**KAUNO TECHNOLOGIJOS UNIVERSITETAS**

**INFORMATIKOS FAKULTETAS**

INTELEKTIKOS PAGRINDAI (P176B101)

**3 laboratorinio darbo ataskaita.**

Atliko:

IFF-4/1 gr. studentas

Mangirdas Kazlauskas

Priėmė:

Doc. Germanas Budnikas

**KAUNAS 2017**

1. Darbo užduotis

Sukurti programą SPAM failams klasifikuoti, panaudojant Bajeso teoremą. Ištirti priklausomybę tarp programoje naudojamų nustatymų ir klasifikatoriaus darbo efektyvumo (*false positive*, *true positive*).

1. Užduoties sprendimo programos kodas

Laboratorinio darbo užduotis buvo išspręsta naudojant JAVA programavimo kalbą. Programos kodas:

package intelektika\_lab3;

import java.io.File;

import java.util.ArrayList;

import java.util.Collections;

import java.util.HashMap;

import java.util.LinkedHashMap;

import java.util.LinkedList;

import java.util.List;

import java.util.Map;

import java.util.Scanner;

import java.util.regex.Pattern;

public class Intelektika\_lab3 {

private static final File SPAM\_DIRECTORY = new File("Training\_data/Spam");

private static final File NOT\_SPAM\_DIRECTORY = new File("Training\_data/NotSpam");

private static final File TESTING\_SPAM\_DIRECTORY = new File("Testing\_data/Spam");

private static final File TESTING\_NOT\_SPAM\_DIRECTORY = new File("Testing\_data/NotSpam");

private static final File ANALYSE\_SPAM\_DIRECTORY = new File("Analyse\_data/Spam");

private static final File ANALYSE\_NOT\_SPAM\_DIRECTORY = new File("Analyse\_data/NotSpam");

private static final Pattern REGEX = Pattern.compile("[^a-zA-Z0-9$'\"]");

private static int spamTokensCounter = 0;

private static int notSpamTokensCounter = 0;

private static int correctAnalyseFiles;

private static class Token{

private int spamCounter;

private int notSpamCounter;

private double spamProbability;

private double notSpamProbability;

private double probability;

public Token(){

this.spamCounter = 0;

this.notSpamCounter = 0;

this.probability = 0.0;

this.spamProbability = 0.0;

this.notSpamProbability = 0.0;

}

public Token(double probability){

this.spamCounter = 0;

this.notSpamCounter = 0;

this.probability = probability;

this.spamProbability = 0.0;

this.notSpamProbability = 0.0;

}

public void incSpamCounter(){

this.spamCounter++;

}

public void incNotSpamCounter(){

this.notSpamCounter++;

}

public void setProbability(double probability){

this.probability = probability;

}

public void setSpamProbability(double probability){

this.spamProbability = probability;

}

public void setNotSpamProbability(double probability){

this.notSpamProbability = probability;

}

public int getSpamCounter(){

return this.spamCounter;

}

public int getNotSpamCounter(){

return this.notSpamCounter;

}

public double getProbability(){

return this.probability;

}

public double getSpamProbability(){

return this.spamProbability;

}

public double getNotSpamProbability(){

return this.notSpamProbability;

}

}

private static void readTokensFromFiles(Map<String, Token> tokens){

String word;

for (File file : SPAM\_DIRECTORY.listFiles()){

try (Scanner scanner = new Scanner(file)) {

while (scanner.hasNext()){

scanner.useDelimiter(REGEX);

word = scanner.next();

word = word.toLowerCase();

if (!word.equals("")){

if(tokens.containsKey(word)){

Token token = tokens.get(word);

token.incSpamCounter();

tokens.put(word, token);

}

else{

Token token = new Token();

token.incSpamCounter();

tokens.put(word, token);

spamTokensCounter++;

}

}

}

} catch (Exception ex) {

System.out.println(ex);

System.exit(1);

}

}

for (File file : NOT\_SPAM\_DIRECTORY.listFiles()){

try (Scanner scanner = new Scanner(file)) {

while (scanner.hasNext()){

scanner.useDelimiter(REGEX);

word = scanner.next();

word = word.toLowerCase();

if (!word.equals("")){

if(tokens.containsKey(word)){

Token token = tokens.get(word);

token.incNotSpamCounter();

tokens.put(word, token);

}

else{

Token token = new Token();

token.incNotSpamCounter();

tokens.put(word, token);

notSpamTokensCounter++;

}

}

}

} catch (Exception ex) {

System.out.println(ex);

System.exit(1);

}

}

}

private static void analyseFile(Map<String, Token> tokens, Map<String, Double> probabilities, File file){

String word;

try (Scanner scanner = new Scanner(file)) {

while (scanner.hasNext()){

scanner.useDelimiter(REGEX);

word = scanner.next();

word = word.toLowerCase();

if (!word.equals("")){

if(tokens.containsKey(word)){

Token token = tokens.get(word);

probabilities.put(word, token.getProbability());

}

else{

probabilities.put(word, 0.4);

}

}

}

} catch (Exception ex) {

System.out.println(ex);

System.exit(1);

}

}

private static void calculateTestingFilesProbabilities(Map<String, Token> tokens, List<Double> spamFilesProbabilities, List<Double> notSpamFilesProbabilities, int testingTokensCount){

for (File file : TESTING\_SPAM\_DIRECTORY.listFiles()){

Map<String, Double> newFileProbabilities = new HashMap<>();

analyseFile(tokens, newFileProbabilities, file);

Map<String, Double> sortedMap = sortMapByValue(newFileProbabilities);

spamFilesProbabilities.add(calculateFileSpamProbability(sortedMap, testingTokensCount));

}

Collections.sort(spamFilesProbabilities);

Collections.reverse(spamFilesProbabilities);

for (File file : TESTING\_NOT\_SPAM\_DIRECTORY.listFiles()){

Map<String, Double> newFileProbabilities = new HashMap<>();

analyseFile(tokens, newFileProbabilities, file);

Map<String, Double> sortedMap = sortMapByValue(newFileProbabilities);

notSpamFilesProbabilities.add(calculateFileSpamProbability(sortedMap, testingTokensCount));

}

Collections.sort(notSpamFilesProbabilities);

Collections.reverse(notSpamFilesProbabilities);

}

private static void calculateAnalyseFilesProbabilities(Map<String, Token> tokens, List<Double> spamFilesProbabilities, List<Double> notSpamFilesProbabilities, int testingTokensCount){

for (File file : ANALYSE\_SPAM\_DIRECTORY.listFiles()){

Map<String, Double> newFileProbabilities = new HashMap<>();

analyseFile(tokens, newFileProbabilities, file);

Map<String, Double> sortedMap = sortMapByValue(newFileProbabilities);

spamFilesProbabilities.add(calculateFileSpamProbability(sortedMap, testingTokensCount));

}

Collections.sort(spamFilesProbabilities);

Collections.reverse(spamFilesProbabilities);

for (File file : ANALYSE\_NOT\_SPAM\_DIRECTORY.listFiles()){

Map<String, Double> newFileProbabilities = new HashMap<>();

analyseFile(tokens, newFileProbabilities, file);

Map<String, Double> sortedMap = sortMapByValue(newFileProbabilities);

notSpamFilesProbabilities.add(calculateFileSpamProbability(sortedMap, testingTokensCount));

}

Collections.sort(notSpamFilesProbabilities);

Collections.reverse(notSpamFilesProbabilities);

}

private static void calculateTokensProbabilities(Map<String, Token> tokens){

tokens.forEach((t, u) -> {

double pws = (u.getSpamCounter() == 0) ? 0 : (double)u.getSpamCounter()/spamTokensCounter;

double pwh = (u.getNotSpamCounter() == 0) ? 0 : (double)u.getNotSpamCounter()/notSpamTokensCounter;

u.setSpamProbability(pws);

u.setNotSpamProbability(pwh);

if (pws == 0 && pwh > 0){

u.setProbability(0.01);

}

else if(pws > 0 && pwh == 0){

u.setProbability(0.99);

}

else{

u.setProbability((double)(pws/(pws + pwh)));

}

});

}

private static double calculateFileSpamProbability(Map<String, Double> tokens, int testingTokensCount){

double numerator = 1;

double denominatorSubstraction = 1;

if(tokens.size() >= testingTokensCount){

List<Map.Entry<String, Double>> list =

new LinkedList<>(tokens.entrySet());

testingTokensCount /= 2;

for (int i = 0; i < testingTokensCount; i++){

numerator \*= list.get(i).getValue();

denominatorSubstraction \*= 1 - list.get(i).getValue();

}

for (int i = list.size()-1; i > list.size() - (testingTokensCount + 1); i--){

numerator \*= list.get(i).getValue();

denominatorSubstraction \*= 1 - list.get(i).getValue();

}

}

else{

for(Map.Entry<String, Double> entry : tokens.entrySet()){

numerator \*= entry.getValue();

denominatorSubstraction \*= 1 - entry.getValue();

}

}

return (double)numerator/(numerator + denominatorSubstraction);

}

private static <K, V> void printMap(Map<K, V> map) {

for (Map.Entry<K, V> entry : map.entrySet()) {

System.out.println("Key : " + entry.getKey()

+ "; Value : " + entry.getValue());

}

}

private static <K, V extends Comparable<? super V>> Map<K, V> sortMapByValue(Map<K, V> unsortMap) {

List<Map.Entry<K, V>> list =

new LinkedList<>(unsortMap.entrySet());

Collections.sort(list, (Map.Entry<K, V> o1, Map.Entry<K, V> o2) -> (o1.getValue()).compareTo(o2.getValue()));

Map<K, V> result = new LinkedHashMap<>();

list.forEach((entry) -> {

result.put(entry.getKey(), entry.getValue());

});

return result;

}

private static double calculateThreshold(List<Double> notSpamFilesProbabilities){

return (double)Math.ceil(notSpamFilesProbabilities.get(0) \* 100)/100;

}

private static double calculateTrueOrFalsePositiveFilesPercentage(List<Double> filesProbabilities, double threshold){

int counter = 0;

for (double i : filesProbabilities) if(i > threshold) counter++;

return (double)Math.round((double)counter/filesProbabilities.size()\*10000)/100;

}

private static void calculateThresholdsWithDifferentTokensCount(Map<Integer, Double> thresholds){

int[] tokenCounts = {2, 4, 8, 12, 16, 24, 32};

System.out.println("Learning... ");

Map<String, Token> tokens = new HashMap<>();

readTokensFromFiles(tokens);

calculateTokensProbabilities(tokens);

for (int i : tokenCounts){

List<Double> spamFilesProbabilities = new ArrayList<>();

List<Double> notSpamFilesProbabilities = new ArrayList<>();

List<Double> spamAnalyseFilesProbabilities = new ArrayList<>();

List<Double> notSpamAnalyseFilesProbabilities = new ArrayList<>();

System.out.println("Testing... ");

calculateTestingFilesProbabilities(tokens, spamFilesProbabilities, notSpamFilesProbabilities, i);

System.out.println("Tokens count: " + i);

double threshold = calculateThreshold(notSpamFilesProbabilities);

System.out.println("Threshold value: " + threshold);

System.out.println("Analysing... ");

calculateAnalyseFilesProbabilities(tokens, spamAnalyseFilesProbabilities, notSpamAnalyseFilesProbabilities, i);

double truePositiveSpamFilesPercentage = calculateTrueOrFalsePositiveFilesPercentage(spamAnalyseFilesProbabilities, threshold);

double falsePositiveSpamFilesPercentage = calculateTrueOrFalsePositiveFilesPercentage(notSpamAnalyseFilesProbabilities, threshold);

System.out.println("True positive files percentage: " + truePositiveSpamFilesPercentage + "%");

System.out.println("False positive files percentage: " + falsePositiveSpamFilesPercentage + "%");

System.out.println("Analysed files probabilities and their status (SPAM or NOT SPAM):");

System.out.println("Files from SPAM directory:");

correctAnalyseFiles = 0;

for(double j : spamAnalyseFilesProbabilities){

System.out.println(j + " " + (j > threshold ? "SPAM" : "NOT SPAM"));

if (j > threshold) correctAnalyseFiles++;

}

System.out.println();

System.out.println("Files from NOT SPAM directory:");

for(double j : notSpamAnalyseFilesProbabilities){

System.out.println(j + " " + (j > threshold ? "SPAM" : "NOT SPAM"));

if (j <= threshold) correctAnalyseFiles++;

}

System.out.println("Bendras programos tikslumas:" + (double)Math.round((double)correctAnalyseFiles/(spamFilesProbabilities.size() + notSpamFilesProbabilities.size())\*10000)/100 + "%");

System.out.println("-----------------------------------------------");

}

}

public static void main(String[] args) {

Map<Integer, Double> thresholds = new HashMap<>();

calculateThresholdsWithDifferentTokensCount(thresholds);

}

}

# Programos veikimo tyrimas

Tyrimo metu SPAM ir NE SPAM failai buvo suskirstyti į tris dalis: programos apmokymo rinkinys (70% visų SPAM ir 70% visų NE SPAM failų), programos testavimo rinkinys (20% visų SPAM ir 20% visų NE SPAM failų) bei programos analizės rinkinys (po 10% likusių visų SPAM ir NE SPAM failų).

Failų spamiškumo riba tyrimo metu buvo nustatoma kelis kartus, naudojant skirtingą leksemų skaičių failų tikimybių nustatymui. Tyrimo metu failų tikimybės buvo nustatomos su tokiais leksemų kiekiais: 2, 4, 8, 12, 16, 24, 32.

Gautus tyrimo rezultatus galime sugrupuoti į tokią lentelę:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Leksemų kiekis | Failų spamiškumo riba (tikimybė) | True positive failų kiekis (%) | False positive failų kiekis (%) | Bendras programos tikslumas |
| 2 | 0,76 | 34,78 | 0,34 | 39,49 |
| 4 | 0,53 | 71,01 | 0,68 | 45,15 |
| 8 | 0,78 | 71,74 | 0 | 45,5 |
| 12 | 0,87 | 65,94 | 0 | 44,57 |
| 16 | 0,75 | 74,64 | 0 | 45,96 |
| 24 | 0,5 | 73,91 | 0,34 | 45,73 |
| 32 | 0,5 | 71,74 | 0,34 | 45,38 |

Remiantis lentelės duomenimis, sudaromas failų spamiškumo ribos priklausomybes nuo leksemų kiekio grafikas:

Iš grafiko matyti, kad leksemų kiekio kitimas daro įtaką failų spamiškumo ribos kitimui.

Kiek reikšmingesnė tyrimui diagrama, remiantis lentelės duomenimis – SPAM failų true positive ir false positive kiekis, kintant maksimaliai savo spamiškumo tikimybės reikšme nutolusių failo leksemų kiekiui.

Iš diagramos matome, kad tiksliausiai SPAM failas nustatytas buvo tada, kai leksemų kiekis buvo 16 (74,64% analizuotų SPAM failų buvo nustatyti kaip SPAM teisingai). Prasčiausiai prognozė buvo atliekama, kai naudotų leksemų skaičius buvo mažiausias. Iš diagramos duomenų galima įžvelgti, kad aiškus programos SPAM failų analizės tikslumo kitimas buvo tik pradžioje, toliau, didinant naudojamų leksemų skaičių failo spamiškumo tikimybei apskaičiuoti, programos veikimo teisingumas išliko panašus.

Kalbant apie false positive failų kiekį, matome, kad labai tiksliai programa veikė, kai apskaičiuojant failų tikimybes buvo naudojamos 8, 12 arba 16 leksemų (nė vienas NE SPAM failas nebuvo priskirtas prie SPAM). Prasčiausiai programa NE SPAM failus atpažino tada, kai failų tikimybių apskaičiavimui buvo naudojamos 4 leksemos.

Iš diagramos taip pat matome, kad bendram programos tikslumui taip pat įtaką daro imamų leksemų kiekis, tačiau dauguma atvejų, bendras programos tikslumas išlieka stabilus (apie 45%).

# Išvados

Laboratorinio darbo metu buvo susipažinta su Bajeso teorema bei, remiantis ja, buvo sukurta programa, kuri klasifikuoja failus į SPAM ir NE SPAM. Iš programos veikimo rezultatų bei atliktos analizės galima teigti, kad programos veikimui įtaką daro ne vienas programoje naudojamas parametras, pvz., naudojamų leksemų kiekis failo tikimybei apskaičiuoti, failo spamiškumo riba ir pan. Iš atliktos analizės rezultatų galime teigti, kad programa sugeba klasifikuoti failus į SPAM ir NE SPAM, nors jos tikslumas, aišku, nėra maksimalus, bet mažas false positive failų kiekis rodo tai, kad programa veikia gana tiksliai, atsižvelgiant į tai, kad NE SPAM failų nepriskiria prie SPAM.