E-Commerce Recommender System Prototype Using Pattern Matching on Recent Purchase History and Recent Search History

Antonio Natthan Krishna - 13521162[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13521162@std.stei.itb.ac.id

Abstract—This paper discusses simple recommender system prototype using Boyer-Moore pattern searching algorithm and Levenshtein pattern similarity algorithm. It starts discussing types of pattern matching algorithms and use it as a core of recommender system this paper proposes. At the end of this paper, a recommendation system will be shown that can select 10 out of 100 most suitable products by using pattern matching on recent search history and recent purchase history which corresponds to the property in e-commerce.

Keywords— Boyer-Moore, Levenshtein, Pattern Matching, E-Commerce, Recommender System.

I. Introduction

Every day, millions of people shop through e-commerce. E-commerce has now become a favorite shopping method for many people because of the ease of shopping it offers. There are many things that e-commerce offers to its users: ease of shopping, discount promos, product prices that are below the average market price, etc. This makes e-commerce a primary shopping method for some people.

E-commerce nowadays has a lot of features that serve to pamper its users and attract more customers. Some make the application look like social media so that users can interact more naturally, some spoil their users by giving big promos on certain days, and there are so many business strategies that e-commerce uses so that users can spend more money through their platform.

To attract users to spend their money, e-commerce must have an effective recommender system. A good recommender system must be able to provide product recommendations that match the characteristics of each user. These characteristics can be seen from the user's personal data, e.g., gender and residential address, users' recent purchase history, users' recent search history, and many more. This will increase the likelihood that users will spend their money by buying products offered by the recommender system.

In this paper, a simple recommender system will be simulated using recent purchase history and recent search history. The recommender system will use a pattern matching algorithm and measure the degree of difference between patterns. Pattern matching algorithm is one of the most powerful basic algorithms in computer science today. In this paper, it will be shown how this algorithm will support the development of this recommender system.

This prototype recommender system is certainly very far from the one that exists and is used in e-commerce today. The real ones already utilize more sophisticated algorithms, e.g., artificial intelligence, machine learning, deep learning, etc. However, every recommender has the same principle: recommend products with the highest level of suitability. This method is expected to provide a simple overview of how recommendation systems work and the application of pattern matching in the real world.

II. Theoretical Basis

A. Pattern Matching

Pattern matching is comparing two patterns in order to determine whether they match (i.e., that they are the same) or do not match (i.e., that they differ) (Tony Hak & Jan Dull, 2009). A string will be evaluated character by character and compared with other strings. The comparison method is different for each algorithm. At the end of the pattern matching algorithm, it will be determined whether the two compared strings are the same, similar, or not the same.

Pattern matching consists of many properties. However, in this paper, only the pattern searching algorithm and the pattern similarity algorithm will be discussed and used in this prototype recommender system. These two algorithms are sufficient to simulate how pattern matching is used in the recommender system's algorithm development.

B. Pattern Searching Algorithms

The three most popular pattern searching algorithms are brute force algorithm, Knuth Morris Pratt algorithm, Boyer Moore algorithm. In the subsequent examples, we are going to locate "abacab" and "acaacc" on string "abacaadaccabacabaabb" using those three algorithms.

1. Brute Force

By using brute force definition: obvious, the idea of searching pattern using brute force algorithm is comparing each character of the pattern and shift pattern to the right by

one character if there is a mismatch. The illustration of pattern searching using brute force algorithm is shown below. For the next and rest pattern searching illustration, T stands for pattern to search, P stands for pattern to match. Every comparison happens is shown with underline on the P character. Bold pattern means there is a match.

a. "abacab"

```
T: a b a c a a d a c c a b a c a b a a b b
P: a b a c a b
     a b a c a b
       a b a c a b
         a b a c a b
           a b a c a b
             a b a c a b
               a b a c a b
                 a b a c a b
                   a b a c a b
                     a b a c a b
                       a b a c a b
                         a b a c a b
                           a b a c a b
```

b. "acaacc"

```
T: a b a c a a d a c c a b a c a b a a b b
P: a c a a c c
     a c a a c c
       a c a a c c
         a c a a c c
           a c a a c c
             a c a a c c
               a c a a c c
                 a c a a c c
                   a c a a c c
                     a c a a c c
                       a c a a c c
                         a c a a c c
                           a c a a c c
                             a c a a c c
                               a c a a c c
                                 a c a a c c
                                   a c a a c c
```

2. Knuth Morris Pratt

This algorithm is a fixation of the brute force algorithm (in context of pattern searching). We no longer shift only one character, but we shift pattern based on how many prefix characters of the pattern which are the same with the suffix character of the pattern. Knuth Morris Pratt algorithm (hereinafter will be referred to as KMP). Hence, KMP will do preprocessing algorithms to pattern to match.

The patterns to match are "abacab" and "acaacc". Hence first we are going to preprocess "abacab" first using this steps,

a. "abacab" is consisted by 6 characters.

a	b	a	c	a	b

b. There is no character before character index 0 (a). If there is a mismatch happens in character index 0, hence there is no prefix which is the same with suffix before index 0.

a	b	a	c	a	b
0					

c. There is only one character before character index 1 (b). If there is a mismatch happens in character index 1, hence there is no prefix which is the same with suffix before index 1 which is not the same substring.

a	b	a	c	a	b
0	0				

d. There are two characters before character index 2 (a). They are character index 0 (a) and character index 1 (b). The only possible arrangement prefix and suffix which is not the same substring and not null is,

Prefix	Suffix	Pre = Su
a	b	0

$a \neq b$, hence,

a	b	a	c	a	b
0	0	0			

e. There are three characters before character index 3 (c). The possible arrangements prefix and suffix which is not the same substring and not null is,

Prefix	Suffix	Pre = Su
a	ba	1
ab	a	1

Match prefix and suffix happens on character index 0 and character index 2, hence we take maximum value of number of character prefix which are the same to the suffix,

a	b	a	c	a	b
0	0	0	1		

f. Do step (e) to the rest of character of the pattern,

Prefix	Suffix	Pre = Su
a	ba	1
ab	a	1

Match prefix and suffix happens on character index 0 and character index 2, hence we take maximum value of number of character prefix which are the same to the suffix,

a	b	a	c	a	b
0	0	0	1	0	1

Do the same step for patten "acaacc", hence the results are,

a	c	a	a	c	c
0	0	0	1	1	2

Number below the pattern will be the guide to shift pattern if there is a mismatch. If there is mismatch at a certain character.

Shift the pattern such that character index i will be parallel to the location of mismatch. If mismatch happens at index 0. Shift pattern by one character to the right. KMP algorithm is illustrated below,

a. "abacab"

```
T: a b a c a a d a c c a b a c a b a a b b
P: a b a c a b
       a b a c a b
         a b a c a b
           a b a c a b
             a b a c a b
               a b a c a b
                 a b a c a b
                   a b a c a b
```

b. "acaacc"

```
T: a b a c a a d a c c a b a c a b a a b b
P: a c a a c c
     a c a a c c
       a c a a c c
         a c a a c c
           a c a a c c
             a c a a c c
               a c a a c c
                 a c a a c c
                   a c a a c c
                     a c a a c c
                       a c a a c c
                         a c a a c c
```

3. Boyer Moore

Boyer-Moore algorithm (hereafter referred to as BM) proposes a different approach to pattern search. It is based on whether the characters in the pattern to search exist in the pattern to match. It also performs preprocessing on the pattern to match before finally performing the string-matching algorithm.

The patterns to match are "abacab" and "acaacc". Hence first we are going to preprocess "abacab" first using this steps,

a. "abacab" is consisted by 3 distinct characters.

a	b	c

b. List the last occurrence of each character in the pattern to match (start from index 0)

a	b	c
4	5	3

Hence, "acaacc",

a. "acaacc" is consisted by 2 distinct characters.

a	c

b. List the last occurrence of each character in the pattern to match (start from index 0)

a	c
3	5

Number below the pattern will be the guide to shift pattern if there is a mismatch. If there is mismatch at a certain character. Shift the pattern such that character index i will be parallel to the location of mismatch. Comparison starts from the last character in pattern to match. There are 3 situations that could be happening when pattern match using BM,

a. If there is a mismatch and the last occurrence of the pattern to search mismatch character in pattern to match have not been evaluated yet. Shift pattern to match so that the character is parallel to location of mismatch.
b. If there is a mismatch and the last occurrence of the pattern to search mismatch character in pattern to match have been evaluated. Shift pattern to match one to the right.
c. If there is a mismatch and the pattern to search mismatch character does not exist in pattern to match. Shift pattern to match all the way to the right of mismatch character.

One thing to note: BM compares characters starting from the last index of pattern to match. BM algorithm is illustrated below,

a. "abacab"

```
T: a b a c a a d a c c a b a c a b a a b b
P: a b a c a b
     a b a c a b
               a b a c a b
                 a b a c a b
                   a b a c a b
```

b. "acaacc"

```
T: a b a c a a d a c c a b a c a b a a b b
P: a c a a c c
       a c a a c c
         a c a a c c
           a c a a c c
             a c a a c c
               a c a a c c
                 a c a a c c
                   ...
```

C. Pattern Similarity Algorithms

In real-world practice, it is often the case that the desired pattern is not available in its exact form. For example, we want to find "aaab" but there is only "aab" or "aaa". Therefore, an algorithm is needed to measure the similarity of two strings (generally comparing strings that are different or not exactly the same). There are 2 algorithms that will be discussed: Hamming Distance and Levenshtein Distance.

1. Hamming Distance

Hamming distance is based on the similarity of characters at the same index. Suppose we want to compare "arose" with "ros". The process is illustrated in the table below,

a	r	o	s	e
r	o	s		
X	X	X	X	X

Every mismatch will increase Hamming value of those strings. Based on the example above, the Hamming distance between "arose" and "ros" is 5 (number of mismatch). Suppose we also want to compare "arose"

with “aro”. Hence, the Hamming process would be,

a	r	o	s	e
a	r	o		
			X	X

Based on the example above, the Hamming distance between “arose” and “aro” is 2 (number of mismatch).

2. Levenshtein Distance

Levenshtein distance talks about how we can modify the first string to become the second with minimum number of edits. Types of edits is either replace, delete, or insert. Levenshtein distance has a general equation,

$$lev_{a,b}(i,j) = \begin{cases} \max(i,j) & \min(i,j) = 0 \\ min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + 1_{(a_i \neq b_i)} \end{cases} & otherwise \end{cases}$$

Suppose we want to compare “arose” and “ros”. Similarity measurements use these steps,

a. Make a matrix table like the following table

	“	a	r	o	s	e
“						
r						
o						
s						

b. Fill the first column and row such that the table looks like the table below

	“	a	r	o	s	e
“	0	1	2	3	4	5
r	1					
o	2					
s	3					

c. Evaluate other cells value using Levenshtein equation. The table would be,

	“	a	r	o	s	e
“	0	1	2	3	4	5
r	1	1	1	2	3	4
o	2	2	2	1	2	3
s	3	3	3	2	1	2

d. The value of levenshtein distance is the value of the most bottom right cell in the matrix. In this case, the value would be 2

With the same algorithm, the Levenshtein score for “arose” and “aro” can be evaluated with levenshtein matrix below.

	“	a	r	o	s	e
“	0	1	2	3	4	5
a	1	0	1	2	3	4
r	2	1	0	1	2	3
o	3	2	1	0	1	2

III. Algorithm And Data Structure

A. Big Picture

In this experiment, there is already 100 product data generated by data faker. Every product data has name and category. For example,

```
ut ultrices_grain
```

Source: Documentation (src/data/product.txt)

The data snippet above shows a product with name “ut ultrices” in category “grain”. This will be target variable since e-commerce recommender system would recommend products to its users.

Based on three string searching algorithms discussed in the previous chapter, we can conclude that BM algorithm is more efficient compared to the others. BM algorithm is becoming more effective with increasing numbers of distinct character used in pattern. Since English alphabet contains 26 distinct characters, BM is the most suitable algorithm for the development of this recommender system.

Based on three string similarity algorithms discussed in the previous chapter, we are going to use Levenshtein distance to determine the degree of differences between two strings, since Levenshtein focused on how one string can be the other, not only comparing character on certain index or on a certain order.

System would recommend product based on users’ recent purchase history and recent search history. System would match every data from those sources and calculate match value to every product exist in product data. And recommend 10 products which have the highest match value.

All program is written in Python.

B. Algorithms

1. Pattern Searching using Boyer-Moore Algorithm

```
# Pattern searching using Boyer Moore algorithm

# Pre process
def generateCharTable(pattern):
    chartable = [-1 for i in range (256)]
    pattern = str(pattern)
    for i in range (len(pattern)):
        chartable[ord(pattern[i])] = i
    return chartable

# Searching algorithm
def search(text, pattern):
    chartable = generateCharTable(pattern)

    m = len(pattern)
    n = len(text)
    s = 0

    while(s <= (n - m)):
        j = m - 1

        while (j >= 0 and
               pattern[j] == text[s + j]):
            j = j - 1

        if (j < 0):
            return s
        else:
```

```
            s += max(1,
               j - chartable[ord(text[s + j])])
    return -1
```

2. Pattern Similarity using Levenshtein Distance

```
def lev_dist(a, b):
    def min_dist(s1, s2):
        if s1 == len(a) or s2 == len(b):
            return len(a) - s1 + len(b) - s2

        # no change required
        if a[s1] == b[s2]:
            return min_dist(s1 + 1, s2 + 1)

        return 1 + min(
            min_dist(s1, s2 + 1),
            min_dist(s1 + 1, s2),
            min_dist(s1 + 1, s2 + 1),
        )

    return min_dist(0, 0)
```

3. Reader data (for product)

```
# Reader for Product
def readerProduct(path):
    f = open(path, "r")
    li = []
    for x in f:
        name = ""
        category = ""
        i = 0

        while (x[i] != "_"):
            name = name + x[i]
            i = i + 1

        i = i+1
        while (x[i] != "\n"):
            category = category + x[i]
            i = i + 1

        li.append(Product(name, category))
    f.close()
    return li
```

4. Reader data (for search history)

```
# Reader for Search History
def readerSearch(path):
    f = open(path, "r")
    li = []
    for x in f:
        li.append(x.splitlines()[0])
    f.close()
    return li
```

C. Data Structure

1. Product (Class)

```
class Product:
    def __init__(self, name, category):
        self.name = name
        self.category = category
        self.match = 0

    def getName(self):
        return self.name

    def getCategory(self):
        return self.category

    def getMatch(self):
        return self.match

    def info(self):
        print(self.name,          self.category,
self.match)

    def updateMatch(self, newmatch):
        self.match = newmatch
```

D. Recommender Algorithm

1. evaluateData
   Type: Function returns match score on a certain product

```
def evaluateData(text, pattern):
    val = 0
    if (search(text, pattern) != -1):
        val = val + 1
    similarity = lev_dist(text, pattern)
    if (len(text) > len(pattern)):
        val = val +
          (len(text) - similarity) / len(text)
    else:
        val = val +
    (len(pattern) - similarity) / len(pattern)
    return val
```

2. processRecommend
   Type: Procedure to update all match score in product database returns dictionary

```
def processRecommend(products,
                     search, purchased):
    for x in products:
        newmatch = 0
        for y in search:
            newmatch +=
              evaluateData(y, x.getName())
            newmatch +=
              evaluateData(y, x.getCategory())

        for y in purchased:
            newmatch +=
        evaluateData(y.getName(), x.getName())
            newmatch +=
evaluateData(y.getCategory(), x.getCategory())

        x.updateMatch(newmatch)
```

3. recommendProduct
   Type: Function returns 10 products with the highest match score.

```
def recommendProduct(products):
    recommend = sorted(products, key=lambda x:
x.getMatch(), reverse=True)
    recommend = recommend[:10]
    return recommend
```

E. Main Program

The subsequent program is written in Python.

```
products = readerProduct("src/data/product.txt")
purchased =
          readerProduct("src/data/purchased.txt")
search = readerSearch("src/data/search.txt")

processRecommend(products, search, purchased)
recommend = recommendProduct(products)
```

```
print("RECOMMENDED PRODUCT")
for item in recommend:
  item.info()
```

IV. EXPERIMENT

1. Test Case 1
   a. Recent Search History

```
vegetable
cras
feli
dairy
mas
```

   b. Recent Purchase History

```
justo_grain
semper_grain
urna_fruit
volutpat_fruit
at diam_meat
```

   c. Recommended Product Results

```
RECOMMENDED PRODUCT
volutpat fruit 8.933333333333334
justo grain 8.858333333333333
semper grain 8.791666666666666
urna fruit 8.665079365079364
eget grain 8.325
creato grain 8.069444444444445
ut ultrices grain 7.858585858585859
pharetra grain 7.833333333333334
diam vitae grain 7.822222222222222
cras in grain 7.793650793650795
```

2. Test Case 2
   a. Recent Search History

```
vegetable
cras
feli
meat
meat
```

   b. Recent Purchase History

```
justo_grain
semper_grain
urna_fruit
integer_meat
at diam_meat
```

   c. Recommended Product Results

```
RECOMMENDED PRODUCT
at diam meat 12.147619047619049
integer meat 12.147619047619047
diam meat 11.754761904761905
eu est meat 11.385714285714286
consequat meat 11.211111111111112
pede meat 11.056349206349207
in meat 11.00079365079365
tristique meat 10.877777777777776
suspendisse meat 10.8676767676767768
malesuada in meat 10.822222222222221
```

3. Test Case 3
   a. Recent Search History

```
semper
cras
metus
arcu
sapien
```

   b. Recent Purchase History

```
justo_grain
semper_grain
urna_fruit
integer_meat
at diam_meat
```

   c. Recommended Product Results

```
RECOMMENDED PRODUCT
semper grain 10.519047619047619
sapien grain 8.85238095238095
metus grain 8.609523809523807
arcu meat 8.426190476190477
integer meat 8.273809523809524
at diam meat 7.988095238095238
justo grain 7.933333333333345
pharetra grain 7.608333333333334
ut ultrices grain 7.460060060060061
in meat 7.428571428571429
```

4. Test Case 4
   a. Recent Search History

```
quis
lorem
nascetur
```

   b. Recent Purchase History

```
quis augue_dairy
semper_grain
urna_fruit
nascetur_dairy
lorem vitae_dairy
```

   c. Recommended Product Results

```
RECOMMENDED PRODUCT
nascetur dairy 11.275
et dairy 10.141666666666667
praesent dairy 9.65
at dairy 9.275
consectetuer dairy 8.441666666666666
eta dairy 7.9
lorem vitae dairy 7.888636363636364
volutpat erat dairy 7.832692307692308
luctus et dairy 7.747222222222223
blandit dairy 7.739285714285714
```

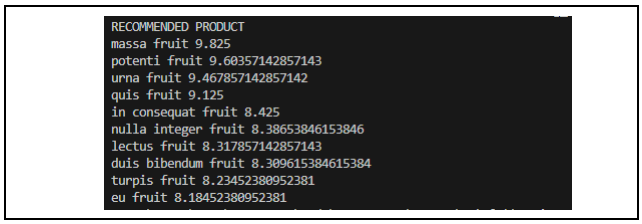
5. Test Case 5
   a. Recent Search History

```
quis
lorem
nascetur
```

   b. Recent Purchase History

```
potenti_fruit
semper_grain
urna_fruit
nascetur_dairy
massa_fruit
```

c. Recommended Product Results



V. Conclusion

Pattern matching related algorithms can be applied in many ways and can solve a wide variety of problems. In this paper, it has been shown that pattern matching algorithms can be applied in the development of a simple recommendation system using only Boyer Moore's string search algorithm and Levenshtein's string similarity algorithm. Both algorithms are simple, yet powerful when brought to bear on a problem. Not only that, but these algorithms can also simplify the way the recommendation system works as explained in the previous section. Hence, this algorithm can be used as an introduction in developing more advanced recommendation system that uses more feature engineering.

VI. Acknowledgment

Firstly, I want to thank myself and God for completing this paper. I cannot express what I am feeling now because I cannot believe that I wrote this paper fully in English. I want to thank Mr. Rinaldi Munir and Ms. Nur Ulfa Maulidevi for the fundamental knowledge that you have taught me earlier this semester. That knowledge has helped me to explore string matching algorithms at a level that I had not previously been able to reach. Hope this paper helps beginner student like me doing experiment on string matching algorithms and apply those in various aspects.

Appendix

1. Code documentation on GitHub
   https://github.com/natthankrish/String_Matching_Recommender_System.git
2. Explanation Video (in Indonesian Bahasa)
   https://youtu.be/5thhHlve-Ew

References

[1] Munir, Rinaldi, Nur Ulfa Maulidevi. 2021. Pencocokan String (String/Pattern Matching). Accessed on https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

[2] Khodra, Masayu Leylia. 2019. String Matching dengan Regular Expression. Accessed on https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2018-2019/String-Matching-dengan-Regex-2019.pdf

[3] Linyuan Lü, Matúš Medo, Chi Ho Yeung, Yi-Cheng Zhang, Zi Ke Zhang, Tao Zho. 2012. Recommender Systems. https://doi.org/10.1016/j.physrep.2012.02.006

[4] Tony Hak, Jan Dul. 2009. Pattern Matching. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=1433934

[5] Nam, Ethan. 2019. Understanding the Levenshtein Distance Equation for Beginners. Accessed on https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0.

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 22 Mei 2023

Antonio Natthan Krishna - 13521162