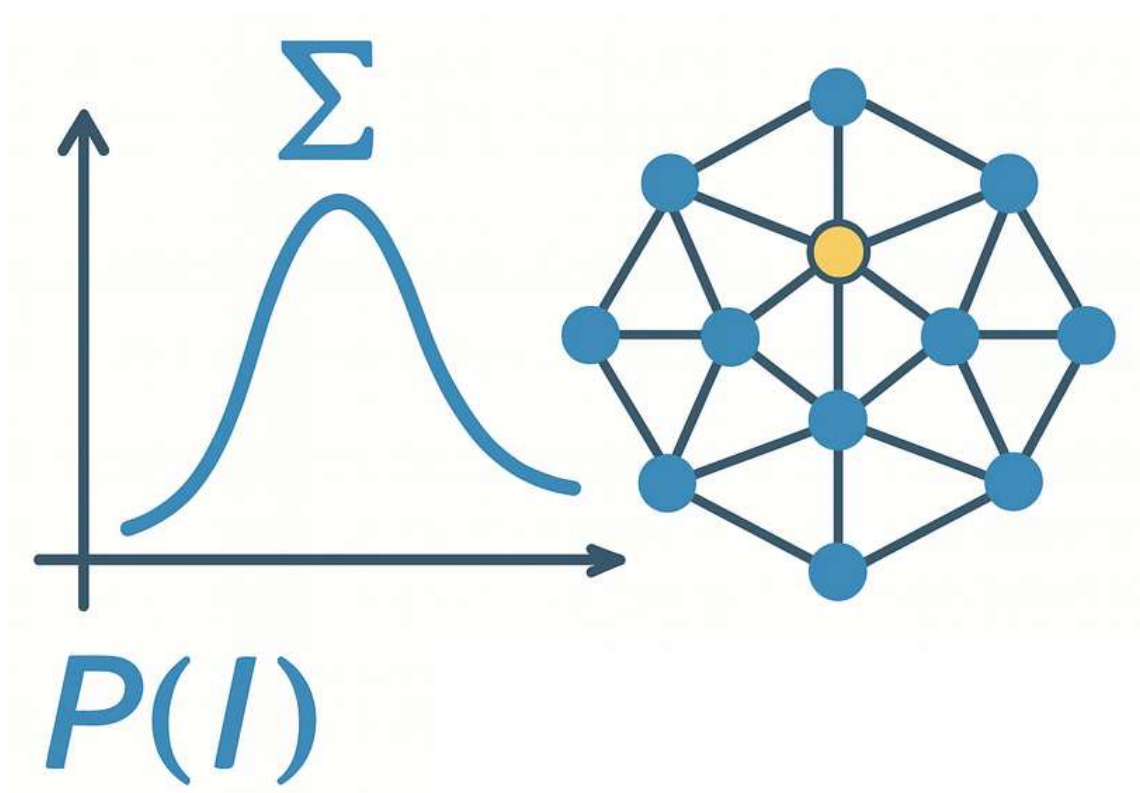


# מתמטיקה אינטואיטיבית למתכנתים לעידן ה-AI



מאת: תומר קדם

יוצר וארכיטקט התוכן של

AI Developer World Class Series

גרסה 2025 | 1.0

## זכויות יוצרים ושימוש

### ספרון 2: מתמטיקה אינטואיטיבית למתכנתים לעידן ה-AI

© 2025 תומר קדם כל הזכויות שמורות.

ספרון זה מופץ ללא עלות במסגרת הסדרה

### AI Developer World-Class Series

ומזמין אותך לבקר באתר הרשמי של הסדרה:

<https://tomkedem.github.io/AI-Developer-World-Class-Series/>

ההפצה מותרת אך ורק באמצעות שיתוף הקישור הרשמי לספרון או לקישור הראשי של הסדרה.

**אין לערוך, למכור או להשתמש בתוכן למטרות מסחריות ללא אישור בכתב מן המחבר.**

התוכן נוצר ונסקר בסיוע כלים מבוססי בינה מלאכותית, בהנחיית המחבר **תומר קדם**.

חלק מהתוכן נוצר ונסקר בסיוע כלים מבוססי בינה מלאכותית בהנחיית המחבר.

הספרון עושה שימוש בכלים בסיסיים של שפת Python ובספריות סטנדרטיות כגון NumPy.

שמות הטכנולוגיות והשפות המוזכרות בספרון הם סימנים מסחריים של בעליהם החוקיים, והשימוש בהם נועד לצורכי לימוד בלבד.

## תוכן העניינים

מבוא .....	1
<b>פרק 1 - למה מתמטיקה היא חלק מהעבודה.....</b>	<b>3</b>
איך מודלים "רואים" את הנתונים.....	3
למה ML זה בעצם "צמצום טעות" .....	5
הקשר בין נתונים, תחזיות והחלטות .....	8
מה מפתח צריך לדעת כדי להבין מה באמת קורה בפנים .....	12
<b>פרק 2 - ממוצע, חציון וסטיית תקן – בלי סיבוכים.....</b>	<b>15</b>
מה זה "מרכז" של דאטה.....	15
מה זה "פיזור" ולמה זה משנה.....	18
דוגמאות על נתונים אמיתיים.....	21
הדגמה קצרה ב-NumPy.....	24
<b>פרק 3 – הסתברות שמדברת בשפה של מתכנת.....</b>	<b>27</b>
הסתברות כתדירות בעולם אמיתי.....	27
איך מייצגים תרחישים בצורה פשוטה.....	29
חיבור ראשוני לעולם של סיווגים (Spam/Ham).....	35
<b>פרק 4 – הסתברות מותנית ובייס – הגרסה אנושית.....</b>	<b>39</b>

39	מה זה הסתברות מותנית באמת
42	בייס דרך טבלאות, לא דרך נוסחאות
45	מה מסתתר מאחורי השאלה "כמה סביר שזה נכון?"
48	הדגמה קטנה בקוד שתגרום לזה להתחבר
52	<b>פרק 5 – וקטורים – הלב של כל מודל</b>
52	איך מציגים אובייקט כסדרה של מספרים
58	למה כמעט כל AI עובד על וקטורים
61	איך טקסט, משתמש או תמונה הופכים לוקטור
65	<b>פרק 6 – נורמה ומרחק – מודדים את העולם</b>
65	מה זה "אורך" של וקטור
67	איך מודלים מודדים כמה דברים דומים או שונים
70	למה מרחק לא מספיק בניתוח טקסט
73	<b>פרק 7 – זווית ודמיון קוסינוס</b>
73	דמיון ככיוון ולא כמרחק
76	איך מודדים דמיון בין שני משפטים
81	חיבור ישיר לעולם ה-Embeddings
85	<b>פרק 8 – פונקציות – איך מודל חושב</b>

85	קלט ← פלט בצורה הכי פשוטה.....
87	מה זה "עקומה" ולמה כל מודל מנסה לרדת בה.....
90	מה זה בכלל מינימום.....
93	<b>פרק 9 - שיפוע - המנוע של הלמידה</b> .....
93	מה זה שיפוע בלי להגיד "נגזרת".....
95	למה שיפוע אומר לנו "כמה טעות יש כאן".....
98	הדגמה קצרה עם גרף פשוט.....
101	<b>פרק 10 - Gradient Descent - הלמידה עצמה</b> .....
101	רעיון ירידת המפל.....
103	למה המודל זז "נגד" השיפוע.....
106	מה קורה כשקצב הלמידה גדול מדי.....
108	הדגמת קוד שמראה ירידה למינימום.....
111	<b>פרק 11 - פרויקט סיום - mini_math_primer</b> .....
111	למה הפרויקט הזה חשוב.....
113	תרגיל: ממוצע וסטיית תקן.....
114	תרגיל: הסתברות מותנית.....
116	תרגיל: בייס על טבלה.....

119.....	תרגיל: נורמה
123.....	תרגיל: צעד Gradient Descent
125.....	<b>פרק 12 – איך כל זה מתחבר ל-ML ול-NLP</b>
125.....	למה כל הדברים שלמדנו חוזרים בכל מודל
127.....	איפה פוגשים אותם שוב בספרון הבא
129.....	מה יקל עליך כשהמודלים נהיים גדולים ומורכבים

## מבוא

כשנכנסים לעולם של בינה מלאכותית, יש רגע שבו מרגישים שכל התהליך קורה מאחורי וילון סגור.

המודלים מייצרים תוצאות מרשימות, אבל הדרך שבה הם מגיעים אליהן נראית **מסתורית** או **לא נגישה**.

התחושה הזו לא מגיעה מקושי אמיתי, אלא מפער קטן בהבנה. אנחנו רגילים לחשוב שמתמטיקה שייכת לעולם אחר, משהו רחוק, משהו שלמדנו פעם ושאין לו מקום בעבודה היומיומית.

המציאות שונה לגמרי.

מאחורי כל מודל עומדים כמה רעיונות **פשוטים**, **אינטואיטיביים**, ו**מאוד שימושיים**.

לא צריך נוסחאות ארוכות, לא צריך סימנים משונים, ולא צריך רקע מתמטי עמוק.

צריך רק להבין מהי השפה שהמודלים מדברים בה.

השפה הזו בנויה ממושגים קטנים וברורים:

**ממוצע**, **פיזור**, **מרחק**, **דמיון**, **שינוי**, **הקשר**, **הסתברות בסיסית**.  
אלה אבני הבניין שמופיעות שוב ושוב בכל מודל, בכל רמה ובכל גודל.

ברגע שהמושגים האלו מתיישבים בראש, משהו משתנה.

טקסט כבר לא נראה כמו רצף תווים, אלא כמו **וקטור של משמעות**.  
שינוי קטן בדאטה כבר לא נראה אקראי, אלא כמו **תנועה במרחב**.

שיפוע כבר לא מילה מתמטית מרוחקת, **אלא אות הכיוון** שהמודל משתמש בו כדי להשתפר.

אחת המטרות של הספר הזה היא להראות שכל זה מתחבר בצורה טבעית לכלים שאתה כבר משתמש בהם בפייתון. חישובים קטנים, יצירת רשימות, עבודה עם NumPy, בדיקת דמיון, וקטעי קוד שממחישים רעיונות בצורה מיידית. כשמבינים את היסודות, אפילו רעיונות שנראים מתקדמים יותר הופכים פתאום **ברורים ונגישים**.

הספר הזה לא מנסה להפוך אותך למתמטיקאי. המטרה היא לתת לך **תחושת שליטה אמיתית** בעולם של מודלים, ולהפוך את מה שקורה "בפנים" למשהו שאתה יכול להסביר לעצמך בקלות.

כל פרק עומד לבד, כל מושג מגיע עם דוגמה ברורה, וכל רעיון מלווה בקטע קוד קצר שמראה אותו בפעולה.

זהו השלב הראשון בסדרה רחבה יותר. מכאן נמשיך בהדרגה למתמטיקה יישומית, למידת מכונה, הבנת שפה, ומודלים גדולים.

וכל זה ירגיש הרבה יותר קל כשיש לך את היסודות האלו ביד. עכשיו אפשר להתחיל.

ולפני שנצלול לעומק – נענה על שאלה אחת:

**למה בכלל צריך את כל זה?**



# פרק 1 - למה מתמטיקה היא חלק מהעבודה

## איך מודלים "רואים" את הנתונים

כשאנחנו עובדים עם AI, אנחנו רגילים לחשוב במונחים של קבצים, טקסטים, תמונות ומידע עסקי.

אבל בתוך המודל – כל זה נעלם.

לא נשארת שפה, לא נשארת תמונה, לא נשארים מושגים.

הכול מתורגם לצורה אחת בלבד: **מספרים**.

וזה לא עניין טכני. זה עניין מהותי.

המודל "מבין" את העולם רק דרך מבנים מתמטיים, ולכן הדרך שאנחנו בונים, מאמנים, בודקים ומשפרים מודלים נשענת על אותם מבנים בדיוק.

## איך זה קורה בפועל?

כשאתה מזין למודל משפט כמו:

"המוצר הגיע מהר ושירות הלקוחות היה מעולה"

המודל לא שומע מילים.

הוא מקבל וקטור – רשימה של מספרים שמייצגים משמעות.

משפט אחר יקבל וקטור אחר, וכשהמודל משווה ביניהם, הוא משווה

**בין שני מבנים מתמטיים**, לא בין שני רעיונות.

אותו דבר קורה עם תמונות:

כל פיקסל מקבל ערך מספרי.

כל שכבה במודל לוקחת את המספרים האלה, משנה אותם קצת, ורצה הלאה.

בסוף כל תמונה – גם אם היא סלפי, וגם אם היא צילום לוויין – היא לא יותר מ-וקטור גדול.

זה הרגע שבו מפתח מבין משהו חשוב:

**AI לא עובד על "תוכן". AI עובד על מבנים.**

ולכן כל מי שמפתח מודלים, משלב אותם במערכות או אפילו רק משתמש בהם, צריך להכיר את השפה שבה הם חושבים. לא לעומק אקדמי – אלא ברמת אינטואיציה.

**למה זה משנה?**

כי כשאתה מבין איך מודל רואה מידע:

- אתה מבין למה משפטים דומים מקבלים תחזיות דומות
- אתה מבין למה מודלים טועים כשקלט יוצא מהטווח שהם מכירים
- אתה מבין למה "רעש" בנתונים מרסק ביצועים
- אתה מבין מה המשמעות של פיצ'רים חסרים או עיוותים בקלט
- אתה מבין איך לשפר את המודל בלי לגעת בקוד שלו בכלל

פתאום דברים שנראו כמו "כישוף" נהיים פשוטים.  
ברגע שאתה רואה את העולם דרך המשקפיים של מודל  
כל שאר הסדרה זורמת בצורה הרבה יותר טבעית.  
והבסיס של המשקפיים האלה הוא מתמטיקה.  
לא מתמטיקה כבדה.

**מתמטיקה שמסבירה איך תבניות הופכות למספרים, ואיך  
מספרים הופכים להחלטות.**

## למה ML זה בעצם "צמצום טעות"

ברגע שמבינים שמודלים רואים את העולם כמספרים, מגיעה  
השאלה הבאה:

איך הם מחליטים מה לעשות עם המספרים האלה?

התשובה תמיד חוזרת לאותה נקודה:

מודלים לא "מבינים" שום דבר.

**הם רק מנסים לצמצם טעות.**

זה אולי נשמע פשוט, אבל זו אחת התובנות הכי חשובות בעבודה  
עם AI.

כל מודל – קטן או ענק – קם בבוקר עם מטרה אחת:

לקבל קלט, לתת פלט, ולנסות להיות **פחות גרוע ממה שהיה**

**קודם.**

## מה זה אומר בפועל?

אם המודל מנחש מחיר של דירה – הוא רוצה שהטעות תהיה קטנה.  
אם הוא מתרגם משפט – הוא רוצה שהטעות הלשונית תהיה קטנה.  
אם הוא מזהה אובייקטים בתמונה – הוא רוצה לעשות כמה שפחות טעויות בזיהוי.

זהו.

Machine Learning הוא לא “למידה” במובן האנושי.  
הוא מערכת שמנסה שוב ושוב להקטין **פונקציה של טעות**.

וזה המקום שבו המתמטיקה נכנסת לתמונה:  
כדי “לרדוף אחרי טעות”, המודל צריך להבין איפה הוא טעה, בכמה  
הוא טעה, ולאיזה כיוון “לזוז” בשביל לשפר.

כל אחד מהמהלכים האלה הוא מהלך מתמטי קטן:

- למדוד את הטעות
- להבין האם היא חיובית או שלילית
- להבין אם צריך לעלות או לרדת
- לבצע צעד קטן בכיוון הנכון

ביחד זה יוצר תהליך שנקרא **ירידת מפל**, נושא שנגיע אליו בהמשך  
הספר.

**למה חשוב להבין את זה כמפתח?**

כי כשאתה מבין שהמודל כל הזמן מנסה לצמצם טעות, אתה גם מבין:

- למה נתונים "מלוכלכים" מבלבלים אותו
- למה שינויים קטנים בנתונים יכולים לגרום לשינוי משמעותי בביצועים

- למה לפעמים מודל חכם מתנהג בצורה טיפשית
  - למה צריך להיזהר ממצבים שבהם המודל "ננעל" על דפוס שגוי
  - איך לקרוא מטריקות בצורה הגיונית ולא כסת"ח מתמטי
- המודל לא מתעקש להיות צודק.

הוא מתעקש **להקטין את הטעות** שלו – ולעיתים זה מוביל אותו למקומות לא אינטואיטיביים.

ברגע שאתה רואה את המודל ככלי שמנסה לצמצם טעות ולא ככלי "חושב",

הרעש יורד, והעולם נהיה הרבה יותר ברור.

## הקשר בין נתונים, תחזיות והחלטות

כל מודל, קטן או גדול, עובד על רצף פשוט של שני שלבים מתמטיים:

### הוא מקבל נתונים ומפיק תחזית.

מכאן ואילך מתחיל שלב שלישי, שהוא **לא חלק מהמנגנון הפנימי של המודל**:

מישהו צריך להחליט מה לעשות עם התחזית הזו.

### המודל לא מקבל החלטות.

הוא לא מאשר עסקאות, לא חוסם מיילים, ולא בוחר פעולה. הוא רק מחשב.

ההחלטה מגיעה מבחוץ: מהמפתח, מהמערכת העסקית, או מתהליך אוטומטי שמופעל מעל התחזיות של המודל.

ולכן חשוב להבין את החיבור בין השלבים:

• **נתונים** הם מה שהמודל רואה.

• **תחזית** היא מה שהמודל מחשב.

• **החלטה** היא מה שאתה או המערכת עושים עם התחזית.

כדי לקבל החלטות נכונות, מפתח חייב להבין איך נוצרה התחזית, אילו מספרים עומדים מאחוריה, ומה המשמעות שלהם.

בלי זה המודל נראה כמו קופסה שחורה.  
עם זה הכול נהיה הגיוני וברור.

## 1. הנתונים

נקודת ההתחלה של כל מערכת AI היא נתונים.  
לא משנה אם זה טקסט, תמונה, סיגנל חיישן או מספרים  
עסקיים.  
המודל מקבל אותם בצורת **וקטור של מספרים** – עולם שבו כל  
תכונה, רעיון או פעולה מקבלת ערך כמותי.  
והרגע הזה קריטי:  
ברגע שהמידע נכנס למודל, הוא הופך לחלק **מתמטי** מהעולם.

## 2. התחזית

התחזית היא לא "תשובה".

היא **חישוב** שמתאר מה המודל חושב שיקרה.

היא יכולה להיות:

- מספר
- הסתברות
- וקטור חדש
- תווית
- דירוג
- טקסט שנוצר על בסיס הדפוסים שראה

אבל בכולן יש עיקרון אחד:

התחזית היא תוצאה של חישובים בין מבנים מתמטיים.

המודל לא "מבין" את התוכן.

הוא משווה, מודד, מחשב ומנסה לצמצם טעות ביחס למה שרצה לתת.



### 3. החלטה

השלב האחרון קורה **לא בתוך המודל**, אלא אצלך כמפתח.

אתה זה שמתרגם את התחזית להחלטה:

- האם המייל הזה הוא ספאם?

- האם כדאי לאשר עסקה?

- האם הרובוט צריך לזוז שמאלה?

- האם המשפט הבא צריך להיכתב בצורה אחרת?

ופה בדיוק עולה הצורך במתמטיקה אינטואיטיבית:

כדי להבין איך לתרגם את התחזית להחלטה אמיתית, אתה צריך להבין

**איך נוצרה התחזית, מה המודל מדד, ומה משמעות המספרים שהוא החזיר.**

בלי זה, התחזיות נראות כמו קופסה שחורה.

עם זה, העבודה הופכת לפשוטה וברורה.

**אז מה הקשר בין השלושה?**

נתונים הם נקודת ההתחלה.

תחזיות הן תוצאה של חישובים.

החלטות הן שימוש של אדם בתוצאה – לעיתים אוטומטי, לעיתים ידני.

וכל החוט שמחבר בין שלושת השלבים האלה  
נשען על הבנה בסיסית של **וקטורים**, **הסתברויות**, **שיפועים ומדדי**  
**מרחק**.

זה בדיוק מה שהספר הזה נותן לך:  
היכולת לראות את הקשר הזה בעיניים נקיות, בלי עומס ובלי  
נוסחאות.

**מה מפתח צריך לדעת כדי להבין מה באמת קורה בפנים**  
ככל שהמודלים נהיים גדולים ומתוחכמים יותר, קל לטעות ולחשוב  
שהם “מבינים” את העולם ברמה אנושית.  
אבל מאחורי הקלעים, גם המערכות המתקדמות ביותר נשענות על  
אותם עקרונות פשוטים שהכרנו בחלקים הקודמים:  
**וקטורים**, **מרחקים**, **הסתברויות**, ושאיפה מתמדת **להקטין טעות**.  
ופה בדיוק מתכנת נכנס לתמונה.

אתה לא צריך לדעת איך לקרוא מאמר אקדמי על רשתות עמוקות.  
אתה לא צריך להבין איך נראות ליבות של מודל עצום מבפנים.  
אבל אתה כן צריך להחזיק כמה אינטואיציות בסיסיות, כאלה  
שמאפשרות לך להבין מה קורה כשמודל מקבל קלט, מעבד אותו,  
ומחזיר תשובה.

## מה הן האינטואיציות שמפתח חייב להכיר?

- **כל מודל רואה את הקלט שלו כוקטור.**  
ברגע שאתה מבין ש"טקסט" הוא בעצם רשימה של מספרים, חצי מהקסם השחור נעלם.
- **כל תחזית מתבססת על מדידה.**  
דמיון, קרבה, שונות, פיזור – הכול מתורגם למספרים שמספרים למודל "כמה זה דומה למה שכבר ראיתי".
- **טעויות הן חלק מהחיים.**  
מודלים תמיד טועים.  
העניין הוא לא למנוע טעויות, אלא להבין **למה** הן קורות ומה אפשר לשפר.
- **למודל אין כוונה. יש לו חישוב.**  
הוא לא "מבין" טקסט, לא "נעלב", ולא "רומז".  
הוא בודק תבניות במספרים.
- **שינוי קטן בנתונים יכול להפוך לשינוי גדול בתחום המתמטי.**  
וזה משפיע על כל שלב בעבודה: איסוף, עיבוד, בדיקות ופרשנות של תוצאות.

## למה זה כל כך חשוב למפתח?

כי ללא האינטואיציות האלה, העבודה מול מודלים מתקדמים הופכת למשחק ניחושים.  
כשאין בסיס מתמטי, כל תוצאה נראית הגיונית או לא הגיונית לפי

תחושת בטן – וזה מסוכן.

כשיש בסיס, גם תוצאות מפתיעות הופכות לשיחה מקצועית:

“המודל הרחיק את הוקטור הזה יותר מדי”,

“הפיזור גבוה מדי ולכן התחזיות רועשות”,

“ההסתברות המותנית כאן נמוכה כי הקלט שייך לתת קבוצה

קטנה”.

ברגע שמפתח מדבר את השפה הזו

הוא לא רק משתמש במודל, הוא **מבין** אותו.

וזה בדיוק מה שהספר הזה בא לתת:

את היכולת להסתכל על מודלים בעיניים נקיות,

להבין את המנגנון הפנימי שלהם בלי לגעת בנוסחאות מסובכות,

ולהכין אותך לשלב הבא שבו כבר מתחילים ליישם את זה עם קוד

אמיתי.

## פרק 2 - ממוצע, חציון וסטיית תקן – בלי סיבוכים

### מה זה “מרכז” של דאטה

כשאנחנו אוספים נתונים, הדבר הראשון שאנחנו מנסים להבין הוא איפה “מרוכז” רוב המידע.

אנשים עושים את זה אינטואיטיבית כל הזמן: כששואלים אותך כמה זמן לוקח להגיע לעבודה, אתה לא מחשב נוסחה.

אתה זורק מספר שקרוב לתחושת הבטן שלך. זו תפיסה של **מרכז**.

במתמטיקה, המרכז הזה נקרא “מדד מרכזי”, והוא עוזר לנו להבין את הדאטה בלי להסתכל על כל ערך בנפרד.

יש שני מדדים מרכזיים שבדרך כלל מספיקים לרוב עבודת ה-AI:

- **ממוצע**

- **חציון**

שניהם מתארים את אותה שאלה: “מה הערך שמסכם את הדאטה בצורה הטובה ביותר?”

אבל כל אחד מהם עושה את זה בדרך אחרת לגמרי.

### ממוצע – הקול של כולם

הממוצע מספר לנו מה הערך שמתקבל אם “נפזר את העומס”  
שווה בשווה בין כל הנקודות.

הוא נותן משקל לכולם – כולל לחריגים.

לכן אם רוב האנשים מרוויחים 10,000 ש"ח וחמישה מנהלים  
מרוויחים חצי מיליון

הממוצע “נמשך” למעלה, למרות שהוא לא מייצג אף אדם אמיתי.

הממוצע טוב כשאין ערכים חריגים.

הוא פחות טוב כשיש פיזור קיצוני.

## חציון – הקול של האמצע

החציון לא מבצע חישובים מסובכים.

הוא פשוט שואל:

**“מה הערך שנמצא בדיוק באמצע הרשימה?”**

זו דרך מדויקת ויציבה יותר להבין מה **רוב** האנשים מרגישים.

היא לא מושפעת ממיעוט קיצוני.

לכן בשכר, בזמן תגובה של שרתים, במספר ביקורים באתר או בכל

דאטה עם “זנב ארוך”

חציון הוא כלי הרבה יותר שימושי מממוצע.

## למה זה חשוב למפתחי AI?

כי מודלים לומדים מתוך המספרים שאנחנו מזינים להם.  
אם הנתונים **מוטים**, אם הם **מעוותים**, ואם יש בהם **ערכים חריגים** המודל ילמד דפוס שגוי.  
הוא לא יודע להתעלם מרעש, הוא פשוט לומד את מה שהוא רואה.  
לכן השאלה “מהו המרכז?” היא לא שאלה אקדמית.  
זו שאלה פרקטית לגמרי שיכולה לקדם מודל קדימה – או להפיל אותו.  
ברגע שמבינים ממוצע וחציון ברמה אינטואיטיבית, הדאטה מפסיק להיות “רשימת מספרים”, ומתחיל להפוך **לתמונה אמיתית** שמראה מה באמת קורה שם.

## מה זה “פיזור” ולמה זה משנה

אחרי שמבינים איפה מרכז הדאטה נמצא, מגיעה השאלה השנייה:

### עד כמה הנתונים מפוזרים סביב המרכז הזה?

שני datasets יכולים לקבל אותו ממוצע ואותו חציון, ועדיין להיות שונים לגמרי.

לדוגמה:

- קבוצה אחת שבה כל הערכים כמעט זהים
- קבוצה שנייה שבה חלק מהערכים נמוכים מאוד וחלק קופצים לשמים

על הנייר זה נראה “אותו דבר”.

במציאות – אלו שני עולמות שונים לגמרי.

ופה נכנס רעיון הפיזור.

### פיזור – כמה הדאטה “רועש”

פיזור הוא מושג פשוט:

### כמה רחוקות הנקודות מהמרכז?

אם רוב הנקודות צמודות למרכז – זה דאטה יציב וצפוי.

אם הן מתפזרות לכל הכיוונים – יש רעש שמקשה על מודל ללמוד דפוס ברור.



דמיין שתי קבוצות של זמני תגובה (Latency) בשרת:

- קבוצה אחת: 88, 90, 91, 92, 89
- קבוצה שנייה: 20, 30, 400, 450, 15

יש להן חציון דומה, אבל הפיזור שונה לחלוטין.  
במערכת אמיתית – הקבוצה השנייה תהיה סיוט תפעולי.

## למה פיזור חשוב ל-AI?

כי מודל לומד מהממוצע והחציון – אבל חי בתוך הפיזור.  
פיזור גדול מדי אומר:

- המודל יתקשה להבחין בדפוסים
  - התוצאות יהיו לא יציבות
  - אותם נתונים יובילו לפעמים לתחזיות שונות
  - המודל “יתבלבל” ויזוז לכיוונים אקראיים בזמן האימון
- פיזור נמוך אומר שהדאטה נקי, עקבי ואמין יותר.  
ברוב פרויקטי ה-AI, לא המרכז מנבא הצלחה או כישלון

## אלא רמת הפיזור.

מתכנתי AI שמסתכלים על המרכז ולא על הפיזור  
בד”כ מפספסים את התמונה הגדולה:  
האם יש כאן דפוס אמיתי – או רק רעש שמתחפש לדפוס?

## סטיית תקן – המדד הפשוט לפיזור

כאן מגיע כלי שנשמע מסובך אבל הוא הכי פשוט בסיפור:

### סטיית תקן.

זו דרך למדוד במדויק כמה רחוקות הנקודות מהממוצע.  
היא לוקחת את כל הפיזור, “ממירה” אותו למספר אחד,  
ומאפשרת להשוות datasets שונים בצורה ברורה.

סטיית תקן גבוהה = דאטה רועש.

סטיית תקן נמוכה = דאטה יציב.

ברוב המערכות – בעיקר בתחזיות, NLP, מודלים התנהגותיים  
ובקרת איכות

סטיית תקן היא אחד המדדים הכי שימושיים שיש.

## דוגמאות על נתונים אמיתיים

כדי שהמושגים "ממוצע", "חציון" ו"סטיית תקן" לא יישארו באוויר, צריך לראות אותם על נתונים אמיתיים. ברגע שרואים מספרים מהעולם האמיתי – הכול מתחיל להרגיש הרבה יותר ברור.

ניקח שלושה סוגים קלאסיים של דאטה שמופיעים כמעט בכל פרויקט AI:

### 1. זמני תגובה במערכת (Latency)

נניח שאתה בודק זמני תגובה של שרת לאורך היום:

98, 102, 100, 101, 99, 103

הממוצע והחציון כמעט זהים.

הסטייה קטנה.

כך בדיוק נראית מערכת יציבה.

עכשיו תסתכל על סדרה אחרת מרובת עומסים:

40, 120, 300, 25, 500, 30

כאן סטיית התקן עצומה.

מבחינת מודל, זו מערכת שמציגה **התנהגות כאוטית**.

אי אפשר למצוא דפוס ברור.

## 2. מחירי מוצרים

נניח שאתה מנתח מחירים של מוצר מסוים ברשת:

- רוב החנויות: 230–260 ₪
- חנות אחת בטעות מוכרת ב-5,000 ₪
- חנות אחרת מוכרת בסייל ב-99 ₪

מה הממוצע יגיד לך?

**משהו באמצע בין כולם – אבל לא מייצג אף נקודה אמיתית.**

החציון, לעומת זאת, מתעלם מהחריגות ומחזיר תמונה נקייה יותר. ככה מתכנתי AI נמנעים מהטיות שנגרמות מערכים קיצוניים.

## 3. דירוגי משתמשים

במערכות המלצה (Recommenders),

משתמשים נותנים ציון 1–5.

אם מוצר מקבל:

5, 4, 5, 4, 1

הממוצע ייפול לאזור ה-3.8.

אבל הציון 1 עשוי להיות פשוט משתמש מתוסכל ביום רע.

פה סטיית התקן עוזרת להבין את העומק:

- סטייה נמוכה ← רוב הדירוגים עקביים
- סטייה גבוהה ← משהו לא יציב, אולי התנהגות חשודה

מודלים שמבינים את הפיזור יכולים לזהות אנומליות הרבה יותר טוב.

## למה הדוגמאות האלו חשובות?

כי בעולם האמיתי, דאטה אף פעם לא “נקי”.

תמיד יש:

- רעש
  - טעויות הזנה
  - ערכים קיצוניים
  - פערים בין ערכים
  - תקופות שקטות מול תקופות עמוסות
- כשמפתח מבין את המרכז והפיזור של הדאטה, הוא יודע:
- איך להכין את הדאטה למודל
  - איפה לנקות
  - איפה להחליק
  - ואיפה להיזהר שלא להרוס מידע חשוב
- סטטיסטיקה בסיסית היא לא “קישוט”.
- היא חלק מהאיכות של המודל.

## הדגמה קצרה ב-NumPy

אחרי שהבנו את הרעיונות מאחורי ממוצע, חציון וסטיית תקן דרך דוגמאות מהעולם האמיתי, הגיע הזמן לראות איך זה נראה בקוד. וזה בדיוק המקום שבו NumPy עושה את החיים פשוטים בצורה כמעט מגוחכת.

NumPy נועדה לעבוד עם מספרים במהירות וביעילות. בכל מה שקשור לדאטה – היא הכלי שהופך מתמטיקה “על נייר” למשהו שמתחבר ישירות למערכות אמיתיות.

### נתחיל מדאטה קטן ופשוט

ניקח רשימה שמייצגת לדוגמה זמני תגובה במערכת:

```
import numpy as np

latencies = np.array([98, 102, 100, 101, 99, 103])
```

זה כל מה שצריך כדי להתחיל לעבוד.

### ממוצע ← נקודת האמצע של כל הערכים

```
avg = np.mean(latencies)
print("ממוצע:", avg)
```

NumPy עושה את החישוב בשורה אחת, בלי לולאות ובלי קוד “מרובע”.

### חציון ← מה שקורה באמצע באמת

```
median = np.median(latencies)
print("חציון:", median)
```

כאן רואים שהחציון נשאר רגוע גם אם יש ערכים קיצוניים.

## סטיית תקן ← האם הדאטה רגוע או רועש

```
std = np.std(latencies)
print("סטיית תקן:", std)
```

זה המספר שמספר למודל אם יש כאן תבנית יציבה או כאוס.

## לראות הכול ביחד

```
print(f"מציון: {median}, מרכז הדאטה → ממוצע {avg}")
print(f"רעש במערכת → סטיית תקן {std}")
```

## מה רואים מהתוצאות?

- אם הממוצע והחציון קרובים ← הדאטה מאוזן
- אם סטיית התקן נמוכה ← המערכת יציבה
- אם סטיית התקן גבוהה ← יש התנהגות רועשת שהמודל יתקשה ללמוד ממנה

זוהי אנליזה בסיסית, אבל כזו שכל מפתח AI משתמש בה כמעט בכל פרויקט — גם כשלא שמים לב.

## למה חשוב לראות את זה בקוד?

כי ברגע שמפעילים את זה על נתונים אמיתיים בפרויקט שלך, המספרים מפסיקים להיות “מושגים מתמטיים”, והופכים לכלים אמיתיים שעוזרים:

- לנקות דאטה
- לזהות בעיות
- להבין דפוסים
- לשפר מודלים

והכי חשוב!

זה נותן תחושת שליטה.

פתאום סטיית תקן כבר לא נשמעת כמו הגדרה מספרית, אלא כמו  
כלי עבודה ברור.



## פרק 3 – הסתברות שמדברת בשפה של מתכנת

### הסתברות כתדירות בעולם אמיתי

לפני שמדברים על נוסחאות או סמלים, חשוב להבין משהו פשוט: הסתברות היא לא מתמטיקה מופשטת.

היא תיאור של **מספר דברים שקורים במציאות**.

כשאומרים “הסתברות של 0.2”, הרבה אנשים רואים סתם מספר. אבל מי שמפתח מערכות מבין שזה אומר: “זה קורה בערך 20 אחוז מהפעמים”.

וזוהו.

אין פה קסם.

### הסתברות היא בעצם ספירה

אם מתוך 100 בקשות לשרת:

20 נכשלות

80 מצליחות

אז ההסתברות לכשלון היא:

**20 מתוך 100 ← 0.2**

השפה הזו טבעית הרבה יותר ממונחים כמו “אירועים”, “מרחבי דגימה” או “התפלגות”.

מפתח לא צריך את כל זה.

הוא צריך להבין **מה קורה בדאטה**.

## הסתברות משקפת תדירות

אם 7 מתוך 10 משתמשים לוחצים על כפתור  $\leftarrow 0.7$

אם רק פעם אחת מתוך 50 מגיעה בקשה חריגה  $\leftarrow 0.02$

אם חצי מהתגובות למודל חיוביות  $\leftarrow 0.5$

המספרים הם תוצאה של **ספירה**, לא של מתמטיקה גבוהה.

## למה זה חשוב בעבודה עם AI?

כי מודלים לא "מנחשים".

הם **משווים תדירויות בעולם שהם ראו**, ומייצרים תחזיות בהתאם.

לדוגמה:

אם מודל טקסט ראה שב-90 אחוז מהמקרים המילה "טוב" מגיעה

אחרי המילה "היה"

הוא ייטה להשלים "היה טוב".

זה לא חוכמה.

זה **סטטיסטיקה פשוטה**.

**בלי תדירות – אין למידה**

כדי שמודל ילמד דפוסים, הוא חייב לראות אותם חוזרים שוב ושוב. אם הדאטה נדיר, מפוזר או בלתי עקבי המודל לא ילמד כלום, גם אם הוא גדול וחכם. זאת ההבנה הראשונה שאתה חייב כדי לדבר "הסתברות של מודל":

לא מספרים, לא נוסחאות.  
רק כמה פעמים משהו קורה.

## איך מייצגים תרחישים בצורה פשוטה

אחרי שמבינים שהסתברות היא בעצם תדירות, מגיעה השאלה הפרקטית:

## איך מייצגים תרחיש בצורה שהמודל מסוגל להבין וללמוד ממנו?

מבחינת המודל, "תרחיש" הוא פשוט רגע שבו משהו קורה. הוא לא יודע אם זה "משתמש לוחץ על כפתור", "מייל שמגיע", או "משפט שמסתיים".

מה שהוא צריך זה ייצוג מספרי שנותן תמונה ברורה של מה קורה ומה קרה בעבר.

## איך אנחנו מייצגים תרחישים?

בדרך כלל בצורה הכי פשוטה שיש:

**ספירה של כמה פעמים כל דבר קורה.**

זה יכול להיות בטבלה, ברשימה, או במערך של NumPy.  
העיקרון תמיד זהה:  
המודל לומד מתוך **תדירויות**.

לדוגמה, ניקח מערכת שמזהה האם הודעה היא ספאם.  
נניח שאנחנו סופרים כמה פעמים מילה מסוימת הופיעה בהודעות  
שסומנו כספאם או כלגיטימיות:

מילה	מופעים בספאם	מופעים בהודעות רגילות
free	42	3
urgent	17	1
hello	2	58

הטבלה הזו מספרת למודל הרבה יותר ממה שנדמה:

- המילה free כמעט תמיד מופיעה בספאם
  - המילה hello מופיעה בעיקר בהודעות רגילות
  - המילה urgent מוטה חזק לספאם, אבל לא ב-100 אחוז
- בלי להבין הסתברות – פשוט מסתכלים על המספרים ורואים את התמונה.

## תרחיש = הקשר

אבל לא רק “מה מופיע”, אלא גם “איפה זה מופיע”.  
למשל, אם משתמש לוחץ על כפתור רק כשהוא מגיע מדפדפן מסוים, זה תרחיש חוזר.  
אם בקשות מסוימות ל-API כושלות בעיקר בלילה, זה תרחיש חוזר.  
אם המילה “בעיה” מופיעה 80 אחוז מהפעמים לפני המילה “דחוף”, זה תרחיש חוזר.  
הסתברות היא כלי שמחבר אותך לאינטואיציה הזו.

## למה זה חשוב בעולם האמיתי?

כי כשאנחנו מפתחים מערכת ML או אפילו כשאנחנו מנתחים ביצועים של מודלים גדולים אנחנו תמיד עובדים סביב תרחישים:

- “מה הסיכוי שהתגובה הזו נכונה?”
- “מה ההסתברות שהמשתמש ינטוש?”
- “כמה פעמים זה קורה ביחס לכלל המקרים?”
- “האם זה דפוס או רעש?”

וכשמבינים איך לייצג תרחישים פשוטים גם הדברים המורכבים יותר נהיים הרבה יותר ברורים.

## דוגמאות: למה דברים קורים יותר או פחות

ברגע שמבינים שהסתברות היא בעצם תדירות, אפשר להתחיל לראות דפוסים כמעט בכל מערכת.

ולפעמים הדפוסים האלה מסבירים לנו דברים שנראים “מוזרים” מבחוץ, אבל הגיוניים לגמרי מבפנים.

### דוגמה 1 – למה מודל טועה שוב ושוב באותה נקודה

נניח שיש לך מערכת שממליצה על כתבות.

בכל פעם שמתמש לוחץ על כתבה בנושא “כסף”, המודל מזהה את זה כתכונה חיובית.

אבל בפועל, המשתמש לוחץ על כתבות כאלה רק בימי ראשון בבוקר.

כאן יש דפוס:

**ימי ראשון ← סיכוי גבוה יותר ללחיצה.**

מי שלא מסתכל על תדירויות מפספס את זה.

המודל, לעומת זאת, רואה שזה “קורה הרבה בימי ראשון”, ולכן הוא לומד לקשר בין שני הדברים – גם אם בכלל לא התכוונת שזה יהיה פיצ'ר.

### דוגמה 2 – למה המילה free נחשבת מסוכנת בספאם

הסתכלנו על טבלה עם ספירות, אבל בוא נראה את זה אינטואיטיבית:

**אם מילה מסוימת מופיעה 14 פעמים בספאם ופעם אחת בלבד**

בהודעה רגילה

אין צורך בשום נוסחה.

גם בן אדם היה מסמן את זה כמחשוד.

זו הסתברות פשוטה:

זה קורה כאן הרבה, וזה כמעט לא קורה שם.

מודלים לומדים בדיוק כך.

**דוגמה 3 – למה מערכת רמזורים “מתחרפנת” בשעות מסוימות**

אם רואים שבדרך כלל מגיעות 300 מכוניות בשעה,

ופתאום ברצף של 5 דקות מגיעות 200 מכוניות

זה דפוס שמערכת בקרה חייבת ללמוד ממנו.

במילים אחרות:

**כשמשו קורה הרבה יותר מהרגיל – ההסתברות שלו גבוהה**

**באותו רגע.**

וכשמשו קורה פחות מהרגיל – ההסתברות שלו יורדת.

זה נשמע טריוויאלי, אבל זו הליבה של רוב המערכות המזהות

אנומליות.

**דוגמה 4 – למה יש יותר משתמשים חוזרים מאשר חדשים**

נניח שבאתר מסוים:

• 25 אחוז מהמבקרים הם חדשים

מאת: תומר קדם

## • 75 אחוז חוזרים

מה המודל ילמד?

שהסבירות להתנהגות של “משתמש חוזר” גבוהה פי שלושה מזו של “משתמש חדש”.

במודלים התנהגותיים, הדפוס הזה משפיע על כל תחזית.

## דוגמה 5 – למה מודל NLP נוטה להשלים מילים מסוימות

אם 80 אחוז מהפעמים שבהן מופיעה המילה מזג מגיע אחריה אוויר

המודל ילמד שזה “המשך טבעי”.

הוא לא עושה קסמים.

הוא פשוט סופר תדירויות.

## מה המשותף לכל הדוגמאות?

העיקרון הבסיסי חוזר שוב ושוב:

**דברים שקורים הרבה ← מקבלים משקל גבוה**

**דברים שקורים מעט ← מקבלים משקל נמוך**

זה הכול.

מזה נבנים מודלים שיכולים לנתח טקסט, לזהות תמונות, להמליץ על תכנים או לנבא התנהגות.



וברגע שאתה כמפתח מתחיל לראות את העולם דרך תדירויות, המודלים מפסיקים להיות קופסאות שחורות, והופכים לכלים שאתה מבין באמת.

## חיבור ראשוני לעולם של סיווגים (Spam/Ham)

עכשיו שהבנו שתדירות היא הלב של הסתברות, אפשר לראות איך זה יושב בדיוק על אחד היישומים הכי בסיסיים בעולם ה-ML: **סיווג (Classification).**

ברוב היישומים הראשונים של מתכנתים בעולמות ה-AI, סיווג הוא הצעד הטבעי:

האם זה ספאם או לא?

האם הלקוח יעזוב או יישאר?

האם התמונה מכילה חתול או כלב?

האם המשפט חיובי או שלילי?

למרות שהמשימות נראות שונות לגמרי, יש להן מבנה זהה:

**המודל מקבל משהו, ומנסה לשייך אותו לאחת משתי קבוצות (או יותר).**

וכמו שראינו בפרקים הקודמים – השיוך הזה מבוסס על תדירויות.

**למה דווקא Spam/Ham הוא הדוגמה הכי קלאסית?**

כי זה מקרה פשוט להבנה והוא מגלה בדיוק איך המודל “חושב”.

אם בתיבת הדואר שלנו:

- 1,200 הודעות מסומנות כספאם

- 800 הודעות מסומנות כרגילות

אז כבר לפני שמסתכלים על המילים בתוך ההודעות, יש עובדה חשובה:

## **הסתברות בסיסית לספאם גבוהה יותר מהסתברות בסיסית להודעה רגילה.**

כלומר: מתוך כלל העולם שהמודל רואה יותר דברים נופלים ל"ספאם".

וכשמודל ניגש להודעה חדשה, הוא שואל את עצמו בשקט: "במה זה דומה יותר למה שכבר ראיתי?"

איך המודל משתמש בתדירויות של מילים?

נניח שיש לנו טבלה כמו זו:

מילה	מופעים בספאם	מופעים רגילים
free	42	3
urgent	17	1
hello	2	58

המודל רואה:

free ← כמעט תמיד בספאם

hello ← בדרך כלל בהודעות רגילות

אין כאן נוסחאות מתוחכמות.  
זו פשוט ספירה.

ברוב המקרים, סיווגים ראשוניים עובדים על מבנה בסיסי כזה:  
הסתברות למילים מסוימות בתוך קבוצות שונות ← תדירות ← החלטה.

## מה זה אומר למפתח בפועל?

כשמפתחים מערכות סיווג, כדאי לשים לב ל-3 שאלות פשוטות:

### 1. מה מופיע הרבה בקבוצה מסוימת?

### 2. מה כמעט לא מופיע?

### 3. אילו דברים משותפים לשתי הקבוצות?

רק מהשאלות האלה אפשר לפעמים לשפר מודל פי כמה עוד לפני שנוגעים בקוד שלו.

## למה זה חשוב לקראת הפרק הבא?

כי בעולם האמיתי, מודלים לא מסתפקים בידע על “כמה פעמים מילה מופיעה”.

הם רוצים לדעת:

- מה הסיכוי למשהו **בהינתן** משהו אחר?
- מה המשמעות של מילה בתוך הקשר?
- איך מחשבים הסתברות כשיש שתי קבוצות עם מאפיינים שונים?

## ופה בדיוק נכנסת **הסתברות מותנית**.

זה אחד הכלים החזקים ביותר בהבנת דפוסים, ולמעשה הבסיס של משפט בייס – שהוא בלב של כל משימות הסיווג.

## פרק 4 – הסתברות מותנית ובייס – הגרסה אנושית

### מה זה באמת הסתברות מותנית

בפרק הקודם דיברנו על שאלה פשוטה:

**“כמה פעמים משהו קורה מתוך כל המקרים?”**

זו הסתברות בסיסית, והיא חשובה.

אבל בחיים האמיתיים – ובטח בעולם ה-AI – זו לא השאלה שאנחנו באמת רוצים לשאול.

השאלה האמיתית היא אחרת לגמרי:

**“כמה פעמים משהו קורה בתוך קבוצה מסוימת?”**

זו הסתברות מותנית.

וברגע שמבינים אותה, העולם משתנה.

### דוגמה יומיומית

אם אתה שואל:

**“מה הסיכוי שירד היום גשם?”**

זו שאלה כללית.

אבל אם אתה שואל:

**“מה הסיכוי שירד היום גשם אם העננים כהים כבר שעה?”**

זו שאלה אחרת לחלוטין.

ברגע שיש **הקשר**, הסיכוי משתנה.

אנחנו כבר לא בודקים את הכול

אלא **מצב חלקי**, תת-קבוצה.

וזה בדיוק מהות ההסתברות המותנית.

**וזה קורה כל הזמן בעולם של מודלים**

מודל לא שואל:

“מה ההסתברות שמהו הוא ספאם?”

אלא:

“מה ההסתברות שההודעה הזו **ספאם** בהינתן **שהיא מכילה את**

**המילה free?**”

או בעולם אחר:

“מה הסיכוי שלקוח יעזוב, בהינתן שזו כבר הפנייה השלישית

**שלו החודש?**”

או ב-NLP:

“מה הסיכוי שמילה מסוימת תגיע עכשיו, בהינתן שלושת

**המילים הקודמות?**”

שמת לב?

אין פה נוסחה.

יש הקשר.

## למה זה כל כך חשוב?

כי מודלים לא עובדים על תדירויות "עיוורות".  
הם לא מחפשים רק "מה קרה הרבה".  
הם מחפשים גם:

### "מה קורה הרבה כשדברים מסוימים כבר קרו?"

זו השפה האמיתית של דפוסים:

- אם X קרה, כמה פעמים גם Y קרה?
- אם מופיעה מילה מסוימת, מה המשמעות של זה בטקסט?
- אם משתמש מתנהג בצורה מסוימת, מה זה אומר על הצעד הבא שלו?

זו הסתברות שמדברת כמו מתכנת:  
פשוטה, הגיונית, מחוברת להקשר.

## וכאן מתחיל החיבור לבייס

בייס הוא לא נוסחה כבדה.  
הוא לא "מתמטיקה של סטטיסטיקאים".  
הוא הדרך לחבר בין:

**העולם הכללי** ← כמה משהו נפוץ באופן כללי

**העולם שבהקשר** ← כמה הוא נפוץ בתוך תת-קבוצה מסוימת

כדי להגיע לתשובה אחת:

**כמה סביר שזה נכון עכשיו.**

לפני שנגיע לבייס עצמו, צריך לבנות את השולחן שעליו הוא עומד: טבלאות פשוטות, מספרים קטנים, וספירה ישרה. וזה בדיוק מה שנעשה בחלק הבא.

## בייס דרך טבלאות, לא דרך נוסחאות

כדי להבין את הרעיון של בייס בצורה **אנושית**, צריך לשים בצד את הנוסחאות ואת כל ה  $P(X|Y)$ .

לפני הכול, בייס מתחיל מהסתכלות על **מספרים אמיתיים שמסודרים בטבלה**.

שם מסתתר כל הרעיון.

אבל לפני שנצלול לתוך הדוגמה, חשוב להבהיר משהו קטן שמונע הרבה בלבול:

**המילה בייס היא לא קיצור ולא ביטוי טכני.**

זה פשוט שמו של תומס בייס, כומר ומתמטיקאי שחי לפני כמה מאות שנים.

בייס הציג רעיון שהיה חדשני לזמנו:

כשיש לנו ידע קודם ומגיע מידע חדש, צריך **לשלב** ביניהם ולא להתעלם משום חלק.

זה הבסיס לכל מה שנקרא היום “חשיבה בייסיאנית”.

עכשיו ניגש לטבלה.



הטבלה מספרת שני דברים חשובים:

**מה קורה בעולם בכלל**

**ומה קורה בתוך ההקשר שמעניין אותנו עכשיו.**

נראה את זה בדוגמה הכי ברורה: זיהוי ספאם.

סרקנו מיילים וספרנו כמה פעמים מופיעה המילה “free”.

זה מה שקיבלנו:

מופע המילה "free"	ספאם	לא ספאם
מופיעה	42	3
לא מופיעה	1158	797

זו טבלה קטנה, אבל היא מספרת סיפור גדול.

**איך קוראים את הטבלה?**

כש free מופיעה:

- 42 מקרים היו ספאם
- 3 בלבד היו הודעות רגילות

כלומר:

**כש free בפנים – הקונטקסט מוטה חזק לכיוון ספאם.**

כש free לא מופיעה:

- 1158 היו ספאם

## • 797 היו רגילות

כאן התמונה הרבה פחות חדה.

הטבלה מציגה שני עולמות:

- העולם הכללי – כמה ספאם יש בכלל
  - העולם שבקונטקסט – מה קורה כשה free מופיעה או לא מופיעה
- ובייס פשוט מחבר בין שני העולמות האלה.

## מה בייס בעצם שואל?

הוא מסתכל על הטבלה ושואל:

בהינתן שאני רואה free

## איזה תרחיש הופיע יותר פעמים?

או בניסוח "אנושי":

מה יותר סביר עכשיו, בהתחשב במה שאני רואה מול העיניים?

## וזה עובד לא רק בטקסט

אותו עיקרון קורה גם בעולם של:

- זיהוי הונאות
- עזיבת לקוחות
- אנומליות במערכות
- זיהוי אובייקטים
- כל תחום שיש בו סיווג

בכל פעם שיש לך קונטקסט, ובתוכו דפוסים שונים  
טבלה כזו היא הכלי הכי חזק להבין מה קורה באמת.  
לפני שנראה "כמה זה סביר" בצורה אינטואיטיבית,  
נבין מה עומד מאחורי השאלה הזו.

## מה מסתתר מאחורי השאלה "כמה סביר שזה נכון?"

אחרי שמבינים איך נראית טבלת שכיחויות, מגיעה השאלה שהמודל  
באמת מנסה לענות עליה.  
זו לא השאלה "האם זה נכון" אלא השאלה "כמה זה סביר".  
זו נקודה עדינה אבל משמעותית.  
מודלים לא עובדים עם אמת מוחלטת.  
הם עובדים עם רמזים, דפוסים והקשרים שמעלים או מורידים את  
הסיכוי למשהו.

## איך מודל חושב על סבירות

כשמודל מקבל הודעה חדשה, הוא לא יודע מראש מה היא.  
הוא בודק מה הוא ראה בעבר.  
הוא מחפש תבניות שמופיעות הרבה בתוך קבוצות מסוימות.

אם free הופיעה בעיקר בספאם  
ואותה המילה מופיעה עכשיו  
המודל מקבל רמז לכיוון מסוים.

אם hello מופיעה בעיקר בהודעות רגילות  
ומופיעה גם כאן  
זה רמז לכיוון אחר.

המודל שואל את עצמו שאלות כמו

- האם הדפוס הזה מתאים למה שקורה בדרך כלל בקבוצה מסוימת?
  - האם מה שאני רואה עכשיו מאפיין מקרים שכבר הכרתי?
  - מה קרה בעבר במצבים דומים?
- זו חשיבה הסתברותית שמבוססת על ניסיון, לא על הגדרה מתמטית.

## למה זו לא החלטה דיכוטומית

המוח שלנו אוהב שחור ולבן.  
מודלים עובדים באפור.  
הם לא אומרים "זו הודעה רגילה" או "זו הודעת ספאם".  
הם אומרים

## בהקשר הזה יש יותר ראיות לכאן מאשר לכאן.

זה מה שהשאלה "כמה סביר שזה נכון" באמת אומרת.

## איך זה מתחבר לעבודה של מפתח

כשאתה מבין שהמודל בודק סבירות ולא אמת מוחלטת אתה פתאום קורא תוצאות בצורה אחרת. אחוזים ומדדים מפסיקים להיות סתם מספרים והופכים לרמזים למבנה הדאטה.

הבנה של סבירות עוזרת לך

- לזהות הטיות
- להבין למה מודל טעה
- לשפר דאטה במקום לשפר קוד
- לבנות לוגיקה עסקית שמתחשבת בסיכון ולא רק בתוצאה

במילים אחרות

לדעת לשאול "כמה סביר" היא מיומנות הנדסית לכל דבר.

בחלק הבא נראה איך כל זה נראה בקוד אמיתי וכמה פשוט אפשר להפוך דפוס מסובך לכמה שורות פייתון.

## הדגמה קטנה בקוד שתגרום לזה להתחבר

אחרי שעברנו דרך הרעיון של הסתברות מותנית והבנו איך מודל מחבר בין הקשר לבין תמונת העולם הרחבה, הגיע הזמן לראות את זה קורה בפועל.

לא נוסחאות.

לא מודל כבד.

רק טבלה, קצת ספירה, וכמה שורות פייתון.

נשתמש בדוגמה שכבר ראינו: המילה free בהודעות ספאם או רגילות.

הטבלה שלנו נראית כך:

מופע המילה "free"	ספאם	לא ספאם
מופיעה	42	3
לא מופיעה	1158	797

בוא נתרגם את הטבלה לפייתון ונראה מה המודל היה "מרגיש" כשהוא מקבל הודעה עם free.

```
import numpy as np

# טבלת שכיחויות
spam_with_free = 42
ham_with_free = 3

spam_without_free = 1158
ham_without_free = 797
```

```
# סך הכל הודעות בכל קטגוריה
total_spam = spam_with_free + spam_without_free
total_ham = ham_with_free + ham_without_free

# הסתברויות בסיסיות
p_spam = total_spam / (total_spam + total_ham)
p_ham = total_ham / (total_spam + total_ham)

# הסתברות מותנית לפי תדירות
p_free_given_spam = spam_with_free / total_spam
p_free_given_ham = ham_with_free / total_ham

print("הסתברות בסיסית לספאם:", round(p_spam, 3))
print("הסתברות בסיסית להודעה רגילה:", round(p_ham, 3))

print("הסתברות free בתוך ספאם:", round(p_free_given_spam, 3))
print("הסתברות free בהודעות רגילות:", round(p_free_given_ham, 3))
```

מה קורה כאן בפועל?

## 1. מחשבים כמה ספאם יש בכלל

לא תמיד צריך את זה, אבל זה נותן תמונת רקע.  
אם יש הרבה ספאם בעולם – הודעה חדשה גם נבחנת בתוך  
רגישות גבוהה יותר.

## 2. בודקים כמה פעמים free מופיעה בכל קבוצה

זה לב הסיפור.  
המודל לא שואל "כמה פעמים free מופיעה".  
הוא שואל "כמה פעמים free מופיעה בתוך ספאם" ו"בתוך לא  
ספאם".

### 3. משווים בין שתי ההסתברויות

אם free מופיעה יותר בספאם מאשר בהודעות רגילות המודל יטה לכיוון ספאם גם אם זה לא ודאות מוחלטת. זה בדיוק "כמה סביר שזה נכון".

### 4. הסיפור מאחורי המספרים

אם תפעיל את הקוד, תראה שמשהו בולט מיד: הסתברות להופעת free בתוך ספאם גבוהה בצורה קיצונית והיא נמוכה מאוד בתוך הודעות רגילות.

במילים אחרות  
אם free מופיעה – הסיכוי לספאם עולה משמעותית.  
זו לא נוסחה.  
זו הבנה דרך דפוסים.  
וזה הלב של בייס:  
שילוב בין תדירות כללית לבין תדירות בתוך הקשר.

### סיכום

הסתברות מותנית ובייס הם לא חומרים תיאורטיים כבדיים, אלא כלים מעשיים שכל מפתח צריך להבין ברמה אנושית. כשאתה מבין



- כמה דברים קורים בתוך הקשר
- איך מודל בודק סבירות
- איך טבלה פשוטה מספרת סיפור

אתה כבר לא מפחד מהסטטיסטיקה שמאחורי מודלים.  
אתה קורא אותה כמו שפה.

פרק זה מספק לך את היכולת להסתכל על מודלים מתוך עולם של  
סבירות ולא מתוך עולם של ניחושים.  
וזו מיומנות שתשרת אותך בכל מודל, בכל קוד, בכל מערכת.

## פרק 5 – וקטורים – הלב של כל מודל

### איך מציגים אובייקט כסדרה של מספרים

אם יש משהו אחד שבאמת משנה את הדרך שבה מפתח מבין AI, זה הרעיון שכל אובייקט בעולם – טקסט, תמונה, משתמש, מוצר, אירוע – יכול להפוך לוקטור.

זאת אומרת: **רשימה מסודרת של מספרים.**

כדי להבין מודלים מודרניים, צריך להבין את המשפט הבא:

**AI לא עובד על תוכן. הוא עובד על מספרים שמייצגים תוכן.**

וזה בדיוק מה שנותן לוקטורים את הכוח שלהם.

### מה זה בכלל וקטור?

זה פשוט מאוד.

וקטור הוא רשימה של מספרים.

לא מטריצה, לא אובייקט מסובך.

רשימה.

לדוגמה:

`[0.2, 1.7, 3.4]`

או

`[0.01, 0.43, -0.12, 0.77, 0.91]`

וקטור יכול להיות קצר או ארוך מאוד.

במודלים גדולים, האורך נמדד לפעמים באלפי ערכים.

### למה זה מעניין אותנו?

כי וקטור הוא הדרך שבה מודל **מבין** דברים.  
כשאומרים למודל "זה משפט חיובי" או "זה חתול",  
אי אפשר לתת לו תמונה או טקסט ישירות.  
צריך לתת לו **מספרים**.

והמספרים האלו הם וקטור.

## איך אובייקט הופך לוקטור?

אפשר לחשוב על זה כעל תרגום.  
אובייקט מגיע מבחוץ – תמונה, מילה, משתמש, מה שלא יהיה  
והמודל צריך דרך לייצג אותו בצורה מתמטית.

הרעיון פשוט:

לוקחים את התכונות החשובות וממירים אותן למספרים.

דוגמאות בסיסיות:

- צבע של פיקסל הופך לערך בין 0 ל-255
  - מילה הופכת לרשימת מספרים שמייצגים משמעות
  - משתמש הופך לקבלת וקטור של התנהגויות
  - מוצר הופך לערכים שמייצגים תכונות שנלמדו מדאטה
- עכשיו נסביר כל דוגמה בצורה פשוטה וברורה.

## צבע של פיקסל הופך למספר

בכל תמונה יש פיקסלים.  
למודל אין מושג מה זה "אדום" או "כחול".  
אז כל צבע בודד מקבל **מספר מ 0 עד 255**.  
0 זה כהה.

255 זה בהיר.  
וביניהם כל הדרגות של הצבע.

כל פיקסל = מספר.  
ככה המודל "רואה" תמונה.

## מילה הופכת לרשימת מספרים שמייצגים משמעות

כאן זה הכי מבלבל אנשים.  
אין "מילים" בתוך מודל.  
אין אותיות.  
הכול הופך ל **וקטור** – רשימת מספרים.  
למשל:

"המבורגר" הופך לרשימה כגון:  
[0.12, 0.83, 1.44, ...]

אבל הרשימה הזו לא מייצגת "אותיות".

היא מייצגת **משמעות**:

אוכל, טעם, מסעדה, שומן, טקסט, הקשר.

המספרים נוצרים מתהליך אימון, לא מניחוש ידני.

כל מילה = רשימת מספרים.

## משתמש הופך לוקטור של התנהגויות

המודל לא עובד עם "פרופיל משתמש".

הוא עובד עם **מספרים** שמייצגים את הדפוסים של אותו אדם:

- כמה פעמים הוא נכנס לאפליקציה
- באיזה שעות
- באיזה מסכים הוא משתמש
- מה משך הפעולה

כל זה הופך לשורה של מספרים, כגון:

[3.1, 0.8, 12.4, 0.02 ...]

זה לא "מי הוא" – זה **איך הוא מתנהג** מבחינת דפוס.

## מוצר הופך לערכים שמייצגים תכונות

אם אתה מוכר מוצרים,

למודל לא משנה "איך המוצר נראה".

הוא צריך מספרים שייצגו:

- מחיר
- קטגוריה
- פופולריות
- תדירות קנייה

- קשר למוצרים אחרים

המודל לוקח את כל זה והופך את המוצר לוקטור כגון:

[0.51, 2.3, 0.12, 15.4 ...]

ככה הוא משווה בין מוצרים.

## מה המשותף בין כל הדוגמאות?

הכול הופך למספרים.

לא כי מפתחים אוהבים מתמטיקה,

אלא כי זו **השפה שהמודל יודע לדבר**.

ברגע שאובייקט בעולם מקבל ייצוג מספרי,

המודל יכול להשוות, למדוד, לזהות דפוסים ולהפיק תחזיות.

הוקטור הוא בעצם **קלף זיכרון** של המודל.

הוא מספר למודל מה הוא ראה.

## למה זו נקודת מפנה בהבנה של AI?

כי ברגע שמבינים שכל אובייקט הופך לוקטור

מבינים גם:

- איך מודלים משווים בין דברים
- איך הם מחשבים דמיון
- למה דברים שונים מתבלבלים לפעמים
- למה שינוי קטן בנתונים יכול לייצר שינוי גדול בתוצאה
- ולמה כל כך חשוב שהוקטורים יהיו נקיים ומייצגים

וקטור הוא לא רק “רשימת מספרים”.

הוא **הזהות המתמטית** של כל דבר שהמודל עובד איתו.

בחלק הבא נבין למה כמעט כל AI בעולם, קטן וגדול, עובד בדיוק עם המבנה הזה.

## למה כמעט כל AI עובד על וקטורים

אחרי שהבנו שוקטור הוא פשוט רשימה של מספרים שמייצגת אובייקט,

עולה שאלה טבעית:

### למה כל עולם ה-AI עובד דווקא עם וקטורים?

יש לזה שלוש סיבות עמוקות שמחברות בין מתמטיקה, יעילות, והדרך שבה מודלים לומדים דפוסים.

#### 1. וקטורים מאפשרים למודל למדוד דמיון

זה הרעיון הכי חשוב.

כדי שמודל יבין ש:

- שני משפטים דומים
- שתי תמונות מציגות אותו אובייקט
- שני משתמשים מתנהגים בצורה קרובה

הוא צריך דרך למדוד קרבה.

וקטור מאפשר לעשות בדיוק את זה

על ידי חישוב של:

- מרחק
- זווית
- דמיון

בין רשימות של מספרים.



לדוגמה, אם שני משפטים מיוצגים כוקטורים קרובים המודל יפרש אותם כבעלי משמעות דומה. זה הבסיס של NLP מודרני.

## 2. וקטורים מאפשרים לבצע חישובים מהירים מאוד

המתמטיקה על וקטורים ממוצעים, סכומים, מכפלות, מרחקים היא מהירה במיוחד לחישוב על חומרה מודרנית. זה חשוב כי מודלים מבצעים מיליארדי פעולות כאלה. וקטורים עובדים מצוין עם GPU, TPU וכל מנוע שמקבל עבודה על מספרים. זו אחת הסיבות שמודלים גדולים בכלל אפשריים.

## 3. וקטורים מאפשרים למודל "ללמוד" תכונות בעצמו

במערכות ישנות היינו צריכים לבחור תכונות ידנית. בינה מלאכותית מודרנית לומדת תכונות לבד ומכניסה אותן לתוך הוקטור. זה אומר שהוקטור מייצג, לא רק מידע גולמי אלא גם משמעות שהמודל למד מתוך דוגמאות.

המילה "כלב" תקבל וקטור שמתקרב  
למילים כמו "גור", "לנבוח", "חיה"  
ומתרחק ממילים כמו "חלון", "פינגווין" או "שולחן".  
זה לא קסם.

זה פשוט מבנה שמאפשר למודל לקלוט הקשרים מתוך מספרים.

#### 4. וקטור הוא מבנה אוניברסלי

תמונה, משפט, משתמש, מוצר, אירוע  
הכול נהיה וקטור.  
ואז המודל לא צריך להתאים את עצמו לכל סוג של מידע.  
הוא תמיד עובד על אותו פורמט:

#### מספרים.

בגלל זה מודלים יכולים לעבור בין

טקסט

תמונה

אודיו

קוד

וגם בין דאטה מובנה ולא מובנה.

הבסיס תמיד אותו בסיס: וקטור.

## איך טקסט, משתמש או תמונה הופכים לוקטור

עכשיו כשהרעיון של וקטורים ברור, מגיעה השאלה המעשית: איך אובייקטים אמיתיים, כאלה שאנחנו רואים במערכות יום יום, הופכים לרשימות מספרים שמודל יכול לעבוד איתן? נפרק את זה לשלושה סוגים נפוצים של מידע.

### טקסט ← וקטור של משמעות

זו אחת ההמרות הכי מעניינות.

בתחילת הדרך

היינו ממירים כל מילה למספר לפי המיקום שלה במילון. זה לא עבד טוב, כי מילים שונות לחלוטין קיבלו מספרים בלי קשר למשמעות שלהן.

במודלים מודרניים

כל מילה וכל רצף מילים מקבל וקטור שמייצג את המשמעות שלו. שתי מילים שמופיעות בהקשרים דומים מקבלות וקטורים קרובים.

לדוגמה, המילים

לקוח, משתמש, צרכן

יהיו קרובות יחסית.

אבל המילה מנורה תהיה רחוקה מהן.

המודל לא "מבין" עברית.  
הוא רואה מספרים שמספרים לו מי קרוב למי.

## משתמש ← וקטור של התנהגות

גם משתמשים הופכים לוקטורים.

הוקטור שלהם יכול להכיל למשל

- כמות ביקורים
- סוגי פעולות
- זמנים אופייניים
- קטגוריות שנצפו
- אורך סשנים
- דפוסי מעבר בין מסכים

המערכת לא צריכה לדעת מי המשתמש בפועל.  
היא רק צריכה וקטור שמייצג את ההתנהגות שלו.

שני משתמשים שונים לחלוטין  
יכולים לקבל וקטורים דומים  
אם הם מתנהגים בדרך דומה.  
וזה בדיוק מה שמאפשר המלצות חכמות.

## תמונה ← וקטור של מאפיינים חזותיים

בכל תמונה יש אלפי פיקסלים.  
אי אפשר לעבוד ישירות עם כל הפיקסלים.  
לכן המודל מעבד את התמונה בשכבות ומייצר וקטור מאפיינים.

הוקטור הזה יכול לייצג

- קווים
- צבעים
- טקסטורות
- צורות
- חלקים של אובייקטים
- דברים שלא תמיד בני אדם שמים לב אליהם

למשל

כל התמונות של חתולים תקבלנה וקטורים קרובים  
גם אם הצבע שונה  
גם אם הזווית שונה  
גם אם הרקע שונה.

שוב

המחשב לא "רואה חתול"  
הוא רואה וקטור שמאפיין תבנית שמתאימה לחתול.

**למה הכול עובד אותו דבר?**

זה היופי.

כיוון שהכול מתורגם לוקטורים

מודלים יכולים לעבוד על סוגים שונים של מידע

באותה לוגיקה.

הם לוקחים וקטורים

משווים

מודדים מרחקים

מזהים דפוסים

לומדים הקשרים

ונותנים תחזיות.

הבסיס תמיד אותו בסיס.

## פרק 6 – נורמה ומרחק – מודדים את העולם

### מה זה “אורך” של וקטור

אם פרק 5 עסק בשאלה איך מייצגים דברים כוקטורים, הפרק הזה עוסק בשאלה הבאה:

**איך מודל יודע כמה הוקטור הזה “גדול”, “חזק” או “קיצוני”?**  
וזו בדיוק הנורמה.

### מהי נורמה?

נורמה היא מספר אחד שמסכם את “הגודל” של הוקטור. אפשר לחשוב עליה כעל אורך. אם תצייר וקטור על לוח כקווים, הנורמה היא האורך של הקו. בפייתון, אם יש לנו וקטור:

```
v = [3, 4]
```

הנורמה שלו היא

5

כי זה האורך של הקו מהנקודה (0,0) לנקודה (3,4).

אצל מודלים הרעיון הזה

רק שהוקטורים ארוכים בהרבה

ולכן האורך שלהם מספר משהו חשוב.

**למה אכפת למודל מהאורך?**

כי נורמה מציגה למודל

כמה חזק או קיצוני אובייקט מסוים ביחס לאחרים.

לדוגמה

- משתמש עם פעילות ענקית יקבל וקטור ארוך יותר
- תמונה שיש בה המון פרטים לפעמים יוצרת וקטור ארוך יותר
- מילה שמופיעה תמיד בסביבה "חזקה" תקבל וקטור שאורכו שונה ממילים פחות משמעותיות
- בנתונים שמתנהגים בצורה קופצנית ולא עקבית, חלק מהוקטורים קופצים באורך בצורה לא יציבה הנורמה הופכת את העולם המרובה ממדים למספר אחד שקל להשוות.

## למה "אורך" עוזר למערכת להבין דברים?

כי כשאתה משווה אורכים

קבלת אינטואיציה על "כמה הדבר הזה גדול יחסית לאחרים".

לא צריך מתמטיקה כבדה.

צריך את ההבנה הפשוטה הזו:

## נורמה עוזרת למודל להבין את עוצמת המידע.

בפרקטיקה

הנורמה משמשת כדי

- לנרמל דאטה
- להבין חריגות
- להשליך נקודות שנמצאות "רחוק מדי"



- לקבוע גבולות במערכות זיהוי אנומליות
  - להכין את הוקטורים להשוואה אמיתית ובפרקים הבאים של הספר
- היא תהיה אחת המילים שיחזרו הכי הרבה.

## איך מודלים מודדים כמה דברים דומים או שונים

אחרי שהבנו מהו האורך של וקטור, מגיעה שאלה בסיסית שכל מודל חייב להתמודד איתה:

## איך יודעים אם שני אובייקטים דומים או שונים, כשהכול מיוצג כוקטורים?

בגלל שכל מודל עובד על רשימות מספרים, הוא צריך דרך להשוות בין שתי נקודות במרחב ולתרגם את ההשוואה הזאת ל"קרבה" או "מרחק".

## הצעד הראשון: מרחק

המדד הכי פשוט לדמיון הוא **מרחק בין וקטורים**. העיקרון מאוד אינטואיטיבי:

- מרחק קטן ← הוקטורים **דומים**
- מרחק גדול ← הוקטורים **שונים**

דוגמה:

```
import numpy as np

v1 = np.array([1, 2])
v2 = np.array([2, 3])

distance = np.linalg.norm(v1 - v2)
print(distance)
```

המספר שמתקבל הוא המרחק הגיאומטרי ביניהם.

## למה מרחק הוא כלי שימושי?

כי הוא נותן למודל מדד ישיר לשאלה

**"כמה שני דברים רחוקים אחד מהשני?"**

וזו שימושי במיוחד ב:

- מערכות המלצה
- זיהוי תמונות
- חיפוש דמיון בין מסמכים
- איתור חריגות
- קיבוץ נתונים דומים

מודל מקבל שתי רשימות של מספרים, מודד את המרחק ביניהן, ומסיק אם מדובר באותו סוג של מידע או לא.

## אבל מרחק הוא רק חלק מהסיפור

כאן מגיעה הנקודה הקריטית:

**מרחק לא יודע להבין משמעות.**

שני משפטים יכולים להיות שונים לגמרי מבחינת המילים, אבל מאוד דומים מבחינת המשמעות. במצב כזה המרחק ביניהם יכול להיות גדול מדי, למרות שהם "אומרים" את אותו הדבר.

מצד שני, שני משפטים שמכילים מילים דומות יכולים להיות קרובים במרחב המספרי למרות שהמשמעות שלהם שונה לחלוטין. במילים פשוטות:

**מרחק מודד שונות גיאומטרית, לא שונות רעיונית. למה זה בעייתי במיוחד בטקסט?**

כי טקסט מבוסס על **משמעות, הקשר וכוונה**.

שתי מילים יכולות להיראות קרובות מאוד מבחינה מספרית, אבל להיות שונות לגמרי בתוכן.

ודרך אחרת – מילים שונות יכולות לשאת אותה משמעות. לכן מודלים צריכים יותר מאשר מרחק.

הם צריכים כלי שמזהה **כיוון משותף** בין וקטורים ולא רק את "הגודל" של ההפרדה ביניהם.

זה בדיוק מה שמוביל אותנו לשיטה שחייבים להכיר בעולם ה-NLP: מדידת זווית בין וקטורים, או במילים פשוטות – **דמיון קוסינוס**.

בחלק הבא נבין למה מרחק לבדו לא מספיק, ואיך זווית בין וקטורים נותנת למודל הבנה הרבה יותר מדויקת של משמעות.

## למה מרחק לא מספיק בניתוח טקסט

מרחק הוא כלי שימושי למדידת דמיון, אבל בעולם של טקסט הוא פשוט לא מספק.

כדי להבין למה, צריך לזכור ש**טקסט הוא משמעות**, ולא רק אוסף של מילים.

כשמודל מקבל משפט, הוא צריך להבין לא רק אילו מילים מופיעות בו, אלא גם מה הן מנסות להגיד.

ופה מרחק כבר מתחיל לחרוק.

### דוגמה שממחישה את הבעיה

שני המשפטים

**"אני מאחר לעבודה"**

|

**"אני מתקשה להגיע בזמן"**

משתמשים במילים שונות, אבל המשמעות שלהם כמעט זהה. לעומת זאת, המשפטים

**"אני אוהב קפה"**

|

**"אני אוהב לשרוף גשרים"**

משתפים חלק מהמילים, אבל המשמעויות שלהם רחוקות לגמרי. מרחק גיאומטרי לא יודע לעשות את ההבחנה הזאת.

הוא רואה **מספרים**, לא רעיונות.

זה אומר שוקטורים של שני משפטים בעלי משמעות דומה יכולים להיות רחוקים מדי,

ושניים בעלי משמעות שונה יכולים להיות קרובים מדי.  
במילים אחרות:

**מרחק מתייחס רק לפער במספרים, לא לתוכן שמסתתר מאחוריהם.**

**למה זה קורה?**

כי מרחק עונה על שאלה אחת בלבד:

**"כמה הוקטורים מתרחקים אחד מהשני במרחב?"**

אבל הוא לא עונה על שאלות כמו:

- האם הכיוון שלהם דומה?
- האם הם מצביעים על אותו רעיון?
- האם הם מתארים אותה כוונה?

בטקסט זה קריטי, כי שני משפטים שנראים שונה מבחינה צורתית יכולים להיות כמעט זהים מבחינה רעיונית.

**מודלים צריכים כלי נוסף שמבין כיוון**

כשמדברים על טקסט, השאלה החשובה היא לא רק "כמה רחוק",  
אלא גם

**"לאיזה כיוון הוקטורים מצביעים?"**

כיוון - מספר למודל, האם שני משפטים "מסתובבים סביב אותו רעיון", גם אם המרחק ביניהם גדול.

זו הסיבה שמערכות NLP ותהליכי הבנה סמנטית כמעט תמיד משתמשים במדידת זווית בין וקטורים ולא רק במרחק. המדד הזה נקרא **דמיון קוסינוס**, והוא אחד הכלים החשובים ביותר להבנת משמעות בשפה. בחלק הבא נעמיק בכלי הזה ונראה איך הוא פותר את כל הבעיות שמרחק לא מצליח לפתור.

## פרק 7 – זווית ודמיון קוסינוס

### דמיון ככיוון ולא כמרחק

בפרק הקודם ראינו שמרחק בין וקטורים הוא כלי שימושי, אבל יש לו מגבלות ברורות.

בעיקר בעולם של טקסט, שבו המשמעות לא תמיד יושבת על "עד כמה שני משפטים רחוקים אחד מהשני", אלא על השאלה **לאיזה כיוון הם מצביעים**.

כאן נכנס הרעיון החשוב הזה:

**דמיון אמיתי בין שני וקטורים נקבע לפי הכיוון שלהם, לא רק לפי המרחק הגיאומטרי ביניהם.**

### למה הכיוון חשוב יותר מהמרחק?

תחשוב על שני משפטים שונים מבחינת המילים, אבל דומים מאוד ברעיון:

"אני מאחר לעבודה"

ו

"אני מתקשה להגיע בזמן"

המשמעות שלהם זורמת באותו כיוון.

גם אם הוקטורים רחוקים יחסית, זווית קטנה ביניהם מספרת שהם "פונים" לאותו כיוון.

לעומת זאת:

"אני אוהב קפה"

ו

"אני אוהב לשרוף גשרים"

מרחק פשוט יגיד ששני המשפטים קרובים, כי הם חולקים מילים דומות.

אבל הכיוון שלהם שונה לגמרי.

הם לא מתארים את אותו רעיון.

זו בדיוק הסיבה שמודלים מודרניים מסתמכים הרבה יותר על **זווית** ופחות על מרחק.

## מה בעצם מודדים בזווית?

כששני וקטורים מצביעים לאותו כיוון

(גם אם יש ביניהם מרחק גדול)

הזווית ביניהם קטנה.

כששני וקטורים מצביעים לכיוונים שונים

הזווית גדולה.

במילים פשוטות:

**הזווית מספרת למודל אם שני אובייקטים "נעים סביב אותו**

**רעיון".**

וזה נכון לטקסט, למשתמשים, למוצרים ולתמונות.

מאת: תומר קדם



## למה זה כל כך חשוב בעולם ה-AI?

כי רוב המשמעות שאנחנו מזהים בעולם היא לא "כמה זה דומה פיזית",  
אלא

### כמה זה דומה רעיונית.

וקטורים שמתקרבים בכיוון  
גם כשהם לא קרובים במרחק  
מספרים למודל:  
"זה דומה במהות".

זה מה שמאפשר למודלים להבין ש:

- שמחה וזכיה מתקרבות באותו כיוון
- עצב ואובדן נעות בכיוון דומה
- 'לקוח מאוכזב' ו'שירות לא תקין' קרובים במשמעות גם אם המילים שונות לחלוטין.

בחלק הבא נראה איך משתמשים בזווית הזו כדי למדוד דמיון בין שני משפטים בפועל.

## איך מודדים דמיון בין שני משפטים

עכשיו כשהרעיון של דמיון ככיוון ברור, אפשר לעבור לשאלה המעשית:

### איך מחשבים בפועל את הדמיון בין שני משפטים?

המודלים המודרניים לא בודקים אם שתי מילים זהות או דומות בצליל.

הם משווים בין שני וקטורים שמייצגים את המשמעות של כל משפט. וכדי להשוות בין שני וקטורים, משתמשים במדד שנקרא **דמיון קוסינוס**.

### מה זה דמיון קוסינוס?

זה מדד שמודד

**לא כמה הוקטורים רחוקים,**

אלא

**כמה הם מצביעים לאותו כיוון.**

ערך גבוה (קרוב ל-1) אומר שהמשפטים דומים מאוד במשמעות. ערך נמוך (קרוב ל-0 או שלילי) אומר שהמשמעות שונה.

## איך זה נראה בקוד?

נניח שיש לנו שני וקטורים שמייצגים משפטים,

v1 ו-v2:

```
import numpy as np

def cosine_similarity(a, b):
    dot = np.dot(a, b)
    norm_a = np.linalg.norm(a)
    norm_b = np.linalg.norm(b)
    return dot / (norm_a * norm_b)

v1 = np.array([0.2, 0.8, 0.4])
v2 = np.array([0.25, 0.75, 0.35])

print(cosine_similarity(v1, v2))
```

אם תקבל מספר גבוה, נניח 0.93,

זה אומר שהמשפטים מצביעים על אותו רעיון.

אם המספר נמוך, נניח 0.18,

המשפטים שונים במובן עמוק.

## מה עושה np.dot?

np.dot(a, b) מחשבת את **המכפלה הסקלרית** בין שני וקטורים.

בפועל זה אומר:

היא מכפילה כל רכיב ברכיב שמתאים לו, ואז מחברת את כל

התוצאות.

לדוגמה:

```
a = [1, 2, 3]
b = [4, 5, 6]

np.dot(a, b)
# 1*4 + 2*5 + 3*6 = 32
```

המשמעות:

ככל ששני וקטורים מצביעים בכיוון דומה יותר, הערך של dot גדול יותר.

**מה עושה np.linalg.norm?**

np.linalg.norm(a) מחשבת את **האורך** של הוקטור a. זה מספר אחד שמייצג כמה הוקטור "רחוק מהמרכז".

לדוגמה:

```
v = [3, 4]
np.linalg.norm(v)
# sqrt(3*3 + 4*4) = 5
```

המשמעות:

וקטורים גדולים או קיצוניים יקבלו נורמה גדולה יותר.

**עכשיו הדוגמה של cosine similarity הופכת לברורה**

```
import numpy as np

def cosine_similarity(a, b):
    dot = np.dot(a, b) # כמה הכיוונים שלהם דומים
    norm_a = np.linalg.norm(a) # אורך של הוקטור הראשון
    norm_b = np.linalg.norm(b) # אורך של הוקטור השני
    return dot / (norm_a * norm_b)
```

וקטורים שמצביעים לאותו כיוון יקבלו ערך קרוב ל 1  
וקטורים בכיוונים שונים יקבלו ערך נמוך.

```
v1 = np.array([0.2, 0.8, 0.4])  
v2 = np.array([0.25, 0.75, 0.35])  
  
print(cosine_similarity(v1, v2))
```

- תוצאה גבוהה, למשל **0.93**, אומרת שהמשפטים דומים מאוד במשמעות.

- תוצאה נמוכה, למשל **0.18**, אומרת שהם שונים לחלוטין.

## למה דמיון קוסינוס עובד כל כך טוב?

### 1. הוא מתעלם מהמרחק האבסולוטי

לא מעניינת אותנו כמות המידע.

מעניינת אותנו המשמעות.

### 2. הוא מתמקד בכיוון המשמעותי

הוא מוצא את "קו המחשבה" המשותף בין שני משפטים.

### 3. הוא רגיש להקשרים

אם שתי מילים מופיעות תמיד באותו הקשר,

הן יקבלו כיוון דומה, גם אם המספרים שונים.

## 4. הוא עמיד לשינויים קטנים בדאטה

תוספת של מילה או שינוי ניסוח קל

לא מרסק את הדמיון.

לכן זה המדד הנפוץ ביותר בעולם של

- עיבוד שפה טבעית
- חיפוש משמעותי במסמכים
- מערכות המלצה
- ניתוח התנהגות משתמשים
- כל מערכת שמשווה משמעויות ולא צורות

בחלק הבא נחבר את כל זה לעולם ה-Embeddings,

כי שם הרעיון של זוויות מתחיל לקבל משמעויות אמיתיות בפרויקטים מודרניים.

## חיבור ישיר לעולם ה-Embeddings

בשלב הזה כל מה שלמדנו על וקטורים, זוויות ודמיון מתחבר לרעיון המרכזי שמניע כמעט כל מודל מודרני.

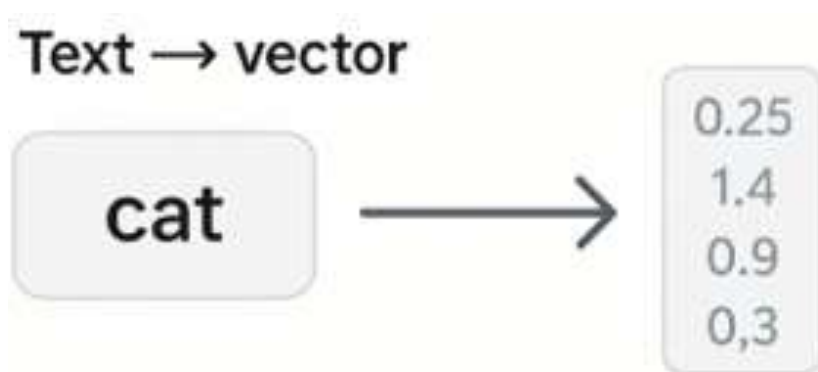
### Embedding הוא וקטור שמייצג משמעות.

המחשב לא מבין מילים, משפטים או תמונות. הוא מבין רק מספרים.

Embedding הוא הדרך להפוך תוכן עשיר ומורכב לרצף מספרים שהמודל יודע לעבוד עליו.

כדי להבין את זה בצורה ברורה, נתחיל בהמחשה חזותית.

Embedding הוא וקטור שמייצג משמעות



באיור זה רואים איך מילה עוברת המרה לוקטור פשוט.

הוקטור לא מתאר אותיות.

הוא מתאר משמעות.

דרך מספרים.

## איך Embeddings ממקמים משמעות במרחב

אחרי שכל משפט או מילה קיבלו וקטור, המודל ממקם אותם בתוך מרחב שבו משמעות דומה מקבלת כיוון דומה.

לדוגמה

מילים כמו חתול גור וחיה יהיו קרובות בכיוון.  
ומילים כמו דפדפן ענן וצבע יהיו רחוקות מהן לגמרי.

Embeddings as points in space



המרחקים פחות חשובים.  
הכיוון הוא מה שמספר למודל מה דומה למה.

## איך כיוון קובע דמיון

כאן נכנס לתמונה דמיון קוסינוס.  
זה המדד שמודד עד כמה שני וקטורים מצביעים לאותו כיוון.  
לא מעניין אותו האורך שלהם.  
מעניינת רק הזווית.

אם הזווית קטנה  
המשפטים מייצגים רעיון דומה.



אם הזווית רחבה  
הם הולכים לכיוונים שונים.

Similarity by direction — cosine similarity



זו הסיבה שדמיון במרחב Embeddings נקבע לפי הכיוון ולא לפי המרחק.

## דוגמה אחת שמאגדת את הכל

נניח שיש לנו Embedding של המשפט  
החתול קפץ על השולחן.

ונרצה למצוא משפטים דומים.

המודל לא יחפש מילים דומות.

הוא יחפש וקטורים שמצביעים לאותו רעיון.

הוא ימצא משפטים כמו

הגור טיפס על הספה

כי הם הולכים לאותו כיוון רעיוני.

משפטים שמדברים על טכנולוגיה או מזג אוויר יהיו רחוקים לחלוטין.  
הכיוון שלהם אחר.

## למה Embeddings הם שכבת היסוד של מודלים מודרניים

כי הם מאפשרים לייצג כל דבר בעולם

מילים

משפטים

משתמשים

מוצרים

תמונות

מסמכים

בצורה שהמודל יודע לחשב עליה.

זה מה שמאפשר למודלים להבין מחשבה אנושית באופן מספרי.

זו הסיבה ש Embeddings נמצאים בלב של

מערכות חיפוש

מערכות המלצה

מודלי שפה

זיהוי תמונות

ניתוח מסמכים

ועוד הרבה מנגנונים שמתרגמים עולם אמיתי לוקטורים.

## פרק 8 – פונקציות – איך מודל חושב

### קלט ← פלט בצורה הכי פשוטה

כמעט כל מודל בעולם ה-AI, מהקטנים ביותר ועד הגדולים ביותר, פועל על רעיון אחד פשוט מאוד:

**מקבלים קלט, מבצעים עליו חישוב, ומחזירים פלט.**

זה הכול.

מודלים יכולים להיות מורכבים, עמוקים ורבים שכבתיים, אבל הבסיס נשאר זהה לחלוטין: פונקציה.

משהו שמקבל משהו אחד ומחזיר משהו אחר.

### מה זה אומר בפועל?

כשמודל מקבל וקטור קלט, הוא מבצע עליו סדרת חישובים. החישובים האלה יכולים להיות פשוטים או מסובכים, אבל העיקרון נשאר זהה:

### המודל ממפה את הקלט לפלט.

למשל

- וקטור של משפט ← סנטימנט חיובי או שלילי
- תמונה ← תג "כלב", "חתול" או "אדם"
- נתוני משתמש ← סיכוי לעזיבה

• וקטור תכונות ← מחיר חזוי

מאחורי כל זה לא עומדת “הבנה” של העולם, אלא **מיפוי מתמטי**.

## למה חשוב לראות מודל כפונקציה?

כי זה מפשט את כל התמונה.

אין קסם, אין אינטואיציות נסתרות.

יש דרך אחת שבה המודל פועל:

**קלט נכנס ← המודל מעבד אותו ← פלט יוצא.**

הפונקציה עצמה היא אוסף של חוקים שהמודל למד מהדאטה.

וזה בדיוק מה שמבדיל בין מודל “טוב” למודל “חלש”:

איזו פונקציה הוא למד.

## ומה שמעניין יותר

המודל לא רק מנסה להחזיר פלט.

הוא מנסה להחזיר **פלט נכון**.

כזה שמקטין את הטעות שלו מהעולם האמיתי.

וזה מוביל אותנו לנושא הבא:

כמעט כל הפונקציות שמודלים לומדים אפשר לצייר כעקומה.

ומודלים תמיד רוצים להגיע לחלק הנמוך של העקומה.

בחלק הבא נבין מה זו “עקומה” ולמה מודלים עושים כל מה שהם

יכולים כדי לרדת בה.

## מה זה “עקומה” ולמה כל מודל מנסה לרדת בה

כשאומרים שמודל “לומד”, בפועל קורה דבר הרבה יותר פשוט: הוא מנסה למצוא **מקום נמוך** על עקומה מתמטית שמייצגת את הטעות שלו.

ואז מגיע רגע שבו הכול מתחבר:

**המודל לא מחפש תשובה. הוא מחפש מינימום.**

## מה זו בכלל “עקומה”?

עקומה היא דרך לייצג את הקשר בין קלט לבין טעות. לכל קלט שהמודל בוחר, הוא מייצר פלט. הפלט הזה מושווה למציאות, וההפרש ביניהם הוא הטעות. כשמציירים את הטעות הזו על גרף, מתקבלת עקומה. נקודה נמוכה על העקומה אומרת “מעט טעות”. נקודה גבוהה אומרת “הרבה טעות”. גם אם זה לא נראה כמו גרף שאתה רואה בבית ספר, זו אותה אינטואיציה:

**יש מקום גבוה, ויש מקום נמוך.**

המודל רוצה לרדת למקום הנמוך.

## למה בכלל יש “עקומה”?

בגלל שהמודל מנסה להתאים את עצמו לדאטה, וכל החלטה שהוא מקבל (משקלים, פרמטרים, כיוונים במרחב) משפיעה על כמה הוא יטעה.

אם הוא מכוון לא נכון  
הטעות שלו גבוהה  
והוא "נמצא" גבוה על העקומה.  
אם הוא מכוון נכון יותר  
הטעות יורדת  
והוא יורד למקום נמוך יותר.

### **למה חשוב להבין את זה כמפתח?**

- כי בלי ההבנה הזו  
למידת מכונה נראית כמו קסם שחור.  
אבל עם ההבנה הזו  
כל התהליך נהיה מאוד הגיוני:
- יש עקומה שמודדת טעות
  - אנחנו רוצים להיות כמה שיותר נמוך
  - כל צעד של המודל הוא ניסיון להתקרב לנקודה נמוכה יותר
- זו כל תורת האימון ב-ML במשפט אחד.

## עקומה לא חייבת להיות דו-ממדית

כשאתה חושב על “עקומה”, אל תחשוב על גרף חלק שמצויר על לוח.

במודלים אמיתיים

העקומה נמצאת במרחב של אלפי או מיליוני משתנים.

אבל האינטואיציה נשארת בדיוק אותה אינטואיציה:

יש מקומות גבוהים ויש מקומות נמוכים.

ולכן כל המודלים בעולם מגיעים לאותה מטרה:

### להקטין טעות.

וכדי להקטין טעות, הם חייבים לרדת על העקומה.

בחלק הבא נבין למה “מינימום” הוא כל כך חשוב,

ומה בעצם קורה כשהמודל סוף סוף מגיע לשם.

## מה זה בכלל מינימום

אחרי שהבנו שהמודל נע על עקומה של טעות ושואף לרדת כל הזמן,

מגיע הרעיון שמחזיק את כל תהליך הלמידה: **המינימום**.

זו המילה שכל מפתח שומע שוב ושוב באימון מודלים.  
אבל מה זה בעצם אומר?

### מינימום הוא המקום שבו המודל טועה הכי מעט

כשמציירים את הטעות כעקומה,  
יש נקודה אחת (או כמה נקודות) שבה הטעות היא הכי נמוכה.  
זו נקודת המינימום.

שם המודל מרגיש "נוח":

הוא מפיק פלטים שמתאימים לדאטה טוב יותר מאשר בכל נקודה  
אחרת על העקומה.

במילים פשוטות:

**מינימום = המקום שבו המודל עובד הכי טוב על הדאטה שראה.**

**אבל לא כל מינימום הוא אותו דבר**

יש שני סוגים חשובים:

#### 1. מינימום מקומי

אזור נמוך, אבל לא הנמוך ביותר.



אפשר לדמיין גבעה קטנה בתוך עמק גדול יותר.  
המודל עלול "להיתקע" שם, וזה חלק טבעי מהאימון.

## 2. מינימום גלובלי

הנקודה הכי נמוכה בכל העקומה.  
זה המקום שבו המודל מגיע לביצועים הכי טובים שהוא מסוגל.  
במודלים גדולים, מרחב העקומות מורכב מאוד,  
ולכן ההבחנה הזו חשובה – גם אם אנחנו לא מציירים את העקומה  
בפועל.

## למה המודל לא תמיד מגיע למינימום הטוב ביותר?

כי העקומה שבה הוא נע היא מאוד מחוספסת.  
יש הרבה ירידות ועליות, הרבה "בורות" קטנים,  
והמודל פשוט נע לפי הכיוון שמקטין טעות ברגע מסוים.

הוא לא רואה את כל המרחב,  
הוא רואה רק את השיפוע המקומי.  
וזה מספיק טוב ברוב המקרים.

## למה מפתחים צריכים להבין מינימום?

כי זה מסביר הרבה התנהגויות של מודלים:

- לפעמים האימון "נתקע"
- לפעמים המודל לא משתפר מעבר לנקודה מסוימת

- לפעמים שינוי קטן בדאטה מזיז את כל המינימום
  - לפעמים הירידה איטית מדי או מהירה מדי
  - ולפעמים יש כמה פתרונות טובים, לא רק אחד
- ברגע שמבינים שמודל רק מחפש נקודה נמוכה על עקומה ולא "מנסה להבין" משהו עמוק יותר הכול נהיה ברור.
- זה הרגע שבו לומדים להבין את המודל כיצירה מתמטית, ולא כקופסה שחורה.

## פרק 9 – שיפוע – המנוע של הלמידה

### מה זה שיפוע בלי להגיד “נגזרת”

כדי שמודל ילמד לרדת בעקומה של הטעות, הוא צריך לדעת **לאן ללכת**.

הוא יודע שיש עקומה, הוא יודע שיש נקודות גבוהות ונמוכות, אבל איך הוא מחליט מהו הצעד הבא?

התשובה היא: **שיפוע**.

ובוא נסביר אותו בצורה הכי אנושית שאפשר, בלי להשתמש במילה “נגזרת”.

### שיפוע הוא פשוט “כמה תלול השטח מתחתיו”

אם אתה עומד על גבעה:

- שיפוע גבוה אומר שהקרקע יורדת חזק לכיוון מסוים
- שיפוע נמוך אומר שהקרקע כמעט שטוחה
- שיפוע חיובי אומר שאתה עולה
- שיפוע שלילי אומר שאתה יורד

המודל מבצע את זה בדיוק

רק על עקומה של טעות במקום על גבעה אמיתית.

### איך השיפוע עוזר למודל?

הוא נותן למודל תשובה לשאלה הכי חשובה במהלך האימון:  
**“אם אזוז קצת שמאלה או ימינה – האם הטעות תגדל או תקטן?”**

זה כל הסיפור.

המודל בודק את השיפוע במקום שבו הוא נמצא:

- אם השיפוע גדול – צריך לזוז מהר
- אם השיפוע קטן – להתקדם בעדינות
- אם אין כמעט שיפוע – אתה קרוב לנקודת מינימום

**למה חשוב להבין את זה?**

כי שיפוע אומר למודל יותר מכל דבר אחר:

**כמה טעות יש כאן, ומה יקרה אם אזוז.**

אפשר לחשוב על זה כעל “מד טעות מקומי” שמכוון את כל תהליך הלמידה.

המודל לא רואה את כל העקומה.

הוא רואה רק את האזור שהוא נמצא בו, ואת השיפוע שמוביל אותו קדימה.

זו הסיבה שבפועל כל תהליך הלמידה נשען על הרעיון הזה:

**לחשב שיפוע ולזוז לכיוון שמקטין טעות.**

בחלק הבא נראה למה השיפוע מייצג בצורה כל כך טובה את כמות הטעות במקום מסוים, ולמה מודלים מסתמכים עליו יותר מכל מדד אחר.

## **למה שיפוע אומר לנו “כמה טעות יש כאן”**

עכשיו כשהרעיון של שיפוע ברור, ניכנס למשהו עמוק יותר: למה בכלל השיפוע מייצג את כמות הטעות במקום מסוים? למה מודלים מסתמכים דווקא עליו כדי לדעת איך להתקדם? התשובה פשוטה יותר ממה שנדמה.

### **שיפוע גבוה = אזור עם הרבה טעות**

כששיפוע חד, זה אומר שהעקומה משתנה מהר. כלומר: הטעות קופצת בצורה משמעותית כשזזים קצת לצד זה או אחר.

אם השטח תלול, אתה יודע שאתה עדיין רחוק מפתרון טוב. זה אומר שהמודל נמצא באזור שבו הוא עדיין טועה הרבה. ולכן צריך לבצע תיקון חזק וברור.

זה כמו להגיד:

**“משהו פה ממש לא מדויק, תזוז חזק לכיוון הירידה.”**

### **שיפוע נמוך = אזור עם מעט טעות**

כשיפוע קטן, העקומה כמעט שטוחה.  
זה אומר שהטעות לא משתנה משמעותית כשזזים קצת ימינה או שמאלה.

זה רמז למודל שהוא מתקרב למקום טוב.  
הוא לא צריך לקפוץ, אלא לבצע צעד קטן ועדין.  
כשהשיפוע כמעט אפס – המודל קרוב לנקודת מינימום.  
אין סיבה לזוז הרבה, כי כל שינוי קטן כבר לא ישפר הרבה.

## השיפוע נותן כיוון וגם עוצמה

זו הנקודה הקריטית:  
לא רק הכיוון של השיפוע חשוב,  
אלא גם **העוצמה** שלו.

- שיפוע חיובי אומר שהטעות גדלה
  - שיפוע שלילי אומר שהטעות יורדת
  - הגודל של השיפוע אומר כמה כדאי לזוז
- זו הסיבה שהשיפוע הוא הכלי המרכזי שמודלים משתמשים בו.  
הוא מסביר גם **לאן** לזוז וגם **כמה** לזוז.

## איך זה נראה ביום יום של מודל?

בכל צעד אימון, המודל בודק:

- מה השיפוע כאן
- האם אני עולה או יורד
- כמה מהר כדאי לי לזוז

ובזכות השיפוע הוא יודע

אם הוא צריך לבצע תיקון גדול

או תיקון קטן

או אולי לעצור כי הוא ממש קרוב לנקודה טובה.

זו כל התבונה של תהליך הלמידה:

למדוד את השיפוע

ולהתקדם בהתאם.

בחלק הבא נראה הדגמה פשוטה וויזואלית שמחברת את כל

הרעיונות האלו לגרף ברור שקשה לטעות בו.

## הדגמה קצרה עם גרף פשוט

כדי שהרעיון של שיפוע יהפוך למוחשי, נבנה רגע את הדוגמה

הפשוטה ביותר:

עקומה בצורת U.

זו עקומה שמייצגת טעות:

גבוהה בצדדים, נמוכה באמצע.

נניח שהטעות מוגדרת כך:

```
import numpy as np
import matplotlib.pyplot as plt

# פונקציית טעות פשוטה
def error(x):
    return (x - 3)**2 + 2

xs = np.linspace(-2, 8, 200)
ys = error(xs)

plt.plot(xs, ys)
plt.xlabel("מיקום על העקומה")
plt.ylabel("טעות")
plt.title("עקומת טעות פשוטה")
plt.grid(True)
plt.show()
```

אם תריץ את הקוד הזה, תראה גרף עם צורה ברורה מאוד:

הטעות גבוהה בקצוות, ונמוכה בנקודה אחת באמצע – באזור של

$x=3$ .

זו נקודת המינימום.



עכשיו נראה מה קורה עם השיפוע.

## מה המודל "מרגיש" לאורך העקומה?

נניח שהמודל נמצא בנקודה גבוהה בצד שמאל.

שם השיפוע **גדול מאוד**.

זה כמו לעמוד על מדרון תלול – ברור לגמרי לאן צריך לזוז.

אם נלך לנקודה גבוהה בצד ימין,

נרגיש שוב מדרון תלול, אבל בכיוון הפוך.

המודל יודע: "תזוז שמאלה, אתה עולה".

באזור המרכז, ליד המינימום, העקומה כמעט שטוחה.

השיפוע קטן מאוד.

המודל מבין שהוא קרוב לפתרון טוב

וצריך לנוע באיטיות או אפילו לעצור.

## איך מודל היה "מיישם" את זה?

נוסיף קטע קוד קטן שמדמה צעד אחד של תהליך הלמידה:

```
# נקודה התחלתית
x = -1

learning_rate = 0.1

for step in range(5):
    slope = 2*(x - 3)    # השיפוע של הפונקציה
    x = x - learning_rate * slope
    print(f"צעד {step+1}: x = {x}")
```

מה תראה כשתריץ את זה?

בכל צעד

**המודל יזוז לכיוון הנכון**

ובקפיצה שמתאימה לגודל השיפוע.

ככל שהוא מתקרב למינימום

הצעדים יהיו קטנים ועדינים יותר.

**למה הדוגמה הזו חשובה לך כמפתח?**

כי היא מתארת בצורה אנושית את כל תהליך הלמידה:

1. יש עקומה שמייצגת את הטעות

2. השיפוע אומר למודל לאן לזוז

3. כל צעד הוא ניסיון להקטין טעות

4. המודל נע עד שהוא מתקרב לנקודה שבה השיפוע כמעט אפס

אין קסם.

אין החלטות נסתרות.

רק סדרה של תיקונים קטנים שמבוססים על השיפוע המקומי.

# פרק 10 – Gradient Descent – הלמידה עצמה

## רעיון ירידת המפל

בפרקים הקודמים דיברנו על עקומת הטעות, על מינימום, ועל שיפוע.

עכשיו אנחנו מחברים את כל החלקים יחד לתהליך שהופך מודל “לא מאומן”

למשהו שמסוגל להבין דפוסים: **Gradient Descent**.

זה השלב שבו המודל מפסיק “לבהות” בדאטה ומתחיל ממש **ללמוד**.

## ירידת מפל – המנגנון שמוריד את הטעות

Gradient Descent הוא פשוט רעיון מתמטי שמדמה ירידה על מדרון.

אם אתה עומד על גבעה ורוצה להגיע לבסיס שלה,

הדרך הטבעית ביותר היא לזוז בכל פעם **לקראת הכיוון שבו הקרקע יורדת**.

אותו עיקרון בדיוק קורה במודל:

הוא “מחפש” את הירידה בעקומת הטעות,

ומתקדם צעד אחר צעד לנקודה עם טעות נמוכה יותר.

**למה זה נקרא “מפל”?**

כי השיפוע מייצג את התלילות של השטח.  
כשערך השיפוע גדול, המפל חד יותר.  
כשערך השיפוע קטן, כבר כמעט הגענו למשטח שטוח ליד  
המינימום.

במילים פשוטות:

**המודל גולש לאורך העקומה עד שהוא מתקרב לנקודה שנוחה לו.**

## **אינטואיציה קצרה**

אם אתה נמצא במקום גבוה על העקומה  
(טעות גדולה)  
השיפוע גדול  
והמודל מבצע צעד גדול.  
אם אתה קרוב יותר לאזור טוב  
השיפוע קטן  
והמודל מבצע צעד עדין יותר.  
זו התאמה דינמית שמאפשרת למודל להתכנס בצורה יציבה.

## **למה זה הגיוני כל כך?**

כי במקום לנסות "לנחש" איפה המינימום נמצא,  
המודל משתמש במידע המקומי שהוא כבר יודע:

הוא פשוט בודק לאן השיפוע מצביע  
וזז בכיוון שמוריד את הטעות.

זו ההבחנה הקריטית:  
Gradient Descent לא מחפש פתרון מושלם.  
הוא שואף לדפוס טוב יותר בכל צעד קטן.  
וזה מה שהופך אותו לאחד הרעיונות החשובים ביותר בעולם ה-AI.

## למה המודל זז “נגד” השיפוע

אחד המשפטים שמבלבלים מפתחים בתחילת הדרך הוא:  
**המודל זז נגד השיפוע.**

למה נגד?

למה לא עם השיפוע?

הרי באופן אינטואיטיבי, אם השיפוע מצביע לכיוון מסוים – לא כדאי  
ללכת אליו?

כאן מגיעה הנקודה המהותית.

## השיפוע מצביע לכיוון שבו הטעות גדלה

כששיפוע חיובי, זה אומר שאם נזוז מעט קדימה בכיוון הזה –  
הטעות תגדל.

כששיפוע שלילי, זה אומר שאם נזוז בכיוון הזה – הטעות תרד.

וזוה הופך את המנגנון לפשוט מאוד:

## **המודל תמיד זז לכיוון שמקטין טעות.**

וכדי להקטין טעות, הוא חייב לזוז **נגד** הכיוון שבו השיפוע עולה את העקומה.

במילים אחרות:

- שיפוע מראה "לאן הטעות גדלה"
- Gradient Descent הולך לכיוון ההפוך: "לאן הטעות יורדת"

## **דוגמה עם אינטואיציה של מדרון**

- אם אתה עומד על מדרון שמטפס למעלה – השיפוע מצביע כלפי מעלה
- כדי לרדת למטה – אתה צריך ללכת בדיוק בכיוון ההפוך

זה כל הסיפור.

זה אפילו לא מתמטיקה, זו תחושת שטח.

## **מה המודל עושה בפועל?**

בכל צעד הוא מחשב את השיפוע באותה נקודה.

אם השיפוע חיובי, הוא זז שמאלה.

אם השיפוע שלילי, הוא זז ימינה.

אם השיפוע קטן מאוד, הוא מאט ומבצע שינוי קטן ועדין.

זו הסיבה ש Gradient Descent הוא תהליך יציב:  
הוא לא עושה קפיצות מיותרות,  
והוא תמיד מחפש כיוון שבו הטעות קטנה יותר.

### **למה חשוב להבין את הרעיון הזה כמפתח?**

כי זה מייצר את ההיגיון שמאחורי כל שינוי שהמודל עושה.  
הוא לא "מנחש" ולא "קופץ".  
הוא פשוט בודק את השיפוע  
ומתקדם בכיוון שמקטין את הטעות.  
זה הסוד של כל למידת מכונה מודרנית.  
בחלק הבא נראה איך קצב הלמידה משפיע על התנועה הזו,  
ומה קורה כשצעד הלמידה גדול מדי או קטן מדי.

## מה קורה כשקצב הלמידה גדול מדי

כל צעד ש Gradient Descent עושה מושפע מפרמטר אחד חשוב במיוחד:

**קצב הלמידה (Learning Rate).**

זה המספר שקובע כמה גדול יהיה כל צעד שהמודל מבצע כשהוא זז נגד השיפוע.

אפשר לחשוב עליו בתור "כמה חזק אני מסובב את ההגה בכל פעם".

הבעיה היא שקצב הלמידה צריך להיות מאוזן. אם הוא גדול מדי או קטן מדי – האימון יוצא משליטה.

### קצב למידה גדול מדי – המודל "קופץ" מעל המינימום

כשקצב הלמידה גבוה מדי, קורה משהו לא נעים: המודל מבצע צעד כל כך גדול שהוא בכלל **עובר את האזור הנמוך** ומתרסק בצד השני של העקומה.

במקום להתקרב למינימום,

הוא קופץ מעליו

ומתגלגל מצד לצד

בלי להתייצב.

זה מוביל לבעיה קלאסית שנקראת "אוסילציות":

המודל פשוט לא מצליח להתייצב במקום טוב.



## למה זה קורה?

כי השיפוע רק אומר האם לעלות או האם לרדת.

הוא לא אומר כמה.

אם קצב הלמידה גדול מדי,

הכיוון נכון

אבל העוצמה מוגזמת.

זה כמו לנסות לעצור על מדרגה

כשאתה יורד בריצה מהירה מדי – אתה פשוט עף קדימה.

## קצב למידה קטן מדי – התקדמות איטית או תקיעה

גם הצד השני בעייתי.

אם הצעדים קטנים מדי,

המודל מתקדם לאט בצורה מתסכלת

ואפילו עלול “להיתקע” באזור שבו השיפוע חלש.

הוא לא מזנק אבל גם לא מצליח לעבור את המדרון.

## איזון נכון

הקסם קורה כשהקצב מאוזן:

- מהיר מספיק כדי לרדת בעקומה
- לא מהיר מדי כדי לא לקפוץ מעל המינימום
- לא איטי מדי כדי לא לבזבז אלפי צעדים

זו האמנות שמסתרת מאחורי אימון מוצלח.

בחלק הבא נראה הדגמה קצרה בקוד שממחישה איך זה נראה כשהקצב נכון, גדול מדי או קטן מדי.

## הדגמת קוד שמראה ירידה למינימום

כדי לראות את כל הרעיונות שלמדנו פועלים ביחד, נבנה הדגמה קטנה של Gradient Descent על עקומה פשוטה. אין כאן מודל אמיתי, אלא רק פונקציית טעות אחת, אבל העיקרון זהה לחלוטין למה שקורה באימון מודלים גדולים.

נשתמש בפונקציית טעות מהסוג שכבר ראינו:

```
def error(x):  
    return (x - 3)**2 + 2
```

הפונקציה הזו מגיעה למינימום כשה- $x$  שווה 3.

עכשיו נכתוב Gradient Descent בסיסי:

```
import numpy as np  
  
def error(x):  
    return (x - 3)**2 + 2  
  
def error_slope(x):  
    # השיפוע של הפונקציה  
    return 2*(x - 3)  
  
x = -1 # נקודת התחלה רחוקה מהמינימום  
learning_rate = 0.1
```

```
print("צעדים:")
for step in range(10):
    slope = error_slope(x)
    x = x - learning_rate * slope
    print(step + 1, "> x =", round(x, 4), "
error =", round(error(x), 4))
```

**מה רואים כשמריצים את זה?**

1. **הערך של  $x$  מתקרב בהדרגה ל-3**, המינימום של הפונקציה.
  2. בכל צעד, השיפוע מכוון את המודל לכיוון הנכון.
  3. כשתתקרב לנקודת המינימום, הצעדים נהיים קטנים ועדינים.
  4. ההתקדמות לא בקפיצות אלא בצורה יציבה ורציפה.
- זה כמעט בדיוק מה שקורה באימון של מודל אמיתי.  
ההבדל היחיד הוא שבמודלים גדולים לא עובדים על מספר אחד  
אלא על מיליוני פרמטרים בבת אחת.

**למה הדוגמה הזו כל כך חשובה?**

כי היא מורידה את כל הרעש מסביב  
ומראה בצורה הכי נקייה שאפשר את העיקרון שבאמת מניע את  
הלמידה:

**המודל מחשב שיפוע**

**זז נגדו**

## והחזר על זה שוב ושוב עד שהוא מתקרב למקום שבו הטעות הכי נמוכה.

אין קסם.

אין קפיצות.

רק רצף של תיקונים קטנים.

## זה כל הסיפור של Gradient Descent

וזה אחד הרעיונות הכי חשובים שכל מפתח בעולם ה-AI צריך  
להבין.

## פרק 11 – פרויקט סיום – mini\_math\_primer

### למה הפרויקט הזה חשוב

אחרי כל הפרקים שעברת עד עכשיו, אתה כבר מחזיק את כל היסודות שצריך כדי להבין מודלים בצורה אינטואיטיבית ומעשית. אבל הסבר תיאורטי לא מספיק, צריך לראות את זה קורה. והכול באמת מתיישב רק כשכותבים קוד.

זה בדיוק התפקיד של הפרויקט הזה:

**לחבר את הבנה של מתמטיקה אינטואיטיבית עם תכלס – כמה שורות פייתון.**

זה לא פרויקט גדול ולא מערכת מורכבת. זה סט של תרגילים קטנים, קטנים ממש, שכל אחד מהם מדגים רעיון אחד מהספר.

כל תרגיל אמור להיות משהו שאתה כותב ב-2-5 דקות. בלי עומס, בלי קפיצה למים עמוקים.

הפרויקט הזה נותן לך שלושה דברים:

1. **יישום אמיתי של כל הרעיונות שלמדנו**  
לא רק להבין – אלא לראות אותם עובדים.

2. **קוד קצר שמייצר "קליק" בראש**  
ברגע שאתה רואה את זה מתקמפל, הטיעון המתמטי הופך למשהו טבעי.

### 3. תשתית מעולה להמשך הסדרה,

במיוחד לספר הבא על מתמטיקה יישומית, שבו נעמיק בנגזרות, מטריצות ותנועות מורכבות יותר.

### איך בנוי הפרויקט?

בכל תרגיל יש:

- הסבר קצר
- קטע קוד בסיסי
- משימה קטנה להרחבה
- שורת הדפסה שמראה תוצאה ברורה

המטרה היא לא לבחון אותך. המטרה היא לייצב את האינטואיציה.

### נתחיל מהתחלה

בחלקים הבאים נכתוב את כל התרגילים לפי הסדר:

#### 1. ממוצע וסטיית תקן

#### 2. הסתברות מותנית

#### 3. בייס על טבלה

#### 4. נורמה

#### 5. cosine similarity

#### 6. Gradient Descent צעד

כל תרגיל יופיע כ"חלק" נפרד כדי שתוכל לעקוב בקלות.

## תרגיל: ממוצע וסטיית תקן

תרגיל קצר שממחיש איך אפשר להבין "מרכז" ו"פיזור" בעזרת מספר שורות פייתון.

### מה עושים כאן?

ניקח רשימת מספרים פשוטה, נחשב את **הממוצע**, את **סטיית התקן**, ונראה איך שני המספרים האלו ביחד מספרים סיפור ברור על הנתונים.

### קוד בסיסי

```
import numpy as np

# רשימת מספרים לדוגמה
data = np.array([10, 12, 13, 9, 8, 15, 14])

# ממוצע
mean = np.mean(data)

# סטיית תקן
std = np.std(data)

print("ממוצע:", mean)
print("סטיית תקן:", std)
```

### מה צריך לראות?

- **הממוצע** אומר איפה "מרכז" המספרים.
- **סטיית התקן** אומרת כמה המספרים רחוקים מהמרכז.

סטייה גדולה מצביעה על פיזור רחב, סטייה קטנה על נתונים מרוכזים.

## משימה קצרה

שנה את הרשימה לנתונים "קיצוניים" יותר, למשל:

```
data = np.array([10, 10, 11, 12, 50])
```

ושים לב איך סטיית התקן "קופצת" בגלל הערך החריג.

## תרגיל: הסתברות מותנית

תרגיל זה מחזק את הרעיון של "הסתברות בתוך הקשר". במקום לבדוק את **השכיחות הכוללת של אירוע מסוים, אנחנו בודקים את השכיחות שלו בתוך קבוצה מצומצמת.**

**הדרך לפתור את זה מתחילה בבניית טבלה של אירועים.**

נבנה טבלה פשוטה של אירועים, ואז נחשב את ההסתברות המותנית:

כמה מתוך האירועים בקבוצה מסוימת מקיימים תנאי נוסף.



## קוד בסיסי

נניח טבלה פשוטה של הודעות:

```
import numpy as np

# נתונים: [ספאם, לא ספאם]
with_free = np.array([42, 3])
without_free = np.array([1158, 797])

total_spam = with_free[0] + without_free[0]
total_ham = with_free[1] + without_free[1]

# בתוך ספאם free הסתברות מותנית: כמה
p_free_given_spam = with_free[0] / total_spam

# בתוך לא ספאם free הסתברות מותנית: כמה
p_free_given_ham = with_free[1] / total_ham

print("בתוך ספאם 'free' הסתברות לראות",
      round(p_free_given_spam, 3))
print("בתוך לא ספאם 'free' הסתברות לראות",
      round(p_free_given_ham, 3))
```

**מה צריך לראות?**

- ההסתברות ל-"free" בתוך ספאם גבוהה משמעותית.
  - בתוך הודעות רגילות כמעט לא מופיעה "free".
  - זה מראה איך הקשר משנה את התמונה.
- זוהי ההבנה המרכזית של הסתברות מותנית.

## משימה קצרה

שנה את המספרים בטבלה.

נסה לדמות מצב שבו רוב ההודעות הן רגילות אבל עדיין free קשור חזק לספאם.

שים לב איך ההסתברות המותנית ממשיכה להראות את הדפוס הנכון גם כשהיחסים הכלליים משתנים.

## תרגיל: בייס על טבלה

תרגיל זה ממשיך ישירות מהתרגיל הקודם. נבין איך משלבים בין התמונה הכללית לבין ההקשר המקומי כדי לקבל הערכה אמיתית של “כמה סביר שזה נכון”.

זה בדיוק מה שבייס עושה, אבל בלי נוסחאות מורכבות. רק טבלה, ספירה, וחישוב אחד קטן.

## מה נעשה כאן?

נחשב את הסיכוי שהודעה היא ספאם בהינתן שמופיע בה free, באמצעות הטבלה הפשוטה.

## קוד בסיסי

נשתמש באותה טבלת שכיחויות:

```
import numpy as np

# טבלה: [ספאם, לא ספאם]
with_free = np.array([42, 3])
without_free = np.array([1158, 797])

# סכומים כלליים
total_spam = with_free[0] + without_free[0]
total_ham = with_free[1] + without_free[1]
total_all = total_spam + total_ham

# הסתברות בסיסית לספאם
p_spam = total_spam / total_all

# בתוך כל קטגוריה free הסתברות לראות
p_free_given_spam = with_free[0] / total_spam
p_free_given_ham = with_free[1] / total_ham

# בכלל free הסתברות לראות
p_free = (with_free[0] + with_free[1]) / total_all

# free בייס: ספאם בהינתן
p_spam_given_free = (p_free_given_spam * p_spam) / p_free

print("free: הסתברות שספאם בהינתן",
      round(p_spam_given_free, 3))
```

## מה צריך לראות?

- זה לא "קסם." זו פשוט דרך לשלב בין מה שקורה בעולם בכלל לבין מה שקורה בתוך ההקשר המעניין אותנו.
- אם free מופיע בעיקר בספאם, בייס משקף בדיוק את זה.
- שינוי קטן במספרים יכול להוציא תוצאות שונות, מה שמחזק את הרעיון: **הקשר משנה הכול.**

## משימה קצרה

שנה את המספרים כך שיהיו הרבה יותר הודעות רגילות בכלל, אבל free עדיין יופיע ברוב המוחלט של הודעות הספאם. בדוק איך זה משפיע על תוצאת בייס, ואיך ההקשר מנצח על התמונה הכללית.

## תרגיל: נורמה

תרגיל זה מדגים בצורה הכי פשוטה מה זה "אורך" של וקטור, ואיך מספר אחד יכול לסכם את כל המידע שמפוזר לאורך רשימה של ערכים.

## מה עושים כאן?

נחשב את הנורמה של כמה וקטורים שונים, ונראה איך אורך הוקטור משתנה בהתאם לערכים שלו.

## קוד בסיסי

```
import numpy as np

# וקטור פשוט
v1 = np.array([3, 4])

# נורמה (אורך)
norm_v1 = np.linalg.norm(v1)

print("נורמה של v1:", norm_v1)

# וקטור ארוך יותר
v2 = np.array([1, 2, 3, 4, 5])
norm_v2 = np.linalg.norm(v2)

print("נורמה של v2:", round(norm_v2, 3))
```

מה צריך לראות?

- הוקטור  $[4, 3]$  מקבל נורמה של 5.
- זה האורך שלו במרחב הדו-ממדי.
- לוקטור ארוך עם ערכים יותר "חזקים" תהיה נורמה גדולה משמעותית.
- זה מייצג אובייקט שנמצא "רחוק יותר" מהמרכז.
- מודלים משתמשים בנורמה כדי להבין חוזק, קיצוניות, או עד כמה האובייקט "רחוק" מהאזור הרגיל של הדאטה.

## משימה קצרה

נסה ליצור שני וקטורים:

אחד עם ערכים קטנים ואחד עם ערכים קפיציים.  
חשב את הנורמות שלהם והשווה ביניהם.

לדוגמה:

```
v_small = np.array([0.1, 0.2, 0.3])  
v_jump = np.array([0.1, 5.0, 0.2])  
  
print(np.linalg.norm(v_small))  
print(np.linalg.norm(v_jump))
```

שים לב איך שינוי קטן בערך אחד יכול להגדיל מאוד את האורך של הוקטור.

## תרגיל: cosine similarity

בתרגיל זה נמדוד דמיון בין שני וקטורים לא לפי המרחק שלהם, אלא לפי **הזווית** ביניהם.

זה בדיוק הכלי שמודלים משתמשים בו כדי לבדוק אם שני משפטים "מצביעים" על אותו כיוון רעיוני.

### מה אנחנו עושים כאן?

נחשב את דמיון הקוסינוס בין שני וקטורים שונים, נראה מה קורה כשהם פונים לאותו כיוון, ומה קורה כשהם שונים לגמרי.

### קוד בסיסי

```
import numpy as np

def cosine_similarity(a, b):
    dot = np.dot(a, b)
    norm_a = np.linalg.norm(a)
    norm_b = np.linalg.norm(b)
    return dot / (norm_a * norm_b)

v1 = np.array([1, 2, 3])
v2 = np.array([1, 2, 3])
v3 = np.array([-3, 0, 1])

print("דמיון v1 ו-v2:", round(cosine_similarity(v1, v2), 3))
print("דמיון v1 ו-v3:", round(cosine_similarity(v1, v3), 3))
```

## מה צריך לראות?

- בין v1 ל-v2 הדמיון כמעט 1  
כי הם מצביעים בדיוק לאותו כיוון.
- בין v1 ל-v3 הדמיון נמוך  
כי הם מצביעים לכיוונים שונים.
- המדד הזה לא מתייחס למרחק.  
הוא מתייחס **רק לזווית** בין הוקטורים.  
ולכן הוא כלי מצוין לניתוח משמעות בטקסט.

## משימה קצרה

נסה ליצור שני וקטורים שונים לחלוטין במרחק,  
אבל בעלי יחס בין רכיבים דומה:

```
a = np.array([1, 2, 3])  
b = np.array([10, 20, 30])
```

בדוק את הדמיון ביניהם.

תראה שהדמיון יהיה כמעט 1, כי הכיוון זהה למרות שהגודל שונה  
לגמרי.



## תרגיל: צעד Gradient Descent

תרגיל זה מחבר את כל מה שלמדנו בפרקים 8–10. הרעיון הוא לראות איך צעד אחד של Gradient Descent משנה את ערך הפרמטר ומקטין את הטעות.

המטרה כאן מאוד פשוטה:

**להראות איך מחשבים שיפוע, איך זזים נגדו, ואיך הטעות יורדת בעקבות הצעד.**

**מה אנחנו עושים כאן?**

נגדיר פונקציית טעות אחת, נמצא את השיפוע בנקודה מסוימת, ונבצע צעד אחד של Gradient Descent.

**קוד בסיסי**

```
import numpy as np

# פונקציית טעות פשוטה
def error(x):
    return (x - 3)**2

# השיפוע של פונקציית הטעות
def slope(x):
    return 2*(x - 3)

# נקודת התחלה
x = -1

learning_rate = 0.1
```

```
print("לפני הצעד:")
print("x =", x, " error =", error(x))

# Gradient Descent צעד אחד של
x_new = x - learning_rate * slope(x)

print("אחרי הצעד:")
print("x =", round(x_new, 4), " error =",
round(error(x_new), 4))
```

## מה צריך לראות?

- הפונקציה מוגדרת כך שהמינימום שלה הוא ב- $x = 3$
  - השיפוע בנקודה  $x = -1$  הוא חיובי
  - לכן Gradient Descent יזיז את  $x$  בכיוון השלילי (נגד השיפוע)
  - אחרי הצעד  $x$ , מתקרב ל-3
  - ועל הדרך, גם הטעות קטנה
- זה בדיוק איך שמודלים גדולים יורדים בעקומת הטעות:  
רצף של צעדים קטנים שכל אחד מהם מקטין את הטעות עוד קצת.

## משימה קצרה

שחק עם קצב הלמידה:

```
learning_rate = 0.8
```

ותראה מה קורה.

האם  $x$  קופץ רחוק מדי?

האם הוא עובר את המינימום?

זה מדגים בצורה ברורה למה קצב למידה חייב להיות מאוזן.

## פרק 12 – איך כל זה מתחבר ל-ML ול-NLP

### למה כל הדברים שלמדנו חוזרים בכל מודל

עברת דרך ארוכה.

וקטורים, נורמה, מרחק, זווית, הסתברות מותנית, בייס, שיפוע, ירידת מפל.

לכאורה זה הרבה נושאים שונים.

אבל האמת הפשוטה היא שכל הדברים האלו חוזרים בכל מודל מודרני, קטן או גדול.

הסיבה מאוד ברורה:

### מודלים לא “מבינים” עולם. הם מבינים מתמטיקה.

הם לא פועלים על מילים, תמונות או משתמשים.

הם פועלים על ייצוגים מתמטיים של הדברים האלה.

ומאותו רגע, כל הכלים שראינו הופכים להיות כלי עבודה קבועים.

### למה זה נכון לכל סוג של מודל?

- כשמודל מקבל טקסט, הוא לא מקבל אותיות. הוא מקבל **וקטור**

**.Embedding**

- ההשוואה בין מילים ומשפטים נעשית דרך **זווית ומרחק** בין וקטורים.

- מציאת פתרון טוב נעשית על **עקומת טעות**.

- כל תהליך הלמידה מבוסס על **שיפוע**.

- התקדמות לקראת פתרון נעשית דרך **Gradient Descent**.
- וכל עיבוד של דאטה מתחיל מהבנה של **ממוצע, פיזור והקשרים**.

ככל שהמודלים גדלים, היסודות האלו לא נעלמים – להפך. הם הופכים להרבה יותר חשובים.

## למה זה משמעותי עבורך כמפתח?

כי אם הבנת את הספרון הזה, אתה כבר יודע לקרוא את השפה הפנימית של המודל.

ברגע שמישהו יגיד לך:

"יש לנו בעיה בהטיית ה-Embedding"

או

"המודל תקוע במינימום מקומי"

או

"ה-cosine similarity לא מספיק גבוה"

זה כבר לא נשמע כמו קסם שחור.

זה פשוט המשך טבעי של החומר שכבר למדת.

אתה לא נשאר בחוץ.

אתה חלק מהשיחה.

## איפה פוגשים אותם שוב בספרון הבא

כל הרעיונות שלמדת כאן הם לא רק בסיס טוב.  
הם **השלב הראשון** לקראת הספר הבא בסדרה, שעוסק  
במתמטיקה יישומית למפתחי AI.

הספר הזה ייקח את כל מה שלמדנו כאן  
ויקדם אותו צעד אחד קדימה, אל העולם שבו מודלים גדולים  
באמת חיים.

## מה חוזר ומעמיק שם?

### • וקטורים ומרחבים גדולים

ראינו כאן וקטורים כסדרה של מספרים.  
בספרון הבא נרחיב איך וקטורים חיים בתוך מרחבים של עשרות  
אלפי ממדים  
ומה קורה כשמרחקים וזוויות משתנים שם.

### • מטריצות

הפעם דיברנו עליהן רק ברמת אינטואיציה.  
בספרון הבא תראה איך כל שכבה במודל היא בעצם מכפלה  
מטריציונית  
ואיך שילוב של מטריצות יוצר את כל מה שמודלים עושים.

### • שיפועים מורכבים

דיברנו כאן על שיפוע של מספר אחד.

בספרון הבא נפגוש שיפועים שמתפזרים על מיליוני פרמטרים ונראה איך מודלים גדולים מחשבים אותם ביעילות.

## • **Gradient Descent אמיתי**

בפרק הזה ראינו רק ירידה על עקומה פשוטה. בהמשך נראה גרסאות מתקדמות של ירידת מפל כגון: Momentum, Adam ואופטימיזציות שבעולם האמיתי אי אפשר להסתדר בלעדיהן.

## • **Embeddings עמוקים**

עבדנו כאן עם דוגמאות פשוטות. בספרון הבא תראה איך Embeddings נוצרים, איך לומדים אותם, איך משנים אותם, ואיך משתמשים בהם במערכות NLP אמיתיות.

## **למה כדאי להמשיך לשם אחרי הספרון הזה?**

כי עכשיו יש לך את **השפה האינטואיטיבית** הנכונה. ברגע שמבינים את אבני הבניין, המעבר למודלים מורכבים כבר לא מאיים אלא מעניין ומאוד הגיוני.

## מה יקל עליך כשהמודלים נהיים גדולים ומורכבים

מודלים היום גדולים יותר, עמוקים יותר, ועובדים על כמויות מידע שפעם לא דמינו.

ואז מגיעה השאלה:

כשהכול נהיה כל כך מורכב,

איך מפתח יכול עדיין להבין מה קורה בפנים?

התשובה פשוטה:

## מבינים רק אם מכירים את היסודות.

החדשות הטובות הן שכשתפסת את אבני הבניין של הספרון הזה, יש דברים שמתחילים פתאום להיראות הרבה יותר פשוטים.

## וקטורים כבר לא מסתוריים

גם אם וקטור Embedding הוא ברוחב 4096, זה עדיין אותו רעיון של רשימת מספרים שמייצגת משמעות. העיקרון נשאר זהה, רק הסקייל משתנה.

## זוויות ומרחקים הופכים לכלי עבודה אמיתי

כשאתה מבין איך דמיון קוסינוס עובד, אתה יכול לזהות למה שני משפטים מתבלבלים, למה חיפוש טקסט מחזיר תוצאות מסוימות, ולמה מודל "ישר קו" בין משמעויות לא קשורות.

הכול מתחיל ונגמר בזווית בין וקטורים.

## שיפוע ו-Gradient Descent כבר לא מפחידים

אימון של מודל ענק נראה כמו קסם עד שאתה זוכר:

הוא עדיין רק מחשב שיפועים

וזז נגדם

מיליוני פעמים.

ברגע שמבינים את המנגנון הקטן,

המנגנון הגדול כבר לא מסתורי.

## הסתברות מותנית ובייס עוזרים לקרוא תוצאות

כשמודל טועה,

הוא לא "טיפש",

הוא פשוט למד באופן שמתאים להסתברויות ולדפוסים שהוא ראה.

ברגע שאתה יודע איך בייס חושב,

אתה מבין למה טעויות מסוימות הן צפויות.

## נורמה ומרחק מסבירים מתי נתון קופץ החוצה

כשמשהו "לא נראה הגיוני",

זה בדרך כלל משתקף באורך חריג של וקטור

או במרחק גדול מדי במרחב.

העין שלך כבר יודעת לזהות את זה.

## מה כל זה נותן לך כמתכנת?

מאת: תומר קדם



בזמן שאחרים רואים מודל כקופסה שחורה,  
אתה רואה אותו כאוסף של צעדים מתמטיים פשוטים.  
אתה יודע לקרוא את ההתנהגות שלו,  
להסביר למה הוא פעל כפי שפעל,  
ולתקן כיוונים בצורה מושכלת.

הבנה של היסודות לא נועדה להפוך אותך למתמטיקאי.  
היא נועדה להפוך אותך למפתח שמסוגל לעבוד לצד מודלים  
גדולים  
בלי להיבהל מהם.