

Министерство образования Республики Беларусь  
Учреждение образования  
«Белорусский государственный университет информатики и  
радиоэлектроники»  
Институт информационных технологий

Факультет компьютерных технологий

Специальность ПОИТ

## **Контрольная Работа**

По дисциплине «Методы и алгоритмы принятия решений»  
Лабораторная работа 1

Выполнил: студент гр. 881072 Пискарев К.А.  
Проверил: Бакунов А.М.

Минск 2020

## **ЛАБОРАТОРНАЯ РАБОТА №1**

### **РАСПОЗНАВАНИЕ ОБРАЗОВ НА ОСНОВЕ КОНТРОЛИРУЕМОГО ОБУЧЕНИЯ**

Цель работы: изучить особенности методов распознавания образов, использующих контролируемое обучение, и научиться классифицировать объекты с помощью алгоритма К-средних.

#### **Порядок выполнения работы**

1. Изучение теоретической части лабораторной работы.
2. Реализация алгоритма К-средних.
3. Защита лабораторной работы.

Процесс распознавания образов напрямую связан с процедурой обучения.

Главная особенность контролируемого обучения заключается в обязательном наличии априорных сведений о принадлежности к определенному классу каждого вектора измерений, входящего в обучающую выборку. Роль обучающего состоит в том, чтобы помочь отнести каждый вектор из тестовой выборки к одному из имеющихся классов. И хотя классы известны заранее, необходимо уточнить и оптимизировать процедуры принятия решений. В основу всех алгоритмов распознавания образов положено понятие «расстояние», выступающее критерием в ходе принятия решений.

В качестве примера метода распознавания образов, использующего процедуру контролируемого обучения, рассмотрим алгоритм К-средних.

Исходные данные – число образов и число классов ( $K$ ), на которое нужно разделить все образы. Количество образов предлагается брать в диапазоне от 1000 до 100 000, число классов – от 2 до 20. Признаки объектов задаются случайным образом, это координаты векторов. Обычно  $K$  элементов из набора векторов случайным образом назначают центрами классов.

Цель и результат работы алгоритма – определить ядрами классов  $K$  типичных представителей классов и максимально компактно распределить вокруг них остальные объекты выборки.

## ДЕМОНСТРАЦИЯ РАБОТЫ ПРОГРАММЫ

После запуска программы отображается форма, представленная на рисунке 1.

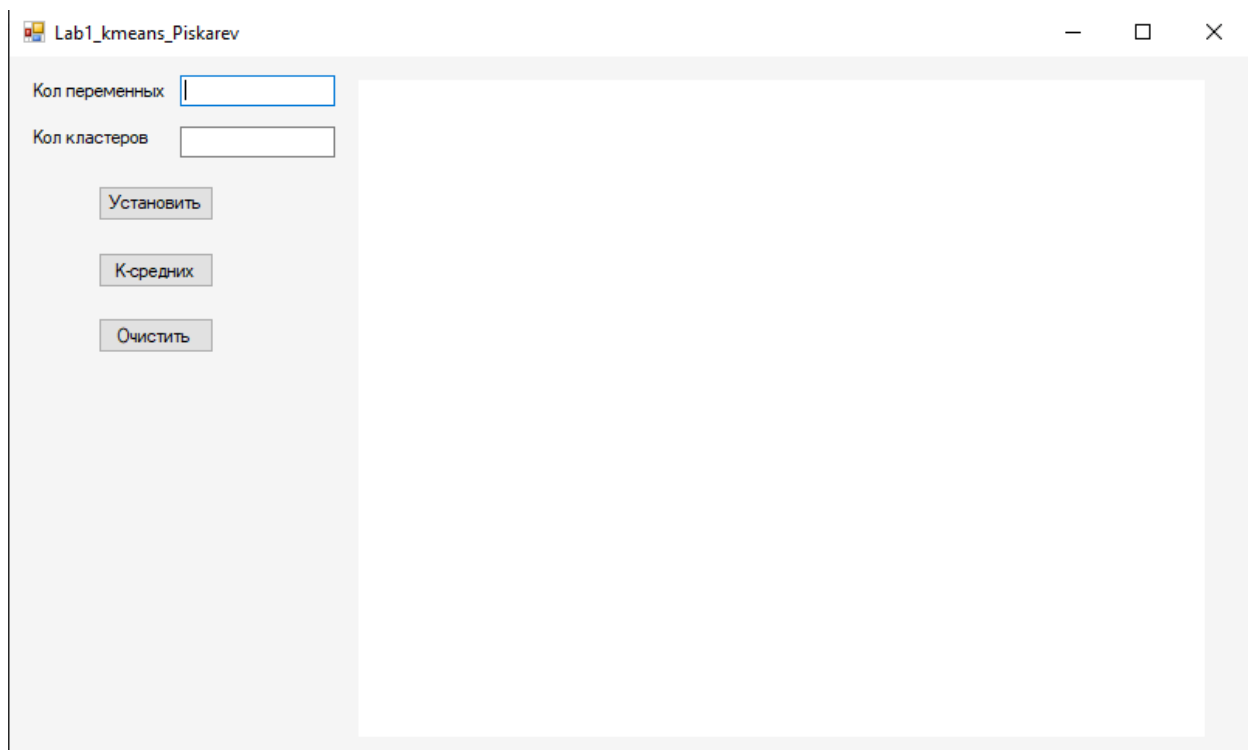
The image shows a screenshot of a Windows application window titled "Lab1\_kmeans\_Piskarev". The window has a standard Windows title bar with minimize, maximize, and close buttons. The main area of the window is divided into two parts. On the left, there is a control panel with two input fields: "Кол переменных" (Number of variables) and "Кол кластеров" (Number of clusters). Below these fields are three buttons: "Установить" (Set), "К-средних" (K-means), and "Очистить" (Clear). The right part of the window is a large, empty white rectangular area, likely intended for displaying the results of the K-means algorithm.

Рисунок 1

Требуется заполнить следующие поля количество переменных и количество кластеров и после нажать на кнопку установить, для первичной инициализации. Результат представлен на рисунке 2.

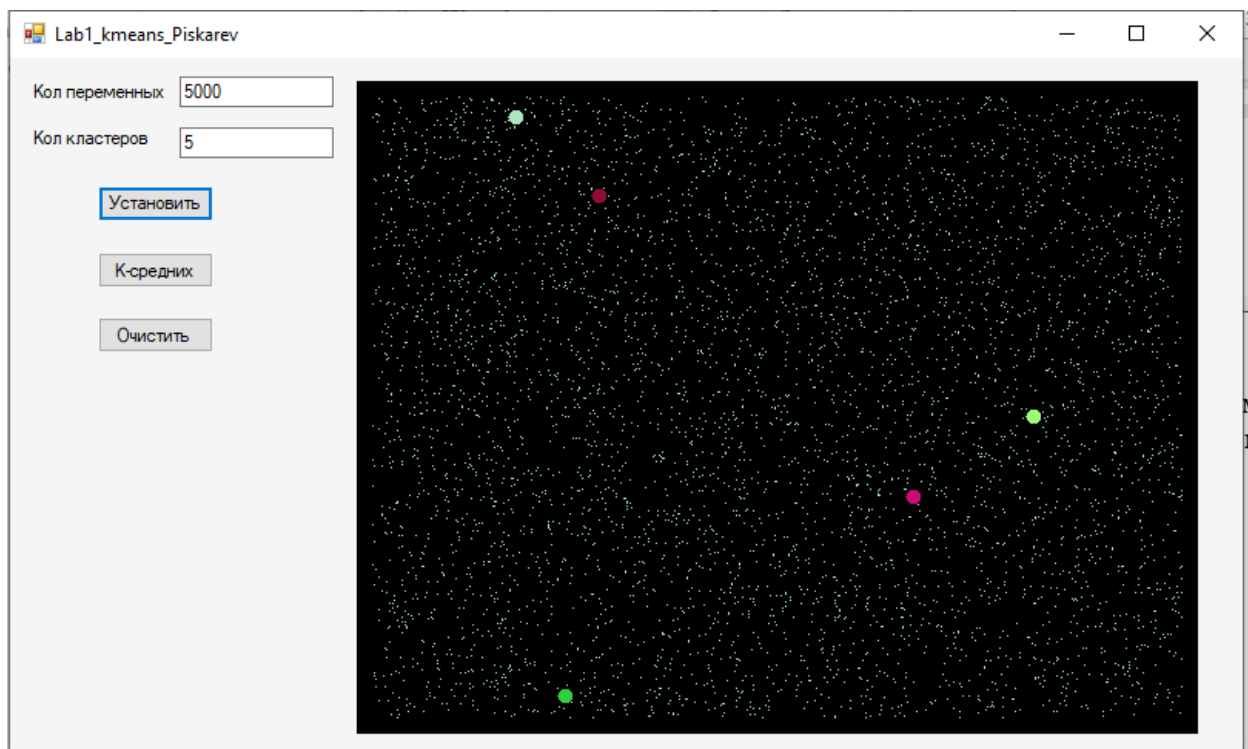


Рисунок 2

Для запуска алгоритма нажать на кнопку К-средних и после этого будет постепенно обновляться изображение. Пример представлен на рисунке 3.

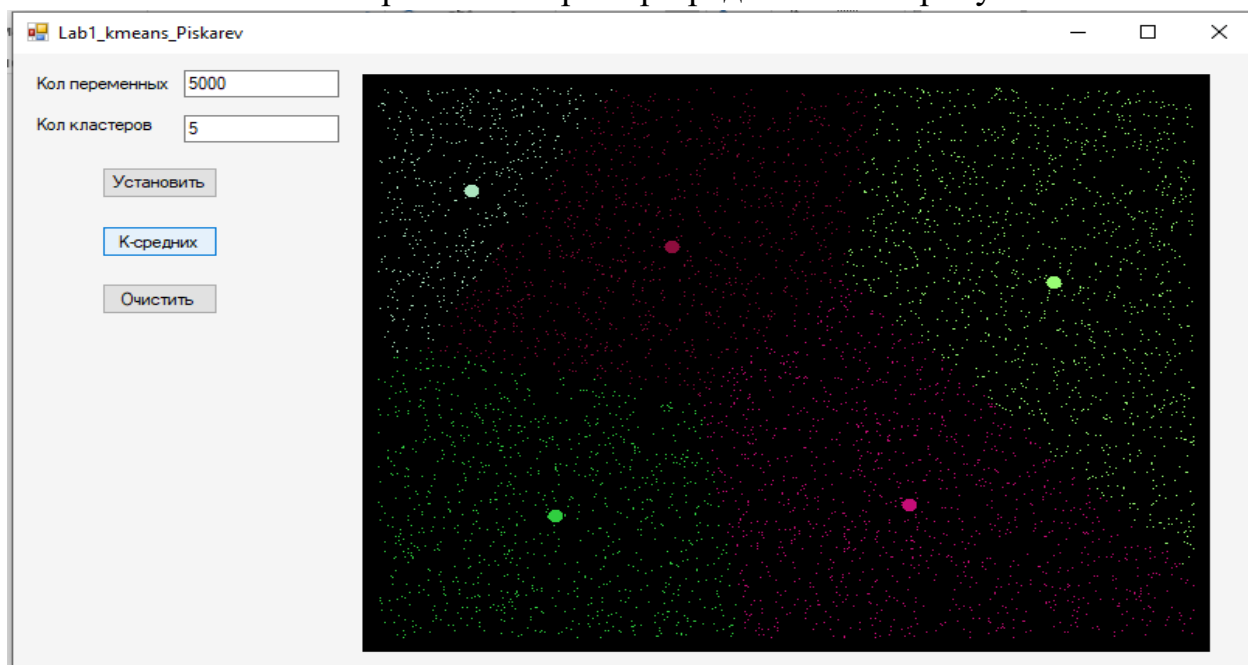
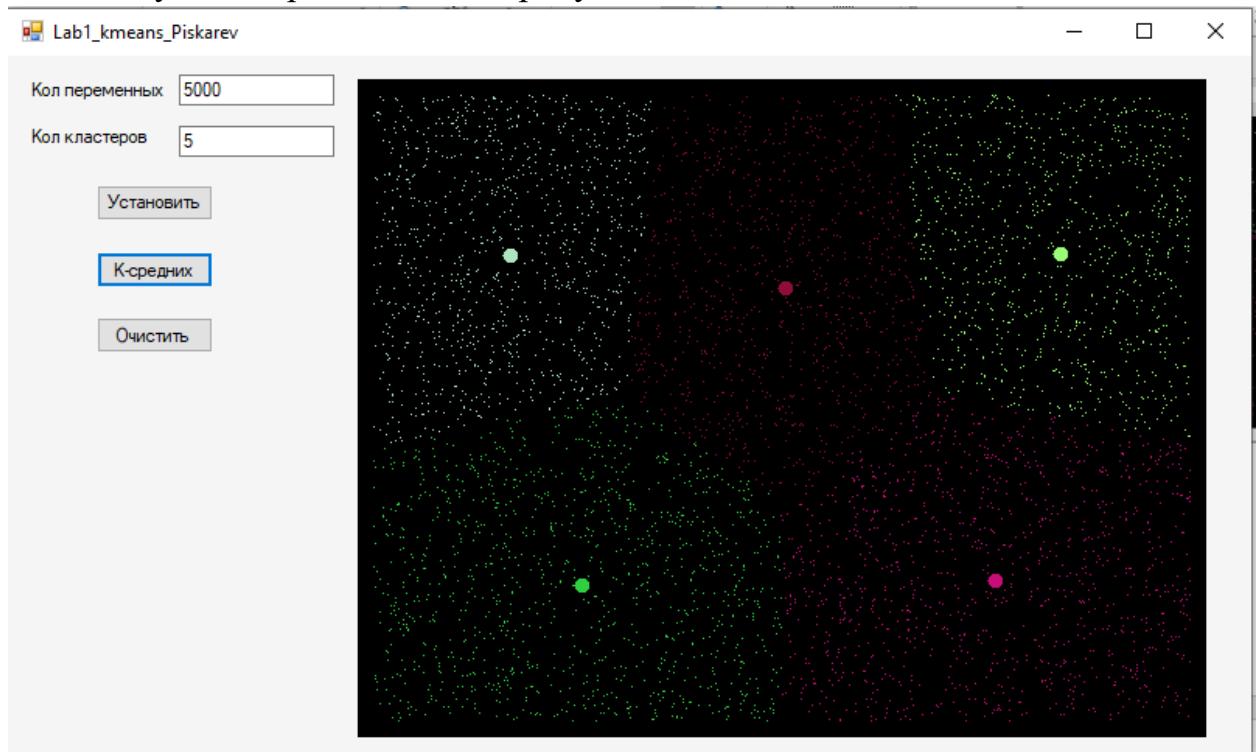


Рисунок 3

После завершения выполнения алгоритма, все кластеры становятся на свои места. Результат представлен на рисунке 4.



## ПРИЛОЖЕНИЕ

### Код программы

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Lab1
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Lab1
{
    public class KPoint
    {
        public int X { get; set; }

        public int Y { get; set; }

        public int Klaster { get; set; }
    }
}

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Lab1
{
    public class Kernel: ICloneable
    {
        public int Klaster { get; set; }

        public Color Color { get; set; }
    }
}
```

```

        public KPoint KPoint { get; set; }

        public bool Equals(Kernel b)
        {
            return this.KPoint.X == b.KPoint.X && this.KPoint.Y == b.KPoint.Y;
        }

        public Object Clone()
        {
            return new Kernel()
            {
                Color = this.Color,
                Klaster = this.Klaster,
                KPoint = new KPoint()
                {
                    X = this.KPoint.X,
                    Y = this.KPoint.Y,
                    Klaster = this.KPoint.Klaster,
                }
            };
        }
    }
}

```

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

```

```

namespace Lab1
{
    public class KMeans
    {
        private const int MIN_KPOINTS_COUNT = 10;
        private const int MAX_KPOINTS_COUNT = int.MaxValue;
        private const int MIN_KLASTER_COUNT = 2;
        private const int MAX_KLASTER_COUNT = 256;
        private const int DEFAULT_KLASTER = 0;
        private const int INDENT = 10;
        private Object lockObject = new Object();

        private int kpoints_count, k_count, xWidth, yHeight = 0;
        private Color[] colors;
        private KPoint[] kPoints;
        private Kernel[] OldKernels, NewKernels;
        private Random rand;
        private BufferedGraphics bufferedGraphics;
        public KMeans(BufferedGraphics bufferedGraphics)
        {
            this.bufferedGraphics = bufferedGraphics;
            rand = new Random();
        }

        public void KInit(int kpoint, int k, int width, int height)

```

```

{
    kpoints_count = kpoint;
    k_count = k;
    xWidth = width;
    yHeight = height;
    colors = new Color[k_count];
    OldKernels = new Kernel[k_count];
    NewKernels = new Kernel[k_count];
    kPoints = new KPoint[kpoints_count];

    for (var i = 0; i < kpoints_count; i++)
    {
        kPoints[i] = new KPoint();
        kPoints[i].X = INDENT + rand.Next(xWidth - 2 * INDENT);
        kPoints[i].Y = INDENT + rand.Next(yHeight - 2 * INDENT);
        kPoints[i].Klaster = DEFAULT_KLASTER;
    }

    for (var i = 0; i < k_count; i++)
    {
        colors[i] = NewColor();
        OldKernels[i] = NewKernels[i] = new Kernel();
        OldKernels[i].KPoint = NewKernels[i].KPoint =
kPoints[rand.Next(kpoints_count)];
        OldKernels[i].Color = NewKernels[i].Color = colors[i];
        OldKernels[i].Klaster = NewKernels[i].Klaster = i;
    }

    Draw();
}

public void ClearBox()
{
    bufferedGraphics.Graphics.Clear(Color.Black);
    bufferedGraphics.Render();
}

public void KmeansDo()
{
    bool isKlasterChanged = true;

    while (isKlasterChanged)
    {
        isKlasterChanged = false;
        Dictionary<Kernel, List<KPoint>> kernelPoints = new Dictionary<Kernel,
List<KPoint>>());
        for (var i = 0; i < k_count; i++)
        {
            OldKernels[i] = (Kernel)NewKernels[i].Clone();
            kernelPoints.Add(NewKernels[i], new List<KPoint>());
        }

        Parallel.ForEach(kPoints,
            new ParallelOptions() { MaxDegreeOfParallelism = kPoints.Count() },
            (kPoint) =>
            {
                KDistance[] kDistances = new KDistance[k_count];
                KDistance Kmin = null;
            }
        );
    }
}

```



```

        int count = 0;
        foreach (var kernel in kernelPoints.Keys)
        {
            kDistances[count] = new KDistance();
            kDistances[count].Kernel = kernel;
            kDistances[count].Distance = EvklidDistance(kPoint,
NewKernels[count].KPoint);

            if (Kmin == null)
            {
                Kmin = kDistances[count];
            }
            else if (Kmin.Distance > kDistances[count].Distance)
            {
                Kmin = kDistances[count];
            }
            count++;
        }

        kPoint.Klaster = Kmin.Kernel.Klaster;
        lock (lockObject)
        {
            kernelPoints[Kmin.Kernel].Add(kPoint);
        }
    });

    Parallel.ForEach(kernelPoints.Keys,
new ParallelOptions() { MaxDegreeOfParallelism =
kernelPoints.Keys.Count() },
(kernel) =>
    {
        int kpointNumber = -1;
        double minAvrgDistance = int.MaxValue;
        var listPoints = kernelPoints[kernel];
        for (var i = 0; i < listPoints.Count(); i++)
        {
            double sumSquareDistance = 0, avrgDistances;
            for (var j = 0; j < listPoints.Count(); j++)
            {
                sumSquareDistance += Math.Pow(EvklidDistance(listPoints[i],
listPoints[j]), 2);
            }
            avrgDistances = Math.Sqrt(sumSquareDistance);
            if (avrgDistances < minAvrgDistance)
            {
                minAvrgDistance = avrgDistances;
                kpointNumber = i;
            }
        }

        if (kpointNumber != -1)
        {
            NewKernels[kernel.Klaster].KPoint = listPoints[kpointNumber];
        }
    });

```

```

        for (int i = 0; i < k_count; i++)
        {
            if (!OldKernels[i].Equals(NewKernels[i]))
            {
                isKlasterChanged = true;
                break;
            }
        }

        Draw();
    }

    private void Draw()
    {
        bufferedGraphics.Graphics.Clear(Color.Black);
        for (var i = 0; i < kpoints_count; i++)
        {
            bufferedGraphics.Graphics.FillRectangle(new
SolidBrush(NewKernels[kPoints[i].Klaster].Color),
                kPoints[i].X, kPoints[i].Y, 1, 1);
        }

        for (var i = 0; i < k_count; i++)
        {
            bufferedGraphics.Graphics.FillEllipse(new
SolidBrush(NewKernels[i].Color),
                NewKernels[i].KPoint.X, NewKernels[i].KPoint.Y, 10, 10);
        }

        bufferedGraphics.Render();
    }

    private Color NewColor()
    {
        return Color.FromArgb(rand.Next(256), rand.Next(256), rand.Next(256));
    }

    private double EvklidDistance(KPoint a, KPoint b)
    {
        return Math.Sqrt(Math.Pow((a.X - b.X), 2) + Math.Pow((a.Y - b.Y), 2));
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Lab1
{
    public class KDistance
    {
        public Kernel Kernel { get; set; }

        public double Distance { get; set; }
    }
}

```

```

}

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Runtime.CompilerServices;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Lab1
{
    public partial class Form1 : Form
    {
        private Graphics graphics;
        private BufferedGraphics bufferedGraphics;
        private KMeans kMeans;
        private bool isSetValue;

        public Form1()
        {
            InitializeComponent();
            graphics = displayBox.CreateGraphics();
            bufferedGraphics = new BufferedGraphicsContext().Allocate(graphics, new
Rectangle(0, 0, displayBox.Width, displayBox.Height));
            kMeans = new KMeans(bufferedGraphics);
            isSetValue = false;
        }

        private void set_Click(object sender, EventArgs e)
        {
            kMeans.KInit(int.Parse(textBox1.Text), int.Parse(textBox2.Text),
displayBox.Width, displayBox.Height);
            isSetValue = true;
        }

        private void clear_Click(object sender, EventArgs e)
        {
            kMeans.ClearBox();
            isSetValue = false;
        }

        private void kmeans_Click(object sender, EventArgs e)
        {
            kMeans.KmeansDo();
        }
    }
}

```