گزارش درس مبانی داده‌کاوی

پروژه جایگزین پایان‌ترم

استاد درس:

دکتر رضا رمضانی

تهیه‌کننده:

محسن دهباشی
۹۵۳۶۱۱۱۳۳۰۲۸

تابستان ۹۹

فهرست مطالب

مقدمه

در این پروژه قصد داریم یک طبقه بند[1] طراحی کنیم که بتواند مؤلف[2] کد مرجع[3] را تشخیص دهد[4]. یعنی داده‌های آموزشی کدهای زده شده به زبان برنامه نویسی مشخصی هستند که برچسب آن‌ها مؤلف آن کد است. زبان برنامه‌نویسی در نظر گرفته شده زبان C++ و محیط پروژه محیط مسابقات برنامه نویسی ACM در نظر گرفته شده است. بنابراین مجموعه داده‌ی Google Code Jam Dataset 2019 شامل ۳۹۸۱۸۰ کد مرجع برای آموزش طبقه بند آماده شده است. روال کلی کار بدین صورت است که ابتدا باید به عنوان پیش پردازش با استخراج مجموعه ویژگی‌های خاصی از یک کد، استایل کدزنی[5] مؤلف را بشناسیم. سپس باید هر کد را به یک بردار[6] در فضای اقلیدسی[7] نگاشت کنیم، برای این کار می‌توانیم از دو روش AST Similarity و TF-IDF استفاده کنیم. در مرحله بعد می‌توان بردارهای نماینده را به الگوریتم‌های معروف طبقه بندی مانند KNN، CNN، SVM، Random Forest و ... داد تا مرحله‌ی یادگیری را انجام دهند. برای تکمیل فرآیند طبقه بندی باید داده‌های آزمون را نیز به بردارهایی در فضای اقلیدسی نگاشت کنیم و به طبقه بند بدهیم، سپس از روی برچسب‌های پیش بینی شده توسط طبقه بند و برچسب‌های واقعی، دقت طبقه بند را محاسبه کنیم. بدیهی است که برای استفاده از مدل یادگیری شده و طبقه بندی کدهای مرجع ورودی، ابتدا باید با پیش پردازش آن‌ها را به بردار نگاشت کنیم.

[1] Classifier
[2] Author
[3] Source Code
[4] Code Authorship Atrribution
[5] Coding Style
[6] Vector
[7] Euclidean Space

مرحله‌ی اول: آماده سازی مجموعه‌ی داده

در این مرحله لازم است همه‌ی کدهای مرجع که در فایل csv ذخیره شده اند، استخراج شوند. بدین منظور از قطعه کد پایتون زیر (preprocess.py) می‌توان استفاده کرد.

```
import os, pandas

if __name__ == "__main__":
    data = pandas.read_csv('codejam2019.csv')
    for row in data.iterrows():
        if not row[1]['file'].endswith('.CPP'):
            continue
        if not os.path.exists(row[1]['username']):
            os.mkdir(row[1]['username'])
        file = open('%s/%s' % (row[1]['username'], row[1]['file']), 'w')
        file.write(row[1]['flines'])
        file.close()
```

پس از اجرای این قطعه کد، همه کدهای مرجع به تفکیک مؤلف پوشه‌بندی می‌شوند.

مرحله‌ی دوم: پیش پردازش

همانطور که گفته شد در این مرحله باید استایل کد زنی مؤلف را شناسایی کنیم. پس از تحلیل اجمالی مجموعه‌ی داده عوامل مهمی که بر استایل کد زنی مؤلف موثرند، شناسایی شد. این عوامل ویژگی‌های هستند که برای یک مؤلف به عادت و رعایت الگو تبدیل شده اند. به همین علت در کدهایی که از یک مؤلف داریم پرتکرار بودن این الگوها[8] را مشاهده می‌کنیم. در ادامه به بررسی این ویژگی‌ها می‌پردازیم.

۱- نحوه‌ی کامنت گذاری در کد

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n;
    cin >> n;
    // This variable will hold factorial(n):
    int fact = 1;
    for(int i = 2; i <= n; i++)
        fact *= i;
    cout << fact << endl;
}
```

[8] Frequent Patterns

$$Comment\ Point\ =\ 0$$

```cpp
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n;
    cin >> n;
    int fact = 1; // This variable will hold factorial(n)
    for(int i = 2; i <= n; i++)
        fact *= i;
    cout << fact << endl;
}
```

$$Comment\ Point\ =\ 1$$

در بررسی این ویژگی اگر یک کد مانند نمونه کد اول کامنت گذاری شده باشد ارزش صفر داشته و اگر مانند نمونه کد دوم کامنت گذاری شده باشد ارزش یک دارد.

پیاده سازی (در زبان کاتلین):

```kotlin
class CodeWrapper(private val codeText: String, private val codeLines: List<String>,
          val name: String, val author: String){

  private var firstCommentType = 0
  private var secondCommentType = 0

  init {
    codeLines.forEach { line ->
      val trim = line.trim()

      if(trim.startsWith("//"))
        firstCommentType++
      else if(trim.contains("//"))
        secondCommentType++
    }
  }

  val commentPoint: Int
  get() = if(firstCommentType >= secondCommentType) 0 else 1

}
```

۲- نحوه‌ی استفاده از } در کد

```cpp
 #include <bits/stdc++.h>
using namespace std;

int main()
{
    int n;
    cin >> n;
    int fact = 1;
    while(n > 1)
    {
        fact *= n;
        n--;
    }
    cout << fact << endl;
}
```

$$Curly\ Braces\ Point\ =\ 0$$

```cpp
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n;
    cin >> n;
    int fact = 1;
    while(n > 1){
        fact *= n;
        n--;
    }
    cout << fact << endl;
}
```

$$Curly\ Braces\ Point\ =\ 1$$

در بررسی این ویژگی اگر یک کد مانند نمونه کد اول باشد ارزش صفر داشته و اگر مانند نمونه کد دوم باشد ارزش یک دارد.

پیاده سازی:

```
import org.antlr.v4.runtime.CharStreams

class CodeWrapper(private val codeText: String, private val codeLines: List<String>,
            val name: String, val author: String){

   private val tokens = CPP14Lexer(CharStreams.fromString(codeText)).allTokens.map { it.text }

   private var firstCurlyBracesType = 0
   private var secondCurlyBracesType = tokens.count { it == "{" }

   init {
      codeLines.forEach { line ->
         val trim = line.trim()

         if(trim.startsWith("{")){
            firstCurlyBracesType++
            secondCurlyBracesType--
         }
      }
   }

   val curlyBracesPoint: Int
   get() = if(firstCurlyBracesType >= secondCurlyBracesType) 0 else 1

}
```

۳- استایل توابع ورودی/خروجی در کد

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n;
    scanf("%d", &n);
    printf("%d", n / 2);
}
```

$$IO\ Point\ =\ 0$$

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n;
    cin >> n;
    cout << n / 2;
}
```

$$IO\ Point\ =\ 1$$

اگر در کدی از scanf و printf استفاده شده باشد ارزش صفر داشته و اگر از cin و cout استفاده شده باشد ارزش یک دارد.

پیاده سازی:

```
import org.antlr.v4.runtime.CharStreams

class CodeWrapper(private val codeText: String, private val codeLines: List<String>,
            val name: String, val author: String){

   private val tokens = CPP14Lexer(CharStreams.fromString(codeText)).allTokens.map { it.text }

   private val firstIOType = tokens.count { it == "scanf" || it == "printf" }
   private val secondIOType = tokens.count { it == "cin" || it == "cout" }

   val IOPoint: Int
   get() = if(firstIOType >= secondIOType) 0 else 1

}
```

۴- موارد دیگر

موارد بیشتر دیگری را می‌توان در این قسمت دخیل کرد که برای ساده‌سازی پروژه از آن‌ها صرف نظر شده است. برای مثال برنامه نویسی را فرض کنید که از کاما در کد زیاد استفاده می‌کند و یا عادت دارد با استفاده از کروشه، متغیرهایش را مقدار دهی اولیه کند. بدیهی است که تحلیل چنین ویژگی‌هایی تاثیر به شدت بالایی در تعیین استایل کد زنی دارند.

یک نمونه کد از برنامه نویس مذکور:

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int a{1}, b{2}, c{3}, d{4};
    int n;
    cin >> n;
    if(n == 1)
        a = 2, b = 1, c = 4, d = 3;
    else if(n == 2)
        a = 4, b = 3, c = 2, d = 1;
}
```

مرحله‌ی سوم: نگاشت کد به بردار

در این مرحله باید کدها را به بردارهایی در فضای اقلیدسی نگاشت کنیم تا برای الگوریتم یادگیری آماده شوند. اگر مجموعه‌ی کدها X باشد به زبان ریاضی به تابعی مانند f نیاز داریم که:

$$f: X \to \mathbb{R}^d$$

$$\forall\, x \in X$$

$$\exists\, y \in \mathbb{R}^d$$

$$f(x) = y$$

که در آن y یک بردار d بعدی است. در ادامه به بررسی دو الگوریتم نگاشت یعنی AST Similarity و TF-IDF می‌پردازیم.

۱- روش AST Similarity

هدف ما در این روش شناسایی روال‌های تکراری دو کد است (حدس می‌زنیم یک مؤلف از روال‌های ساده تکراری[9] در بسیاری از کدهای مرجع خودش استفاده می‌کند). اگر بتوانیم شباهت هر دو کد را بسنجیم، فاصله‌ی آن دو از هم را نیز می‌توانیم بدست آوریم، آنگاه داده‌ها برای یادگیری مدل طبقه بند آماده هستند. در این روش به کامپایلر C++14 نیاز داریم. بدین منظور می‌توان از گرامر، لگزر[10] و پارسر[11] C++14 پیاده سازی شده توسط antrl4 در زبان کاتلین استفاده کرد. برای شناسایی روال‌های تکراری دو معیار برای سنجش شباهت داریم که بهتر است از هردوی آن‌ها استفاده کنیم. معیار اول که یک معیار لغوی[12] است، Longest Common Sub-Lexemes نام دارد. برای مقایسه‌ی دو کد تحت این معیار ابتدا باید کدها را توسط لگزر سی پلاس پلاس tokenize کنیم و الگوریتم برنامه‌نویسی پویا[13]ی Longest Common Subsequence را روی واژه‌ها[14]ی tokenize شده‌ی دو کد اجرا کنیم. مقداری که به دست می‌آید میزان شباهت دو کد از لحاظ وجود روال‌های تکراری لغوی[15]می‌باشد. همچنین برای تبدیل مقدار شباهت به فاصله lexical distance را تعریف می‌کنیم که عدد حقیقی‌ای در بازه‌ی صفر تا یک است. هرچه این عدد کمتر باشد دو کد مرجع بیشتر به هم شبیه هستند.

[9] Simple Frequent Procedures
[10] Lexer
[11] Parser
[12] Lexical
[13] Dynamic Programming
[14] Lexemes
[15] Frequent Lexical Procedures

$$lexical\ distance = 1 - \frac{Longest\ Common\ Sub - Lexemes}{\min\left(|first\ code\ tokens|,\ |second\ code\ tokens|\right)}$$

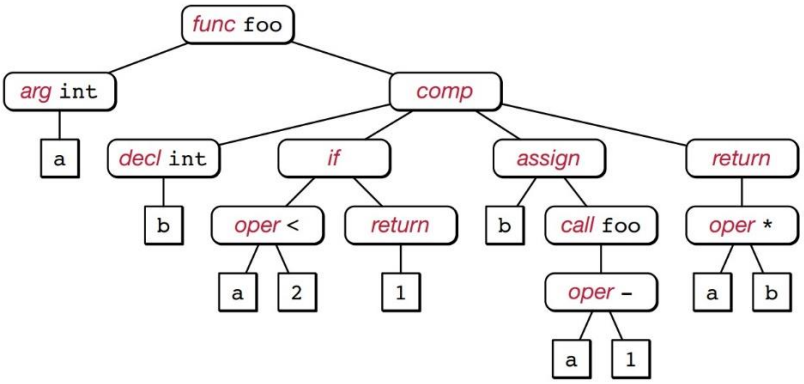
پیاده سازی:

```
import org.antlr.v4.runtime.CharStreams
import kotlin.math.max
import kotlin.math.min

class CodeWrapper(private val codeText: String, private val codeLines: List<String>,
            val name: String, val author: String){

   private val tokens = CPP14Lexer(CharStreams.fromString(codeText)).allTokens.map { it.text }

   infix fun lexicalDistance(other: CodeWrapper): Double{
      val dp = Array(this.tokens.size){IntArray(other.tokens.size)}
      for(i in this.tokens.indices)
         for(j in other.tokens.indices){
            if(i > 0)
               dp[i][j] = max(dp[i][j], dp[i-1][j])
            if(j > 0)
               dp[i][j] = max(dp[i][j], dp[i][j-1])
            if(this.tokens[i] == other.tokens[j])
               dp[i][j] = max(dp[i][j], 1 + if(i > 0 && j > 0) dp[i-1][j-1] else 0)
         }
      val ans = dp[this.tokens.size-1][other.tokens.size-1]
      return 1 - ans.toDouble() / min(this.tokens.size, other.tokens.size)
   }

}
```

معیار دوم که یک معیار نحوی[16]است، AST Similarity نام دارد. برای مقایسه‌ی دو کد تحت این معیار باید از درخت پارس گرامر[17] کد مرجع یعنی Abstract Syntax Tree استفاده کنیم. به عنوان مثال کد زیر و درخت پارس آن را در نظر بگیرید.

```
1  int foo(int a){
2    int b;
3    if (a < 2)       // base case
4      return 1;
5    b = foo(a - 1); // recursion
6    return a * b;
7  }
```

[16] Syntactical
[17] Grammar Parse Tree



در این درخت، هر گره[18] یک نوع دارد و یک مقدار که به ترتیب با رنگ‌های قرمز و مشکی در شکل مشخص شده‌اند. در گره‌های این درخت، ویژگی حائز اهمیت نوع گره است و نه مقدار آن (برای مثال فرض کنید مؤلف از نام‌های متغیر و تابع مختلفی استفاده کند اما از یک روال الگوریتمی تکراری استفاده کند، بدیهی است که این موضوع نباید در شناسایی روال‌های تکراری نحوی[19] تاثیر بگذارد). بنابراین برای سنجش میزان شباهت دو کد، ابتدا باید مقادیر را در درخت‌های پارس حذف کنیم و بعد از آن تعداد زیر درخت‌های کاملا یکسان دو درخت پارس را با استفاده از برنامه نویسی پویا بدست آوریم. همچنین برای تبدیل مقدار شباهت به فاصله syntactical distance را تعریف می‌کنیم که عدد حقیقی‌ای در بازه‌ی صفر تا یک است. هرچه این عدد کمتر باشد دو کد مرجع بیشتر به هم شبیه هستند.

$$syntactical\ distance = 1 - \ scale(Number\ of\ Common\ Sub - trees)$$

$$scale(Number\ of\ Common\ Sub - trees) = \frac{Number\ of\ Common\ Sub - trees}{\max\left(|first\ tree|,\ |second\ tree|\right)}$$

پیاده سازی:

```
class CPP14TreeMaker: CPP14BaseListener() {

  private val id = ParseTreeProperty<Int>()
  val nodeTypes = ArrayList<String>()
  val nodeChilds = ArrayList<ArrayList<Int>>()
  val nodeCount: Int
  get() = nodeTypes.size

  override fun enterEveryRule(ctx: ParserRuleContext?) {
    super.enterEveryRule(ctx)
    id.put(ctx, nodeCount)
    nodeTypes.add(ctx!!::class.toString())
    nodeChilds.add(ArrayList())
    if(ctx.getParent() != null)
      nodeChilds[id.get(ctx.getParent())].add(id[ctx])
  }}
```

[18] Node
[19] Frequent Syntactical Procedures

```
import org.antlr.v4.runtime.CharStreams
import org.antlr.v4.runtime.CommonTokenStream
import org.antlr.v4.runtime.tree.ParseTreeWalker
import kotlin.math.max
import kotlin.math.min

class CodeWrapper(private val codeText: String, private val codeLines: List<String>,
                  val name: String, val author: String){

    private val tree = CPP14TreeMaker()

    init {

        ParseTreeWalker().walk(tree,
CPP14Parser(CommonTokenStream(CPP14Lexer(CharStreams.fromString(codeText)))).translationunit())

    }

    infix fun commonSubtrees(other: CodeWrapper): Int{
        val dp = Array(this.tree.nodeCount){IntArray(other.tree.nodeCount){-1}}
        var commonSubtrees = 0
        for(i in 0 until this.tree.nodeCount)
            for(j in 0 until other.tree.nodeCount)
                if(cdp(dp, i, j, other) == 1)
                    commonSubtrees++
        return commonSubtrees
    }

    private fun cdp(dp: Array<IntArray>, i: Int, j: Int, other: CodeWrapper): Int{
        if(dp[i][j] != -1)
            return dp[i][j]
        if(this.tree.nodeTypes[i] != other.tree.nodeTypes[j]){
            dp[i][j] = 0
            return 0
        }
        if(this.tree.nodeChilds[i].size != other.tree.nodeChilds[j].size){
            dp[i][j] = 0
            return 0
        }
        for(c in 0 until this.tree.nodeChilds[i].size)
            if(cdp(dp, this.tree.nodeChilds[i][c], other.tree.nodeChilds[j][c], other) == 0){
                dp[i][j] = 0
                return 0
            }
        dp[i][j] = 1
        return 1
    }
}
```

پس از بررسی دو معیار مذکور، نوبت به بدست آوردن فاصله‌ی اقلیدسی دو کد می‌رسد. لازم بذکر است که اگر چه در این روش برداری تولید نمی‌شود اما در آن فاصله‌ی اقلیدسی بین هر دو کد محاسبه می‌شود که برای الگوریتم‌های یادگیری کافی است (گوئی بردارهای انتزاعی[20]ای وجود داشته اند که فاصله‌ی بین هر دوی آن‌ها را حساب کرده‌ایم). برای بدست آوردن فاصله‌ی اقلیدسی بین دو کد مرجع x_1 و x_2 می‌توان از ظابطه‌ی زیر کرد.

$$a = |Comment\ Point(x_1) - Comment\ Point(x_2)|$$

$$b = |Curly\ Braces\ Point(x_1) - Curly\ Braces\ Point(x_2)|$$

$$c = |IO\ Point(x_1) - IO\ Point(x_2)|$$

$$d = lexical\ distance(x_1, x_2)$$

$$e = syntactical\ distance(x_1, x_2)$$

$$distance(x_1, x_2) = \sqrt{a^2 + b^2 + c^2 + d^2 + e^2}$$

پیاده سازی (main پروژه):

```
import java.io.File
import kotlin.math.abs
import kotlin.math.max
import kotlin.math.sqrt

val wrappers = ArrayList<CodeWrapper>()
val distances = HashMap<Pair<Int, Int>, MutableList<Double>>()

fun main() {
    val datasetFolder = File("dataset")
    datasetFolder.list()!!.forEach{ author ->
        val authorFolder = File("dataset/$author")
        authorFolder.list()!!.forEach { code ->
            val codeFile = File("dataset/$author/$code")
            wrappers.add(CodeWrapper(codeFile.readText(), codeFile.readLines(), code, author))
        }
    }
    var maxCommonSubtrees = 0.0
    for(i in wrappers.indices)
        for(j in i+1 until wrappers.size){
            distances[i to j] = mutableListOf(
                abs(wrappers[i].commentPoint - wrappers[j].commentPoint).toDouble(),
                abs(wrappers[i].curlyBracesPoint - wrappers[j].curlyBracesPoint).toDouble(),
                abs(wrappers[i].IOPoint - wrappers[j].IOPoint).toDouble(),
                wrappers[i] lexicalDistance wrappers[j],
                (wrappers[i] commonSubtrees wrappers[j]).toDouble().also {
                    maxCommonSubtrees = max(maxCommonSubtrees, it)
                }
```

[20] Abstract

```
        )
    }
  distances.forEach { (t, u) ->
    u[u.size-1] = 1.0 - u.last() / maxCommonSubtrees
    val distance = sqrt(u.fold(0.0, {acc, d -> acc + d * d}))
    val c1 = wrappers[t.first]
    val c2 = wrappers[t.second]
    println("distance(${c1.author}/${c1.name}, ${c2.author}/${c2.name}) = $distance")
  }
}
```

نمونه خروجی از اجرای پروژه:

```
distance(mohsen/fact.cpp, mohsen/fib.cpp) = 1.0081155523814582
distance(mohsen/fib.cpp, mohsen/mod.cpp) = 1.5813740343109277
distance(mohsen/mod.cpp, mohsen/sum.cpp) = 1.2364089902105937
distance(mohsen/fact.cpp, mohsen/mod.cpp) = 1.3088401052814793
distance(mohsen/fib.cpp, mohsen/sum.cpp) = 1.2209709504829427
distance(mohsen/fact.cpp, mohsen/sum.cpp) = 1.643307142409667
```

۲- روش TF-IDF

در این روش که در تحلیل مؤلف متن[۲۱] نیز استفاده می‌شود، تعداد تکرار[۲۲] یک واژه (token) در مجموعه‌ای از کدهای مرجع حائز اهمیت است. نکته‌ای که در این روش مورد تأکید است بررسی تکرار واژه‌ها با در نظر گرفتن همه‌ی کدها است. بنابراین نباید کدهای مرجع را مستقلاً بررسی کرد. همچنین این روش برخلاف روش قبل، هر کد را به یک بردار در فضای اقلیدسی نگاشت می‌کند. اگر D مجموعه‌ی کدها و T مجموعه‌ی کل واژه‌ها باشد داریم:

$$\forall\, t \in T$$

$$\forall\, d \in D$$

$$TF - IDF(t, d, D) = TF(t, d) * IDF(t, D)$$

که در آن $TF(t, d)$ فراوانی واژه‌ی t در کد d است. همچنین:

$$IDF(t, D) = \log\left(\frac{|D|}{DF(t, D)}\right) + 1$$

$DF(t, D)$ نیز فراوانی کدهایی است که شامل واژه‌ی t می‌باشند.

[21] Text Authorship Attribution
[22] Frequency

نهایتا بردار معادل کد d به صورت زیر است:

$$(TF-IDF(t_1,d,D), TF-IDF(t_2,d,D), \dots, TF-IDF(t_{|T|},d,D))$$

در پیاده سازی باید ویژگی‌هایی که تمایزی میان مؤلفان کد ایجاد نمی‌کنند را حذف کرد مانند stop wordها.

پیاده سازی (در زبان کاتلین):

```
import org.antlr.v4.runtime.CharStreams
import java.io.File
import kotlin.math.log

val stopWords = listOf("#include", "using", "namespace", ";", "{", "}", "(", ")")
val tokenSet = HashSet<String>()

class Document(private val codeText: String, val name: String, val author: String){

    private val tokens = CPP14Lexer(CharStreams.fromString(codeText)).allTokens.map { it.text }
    val tf = tokens.filter { it !in stopWords } .groupingBy { it }.eachCount()
    fun containsToken(token: String): Boolean = token in tf.keys

    init {
        tokenSet.addAll(tf.keys)
    }

}

val documents = ArrayList<Document>()

fun df(token: String) = documents.count { it.containsToken(token) }
fun idf(token: String) = log(documents.size.toDouble() / df(token), 2.0) + 1

fun tfidf(token: String, document: Document) = document.tf.getOrElse(token, { 0 }) * idf(token)

fun main() {
    val datasetFolder = File("dataset")
    datasetFolder.list()!!.forEach{ author ->
        val authorFolder = File("dataset/$author")
        authorFolder.list()!!.forEach { code ->
            val codeFile = File("dataset/$author/$code")
            documents.add(Document(codeFile.readText(), code, author))
        }
    }
    documents.forEach { document ->
        print("${document.author}/${document.name} = ")
        println(tokenSet.map { token -> tfidf(token, document) }
                .joinToString(prefix = "(", postfix = ")", separator = ", "))
    }
}
```

نمونه خروجی از اجرای پروژه:

```
mohsen/fact.cpp = (2.0, 4.0, 2.0, 9.0, 0.0, 2.0, 2.0, 1.0, 0.0, 0.0, 0.0, 0.0, 3.0, 0.0, 0.0, 2.0, 2.0, 0.0, 1.0, 0.0, 2.0, 0.0, 0.0, 8.0, 0.0, 0.0, 0.0, 4.0, 6.0, 1.0, 2.0, 2.0, 0.0, 0.0, 4.0)
mohsen/fib.cpp = (2.0, 4.0, 2.0, 0.0, 0.0, 2.0, 2.0, 1.0, 0.0, 21.0, 0.0, 0.0, 0.0, 21.0, 21.0, 2.0, 2.0, 0.0, 1.0, 0.0, 2.0, 0.0, 0.0, 12.0, 2.0, 0.0, 6.0, 4.0, 8.0, 1.0, 6.0, 4.0, 3.0, 3.0, 8.0)
mohsen/mod.cpp = (0.0, 0.0, 0.0, 0.0, 2.0, 0.0, 0.0, 1.0, 2.0, 0.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 6.0, 1.0, 6.0, 0.0, 3.0, 4.0, 0.0, 0.0, 8.0, 0.0, 2.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0)
mohsen/sum.cpp = (0.0, 0.0, 0.0, 0.0, 2.0, 0.0, 0.0, 1.0, 2.0, 0.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 6.0, 1.0, 6.0, 0.0, 0.0, 4.0, 0.0, 2.0, 8.0, 0.0, 2.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```

مشاهده می‌شود که بردارهای بدست آمده همگی $|T|$ بعدی هستند و با بزرگ شدن مجموعه‌ی داده یا زیاد شدن متن کدهای مرجع، ابعاد بردارها بسیار زیاد خواهد شد. برای رفع این مشکل باید تعدادی از ویژگی‌های کم‌اهمیت را حذف کرده و ابعاد بردارها را کاهش دهیم. بدین منظور برای هر واژه‌ی t تعریف می‌کنیم:

$$x_t = \bigcup_{d \in D} TF - IDF(t, d, D)$$

U یک عمل‌گر انتخاب کننده‌ی ویژگی است. به عنوان مثال می‌تواند آزمون X^2 پیرسون[۲۳] باشد. حال می‌توان در جهت کاهش بعد[۲۴] از میان $|T|$ ویژگی، k ویژگی‌ای را انتخاب و فیلتر کرد که x_tهای آن‌ها بیش‌ترین مقادیر را دارند.

[23] Pearson’s Chi-squared Test
[24] Dimensionality Reduction

مراجع

Erwin Quiring, Alwin Maier, and Konrad Rieck, TU Braunschweig 2019. Misleading Authorship Attribution of Source Code using Adversarial Learning

Brian N. Pellin, University of Wisconsin 2006. Using Classification Techniques to Determine Source Code Authorship

Mohammed Abuhamad, Aziz Mohaisen, DaeHun Nyang, Tamer AbuHmed 2019. Large-Scale and Language-Oblivious Code Authorship Identification