IDENTIFIKASI DUPLIKASI LAPORAN BUG PADA REPOSITORI LAPORAN BUG UNTUK MENGHASILKAN SARAN RESOLUSI BUG PERANGKAT LUNAK

Sugiyanto[1)], **Widhy Hayuhardhika Nugraha Putra**[2)]
[1)] Jurusan Teknik Informatika Institut Teknologi Adhi Tama Surabaya
[2)] Program Teknik Informatika dan Ilmu Komputer Universitas Brawijaya Malang
Email : [1)]sugianto@itats.ac.id, [2)]widhy@ub.ac.id

ABSTRAK

Repositori laporan bug *perangkat lunak sebenarnya menyediakan informasi dan saran resolusi* bug *tertentu yang dapat digunakan untuk memperbaiki* bug *yang terjadi pada tahap perbaikan perangkat lunak. Pengembang dapat mencari saran resolusi* bug *perangkat lunak tertentu dengan mengidentifikasi duplikasi* bug *pada repositori laporan* bug *perangkat lunak. Penelitian ini mengusulkan sistem identifikasi duplikasi laporan* bug *pada repositori laporan* bug *untuk menghasilkan saran resolusi* bug *perangkat lunak menggunakan teknik pendekatan temu kembali informasi. Pengembang dapat menemukan informasi alasan kegagalan perangkat lunak dan memperoleh saran resolusi untuk memperbaiki* bug *tersebut. Hasil penelitian ini menunjukkan bahwa teknik pendekatan temu kembali informasi dapat digunakan untuk mengidentifikasi duplikasi laporan* bug *pada repositori laporan* bug *untuk menghasilkan saran resolusi* bug *perangkat lunak.*

***Kata Kunci**: duplikasi laporan* bug*, repositori laporan* bug*, temu kembali informasi*

1 PENDAHULUAN

Dalam SDLC *(Software Development Life Cycle),* aktifitas evolusi perangkat lunak berada dalam tahap pemeliharaan (*maintenance*). Dalam aktivitas ini, pengembang dihadapkan pada banyak sekali masalah, terutama masalah untuk mengatur sumber daya. Mulai dari jumlah tim pengembang yang terbatas, dana yang terbatas, dan waktu yang diberikan untuk menyelesaikan suatu masalah juga terbatas. Oleh karena itu, pengembang memerlukan suatu tindakan yang efektif dan efisien untuk dapat menyelesaikan permasalahan tersebut, terutama untuk masalah *bug* [1]. *Bug* perangkat lunak adalah kesalahan atau cacat dalam perangkat lunak [2],[3]. *Bug* adalah sesuatu yang seharusnya tidak dilakukan oleh perangkat lunak atau perangkat lunak yang tidak melakukan seperti yang seharusnya dilakukan [4].

Seringkali dua atau lebih laporan *bug* mempunyai jenis kerusakan yang sama [1]. Dalam proyek perangkat lunak berskala besar, banyak laporan *bug* yang teridentifikasi sebagai duplikasi laporan *bug* (lebih dari 30%) [5]. Identifikasi duplikat laporan *bug* dapat membantu dalam memperbaiki *bug*. Pengembang masih melakukan pencarian secara manual berdasarkan ingatannya dan pencarian tekstual [6]. Pencarian ini memberikan hasil yang tidak akurat.

Repositori *bug* sebenarnya menyimpan informasi tentang kegagalan perangkat lunak seperti bagaimana kegagalan terjadi dan bagaimana hal itu dapat diatasi [7]. Salah satu permasalahannya adalah repositori laporan *bug* bersifat tidak terstruktur. Membuka *file* dokumen laporan *bug* satu persatu untuk mencari informasi *bug* merupakan tindakan yang tidak efektif. Saat ini, pencarian data sederhana untuk mendapatkan informasi berdasarkan kata dan mencocokkannya dengan suatu dokumen sudah umum ditemukan pada sistem komputer. Proses ini akan menampilkan hasil pencarian dokumen yang ditemukan pada sistem, baik hasilnya relevan maupun tidak relevan. Namun pemrosesan ini memiliki banyak kelemahan seperti waktu proses yang lama, redundansi hasil, dan tidak sesuainya hasil dengan keinginan pengguna [8]. Oleh karena itu, diperlukan suatu metode untuk pencarian informasi yang efektif.

Temu kembali informasi *(information retrieval)* merupakan tindakan, metode dan prosedur untuk menemukan kembali informasi data yang tersimpan sesuai subyek yang dibutuhkan [8]. Tindakan tersebut mencakup pembentukan indeks teks, analisis praproses, dan analisis relevansi.

Informasi *bug* perangkat lunak dapat diekstrak dan diprediksi dari historinya [9],[10]. Kendala utama dari pendekatan model data relasional adalah penggunaan konsep dari model data yang menjaga konsistensi, menghilangkan redundansi, atau menghilangkan anomali. Hal ini dilakukan dengan membentuk model data secara relasi-relasi tabel dalam bentuk normalisasi. Pendekatan temu kembali informasi dapat digunakan untuk

menyediakan sebuah antarmuka yang menampilkan pesan resolusi *bug* dan menilai kualitas artefak perangkat lunak yang terkait. Pendekatan temu kembali informasi terbukti mampu digunakan dalam berbagai kasus pencarian [11],[12].

Beberapa penelitian telah mengusulkan antarmuka dan struktur data generik untuk menangani masalah atau manajemen *bug* [13], [14], [15], [16], [17], [18]. Sistem pelacakan *bug* dengan menggunakan bugzilla dan klasifikasi *severity* dari *bug* [19] kurang efektif dan efisien dalam pencarian informasi *bug*. Atribut dalam bugzilla terbatas dan data yang dimasukkan oleh pengguna aplikasi tersebut harus spesifik. Atribut-atribut *bug* tersebut harus diisi dengan nilai atau isian yang sesuai oleh pengguna. Kolom *severity* menjadi suatu masalah jika dihadapkan pada beberapa hal seperti jika pengguna mempunyai latar belakang yang beragam. Selain itu, pengguna dihadapkan pada waktu yang singkat sehingga tidak dapat melakukan pengecekan semua kolom secara menyeluruh.

Model data yang terpadu seperti dokumen laporan *bug* dengan model 4C memiliki atribut yang lebih banyak dapat digunakan untuk mendukung pencarian *bug* [20]. Tujuan dari sistem temu kembali informasi adalah memenuhi kebutuhan informasi pengguna dengan mengembalikan semua dokumen yang relevan, pada waktu yang sama mengembalikan sesedikit mungkin atau tidak sama sekali dokumen yang tidak relevan [8]. Sistem ini menggunakan metode tertentu untuk mendapatkan dokumen-dokumen yang relevan dengan kueri pengguna. Sistem temu kembali informasi yang baik memungkinkan pengguna menentukan secara cepat dan akurat apakah isi dari dokumen yang diterima memenuhi kebutuhannya. Jika laporan *bug* perangkat lunak yang sesuai dapat diambil untuk memperbaiki *bug* tersebut, maka kualitas dan produktivitas pengembangan perangkat lunak dapat ditingkatkan.

Penelitian ini mengusulkan sistem identifikasi duplikasi laporan *bug* pada repositori laporan *bug* untuk menghasilkan saran resolusi *bug* perangkat lunak menggunakan teknik pendekatan temu kembali informasi. Tulisan ini dibagi menjadi 5(lima) bagian. Latar belakang permasalahan dikemukakan pada bagian 1. Pada bagian 2 dijelaskan model, analisis, desain, dan implementasi dari sistem ini. Rancangan sistem dan uji coba dijelaskan pada bagian 3 dan 4. Pada akhir tulisan diuraikan kesimpulan yang diambil dari hasil penelitian.

2 MODEL, ANALISIS, DESAIN, DAN IMPLEMENTASI

2.1 *Bug* Perangkat Lunak

Bug perangkat lunak adalah kesalahan atau cacat dalam perangkat lunak [2],[3]. *Bug* adalah sesuatu yang seharusnya tidak dilakukan oleh perangkat lunak atau perangkat lunak yang tidak melakukan seperti yang seharusnya dilakukan [4]. *Bug* pada perangkat lunak sendiri bermacam-macam. Jenis *bug* berdasarkan karakteristiknya [2] antara lain sebagai berikut:

1. *Infinite Loop*

Loop merupakan perulangan, yang seringkali digunakan dalam proses pemrograman. Penggunaan *loop* yang salah akan menyebabkan sebuah program menjalankan sebuah prosedur tanpa akhir.

2. *Arithmatic Overflow or Underflow*

Overflow akan terjadi jika sebuah nilai hasil perhitungan lebih besar daripada nilai yang dapat ditampung oleh variabel penyimpanan. Sementara *underflow* akan terjadi jika sebuah perhitungan menghasilkan nilai yang lebih kecil daripada nilai yang dapat ditampung oleh variabel penyimpanan. Hal ini sering ditemukan pada perhitungan aritmatika dan dapat menjadi masalah.

3. *Exceeding Array Bounds*

Array adalah variabel berdimensi yang memiliki indeks. Program dapat menjadi *error* saat mengakses indeks di luar *array* yang ditentukan.

4. *Access Violation*

Access Violation terjadi saat sebuah proses mencoba melewati batas yang diinginkan sistem. Misalnya menulis sebuah nilai pada alamat memory, media atau segmen yang diproteksi.

5. *Memory leak*

Memory leak adalah penggunaan memori yang tidak diinginkan, Hal ini dapat terjadi karena program gagal melepaskan memori yang sudah tidak digunakan.

6. *Stack Overflow or Underflow*

Stack merupakan struktur data yang menggunakan pendekatan LIFO *(Last in First Out)*. Pada program dapat diimplementasikan logika *stack* untuk suatu tujuan. *Stack Overflow/Underflow* terjadi jika *stack* melebihi atau di bawah nilai yang diijinkan oleh program.

7. *Buffer Overflow*

Buffer adalah media penyimpanan sementara dalam teknik pemrograman. *Buffer overflow* terjadi jika *buffer* yang disediakan tidak dapat menampung atau menyimpan terlalu banyak data.

8. *Deadlock*

Deadlock adalah suatu kondisi saat dua atau lebih proses saling menunggu satu sama lain untuk menyelesaikan prosesnya. *Deadlock* sering terjadi saat program menjalankan lebih dari satu proses.

Hal ini menyebabkan tidak satu pun dari dua proses tersebut yang selesai.

9. *Off by One Error*

Off by One Error adalah istilah untuk menggambarkan perulangan yang terlalu banyak atau terlalu sedikit. Misalnya perulangan yang diinginkan adalah 3 kali, tetapi yang terjadi adalah aplikasi tersebut mengulang proses tersebut sebanyak 2 atau 4 kali. *Off by One Error* umumnya terjadi karena kesalahan logika penulisan kode pada proses perulangan.

10. *Divide by Zero*

Divide by Zero terjadi jika pada sebuah proses pembagian, pembagi bernilai 0, sehingga program akan terhenti dan mengalami *error*.

2.2 Repositori Laporan *Bug*

Repositori laporan *bug* yang digunakan dalam penelitian ini menggunakan laporan *bug* dengan struktur berbasis 4C meta-model [20] seperti yang terlihat pada Gambar 1.

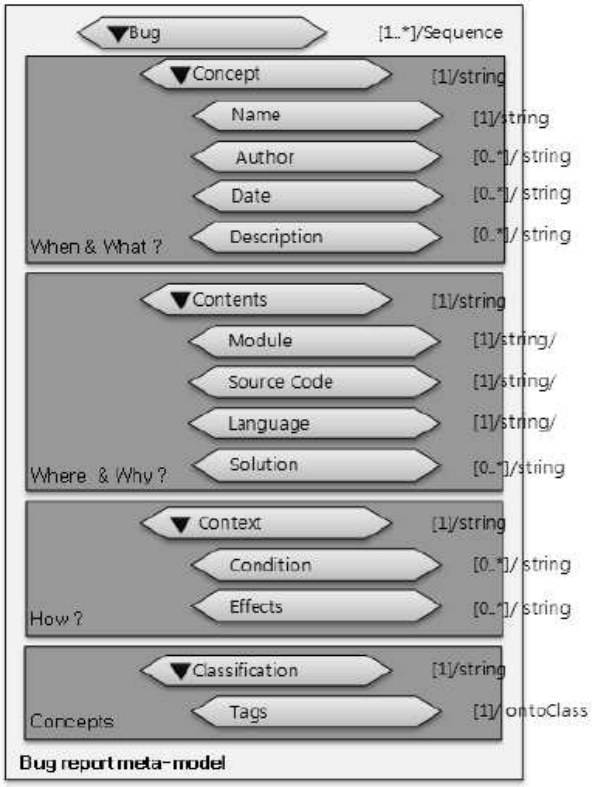
Struktur laporan *bug* 4C meta-model memiliki konsep, isi, konteks, dan sifat klasifikasi. Model ini harus sederhana dan mudah digunakan untuk pencarian semantik. *Properties Concepts* menunjukkan informasi dasar seperti nama *bug*, penulis, tanggal dan deskripsi. *Properties Content* merupakan informasi perangkat lunak termasuk *bug* seperti modul, kode sumber, bahasa dan solusi. *Properties Context* menggambarkan situasi yang terjadi disebabkan adanya *bug*, termasuk kondisi dan efek. *Properties Classification* menunjukkan *tag* label oleh pengembang dan pengguna seperti informasi jenis. Informasi lain dapat ditambahkan ke masing-masing *properties* jika perlu.

2.3 Proses Perangkingan Dokumen

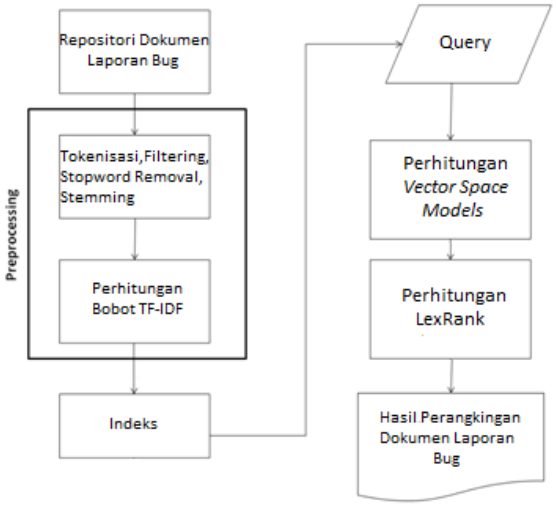
Sebelum dilakukan proses perangkingan perlu dilakukan tahapan pengindeksan seperti terlihat pada Gambar 2. Pada tahap ini terdapat beberapa proses yang saling berkesinambungan. Proses-proses dalam tahap ini diantaranya tokenisasi, *filtering*, *stopwords removal*, *stemming* dan penghitungan bobot kata/*term*.

2.3.1 Praproses Laporan *Bug*

Data dalam repositori laporan *bug* yang berupa teks/string harus diubah menjadi bentuk representasi kata dengan menggunakan metode tokenisasi, *stop word removal* dan *stemming*. Tokenisasi adalah proses memecah suatu teks/string menjadi kumpulan subString dengan menggunakan teks/string pemisah tertentu. Ada dua macam kelas tokenisasi yang dapat digunakan, yaitu *WordTokenizer* dan *AlphabeticTokenizer*. *WordTokenizer* adalah kelas tokenisasi sederhana yang menggunakan kelas *java.util.StringTokenizer* untuk proses tokenisasinya, sedangkan *AlphabeticTokenizer* melakukan tokenisasi berdasarkan urutan *alphabetic* dari kata yang didapat. Aplikasi menggunakan *String* ' \r\n\t.,;:'"()?!' sebagai *String* pemisah dalam tokenisasi.



Gambar 1. Struktur laporan *bug*



Gambar 2. Proses perangkingan dokumen

Stopword Removal adalah proses menghilangkan kata-kata yang umum digunakan dan tidak mempunyai informasi yang berharga pada

suatu konteks. Sistem ini menggunakan list *stopwords* standar yang sudah disediakan oleh pustaka *Weka*.

Stemming adalah proses mengurangi kata yang berimbuhan atau kata turunan menjadi bentuk dasar. Contohnya kata 'runs', 'ran', 'running' adalah kata yang ditulis dari bentuk dasar yang sama yaitu 'run', oleh karena itu hanya perlu disimpan satu kali saja. Setiap macam akhiran diasosiasikan dengan satu kondisi. Pada tahap awal sebuah kata dicari akhirannya dengan mencari potongan akhiran yang terpanjang yang sesuai dengan daftar akhiran. Kemudian dilakukan pengecekan kondisi sesuai dengan akhiran yang digunakan. Jika terpenuhi selanjutnya dilakukan proses transformasi sesuai dengan daftar aturan transformasi.

2.3.2 Perhitungan Bobot TF-IDF

Setelah laporan *bug* diseleksi menjadi susunan kata, maka proses pembobotan dapat dilakukan. Perangkingan laporan *bug* menggunakan representasi *vector space model* dari kumpulan data. Dokumen dalam *vector space model* direpresentasikan dalam matriks yang berisi bobot kata pada dokumen. Bobot tersebut menyatakan kepentingan/kontribusi kata terhadap suatu dokumen dan kumpulan dokumen. Kepentingan suatu kata dalam dokumen dapat dilihat dari frekuensi kemunculannya terhadap dokumen.

Metode TF-IDF *(Term Frequency-Inverse Document Frequency*) merupakan suatu metode yang digunakan untuk menghitung bobot hubungan suatu kata (*term*) tertentu terhadap dokumen [21]. Metode ini menggabungkan dua jenis konsep untuk perhitungan bobot antara lain frekuensi kemunculan sebuah kata di dalam sebuah dokumen tertentu (TF) dan *inverse* frekuensi dokumen yang mengandung kata tersebut (IDF). Frekuensi kemunculan kata dalam dokumen akan menunjukkan seberapa penting kata tersebut di dalam dokumen tersebut.

Frekuensi dokumen yang mengandung kata akan menunjukkan seberapa umum kata tersebut. Sehingga bobot hubungan antara sebuah kata dan sebuah dokumen akan tinggi apabila frekuensi kata tersebut tinggi di dalam dokumen dan frekuensi keseluruhan dokumen yang mengandung kata tersebut yang rendah pada kumpulan dokumen laporan *bug* perangkat lunak (repositori).

Rumus umum untuk TF-IDF dapat dilihat pada Persamaan 1.

$$w_{t,d} = (1 + \log tf_{t,d}) \times \log_{10} N / df_t \quad (1)$$

dimana:

w_{td} = bobot kata/*term td* terhadap dokumen *df*

tf_{td} = jumlah kemunculan kata/*term td* dalam *df*

N = jumlah semua dokumen yang ada dalam repositori

df = jumlah dokumen yang mengandung kata/*term td* (minimal ada satu kata yaitu *term td*)

2.3.3 Perhitungan *Vector Space Models*

Hasil algoritma TF-IDF sering mendapatkan nilai bobot (w) yang sama untuk dua dokumen yang berbeda. Hal ini dikarenakan perhitungan bobot menggunakan TF-IDF berdasarkan perhitungan kueri sehingga pengurutan yang dilakukan untuk mendapatkan nilai similaritas kurang akurat. Oleh karena itu, diperlukan metode lain untuk mendapatkan nilai similaritas antara kata/kalimat kunci terhadap sebuah dokumen, salah satunya dapat digunakan metode *vector space models* [22] yang direpresentasikan dalam Gambar 3.

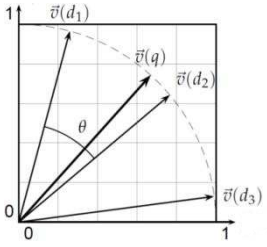
Dasar metode ini adalah menghitung nilai cosinus sudut dari dua vektor, yaitu bobot (w) dari setiap dokumen dan bobot (w) dari kata/kalimat kunci. Dengan menggunakan *vector space models* untuk menghitung nilai similaritas, formula similaritas yang dipakai dapat dilihat pada persamaan 2.

$$Sim(Q, D_i) = \frac{\sum_i w_{Q,j} w_{i,j}}{\sqrt{\sum_j w_{Q,j}^2} \sqrt{\sum_i w_{i,j}^2}} \quad (2)$$

Notasi Q adalah *Query*, D adalah Dokumen, i adalah indeks dokumen, w adalah *weight* (bobot kata/*term*) dan *j* adalah indeks *term.* Nilai *weight (w)* untuk TF-IDF yang telah ternormalisasi dinyatakan dalam persamaan 3.

$$w_{Q,i} = \left\{ \frac{tf_{Q,i}}{\max tf_{Q,i}} \right\} * log \left\{ \frac{D}{df_i} \right\} \quad (3)$$

Notasi Q adalah *query*, *i* adalah indeks *term*, tf adalah *term frequency*, D adalah jumlah dokumen dan *dfi* adalah jumlah dokumen yang mengandung *term* ke-*i*. Rumus *vector space models* ini nantinya akan digunakan sebagai rumus untuk mencari nilai similaritas dari dua kalimat. Kalimat-kalimat akan direpresentasikan sebagai dokumen dan dihitung similaritas dengan dokumen lain yang juga berupa kalimat.



Gambar 3. Representasi *vector space models*

2.3.4 Perhitungan LexRank

LexRank diadopsi dari PageRank, yang merupakan algoritma perangkingan untuk halaman web. Metode LexRank mengaplikasikan graf matematis sebagai penyelesaiannnya. Persamaannya dapat dilihat pada persamaan 4 dan 5.

$$S(V_i) = (1-d) + d * \sum_{V_j \in I_n(V_i)}^{n} \frac{1}{|\mathrm{Out}(\mathrm{V_j})|} \mathrm{S}(\mathrm{V_j}) \quad (4)$$

$$WS(V_i) = (1-d) + d * \sum_{V_j \in I_n(V_i)} \frac{\mathrm{w_{ji}}}{\sum_{\mathrm{V_k} \in \mathrm{Out}(\mathrm{V_j})} \mathrm{w_{jk}}} WS(V_j) \quad (5)$$

Dimana: V_i = *vertex* yang dihitung nilai skornya; V_j = *vertex* yang bertetanggaan dengan V_i; V_k = *vertex* yang bertetanggaan dengan V_j; dan d = *damping factor* yang nilainya biasanya antara 0 dan 1, misalnya 0,85.

Saat diterapkan pada graf tekstual, biasanya terdapat bobot antara dua *vertex*. Formula diatas mengasumsikan graf berarah. Namun algoritma ini dapat juga diterapkan pada graf tidak berarah dan berbobot (*undirected and weighted graph*), sehingga formulasinya seperti dalam Persamaan 6.

$$WS(V_i) = d + (1-d) * \sum_{V_j \in Adj(V_i)} \frac{\mathrm{w_{ji}}}{\sum_{\mathrm{V_k} \in \mathrm{Adj}(\mathrm{V_j})} \mathrm{w_{jk}}} WS(V_j) \quad (6)$$

3 SKENARIO UJI COBA

Skenario uji coba dirancang dengan tujuan untuk menilai kualitas sistem dengan menggunakan *precison* dan *recall*. Uji coba dilakukan dengan menggunakan data uji yang didapatkan dari *event log* perangkat lunak *open source*. Data yang digunakan dalam uji coba ini merupakan *corpus* atau kumpulan dokumen teks laporan *bug* perangkat lunak.

Kumpulan dokumen tersebut diubah menjadi sebuah repositori laporan *bug* perangkat lunak yang terdiri atas 200 dokumen laporan *bug* perangkat lunak. Kemudian dipilih sebuah dokumen laporan *bug* perangkat lunak untuk dihitung nilai kemiripannya dengan dokumen-dokumen yang ada pada repositori. Nilai kemiripan tertinggi yang dihasilkan oleh sistem dianggap sebagai duplikasi laporan *bug* dan selanjutnya dievaluasi dengan cara dibandingkan dengan hasil penilaian kemiripan berdasarkan asumsi manusia. Semakin tinggi nilai kemiripan dokumen maka semakin tinggi duplikasi laporan *bug*.

Tabel 1. Tabel Evaluasi

Informasi	Relevan	Tidak Relevan
Ditampilkan	True Positive	False Positive
Tidak Ditampilkan	False Negative	True Negative

Hasil dari evaluasi sistem dihitung menggunakan rumus *precision* dan *recall* dengan pendekatan dokumen yang ditampilkan dan relevan seperti pada Tabel 1.

Tabel 1 tersebut menunjukkan beberapa item yang diperlukan untuk mengukur performa sistem. Item-item tersebut akan digunakan untuk menghitung *precision* dan *recall*. Jika peneliti mengasumsikan A sebagai himpunan dokumen laporan *bug* yang seharusnya dikembalikan oleh sistem (*retrieved document*), B adalah himpunan dokumen laporan *bug* yang relevan dengan kueri, dan A∩B adalah himpunan dokumen benar yang diberikan sistem, maka nilai *precision* dan *recall* sistem ini dapat dicari menggunakan Persamaan 7 dan Persamaan 8.

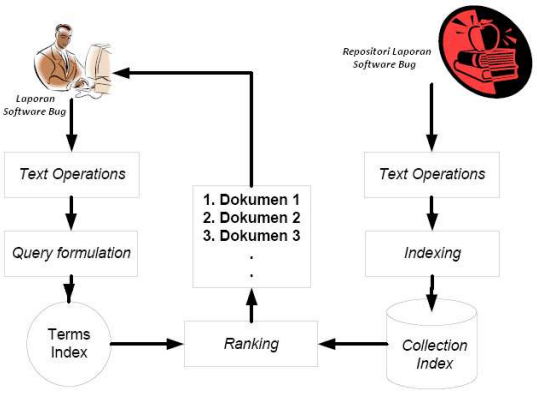
Precision (P) = TP / (TP + FP) (7)
Recall (R) = TP / (TP + FN) (8)

Pada dasarnya, nilai *recall* dan *precision* berada pada rentang antara 0 sampai dengan 1. Oleh karena itu, suatu sistem yang baik adalah yang dapat memberikan nilai *precision* dan *recall* mendekati 1.

3.1 Rancangan Sistem

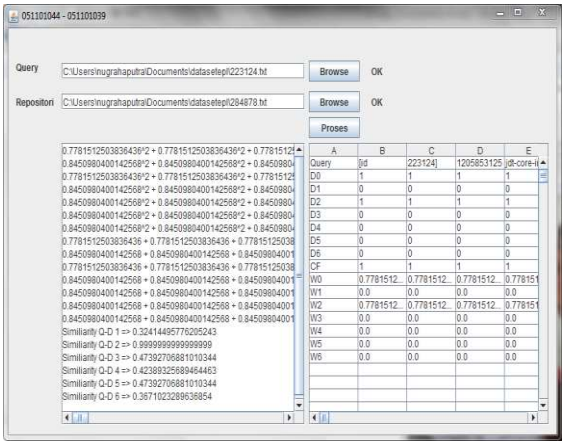
Sistem identifikasi duplikasi *bug* ini dikembangkan menggunakan bahasa pemrograman Java. Sistem tersebut selanjutnya dibangun dengan arsitektur yang diilustrasikan pada Gambar 4.

Unit antar muka akan menerima masukan laporan *bug* perangkat lunak yang dialami oleh pengguna atau pengembang perangkat lunak. Sistem akan melakukan operasi teks (tokenisasi, *filtering, stopword removal*, *stemming*) dan formulasi *query*. Proses ini akan menghasilkan *term index* (indeks yang berisi bobot tiap kata). Selanjutnya sistem akan membandingkan *term index* dan *collection index* berdasarkan perhitungan *vector space model* dan LexRank. Proses ini akan menghasilkan perangkingan laporan *bug* yang diurutkan berdasarkan tingkat kemiripan dokumen tersebut terhadap laporan *bug* perangkat lunak yang diinputkan oleh pengguna. Laporan *bug* yang terpilih akan menampilkan saran resolusi *bug*.



Gambar 4. Arsitektur Sistem

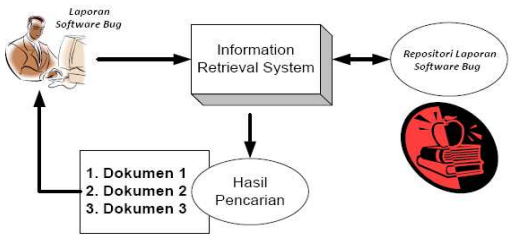
Antarmuka aplikasi dibuat untuk menampung masukan dokumen laporan *bug* dalam bentuk file teks (.txt) terstruktur sesuai struktur *bug* pada repositori. Disamping itu, antarmuka aplikasi juga menampilkan informasi perhitungan yang dilakukan. Antarmuka aplikasi ditunjukkan pada Gambar 5.



Gambar 5. Antarmuka aplikasi

3.3 Langkah-langkah Uji Coba

Pengujian dilakukan pada kueri dokumen yang memiliki lebih dari satu laporan *bug* hasil pencarian yang relevan. Pengujian dilakukan sebanyak 170 kali dengan memakai beberapa variasi dokumen laporan *bug*. Hasil pencarian kemudian dievaluasi oleh peneliti. Gambar 6 mengilustrasikan proses yang dilakukan dalam uji coba.



Gambar 6. Proses uji coba

4 HASIL UJI COBA

Hasil perhitungan awal berupa matriks keterhubungan dan nilai kemiripan antar dokumen dapat dilihat pada Lampiran 1. Hasil perhitungan TF-IDF yang berupa matriks keterhubungan dan nilai kemiripan antar dokumen laporan *bug* digunakan sebagai acuan dalam perhitungan bobot untuk masing-masing kueri menggunakan algoritma LexRank. Langkah pertama dalam menentukan bobot tiap dokumen laporan *bug* adalah dengan memberikan nilai awal 1 kepada masing-masing dokumen seperti pada Tabel 2 sebagai nilai *vertex* untuk menghitung *error rate* nilai *vertex* menggunakan persamaan 6.

Tabel 2. Tabel LexRank

u	v	Z	p(v)	sim(u,v)	Σsim(z,v)	Temp	p(u)
a	b	a,c,d,g	1	0.52915	2.33578	0.22654	0.32665
	c	a,b,d,g	1	0.50199	2.25431	0.22268	
	d	b,c,f,g	1	1	3.06229	0.32655	
	f	a,d,e	1	0.52915	1.56029	0.33913	
	g	a,b,c,d	1	0.52915	2.08944	0.25324	
b	a	b,c,d,f,g	1	0.52915	3.08944	0.17127	0.260819
	c	a,b,d,g	1	0.74833	2.25431	0.33195	
	d	b,c,f,g	1	0.52915	3.06229	0.17279	
	g	a,b,c,d	1	0.52915	2.08944	0.25324	
c	a	b,c,d,f,g	1	0.5019	3.08944	0.16248	0.254485
	b	a,c,d,g	1	0.74833	2.33578	0.32037	
	d	b,c,f,g	1	0.5019	3.06229	0.16392	
	g	a,b,c,d	1	0.5019	2.08944	0.24025	
d	a	b,c,d,f,g	1	1	3.08944	0.32368	0.398611
	b	a,c,d,g	1	0.52915	2.33578	0.22654	
	c	a,b,d,g	1	0.5019	2.25431	0.22268	
	e	F	1	0.26457	0.52915	0.5	
	f	a,d,e	1	0.5019	1.56029	0.32173	
	g	a,b,c,d	1	0.52915	2.08944	0.25324	
e	d	b,c,f,g	1	0.26457	3.06229	0.08639	0.185258
	f	a,d,e	1	0.52915	1.56029	0.33913	
f	a	b,c,d,f,g	1	0.52915	3.08944	0.17127	0.32170
	d	b,c,f,g	1	0.5019	3.06229	0.16392	
	e	f	1	0.52915	0.52915	1	
g	a	b,c,d,f,g	1	0.52915	3.08944	0.17127	0.240422
	b	a,c,d,g	1	0.52915	2.33578	0.22654	
	c	a,b,d,g	1	0.5019	2.25431	0.22268	
	d	b,c,f,g	1	0.52915	3.06229	0.17279	

Dimana: u = *vertex* yang dihitung nilai skornya; v = *vertex* yang bertetanggaan dengan u; Z = *vertex* yang bertetanggaan dengan v. Dengan mengunakan persamaan 6 didapatkan ranking masing-masing *vertex* dokumen *bug* pada repositori *bug*.

Dari hasil perhitungan LexRank, didapatkan nilai bobot masing-masing dokumen laporan *bug* (p(U)). Kemudian dilakukan pengurutan berdasarkan nilai LexRank. Nilai LexRank menandakan tingkat keutamaan dokumen *bug* pada repositori terhadap dokumen *bug* yang diujikan. Seperti yang ditampilkan pada Tabel 3 berikut.

Tabel 3. Hasil LexRank

U	p(u)
A	0.179317648
B	0.15355745
F	0.149612908
D	0.141298726
G	0.140254041
C	0.137985775
E	0.133451077

Pada tabel tersebut digambarkan bahwa dokumen uji yang digunakan memiliki tingkat kemiripan paling tinggi dengan dokumen A dan paling rendah pada dokumen E. Sehingga dapat disimpulkan bahwa dokumen uji terduplikasi paling mirip pada dokumen A. Sehingga informasi pada dokumen A seperti *assign to, resolution*, dan lain-lain dapat diambil sebagai referensi pada dokumen uji.

Untuk menguji kehandalan sistem dihitung nilai *precision* dan *recall* berdasarkan hasil perangkingan duplikasi yang dilakukan oleh manusia. *Precision* adalah rasio jumlah laporan *bug* relevan yang ditemukan dengan total jumlah laporan *bug* yang ditemukan sistem. *Recall* adalah rasio jumlah laporan *bug* relevan yang ditemukan kembali dengan total jumlah laporan bug dalam repositori yang dianggap relevan. Dalam percobaan diberikan hasil ringkasan yang dilakukan manusia.
Dari hasil perbandingan dapat diketahui jumlah *True Positive* = 111, *False Positive* = 29, jumlah *False Negative* = 30. Sehingga dapat dihitung nilai *precision* dan *recall* sebagai berikut :

$$Precision = \frac{111}{111 + 29} = 0.793$$

$$Recall = \frac{111}{111 + 30} = 0{,}787$$

Berdasarkan hasil uji coba, dapat dilihat bahwa sistem ini memiliki performa yang bagus dengan mempunyai nilai *precision* sebesar 79.3% dan *recall* sebesar 78.7%. Nilai *precision* dan *recall* ini masih dapat ditingkatkan dengan melakukan pengembangan pada praproses perangkingan dokumen.

5 KESIMPULAN

Berdasarkan sistem yang telah dibuat dan hasil yang didapat dari serangkaian uji coba yang telah dilakukan, sistem ini memiliki performa yang bagus dengan mempunyai nilai *precision* sebesar 79.3% dan *recall* sebesar 78.7% maka dapat ditarik kesimpulan atas penelitian ini bahwa teknik pendekatan temu kembali informasi dapat digunakan untuk identifikasi duplikasi *bug* pada repositori laporan *bug* untuk menghasilkan saran resolusi *bug* perangkat lunak. Dimana ekstraksi dokumen *bug* seperti kalimat-kalimat penyusun, kata-kata kunci dapat dijadikan acuan untuk menentukan perangkingan kemiripan dokumen uji terhadap pada repositori laporan *bug*. Penelitian selanjutnya dapat dilakukan dengan memperhitungkan nilai *Inverse Class Frequency* dalam metode pembobotan *term* sehingga dapat meningkatkan nilai *precision* dan *recall* sistem ini, dan juga diperhitungkan juga penggunaan semantik untuk menentukan tingkat kemiripan dokumen uji terhadap dokumen repositori *bug*.

6 DAFTAR PUSTAKA

[1] Sun, C., Lo, D., Wang, X., Jiang, J., dan Khoo, S. C, 2010. “A Discriminative Model Approach for Accurate Duplicate Bug Report Retrieval”. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Volume 1, pages 45–54.

[2] Weyuker, E.J., Bell, R. M., dan Ostrand, T. J., 2010. ”We're Finding Most of the Bugs, but What are We Missing?”. Proceeding International Conference on Software Testing, pp. 313 – 322.

[3] Lukins, S. K., Kraft, N. A., dan Etzkorn, L. H. 2008. ”Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation”. Proceeding Working Conference on Reverse Engineering, pp. 155-164.

[4] Telles, M., dan Hsieh, Y., 2001. The Science of Debugging. Scottsdale: Coriolis.

[5] Ritchey, S., dan Turner, R., 2014. “Classification Algorithms for Detecting Duplicate Bug Reports in Large Open Source Repositories”. Youngstown State University CREU Reports.

[6] Sureka, A., Jalote, P., 2010. "Detecting Duplicate Bug Report Using Character N-Gram-Based Features". IEEE Proceedings of Asia Pacific Software Engineering Conference.
[7] Alipour, A., Hindle, A., dan Stroulia, E., 2013. "A Contextual Approach towards More Accurate Duplicate Bug Report Detection". Proceedings of the 10th Working Conference on Mining Software Repositories, IEEE Press Piscataway, NJ, USA, pages 183-192.
[8] Manning, C.D., Prabhakar, R., dan Hinrich, S., 2009. An Introduction to Information Retrieval. Cambridge, England: Cambridge University Press.
[9] Dallmeier, V., dan Zimmermann, T., 2007. "Extraction of Bug Localization Benchmarks from History". Proceeding IEEE/ACM International Conference on Automated Software Engineering, pp. 433 - 436.
[10] Zimmermann, T., Nagappan, N., dan Zeller, A., 2008. "Predicting Bugs From History". Software Evolution, Springer, ch. 4, pp. 69 - 88.
[11] Ankolekar, A., Sycara, K., Herbsleb, J., Kraut, R., dan Welty, C., 2006. "Supporting Online Problem-Solving Communities with the Semantic Web". Proceeding International Conference on World Wide Web, pp. 575-584.
[12] Tran, H. M., Lange, C., Chulkov, G., Schonwalder, J., dan Kohlhase, M., 2009. "Applying Semantic Techniques to Search and Analyze Bug Tracking Data". Journal of Network and Systems Management, vol. 17, no. 3, pp. 285-308.
[13] Wang, X., Zhang, L., Xie, T., Anvik, J., dan Sun, J., 2008. "An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information". Proceeding Industrial Electronics, Control and Instrumentation. IECON '91.
[14] Alipour, A., Hindle, A., dan Stroulia, E., 2013. "A Contextual Approach towards More Accurate Duplicate Bug Report Detection". Proceedings of the 10th Working Conference on Mining Software Repositories, IEEE Press Piscataway, NJ, USA, pages 183-192.
[15] Sean Banerjee, S., Cukic, B., dan Adjeroh, D., 2012. "Automated Duplicate Bug Report Classification Using Subsequence Matching". IEEE 14th International Symposium on High-Assurance Systems Engineering.
[16] Sun, C., Lo, D, Khoo, S.C., dan Jiang, J, 2011. "Towards More Accurate Retrieval of Duplicate Bug Reports". Automated Software Engineering (ASE) 26th IEEE/ACM International Conference.
[17] Tian, Y., Sun, C., dan Lo, D., 2012. "Improved Duplicate Bug Report Identification". IEEE Software Maintenance and Reengineering (CSMR) 16th European Conference.
[18] Tomasev, N., Leban, G., dan Mladenic, D., 2013. "Exploiting Hubs for Self-Adaptive Secondary Re-Ranking In Bug Report Duplicate Detection". IEEE Proceedings of the ITI 35th International Conference.
[19] Putra, T.R., Siahaan, D.O., dan Yuhana, U.L., 2011. "Klasifikasi Severity Dari Bug Untuk Proyek Perangkat Lunak". Tesis Jurusan Teknik Informatika, Fakultas Teknologi Informasi, Institut Teknologi Sepuluh Nopember.
[20] Edwards, S., 1990. "The 3C Model of Reusable Software Components". Proceeding Third Annual Workshop: Methods and Tools for Reuse.
[21] Mingyong, L., dan Jiangang, Y., 2012, "An improvement of TFIDF weighting in text categorization". International Conference on Computer Technology and Science (ICCTS 2012).
[22] Kurniawan, H., dan Aji, R.F., 2006. "Otomatisasi Pengelompokkan Koleksi Perpustakaan Dengan Pengukuran Cosine Similarity Dan Euclidean Distance". Seminar Nasional Aplikasi Teknologi Informasi 2006 (SNATI 2006), Yogyakarta.