Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Кафедра прикладной математики и кибернетики

Отчет

По лабораторной работе №1

Метод k-ближайших соседей.

Метод парзеновского окна с фиксированным h.

Выполнил:

студент гр. ИП-712

Алексеев С.В.

Работу проверил: Ассистент кафедры  
Морозова К.И.

Новосибирск 2020 г.

**Теория.**

**Метод k ближайших соседей**

**Теоретический базис:**

Пусть на множестве объектов задана функция расстояния . Существует целевая зависимость , значения которой известны только на объектах обучающей выборки . Множество классов конечно. Требуется построить алгоритм классификации , аппроксимирующий целевую зависимость на всём множестве .

Для произвольного объекта расположим элементы обучающей выборки в порядке возрастания расстояний до : , где через обозначается i-й сосед объекта u. Соответственно, ответ на i-м соседе объекта u есть . Таким образом, любой объект порождает свою перенумерацию выборки.

***Определение:*** *Метрический алгоритм классификации с обучающей выборкой относит объект к тому классу , для которого суммарный вес ближайших обучающих объектов максимален:*

*где весовая функция оценивает степень важности -го соседа для классификации объекта . Функция называется оценкой близости объекта к классу .*

***Алгоритм k ближайших соседей*** (k nearest neighbors, kNN).

Чтобы сгладить влияние выбросов, будем относить объект к тому классу, элементов которого окажется больше среди ближайших соседей :

При этот алгоритм совпадает с предыдущим, следовательно, неустойчив к шуму. При , наоборот, он чрезмерно устойчив и вырождается в константу.

Таким образом, крайние значения нежелательны. На практике оптимальное значение параметра определяют по критерию скользящего контроля с исключением объектов по одному (leave-one-out, LOO). Для каждого объекта проверяется, правильно ли он классифицируется по своим ближайшим соседям.

Заметим, что если классифицируемый объект не исключать из обучающей выборки, то ближайшим соседом всегда будет сам , и минимальное (нулевое) значение функционала будет достигаться при . Существует и альтернативный вариант метода kNN: в каждом классе выбирается ближайших к объектов, и объект относится к тому классу, для которого среднее расстояние до ближайших соседей минимально.

***Метод парзеновского окна***

Ещё один способ задать веса соседям — определить как функцию от расстояния , а не от ранга соседа . Введём функцию ядра , невозрастающую на . Положив в общей формуле, получим алгоритм.

Параметр называется шириной окна и играет примерно ту же роль, что и число соседей . «Окно» — это сферическая окрестность объекта радиуса , при попадании в которую обучающий объект «голосует» за отнесение объекта к классу . Мы пришли к этому алгоритму чисто эвристическим путём, однако он имеет более строгое обоснование в байесовской теории классификации, и, фактически, совпадает с методом парзеновского окна. Параметр можно задавать априори или определять по скользящему контролю. Зависимость , как правило, имеет характерный минимум, поскольку слишком узкие окна приводят к неустойчивой классификации; а слишком широкие — к вырождению алгоритма в константу.

Выбор ядра следует осуществлять из вариантов, представленных на рисунке:



Вариант «Метод парзеновского окна с фиксированным h + Q - квартическое».

**Код Программы.**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.IO;

using System.Diagnostics;

using System.Threading;

using System.Collections.Concurrent;

class Entry

{

public int MrotInHour, Salary, Class;

public Entry(string str)

{

var blocks = str.Split(',');

MrotInHour = Convert.ToInt32(blocks[0]);

Salary = Convert.ToInt32(blocks[1]);

Class = Convert.ToInt32(blocks[2]);

}

}

class Program

{

static Random random = new Random();

static Entry[] ReadAllData(string filename)

{

string[] lines = File.ReadAllLines(filename);

var result = lines.Skip(1).Select(line => new Entry(line)).ToArray();

return result;

}

static (Entry[], Entry[]) SplitData(Entry[] data)

{

int n = data.Length;

int nTraining = n \* 2 / 3;

int nTesting = n - nTraining;

var trainingData = new List<Entry>();

var testingData = new List<Entry>();

var grouppedByClass = new Dictionary<int, List<Entry>>();

foreach (var d in data)

{

if (!grouppedByClass.TryGetValue(d.Class, out var list))

{

list = new List<Entry>();

grouppedByClass[d.Class] = list;

}

list.Add(d);

}

int classI = 0;

int i = 0;

while (i < n)

{

classI = (classI + 1) % grouppedByClass.Keys.Count;

var @class = grouppedByClass.Keys.ElementAt(classI);

var group = grouppedByClass[@class];

if (group.Count == 0)

{

grouppedByClass.Remove(@class);

continue;

}

int index = random.Next(group.Count);

var item = group[index];

group.RemoveAt(index);

var target = i < nTraining ? trainingData : testingData;

target.Add(item);

i++;

}

return (trainingData.ToArray(), testingData.ToArray());

}

static int SqrDistance(Entry a, Entry b)

{

int dx = a.MrotInHour - b.MrotInHour;

int dy = a.Salary - b.Salary;

return dx \* dx + dy \* dy;

}

static float Distance(Entry a, Entry b)

{

return (float)Math.Sqrt(SqrDistance(a, b));

}

static float Kernel(float x)

{

if (x >= 1)

{

return 0;

}

float t = (1 - x \* x)\* (1 - x \* x);

return t \* t;

}

static Dictionary<Entry, Entry[]> SortDatasByDistance(Entry[] allData, Entry[] trainingData)

{

var result = new ConcurrentDictionary<Entry, Entry[]>();

int j = 1;

Action progressShow = async () =>

{

while (j < allData.Length - 1)

{

Console.Write($"Sorting by distance: {j}/{allData.Length} ");

Console.CursorLeft = 0;

await Task.Delay(10);

}

Console.Write($"Sorting by distance: {j}/{allData.Length} ");

Console.WriteLine();

};

progressShow();

Parallel.For(0, allData.Length, i =>

{

Interlocked.Increment(ref j);

var trainingDataSortedByDistanceToD = (Entry[])trainingData.Clone();

var d = allData[i];

Array.Sort(trainingDataSortedByDistanceToD, (a, b) => SqrDistance(a, d) - SqrDistance(b, d));

result[d] = trainingDataSortedByDistanceToD;

});

return new Dictionary<Entry, Entry[]>(result);

}

//first

static int Categorize(Entry item, Entry[] sortedTrainingData, float h)

{

float class0 = 0, class1 = 0;

foreach (var neighbor in sortedTrainingData)

{

if (neighbor == item) continue;

var measure = Kernel(Distance(item, neighbor) / h);

if (measure <= 0) break;

if (neighbor.Class == 0) class0 += measure;

else class1 += measure;

}

return class0 > class1 ? 0 : 1;

}

static int CountMatches(Entry[] data, Dictionary<Entry, Entry[]> sortedTrainingDatas, float k)

{

int count = 0;

foreach (var d in data)

{

var trainingData = sortedTrainingDatas[d];

if (d.Class == Categorize(d, trainingData, k))

count++;

}

return count;

}

static float FindBestK(Dictionary<Entry, Entry[]> sortedTrainingDatas, int maxK)

{

int maxMatches = 0;

object lockObj = new object();

float bestK = 0;

var anyOrderTrainingData = sortedTrainingDatas.First().Value;

int i = 1;

double r = 2;

var results = new Dictionary<float, float>();

for (float h = 0; h < 10; h += 0.1f)

{

int matches = CountMatches(anyOrderTrainingData, sortedTrainingDatas, h);

lock (lockObj)

{

if (matches > maxMatches)

{

maxMatches = matches;

bestK = h;

}

results[h] = (float)matches / anyOrderTrainingData.Length;

//results[k] = (float)Math.Pow((1 - (float)(Math.Pow(((float)matches / anyOrderTrainingData.Length), r))),r);

}

};

File.AppendAllText("results.txt", string.Join("\n", results.OrderBy(pair => pair.Key).Select(pair => pair.Value)) + "\n\n");

return bestK;

}

static void Main(string[] args)

{

Console.CursorVisible = false;

//for (int i = 0; i < 10; i++)

{

var allData = ReadAllData("data3.csv");

(var trainingData, var testingData) = SplitData(allData);

var sortedTrainingData = SortDatasByDistance(allData, trainingData);

float k = FindBestK(sortedTrainingData, 13);

Console.WriteLine("Best h: " + k);

Console.WriteLine("Training: " + (float)CountMatches(trainingData, sortedTrainingData, k) / trainingData.Length);

Console.WriteLine("Testing: " + (float)CountMatches(testingData, sortedTrainingData, k) / testingData.Length);

Console.WriteLine();

Console.WriteLine();

}

Console.ReadLine();

}

}

**Результат.**

