הנקודות וה- Flat שנוצר.

אופן פעולת האלגוריתם ומבנה הקוד

הפונקציה הראשית בקוד היא *good_subspace*, אשר משתמשת בפונקציות עזר במהלך ריצתה. הפונקציה מקבלת כארגומנטים נקודות ממרחב גבוה, את k שהוא ממד הקטן מממד הנקודות, ואת פרמטר הדיוק ϵ, , ומחזירה בסיס אורתונורמלי שמתאר את תת המרחב מממד k המקרב בצורה הטובה ביותר את הנקודות. הפונקציה פועלת בשיטת הפרד ומשול, ובאופן רקורסיבי (על הפרמטר k), כאשר בכל שלב היא מפחיתה את k ב- 1. בשלב הראשון היא בונה סדרה של קווים ע"י דגימה רנדומלית מהנקודות הנתונות, בהסתברות התלויה בנורמה שלהן, מנרמלת אותן ובונה מהם קווים. לאחר מכן, היא בוחרת אחד מהקווים הללו באופן רנדומלי, ומקרינה את כל הנקודות על מרחבו האורתוגונלי. לבסוף, הפונקציה קוראת לעצמה עם ממד k-1 והנקודות המוקרנות. התהליך ממשיך עד למקרה הבסיס שבו k=1, אז נבחר קו המתאים לנקודות באופן רנדומלי.

הפונקציות השונות בהן נעשה שימוש:

- ***load_points(file_path)***: הפונקציה שמחלצת את הנקודות מקובץ נתון, והופכת אותם לייצוג מטריציוני (*Numpy array*).
  - קלט: קובץ טקסט (*txt*) בעל הפורמט הבא: כל נקודה נמצאת בשורה, וערכי הממדים בכל נקודה מופרדים ע"י רווחים.
  - פלט: מטריצה מגודל $n*d$, מסוג *numpy*, כאשר n הוא מספר הנקודות ו- d הוא הממד שלהן.
- ***good_subspace(P, k, epsilon)***: הפונקציה המרכזית, מחפשת תת מרחב מממד k המתאים לנקודות P, עם שגיאה שהיא לכל היותר $(1+\varepsilon)$ מהפתרון האופטימלי.

- קלט: מטריצת *numpy* דו-ממדית מגודל *n*d* שמייצגת את *n* הנקודות ממד *d* (*P*), מספר שלם שהוא ממד התת המרחב הרצוי (*k*), מספר ממשי בין 0 ל- 1 שהוא רמת הקירוב (*epsilon*).
- פלט: מערך דו ממדי בגודל *k*d*, מסוג *numpy*, שבו כל שורה מייצגת וקטור בבסיס האורתונורמלי של תת-המרחב שנמצא.

- ***pick_random_line(P, epsilon)***: בוחרת קו המקרב את הנקודות.
  - קלט: מטריצת *numpy*, דו-ממדית, המייצגת אוסף נקודות (*P*), מספר ממשי בין 0-1 שהוא רמת הקירוב (*epsilon*).
  - פלט: וקטור יחידה, מסוג *numpy*, המייצג את הקו.
  - דוגמה:

```
#input
P = np.array([
    [1.0, 2.0, 3.0],
    [2.0, 3.0, 4.0],
    [3.0, 4.0, 5.0],
    [4.0, 5.0, 6.0],
    [5.0, 6.0, 7.0]])

epsilon = 0.1

#output
[0.45584231 0.56980288 0.68376346]
```

- ***sample_point_norm_weighted(P)***: בוחרת נקודה מתוך אוסף נקודות, בהסתברות יחסית לנורמה שלה.
  - קלט: מטריצת *numpy* דו-ממדית המייצגת אוסף נקודות (*P*).
  - פלט: וקטור, מסוג *numpy*, המייצג את הנקודה.
  - דוגמה:

```
#input
P = np.array([
    [1.0, 2.0, 3.0],
    [2.0, 3.0, 4.0],
    [3.0, 4.0, 5.0],
    [4.0, 5.0, 6.0],
    [5.0, 6.0, 7.0]])

#output
[4. 5. 6.]
```

- ***unit_vector(v)***: מנרמלת את הווקטור.
  - קלט: וקטור *numpy* (*v*).
  - פלט: וקטור *numpy* בעל נורמה 1, שהוא הווקטור המנורמל.
  - דוגמה:

```
#input
v = np.array([3, 4, 1, 2])

#output
[0.54772256 0.73029674 0.18257419 0.36514837]
```

- ***project_onto_complement(P, v)***: מקרינה (*projects*) כל אחת מאוסף נקודות נתון למרחב האורתוגונלי של וקטור נתון.
  - קלט: מטריצת *numpy* דו-ממדית המייצגת אוסף נקודות (*P*), וקטור (*v*).
  - פלט: מטריצת *numpy* דו-ממדית, המייצגת את אוסף הנקודות המוקרן.
  - דוגמה:

```
#input
P = np.array([
    [1.0, 2.0, 3.0],
    [2.0, 3.0, 4.0],
    [3.0, 4.0, 5.0],
    [4.0, 5.0, 6.0],
    [5.0, 6.0, 7.0]])

line = np.array([1,0,0])

#output
[[0. 2. 3.]
 [0. 3. 4.]
 [0. 4. 5.]
 [0. 5. 6.]
 [0. 6. 7.]]
```

- ***orthonormalize(V)***: מייצרת בסיס אורתונורמלי לקבוצת וקטורים נתונה באמצעות תהליך *Gram-Schmidt*.
  - קלט: מטריצת *numpy*, המייצגת אוסף וקטורים (*V*).
  - פלט: מטריצת *numpy* המייצגת וקטורים המהווים את הבסיס האורתונורמלי.
  - דוגמה:

```
#input
V = np.array([[1, 0, 0],
              [1, 1, 0]])

#output
[[1. 0. 0.]
 [0. 1. 0.]]
```

- ***print_subspace_formula(mu, basis)***: מדפיסה את הנוסחה הפרמטרית של תת המרחב שעובר דרך נקודה נתונה ונפרס ע"י אוסף וקטורים נתון.
  - קלט: וקטור *numpy* שהוא הראשית\וקטור הממוצע (*mu*), מטריצה דו-ממדית מסוג *numpy*, שהיא קבוצת וקטורים שמהווים בסיס (*basis*).
  - פלט: אין.
  - דוגמה:

```
#input
mu = [3. 4. 5.]
basis = np.array([[1,0,0]])

#output
x(α) = [3.0000, 4.0000, 5.0000] + α1*[1.0000, 0.0000, 0.0000]
```

- ***compute_rd_cost(P, mu, basis, tau=2)***: מחשבת את ה- *RD cost*, שהוא סכום המרחקים (החזקות של המרחקים) של הנקודות מתת המרחב הנתון, שהוא מדד לכמה הוא מקרב אותם.
  - קלט: מריצת *numpy* דו-ממדית שהיא אוסף נקודות (*P*), וקטור *numpy* שהוא הראשית\וקטור הממוצע (*mu*), מריצת *numpy* דו-ממדית שהיא קבוצת הווקטורים שמהווים בסיס (*basis*).
  - פלט: מספר ממשי שהוא ערך ה- *RD cost*.
  - דוגמה:

```python
#input
P = np.array([
    [1.0, 2.0, 3.0],
    [2.0, 3.0, 4.0],
    [3.0, 4.0, 5.0],
    [4.0, 5.0, 6.0],
    [5.0, 6.0, 7.0]])
mu = np.mean(P, axis=0)
basis = np.array([[1,0,0]])

#output
4.47213595499958
```

- ***optimal_pca_cost(P, k)***: מחשבת את עלות ההתאמה האופטימלית של אוסף נקודות נתון ע"י *PCA* עבור ממד *k*.
  - קלט: מטריצת *numpy* דו-ממדית שהיא אוסף נקודות (*P*), מספר שלם שהוא הממד (*k*).
  - פלט: מספר ממשי שהוא עלות ההתאמה.
  - דוגמה:

```python
#input
P = np.array([
    [1.0, 2.0, 3.0],
    [2.0, 3.0, 4.0],
    [3.0, 4.0, 5.0],
    [4.0, 5.0, 6.0],
    [5.0, 6.0, 7.0]])
k = 1

#output
4.440892098500626e-16
```

- ***visualize_plane_3d(P, basis), visualize_line_2d(P, basis), visualize_line_3d(P, basis)***: פונקציות ויזואליזציה, המייצרות *plot* שמכיל את הנקודות הנתונות בנוסף לתת המרחב הנפרס ע"י ווקטורי הבסיס הנתונים.
  - קלט: מטריצת *numpy* דו-ממדית שהיא אוסף נקודות (*P*), מטריצת *numpy* דו-ממדית שהיא אוסף וקטורים המהווים בסיס (*basis*).
  - פלט: אין.

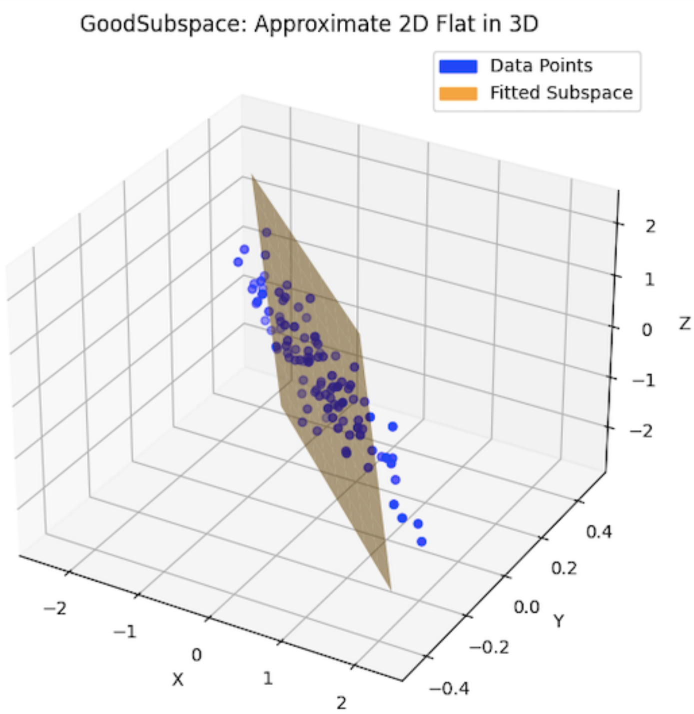
דוגמאות הרצה

<u>דוגמת הרצה 1</u>

python good_subspace.py --n 100 --d 3 --k 2 --epsilon 0.5

```
The Fitted Subspace:
  x(α) = [-0.0424, 0.0218, -0.1217] + α1*[-0.2153, 0.0370, 0.9758]
                                    + α2*[0.9555, -0.1982, 0.2184]

Optimal Cost (PCA): 0.482672
Allowed Bound: 2.2500 × 0.482672 = 1.086011
RD Cost: 0.895169
     ✅ The Algorithm satisfies the guarantee.
```

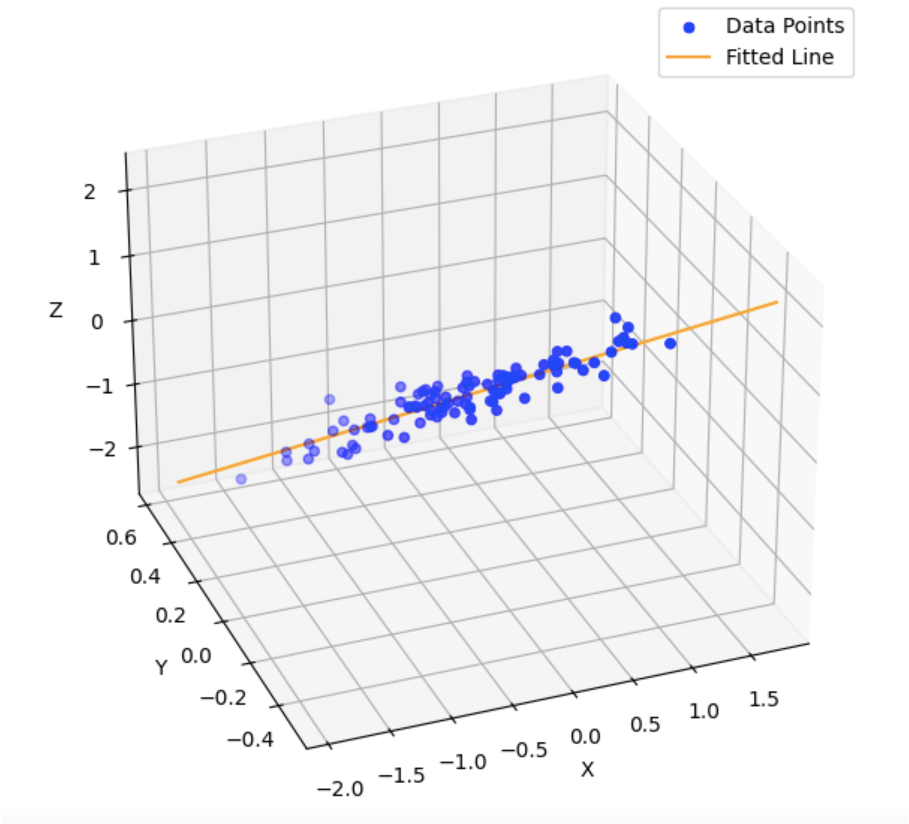


<u>דוגמת הרצה 2</u>

python good_subspace.py --n 100 --d 3 --k 1 --epsilon 0.1

```
The Fitted Subspace:
  x(α) = [-0.0891, 0.0322, -0.0983] + α1*[-0.6062, 0.1827, -0.7740]

Optimal Cost (PCA): 0.668118
Allowed Bound: 1.1000 × 0.668118 = 0.734929
RD Cost: 0.679274
     ✅ The Algorithm satisfies the guarantee.
```

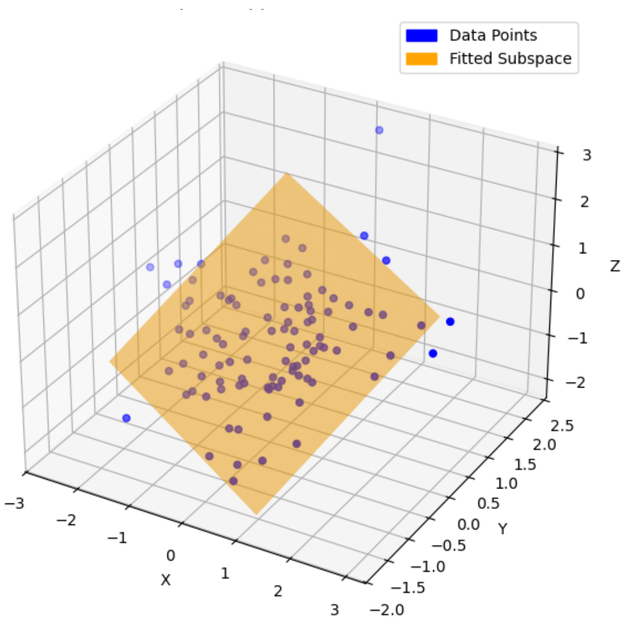


<u>דוגמת הרצה 3</u>

python good_subspace.py --points points_3d.txt --k 2 --epsilon 0.5

```
The Fitted Subspace:
  x(α) = [0.0788, -0.0491, -0.0631] + α1*[0.9219, -0.2620, -0.2854]
                                      + α2*[0.3867, 0.5779, 0.7186]

Optimal Cost (PCA): 0.515116
Allowed Bound: 2.2500 × 0.515116 = 1.159011
RD Cost: 0.527254
      ✅ The Algorithm satisfies the guarantee.
```



<u>דוגמאות הרצה של מקרי קצה</u>

1. המקרה בו **כל הנקודות שווה ל- 0**. דוגמת:

```
P = np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
```

הפלט מציג שגיאה כמצופה בשל אי-יכולת האלגוריתם לעשות דגימה ממושקלת לפי הנורמה:

```
ValueError: All points have zero norm. Cannot perform norm-weighted sampling.
```

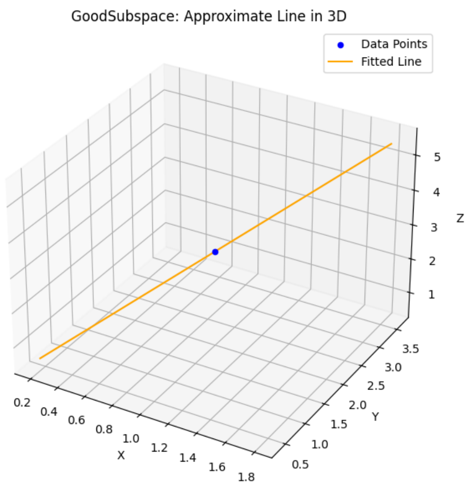
2. המקרה בו **אוסף הנקודות מכיל נקודה אחת בלבד**, דוגמת:

```
P = np.array([[3, 2, 5]])
k = 1
```

הפלט כמצופה שונה בכל פעם, אולם תמיד מייצר קו העובר בדיוק בתוך הנקודה, לכן מתקבל שה- RD Cost שווה לאפס.

```
The Fitted Subspace:
  x(α) = [1.0000, 2.0000, 3.0000] + α1*[0.2673, 0.5345, 0.8018]

Optimal Cost (PCA): 0.000000
Allowed Bound: 1.1000 × 0.000000 = 0.000000
RD Cost: 0.000000
     ✅ The Algorithm satisfies the guarantee
```



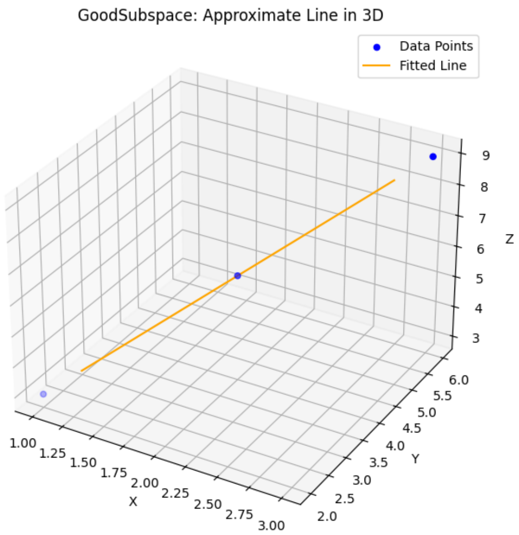
3. המקרה בו **כל הנקודות נמצאות על קו ישר**. דוגמת:

```
P = np.array([[1, 2, 3], [2, 4, 6], [3, 6, 9]])
k = 1
```

מתקבל תמיד אותו פלט שהוא הקו הישר העובר דרך 3 הנקודות. מכאן ה- RD Cost שווה לאפס:

```
The Fitted Subspace:
  x(α) = [2.0000, 4.0000, 6.0000] + α1*[0.2673, 0.5345, 0.8018]

Optimal Cost (PCA): 0.000000
Allowed Bound: 1.1000 × 0.000000 = 0.000000
RD Cost: 0.000000
```



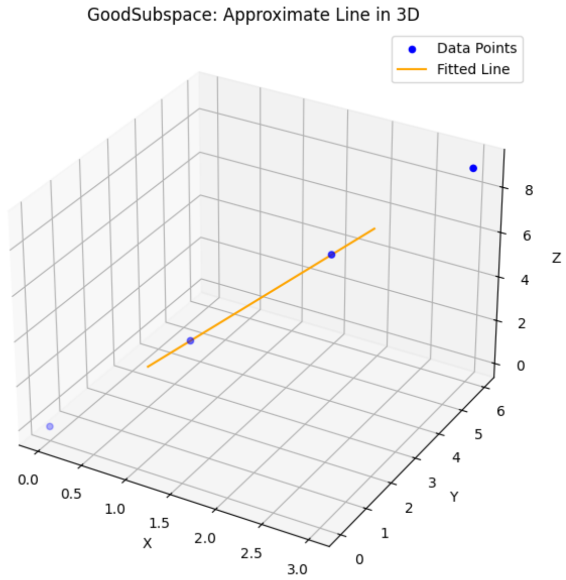
4. המקרה בו **הנקודות כולן על קו ישר, בנוסף לנקודה האפס**: אוסף הנקודות במקרה הזה דומה לזה שבמקרה הקודם, בנוסף לנקודת האפס. כלומר:

```
P = np.array([[1, 2, 3], [2, 4, 6], [3, 6, 9], [0, 0, 0]])
k = 1
```

הפלט הוא אותו פלט של המקרה הקודם, זה נובע מכך שנקודת האפס בעלת נורמה אפס, לכן היא לעולם לא תיבחר לדגימה של כיוון, ולא משפיעה על הבנייה של ווקטורי הבסיס.

```
The Fitted Subspace:
  x(α) = [1.5000, 3.0000, 4.5000] + α1*[0.2673, 0.5345, 0.8018]

Optimal Cost (PCA): 0.000000
Allowed Bound: 1.1000 × 0.000000 = 0.000000
RD Cost: 0.000000
```



5. המקרה הבא:

```
P = np.array([[1, 0, 0], [2, 0, 0], [3, 0, 0]])
```

הפלט:

```
ValueError: All points have zero norm. Cannot perform norm-weighted sampling.
```

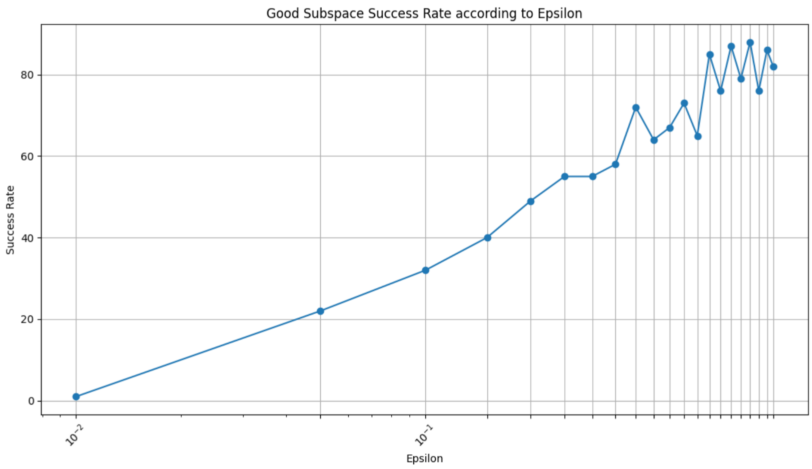
הבעיה נוצרת לאחר האיטרציה הראשונה בה האלגוריתם בוחר אחד מהקווים (למשל $[1,0,0]$), ומקרין את כל הנקודות על המרחב האורתוגונלי לקו הזה. מכיוון שכל הנקודות נמצאות על ציר x בלבד (קו ישר), ההקרנה מביאה את כולם להיות אפס. כלומר:

$$P\_proj = [[1,0,0],[2,0,0],[3,0,0]] - [[1,0,0],[2,0,0],[3,0,0]]$$
$$= [[0,0,0],[0,0,0],[0,0,0]]$$

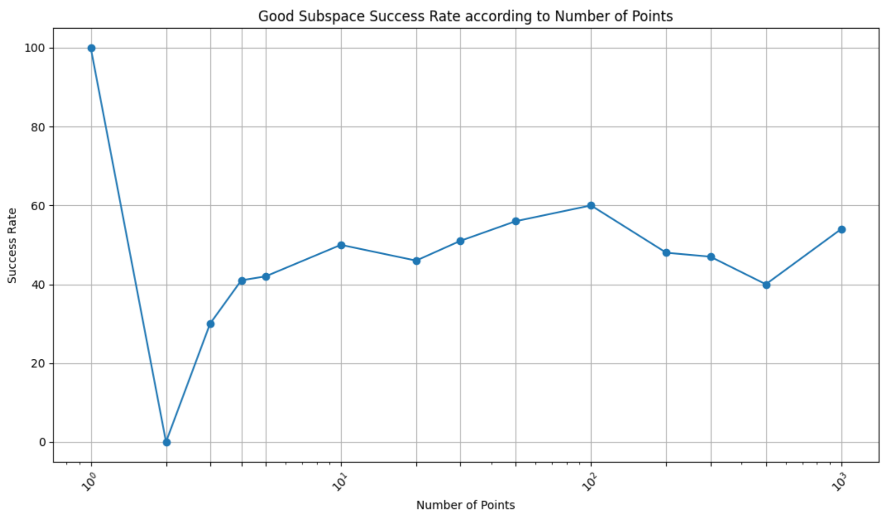
מה שיוצר את אותה בעיה שבמקרה 1, רק בשלב מתקדם.

השוואה בין ערכים שונים לפרמטרים

- **מידת השגיאה** (ε): האלגוריתם נבחן תחת ערכי אפסילון שונים שבין הטווח 0-1, ע"י השוואה בין סטים של 100 טסטים שהשתמשו באותה קונפיגורציה מלבד ערך האפסילון. התוצאות, המצורפות למטה, הראו קשר ישיר וחזק בין ערך השגיאה ומידת הצלחת האלגוריתם.

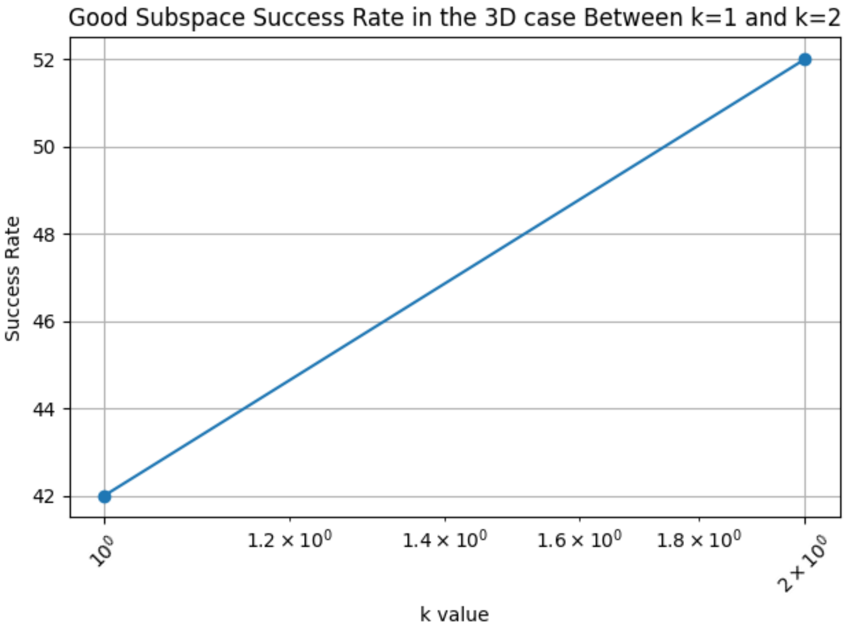


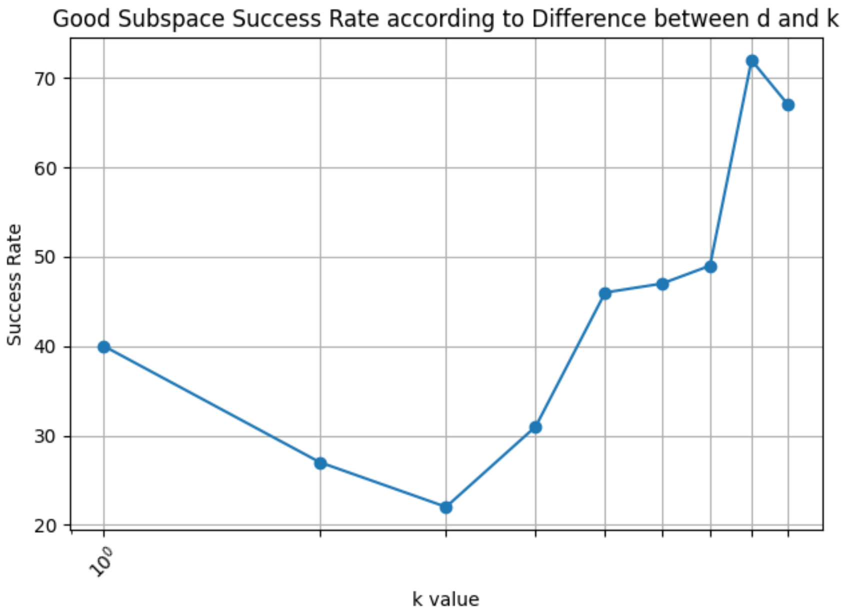
- **מספר הנקודות**: בדומה להשוואה הקודמת, השוואה נוספת בוצעה עבור מספר הנקודות שהאלגוריתם קיבל. גורם זה נחשד כבעל השפעה על הצלחת האלגוריתם בשל כך שיותר נקודות משמעה יותר מידע על המרחב. למרות שאחוז ההצלחה עלה בהתאם למספר הנקודות למספרים קטנים בטווח 1-10, מגמה זאת לא נצפתה במספרי נקודות גדול.



- **הפרש בין d ל-k**: הנחנו שזהו גורם משפיע גם כן על סמך כך שהאלגוריתם היה מצליח יותר עבור מרחב תלת-מימדי (d=3) במקרה שהתת-מרחב ממימד 2 (k=2) לעומת כאשר התת מרחב ממימד 1 (k=1), כפי שרואים בגרף הראשון למטה

המשווה בין שני המקרים. דבר זה נמדד באמצעות כך שהשווינו בין ערכי ה- k-ים השונים שבין 1-9 כאשר המרחב הוא ממימד 10, התוצאות תאמו חלקית לציפיות שלנו, וכנראה שיש גורמים אחרים המעורבים.



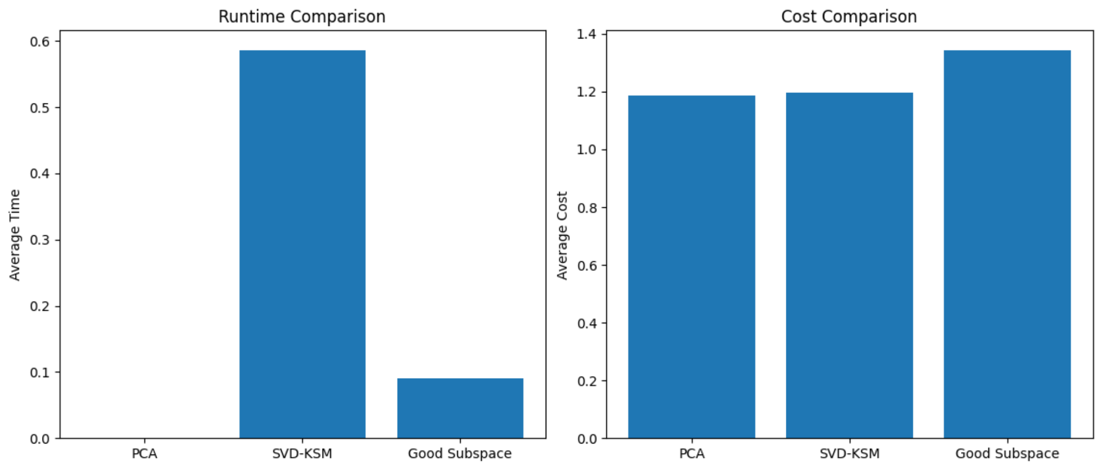


השוואה עם אלגוריתם SVD-KSM

השווינו את התוצאות עם מימוש של אלגוריתם SVD-KSM הפותר את אותה בעיה, בנוסף ל-PCA. ההשוואה נעשתה במונחים של התוצאות שהתקבלו (ה- RD Score במקרה של האלגוריתם שלנו, והמרחק האורתוגונלי במקרה של אלגוריתם SVD-KSM), וזמן הריצה. ההשוואה לקחה בחשבון את המקרים הבאים:

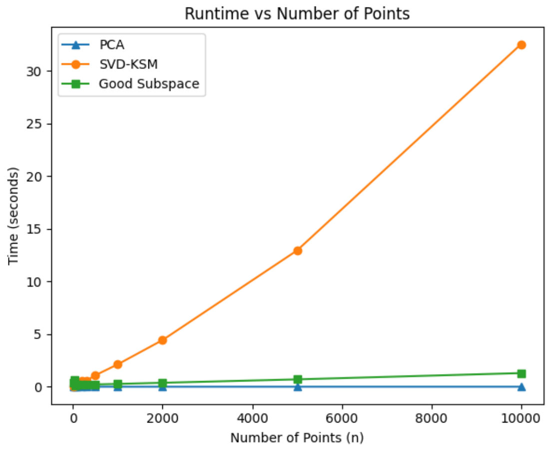
```python
test_cases = [
    {'n': 50, 'd': 2, 'k': 1},
    {'n': 50, 'd': 3, 'k': 2},
    {'n': 50, 'd': 3, 'k': 1},
    {'n': 200, 'd': 2, 'k': 1},
    {'n': 200, 'd': 3, 'k': 2},
    {'n': 200, 'd': 3, 'k': 1}]
```

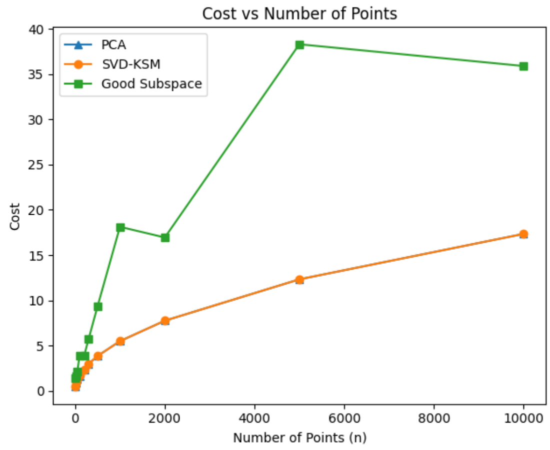
ולהלן התוצאות שהתקבלו:



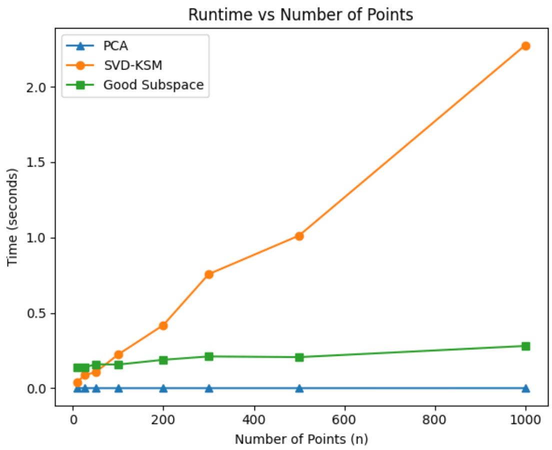
כמו כן, נעשו השוואות יותר מעמיקות שלקחו בחשבון את המשתנים המשותפים בין שני האלגוריתמים, שהם n,d,k. בשל אופיו הרנדומלי של האלגוריתם ועל מנת לצמצם את אפקט הרעש, כל מקרה מהמקרים המצוינים למטה נבחן ע"י 5 ריצות שהתמצעו.

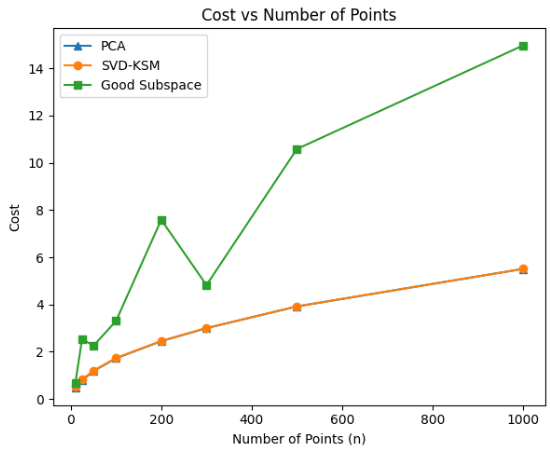
<u>ההשוואה כפונקציה של מספר הנקודות</u> (n): ערכי ה- d,k קובעו להיות 3 ו- 2 בהתאמה. להלן הגרף המציג את התוצאות של שני האלגוריתמים על פני ערכי n שונים:



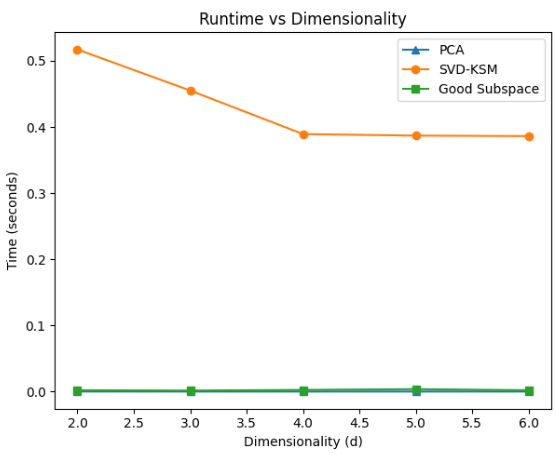


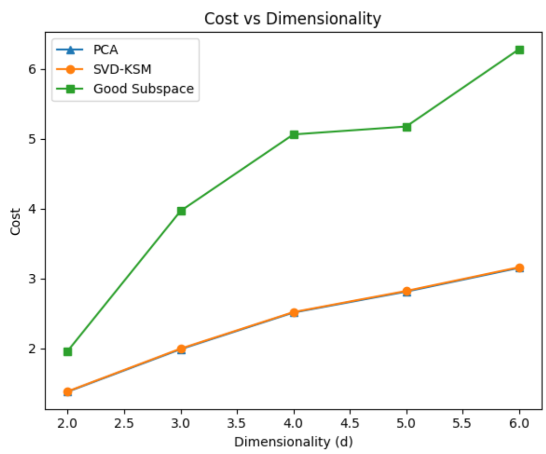
גרף היותר ממוקד בערכים עד 1000:





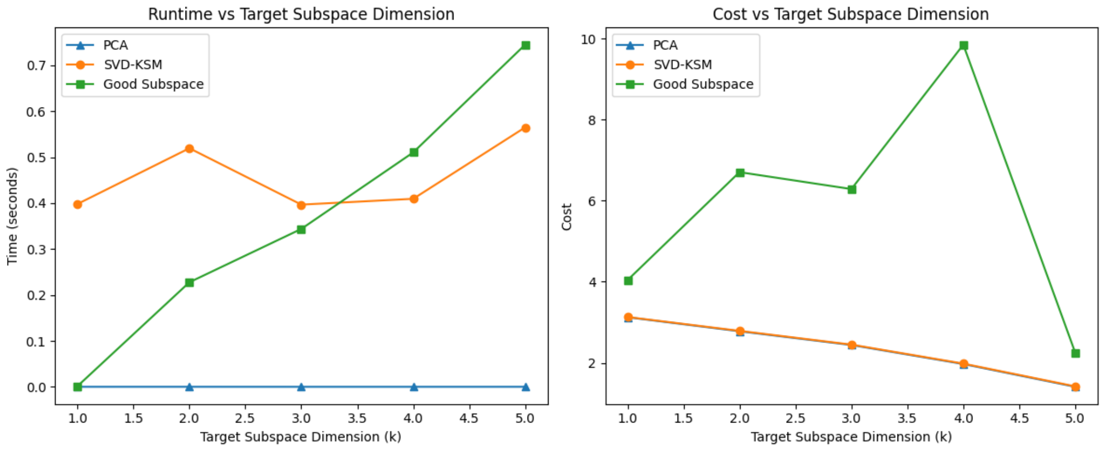
ההשוואה כפונקציה של הממד (d): ערכי ה- n,k קובעו להיות 200 ו- 1 בהתאמה. להלן הגרף המציג את התוצאות של שני האלגוריתמים על פני ערכי ה- d שבין 2 ל- 6:





ההשוואה כפונקציה של ממד המטרה (k): ערכי ה- n,d קובעו להיות 200 ו- 6 בהתאמה.

להלן הגרף המציג את התוצאות של שני האלגוריתמים על פני ערכי ה- k שבין 1 ל- 5:



אלגוריתם ה- Good Subspace מגלה יעילות מבחינת זמן ריצה, באופן יותר טוב לעומת אלגוריתם SVD-KSM, ולא נראה שהוא מושפע רבות מהשתנות ערכי המשתנים השונים.

מבחינת תוצאות האלגוריתמים, תוצאות המרחק האורתוגונלי של SVD-KSM כמעט תמיד היו טובות יותר משל האלגוריתם שלנו. ההפרש הזה גדל ככל ש- n היה יותר גדול, עד ל- n=5000 שם המרחקים התחילו להצטמצם. מגמה דומה נצפתה גם בממד (d) כאשר ככל שהממד נהיה גדול יותר ההפרש היה גדל לטובת ה- SVD-KSM. מהתוצאות שלנו לא נראה שלערך ה- k הייתה השפעה כלשהי על ערכי התוצאה.

מקורות וקרדיט

- **המאמר** להלן, שהציע את האלגוריתם:

Shyamalkumar, N. D., & Varadarajan, K. (2012). Efficient subspace approximation algorithms. Discrete & Computational Geometry, 47(1), 44-63.

- **מודלי שפה**: נעזרנו ב- Microsoft Copilot המותקן כתוסף ב- VS Code שבו השתמשנו לפיתוח הקוד. העזרה של המודל התבטאה בעיקר בהשלמת שורות ותיקון שגיאות. כמו כן, נעשה שימוש ב- ChatGPT, בהסבר לגבי דברים לא מובנים במהלך למידת המאמר, במקומות שונים במהלך מימוש הקוד, ובהצעות לגבי מקרי קצה שחלקם נכללו בדו"ח. למרות זאת, רוב הקוד, ובעיקר השלד שלו, נכתב על ידינו ועל פי ההבנה שלנו לאלגוריתם.

- **העמיתות** קלוד עזאם ונסיל כעביה, שעבדו על אלגוריתם הפותר את אותה בעיה, עזרו לנו בפיתוח הסקריפט המשווה בין שני האלגוריתמים, שלנו ושלהם, דבר שנתן לנו להעריך את טיב הביצועים של האלגוריתם.