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

Muhammad Zhafran Bahij[1], Muhammad Eka Suryana[2], Med Irzal[3]
Program Studi Ilmu Komputer, Fakultas Matematika dan Ilmu Pengetahuan Alam Universitas Negeri Jakarta, Jakarta Timur, Indonesia
muhammadzhafranbahij@gmail.com[1], eka-suryana@unj.ac.id[2], medirzal@unj.ac.id[3]

***Abstrak*— 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. 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. Hal ini terjadi karena program yang telah dibuat tidak bisa membagi Bipartite Graph Partition sebanyak K lebih dari dua.**

***Kata Kunci*— Tagging, Database, Bipartite Graph, Poisson Mixture Model**

I. PENDAHULUAN

Saat ini, penggunaan search engine atau mesin pencari telah digunakan oleh khalayak umum untuk mencari berbagai informasi yang ada, *Google* merupakan *search engine* yang menempati urutan pertama terpopuler, selanjutnya diikuti dengan *Bing*, *Baidu*, *Yahoo!*, *Yandex*, dan *DuckDuckGo*. [2]

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*. [11]

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 didapat 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 [11].

Pada penelitian *search engine Telusuri*, pengindeksan masih melalui judul dan belum menggunakan *tag*. Oleh karena itu, *automatic tagging* diperlukan agar bias membuat *tag* secara otomatis yang nantinya mampu melakukan pengindeksan lebih efisien dan cepat.

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. [14]

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*).

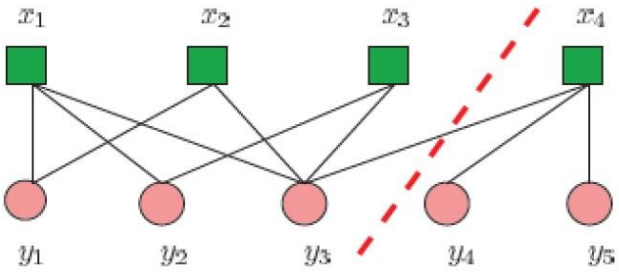
Rekomendasi *tag* dapat dilakukan dengan dua jenis menurut Song yaitu jenis pendekatan melalui pengguna dan jenis pendekatan melalui dokumen. Rekomendasi *tag* merupakan suatu sistem yang di mana sistem tersebut menampilkan *tag* yang relevan pada suatu dokumen agar pengguna bisa memperhitungkan apakah *tag* yang ditampilkan itu ingin dipakai atau tidak. Melalui pendekatan *user*, sistem ini akan mengolah rekomendasi *tag* berdasarkan *tag* 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, 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*. [13]

II. KAJIAN PUSTAKA

A. *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. [13]



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*. [13]

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, perlu menggunakan *normalized graph Laplacian* untuk aproksimasi W. *Normalized Laplacian* $L(W)$ sebagai berikut. [13]

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

di mana di 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, normalisasi Laplacian dapat direpresentasikan sebagai berikut. [13]

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

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. [13]

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:

$$\lambda \geq ... \geq \lambda_n \geq 0 \quad (3)$$

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:

$$||A - T||_F \leq e||A||_F \quad (4)$$

Di mana ||.||F denotasi *Frobenius norm*, dengan *e* sebagai variabel terkendali. Sebagai contoh, untuk menangkap 95% *varians* dari *A*, *e* diatur sebagai 0.05. [13]

Untuk melakukan multi-klastering pada *bipartite graph*, perlu 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. [13]

$$Ncut(A,B) = \frac{cut(A,B)}{W(A,Y) + W(X,B)} + \frac{cut(A^c,B^c)}{W(A^c,Y) + W(X,B^c)}, \quad (5)$$

$$\begin{aligned} cut(A,B) &= W(A,B^c) + W(A^c,B) \\ &= \sum_{i \in A, j \in B^c} w_{ij} + \sum_{i \in A^c, j \in B} w_{ij}, \end{aligned} \quad (6)$$

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. [13]

Menggunakan dua pengukuran baru yaitu *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)*,

$$np_i = \frac{\sum_{j=1}^{n} w_{ij}\Pi[C(j) = C(i)]}{\sum_{j,k=1}^{n} w_{jk}\Pi[C(j) = C(k) = C(i)]}, j, k \neq i, \quad (7)$$

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. [13]

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. [13]

$$nr_i = \frac{|E_i|}{\sum_{j=1}^{n} w_{ij}\Pi[C(j) = C(i)]}. \quad (8)$$

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. [13]

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

$$Rank_i = \begin{cases} \exp(-\frac{1}{r(i)^2}), & r(i) \neq 0, \\ 0, & r(i) = 0, \end{cases} \textbf{ di mana } r(i) = (np_i) * log(nr_i) \quad (9)$$

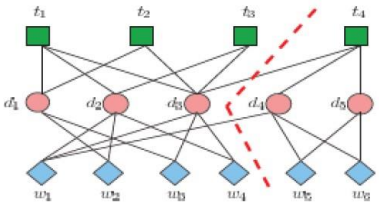
Berdasarkan persamaan di atas, fungsi perangkingan ini merupakan pengganti yang dihaluskan yang proporsional untuk presisi titik (*node precision*) dan *recall*, hasilnya berada di kisaran *range* 0 sampai 1. Potensi pengalikasian dari metodologi *bipartite graph node ranking* termasuk interpretasi relasi dokumen dan *author* yaitu menentukan relasi sosial (misal "*hub*" dan "i") 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.

B. 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.

$$W = \begin{pmatrix} 0 & A & 0 \\ A^T & 0 & B \\ 0 & B^T & 0 \end{pmatrix} \quad (10)$$

di mana *A* dan *B* sebagai matriks-matriks inter-relasi antara tag dengan dokumen dan dokumen dengan kata-kata. [13]



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*, perlu 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*. [13]

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

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

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

Online Recommendation

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

(11)

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 klister 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, akan merujuk kepada klaster yang diinginkan dengan mempartisi algoritima di dalam tahap pertama sebagai classess di dalam sesi selanjutnya. [13]

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*. [13]

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. [13]

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

$$P(D = d|C = k) = \sum_{m=1}^{M} \pi_m \Pi(F(m) = k) \prod_{j=1}^{p} \phi(d_j|\lambda_{j,m}), \quad (12)$$

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*, $\phi(dj\ |\lambda j,m) = e^{-\lambda j,m}(\lambda_{j,m})^{dj}\ /d_j\ !$. Pada jalur ini, setiap kelas adalah *mixture model* dengan distribusi yang multivariasi dengan memiliki variabel yang mengikuti distribusi *Poisson*. menunjukkan histogram pada dua *mixture* yang bisa dianggap sebagai *pmf* dari dua *Poisson mixture*.

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, terdapat asumsi 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 $L \ll p$):

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

(13)

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 $c(k, j) \in \{1, ...,L\}$, $k \in \{1, ...,K\}$, $j \in \{1, ..., p\}$. [13]

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

$$p_{i,m} \propto \pi_m^{(t)} \mathbb{I}(C(i)) \prod_{j=1}^{p} \theta\left(d(i,j) \mid \tilde{\lambda}_{m,i,j}^{(t)}\right) \quad (14)$$

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

$$L\left(\pi_m^{(t+1)}, \tilde{\lambda}_{m,l}^{(t+1)}, c^{(t+1)}(k,j) \mid \pi_m^{(t)}, \tilde{\lambda}_{m,l}^{(t)}, c^{(t)}(k,j)\right) = \max \sum_{i=1}^{n} \sum_{m=1}^{M} p_{i,m} \log\left(\pi_m^{(t+1)} \mathbb{I}(C(i)) \prod_{j=1}^{p} \theta\left(d(i,j) \mid \tilde{\lambda}_{m,i,j}^{(t+1)}\right)\right), \quad (15)$$

dan *update* parameter

$$\pi_m^{(t+1)} = \frac{\sum_{i=1}^{n} p_{i,m}}{\sum_{m'=1}^{M} \sum_{i=1}^{n} p_{i,m'}}, \quad (16)$$

$$\tilde{\lambda}_m^{(t+1)} = \frac{\sum_{i=1}^{n} p_{i,m} \sum_j d(i,j) \mathbb{I}(C(i))}{|d(i,j)| \sum_{i=1}^{n} p_{i,m}}, \quad (17)$$

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

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

$$c^{(t+1)}(k,j) = \arg\max_{l} \sum_{i=1}^{n} \sum_{m=1}^{M} \log\left(d(i,j) \mid \tilde{\lambda}_{m,l}^{(t+1)}\right) \quad (18)$$

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

$$P(C = k|D = d_t) = \frac{P(D = d_t|C = k)P(C = k)}{P(D = d_t)} = \frac{\sum_{m=1}^{M} \pi_m \mathbb{III}(F(m) = k) \prod_{l=1}^{L} \phi(d_{k,l}\tilde{\lambda}_{l,m}) P(C = k)}{P(D = d_t)} \quad (19)$$

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

$$R(T_i, d_t) = P(T = T_i|D = d_t) = Rank_{T_i} * P(C = x|D = d_t) \quad (20)$$

Dengan perangkingan tag dari terbesar ke terkecil pada probabilitas mereka, *top ranked tags* dipilih untuk rekomendasi. [13]

C. *Lanczos Iteration*

Berikut ini adalah algoritma *lanczos iteration*[4].

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

(21)

Dalam algoritma ini, nilai *alpha*, *beta*, dan q diperlukan untuk membentuk matriks Q_k dan T_k.

$$T_k = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \beta_2 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \beta_{k-1} \\ & & & \beta_{k-1} & \alpha_k \end{bmatrix}. \quad (22)$$

$$Q_k = \begin{pmatrix} q_1 & | & q_2 & | & \dots & | & q_k \end{pmatrix} \quad (23)$$

D. *Spectral Recursive Embedding*

Untuk membentuk *bipartite graph partition*, perlu algoritma *SRE*. Berikut adalah algoritma *SRE*. [17]

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 $G(A^c, B^c)$

(24)

III. METODOLOGI PENELITIAN

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

Mencari salah satu situs yang ingin di-crawling terlebih dahulu. Kemudian, lakukan crawling data dari situs tersebut menggunakan *crawler*[5]. 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 bias memprediksi tag yang diinginkan. Setelah itu, lakukan pengujian terhadap tag yang telah dihasilkan.

A. *Penginputan*

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

D, S, dan *T* didapat dari data yang telah diolah dari hasil *crawling*. Untuk *K, M,* dan *L*, menggunakan nilai dua pada penelitian ini.

Berikut adalah kode intinya.

```
def document_processing(dataset_document):
  id_before = 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 = []
  document_tag = []
  for data in dataset_document:
    if id_before != data[1]:
      id_list.append(id_before)
      tag_dictionary_list.append(tag_dictionary.copy())
      tags.append(len(tag_dictionary.copy()))
      tag_dictionary.clear()
      title_id_document.append((data[3].replace('| The Hill', ''), data[1]))
      word_dictionary_list.append(documentWordProcessing(data[2]))
      id_before = data[1]
    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()
  document_word = pandas.DataFrame(word_dictionary_list,
index=id_list)
  document_word = document_word.fillna(0)
  document_tag = pandas.DataFrame(tag_dictionary_list, index=id_list)
  document_tag = document_tag.fillna(0)
  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
```

B. *Menentukan Matriks W*

Untuk membentuk matriks *W* diperlukan matriks *A* dan matriks *B*. Matriks *A* didapat dari relasi antara *tag* dengan dokumen, sedangkan matriks *B* didapat dari relasi dokumen dengan *word*.

Di bawah ini merupakan contoh kodingan intinya.

```
def matrixABtoW(A, B):
  AT = A.transpose()
  BT = B.transpose()
  tag_count, document_count = A.shape
  document_count, word_count = B.shape
  all_count = tag_count + document_count + word_count
  W = np.zeros((all_count, all_count))
  for i in range(tag_count):
    W[i][tag_count:-word_count] = A[i]
  for i in range(1, word_count+1):
    W[-i][tag_count:-word_count]= BT[-i]
  for i in range(document_count):
    W[tag_count+i][0:tag_count] = AT[i]
  for i in range(document_count):
    W[tag_count+i][-word_count:] = B[i]
  return W
```

C. *Menghitung Low Rank Approximation Matrix Menggunakan Lanczos Iteration*

Pada algoritma *Lanczos Iteration*[4], ada beberapa inputan agar prosesnya bisa berjalan yaitu β_0, q_0, b, dan q_1. Setelah itu, nilai dari matriks Q_k dan T_k bias ditemukan.

Di bawah ini adalah contoh kode untuk mencati matriks *Q* dan *T*.

```
def lanczos_iteration(A, one_b = 1):
  k = 50
  row, column = A.shape
  T = np.zeros((k, k))
  Q = np.zeros((row, k))
  beta = 0
  q_before = 0
  b = ""
  if (one_b != 1):
    b = np.random.default_rng().random((row, 1))
  else:
    b = np.ones((row, 1))
  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]
    v = v - beta*q_before - alpha*q_now
    beta = normalization_vector(v)
    q_before = q_now
    q_now = v / beta
    T[i-1, i-1] = alpha
    if i < row:
      T[i-1, i] = beta
      T[i, i-1] = beta
      Q[:, i] = q_now.transpose()
  return Q, T
```

D. Melakukan partisi ˆW ke dalam klaster K menggunakan SRE

Untuk membuat algoritma SRE[17], ada lima langkah yang diperlukan.

Langkah pertama adalah mencari matriks $\hat{W}$ dengan menggunakan algoritma. 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 mencari keduanya, strategi tersimpelnya adalah dengan memasang $c_x = 0$ dan $c_y = 0$.

Langkah keempatnya membentuk suatu partisi dengan A dan A^C untuk vertex X dan B dan B^C untuk *vertex Y* .
Langkah kelima adalah mengulangi partisi pada sub *graph G(A,B)* dan *G(Ac,Bc)* jika diperlukan.

Berikut adalah kode inti untuk menjalankan *SRE*.

```
def spectral_recursive_embedding(W_hat, W):
  all_matrix = []
  all_cluster = []
  x, y = second_largest_singular_vector(W_hat)
  cx, cy = find_cut_point(x, y)
  A, Ac, B, Bc = form_partition(cx, cy, x, y)
  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)
  return all_matrix, all_cluster
```

E. 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. Berikut adalah langkah dalam melakukan pelabelan.

```
def assign_label_cluster(title_id_document, all_tag_list, all_word_list,
all_cluster):
  index = 0
  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
```

F. Menghitung Node Rank Rank(T) untuk Setiap Tag

Untuk menghitung *Rank(T)*, perlu menghitung np_i dan nr_i terlebih dahulu. Kemudian barulah menghitung *Rank(T)*. Berikut adalah kode intinya.

```
def node_rankt(tag_list, matrix_w_original, all_matrix_partition):
  all_tag_list_with_rank = []
  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)
  for tag, cluster, nodes in tag_list:
    index_cluster = 0
    rank_i_list = []
    np_i_list = []
    for k in cluster:
      np = all_np_list[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
```

G. Membuat Two Way Poisson Mixture Model

Dalam membuat ini, langkah awalnya adalah menghitung persamaan 12. Dilanjut dengan menghitung persamaan 14, 16, dan 17. Jika dirasa belum optimal, lakukan iterasi kembali di persamaan 14, 16, dan 17.

Kodingan tahap awal.

```
def p_im_list(doc_list, pi_m, word_list, dataframe_document_word, M):
  new_doc_list = []
  for title_id, cluster, indexes, word_count, m_component, probability in
doc_list:
    p_im = []
    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:
        prod_teta_list[m-1] *= probability_mass_function(word_value,
word_list[i][4][m-1])
      i+=1
    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
```

Kodingan tahap berikutnya.

```
def pi_m_with_t(doc_list, M):
  pi_m_list = np.zeros(M)
  sum_p_im_list = np.zeros(M)
  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
  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):
  new_word_list = [] # word list baru
  top_lambda_mt_list = np.zeros(M) # Untuk perhitungan pada persamaan
lambda_mt bagian atas
  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
  for word, cluster, indexes, word_count, lambda_m_j in word_list:
    lambda_m_j_temp = []
    for m in range(1, M+1):
      bottom_lambda_mt = word_count[m-1] * sum_p_im_list[m-1]
      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):
  new_doc_list = []
  for title_id, cluster, indexes, word_count, m_component, p_im,
probability in doc_list:
    p_im = []
    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]))
    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
```

H. Rekomendasi Tag Untuk Dokumen Baru

Lakukan melalui persamaan 19 lalu untuk mencari beberapa parameternya bias ditemukan dengan persamaan 11. Lalu, masukan dokumen yang baru ke dalam suatu klaster yang memiliki probabilitas terbesar. Berikut adalah kode inti dari alur tersebut.

```
def tag_recommendation(all_tag_list_with_rank, all_cluster,
total_doc_in_cluster, p_dt_ck):
  p_cluster = 1/len(all_cluster) # P(C=k)
  p_document = [1/tdic for tdic in total_doc_in_cluster
  R_Ti_dt = []
  all_tag_name = []
  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]
      rti = rank[index_cluster] * probability
      R_Ti_dt.append(rti)
      all_tag_name.append(tag)
      index_cluster += 1
```

I. Rekomendasi Tag Berdasarkan Ranks Tag

Langkah berikutnya adalah merekomendasikan *tag* berdasarkan klaster dari dokumen tersebut. Cara melakukannya adalah dengan menggunakan persamaan 20 pada setiap *tag* yang ada di klaster tersebut. Kemudian, lakukan pengurutan $R(T_i, d_t)$ dari yang terbesar ke yang terkecil. Kode ini merupakan masih dalam *function tag_recommendation* dan lanjutan dari kode di atas.

```
  dff = pandas.DataFrame([all_tag_name, R_Ti_dt], ["Tag", "Value"])
  dffT = dff.T.sort_values(by=['Value'], ascending=False)
  big_rank = [tag for tag in dffT.head(6)["Tag"]]
  return big_rank
```

J. Skenario Pengujian

Langkah pertama adalah Sumber data yang didapat dari hasil crawling dibagi menjadi dua bagian yaitu untuk data latih dan data uji. Kemudian, data uji mulai dimasukkan. Selanjutnya, Program akan memberikan beberapa prediksi tag pada data uji. Setelah itu, menghitung banyaknya tag yang cocok dengan data uji.

IV. HASIL DAN PEMBAHASAN

A. Implementasi

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.

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*.

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

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.

Data-data tersebut nantinya akan diolah menjadi matriks berdasarkan persamaan 10 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*.

Lalu, kedua matriks *A* dan *B* disatukan hingga menjadi matriks *W* berdasarkan persamaan 10 dengan menggunakan fungsi *matrixABtoW* pada file *matrix_processing.py*.

Kemudian, 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*.

Selanjutnya, mencari nilai dari matriks $\tilde{W}$ dengan menggunakan function *low_rank_approximation_matrix* pada *file low_rank_approximation_matrix.py*.

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*.

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*.

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*.

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.

Selanjutnya, mencari nilai $\lambda\tilde{}_m$ untuk setiap kata. Nilai $\lambda\tilde{}_m$ 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*.

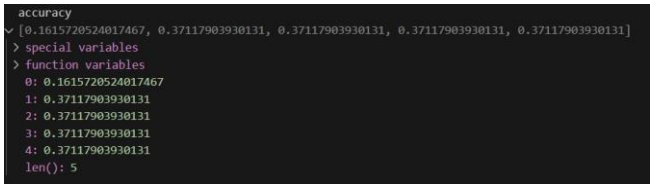
Kemudian, mencari nilai $p_{i,m}$ dalam setiap dokumen untuk memulai fase *E-step* dengan menggunakan fungsi *p_im_list* pada *file two_way_poisson_mixture_model.py*.

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*.

Gambar 1 Hasil rekomendasi tag

B. Hasil Pengujian

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



Gambar 2 Akurasi rekomendasi tag yang dihasilkan

C. Hasil Analisa

Berdasarkan hasil pengujian, jumlah akurasi terbilang sedikit yaitu 37%. Tentunya, terdapat beberapa faktor kenapa hasilnya rendah yaitu Penggunaan *K* yang sangat sedikit saat melakukan *Bipartite Graph Partition*, nilai *P(D = d|K = k)* yang terlalu sedikit juga menentukan rendahnya akurasi. dan Adanya kemungkinan hasil kodingan program yang kurang baik.

Selain itu, kelemahan kodingan program yang telah penulis buat adalah tidak bisa membagi Bipartite Graph Partition sesuai dengan rujukan aslinya. Jika program yang penulis buat mampu membagi Bipartite Graph Partition sesuai dengan K yang diinginkan, akurasinya bisa saja meningkat.

V. 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.

DAFTAR PUSTAKA

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

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

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

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

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

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

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

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

[9] Reynolds, D. (2009). Gaussian mixture model.

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

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

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

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

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

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

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

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