10

Dimensionality Reduction dan Representation Learning

> "The goal is to turn data into information, and information into insight."
>
> Carly Fiorina

Bab ini memuat materi yang relatif sulit (karena agak *high level*). Bab ini memuat materi autoencoder serta penerapannya pada pemrosesan bahasa alami (***natural language processing*** - NLP). Berhubung aplikasi yang diceritakan adalah aplikasi pada NLP, kami akan memberi sedikit materi (*background knowledge*) agar bisa mendapat gambaran tentang persoalan pada domain tersebut. Bagi yang tertarik belajar NLP, kami sarankan untuk membaca buku [56]. Teknik yang dibahas pada bab ini adalah ***representation learning*** untuk melakukan pengurangan dimensi pada *feature vector* (*dimensionality reduction*), teknik ini biasanya digolongkan sebagai *unsupervised learning*. Artinya, *representation learning* adalah mengubah suatu representasi menjadi bentuk representasi lain yang ekuvalen, tetapi berdimensi lebih rendah; sedemikian sehingga informasi yang terdapat pada representasi asli tidak hilang/terjaga. Ide dasar teknik ini bermula dari aljabar linear, yaitu dekomposisi matriks.

10.1 Curse of Dimensionality

Pada bab model linear, kamu telah mempelajari ide untuk mentransformasi data menjadi dimensi lebih tinggi agar data tersebut menjadi *linearly separable*. Pada bab ini, kamu mempelajari hal sebaliknya, yaitu mengurangi dimensi. *Curse of dimensionality* dapat dipahami secara mendalam apabila kamu membaca buku [57]. Untuk melakukan klasifikasi maupun *clustering*, kita membutuhkan fitur. Fitur tersebut haruslah dapat membedakan satu *instance* dan *instance* lainnya. Seringkali, untuk membedakan *instance* satu

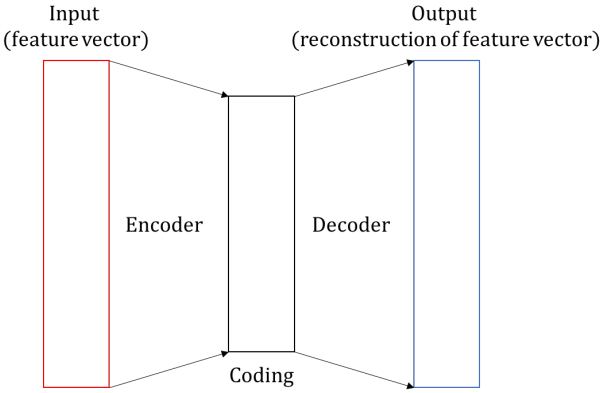
dan *instance* lainnya, kita membutuhkan *feature vector* yang berdimensi relatif "besar". Karena dimensi *feature vector* besar, kita butuh sumber daya komputasi yang besar juga. Untuk itu, terdapat metode-metode ***feature selection***[1] untuk memilih fitur-fitur yang dianggap "representatif" dibanding fitur lainnya. Sayangnya, bila kita menggunakan metode-metode *feature selection* ini, tidak jarang kita kelihangan informasi yang memuat karakteristik data. Dengan kata lain, ada karakteristik yang hilang saat menggunakan *feature selection*. Karena alasan tersebut, permasalahan ini disebut "*curse*".

Pertanyaan yang kita ingin jawab adalah apakah ada cara untuk merepresentasikan data ke dalam bentuk yang membutuhkan memori lebih sedikit tanpa adanya kehilangan informasi? Kita bisa menggunakan cara lain untuk mengurangi kompleksitas komputasi adalah dengan melakukan kompresi *feature vector*. ***Representation learning*** adalah metode untuk melakukan **kompresi** *feature vector*, umumnya (*typically*) menggunakan *neural network*[2]. Proses melakukan kompresi disebut ***encoding***, hasil *feature vector* dalam bentuk terkompres disebut ***coding***, proses mengembalikan hasil kompresi ke bentuk awal disebut (atau secara lebih umum, proses menginterpretasikan *coding*) ***decoding***. *Neural network* yang mampu melakukan hal ini disebut ***encoder*** [58, 59, 60, 61, 62]. Contoh *representation learning* paling sederhana kemungkinan besar adalah ***autoencoder*** yaitu *neural network* yang dapat merepresentasikan data kemudian merekonstruksinya kembali. Ilustrasi autoencoder dapat dilihat pada Gambar. 10.1. Karena tujuan *encoder* untuk kompresi, bentuk terkompresi haruslah memiliki dimensi lebih kecil dari dimensi *input*. *Neural network* mampu melakukan "kompresi" dengan baik karena ia mampu menemukan *hidden structure* dari data. Ukuran *utility function* atau *performance measure* untuk *autoencoder* adalah mengukur *loss*. Idealnya, *output* harus sama dengan *input*, yaitu *autoencoder* dengan tingkat *loss* 0%.

Contoh klasik lainnya adalah ***N-gram language modelling***, yaitu memprediksi kata y_t diberikan suatu konteks (*surrounding words*) misal kata sebelumnya y_{t-1} (bigram). Apabila kita mempunyai *vocabulary* sebesar $40,000$ berarti suatu bigram model membutuhkan *memory* sebesar $40,000^2$ (kombinatorial). Apabila kita ingin memprediksi kata diberikan *history* yang lebih panjang (misal dua kata sebelumnya - trigram) maka kita membutuhkan *memory* sebesar $40,000^3$. Artinya, *memory* yang dibutuhkan berlipat secara eksponensial. Tetapi, terdapat strategi menggunakan *neural network* dimana parameter yang dibutuhkan tidak berlipat secara eksponensial walau kita ingin memodelkan konteks yang lebih besar [63].

[1] `http://scikit-learn.org/stable/modules/feature_selection.html`

[2] Istilah *representation learning* pada umumnya mengacu dengan teknik menggunakan *neural network*.



Gambar 10.1. Contoh autoencoder sederhana.

10.2 Singular Value Decomposition

Sebelum masuk ke *autoencoder* secara matematis, penulis akan memberikan sedikit *overview* tentang dekomposisi matriks. Seperti yang sudah dijelaskan pada bab-bab sebelumnya, dataset dimana setiap instans direpresentasikan oleh *feature vector* dapat disusun menjadi matriks $\mathbf{D}$ berukuran $M \times N$, dimana M adalah banyaknya instans[3] dan N adalah dimensi fitur. Pada *machine learning*, dekomposisi atau reduksi dimensi sangat penting dilakukan terutama ketika dataset berupa *sparse matrix*. Matriks $\mathbf{D}$ dapat difaktorisasi menjadi tiga buah matriks, dimana operasi ini berkaitan dengan mencari eigenvectors, diilustrasikan pada persamaan 10.1.

$$\mathbf{D} = \mathbf{U}\ \mathbf{V}\ \mathbf{W} \tag{10.1}$$

dimana $\mathbf{U}$ berukuran $M \times L$, $\mathbf{V}$ berukuran $L \times L$, dan $\mathbf{W}$ berukuran $L \times N$. Perlu diperhatikan, matriks $\mathbf{V}$ adalah sebuah diagonal matriks (elemennya adalah nilai eigenvectors dari $\mathbf{D}$). Misalkan kita mempunyai sebuah matriks lain $\hat{\mathbf{V}}$ berukuran $K \times K$ ($K < L$), yaitu modifikasi matriks $\mathbf{V}$ dengan mengganti sejumlah elemen diagonalnya menjadi 0 (analogi seperti menghapus beberapa elemen yang dianggap kurang penting). Sebagai contoh, perhatikan ilustrasi berikut untuk $L = 3$ dan $K = 2$!

$$\mathbf{V} = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_L \end{bmatrix} \quad \hat{\mathbf{V}} = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_K & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Kita juga dapat me-nol-kan sejumlah baris dan kolom pada matriks $\mathbf{U}$ dan $\mathbf{W}$ menjadi $\hat{\mathbf{U}}$ ($M \times K$) dan $\hat{\mathbf{W}}$ ($K \times N$). Apabila kita mengalikan semuanya,

[3] Dapat dianalogikan dengan banyaknya training data.

kita akan mendapat matriks $\hat{\mathbf{D}}$ yang disebut *low rank approximation* dari matriks asli $\mathbf{D}$, seperti diilustrasikan pada persamaan 10.2.

$$\hat{\mathbf{D}} = \hat{\mathbf{U}}\ \hat{\mathbf{V}}\ \hat{\mathbf{W}} \tag{10.2}$$

Suatu baris dari matriks $\mathbf{E} = \hat{\mathbf{U}}\ \hat{\mathbf{V}}$ dianggap sebagai aproksimasi baris matriks $\mathbf{D}$ berdimensi tinggi [1]. Artinya, menghitung *dot-product* $\mathbf{E}_i \cdot \mathbf{E}_j = \hat{\mathbf{D}}_i \cdot \hat{\mathbf{D}}_j$. Artinya, operasi pada matriks aproksimasi (walaupun berdimensi lebih rendah), kurang lebih melambangkan operasi pada matriks asli. Konsep ini menjadi fundamental *autoencoder* yang akan dibahas pada bab berikutnya.

10.3 Ide Dasar Autoencoder

Seperti yang sudah dijelaskan *autoencoder* adalah *neural network* yang mampu merekonstruksi *input*. Ide dasar *autoencoder* tidak jauh dari konsep dekomposisi/*dimentionality reduction* menggunakan *singular value decomposition*. Diberikan dataset $\mathbf{D}$, kita ingin mensimulasikan pencarian matriks $\hat{\mathbf{D}}$ yang merupakan sebuah *low rank approximation* dari matriks asli. Arsitektur dasar *autoencoder* diberikan pada Gambar. 10.1. Kita memberi input matriks $\mathbf{D}$ pada *autoencoder*, kemudian ingin *autoencoder* tersebut menghasilkan matriks yang sama. Dengan kata lain, *desired output* sama dengan *input*. Apabila dihubungkan dengan pembahasan ANN pada bab sebelumnya, *error function* untuk melatih *autoencoder* diberikan pada persamaan 10.3, dimana $\mathbf{o}$ adalah output dari jaringan, θ adalah ANN (kumpulan *weight matrices*)[4], N adalah dimensi *output*, dan $\mathbf{d}_i$ adalah data ke-i (*feature vector* ke-i).

$$E(\theta) = \frac{1}{N}\sum_{i=j}^{N}\left(\mathbf{d}_{i[j]} - \mathbf{o}_{i[j]}\right)^2 \tag{10.3}$$

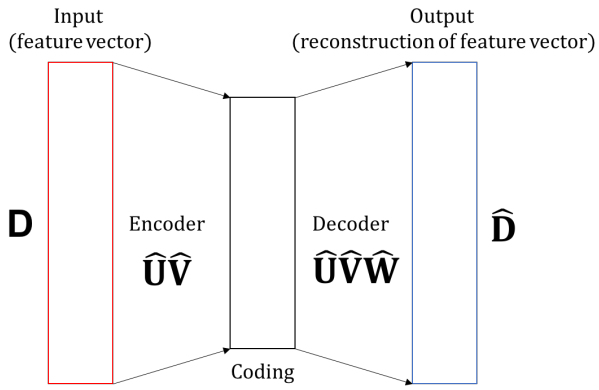
Persamaan 10.3 dapat kita tulis kembali sebagai persamaan 10.4, dimana f melambangkan fungsi aktivasi.

$$E(\theta) = \frac{1}{N}\sum_{j=1}^{N}\left(\mathbf{d}_{i[j]} - f(\mathbf{d}_i, \theta)_{[j]}\right)^2 \tag{10.4}$$

Seperti yang sudah dijelaskan sebelumnya, *desired output* sama dengan *input*. Tetapi seperti yang kamu ketahui, mencapai *loss* sebesar 0% adalah hal yang susah. Dengan demikian, kamu dapat memahami secara intuitif bahwa *autoencoder* melakukan aproksimasi terhadap data asli. Gambar. 10.2 mengilustrasikan hubungan antara *autoencoder* dan *singular value decomposition*[5]. Perhatikan, *hidden layer*/*coding* adalah $\mathbf{E} = \hat{\mathbf{U}}\ \hat{\mathbf{V}}$. Dengan kata lain, kita

[4] Pada banyak literatur, kumpulan *weight matrices* ANN sering dilambangkan dengan θ

[5] Hanya sebuah analogi.



Gambar 10.2. Hubungan autoencoder dan *singular value decomposition*.

dapat melakukan operasi *dot-product* pada *coding* untuk merepresentasikan *dot-product* pada data asli **D**. Ini adalah ide utama *autoencoder*, yaitu mengaproksimasi/mengkompresi data asli menjadi bentuk lebih kecil *coding*. Kemudian, operasi pada bentuk *coding* merepresentasikan operasi pada data sebenarnya.

Autoencoder terdiri dari *encoder* (sebuah *neural network*) dan *decoder* (sebuah *neural network*). *Encoder* merubah *input* ke dalam bentuk dimensi lebih kecil (dapat dianggap sebagai kompresi). *Decoder* berusaha merekonstruksi *coding* menjadi bentuk aslinya.

Sekarang kamu mungkin bertanya-tanya, bila *autoencoder* melakukan hal serupa seperti *singular value decomposition*, untuk apa kita menggunakan *autoencoder*? (mengapa tidak menggunakan aljabar saja?). Berbeda dengan teknik SVD, teknik *autoencoder* dapat juga mempelajari fitur non-linear[6]. Pada penggunaan praktis, *autoencoder* adalah *neural network* yang cukup kompleks (memiliki banyak *hidden layer*). Dengan demikian, kita dapat "mengetahui" berbagai macam representasi atau transformasi data. *Framework autoencoder* yang disampaikan sebelumnya adalah *framework* dasar. Pada kenyataannya, masih banyak ide lainnya yang bekerja dengan prinsip yang sama untuk mencari *coding* pada permasalahan khusus. *Output* dari *neural network* juga bisa tidak sama *input*-nya, tetapi tergantung permasalahan (kami akan memberikan contoh persoalan *word embedding*). Selain itu, *autoencoder* juga relatif fleksibel; dalam artian saat menambahkan data baru, kita hanya perlu memperbaharui parameter *autoencoder* saja. Kami sarankan untuk membaca *paper* [64, 65] perihal penjelasan lebih lengkap tentang perbedaan dan persamaan SVD dan *autoencoder* secara lebih matematis.

Apabila kamu hanya ingin mengerti konsep dasar *representation learning*, kamu dapat berhenti membaca sampai subbab ini. Secara sederhana

[6] Hal ini abstrak untuk dijelaskan karena membutuhkan pengalaman.

representation learning adalah teknik untuk mengkompresi *input* ke dalam dimensi lebih rendah tanpa (diharapkan) ada kehilangan informasi. Operasi vektor (dan lainnya) pada level *coding* merepresentasikan operasi pada bentuk aslinya. Untuk pembahasan *autoencoder* secara lebih matematis, kamu dapat membaca pranala ini[7]. Apabila kamu ingin mengetahui lebih jauh contoh penggunaan *representation learning* secara lebih praktis, silahkan lanjutkan membaca materi subbab berikutnya. Buku ini akan memberikan contoh penggunaan *representation learning* pada bidang *natural language processing* (NLP).

10.4 Representing Context: Word Embedding

Pada domain NLP, kita ingin komputer mampu mengerti bahasa selayaknya manusia mengerti bahasa. Misalkan komputer mampu mengetahui bahwa "meja" dan "kursi" memiliki hubungan yang erat. Hubungan seperti ini tidak dapat terlihat berdasarkan teks tertulis, tetapi kita dapat menyusun kamus hubungan kata seperti WordNet[8]. WordNet memuat ontologi kata seperti hipernim, antonim, sinonim. Akan tetapi, hal seperti ini tentu sangat melelahkan, seumpama ada kata baru, kita harus memikirkan bagaimana hubungan kata tersebut terhadap seluruh kamus yang sudah dibuat. Pembuatan kamus ini memerlukan kemampuan para ahli linguistik.

Oleh sebab itu, kita harus mencari cara lain untuk menemukan hubungan kata ini. Ide utama untuk menemukan hubungan antarkata adalah ***statistical semantics hypothesis*** yang menyebutkan pola penggunaan kata dapat digunakan untuk menemukan arti kata [66]. Contoh sederhana, kata yang muncul pada "**konteks**" yang sama cenderung memiliki makna yang sama. Perhatikan "konteks" dalam artian NLP adalah kata-kata sekitar (*surrounding words*)[9]; contohnya kalimat "budi menendang bola", "konteks" dari "bola" adalah "budi menendang". Kata "cabai" dan "permen" pada kedua kalimat "budi suka cabai" dan "budi suka permen" memiliki kaitan makna, dalam artian keduanya muncul pada konteks yang sama. Sebagai manusia, kita tahu ada keterkaitan antara "cabai" dan "permen" karena keduanya bisa dimakan.

Berdasarkan hipotesis tersebut, kita dapat mentransformasi kata menjadi sebuah bentuk matematis dimana kata direpresentasikan oleh pola penggunaannya [56]. Arti kata ***embedding*** adalah transformasi kata (beserta konteksnya) menjadi bentuk matematis (vektor). "Kedekatan hubungan makna" (***semantic relationship***) antarkata kita harapkan dapat tercermin pada operasi vektor. Salah satu metode sederhana untuk merepresentasikan kata sebagai vektor adalah ***Vector Space Model***. Konsep *embedding* dan *autoencoder*

[7] `https://jaan.io/what-is-variational-autoencoder-vae-tutorial/`

[8] `https://wordnet.princeton.edu/`

[9] Selain *surrounding words*, konteks dalam artian NLP dapat juga berupa kalimat, paragraph, atau dokumen.

	Dokumen 1	Dokumen 2	Dokumen 3	Dokumen 4	⋯
King	1	0	0	0	⋯
Queen	0	1	0	1	⋯
Prince	1	0	1	0	⋯
Princess	0	1	0	1	⋯
⋯					

Tabel 10.1. Contoh 1-of-V encoding.

sangatlah dekat, tapi kami ingin menakankan bahwa *embedding* adalah bentuk representasi konteks.

Semantic relationship dapat diartikan sebagai ***attributional*** atau ***relational similarity***. *Attributional similarity* berarti dua kata memiliki atribut/sifat yang sama, misalnya anjing dan serigala sama-sama berkaki empat, menggongong, serta mirip secara fisiologis. *Relational similarity* berarti derajat korespondensi, misalnya *anjing* : *menggongong* memiliki hubungan yang erat dengan *kucing* : *mengeong*.

10.4.1 Vector Space Model

Vector space model (VSM)[10] adalah bentuk *embedding* yang relatif sudah cukup lama tapi masih digunakan sampai saat ini. Pada pemodelan ini, kita membuat sebuah matriks dimana baris melambangkan kata, kolom melambangkan dokumen. Metode VSM ini selain mampu menangkap hubungan antarkata juga mampu menangkap hubungan antardokumen (*to some degree*). Asal muasalnya adalah *statistical semantics hypothesis*. Tiap sel pada matriks berisi nilai 1 atau 0. 1 apabila $kata_i$ muncul di $dokumen_i$ dan 0 apabila tidak. Model ini disebut ***1-of-V/1-hot encoding*** dimana V adalah ukuran kosa kata. Ilustrasi dapat dilihat pada Tabel. 10.1.

Akan tetapi, *1-of-V encoding* tidak menyediakan banyak informasi untuk kita. Dibanding sangat ekstrim saat mengisi sel dengan nilai 1 atau 0 saja, kita dapat mengisi sel dengan frekuensi kemunculan kata pada dokumen, disebut ***term frequency*** (TF). Apabila suatu kata muncul pada banyak dokumen, kata tersebut relatif tidak terlalu "penting" karena muncul dalam berbagai konteks dan tidak mampu membedakan hubungan dokumen satu dan dokumen lainnya (***inverse document frequency***/IDF). Formula IDF diberikan pada persamaan 10.5. Tingkat kepentingan kata berbanding terbalik dengan jumlah dokumen dimana kata tersebut dimuat. N adalah banyaknya dokumen, $\|d \epsilon D; t \epsilon d\|$ adalah banyaknya dokumen dimana kata t muncul.

$$IDF(t, D) = log\left(\frac{N}{|d \epsilon D; t \epsilon d|}\right) \tag{10.5}$$

[10] Mohon bedakan dengan VSM (*vector space model*) dan SVM (*support vector machine*)

Dengan menggunakan perhitungan TF-IDF yaitu $TF*IDF$ untuk mengisi sel pada matriks Tabel. 10.1, kita memiliki lebih banyak informasi. TF-IDF sampai sekarang menjadi *baseline* pada ***information retrieval***. Misalkan kita ingin menghitung kedekatan hubungan antar dua dokumen, kita hitung ***cosine distance*** antara kedua dokumen tersebut (vektor suatu dokumen disusun oleh kolom pada matriks). Apabila kita ingin menghitung kedekatan hubungan antar dua kata, kita hitung *cosine distance* antara kedua kata tersebut dimana vektor suatu kata merupakan baris pada matriks. Tetapi seperti intuisi yang mungkin kamu miliki, mengisi *entry* dengan nilai TF-IDF pun akan menghasilkan *sparse matrix*.

Statistical semantics hypothesis diturunkan lagi menjadi empat macam hipotesis [66]:

1. *Bag of words*
2. *Distributional hypothesis*
3. *Extended distributional hypothesis*
4. *Latent relation hypothesis*

Silakan pembaca mencari sumber tersendiri untuk mengerti keempat hipotesis tersebut atau membaca *paper* Turney dan Pantel [66].

10.4.2 Sequential, Time Series, dan Compositionality

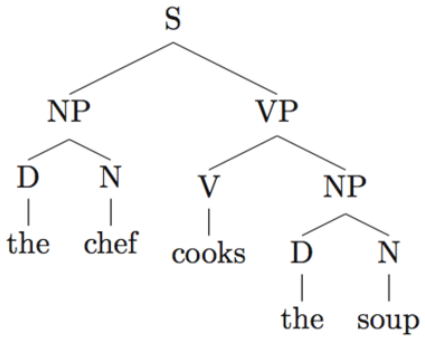
Bahasa manusia memiliki dua macam karakteristik yaitu adalah data berbentuk ***sequential data*** dan memenuhi sifat ***compositionality***. *Sequential data* adalah sifat data dimana suatu kemunculan $data_i$ dipengaruhi oleh data sebelumnya ($data_{i-1}, data_{i-2}, ...$). Perhatikan kedua kalimat berikut:

1. Budi melempar bola.
2. Budi melempar gedung bertingkat.

Pada kedua kalimat tersebut, kalimat pertama lebih masuk akal karena bagaimana mungkin seseorang bisa melempar "gedung bertingkat". Keputusan kita dalam memilih kata berikutnya dipengaruhi oleh kata-kata sebelumnya, dalam hal ini "Budi melempar" setelah itu yang lebih masuk akal adalah "bola". Contoh lain adalah data yang memiliki sifat ***time series*** yaitu gelombang laut, angin, dan cuaca. Kita ingin memprediksi data dengan rekaman masa lalu, tapi kita tidak mengetahui masa depan. Kita mampu memprediksi cuaca berdasarkan rekaman parameter cuaca pada hari-hari sebelumnya. Ada yang berpendapat beda *time series* dan *sequential* (sekuensial) adalah diketahuinya sekuens kedepan secara penuh atau tidak. Penulis tidak dapat menyebutkan *time series* dan sekuensial sama atau beda, silahkan pembaca menginterpretasikan secara bijaksana.

Data yang memenuhi sifat ***compositionality*** berarti memiliki struktur hirarkis. Struktur hirarkis ini menggambarkan bagaimana unit-unit lebih kecil berinteraksi sebagai satu kesatuan. Artinya, interpretasi/pemaknaan unit

yang lebih besar dipengaruhi oleh interpretasi/pemaknaan unit lebih kecil (subunit). Sebagai contoh, kalimat "saya tidak suka makan cabai hijau". Unit "cabai" dan "hijau" membentuk suatu frasa "cabai hijau". Mereka tidak bisa dihilangkan sebagai satu kesatuan makna. Kemudian interaksi ini naik lagi menjadi kegiatan "makan cabai hijau" dengan keterangan "tidak suka", bahwa ada seseorang yang "tidak suka makan cabai hijau" yaitu "saya". Pemecahan kalimat menjadi struktur hirarkis berdasarkan *syntactical role* disebut ***constituent parsing***, contoh lebih jelas pada Gambar. 10.3. *N* adalah *noun*, *D* adalah *determiner*, *NP* adalah *noun phrase*, *VP* adalah *verb phrase*, dan *S* adalah *sentence*. Selain bahasa manusia, gambar juga memiliki struktur hirarkis. Sebagai contoh, gambar rumah tersusun atas tembok, atap, jendela, dan pintu. Tembok, pintu, dan jendela membentuk bagian bawah rumah; lalu digabung dengan atap sehingga membentuk satu kesatuan rumah.



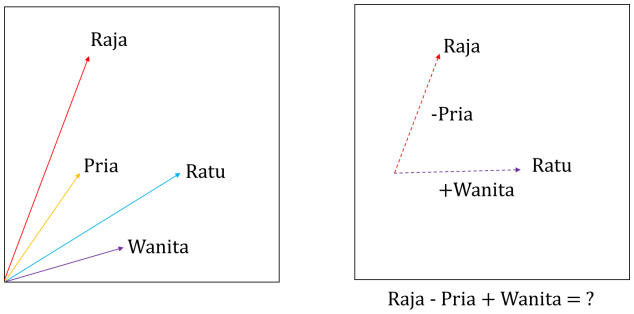
Gambar 10.3. Contoh Constituent Tree[11].

10.4.3 Distributed Word Representation

Seperti yang disebutkan pada bagian sebelumnya, kita ingin hubungan kata (yang diinferensi dari konteksnya) dapat direpresentasikan sebagai operasi vektor seperti pada ilustrasi Gambar. 10.4. Kata "raja" memiliki sifat-sifat yang dilambangkan oleh suatu vektor (misal 90% aspek loyalitas, 80% kebijaksanaan, 90% aspek kebangsaan, dst), begitu pula dengan kata "pria", "wanita", dan "ratu". Jika sifat-sifat yang dimiliki "raja" dihilangkan bagian sifat-sifat "pria"-nya, kemudian ditambahkan sifat-sifat "wanita" maka idealnya operasi ini menghasilkan vektor yang dekat kaitannya dengan "ratu". Dengan kata lain, raja yang tidak maskulin tetapi fenimin disebut ratu. Seperti yang disebutkan sebelumnya, ini adalah tujuan utama *embedding* yaitu merepresentasikan "makna" kata sebagai vektor sehingga kita dapat memanipulasi banyak hal berdasarkan operasi vektor. Hal ini mirip (tetapi tidak sama)

[11] source: Pinterest

dengan prinsip *singular value decomposition* dan *autoencoder* yang telah dijelaskan sebelumnya.

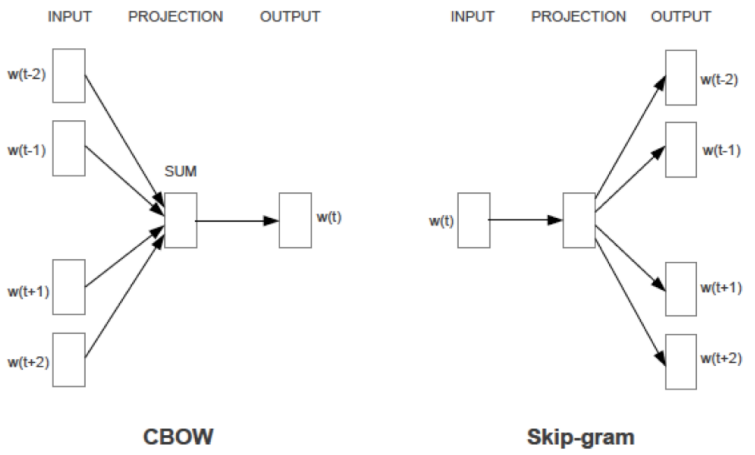


Gambar 10.4. Contoh Operasi Vektor Kata.

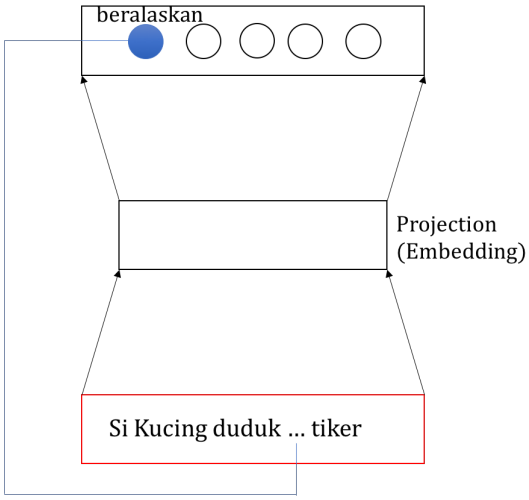
Selain *vector space model*, apakah ada cara lain yang mampu merepresentasikan kata dengan lebih baik? Salah satu kekurangan VSM adalah tidak memadukan sifat sekuensial pada konstruksi vektornya. Cara lebih baik ditemukan oleh [40, 41] dengan ekstensi pada [62]. Idenya adalah menggunakan teknik *representation learning* dan prinsip *statistical semantics hypothesis*. Metode ini lebih dikenal dengan sebutan *word2vec*. Tujuan *word2vec* masih sama, yaitu merepresentasikan kata sebagai vektor, sehingga kita dapat melakukan operasi matematis terhadap kata. *Encoder*-nya berbentuk ***Continous bag of words*(CBOW)** atau **Skip Gram**. Pada CBOW, kita memprediksi kata diberikan suatu "konteks". Pada arsitektur "Skip Gram" kita memprediksi konteks, diberikan suatu kata. Ilustrasi dapat dilihat pada Gambar. 10.5. Bagian *projection layer* pada Gambar. 10.5 adalah *coding* layer. Kami akan memberikan contoh CBOW secara lebih detil.

Perhatikan Gambar. 10.6. Diberikan sebuah konteks "si kucing duduk ... tiker". Kita harus menebak apa kata pada "..." tersebut. Dengan menggunakan teknik autoencoder, *output layer* adalah distribusi probabilitas $kata_i$ pada konteks tersebut. Kata yang menjadi jawaban adalah kata dengan probabilitas terbesar, misalkan pada kasus ini adalah "beralaskan". Dengan arsitektur ini, prinsip sekuensial atau *time series* dan *statistical semantics hypothesis* terpenuhi (*to a certain extent*). Teknik ini adalah salah satu contoh penggunaan *neural network* untuk *unsupervised learning*. Kita tidak perlu mengkorespondensikan kata dan *output* yang sesuai karena *input vektor* didapat dari statistik penggunaan kata. Agar lebih tahu kegunaan vektor kata, kamu dapat mencoba kode dengan bahasa pemrograman Python 2.7 yang disediakan penulis[12]. Buku ini telah menjelaskan ide konseptual *word embedding* pada level abstrak. Apabila kamu tertarik untuk memahami detilnya secara matem-

[12] `https://github.com/wiragotama/GloVe_Playground`



Gambar 10.5. CBOW vs Skip Gram [41].



Gambar 10.6. CBOW.

atis, kamu dapat membaca berbagai penelitian terkait[13]. Silahkan baca *paper* oleh Mikolov [40, 41] untuk detil implementasi *word embedding*.

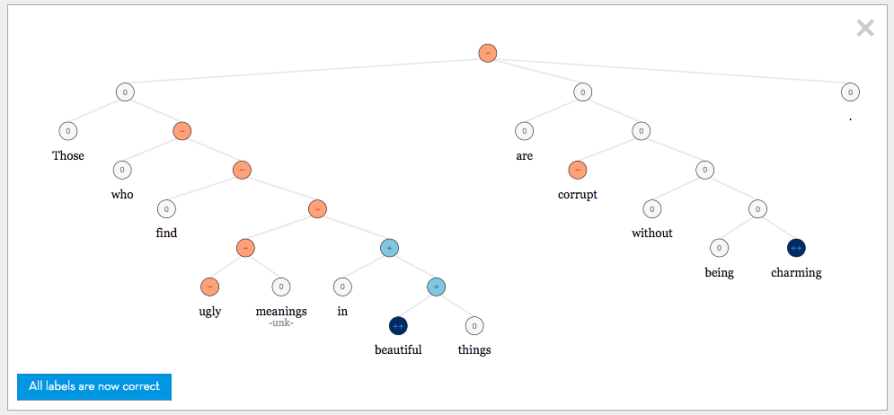
10.4.4 Distributed Sentence Representation

Kita sudah dapat merepresentasikan kata menjadi vektor, selanjutnya kita ingin mengonversi unit lebih besar (kalimat) menjadi vektor. Salah satu cara

[13] Beberapa orang berpendapat bahwa *evil is in the detail*.

paling mudah adalah menggunakan nilai rata-rata representasi *word embedding* untuk semua kata yang ada pada kalimat tersebut (*average of its individual word embeddings*). Cara ini sering digunakan pada bidang NLP dan cukup *powerful*, sebagai contoh pada *paper* oleh Putra dan Tokunaga [67]. Pada NLP, sering kali kalimat diubah terlebih dahulu menjadi vektor sebelum dilewatkan pada algoritma *machine learning*, misalnya untuk analisis sentimen (kalimat bersentimen positif atau negatif). Vektor ini yang nantinya menjadi *feature vector* bagi algoritma *machine learning*.

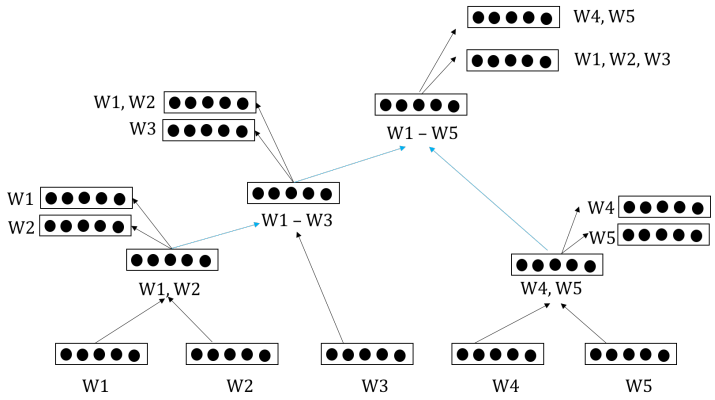
Kamu sudah tahu bagaimana cara mengonversi kata menjadi vektor, untuk mengonversi kalimat menjadi vektor cara sederhananya adalah merata-ratakan nilai vektor kata-kata pada kalimat tersebut. Tetapi dengan cara sederhana ini, sifat sekuensial dan *compositional* pada kalimat tidak terpenuhi. Sebagai contoh, kalimat "anjing menggigit Budi" dan "Budi menggigit anjing" akan direpresentasikan sebagai vektor yang sama karena terdiri dari kata-kata yang sama. Dengan demikian, representasi kalimat sederhana dengan merata-ratakan vektor kata-katanya juga tidaklah sensitif terhadap urutan[14]. Selain itu, rata-rata tidak sensitif terhadap *compositionality*. Misal frase "bukan sebuah pengalaman baik" tersusun atas frase "bukan" yang diikuti oleh "sebuah pengalaman baik". Rata-rata tidak mengetahui bahwa "bukan" adalah sebuah *modifier* untuk sebuah frase dibelakangnya. Sentimen dapat berubah bergantung pada komposisi kata-katanya (contoh pada Gambar. 10.7).



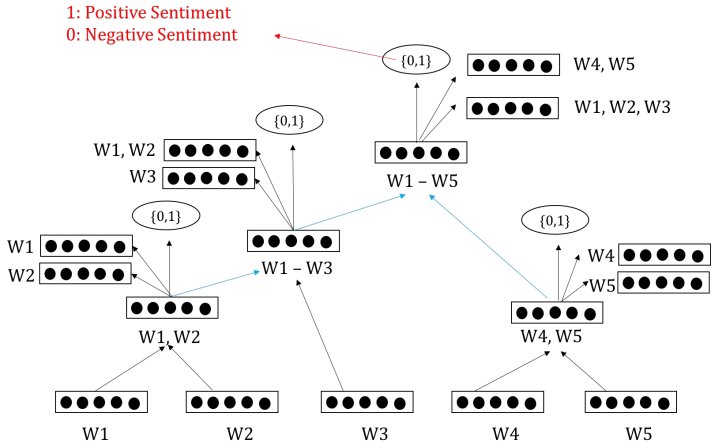
Gambar 10.7. Contoh analisis sentimen (Stanford)[15]

Cara lainnya adalah meng-*encode* kalimat sebagai *vektor* menggunakan *recursive autoencoder*. *Recursive* berarti suatu bagian adalah komposisi dari bagian lainnya. Penggunaan *recursive autoencoder* sangat rasional berhubung

[14] Karena ini *recurrent neural network* bagus untuk *language modelling*.
[15] `http://nlp.stanford.edu:8080/sentiment/rntnDemo.html`



Gambar 10.8. Contoh recursive autoencoder.



Gambar 10.9. Contoh recursive autoencoder dengan sentiment[59].

data memenuhi sifat *compositionality* yang direpresentasikan dengan baik oleh topologi *recursive neural network*. Selain itu, urutan susunan kata-kata juga tidak hilang. Untuk melatih *recursive autoencoder*, *output* dari suatu layer adalah rekonstruksi *input*, ilustrasi dapat dilihat pada Gambar. 10.8. Pada setiap langkah *recursive*, *hidden layer/coding layer* berusaha men-*decode* atau merekonstruksi kembali vektor *input*.

Lebih jauh, untuk sentimen analisis pada kata, kita dapat menambahkan output pada setiap *hidden layer*, yaitu sentimen unit gabungan, seperti pada Gambar. 10.9. Selain menggunakan *recursive autoencoder*, kamu juga dapat menggunakan *recurrent autoencoder*. Kami silahkan pada pembaca untuk memahami *recurrent autoencoder*. Prinsipnya mirip dengan *recursive autoencoder*.

Teknik yang disampaikan mampu mengonversi kalimat menjadi vektor, lalu bagaimana dengan paragraf, satu dokumen, atau satu frasa saja? Teknik umum untuk mengonversi teks menjadi vektor dapat dibaca pada [61] yang lebih dikenal dengan nama ***paragraph vector*** atau ***doc2vec***.

10.5 Tips

Bab ini menyampaikan penggunaan *neural network* untuk melakukan *kompresi data* (*representation learning*) dengan teknik *unsupervised learning*. Hal yang lebih penting untuk dipahami bahwa ilmu *machine learning* tidak berdiri sendiri. Walaupun kamu menguasai teknik *machine learning* tetapi tidak mengerti domain dimana teknik tersebut diaplikasikan, kamu tidak akan bisa membuat *learning machine* yang memuaskan. Contohnya, pemilihan fitur *machine learning* pada teks (NLP) berbeda dengan gambar (*visual processing*). Mengerti *machine learning* tidak semata-mata membuat kita bisa menyelesaikan semua macam permasalahan. Tanpa pengetahuan tentang domain aplikasi, kita bagaikan orang buta yang ingin menyetir sendiri!

Soal Latihan

10.1. Feature Selection
Sebutkan dan jelaskan berbagai macam teknik *feature selection*!

10.2. LSI dan LDA

(a) Jelaskanlah *matrix factorization* dan *principal component analysis*!
(b) Jelaskanlah Latent Semantic Indexing (LSI) dan Latent Dirichlet Allocation (LDA)!
(c) Apa persamaan dan perbedaan antara LSI, LDA, dan Autoencoder?

10.3. Variational Autoencoder
Jelaskan apa itu *Variational autoencoder*! Deskripsikan perbedaannya dengan *autoencoder* yang sudah dijelaskan pada bab ini?