

Dominik Wróbel

Inżynieria oprogramowania i systemów

Informatyka, II stopień, 2018/19

Metody eksploracji danych

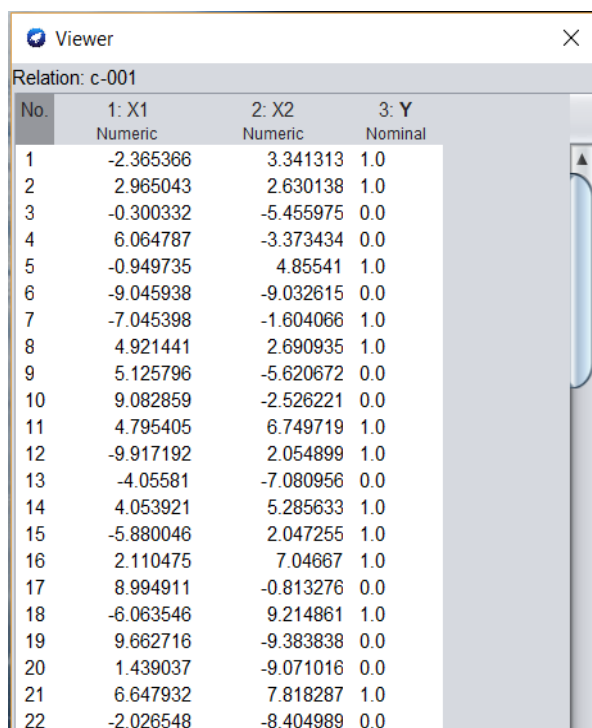
Laboratorium 5 – 30.04.2019

Programowe użycie bibliotek Weka z poziomu języka Java

Klasyfikatory: NaiveBayes, drzewa decyzyjne, SVM (Weka: SMO) i kNN (Weka: IBk)

### 5.1 Załadowanie zbioru uczącego i klasyfikacja

W oprogramowaniu weka otworzono plik c-001.arff.



Relation: c-001

No.	1: X1 Numeric	2: X2 Numeric	3: Y Nominal
1	-2.365366	3.341313	1.0
2	2.965043	2.630138	1.0
3	-0.300332	-5.455975	0.0
4	6.064787	-3.373434	0.0
5	-0.949735	4.85541	1.0
6	-9.045938	-9.032615	0.0
7	-7.045398	-1.604066	1.0
8	4.921441	2.690935	1.0
9	5.125796	-5.620672	0.0
10	9.082859	-2.526221	0.0
11	4.795405	6.749719	1.0
12	-9.917192	2.054899	1.0
13	-4.05581	-7.080956	0.0
14	4.053921	5.285633	1.0
15	-5.880046	2.047255	1.0
16	2.110475	7.04667	1.0
17	8.994911	-0.813276	0.0
18	-6.063546	9.214861	1.0
19	9.662716	-9.383838	0.0
20	1.439037	-9.071016	0.0
21	6.647932	7.818287	1.0
22	-2.026548	-8.404989	0.0

Następnie w IDE IntelliJ dodano do projektu plik weka.jar i napisano program przeprowadzający klasyfikację stworzonej w kodzie instancji na podstawie danych z pliku c-001.arff.

```

public class MainWeka {

    public static void main(String[] args) {

        System.out.println("This is Java Weka library !");

        try{
            // 5.1.1
            ConverterUtils.DataSource source = new ConverterUtils.DataSource( "c-001.arff");
            Instances data = source.getDataSet();

            // 5.1.2
            if (data.classIndex() == -1)                // if classIndex not set
                data.setClassIndex(data.numAttributes()-1);    // last attribute is the class

            // 5.1.3
            Classifier cls = new NaiveBayes();           // build model
            cls.buildClassifier(data);

            // 5.1.4
            Instance inst = new DenseInstance( numAttributes: 3);    // create instance
            inst.setDataset(data);
            inst.setValue( 0, -15);                // X1 value
            inst.setValue( 1, -10);                // X2 value

            // 5.1.5
            double y = cls.classifyInstance(inst);    // classify instance
            System.out.println("Classified as " + y);

            // 5.1.6
            double[] distrib = cls.distributionForInstance(inst);
            System.out.printf(Locale.US, format: "%d->%f %d->%f\n", ...args: 0,distrib[0],1,distrib[1]);

        }catch(Exception e){
            e.printStackTrace();
        }

    }
}

```

Testy przeprowadzono dla różnych wartości wejściowych x1 i x2. Wybrane wyniki przedstawiono w tabeli poniżej.

Nr instancji	X1	X2	Zakwalifikowano jako	Prawdopodobieństwo 0	Prawdopodobieństwo 1
1.	-15	-10	0	0.77	0.23
2.	-10	-10	0	0.85	0.15
3.	-5	-5	0	0.59	0.41
4.	-2	2	1	0.02	0.98
5.	5	5	1	0.01	0.99

## 5.2 Klasyfikacja wygenerowanego zbioru instancji

W tym zadaniu ponownie skorzystano ze zbioru danych c-001.arff. W kodzie stworzono zbiór danych składający się z numerycznych atrybutów wejściowych X1, X2 oraz atrybutu wyjściowego Y (tak lub nie). Następnie w pętli wygenerowano dane testowe z zakresu -10 do 10 i stworzono z nich instancje, które poddano klasyfikacji. Na końcu programu stworzono z danych plik .arff.

```
// 5.1.3
Classifier cls = new NaiveBayes(); // build model
cls.buildClassifier(data);

// 5.2.2
List<Attribute> atts= Arrays.asList( // create output data set
    new Attribute( attributeName: "X1"),
    new Attribute( attributeName: "X2"),
    new Attribute( attributeName: "Y", Arrays.asList("tak","nie"))
);

Instances result = new Instances( name: "some-relation", new ArrayList<>(atts), capacity: 0);
result.setClassIndex(result.numAttributes()-1);

// 5.2.3
for(double x1 = -10; x1 <= 10; x1 += 0.1){
    for(double x2 = -10; x2 <= 10; x2 += 0.1){
        Instance inst = new DenseInstance( numAttributes: 3); // create new instance
        inst.setValue( 0, x1);
        inst.setValue( 1, x2);
        inst.setDataset(result);
        double y = cls.classifyInstance(inst); // classify the instance
        inst.setClassValue(y);
        result.add(inst);
    }
}

// 5.2.4
ArffSaver saver = new ArffSaver(); // save file
saver.setInstances(result);
saver.setFile(new File( pathname: "c-001-result.arff"));
saver.writeBatch();

} catch (Exception e) {
    e.printStackTrace();
}

}
```

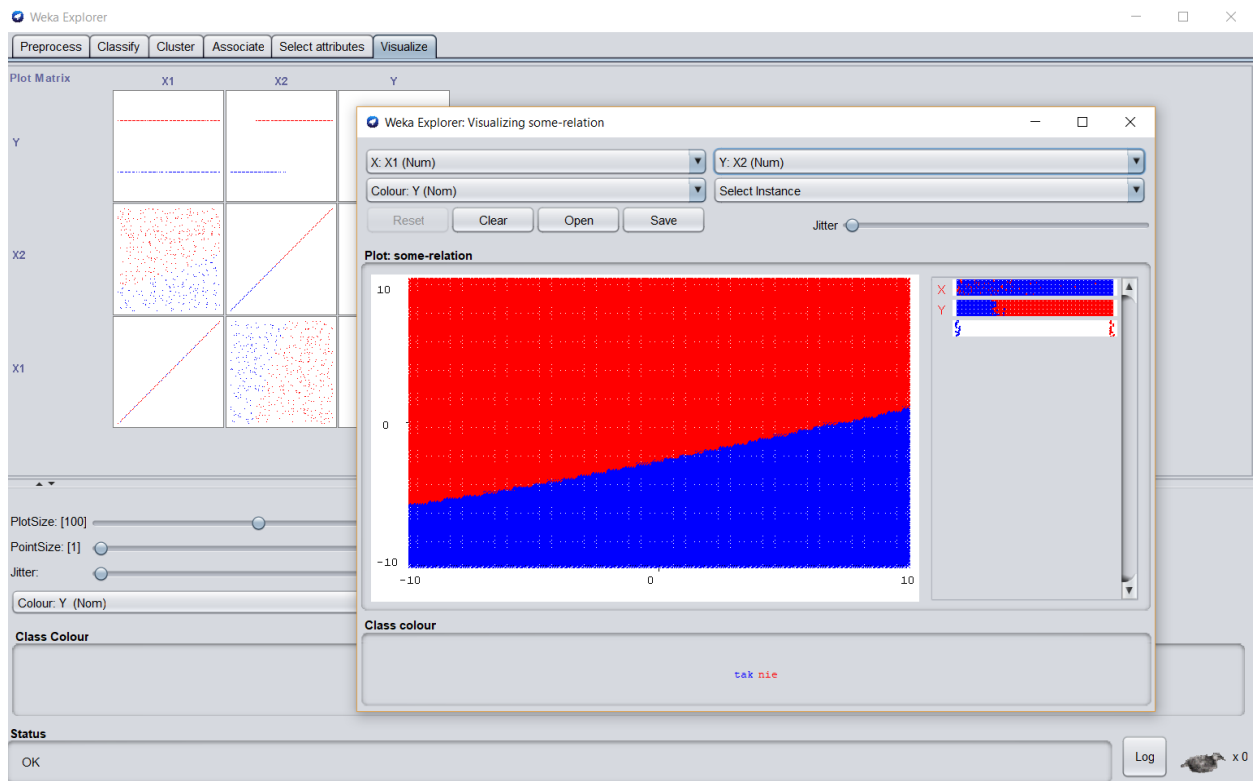
Fragment utworzonego pliku .arff :

```
@relation some-relation

@attribute X1 numeric
@attribute X2 numeric
@attribute Y {tak,nie}

@data
-10,-10,tak
-10,-9.9,tak
-10,-9.8,tak
-10,-9.7,tak
-10,-9.6,tak
-10,-9.5,tak
-10,-9.4,tak
-10,-9.3,tak
-10,-9.2,tak
-10,-9.1,tak
-10,-9,tak
-10,-8.9,tak
-10,-8.8,tak
```

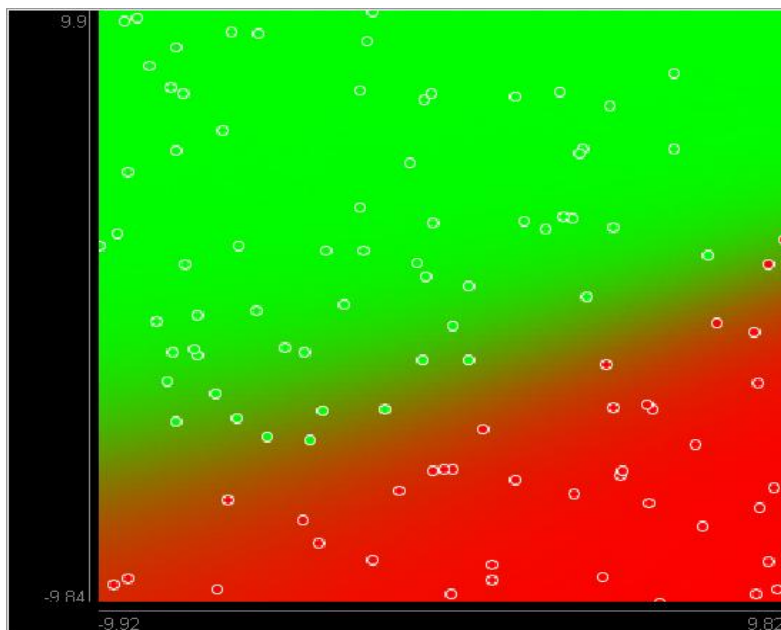
Uzyskany plik otworzono w eksploratorze Weka.



W tej klasyfikacji użyty został naiwny klasyfikator Bayesa. Granica klas jest w przybliżeniu linią prostą, znacznie więcej instancji zostało zakwalifikowane jako 'Nie'.

### 5.3 Wypróbuj gotowe rozwiązanie

W oprogramowaniu Weka skorzystano z gotowego rozwiązania do klasyfikacji przy użyciu naiwnego klasyfikatora Bayesa. Uzyskany wynik przedstawiono poniżej



Gotowe rozwiązanie w Weka dało zbliżone rozwiązanie do tego użytego w kodzie Java.

## 5.4 Ocena jakości klasyfikatora

W programie załadowano plik c-001.arff i stworzono dla danych naiwny model Bayesa. Następnie przy pomocy 10-krotnej walidacji krzyżowej przeprowadzono proces walidacji modelu.

```
public class MainWeka3 {  
  
    public static void main(String[] args) {  
  
        System.out.println("This is Java Weka library !");  
  
        try{  
            // 5.1.1  
            ConverterUtils.DataSource source = new ConverterUtils.DataSource( location: "c-001.arff");  
            Instances data = source.getDataSet();  
  
            // 5.1.2  
            if (data.classIndex() == -1)                // if classIndex not set  
                data.setClassIndex(data.numAttributes()-1);    // last attribute is the class  
  
            // 5.4.2  
            Classifier cls = new NaiveBayes();                // build model  
            Evaluation eval= new Evaluation(data);  
            eval.crossValidateModel(cls, data, numFolds: 10, new Random( seed: 1));  
  
            eval.toSummaryString();  
            eval.confusionMatrix();  
            System.out.printf( Locale.US,  
                format: "[precrecallfmeasure]:\t%f\t%f\t%f\n",  
                    eval.weightedPrecision(),  
                    eval.weightedRecall(),  
                    eval.weightedFMeasure()  
                );  
        }catch(Exception e){  
            e.printStackTrace();  
        }  
    }  
}
```

Uzyskano wynik:

[prec recall fmeasure]: 0.970047	0.970000	0.969920
----------------------------------	----------	----------

Wskaźnik prec określa procentową liczbę przypadków w których klasyfikator zwrócił poprawny wynik. W tym przypadku wskaźnik ten jest dość wysoki wynosi 97%.

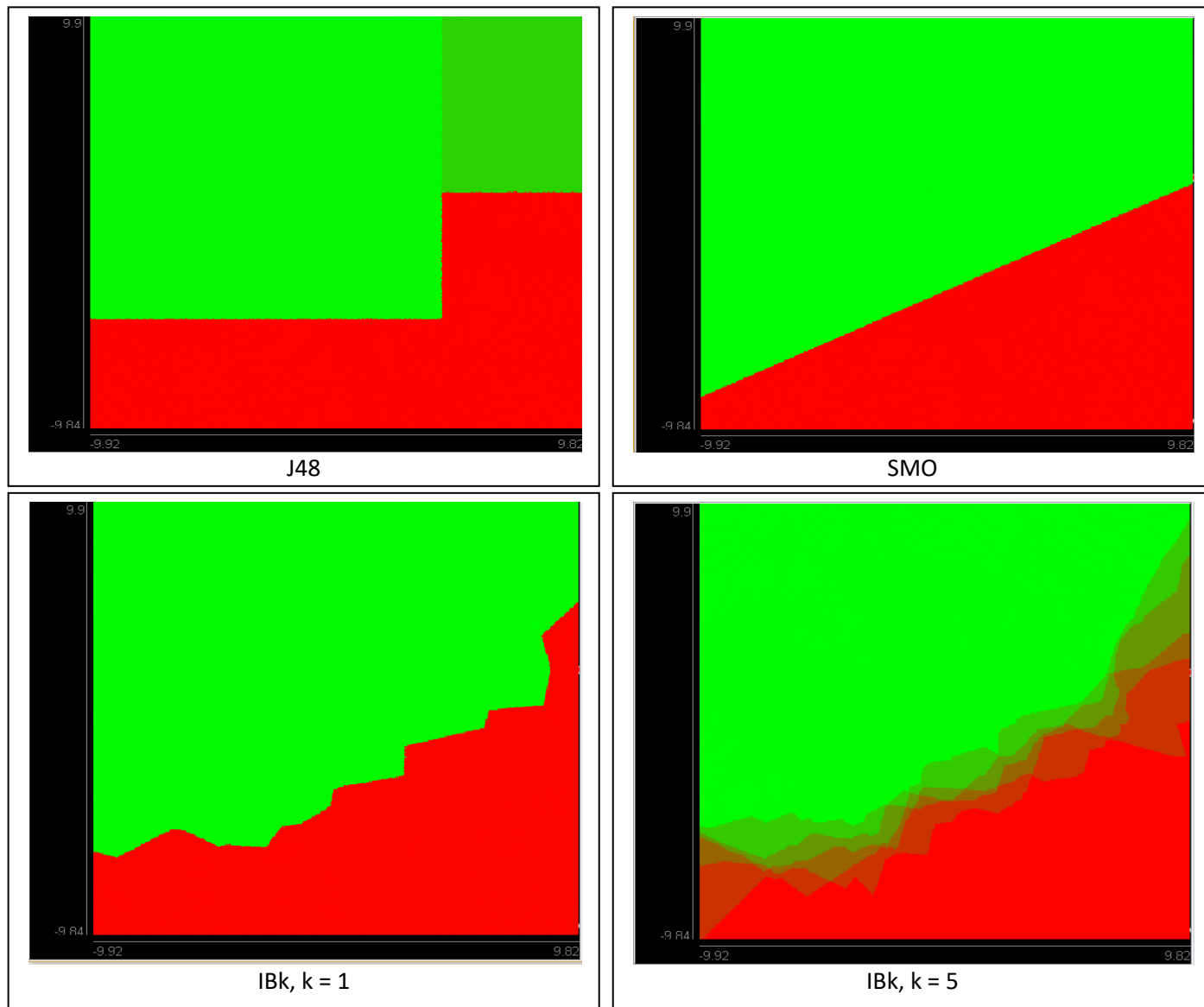
Wskaźnik recall jest to stosunek dobrze sklasyfikowanych instancji do ilości wszystkich instancji, jego wysoka wartość świadczy o dobrej jakości klasyfikacji.

Wskaźnik fmeasure to wskaźnik obliczany na podstawie prec i recall. W tym przypadku oba te wskaźniki mają zbliżoną wartość co daje w wyniku wysoką wartość wskaźnika fmeasure.

## 5.5 Ocena jakości kolejnych klasyfikatorów

W tym zadaniu zbadano jakość innych klasyfikatorów w celu porównania ich działania.

W BoundaryVisualizer zbadano klasyfikację korzystając z klasyfikatora J48, SMO, k-NN otrzymano wyniki:



Dla tych samych klasyfikatorów wyznaczono wskaźniki w kodzie:

J48 [prec recall fmeasure]:	0.950989	0.950000	0.949563
SMO [prec recall fmeasure]:	0.980625	0.980000	0.979888
IBk, k = 1 [prec recall fmeasure]:	0.990256	0.990000	0.990024
IBk, k = 5 [prec recall fmeasure]:	0.990256	0.990000	0.990024

Wskaźniki o najlepszych wartościach udało uzyskać się dla k-NN. Najgorzej poradził sobie klasyfikator J48, dla którego wskaźniki wynoszą w przybliżeniu 0,95.

```
public static void main(String[] args) {

    System.out.println("This is Java Weka library !");

    try{
        // 5.1.1
        ConverterUtils.DataSource source = new ConverterUtils.DataSource("c-001.arff");
        Instances data = source.getDataSet();

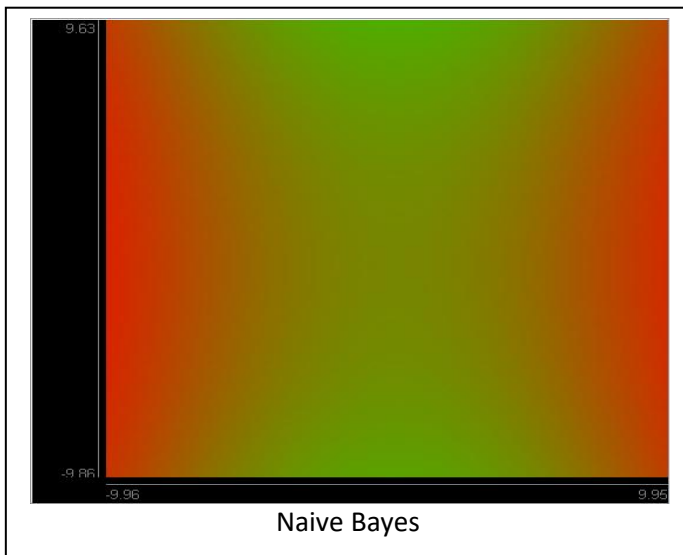
        // 5.1.2
        if (data.classIndex() == -1) // if classIndex not set
            data.setClassIndex(data.numAttributes()-1); // last attribute is the class

        // 5.4.2
        // Classifier cls = new NaiveBayes(); // build model
        // Classifier cls = new J48();
        // Classifier cls = new SMO();
        // Classifier cls = new IBk();
        Classifier cls = new IBk();

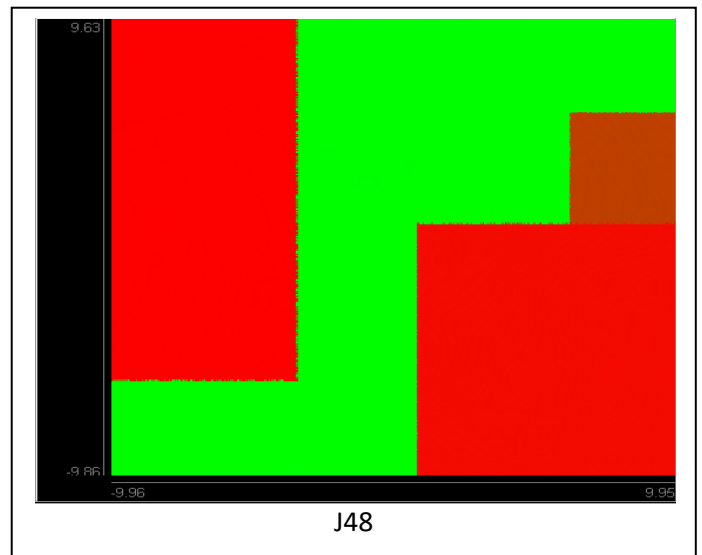
        Evaluation eval = new Evaluation(data);
        eval.crossValidateModel(cls, data, numFolds: 10, new Random(seed: 1));

        eval.toSummaryString();
        eval.confusionMatrix();
        System.out.printf(Locale.US,
            format: "[precrecallfmeasure]:\t%f\t%f\t%f\n",
            eval.weightedPrecision(),
            eval.weightedRecall(),
            eval.weightedFMeasure()
        );
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

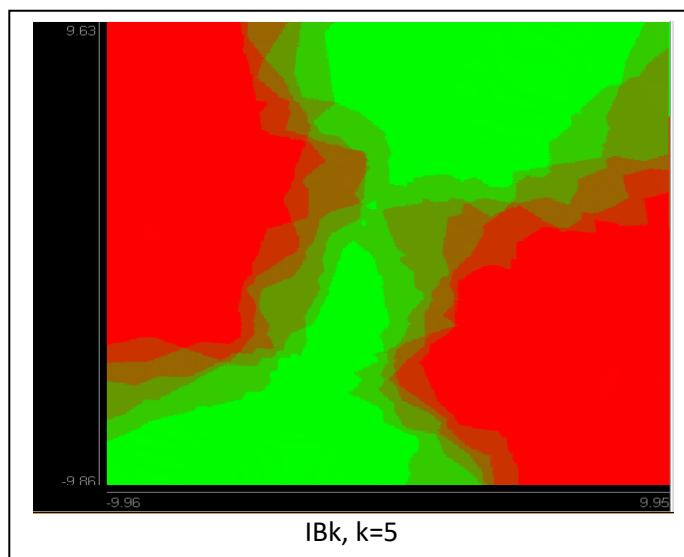
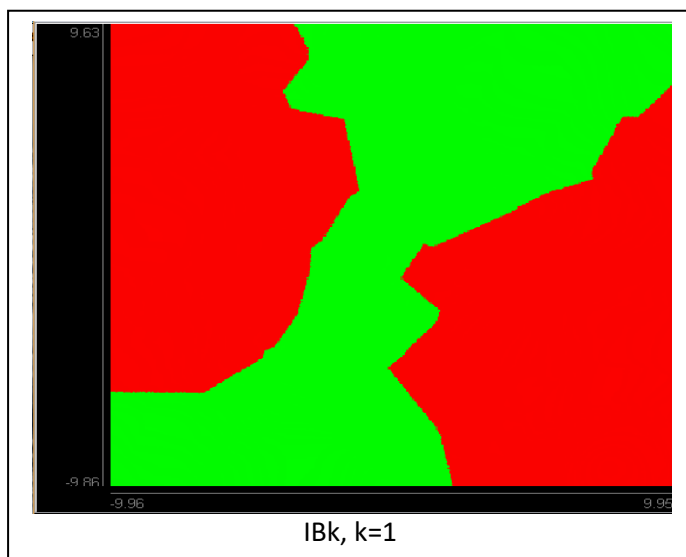
## 5.6 Przetwarzamy c-002.arff



Naive Bayes



J48

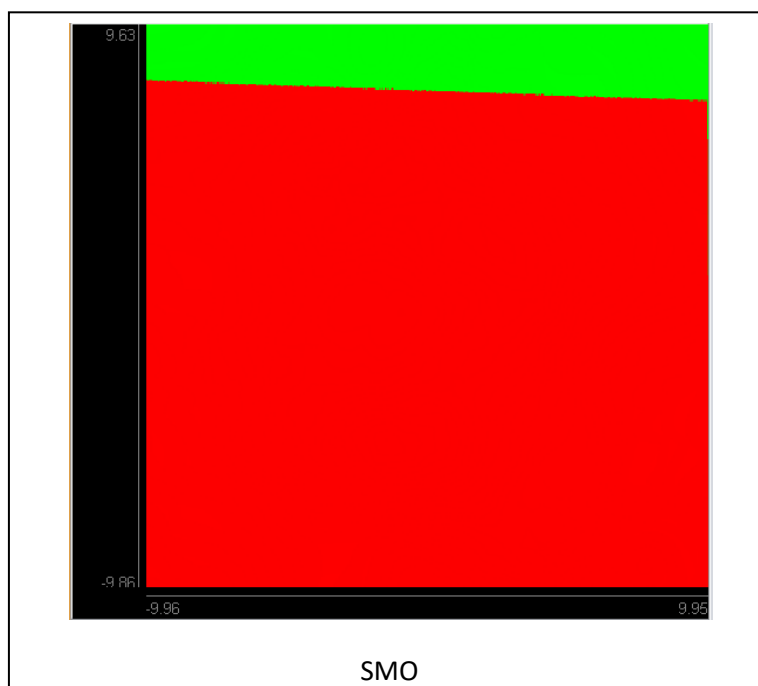


NB [prec recall fmeasure]:	0.761500	0.760000	0.757944
J48 [prec recall fmeasure]:	0.930285	0.930000	0.930049
IBk, k=1 [prec recall fmeasure]:	0.911909	0.910000	0.910135
IBk, k=5 [prec recall fmeasure]:	0.934862	0.930000	0.930105

Tym razem najlepsze wartości wskaźników udało uzyskać się dla IBk przy k=5 oraz dla J48. Dla tego zbioru danych bardzo słabe wyniki uzyskano dla klasyfikatora NaiveBayes.

## 5.7 Stosujemy SMO

Dla pliku c-002.arff zastosowano klasyfikator SMO. Uzyskane wyniki nie są najlepsze.

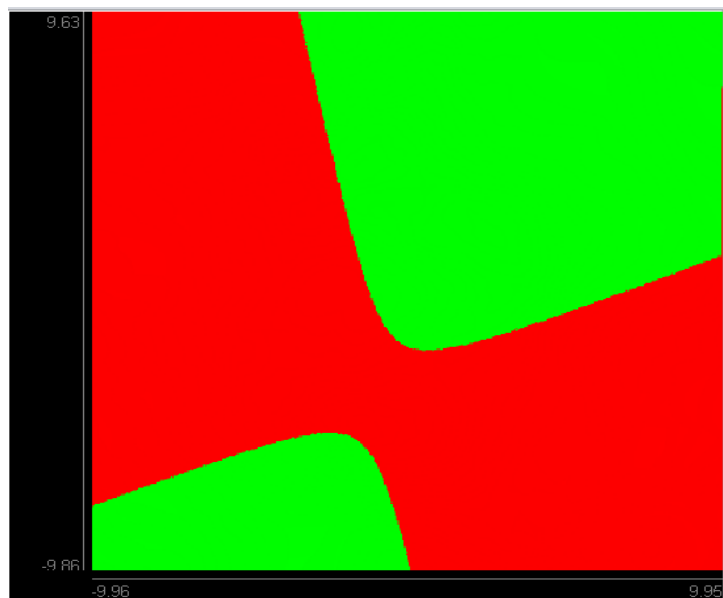




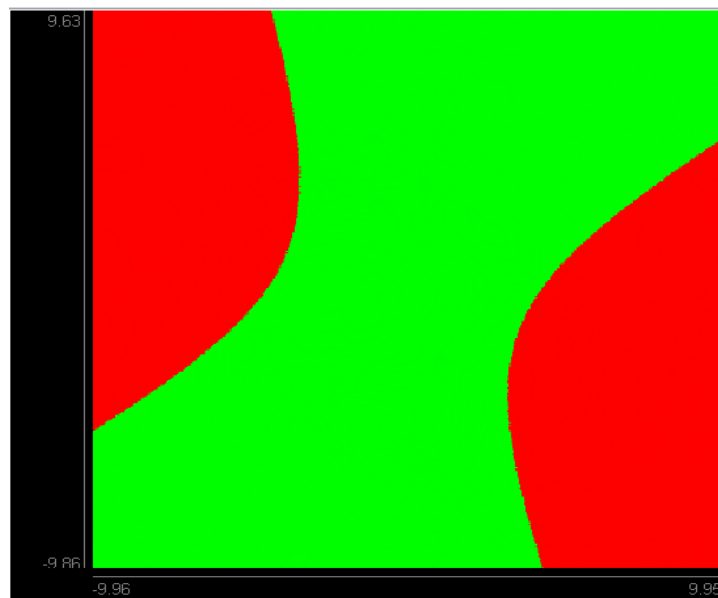
Niezadowolające działanie tego klasyfikatora potwierdzają również wskaźniki uzyskane z wykonania kodu:

SMO [prec recall fmeasure]:	0.754545	0.550000	0.400751
-----------------------------	----------	----------	----------

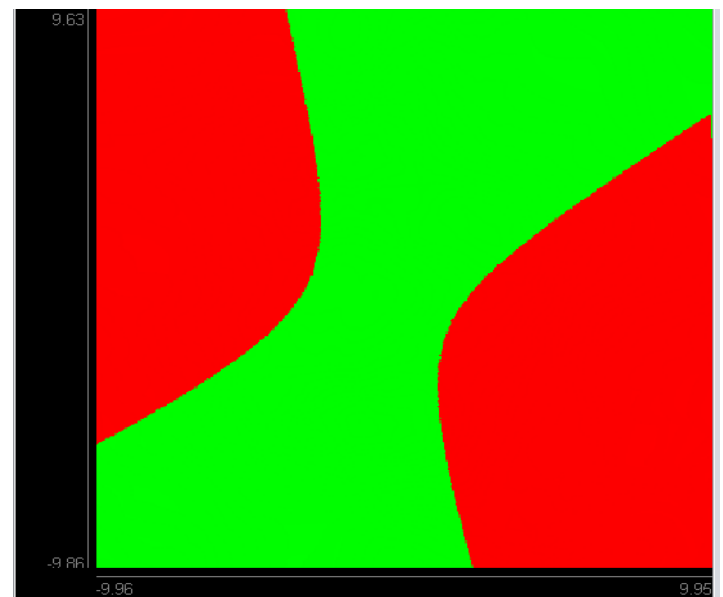
Dla klasyfikatora wypróbowano różne wartości parametru Exponent dla PolyKernel w BoundaryView.



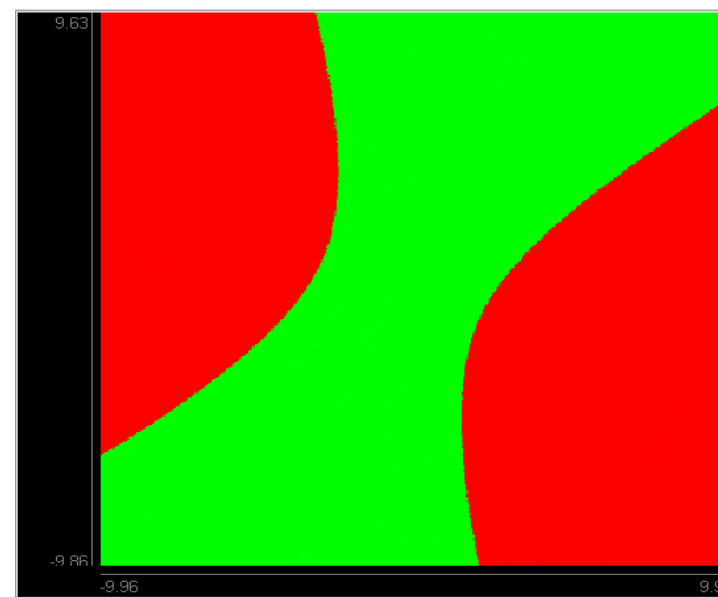
SMO, Exponent = 2.0



SMO, Exponent = 3.0



SMO, Exponent = 5.0



SMO, Exponent = 8.0

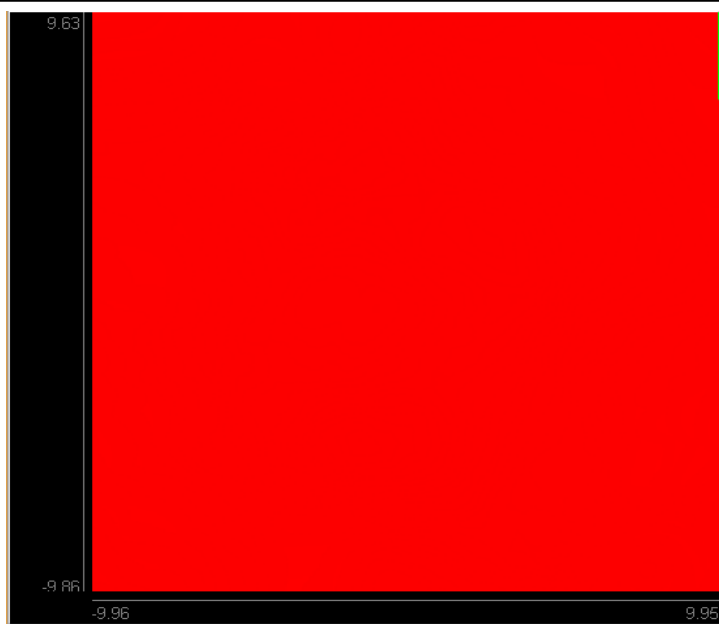
### 5.8 Przenieśmy te opcje do kodu

SMO, Exponent = 2 [prec recall fmeasure]:	0.767118	0.760000	0.755535
SMO, Exponent = 3 [prec recall fmeasure]:	0.898644	0.870000	0.869153
SMO, Exponent = 5 [prec recall fmeasure]:	0.911594	0.900000	0.900000
SMO, Exponent = 8 [prec recall fmeasure]:	0.960865	0.960000	0.960048

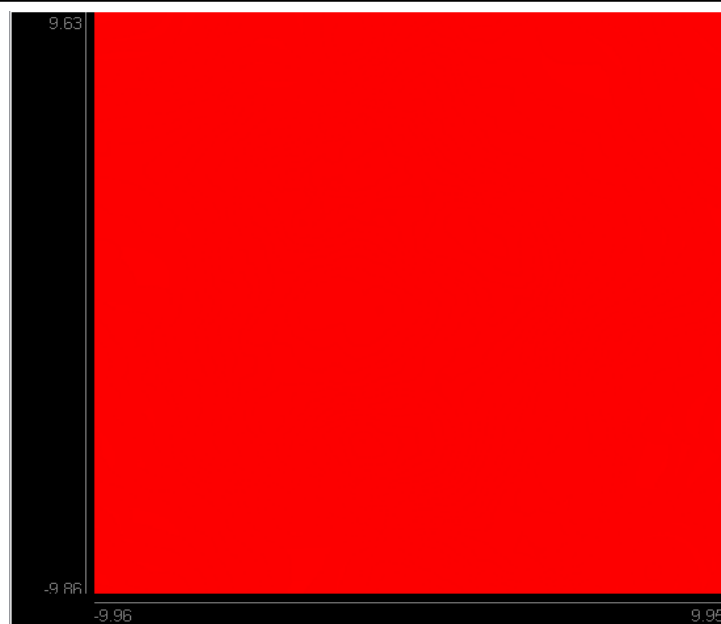
Zadanie to pokazało, że zmiana funkcji jądra ma bardzo duży wpływ na poprawność działania klasyfikatora. Zmiana parametru Exponent na większy za każdym razem przyniosła poprawę klasyfikacji, ostatecznie udało uzyskać się wartości wskaźników lepsze niż dla jakiegokolwiek innego klasyfikatora już przy Exponent = 8.0.

### 5.9 Kernel RBF

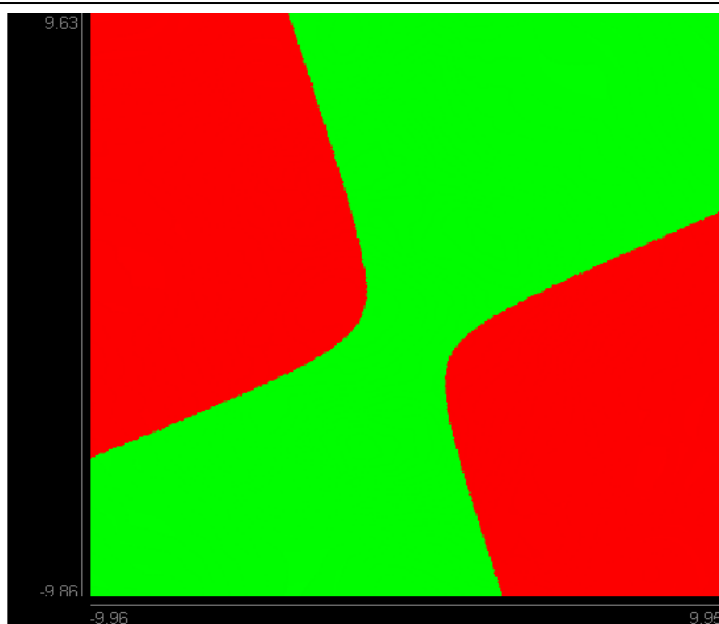
W tym zadaniu ponownie skorzystano z klasyfikatora SMO, jednak tym razem przyjęto funkcję jądra RBF. Zbadano różne wartości współczynnika gamma w BoundaryView oraz w kodzie programu.



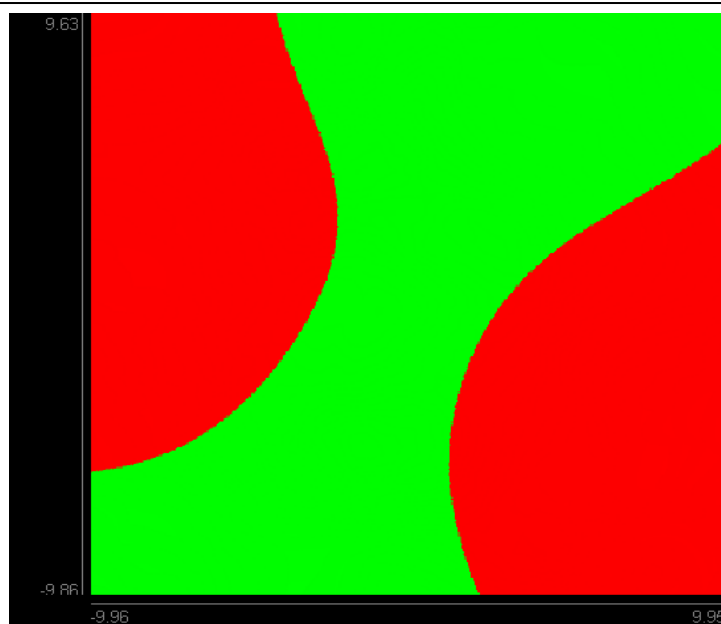
SMO, gamma = 0.01



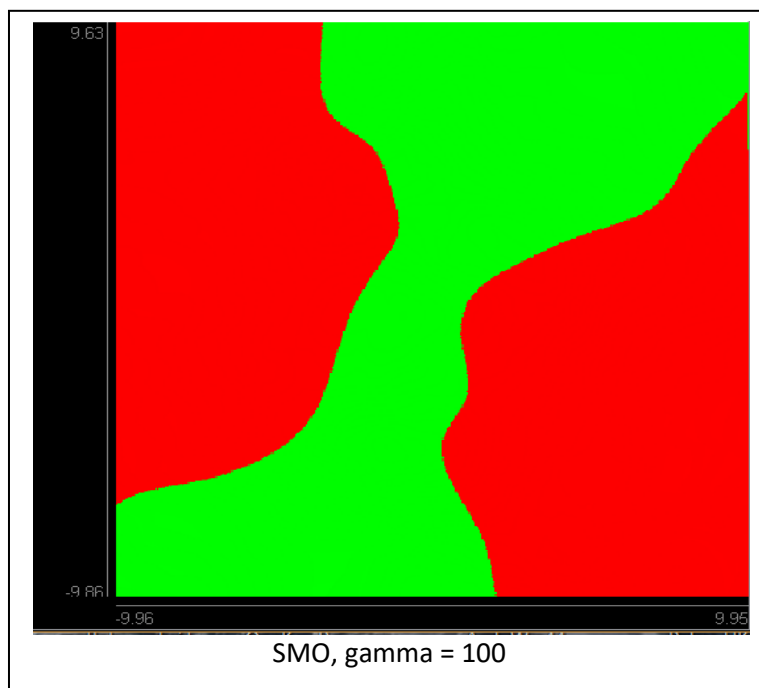
SMO, gamma = 0.1



SMO, gamma = 1



SMO, gamma = 10

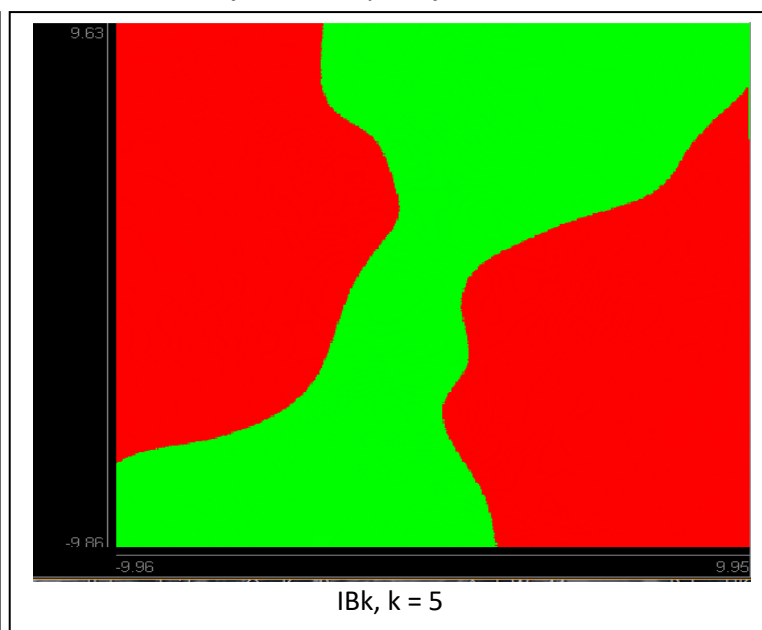
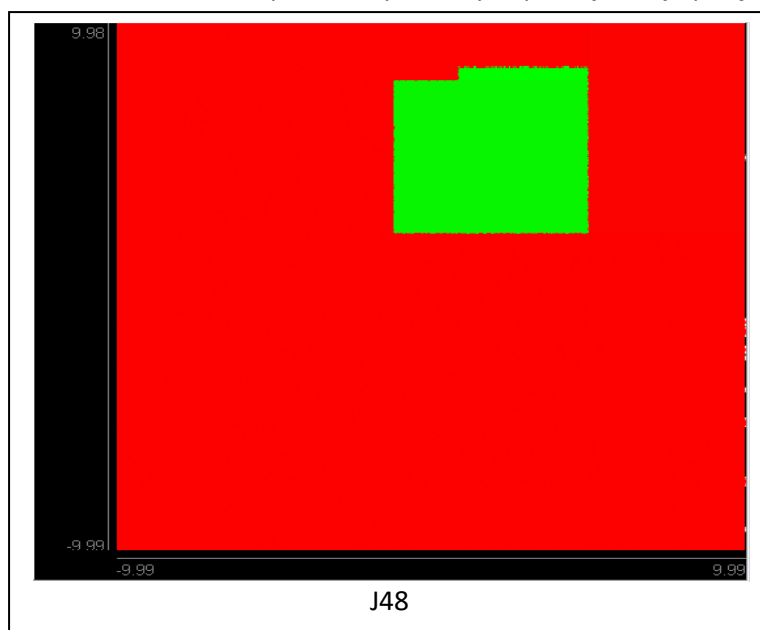


SMO, RBF, gamma = 0.01 [prec recall fmeasure]:	NaN	0.540000	NaN
SMO, RBF, gamma = 0.1 [prec recall fmeasure]:	NaN	0.540000	NaN
SMO, RBF, gamma = 1 [prec recall fmeasure]:	0.863200	0.860000	0.860224
SMO, RBF, gamma = 10 [prec recall fmeasure]:	0.939245	0.930000	0.930049
SMO, RBF, gamma = 100 [prec recall fmeasure]:	0.911909	0.910000	0.910135

W tym przypadku zmiana funkcji jądra nie przyniosła lepszej wartości wskaźników niż w przypadku wielomianowej funkcji jądra. Można zaobserwować wzrost jakości klasyfikacji wraz ze wzrostem współczynnika gamma, jednak wzrost ten nie jest nieskończony (np. dla gamma = 100 wartość wskaźników jest gorsza niż dla gamma = 10).

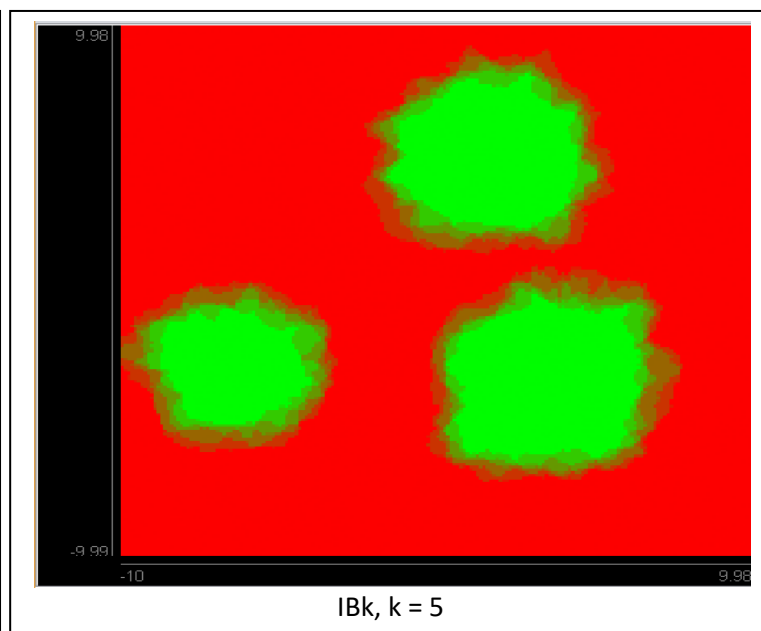
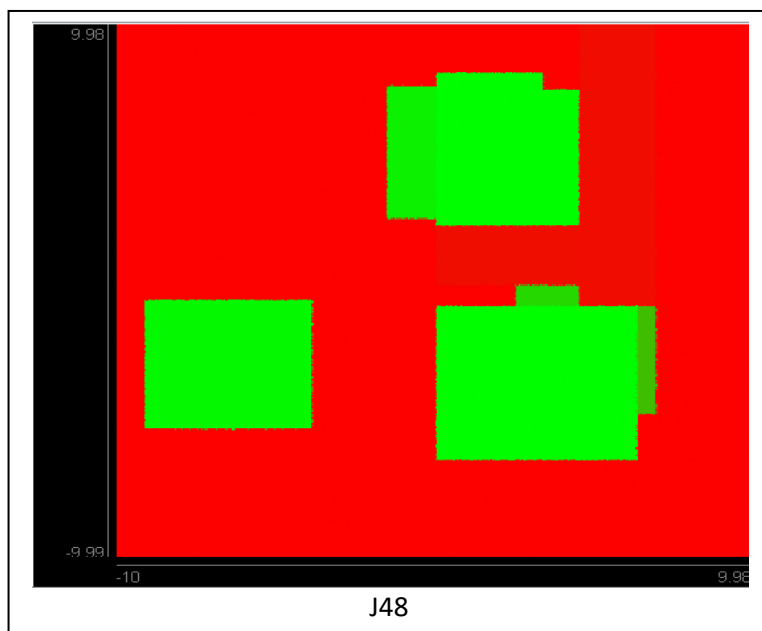
### 5.10 Plik c-003.arff

W tym i kolejnych zadaniach przeprowadzono klasyfikację przy użyciu różnych metod oraz dobierając różne ich parametry tak aby uzyskać jak najlepszą wartość wskaźników jakości klasyfikacji.



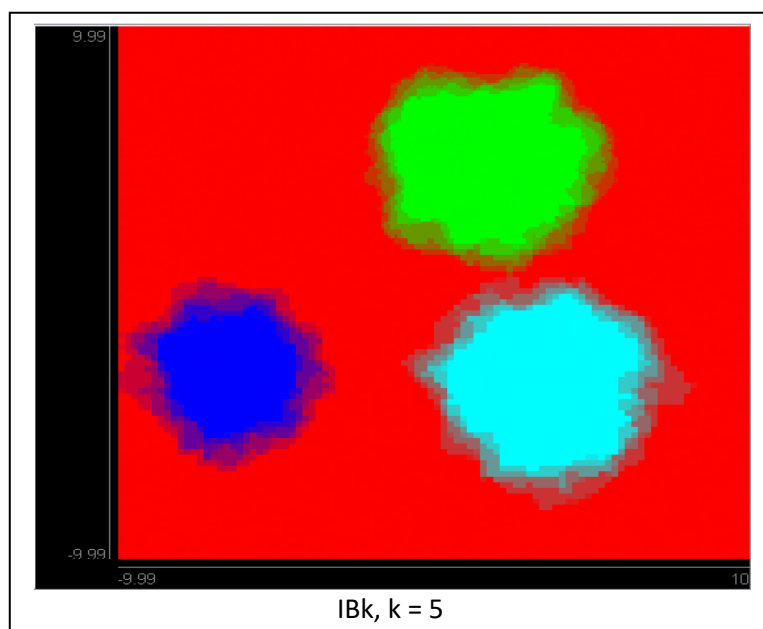
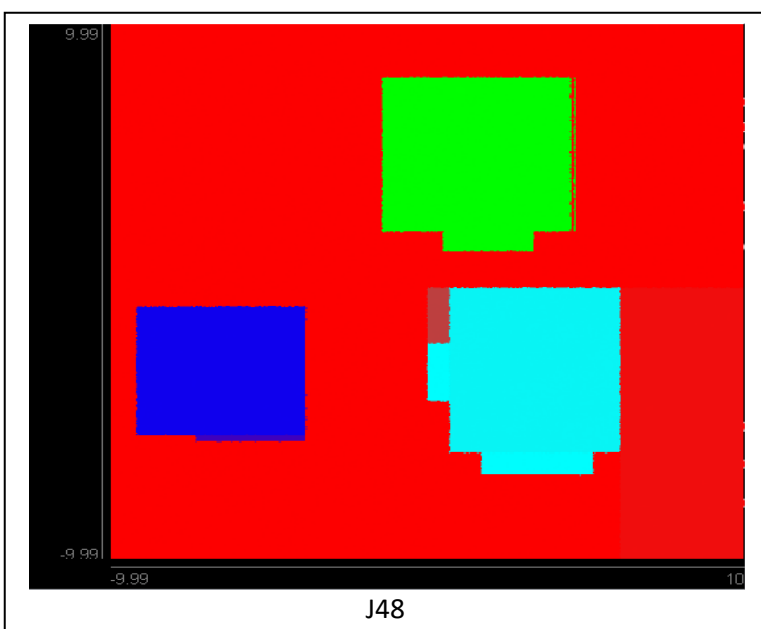
Klasyfikator	Parametry	Precision	Recall	F measure
Naive Bayes		0.958967	0.957000	0.951831
J48		0.988281	0.988000	0.988101
IBk	k=1	0.994984	0.995000	0.994989
SMO+PolyKernel	Exp=5	0.998000	0.998000	0.998000
SMO+RBFKernel	Gamma=100	0.999001	0.999000	0.998998

### 5.11 Plik c-004.arff



Klasyfikator	Parametry	Precision	Recall	F measure
Naive Bayes		NaN	0.751000	NaN
J48		0.961907	0.962000	0.961949
IBk	k=5	0.969846	0.970000	0.969877
SMO+PolyKernel	Exp=10	0.896211	0.898000	0.893951
SMO+RBFKernel	Gamma=100	0.972955	0.973000	0.972754

### 5.15 Plik c-005.arff



Klasyfikator	Parametry	Precision	Recall	F measure
Naive Bayes		0.891733	0.873000	0.855940
J48		0.934415	0.934000	0.934119
IBk	k=5	0.969501	0.969000	0.969130
SMO+PolyKernel	Exp=10	0.986412	0.986000	0.986110
SMO+RBFKernel	Gamma=100	0.967914	0.968000	0.967596