AUTOMATIC TAGGING DENGAN MENGGUNAKAN ALGORITMA BIPARTITE GRAPH PARTITION DAN TWO WAY POISSON MIXTURE MODEL

Skripsi

Disusun untuk memenuhi salah satu syarat memperoleh gelar Sarjana Komputer

Mencerdaskan dan Memartabatkan Bangsa

Oleh:
Muhammad Zhafran Bahij
1313619012

PROGRAM STUDI ILMU KOMPUTER
FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS NEGERI JAKARTA

2023

LEMBAR PENGESAHAN

Dengan ini saya mahasiswa Fakultas Matematika dan Ilmu Pengetahuan Alam, Universitas Negeri Jakarta

Nama	: Muhammad Zhafran Bahij
No. Registrasi	: 1313619012
Program Studi	: Ilmu Komputer
Judul	: Automatic Tagging dengan Menggunakan Algoritma Bipartite Graph Partition dan Two Way Poisson Mixture Model

Menyatakan bahwa skripsi ini telah siap diajukan untuk sidang skripsi.

Menyetujui,

Dosen Pembimbing I	Dosen Pembimbing II
Muhammad Eka Suryana, M. Kom	**Med Irzal, M.Kom.**
NIP. 19851223 201212 1 002	NIP. 19770615 200312 1 001

Mengetahui,
Koordinator Program Studi Ilmu Komputer

Dr. Ria Arafiyah, M.Si.
NIP. 19751121 200501 2 004

LEMBAR PERSETUJUAN HASIL SIDANG SKRIPSI

Automatic Tagging* dengan Menggunakan Algoritma *Bipartite Graph Partition* dan *Two Way Poisson Mixture Model

Nama : Muhammad Zhafran Bahij
No. Registrasi : 1313619012

	Nama	Tanda Tangan	Tanggal
Penanggung Jawab			
Dekan	: Prof. Dr. Muktiningsih N, M.Si. NIP. 196405111989032001	.....	.....
Wakil Penanggung Jawab			
Wakil Dekan Bidang Akademik	: Dr. Esmar Budi, S.Si., MT. NIP. 197207281999031002	.....	.....
Ketua	: Ir. Fariani Hermin Indiyah, MT. NIP. 196605171994031003	.....	.....
Sekretaris	: Ari Hendarno, S.Pd, M.Kom NIP. 198811022022031002	.....	.....
Penguji Ahli	: Dr. Ria Arafiyah, M.Si. NIP. 197511212005012004	.....	.....
Pembimbing I	: Muhammad Eka Suryana, M.Kom. NIP. 198512232012121002	.....	.....
Pembimbing II	: Med Irzal, M.Kom. NIP. 197706152003121001	.....	.....

Dinyatakan lulus ujian skripsi tanggal: 22 Agustus 2023

LEMBAR PERNYATAAN

Saya menyatakan dengan sesungguhnya bahwa skripsi dengan judul ***Automatic Tagging*** **dengan Menggunakan Algoritma** ***Bipartite Graph Partition*** **dan** ***Two Way Poisson Mixture Model*** yang disusun sebagai syarat untuk memperoleh gelar Sarjana komputer dari Program Studi Ilmu Komputer Universitas Negeri Jakarta adalah karya ilmiah saya dengan arahan dari dosen pembimbing.

Sumber informasi yang diperoleh dari penulis lain yang telah dipublikasikan yang disebutkan dalam teks skripsi ini, telah dicantumkan dalam Daftar Pustaka sesuai dengan norma, kaidah dan etika penulisan ilmiah.

Jika dikemudian hari ditemukan sebagian besar skripsi ini bukan hasil karya saya sendiri dalam bagian-bagian tertentu, saya bersedia menerima sanksi pencabutan gelar akademik yang saya sanding dan sanksi-sanksi lainnya sesuai dengan peraturan perundang-undangan yang berlaku.

Jakarta, 10 Agustus 2023

Muhammad Zhafran Bahij

KATA PENGANTAR

Ungkapan Puji dan Syukur penulis panjatkan kehadirat Tuhan Yang Maha Esa, atas segala rahmat dan karunia-Nya sehingga penulis dapat menyelesaikan skripsi ini dengan baik. Adapun jenis penelitian yang dengan judul *Automatic Tagging dengan Menggunakan Algoritma Bipartite Graph Partition dan Two Way Poisson Mixture Model*

Dalam menyelesaikan skripsi ini, penulis selalu mendapat dorongan dan bantuan. Oleh karena itu, penulis menyampaikan terima kasih kepada:

1. Para petinggi di lingkungan FMIPA Universitas Negeri Jakarta.
2. Ibu Dr. Ria Arafiyah, M.Si. selaku Koordinator Program Studi Ilmu Komputer.
3. Bapak Muhammad Eka Suryana, M.Kom. selaku Dosen Pembimbing I yang telah membimbing, mengarahkan, serta memberikan saran dan koreksi terhadap skripsi ini.
4. Bapak Med Irzal M.Kom. selaku Dosen Pembimbing II yang telah membimbing, mengarahkan, serta memberikan saran dan koreksi terhadap skripsi ini.
5. Ayah dan Ibu penulis yang selama ini telah mendukung penulis dalam menyelesaikan skripsi ini dalam berbagai hal.
6. Sepupu penulis yang telah menginspirasi penulis untuk menjalani kehidupan di Program Studi Ilmu Komputer.
7. Teman-teman Ilmu Komputer 2019 yang telah menyemangati penulis dalam penulisan skripsi.
8. Rekan-rekan PT Fhadira Inovasi Teknologi yang telah mendukung penulis untuk menyelesaikan skripsi.
9. Teman-teman KANAU yang secara tidak langsung melatih penulis agar terbiasa melakukan penulisan.
10. Kawan-kawan penulis yang tidak bisa disebutkan satu persatu.

Dalam penulisan skripsi ini, penulis menyadari bahwa dengan keterbatasan ilmu dan pengetahuan penulis, skripsi ini masih jauh dari sempurna, baik dari segi penulisan, penyajian materi, maupun bahasa. Oleh karena itu, penulis sangat membutuhkan kritik dan saran yang dapat dijadikan sebagai pembelajaran serta dapat membangun penulis agar lebih baik lagi kedepannya.

Akhir kata, penulis berharap ini bermanfaat bagi semua pihak khususnya penulis sendiri, serta menjadi semangat dan motivasi bagi rekan-rekan yang akan melaksanakan skripsi berikutnya. Semoga Tuhan Yang Maha Esa senantiasa membalas kebaikan semua pihak yang telah membantu penulis dalam menyelesaikan proposal ini.

Terima kasih,

Jakarta, 10 Agustus 2023

Penulis

ABSTRAK

MUHAMMAD ZHAFRAN BAHIJ. *Automatic Tagging* dengan Menggunakan Algoritma *Bipartite Graph Partition* dan *Two Way Poisson Mixture Model*. Skripsi. Fakultas Matematika dan Ilmu Pengetahuan Alam, Universitas Negeri Jakarta. 2023. Di bawah bimbingan Med Irzal, M. Kom dan Muhammad Eka Suryana, M. Kom.

Automatic Tagging merupakan hal yang dilakukan untuk menentukan suatu kata kunci atau frasa kunci yang relevan pada suatu artikel, dokumen, gambar, atau video secara otomatis. Penelitian ini merupakan salah satu payung penelitian dari mesin pencari atau *search engine Telusuri* yang berpondasi kepada penelitian milik Lazuardy Khatulistiwa dengan judul *Perancangan Arsitektur Search Engine dengan Mengintegrasikan Web Crawler, Algoritma Page Ranking, dan Document Ranking*. Agar bisa melakukan pencarian dengan lebih efisien, salah satu caranya adalah pencarian melalui *tag*. Sebelum mencapai tahap pencarian melalui *tag*, langkah awal yang diperlukan adalah membuat program *Automatic Tagging*. Dalam penelitian ini, algoritma yang digunakan adalah *Bipartite Graph Partition* dan *Two Way Poisson Mixture Model* dengan menggunakan data latih dari suatu *website thehill.com*. Proses pembentukkan algoritma tersebut menggunakan bahasa pemrograman *Python*. Hasil akhir dari penelitian ini adalah mampu memberikan enam *tag* dengan ketepatan akurasi sebesar 37% dengan menggunakan data 229 dokumen atau artikel dan 723 *tag*.

Kata kunci : *tagging*, *database*, *bipartite graph*, *Poisson Mixture Model*

DAFTAR ISI

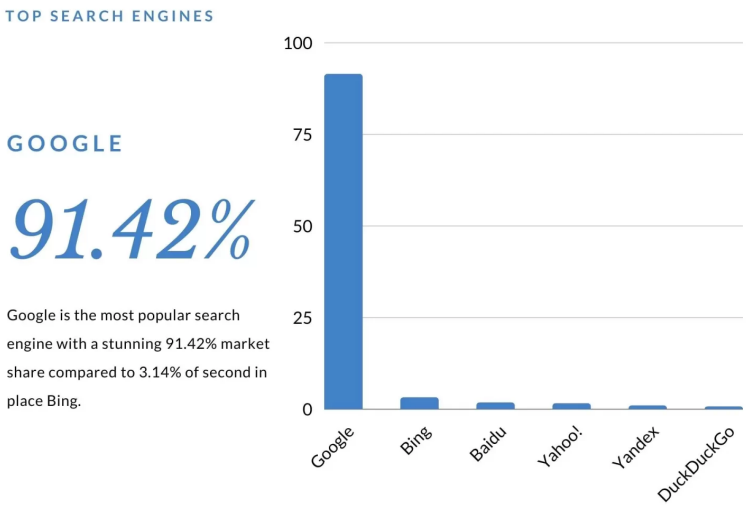
DAFTAR GAMBAR

BAB I

PENDAHULUAN

1.1 Latar Belakang Masalah

Saat ini, penggunaan *search engine* atau mesin pencari telah digunakan oleh khalayak umum untuk mencari berbagai informasi yang ada. Berdasarkan data dari (Christ, 2022), Google merupakan *search engine* yang menempati urutan pertama terpopuler, selanjutnya diikuti dengan Bing, Baidu, Yahoo!, Yandex, dan DuckDuckGo.

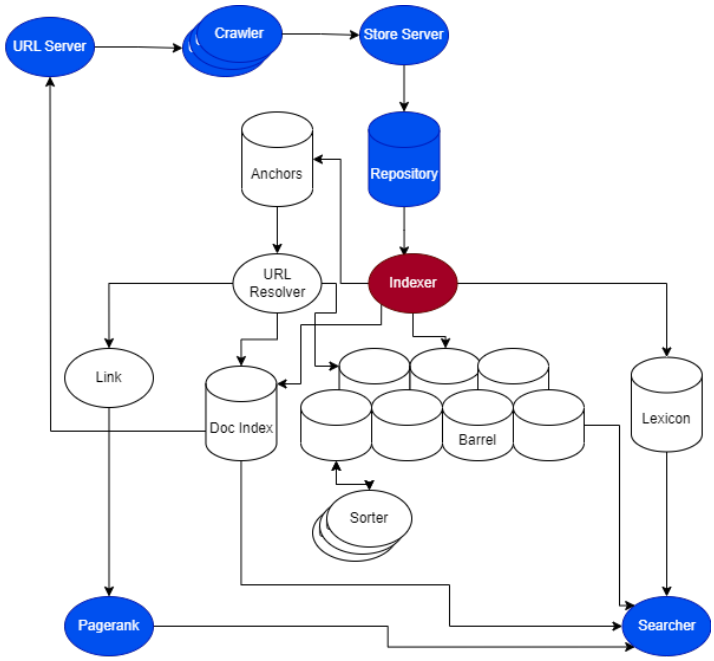


Gambar 1.1: Pengguaan search engine terpopuler (Christ, 2022)

Web Search Engine atau mesin pencari web merupakan suatu perangkat lunak yang digunakan untuk mencari sesuatu di internet berdasarkan kata-kata yang diberikan oleh pengguna sebagai *search terms*. Pembuatan *search engine* pertama kali dilakukan oleh Alan Emtage, Bill Heelan, dan J. Peter Deutsch pada tahun 1990. Mereka menamai *search engine* tersebut yaitu Archie (Seymour et al., 2011).

Pekerjaan utama dari *search engine* ada tiga yaitu *web crawling, indexing,* dan *searching*. *Search engine* bekerja dengan cara mengirimkan informasi tentang halaman web, Halaman tersebut di dapat dari *web crawler* suatu *automated web browser* yang mengikuti seluruh pranala yang ada di situs. Pengecualian situs yang

dicari dapat dilakukan melalui "*robots.txt*". Kemudian, konten dari setiap halaman akan dianalisis untuk menentukan urutan index. Data tentang halaman web dikirim ke dalam *index database* yang nantinya akan dilakukan *query*. *Query* bisa satu kata atau lebih. Tujuan pengindeksan adalah untuk menemukan informasi secepat mungkin (Seymour et al., 2011).



Gambar 1.2: High Level Google Architecture (Brin & Page, 1998)

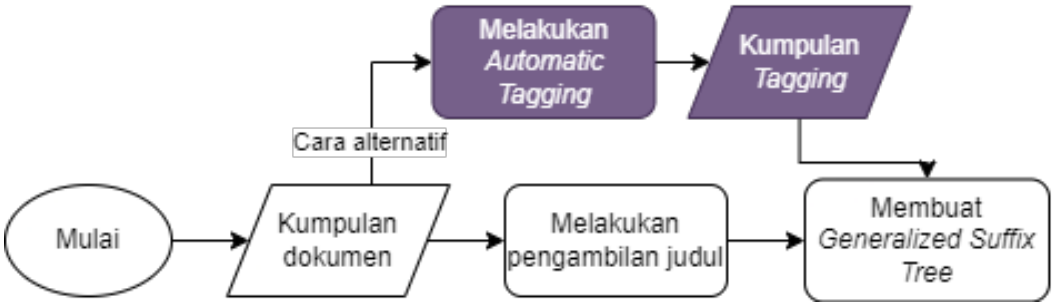
Pada Gambar 1.1, telah diperlihatkan bahwa Google merupakan *search engine* terfavorit. Google ditemukan oleh Larry Page dan Sergey Brin pada tahun 1998. Salah satu keunggulan Google adalah pengaplikasian PageRank yaitu mengatasi *underspecified queries*. Sebagai contohnya, jika kita mencari kata Real Madrid, maka situs pertama kali yang terlihat adalah situs resmi Real Madrid.

Gambar 1.2 merupakan struktur arsitektur Google. Warna biru menunjukkan hasil penelitian yang dilakukan oleh Lazuardy Khatulistiwa dalam penelitian yang berjudul "Perancangan Arsitektur Search Engine dengan Mengintegrasikan *Web Crawler, Algoritma Page Ranking*, dan *Document Ranking*" dan warna merah menunjukkan proses penelitian dari Zaidan Pratama dalam judul "Perancangan Modul Pengindeks pada *Search Engine* Berupa *Induced Generalized Suffix Tree* untuk Keperluan Perangkingan Dokumen". Salah satu komponen dalam *search engine* milik Google adalah *Indexer*. Dalam penelitian Zaidan Pratama, di sana

menjelaskan tentang melakukan pengindeksan melalui algoritma *General Suffix Tree* (*GST*) yang termodifikasi. (Pratama (2022))

Proses pengindeksan dimulai dengan memasukkan kumpulan dokumen yang dibuat menjadi *GST*. Kemudian, membuat *GST* dari kumpulan dokumen tersebut. Dari *GST* tersebut, nantinya akan dilakukan reduksi untuk node yang redundan atau node yang mengalami perulangan yang tidak diperlukan sehingga akan terbentuk pohon yang terinduksi untuk frekuensi f yang bernama *Induced Generalized Suffix Tree-f*.

IGST-f ini menjadi komponen utama dalam pengindeksan. Setelah itu, program menerima masukan berupa pola kata dan batas k untuk dicari pada kumpulan dokumen. Dari sinilah kita akan mencari nilai *count* dari setiap *node*. Kemudian, mencatat jumlah dokumen dalam *sublist* yang tereduksi untuk setiap *node*. Selanjutnya, mencari nilai *counter lowest common ancestor* dari setiap *node*. Terakhir, mengenai Indeks Efisien. Untuk *Top-k Document Retrieval Problem* dilakukan pengurutan terhadap representasi *array IGST-f* yang sudah memiliki nilai *count* dan mengembalikan hasil *top-k* yang meiliki pola *P*. Pratama (2022)



Gambar 1.3: Bagian *Automatic Tagging* pada *Indexer*

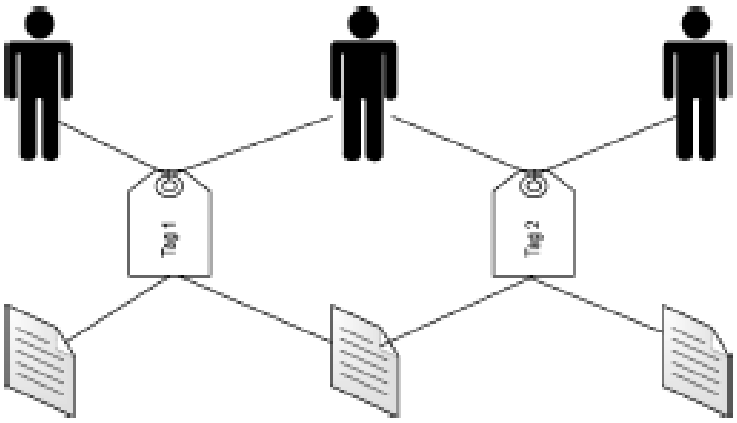
Akan tetapi, pengindeksan tersebut masih hanya melalui judul dan belum melalui *tag*. Oleh karena itu, salah satu alternatifnya adalah melakukan pengindeksan melalui *tag*. Pada gambar 1.3 Warna ungu adalah salah satu alternatif yang dapat dilakukan yaitu melalui *automatic tagging* yang nantinya peneliti akan lakukan. Hal ini dapat dimanfaatkan agar bisa melakukan pengindeksan lebih akurat.

Tagging merupakan hal yang biasa dilakukan untuk menggambarkan suatu kata kunci yang relevan atau frasa kunci pada suatu dokumen, gambar, atau video. Dalam merambatnya perkembangan Web 2.0 aplikasi seperti Del.icio.us dan Flickr, pelayanan *tagging* mulai populer dan menarik perhatian pihak akademis dan

industri. Penelitian tentang cara *automatic tag* membuahkan hasil. Cara melakukannya dengan algoritma *Poisson Mixture Model*. Dengan cara ini, kecepatan untuk membuat *automatic tag* bisa lebih cepat dibandingkan SimFusion dan VS+IG. Contohnya pada saat *Delicious Test Time*, *PMM* mampu menghasilkan 1,23 detik saat proses *automatic tag*, sedangkan SimFusion membutuhkan waktu 6,4 detik dan *VS+IG* membutuhkan waktu 77,43 detik. Selain kecepatan, *PMM* juga mampu di atas *SimFusion* serta *VS+IG* secara signifikan dalam hal akurasi, presisi, dan *recall*. (Song et al., 2008)

Selain itu, *tagging* juga digunakan untuk membantu pengorganisasian, *browsing*, dan pencarian. Seperti *image tagging* yang digunakan oleh Flickr, *web page tagging* yang digunakan oleh Del.ico.us, dan *social tagging* yang digunakan oleh Facebook, semua sistem tersebut menjadi populer dan dipergunakan di penjuru Web. (Sood, 2007)

Secara umum, sumber yang memiliki *tag* biasanya berasosiasi *tag* yang lain. Selain itu, sumber yang memiliki *tag* berasosiasi terhadap *user*. Sebagai contoh, *tagging* terhadap dokumen d yang dilakukan oleh *user* u dengan *tag* t dapat direpresentasikan sebagai tiga kesatuan (u, d, t). Dengan menggunakan pendekatan itu, dapat terbentuk suatu graf yang digambarkan sebagai berikut.



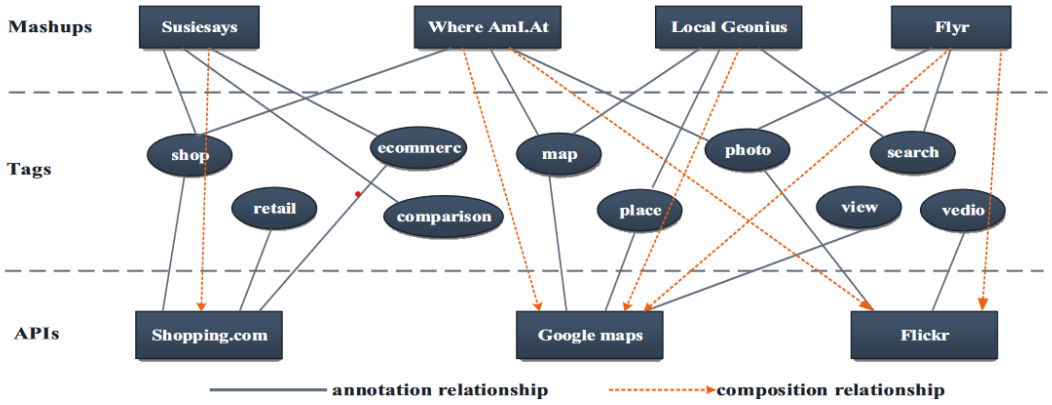
Gambar 1.4: Relasi antara user, tag, dan dokumen (Song et al., 2011)

Dengan relasi pada gambar 1.4 , rekomendasi *tag* dapat dilakukan dengan dua jenis menurut Song yaitu jenis pendekatan melalui pengguna dan jenis pendekatan melalui dokumen. Rekomendasi *tag* merupakan suatu sistem yang di mana sistem tersebut menampilkan *tag* yang relevan pada suatu dokumen agar pengguna bisa memperhitungkan apakah *tag* yang ditampilkan itu ingin dipakai atau

tidak. Melalui pendekatan *user*, sistem ini akan mengolah rekomendasi *tag* berdasarkan *tag-tag* yang telah dilakukan *user* sebelumnya dan merekomendasikan tag yang mirip dengan *user* ini atau kelompok dari *user* tersebut. Berbeda halnya dengan pendekatan dokumen, cara ini dilakukan dengan cara mengklasterisasikan dokumen-dokumen tersebut ke dalam topik-topik yang berbeda. Topik yang sama pada suatu dokumen akan memiliki *tag* yang diasumsikan lebih mirip dibandingkan dokumen yang berbeda topik. Namun, di antara kedua cara ini, yang dinilai kurang efektif adalah melalui pendekatan *user*. Pertama, berdasarkan penelitian dari Farooq et al. (2007), distribusi dari *user* vs *tag* mengikuti *long tail power law distribution*. Itu artinya, hanya sebagian kecil porsi dari *user* yang melakukan *tag* dengan panjang atau meluas. Sebagai tambahan, penggunaan *tag* yang berulang juga terbilang rendah, tetapi pertumbuhan perbendaharaan *tag* terus berkembang. Dengan sedikit pengguna relatif yang didapat, pendekatan *user* akan sulit untuk mencari model mana yang cocok buat untuk melakukan rekomendasi *tag* yang efektif. Berbanding terbalik dengan pendekatan dokumen yang lebih kokoh karena kekayaan informasi yang ada di dokumen. Bahkan, *tag* dan kata akan menciptakan relasi yang potensial antara topik dan konten di suatu dokumen yang di mana tag dianggap sebagai kelas label untuk dokumen dalam skenario *supervised learning* atau kesimpulan dari dokumen dalam skenario *unsupervised learning*. (Song et al., 2011)

Namun, percobaan ini hanya terbatas pada CiteULike dan del.icio.us. Untuk saat ini, kedua situs tersebut sudah tidak dapat diakses dengan semestinya. CiteULike beralih menjadi situs judi, sedangkan del.icio.us tidak dapat diakses oleh umum.

Beberapa tahun kemudian, suatu penelitian membahas mengenai *automatic mashup tag*. Secara sederhana, *mashup* adalah suatu *web service* yang di mana merupakan kumpulan dari kombinasi beberapa Web *API* dan konten dari berbagai sumber. Berbeda dengan rekomendasi *tag* yang menggunakan pendekatan dengan konten tekstual, di dalam *Web services* terdapat banyak sekali relasi seperti komposisi relasi antara *mashup* dengan *API* dan anotasi berelasi antara *API* dan *tag*. (Shi et al., 2016)



Gambar 1.5: Skema *Mashup*, *Tag*, dan *API* (Shi et al., 2016)

Selain teks, *tag* juga digunakan dalam hal yang bersifat non teks seperti video, musik, dan gambar. Dalam suatu video, *tag* sangat diperlukan untuk menentukan relevansi antara pencarian yang diinginkan dengan isi video. Meskipun beberapa platform video seperti Youtube menyediakan judul dan deskripsinya, bisa saja judul tersebut tidak ada keterkaitannya dengan video dan deskripsinya yang sangat panjang sehingga orang malas untuk membaca. Manfaat dalam *tag* video ada dua yaitu bisa menemukan daftar video yang representatif dan *tag* dapat mendukung untuk melakukan penemuan tentang video yang kontennya berhubungan dengan video yang telah ditonton. (Parra et al., 2018)

Untuk kasus *automatic tag* pada musik, *Automatic music tagging* adalah *multi label binary classification* yang bertujuan untuk memprediksi *tag* yang relevan pada suatu lagu. *Tag* tersebut membawa informasi musik semantik yang nantinya dapat digunakan untuk membuat aplikasi seperti rekomendasi musik. (Won et al., 2020)

Dengan demikian, peneliti ingin membuat penelitian terkait *automatic tagging* dengan menggunakan penelitian dari Song et al. (2008) yang terdapat dua algoritma utama yaitu *Bipartite Graph Partition* dan *Two Way Poisson Mixture Model*.

1.2 Rumusan Masalah

Berdasarkan latar belakang masalah yang telah diuraikan, maka perumusan masalah pada penelitian ini adalah "Bagaimana cara melakukan Automatic Tagging dengan menggunakan algoritma *Bipartite Graph Partition* dan *Two Way Poisson Mixture Model*?".

1.3 Batasan Masalah

Batasan masalah dalam penelitian ini yaitu:

1. Data latih dan data uji yang digunakan berasal dari satu sumber *web*.
2. Untuk *crawling* data, akan digunakan *crawler* dari sistem yang telah dibuat oleh Khatulistiwa (2022) berjudul "Perancangan Arsitektur Search Engine dengan Mengintegrasikan Web Crawler, Algoritma Page Ranking, dan Document Ranking".
3. Banyaknya K yang digunakan adalah dua.

1.4 Tujuan Penelitian

Tujuan dari penelitian ini adalah untuk membuat program *automatic tagging* dengan menggunakan algoritma *Bipartite Graph Partition* dan *Two Way Poisson Mixture Model* sesuai penelitian Song et al. (2008).

1.5 Manfaat Penelitian

Dalam penelitian ini, manfaat yang bisa diperoleh yaitu:

1. Bagi Peneliti

   Menambah pengetahuan penulis tentang *automatic tagging* terhadap dokumen.

2. Bagi Peneliti Selanjutnya

   Diharapkan metode yang diusulkan pada penelitian ini dapat membantu penelitian selanjutnya dalam mengembangkan sistem yang lebih kompleks dan bermanfaat.

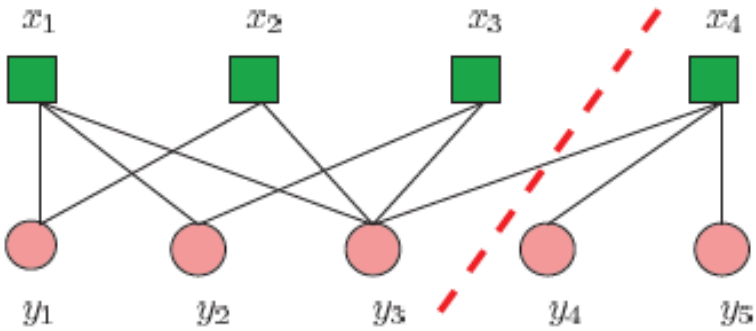
BAB II

KAJIAN PUSTAKA

2.1 Representasi Bipartite Graph

Mendefinisikan suatu graf $G = (V, E, W)$ sebagai suatu set dari titik-titik V dan hubungan mereka dengan garis E, dengan W sebagai bobot dari garis-garis tersebut. Sebagai contoh, w_{ij} sebagai bobot dari garis di antara titik i dan j.

Graf G disebut *bipartite* jika mengandung dua kelas titik X dan Y sebagai berikut $V = X \cup Y$ dan $X \cap Y = \emptyset$ setiap garis $e_{ij} \in E$ memiliki *endpoint* i di dalam X dan *endpoint* lain j di dalam Y. Biasanya, X dan Y merujuk kepada perbedaan tipe objek dan E merepresentasikan relasi antara keduanya. Di dalam konteks representasi dokumen, X merupakan suatu set dokumen, sedangkan Y adalah *terms* dan w_{ij} merupakan banyaknya *term* j yang muncul di dalam dokumen i. Perlu dicatat bahwa *weighted adjacency matrix* W untuk *bipartite graph* selalu simetris. Sebagai contoh, gambar berikut adalah *undirected bipartite graph* dengan 4 dokumen dan 5 *term*.



Gambar 2.1: Suatu *bipartite graph* X dan Y Song et al. (2008)

2.1.1 Normalisasi dan Aproksimasi

Normalisasi biasanya digunakan pertama kali untuk bobot matriks W untuk mengeliminasi *bias*. Jalur yang paling lurus untuk menormalisasikan W adalah normalisasi baris yang tidak mengambil akun dari simetri pada W. Namun, untuk memahami simetri pada W, Song dkk menggunakan *normalized graph Laplacian* untuk aproksimasi W. *Normalized Laplacian* $L(W)$ sebagai berikut.

$$L(W)_{ij} = \begin{cases} 1 - \frac{w_{ij}}{d_i} & \text{jika } i = j, \\ -\frac{w_{ij}}{\sqrt{d_i d_j}} & \text{jika } i \text{ dan } j \text{ terhubung} \\ 0 & \text{selain itu,} \end{cases}$$

di mana d_i adalah *degree* luar dari titik i, pada persamaan $d_i = \sum w_{ij}, \forall j \in V$. Kemudian, kita bisa mendefinisikan matriks diagonal D di mana $D_{ii} = d_i$. Oleh karena itu, *normalize Laplacian* dapat direpresentasikan sebagai berikut.

$$L(W) = D^{(-1/2)} W D^{(-1/2)} \tag{2.1}$$

Untuk *dataset* berskala besar seperti situs *corpora* dan koleksi gambar, spasi fitur mereka biasanya mengandung jutaan vektor dengan dimensi yang sangat tinggi (contohnya, x = 10^6, y = 10^7). Oleh karena itu, biasanya ini sering diinginkan untuk mencari *low rank matrix* W *komplemen* untuk aproksimasi $L(W)$ dengan tujuan untuk mengurangi biaya komputasi, mengekstrak korelasi, dan menghapus *noise*. Metode dekomposisi matriks tradisional misalnya adalah *Single Value Decomposition* dan *eigenvalue decomposition*, membutuhkan waktu *superlienar* untuk perkalian matriks vektor jadi biasanya mereka tidak menggunakannya sampai pengaplikasian di dunia nyata.

Untuk *symmetric low rank apporiximation*, di sini menggunakan algoritma Lanczos yang secara iteratif mencari nilai eigen dan vektor eigen dari matriks persegi. Diberikan n x n *sparse symmetric matrix* A dengan nilai eigen:

$$\lambda \geq ... \geq \lambda_n \geq 0 \tag{2.2}$$

Algoritma Lanczos menghitung k x k *symmetric tridiagonal matrix* T, yang nilai eigennya mengaproksimasi nilai eigen dari A, dan vektor eigen dari T bisa digunakan untuk mengaproksimasi vektor eigen A, dengan k lebih kecil dibandingkan n. Dengan kata lain, T yaitu:

$$||A - T||_F \leq e||A||_F \tag{2.3}$$

di mana $||.||_F$ denotasi *Frobenius norm*, dengan e sebagai variabel terkendali. Sebagai contoh, untuk menangkap 95% varians dari A, e diatur sebagai 0.05.

2.1.2 Bipartite Graph Partitioning

Untuk melakukan multi-klastering pada *bipartite graph*, Song et al. (2008) menggunakan algoritma *Spectral Recursive Embedding (SRE)*. Secara esensial, *SRE* digunakan untuk menkontruksi *partition* dengan meminimalisir normalisasi dari total pada bobot garis di antara pasangan yang tidak cocok pada suatu garis, misalnya $min_{\Pi(A,B)}Ncut(A,B)$, di mana A dan B adalah pasangan yang cocok dalam partisi dengan A^c dan B^c menjadi yang lain. Normalisasi varian dari *edge cut* $Ncut(A,B)$ didefinisikan sebagai berikut.

$$Ncut(A,B) = \frac{cut(A,B)}{W(A,Y)+W(X,B)} + \frac{cut(A^c,B^c)}{W(A^c,Y)+W(X,B^c)}, \tag{2.4}$$

di mana

$$\begin{aligned} cut(A,B) &= W(A,B^c) + W(A^c,B) \\ &= \sum_{i\in A, j\in B^c} w_{ij} + \sum_{i\in A^c, j\in B} w_{ij}, \end{aligned} \tag{2.5}$$

Rasional dari $Ncut$ tidak hanya mencari partisi dengan perpotongan garis kecil, tetapi juga mepartisinya sepadat mungkin. Ini berguna untuk aplikasi dari *tagging document* di mana dokumen dengan setiap partisi secara ideal berfokus kepada satu topik spesifik. Sebagai hasil, semakin padat suatu partisi, semakin baik yang dokumen relevan dan *tag* terkelompokkan bersamaan.

2.1.3 Dengan Cluster Node Ranking

Song et al. (2008) mendefinisikan dua pengukuran baru *N-Precision* dan *N-Recall* untuk *node ranking*. *N-Precision* dari titik i merupakan jumlah total dari bobot pada garis yang terkoneksi kepada titik dalam klaster yang sama, dibagi dengan jumlah total dari bobot garis yang ada di dalam klaster. Label klaster i sebagai $C(i)$,

$$np_i = \frac{\sum_{j=1}^{n} w_{ij}\Pi[C(j)=C(i)]}{\sum_{j,k=1}^{n} w_{jk}\Pi[C(j)=C(k)=C(i)]}, j,k \neq i, \tag{2.6}$$

Untuk graf tidak berbobot, persamaan di atas setara dengan banyaknya garis

yang berasosiasi dengan titik i di dalam klaster $C(i)$, dibagi dengan total garis yang ada di dalam klaster $C(i)$. Secara umum, *N-precision* mengukur seberapa pentingnya titik pada suatu klaster, di dalam perbandingan dengan titik lain. Dalam konteks pada suatu dokumen, klasternya adalah suatu set topik dari dokumen dan bobot dari titik-titik kata menunjukkan frekuensi dari kata-kata yang muncul pada topik tersebut. Dengan *cluster determined*, *denominator equation* bersifat konstan, jadinya semakin banyak bobot yang dimiliki pada suatu titik, semakin penting pula.

Kontrasnya, *N-recall* digunakan untuk menghitungkan probabilitas posterior dari titik i pada klaster yang diberikan dan *inverse* pembagian dari garis i yang berasosiasi dengan klasternya.

$$nr_i = \frac{|E_i|}{\sum_{j=1}^{n} w_{ij} \Pi[C(j) = C(i)]}. \tag{2.7}$$

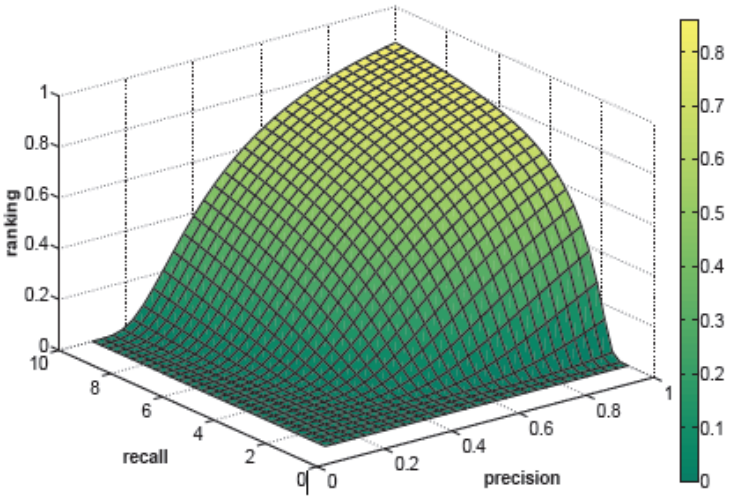
Hal tersebut merupakan bukti bahwa *N-Recall* selalu tidak kurang dari satu. Semakin lebar *N-Recall*, maka semakin mungkin bahwa kata tersebut berasosiasi dengan suatu topik yang spesifik.

Diberikan np_i dan nr_i, kita akan melakukan estimasi dari peringkat i:

$$Rank_i = \begin{cases} \exp(-\frac{1}{r(i)^2}), & r(i) \neq 0, \\ 0, & r(i) = 0, \end{cases} \textbf{di mana } r(i) = (np_i) * log(nr_i) \tag{2.8}$$

Berdasarkan persamaan di atas, fungsi perangkingan dari Song et al. (2008) merupakan pengganti yang dihaluskan yang proporsional untuk presisi titik (*node precision*) dan *recall*, hasilnya terjamin berada di kisaran *range (0, 1)*.

Potensi pengalikasian dari metodologi *bipartite graph node ranking* termasuk interpetasi relasi dokumen dan *author* yaitu menentukan relasi sosial (misal "*hub*" dan "*authority*") dari *author* di dalam satu topik penelitian yang sama, dan mencari representatif dokumen dari suatu topik. Di sini, mereka mengaplikasikan *framework* ini untuk melakukan rekomendasi *tag* dengan *ranking node* yang merepresentasikan *tag* pada setiap klaster.



Gambar 2.2: Smoothed Ranking Function Song et al. (2008)

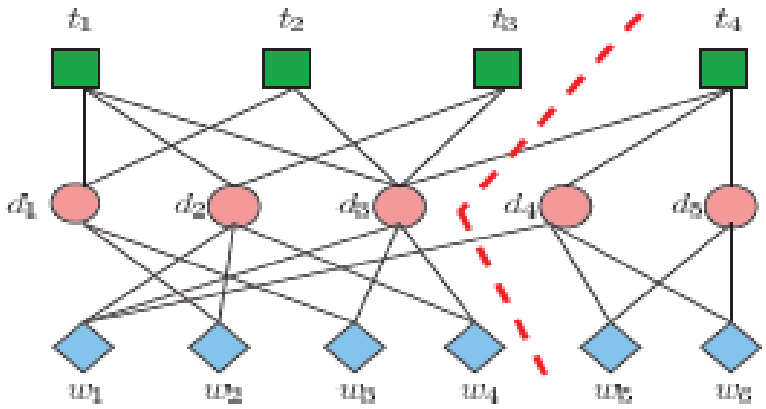
2.2 Online Tag Recommendation

Suatu dokumen biasanya mengandung beberapa kata dan beberapa *tag* yang dibuat oleh *user*. Hubungan antara dokumen, kata-kata, dan *tag* bisa direpresentasikan dengan gambar dua *bipartite graph* yang akan ditunjukkan oleh gambar setelah ini.

Graf yang berbobot dapat ditulis dsebagai berikut

$$W = \begin{pmatrix} 0 & A & 0 \\ A^T & 0 & B \\ 0 & B^T & 0 \end{pmatrix} \tag{2.9}$$

di mana A dan B sebagai matriks-matriks inter-relasi antara *tag* dengan dokumen dan dokumen dengan kata-kata.



Gambar 2.3: Dua bipartite graph dari dokumen-dokumen, kumpulan kata, dan kumpulan tag. Song et al. (2008)

Diberikan suatu representasi matriks, pendekatan lurus untuk *tag* rekomendasi adalah dengan melihat kemiripannya antara dokumen *query* dan dokumen *training* dengan fitur-fitur kata, kemudian lakukan *top ranked tags* dari dokumen yang paling mirip. Pendekatan ini biasanya direferensikan sebagai filter kolaborasi. Namun, pendekatan ini tidaklah efisien untuk skenario dunia nyata. Untuk mengambil kelebihan dari algoritma *node ranking*, Song dkk menggunakan *Possion Mixture Model (PMM)* yang secara efisien menentukan sampel *membership* sebaik klastering kata-kata dengan makna yang mirip. Sebelum melakukan *mixture model*, di sini terdapat rangkuman algoritma yang digunakan untuk rekomendasi *tag* di dalam Algoritma 1. (Song et al. (2008))

Algorithm 1 Online Tag Recommendation (Song et al., 2008)

1: **Input** $(\mathcal{D}, S, T), K, M, L$
Kumpulan dokumen: $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_m\}$
Word vocabulary: $S = \{S_1, \ldots, S_k\}$
Tag vocabulary: $T = \{T_1, \ldots, T_n\}$
banyaknya klaster: $K \in \mathbb{R}$
banyaknya komponen-komponen: $M \in \mathbb{R}$
banyaknya klaster-klaster kata: $L \in \mathbb{R}$

Offline Computation

2: Menunjukkan bobot terdekat matriks W seperti persamaan (2.9)
3: Normalisasi W menggunakan *Normalized Laplacian* persamaan (2.1)
4: Komputasi *low rank approximation matrix* menggunakan Lanczos:
$\tilde{W} \simeq L(W) = Q_k T_k Q_k^T$
5: Partisi $\tilde{W}$ ke dalam klaster K menggunakan SRE, $\tilde{W} = \left\{\tilde{W}_1, \ldots, \tilde{W}_K\right\}$
6: Tandai label ke dalam setiap dokumen $\mathcal{D}_j, j \in \{1, \ldots m\}$
$C(\mathcal{D}_j) \in \{1, \ldots, K\}$
7: Hitung *node rank* $Rank(T)$ untuk setiap tag $T_{i,k}$ di dalam klaster
$k, i \in \{1, \ldots, n\}, k\{1, \ldots, K\}$ persamaan (2.8)
8: Buat *Poisson Mixture Model* untuk $(\tilde{B}, C(\mathcal{D}))$ dengan M komponen-komponen dan L klaster kata-kata, di mana $\tilde{B}$ denotasi matriks inter-relationship pada suatu dokumen-dokumen dan kata-kata di dalam $\tilde{W}$ persamaan (2.9)

Online Recommendation

9: Untuk setiap dokumen tes $\mathbb{Y}$, kalkulasikan posterior probabilitas
$P(C = k \mid D = \mathbb{Y})$ di dalam setiap klaster k, dan denotasi membership pada $\mathbb{Y}$ sebagai $C(\mathbb{Y}) = \{c(\mathbb{Y}, 1), \ldots, c(\mathbb{Y}, K)\}$ persamaan (2.16)
10: Tag rekomendasi berdasarkan perangkingan pada tag, yaitu *joint probability* pada tag-tag T dan dokumen Y, $R(T, \mathbb{Y})$ persamaan (2.17)

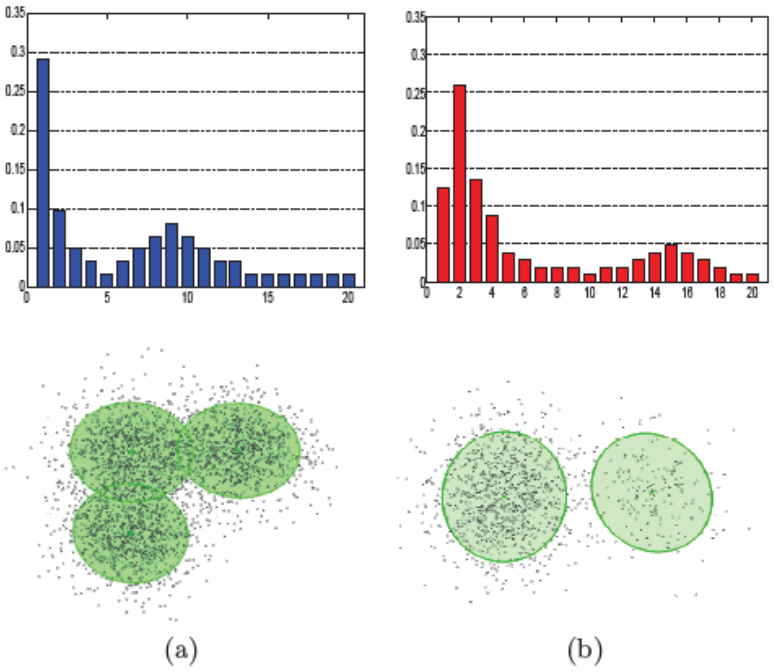
Dua tahap *framework* ini bisa diinterpretasikan sebagai prosedur *unsupervised-supervised learning*. Saat tahap *offline learning stage*, titik-titik akan dipartisi ke dalam klaster-klaster menggunakan *unsupervised learning*, label klaster dipasangkan kepada *document node* sebagai *class label* mereka, dan tag akan diberikan *rank* di dalam setiap klaster. *Mixture model* kemudian dibangun berdasarkan distribusi dari dokumen-dokumen dan kata-kata. Di dalam *online*

recommendation stage, suatu dokumen diklasifikasikan ke dalam *predefined cluster* yang telah didapatkan di dalam tahap pertama oleh Naive Bayes jadi *tag* tersebut bisa direkomendasikan di pengurutan secara terbalik dari rank mereka. Untuk menghindari kebingungan, Song et al. (2008) akan merujuk kepada klaster yang diinginkan dengan mempartisi algoritima di dalam tahap pertama sebagai *classess* di dalam sesi selanjutnya. (Song et al. (2008))

2.2.1 Two-Way Poisson Mixture Model

Song et al. (2008) mengusulkan untuk menggunakan *Poisson Mixture Model* untuk mengestimasi distribusi data pada vektor dokumen sebab algoritma tersebut cocok digunakan dibandingkan *standard Poissons* dengan memproduksi estimasi lebih baik pada data varians dan cukup mudah untuk estimasi parameter. Akan tetapi, itu membutuhkan waktu untuk mencocokkan data latihan. Algoritma ini efisien untuk memprediksi label kelas dari dokumen baru setelah model tersebut selesai dibuat. Karena stabilitas numerikal pada pendekatan statistika ini, biasanya hasilnya dapat diandalkan. Sejak hanya estimasi probabilitas yang dilibatkan, ini dapat diandalkan untuk proses secara *real-time*.

Namun, pendekatan tradisional *unsupervised learning* dari *mixture model* tidak selalu diandalkan untuk menghadapi klasifikasi dokumen. Mempertimbangkan kelemahan dan tingginya dimensi pada matriks *document-word* di mana kebanyakan masukkan berupa 0 dan 1, model bisa saja gagal untuk memprediksi distribusi yang benar (yaitu *probability mass faunction*) pada komponen yang berbeda. Sebagai hasilnya, klasterisasi kata adalah langkah yang diperlukan sebelum mengestimasi komponen-komponen di dalam model. Di sini akan dilakukan *two-way Poisson Mixture Model* untuk secara bersamaan melakukan *cluster word feature* dan klasifikasi dokumen.



Gambar 2.4: Distribusi Poisson dalam dua klaster. Bagian atas menggambarkan histogram dari *mixture components*. Bagian bawah menggambarkan hasil dari klasifikasi *mixture model*. Bagian (a) *three component mixtures* dan bagian (b) *two component mixtures* Song et al. (2008)

Diberikan suatu dokumen $D = \{D_1, ..., D_p\}$, di mana p adalah dimensi, distribusi pada vektor dokumen di setiap kelas dapat diestimasikan dengan menggunakan *parametric mixture model*.

kelas label $C = \{1, 2, ..., K\}$, kemudian

$$P(D = d|C = k) = \sum_{m=1}^{M} \pi_m \Pi(F(m) = k) \prod_{j=1}^{p} \phi(d_j|\lambda_{j,m}), \tag{2.10}$$

di mana π_m adalah *prior probability* dari komponen m, dengan $\sum_{m=1}^{M} \pi_m = 1.\Pi(F(m) = k)$ adalah fungsi indikator yaitu apakah komponen m milik kelas k, dan ϕ merujuk kepada *probability mass function* dari distribusi Poisson, $\phi(d_j|\lambda_{j,m}) = e^{-\lambda_{j,m}} \lambda_{j,m}{}^{d_j}/d_j!$.

Pada jalur ini, setiap kelas adalah *mixture model* dengan distribusi yang multivariasi dengan memiliki variabel yang mengikuti distribusi Poisson. Gambar 2.4 menunjukkan histogram pada dua *mixture* yang bisa dianggap sebagai pmf dari dua *Poisson mixture*.

Asumsi Song et al. (2008) terkait setiap kelas, kata-kata pada dokumen yang berbeda memiliki parameter Poisson yang setara, saat dokumen-dokumen di dalam kelas yang berbeda, kata-kata bisa saja mengikuti perbedaan distribusi Poissson. Untuk mempersimpel, Song et al. (2008) mengasumsi bahwa semua kelas memiliki nomor yang dari klaster-klaster kata. *Denote* $l = \{1,,,,L\}$ untuk menjadi klaster-klaster kata, kata-kata yang sama klaster kata m akan memiliki parameter yang sama yaitu $\lambda_{i,m} = \lambda_{j,m} = \lambda_{l,m}$ untuk $c(i,k) = c(j,k)$ di mana $c(i,k)$ *denote* label klaster pada kata i di dalam kelas k. Berarti, persamaan sebelumnya dapat dipermudah menjadi (dengan $L \ll p$):

$$P(D = d|C = k) \propto \sum_{m=1}^{M} \pi_m \Pi(F(m) = k) \prod_{l=1}^{L} \phi(d_{k,l}|\lambda_{l,m}), \tag{2.11}$$

Estimasi Parameter Dengan kelas yang dideterminasi, berikutnya masukkan algoritma EM untuk mengestimasi parameter Poisson $\lambda_{l,m}, l \in \{1,...,L\}, m \in \{1,...,M\}$, *prior of mixture component* π_m, dan indeks klaster kata $c(k,j) \in \{1,...,L\}, k \in \{1,...,K\}, j \in \{1,...,p\}$.

Estimasi *E-step posterior probability* $p_{i,m}$ sebagai berikut.

$$p_{i,m} \propto \pi_m^{(t)} \mathbb{III}(C(i)) \prod_{j=1}^{p} \theta\left(d(i,j) \mid \tilde{\lambda}_{m,i,j}^{(t)}\right) \tag{2.12}$$

M-step menggunakan $p_{i,m}$ untuk memaksimalkan *objective function*

$$\begin{aligned} &L\left(\pi_m^{(t+1)}, \tilde{\lambda}_{m,l}^{(t+1)}, c^{(t+1)}(k,j) \mid \pi_m^{(t)}, \tilde{\lambda}_{m,l}^{(t)}, c^{(t)}(k,j)\right) \\ &= \max \sum_{i=1}^{n} \sum_{m=1}^{M} p_{i,m} \log\left(\pi_m^{(t+1)} \mathbb{I}(C(i)) \prod_{j=1}^{p} \theta\left(d(i,j) \mid \tilde{\lambda}_{m,i,j}^{(t+1)}\right)\right), \end{aligned} \tag{2.13}$$

dan *update* parameter

$$\pi_m^{(t+1)} = \frac{\sum_{i=1}^{n} p_{i,m}}{\sum_{m'=1}^{M} \sum_{i=1}^{n} p_{i,m'}}, \tag{2.14}$$

$$\tilde{\lambda}_m^{(t+1)} = \frac{\sum_{i=1}^{n} p_{i,m} \sum_j d(i,j) \mathbb{I}(C(i))}{|d(i,j)| \sum_{i=1}^{n} p_{i,m}}, \tag{2.15}$$

di mana $|d(i,j)|$ denotasi bilangan dari j di dalam komponen l.

Setelah $\tilde{\lambda}_m^{(t+1)}$ telah diperbaiki, indeks klaster kata $c^{(t+1)}(k,j)$ bisa ditemukan dengan melakukan *linear search* pada semua komponen-komponen:

$$c^{(t+1)}(k,j) = \arg\max_l \sum_{i=1}^{n} \sum_{m=1}^{M} \log\left(d(i,j) \mid \tilde{\lambda}_{m,l}^{(t+1)}\right)$$

(Song et al. (2008))

2.2.2 Tag Recommendation for New Documents

Normalnya, label kelas $C(d_t)$ dari suatu dokumen baru d_t ditentukan oleh $\hat{C}(x) = \arg\max_k P(C = k \mid D = d_t)$. Namun, di kasus ini, Song et al. (2008) determinasi *mixed membership* pada suatu dokumen dengan menkalkulasi *posterior probabilities* pada suatu kelas dengan $\sum_{k=1}^{K} P(C = k \mid D = d_t) = 1$. Mengaplikasikan persamaan (2.12) dan *Bayes Rule*,

$$\begin{aligned} P(C = k|D = d_t) &= \frac{P(D = d_t|C = k)P(C = k)}{P(D = d_t)} \\ &= \frac{\sum_{m=1}^{M} \pi_m \mathbb{III}(F(m) = k) \prod_{l=1}^{L} \phi(d_{k,l}\tilde{\lambda}_{l,m}) P(C = k)}{P(D = d_t)} \end{aligned} \tag{2.16}$$

di mana $P(C = k)$ adalah *prior probabilites* dari kelas k dan set seragam. Akhirnya, probabilitas untuk setiap *tag* $T_i, i \in \{1, \ldots, n\}$ diasosiakan dengan sampel adalah

$$\begin{aligned} R(T_i, d_t) &= P(T = T_i|D = d_t) \\ &= Rank_{T_i} * P(C = x|D = d_t) \end{aligned} \tag{2.17}$$

Dengan perangkingan *tag* dari terbesar ke terkecil pada probabilitas mereka, *top ranked tags* dipilih untuk rekomendasi. (Song et al. (2008))

2.3 Mixture Model

Salah satu *Mixture Model* yang biasa dipakai yaitu *Gaussian Mixture Model* (*GMM*), *GMM* adalah suatu *parametric probability density function* yang merepresentasikan *weighted sum* dari kepadatan komponen *Gaussian*. Biasanya,

GMM digunakan sebagai *parametric model* dari distribusi probabilitas pada pengukuran yang kontinu. Parameter *GMM* diestimasi dari data latih menggunakan algoritma *Expectation-Maximization* (*EM*).

Gaussian Mixture Model adalah *weighted sum* dari *M* kepadatan komponen *Gaussian* dengan persamaan sebagai berikut.

$$p(x|\lambda) = \sum_{i=1}^{M} w_i g(x|\mu_i, \Sigma_i) \tag{2.18}$$

Di mana x adalah D *dimensional continuous* vektor data, w_i, i = 1, ...,M adalah bobot mikstur, dan $g(x|miu_i, \Sigma_i)$, i = 1,...,M adalah kepadatan komponen *Gaussian*. Kepadatan komponen *Gaussian* adalah sebagai berikut.

$$g(x|\mu_i, \Sigma_i) = \frac{1}{(2\Phi)^{D/2}|\Sigma_i|^{1/2}} exp\{-\frac{(x-\mu_i)'\Sigma_i^{-1}(x-\mu_i)}{2}\} \tag{2.19}$$

dengan μ_i sebagai *mean vector* dan σ_i sebagai matriks kovarian. *Mixture Weights* memenuhi persamaan $\sum_{i=1}^{M} w_i = 1$.

Beberapa parameter tersebut dikumpulkan menjadi persamaan berikut.

$$\lambda = \{w_i, \mu_i, \sigma_i\} i = 1, ..., M \tag{2.20}$$

Pada tahap selanjutnya adalah menghitung *Maximum Likelihood Parameter*. Diberikan data latih berupa vektor-vektor dan konfigurasi *GMM*. Dari sini akan dilakukan estimasi dari parameter *GMM* λ. Metode paling populer dalam menentukan ini adalah estimasi *Maximum Likelihood*.

Tujuan *Maximum Likelihood* adalah mencari parameter model yang memaksimalkan *likelihood* pada GMM dari data latih yang diberikan. Untuk setiap T *training vector* $X = \{x_1, ..., x_T\}$, *GMM likelihood*, asumsikan antar vektor adalah independen, maka sebagai berikut.

$$p(X|\lambda) = \prod_{t=1}^{T} p(x_t|\lambda) \tag{2.21}$$

Sayangnya, persamaan ini adalah fungsi tidak linear dari parameter λ dan tidak mungkin untuk melakukan pemaksimalan secara langsung. Namun, parameter *Maximum Likelihood* dapat diestimasikan dengan cara iteratif dengan menggunakan

algoritma *Expectation-maximization* (*EM*).

Ide dasar dari algoritma *EM* adalah memulai model awal λ untuk mengistmasi model baru $\tilde{\lambda}$ lalu menjadi $p(X|\tilde{\lambda}) \geq p(X|\lambda)$. Model baru akan menjadi model awal untuk iterasi selanjutnya dan proses tersebut akan berlanjut sampai batas tertentu yang ditetapkan. Model awal biasanya *derived* dengan menggunakan *VQ estimation*. (Reynolds (2009))

Dalam setiap iterasi *EM*, formula reestimasi selanjutnya akan digunakan untuk meningkatkan nilai *likelihood*.

Mixture Weights

$$\tilde{w}_i = \frac{1}{T}\sum_{t=1}^{T} Pr(i|x_t, \lambda) \tag{2.22}$$

Means

$$\tilde{\mu}_i = \frac{\sum_{t=1}^{T} Pr(i|x_t, \lambda)x_t}{\sum_{t=1}^{T} Pr(i|x_t, \lambda)} \tag{2.23}$$

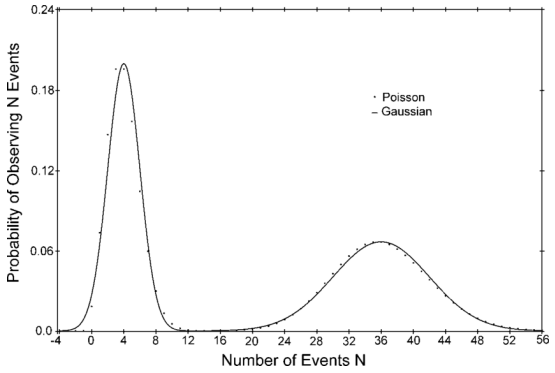
Varians

$$\tilde{\sigma}_i^2 = \frac{\sum_{t=1}^{T} Pr(i|x_t, \lambda)x_t^2}{\sum_{t=1}^{T} Pr(i|x_t, \lambda)} - \tilde{\mu_i^2} \tag{2.24}$$

Kemudian *posterior probability* untuk komponen i sebagai berikut.

$$Pr(i|x_t, \lambda) = \frac{w_i g(x_t|\mu_i, \Sigma_i)}{\sum_{k=1}^{M} w_k g(x_t|\mu_i, \Sigma_i)} \tag{2.25}$$

Perbedaan antara Distribusi Gaussian dengan Distribusi Poisson sebagai berikut.

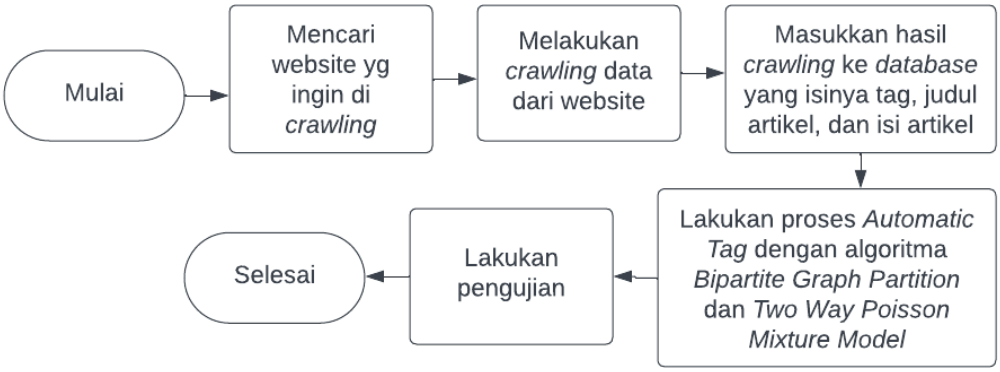


Gambar 2.5: Perbedaan antara Distribusi Gaussian dengan Distribusi Poisson Rzeszotarski (1999)

BAB III
DESAIN MODEL

3.1 Tahapan Penelitian

Berikut merupakan *flowchart* dari tahapan penelitian *Automatic Tagging* dengan menggunakan algoritma *Bipartite Graph Partition* dan *Two Way Poisson Mixture Model*.



Gambar 3.1: Flowchart alur penelitian

Mencari salah satu situs yang ingin di-*crawling* terlebih dahulu. Kemudian, lakukan *crawling* data dari situs tersebut menggunakan *crawler* dari penelitian Khatulistiwa (2022). Hasil *crawling* tersebut dimasukkan ke dalam *database*.

Kemudian, data tersebut diolah menggunakan algoritma *Bipartite Graph Partition* dan *Two Way Poisson Mixture Model* untuk melatih program supaya bisa memprediksi *tag* yang diinginkan. Setelah itu, lakukan pengujian terhadap *tag* yang telah dihasilkan.

3.2 Algoritma *Automatic Tag*

Berikut adalah algoritma dari *Online Tag Recommendation*

Algorithm 2 Online Tag Recommendation (Song et al., 2008)

1: **Input** $(\mathcal{D}, S, T), K, M, L$

Kumpulan dokumen: $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_m\}$

Word vocabulary: $S = \{S_1, \ldots, S_k\}$

Tag vocabulary: $T = \{T_1, \ldots, T_n\}$

banyaknya klaster: $K \in \mathbb{R}$

banyaknya komponen-komponen: $M \in \mathbb{R}$

banyaknya klaster-klaster kata: $L \in \mathbb{R}$

Offline Computation

2: Menunjukkan bobot terdekat matriks W seperti persamaan (2.9)

3: Normalisasi W menggunakan *Normalized Laplacian* persamaan (2.1)

4: Komputasi *low rank approximation matrix* menggunakan Lanczos:
$\tilde{W} \simeq L(W) = Q_k T_k Q_k^T$

5: Partisi $\tilde{W}$ ke dalam klaster K menggunakan SRE, $\tilde{W} = \left\{\tilde{W}_1, \ldots, \tilde{W}_K\right\}$

6: Tandai label ke dalam setiap dokumen $\mathcal{D}_j, j \in \{1, \ldots m\}$
$C(\mathcal{D}_j) \in \{1, \ldots, K\}$

7: Hitung *node rank* $Rank(T)$ untuk setiap tag $T_{i,k}$ di dalam klaster
$k, i \in \{1, \ldots, n\}, k\{1, \ldots, K\}$ persamaan (2.8)

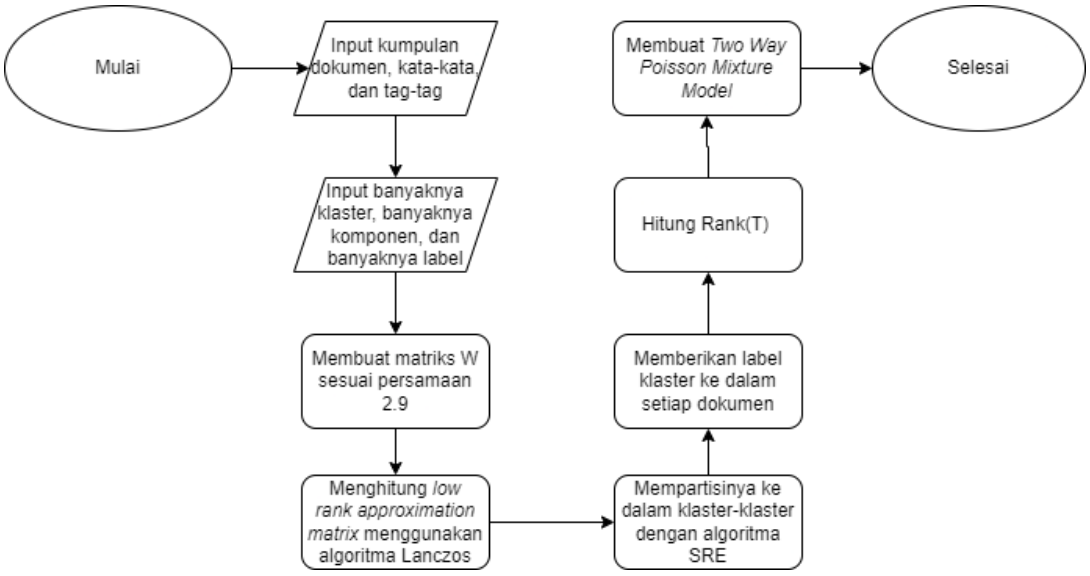
8: Buat *Poisson Mixture Model* untuk $(\tilde{B}, C(\mathcal{D}))$ dengan M komponen-komponen dan L klaster kata-kata, di mana $\tilde{B}$ denotasi matriks inter-relationship pada suatu dokumen-dokumen dan kata-kata di dalam $\tilde{W}$ persamaan (2.9)

Online Recommendation

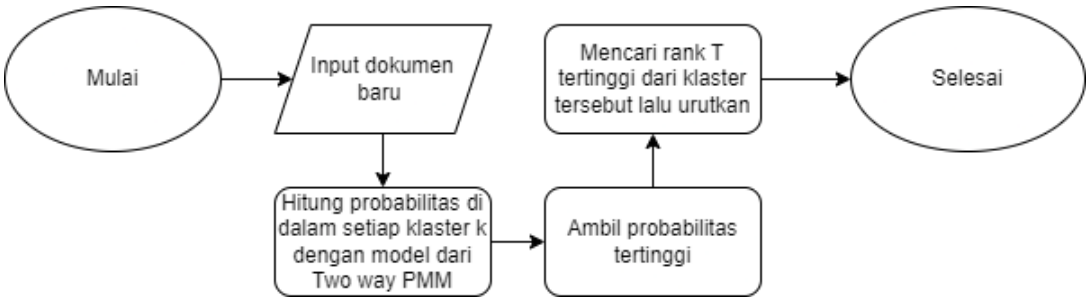
9: Untuk setiap dokumen tes $\mathbb{Y}$, kalkulasikan posterior probabilitas
$P(C = k \mid D = \mathbb{Y})$ di dalam setiap klaster k, dan denotasi membership pada $\mathbb{Y}$ sebagai $C(\mathbb{Y}) = \{c(\mathbb{Y}, 1), \ldots, c(\mathbb{Y}, K)\}$ persamaan (2.16)

10: Tag rekomendasi berdasarkan perangkingan pada tag, yaitu *joint probability* pada tag-tag T dan dokumen Y, $R(T, \mathbb{Y})$ persamaan (2.17)

3.3 Flowchart Automatic Tag



Gambar 3.2: Diagram alir untuk tahap *offline computation*



Gambar 3.3: Melakukan *online recommendation* berdasarkan hasil data training

3.4 Alat dan Bahan Penelitian

Dalam penelitian ini, setidaknya ada perangkat keras sebagai berikut.

1. Laptop dengan prosesor Intel Core i5 8th Gen *series* dan *RAM* 16 *GB*.
2. Koneksi berbasis Wi-Fi dan berbasis kuota internet dari ponsel pintar.

Perangkat lunaknya sebagai berikut.

1. Windows 10 64 bit OS.

2. Visual Studio Code sebagai *Code Editor*.

3. Python 3 untuk menjalankan program Python.

4. Sumber data berasal dari *Thehill.com* .

3.5 Tahapan Penelitian Automatic Tag

Setidaknya terdapat 10 tahapan yang perlu dijalankan agar *automatic tag* ini akan berjalan dengan baik yaitu

3.5.1 Penginputan

Pada tahap ini, akan ditentukan enam komponen atau inputan yang diperlukan adalah dokumen (D), *word vocabulary* (S), *Tag vocabulary* (T), banyaknya klaster (K), banyaknya komponen-komponen (M), dan banyaknya klaster-klaster kata (L).

Untuk dokumen (D) yang diinput, berasal dari data hasil *crawling* dengan menggunakan *crawler* milik Lazuardy Khatulistiwa berjudul Perancangan Arsitektur *Search Engine* dengan Mengintegrasikan *Web Crawler*, Algoritma *Page Ranking*, dan *Document Ranking*.

Untuk *word vocabulary* (S), kata-kata yang diinput merupakan kata-kata yang berasal dari dokumen hasil *crawling*.

Pada *tag vocabulary* (T), *tag* yang didapat berasal dari dokumen *tag* hasil *crawling*.

Pada banyaknya klaster (K), digunakan untuk menentukan berapa klaster yang ingin dibuat saat menjalankan algoritma *SRE*. Banyaknya klaster ini ditentukan sesuai dengan keinginan sang pengguna.

Banyaknya komponen-komponen (M), biasanya M ini akan digunakan pada algoritma *Two Way Poisson Mixture Model*.

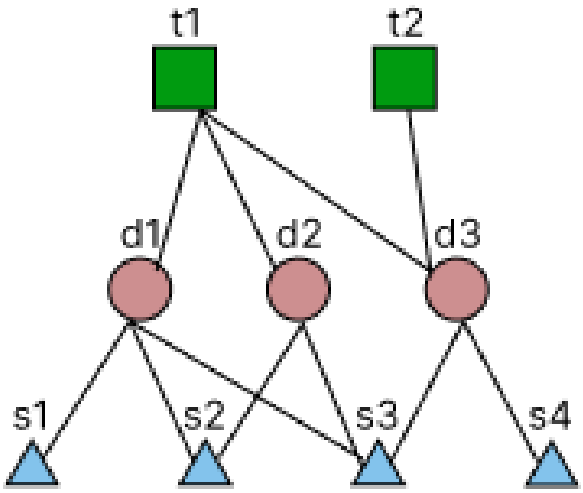
Terakhir adalah banyaknya klaster kata (L) untuk melakukan pelabelan (L). Variabel ini akan digunakan pada algoritma *Two Way Poisson Mixture Model*.

3.5.2 Menentukan Matriks W

Untuk matriks W akan terdiri dari beragam matriks yaitu matriks A, matriks A^T, matriks B, dan matriks B^T.

Untuk matriks A, terbuat dari relasi antara *tag vocabulary* dengan dokumen, sedangkan untuk matriks B diperoleh dari relasi antara dokumen dengan *word vocabulary*.

Untuk membuat matriks W akan menggunakan persamaan (2.9). Untuk contohnya, akan mengambil gambar berikut.



Gambar 3.4: Contoh simpel dua *bipartite graph*

Relasi antara t (*tag*) dengan d (*document*) akan membentuk matriks A dan A^T sebagai berikut.

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.1}$$

$$A^T = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \tag{3.2}$$

Selanjutnya, mencari relasi antara D (*document*) dengan S (*words*) agar membentuk **matriks** B sebagai berikut.

$$B = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \tag{3.3}$$

$$B^T = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.4}$$

Selanjutnya, adanya penggabungan yang nantinya membentuk **matriks** W sebagai berikut

$$W = \begin{pmatrix} 0 & A & 0 \\ A^T & 0 & B \\ 0 & B^T & 0 \end{pmatrix}$$

Nantinya, matriks A dan matriks B ditempatkan ke dalam matriks W. Angka 0 di atas merupakan matriks 0 yang isinya menyesuaikan baris dan kolom matriks sekitarnya.

$$W = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{3.5}$$

3.5.3 Menghitung *Low Rank Approximation Matrix* menggunakan algoritma Lanczos

Pada algoritma Lanczos yang terdapat di (Golub & Loan (1996)), ada beberapa inputan agar prosesnya bisa berjalan yaitu β_0, q_0, b, dan q_1. Berikut adalah beberapa inputannya.

$$\beta_0 = 0$$

$$q_0 = 0$$

$$b = arbitrary$$

Karena b adalah *arbitrary* yang memiliki makna bahwa nilai yang diinput itu bebas. Oleh karena itu, b dibuat menjadi

$$b = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Selanjutnya nilai q_1 adalah

$$q_1 = b/||b|| = \begin{pmatrix} 0.33 & 0.33 & 0.33 & 0.33 & 0.33 & 0.33 & 0.33 & 0.33 & 0.33 \end{pmatrix}$$

$$\hat{W} \simeq L(W) = Q_k T_k Q_k^T \tag{3.6}$$

Karena persamaan ini, hasil dari $L(W)$ juga bisa dicari dengan menggunakan $\hat{W}$. Selain itu, algoritma Lanczos dinilai lebih efisien untuk data yang banyak dibandingkan menggunakan normalisasi Laplacian. Untuk mencari Q_k dan T_k, perlu menggunakan iterasi Lanczos.

Untuk matriks T_k memiliki format sebagai berikut

$$T_k = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \beta_2 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \beta_{k-1} \\ & & & \beta_{k-1} & \alpha_k \end{bmatrix}.$$

Dalam matriks di atas, diperlukan pencarian nilai α dan β agar bisa melengkapi hasil T_k.

Untuk matriks Q_k didapat dari

$$Q_k = \begin{pmatrix} q_1 & | & q_2 & | & \dots & | & q_k \end{pmatrix}$$

q_1 adalah matriks kolom yang ke-1. q_2 adalah matriks kolom yang ke-2. q_k

adalah matriks kolom yang ke-k. q_1 sampai q_k akan didapat melalui iterasi Lanczos sebagai berikut.

Pada algoritma di bawah, asumsikan bahwa $n = k$ dan $A = W$.

Algorithm 3 *Lanczos Iteration* (Golub & Loan, 1996)

Require: $\beta_0 = 0$, $q_0 = 0$, $b = arbitrary$, $q_1 = b/||b||$

while $n = 1, 2, 3, ..$ **do**

$v = Aq_n$

$\alpha_n = q_n^T v$

$v = v - \beta_{n-1} q_{n-1} - \alpha_n q_n$

$\beta_n = ||v||$

$q_{n+1} = v/\beta_n$

end while

3.5.4 Melakukan partisi $\hat{W}$ ke dalam klaster K menggunakan *SRE*

Untuk membuat algoritma *SRE* berdasarkan Zha et al. (2001), ada lima langkah yang diperlukan.

Pertama adalah menghitung D_X dan D_Y dan membentuk matriks $\hat{W} = D_X^{-1/2} W D_Y^{-1/2}$. Pada langkah ini, matriks $\hat{W}$ telah dibuat dengan menggunakan algoritma Lanczos atau *normalized Laplacian*. Alasannya adalah pola untuk mendapatkan $\hat{W}$ yang mirip.

Langkah kedua ialah mencari vektor terluas kedua singular kiri dan singular kanan dengan menggunakan *Singular Vector Decomposition* (*SVD*) .

Dalam algoritma *SRE*, langkah ketiga adalah mencari titik potong yaitu c_x dan c_y untuk $x = D_X^{-1/2}\hat{x}$ dan $y = D_Y^{-1/2}\hat{y}$. Untuk mencari keduanya, strategi tersimpelnya adalah dengan memasang $c_x = 0$ dan $c_y = 0$.

Langkah keempatnya membentuk suatu partisi dengan $A = \{i|x_i \geq c_x\}$ dan $A^C = \{i|x_i < c_x\}$ untuk *vertex* X dan $B = \{i|y_j \geq c_y\}$ dan $B^C = \{j|y_j < c_x\}$ untuk *vertex* Y.

Langkah kelima adalah mengulangi partisi pada *sub graph* $G(A, B)$ dan $G(A^c, B^c)$ jika diperlukan.

Berikut ini adalah algoritma *Spectral Recursive Embedding* atau *SRE* dari (Zha et al., 2001)

Algorithm 4 *Spectral Recursive Embedding (SRE)* (Zha et al., 2001)

Diberikan *weighted bipartite graph* $G = (X, Y, E)$ dengan bobot garis matriks W

1. Komputasi D_x dan D_y dan bentuk *scaled weight matrix* $\hat{(W)} = D_x^{-1/2} W D_y^{-1/2}$
2. Hitung singular vektor terluas kiri dan kanan kedua dari vektor $\hat{W}$, $\hat{x}$, dan $\hat{y}$
3. Temukan titik potong c_x dan c_y untuk $x = D_X^{-1/2}\hat{x}$ dan $y = D_Y^{-1/2}\hat{y}$, secara berulang.
4. Bentuk partisi $A = \{i|x_i \geq c_x\}$ dan $A^c = \{i|x_i < c_x\}$ untuk verteks set X, dan $B = \{j|y_j \geq c_y\}$ dan $B^c = \{j|y_j < c_y\}$ untuk verteks set Y.
5. Lakukan partisi secara rekursif untuk *sub-graphs* $G(A, B)$ dan $G(A^c, B^c)$

3.5.5 Melakukan pelabelan setiap dokumen

Selanjutnya adalah memberikan label kepada setiap dokumen yang ada sesuai pembagian klasternya. Misalnya, terdapat 4 dokumen yaitu D_1, D_2, D_3, dan D_4 dan ada 2 klaster sesuai jumlah K. Nantinya, dokumen tersebut dimasukkan ke dalam klaster sesuai dengan hasil partisi $\hat{W}$ sesuai dengan banyaknya klaster yang dibentuk.

3.5.6 Menghitung *Node Rank* Rank(T) untuk Setiap Tag

Cara menghitung Rank(T) untuk setiap tag T_i di dalam klaster k dengan menggunakan persamaan (2.6), persamaan (2.7), dan persamaan (2.8).

Untuk persamaan (2.6) adalah menghitung *N-Precision* np_i. Untuk mencari np_i, diharuskan menghitung total seluruh bobot *edges* dari *node* i di dalam klaster yang sama, lalu hasilnya dibagi dengan total dari seluruh bobot pada *edges* di klaster tersebut. Semakin besar nilai np_i-nya, semakin penting keberadaan *tag* di dalam klaster tersebut.

Untuk persamaan (2.7) adalah menghitung *N-recall*. $|E_i|$ didapat dari banyaknya garis yang terhubung ke Tag i T_i baik itu di dalam klaster maupun di luar klaster. Lalu, hasilnya dibagi dengan banyaknya garis yang terhubung dengan T_i yang di luar klaster.

Setelah berhasil menghitung np_i dan nr_i, $Rank_i$ dapat dihitung dengan $exp(\frac{-1}{r(i)^2})$ untuk $r(i) = np_i * log(nr_i)$ dengan syarat $r(i)$ tidak boleh sama dengan 0. Jika $r(i) = 0$, $Rank_i$ hasilnya adalah 0 berdasarkan persamaan (2.8).

3.5.7 Membuat Two Way Poisson Mixture Model

Untuk membuat salah satu komponen dari *Two Way Poisson Mixture Model*, diperlukan persamaan (2.11). Salah satu komponen dari persaman tersebut adalah ϕ_m yang didapat dari persamaan (2.15).

Untuk $II(F(m) = k)$ adalah suatu *indicator function*. Jika komponen m milik klaster k, berarti bernilai 1. Dalam kondisi sebaliknya, bernilai 0.

Untuk mengisi t pada persamaan (2.14), inisialisasi $t = 0$ (Li & Zha (2004)). Kemudian, saat mencari persamaan (2.14), terdapat komponen $p_{i,m}$ yang perlu dicari terlebih dahulu dengan menggunakan persamaan (2.11).

Pada $p_{i,m}$ terdapat komponen $\phi_m^{(t)}$. Kemudian, $d(i, j)$ didapat dari berapa banyaknya kata j di dalam dokumen i. Hal ini ada hubungannya dengan matriks B dari persamaan (2.9). Mencari θ dapat dicari dengan cara $\theta(d_j|\lambda_{j,m}) = e^{-\lambda_{j,m}}\lambda_{j,m}^{d_j}/d_j!$. Untuk p, ia adalah *dimension*. Selain itu, $IIC(i)$ untuk mengecek apakah kata i ada di klaster tersebut.

Selain itu, di persamaan (2.11) terdapat komponen $\tilde{\lambda}_{m,l}$ yang bisa ditemukan di persamaan (2.15). Pada variabel $|d(i, j)|$ dapat dicari dengan menentukan apakah j ada di label l.

3.5.8 Rekomendasi Tag Untuk Dokumen Baru

Lakukan melalui persamaan (2.16) lalu untuk mencari $\frac{P(D=d_t|C=k)P(C=k)}{P(D=d_t)}$ bisa ditemukan dengan persamaan (2.11). Lalu, masukan dokumen yang baru ke dalam suatu klaster yang memiliki probabilitas terbesar.

3.5.9 Rekomendasi Tag Berdasarkan Ranks Tag

Setelah melakukan perhitungan $P(C = k|D = Y)$, langkah berikutnya adalah merekomendasikan *tag-tag* berdasarkan klaster dari dokumen tersebut. Cara melakukannya adalah dengan menggunakan persamaan (2.17) pada setiap *tag* yang ada di klaster tersebut. Kemudian, lakukan pengurutan $R(T_i, d_t)$ dari yang terbesar ke yang terkecil.

3.6 Skenario Pengujian

Berikut adalah skenario pengujian untuk *Automatic Tag Recommendation* dengan Algoritma *Bipartite Graph Partition* dan *Two Way Poisson Mixture Model*.

1. Sumber data yang didapat dari hasil *crawling* dibagi menjadi dua bagian yaitu untuk data latih dan data uji.
2. Setelah seluruh proses algoritmanya berjalan, data uji mulai dimasukkan.
3. Program akan memberikan beberapa prediksi *tag* pada data uji.
4. Kemudian, menghitung banyaknya *tag* yang cocok dengan data uji.

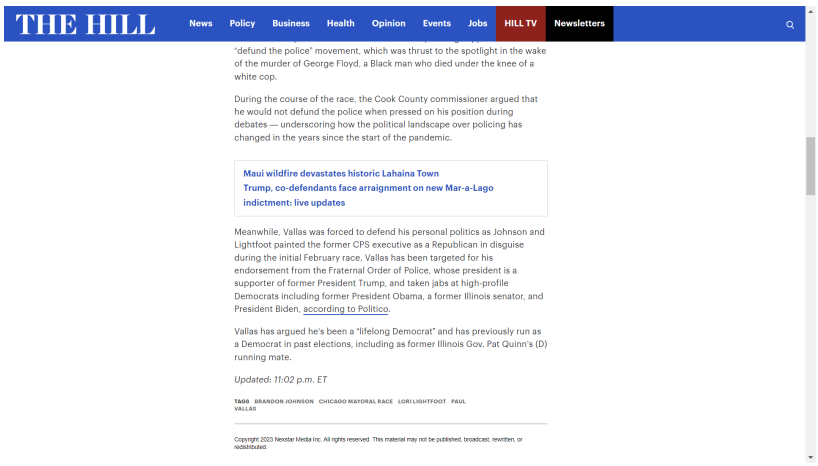
BAB IV

HASIL DAN PEMBAHASAN

4.1 Implementasi

4.1.1 Hasil Crawling

Dalam penelitian ini, situs yang digunakan adalah *thehill.com* karena artikel dari situs tersebut menyediakan *tag* di dalam artikel. Selain itu, adanya variasi *tag* di dalam situs tersebut juga menjadi alasan menggunakan situs ini sebagai bahan untuk penelitian.



Gambar 4.1: Contoh *tag* dari artikel thehill.com

Untuk mendapatkan data-data seperti itu, penulis menggunakan *crawler*. Dalam *table* "*page_informations*", kolom yang dipakai nantinya adalah kolom *id_page*, *content_article*, dan *title*. Kemudian, di dalam *page_tags*, kolom yang digunakan adalah kolom *tag* dan *page_id*.

Table	Action	Rows	Type	Collation	Size	Overhead
crawling	Browse Structure Search Insert Empty Drop	1	InnoDB	latin1_swedish_ci	16.0 KiB	-
pagerank	Browse Structure Search Insert Empty Drop	0	InnoDB	latin1_swedish_ci	32.0 KiB	-
pagerank_changes	Browse Structure Search Insert Empty Drop	0	InnoDB	latin1_swedish_ci	32.0 KiB	-
page_forms	Browse Structure Search Insert Empty Drop	0	InnoDB	latin1_swedish_ci	32.0 KiB	-
page_images	Browse Structure Search Insert Empty Drop	0	InnoDB	latin1_swedish_ci	32.0 KiB	-
page_information	Browse Structure Search Insert Empty Drop	493	InnoDB	latin1_swedish_ci	4.5 MiB	-
page_linking	Browse Structure Search Insert Empty Drop	~120,327	InnoDB	latin1_swedish_ci	15.0 MiB	-
page_list	Browse Structure Search Insert Empty Drop	0	InnoDB	latin1_swedish_ci	32.0 KiB	-
page_scripts	Browse Structure Search Insert Empty Drop	0	InnoDB	latin1_swedish_ci	32.0 KiB	-
page_styles	Browse Structure Search Insert Empty Drop	0	InnoDB	latin1_swedish_ci	32.0 KiB	-
page_tables	Browse Structure Search Insert Empty Drop	0	InnoDB	latin1_swedish_ci	32.0 KiB	-
page_tags	Browse Structure Search Insert Empty Drop	1,478	InnoDB	latin1_swedish_ci	144.0 KiB	-
tfidf	Browse Structure Search Insert Empty Drop	0	InnoDB	latin1_swedish_ci	32.0 KiB	-
tfidf_word	Browse Structure Search Insert Empty Drop	0	InnoDB	latin1_swedish_ci	32.0 KiB	-

Gambar 4.2: Dataset yang digunakan

	id_tag	page_id	tag
Edit Copy Delete	1	78	alvin bragg
Edit Copy Delete	2	78	alvin bragg
Edit Copy Delete	3	78	donald trump
Edit Copy Delete	4	78	kentucky
Edit Copy Delete	5	78	kevin mccarthy
Edit Copy Delete	6	78	mitch mcconnell
Edit Copy Delete	7	78	mitch mcconnell
Edit Copy Delete	8	78	mitt romney
Edit Copy Delete	9	78	new york
Edit Copy Delete	10	78	trump indictment
Edit Copy Delete	11	79	2024

Gambar 4.3: Dataset pada tabel *page_tags*

	id_page	crawl_id	url	html5	title	description	keywords	content_article	content_text	hot_url
Edit Copy Delete	76	1	https://thehill.com/sponsored-content/?ntv_adpz=37...	0	The Hill	-	-	NULL	Skip to content Toggle Menu Close search form News...	0
Edit Copy Delete	77	1	https://thehill.com/sponsored-content/?prx_t=DjsIA...	0	The Hill	-	-	NULL	Skip to content Toggle Menu Close search form News...	0
Edit Copy Delete	78	1	https://thehill.com/homenews/senate/3934484-mcconn...	1	McConnell lets an indicted Trump twist in the wind...	Senate Republican Leader Mitch McConnell (Ky.) and...	Alvin Bragg, Donald Trump, Kentucky, Kevin McCarth...	Senate Republican Leader Mitch McConnell (Ky.) a...	Senate Alexander Bolton - 04/05/23 6:00 AM ET Shar...	0
Edit Copy Delete	79	1	https://thehill.com/newsletters/morning-report/393...	1	The Hill's Morning Report — Trump: 'There is no cr...	Editor's note: The Hill's Morning Report is our da...	-	Editor's note: The Hill's Morning Report is our ...	Morning Report Alexis Simendinger and Kristina Kar...	0
Edit Copy Delete	80	1	https://thehill.com/author/alexis-simendinger	1	Alexis Simendinger - Author at The Hill | Page 1	Find all of the posts authored by Alexis Simending...	-	NULL	/,	0

Gambar 4.4: Dataset pada tabel *page_informations*

4.1.2 Pengambilan Data dari Database

Data-data yang diambil adalah *tag*, *id* dari artikel, judul artikel, dan isi artikel melalui fungsi *get_data* pada *file data_from_database.py*.

```
dataset_document
(('alvin bragg', 78, '\n\nSenate Republican ...pitol. \xa0\n\n', 'McConnell lets an in...| The Hill'), ('donald trump', 78, '\n\nSenate Republican ...
pitol. \xa0\n\n', 'McConnell lets an in...| The Hill'), ('kentucky', 78, '\n\nSenate Republican ...pitol. \xa0\n\n', 'McConnell lets an in...| The Hil
l'), ('kevin mccarthy', 78, '\n\nSenate Republican ...pitol. \xa0\n\n', 'McConnell lets an in...| The Hill'), ('mitch mcconnell', 78, '\n\nSenate Repu
blican ...pitol. \xa0\n\n', 'McConnell lets an in...| The Hill'), ('mitt romney', 78, '\n\nSenate Republican ...pitol. \xa0\n\n', 'McConnell lets an i
n...| The Hill'), ('new york', 78, '\n\nSenate Republican ...pitol. \xa0\n\n', 'McConnell lets an in...| The Hill'), ('trump indictment', 78, '\n\nSen
ate Republican ...pitol. \xa0\n\n', 'McConnell lets an in...| The Hill'), ('2024', 79, '\n\nEditor's note: The...riends!\n\n\n', 'The Hill's Morning
R...| The Hill'), ('alvin bragg', 79, '\n\nEditor's note: The...riends!\n\n\n', 'The Hill's Morning R...| The Hill'), ('car prices', 79, '\n\nEditor's
note: The...riends!\n\n\n', 'The Hill's Morning R...| The Hill'), ('chicago', 79, '\n\nEditor's note: The...riends!\n\n\n', 'The Hill's Morning R...|
The Hill'), ('desantis', 79, '\n\nEditor's note: The...riends!\n\n\n', 'The Hill's Morning R...| The Hill'), ('donald trump', 79, '\n\nEditor's note:
The...riends!\n\n\n', 'The Hill's Morning R...| The Hill'), ...)
```

Gambar 4.5: Dataset yang digunakan

4.1.3 Penginputan

Untuk inputnya, D didapat dari *id* dan judul dari artikel yang ada. T didapat dari *tag* yang telah dikumpulkan. Lalu, S didapat dari kata yang unik dari isi-isi artikel. Selanjutnya, K bernilai 2, M bernilai 2, dan L bernilai 2.

4.1.4 Mengolah Data Menjadi Matriks

Data-data tersebut nantinya akan diolah menjadi matriks berdasarkan persamaan (2.9) yang di mana matriks A adalah relasi antara *tag* dengan dokumen dan matriks B adalah relasi antara dokumen dengan *word*. Untuk melakukan proses ini, diperlukan fungsi *document_processing* pada *file input_processing.py*.

```
matrix_tag_document
array([[1., 1., 1., ..., 1., 0., 0.],
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 0., 0., 1.]])
matrix_document_word
array([[22., 12.,  8., ...,  0.,  0.,  0.],
       [40.,  1.,  5., ...,  0.,  0.,  0.],
       [19.,  0.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 4.,  0.,  1., ...,  0.,  0.,  0.],
       [ 6.,  0.,  1., ...,  0.,  0.,  0.],
       [10.,  0.,  1., ...,  1.,  1.,  1.]])
```

Gambar 4.6: Matrix Tag Document dan Matrix Document Word

Lalu, kedua matriks tersebut disatukan hingga menjadi matriks W berdasarkan persamaan (2.9) denggan menggunakan fungsi *matrixABtoW* pada *file matrix_processing.py*.

```
matrix_w
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

Gambar 4.7: Matrix W

4.1.5 Menghitung Low Rank Approximation Matrix Menggunakan Algoritma Lanczos

Untuk merubah matriks W menjadi matriks $\tilde{W}$, diperlukan matriks Q_k dan T_k. Algoritma *Lanczos Iteration* dapat digunakan untuk mencari nilai dari matriks tersebut.

Dalam mencari nilai matriks Q_k dan T_k, diperlukan *function lanczos_iteration* pada *file low_rank_approximation_matrix.py*. Hasilnya adalah sebagai berikut.

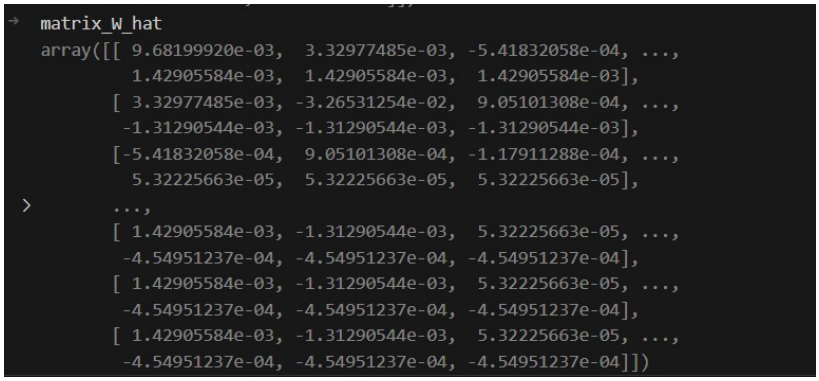
```
matrix_Q
array([[ 9.51216909e-03,  3.58878687e-03,  8.26741577e-03, ...,
         1.46193767e-03,  2.13908093e-02, -1.80002540e-03],
       [ 9.51216909e-03,  9.12402083e-03,  1.30905755e-02, ...,
        -3.70160939e-03,  2.50930092e-02,  3.65486351e-03],
       [ 9.51216909e-03, -1.94644709e-03, -1.82925872e-03, ...,
         6.26298668e-05,  7.58647357e-04, -1.44779253e-04],
       ...,
       [ 9.51216909e-03, -1.94644709e-03, -2.28168414e-03, ...,
        -5.61618922e-04,  4.74527232e-05,  6.30766058e-04],
       [ 9.51216909e-03, -1.94644709e-03, -2.28168414e-03, ...,
        -5.61618922e-04,  4.74527232e-05,  6.30766058e-04],
       [ 9.51216909e-03, -1.94644709e-03, -2.28168414e-03, ...,
        -5.61618922e-04,  4.74527232e-05,  6.30766058e-04]])
```

Gambar 4.8: Matriks Q

```
matrix_T
array([[ 13.30763663,  60.14667501,   0.        , ...,   0.        ,
          0.        ,   0.        ],
       [ 60.14667501, 112.46732951, 143.31934023, ...,   0.        ,
          0.        ,   0.        ],
       [  0.        , 143.31934023, -90.87444223, ...,   0.        ,
          0.        ,   0.        ],
       ...,
       [  0.        ,   0.        ,   0.        , ..., -66.01256203,
        142.60793428,   0.        ],
       [  0.        ,   0.        ,   0.        , ..., 142.60793428,
          1.34209621, 137.59667632],
       [  0.        ,   0.        ,   0.        , ...,   0.        ,
        137.59667632,   0.        ]])
```

Gambar 4.9: Matriks T

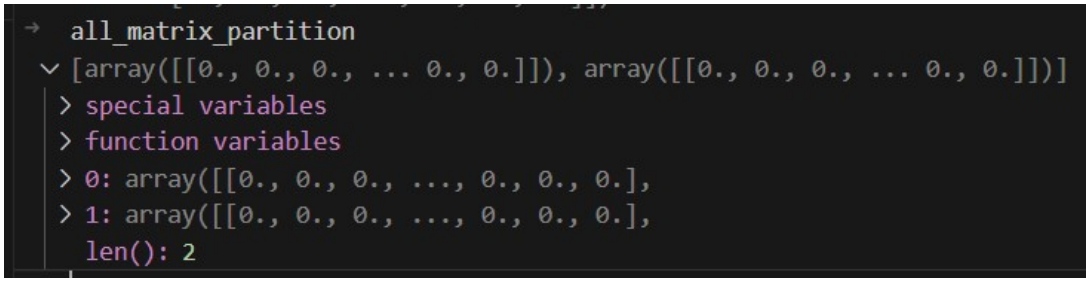
Selanjutnya, mencari nilai dari matriks $\tilde{W}$ dengan menggunakan *function low_rank_approximation_matrix* pada *file low_rank_approximation_matrix.py*. Hasilnya bisa dilihat di bawah ini.



Gambar 4.10: Matriks $\tilde{W}$

4.1.6 Melakukan Partisi W ke dalam Klaster K

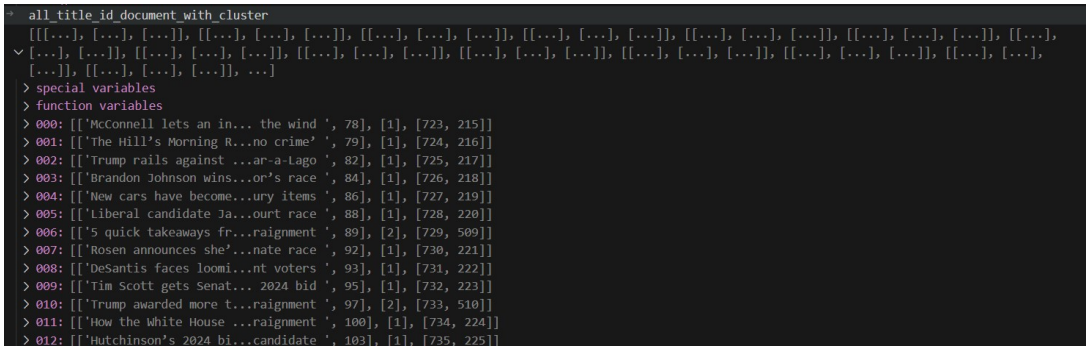
Agar dapat melakukan *Bipartite Graph Partition*, perlu algoritma bernama *Spectral Recursive Embedding*. Hasil dari partisi ini adalah membentuk dua *graph* terbaru dalam bentuk matriks. Untuk partisinya, ditentukan oleh banyaknya K yang telah diatur. Partisi ini menggunakan fungsi *spectral_recursive_embedding* pada *file spectral_recursive_embedding.py*.



Gambar 4.11: Matriks hasil partisi

4.1.7 Melakukan Pelabelan Setiap Dokumen

Setiap dokumen dilabelkan berdasarkan hasil partisi yang telah dilakukan. Pelabelan ini tergantung banyaknya klaster yang disediakan. Jika terdiri dari dua klaster, berarti dokumen tersebut akan dimasukkan ke dalam klaster pertama atau klaster kedua sesuai hasil dari *Bipartite Graph Partition* dengan menggunakan fungsi *assign_label_cluster* pada *file assign_label.py*.



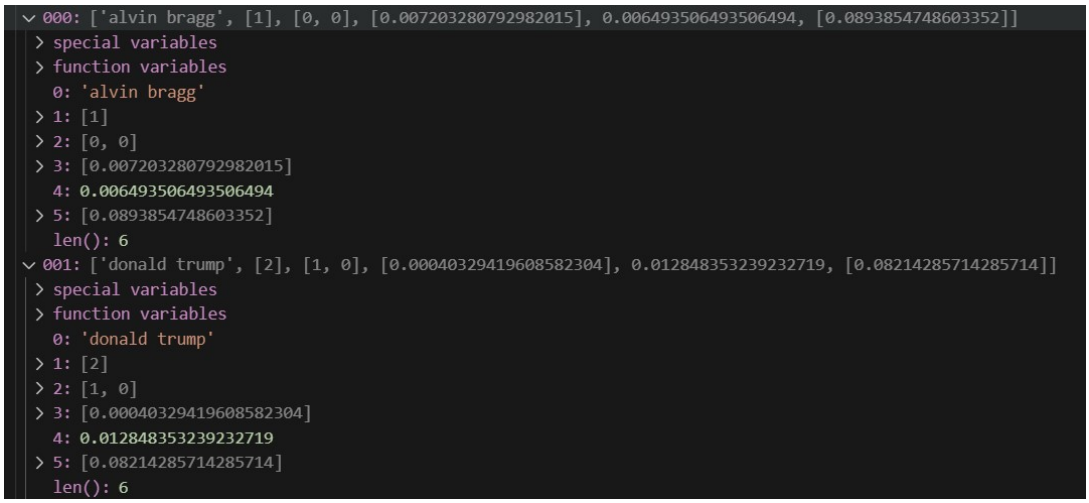
Gambar 4.12: Pelabelan dokumen

Pada variabel *all_title_id_document_with_cluster*, terdapat beberapa data di setiap *index*. Data tersebut yaitu

1. Judul dokumen dan *id*.
2. Label klaster.
3. *Node* keberapa dalam suatu *bipartite graph*.

4.1.8 Menghitung Rank(T) untuk setiap Tag

Setiap *tag* yang ada di dataset, perlu dihitung terlebih dahulu nilai *Rank(T)*. Untuk menghitung *Rank(T)*, akan menggunakan fungsi *node_rankt* dalam *file node_rank_t.py*. Hasilnya sebagai berikut.



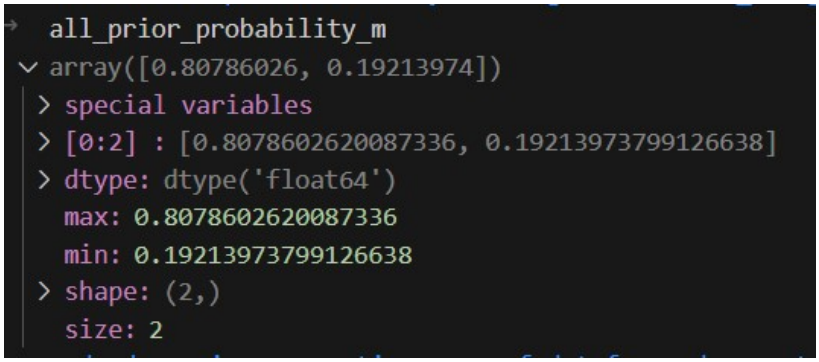
Gambar 4.13: Rank(T)

Dari hasil ini, terdapat dua *tag* yang digunakan sebagai contoh yaitu *Alvin Bragg* dan *Donald Trump*. *Alvin Bragg* memiliki nilai *N Precision* sebesar 0,893855 , *N Recall* sebesar 0,006494, dan *Rank(T)* sebesar 0,007203.

4.1.9 Membuat Two Way Poisson Mixture Model

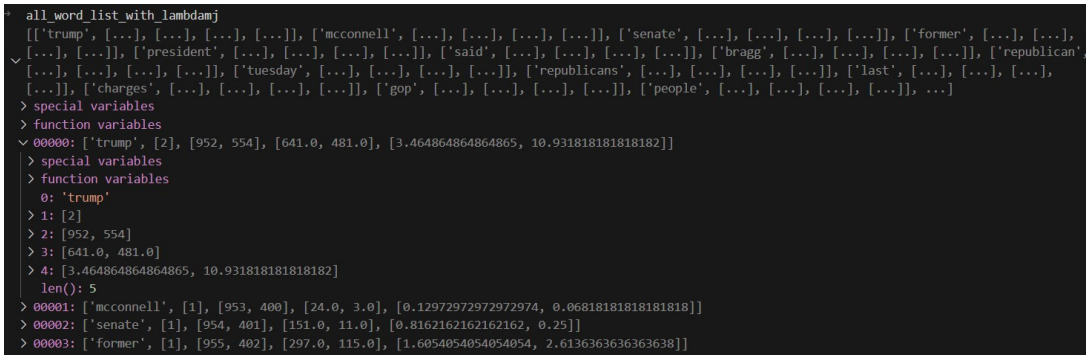
Dalam membuat *Two Way Poisson Mixture Model*, terdapat beberapa hal yang diperlukan yaitu relasi antara dokumen dengan *word vocabulary*, banyaknya komponen M, banyaknya klaster K, dan banyaknya *word cluster* L.

Awalnya, perlu mencari π_m dengan cara menghitung seluruh dokumen dalam komponen m dibagi dengan seluruh dokumen yang ada menggunakan fungsi *first_prior_probability* pada *file two_way_poisson_mixture_model.py*. Dengan menggunakan banyaknya komponen M berjumlah dua, menghasilkan hasil sebagai berikut.



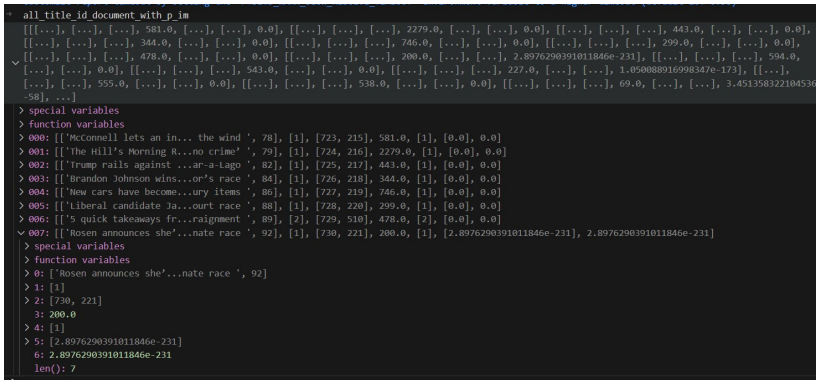
Gambar 4.14: π_m

Selanjutnya, mencari nilai $\tilde{\lambda_m}$ untuk setiap kata. Nilai $\tilde{\lambda_m}$ untuk setiap kata didapat dari nilai rata-rata kemunculan kata tersebut di setiap dokumen d dalam setiap komponen m. Dengan menggunakan fungsi *lambda_m_j_list* pada *file two_way_poisson_mixture_model.py* hasilnya sebagai berikut.



Gambar 4.15: Daftar kata dengan nilai $\tilde{\lambda_m}$

Kemudian, mencari nilai $p_{i,m}$ dalam setiap dokumen untuk memulai fase *E-step* dengan menggunakan fungsi *p_im_list* pada *file two_way_poisson_mixture_model.py*. Hasil yang didapat sebagai berikut.



Gambar 4.16: Nilai $p_{i,m}$ pada setiap dokumen

Setiap *index* dari variabel tersebut mengandung beberapa nilai yaitu

1. Nilai dari judul dokumen dan *id* dokumen.
2. Klaster k dari dokumen tersebut.
3. *Node* keberapa dari suatu *bipartite graph* awal dan *bipartite graph* setelah partisi.
4. Banyaknya kata dalam dokumen.
5. Komponen m dari dokumen tersebut.
6. Nilai dari $p_{i,m}$.

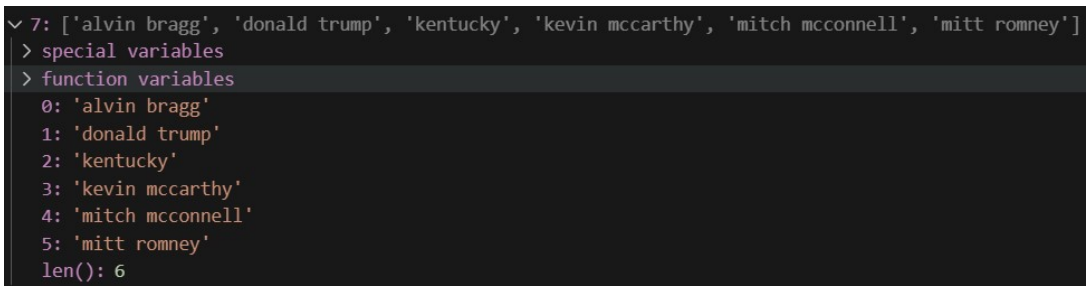
7. Nilai dari $P(D = d|K = k)$ dari dokumen.

Dari data di atas, nilai $p_{i,m}$ sangat rendah dan beberapa diantaranya 0. Hal ini dikarenakan perhitungan *product* dari nilai θ yang rendah, tapi nilai p (banyaknya kata unik) yang banyak dalam persamaan 2.12.

Selanjutnya adalah melakukan *M-step*. Dalam penelitian ini, iterasi yang digunakan yaitu lima kali.

4.1.10 Rekomendasi Tag

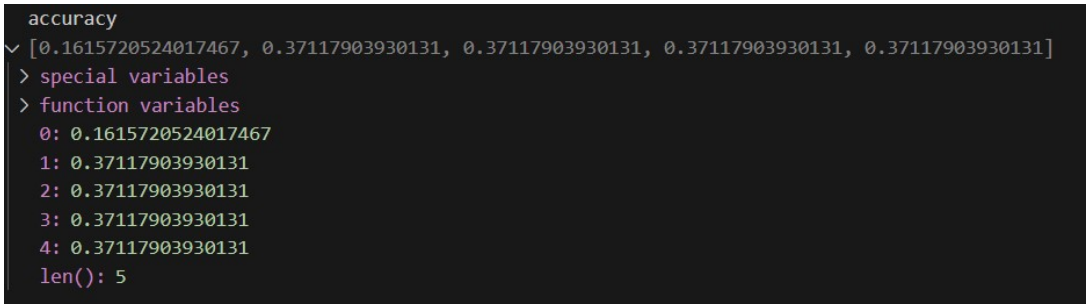
Fungsi utama dari algoritma ini adalah membuat rekomendasi *tag* berdasarkan data latih yang tersedia. Saat ini, algoritma tersebut mampu memberikan sepuluh pilihan *tag* sesuai hasil rekomendasi. Dengan menggunakan fungsi *tag_recommendation_mass* pada *file tag_recommendation_for_new_document.py*, hasil yang didapat sebagai berikut.



Gambar 4.17: Contoh dari hasil rekomendasi *Tag*

4.2 Hasil Pengujian

Berdasarkan hasil pengujian, akurasi dari rekomendasi *tag* dengan menggunakan algoritma *Bipartite Graph Partition* dan *Two Way Poisson Mixture Model* adalah sebagai berikut.



Gambar 4.18: Contoh dari hasil rekomendasi *Tag*

Saat melakukan iterasi di *Poisson Mixture Model* sebanyak lima kali, akurasi tertinggi sebanyak 37% dengan jumlah data 229 dokumen dan 723 *tag*.

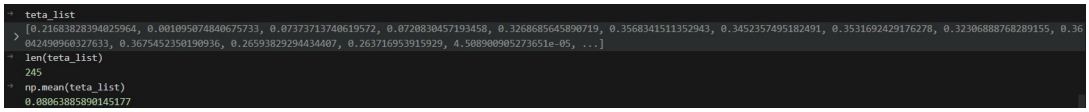
Untuk pengujian ini, tidak menggunakan data uji dan justru melakukan pengujian dengan data yang sama seperti data latih.

4.3 Hasil Analisa

Berdasarkan hasil pengujian, jumlah akurasi terbilang sedikit yaitu 37,118%. Tentunya, terdapat beberapa faktor kenapa hasilnya rendah yaitu

1. Penggunaan K yang sangat sedikit saat melakukan *Bipartite Graph Partition*.
2. Nilai $P(D = d|K = k)$ yang terlalu sedikit juga menentukan rendahnya akurasi.
3. Adanya kemungkinan hasil kodingan program yang kurang baik.

Salah satu penyebabnya adalah saat perhitungan dengan menggunakan persamaan 2.12. Dalam melakukan perhitungan ini, nilai rata-rata pada $\theta\left(d(i,j) \mid \tilde{\lambda}^{(t)}_{m,i,j}\right)$ adalah sebagai berikut.



Gambar 4.19: Nilai $\theta\left(d(i,j) \mid \tilde{\lambda}^{(t)}_{m,i,j}\right)$

Selain itu, kelemahan kodingan program yang telah penulis buat adalah tidak bisa membagi *Bipartite Graph Partition* sesuai dengan sumber aslinya. Jika program

yang penulis buat mampu membagi *Bipartite Graph Partition* sesuai dengan K yang diinginkan, akurasinya bisa saja meningkat.

BAB V
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Berdasarkan hasil dari implementasi dan pengujian *Automatic Tagging* dengan menggunakan algoritma *Bipartite Graph Partition* dan *Two Way Poisson Mixture Model* , maka diperoleh kesimpulan sebagai berikut. Program yang telah dibuat dapat menghasilkan beberapa rekomendasi *tag*. Akan tetapi, akurasi yang dihasilkan masih sangat rendah yaitu 37%. Hal ini dikarenakan banyaknya K yang digunakan dalam program ini untuk melakukan *Bipartite Graph Partition* hanyalah dua. Selain itu, nilai $\theta\left(d(i,j) \mid \tilde{\lambda}_{m,i,j}^{(t)}\right)$ yang dihasilkan mempengaruhi nilai $p_{i,m}$.

5.2 Saran

Adapun saran untuk penelitian selanjutnya adalah:

1. Menggunakan jumlah K yang lebih optimal.
2. Menggunakan dataset yang lebih besar karena tingkat akurasinya akan lebih meningkat dibandingkan dengan dataset yang lebih kecil.
3. Dapat menggunakan bahasa Indonesia dalam pemilihan artikel atau situs yang digunakan untuk penelitian.
4. Mengintegrasikan algoritma ini ke dalam aplikasi *website* agar pengguna mampu menggunakannya lebih mudah.
5. Melakukan sinkronisasi dari algoritma *Automatic Tagging* ke dalam sistem mesin pencari *Telusuri*.

DAFTAR PUSTAKA

Brin, S. & Page, L. (1998). The anatomy of a large-scale hypertextual web search engine.

Christ, A. (2022). Top 10 search engines in the world (2022 update).

Farooq, U., Song, Y., Caroll, J. M., & Giles, C. L. (2007). Social bookmarking for scholarly digital libraries.

Golub, G. H. & Loan, C. F. V. (1996). *Matrix Computations Third Edition*. Matrix Computations. The John Hopkins University Press.

Khatulistiwa, L. (2022). Perancangan arsitektur search engine dengan mengintegrasikan web crawler, algoritma page ranking, dan document ranking.

Li, J. & Zha, H. (2004). Two-way poisson mixture models for simultaneous document classification and word clustering.

Parra, E., Escobar-Avila, J., & Haiduc, S. (2018). Automatic tag recommendation for software development video tutorials.

Pratama, Z. (2022). Perancangan modul pengindeks pada search engine berupa induced generalized suffix tree untuk keperluan perangkingan dokumen.

Reynolds, D. (2009). Gaussian mixture model.

Rzeszotarski, M. (1999). The aapm/rsna physics tutorial for residents1. *RadioGraphics*, 19:765–782.

Seymour, T., Frantsvog, D., & Kumar, S. (2011). History of search engines. *International Journal of Management & Information Systems (IJMIS)*, 15(4):47–58.

Shi, M., Liu, J., Tang, M., Xie, F., & Zhang, T. (2016). A probabilistic topic model for mashup tag recommendation.

Song, Y., Zhuang, L., & Giles, L. (2011). Real-time automatic tag recommendation.

Song, Y., Zhuang, Z., Li, H., Zhao, Q., Li, J., Lee, W.-C., & Giles, L. (2008). Real-time automatic tag recommendation.

Sood, S. (2007). Tagassist: Automatic tag suggestion for blog posts.

Won, M., Ferraro, A., Bogdanov, D., & Serra, X. (2020). Evaluation of cnn-based automatic music tagging models.

Zha, H., He, X., Ding, C., Simon, H., & Gu, M. (2001). Bipartite graph partitioning and data clustering.

LAMPIRAN A

main.py

```
# Source dari file
import data_from_database as dfd
import matrix_processing as mp
import input_processing as ip
import normalized_laplacian as nl
import low_rank_approximation_matrix as lram
import spectral_recursive_embedding as sre
import the_moment as tm
import assign_label as al
import node_rank_t as nrt
import word_count_in_matrix as wcim
import two_way_poisson_mixture_model as twpmm
import word_count_in_list as wcil
import tag_recommendation_for_new_document as trfnd
import top_k_accuracy as tka
import data_testing_converter as dtc

import numpy as np

tm.this_moment("Menjalankan Algoritma :")

# Mengambil data dari database
dataset_document = dfd.get_data()
tm.this_moment("Mengambil dataset :")

# Mensetting K, M, dan L
K = 2
M = 2
L = 2

# Memproses dataset menjadi matrix
matrix_tag_document, matrix_document_word, title_id_document, all_tag_list,
     all_word_list, dataframe_document_tag, dataframe_document_word = ip.
    document_processing(dataset_document)
tm.this_moment("dataset ke matrix :")

matrix_w = mp.matrixABtoW(matrix_tag_document, matrix_document_word)
```

```
tm.this_moment("matrix A & B menjadi W :")

matrix_Q, matrix_T = lram.lanczos_iteration(matrix_w, 1)
matrix_W_hat = lram.low_rank_approximation_matrix(matrix_Q, matrix_T)
tm.this_moment("Low Rank Approximation :")
# print(matrix_W_hat)

all_matrix_partition, all_cluster = sre.spectral_recursive_embedding(
    matrix_W_hat, matrix_w)
tm.this_moment("Spectral Recursive Embedding :")
# print("X")

all_matrix_w_hat_partition = []
for matrix_partition in all_matrix_partition:
        Q, T = lram.lanczos_iteration(matrix_partition, 1)
        matrix_W_hat_partition = lram.low_rank_approximation_matrix(Q, T)
        all_matrix_w_hat_partition.append(matrix_W_hat_partition)
tm.this_moment("Create W hat partition :")

all_tag_list_with_cluster, all_title_id_document_with_cluster,
    all_word_list_with_cluster = al.assign_label_cluster(title_id_document,
     all_tag_list, all_word_list, all_cluster)
tm.this_moment("Assign Label :")

all_tag_list_with_rank = nrt.node_rankt(all_tag_list_with_cluster, matrix_w
    , all_matrix_partition)
tm.this_moment("Node Rank T :")

# Menghitung banyaknya document dari
all_title_id_document_with_word_count, total_doc, total_doc_in_cluster =
    wcim.word_count_in_matrix(all_title_id_document_with_cluster,
    all_matrix_partition, dataframe_document_word)
tm.this_moment("Word Count setiap Document dari Matrix partisi :")
# print("X")

# Menghitung banyaknya word serta banyaknya word di masing-masing klaster
all_word_list_with_count, total_word, total_word_in_cluster = wcil.
    word_count_in_list(all_word_list_with_cluster, matrix_w,
    all_matrix_partition)
tm.this_moment("Word Count setiap word :")

# Two Way Poisson Mixture Model
```

```
# Memilih m component
all_title_id_document_with_m_component, total_doc_in_component = twpmm.
    set_m_component_to_document(all_title_id_document_with_word_count, M ,K
    )
tm.this_moment("Menentukan m component pada suatu klaster :")

# Menghitung banyaknya word serta banyaknya word di masing-masing komponen
all_word_list_with_count = twpmm.set_word_count_in_every_m(
    all_title_id_document_with_m_component, all_word_list_with_count, M, K,
     matrix_document_word)
tm.this_moment("Word Count setiap word :")
# Menghitung prior probability
all_prior_probability_m = twpmm.first_prior_probability(total_doc,
    total_doc_in_component)
tm.this_moment("prior probability :")
# Menghitung nilai lambda
all_word_list_with_lambdamj = twpmm.lambda_m_j_list(
    all_word_list_with_count, total_doc_in_component)
tm.this_moment("lambda(m,j) :")
# Menghitung probabilitas
all_title_id_document_with_probability = twpmm.probability(
    all_title_id_document_with_m_component, all_prior_probability_m,
    all_word_list_with_lambdamj, dataframe_document_word, M)
tm.this_moment("P(D = d|C = k) :")
# Menghitung nilai p(i,m)
all_title_id_document_with_p_im = twpmm.p_im_list(
    all_title_id_document_with_probability, all_prior_probability_m,
    all_word_list_with_lambdamj, dataframe_document_word, M)
tm.this_moment("p(i,m) :")

# Looping Expectation Maximization
log_likelihood = []
top_k_accuracy_list = []
for i in range(1, 6):
        # Menghitung Prior probability (pi_m) t+1
        all_prior_probability_m, sum_p_im_list = twpmm.pi_m_with_t(
            all_title_id_document_with_p_im, M)
        tm.this_moment("pi(m) (t+1) :")
        # Menghitung lambda t+1
        all_word_list_with_lambdamj = twpmm.lambda_mt(
            all_word_list_with_lambdamj, sum_p_im_list,
            all_title_id_document_with_p_im, M)
```

```
        tm.this_moment("lambda(m) (t+1) :")
        # Menghitung nilai likelihood
        new_all_title_id_document_with_p_im = twpmm.p_im_list_t_more_than_1
            (all_title_id_document_with_p_im, all_prior_probability_m,
            all_word_list_with_lambdamj, dataframe_document_word)
        tm.this_moment("p(i,m) (t+1) :")
        log_likelihood.append(twpmm.get_log_likelihood(
            all_title_id_document_with_p_im,
            new_all_title_id_document_with_p_im))
        all_title_id_document_with_p_im =
            new_all_title_id_document_with_p_im
        all_title_id_document_with_probability = twpmm.
            set_new_probability_t(all_title_id_document_with_p_im)
        tm.this_moment("Menghitung nilai Log Likelihood :")

        all_title_id_document_with_tag_recommendation = trfnd.
            tag_recommendation_mass(all_title_id_document_with_probability,
             all_tag_list_with_rank, all_cluster, total_doc_in_cluster)
        tm.this_moment('Tag Recommendation: ')

        top_k_accuracy_value = tka.top_k_accuracy(
            all_title_id_document_with_tag_recommendation,
            dataframe_document_tag)
        tm.this_moment('Top 6 Tag: ')

        top_k_accuracy_list.append(top_k_accuracy_value)

accuracy = []
for tkal in top_k_accuracy_list:
        accuracy.append(tkal.count(1) / len(tkal))

tm.this_moment('Menghitung akurasi: ')

print("X")
```

LAMPIRAN B

data_from_database.py

```
import pymysql.cursors

connection = pymysql.connect(host='localhost', user='root', password='',
    database='autotag-crawl', autocommit=True,)

# Mengambil data dari database
def get_data():
        with connection:
                with connection.cursor() as cursor:

                        # Mengambil data tag, title, dan isi artikel
                        cursor.execute("SELECT DISTINCT page_tags.tag,
                            page_information.id_page, page_information.
                            content_article, page_information.title FROM `
                            page_tags` INNER JOIN page_information ON
                            page_tags.page_id = page_information.id_page")
                        result = cursor.fetchall()
                        return result
```

LAMPIRAN C

input_processing.py

```
import nltk
import numpy as np
import pandas
import re
import the_moment as tm

from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

unique_words = []
words = []
tags = []
documents = []

def wordProcessing(content_article):
        """
        Fungsi untuk menghitung banyaknya word dalam suatu artikel

        Args:
                content_article: Isi dari artikel

        Sumber library untuk menghitung kata:
                https://www.nltk.org/book/ch01.html

        Returns:
                        word_document_dictionary: berupa dictionary untuk
                            menghitung banyaknya
                dan beragamnya word dalam suatu document
        """
        tokens = word_tokenize(re.sub('[^ 0-9a-z]+', ' ', content_article.
            lower())) # Menghilangkan tanda baca
        english_stopwords = stopwords.words('english') # Menampilkan daftar
             stopwords
        english_stopwords.extend(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', '
            i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u',
             'v', 'w', 'x', 'y', 'z']) #menambahkan stopwords
        tokens_wo_stopwords = [t for t in tokens if t not in
```

```
            english_stopwords]

        # Menghitung banyaknya word
        freq = nltk.FreqDist(tokens_wo_stopwords)
        word_document_dictionary = {}

        # Menampung word tersebut dalam format dictionary
        # Alasan menggunakan freq.most_common(2000) untuk mendapatkan 2000
            kata yg sering muncul
        for word, count in freq.most_common(2000):
                        word_document_dictionary.update({word: count})

        unique_words.append(len(word_document_dictionary))
        words.append(len(tokens_wo_stopwords))
        # len(word_document_dictionary)

        # print("Dictionary : ", word_document_dictionary)
        return word_document_dictionary

def documentWordProcessing(content_article):
                """
                Fungsi untuk membuat dataframe antara document(title)
                    dengan word

                Args:
                        content_article: Isi dari artikel
                        id_article: ID dari artikel

                Returns:
                                word_document: word_document dalam bentuk
                                    dataframe
                """
                word_document_dictionary = wordProcessing(content_article)
                # word_document = pandas.DataFrame(word_document_dictionary
                    ,
                #     index=[id_article]
                # )
                return word_document_dictionary

def document_processing(dataset_document):
        """
        Memproses dataset yang masuk, lalu mengolahnya menjadi kumpulan
```

```
    dataframe antara tag dgn dokumen
dan dokumen dgn word

Args:
        dataset_document: data-data yg diambil dari database dengan
             isi "tag, id_article, content_article"

Returns:
        matrix_tag_document: matriks antara tag dgn document
        matrix_document_word: matriks antara document dgn word
        title_id_document: relasi antara title dan id dari suatu
            artikel
"""

# tm.this_moment('Mulai Document Processing :')
id_before = dataset_document[0][1]
# document_word = []
# document_word.append(documentWordProcessing(dataset_document
    [0][2], dataset_document[0][1]))

title_id_document = []
title_id_document.append((dataset_document[0][3].replace('| The
    Hill', ''), dataset_document[0][1]))

tag_dictionary = {}
tag_dictionary_list = []

word_dictionary = {}
word_dictionary_list = []
word_dictionary_list.append(documentWordProcessing(dataset_document
    [0][2]))

id_list = []
# Tempat untuk menghitung banyaknya tag dalam suatu dokumen
document_tag = []

# tm.this_moment('Mulai looping :')
for data in dataset_document:

        # Jika judul data berbeda dengan id_before
        if id_before != data[1]:
```

```
                    # Menampung tag-tag yg telah didapat di
                        tag_dictionary ke document_tag
                    id_list.append(id_before)
                    tag_dictionary_list.append(tag_dictionary.copy())
                    tags.append(len(tag_dictionary.copy()))
                    # document_tag.append(datafr)
                    tag_dictionary.clear()

                    # Melakukan proses untuk menghitung banyaknya kata
                        dalam suatu dokumen
                    title_id_document.append((data[3].replace('| The
                        Hill', ''), data[1]))
                    word_dictionary_list.append(documentWordProcessing(
                        data[2]))
                    id_before = data[1]

            #tag yg didapat akan dimasukkan ke tag_dictionary
            tag_dictionary.update({data[0]: 1})

id_list.append(id_before)
tag_dictionary_list.append(tag_dictionary.copy())
tags.append(len(tag_dictionary.copy()))
tag_dictionary.clear()

# tm.this_moment('Mulai concat docword :')
document_word = pandas.DataFrame(word_dictionary_list, index=
    id_list)
document_word = document_word.fillna(0)

# tm.this_moment('Mulai concat doctag :')
document_tag = pandas.DataFrame(tag_dictionary_list, index=id_list)
document_tag = document_tag.fillna(0)


# tm.this_moment('Mulai membuat matrix :')
matrix_tag_document = document_tag.to_numpy().transpose()
matrix_document_word = document_word.to_numpy()
return matrix_tag_document, matrix_document_word, title_id_document
    , document_tag.columns, document_word.columns, document_tag,
    document_word
```

LAMPIRAN D

matrix_processing.py

```
import numpy as np

def matrixABtoW(A, B):
        """
        Fungsi untuk menggabungkan matriks A dan B menjadi W

        Args:
                A: Matriks A dengan tag sebagai row dan document sebagai
                    column
                B: Matriks B dengan document sebagai row dan word sebagai
                    column

        Returns:
                W: Matriks gabungan antara A dengan B
        """
        AT = A.transpose()
        BT = B.transpose()

        # Menghitung banyaknya tag, dokumen, dan word
        tag_count, document_count = A.shape
        document_count, word_count = B.shape

        # Membuat matriks W dengan panjang dan lebar dari "tag + dokumen +
            word"
        all_count = tag_count + document_count + word_count
        W = np.zeros((all_count, all_count))

        # Menempelkan matriks A ke W
        for i in range(tag_count):
                W[i][tag_count:-word_count] = A[i]

        # Menempelkan matriks B Transpose ke W
        for i in range(1, word_count+1):
                W[-i][tag_count:-word_count]= BT[-i]

        # Menempelkan matriks A Transpose ke W
        for i in range(document_count):
```

```
            W[tag_count+i][0:tag_count] = AT[i]

    # Menempelkan matriks B ke W
    for i in range(document_count):
            W[tag_count+i][-word_count:] = B[i]

    return W
```

LAMPIRAN E

low_rank_approximation_matrix.py

```
import numpy as np

def normalization_vector(vector):
        vector_power = [np.power(x, 2) for x in vector]
        vector_normalize = np.sqrt(np.sum(vector_power))
        return vector_normalize

def lanczos_iteration(A, one_b = 1):
        """
        Melakukan Lanczos iteration dengan membuat matriks Q dan T
        Kemudian, membuat kedua matriks tersebut menjadi W_hat

        Args:
                A: Inputan dari matriks W

        Returns:
                Q: Hasil perkalian matriks Q
        """

        k = 50

        row, column = A.shape

        # Buat matriks T untuk menampung alpha dan beta
        T = np.zeros((k, k))

        # buat matriks Q untuk menampung q_now di setiap looping
        Q = np.zeros((row, k))

        beta = 0
        q_before = 0
        b = "" # Bentuk matriks b secara arbitrary (bebas)
        if (one_b != 1):
                b = np.random.default_rng().random((row, 1))
        else:
                b = np.ones((row, 1))
```

```
        # Matriks q_now adalah matriks yg telah normalisasi
        # panjang q_now = 1, cek panjang q_now dengan np.sqrt(np.sum([x*x
            for x in q_now]))
        q_now = b / normalization_vector(b)

        Q[:, 0] = q_now.transpose()

        for i in range(1, k):
                v = A.dot(q_now)
                alpha = q_now.transpose().dot(v)
                alpha = alpha[0][0] # Membuat alpha agar menjadi skalar
                v = v - beta*q_before - alpha*q_now
                beta = normalization_vector(v)
                q_before = q_now
                q_now = v / beta

                # Tampung alpha dan beta ke matriks T
                T[i-1, i-1] = alpha
                if i < row:
                        T[i-1, i] = beta
                        T[i, i-1] = beta

                        # Tampung nilai q_now ke matriks Q
                        Q[:, i] = q_now.transpose()

        return Q, T

def low_rank_approximation_matrix(Q, T):
        """
        Melakukan perkalian matriks Q, T, dan Q transpose

        Args:
                Q: Matriks Q yang didapat dari Lanczos iteration
                T: Matriks T yang didapat dari Lanczos iteration

        Returns:
                W_hat: Hasil perkalian matriks Q, T, dan Q transpose
        """

        return Q.dot(T).dot(Q.transpose())
```

LAMPIRAN F

spectral_recursive_embedding.py

```
import numpy as np
import scipy as sp
import normalized_laplacian as nl
import matrix_processing as mp
import low_rank_approximation_matrix as lram

import the_moment as tm

def second_largest_singular_vector(W_hat):
        """
                Menghitung singular vector menggunakan W_hat dari library
                https://docs.scipy.org/doc/scipy/reference/sparse.linalg.
                    svds-lobpcg.html

                Args:
                        W_hat: Matrix W_hat

                Returns:
                        second_largest_left: Mengambil vektor U
                        second_largest_right: Mengambil vektor Vh
        """

        U, s, Vh = sp.sparse.linalg.svds(W_hat, solver='lobpcg')
        # U, s, Vh = sp.linalg.svd(W_hat)
        second_largest_left = U[:, 1] # Mengambil left singular vector
            kedua
        second_largest_right = Vh[1, :] # Mengambil right singular vector
            kedua

        return second_largest_left, second_largest_right

def find_cut_point(singular_vector_left, singular_vector_right ):
        """
                Mencari cut point

                Returns:
                        cx: cx
```

```
                cy: cy
        """

        cx = np.median(singular_vector_left)
        cy = np.median(singular_vector_right)
        return cx, cy

def form_partition(cx, cy, x, y):
        """
                Melakukan form partition

                Args:
                        cx: cut point untuk x
                        cy: cut point untuk y
                        x: Hasil dari second_largest_left
                        y: Hasil dari second_largest_right

                Returns:
                        A_partition
                        Ac_partition
                        B_partition
                        Bc_partition
        """

        A_partition = []
        Ac_partition = []
        B_partition = []
        Bc_partition = []

        i = 0
        for value in x:
                if value >= cx:
                        A_partition.append(i)
                else:
                        Ac_partition.append(i)
                i+=1

        j = 0
        for value in y:
                if value >= cy:
                        B_partition.append(j)
                else:
```

```
                Bc_partition.append(j)
            j+=1

    return A_partition, Ac_partition, B_partition, Bc_partition


def create_matrix_from_two_vertex(X, Y, W):
    """
        Menggabungkan dua partisi ke dalam satu matrix berdasarkan
        'Bipartite graph partitioning and data clustering'

        Args:
            X: Vertex x
            Y: Vertex y
            W: Matrix W awal

        Returns:
            matrix: Matrix dari bipartite graph yg terbaru
            XY: Tanda untuk pada vertex keberapa ia di klaster
                ini
    """

    XY = list(set(X).union(set(Y)))
    len_xy = len(XY)
    matrix = np.zeros((len_xy , len_xy))

    # Looping untuk membuat matrix_w hasil partisi
    for i in range(0, len_xy):
        xy_i = XY[i]
        for j in range(0, len_xy):
            xy_j = XY[j]
            matrix[i][j] = W[xy_i][xy_j]

    return matrix, XY

def spectral_recursive_embedding(W_hat, W):
    """
        Melakukan bipartite graph partition dengan menggunakna
        spectral recursive embedidng

        Args:
            W_hat: Matrix W_hat
```

```
                Y: Vertex y
                W: Matrix W awal

        Returns:
                all_matrix: List Matrix w hasil klasterisasi
                all_cluster: Klaster
"""
all_matrix = []
all_cluster = []
# tm.this_moment("Start :")
x, y = second_largest_singular_vector(W_hat)
# tm.this_moment("Second Largest Singular Vector :")
cx, cy = find_cut_point(x, y)
# tm.this_moment("Cut Point :")
A, Ac, B, Bc = form_partition(cx, cy, x, y)
# tm.this_moment("Form Partition :")
matrix, cluster = create_matrix_from_two_vertex(A, B, W)
all_matrix.append(matrix)
all_cluster.append(cluster)
matrix, cluster = create_matrix_from_two_vertex(Ac, Bc, W)
all_matrix.append(matrix)
all_cluster.append(cluster)
# tm.this_moment("Fusion the matrix :")

return all_matrix, all_cluster
```

LAMPIRAN G

assign_label.py

```
def assign_label_for_tag(all_tag_list, all_cluster, index):
  all_tag_list_with_cluster = []

  # Looping seluruh isi all_tag_list
  for tag in all_tag_list:
    tag_cluster = []
    tag_index_in_matrix = [] # Kumpulan index dalam satu tag di dalam
        setiap matrix yg ditempati tag tersebut
    index_cluster = 0 # index klaster

    tag_index_in_matrix.append(index)
    # Looping selurung klaster
    for cluster in all_cluster:
      if index in cluster:
        tag_cluster.append(index_cluster + 1)
        tag_index_in_matrix.append(cluster.index(index))
      index_cluster+=1

    index += 1
    all_tag_list_with_cluster.append([tag, tag_cluster, tag_index_in_matrix
        ])

  return index, all_tag_list_with_cluster

def assign_label_for_document(title_id_document, all_cluster, index):
  # Memasukkan label klaster ke dalam document
  all_title_id_document_with_cluster = []

  # Looping seluruh isi title_id_document
  for title, id in title_id_document:
    document_cluster = []
    document_index_in_matrix = [] # Kumpulan index dalam satu document di
        dalam setiap matrix yg ditempati document tersebut
    index_cluster = 0

    document_index_in_matrix.append(index)
    # Looping isi klaster
```

```
    for cluster in all_cluster:
      # Jika index tersebut ada di suatu klaster
      if index in cluster:
        document_cluster.append(index_cluster + 1)
        document_index_in_matrix.append(cluster.index(index))
      index_cluster+=1

    index += 1
    all_title_id_document_with_cluster.append([[title, id],
        document_cluster, document_index_in_matrix])

  return index, all_title_id_document_with_cluster

def assign_label_for_word(all_word_list, all_cluster, index):
  # Memasukkan label klaster ke dalam word
  all_word_list_with_cluster = []

  # Looping seluruh word di all_word_list
  for word in all_word_list:
    word_cluster = []
    word_index_in_matrix = [] # Kumpulan index dalam satu word di dalam
        setiap matrix yg ditempati word tersebut
    index_cluster = 0

    word_index_in_matrix.append(index)
    # Looping isi klaster
    for cluster in all_cluster:
      if index in cluster:
        word_cluster.append(index_cluster + 1)
        word_index_in_matrix.append(cluster.index(index))
      index_cluster+=1

    index += 1
    all_word_list_with_cluster.append([word, word_cluster,
        word_index_in_matrix])

  return index, all_word_list_with_cluster

def assign_label_cluster(title_id_document, all_tag_list, all_word_list,
    all_cluster):

  # Memasukkan label klaster ke dalam tag
```

```
index = 0 # Index row matrix awal
index, all_tag_list_with_cluster = assign_label_for_tag(all_tag_list,
    all_cluster, index)
index, all_title_id_document_with_cluster = assign_label_for_document(
    title_id_document, all_cluster, index)
index, all_word_list_with_cluster = assign_label_for_word(all_word_list,
    all_cluster, index)

return all_tag_list_with_cluster, all_title_id_document_with_cluster,
    all_word_list_with_cluster
```

LAMPIRAN H

node_rank_t.py

```
import numpy as np

import matrix_processing as mp

def n_recall(matrix_w_origin, matrix_w_partition, node_i_origin):
        """
                Menghitung N Precision dari suatu tag

                Args:
                        matrix_w_origin: matrix w yg dari awal dibuat
                        tag_cluster:
                        node_i_origin: node di matrix w yg dari awal dibuat

                Returns:
                        nri: N Recall dari suatu tag
        """

        row, col = matrix_w_partition.shape
        nri_top = sum(matrix_w_origin[node_i_origin])
        nri_bottom = row
        nri = nri_top / nri_bottom
        return nri

def rank(npi, nri):
        """
                Menghitung Rank T

                Args:
                        npi: N Precision dari suatu tag
                        nri: N Recall dari suatu tag

                Returns:
                        ranki: Rank dari suatu tag
        """
        ri = npi * np.log(nri)
        ranki = 0
        if (ri != 0):
```

```
                ranki =  np.exp(-1 / np.power(ri, 2))

        return ranki

# Untuk mendapatkan np top di setiap tag
def get_all_np_top_list(tag_list, all_matrix_partition):
        np_top_list = []
        k_list = []

        for tag, cluster, nodes in tag_list:
                index_cluster = 0
                for k in cluster:
                        if not k in k_list:
                                start_range = 0
                                if len(k_list) > 0:
                                        start_range = k_list[-1]
                                for ik in range(start_range+1, k+1):
                                        k_list.append(ik)
                                        np_top_list.append([])

                        npi_top = sum(all_matrix_partition[k-1][nodes[
                            index_cluster + 1]])
                        np_top_list[k-1].append(npi_top)
                        index_cluster += 1

        return np_top_list

# mendapatkan nilai N Precission di setiap tag
def get_all_np_list(tag_list, np_top_list):
        np_list = []
        k_list = []
        sum_np_top = []

        for tag, cluster, nodes in tag_list:
                index_cluster = 0

                # Jika klaster k belum ada di k_list
                for k in cluster:
                        if not k in k_list:
                                start_range = 0
                                if len(k_list) > 0:
                                        start_range = k_list[-1]
```

```
                                for ik in range(start_range+ 1, k+1):
                                        k_list.append(ik)
                                        np_list.append([])
                                        sum_np_top.append(sum(np_top_list[
                                            ik-1]))

                        # Mengammbil np_top
                        np_top_i = np_top_list[k-1][nodes[index_cluster+1]]
                        # Menghitung np_bottom
                        np_bottom = sum_np_top[k-1]

                        # Menghitung np pada suatu tag
                        npi = np_top_i / np_bottom

                        # Menampung nilai np
                        np_list[k-1].append(npi)
                        index_cluster += 1

        return np_list


def node_rankt(tag_list, matrix_w_original, all_matrix_partition):
        """
                Menghitung seluruh nilai Rank T yg ada di dalam tag_list

                Args:
                        tag_list: Kumpulan tag
                        matrix_w_original: Matrix W awal
                        all_matrix_partition: list dari matrixs W yg telah
                            dipartisi

                Returns:
                        all_tag_list_with_rank: Tag list yg telah ada nilai
                             Rank T
        """
        all_tag_list_with_rank = []

        # Menghitung N Precision dengan menggunakan tag & document
        all_np_top_list = get_all_np_top_list(tag_list,
            all_matrix_partition)
        all_np_list = get_all_np_list(tag_list, all_np_top_list)
```

```
# Looping seluruh data di tag list
for tag, cluster, nodes in tag_list:

        # nr = n_recall(matrix_w_original, cluster, nodes[0])
        # Menghitung np & ranki
        index_cluster = 0
        rank_i_list = []
        np_i_list = []
        for k in cluster:
                np = all_np_list[k-1][nodes[index_cluster + 1]]
                # np = n_precision(all_matrix_partition[k-1], nodes
                    [index_cluster + 1])
                nr = n_recall(matrix_w_original,
                    all_matrix_partition[k-1], nodes[0])
                ranki = rank(np, nr)
                index_cluster += 1
                np_i_list.append(np)
                rank_i_list.append(ranki)

        all_tag_list_with_rank.append([tag, cluster, nodes,
            rank_i_list, nr, np_i_list])

return all_tag_list_with_rank
```

LAMPIRAN I

word_count_in_matrix.py

```
import numpy as np


def word_count_in_matrix(document_list, all_matrix_partition,
    dataframe_document_word):
        """
                Melakukan perhitungan berapa banyak kata dalam per dokumen

        Args:
                document_list: Daftar list dokumen
                all_matrix_partition: sebuah list yang berisi mengenai
                    matriks-matriks yang telah dipartisi

        Returns:
                new_document_list: dokumen list dengan tambahan banyaknya
                    kata dalam satu dokumen
                total_doc: menyimpan hasil berupa jumlah seluruh dokumen yg
                     ada
                total_doc_per_cluster: banyaknya dokumen per klaster
        """
        new_document_list = []
        total_doc = 0 # Menyimpan total doc yg ada
        total_doc_per_cluster = np.zeros(len(all_matrix_partition)) #
            Menyimpan total doc di masing-masing klaster

        # Looping sesuai banyaknya dokumen
        for title_and_id, cluster, nodes in document_list:
                # index_cluster = 0
                total_doc += 1

                # Mengambil row dari matriks partisi yang menandakan
                    dokumen tersebut
                # row_doc = matrix_origin[nodes[0]]

                # Menghitung banyaknya kata
                word_count = sum(dataframe_document_word.loc[title_and_id
                    [1]])
```

```
            # Looping klaster
            for k in cluster:
                    total_doc_per_cluster[k-1] += 1

            new_document_list.append([title_and_id, cluster, nodes,
                word_count])

    return new_document_list, total_doc, total_doc_per_cluster.tolist()
```

LAMPIRAN J

word_count_in_list.py

```
import numpy as np

# Versi di mana word yg di cluster dihitung menggunakan jumlah di
    matrix_origin
def word_count_in_list(word_list, matrix_origin, all_matrix_partition):
        """
                Menghitung banyaknya word dalam

        Args:
                document_list: Daftar list dokumen
                all_matrix_partition: sebuah list yang berisi mengenai
                    matriks-matriks yang telah dipartisi

        Returns:
                new_document_list: dokumen list dengan tambahan banyaknya
                    kata dalam satu dokumen
                total_doc: menyimpan hasil berupa jumlah seluruh dokumen yg
                     ada
                total_doc_per_cluster: banyaknya dokumen per klaster
        """
        new_word_list = []
        total_word = 0 # Menyimpan total word yg ada
        total_word_per_cluster = np.zeros(len(all_matrix_partition)) #
            Menyimpan total word di masing-masing klaster

        # Looping word list
        # name = nama word
        # cluster = word tersebut termasuk klaster berapa
        # indexes = lokasi row index pada suatu matrix
        for name, cluster, indexes in word_list:
                sum_word = []

                # Menghitung banyaknya word yang ada di seluruh dokumen
                total_of_this_word = sum(matrix_origin[indexes[0]]) #
                    Menghitung seluruh kata tersebut di dalam matrix
                sum_word.append(total_of_this_word)
                total_word += total_of_this_word
```

```
            index_cluster = 0
            for k in cluster:
                    # total_of_this_word_in_cluster = sum(
                        all_matrix_partition[k-1][indexes[index_cluster
                        +1]])
                    # total_of_this_in_cluster = sum(matr)
                    sum_word.append(total_of_this_word)

                    total_word_per_cluster[k-1] += total_of_this_word

            new_word_list.append([name, cluster, indexes, sum_word])

    return new_word_list, total_word, total_word_per_cluster
```

LAMPIRAN K

two_way_poisson_mixture_model.py

```
import numpy as np
import pandas as pd

def set_new_probability_t(doc_list):

        new_doc_list = []

        for title_id, cluster, indexes, word_count, m_component, p_im,
            probability in doc_list:
                probability = sum(p_im)

                new_doc_list.append([title_id, cluster, indexes, word_count
                    , m_component, p_im, probability])

        return new_doc_list


def set_m_component_to_document(doc_list, M, K):
        """
        Melabelkan dokumen dengan m komponen

        Args:
                doc_list: daftar dari dokumen
                M: banyaknya m komponen
                K: banyaknya klaster

        Returns:
                new_doc_list: prior probability
                total_doc_in_component: Banyaknya dokumen dalam suatu
                    komponen
        """
        new_doc_list = []
        total_doc_in_component = np.zeros(M)
        index_component = np.zeros(K)

        for title_id, cluster, indexes, word_count in doc_list:
                m_list = []
```

```
                # cluster = klaster dari dokumen
                for k in cluster:
                        # Mendapatkan m ke berapa
                        # (M / K)*(k-1) + 1 berguna untuk
                            menentukan titik mulai-nya komponen
                            berdasarkan K dan M
                        # index_component[k-1] % (M / K) berguna
                            untuk membirkan value m secara
                            bergantian setiap klaster
                        m_component = int((int(M / K)*(k-1) + 1) +
                            (index_component[k-1] % int(M / K)))
                        m_list.append(m_component)
                        total_doc_in_component[m_component - 1] +=
                            1
                        index_component[k-1] += 1

                # Membuat list dokumen terbaru
                new_doc_list.append([title_id, cluster, indexes, word_count
                    , m_list])

        return new_doc_list, total_doc_in_component

def set_word_count_in_every_m(doc_list, word_list, M, K,
    matrix_document_word):
        """
        Menghitung banyaknya masing-masing word di setiap komponen

        Args:
                        doc_list: daftar dari dokumen
                        word_list: daftar word
                        M: banyaknya m komponen
                        K: banyaknya klaster
                        matrix_document_word: matrix relasi antara dokumen
                            sbg row dgn word sbg column

        Returns:
                        new_doc_list: prior probability
                        total_doc_in_component: Banyaknya dokumen dalam
                            suatu komponen
        """
        new_word_list = []
```

```
        row, col = matrix_document_word.shape
        total_every_word_in_component = np.zeros((M, col)) # Inisiasi word
            dalam component

        index = 0
        # Proses menghitung banyaknya word di dalam suatu komponen doc
        for title_id, cluster, indexes, word_count, m_component in doc_list
            :
                for m in m_component:
                        total_every_word_in_component[m-1] +=
                            matrix_document_word[index]
                index += 1


        index = 0
        for word, cluster, indexes, word_count in word_list:
                # Memindahkan total_every_word_in_component ke dalam
                    word_count sesuai word-nya
                word_count = total_every_word_in_component[..., index]
                index += 1

                new_word_list.append([word, cluster, indexes, word_count.
                    tolist()])

        return new_word_list

def first_prior_probability(total_doc, total_doc_in_component):
        """
        Menghitung pi_m dengan cara mencari prior probability setiap m
        Dengan asumsi banyaknya M adalah banyaknya K

        Args:
                total_doc: Keseluruhan dokumen dari dataset yang diberikan
                total_doc_in_component: Keselurhan dokumen dalam satu M

        Returns:
                pi_m: prior probability
        """

        # Hitung nilai pi_m di setiap M
        pi_m = np.array(total_doc_in_component) / total_doc
        return pi_m
```

```
def lambda_m_j_list(word_list, total_doc_in_component):
        """
        Menghitung nilai lambda untuk setiap kata

        Args:
                word_list: list seluruh word yg ada di dataset
                total_doc_in_component: total dokumen dalam 1 komponen

        Returns:
                new_word_list: word list terbaru
        """

        new_word_list = [] # word list baru

        # Looping word list untuk mencari lambda_m_j
        for word, cluster, indexes, word_count in word_list:
                lambda_m_j = []
                index_m = 0
                for tdic in total_doc_in_component:
                                lambda_m_j.append(word_count[index_m] /
                                    tdic)
                                index_m += 1
                                # lambda_m_j.append(1)

                new_word_list.append([word, cluster, indexes, word_count,
                    lambda_m_j])

        return new_word_list

def probability_mass_function(d_ij, lambda_mij):
        """
        Menghitung probability mass function pada suatu word

        Args:
                d_ij: banyaknya word j dalam dokumen i
                total_doc_in_cluster: lambda dari word j

        Returns:
                teta: probability mass function
        """
```

```
        teta = np.exp(-lambda_mij) * np.power(lambda_mij, d_ij) / np.prod(
            np.arange(1, d_ij+1))
        return teta

def probability(doc_list, pi_m, word_list, dataframe_document_word, M):
        """
        Menghitung P(D = d|C = k) untuk setiap dokumen

        Args:
                doc_list: list dari suatu dokumen
                pi_m: nilai pi_m (prior probability) setiap komponen
                word_list: list dari suatu word
                dataframe_document_word: Dataframe relasi antara doc dgn
                    word

        Returns:
                new_doc_list: list doc dgn tambahan probability
        """
        new_doc_list = []

        for title_id, cluster, indexes, word_count, m_component in doc_list
            :
                probability = [] # Nilai probability yg akan distore di doc
                     list baru

                # Menghitung teta di setiap kata di dalam 1 dokumen
                prod_teta_list = np.ones(M)
                i = 0
                for word_value in dataframe_document_word.loc[title_id[1]]:
                        if(word_value < 1):
                                        i += 1
                                        continue

                        for m in m_component:
                                # Memasukkan probability mass function dgn
                                # word_value = banyaknya word dari doc ini
                                # word_list[i][4][m-1] = lambda_mj dari
                                    word tersebut
                                prod_teta_list[m-1] *=
                                    probability_mass_function(word_value,
                                    word_list[i][4][m-1])
                        i+=1
```

```
                # Menghitung probability
                for m in m_component:
                        # prod_teta_list = np.prod(teta_list[m-1])
                        probability.append(pi_m[m-1] * prod_teta_list[m-1])

                new_doc_list.append([title_id, cluster, indexes, word_count
                    , m_component, sum(probability)])

        return new_doc_list

def p_im_list(doc_list, pi_m, word_list, dataframe_document_word, M):
        """
                Memproses p_im

                Args:
                        doc_list: daftar dokumen
                        pi_m: prior probability dari komponen m dgn asumsi
                            banyaknya K = banyaknya M
                        word_list: list dari word
                        dataframe_document_word: dataframe dengan document
                            sebagai row dan word sebagai column

                Returns:
                        new_doc_list: list doc terbaru
        """

        new_doc_list = []

        # Looping doc_list dgn:
        # title_id: judul dan id dari doc
        # cluster: klaster dari dokumen
        # indexes: posisi row index pada matrix w dan matrix w partition
        # word_count: banyaknya jumlah word dalam dokumen
        for title_id, cluster, indexes, word_count, m_component,
            probability in doc_list:
                p_im = [] # Nilai p_im yg akan distore di doc list baru

                # Menghitung teta di setiap kata di dalam 1 dokumen
                prod_teta_list = np.ones(M)
                i = 0
                for word_value in dataframe_document_word.loc[title_id[1]]:
```

```
                        if(word_value < 1):
                                i += 1
                                continue

                        for m in m_component:
                                # Memasukkan probability mass function dgn
                                # word_value = banyaknya word dari doc ini
                                # word_list[i][4][m-1] = lambda_mj dari
                                    word tersebut
                                prod_teta_list[m-1] *=
                                    probability_mass_function(word_value,
                                    word_list[i][4][m-1])
                        i+=1

                # Menghitung p_im
                for m in m_component:
                        p_im.append(pi_m[m-1] * prod_teta_list[m-1])

                new_doc_list.append([title_id, cluster, indexes, word_count
                    , m_component, p_im, probability])

        return new_doc_list

def pi_m_with_t(doc_list, M):
        """
                Mencari nilai p_im jika pencarian p_im lebih dari 1 turn

                Args:
                        doc_list: daftar dokumen
                        M: banyaknya komponen
                Returns:
                        pi_m_list: Daftar pi_m terbaru
                        sum_p_im_list: sum dari p_im pada turn saat ini di
                            setiap komponen
        """

        pi_m_list = np.zeros(M)
        sum_p_im_list = np.zeros(M)

        # Mencari nilai sum(p_im)
        for title_id, cluster, indexes, word_count, m_component, p_im,
            probability in doc_list:
```

```
                index_m = 0
                for m in m_component:
                        sum_p_im_list[m-1] += p_im[index_m]
                        index_m += 1

        # Mencari nilai pi_m
        index = 0
        for value in sum_p_im_list:
                pi_m_list[index] = value/sum(sum_p_im_list)
                index += 1

        return pi_m_list, sum_p_im_list

def lambda_mt(word_list, sum_p_im_list, doc_list, M):
        """
                Mencari nilai lambda jika pencarian lambda lebih dari 1
                    turn

                Args:
                        word_list: daftar word
                        sum_p_im_list: sum dari p_im pada turn saat ini di
                            setiap komponen
                Returns:
                        new_word_list: daftar word terbaru
        """

        new_word_list = [] # word list baru
        top_lambda_mt_list = np.zeros(M) # Untuk perhitungan pada persamaan
             lambda_mt bagian atas

        # Looping setiap dokumen
        for title_id, cluster, indexes, word_count, m_component, p_im,
            probability in doc_list:

                index_m = 0
                for m in m_component:
                        top_lambda_mt_list[m-1] += p_im[index_m] *
                            word_count
                        index_m += 1

        # Looping word list untuk mencari lambda_m_j
        for word, cluster, indexes, word_count, lambda_m_j in word_list:
```

```
                lambda_m_j_temp = []

                # Kalkulasi nilai lambda berdasarkan word dan klasternya
                for m in range(1, M+1):
                        bottom_lambda_mt = word_count[m-1] * sum_p_im_list[
                            m-1]

                        # Jika nilai bottom_lambda_mt terbaru bernilai 0
                            dan mencegah lambda_m_j_temp bernilai inf
                        if bottom_lambda_mt == 0:
                                lambda_m_j_temp.append(0)
                        else:
                                lambda_m_j_temp.append(top_lambda_mt_list[m
                                    -1] / word_count[m-1] * sum_p_im_list[m
                                    -1])


                new_word_list.append([word, cluster, indexes, word_count,
                    lambda_m_j_temp])

        return new_word_list

def p_im_list_t_more_than_1(doc_list, pi_m, word_list,
    dataframe_document_word):
        """
        Mencari nilai p_im jika pencarian p_im lebih dari 1 turn

        Args:
                doc_list: daftar dokumen
                pi_m: prior probability dari komponen m
                word_list: list dari word
                dataframe_document_word: dataframe dengan document sebagai
                    row dan word sebagai column
        Returns:
        """

        new_doc_list = []

        # Looping doc_list dgn:
        # title_id: judul dan id dari doc
        # cluster: klaster dari dokumen
```

```
        # indexes: posisi row index pada matrix w dan matrix w partition
        # word_count: banyaknya jumlah word dalam dokumen
        # p_im: Nilai dari p_im
        for title_id, cluster, indexes, word_count, m_component, p_im,
            probability in doc_list:
                p_im = [] # Nilai p_im yg akan distore di doc list baru

                # Menghitung teta di setiap kata di dalam 1 dokumen
                teta_list = []
                i = 0
                for word_value in dataframe_document_word.loc[title_id[1]]:
                        if(word_value < 1):
                                i += 1
                                continue
                        teta_list.append(probability_mass_function(
                            word_value, word_list[i][4][0]))

                # Menghitung p_im
                for m in m_component:
                        prod_teta_list = np.prod(teta_list)
                        p_im.append(pi_m[m-1] * prod_teta_list)

                new_doc_list.append([title_id, cluster, indexes, word_count
                    , m_component, p_im, probability])

        return new_doc_list

def get_log_likelihood(doc_list, new_doc_list):
        log_likelihood = 0

        # Looping dokumen
        for i in range(0, len(doc_list)):

                # Looping sebanyak label m yg ada di dokumen
                for m in range(0, len(doc_list[i][4])):
                        log_p_im = np.log(new_doc_list[i][5][m])
                        log_likelihood += doc_list[i][5][m] * log_p_im

        return log_likelihood
```

LAMPIRAN L

tag_recommendation_for_new_document.py

```
import pandas

def tag_recommendation(all_tag_list_with_rank, all_cluster,
    total_doc_in_cluster, p_dt_ck):

        """
        Melakukan rekomendasi tag terhadap dokumen baru
        Args:
                all_tag_list_with_rank: Daftar tag dgn rank
                all_cluster: Daftar klaster
                total_doc_in_cluster: total banyaknya dokumen dalam 1
                    klaster
                p_dt_ck: peluangnya

        Returns:
                big_rank: Mengurutkan rank
        """

        p_cluster = 1/len(all_cluster) # P(C=k)
        p_document = [1/tdic for tdic in total_doc_in_cluster] # list P(D =
             dt) || Setiap klaster berbeda valuenya
        # p_dt_ck = 0.25

        R_Ti_dt = [] # Tampungan untuk nilai rank akhir
        all_tag_name = [] # Tampungan untuk nama tag

        for tag, cluster, nodes, rank, nr, np_i in all_tag_list_with_rank:
                index_cluster = 0

                for k in cluster:
                        probability = p_dt_ck * p_cluster / p_document[k-1]
                             # Hitung P(C=k|D=dt)
                        rti = rank[index_cluster] * probability # Hitung R(
                            Ti, dt)
                        R_Ti_dt.append(rti)
                        all_tag_name.append(tag)
                        index_cluster += 1
```

```
        # Buat dataframe dgn kolom tag & rank akhir lalu urutkan
        dff = pandas.DataFrame([all_tag_name, R_Ti_dt], ["Tag", "Value"])
        dffT = dff.T.sort_values(by=['Value'], ascending=False)

        # Ambil 6 tag dgn rank akhir terbesar
        big_rank = [tag for tag in dffT.head(6)["Tag"]]

        return big_rank

def tag_recommendation_mass(doc_list, all_tag_list_with_rank, all_cluster,
    total_doc_in_cluster):

        """
        Melakukan rekomendasi tag terhadap dokumen baru secara massal
        Args:
                all_tag_list_with_rank: Daftar tag dgn rank
                all_cluster: Daftar klaster
                total_doc_in_cluster: total banyaknya dokumen dalam 1
                    klaster
                doc_list: daftar dokumen

        Returns:
                doc_list: daftar dokumen baru
        """
        new_doc_list= []

        index = 0
        for doc in doc_list:
                tag_recommend = tag_recommendation(all_tag_list_with_rank,
                    all_cluster, total_doc_in_cluster, doc[-1])
                doc_list[index].append(tag_recommend)
                index += 1

        return doc_list
```

LAMPIRAN M

top_k_accuracy.py

```
import numpy as np

def top_k_accuracy(doc_list, dataframe_document_tag):
        """
                Mengkonversi data testing

        Args:
                doc_list: Daftar list dokumen
                dataframe_document_tag: Dataframe untuk dokumen dan tag

        Returns:
                success_list: list berapa tebakan yang benar
        """
        success_list = []
        for doc in doc_list:

                value = 0
                for tag in doc[-1]:
                        if tag in dataframe_document_tag.columns:
                                value += dataframe_document_tag.loc[doc
                                    [0][1], tag]
                                if(value == 1):
                                        success_list.append(1)
                                        break

                if(value == 0):
                        success_list.append(0)

        return success_list
```