# דו"ח מסכם תרגיל בית 4 – מבוא ללמידה ממוכנת:


## מגישים: איתי גיא, ת.ז. – 305104184

## רועי נתן, ת.ז. - 203842885


א. קריאת קובץ הנתונים $iris.data$ – בוצע ע"י קריאת הקובץ שורה אחרי שורה והכנסת כל שורה לאובייקט שנקרא $FeatureVector$ שמתאר כל תצפית בנתונים אלו – פיצ'רים של עלי הכותרת והתיוג של השורה (המחרוזות של המחלקה שאליה היא מתאימה)

```
/**
 * Handling FeatureVector operations
 * @author Itay_Guy
 */
public class FeatureVector {
    private List<Double> _feature_vector; // numeric dat
    private String _name; //data labeling

    public FeatureVector() {
        this._feature_vector = new ArrayList<Double>();
        this._name = new String();
    }

    public FeatureVector(String[] features) {
        this();
        for (String f : features) {
            try {
                double d = Double.parseDouble(f);
                this._feature_vector.add(d);
            }catch(Exception ex) {
                _name = new String(f.trim());
            }
        }
    }
```

```
Training-set size = 120 , Test-set size = 30
```

ב. קידוד פונקציית הפיצול שמכינה את הנתונים בקובץ לאימון מודל – תוספת הפרמטרים הם בכדי למחזר את הפונקציה גם לטובת המחלקה של $DT$ – באופן כללי כל הטיפול בנתונים מתבצע במחלקה בשם $Dataset$.

```java
    /**
     * prepare data to the model usage
     * @param split_perc    training-set size percentage to split
     * @param uniform   selection distribution type - if false there is random type
     * @param headers   usage for DT-ID3 only and it says if there are headers to this data file
     * @throws IOException
     */
    public void prepare_data(int split_perc,boolean uniform,boolean headers) throws IOException {
        List<String> lines = Files.readAllLines(this._data_name); // taking data from Path file you provided in the constructor
        // handling headers:
        if(headers) {
            this._headers = new ArrayList<String>();
            String[] variables = lines.get(0).split(",");
            int i = 0;
            for (String var : variables) {
                this._headers.add(new String(var.trim()));
                i++;
            }
            lines.remove(0); // remove the labeling from headers array
        }
        int lines_amount = lines.size();
        int set_perc = (split_perc*lines.size())/100;
        List<Integer> seq = new ArrayList<Integer>();
        for (int i = 0; i < lines_amount; i++) {
            seq.add(i);
        }
        List<Integer> permut = this.make_permut(seq, lines_amount); // make randomized data lines
        if(!uniform) {
            // this is not uniform distribution -> take the first split_perc data lines to training set and the remain to test set
            for (int i = 0; i < lines_amount; i++) {
                String[] data_line = lines.get(permut.get(i)).split(",");
                if(i >= set_perc) {
                    this._test_set.add(new FeatureVector(data_line));
                } else {
                    this._train_set.add(new FeatureVector(data_line));
                }
            }
        } else {
            // this is distributed uniformly -> handling the issue there is same amounts in the training set:
            Map<String,Integer> types = new HashMap<String,Integer>();
```

```java
        for (String line : lines) {
            String[] features = line.split(",");
            String name = features[features.length - 1].trim();
            if(types.containsKey(name)) {
                types.put(name,types.get(name).intValue()+1);
            } else {
                types.put(name,1);
            }
        }
        int uniform_chunk = lines_amount/types.keySet().size();
        for (String key : types.keySet()) {
            if(types.get(key).intValue() > uniform_chunk) {
                types.put(key,types.get(key).intValue()-(types.get(key).intValue()-uniform_chunk));
            }
        }
        boolean[] bucket = new boolean[lines_amount];
        for (int i = 0;i < bucket.length;i++) {
            bucket[i] = false;
        }
        for (int i = 0;i < lines_amount;i++) {
            String[] features = lines.get(permut.get(i)).split(",");
            String line_name = features[features.length - 1].trim();
            if(this._train_set.size() < set_perc){
                if(types.get(line_name).intValue() > 0) {
                    types.put(line_name,types.get(line_name).intValue() - 1);
                    this._train_set.add(new FeatureVector(features));
                    bucket[permut.get(i)] = true;
                }
            } else {
                break;
            }
        }
        for (int i = 0;i < bucket.length;i++) {
            if(bucket[i] == false) {
                this._test_set.add(new FeatureVector(lines.get(i).split(",")));
            }
        }
    }
    this.prepare_class_mapping(this._train_set,this._test_set); // build class tag mapping
    System.out.println("Training-set size = " + this._train_set.size() + " , Test-set size = " + this._test_set.size());
}
```

ג. פונקציית המרחק בין אירוסים נכתבה במחלקה $FeatureVector$ כאשר היא ממשת באופן כללי את פונקציית מינקובסקי כאשר $p = 2$ זה מרחק אוקלידי לכן ניתן לראות אותו בפעולה עבור $p$-ים שונים שזה בעצם עבור סוגים שונים של פונקציות מרחק מאותה המשפחה.

```java
    /**
     * Minkovski distance
     * @param other FeatureVector to comparison with
     * @param p if p=1 -> (Manhattan distance), if p=2 -> (euclidian distance),...
     * @return  the distance number
     */
    public double distance_p(FeatureVector other,int p) {
        double sum = 0.0;
        for (int i = 0; i < this._feature_vector.size(); i++) {
            sum += Math.pow(Math.abs(this.getFeature(i) - other.getFeature(i)),p);
        }
        return Math.pow(sum,1.0/p);
    }
```

ד. פונקציית סיווג שמחזירה את $k$ השכנים הקרובים ביותר לפרח המתקבל כפרמטר נכתבה במחלקה $KNN$ שמממשת את המודל.

```
/**
 * pick k neighbors
 * @param data  data-set object
 * @param f sample from the test-set
 * @param k for picking the k-neighbors
 * @return  k neighbors into array
 */
public List<FeatureVector> make_subset(Dataset data,FeatureVector f,int k) {
    List<FeatureVector> train_set = data.getTrainingset();
    Collections.sort(train_set,new Comparator<FeatureVector>() { // sorting bottom up
        @Override
        public int compare(FeatureVector o1, FeatureVector o2) {
            double x = f.distance_p(o1,P);
            double y = f.distance_p(o2,P);
            if(x < y) return -1;
            else if(x > y) return 1;
            else return 0;
        }
    });
    List<FeatureVector> k_neighbors = new ArrayList<FeatureVector>();
    for (int i = 0; i < k;i++) {
        k_neighbors.add(new FeatureVector(train_set.get(i)));
    }
    return k_neighbors;
}
```

<u>למשל עבור $k = 3$ נקבל:</u>

```
For Test Example: Feature vector = [6.7,3.0,5.2,2.3,Label:Iris-virginica]
        Neighbor 0: Feature vector = [6.9,3.1,5.1,2.3,Label:Iris-virginica]
        Neighbor 1: Feature vector = [6.5,3.0,5.2,2.0,Label:Iris-virginica]
        Neighbor 2: Feature vector = [6.9,3.1,5.4,2.1,Label:Iris-virginica]
```

קביעת סוג האירוס עפ"י המחלקה של הרוב מתוך $k$ השכנים שהתקבלו:

```java
/**
 * finding the class that the sample from test-set is belong to
 * @param best_k    array of best k neighbor from "make_subset"
 * @param weighted  true if need weighted technique and false otherwise
 * @return  classified class for the test-set's sample
 */
public String choose_best(List<FeatureVector> best_k,boolean weighted) { // best_k is already sorted
    int k = best_k.size();
    Map<String, HashMap<String,Double>> freq = new HashMap<String,HashMap<String,Double>>();
    for (int i = 0; i < k; i++) {
        double eps = 0.0;
        if(weighted) {
            eps = 1.0*k-i; // weight for each neighbor by their distance from f
        }

        HashMap<String,Double> h = new HashMap<String,Double>();
        h.put("weight",1.0 + eps);
        freq.put(best_k.get(i).getName() + "@" + i, h);
    }
    Map<String,Double> counter = new HashMap<String,Double>();
    for (String key : freq.keySet()) {
        HashMap<String,Double> h = freq.get(key);
        double val = h.get("weight").doubleValue();
        key = key.split("@")[0];
        if(!counter.containsKey(key)) {
            counter.put(key,val);
        } else {
            counter.put(key,counter.get(key) + val);
        }
    }
    String max_label = "";
    double max_appearances = 0.0;
    for (String key : counter.keySet()) {
        double val = counter.get(key).doubleValue();
        if(val > max_appearances) {
            max_appearances = val;
            max_label = key;
        }
    }
    return  max_label;
}
```

- פרמטר $weighted$ נכנס בכדי להשתמש בקוד עבור הדרישה המתקדמת בתרגיל.

ה. <u>פונקציה הקובעת את אחוז הדיוק של קבוצת הבדיקה – הפונקציה כתובה במחלקה *KNN* שמממשת את המודל.</u>

```
/**
 * classification for same data examples
 * @param data  Data-set
 * @param k number of neighbors
 * @param weighted  need to use weighted method or not
 * @param need_confution_matrix_disp    if we would like to display the confusion matrix
 * @param validate_training_set true if we need to classify the training-set
 * @return  match accuracy of this model
 */
public double classify_by_accuracy(Dataset data,int k,boolean weighted,boolean need_confution_matrix_disp,boolean validate_training_set) {
    Map<String,HashMap<String,Integer>> confusion_matrix = this.classify(data, k, weighted,validate_training_set);
    if(need_confution_matrix_disp) {
        System.out.println(KNN.disp_confusion_matrix(confusion_matrix));
    }
    int total = 0;
    int correct = 0;
    //counting how many is correct predictions:
    for (String key : confusion_matrix.keySet()) {
        for(String name : confusion_matrix.get(key).keySet()){
            if(key.equals(name)) {
                correct += confusion_matrix.get(key).get(name);
            }
            total += confusion_matrix.get(key).get(name);
        }
    }
    if(!need_confution_matrix_disp) {
        System.out.println("******");
    }
    double accuracy = correct*100.0/total;
    System.out.println("Accuracy = " + accuracy);
    return accuracy;
}
```

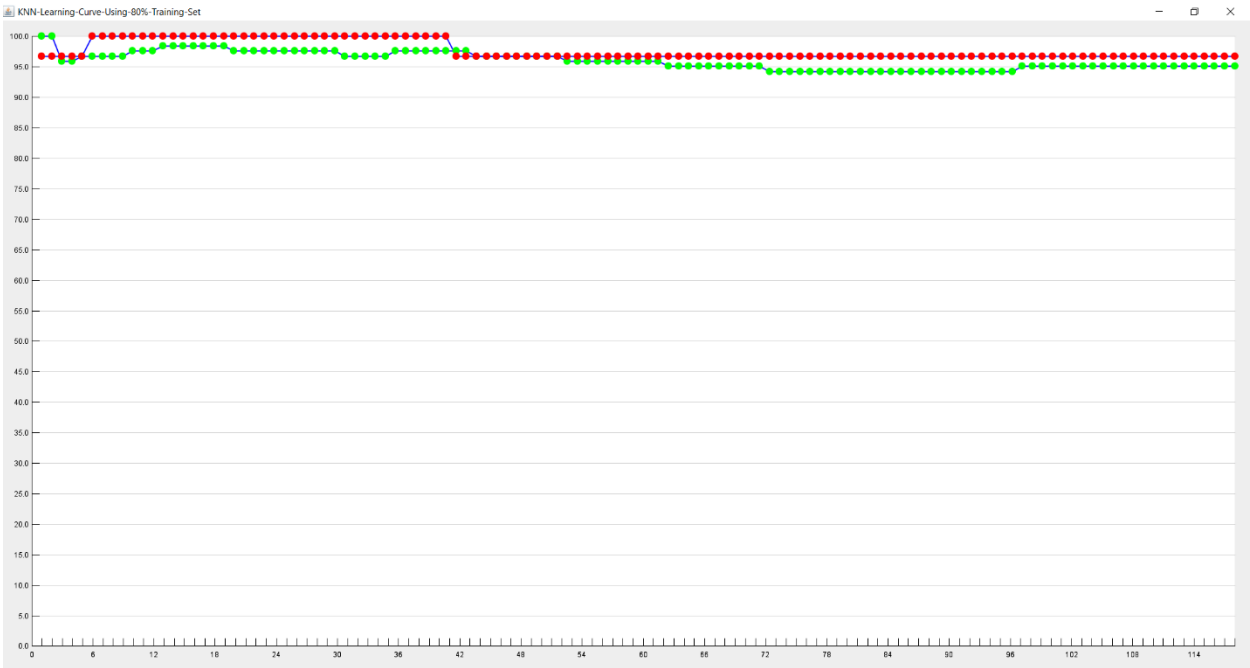
ו. שילוב כל הפונקציה לתוך מסווג.

```java
/**
 * making classification for some data - training-set or test-set
 * @param data  Data-set
 * @param k number of neighbors
 * @param weighted  need to use weighted method for finding the class between k neighbors
 * @param validate_training_set if need to classify the training-set for learning curve for example
 * @return  confusion matrix
 */
public Map<String,HashMap<String,Integer>> classify(Dataset data,int k,boolean weighted,boolean validate_training_set) {
    List<FeatureVector> test = null;
    if(validate_training_set) {
        System.out.println("******\nClassification: Training-Set");
        test = data.getTrainingset();
    } else {
        System.out.println("******\nClassification: Test-Set");
        test = data.getTestSet();
    }
    Map<String,HashMap<String,Integer>> confusion_matrix = new HashMap<String,HashMap<String,Integer>>();
    System.out.println("\nUsing k = " + k + " and Weighted-Sum = " + weighted + "\n******");
    for (FeatureVector f_test : test) {
        System.out.println("For Test Example: " + f_test);
        List<FeatureVector> best_k = this.make_subset(data,f_test, k);
        int i = 0;
        for (FeatureVector fv : best_k) {
            System.out.println("\tNeighbor " + i + ": " + fv);
            i++;
        }
        String res = this.choose_best(best_k, weighted); // weighted = true of false
        if(res.equals(f_test.getName())) {
            System.out.println("\t+ " + res + " Classified correctly");
        } else {
            System.out.println("\t- Truth value: " + f_test.getName() + " => but classified as: " + res);
        }
        // build confusion matrix
        HashMap<String,Integer> h = new HashMap<String,Integer>();
        if(confusion_matrix.keySet().contains(f_test.getName())){
            if(confusion_matrix.get(f_test.getName()).keySet().contains(res)) {
                confusion_matrix.get(f_test.getName()).put(res,confusion_matrix.get(f_test.getName()).get(res) + 1);
            } else {
                h.put(res,1);

            confusion_matrix.put(f_test.getName(),h);
        }
    }
    ArrayList<String> titles = new ArrayList<String>();
    for (String name : confusion_matrix.keySet()) {
        titles.add(name);
    }
    for (String name : confusion_matrix.keySet()) {
        for (int i = 0; i < titles.size(); i++) {
            if(!confusion_matrix.get(name).containsKey(titles.get(i))) {
                confusion_matrix.get(name).put(titles.get(i),0);
            }
        }
    }
    return confusion_matrix;
}
```

ז. ניתן לראות בגרף שצויר למטה את ההשוואה בין ה-$training - set$ ל-$test - set$ כאשר ב-$test - set$ (**העקום האדום**) גילינו שה-$k$ האופטימלי מתקבל בטווח בין $k = 6$ ל-$k = 42$ עם דיוק של $100\%$ ו-$train$ $80\%$ (אם צריך לתת $k$ מסוים אז נבחר את $k = 7$).

ח. בנוסף, גילינו ע"י ציור 2 העקומים למטה שבאזור של $k = 5$ מתקבלת התאמה של הפונקציה. כאשר לפני כאן ב-$k$ נמוך נראה אזור של $underfitting$ מכיוון שאין דיוק של ה-$test - set$ כמו של ה-$training - set$ ובאזור שאחרי עד $k = 42$ מתקבל $overfitting$ מכיוון שדיוק ה-$training - set$ עולה על הדיוק של ה-$test - set$. בהמשך הגרף החל מהאזור של $k = 42$ נראה שהאלגוריתם מתייצב בעיקר באזור שבין $k = 42$ עד $k = 60$ (בערך) ומדייק באופן דומה ל-$training - set$.



דגימה של קבוצת הבדיקה – הדפסת תוצאת הסיווג לכל אירוס, הסיווג הנכון והאם הוא היה נכון וכן בסוף את הדיוק הכללי של המסווג:

```
Distance Method: Minkovski Distance with P = 2
Training-set size = 120 , Test-set size = 30
Using k = 3 and Weighted-Sum = false
******
For Test Example: Feature vector = [6.2,3.4,5.4,2.3,Label:Iris-virginica]
        Neighbor 0: Feature vector = [6.3,3.4,5.6,2.4,Label:Iris-virginica]
        Neighbor 1: Feature vector = [6.4,3.2,5.3,2.3,Label:Iris-virginica]
        Neighbor 2: Feature vector = [6.5,3.2,5.1,2.0,Label:Iris-virginica]
        + Iris-virginica Classified correctly
For Test Example: Feature vector = [5.2,4.1,1.5,0.1,Label:Iris-setosa]
        Neighbor 0: Feature vector = [5.5,4.2,1.4,0.2,Label:Iris-setosa]
        Neighbor 1: Feature vector = [5.1,3.8,1.5,0.3,Label:Iris-setosa]
        Neighbor 2: Feature vector = [5.4,3.7,1.5,0.2,Label:Iris-setosa]
        + Iris-setosa Classified correctly
For Test Example: Feature vector = [5.1,3.8,1.6,0.2,Label:Iris-setosa]
        Neighbor 0: Feature vector = [5.1,3.8,1.5,0.3,Label:Iris-setosa]
        Neighbor 1: Feature vector = [5.1,3.7,1.5,0.4,Label:Iris-setosa]
        Neighbor 2: Feature vector = [5.0,3.6,1.4,0.2,Label:Iris-setosa]
        + Iris-setosa Classified correctly
For Test Example: Feature vector = [4.9,3.1,1.5,0.1,Label:Iris-setosa]
        Neighbor 0: Feature vector = [4.9,3.1,1.5,0.1,Label:Iris-setosa]
        Neighbor 1: Feature vector = [4.9,3.1,1.5,0.1,Label:Iris-setosa]
        Neighbor 2: Feature vector = [4.9,3.0,1.4,0.2,Label:Iris-setosa]
        + Iris-setosa Classified correctly
For Test Example: Feature vector = [4.6,3.1,1.5,0.2,Label:Iris-setosa]
        Neighbor 0: Feature vector = [4.6,3.2,1.4,0.2,Label:Iris-setosa]
        Neighbor 1: Feature vector = [4.7,3.2,1.6,0.2,Label:Iris-setosa]
        Neighbor 2: Feature vector = [4.8,3.1,1.6,0.2,Label:Iris-setosa]
        + Iris-setosa Classified correctly


...

For Test Example: Feature vector = [4.9,2.5,4.5,1.7,Label:Iris-virginica]
        Neighbor 0: Feature vector = [5.4,3.0,4.5,1.5,Label:Iris-versicolor]
        Neighbor 1: Feature vector = [5.5,2.6,4.4,1.2,Label:Iris-versicolor]
        Neighbor 2: Feature vector = [5.6,2.7,4.2,1.3,Label:Iris-versicolor]
        - Truth value: Iris-virginica => but classified as: Iris-versicolor
For Test Example: Feature vector = [7.3,2.9,6.3,1.8,Label:Iris-virginica]
        Neighbor 0: Feature vector = [7.4,2.8,6.1,1.9,Label:Iris-virginica]
        Neighbor 1: Feature vector = [7.1,3.0,5.9,2.1,Label:Iris-virginica]
        Neighbor 2: Feature vector = [7.7,2.8,6.7,2.0,Label:Iris-virginica]
        + Iris-virginica Classified correctly
For Test Example: Feature vector = [4.6,3.6,1.0,0.2,Label:Iris-setosa]
        Neighbor 0: Feature vector = [4.6,3.4,1.4,0.3,Label:Iris-setosa]
        Neighbor 1: Feature vector = [5.0,3.5,1.3,0.3,Label:Iris-setosa]
        Neighbor 2: Feature vector = [4.4,3.2,1.3,0.2,Label:Iris-setosa]
        + Iris-setosa Classified correctly
For Test Example: Feature vector = [6.4,3.2,4.5,1.5,Label:Iris-versicolor]
        Neighbor 0: Feature vector = [6.3,3.3,4.7,1.6,Label:Iris-versicolor]
        Neighbor 1: Feature vector = [6.7,3.1,4.4,1.4,Label:Iris-versicolor]
        Neighbor 2: Feature vector = [6.7,3.1,4.7,1.5,Label:Iris-versicolor]
        + Iris-versicolor Classified correctly
******
Confusion Matrix:
                        Iris-versicolor         Iris-virginica          Iris-setosa
        Iris-versicolor         9                       0                       0
        Iris-virginica          1                       10                      0
        Iris-setosa             0                       0                       10
******
```

```
Iris-versicolor Precision = 100.0
Iris-virginica Precision = 90.9090909090909
Iris-setosa Precision = 100.0
******
Iris-versicolor Recall = 90.0
Iris-virginica Recall = 100.0
Iris-setosa Recall = 100.0
******
F-[Iris-versicolor]-measure = 94.73684210526316
F-[Iris-virginica]-measure = 94.73684210526316
F-[Iris-setosa]-measure = 100.0
******
Accuracy = 96.66666666666667
```

<u>דגימה של קבוצת האימון – הדפסת תוצאת הסיווג לכל אירוס, הסיווג הנכון והאם הוא היה נכון וכן בסוף את הדיוק הכללי של המסווג:</u>

```
Distance Method: Minkovski Distance with P = 2
Training-set size = 120 , Test-set size = 30
Classification: Training-Set

Using k = 3 and Weighted-Sum = false
******
For Test Example: Feature vector = [4.8,3.4,1.6,0.2,Label:Iris-setosa]
        Neighbor 0: Feature vector = [4.8,3.4,1.6,0.2,Label:Iris-setosa]
        Neighbor 1: Feature vector = [5.0,3.4,1.5,0.2,Label:Iris-setosa]
        Neighbor 2: Feature vector = [5.0,3.4,1.6,0.4,Label:Iris-setosa]
        + Iris-setosa Classified correctly
For Test Example: Feature vector = [7.3,2.9,6.3,1.8,Label:Iris-virginica]
        Neighbor 0: Feature vector = [7.3,2.9,6.3,1.8,Label:Iris-virginica]
        Neighbor 1: Feature vector = [7.4,2.8,6.1,1.9,Label:Iris-virginica]
        Neighbor 2: Feature vector = [7.2,3.2,6.0,1.8,Label:Iris-virginica]
        + Iris-virginica Classified correctly
For Test Example: Feature vector = [7.7,2.6,6.9,2.3,Label:Iris-virginica]
        Neighbor 0: Feature vector = [7.7,2.6,6.9,2.3,Label:Iris-virginica]
        Neighbor 1: Feature vector = [7.7,2.8,6.7,2.0,Label:Iris-virginica]
        Neighbor 2: Feature vector = [7.7,3.0,6.1,2.3,Label:Iris-virginica]
        + Iris-virginica Classified correctly
For Test Example: Feature vector = [7.7,3.0,6.1,2.3,Label:Iris-virginica]
        Neighbor 0: Feature vector = [7.7,3.0,6.1,2.3,Label:Iris-virginica]
        Neighbor 1: Feature vector = [7.4,2.8,6.1,1.9,Label:Iris-virginica]
        Neighbor 2: Feature vector = [7.3,2.9,6.3,1.8,Label:Iris-virginica]
```

...

```
For Test Example: Feature vector = [6.7,3.1,4.7,1.5,Label:Iris-versicolor]
        Neighbor 0: Feature vector = [6.7,3.1,4.7,1.5,Label:Iris-versicolor]
        Neighbor 1: Feature vector = [6.9,3.1,4.9,1.5,Label:Iris-versicolor]
        Neighbor 2: Feature vector = [6.7,3.1,4.4,1.4,Label:Iris-versicolor]
        + Iris-versicolor Classified correctly
For Test Example: Feature vector = [5.4,3.4,1.5,0.4,Label:Iris-setosa]
        Neighbor 0: Feature vector = [5.4,3.4,1.5,0.4,Label:Iris-setosa]
        Neighbor 1: Feature vector = [5.4,3.4,1.7,0.2,Label:Iris-setosa]
        Neighbor 2: Feature vector = [5.2,3.4,1.4,0.2,Label:Iris-setosa]
        + Iris-setosa Classified correctly
******
Confusion Matrix:
                        Iris-versicolor         Iris-virginica          Iris-setosa
        Iris-versicolor         39                      0                       0
        Iris-virginica          3                       36                      0
        Iris-setosa             0                       0                       42
******
```

```
Iris-versicolor Precision = 100.0
Iris-virginica Precision = 92.3076923076923
Iris-setosa Precision = 100.0
******
Iris-versicolor Recall = 92.85714285714286
Iris-virginica Recall = 100.0
Iris-setosa Recall = 100.0
******
F-[Iris-versicolor]-measure = 96.2962962962963
F-[Iris-virginica]-measure = 96.2962962962963
F-[Iris-setosa]-measure = 100.0
******
Accuracy = 97.5
```

.2

א. בפונקציית הדגימה ($prepare\_data$) יש 2 מצבים – בהתאם לפרמטר $boolean\ uniform$.

<u>הקוד לדגימה פרופורציונלית (**מקרה** בתוך הפונקציה $prepare\_data$):</u>

```
        } else {
            // this is distributed uniformly -> handling the issue there is same amounts in the training set:
            Map<String,Integer> types = new HashMap<String,Integer>();
            for (String line : lines) {
                String[] features = line.split(",");
                String name = features[features.length - 1].trim();
                if(types.containsKey(name)) {
                    types.put(name,types.get(name).intValue()+1);
                } else {
                    types.put(name,1);
                }
            }
            int uniform_chunk = lines_amount/types.keySet().size();
            for (String key : types.keySet()) {
                if(types.get(key).intValue() > uniform_chunk) {
                    types.put(key,types.get(key).intValue()-(types.get(key).intValue()-uniform_chunk));
                }
            }
            boolean[] bucket = new boolean[lines_amount];
            for (int i = 0;i < bucket.length;i++) {
                bucket[i] = false;
            }
            for (int i = 0;i < lines_amount;i++) {
                String[] features = lines.get(permut.get(i)).split(",");
                String line_name = features[features.length - 1].trim();
                if(this._train_set.size() < set_perc){
                    if(types.get(line_name).intValue() > 0) {
                        types.put(line_name,types.get(line_name).intValue() - 1);
                        this._train_set.add(new FeatureVector(features));
                        bucket[permut.get(i)] = true;
                    }
                } else {
                    break;
                }
            }
            for (int i = 0;i < bucket.length;i++) {
                if(bucket[i] == false) {
                    this._test_set.add(new FeatureVector(lines.get(i).split(",")));
                }
            }
        }
```

ב. בפונקציית הסיווג יינתן משקל גבוהה יותר לאירוסים קרובים במידה ופרמטר *boolean weighted* יהיה שווה ל-*true*.

```java
/**
 * finding the class that the sample from test-set is belong to
 * @param best_k    array of best k neighbor from "make_subset"
 * @param weighted  true if need weighted technique and false otherwise
 * @return  classified class for the test-set's sample
 */
public String choose_best(List<FeatureVector> best_k,boolean weighted) { // best_k is already sorted
    int k = best_k.size();
    Map<String, HashMap<String,Double>> freq = new HashMap<String,HashMap<String,Double>>();
    for (int i = 0; i < k; i++) {
        double eps = 0.0;
        if(weighted) {
            eps = 1.0*k-i; // weight for each neighbor by their distance from f
        }

        HashMap<String,Double> h = new HashMap<String,Double>();
        h.put("weight",1.0 + eps);
        freq.put(best_k.get(i).getName() + "@" + i, h);
    }
    Map<String,Double> counter = new HashMap<String,Double>();
    for (String key : freq.keySet()) {
        HashMap<String,Double> h = freq.get(key);
        double val = h.get("weight").doubleValue();
        key = key.split("@")[0];
        if(!counter.containsKey(key)) {
            counter.put(key,val);
        } else {
            counter.put(key,counter.get(key) + val);
        }
    }
    String max_label = "";
    double max_appearances = 0.0;
    for (String key : counter.keySet()) {
        double val = counter.get(key).doubleValue();
        if(val > max_appearances) {
            max_appearances = val;
            max_label = key;
        }
    }
    return  max_label;
}
```

ג. <u>ניתן לראות שע"י שימוש במרחק אוקלידי כפרמטר למרחק קיבלנו את התוצאות הבאות עבור $k = 3$:</u>

```
                    Iris-versicolor        Iris-virginica        Iris-setosa
        Iris-versicolor        11                    1                    0
        Iris-virginica         0                     8                    0
        Iris-setosa            0                     0                    10
******
Iris-versicolor Precision = 91.66666666666667
Iris-virginica Precision = 100.0
Iris-setosa Precision = 100.0
******
Iris-versicolor Recall = 100.0
Iris-virginica Recall = 88.88888888888889
Iris-setosa Recall = 100.0
******
F-[Iris-versicolor]-measure = 95.65217391304348
F-[Iris-virginica]-measure = 94.11764705882352
F-[Iris-setosa]-measure = 94.11764705882352
******
Accuracy = 96.66666666666667
```

<u>וע"י שימוש בפונקציית מינקובסקי אחרת - $p = 10$ קיבלנו את התוצאות הבאות עבור $k = 3$:</u>

```
                    Iris-versicolor        Iris-virginica        Iris-setosa
        Iris-versicolor        10                    0                    0
        Iris-virginica         0                     9                    0
        Iris-setosa            0                     0                    11
******
Iris-versicolor Precision = 100.0
Iris-virginica Precision = 100.0
Iris-setosa Precision = 100.0
******
Iris-versicolor Recall = 100.0
Iris-virginica Recall = 100.0
Iris-setosa Recall = 100.0
******
F-[Iris-versicolor]-measure = 100.0
F-[Iris-virginica]-measure = 100.0
F-[Iris-setosa]-measure = 100.0
******
Accuracy = 100.0
```

כלומר, אכן ניתן לראות את השפעת השינוי על התוצאות באופן משמעותי מאוד בין מינקובסקי $p = 2$ מול מינקובסקי $p = 10$ (במאמר מוסגר ניתן לומר שעלות חישוב של מינקובסקי עם $p = 10$ לווקטורים גדולים יותר עלולה להיות עלות יקרה בזמן חישוב מודל $KNN$ ולהאט משמעותית את זמן החישוב).

.3

א. מימוש עץ ההחלטה בוצע במחלקה בשם $DecisionTree$ שנעזרת במחלקות $Node, Tree$ ו-$FeatureVector$ בכדי לייצג את הנתונים שבקובץ $votes.csv$.

* בפונקציה $fit$ יש קריאה לפונקציית בנייה של העץ.

```
/**
 * build new DT model
 */
public void fit() {
    List<FeatureVector> train = this._data.getTrainingset();
    List<String> attributes = this._data.getAttributes(false); // false is for taking attribute without labels
    String defy = this.majority_value(train); // pickup the majority label in the training-set samples
    this._hypotheses_tree = this.build_decision_tree(train,attributes,defy);
}

private Tree build_decision_tree(List<FeatureVector> examples,List<String> attributes,String defy) {
    if(examples == null || examples.isEmpty()) {
        return new Tree(new Node(this._data.convert_class(defy),attributes));
    }else if(this.has_homogenous_class(examples) || attributes.isEmpty()) {
        return new Tree(new Node(this._data.convert_class(this.majority_value(examples)),attributes));
    } else {
        String best = this.choose_attribute(examples, attributes);
        Tree tree = new Tree(new Node(best,attributes));
        for(int vi : DecisionTree.BINARY_SET) { // build node options
            List<FeatureVector> examplesi = this.choose_subtrain(examples, vi, best);
            Tree subtree = this.build_decision_tree(examplesi,this.subtract_attr(attributes,best),this.majority_value(examples));
            tree.add_branch(subtree,vi,best); // add new sub-tree to this vi side in this node
        }
        return tree;
    }
}
```

ב. חלוקת הנתונים לקבוצת אימון וקבוצת בדיקה: שימוש בפונקציה שהוצגה הנ"ל בשם $prepare\_data$ כאשר הפרמטר $boolean\ header$ שווה ל-$true$ זה אומר שיש שמות לעמודות והן נאספות בפונקציה לאובייקט נפרד לצורך ביצוע מיפוי בין אינדקס לשם בהמשך.

```
Training-set size = 279 , Test-set size = 156
```

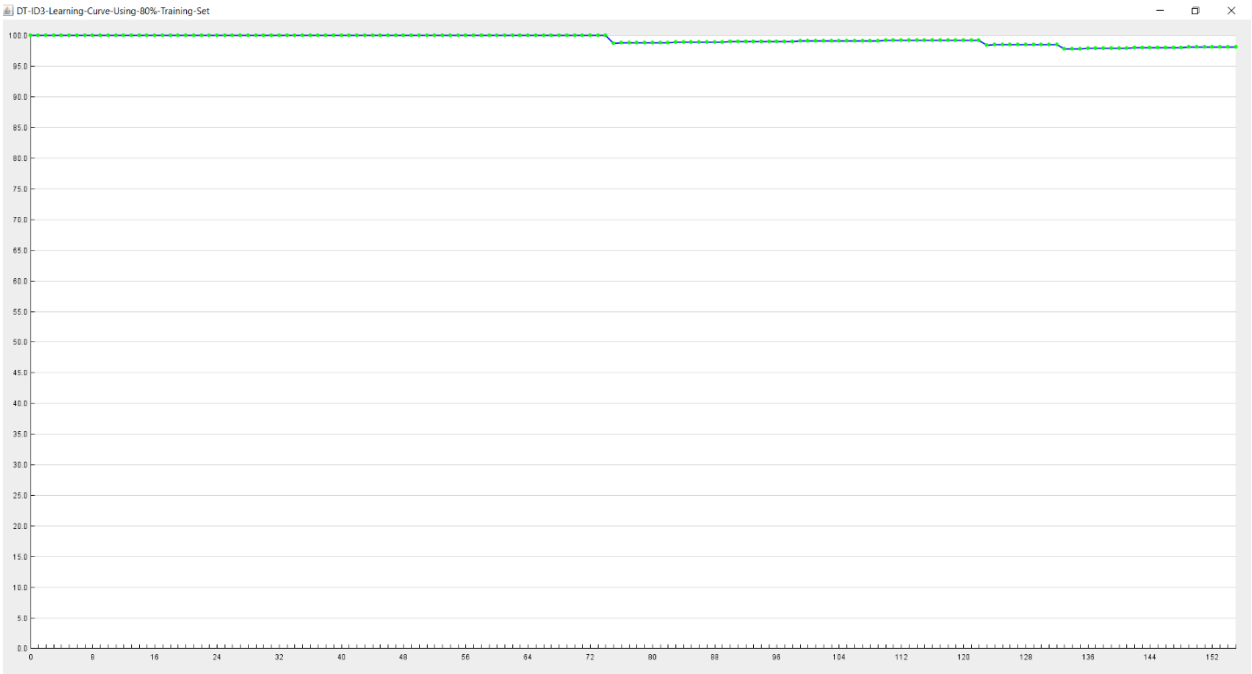
ג. פונקציית פיצול מקסימום $Gain$ נכתבה במחלקה $DecisionTree$ ולוקחת את הפיצ'ר בעל ה-$Gain$ המקסימאלי באופן הפוך לאנטרופיה.

<u>פונקציית ה-$Gain$:</u>

```
/**
 * computing Gain function
 * @param train current training-set
 * @param attributes    current attributes at this node to compute the best from this
 * @return  best attribute by gain function
 */
public String select_by_gain(List<FeatureVector> train,List<String> attributes) {
    String target_name = train.get(0).getName();
    double class_entropy = this.calc_entropy(train,target_name,true);
    double max_cond_ent = 0.0;
    String future_attr = attributes.get(0);
    for (String cond_attr : attributes) {
        double prob = this.cond_prob_by_class(train, target_name, cond_attr); // probability
        double attr_entropy = this.calc_entropy(train, cond_attr, false); // attribute entropy value
        double cond_entropy = class_entropy - prob*attr_entropy;    //confitinal entropy computation
        if(cond_entropy > max_cond_ent) { // comparison to pick the best
            max_cond_ent = cond_entropy;
            future_attr = cond_attr;
        }
    }
    return future_attr;
}
```

<u>השוואה בין עץ החלטה עם פונקציית פיצול מקסימום $Gain$ לבין $KNN$ עם $k$ אופטימלי:</u>

**הגרף של הרצת $DT$ – דיוק כפונקציה של כמות תצפיות:**




ניתן לראות בגרף הנ"ל שהאלגוריתם חוזה באופן מרשים את נתונים קבוצת הבדיקה וככל שהתצפיות גדלות כך **הטעות עולה** אבל לא בהרבה – הפרש של 10% לכל היותר בין הדגימות.

**<u>השוואה בין דיוקי המודלים:</u>**

| השוואה | $KNN\ with\ optimal\ k$ | $Decision\ Tree\ with\ Gain$ |
|---|---|---|
| זמן למידה עד שמגלה את $h$ | דורש זמן גבוה למציאת ה-$k$ | מתייצב באופן מהיר יחסית |
| דיוק לפי שיטת השוואה | משווא ע"י "מרחקים" בין התצפיות כאשר הגדרת המרחק משפיע גם הוא על ה-$k$ בכדי להיות אופטימלי ומשפר גם הוא את הדיוק כפי שראינו הנ"ל | פונקציית ה-$Gain$ כפי שהוסבר הנ"ל מוצאת באופן חכם יותר את הפיצ'רים שאיתם נפצל את העץ ולכן מכלילה את התצפיות ב-$test - set$ יותר ולכן דיוק המודל יכול לעבוד לאורך זמן ולהתאים ליותר נתונים כפי שהוצג בגרף |
| אחוזי דיוק | כאשר ה-$k$ הוא אופטימלי אחוז הדיוק כאן גבוה מאוד בדומה ל-$DT$ | אחוז הדיוק גבוה מאוד בדומה ל-$KNN$ עם $k$ אופטימלי |

הדפסת ה-$F1$, $confusion\ matrix, precision, recall$ ועקומת הלמידה ל-$KNN$ $(k = 3, p = 2, 80\%\ train)$:

```
******
Confusion Matrix:
                        Iris-versicolor         Iris-virginica          Iris-setosa
        Iris-versicolor         8                       1                       0
        Iris-virginica          0                       13                      0
        Iris-setosa             0                       0                       8
******
Iris-versicolor Precision = 88.88888888888889
Iris-virginica Precision = 100.0
Iris-setosa Precision = 100.0
******
Iris-versicolor Recall = 100.0
Iris-virginica Recall = 92.85714285714286
Iris-setosa Recall = 100.0
******
F-[Iris-versicolor]-measure = 94.11764705882352
F-[Iris-virginica]-measure = 96.2962962962963
F-[Iris-setosa]-measure = 96.2962962962963
******
```

הדפסת ה-$F1$, $confusion\ matrix, precision, recall$ ועקומת הלמידה ל-$DecisionTree$ $(80\%\ train)$:

```
Confusion Matrix:
=================
          democrat        republican
democrat        91        3
republican      2         60
******
democrat Precision: 0.9680851063829787
democrat Precision: 0.967741935483871
******
democrat Recall: 0.978494623655914
democrat Recall: 0.9523809523809523
******
F-[democrat]-measure: 0.9732620320855615
F-[democrat]-measure: 0.96
******
```

**סיכום ההבדלים בין 2 אלגוריתמי הלמידה המונחית שהוצגו הנ"ל:**

***KNN:***

**יתרונות** – מימוש פשוט לביצוע ושיטה מובנת ואף טריוויאלית, נותן ביצועים טובים על נתונים קטנים כאשר פונקציית המרחק בין התצפיות מוגדרת היטב.

**חסרונות** – דורש כמות אכסון גבוהה בכדי להריץ את המודל ביחס ל-$DT$ שלא שומר את כל הנתונים אלה רק את הפונקציה שעץ ההחלטה מייצר וזה בעצם ייצוג קומפקטי יותר מ-$KNN$ בהשוואה בין המודלים, מכמות אכסון גבוהה יוצא גם שבשלב הסיווג האלגוריתם רץ על כל התצפיות שניתנו לו בלמידה ולכן הוא בעצם לא בדיוק לומד אלה יותר אלגוריתם עצל ששומר נתונים בצורה נוחה ומביא באופן $broote\ force$ את התוצאה ולכן זמן הריצה שהוא דורש בכמויות גדולות יותר של נתונים נחשבת משמעותית יותר ויותר.

***:DT***

**יתרונות** – ייצוג קומפקטי לפונקציה המקורבת $h$ של התצפיות שנתנו לה בזמן הלמידה ולכן בניגוד ל-$KNN$ תופס שטח אכסון קטן יותר משמעותית כפונקציה של הקלט, מכליל הרבה יותר טוב את הנתונים מאשר $KNN$ בכמויות גדולות יותר של נתונים מכיוון שדורש פחות פרמטרי קלט מ-$KNN$ כגון: $k$ וגם עצם זה שמתייחס לאזורים במרחב ולא לכל תצפית. לכן, האלגוריתם הוא כמעט "עצמאי" למצוא את העץ שמתאר באופן הטוב ביותר את התצפיות שקיבל, מהר מאוד ניתן למצוא את כמות התצפיות שתיתן תוצאה טובה ומוכללת בניגוד ל-$KNN$ שמקיימת את זה עבור $k$ גדול יחסית (לפני כן זה $underfitting$ ולאחר מכן זה $overfitting$), משתמש בפונקציית ה-$Gain$ שהיא בעצם פונקציית ה-$mutual\ information$ שמזהה תלות בין משתנים וע"י ביצוע מקסימום ניתן להשים פיצולים שמסבירים באופן כללי יותר את תופעת התצפיות מכאן נובע שעץ החלטה זה אלגוריתם לומד ומשתמש ביותר כלים חכמים ובעלי משמעות מאשר $KNN$ ומכאן גם ההכללה הטובה יותר שלו.

**חסרונות** – ישנם פונקציות ידועות שנלמדו בהרצאה שאינו יודע לבצע בגלל צורת החלוקה של העולם שהוא מייצר, עלול להיכנס מהר מאוד ל-$overfitting$ ולשם כך ישנם פעולות שמטפלות בתופעה זו אבל מייקרות את תהליך למידה בזמן הריצה, כאשר מספר המשתנים/הפיצ'רים גדל איתו גם כמות האפשרויות לפיצול בכל צומת ולכן אלגוריתם זה במקרים מסוימים יכול להיות יותר מורכב מ-$KNN$ ופחות ברור לסביבה כך שמייצר בעיות כגון: $debugging$.