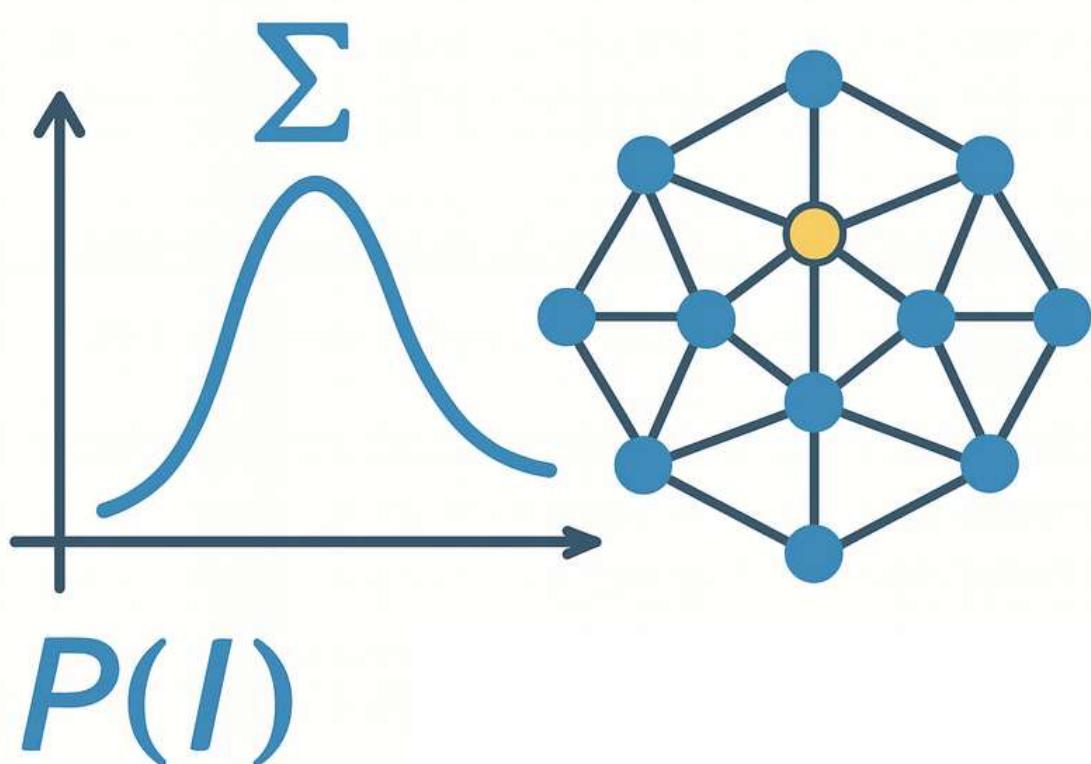


מתמטיקה אינטואיטיבית למתכנתים לעידן ה-AI



מאת: תומר קדם

יוצר וארכיטקט התוכן של
AI Developer World Class Series

גרסה 1.0 | 2025

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 בספרון או לקישור הראשי של הסדרה.

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

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

חלק מהתוכן נוצר ונסקר בסיווע 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 בעלייהם החוקיים, והשימוש בהם נועד לצורכי לימוד בלבד.



תוכן העניינים

מבוא	1
פרק 1 - למה מטמטיקה היא חלק מהעבודה	3
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
פרק 2 - ממוצע, חציון וסטיית תקן – בלי סיבוכים	15
מה זה "מרכז" של DATA	15
מה זה "פיזור" ולמה זה משנה	18
דוגמאות על נתונים אמיתיים	21
הדגמה קצרה ב-NumPy	24
פרק 3 – הסתברות שמדובר בשפה של מתכנת	27
הסתברות כתדרות בעולם אמיתי	27
איך מייצגים תרחישים בצורה פשוטה	29
חיבור ראשוןינו לעולם של סיוגים(Spam/Ham)	35
פרק 4 – הסתברות מותנית וביס – הגרסה אנושית	39

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

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

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

מבוא

כשנכנים לעולם של בינה מלאכותית, יש רגע שבו מרגשים שככל התהlixir קורה מאחריו וילון סגור.

המודלים מייצרים תוצאות מרשים, אבל הדרך שבהם מגעים אליו נראית **מסתורית או לא נגישות**.

התחששה הזו לא מגיעה מkusי אמיתי, אלא מפער קטן בהבנה. אנחנו רגילים לחשב שמתמטיקה שייכת לעולם אחר, שהוא רחוק, שהוא שלמדנו פעם ושאין לו מקום בעבודה היומיומית. המזיאות שונה לגמרי.

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

לא צריך נוסחאות ארוכות, לא צריך סימנים שונים, ולא צריך רקע מתמטי عمוק.

צריך רק להבין מהי השפה שהמודלים מדברים בה.

השפה הזו בניה ממושגים קטנים וברורים:

ממוצע, פיזור, מרחק, דמיון, שינוי, הקשר, הסתבות בסיסית.

אליה אבני הבניין שמוסיפות שוב ושוב בכל מודל, בכל רמה ובכל גודל.

ברגע שהמושגים האלה מתיישבים בראש, שהוא משתנה.

סקפט כבר לא נראה כמו רצף תווים, אלא כמו **וktor של משמעות**. שינוי קטן בדעתה כבר לא נראה אקראי, אלא כמו **תנועה במרחב**.

шиוף כבר לא מילה מתמטית מרוחקת, אלא **אות הcyon** שהמודל משתמש בו כדי להשתפר.

אחת המטרות של הספר זהה היא להראות שכל זה מתחבר בצורה טبيعית לכלים שאתה כבר משתמש בהם בפייתון.

чисובים קטנים, יצירת רשימות, עבודה עם NumPy, בדיקת דמיון, וקטעי קוד שימושיים רעיונות בצורה מיידית.

כשمبינים את היסודות, אפילו רעיוות שנראים מתקדמים יותר הופכים פתאום **ברורים ונגישים**.

הספר הזה לא מנסה להפוך אותך למתרטט.

המטרה היא לחת 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 מלאוה בקטעי קוד קצר שומרה אותו בפועלה.

זהו השלב הראשון בסדרה רחבה יותר.

מכאן נמשיך בהדרגה למתמטיקה יישומית, למידת מכונה, הבנת שפה, ומודלים גדולים.

ולזה ירגיש הרבה יותר קל להשתמש לך את היסודות האלה ביד. עכשו אפשר להתחיל.

ולפנינו שנסכלול 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 לחשב במונחים של קבצים, טקסטים, תמונות ומידע עסקי.

אבל בתחום המודל – כל זה נעלם.

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

הכל מתרגם לצורה אחת בלבד: **מספרים**.

זה לא עניין טכני. זה עניין מהותי.

המודל "מビין" את העולם רק דרך מבנים מתמטיים, ולכן הדרך שאנחנו בונים, מאמנים, בודקים ומשפרים מודלים נשענת על אותם מבנים בדיק.

איך זה קורה בפועל?

כשהת� מגין למודל משפט כמו:

"ה מוצר הגיע מהר ושירות הליקוחות היה מעולה"

המודל לא שומע מילימ.

הוא מקבל וקטור – רשימה של מספרים שמייצגים ממשמעות.

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 מסווני.

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

בסוף כל תמונה – גם אם היא סלפי, וגם אם היא צילום לוין – 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 שפתח מודלים, משלב אותם במערכות או אפילו רק משתמש בהם, צריך להכיר את השפה שבה הם חושבים.
לא עמוק אקדמי – אלא ברמת אינטואיציה.

למה זה משנה?

- כי כשאתה מבין איך מודל רואה מידע:
- אתה מבין למה משפטים דומים מקבלים תוצאות דומות
 - אתה מבין למה מודלים טוענים כשליט יוצא מהטוויח שהם מכירים
 - אתה מבין למה "רעש" בננתונים מפרק ביצועים
 - אתה מבין מה המשמעות של פיצ'רים חסריים 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 بصورة הרבה יותר טבעית. והבסיס של המשקפיים האלה הוא מתמטיקה. לא מתמטיקה כבדה. מתמטיקה שמסבירה **איך תכניות הופכות למספרים, ואיך מספרים הופכים להחלטות.**

למה LM זה בעצם "צמצום טעות"

ברגע שمبرינים שמודלים רואים את העולם כמספרים, מגיעה השאלה הבאה:
איך הם מחליטים מה לעשות עם המספרים האלה? התשובה תמיד חוזרת לאותה נקודה:
מודלים לא "مبرינים" שום דבר.
הם רק מנסים **לצמצם טעות.**

זה אולי נשמע פשוט, אבל זו אחת התרומות הכי חשובות בעבודה עם AI:
כל מודל – קטן או ענק – קם ב�отך עם מטרת אחת:
לקבל קלט, לחת פלט, ולנסות להיות **פחות גרווע ממה שהיא קודם.**

מה זה אומר בפועל?

אם המודל מನחש מחיר של דירה – הוא רוצה שהטעות תהייה קטנה.
 אם הוא מתרגם משפט – הוא רוצה שהטעות הלשונית תהייה 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 להקตน **פונקציה של טעות**.
 וזה המקום שבו המתמטיקה נכנסת לתמונה:
 כדי "לרדוף אחרי טעות", המודל צריך להבין אילו הוא טעה, בכמה הוא טעה, ולאיזה כיוון "לזוז" בשביל לשפר.

כל אחד מההלים 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 שנגע אליו בהמשך הספר.

למה חשוב להבין את זה כמפתח?

כִּי כָּשָׂתָה מִבֵּין שֶׁמּוֹדֵל כָּל הַזְּמָן מְנֻסָּה לְצַמֵּץ טֻועָת, אַתָּה גַּם
מִבֵּין:

- לִמְהָ נְתּוֹנִים "מִלּוּכָּלִים" מְבָלְבָלִים אָתוּ
- לִמְהָ שִׁינוֹנִים קָטָנים בְּנְתּוֹנִים יְכוֹלִים לְגַרְוּם לְשִׁינוֹנִים שִׁמְעוֹתִי

בְּבִיצּוּעִים

- לִמְהָ לְפָעָמִים מְוֹדֵל חֲכָם מַתְנָהָג בְּצָורה טִיפְשִׁית
- לִמְהָ צָרֵיךְ לְהִזְהָר מִמְצָבִים שְׁבָהָם המְוֹדֵל "נְנָעַל" עַל דְּפָס שָׁגָן
- אֵיךְ לְקַרְוֹא מִטְרִיקּוֹת בְּצָורה הָגִינִית וְלֹא כִּסְתָּחַד מַתְמָטִי
הַמּוֹדֵל לֹא מַתְעַקֵּשׁ לְהִוָּת צָודָק.

הָא מַתְעַקֵּשׁ **לְהַקְטִין אֶת הַטּוֹעָות שְׁלָו** – וְלֹעִיתִים זָה מִזְבֵּיל אָתוּ
לְמִקְומֹת לֹא אִנְטּוֹאִיטִיבִּים.

בַּרְגָּעָה שָׁאַתָּה רֹאֶה אֶת המְוֹדֵל כָּלִי שְׁמַנְסָה לְצַמֵּץ טֻועָת וְלֹא כָּלִי
"חוֹשֶׁב",
הַרְעֵשׁ יוֹרֵד, וְהַעוֹלָם נְהִיהָ הַרְבָּה יוֹתֵר בָּרוֹר.

הקשר בין נתונים, תחזיות והחלטות

כל מודל, קטן או גדול, עובד על רצף פשוט של שני שלבים מתמטיים:
הוא מקבל נתונים וմפיק תחזית.

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

מיshaו צריך להחליט מה לעשות עם התחזית זו.

המודל לא מקבל החלטות.

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

ההחלטה מגיעה מבחוץ: מהפתח, מה המערכת העסקית, או מטהlixir אוטומטי שモפעל 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 מאחורייה, ומה המשמעות שלהם.

בלי זה המודל נראה כמו קופסה שחורה.
עם זה הכל נהייה הגיוני וברור.

1. הנתונים

נקודת ההתחלה של כל מערכת AI היא נתוניים.
לא משנה אם זה טקסט, תמונה, סיגナル חיישן או מספרים
עסקיים.

המודל מקבל אותם בצורה **וקטור של מספרים** – עולם שבו כל
תכונה, רעיון או פעולה מקבלת ערך מסוימי.

והרגע הזה קרייטי:
ברגע שהמیدע נכנס למודל, הוא הופך לחלק **מתמטי** מהעולם.

2. התחזית

התחזית היא לא "תשובה".

היא **чисוב** שמתאר מה המודל חושב שיקра.

היא יכולה להיות:

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

אבל ביכול יש עיקרון אחד:

התחזית היא תוצאה של חישובים בין מבנים מתמטיים.

המודל לא "מבין" את התוכן.

הוא משווה, מודד, מחשב ומנסה למצמצם טווח ביחס למה שרצה
לחת.

3. החלטה

השלב האחרון קורה **לא בתוך המודל**, אלא אצל כמפתח. אתה זה שתרגם את התחזית להחלטה:

- האם המיל הזה הוא ספאם?

- האם כדאי לאשר עסקה?

- האם הרובוט צריך לזרז שמאליה?

- האם המשפט הבא צריך להיכתב בצורה אחרת?

ופה בדיק עליה הצורך במתמטיקה אינטואיטיבית:

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

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

בלי זה, התחזיות נראהות כמו קופסה שחורה.

עם זה, העבודה הפכת פשוטה 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, הסתבריות, שיפורים ומדדי מרחק**.

זה בדיק מה שהספר הזה נותן לנו: היכולת לראות את הקשר הזה בעניינים נקיים, בלי עומס ובלוי נסחאות.

מה מפתח צריין לדעת כדי להבין מה באמת קורה בפנים

כל שהמודלים נהנים גדולים ומתחכמים יותר, קל לטעות ולהשווים בהם "מבנהים" את העולם בرمאה אנושית.

אבל מאחוריו הקליים, גם הממערכות המתקדמיות ביותר נשענות על אותן עקרונות פשוטים שהכרנו בחלוקת הקודמים:
וקטורים, מרחקים, הסתבריות, ושאייה מתמדת להקטין טעות.

ופה בדיק מתכנת נכנס לתמונה.

אתה לא צריך לדעת איך לקרוא מאמר אקדמי על רשתות עמוקות.
אתה לא צריך להבין איך נראה ליבות של מודל עצום מבפנים.

אבל אתה כן צריך להחזיק כמה אינטואיציות בסיסיות, כאלה שמאפשרות לך להבין מה קורה כמודל מקבל קלט, מעבד אותו, ומחזיר תשובה.

מה הן האינטואיציות שפתח חייב להכיר?

• כל מודל רואה את הקלט שלו כוקטור.

ברגע שאתה מבין ש"טקט" הוא בעצם רישימה של מספרים, חצי מהקסם השחור נעלם.

• כל תחזית מתבססת על מדידה.

דמיון, קרבה, שנות, פיזור – הכל מתורגם למספרים שמספרים למודל "כמה זה דומה למה שכבר ראיתי".

• טוויות הן חלק מהחיים.

מודלים תמיד טועים.

העניין הוא לא למנוע טוויות, אלא להבין **למה** הן קורות ומה אפשר לשפר.

• מודל אין כוונה. יש לו חשוב.

הוא לא "מבין" טקט, לא "נעלב", ולא "רומז".
הוא בודק תבניות במספרים.

• שינוי קען בנתונים יכול להשפוך לשינוי גדול בתחום המתמטי.

זה משפיע על כל שלב בעבודה: איסוף, עיבוד, בדיקות ופרשנות של תוצאות.

למה זה כל כך חשוב לפתח?

כי ללא האינטואיציות האלה, העבודה מול מודלים מתקדמים הופכת למשחק ניחושים.

כשאין בסיס מתמטי, כל תוצאה נראהיה או לא הגיונית לפי

תחושת בין – זה מסוכן.
כשיש בסיס, גם תוצאות מפתיעות הופכות לשיחה מקצועית:
“המודל הרחיק את הוקטור זהה יותר מדי”,
“הפייזור גבוה מדי ולכן התוצאות רועשות”,
“הסתברות המותנית כאן נמוכה כי הקלט שייר לחת קבוצה
קטנה”.

ברגע שמספר מדבר את השפה הזה
הוא לא רק משתמש במודל, הוא **מבין** אותו.

זה בדיק מה שהספר הזה בא לחת:
את היכולת להסתכל על מודלים בעניינים נקיים,
להבין את המנגנון הפנימי שלהם בלי לגעת בנוסחים מסובכים,
ולהcinן אותו לשלב הבא שבו כבר מתחילה לישם את זה עם קוד
אמיתי.

פרק 2 - ממוצע, חציון וסטטיסטיקה – בלי סיבוכים

מה זה “מרכז” של דedataה

כשאנו אוסף נתונים, הדבר הראשון שאנו מנסים להבין הוא איפה “מרכז” רוב המידע.

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

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

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

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

- **ממוצע**
- **חציון**

שניהם מתארים את אותה שאלה: “מה הערך שמסכם את הדadataה 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 נוחן משקל לכולם – כולל לחריגים.

לכן אם רוב האנשים מרוויחים 10,000 ש"ח וחמשה מנהלים מרוויחים חצי מיליון ה ממוצע “נמשך” למעלה, למرات שהוא לא מייצג אף אדם אמיתי. המוצע טוב כשאין ערכיהם חריגים. הוא פחות טוב כשהיש פיזור קיצוני.

חציוון – הקול של האמצע

החציוון לא מבצע חישובים מסובכים. הוא פשוט שואל:

“מה הערך שנמצא בבדיקה באמצעות הרשימה?”

זו דרך מדוקقة ויציבה יותר להבין מה **רוב** האנשים מרגישים. היא לא מושפעת ממיעוט קיצוני.

לכן בשכר, בזמן תגובה של שרתים, במספר ביקורים באתר או בכל דאטא עם “זנב ארוך” חציוון הוא כלי הרבה יותר שימושי ממוצע.

למה זה חשוב למפתחי AI?

כִּי מודלים לומדים מתוך המספרים שאנו מזינים להם.
אם הנתונים **מוסעים**, 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 שיכולהקדם מודל קדימה – או להפיל
אותו.

ברגע שambilנים ממוצע וחציוון ברמה אינטואיטיבית, הדעתה מפסיק
להיות “רשימת מספרים”, ומתחילה להפוך **لتמונה אמיתית**
שמראה מה באמת קורה שם.

מה זה "פייזור" ולמה זה משנה

אחריו שuibנים איפה מרכז הדאטה נמצא, מגיעה השאלה השנייה:

עד כמה הנתונים מפוזרים סביב המרכז זהה?

שני datasets יכולים לקבל אותו ממוצע והוא ציון, ועודין להיות
שוניים לגמרי.

לדוגמא:

- קבוצה אחת שבה כל הערכים כמעט זהים
- קבוצה שנייה שבה חלק מהערכים נמצאים מאוד וחלק קופצים
לשים

על הניר זה נראה "אותו דבר".
במציאות – אלו שני עולמות שונים לגמרי.
ופה נכנס רעיון הפיזור.

פייזור – כמה הדאטה "רואה"

פייזור הוא מושג פשוט:

כמה רחוקות הנקודות מהמרכז?

אם רוב הנקודות צמודות למרכז – זה דאטה יציב וצפוי.
אם הן מתרפירות לכל הצדדים – יש רעש שמקשה על מודל ללמידה
דפוס ברור.

דמיין שני קבוצות של זמני תגובה (Latency) בשרת:

- קבוצה אחת: 88, 90, 91, 92, 89

- קבוצה שנייה: 15, 30, 400, 450

יש להן חציון דומה, אבל הפיזור שונה לחלווטין.

במערכת אמיתי – הקבוצה השנייה תהיה סיווט תפעולי.

למה פיזור חשוב לאי?

כי מודל לומד מהממוצע והחציון – אבל חי בתוך הפיזור.

פיזור גדול מדי אומר:

- המודל יתקשה לבדוק בדפוסים

- התוצאות יהיו לא יציבות

- אותם נתונים יובילו לפעמים לתוצאות שונות

- המודל "יתבלבל" ויזוז לכיוונים אקראים בזמן האימון

פיזור נמוך אומר שהדעתה נקי, עקבי ואמין יותר.

ברוב פרויקטי ה-AI, לא המרכז מנבה הצלחה או כישלון

אלא רמת הפיזור.

מתכנת 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 – המדריך הפשטוני לפיזור

כאן מגיע כלוי שנשמע מסביר אבל הוא הכיו פשטוני בסיפור:
סטיית תקן.

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

סטיית תקן גובהה = דאטה רועש.

סטיית תקן נמוכה = דאטה יציב.

ברוב המערכות – בעיקר בתחום תחומיות, NLP, מודלים התנהגותיים
ובקרת איכות
סטיית תקן היא אחד המדרדים הכיו שימושיים שיש.

דוגמאות על נתוני אמיתיות

כדי שהמושגים "ממוחע", "חציוון" ו"סטיית תקן" לא יישארו באוויר, צריך לראות אותם על נתונים אמיתיים.

ברגע שראוים מספרים מהעולם האמיתי – הכול מתחילה להרגיש
הרבה יותר ברוח.

נוקח שלושה סוגים קלאסיים של נתונים שמוספיים כמעט בכל פרויקט AI:

1. זמני תגובה במערכת (Latency)

כנнич שאתה בודק זמני תגובה של שרת לאורר היום:

98, 102, 100, 101, 99, 103

המוציא והחציון כמעט זהים.

הספירה קטנה.

כ ר ב ד יו ק נ ר א י ת מ ע ר כ ת י צ י ב ה .

עכשו תסתכל על סדרה אחרת מרובה עומסים:

40, 120, 300, 25, 500, 30

כائ סטיית התקן עצומה.

מבוחנת מודל, זו מערכת שמצויה **התנהגות כאוית**.
או אפשר למצוא דפוס ברור.

2. מחירי מוצרים

נניח שאתה מנהח מחירים של מוצר מסוים ברשות:

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

מה המחיר יגיד לך?

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

החזון, לעומת זאת, מtellם מהחריגות ומהציג תמונה נקייה יותר. ככה מתכנת AI נמנעים מהניסיונות שנגרמו ממערכות קיזוניים.

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 עקבאים
- סטייה גבוהה ← משהו לא יציב, אולי התנהגות חשודה

מודלים שבונים את הפיזור יכולים לזהות אנומליות הרבה יותר טוב.

למה הדוגמאות האלו חשובות?

כי בעולם האמיתי, DATA אף פעם לא "נקה".
תמיד יש:

- רעש
- בעיות הזנה
- ערכים קיצוניים
- פערים בין ערכים
- תקופות שקטות מול תקופות עמוסות

כשפתח מבין את המרכז והפיזור של הדאטה, הוא יודע:

- איך להכין את הדאטה למודל
 - איפה לנוקות
 - איפה להחליק
 - איפה להיזהר שלא להרוס מידע חשוב
- סטטיסטיקה בסיסית היא לא "KİSHOT".
- היא חלק מהaicות של המודל.

הדגמה קצרה ב-NumPy

אחריו שהבנו את הרעיונות מאחוריו ממוצע, חציון וסטיית תקן דרך דוגמאות מהעולם האמיתי, הגיע הזמן לראות איך זה נראה בקוד. וזה בדיקת המקומ שבו NumPy עושה את החישום פשוטים בצורה כמעט מוגחתת.

NumPy נועדה לעבוד עם מספרים במהירות וביעילות. בכל מה הקשור לדאטה – היא הכליל שהופך מתמטיקה “על ניר” למשהו שמתחבר ישירות למערכות אמיתיות.

начילה 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 שקרה באמצע באממת

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

כאן רואים שהחציון נשאר רגוע גם אם יש ערכים קיצוניים.

سطית תקן ← האם הדאטה רגוע או רועש

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

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

ראות הכל ביחד

```
print("median", median)
print("std", 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

לימוד ממנה

זהו אנליזה בסיסית, אבל צזו של מפתח AI משתמש בה כמעט בכל פרויקט — גם כשהוא שםם לב.

למה חשוב לראות את זה בקוד?

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

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

והci חשוב!
זה נותן תחושת שליטה.
פתאום סטיית תקן כבר לא נשמעת כמו הגדרה מספרית, אלא כמו
כלי עבודה ברור.

פרק 3 – הסתברות שמדובר בשפה של תוכנת הסתברות כתדיות בעולם אמיתי

לפני שמדובר על נסחאות או סמלים, חשוב להבין מהו פשוט:
הסתברות היא לא מתמטיקה מופשטת.
היא תיאור של **מספר דברים שקרוים בנסיבות**.

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

וזהו.

אין פה קسم.

הסתברות היא בעצם ספירה

אם מתוך 100 בקשות לשרת:
20 נכשלות
80 מצליחות

از ההסתברות לכשלון היא:

20 מתוך 100 ← 0.2

השפה הזאת טבعت הרבה יותר מMONOCHROME כמו “AIRUIIM”, “MARHBBI
DGINHA” או “HTPFLGOT”.

מפתח לא צריך את כל זה.
הוא צריך להבין **מה קורה בדעתה**.

הסתברות משקפת תדירות

אם 7 מתוך 10 משתמשים לוחצים על כפתור $\leftarrow 0.7$
אם רק פעם אחת מתוך 50 מגיעה בקשה חriga $\leftarrow 0.02$
אם חצי מהתגובה למודל חיובית $\leftarrow 0.5$
המספרים הם תוצאה של **ספריה**, לא של מתמטיקה גבוהה.

למה זה חשוב בעבודה עם AI?

כי מודלים לא "מנחשים".
הם **משווים תדירות בעולם שהם ראו**, ומיצרים תחזיות בהתאם.

לדוגמה:
אם מודל טקסט ראה שב-90 אחוז מהמקרים המילה "טוב" מגיעה אחרי המילה "היה"
הוא יטה 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 עיקבי המודל לא יוכל כלום, גם אם הוא גדול וחזק.

זאת ההבנה הראשונה שאתה חייב כדי לדבר "הסתברות של מודל":

לא מספרים, לא נסחאות.
רק כמה פעמים ממש קורה.

איך מייצגים תרחישים בצורה פשוטה

אחרי שבבינים שהסתברות היא בעצם תדירות, מגיעה השאלה הפרקטייה:

איך מייצגים תרחיש בצורה שהמודל מסוגל להבין וללמוד ממנו?

מבחינת המודל, "תרחיש" הוא פשוט רגע שבו ממש קורה. הוא לא יודע אם זה "משתמש לווח על כפתור", "מイル שמניע", או "משפט שמסתיים".

מה שהוא צריך זה יציג מספרי שונים תמונה ברורה של מה קורה ומה קרה בעבר.

איך אנחנו מייצגים תרחישים?

בדרכם הכללי בצורה הכליא פשוטה שיש:

ספרה של כמה פעמים כל דבר קורה.

זה יכול להיות בטבלה, ברשימה, או במערך של `NumPy`.

העיקרונו תמיד זהה:

המודל לומד מתוך **תדריות**.

לדוגמא, ניקח מערכת שמצויה האם הודעה היא ספאם.

נניח שאנו סופרים כמה פעמים מילה מסוימת הופיעה בהודעות שסוכמנו כספרם או כלגיטימיות:

מילה	מספרם	מופיעים בהודעות ספאם	מופיעים בהודעות רגילות
free	3	42	
urgent	1	17	
hello	58	2	

הטבלה זו מספרת למודל הרבה יותר מאשר מה שנדרשה:

- המילה `free` כמעט תמיד מופיעה בספרם
- המילה `hello` מופיעה בעיקר בהודעות רגילות
- המילה `urgent` מוטה חזק בספרם, אבל לא ב-100 אחוז

בליעוב הסתברות – פשוט מסתכלים על המספרים ורואים את התמונה.

תרחיש = הקשר

אבל לא רק "מה מופיע", אלא גם "איפה זה מופיע".
למשל, אם משתמש לווח על כפתרו רק כשהוא מגיע מדף
מספרים, זה תרחיש חוזר.
אם בקשות מסוימות ל-API כשלות בעיקר בלילה, זה תרחיש חוזר.
אם המילה "בעיה" מופיעה 80 אחוז מהפעמים לפני המילה
"דוחף", זה תרחיש חוזר.
הסתברות היא כדי שמחבר אותו לאינטואיציה זו.

למה זה חשוב בעולם האמיתי?

כי כשאנחנו מפתחים מערכת TM או אפילו כשאנחנו מנהחים
ביצועים של מודלים גדולים

אנחנו תמיד עובדים סביב תרחישים:

- "מה הסיכוי שהתגובה זו נcona?"
- "מה ההסתברות שהמשתמש ינטוש?"
- "כמה פעמים זה קורה ביחס לכל המקרים?"
- "האם זה דפוס או רעש?"

וכשambilנים איך ליצג תרחישים פשוטים
גם הדברים המורכבים יותר נהנים הרבה יותר ברורים.

דוגמאות: למה דברים קוראים יותר או פחות

ברגע שambilנים שהסתברות היא עצם 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 טעה שוב ושוב באותה נקודה

נניח שיש לך מערכת שמליצה על כתבות.

בכל פעם משתמש לווח על כתבה בנושא "ספר", המודל מזהה את זה כתוכנה חיובית.

אבל בפועל, המשתמש לווח על כתבות כאלה רק ביום ראשון בוקר.

כאן יש דפוס:

ימי ראשון ← סיכוי גבוהה יותר ללחיצה.

מי שלא מסתכל על תדירות מפספס את זה.

המודל, לעומת זאת, רואה שזה "קוראה הרבה ביום ראשון", ולכן לומד קשר בין שני הדברים – גם אם בכלל לא התכוונת שזה יהיה פיצ'ר.

דוגמה 2 – למה המילה free נחשבת מסוכנת בספרם

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

אם מילה מסוימת מופיעה **14 פעמים בספאם ופעם אחת בלבד**
בהתשובה רגילה
אין צורך בשום נוסחה.
גם בן אדם היה מסמן את זה כמחשביד.

זו הסתברות פשוטה:
זה קורה אכן הרבה, וזה כמעט לא קורה שם.
מודלים לומדים בדיק קר.

דוגמה 3 – למה מערכת רמזורים “מתחרפת” בשעות מסוימות
אם רואים שבדרך כלל מגיעות 300 מכוניות בשעה,
ופתאום ברגע של 5 דקות מגיעות 200 מכוניות.
זה דפוס שמערכת בקירה חיבת ללמידה ממנה.

ב的日子里 האחרות:
**כשמשהו קורה הרבה יותר מהרגיל – ההסתברות שלו גבוהה
באותו רגע.**

וכשמשהו קורה פחות מהרגיל – ההסתברות שלו יורדת.
זה נשמע טריוויאלי, אבל זו הליבה של רוב המערכות המודפסות
אנומליות.

דוגמה 4 – למה יש יותר משתמשים חוזרים מאשר חדשים
נניח שבאתר מסוים:
• 25 אחוז מהמבקרים הם חדשים

• 75 אחוז חוזרים מה המודל למד? שהסבירות להתנהגות של " משתמש חדש" גבואה פי שלושה מזו של " משתמש חדש".

במודלים התנהגותיים, הדפוס הזה משפיע על כל תחזית.

דוגמה 5 – למה מודל NLP נוטה להשלים מילים מסויימות

אם 80 אחוז מהפעמים שבוחן מופיעה המילה מזג מגע אחרת אוויר

המודל למד שזה "המשך טبعו".

הוא לא עושה קסמים.

הוא פשוט סופר תדிரיות.

מה המשותף לכל הדוגמאות?

העיקרון הבסיסי חוזר שוב ושוב:

דברים שקורים הרבה ← מקבלים משקל גבוה

דברים שקורים מעט ← מקבלים משקל נמוך

זה הכל.

זה נבנים מודלים שיכולים לנתח טקסט, לזהות תМОנות, להמליץ על תכנים או לנבא התנהגות.

וברגע שאתה כמפתח מתחילה לראות את העולם דרך תדירות, המודלים מפסיקים להיות קופסאות שחורות, והופכים לכליים שאתה מבין באמת.

חיבור ראשוני לעולם של סיוגים (Spam/Ham)

עכשו שהבנו שתדירות היא הלב של הסתבות, אפשר לראות איך זה יושב בבדיקה על אחד היישומים הכחיסיסים בעולם ה-ML: **סיוג (Classification)**.

ברוב היישומים הראשונים של מתכנתים בעולם ה-AI, סיוג הוא הצעד הטבעי:

אם זה ספאם או לא?

אם הלקוח יעוזב או ישאר?

אם התמונה מכילה חתול או כלב?

אם המשפט חיובי או שלילי?

למרות שהמשימות נראות שונות לגמרי, יש להן מבנה זהה:

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

וכמו שראינו בפרקם הקודמים – השיר זהה מבוסס על תדירות.

למה דווקא Ham/Spam הוא הדוגמה הכחיקאליסטית?

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

אם בתיית הדואר שלנו:

- 1,200 הודעות מסומנות כספרם
- 800 הודעות מסומנות כרגילות

از כבר לפני שמתכוילים על המילים בתחום ההודעות, יש עובדה
חשובה:

הסתברות בסיסית לספרם גבואה יותר מהסתברות בסיסית להודעה רגילה.

כלומר: מתחום כלל העולם שהמודל רואה
יותר דברים נופלים ל"ספרם".

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

איך המודל משתמש בתדריות 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 נסחאות מתחככות.

זו פשוט ספירה.

ברוב המקרים, סיוגים ראשוניים עובדים על מבנה בסיסי כזה:
הסתברות למילים מסוימות בתחום קבוצות שונות ← תדריות ←
החלטה.

מה זה אומר למפתח בפועל?

כשפתחים מערכות סיוג, 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 כמו
עוד לפני שנוגעים בקוד שלו.

למה זה חשוב לקרוא הפרק הבא?

כי בעולם האמיתי, מודלים לא מסתפקים במידע על "כמה פעמים
מילה מופיעה".

הם רוצים לדעת:

- מה הסיכוי למשהו **בהתנתק** משהו אחר?
- מה המשמעות של מילה בתחום הקשר?
- איך מחשבים הסתברות כישר שתי קבוצות עם מאפיינים שונים?

ופה בדיק נכנסת **הסתברות מותנית**.

זה אחד הכלים החזקים ביותר בהבנת דפוסים,
ולמעשה הבסיס של משפט בייס – שהואقلب של כל שימוש
הסיוג.

פרק 4 – הסתירות מותנית ובייס – הגרסה אנושית

מה זה באמת הסתירות מותנית

בפרק הקודם דיברנו על שאלה פשוטה:

“כמה פעמים משוה קורה מtower כל המקרים?”

זו הסתירות בסיסית, והיא חשובה.

אבל בחים האמיתיים – ובטח בעולם ה-AI – זו לא השאלה שאנוחנו באמת רצים לשאול.

השאלה האמיתית היא אחרת לגמרי:

“כמה פעמים משוה קורה בתower קבוצה מסוימת?”

זו הסתירות מותנית.

וברגע שambilנים אותה, העולם משתנה.

דוגמה יומיומית

אם אתה שואל:

“מה הסיכוי שריד היום גשם?”

זו שאלה כללית.

אבל אם אתה שואל:

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

זו שאלה אחרת לחלוטין.

ברגע שיש **הקשר**, הסיכוי משתנה.
אנחנו כבר לא בודקים את הכל
אלא **מצב חלקי**, תחת-קובוצה.
זה בדיק מהות ההסתברות המותנית.
זה קורה כל הזמן בעולם של מודלים
מודל לא שואל:
"מה ההסתברות שמשהו הוא ספאם?"
אלא:
"מה ההסתברות שהודעה זו ספאם בהינתן שהיא מכילה את המילה free?"
או בעולם אחר:
"מה הסיכוי שלקוח יעצוב, בהינתן שגם הפניה השלישית שלו החודש?"
או ב-NLP:
"מה הסיכוי שמילה מסימת תגיע עכשו, בהינתן שלושת המילים הקודמות?"
שנת לב?
אין פה נוסחה.
יש הקשר.

למה זה כל כך חשוב?

כי מודלים לא עובדים על תדிரיות "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 מסויימת, מה המשמעות של זה בטקסט?
- אם משתמש מתרגם בצורה מסויימת, מה זה אומר על הצעד הבא שלו?

זו הסתברות שמדובר במקרה מתכונת:

פשוטה, הגיונית, מחוברת להקשר.

וכאן מתחילה החיבור לבייס

בייס הוא לא נוסחה כבדה.

הוא לא "מתמטיקה של סטטיסטיקים".

הוא הדרך לחבר בין:

העולם הכללי ← כמה משהו נפוץ באופן כללי

העולם שבקשר ← כמה הוא נפוץ בתחום תח-קבוצה מסוימת

כדי הגיעו לתשובה אחת:

כמה סביר שזה נכון עכשו.

לפני שנגיע לביס עצמו, צריך לבנות את השולחן שעליו הוא עומד: טבלאות פשוטות, מספרים קטנים, וספרה ישרה. זה בדיק מה שנעשה בחלק הבא.

ביס דרך טבלאות, לא דרך נוסחאות

כדי להבין את הרעיון של ביס בצורה **אנוונית**, צריך לשים לצד את הנוסחאות ואת כל ה(ז|(א)פ).

לפני הכל, ביס מתחילה מהסתכלות על **מספרים אמיתיים שמסודרים בטבלה**.

שם מסתתר כל הרעיון.

אבל לפני שנצלול לתוך הדוגמה, חשוב להבהיר מהו קטן שמנוע הרבה בלבול:

המילה ביס היא לא קיצור ולא ביטוי טכני.

זה פשוט שמו של תומס ביס, כומר ומתמטיקאי שחי לפני כמא מאות שנים.

ביס הציג רעיון שהוא חדשני לזמןנו:

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

זה הבסיס לכל מה שנקרה היום “**חישיבה ביסיאנית**”.

עכשו ניגש לטבלה.

הטבלה מספרת שני דברים חשובים:

מה קורה בעולם בכלל

ומה קורה בתחום ההקשר שמעניין אותנו עכשיו.

נראה את זה בדוגמה הכיו בrhoה: זיהוי ספאם.

סרקנו מיילים וספרנו כמה פעמים מופיעה המילה "free".

זה מה שקיבלנו:

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

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 – הקונקטסט מוטה חזק לכיוון ספאם.

כש free לא מופיעה:

- 1158 היו ספאם

• 79. היו רגילותות

כאן התמונה הרבה פחותה חדה.

הטבלה מציגה שני עולמות:

- העולם הכללי – כמה ספאם יש בכלל
- העולם שבקונטקט – מה קורה כשהיא free מופיעה או לא מופיעה
ובויס פשוט מחבר בין שני העולמות האלה.

מה בvais בעצם שואל?

הוא מסתכל על הטבלה ושואל:

בhinthan שאני רואה free

איזה תרחיש הופיע יותר פעמים?

או בניסוח "אנושי":

מה יותר סביר עכשו, בהתחשב بما שאני רואה מול העיניים?

זה עובד לא רק בטקסט

אותו עיקרונו קורה גם בעולם של:

- זיהוי הונאות
- עדיבת ל��חות
- אנומליות במערכות
- זיהוי אובייקטים
- כל תחום שיש בו סיוג

בכל פעם שיש לר' קונטקט, ובתוכו דפוסים שונים. טבלה כזו היא הכליל הכי חזק להבין מה קורה באמת. לפני שנראה "כמה זה סביר" בצורה אינטואיטיבית, נבין מה עומד מאחורי השאלה זו.

מה מסתתר מאחורי השאלה "כמה סביר שזה נכון?"

אחרי שמבינים איך נראה טבלת שכיחיות, 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 וקשרים 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 שופיעו הרבה בתוך קבוצות מסוימות.

אם free הופיעה בעיקר בספרם
ואותה המילה מופיעה עכשו
המודל מקבלرمز לכיוון מסויים.

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

המודל שואל את עצמו שאלות כמו

- האם הדפוס הזה מתאים למה שקרה בדרך כלל בקבוצה מסויימת?
- האם מה שאני רואה עכשו אפילו מקרים שכבר הכרת?
- מה קרה בעבר במצבים דומים?

זו חשיבה הסתברותית שבבסיסה על ניסיון, לא על הגדרה מתמטית.

למה זו לא החלטה דיקוטומית

המוח שלנו אוהב שחור לבן.
מודלים עובדים באפור.

הם לא אומרים "זו הودעה רגילה" או "זו הודעת ספרם".
 הם אומרים

בקשר זה יש יותר ראיות לכך מאשר לכך.

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

איך זה מתחבר לעבודה של מפתח

כשהתבין שהמודל בודק סבירות ולא אמת מוחלטת אתה פתאום קורא תוצאות בצורה אחרת. אחזים ומדדים מפסיקים להיות סתם מספרים והופכים לرمזים לבנייה הדאטה.

המבנה של סבירות עוזרת לך:

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

במילים אחרות

לדעת לשאל "כמה סביר" היא מיומנות הנדרשת לכל דבר.

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

הדגמה קטנה בקוד שתגבור לזה להתחבר

אחרי שעברנו דרך הרעיון של הסתירות מותנית והבנו איך מודל מחבר בין הקשר לבין תמונה העולם הרחבה, הגיע הזמן לראות את זה קורה בפועל.
 לא נסחאות.
 לא מודל כבד.
 רק טבלה, קצת ספירה, וכמה שורות פיתון.
 נשתמש בדוגמה שכבר ראיינו: המילה `free` בהודעות ספאם או ריגילות.

הטבלה שלנו נראהthus:

מופיע המילה "free"	ספאם	לא ספאם
מופיע	3	42
לא מופיע	797	1158

בוא נתרגם את הטבלה לפיתון ונראה מה המודל היה "Margin".
 כשהוא מקבל הודעה עם `free`.

```
import numpy as np

# טבלת שכיחיות
spam_with_free = 42
ham_with_free = 3

spam_without_free = 1158
ham_without_free = 797
```

```

# סך הכל הודעות בכל קטgorיה
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 תזרירות
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("בתוך ספאם free הסתברות לראות:", round(p_free_given_spam, 3))
print("בהודעות רגילות free הסתברות לראות:", round(p_free_given_ham, 3))

```

מה קורה כאן בפועל?

1. מחשבים כמה ספאם יש בכלל

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

2. בודקים כמה פעמים free מופיעה בכלל קבוצה

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

3. משויים בין שתי הסתברויות

אם free מופיעה יותר בספרם מאשר בהודעות רגילות המודל יטה לכיוון ספרם גם אם זה לא ודאות מוחלטת.
זה בדיק "כמה סביר שהוא נכון".

4. הסיפור מאחוריו המספרים

אם תפעיל את הקוד, תראה שימושו בולט מיד:
הסתברות להופעת free בתחום ספרם גבוה בצורה-kitzhnit
והיא נמוכה מאוד בתחום ה Hodutot רגילות.

במילים אחרות
אם free מופיעה – הסיכוי לספרם עולה משמעותית.

זו לא נוסחה.

זו הבנה דרך דפוסים.

זה הלב של ביס:
שילוב בין תדיות כללית לבין תדיות בתחום הקשר.

סיכום

הסתברות מותנית וביס הם לא חומרים תיאורתיים בלבד, אלא כלים מעשיים שכל מפתח צריך להבין ברמה אנושית.

כשהתה מבין

- כמה דברים קורים בתחום הקשור
- איך מודל בודק סבירות
- איך טבלה פשוטה מספרת סיפור

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

פרק זה מספק לך את היכולת להשתכל על מודלים מתחומי עולם של
סבירות ולא מתחומי עולם של ניחושים.
וזו מiomנות שתשרה 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 שימושיים תוכן.
זה בדיק מה שנוטן לוקטורים את הכוח שלהם.

מה זה בכלל וקטור?

זה פשוט מאד.

וקטור הוא רשימה של מספרים.
לא מטריצה, לא אובייקט מסוים.
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 **מביין** דברים.
כשאומרים למודל "זה משפט חיובי" או "זה חתול",
אי אפשר לחת לו תמונה או טקסט ישירות.
צריך לחת לו מספרים.
והמספרים האלו הם וקטור.

AIR אובייקט הופך לוקטור?

אפשר לחשב על זה כעל תרגום.
אובייקט מגע מבוחץ – תמונה, מילה, משתמש, מה שלא יהיה
ומודל צריך דרך לייצג אותו בצורה מתמטית.

הרעיון פשוט:
לקחמים את התכונות החשובות ומmirים 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 תוכנות שנלמדו מدادה

עכשו נסביר כל דוגמה בצורה פשוטה וברורה.

צבע של פיקסל הופך למספר

בכל תמונה יש פיקסלים.
למודל אין מושג מה זה "אדום" או "כחול".
از 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 לרשימה מספרים שמייצגים משמעות
כאן זה הכى מבלב>Anshim.
אין "מילים" בתוך מודל.
אין אותיות.
הכול הופך ל **וקטור** – רשימה מספרים.

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

אבל הרשימה זו לא מייצגת "אותיות".
היא מייצגת משמעות:
אוכל, טעם, מסעדה, שומן, טקסט, הקשר.
המספרים נוצרים מהתהילה אימון, לא מניחוש ידני.

כל מילה = רשימת מספרים.

משתמש הופך לוקטור של התנהוגיות

המודל לא עובד עם "פרופיל משתמש".

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

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

כל זה הופך לשורה של מספרים, כגון:

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

זה לא "מי הוא" – זה **איך הוא מתנהג** מבחינה דפוא.

מוצר הופך לערכים שמייצגים תוכנות

אם אתה מוכר מוצרים,
למודל לא משנה "איך המוצר נראה".

הוא צריך מספרים שייצגו:

- מחיר
- קטגוריה
- פופולריות
- תדיות 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?

הכל הופך למספרים.
לא כי מפתחים אוحبים מתמטיקה,
אלא כי זו **השפה שהמודל ידע לדבר**.

ברגע שאובייקט בעולם מקבל ייצוג מספרי,
המודל יכול להשוו, 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 שمبرינים שכל אובייקט הופך לocketor
MBERINS גם:

- איך מודלים משווים בין דברים
- איך הם מחשבים דמיון
- למה דברים שונים מתבלבלים לעיתים
- למה שינוי קטן בנתונים יכול ליצור שינוי גדול בתוצאה
- ולמה כל כרך חשוב שהוקטוריים יהיו נקיים ומיצגים
ocketor הוא לא רק "רשימת מספרים".

הו **זהות המתמטית** של כל דבר שהמודול עבד אותו.

בחלק הבא נבין למה כמעט כל AI בעולם, קטן וגדול, עובד בבדיקה
עם המבנה זהה.

למה כמעט כל AI עובד על וקטורים

אחריו שהבנו שוקטור הוא פשוט רשימה של מספרים שמייצגת אובייקט, עולה שאלת טבעיות:

למה כל עולם ה-AI עובד דואק עם וקטורים?

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

1. וקטורים מאפשרים מודל למדוד דמיון

זה הרעיון הכלי חשוב.

כדי שמודל יוכל ש:

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

הוא צריך דרך למדוד קרבה.

וקטור מאפשר לעשות בדיקות אלה

על ידי חישוב של:

- מרחק
- זווית
- דמיון

בין רשימות של מספרים.

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

2. וקטורים אפשריים לבצע חישובים מהירים מאוד

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

3. וקטורים אפשריים למודל "למוד" תוכנות בעצמו

במערכות ישנות היו צריכים לבחור תוכנות ידנית.
בינה מלאכותית מודרנית לומדת תוכנות בלבד
ומכניתה אותן לתרוך הוקטור.

זה אומר שהוקטור מייצג, לא רק מידע גולמי
אליא גם משמעות שהוא מודל למד מתוך דוגמאות.

המילה "כלב" תקבל וקטור שמתקרב
למילים כמו "גור", "לנבות", "חיה"
ומתרחק מילים 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 יהיה וקטור.
וזה 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 דאטא מובנה ולא מובנה.
הבסיס תמיד אותו בסיס: וקטור.

איך טקסט, משתמש או תמונה הופכים לocketor

עכשו כשרעון של וקטורים ברור, מגיעה השאלה המעשית:
איך אובייקטים אמיתיים, كالה שאנו רואים במערכות יומם,
הופכים לרשימות מספרים שמודל יכול לעבוד איתן?
נפרק את זה לשישה סוגים נפוצים של מידע.

טקסט ← וקטור של משמעות

זו אחת ההמרות הכי מעניינות.

בתחילת הדרך
הינו מmirים כל מילה במספר לפי המיקום שלה במלון.
זה לא עבד טוב, כי מילים שונות לחלוין קיבלו מספרים בלי קשר
למשמעות שלהן.

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

לדוגמה, המילים
לקוח, משתמש, צרכן
יהו קרובות יחסית.
אבל המילה מנורה תהיה רחוקה מהן.

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

משתמש ← וקטור של ההתנהגות

גם משתמשים הופכים לוקטורים.

הוקטור שלהם יכול להכיל למשל

- כמות ביקורים
- סוג פעולות
- זמנים אופייניים
- קשריות שנצפו
- אורך שנים
- דפוסי מעבר בין מסכים

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

שני משתמשים שונים לחולטן
יכולים לקבל וקטורים דומים
אם הם מתנהגים בדרך דומה.
זה בדיק 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 ומיציר וקטור מאפיינים.

הוקטור זהה 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 הזרזורה שונה
גם אם הרקע שונה.

שוב

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

למה הכל עובד אותו דבר?

זה היופי.

כיוון שהכל מתרגם לוקטורים
מודלים יכולים לעבוד על סוגים שונים של מידע
באותה לוגיקה.

הם לוקחים וקטורים
משווים
מודדים מרחקים
מצהירים דפוסים
לומדים קישורים
ונותנים תחזיות.

הבסיס תמיד אותו בסיס.

פרק 6 – נורמה ומרחק – מודדים את העולם

מה זה “אורק” של וקטור

אם פרק 5 עסוק בשאלת איך מייצגים דברים כוקטוריים, הפרק הזה עוסק בשאלת הבאה:
איך מודל יודע כמה הוקטור זהה “גדול”, “חזק” או “קיצוני”?
וזו בדיקת הנורמה.

מהי נורמה?

נורמה היא מספר אחד שמסכם את “הגודל” של הוקטור. אפשר לחשב עליה כעל אורק.
אם תצייר וקטור על לוח קוקויים, הנורמה היא האורק של הקו.
בפייתון, אם יש לנו וקטור:

$$v = [3, 4]$$

הנורמה שלו היא

5
כי זה האורק של הקו מהנקודה (0,0) לנקודה (3,4).

אצל מודלים הרעיוניים זהה רק שהוקטורים ארוכים בהרבה
ולכן האורק שלהם מספר שהוא חשוב.

למה אכפת למודל מהאורק?

נורמה מציגה למודל

כמה חזק או קיצוני אובייקט מסוים ביחס לאחרים.

לדוגמה

- משתמש עם פעילות ענקית קיבל וקטור ארוך יותר
- תמונה שיש בה המון פרטים לפחות יוצרת וקטור ארוך יותר
- מילה שמשמעותה תמיד בסביבה "חזקת" מקבל וקטור שאורךו שונה ממלים פחות משמעותית
- בנתונים שמתנהגים בצורה קופצנית ולא עקבית, חלק מהוקטורים קופצים באורך בצורה לא יציבה הנורמה הופכת את העולם המרובה ממדיים למספר אחד שקל להשוות.

למה "אורך" עוזר למערכת להבין דברים?

כי כשאתה משווה אורךים

תקבלת אינטואיציה על "כמה הדבר זהה גדול יחסית לאחרים".

לא צריך מתמטיקה כבדה.

צריך את ההבנה פשוטה זו:

נורמה עוזרת למודל להבין את עצמת המידע.

בפרקטייה

הנורמה משמשת כדי

לנរמל דאטा

להבין חריגות

להשליך נקודות שנמצאות "רחוק מדי"

- לקבוע גבולות במערכות זיהוי אונומליות
- להכין את הוקטורים להשוואה אמיתית
ובפרקם הבאים של הספר
היא תריהacha אחת המילים שיחזרו הכיו הרבה.

איך מודלים מודדים כמה דברים דומים או שונים

אחרי שהבנו מהו האורך של וקטור, מגיעה שאלת בסיסית שכל מודול חייב להתמודד איתה:

איך יודעים אם שני אובייקטים דומים או שונים, כשהם מוצגים כוקטורים?

בגלל שכל מודול עובד על רשימות מספרים, הוא צריך דרך להשוות בין שתי נקודות למרחב וltrגם את ההשוואה הזאת ל"קרבה" או "רחוק".

הצעד הראשון: מרחק

המדד הכיו פשוט לדמיון הוא **מרחק בין וקטורים**.

העיקרונו מאד אינטואיטיבי:

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

דוגמאות:

```
import numpy as np

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

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

המספר שמקבל הוא המרחק הגיאומטרי ביניהם.

למה מרחק הוא כלי שימושי?

כי הוא נותן לנו מודל מודד ישיר לשאלת

"כמה שני דברים רחוקים אחד מהשני?"

זהו שימושי במיוחד ב:

- מערכות המלצה
- זיהוי תסבוכות
- חיפוש דמיון בין מסמכים
- איתור חריגות
- קיבוץ נתוניים 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 הקritisית:

מרחק לא יודע להבין משמעותו.

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 משפטיים שמקילים מילים דומות יכולים להיות קרובים במרחב המספרי למרות שהמשמעות שלהם שונה.

במילים פשוטות:

מרחב מודד שונות גיאומטרית, לא שונות רעיונית.

למה זה בעיתי במיוחד בטקסט?

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

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

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

הם צריכים כל שפה **כיוון משותף** בין קטורים ולא רק את "הגודל" של ההפרדה ביניהם.

זה בדיק מה שmobail אותנו לשיטה שחיבים להכיר בעולם ה-NLP:

מדידת זווית בין קטורים, או במילים פשוטות – דמיון קוסינוס.

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

למה מרחק לא מספיק בניתוח טקסט

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

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

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

ופה מרחק כבר מתחילה לחרוק.

דוגמה שמחישה את הבעיה

שני המשפטים

"אני מאחר לעובדה"

|

"אני מתקשה להגיע בזמן"

משתמשים במילים שונות, אבל המשמעות שלהם כמעט זהה.

לעומת זאת, המשפטים

"אני אוהב קפה"

|

"אני אוהב לשורף גשרים"

משתפים חלק מהמילים, אבל המשמעות שלהם רוחקota לגמרי.

מרחק גיאומטרי לא יודע לעשות את הבדיקה 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:

מרחך מתייחס רק לפער במספרים, לא לתוכן שמסתתר מאחוריהם.

למה זה קורה?

כי מרחך עונה על שאלה אחת בלבד:

"כמה הוקטורים מתרחקים אחד מהשני למרחב?"

אבל הוא לא עונה על שאלות כמו:

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

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

מודלים צריכים כדי נסף שמיין כיוון

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

"איזה 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 במרחב. המצד הזה נקרא **דמיוון קוסינוס**, והוא אחד הכלים החשובים ביותר להבנת משמעות בשפה.

בחלק הבא עמוק בכל זה ונראה איך הוא פותר את כל הבעיות שמרחב לא מצליח לפתור.

פרק 7 – זווית ודמיון קוסינוס

דמיון ככיוון ולא כמרחך

בפרק הקודם רأינו שמרחב בין וקטורים הוא כלי שימושי, אבל יש לו מגבלות ברורות.

בעיקר בעולם של טקסט, שבו המשמעות לא תמיד יושבת על "עד כמה שני משפטי רוחקים 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 מה מרחק?

תחשוב על שני משפטיים שונים מבחינת המילויים, אבל דומים מאוד בReLUוּן:

"אני מאחר לעבודה"

|

"אני מתקשה להגיע בזמן"

המשמעות שלהם זורמת באותו כיוון.

גם אם הוקטורים רוחקים יחסית, זווית קטנה ביניהם מספרת שהם "פונים" באותו כיוון.

לעומת זאת:

"אני אוהב קפה"

|

"אני אוהב לשrox גשרים"

מרחוק פשוט יגיד שני המשפטים קרובים, כי הם חולקים מילים דומות.

אבל הכוון שלהם שונה לגמרי.

הם לא מתארים את אותו רעיון.

זו בדיקת הסיבה שמודלים מודרניים משתמשים הרבה יותר על **חזיות** ופחות על מרחק.

מה בעצם מודדים בחזיות?

כשנוי וקטוריים מצבעים לאותו כיוון
(גם אם יש ביניהם מרחק גדול)
החזיות ביניהם קטנה.

כשנוי וקטוריים מצבעים לכיוונים שונים
החזית גדולה.

במילים פשוטות:

החזית מספרת למודל אם שני אובייקטים "נעימים סביב אותו רעיון".

זה נכון לטקסט, למשתמשים, למוצרים ולתמונהות.

למה זה כל כך חשוב בעולם ה-AI?

כפי רוב המשמעות שאנו מזהים בעולם היא לא "כמה זה דומה פיזית", אלא **כמה זה דומה רעיונית**.

וקטורים שמתקרבים בכוון גם כהם לא קרובים למרחק מספרים למודל: "זה דומה במעטות".

זה מה שמאפשר למודלים להבין ש:

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

בחילק הבא נראה איך משתמשים בזווית זו כדי למדוד דמיון בין שני משפטיים בפועל.

AIR MODDEDIM DEMION BIN SHNI MASHPATIM

עכשו כהרעין של דמיון ככיוון ברור, אפשר לעבור לשאלת המעשית:

AIR MACHSHAVIM BAFUOL AT HADMION BIN SHNI MASHPATIM?

המודלים המודרניים לא בודקים אם שתי מילימ זחות או דומות בצליל.

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

מה זה דמיון קוסינוס?

זה ממד שמודד
לא כמה הוקטורים רוחקים,
אלא
כמה הם מצביעים לאותו כיוון.

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

אייר 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 קיבל מספר גובה, נניח 0.93
זה אומר שהמשפטים מצבעים על אותו רעיון.

אם המספר נמוך, נניח 0.18,
המשפטים שונים ביסודן עמוק.

מה עושה np.dot?

(b,a).dot() מחשבת את **המכפלה הסקלרית** בין שני וקטורים.
בפועל זה אומר:
היא מכפילה כל רכיב ברכיב שמתאים לו, ואז לחברת את כל
התוצאות.

לדוגמה:

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

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

המשמעות:

כל שני וקטורים מצבעים בכוון דומה יותר, הערך של dot גדול יותר.

מה עושה np.linalg.norm ?

np.linalg.norm מחשבת את **האור** של הוktor a. זה מספר אחד שמייצג כמה הוktor “רחוק מהמרכז”.

לדוגמה:

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

המשמעות:

ווקטורים גדולים או קיצוניים יקבלו נורמה גדולה יותר.

עכשו הדוגמה של cosine similarity הופכת לברורה

```
import numpy as np

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

כמה הכוונים שלהם דומים #
אורק של הוktor הראשון #
אורק של הוktor השני #

וקטורים שמצבעים אותו 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 המשפטים.

3. הוא רגיש להקשרים

אםשתי מילים מופיעות תמיד באותו הקשר, הן יקבלו כיוון דומה, גם אם המספרים שונים.

4. הוא עמיד לשינויים קטנים בדעתה

תוספת של מילה או שינוי ניסוח קל לא מרסק את הדמיון.

לכן זה המדרד הנפוץ ביותר בעולם של

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

בחלק הבא נחבר את כל זה לעולם ה-*embeddings*, כדי שם הרעיון של זוויות מתחילה לקבל משמעות אמיתי בתפקידים מודרניים.

חיבור ישיר לעולם ה-**Embeddings**

בשלב זהה כל מה שלמדנו על וקטורים, זווית ודמיון מתחבר לרעיון המרכז שמניע כמעט כל מודל מודרני.

הוא וקטור שמייצג משמעות.

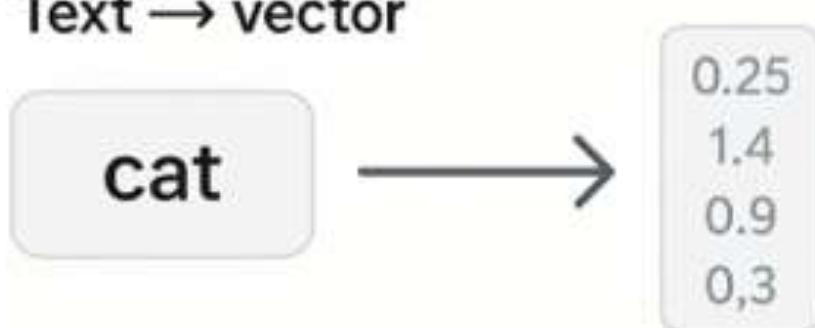
המחשב לא מבין מילים, משפטים או תמונות.
הוא מבין רק מספרים.

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

כדי להבין את זה בצורה ברורה, נתחל בהמחשה חזותית.

הוא וקטור שמייצג משמעות Embedding

Text → vector



באյור זה רואים איך מילה עוברת המرة לוקטור פשוט.

הוקטור לא מתאר אותן.

הוא מתאר משמעות.

דרך מספרים.

AIR **Embeddings** ממקמים משמעות במרחב

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

לדוגמה

מילים כמו חתול גור וחיה יהיו קרובות彼此. ומילים כמו דפדן ענן וצבע יהיו רחוקות מהן לגמרי.

Embeddings as points in space



המרחקים קטנים חשובים. הכוון הוא מה שמספר למודל מה דומה למה.

AIR **כיוון קבוע דמיון**

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

אם הזווית קטנה
המשפטים מייצגים רעיון דומה.

אם הזריות רחבה
הן הולכים לכיוונים שונים.

Similarity by direction — cosine similarity



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

דוגמה אחת שמאגדת את הכל

נניח שיש לנו Embedding של המשפט החתול קופץ על השולחן.

ונרצה למצוא משפטים דומים.
המודל לא יחפש מילים דומות.
הוא יחפש וקטים שמצביעים לאותו רעיון.

הוא ימצא משפטים כמו
הגור טיפס על הספה
כי הם הולכים לאותו כיוון רעיון.

משמעותם שמדוברים על טכנולוגיה או מג אוויר יהו רחוקים לחולוטין.
הכוון שלהם אחר.

למה Embeddings הם שכבת היסוד של מודלים מודרניים

כי הם מאפשרים לייצג כל דבר בעולם
מילאים

משמעותים

משתמשים

מושרים

תמונהות

משמעותים

בצורה שהמודל ידע 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 תמונהות

ניתוח משמעותים

ועוד הרבה מנגנים שתרגמים עולם אמיתי לוקטורים.

פרק 8 – פונקציות – איך מודל חישוב

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 הכל.

מודלים יכולים להיות מורכבים, עמוקים ורבים שכבותיים, אבל הבסיס נשאר זהה לחלופין:
פונקציה.

משהו שמקבל משהו אחד ומחזיר משהו אחר.

מה זה אומר בפועל?

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

המודל ממפה את הקלט לפט.

למשל

- וקטור של משפט → סנטימנט חיובי או שלילי
- תמונה → תג “כלב”, “חתול” או “אדם”
- נתוני משתמש → סיכון לעזיבה

- **וקטור תכונות → מחיר חזו'**

מאחורי כל זה לא עומדת "הבנה" של העולם, אלא **מייפוי מתמטי**.

למה חשוב לראות מודל פונקצייה?

כי זה מפשט את כל התמונה.

אין קסם, אין אינטואיציות נסתרות.

יש דרך אחת שבה המודל פועל:

קלט כניסה ← המודל מעבד אותו ← פלט יוצא.

הfonקציה עצמה היא אוסף של חוקים שהמודל למד מהדата.

זה בדיק מה שմבדיל בין מודל "טוב" למודל "חלש":

איזו פונקציה הוא למד.

ומה שמשמעותו יותר

המודל לא רק מנסה להחזיר פלט.

הוא מנסה להחזיר **פלט נכון**.

כזה שמקטין את הטעות שלו מהעולם האמיתי.

זה מוביל אותנו לנושא הבא:

כמעט כל הפונקציות שמודלים לומדים אפשר לציר כעוקמה.

ומודלים תמיד רוצים להגיע לחלק הנמוך של העוקמה.

בחולק הבא נבין מה זו "עוקמה" ולמה מודלים עושים כל מה שהם

יכולים כדי לרדת בה.

מה זה "עוקמה" ולמה כל מודל מנשה לרדת בה

כשאומרים שמודל "לומד", בפועל קורה דבר הרבה יותר פשוט: הוא מנשה למצוא **מקום נמור** על עוקמה מתמטית שמייצגת את הטעות שלו.

ואז מגיע רגע שבו הכל מתחבר:
המודל לא מחפש תשובה. הוא מחפש מינימום.

מה זו בכלל "עוקמה"?

עוקמה היא דרך לייצג את הקשר בין קלט לבין טעות.
לכל קלט שהמודל בוחר, הוא מייצר פלט.
פלט זהה מושווה למציאות, והפרש ביניהם הוא הטעות.
שמצירים את הטעות זו על גרפ, מתקבלת עוקמה.
נקודה נמוכה על העוקמה אומרת "מעט טעות".
נקודה גבוהה אומרת "רבה טעות".

גם אם זה לא נראה כמו גרפ שאתה רואה בבית ספר,
זו אותה אינטואיציה:

יש מקום גבוה, ויש מקום נמור.

המודל רוצה לרדת למקום הנמור.

למה בכלל יש "עוקמה"?

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

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

למה חשוב להבין את זה כמפתח?

כיבלי ההבנה הזה
למיידת מכונה נראה כמו קסם שחור.
אבל עם ההבנה הזאת
כל התהילה נהיה מאוד הגיוני:

- יש עקומה שמודדת טעות
- אנחנו רוצים להיות כמה שיותר נמוך
- כל צעד של המודל הוא ניסיון להתקرب לנקודה נמוכה יותר

זו כל תורת האימון ב-**TLW** במשפט אחד.

עקוּמָה לא חִיַּבָּת לְהִיּוֹת דּוּ-מִמְדִית

כשאתה חושב על "עקוּמָה", אל תחשוב על גראף חלק שמצויר על לוח.

במודלים אמיתיים העקוּמָה נמצאת למרחב של אלפי או מיליון מישתנים. אבל האינטואיציה נשארת בדיק אותה אינטואיציה: יש מקומות גבוהים ויש מקומות נמוכים.

ולכן כל המודלים בעולם מגיעים לאותה מטרה:
להקטין טעות.

וכדי להקטין טעות, הם חייבים לרדת על העקוּמָה. בחלק הבא נבין למה "מינימום" הוא כל כך חשוב, ומה בעצם קורה כשהמודל סופר סוף מגיע לשם.

מה זה בכלל מינימום

אחרי שהבנו שהמודל נעה על עקומה של טעות ושוואף לרדת כל הזמן,

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

זו המילה שכל מפתח שומע שוב ושוב באימון מודלים.

אבל מה זה בעצם אומר?

מינימום הוא המקום שבו המודל טועה הכי מעט

כשמציררים את הטעות בעקומה,

יש נקודת אחת (או כמה נקודות) שבה הטעות היא הצעירה ביותר.

זו נקודת המינימום.

שם המודל מרגיש "נוח":

הוא מפיק פלטים שמתאימים לדאטה טוב יותר מאשר בכל נקודת

אחרת על העקומה.

במילים פשוטות:

מינימום = המקום שבו המודל עובד הכי טוב על הדאטה שראתה.

אבל לא כל מינימום הוא אותו דבר

יש שני סוגי 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 הći נמוכה בכל העקומה. זה המקום שבו המודל מגיע לביצועים הכי טובים שהוא מסוגל. במודלים גדולים, מרחב העקומות מורכב מאוד, ולכן הבדיקה זו חשובת – גם אם אנחנו לא מציררים 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 לפיה הכוון שמקטיין 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:

- לעיתים האימון "נתקע"
- לעיתים המודל לא משתפר מעבר לנקודת מסיוםת

- לעיתים שינוי קטן בדעתה מזיז את כל המינים
- לעיתים הירידה איטית מדי או מהירה מדי
- ולפעמים יש כמה פתרונות טובים, לא רק אחד

ברגע שבניים שמודל רק מחפש נקודה נמוכה על עוקמה ולא "מנסה להבין" משהו עמוק יותר הכל יהיה ברור.

זה הרגע שבו לומדים להבין את המודל יוצרה מתמטית, ולא כקופסה שחורה.

פרק 9 – שיפוע – המנווע של הלמידה

מה זה שיפוע בלי להגיד "נגזרת"

כדי שמודל יוכל לרדת בעקבות הטענות, הוא צריך לדעת **לאן ליכת**.

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

התשובה היא: **SHIPOU**.

ובוא נסביר אותו בצורה הכיו אנושית שאפשר, בלי להשתמש במילה "נגזרת".

SHIPOU הוא פשוט "כמה תלול השטח מתחת"

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 בבדיקה רק על עוקמה של טעות במקום על גבעה אמיתי.

איך השיפוע עוזר למודל?

הוא נותן לנו מודל תשובה לשאלת הcoli חשובה במהלך האימון:
“אם איזוז קצת שמאל או ימינה – האם הטעות תגדל או תקטן?”

זה כל הסיפור.

המודל בודק את השיפורו במקום שבו הוא נמצא:

- אם השיפור גדול – צריך ליזוז מהר
- אם השיפור קטן – להתקדם בעדינות
- אם אין כמעט שיפור – אתה קרוב לנקודת מינימום

למה חשוב להבין את זה?

כי שיפור אומר לנו יותר מכל דבר אחר:

כמה טעות יש כאן, ומה יקרה אם איזוז.

אפשר לחשב על זה כעל “מד טעות מקומי” שמכoon את כל תהליכי הלמידה.

המודל לא רואה את כל העקומה.

הוא רואה רק את האזור שהוא נמצא בו, ואת השיפור שסביר לו אותו קידימה.

זו הסיבה שבפועל כל תהליכי הלמידה נשען על הרעיון הזה:

לחשב שיפור וליזוז לכיוון שמקlein טעות.

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

למה שיפוע אומר לנו "כמה טעות יש כאן"

עכשו כשהרעיון של שיפוע ברור, ניכנס למשהוعمוק יותר: למה בכלל השיפוע מייצג את כמות הטעות במקום מסוים? ומה מודלים משתמשים דווקא עליו כדי לדעת איך להתקדם? התשובה פשוטה יותר ממה שנדרשה.

SHIPOU GBOH = AZOR UM HORBA TEUOT

כשSHIPOU CHD, זה אומר שהעוקמה משתנה מהר. כלומר: הטעות קופצת בצורה ממשמעותית כשזדים קצת לצד זה או אחר.

אם השטח תלול, אתה יודע שאתה עדין רחוק מפתרון טוב. זה אומר שהמודל נמצא באзор שבו הוא עדין טעה הרבה. וכן צריך לבצע תיקון חזק וברור.

זה כמו להגיד:

"משהו פה ממש לא מדויק, תזוז חזק לכיוון הירידה."

SHIPOU NAMOR = AZOR UM MEUT TEUOT

כשיעור קטן, העוקמה כמעט שטוחה.
זה אומר שהטעות לא משתנה ממשמעותית כשזדים קצת ימינה או
שמאלה.

זה רמז למודל שהוא מתרב למקומות טוב.
הוא לא צריך לקפוץ, אלא לבצע צעד קטן ועדיין.
כשיעור כמעט אף – המודל קרוב לנקודת מינימום.
אין סיבה לוזד הרבה, כי כל שינוי קטן כבר לא יספר הרבה.

שיעור נוטן כיוון וגם עוצמה

זו הנקודה הקritisית:

לא רק הכיוון של השיעור חשוב,
אלא גם **העוצמה שלו**.

- שיעור חיובי אומר שהטעות גדלה
- שיעור שלילי אומר שהטעות יורדת
- הגודל של השיעור אומר כמה כדאי לוזד

זו הסיבה שהשיעור הוא הכלי המרכזי שמודלים משתמשים בו.
הוא מסביר גם **לאן** לוזד וגם **כמה** לוזד.

איך זה נראה ביום יום של מודל?

בכל צעד אימון, המודל בודק:

- מה השיפורע כאן
- האם אני עולה או יורדת
- כמה מהר כדאי לי לזרז

ובצתות השיפורע הוא יודע

אם הוא צריך לבצע תיקון גדול
או תיקון קטן

או אולי לעצור כי הוא ממש קרוב לנקודת טובה.

זו כל התבוננה של תהליכי הלמידה:

למדוד את השיפורע
ולהתקדם בהתאם.

בחלק הבא נראה הדוגמה פשוטה ויזואלית שמחברת את כל הרעיוןנות האלו לgraf ברור שקשה לטעות בו.

הדגמה קצרה עם גרף פשוט

כדי שהרעיון של שיפוע והפוך למוחשי, נבנה רגע את הדוגמה פשוטה ביותר ביותר:

עוקמה בצורת U.

זו עוקמה שמייצגת טעות: גבואה בצדדים, נמוכה במרכז.

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)
```

(*"מיקום על העוקמה"*)
`plt.xlabel("טעות")`
`plt.title("עקומת טעות פשוטה")`
`plt.grid(True)`
`plt.show()`

אם תריץ את הקוד זהה, תראה גרף עם צורה ברורה מאוד: העות גבואה בקצוות, נמוכה בנקודת אחת במרכז – באיזור של $x=3$.

זו נקודת המינימום.

עכשו נראה מה קורה עם השיפוע.

מה המודל "מרגיש" לאור העקומה?

נניח שהמודל נמצא בנקודת גובהה מצד שמאל.

שם השיפוע **גדול מאוד**.

זה כמו לעמוד על מדרון תלול – ברור למורי لأن צריך לזוז.

אם נלך לנקודת גובהה מצד ימין,

נרגיש שוב מדרון תלול, אבל בכיוון הפוך.

המודל יודיע: "תזוז שמאלה, אתה עולה".

באזור המרכז, ליד המינימום, העקומה כמעט שטוחה.

הSHIPוע קטן מאוד.

המודל מבין שהוא קרוב לפתרון טוב

וכיריך לנوع באיכות או אפילו לעצורך.

איך מודל היה "מיישם" את זה?

נוסיף קטע קוד קטן שמדמה צעד אחד של תהליכי הלמידה:

```
# נקודת 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 כשתרץ את זה?

בכל צעד

המודל יוזד לכיוון הנכוון

ובקפיצה שמתאימה לגודל השיפוע.

כל שהוא מתקrab למינימום

הצעדים יהיו קטנים ועדים יותר.

למה הדוגמה זו חשובה לר כמפתח?

כי היא מתארת בצורה אנושית את כל תהליך הלמידה:

1. יש עיקומה שמייצגת את הטעות

2. השיפוע אומר למודל לאן לוזד

3. כל צעד הוא ניסיון להקטין טעות

4. המודל נע עד שהוא מתקrab לנקודה שבה השיפוע כמעט אף

אין קסם.

אין החלטות נסתרות.

רק סדרה של תיקונים קטנים שמובוססים על השיפוע המקומי.

פרק 10 – הלמידה עצמה

רעיון ירידת המפל

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

עכשו אנחנו מחברים את כל החלקים יחד לתהילר שהופך מודל “לא מאוכן”

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

זה השלב שבו המודל מפסיק “לבחות” בDATA
ומתחל ממש **לŁmod**.

ירידת מפל – המנגנון שמוריד את הטעות

Gradient Descent הוא פשוט רעיון מתמטי שمدמה ירידת על מדרון.

אם אתה עומד על גבעה ורוצה להגיע לבסיס שלה,
הדרך הטבעית ביותר היא לזרז בכל פעם **לקראת הכיוון שבו**
הקרקע יורדת.

אותו עיקנון בדיק 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 התיליות של השטח.
כשער השיפוע גדול, המפל חד יותר.
כשער השיפוע קטן, כבר כמעט הגענו למשטח שטוח ליד
המינימום.

במילים פשוטות:

המודל גולש לאורק העקומה עד שהוא מתקרב לנקודה שנואה לו.

אינטואיציה קצרה

אם אתה נמצא במקום גבוה על העקומה
(פעות גדולות)
הSHIPוע גדוול
ו**המודל מבצע צעד גדול**.

אם אתה קרוב יותר לאזור טוב
הSHIPוע קטן
ו**המודל מבצע צעד עדין יותר**.

זו התאמה דינמית שמאפשרת למודל להתכנס בצורה יציבה.

למה זה הגיוני כל כך?

כי במקרה לנסות “לנחש” איפה המינימום נמצא,
המודל משתמש במידע המקומי שהוא כבר ידוע:

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

זו הבדיקה הקritisית:
לא מחפש פתרון מושלם.
הוא שואף לדפוס טוב יותר בכל צעד קטן.

זה מה שהופך אותו לאחד הרוינוות החשובים ביותר בעולם ה-AI.

למה המודל זו “נגד” השיפוע

אחד המשפטים שמבלבלים מפתחים בתחום הדריך הוא:
המודל זו נגד השיפוע.

למה נגד?

למה לא עם השיפוע?

הרי באופן אינטואיטיבי, אם השיפוע מצבייע לכיוון מסויים – לא כדאי
ללכת אליו?

כאן מגיע הנקודה המהותית.

השיפוע מצבייע לכיוון שבו הטעות גדולה

כשיעור חיובי, זה אומר שם נזוז מעט קדימה בכיוון זהה –
הטעות גדל.

כשיעור שלילי, זה אומר שם נזוז בכיוון זהה – הטעות תרד.

זה הופך את המנגנון פשוט מאוד:
המודל תמיד זו לכיוון שמקטין טעות.

וכדי להקטין טעות, הוא חייב לזרז **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 ללקת בדיק 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 זו שמאליה.
אם השיפוע שלילי, הוא זו ימינה.
אם השיפוע קטן מאוד, הוא מאט ומבצע שינוי קטן ועודין.

זו הסיבה ש Gradient Descent הוא תחילה 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 זו,
ומה קורה כשצעד הלמידה גדול מדי או קטן מדי.

מה קורה כאשר הלמידה גדול מדי

כל צעד Gradient Descent עושה מושפע מפרמטר אחד חשוב במיוחד: **קצב הלמידה** (Learning Rate).

זה המספר שקבע כמה גדול יהיה כל צעד שהמודל מבצע כשהוא זו נגד השיפוע.

אפשר לחשב עליו בתרור "כמה חזק אני מסובב את ההגה בכל פעם".

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

קצב למידה גדול מדי – המודל " קופץ" מעל המינימום

כאשר קצב הלמידה גבוהה מדי, קורה שהוא 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

בליל להתיצב.

זה מוביל לבעה קלאסית שנקראת "אוסילציות": המודל פשוט לא מצליח להתיצב במקום טוב.

למה זה קורה?

כי השיפור רק אומר האם לעלות או האם לרדת.
הוא לא אומר כמה.

אם קצב הלמידה גדול מדי,

הכוון נכון

אבל העוצמה מוגזמת.

זה כמו לנשות לעזר על מדרגה
כשהתיהם יורד בריצה מהירה מדי – אתה פשוט עף קדימה.

קצב למידה קטן מדי – התקדמות איטית או תקיעה

גם הצד השני בעיתוי.

אם הצעדים קטנים מדי,

המודל מתקדם לאט בצורה מתסכלת

ואפיוּן עולל "להיתקע" באזור שבו השיפור חלש.

הוא לא מזמן אבל גם לא מצליח לעבור את המדרון.

אייזון נכון

הקסם קורה כשהקצב מאוזן:

- מחיר מספיק כדי לרדת בעקבותה
- לא מהיר מדי כדי לא לקפוץ מעל המינימום
- לא איטי מדי כדי לא לבזבז אלף צעדים

זו האמנות שמסתתרת מאחורי אימון מוצלח.

בחלק הבא נראה הדוגמה קצרה בקוד שסמחישה איך זה נראה כשהקצב נכון, גדול מדי או קטן מדי.

הדוגמת קוד שמרת ירידה למינימום

כדי לראות את כל הרוונות של מזנו פועלים ביחד, נבנה הדוגמה קטנה של Gradient Descent על עוקמה פשוטה.

אין כאן מודל אמיתי, אלא רק פונקציה טווח אחת, אבל העיקרון זהה לחילופין למה שקרה באימון מודלים גדולים.

נשתמש בפונקציה טווח מהסוג שכבר רأינו:

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

הפונקציה זו מגיעה למינימום כשה-x שווה 3.

עכשו נכתב 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 התחלת רוחקה 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 כשריצים את זה?

1. **הערך של x מתקרב בהדרגה ל-3**, המינימום של הפונקציה.
2. בכל צעד, השיפוע מכוען את המודל לכיוון הנכוון.
3. כשהתקרב לנקודת המינימום, הצעדים נהיים קטנים ועדים.
4. ההתקדמות לא בקפיצות אלא בצורה יציבה ורציפה.

זה כמעט בדיק מה שקרה באימון של מודל אמיתי.
ההבדל היחיד הוא שבמודלים גדולים לא עובדים על מספר אחד
אלא על מיליון פרמטרים בבהא אחת.

למה הדוגמה זו כל כך חשובה?

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

המודל מחשב שיפוע זו נגדו

**וחזר על זה שוב ושוב
עד שהוא מתרב למועדן שבו הטעות הći נמוכה.**

אין קסם.
אין כפיצות.
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. והכל באמת מתישב רק כשוכותבים קוד.

זה בדיקת התפקיד של הפרויקט הזה:

לחבר את הבנה של מתמטיקה אינטואיטיבית עם תלס – כמה שורות פיתון.

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

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

הפרויקט הזה נותן לך שלושה דברים:

1. יישום אמיתי של כל הרעיונות שלמדו

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. תשתיות מעולה להמשך הסדרה,
במיוחד בספר הבא על מתמטיקה יישומית, שבו נעמיק בנגזרות,
מטריצות ותנועות מורכבות יותר.

איך בינוי הפרויקט?

בכל תרגיל יש:

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

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

נתחיל מהתחלה

בחלקים הבאים נכתוב את כל התרגילים לפי הסדר:

1. ממוצע וסטיית תקן

2. הסתברות מותנית

3. ביסס על טבלה

4. נורמה

cosine similarity.5

Gradient Descent.6

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

תרגיל: ממוצע וסטיית תקן

תרגיל קצר שמאפשרoir איר להבין "מרכז" ו"פיזור" בעזרת מספר שורות פיתון.

מה עושים כאן?

ניקח רשימה של מספרים פשוטה, נחשב את **הממוצע**, את **סטיית התקן**, ונראה איך שני המספרים הללו ביחיד 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 התקן #
std = np.std(data)

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

מה צריך לראות?

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

סטייה גדולה מציביעה על פיזור רחוב, סטייה קטנה על נתונים מרכזים.

משימה קצרה

שנה את הרשימה לנתונים "קיצוניים" יותר, למשל:

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

ושם לב איר סטיית התקן " קופצת" בגלל הערך החיריג.

תרגיל: הסתברות מותנית

תרגיל זה מחזק את הרעיון של "הסתברות בתוך הקשר".

במוקם לבדוק את **השכיחות הכללית של אירוע מסוים, אנחנו**

בודקים את השכיחות שלו בתוך קבוצה מצומצמת.

הדרך לפתרו את זה מתחילה בבנייה טבלה של אירועים.

نبנה טבלה פשוטה של אירועים, ואז נחשב את ההסתברות

המוחנית:

כמה מתוך האירועים בקבוצה מסוימת מקיימים תנאי נוסף.

קוד בסיסי

נניח טבלה פשוטה של הודעות:

```
import numpy as np

# נתוניים: [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]

# בتوز' ספאם הסתברות מותנית: כמה
p_free_given_spam = with_free[0] / total_spam

# בتوز' לא ספאם הסתברות מותנית: כמה
p_free_given_ham = with_free[1] / total_ham

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

מה צריך לראות?

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

משימה קצרה

שנה את המספרים בטבלה.
נסה לדמות מצב שבו רוב ההודעות הן רגילות אבל עדין free קשור
חזק למספרם.
שים לב איך הסתברות המותנית ממשיכה להראות את הדפוס
הנכון גם כשהיחסים הכלליים משתנים.

תרגיל: ביס על טבלה

תרגיל זה ממשיר יישור מהתרגיל הקודם.
נ賓אריך משלבים בין התמונה הכללית לבין הקשר המקומי כדי
לקבל הערכה אמיתית של "כמה סביר שזה נכון".
זה בדיק מה שביס עשה, אבל בלי נסחאות מורכבות.
רק טבלה, ספירה, וחישוב אחד קטן.

מה נעשה כאן?

נחשב את הסיכוי שהודעה היא ספאם בהינתן שמוופיע בה free,
באמצעות הטבלה פשוטה.

קוד בסיסי

נשתמש באלה טבלת שכיחיות:

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

# ביחס לספאם בהינתן free
p_spam_given_free = (p_free_given_spam * p_spam) / p_free

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

מה צריך לראות?

- זה לא "קסם". זה פשוט דרך לשלב בין מה שקרה בעולם בכלל לבין מה שקרה בתחום ההקשר המעניין אותנו.
- אם free מופיע בעיקר בספרם, ביס משקף לבדוק את זה.
- שינוי קטן במספרים יכול להזעיא תוצאות שונות, מה שמחזק את הרעיון: **הקשר משנה הכל**.

משימה קצרה

שנה את המספרים כך שיהיו הרבה יותר הודעות רגילים בכלל, אבל free עדין יופיע ברוב המוחלט של ה Hodoutes הספרם. בדוק איך זה משנה על תוצאה ביס, ואיך ההקשר מנתח על התמונה הכללית.

תרגיל: נורמה

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

מה עושים כאן?

נחשב את הנורמה של כמה וקטורים שונים, ונראה איך אורך הווקטור 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("v1:", norm_v1)

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

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

מה צריך לראות?

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

משימה קצרה

נסה ליצור שני וקטורים:

אחד עם ערכים קטנים ואחד עם ערכים קבועים.
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 נמדד דמיון בין שני וקטורים לא לפי המרחק שלהם, אלא **לפי הזווית** ביניהם.

זה בדיקת הכלים שמודלים משתמשים בו כדי לבדוק אם שני מושפעים "מצבעים" על אותו כיוון רעוני.

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

מה צריך לראות?

- בין 1st ל-2nd הדמיון כמעט 1
כי הם מצביעים בדיק לאותו כיוון.
- בין 1st ל-3rd הדמיון נמוך
כי הם מצביעים לכיוונים שונים.
- המדריך זה לא מתייחס למרחק.
הוא מתייחס **רק לזרות** בין הוקטורים.
ולכן הוא כלי מצוין לניתוח משמעות בטקסט.

משימה קצרה

נסה ליצור שני וקטורים שונים לחלוטין למרחוק,
אבל בעלייחס 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 הכוון זהה למרחות שהגודל שונה
לגמר.

תרגיל: צעד Gradient Descent

תרגיל זה מחבר את כל מה שלמדנו בפרק 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?

נגיד פונקציית טעות אחת, נמצא את השיפוע בנקודת מסויימת, וביצע 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 התחלת
x = -1

learning_rate = 0.1
```

```

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

# Gradient Descent
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?

- הfonקציה מוגדרת כך שהמינימום שלו הוא $x = -3$
- השיפוע בנקודה $-1 = x$ הוא חיובי
- لكن Gradient Descent יזיז את x בכיוון השילוי (נגד השיפוע)
- אחרי הצעד x , מתקרב ל- -3
- ועל הדרך, גם הטעות קטנה

זה בדיק איך שמודלים גדולים יורדים בעקבות 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 מאדן.

פרק 12 – איך כל זה מתחבר ל-**LM** ול-**NLP**

למה כל הדברים שלמדנו חוזרים בכל מודל

בערתך ארוכה.

וקטורים, נורמה, מרחק, זווית, הסתירות מותנית, ביס, שיפוע, ירידת מפל.

לכורה זה הרבה נושאים שונים.

אבל האמת הפשטה היא שכל הדברים האלה חוזרים בכל מודל מודרני, קטן או גדול.

הסיבה מאוד ברורה:

מודלים לא “مبינים” 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?

- כamodel 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 **עקבות טעות**.
- כל תהליכי הלמידה מבוססים על **SHIPOU**.

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

כל שהמודלים גדלים, היסודות האלה לא נעלמים – להפר.
הם הופכים להרבה יותר חשובים.

למה זה שימושי עבורך כמפתח?

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

ברגע שמשהו 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 מהשיטה.

איפה פוגשים אותם שוב בספרון הבא

כל הרוינות שלמדת כאן הם לא רק בסיס טוב.

הם **השלב הראשון** לקרוא הספר הבא בסדרה, שעוסק במתמטיקה יישומית למפתחי AI.

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

מה חוזר ועמוק שם?

• **קטורים ומרחבים גדולים**

ראינו כאן וקטורים כסדרה של מספרים.

בספרון הבא נרחב איר וקטורים חיים בתחום מרחבים של عشرות אלפי ממדים

ומה קורה כשמרחקים וזרויות משתנים שם.

• **מטריצות**

הפעם דיברנו עליהם רק ברמת אינטואיציה.

בספרון הבא תראה איך כל שכבה במודל היא בעצם מכפלה מטריצונית

ואיך שילוב של מטריצות יוצר את כל מה שמודלים עושים.

• **שיפועים מורכבים**

דיברנו כאן על שיפוע של מספר אחד.

בספרון הבא נפגש שיפורים שמתפזרים על מיליון פרמטרים ונראה איך מודלים גדולים מחשבים אותם ביעילות.

• **Gradient Descent אמיתי**

בפרק זהה ראיינו רק ירידה על עוקמה פשוטה. בהמשך נראה גרסאות מתקדמות של ירידת מפל כגון: Momentum, Adam ואופטימיזציות שבעולם האמיתי אי אפשר להסתדר בלבדיהן.

• **Embeddings عمוקים**

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

מה יכול עלייר כשהמודלים נהיים גדולים ומורכבים

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

ואז מגיעה השאלה:

כשהכל נהיה כל כך מורכב,
איך מפתח יוכל עדין להבין מה קורה בפנים?

התשובה פשוטה:

מבינים רק אם מכירים את היסודות.

החדשנות הטובות הן שכשתפשת את אבני הבניין של הספרון זהה, יש דברים שמתחלים בהתאם להיראות הרבה יותר פשוטים.

וקטורים כבר לא מסתוראים

גם אם וקטור Embedding הוא ברוחב 4096, זה עדין אותו רעיון של רשימת מספרים שמייצגת ממשמעות. העיקרונו נשאר זהה, רק הסקיל משתנה.

חוויות ומרחקים הופכים לכלי עבודה אמיתי

כשהאתה מבין איך דמיון קוסינוס עובד, אתה יכול לזהות למה שני משפטיים מתבלבלים, ומה חיפוש טקסט מחזיר תוצאות מסויימות, ולמה מודל “יישר קו” בין משמעותם לא קשורות.

הcoil מתחילה ונגמר בדווית בין וקטורים.

SHIPOU-Gradient Descent כבר לא מפחדים

אימון של מודל ענק נראה כמו קסם עד שאתה זכר:

הוא עדין רק מחשב שיפורים

וזד נגדם

מיליוני פעמים.

ברגע שמבצעים את המנגנון הקטן,

המנגנון הגדל כבר לא מסתורי.

הסתברות מותנית וביס עוזרים לקרוא תוצאות

כשמודל טעה,

הוא לא "טיפש",

הוא פשוט למד באופן שמתאים להסתברויות ולדפוסים שהוא ראה.

ברגע שאתה יודע איך ביס חושב,

אתה מבין למה טוויות מסוימות הן צפויות.

נורמה ומרחיק מסבירים מתי נתון קופץ החוצה

כשמשהו "לא נראה הגיוני",

זה בדרך כלל משתקף באורך חריג של וקטור

או במרחב גדול מדי במרחב.

העינ שיר כבר יודעת לזהות את זה.

מה כל זה נותן לך כמתכנת?

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

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