מבוא ללמידת מכונה- סיכום:

שקד שרון.

בינה מלאכותית:

בעיות חיפוש:

נפוצות בעזרת עצים/ רשתות.
נמפה את הבעיה האמתית לגרף/ עץ שיתאר אותה.
גרף מכיל: צמתים (vertices/ nodes), חיבורים (edges), דרכים (Paths) ומחירים (Costs).

הגדרות בסיסיות:

סוכן: חי בסביבה ומגיב בהתאם לתנאים בה.

סביבה: הסביבה בה "חי" הסוכן שלנו.

סוגי סביבה:

1. Fully vs. Partially observable:
כל הנתונים ידועים (שח) / חלק מהנתונים ידועים (פוקר).
2. Deterministic vs. non-deterministic:
סביבה דטרמיניסטית (שח) / לא ( ). דטרמיניסטי: אם ביצעתי פעולה- ודאי שהיא תקרה.
3. Episodic vs. Sequential:
סדרתי (מענה טלפוני) / מתמשך (שח). סדרתי: הפעולה לא משפיעה על העתיד.
4. Static vs. Dynamic:
סטטי (שח) / דינמי (וויז). דינמי: יכולים להיות שינויים בסביבה שלא נצפה אותם.
5. Discrete vs. Continuous:
בדיד (שח) / רציף (וויז).

מצב יעד/מטרה (goal) מטרת הסוכן. נצטרך להגדיר ולאפיין את היעד של כל סוכן. (לדוגמא פתרון פאזל).

מצבים (states): כלל המצבים והאפשרויות בהן יכול להיתקל הסוכן.

מצב התחלתי (initial state): המצב בו אנחנו מתחילים.

פעולות (actions): הפעולות המותרות במשחק/ בסביבה.

עלות (cost): כמה צעדים "עלה" לנו כדיי להגיע לפתרון. (כך נוכל להשוות בין סוכנים/ בין דרכים).

פתרון (solution): רצף הפעולות שנדרשנו לבצע מהמצב ההתחלתי עד מצב הסיום. (לפעמים יספיק לנו רק מצב הסיום שנמצא, ולא נידרש לדרך).

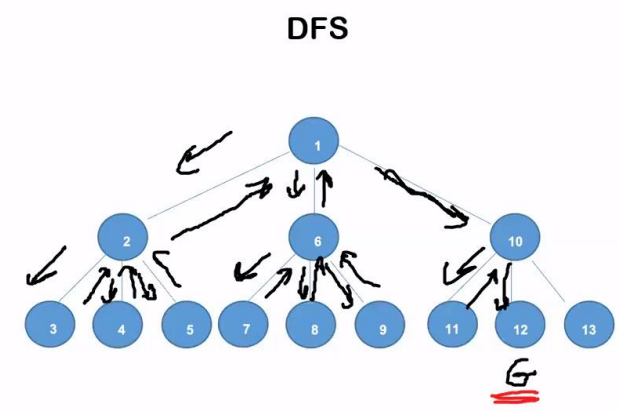
פתרון אופטימלי- רצף הפעולות עם העלות הנמוכה ביותר שמצאנו- בדרך כלל עם הדרך הקצרה ביותר. (לא לבלבל עם פתרון שלקח הכי פחות זמן למצוא!!!).

פרונטיר (frontier): "מחסן מצבים". מכיל מצב התחלתי. אם הפרונטיר ריק- אין פתרון לבעיה.
אחרת: נבדוק את המצבים המוכלים בפרונטיר אם הם מצב יעד/ אפשר לפתח מהם מצבים נוספים ולהכניס לפרונטיר.

סוגי חיפושים ושימוש בפרונטיר:

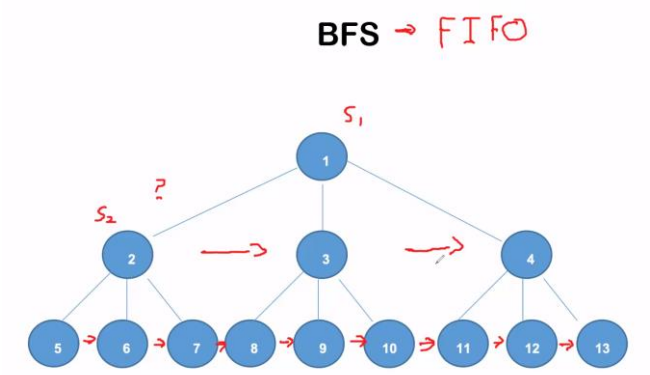
חיפוש עיוור (uninformed search)- חיפוש על ידי בדיקה של כלל המצבים (ללא אינדיקציה לדרך הפתרון).

1. **DFS- depth first search : חיפוש לעומק:**
מנהל לפרונטיר= מחסנית(stack). (**LIFO**).
ראשית נבדוק את המצב האחרון שפיתחנו לתוך הפרונטיר.



חיפוש בשיטת דפ"ס לא מבטיח אופטימליות. (אלא מוצא את הפתרון לפי הגדרת סדר פיתוח המצבים). מפתח דרך דרך- ולכן תופס פחות מקום בזיכרון הפרונטיר משיטת בפ"ס.

2. **BFS- breadth first search: חיפוש לרוחב:**
מנהל לפרונטיר= תור (queue) (**FIFO**).
ראשית נבדוק את המצב שפיתחנו הראשון (הכי מוקדם).



בפ"ס עובד בשיטה שטוחה, ולכן מבטיח אופטימליות. מצד שני- ממלא את הפרונטיר בהרבה מצבים אפשריים, ולכן בדרך כלל יתפוס יותר מקום בזיכרון.

3. **ID- iterative deepening:**
שילוב של היתרונות מכל אחד משני סוגי החיפוש לעיל.
החיפוש נעשה בשיטת דפ"ס, אך עם הגבלת גובה בכל איטרציה. בשלב הראשון מחפשים עד עומק 1. לאחר כך עד עומק 2. וכן הלאה.
אם יש פתרון אופטימלי בשלבים הראשונים- נמצא אותו מיד. (יתרון של בפ"ס).
בין האיטרציות הזיכרון הרוחבי נמחק- כך שלא מילאנו את הזיכרון בצורה חריגה. (יתרון של דפ"ס).

<u>חיפוש אינפורמטיבי (informed search)-</u> שימוש ביוריסטיקות -"מצפן" על מנת לייעל את החיפוש ובדיקת המצבים.

פונ' יוריסטיקה- h(n) (heuristic): שימוש בהערכה בהתאם לתנאי ה"משחק" שתיתן לנו עבור כל מצב עד כמה הוא קרוב לפתרון.

שימוש בפונ' יוריסטיקה הוא סוג שימוש חדש בפרונטיר ("מנהל" חדש).

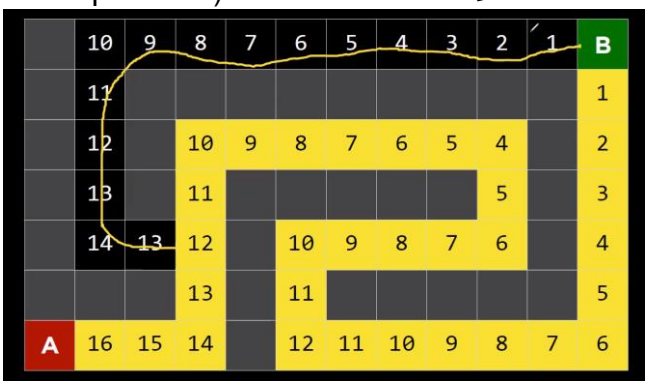
סוגי יוריסטיקות שימושיות:

<u>מרחק מנהטן</u>- סכום מרחק אופקי ואנכי ממיקום המטרה. (פיתגורס). שימושי במבוכים.

#<u>מרחק אווירי</u>- שימושי בניווטים/ וויז לדוגמא.

4. Greedy best first search:

עבור סוכן שמשתמש ביוריסטיקות- עבור כל מצב שאינו מצב היעד, הוא מפתח את הצאצאים וניגש אליהם לפי ההערכה היוריסטית שלהם (ייגש ראשון לזה שהכי קרוב ליעד).

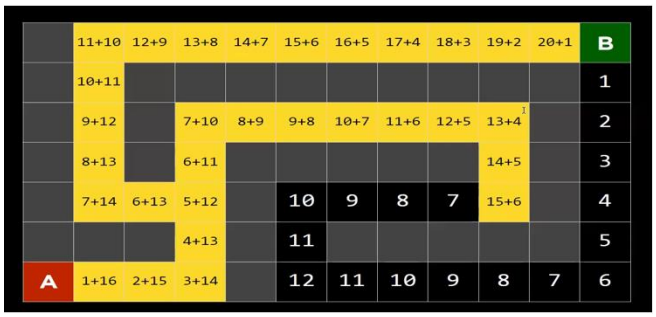


חסרון בשיטה זו- לא מבטיח את הדרך האופטימלית. (יכול לבחור כיוון עם יוריסטיקה טובה יותר שבסוף יתבאר כיותר ארוך).

5. A* search:

מצרף ליוריסטיקה הבסיסית את מרחק הצעדים (דרך) שנעשו ממצב ההתחלה עד למצב הנוכחי, וכך מבטיח אופטימליות.

Evaluation function $f(n) = g(n) + h(n)$



תנאי להבטחת מציאת הדרך האופטימלית בשיטה זו:

היוריסטיקה צריכה להיות **אדמיסיבלית** (admissible) כלומר תמיד היוריסטיקה צריכה להיות קטנה או שווה לדרך האמתית.

יוריסטיקה תיקרא "טובה" אם היא תיתן הערכה לעלות שתהיה בין העלות בדרך האמתית לבין חצי ממנה.

It is a relatively accurate estimator of the cost to reach a goal.

Usually a *"good"* heuristic is if ***½ opt(n)<h(n)<opt(n)***

<u>דוגמא ספציפית</u>: יוריסטיקות ל"פאזל8":

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

1. מספר האריחים שלא במקום.
2. סכום מרחקי מנהטן של כל האריחים מהמקום שלהם.

מנסיון, עדיף את היוריסטיקה השנייה, היא פחות משתמשת בזיכרון מאשר הראשונה.

	Search Cost (nodes generated)		
d	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6
4	112	13	12
6	680	20	18
8	6384	39	25
10	47127	93	39
12	3644035	227	73
14	–	539	113
16	–	1301	211
18	–	3056	363
20	–	7276	676
22	–	18094	1219
24	–	39135	1641

חיפוש מקומי (localized search):

חיפוש בו מעניין אתנו רק המצב שלנו והמצבים השכנים לו (מה שאפשר לפתח ממנו).

לא מעניינת הדרך, אלא רק בירור של מהו מצב היעד. ואיזה צאצא כדאי לפתח כדיי למצוא פתרון אופטימלי לבעיה. (אין שימוש בפרונטיר עם כל המצבים, אלא 'זוכרים' רק את השכנים שקיימים למצב הנוכחי).

State space landscape- שיטת ייצוג של המצבים והערכים לכל מצב.

נגדיר:

גובה עמודה= הערך של המצב. עמודות שכנות= מצבים שכנים.

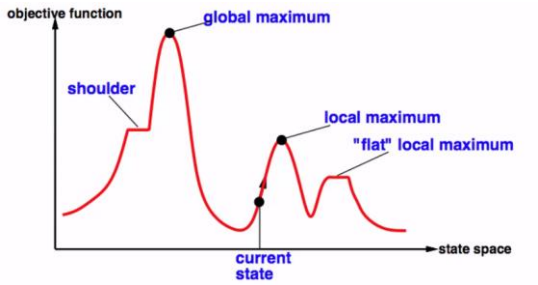
המטרה- למצוא ערך מינימלי/מקסימלי בין המצבים.

קיצון גלובאלי- הכי גבוה/ נמוך מבין כלל המצבים האפשריים.

קיצון לוקאלי- הכי גבוה/ נמוך מבין השכנים שלו.

כתף (sholder)-- ערכים זהים למצבים סמוכים כשבצדדים ערכים גבוהים ונמוכים. (בצד שמאל בתמונה).

רמה (plato) – ערכים זהים למצבים סמוכים כשבשני הצדדים ערך נמוך יותר. (בצד ימין בתמונה).

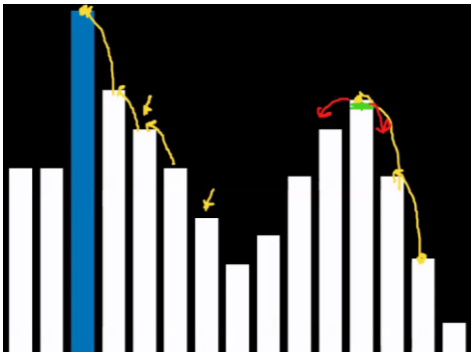


ציר העמודות ייקרא:

Objective function עבור בעיות חיפוש של מקסימום גלובלי.
Cost function עבור בעיות חיפוש של מינימום גלובלי.

אלגוריתמים לחיפוש לוקאלי:

Hill climbing–טיפוס הרים:
אם נחפש לדוגמא ערך מקסימלי- מחפש בין 2 השכנים שלו את הערך המקסימלי מביניהם, ומפתח אותו. כך מתקדם עד שמגיע למצב אשר אין לו שכן גבוה יותר. הגענו למקסימום מקומי.
(כמובן שלא נגיע למקסימום גלובלי תמיד, זה תלוי במצב ההתחלתי שלנו.)



אלגוריתמים (פתרונות) להעלאת הסיכוי למציאת הקיצון הגלובלי: (בפירוט אני מניח שמחפשים מקסימום).

1.steepest- ascent:

האלגוריתם הבסיסי. מתוך השכנים בוחר את השכן עם הערך הכי גבוה.

2. stochastic:

בוחר שכן אקראי מבין השכנים. (אפשר לתת משקל שונה לשכנים עם ערכים גבוהים יותר)

3.first- choice:

בוחר את השכן הראשון שבדקנו (מזכיר FIFO).

4.random- restart:

מוצא פתרון בשיטת טיפוס הרים, מגיע לפתרון, ומתחיל את החיפוש מחדש ממקום אחר.
לבסוף משווה בין ערכי המקסימום הגלובליים שמצא ובוחר את הגבוה מביניהם.
המומלץ.

5.local beam search:

בוחר K שכנים הכי גבוהים, וממשיך איתם את האלגוריתם. משווה בין המקסימום המקומיים שמוצא.

6.simulated annealing:

הסוכן 'מטייל' במרחב המצבים, תוך כדיי שהוא ללוקח סיכונים מדי פעם ולא תמיד בוחר בשכן המיטבי.
בנוסף, יש לנו פונ' טמפרטורה (temperature) ש'מתקררת' עם הזמן, כך שככל שעובר הזמן, הסיכוי שהסוכן ייקח סיכון יורד.
כתוצאה מכך בדרך כלל נתכנס למצב הטוב ביותר.

```
function SIMULATED-ANNEALING(problem, max):
   current = initial state of problem
   for t = 1 to max:
      T = TEMPERATURE(t)
      neighbor = random neighbor of current
      ΔE = how much better neighbor is than current
      if ΔE > 0:
         current = neighbor
      with probability e^(ΔE/T) set current = neighbor
   return current
```

<u>Backtracking :</u>

עקרון למימוש בסוגי חיפוש שונים למקרה שנתקענו במצב לא אפשרי.
לפי עקרון זה, מתקדמים עם בחירה של המצב הבא בכל פעם, עד שנתקלים בבעיה או במצב לא אפשרי. כעת <u>נחזור</u> למצב האחרון בו היו לנו ברירות אחרות חוץ ממה שבחרנו בו, ונבחר את האפשרות הבאה בתור (כי אולי היא לא תגרום לבעיה בהמשך), משם נמשיך עם האלגוריתם שוב.
שיטת ה'באקטראקינג' חייבת לדעת מה ייעד הפתרון, כדיי לדעת לזהות אותו. (אי אפשר להמשיך ולהמשיך בלי לדעת איזה יעד מכוונים בשיטה זו).

חיפוש במשחקים מול יריב (adversarial search):

עבור משחקים של שני שחקנים- נצא מנקודת הנחה ששני השחקנים רציונאליים, ובוחרים בכל מצב את השכן שייתן להם את הערך הטוב ביותר עבורם בעתיד.
דוגמאות למשחקי 2 שחקנים- איקס עיגול, שחמט, וכו'...

אלגוריתם המינימקס (minimax):

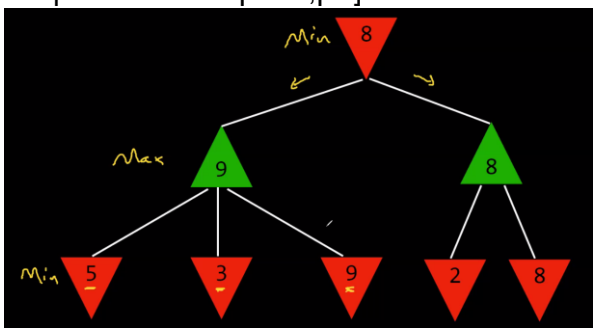
העיקרון:
עבור אחד השחקנים (ייקרא **max**), מטרת המשחק תהיה להגדיל את הערך של המצב הבא (על ידי בחירה של הפעולה שלו) כמה שאפשר.
עבור השחקן הנוסף (ייקרא **min**), המטרה תהיה למזער את ערך המצב הבא ככל האפשר.

לכל לוח ניתן ערך (מינימלי מאוד אם שחקן המינ' מנצח בו, מקסימלי אחרת, ואפשר גם בין לבין).
כל שחקן מקבל את הלוח שנתן לו היריב, ובוחר את הפעולה שלו (וכך בעצם קובע את המצב הבא לשחקן היריב) לפיי הערך הכי טוב מבין הלוחות הבאים האפשריים (מינ' ישאף להכי נמוך ולהפך).

כדיי לדעת את הערך של הלוחות הבאים- הסוכן מניח שהיריב רציונלי וממטב את המצב שלו גם, ועושה את אותו האלגוריתם, הפעם עבור היריב.
בפועל- נתחיל מהלוחות הסופיים ומהם נחשב עבור כל תור את הלוח האופטימלי שייבחר. (כמו רקורסיה הדדית).

- Given a state s:
  - MAX picks action a in ACTIONS(s) that produces highest value of MIN-VALUE(RESULT(s, a))
  - MIN picks action a in ACTIONS(s) that produces smallest value of MAX-VALUE(RESULT(s, a))

כמובן שברוב המשחקים לא נוכל להתחיל מהלוחות הסופיים, בהם מתרחש ניצחון או תיקו (ידרוש זיכרון שלא קיים במחשב), ולכן נרד עד עומק מסוים- לפי הגדרה שלנו, ועבור אותו עומק קצה, נקבע את ערך הלוח לפי יוריסטיקות מסוימות. משם נמשיך כרגיל.
(יוריסטיקה לדוגמא עבור שח- יכולה להיות לתת לכלים שחורים ערכים חיוביים בהתאם לשווי הכלי, וללבנים- שליליים בהתאמה [כאן, כמובן השחור יהיה מקס' והלבן מינ']).



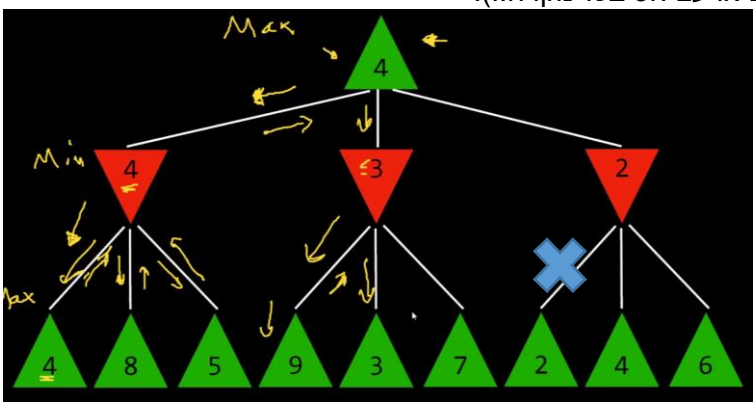
לפי המקובל אצלנו בקורס נגדיר:
משולש אדום מטה- הבחירה שבחר מינ'.
משולש ירוק מעלה- הבחירה של מקס'.
(בעצם בתור של המקס' הוא מקבל את הערכים 9,3,5 ובוחר את הערך הגבוה מבניהם- 9).

הפעולות שנשתמש בהן מבחינת פסאודו קוד:

- S_0 : initial state
- PLAYER(s) : returns which player to move in state s
- ACTIONS(s) : returns legal moves in state s
- RESULT(s, a) : returns state after action a taken in state s
- TERMINAL(s) : checks if state s is a terminal state
- UTILITY(s) : final numerical value for terminal state s

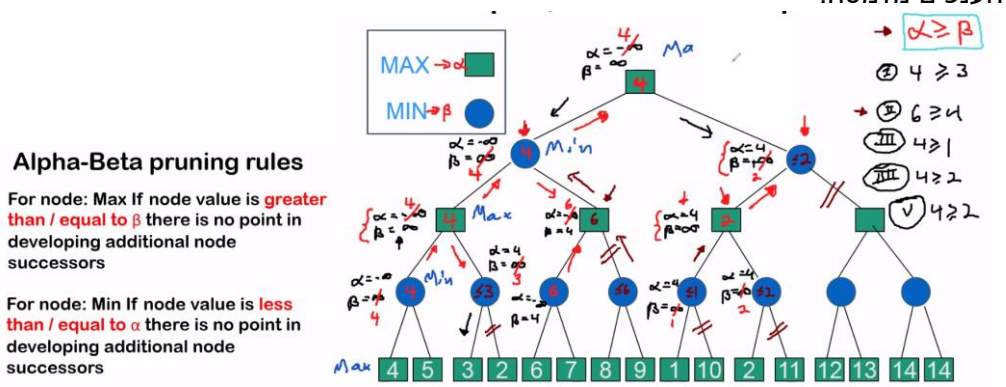
אופטימיזציה בשיטת **קיצוץ אלפא-ביתא** (alpha- beta pruning):
הבעיה המרכזית בחישוב בשיטת המינימקס היא תפוסה רבה של זיכרון (כי בכל שלב מפתחים המון מצבים), וצריך להתגבר עליה בדרך כלשהי.
הפתרון: קיצוץ של ענפיי המשך שאינם רלוונטיים אלינו.

נבין את הרעיון עם דוגמא:
אם בשלב מסוים בענף מסוים מינ' הציע למקס ערך, נניח 4, ובענף הבא בבחירה של מינ' אנחנו נתקלים בערך של 3 באחד הענפים- מקס כבר יודע שמינ' יציע לו בענף הזה את המספר המינימלי, כלומר ערכים שקטנים או שווים ל3, ולכן ברור למקס' שמבין שני הענפים שהצגנו עדיף לו לבחור ב4.
מקס' יודע את זה, גם מינ' יודע את זה (שני השחקנים מניחים רציונליות של היריב), ולכן לשניהם אין טעם בכלל לבדוק את **שאר** האופציות בענף שהיה בו את ה3. וניתן לכתוב בתור של מינ' ש"בחר" 3. (שוב, גם אם יהיו ערכים מינימליים יותר, זה לא משנה כי מקס' לא ימשיך איתם או עם ה3 בכל מקרה..).



מבחינת האלגוריתם:
נאתחל בראש העץ את אלפא למינוס אינסוף (הערך הרע ביותר עבור מקס'), ואת בטא לפלוס אינסוף (הערך הרע ביותר עבור מינ').
נשרשר את הערכים האלו מטה עד עומק העץ שבחרנו.
עבור שלב העלים- כל שחקן בתורו, שואל אם הערך בענף הנבדק טוב לו יותר מהערך הקיים אצלו (אלפא או בטא בהתאמה), אם כן, הוא מעדכן את הערך הקיים אצלו (ואת ערך הקובייה שלו) להיות כמו הערך בענף הנבדק.
לאחר בדיקת כלל הענפים היוצאים מהקובייה (ועדכון הקובייה בהתאם), עולים שלב לשחקן הבא.. (אם קיים ענף עם קובייה ריקה, נשרשר את אלפא ובטא מטה ונמשיך).
בתור של מקס'- מעדכנים את אלפא. בתור של מינ'- מעדכנים את בטא.

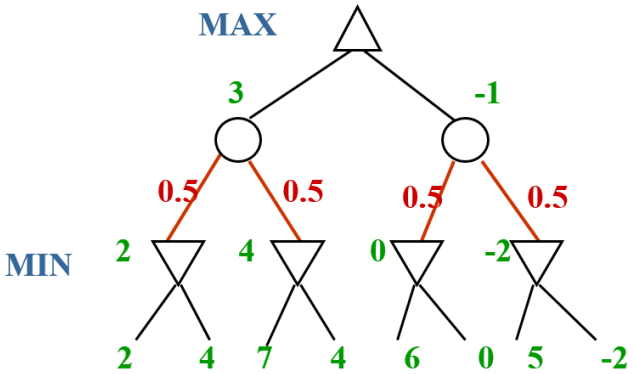
בכל שלב ושלב **(!)** - אם מתישהו **אלפא גדול/שווה לבטא**- אין טעם להמשיך ולבדוק את הענפים הנותרים עבור אותה קובייה, מהסיבות שהזכרנו לעיל. **נבצע קיצוץ** של שאר הענפים מלמטה.



<u>משחקים עם הסתברויות:</u>

עבור משחקים הכוללים הסתברויות (לדוג' שש-בש), לא משנה איזו פעולה נבחר לעשות, לא נוכל לדעת בוודאות מוחלטת מה יהיה הלוח הבא שיגיע ליריב.

במקרה כזה, נכמת כל לוח מהלוחות שיוכלו להיות ליריב, וניתן להם ערכים מסוימים, ואז נכפיל כל ערך <u>בהסתברות לקבל את הלוח הנוכחי</u>. (מזכיר מאוד את מושג התוחלת).



[בדוגמא: $0.5 * 2 + 0.5 * 4 = 3$]

<u>בעיות סיפוק אילוצים</u> <u>(CSP):</u>

קצת לוגיקה:

1. And: A ^ B אמת רק כאשר גם משפט A וגם משפט B אמת.
2. Or: A v B אמת כאשר או משפט A או משפט B או שניהם אמת.
3. Not: 'A אמת כאשר A שקר ולהפך.
4. Xor: אמת כאשר משפט A שונה ממשפט B.
5. יש הגדרות רבות נוספות, כמו 'אאם', ואחרות, הניתנות לייצוג על ידי ההגדרות לעיל.

בבעיות סיפוק אילוצים:

ישנם **משתנים** (variables) (לדוגמא אזורים בשולחן- רגליים, בסיס, לוח וכו'), ישנו **הדומיין** (domain), המכיל את **הערכים** (values) עבור המשתנים שלנו (לדוגמא צבעים עבור כל חלק בשולחן), ונניח **אילוצים** (constraints) שמגדירים אילו אפשרויות אפשר לקיים במשתנים שלנו ואיזה לא (לדוג' שאסור לצבוע חלקים סמוכים באותו הצבע).

נרצה לבצע השמה של ערכים למשתנים, בהתאם לאילוצי הבעיה.

הדוגמא המוכרת ביותר- **<u>בעיית צביעת מפות</u>**:

נתונה מפה עם אזורים נתונים. המטרה: לצבוע את כלל האזורים (עם 3 צבעים, בדוגמה שלנו), כך ששני אזורים סמוכים לא ייצבעו בצבע זהה.

עד עכשיו, למדנו איך לעשות זאת על ידי חיפוש עיוור- כלומר כל עוד אני לא מצב הייעד, הסוכן מפתח את הצאצאים של המפה הנתונה, ובודק אם היא מצב היעד. נשים לב כי **כבר בשלב הזה- לא נוכל לפתח מצב שסותר את האילוצים שלנו** (כמו צאצא שייצבע באדום עיר ליד עיר אדומה).

ישנן כמה יוריסטיקות שיוכלו לייעל לנו את החיפוש (לחסוך backtracking), כך שנימנע מראש מלפתח צאצאים שבעתיד יתבררו כסותרים את האילוצים שלנו:

1. **Most constrained variable- המשתנה הכי מוגבל:**
לפי היוריסטיקה הזו, נבחר לעבוד (לפתח ראשון בדפ"ס) בכל שלב עם ה<u>משתנה</u> עבורו נשארו הכי פחות ערכים אפשריים לשיבוץ. (במפה- הערים הצמודות למה שכבר צבוע).
2. **Most constraining variable- המשתנה הכי מגביל:**
לפי היוריסטיקה הזו, נתחיל לפתח את ה<u>משתנה</u> עם הכי הרבה השפעה על משתנים אחרים. (במפה- העיר עם הכי הרבה גבולות).
3. **Least constraining value- הערך הכי מגביל:**
כאן, נבחר לעבוד עם צאצא אקראי (או לפי סדר כלשהו אחר), אך נפתח ממנו את הצאצאים עם ה<u>ערך</u> הכי פחות מגביל, כך שישאיר הכי הרבה אפשרויות לערכים בהמשך. (במפה- נבחר לצבוע עיר מסוימת בצבע שישאיר הכי הרבה אפשרויות לצביעה בהמשך של הערים הבאות).
4. **Forward checking**– 'הסתכלות קדימה':
לאי יוריסטיקה זו, לאחר שבחרנו ערך למשתנה מסוים, "נמחק" לפי האילוץ את הערך הזה מהדומיין עבור כל הצאצאים השכנים למצב הנוכחי. (במפה- לאחר בחירת צבע לעיר מסוימת, נמחק מהדומיין את הצבע הזה עבור כל הערים השכנות אליה).
#ניתן גם להסביר על ידי משחק המלכות: לאחר הצבה של מלכה מסוימת, נמחק משאר המלכות בעמודות האחרות את האופציה להיות בשורה/ באלכסונים שלה.

בפועל, נדע איזה יוריסטיקה הכי טובה לכל מקרה על ידי בדיקת והשוואת פתרונות של כלל היוריסטיקות, ושימוש בזו שנתנה את התוצאה האופטימלית.

למידת מכונה (data mining/ machine learning):

למידה מפוקחת (supervised learning):

בעיות סיווג (classification):

בסוג זה של למידה, צריך לתת המון דוגמאות לסוכן, ולגרום לו לבנות מודל שיידע לזהות את התבניות האלו (לדוגמא תמונות של כלבים וחתולים, ויכולת לזהות האם המוצג בתמונה חדשה הוא כלב או חתול...).
הסוכן מקבל בשלב הראשון גם קלט וגם פלט, ועושה **סיווג** (classification), בשביל ההמשך.
הקטלוג הינו **בדיד**. (יש גשם/ אין גשם. אין אפשרות לקטלג **כמה** גשם יירד..).

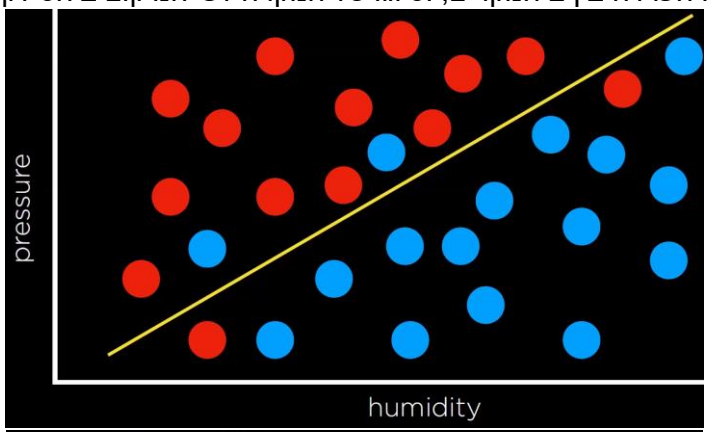
ישנן שיטות סיווג שונות למידע החדש שנבדק:

KNN- k nearest neighbor classification:

מציאת המקרה (השכן) הכי קרוב וסיווג לפי הזהות שלו.
כמובן, נעדיף לתת משקל לכמה נקודות ולא לאחת (K, כמה שנחליט), ועליהן מיצוע.
על רעיון זה נדבר בהרחבה בהמשך. הוא בשימוש בתחום אחר.

יצירת קו מגמה:

מציאת קו הפרדה בין 2 המקרים, וסיווג של המקרה לפי המיקום ביחס לקו שיצרנו.



x_1 = Humidity

x_2 = Pressure

$$h(x_1, x_2) = \begin{cases} \text{Rain if } w_0 + w_1x_1 + w_2x_2 \geq 0 \\ \text{No Rain otherwise} \end{cases}$$

הווקטור W(w0,w1,w2) בעצם מסמל את ערך הקו לפי הנוסחה לעיל,
נוכל להגדיר וקטור ערכים X(1,x1,x2) כך שהתנאי יהיה אם W*X>=0 תקטלג 1 אחרת 0.

$\mathbf{w} \cdot \mathbf{x}$: $w_0 + w_1x_1 + w_2x_2$

$$h(x_1, x_2) = \begin{cases} 1 \text{ if } w_0 + w_1x_1 + w_2x_2 \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

Perceptron learning rule:

שיטה ל'תיקון' של ווקטור W תוך כדיי קבלת נתונים ללמידה.
בהתחלה, נזין ערכי W כלשהם, ונתקן אותם לפי הסטייה של הסיווג האמיתי לפי הנתונים מהסיווג שהמערכת נתנה על ידי האלגוריתם.
כך לאחר קבלת נתונים רבים, נקבל ערך W יציב ונכון עד כמה שאפשר.

Given data point $(\mathbf{x}, y)$, update each weight according to:

$$w_i = w_i + \alpha(y - h_{\mathbf{w}}(\mathbf{x})) \times x_i$$

$$w_i = w_i + \alpha(\text{actual value} - \text{estimate}) \times x_i$$

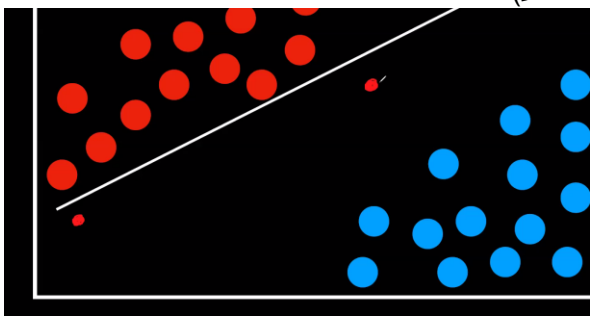
מקדם alpha קובע כמה משקל לתת לתיקון ביחס לערך הקיים. (בהמשל נוכל לעשות גם עליו אופטימיזציה).

את $h_w(x)$ (המקטלג) נוכל לקבוע בדרכים שונות, למשל:

1. על ידי פונ' מדרגה. קל להבנה אך לא ניתן למימוש, כי לא גזירה בכל התחום.
2. על ידי פונ' עם **מעבר רך** יותר בין 0-1 (soft threshold). כך שעבור נתונים שונים היא תיתן ערך שאינו 0/1 בשלב הלמידה.

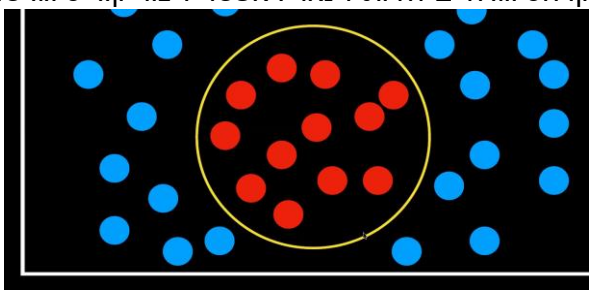
STM-Support vector machines:

נרצה שהקו שנעביר יהיה רחוק כמה שאפשר מכלל הנקודות בקבוצות השונות (כמה שיותר באמצע). לשם כך, ניצור שני קווים מקבילים שיהיה צמודים לקבוצות השונות, ובחישובים הלאה נשתמש בקו שנמצא בדיוק ביניהם ומקביל להם. (כדיי לקבל רגישות מקסימלית עבור המידע).



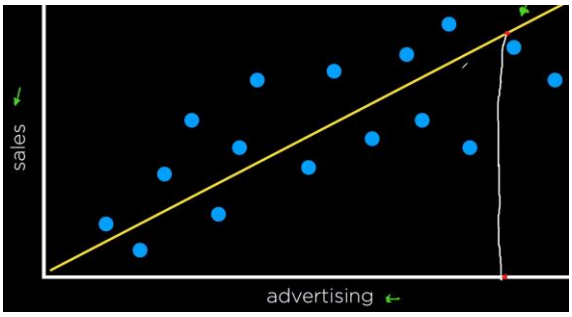
בתמונה- המערכת תקטלג את הנקודות המצויירות ככחול, למרות שמתאים להן יותר סיווג כאדום..

הערה: לא תמיד קו הסיווג חייב להיות לינארי. אפשר ליצור קווי סיווג שונים ואחרים..



<u>בעיות רגרסיה</u> <u>(regression):</u>

עד עכשיו דיברנו על חיזוי לערך קטגורי (**בדיד**, יש/אין גשם), כעת נרצה לחזות ערך רציף (לדוג' **כמות** הגשם שירד).



בצורת הרגרסיה (בדרך כלל הלינארית), לפי כל הנתונים שקיבלנו ניצור איזה קו ישר שינבא לנו בהמשך עבור כל קלט שנקבל איזה פלט לתת.

כדאי לשים לב: כאן, בשונה מהדוגמאות לעיל, ציר הX הוא הקלט- הנתונים, וציר Y הוא הפלט- הסיווג).

הערכת השערות (Evaluating hypotheses):

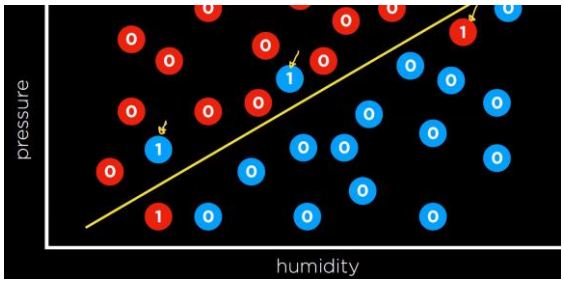
גם בבעיות רגרסיה, וגם בבעיות קטלוג, ישנן שיטות רבות למציאת הקו הלינארי ה**אידיאלי**.

Loss function- היא פונ' המתארת את כמות השגיאות שישנן לפי הקו בחרנו.

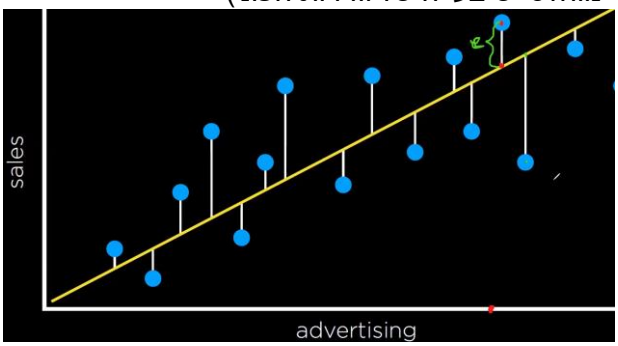
לאחר בדיקה של כל הקווים האפשריים, נבחר את זה בעל פונ' ההפסד הנמוכה ביותר.

דוגמאות לפונ' loss:

1. **0-1 loss function:**
תתאים לבעיות קטלוג, מקבלת את הנתונים האמיתיים, משווה לחיזוי שלה לפי הקו הנבדק, וסופרת את כמות הטעיות בחיזוי.



2. **L1 loss function:**
תתאים לבעיות רגרסיה, סוכמת את המרחקים של ערך הניבוי מהערך האמיתי בערך מוחלט (השגיאה בציר הערך).
(בעייתי כי עם ערך מוחלט יש בעייה של גזירות הפונ')



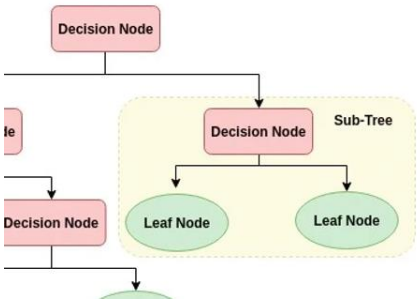
3. **L2 loss function:**
זהה לL1 , אך סוכמת את המרחקים בריבוע (במקום בערך מוחלט).
שני יתרונות על L1: גם גזיר וכו', וגם "מעניש" טעויות גדולות- כי אם נעלה אותן בריבוע הן יגדילו משמעותית את פונ' ההפסד.

למידה על ידי **עצי החלטה** (decisions trees):

בעולם הסיווג, נוח מאוד לדעת לסווג גוף כלשהו לקבוצה המתאימה בעזרת עץ החלטה.

העץ מכיל:

1. צמתי החלטה (decision node)- הצומת בה אנחנו מקבלים החלטה.
2. ענפים (edges/ branch)- ההסתעפויות היוצאות מהצומת.
3. עלים בעץ (leaf node/ terminal nodes)- נקודות סיום, בהן אנחנו מגיעים למסקנה.



בכל שלב בשלב הסיווג, נבחן את הגוף הנתון, ונראה לאיזה ענף הוא מתאים יותר, כשנגיע לעלה מסוים, נקבל סיווג לגוף לפי תוכן העלה.

איך בונים עץ החלטה מתוך נתונים קיימים?

נשתמש בעקרון האנטרופיה (entropy). אנטרופיה של מערכת היא מדד לחוסר הסדר שלה (כך שעבור מערכת הומוגנית האנטרופיה היא אפס, ועבור מערכת בחוסר סדר מוחלט האנטרופיה מקסימלית).

העיקרון שהאלגוריתם יעבוד לפיו: נתחיל את העץ עם האטריביוט (attribute) (הגורם, הסיווג אותו נרצה לבדוק כדיי לקבל החלטה) עבורו האנטרופיה מינימלית, כלומר ההחלטות הכי הומוגניות, וכך נוכל כבר בשלבים הראשונים 'לצמצם' את הסתעפויות העץ בעתיד.

נמשיך שוב, עם האטריביוט הבא, וכן הלאה. עד לקבלת כלל העלים.

לתשומת לב, בפועל, לא נחשב את האנטרופיה המינימלית, אלא את ההגבר המקסימלי- נראה בהמשך.

נגדיר מתמטית:

- Information Gain (IG) of the vertex
  - Measures how much information we gain if we break down this vertex

$$Gain(T,X) = Entropy(T) - Entropy(T,X)$$

  - T – target value, X - attribute
- Entropy of a single value

$$E(S) = \sum_{i=1}^{c} -p_i \log_2 p_i$$

- Entropy (target value, attribute)

$$E(T,X) = \sum_{c \in X} P(c)E(c)$$

הסבר קצר:

אנטרופיה של הטארגט **E(S)** :

נעבור על כל הקבוצות בטארגט, ונחשב עבורן את הפרופורציה שלהן ביחס להכל, ונכפיל בלוג הפרופורציה. נסכם עבור כלל הקבוצות (עם סימן מינוס) בטארגט וסיימנו.

בדוגמא עם הגולף- נבדוק כמה 'כן' כללי יש וכמה 'לא' לצאת לשחק, ונסכם את הפרופורציה של שניהם כפול הלוג...

אנטרופיה של האטריביוט **E(T,X)** :

נסתכל על אטריביוט מסוים ו'נבנה' את הטבלה עבורו. כעת, עבור כל שורה, נחשב את האנטרופיה של הטארגט שלה (כמו קודם), ונכפיל בשכיחות של השורה הזו באטריביוט. נסכום את הכל.

בדוגמת הגולף- נסווג את כלל הקבוצות לפי מזג אוויר. גשום/ שמשי/ נוח. ועבור כל מזג אוויר נחשב אנטרופיה לכן/לא ונכפיל בשכיחות מזג האוויר ביחס לכלל הימים.

'הגבר'- **information gain** IG:

ההגבר עבור אטריביוט מסוים יהיה האנטרופיה של הטארגט פחות האנטרופיה עבור אותו אטריביוט, כך שעבור אטריביוט עם קבוצות הומוגניות יותר, האנטרופיה שלו קטנה יותר, ונקבל עבורו הגבר גדול יותר.

נבחר להחלטה הראשונה בעץ שלנו את הסיווג לפי האטריביוט **עם ההגבר הגדול ביותר**.

יחס הגבר (GR- gain ratio):

Overfitting- בעיה בעצים, שמתקיימת כשהעץ מפורט מאוד, כלומר ישנם הרבה הסתעפויות וקריטריונים לקבלת ההחלטות השונות (לדוגמא אם אחד האטריבוטים הוא תאריך, ברור שיהיו הרבה קבוצות הומוגניות, ולפי האלגוריתם שהצענו הוא יהיה בראש העץ, אך העץ יהיה מפורט מדי!)

נרצה שהמודל שלנו יהיה נכון, ופשוט ככל האפשר, ולא בעל הסתעפויות רבות מדי. (ככה המודל יחמיר עם חלק מהנתונים ויקל עם אחרים..).

הפתרון- שימוש בGR במקום בIG.

נחלק את ההגבר בספליט של האטריביוט המסוים, וככה ניתן פחות חשיבות לאטריבוטים עם הרבה איברים שונים (שמראש נוטה להיות הומוגנית יותר).

חישוב פונ' הספליט עבור אטריביוט מסוים תהיה סכימה של כמות הטארגט הכוללת בכל הגדרה באטריביוט בפרופורציה לכל המקרים, כפול הלוג וכו'..

(עבור אטריביוט עם יותר אפשרויות הספליט יהיה גדול יותר, וככה יחס ההגבר יקטן).

$$\text{GainRatio}(T,X) = \frac{\text{Gain}(T,X)}{\text{SplitInformation}(T,X)}$$

$$Split(T,X) = -\sum_{c \in A} P(c)\log_2 P(c)$$

עבור דוגמת הגולף- באטריביוט מזג האוויר, נבדוק כמה שכיחים ימי הגשם ביחס לכל הימים, וכמה שכיחים ימי השמש וכו'...

(כלל אצבע- עבור קבוצות של 2 רכיבים- הספליט בערך 1. עבור 3 בערך 1.5. עבור 4 בערך 2).

מתי נגדיר שקיבלנו עלה? מתי שהגענו לקבוצה הומוגנית- עלה ודאי, או כשנגיע לתנאי עצירה שהגדרנו, שם נגדיר עלה (למרות שהוא לא ודאי לפי הנתונים), כדיי לקצר את זמן הריצה/ הזיכרון של התוכנית.

למידה לא מפוקחת (unsupervised learning):

למידה לא מפוקחת: הסוכן יבנה עצמאית מודל לסיווג וקטלוג של הנתונים, לפי דוגמאות רבות שיינתנו לו, ללא התערבות אדם.

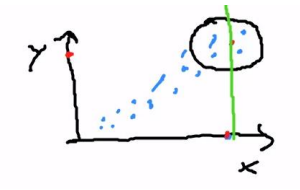
המודל יתבקש למצוא קבוצות שונות, אך לא ברור מראש מה הקבוצות שהסוכן יחלק לפיהן... (אם ניתן לו ארגז תפוחים, הוא יכול לחלק אותם לפי צבא, אך גם לפי צורה.. מה שהוא יזהה).

#כדי לעבוד עם השיטות השונות בלמידה לא מפוקחת נעבוד עם העיקרון:

"אם הם נראים דומים, הם כנראה דומים"..

לדוגמא בעבודה עם השוואת משפטים, נוכל לבנות מחסן מילים, ועבור כל משפט, למלא מחסן בגודל זהה למחסן המילים שבו יצוינו השכיחויות של כל מילה במשפט. (אפשר גם בבינארי, האם מופיעה/לא).

רעיון, שכבר נתקלנו בו ללמידה ללא בניית מודל סיווג (lazy learning) יהיה **אלגוריתם K-nn** כלומר התייחסות ל k השכנים הקרובים, ועל פיהם להחליט איך לסווג (מתאים גם לבעיות רגרסיה) את הקלט החדש.



נבחר את K כרצוננו, נשים לב שלא בוחרים K קטן מדי (כי אז נהיה רגישים לטעויות), ולא K גדול מדי, כי אז נוכל "לפלוש" לקבוצה אחרת צפופה שתשפיע על התוצאות שלנו.

בכל מקרה, נבחר K אי-זוגי, כדי להימנע מתיקו. (עבור בעיות סיווג).

כדי להחליט מה הקרבה של כל שכן אלינו נוכל להשתמש בכמה 'הגדרות קרבה':

1. מרחק אוקלידי:

$$D(a,b) = \sqrt{\sum_{i=1}^{n} (b_i - a_i)^2}$$

2. מרחק מנהטן:

$$\text{Distance} = \sum_{i=1}^{N} |a_i - b_i|$$

3. Hamming distance:

סוכם את כמות הפעמים בהן כמות ההופעות של אותה המילה שונה מהמשפט המקורי.

```
count += int(s1[i] != s2[1])
```

בסופו של דבר- נבחר את המרחק הקצר ביותר של המשפט שלנו מהמשפטים המוכרים, ונסווג את המשפט שלנו כמשפט הזה. (כמובן, התוצאה יכולה להיות תלויה באיזה הגדרת מרחק אנחנו משתמשים).

במקרה של K מעל 1, עבור סיווג נסווג לפי רוב הנקודות בK הקרובות, ועבור רגרסיה, נחזיר ממוצע של K הנקודות שמצאנו שהכי קרובות לנתון החדש.

K-means:

האלגוריתם הזה יעזור לנו לסווג קבוצת דאטא גדולה לפי אשכולות (זוהי הדוגמה הגדולה ללמידה לא מפוקחת), בהן נוכל להשתמש בעתיד כדיי לסווג נתונים חדשים.

כאן, K זהו מספר האשכולות שנרצה שאליהן האלגוריתם יחלק את הדאטא, נצטרך לפי הצורך המעשי, לדעת לכמה אשכולות אנחנו מצפים, ולתת את K כנתון לאלגוריתם. (שוב, לא בטוח שיסווג לפי המאפיין שאנחנו חשבנו עליו. רק יסווג לK אשכולות שונות ביניהן).

מבחינה פסאודו-קוד, סדר הפעולות של האלגוריתם:

K-MEAN Clustering Sketch

1. Define K as the number of Clusters to find
2. Randomly select k values within the records as Seeds
3. Find the distance of every point from every seed
4. Each point will be joined to the cluster with the smallest distance
5. After all points have been assigned recalculate the center of each cluter
6. Repeat steps 3-5 until the distances don't decrease (e.g when the center stops moving)

כמובן, הסיווג הסופי תלוי בבחירת הנקודות המרכז הראשוניות.. ישנן דרכים להתגבר על כך, אך לא נלמד אותן במסגרת הקורס הנוכחי.

לסיכום:

עבור **למידה מפוקחת** ,נבנה מודל, על פי דוגמאות רבות, כך שבהגיע נתון חדש, נוכל לשייך אותו לקבוצה לפי המודל שבנינו.

דוגמאות: קווי רגרסיה, עצי החלטה.

ישנו ה **lazy learning**, בו אין מודל (free model), אבל יש הרבה דאטא עם סיווג, וכשמגיע נתון חדש נוכל לסווג אותו לפי מה שהסוכן למד.

דוגמא: K-nn.

ויש **למידה לא מפוקחת**, לתת הרבה דאטא לסוכן, הוא יסווג אותו לקבוצות, ויסווג גם את הנתונים החדשים לפי מה שהחליט.

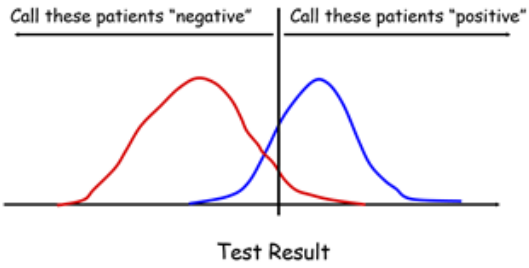
דוגמא: K-means.

<u>שיטות להערכת האיכות</u> <u>(Evaluation methods):</u>

שיטה (סטטיסטית) לבדיקה עד כמה האלגוריתם שלנו עשה את הסיווג טוב.

כמובן, נוכל לשנות את שיטת הסיווג, או את ערך הסף, כדיי לקבל סיווג טוב יותר על פי המודל שנבנה.

השפעת ערך הסף:



נגדיר **מטריצת בלבול** (confusion matrix):

שורות המטריצה- ייצגו את **הערך האמיתי** (real/ actual) של הנתון, עמודות המטריצה, ייצגו את **הערך שהתוכנה סיווגה לו** (Predicted), כך שנוכל ל'מקם' בשורה ועמודה מסוימת את הנתון שהבאנו עכשיו, יחד עם המיקום אליו סיווגנו אותו.

בשביל להסביר את החישובים בשיטה, נשתמש במטריצת בסיס, בעלת 2 שורות ועמודות. חיובי ושלילי. לתאים נקרא בהתאמה כמו בתמונה הבאה:

		Predicted tag (by model)	
		Positive (A)	Negative (B)
A known tag	Positive (A)	True Positive (TP)	False Negative (FN)
	Negative (B)	False Positive (FP)	True Negative (TN)

TP- נתון חיובי שסווג כחיובי (הצלחה).

FN- נתון חיובי שסווג כשלילי (טעות סיווג).

FP- נתון שלילי שסווג כחיובי (טעות סיווג).

TN- נתון שלילי שסווג כשלילי (הצלחה).

(בעצם תמיד האלכסון הראשי יראה הצלחה בסיווג, בעוד ששאר התאים- טעות סיווג).

נמשיך להגדיר:

Recall:

הריקול של קטגוריה מתייחס לערך האמיתי (**שורה**) של הקטגוריה, ומבטא את היחס בין כמות הפריטים שאנחנו שהצלחנו לסווג נכון בקטגוריה, לבין כמות הפריטים מהקטגוריה הזו סך הכל.

Precision:

הפרסיז'ן של קטגוריה מתייחס לפריטים שסיווגנו לקטגוריה הנבחנת (**עמודות**). מבטא את היחס בין כמות הפריטים שבאמת שייכם לקטגוריה שסיווגנו אליה, לבין הפריטים שזיהינו שבקטגוריה והם טעות סיווג.

בצורה כמותית:

$$precision = \frac{TP}{TP + FP}$$
$$recall = \frac{TP}{TP + FN}$$

כמובן, נרצה גם ריקול וגם פרסיז'ן גדולים ככל האפשר. אם הם קטנים, נבין ששיטת הסיווג של התוכנה כנראה לא טובה מספיק או לא מתאימה למקרה שלנו.

ממוצע הרמוני (F-measure/ harmonic mean/F1):

ממוצע הרמוני משכלל את שני הפרמטרים שהגדרנו קודם, כך שאם אחד מהם קטן באופן חריג, הממוצע ההרמוני יהיה קטן גם כן.

נגדיר אותו מתמטית כך:

$$F_1 = \left(\frac{\text{recall}^{-1} + \text{precision}^{-1}}{2}\right)^{-1} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}.$$

במערכת- נבחן את הממוצע ההרמוני וניתן לו ערך סף מסוים. אם יהיה קטן ממנו- סימן שיש לנו הרבה טעויות סיוג במערכת (אין אינדיקציה לגביי שורות/עמודות), ונצטרך לבחון את אפיון התוכנה מחדש.

עד עכשיו ההסבר היה לגביי מטריצת בסיס של 2X2. עבור מטריצות גדולות יותר, כמו בדוגמא בתמונה הבאה, נחשב את הריקול והפרסיז'ן של סיווג מסוים, למשל מהו ערך הריקול של ציפור? (יתייחס לשורה של ציפור), או מה ערך הפרסיז'ן של דג? (יתייחס לעמודה של הדג).

Precision	Recall	F-Measure	ROC Area	Class
1	1	1	1	mammal
1	1	1	1	bird
0.75	0.6	0.667	0.793	reptile
0.929	1	0.963	0.994	fish
1	0.75	0.857	0.872	amphibian
0.625	0.625	0.625	0.92	insect
0.727	0.8	0.762	0.986	invertebrate

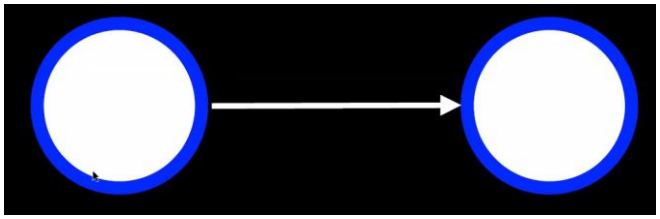
```
=== Confusion Matrix ===

  a  b  c  d  e  f  g   <-- classified as
 41  0  0  0  0  0  0 |  a = mammal
  0 20  0  0  0  0  0 |  b = bird
  0  0  3  1  0  1  0 |  c = reptile
  0  0  0 13  0  0  0 |  d = fish
  0  0  1  0  3  0  0 |  e = amphibian
  0  0  0  0  0  5  3 |  f = insect
  0  0  0  0  0  2  8 |  g = invertebrate
```

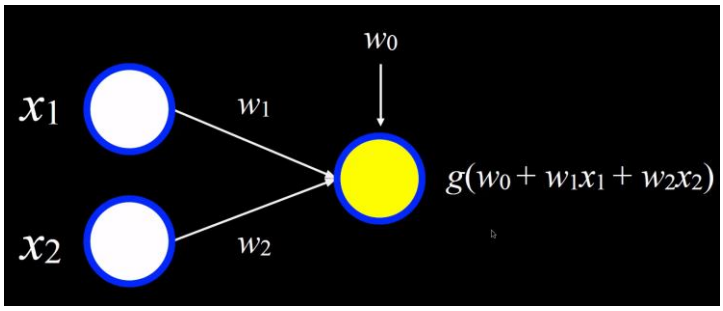
רשת נוירונים (Neural networks):

הרעיון: לחקות רשת ניוירונים ביולוגית תוך שימוש באמצעיים מתמטיים וכוח חישוב, כך שהסוכן יוכל לקבל החלטות ולבצע משימות בעצמו, ללא מודל נתון, אלא על ידי בנייה ובחירה של המודל הכי טוב למשימה.

האנלוגיה לעולם הנוירונים- כל נוירון מלאכותי יקבל קלט (input) וייתן פלט (output) החוצה, או לנוירון הבא (כקלט).



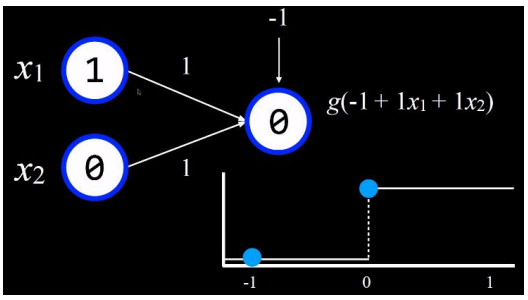
פונ' אקטיבציה (activation function): נסמן בG פונ' שתחליט ע"פ הקלט (או הקלטים, הW) המגיעים לנוירון איזה פלט הוא יוציא. (פונ' מדרגה, arctan, ReLU...).



הרשת תמצא את הW המתאימים (או כל פרמטר קובע אחר) עבור המימוש המתאים.

לדוגמא- מימוש שער or/and (עבור פונ' מדרגה):

$W_{or}=(-1,1,1)$, $W_{and}=(-2,1,1)$



אפשר לבנות טבלת אמת ולראות שפונ' G(X*W) של מדרגה תיתן את השערים הלוגים לעיל..

#אפשר שנוירון יקבל כקלט מספר רב של נתונים, כך שאפשר לפתור בעיות מרובות ממדים בעזרת הרשתות.

$$g\left(\sum_{i=1}^{n} x_i w_i + w_0\right)$$

#אפשר גם שמספר נוירונים יקבלו את הקלטים ויחשבו סיכוי לאפשרויות מנוגדות (נבחר את הנוירון ש'יורה' הכי חזק- בעל הסיכוי הכי גבוה).

gradient descent:

שיטה לשינוי וקטור הW על מנת לקבל מינימום של פונ' ההפסדים.

נגזור (גרדיאנט) את פונ' ההפסדים לפי הW הנוכחי, ונעדכן את וקטור הW לפי המרחק של הנגזרת מאפס (כך שבסוף נגיע לנגזרת קרובה לאפס- מינימום הפסדים).

$$Wi = Wi - \alpha * \frac{dL(W)}{dWi}$$

כאשר alpha נקרא **learning rate** והוא ייתן משקל עד כמה נשנה את וקטור הW עבור כל איטרציה.

(alpha צריך להיות מדויק עבור היישום שלנו. אם יהיה גדול מדי- נוכל להתבדר ולפספס את המינ'.
אם יהיה קטן מדי- התיקונים יהיו קטנים מאוד וייקח זמן רב עד להתייצבות של וקטור הW).

- Start with a random choice of weights.
- Repeat:
  - Calculate the gradient based on **one data point**: direction that will lead to decreasing loss.
  - Update weights according to the gradient.

הגרדיאנט הרגיל מערכן את וקטור הW לאחר מעבר ושכלול של כלל נקודות הקלט. זה עובד מצוין ובסוף הוא מגיע להתייצבות יפה, אבל לוקח המון זמן ריצה במקרים מסוימים.

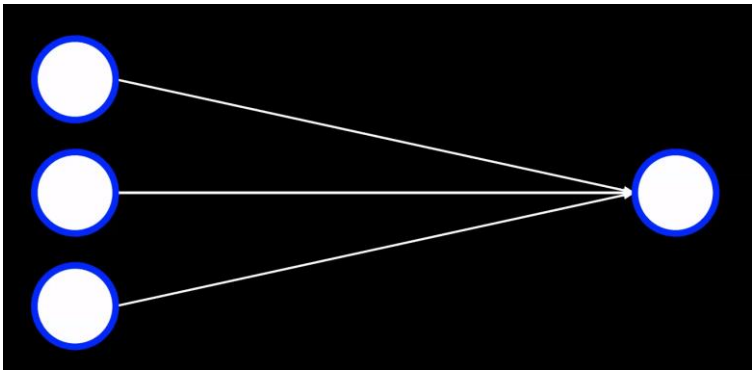
ישנן 2 שיטות מרכזיות לייעול האלגוריתם:

1. **Stochastic gradient descent:**
   העדכון של וקטור הW נעשה לאחר דגימה של נקודה אקראית אחת בכל פעם. כך הרבה איטרציות, אבל משמעותית פחות מהרגיל.
2. **Mini- batch gradient descent:**
   לאחר חילוק של קבוצת הנתונים לכמה מאגרים קטנים יותר, נלמד על וקטור W עבור כל מאגר בנפרד. בסוף אפשר לשכלל אותם.

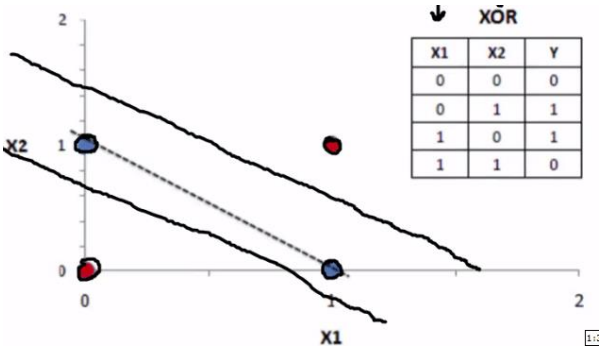
<u>רשת מרובת שכבות</u> <u>(multilayer neural network):</u>

עד עכשיו התעסקנו בבעיות המצריכות הפרדה לינארית של שתי הקבוצות כמו קו יחיד בין אדומים לכחולים, או מימוש של שער and ושער or.

מבנה כזה של מערכת, המשלבת חישוב על ידי נוירון בודד נקראת **<u>perceptron</u>**.

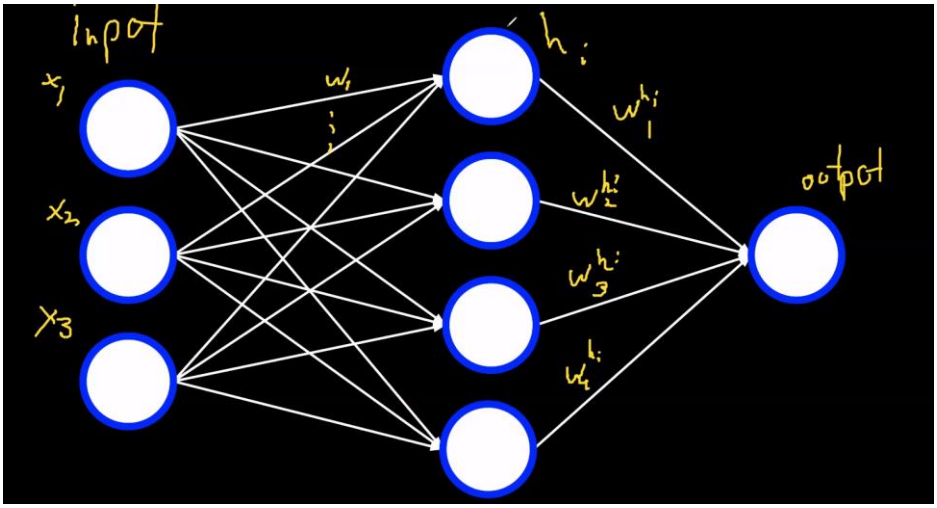


אך מה יקרה כשנרצה לסווג בין קבוצות שלא מופרדות לינארית? כמו כחולים בתוך עיגול ביחס לאדומים, או מימוש של שער xor?



במקרה כזה נשתמש בפונקציות אקטיבציה נוספות, ונשלב נוירונים נוספים בחישוב.

נקרא לחישוב בדרך כזו **<u>multilayer neural network</u>**, כך שלפני הנוירונים שייתנו פלט נשתמש בשכבות נוירונים נוספות לחישובי ביניים, הן ייקראו- **<u>hidden layer</u>**.



Backpropagation:

שיטה ללמידת הW ברשת, עבור רשתות בעלות שכבות נוירונים נסתרות.

(בשכבות כאלו לא תספיק שיטת הגרדיאנט, כי כל וקטור Wh_i תלוי בווקטור הWx שלפניו).

הרעיון- לשרשר את השגיאה בחיזוי אחורה וכך לתקן את הW:

- Start with a random choice of weights.
- Repeat:
  - Calculate error for output layer.
  - For each layer, starting with output layer, and moving inwards towards earliest hidden layer:
    - Propagate error back one layer.
    - Update weights.

$$\frac{\partial Error}{\partial W_6} = \frac{\partial Error}{\partial prediction} * \frac{\partial prediction}{\partial W_6}$$

chain rule

$$Error = \frac{1}{2}(prediction - actual)^2$$

$$prediction = (i_1 w_1 + i_2 w_2) w_5 + (i_1 w_3 + i_2 w_4) w_6$$

$$\frac{\partial Error}{\partial W_6} = \frac{\frac{1}{2}(predictoin - actula)^2}{\partial prediciton} * \frac{\partial (i_1 w_1 + i_2 w_2) w_5 + (i_1 w_3 + i_2 w_4) w_6}{\partial W_6}$$

$$\frac{\partial Error}{\partial W_6} = 2 * \frac{1}{2}(\mathbf{predictoin} - \mathbf{actula}) \frac{\partial (predictoin - actula)}{\partial prediciton} * (i_1 w_3 + i_2 w_4)$$

$$h_2 = i_1 w_3 + i_2 w_4$$

$$\frac{\partial Error}{\partial W_6} = (\mathbf{predictoin} - \mathbf{actula}) * (h_2)$$

$$\Delta = prediction - actual$$

delta

$$\frac{\partial Error}{\partial W_6} = \Delta h_2$$

$$*W_6 = W_6 - a\left(\frac{\partial Error}{\partial W_6}\right) \qquad *W_6 = W_6 - a\,\Delta h_2$$

אפשר לראות מהתמונה שההפסדים תלויים בW6 לפי שינוי של W3,4. וככה נשנה אותם אחד אחריי השני..

Overfitting:

גם בנושא של רשתות צריך להתמודד עם דיוק רב מדי- שבסוף יגביל אותנו ויקבע אותנו לנוירונים בודדים.

השימוש יהיה ב**dropout-** מדי פעם, באופן זמני, מסירים כמה נוירונים אקראיים מהרשת ועובדים איתה (מדי פעם מחליפים), כך שלא נתקבע על נוירונים חורגים בודדים...