בית הספר להנדסה ומדעי המחשב ע"ש רחל וסלים בנין

מבוא למדעי המחשב 67101

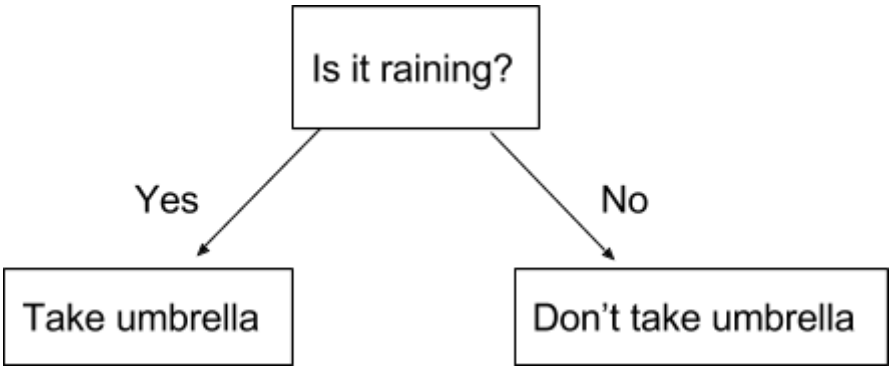
תרגיל 10 - עצים

להגשה בתאריך 03/01/2018 בשעה 22:00

בתרגיל זה נתרגל שימוש בעצים כמבנה נתונים.

בתרגיל נעבוד עם עצים בינאריים מושרשים, ובפרט נעבוד עם עצי החלטה. עץ בינארי הוא עץ בו לכל קודקוד יש לכל היותר שני בנים. אנו נעסוק בעצים בהם לכל קודקוד יש בדיוק 2 בנים או 0 בנים (כלומר הוא עלה). יש לציין ששורש ללא בנים הוא גם עלה. עץ החלטה הוא עץ בינארי בו כל קודקוד שאיננו עלה מייצג שאלת כן ולא. לכל קודקוד יש בן אחד המתאים לתשובה "כן" ובן אחד המתאים לתשובה "לא". כשנעבור על קודקודי העץ, בכל שלב נתבונן בשאלה על הקודקוד הנוכחי, נבדוק אם התשובה היא כן או לא, ואז נתקדם לבן המתאים. כל עלה מייצג החלטה שנקבל בהתאם למסלול שעברנו בו.

דוגמא לעץ החלטה פשוט הינה:



בעץ ההחלטה הנ"ל השורש הוא קודקוד המייצג את השאלה "האם יורד גשם?". לקודקוד זה יש שני בנים. הבן השמאלי מתאים לתשובה "כן" ומשום שהוא עלה הוא מייצג את ההחלטה לקחת מטריה. הבן הימני מתאים לתשובה "לא" ומשום שהוא עלה הוא מייצג את ההחלטה לא לקחת מטריה.

שלד הקובץ ex10.py

כדי לפתור את התרגיל תקבלו שלד עבור התרגיל. בקובץ תהיה מוגדרת המחלקה Node. כל אובייקט מסוג Node הינו קודקוד בעץ.

ל-Node יהיו השדות הבאים:

- השדה data. אמור להכיל ערך מסוג String. אם ה-Node איננו עלה, השדה מייצג שאלה לשאול בקודקוד זה. אם ה-Node הינו עלה, השדה מייצג את ההחלטה שהתקבלה בעלה זה.
- השדות positive_child ו-negative_child. אם ה-Node איננו עלה בשדות יופיעו אובייקטים מסוג Node, כאשר ב-positive_child יהיה הבן התואם לתשובה "כן" על השאלה ב-data וב-negative_child יופיע הבן התואם לתשובה "לא". אם ה-Node הינו עלה, בשני השדות יהיה הערך None.

שדות המחלקה מתוארים גם בטבלה הבאה:

Field	Type	Description
data	String	Non-leaf node - The question asked at this node

		Leaf node - The decision indicated by this leaf
positive_child	Node	Non-leaf node - The node that matches a positive answer to the question Leaf node - None
negative_child	Node	Non-leaf node - The node that matches a negative answer to the question Leaf node - None

כמו כן, ל-Node יהיה הבנאי הבא:

Node(data, positive_child, negative_child)

הבנאי ייצור אובייקט מסוג Node עם הארגומנטים כשדות.

בנוסף, תוגדר מחלקה בשם Record (מעתה ואילך Record ייקרא גם רשומה). מבנה הנתונים יכיל שני שדות:

- השדה illness. בשדה זה יהיה String עם שם של מחלה כלשהי.
- השדה symptoms. בשדה זה תהיה רשימה של אובייקטים מסוג String, שכל אחד מהם מייצג סימפטום אפשרי. שימו לב כי רשימה זה יכולה להיות ריקה.

לדוגמא, ברשומה מסויימת יכולים להיות השדות הבאים:

record.illness == “influenza”

record.symptoms == [“fever”, “fatigue”, “headache”, “nausea”]

בקובץ תהיה גם הפונקציה הבאה:

parse_data(filepath)

הפונקציה תקבל נתיב של קובץ ותחזיר רשימה של אובייקטים מסוג Record. כל איבר ברשימה מייצג חולה אחד שאובחן במחלה illness ויכיל את הסימפטומים symptoms שנלוו לה.

לבסוף, תוגדר המחלקה Diagnoser. למחלקה יהיה הבנאי הבא:

Diagnoser(root)

הבנאי ישמור את השורש בשדה __root, ואתם תממשו את יתר המתודות.

ניתן להוסיף מתודות למחלקות הקיימות בקובץ השלד לפי הצורך.

חלק ראשון: שימוש בעץ החלטה

בחלק זה נניח שכבר בנינו עץ החלטה. עץ ההחלטה כולו ייוצג על ידי שורש העץ שהינו אובייקט מסוג Node. שימו לב שבקובץ השלד שקיבלתם נבנה עבורכם עץ ידנית ונכתבה דוגמא לשימוש בו (בחלק השני של התרגיל תבנו עצי החלטה). כל הפונקציות בחלק זה הינן מתודות של המחלקה Diagnoser. המחלקה Diagnoser תקבל שורש של עץ החלטה, תשמור אותו כשדה, וכל יתר הפונקציות ישתמשו באותו העץ. חשוב לציין שעל כל המתודות של Diagnoser לא לשנות את מבנה העץ או את המידע השמור עליו כלל.

1. ממשו את המתודה:

def diagnose(self, symptoms):

הפונקציה תקבל רשימה של סימפטומים ותשתמש בשורש עץ ההחלטה השמור ב-self. על הפונקציה להתחיל מהשורש, לבדוק האם הסימפטום עליו שואל השורש נמצא ברשימת הסימפטומים, ולהתקדם לבן המתאים. כלומר, אם הסימפטום נמצא ברשימה להתקדם לבן שתואם לתשובה "כן" ואם הוא לא נמצא ברשימה להתקדם לבן שתואם לתשובה "לא". לאחר מכן יש לחזור על אופן הפעולה עד שמגיעים לעלה.

הפונקציה תחזיר את המחלה שנמצאת על העלה שהגיעה אליו.

2. ממשו את המתודה:

def calculate_error_rate(self, records):

כשנבנה עץ, נקבל רשימה של Records מ-parse_data שבכל אחת רשימת סימפטומים symptoms ומחלה תואמת illness. כשנבנה עץ, נרצה שלאחר מעבר על העץ עם הרשימה symptoms נענה את המחלה illness לכמה שיותר רשומות. כדי לבדוק את איכות העץ נממש את הפונקציה calculate_error_rate.

הפונקציה תקבל רשימה של אובייקטים מסוג Record ותשתמש בשורש של עץ ההחלטה השמור ב-self. הפונקציה תחזיר את היחס בין מספר השגיאות של העץ על הרשומות ב-Records לבין מספר הרשומות בסך הכל. כדי לעשות זאת, הפונקציה תעבור על כל אחת מהרשומות ברשימה records, תחשב דיאגנוזה לפי העץ של הרשימה symptoms מתוך הרשומה ותבדוק האם התקבלה המחלה illness מתוך אותה רשומה. הפונקציה תחלק את מספר הפעמים בהן לא התקבלה המחלה הנכונה במספר הרשומות בסך הכל ותחזיר את התוצאה.

3. ממשו את הפונקציה:

def all_illnesses(self):

הפונקציה תשתמש בשורש עץ ההחלטה השמור ב-self ותחזיר רשימה של כל המחלות השמורות על עלי העץ. כלומר, יש להגיע לכל עלי העץ ולשמור את שדה ה-data של כל עלה. על כל מחלה להופיע ברשימה פעם אחת בלבד. כמו כן, על הרשימה להיות ממוינת לפי סדר לקסיקוגרפי. ניתן וכדאי להשתמש באלגוריתמים שנלמדו בכיתה למעבר על קודקודי עץ.

4. ממשו את הפונקציה:

def most_common_illness(self, records):

הפונקציה תקבל רשימה של records כארגומנט, תעבור על כל אחת מהרשומות ותבדוק לאיזו מחלה נגיע אם נתקדם לפי רשימת הסימפטומים ברשומה (כלומר, מה הערך שהפונקציה diagnose הייתה מחזירה על רשימת הסימפטומים ברשימה). הפונקציה תעשה זאת עבור כל רשומה ולבסוף תחזיר את המחלה אליה הגענו הכי הרבה פעמים. שימו לב כי ייתכן שיש יותר מעלה אחד המייצג כל מחלה.

5. ממשו את הפונקציה:

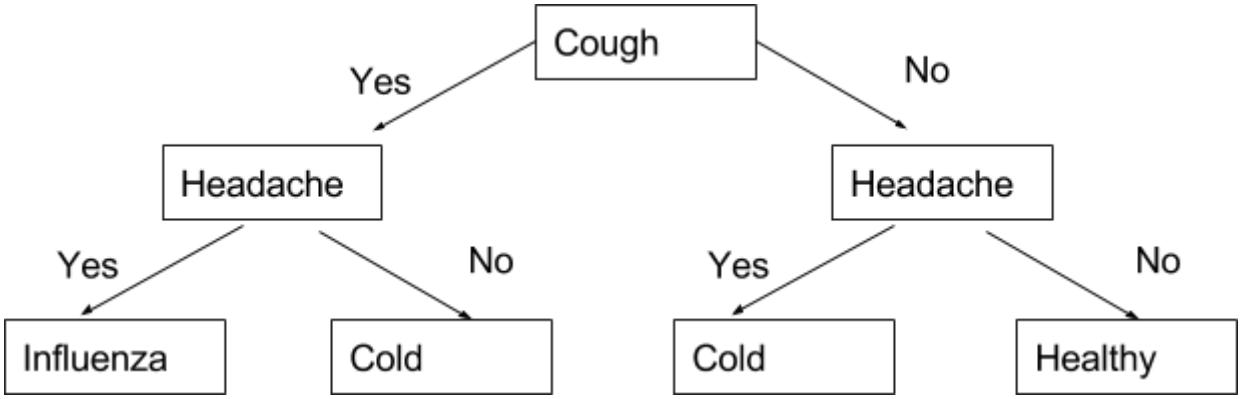
def paths_to_illness(self, illness):

הפונקציה תקבל ארגומנט illness מסוג String. הפונקציה תעבור בכל המסלולים בעץ ותחזיר רשימה של כל המסלולים שמגיעים לעלה שעליו המחלה illness. כדי לייצג מסלול נשים לב שמספיק להגיד מה התשובה שנתנו בכל אחד מקודקודי העץ. כלומר, עבור כל מסלול שמסתיים במחלה illness מתאימה רשימה של ערכי True,False כך שבמקום ה-i ברשימה יהיה True אם בצעד ה-i ענינו כן ואחרת במקום ה-i ברשימה יהיה False. שימו לב כי אנו

מתחילים לספור את הצעדים בצעד ה-0. כמו כן, חשוב לציין כי ייתכן שיהיו כמה מסלולים שיגיעו לאותה המחלה (בעלים שונים בעץ).

לכן, ערך ההחזרה אמור להיות רשימה של רשימות. בכל אחת מהרשימות הפנימיות יהיו רק ערכי True,False המייצגים את המסלולים. אם אין אף מסלול המגיע למחלה יש להחזיר רשימה ריקה. אין חשיבות לסדר הופעת הרשימות.

למשל, עבור העץ הנ"ל:



הקריאה

```
paths_to_illness(self, "Cold")
```

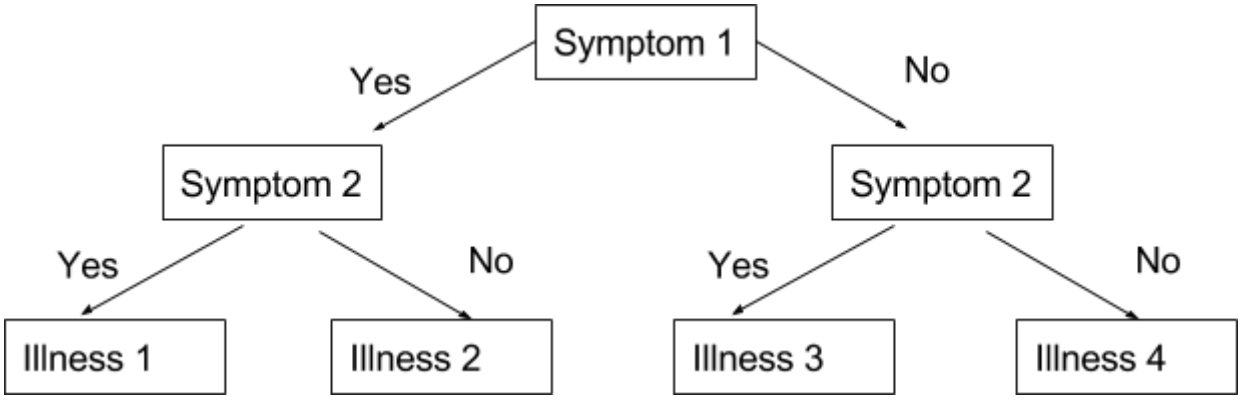
תחזיר את הרשימה:

```
[[True, False],[False,True]]
```

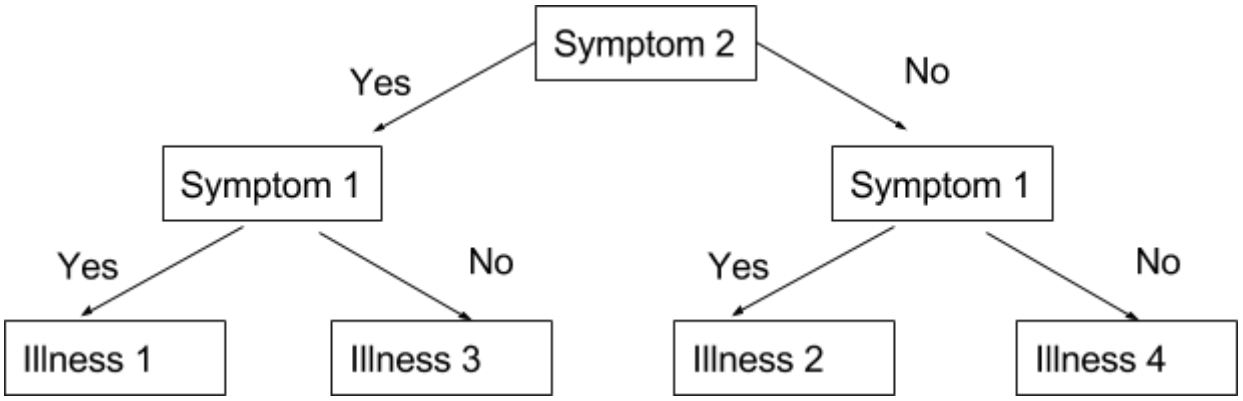
חלק שני: בניית עץ החלטה

בחלק זה נבנה עץ החלטה עם אחוז שגיאה נמוך עבור המידע שהתקבל מ-parse_data. שימו לב כי אם נשאל על כל אחד מהסימפטומים האפשריים נוכל יחסית בקלות לבנות עץ החלטה אופטימלי (כיצד היינו עושים כזה דבר?). לכן נרצה להגביל את עצי ההחלטה שלנו לעצים ששואלים על מספר קטן יחסית של סימפטומים.

כמו כן, שימו לב שאם עץ תמיד שואל על רשימת סימפטומים כלשהי symptoms לפני שהוא מגיע לעלה, סדר השאלות לא משנה. כלומר העץ:



והעץ:



יחזירו בדיוק את אותו אחוז שגיאה. שכנעו את עצמכם שקביעה זו נכונה.

שימו לב כי הפונקציות הבאות אינן מתודות של Diagnoser.

6. ממשו את הפונקציה:

def build_tree(records, symptoms):

הפונקציה תבנה עץ השואל בדיוק על הסימפטומים ברשימה symptoms לפי סדר הופעתם ברשימה ותחזיר את שורש העץ שנבנה. כלומר בשורש ישאלו על symptoms[0], בבניו ישאלו על symptoms[1] וכן הלאה. ניתן לעשות זאת בכך שבשורש העץ נשים את הערך הראשון ברשימה, symptoms[0], ואז בכל אחד מבניו נבנה תת עץ השואל על כל יתר הסימפטומים ב-symptoms, כך שבכל עומק בעץ, כל הצמתים המקבילים יבדקו את אותו הסימפטום. שימו לב כי זוהי הגדרה רקורסיבית.

ניתן להניח שהרשימות records,symptoms לא ריקות.

כשנגיע לעלי העץ נצטרך לבחור מחלה כלשהי שהיא האבחנה שמספק העץ במקרה זה. בכל עלה נבדוק מה הסימפטומים שמובילים לעלה מהשורש. כדי למזער את השגיאה של דיאגנוזה עתידית, נבחר לבנות את העץ כך שבכל עלה תופיע המחלה המופיעה במספר המרבי של רשומות התואמות למסלול שמגיע לאותו עלה. נאמר שרשומה תואמת למסלול אם כל הסימפטומים עליהם אמרנו "כן" לאורך המסלול מופיעים ברשימת הסימפטומים ברשומה וכל הסימפטומים עליהם אמרנו "לא" לא מופיעים ברשימת הסימפטומים. שימו לב כי העץ לא בהכרח יכיל את כל הסימפטומים או המחלות המופיעים ב-records.

במקרה של שוויון בין דיאגנוזות שונות, ניתן לבחור כל אחת מהדיאגנוזות עם המספר המקסימלי של רשומות תואמות. בפרט עבור המקרה בו לא נמצאה אף רשומה תואמת, ניתן לבחור בכל אחת מהדיאגנוזות האפשריות.

<u>הערה</u>: שימו לב כי ייתכן שכדאי להגדיר פונקציית עזר רקורסיבית עבור סעיף זה.

7. ממשו את הפונקציה:

def optimal_tree(records, symptoms, depth):

ניתן להניח כי הרשימות records,symptoms לא ריקות וכי len(symptoms) >= depth > 0.

הפונקציה תחזיר את העץ עם אחוז השגיאה הנמוך ביותר שתמיד שואל על מספר סימפטומים השווה ל-depth.

כדי לעשות זאת, הפונקציה תעבור על כל תתי הקבוצות בגודל depth של קבוצת הסימפטומים symptoms, תבנה עץ השואל בדיוק על הסימפטומים בתת הקבוצה שנבחרה, ותבדוק את אחוז השגיאה על אותו העץ. כדי לבחור את תתי הקבוצות בגודל מסוים, ניתן להשתמש בפונקציה combinations כדי לבנות את כל תתי הקבוצות.

כדי לקרוא על הפונקציה ולראות דוגמאות לשימוש בה ניתן לקרוא את הדוקומנטציה של פייתון בלינק הבא:

https://docs.python.org/3.6/library/itertools.html#itertools.combinations

בסוף התהליך הפונקציה תחזיר את השורש של העץ עם אחוז השגיאה הנמוך ביותר. ניתן לבנות עצים אלו בעזרת הפונקציה build_tree.

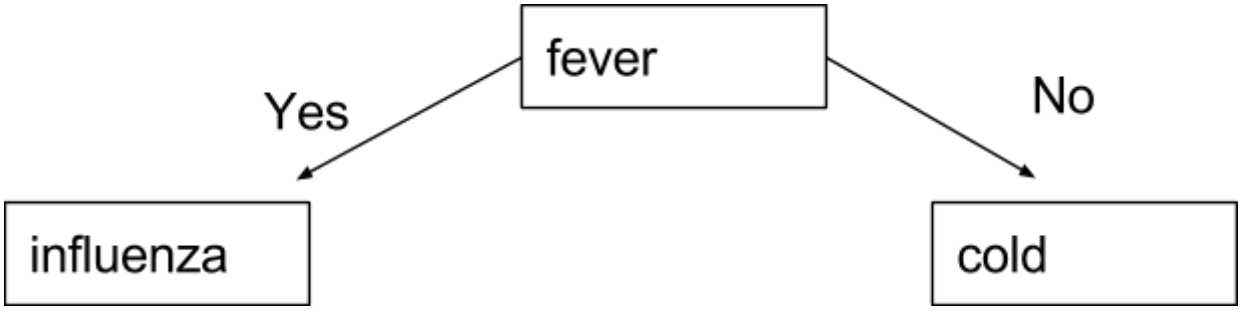
לדוגמא, עבור הפונקציות 6,7 עם הרשימה records הבאה:

```
record1 = Record("influenza", ["cough", "fever"])
record2 = Record("cold", ["cough"])
records = [record1, record2]
```

נצפה ששתי הקריאות:

```
build_tree(records, [“fever”])
optimal_tree(records, [“cough”,”fever”], 1)
```

יחזירו את שורש העץ:



נהלי הגשה

הלינק להגשה של התרגיל הוא תחת השם: ex10

בתרגיל זה עליכם להגיש את הקבצים הבאים:

1. ex10.py – עם המימושים שלכם לפונקציות.
2. README (על פי פורמט ה-README לדוגמא שיש באתר הקורס, ועל פי ההנחיות לכתיבת README המפורטות בקובץ נהלי הקורס).

יש להגיש קובץ zip הנקרא ex10.zip המכיל בדיוק את שני הקבצים הנ"ל.

בהצלחה!