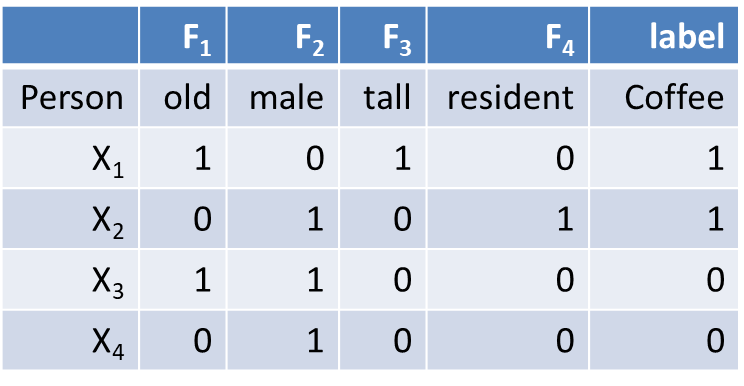
# Winnow & Perceptron

## אלגוריתם Winnow

זהו אלגוריתם שמטרתו למצוא מסווג ליניארי בינארי. נתונים מדגם של S אובייקטים כך שלכל אובייקט יש n תכונות כלשהן , ו-label המסווג את האובייקט כ-1 או 0 (המייצגים בדרך כלל תשובה לשאלת כן או לא). מטרת האלגוריתם היא למצוא וקטור W עם n משקלים, כך שאם נכפיל וקטור זה בכל אובייקט x עם n תכונות (ולא רק אלו שנתונים לנו), יעזור לנו לסווג את x האם שייך למחלקה 1 או שייך למחלקה 0.

ההנחה המרכזית של אלגוריתם Winnow היא שמתוך כל התכונות יש תכונות בלבד שהן קריטיות והשאר אינן חשובות, כך שאם לאובייקט כלשהו יש ערך שונה מ-0 **בלפחות אחד** מהתכונות הקריטיות, אזי נסווג אותו כ-1, אך אם בכל התכונות הערך של האובייקט הוא 0 נסווג אותו כ-0. אם הנחה זו לא מתקיימת על המדגם האלגוריתם לא יעבוד. מטרת האלגוריתם היא לזהות את כל התכונות הקריטיות ולנפות את כל שאר התכונות. מכאן שמו של האלגוריתם Winnow (לנפות).

דוגמה: בטבלה הבאה ניתן לראות שהתכונות ו- הן התכונות הקריטיות.

### תיאור האלגוריתם

האלגוריתם מאתחל את כל n המשקלים ב-W ל-1, ולאחר מכן עובר בלולאה על כל האובייקטים. עבור כל אובייקט מכפיל אותו בוקטור W וסוכמים את כל הערכים. אם הסכום שהתקבל גדול שווה ל-n מסווג אותו כ-1, אחרת מסווג אותו כ-0. לאחר סיווג של כל אובייקט בודקים את ה-label של האובייקט. אם סיווגנו נכון עוברים לאובייקט הבא, אך אם טעינו בודקים איזה סוג טעות זו. אם סיווגנו 1 אבל הוא באמת 0, משמע שכל התכונות בהן יש לאובייקט זה 1 הן אינן חשובות ולכן נאפס את המשקלים בהן. אך אם סיווגנו 0 והוא באמת 1, נכפיל פי 2 את כל המשקלים ב-W שבהן האובייקט שונה מ-0, כדי שבאיטרציה הבאה יהיה יותר סיכוי שנסווגו כ-1. כך חוזרים על פעולה זו עד שהמודל שיתקבל מסווג נכון את כל המדגם.

1. initialize = 1.
2. For each object :

* if classify as 1
* else classify as 0
* if we classified incorrectly:

A - if we classify 1 but was 0:

Set for all

B - if we classify 0 but was 1:

Set for all

1. return step 2 until no mistake is found on all objects

### סיבוכיות

נוכיח כי מספר הטעויות המקסימלי שהאלגוריתם מבצע על המדגם הוא לכל היותר . כתוצאה מכך נגיע למסקנה שמספר הפעמים שחוזרים על שלב 2 הוא , ולכן הסיבוכיות של אלגוריתם Winnow היא .

הוכחה:

* בכל טעות מסוג B מכפילים משקל של לפחות תכונה קריטית 1 פי 2. אחרי טעויות שבהן הכפלנו משקל של תכונה קריטית פי 2 נקבל שהמשקל של תכונה זו לבדה גדול מ-n ולכן לא נוכל לטעות בה יותר. כיוון שיש r תכונות, טעויות מסוג B יקרו לכל היותר פעמים.
* בכל טעות מסוג A מפחיתים את סכום המשקולות ב-W בלפחות n, ואילו טעות מסוג B מוסיפה לכל היותר n לסכום המשקולות. כיוון שסכום המשקולות יכול להגיע לכל היותר ל- לפי החלק הראשון בהוכחה, אזי על A ניתן לחזור לכל היותר גם כן פעמים, כי לא ניתן לאתחל משקל לערך שלילי לפי האלגוריתם.

## אלגוריתם Perceptron

כמו Winnow מטרת אלגוריתם Perceptron היא למצוא מסווג ליניארי בינארי, אלא שבנוסף מנסה שהמרווח של המישור הליניארי, המשמש כמסווג, מן הנקודות במדגם יהיה כמה שיותר גדול. המניע לכך הוא, שכמו שלמדנו בעקרונות למידת מכונה, ככל שהמסווג רחוק יותר מן הנקודות כך הטעות שלו על העולם תהיה קטנה יותר והמסווג יהיה טוב יותר.

האלגוריתם מסתמך על תכונה באלגברה ליניארית שאם מכפילים נקודה במרחב בוקטור אורתוגונלי למישור מקבלים את המרחק של הנקודה מהמישור. אם הערך חיובי הנקודה בצד אחד של המישור, אם שלילי הנקודה בצד שני של המישור ואם 0 הנקודה על המישור.

### תיאור האלגוריתם

האלגוריתם עובר בסבבים על כל האובייקטים במדגם עד שמסווג נכון את כל האובייקטים. עבור כל סבב האלגוריתם שומר וקטור משקלים . בהתחלה מאתחל כל המשקלים ב- ל-0. עבור כל אובייקט, אם המכפלה שלו בוקטור משקלים נוכחי גדול מ-0 נסווג אותו כ-1, אחרת נסווגו כ-0. לאחר סיווג של כל אובייקט בודקים את ה-label שלו. אם סיווגנו נכון עוברים לאובייקט הבא, אך אם טעינו בודקים איזה סוג טעות זו. אם סיווגנו 1 אבל הוא באמת 0, נגדיר  *כדי שבסבב הבא יפחת הסיכוי שנסווג אותו 1*. אך אם סיווגנו 0 והוא 1, נגדיר  *כדי שבסבב הבא* יגדל הסיכוי שנסווגו כ-1. בכל סבב בו גילינו טעות ראשונה, מסיימים את הסבב ועוברים לסבב הבא.

1. set
2. for round
3. for each object :

* If classify as 1
* else classify as 0
* if we classified incorrectly:

A - if we classify 1 but was 0:

Set

B - if we classify 0 but was 1:

Set

exit round t (on first mistake)

1. if no mistakes this round, exit algorithm

### סיבוכיות

נוכיח כי מספר המקסימלי של סבבים עבור הפעלה של האלגוריתם Perceptron על מדגם S היא לכל היותר , כאשר הוא המרווח המקסימלי של מישור שעקבי על S מאיזושהי נקודה ב-S. מכאן שסיבוכיות האלגוריתם היא .

הוכחה:

נסמן ב- את הוקטור המאונך למישור עם מרווח אותו אנו רוצים למצוא.

1. *נוכיח כי מתקיים .*

*עבור טעות מסוג A אגן מתקיים: .* שהרי מחזיר את המרחק מהמישור שהוא לכל הפחות .

באותו אופן, עבור טעות מסוג B מתקיים: .

1. *נוכיח כי מתקיים: .*

*עבור טעות מסוג A מתקיים: . שהרי אנו מניחים שכל הנקודות* x *הן על מעגל היחידה או בתוכו, ולכן . בנוסף, כי אחרת לא היינו מסווגים את* x *להיות 1.*

באותו אופן, עבור טעות מסוג B מתקיים: .

כעת, נסמן את המספר המקסימלי של סבבים באלגוריתם Perceptron ב-. לפי טענה 1 יוצא שבכל סיבוב מוסיפים ל- לפחות , ולכן לאחר סבבים מתקיים: **.** ולפי טענה 2 יוצא שבכל סיבוב מוסיפים ל- לפחות 1, ולכן לאחר סבבים מתקיים: . מכאן שמתקיים:

# SVM - Support Vector Machines

## הסבר האלגוריתם

לפי משפט שלמדנו בעקרונות למידת מכונה, מסווג ליניארי (המייצג מישור במרחב) הוא טוב יותר ככל שהמרווח ממנו לנקודה במדגם שהכי קרובה אליו גדול יותר. מרווח זה מסומן ב-. מטרת אלגוריתם SVM היא למצוא מסווג לינארי שהמרווח שלו מקסימלי. ישנם מספר שיטות כיצד למצוא מסווג לינארי זה:

### Hard Margin SVM

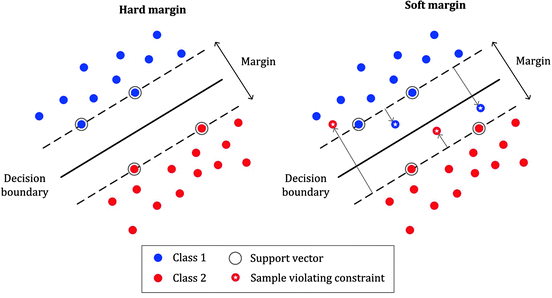
בהינתן S אובייקטים, , כך שלכל אובייקט יש n תכונות ותיוג , המסווג את האובייקט כ-1 או -1, נרצה למצוא מישור W במימד n הממקסם את תחת האילוץ כאשר W מנורמל (). אילוץ זה דורש שמרחק כל נקודה במדגם מהמישור הוא לפחות . הוקטור W צריך להיות מנורמל כי אז המכפלה מחזירה את המרחק של מ-W. אם אזי , ואם אזי .

דרך נוספת לנסח את הבעיה היא לא לנרמל את W ואז לקבוע שהמרחק היחסי בין המישור לנקודה במדגם הכי קרובה אליו היא . במצב זה ש-W אינו מנורמל המרחק מכל נקודה למישור W היא . אנו רוצים למצוא מישור W שממקסם את תחת האילוץ . ביטוי זה שקול למציאת מינימום של תחת האילוץ .

ניתן לפתור בעיות אלו באמצעות Convex optimization.

### Soft Margin SVM

לפי האילוץ אנו לא מאפשרים למצוא מישור שטועה על אף נקודה במדגם. לכן השיטה הקודמת מתאימה למקרה שניתן להבדיל בין המחלקות באופן מוחלט, כלומר קיים מישור שלעולם לא טועה. השיטה שנלמד כאן היא למקרה שבו לא ניתן להבדיל בין המחלקות, כלומר לכל מישור יש טעות כלשהי.

נשים לב כי הטעות האמיתית של המישור תלוי בטעות על המדגם ובמימד VC התלוי בגודל המרווח . מצד אחד אנו רוצים טעות כמה שיותר קטנה על המדגם אך מצד שני מרווח כמה שיותר גדול. לפי שיטת Soft Margin SVM אנו יכולים להגדיל את המרווח על ידי כך שנתייחס לכל הנקודות שנופלות בתוך המרווח מהמישור כטעות על המדגם (ואפילו אם סווגו נכון). כך אנו מגדילים את המרווח אבל גם מגדילים את הטעות על המדגם. במילים אחרות, ישנו tradeoff בין גודל המרווח לטעות על המדגם.

לצורך הגדרת הבעיה נחשב לכל נקודה ערך , כך שאם אינה במרחק של 1 מהמישור בצד המתאים לה שווה למרחק שלה מלהיות כזו, ואם הנקודה נמצאת בצד המתאים לה במרחק של 1 מהמישור אזי . זהו בעצם עונש של המודל שאנו רוצים שיהיה קטן כמה שיותר בממוצע. אנו גם רוצים מינימום של כמו בשיטה קודמת. לסיכום, אנו רוצים למצוא מינימום של:

תחת האילוצים:

מכפילים ב- *כי רוצים למצוא מינימום של שני גורמים שלא יודעים מה היחס בין הסדרי גודל שלהם. בוחרים את באמצעות* cross validation*.*

### Non-Linear SVM

ישנם שיטות נוספות של SVM, כמו גרעין פולינומי ו-RBF, המאפשרות למצוא מסווג שאינו ליניארי על ידי הגדלת מימד הבעיה. בפועל מעלים מאוד את אחוזי ההצלחה של המודל.

K-NN

## הסבר האלגוריתם

אלגוריתם KNN (k-Nearest Neighbors) הוא אלגוריתם המשמש כמסווג לא ליניארי, אך יכול גם לפתור בעיית רגרסיה. הרעיון המרכזי באלגוריתם זה, שבשלב האימון הוא זוכר מדגם מייצג של אובייקטים עם ה-label שלהם, ועבור כל אובייקט x חדש שרוצה לסווג, האלגוריתם צריך למצוא את k האובייקטים הכי קרובים אליו (k השכנים שלו), ואז לסווג את x אל המחלקה הנפוצה ביותר בקרב k השכנים הקרובים. שובר שוויון יכול להיות סכום המרחקים הכי קטן או השכן הכי קרוב.

אלגוריתם זה נחשב לפשוט מאוד. מצד אחד זמן האימון שלו הוא , אך הזמן שלוקח לו לסווג אובייקט הוא . זהו זמן ריצה גרוע מאוד. אנו מעדיפים שזמן האימון יהיה ארוך, כיוון שמבצעים אותו פעם אחת בלבד, ואילו זמן הסיווג יהיה מהיר.

כעקרון, אלגוריתם KNN יעיל כאשר יש הרבה דוגמאות בדאטה ומספר התכונות הוא נמוך. אולם ככל שיש פחות דוגמאות בדאטה או מספר תכונות גבוה, פחות כדאי להשתמש ב-KNN.

## פונקציות מרחק

כדי למצוא את k השכנים הקרובים צריך להגדיר ל-K-NN פונקציית מרחק. יש מספר סוגים מומלצים של פונקציות מרחק בהתאם לסוג האובייקטים ביניהם רוצים למדוד מרחק.

### פונקציות מרחק בין וקטורים -

יש שלוש פונקציית מרחק מרכזיות:

1. L1 Distance (Manhattan) - זהו סכום ההפרשים בערך מוחלט של כל f התכונות של שני האובייקטים.
2. L2 Distance (Euclidean) - זהו שורש הסכום של ההפרשים בריבוע של כל f התכונות של שני האובייקטים.
3. L Distance (Frechet) - זהו ההפרש המקסימלי מבין כל f התכונות של שני האובייקטים.

באופן כללי המרחק בין שני אובייקטים מוגדר כך . ככל שהערך p גדל המרחק בין שני הנקודות יכול רק לקטון.

### פונקציות מרחק בין מחרוזות - Levenshtein Distance

בהינתן שני מחרוזות s ו-t, נמדוד את המרחק ביניהן על ידי מספר הפעולות של מחיקה, החלפה והוספת אות שצריך לבצע על s כדי לקבל את t. האלגוריתם המחשב מספר פעולות אלו נקרא “Edit Distance”. ניתן לבצע את האלגוריתם בשיטת תכנות דינאמי בזמן ריצה של .

### פונקציות מרחק בין תמונות - Earth Mover’s

המרחק בין שתי תמונות A ו-B הוא מספר ההזזות של פיקסלים מתמונה A כדי שנקבל את תמונה B. ניתן להתייחס לבעיה זו כבעיית שידוך, שבה צריך להתאים כל פיקסל ב-A אל מיקום פיקסל ב-B הזהה לו. ניתן לפתור בעיה זו באמצעות אלגוריתם “Hungarian Method” בזמן ריצה של .

## קביעת היפר-פרמטרים

ההיפר-פרמטרים של אלגוריתם K-NN הם הערך k ופונקציית המרחק. k צריך להיות אי-זוגי כדי למנוע מקרים של שוויון בין מחלקות. כדי לוודא אילו ערכים הכי טובים יש לחלק את הדאטה כך שיהיה חלק של test שבו נבדוק את איכות המודל הסופי. יש כמה דרכים כיצד לחלק את הדאטה ולקבוע את ההיפר-פרמטרים.

* **train-validation-test**. נקבע ערכים שרירותיים ל-k, נמצא k שכנים הכי קרובים מה-train ונעריך את המודל על ה-validation. ניקח את ערכי ה-k הכי טובים על ה-validation, ולבסוף נבדוק את איכות המודל על ה-test.
* **Cross-Validation** - נחלק את הדאטה למספר חלקים שווים הנקראים folds ול-test. עבור כל ערך של k שנרצה לבדוק נעבור בלולאה כמספר ה-folds. בכל איטרציה fold אחר ישמש כ-validation ושאר ה-folds ישמשו כ-train. רמת הדיוק של ערך k יהיה ממוצע רמות הדיוק של כל האיטרציות. נבחר את ה-k שבו ממוצע רמת הדיוק היה מקסימלי. לבסוף נבדוק את איכות המודל על ה-test. שיטה זו שימושית בעיקר שאין הרבה דאטה.

## K-NN לרגרסיה

בהינתן דוגמה חדשה, האלגוריתם מחזיר ערך מאפיין, לדוגמה, ממוצע ערכים של ה-labels ב-k השכנים הקרובים ביותר.

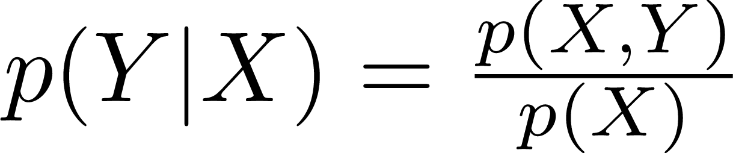
Naïve Bayes

## הסבר האלגוריתם

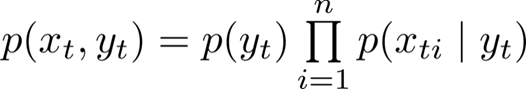
Naïve Bayes זהו אלגוריתם המייצר מסווג (Classifier) שבו הסיווג למחלקות אינו מתחשב בתלות שבין התכונות (ולכן נקרא נאיבי). מסיבה זו הוא יכולה לעבוד גם עבור אובייקטים שבהם **חסרים תכונות**. נסמן את כל סוגי המחלקות ב-. הרעיון באלגוריתם זה הוא לחשב עבור כל אובייקט שאנו רוצים לסווג למחלקה, מהי ההסתברות שהוא שייך לכל מחלקה, ולבחור את המחלקה עם ההסתברות הגבוהה ביותר, נסמן אותה ב-. במילים אחרות נרצה לחשב:

.

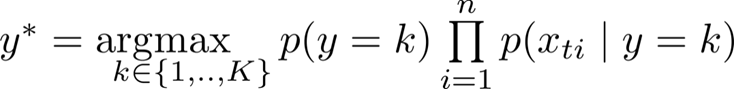
כאשר הוא ההסתברות שאובייקט שייך למחלקה .



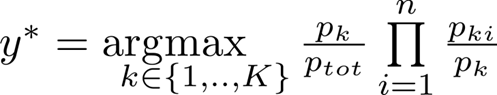
* לפי חוק בייס מתקיים: .
* ולפי נוסחת ההסתברות השלימה מתקיים:
* אנו מניחים באלגוריתם זה שתמיד אין תלות בין תכונות שונות. אנו מניחים זאת למרות שזה כמובן לא תמיד נכון, מפני שזה יכול להוביל לשגיאות, וגם כי בפועל זה מביא תוצאות טובות. לכן נקבל:



=

* **משילוב המשוואות נקבל את נוסחת Naïve Bayes:

כאשר היא ההסתברות שאובייקט כלשהו שייך למחלקה , ו-היא ההסתברות שהתכונה i ב- שייכת למחלקה .

כדי לפשט את המודל, שלא נצטרך לחשב את כל הנתונים בנפרד עבור כל אובייקט, נחשב את שלושת הערכים הבאים כדי לזרז את החישוב. הראשון משותף לכל המחלקות , ואילו השניים האחרים מיוחדים לכל מחלקה :

* - מספר האובייקטים שכבר נתונים לנו.
* - סכום כל האובייקטים השייכים למחלקה k.
* - לכל תכונה i ולכל מחלקה k, נחשב בכמה אובייקטים ממחלקה k התכונה i מופיעה.

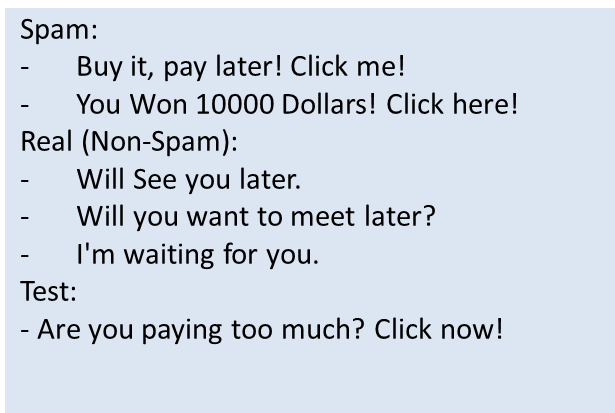
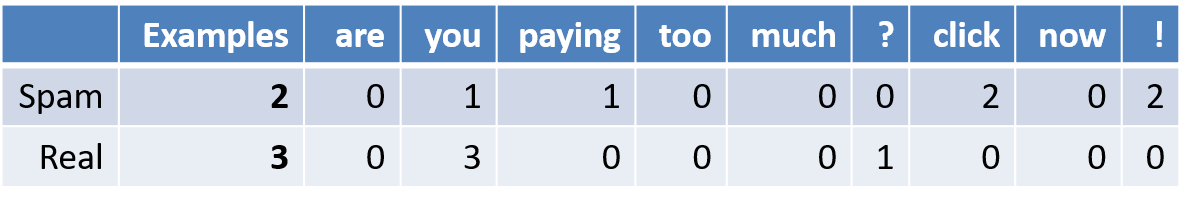
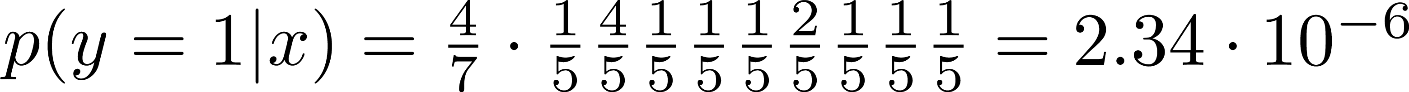
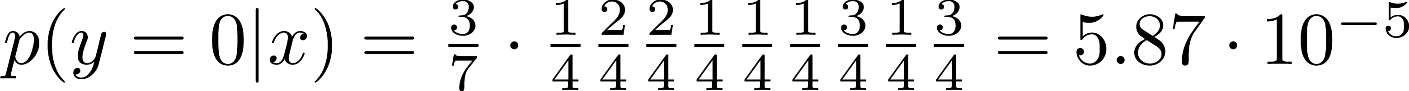
כעת כל שנותר הוא לחשב עבור כל אובייקט:

הערה - כאשר נתונים המון אובייקטים, כיוון שמכפילים הסתברויות, יכול להיות מצב שכל הערכים יצאו קטנים מיכולת הדיוק של המחשב, ולכן כל ההסתברויות יהיו 0. מסיבה זו, בפועל בדרך כלל במקום להכפיל הסתברויות, נפעיל פונקציית log לכל הסתברות ונסכום את כל ערכים שהתקבלו.

## Laplace's Smoothing

בעיה נוספת שיכולה להתעורר מהנוסחה שתיארנו היא כאשר נקבל שהסתברות עבור עמודה מסוימת היא 0. דבר זה יכול לאפס את המשוואה. כדי לפתור בעיה זו נוסיף לנתונים עוד אובייקט לכל מחלקה, שבו מופיעים כל הערכים האפשריים. פעולה זו תוסיף k ל-, 1 לכל , 1 ל- במונה מחוץ למכפלה, ו-2 ל- של המכנה במכפלה.

**דוגמה:** בדיקה האם הודעה היא ספאם או אמיתית



## דוגמה ב-Python

נמשיך את הדוגמה מסעיף קודם על סיווג הודעות אמיתיות לספאם. נניח שהמידע שמור ב-RDD בתור (הודעה, מחלקה).

input\_data = sc.parallelize([("hello there", 0), ("hi there", 0), ("go home", 1), ("see you",1), ("bye to you", 1)])

#We need to find , and all :

>>> pk = input\_data.map(lambda (message, cls): (cls, 1)).reduceByKey(lambda a,b: a+b).collectAsMap()

>>> ptot = sum(pk.values())

>>> pki = input\_data \

.flatMap(lambda (message, cls): list(set([(cls,w) for w in message.split()]))) \

.map(lambda (cls, word): ((cls,word), 1)) \

.reduceByKey(lambda a,b: a+b).collectAsMap()

>>> import numpy as np

>>> query = "hello hi"

>>> class\_probs = [pk[k]/float(ptot)\*np.prod (np.array([pki.get((k,i),0)/float(pk[k]) for i in query.split()])) for k in range(0,2)]

>>> print(class\_probs)

[0.10000000000000001, 0.0]

>>> y\_star = np.argmax(np.array(class\_probs))

>>> print(y\_star)

[0]

# Adaboost

## הסבר האלגוריתם

הרעיון באלגוריתם Adaboost הוא למצוא דרך "לחבר" חוקים (מסווגים) שהם חלשים יחסית לחוק אחד חזק יותר. חוק חלש הוא חוק שהוא מעט יותר טוב מבחירה אקראית, כלומר צודק על קצת יותר מחצי האובייקטים. חוק חזק הוא חוק שמתקרב לביצועים אופטימליים. הרעיון הכללי הוא לתת לכל חוק חלש משקל המציין את החשיבות שלו. עבור כל אובייקט שנרצה לסווג סוכמים את כל התוצאות של כל החוקים החלשים ביחס למשקל שלהם (הצבעה ממושקלת). נסביר ביותר פירוט.

### נתינת משקלים לחוקים

האלגוריתם מקבל:

* מדגם של אובייקטים S כך שלכל אובייקט יש n תכונות ותיוג .
* קבוצת חוקים חלשים H כך שכל חוק מחזיר 1 או -1 לכל אובייקט .
* מספר איטרציות k. כל איטרציה ניתן משקל לחוק אחד.

האלגוריתם מחזיר: משקל לכל חוק .



האלגוריתם מתחזק וקטור משקלים לכל הנקודות ב-S עבור כל איטרציה t. ראשית, מאתחלים את כל הנקודות ב- ל-. לאחר מכן, בכל לולאה t מחשבים לכל חוק את סכום משקלי הנקודות ב- שטעה עליהם. מתוך כל החוקים בוחרים את החוק שהטעות שלו הכי קטנה ונותנים לו משקל לפי הטעות שלו - . ככל שהטעות של החוק המינימלי קטנה יותר כך המשקל שלו גדול יותר. אם הטעות מתקרב או גדול מחצי אזי המשקל של יהיה קרוב ל-0, כלומר אינו משמעותי. רק חוקים שמדייקים יותר מחצי הם רלוונטיים.

לבסוף, נרצה לחשב את כהכנה לאיטרציה הבאה. נרצה שמשקל כל הנקודות ש- טעה עליהם באיטרציה נוכחית יגדל באיטרציה הבאה, כדי שנוכל למצוא חוק טוב עבור נקודות אלו, ואילו משקל כל הנקודות ש- צדק עליהן באיטרציה נוכחית יקטן באיטרציה הבאה. כך נקבל שחיבור החוקים הסופי יהיה יעיל על כל הנקודות. נחשב כך

נשים לב שאם צדק על מתקיים , ואז הכפלה ב- מקטינה את המשקל. לעומת זאת, כאשר טעה על מתקיים , ואז הכפלה ב- מגדילה את המשקל. מחלקים בערך קבוע כדי שסכום כל המשקלים ב- יהיה שווה ל-1. .

### סיווג אובייקט חדש

עבור כל אובייקט x חדש נחשב את סכום חיזוי החוקים כפול המשקל שלהם:

אם ערך זה חיובי אזי הסיווג הסופי הוא , ואם שלילי הוא .

## כמה חוקים לחבר

מצד אחד אם נחבר הרבה חוקים נוכל ליצור חוק חזק עם ביצועים טובים על המדגם. אמנם לפי משפט איחוד חוקים, ככל שיש יותר חוקים כך המימד VC גדל והטעות על העולם גדלה, כלומר נקבל overfitting. מצד שני, אם נחבר מעט חוקים נקבל מימד VC קטן אך יכול להיות שלא נצליח ליצור חוק טוב על המדגם. יש כאן tradeoff בין הטעות על המדגם למימד VC (זהו bias-variance tradeoff). כדי למצוא מודל אופטימלי באמצעות Adaboost צריך למצוא את האיזון במספר החוקים, על ידי השוואה של מודלים שנוצרו עם קבוצות שונות ובגדלים שונים של חוקים.

## מימוש בפייתון

**def train(self, points, iters, rules):**

num\_points = len(points)

num\_rules = len(rules)

# init numpy arrays

D = np.zeros(shape=(iters+1, num\_points))

self.best\_rules = np.zeros(shape=iters, dtype=object)

self.weights = np.zeros(shape=iters)

# Adaboost algorithm

D[0] = np.ones(shape=num\_points) / num\_points

for i in range(iters):

errors = np.zeros(shape=num\_rules)

for (j, rule) in enumerate(rules):

for (p, point) in enumerate(points):

if rule.classify(point) != point.label:

errors[j] += D[i][p]

# Find rule with min error and calculate its weight

minErrorIdx = np.argmin(errors)

minError = errors[minErrorIdx]

minErrorRule = rules[minErrorIdx]

self.weights[i] = 0.5 \* np.log((1 - minError) / minError)

self.best\_rules[i] = minErrorRule

# update next iteration points weight

for (p, point) in enumerate(points):

D[i+1][p] = D[i][p] \* (np.exp(-1\*self.weights[i]\*minErrorRule.classify(point)\*point.label))

z = np.sum(D[i+1])

D[i+1] /= z

return (self.best\_rules, self.weights)

**def predict(self, point):**

sumPred = 0

for (i, rule) in enumerate(self.best\_rules):

sumPred += self.weights[i]\*rule.classify(point)

if(sumPred > 0):

return 1

else:

return -1

# עצי החלטה

## הסבר האלגוריתם

Linear Regression

## הסבר האלגוריתם

זהו אלגוריתם שפותר בעיית רגרסיה, כלומר מחזיר ערך מטווח רציף ולא בדיד. רגרסיה ליניארית היא שיטה מתמטית המאפשרת לנו לבנות פונקציה ליניארית, כך שבהינתן אובייקט חדש x, שאנו יודעים את **כל** k התכונות שלו, נוכל לנבא מהו הערך y עבור x.

כל מייצג את המשקל (weight) שאנו נותנים לכל יחידה מהתכונה ה-i, ו-b הוא ערך מינימלי כלשהו (bias), שמטרתו להוסיף כוח חישוב שאינו תלוי בתכונות. הסיבה שאנו רוצים דווקא פונקציה שהיא ליניארית היא מפני שאנו מחפשים חוק פשוט, שלפי משפט Sauer הוא חוק טוב.

הרעיון המרכזי הוא שהפונקציה הליניארית שאותה אנו מחפשים היא הפונקציה שסך הסטיות של כל m האובייקטים בדאטה ממנה הוא מינימלי. ככל שיש לנו יותר מידע, כלומר יש בידינו המון אובייקטים נתונים, כך יכולות הניבוי של המודל יהיו טובים יותר. אמנם יש איזשהו גבול של מספר אובייקטים שכבר לא ישפר את המודל.

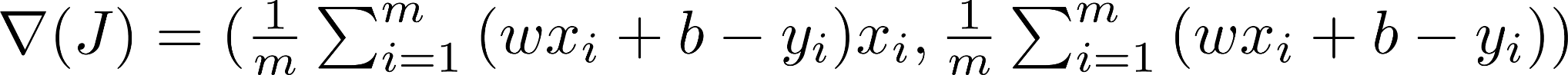
נתחיל עם רגרסיה ליניארית פשוטה שבה מספר התכונות הוא , ולאחר מכן נלמד כיצד לבנות משוואה עבור מודל עם k תכונות.

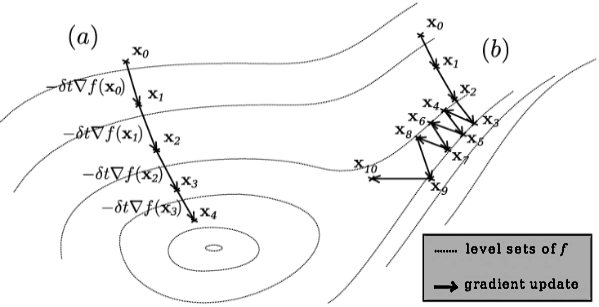
## רגרסיה ליניארית פשוטה

כאשר לכל אובייקט יש תכונה אחת בלבד, אנו מחפשים פונקציה פשוטה מהצורה , שבה נוכל להציב את התכונה של x ולקבל את הערך הצפוי y אותו אנו רוצים לנבא. w ו-b הם קבועים שבהמשך נלמד כיצד לחשב אותם. כאמור לעיל, אנו מחפשים פונקציה שעבור סכום המרחקים של הערך של כל אובייקט מהערך שהפונקציה צופה לו הוא מינימלי. ניתן לתאר את המרחק בערך מוחלט או בהעלאה בריבוע של ההפרש. נבחר בהעלאה בריבוע מפני שהוכח הסתברותית שמקבלים תוצאה טובה יותר על ידי העלאה בריבוע, וגם בגלל שערך מוחלט הוא לא גזיר ויכול להיות גם לא רציף. בנוסף, נחלק ב-2m כדי לקבל סטייה ממוצעת בריבוע (החלוקה ב-2 כי מצטמצמת עם הנגזרת). אם כן, מטרתנו היא למצוא w ו-b שמצמצמים את ה-Loss Function, שנסמן אותה , והיא:

דרך אחת לעשות זאת היא לגזור את הפונקציה ולהשוות ל-0, אמנם עבור מודלים מסוימים שבהם יש המון מידע וכן כהכנה לשיטה הבאה שנלמד, Logistic Regression, משתמשים בשיטת Gradient Descent.

## Gradient Descent

וקטור הגרדיאנט זהו וקטור שמייצג את הנגזרת של פונקציה עם מספר משתנים, ומסומן ב-. בכל כניסה בוקטור נמצא הנגזרת של הפונקציה לפי משתנה אחר. גודל הוקטור כמספר המשתנים בפונקציה. וקטור הגרדיאנט עבור פונקציית ה-Loss ב-Linear Regression הוא:

הרעיון בשיטת ה- Gradient Descentהוא להתחיל בערכים שרירותיים כלשהם של w ו-b, ובלולאה לשנות ערכים אלו בצעדים של קצב למידה ביחס לערך הגרדיאנט לפי ערכי w ו-b הנוכחיים, עד שמגיעים למצב של התכנסות (convergence). התכנסות זה מצב שבו הצבתם של w ו-b הנוכחיים בוקטור הגרדיאנט ייתן לנו את וקטור ה-0 או שואף לכך. כאשר מגיעים למצב של התכנסות מצאנו את ערכי w ו-b המינימליים.

חשוב ש- לא יהיה גדול מדי, שכן אז עלולים לפספס את המינימום ובצעד חזור שוב לפספס, וכך אף פעם לא להגיע למינימום. לכן צריך ש- יהיה קטן יחסית, בדר"כ באזור 0.001. על בסיס שיטה זו בנויים שיטות למידה רבות כמו רגרסיה לוגיסטית ולמידה עמוקה בכללותה.

האלגוריתם שבו נשתמש כדי למצוא ערכי w ו-b ברגרסיה ליניארית עבורם נקבל מינימלי הוא:

* בחר w ו-b רנדומליים כלשהם (בדרך כלל 0 לשניהם).
* קבע איזשהו שהוא גודל הצעד על w ו-b.
* בלולאה עד שמשיגים התכנסות (convergence) או מספר סופי של צעדים:
* חשב את .
* חשב את .
* עדכן את w להיות .
* עדכן את b להיות .
* החזר את w ו-b.

## דוגמה ב-Python

דוגמה לחישוב משוואה שתנבא מחיר של פלאפון גלאקסי S5, כאשר ידוע מחירם של גרסאות אחרות של גלאקסי. בנוסף, תוכנית זו תדפיס את ערכי רכיבי הגרדיאנט של w ו-b כל 200 איטרציות, ולבסוף תדפיס את המחיר המנובא.

import numpy as np

galaxy\_data = np.array([[2,70],[3,110],[4,165],[6,390],[7,550]]) #Galaxi versions and their price

w = 0

b = 0

alpha = 0.01

for iteration in range(10000):

deriv\_b = np.mean(1\*((w\*galaxy\_data[:,0]+b)-galaxy\_data[:,1]))

deriv\_w = np.mean(galaxy\_data[:,0]\*((w\*galaxy\_data[:,0]+b)-galaxy\_data[:,1]))

b -= alpha\*deriv\_b

w -= alpha\*deriv\_w

if iteration % 200 == 0 :

print("it:%d, grad\_w:%.3f, grad\_b:%.3f, w:%.3f, b:%.3f" %(iteration, deriv\_w, deriv\_b, w, b))

print("Estimated price for Galaxy S5: ", w\*5 + b)

## רגרסיה ליניארית מרובה

כאשר יש k תכונות לכל אובייקט ולא רק תכונה אחת, השיטה מאוד דומה אלא שכעת כל אובייקט אינו ערך בודד אלא וקטור , כאשר כל מייצג את התכונה ה-j של האובייקט . וכן w כבר לא יהיה ערך בודד אלא וקטור , כאשר כל מייצג את המשקל שאנו נותנים לכל יחידה מהתכונה j. הוא למעשה מישור שהמכפלה שלו ב- היא המרחק של האובייקט מהמישור. מטרתנו היא למצוא ו-b שמצמצמים את , כך שמרחק כל האובייקטים בדאטה מהמישור הוא מינימלי.

ישנם וריאציות שמוסיפים תכונה נוספת (אינדקס 0) ל-k התכונות שתייצג את b וכך מפשטים את המשוואה. בדרך כלל לא נעשה זאת אך יש כאלו שכן וצריך להכיר את זה. לשם כך צריך להגדיר בכל האובייקטים: , וכן . כעת המטרה היא לצמצם את:

האלגוריתם שתיארנו לעיל נשאר אותו הדבר אלא שכעת בוקטור הגרדיאנט יש k+1 כניסות, 1 לכל תכונה ועוד b. כל הכניסות נראות אותו הדבר כמו בוקטור הגרדיאנט שבנינו לעיל. בכל איטרציה של הלולאה, עבור כל , יש לחשב את הנגזרת החלקית שלו ולהציב את , ואז ולעדכן . אותו הדבר בכל איטרציה גם עבור b.

## איכות המודל

כדי לבדוק אם מודל רגרסיה ליניארית איכותי, נשווה בין המודל שלנו למודל שתמיד מנבא את הערך הממוצע של כל ערכי ה-Y (labels). ההשוואה תהיה בין הערכים המוחזרים מפונקציית ה-Loss הסופי של שני המודלים. כאמור לעיל, הערך המוחזר מפונקציית Loss מייצג את הסטייה הממוצעת של ניבוי המודל מ-Y. ככל שהסטייה של המודל שלנו קטנה מהסטייה של מודל שתמיד מנבא ממוצע כך המודל שלנו יותר טוב. ההשוואה האמיתית צריכה להיות כאשר הסטייה של המודל שלנו מחושבת על ה-test ואילו סטיית המודל שתמיד מנבא ממוצע מחושבת על ה-train.

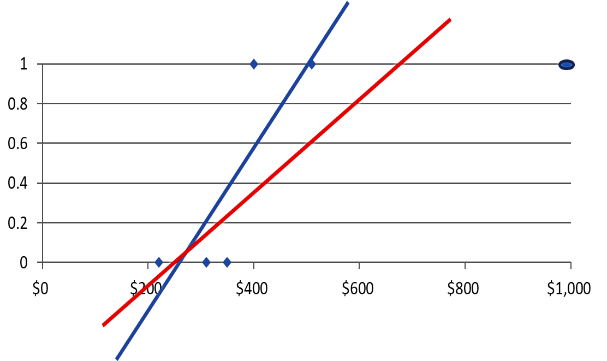
## סוגי רגרסיה נוספים

עבור דאטה שבו הסיווג תלוי בגורם נוסף של זמן, כמו שהסיווג הוא מחזורי או תלוי ברצך של זמן (time series), יש מודלים נוספים לרגרסיה כמו ARMA ו-ARIMA המתחשבים בנתונים אלו.

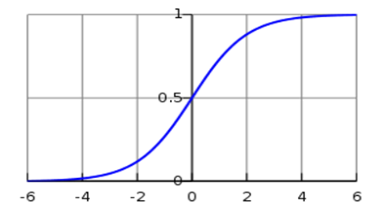
# Logistic Regression

## הסבר השיטה

רגרסיה לוגיסטית זוהי שיטה המנסה להתאים את הרגרסיה הליניארית כדי שתוכל לפתור בעיית Classification, כלומר במקום לנבא ערך לאובייקט נרצה לסווג אותו למחלקה מתאימה. בשלב ראשון נסביר כיצד לבצע התאמה זו עבור בעיית סיווג של שתי מחלקות. בעיות אלו הן לרוב בעיות שבאות לענות על שאלת כן או לא. לדוגמה, האם אדם הוא מועסק או לא או האם לקוח יבצע רכישה או לא. בהמשך נלמד כיצד לבצע את ההתאמה עבור בעיות סיווג של מספר מחלקות.



המחשבה הראשונית היא לייצג את שתי המחלקות באמצעות 0 ו-1, כאשר 0 התשובה היא לא ו-1 היא כן. במצב כזה נוכל להפעיל את שיטת הרגרסיה הליניארית על אובייקט x, כמו שלמדנו לעיל, ואזי אם הערך y שהתקבל קרוב יותר ל-1 מאשר ל-0 נעריך ששייך למחלקה 1, אחרת נסווג אותו למחלקה 0. אמנם בשיטה כזו במקרים שבהם יש אובייקט עם ערך מאוד גבוה או מאוד נמוך ביחס לקבוצה של אובייקטים שהוא נמצא איתם באותה מחלקה, כל המשוואה הליניארית תינטה אל אותו אובייקט. טעות זו יכולה להוביל לשגיאות חמורות.



לכן במקום פונקציה ליניארית נרצה למצוא פונקציה שבה סך הסטיות של כל m האובייקטים ממנה הוא מינימלי אך גם תצמצם את הפגיעה באיכות המשוואה במקרי קיצון שתיארנו. פונקציה כזו היא Sigmoid Function, שבה ככל שערכי ה-x גבוהים היא הולכת ומתקרבת ל-1, וככל שערכי x הולכים וקטנים היא מתקרבת ל-0.

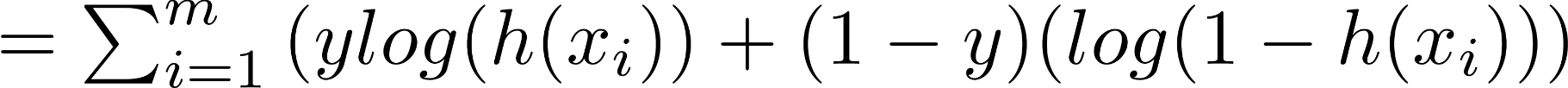
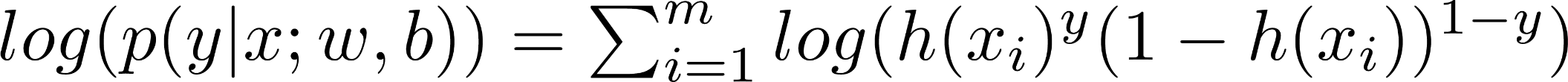
לאחר שנמצא את w ו-b באמצעות שיטת Gradient Descent, נציב אותם בפונקציית סיגמואיד h(x). כל אובייקט x שנרצה לסווג אותו למחלקה נציב אותו בפונקציה שהתקבלה. הערך שיחזור מהפונקציה הוא מה ההסתברות שהאובייקט שייך למחלקה 1, . אם ערך זה גדול מ-0.5 נעריך ששייך למחלקה 1, אחרת נסווג אותו למחלקה 0.

### חישוב פונקציית Loss

כאמור, *ההסתברות שהחיזוי עבור אובייקט יהיה 1 או 0 הוא*  ו- *בהתאמה. מכאן שההסתברות שהחיזוי עבור אובייקט יהיה הוא: .*

נרצה לחשב עבור חלק ה-train בדאטה מה ההסתברות שהחיזוי שלנו עבור כל אובייקט יהיה שווה בדיוק לדאטה. לשם כך נצטרך להכפיל את החישוב לעיל עבור כל אובייקט . נרצה כמובן למקסם ערך זה. .

אם נפעיל פונקציית log על הביטוי לעיל ונשתמש בחוקי log, נוכל לקבל סכום במקום מכפלה.



כעת כיוון שמקובל למזער ולא למקסם נוסיף מינוס בהתחלה. בנוסף, נחלק ב-m כדי לקבל סטייה ממוצעת לכל אובייקט, ולא שאם נגדיל מספר האובייקטים אזי גם הסטייה תגדל. סך הכל פונקציית ה-loss אותה נרצה למזער עבור w ו-b היא:

נזכיר שאם לכל אובייקט יש k תכונות ולא רק תכונה אחת, אזי כל אובייקט אינו ערך בודד אלא וקטור , כאשר כל מייצג את התכונה ה-j של האובייקט . וכן w כבר לא יהיה ערך בודד אלא וקטור , כאשר כל מייצג את המשקל שאנו נותנים לכל יחידה מהתכונה j.

### חישוב הגרדיאנט

נוכל למצוא w ו-b מינימליים באמצעות Gradient Descent. וקטור הגרדיאנט עבור נראה אותו דבר, אלא שיש להחליף את הפונקציה הליניארית בפונקציית סיגמואיד . וקטור הגרדיאנט יראה כך:

האלגוריתם שבו נשתמש כדי למצוא ערכי w ו-b עבורם נקבל מינימלי זהה לאלגוריתם שלמדנו ברגרסיה ליניארית:

* קבע איזשהו שהוא גודל הצעד על w ו-b.
* בלולאה עד שמשיגים התכנסות (convergence):
* חשב את , *כאשר ב-*מציבים את w ו-b הנוכחיים.
* חשב את , *כאשר ב-*מציבים את w ו-b הנוכחיים.
* עדכן את w להיות .
* עדכן את b להיות .
* החזר את w ו-b.

## השוואה ל- Naïve Bayes

Naïve Bayes הוא מודל יוצר (generative model) שבו מחשבים עבור כל מחלקה y מה ההסתברות שאובייקט x שייך למחלקה זו , וכן מה ההסתברות שאובייקט כלשהו שייך למחלקה y . לעומת זאת, Logistic Regression הוא מודל מפלה (discriminative model) שבו מחשבים רק מה ההסתברות שאובייקט x שייך למחלקה כלשהי y *. בדרך כלל מודל מפלה הוא יעיל יותר מאשר מודל יוצר.*

## דוגמה ב-Python

יש לנו בסיס נתונים של אנשים ואנחנו רוצים לשלוח להם הצעות עבודה, אך לשם כך עלינו להעריך מי מועסק ומי לא. אנו מכירים מידע זה רק על חלק קטן מהנתונים. ברצוננו לבנות Classifier הקובע אם משתמש מועסק או לא, בהתבסס על מגדר (0-זכר, 1-נקבה), גיל ושנות הניסיון של המשתמש.

import numpy as np

data\_x = np.array([[1,28,4],[1,60,34],[1,25,3],[0,54,20],[0,24,2],[0,39,12],[0,30,4],[1,36,10],[1,26,1],[0,44,9]])

data\_y = np.array([1,1,1,1,1,1,1,0,0,0]) # data\_x is the gender, age, and experience. data\_y 1 means employed

def h(x,w,b): # sigmoid function

return 1 / (1+np.exp(-(np.dot(x,w) + b)))

w = np.array([0.,0,0])

b = 0

alpha = 0.001

for iteration in range(100000):

deriv\_b = np.mean(1\*( (h(data\_x,w,b)- data\_y)))

deriv\_w = np.dot((h(data\_x,w,b) - data\_y), data\_x)\*1/len(data\_y)

b -= alpha\*deriv\_b

w -= alpha\*deriv\_w

print("User [1, 49, 8] probability of working: ", h(np.array([[1, 49, 8]]),w,b))

## ריבוי מחלקות

כאשר יש מחלקות שאנו רוצים לסווג אליהן, ולא רק 2, משתמשים בשיטה שנקראת Softmax כדי לסווג אובייקט x למחלקה מתאימה. שיטה זו מתוארת בפרק הבא.

## איכות המודל - F-Measure

כדי לוודא את איכות המודל נבנה Confusion Matrix על התוצאות שנלקחו מהרצת המודל על ה-validation או על ה-test. במטריצה זו נציין כמה אובייקטים המודל ניבא נכון וכמה לא ניבא נכון מתוך החיוביים ומתוך השליליים. המטריצה משמאל מתאימה לבעיית Classification עם שתי מחלקות. נגדיר:

* Recall – מהו האחוז שהגדרנו שהם מתאימים למחלקה מבין כל אלו שאכן מתאימים למחלקה. בציור לעיל נחשב:

.

* Precision – מהו האחוז של המתאימים למחלקה מבין אלו שהמודל זיהה שהם מתאימים למחלקה. בציור לעיל נחשב:

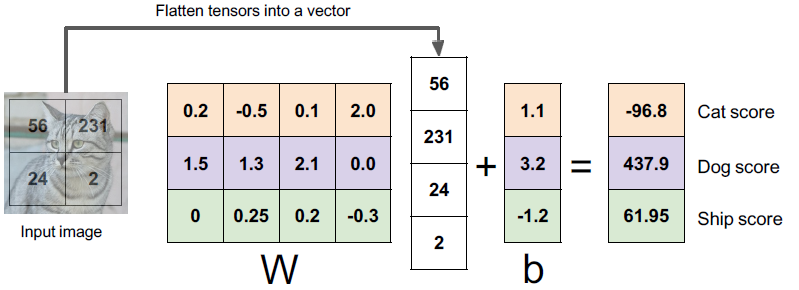
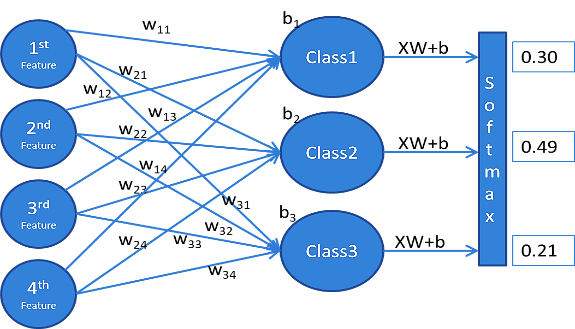
השיטה שבה נמדוד את האיכות של המודל נקראת F-Measure המחשבת ממוצע הרמוני באמצעות שני הערכים שהגדרנו לעיל. ככל שערך זה גדול יותר כך המודל מדויק יותר.

# Softmax

## הגדרה

כאשר ישנם k מחלקות שאנו רוצים לנבא עבור אובייקט לאיזה מהן הוא שייך, נשתמש בשיטת Softmax כדי לעשות זאת. בשיטה זו בונים מודל כמו Logistic Regression, אלא שכאן המודל ינבא מה ההסתברות שהוא שייך לכל אחד מ-k המחלקות. כלומר, לכל מחלקה i, המודל יחזיר מה ההסתברות שאובייקט uשייך ל-i, כך שסכום כל ההסתברויות הוא בדיוק 1. נשייך את u למחלקה עם ההסתברות הגבוהה היותר.

כיצד מודל Softmax עובד? נסמן את הדאטה X, את מספר האובייקטים בדאטה n, מספר התכונות f ומספר המחלקות k. במקום ש-W יהיה וקטור באורך f ו-b ערך בודד, כמו ב-Logistic רגיל, **מייצרים W שהוא מטריצה מסדר גודל של** , **כלומר מספר השורות הוא מספר התכונות ומספר העמודות הוא מספר המחלקות, ו-b הוא וקטור באורך k**. כעת הכפלה של הדאטה ב-W והוספת b ייתן לנו מטריצה בגודל , כלומר יש k ערכים לכל אובייקט. k ערכים אלו נקראים “logits”. כדי לקבל את k ההסתברויות לכל אובייקט, נחלק כל אחד מ-k הערכים בסכום k הערכים.



לסיכום, החישוב שהמודל יבצע כדי לקבל את וקטור ההסתברות y המכיל את ההסתברות שכל אובייקט שייך למחלקה i הוא:

פונקציה זו נקראת פונקציית Softmax. נשים לב שכאשר Softmax הוא Logistic Regression רגיל, לכן Softmax הוא הכללה של Logistic Regression. ב-TensorFlow נוכל להשתמש בפקודות הבאות כדי לקבל את k ההסתברויות לכל אובייקט ב-X:

W = tf.Variable(tf.zeros([features,categories]))

b = tf.Variable(tf.zeros([categories]))

y = tf.nn.softmax(tf.matmul(X, W) + b)

## פונקציית Loss - Cross Entropy

בשיטה זו פונקציית ה-Loss היא האנתרופיה, המייצגת שיעור אי-ודאות של מאורע. חישוב האנתרופיה למאורע X שיש לו מספר אפשרויות עם הסתברות שונה לכל אפשרות הוא:

אמנם כאן ההסתברויות אינן בהכרח תואמות את ההתפלגות האמיתית, לכן אנו נחשב cross entropy, שבה **מצליבים** את ההסתברויות שקיבלנו עם ההסתברויות האמיתיות. בנוסף, נרצה לחשב את האנתרופיה הממוצעת לכן נחלק במספר האובייקטים n. חישוב פונקציית ה-Loss עבור כל מחלקה i הוא:

כאשר זוהי ההסתברות האמיתית שאובייקט x שייך למחלקה i, ו- זה ההסתברות ש-x שייך למחלקה i לפי המודל שלנו. נשים לב שכיוון שאנו יודעים בוודאות לאיזה מחלקה שייך כל אובייקט, היא 1 עבור אותה מחלקה ו-0 עבור שאר המחלקות. לכן חשוב לשמור את ערכי האמת לכל אובייקט בוקטור בגודל k בו יש 1 בכניסה i אם האובייקט שייך למחלקה i, ו-0 בכל שאר הכניסות (one-hot encoding). למעשה, החישוב של פונקציית ה-Loss עבור כל מחלקה i הוא אם כן:

### בעיות חישוב

אם פונקציית ה-Softmax תחזיר עבור אובייקט במחלקה מסוימת 0 או 1 אזי נקבל בחישוב פונקציית Loss ערך NaN (Not a Number). מאוד קל להגיע למצב זה שכן יכול להגיע בקלות לאינסוף או 0 ואז הניבוי יהיה 0 או 1. כדי לפתור זאת נשתמש בפונקציה מותאמת של TensorFlow שמחשבת בבת אחת את פונקציית ה-Softmax עם פונקציית ה-Loss, ואז ניתן להפריד בין ה-logits מהחזקה של e ולקבל חישוב מדויק שלא מגיע לטעות כזו. פונקציה זו היא:

loss = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits\_v2(y\_, z))

## תבנית למודל Softmax

categories = 10

x = tf.placeholder(tf.float32, [None, features])

y\_ = tf.placeholder(tf.float32, [None, categories])

W = tf.Variable(tf.zeros([features,categories]))

b = tf.Variable(tf.zeros([categories]))

z = tf.matmul(x, W) + b

loss = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits\_v2(y\_, z))

update = tf.train.GradientDescentOptimizer(0.001).minimize(loss)

…

# שיפורים כלליים למודל למידת מכונה

## רגולריזציה

במודלים רבים נותנים "משקלים" לכל תכונה. רגולריזציה זוהי שיטה שבה אנו נותנים "עונש" למודל על כל משקל שהוא יקבע. מטרתנו היא להוריד כוח מהמודל ולגרום לו להיות זהיר יותר בנתינת משקלים גדולים מדי לתכונות אלא אם כן זה מאוד משתלם. המטרה כאן היא למנוע מהמודל להיכנס ל-overfitting, כלומר שהמודל טוב מדי על ה-train אך פחות טוב על ה-validation. נעשה זאת על ידי הוספה לפונקציית Loss סכום כלשהו התלוי במשקלים שהמודל קבע במצב נוכחי. ככל שהמשקל יהיה גדול יותר כך ה-Loss יגדל, ולכן יהיה יותר זהיר. יש שתי שיטות מרכזיות לרגולריזציה. שתי השיטות מתבססות על נוסחאות p-Norm ועל המשקלים W.

Spam:

* Buy it, pay later! Click me!
* You Won 10000 Dollars! Click here!

Real (Non-Spam):

* Will See you later.
* Will you want to meet later?
* I am waiting for you.

Test:

- Are you paying too much? Click now!

### Lasso

בשיטה זו מוסיפים לפונקציית Loss את סכום כל המשקלים בערך מוחלט כפול קבוע . הוא היפר-פרמטר, הנקרא “regularization strength”, שיש לבצע מספר ניסויים כדי למצוא את הערך הטוב ביותר עבורו.

Lasso מאוד אגרסיבית עבור תכונות עם משקלים קטנים ועלולה לגרום למודל לאפס את המשקלים עליהם כך שלא ישפיעו כלל. בעייתיות נוספת של שיטה זו היא שקשה לגזור ערך מוחלט. יש לכך מספר פתרונות:

1. להשתמש Coordinate Gradient Descent. לא ממומש ב-TensorFlow.
2. להשתמש באופטימייזר Adam.
3. להפחית את גודל הצעד ב- Gradient Descentככל שמתקדמים עם הצעדים - alpha\*=0.99.

### Ridge

בשיטה זו מוסיפים לפונקציית Loss את סכום כל המשקלים בריבוע חלקי 2, ומכפילים בקבוע .

מחלקים ב-2 כדי שיצטמצם עם הנגזרת. Ridge תגרום למודל להפחית משקולות גבוהים אך לא תאפס את הנמוכים, אלא תגרום למשקלים להיות מבוזרים פחות או יותר בצורה אחידה. משתמשים ברגולריזציה זו כאשר רוצים לגרום למודל להתחשב גם בתכונות פחות חשובות. ב-TensorFlow ניתן להשתמש בפונקציה המובנית tf.nn.l2\_loss(W) שמבצעת את אותה פעולה.

### Elastic Net

שילוב של Lasso עם Ridge.

## Imbalanced Data

כאשר יש המון מידע ממחלקה אחת אך יש רק מעט מידע ממחלקה אחרת, המודל יכול ללמוד לחזות נכון רק עבור המחלקה הראשונה, שכן עיקר ה-Loss נובע משגיאות על מחלקה זו. אולם בדרך כלל נהיה מעוניינים להיות מדויקים יותר עבור המחלקה השנייה. לדוגמה, כאשר נרצה לחזות האם לאדם יש מחלה כלשהי, לרוב האנשים התשובה תהיה שלילית אך אנו מעוניינים לדייק יותק דווקא עבור אלו שכן חולים. יש מספר דרכים להתמודד עם דאטה לא מאוזן:

1. Min-Batch - נחלק את קבוצת ה-train למחלקה שמיוצגת הרבה ולמחלקה שמיוצגת מעט. נשתמש ב-min-batch כך שמידע אותו נריץ יהיה כל פעם חלק אחר מהמחלקה שמיוצגת הרבה אך נצרף אליה כל פעם את המידע של המחלקה שמיוצגת מעט.
2. Under-sampling - לא להשתמש בכל המידע שיש לנו אלא לדגום ב-train ממחלקת הרוב קבוצה שווה לגודל מחלקת המיעוט וזה יהיה ה-train החדש שלנו.
3. Over-sampling - נשכפל את המידע במחלקה הקטנה מספר פעמים. יש דרך קלה ויעילה לעשות זאת באמצעות טכניקת SMOT (Synthetic Minority Oversampling Technique).
4. Loss function intervention - בדר"כ 1 יהיה המחלקה הקטנה ו-0 המחלקה הגדולה. נרצה "להעניש" את המודל על כל אובייקט שינבא 0 ובפועל הוא 1. לכן נכפיל בפונקציית Loss את הרכיב שסוכם טעות זו בקבוע , כלומר נכפיל במנה של מספר האובייקטים הכולל ב-train במספר האובייקטים השייכים למחלקת המיעוט. **זוהי השיטה המומלצת לבעיית דאטה לא מאוזן**. בספריות למידת מכונה לכל מודל יש פרמטר שאחראי לאזן את הדאטה בצורה זו.
5. Reduce prediction threshold - במקום לקבוע ש0.5 זהו הסף שמעליו נשייך למחלקה 1 ומתחתיו נשייך למחלקה 0, נקבע סף נמוך יותר כך שיותר אובייקטים ינובאו שייכים למחלקה 1. זוהי שיטה לא מומלצת.