תכנות מתקדם 2: 89-211 – מועד א' תשע"ז

זמן המבחן: **שעתיים וחצי**, יש לענות על 4 מתוך 4 שאלות, **בגוף השאלון בלבד**. חומר **סגור**.

שתי שאלות בקיאות + שתי שאלות עיצוב קוד ותכנות (java).

בקיאות

שאלה1 (24 נק'): מנגנון לסיווג (קלסיפיקציה) של טקסטים פועל באופן הבא. בשלב הלמידה אנו סורקים ה-מ-ו-ן טקסטים שסווגו מראש, ובודקים מהן המילים הנפוצות ביותר השייכות לסיווג ה i ושאינן נפוצות בשאר הסיווגים. כך לכל סיווג משויכת רשימה של מילים שמאפיינת דווקא אותו. למשל כל המילים הנפוצות ביותר שגברים כותבים אך נשים כמעט ולא משתמשות בהן. רשימה כזו נקראת features.

בהינתן טקסט חדש, אנו עוברים על רשימת המילים של כל סיווג ומודדים את שכיחותן של מילים אלה בטקסט החדש. נסווג את הטקסט החדש על פי רשימת המילים שזכתה לשכיחות הגבוהה ביותר.

מידי פעם נרצה לרענן את תהליך הלמידה ולעדכן את features שלמדנו.

לצורך שמירת המילים הנפוצות ביותר עבור כל סיווג, הגדרנו מחלקה בשם Classification המכילה רשימה של מילים. לאחר שלב הלמידה יש לאכסן את המופעים של Classification בדיסק. עבור כל אחת מהשיטות הבאות נמקו בקצרה את היתרונות והחסרונות של כל שיטה (3 נק' ליתרונות, 3 נק' לחסרונות).

א. פשוט נשמור אובייקט מסוג List<Classification> בדיסק (זה הרי serializable ולכן זה אפשרי)

יתרונות:

קריאה וכתיבה פשוטה מה\לדיסק.

קל לעדכן את ה features היות ולאחר הקריאה הכל מוחזק בזיכרון (RAM).

חסרונות:

חייבים לקרוא את *כל* האובייקטים בבת אחת מהדיסק ולהחזיק את כולם בזיכרון.

לא ניתן לשמור רק אחד מהם כאשר יש שיוניים, אלא את כולם. לא נוכל לעשות זאת כל הזמן, ואם תהיה נניח הפסקת חשמל נאבד את המידע ששינינו.

גישה ב O(n) לאובייקט הרצוי.

מכיוון שמדובר בקבצים, לא ניתן לבצע בקלות שאילתות מורכבות. יש לתכנת אותן.

ב. כל אובייקט Classification יישמר בקובץ אחר

יתרונות:

גישה ב O(1) לאובייקט הרצוי. ניתן בעת הצורך לטעון ולשמור רק אובייקט אחד בכל פעם (commit).

חסרונות:

מכיוון שמדובר בקבצים, לא ניתן לבצע בקלות שאילתות מורכבות. יש לתכנת אותן.

ג. נשתמש במסד נתונים. נמפה כל אובייקט Classification לשורה בטבלה. לטבלה יהיו N עמודות המשמשות לאכסון ה features של כל סיווג. שיטת schema on write.

יתרונות:

מסד הנתונים מתנהל מול הקבצים במקומנו, ומאפשר לנו לבצע שאילתות מורכבות בקלות.

באמצעות המיפוי אנו עדיין עובדים עם אובייקטים.

חסרונות:

בעיית סקאלביליות, מאד קשה לשנות את הסכמה לאחר שהתחלנו לעבוד איתה. בפרט, יהיה קשה להגדיל את מספר העמודות כאשר מספר ה features יגדל.

מכיוון שמדובר באוסף גדול של features, החל מנקודה מסוימת הפיצול ידרוש מחשב נוסף ויהיה קשה לחלק את הטבלאות מחדש בין המחשבים השונים.

שינוי בטבלאות יגרור גם שינוי בקוד המיפוי, וצורך בשמירה ובטעינה מחדש של כל מסד הנתונים בפוטנציה.

בנוסף, הגדרת הסכמה בעת הכתיבה גורם לכך שלרוב רק האפליקציה שלנו יכולה לעבוד מול מסד הנתונים ואין גמישות עבור אפליקציות אחרות שרוצות את המידע בצורה קצת שונה או לתעדף אותו אחרת. יש צורך לחשוב מראש איך ניתן לספק את כולם, ובפועל אף אחד לא ייצא מרוצה.

ד. נשתמש במסד נתונים בסגנון no SQL. שיטת schema on read.

יתרונות:

בשיטת הגדרת הסכמה בזמן הקריאה אנו נמנעים מהחיסרון לעיל, שכן כל אחד יכול לקרוא את הנתונים בדרכו שלו. יקרא את המידע כפי שהוא, ובאמצעות קוד ישלוף את מה שהכי רלוונטי לגביו.

חסרונות:

שיטה זו יקרה מאד מבחינת משאבי המערכת. יש צורך בחישובים כל הזמן.

ה data אינו מסביר את עצמו (לא ניתן למשל להסתכל על סכמה כלשהי כדי להבין כיצד ה data מורכב).

לכן, גם כתיבת השאילתות וגם הביצוע שלהם נורא איטי.

בנוסף, מכיוון שה data אינו structured, כל הזמן אנו מתמודדים עם האתגר של מידע חסר \ כפול \ לא תקין וכו'.

שאלה 2: (12 נק')

הקיפו בעיגול את התשובות הנכונות:

א. פתרון memcacheD מספק סקלאבליות ליניארית.
ב. סביבת Android Runtime חוסכת ב RAM ביחס לגרסאות קודמות.
ג. Service Provider רושם את עצמו אצל ה broker באמצעות פרוטוקול SOAP
ד. ל REST יש תמיכה בשליחת מידע רק בפורמט XML

שאלה 3 (31 נק'):

הביטו ב main הבא, המגדיר את המטרות שעליכם להשיג.

```
BlockingQueue<Point> result;
// define the stream
Stream<Point> s=new Stream<>();
result = s.filter(p->p.x>=0).filter(p->p.y<=0).getBuffer();
// the stream is still empty.

// printing thread
final boolean[] stop={false};
new Thread(()->{
      try {
            while(!stop[0])
                  while(!result.isEmpty())
                        System.out.println(result.take());
      } catch (InterruptedException e) {}
}).start();

// a demo of a slow stream-generation
Random r=new Random();
for(int i=0;i<500;i++){
      s.push(new Point(-100+r.nextInt(201),-100+r.nextInt(201)));
      Thread.sleep(50);
}
// stopping the stream(s)
s.endOfStream();

// stopping the printing thread
stop[0]=true;

// result: as the new points are generated,
//         only points with x>=0 & y<=0 are printed
```

ב main לעיל אנו מייצרים מופע של Stream<Point> המאפשר ארכיטקטורת pipes and filters ו fluent programming. באמצעות ביטוי למדה המתודה filter מאפשרת להעביר הלאה את כל הנקודות עם x לא שלילי, ומאלה להשאיר רק את הנקודות עם y לא חיובי. התוצאה תישמר ב result. אולם, בינתיים לכאורה לא קורה דבר, שכן ה stream ריק ממידע.

כעת אנו מגדירים ת'רד אנונימי שפשוט מדפיס את התוכן של result, ככל שיתקבלו לתוכו אובייקטים. הוא חי ברקע.

לאחר מכן אנו מייצרים 500 נקודות אקראיות עם ערכי x,y בין 100- ל 100, ומכנסים אותן ל stream. תוך כדי הכנסתן (ולא רק לאחר שמסתיים הקלט) הן יעברו סינון בהתאם להגדרות לעיל; "השורדים" יכנסו ל result, ויודפסו ע"י ת'רד שהגדרנו.

הפקודה endOfStream מורה על סיום הקלט הנכנס ל stream וכל משאב שצרכנו ישוחרר.

עליכם להשלים את הקוד של המחלקה Stream<T> כך שנוכל להפעיל את המתודות filter ו endOfStream בהצלחה ולקבל את התוצאה הרצויה להפעלה דומה לזו שב main לעיל.

תשובה:

```
public static class Stream<T>{

                                        בהינתן איזשהו E, החזר אמת או שקר
      public interface Predicate<E>{
            boolean test(E e);  // 5 points
      }

      BlockingQueue<T> buffer;
      volatile boolean stop;
      Thread myThread; // 2 points- take() כדי שנוכל לעצור אותו גם אם יהיה תקוע על ה
      Stream<T> next;  // 2 points- הבא Stream כדי שנוכל להורות על עצירת ה
                        שימו לב ש 4 הנקודות הללו הן למעשה חלק מהסעיף של עצירת התהליכים.

      public Stream() {
            buffer=new LinkedBlockingQueue<T>();
            stop=false;
      }


      public void push(T t){
            buffer.add(t);
      }
        אחת הטעויות הנפוצות היו עבודה על אותו ה buffer או שימוש ב remove מה שפגע במהות השאלה
      public Stream<T> filter(Predicate<T> p){      // total of 16 points
            next=new Stream<T>();
            myThread = new Thread(()->{      // 5 points
                  while(!stop){
                        try {
                              T item = buffer.take();
                              if(p.test(item))    // 5 points
                                    next.buffer.add(item);
                        } catch (InterruptedException e) {}
                  }
            });
            myThread.start();
            return next; // 5 points
      }

      public BlockingQueue<T> getBuffer(){
            return buffer;
      }
  לרוב היו כאן טעויות נגררות (סימנתי היכן שזה קיים). צריך לעצור גם את הת'רד הנוכחי וגם את ה stream הבא.
      public void endOfStream() { // 6 points
            stop=true;
            if(myThread!=null)
                  myThread.interrupt();
            if(next!=null)
                  next.endOfStream();
      }
}
```

שימו לב שלשאלה זו לא צריך להשתמש ב java8 אלא לממש בכלים המוכרים של threads ותכנות מוכוון עצמים את הארכיטקטורה של pipes & filters עליה למדנו לעומק בכיתה.

שאלה 4 (33 נק'):

ברצוננו ליצור תשתית מחלקות עבור מכירה פומבית. במכירה פומבית יש שני סוגים של שחקנים – המוכר Auctioneer, והקונה Bidder. כל הקונים מכירים את המוכר. המכירה מתחילה מאיזשהו מחיר התחלתי וכל קונה במקביל מעלה הצעות מחיר ע"פ מדיניות כלשהי משלו. המוכר מכריז לכל הקונים על המחיר העדכני, והם ממשיכים להציע מחירים חדשים. לאחר זמן מה המכירה מסתיימת והקונה שהציע את המחיר הגדול ביותר זוכה.

להלן דוגמת קוד להפעלת התשתית שברצוננו ליצור:

```
Auctioneer a=new Auctioneer();
Bidder b1=new Bidder(a, "b1", (x)->x+10);
Bidder b2=new Bidder(a, "b2", (x)->(x<=90 ? x+10 : 100));
Bidder b3=new Bidder(a, "b3", (x)->x+5);

a.startAuction(50);

Thread.sleep(50); // after some time

Bidder winner = a.endAuction();
System.out.println(winner.name+" "+ winner.currentBid); // b1
```

- יצרנו מופע של Auctioneer
- b1 הוא Bidder שהמדיניות שלו היא בהינתן המחיר x העלה את המחיר ל x+10
  - הגדרנו פונקציה מאד פשוטה, היא תעלה את המחיר ללא הגבלה, גם אם b1 הוא זה שקבע את המחיר הקודם...
- b2 נוהג באופן דומה עד לתקרה של 100, ואילו b3 תמיד יעלה ב 5.
- התחלנו את המכירה במחיר התחלתי של 50.
- לאחר זמן מה עצרנו את המכירה וקבלנו את ה Bidder שזכה.
- ע"פ המדיניות שהזרקנו b1 כמעט תמיד יוצא מנצח.

כדי לממש תשתית זו יש צורך בשתי תבניות עיצוב חשובות. עליכם להשלים את הקוד בטופס המבחן במקומות המתאימים בהתאם לתבניות העיצוב, הקוד והדרישות לעיל.

```
public class Auctioneer extends Observable{ // 1 points

	private double currentPrice;
	private Bidder currentBidder;
	private volatile boolean stop;   // 2 points

	// registers a bidder to an auction
	public void registerTheAuction(Observer bidder){  // 2 points
		addObserver(bidder);
	}
```

עד כאן לרוב לא היו טעויות.

מכאן היו טעויות מגוונות. המתודה הבאה צריכה להיות מסונכרנת, הרי כל הקונים ינסו להפעיל אותה במקביל אצל אותו המוכר. טעויות נפוצות:

- לשכוח מ stop
- While(!stop) במקום IF (כ 3 נק'). הרי acceptBid לא אמורה לפעול כל הזמן. הקונים אמורים להפעיל אותה כאשר הם רוצים בכך.
- להכניס כאן את הת'רד (כ 4 נק'). המשאב המשותף של כל הקונים הוא המוכר. לכן הוא זה שפועל בצורה מסונכרנת ואילו הקונים הם אלו שפועלים במקביל על המוכר. אם המוכר היה פותח ת'רד לכל אחד מהקונים אז הם פועלים במקביל על אותו ה data שלו. הסנכרון לא היה עוזר במקרה זה כי בתוך ה synchronized נפתח ת'רד לכל קונה...
- כל הזמן לעשות notifyObservers, גם כשהמחיר לא משתנה (כ 2 נק').
- לא לשלוח פרמטר ל notify ומאוחר יותר להניח שקיים (כנקודה).

```
	// accepts a bid by some bidder
	public synchronized void acceptBid(Bidder b,double price){
	// total of 8 points (synchronized is 2)
		if(!stop && price>currentPrice){  // 2 points
			currentPrice=price;
			currentBidder=b;
			// 4 points:
			setChanged();
			notifyObservers(price);
		}
	}
```

היו כאן כמה ששכחו לשלוח נוטיפיקציה לכל הקונים. הרי זה הטריגר לכל הפעולות 2 נק'

```
	public void startAuction(double initialPrice){ // 2 points
		currentPrice=initialPrice;
		setChanged();
		notifyObservers(currentPrice);
	}

	public Bidder endAuction(){
		stop=true;
		return currentBidder;
	}
}
```

כאן וויתרתי למי שרשם int. אבל כן דרשתי ערך קבלה וערך חזרה, ושיהיו מאותו הסוג. היו כמה ששכחו שכדי לכתוב ביטוי למדה צריך פשוט לכתוב ממשק בעל מתודה אחת המצריכה מימוש.

```
public static interface Policy{ // 3 points
	double bid(double price);
}

public static class Bidder implements Observer{ // 1 points
	Policy policy; // 2 points
	double currentBid;
	Auctioneer auctioneer;
	String name;
```

טעות נפוצה: לשכוח לבצע את הרישום כ observer. הרי ב main רואים שזה לא נעשה, ולכן יש לדאוג לכך פנימית. כאן זה המקום הכי הגיוני לכך. קבלתי גם תשובות שבצעו זאת במקומות אחרים כל עוד זה קורה פעם אחת.

```
public Bidder(Auctioneer auctioneer,String name,Policy policy) {
        // total of 5 points
        this.policy=policy;
        this.auctioneer=auctioneer;
        auctioneer.registerTheAuction(this); // 2 points
        this.name=name;
}
```

טעויות נפוצות: לא להשתמש בת'רד, לא לחלץ נכון את המחיר (יש לא מעט אופציות נכונות לחילוץ). או לא להשתמש ב acceptBid.

```
@Override
public void update(Observable arg0, Object arg1) { // total of 7 points
        new Thread(()->{ // using a thread correctly 3 points
                // using the policy 2 points
                double bid=policy.bid((Double)arg1);
                if(bid!=currentBid){
                        currentBid=bid;
                        auctioneer.acceptBid(this, currentBid); // 2 points
                }
        }).start();
}
}
```

מדיניות ערעורים:

כפי שאתם רואים סיפקתי לכם פתרון מפורט + מפתח בדיקה. הכל שקוף. לכן, אני גם מצפה מכם להוגנות דומה ולא לערער סתם כדי לחלוב עוד נקודות. אנא תערערו **רק אם היתה טעות בבדיקה**, ולא על שיקול הדעת בחלוקת הנקודות או כל דבר אחר שהוא אינו טעות בבדיקת השאלה. אף אחד לא רוצה לבדוק כל מבחן פעמיים.

כמו כן, בוודאי שמתם לב שאת השאלה הראשונה בדקתי בצורה **מ-א-ו-ד** מקלה. הדיל הוא כזה: אני שומר לעצמי את הזכות בעת ערעור לבדוק את כל המבחן מחדש ובפרט את שאלה 1 בצורה הוגנת לפי התשובה שפרסמתי. אז אנא, לפני שתערערו על נקודה אחת פה ושם, פשוט בדקו אם הדבר אכן משתלם.