BAB 2

LANDASAN TEORI

2.1 Representasi Teks

Data yang diperoleh secara mentah (*raw data*) biasanya masih berbentuk teks. Teks tersebut harus diubah menjadi data yang memiliki struktur agar dapat digunakan sebagai *input*, sesuai dengan algoritma dan model yang akan digunakan, misalnya data tabular (Žižka, dkk., 2019).

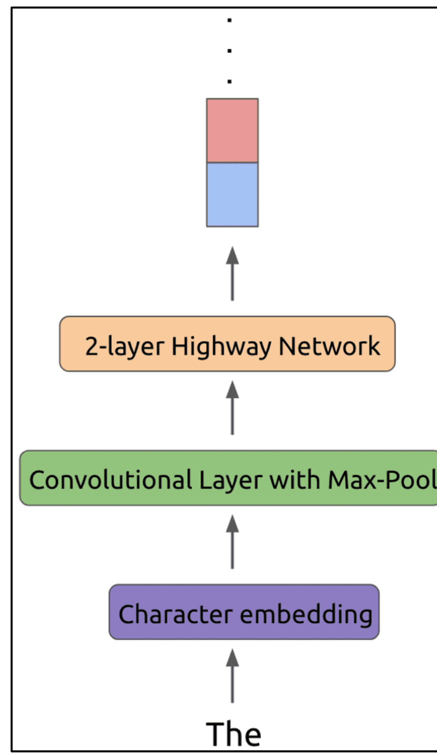
Berdasarkan TensorFlowHub (2020), *pre-trained* ELMo dapat menerima 2 bentuk *input*, yaitu kalimat atau *untokenized sentences* dan kata atau *tokenized sentence. Input* berupa kalimat menggunakan pembatas antar kata menggunakan spasi, sementara *input* berupa kata menggunakan pembatas berupa koma. Oleh sebab itu, representasi teks yang dilakukan dapat berupa *sentence detection* atau *tokenization*, tergantung dengan pilihan *input* pengguna (Žižka, dkk., 2019).

2.2 Embedding from Language Model (ELMo)

ELMo merupakan *deep contextualized word embeddings* yang menggunakan CNN dan kombinasi linear dari *internal states* lapisan model Long Short-Term Memory (LSTM) dua arah (BiLM) (Peters, dkk., 2018). ELMo merepresentasikan kata dengan melihat konteks kata dalam kalimat terkait, sehingga dapat memperoleh informasi *syntactic* dan *semantic* (Peters, dkk., 2018).

ELMo menerima *input* awal berupa kalimat yang setiap katanya akan direpresentasikan menjadi *token* menggunakan CNN karakter yang terdiri dari *character embedding, convolutional layer* dengan *maxpool*, serta 2 *layer highway*

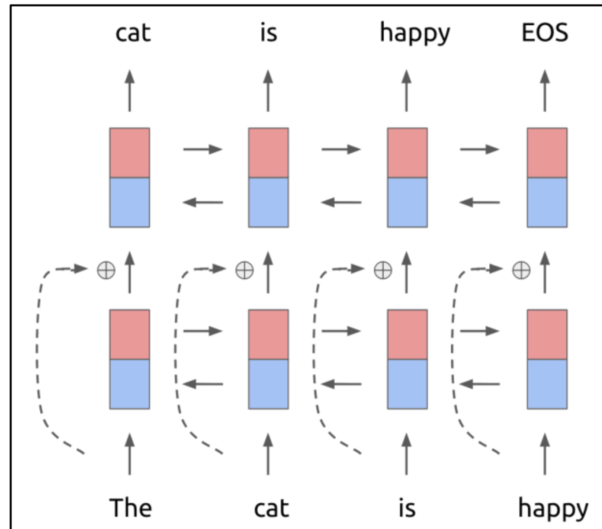
network (Bouadjenek, 2018). Selanjutnya *output* dengan dimensi 512 tersebut akan dijadikan *input* dari dua *layer bidirectional* LSTM. (Reimers, dkk., 2019).



Gambar 2.1 Transformasi untuk Setiap Token

Sumber: (Bouadjenek, 2018).

Dalam *bidirectional* LSTM, *input* awal akan diberikan kepada *forward layer* dan *backward layer* secara terpisah, kemudian menggabungkan kedua *output*-nya untuk dijadikan *input* dari BiLSTM *layer* selanjutnya. Antara *layer* pertama dan kedua BiLSTM, ditambahkan koneksi residual untuk mengurangi nilai gradien yang hilang akibat penggunaan *activation function* dari *backpropagation* (Bouadjenek, 2018).



Gambar 2.2 Bidirectional Language Model

Sumber: (Bouadjenek, 2018)

Bidirectional Language Model mengombinasikan *Forward Recurrent Unit Model* (yang direpresentasikan dengan kotak berwarna merah) dan *Backward Recurrent Unit Model* (yang direpresentasikan dengan kotak berwarna biru). *Forward* LM dapat menghitung kemungkinan *token* saat ini dengan mengetahui *token* sebelumnya, sementara *Backward* LM dapat menghitung *token* saat ini dengan melakukan perhitungan menggunakan *token* selanjutnya. Atas alasan tersebut, ELMo dapat melihat konteks kata melalui informasi yang diperoleh dari urutan kata sebelum dan sesudah kata tersebut (Peters, dkk., 2018).

Hasil representasi teks ELMo memperhitungkan hasil dari ketiga layer menjadi 1 vektor menggunakan rumus berikut (Peters, dkk., 2018):

$$ELMO_k^{task} = \gamma^{task} \sum_{j=0}^{L} s_j^{task} \, h_{k,j}^{LM} \qquad \ldots(2.1)$$

dengan s^{task} adalah *weight soft max* yang telah dinormalisasi, γ^{task} adalah tingkat kegunaan ELMo dalam *task* tersebut, h^{LM} merupakan *hidden state* dari *language model*, serta *k* merupakan urutan dari kata yang akan diubah menjadi vektor.

$$\text{ELMo}_k^{\text{task}} = \gamma^{\text{task}} \times \sum \begin{cases} s_2^{\text{task}} \times h_{k2}^{LM} & \overrightarrow{h}_{k2}^{LM} \quad \overleftarrow{h}_{k2}^{LM} \\ s_1^{\text{task}} \times h_{k1}^{LM} & \overrightarrow{h}_{k1}^{LM} \quad \overleftarrow{h}_{k1}^{LM} \\ s_0^{\text{task}} \times h_{k0}^{LM} & x_k \quad x_k \end{cases}$$

Gambar 2.3 Perhitungan ELMo untuk Setiap Layer

Sumber: (Chen, 2020)

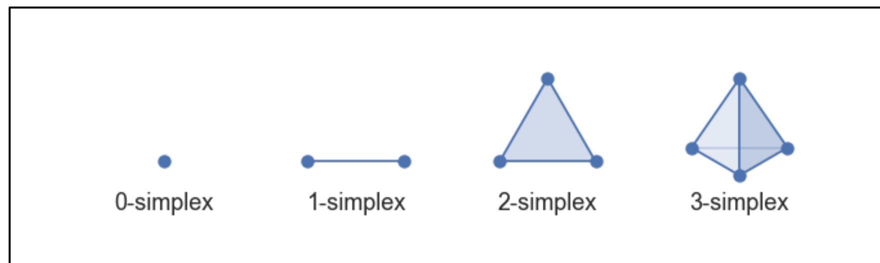
ELMo menyediakan model *pre-trained,* sehingga tidak memerlukan data yang besar untuk melakukan *training model* lagi. *Pre-trained* model dapat secara langsung digunakan untuk mengubah teks menjadi vektor.

2.3 Uniform Manifold Approximation and Projection (UMAP)

UMAP merupakan salah satu teknik yang berfungsi untuk mengurangi dimensi data dengan menggunakan teori topologi and *Riemannian geometry* (McInnes, dkk., 2018). Secara singkat, UMAP bekerja dengan mengoptimalkan representasi grafik dalam dimensi rendah semirip mungkin dengan representasi data dalam dimensi tinggi (Coenen dan Pearce, 2019). Optimasi tersebut dihitung menggunakan *cross entropy* (McInnes, 2020).

Pembuatan grafik awal berdimensi tinggi dilakukan menggunakan Čech *complex*, yaitu cara untuk merepresentasikan topologi menggunakan data yang terbatas (*finite sets*) melalui *simplicial complex* (Coenen dan Pearce, 2019). *Simplicial complex* merupakan kumpulan *simplex* yang saling terhubung, sementara *simplex* adalah objek dalam dimensi k yang terbentuk dengan cara menghubungkan k+1 titik (McInnes, 2020). Jika seluruh data diibaratkan seperti *simplex* kemudian dikombinasikan sedemikian rupa hingga terbentuk Čech

complex, maka dapat diperoleh representasi topologi yang baik dan terbukti secara teoritis (Coenen dan Pearce, 2019).



Gambar 2.4 *Simplices* dalam Dimensi Rendah

Sumber: (McInnes, 2020)

Untuk menghubungkan data, dibuat radius dari masing-masing *simplex* menggunakan *Riemannian geometry* (McInnes, 2020). Apabila suatu *simplex* terletak dalam radius dari *simplex* lain, maka keduanya akan saling terhubung. Melalui implementasi *Riemannian geometry,* radius masing-masing *simplex* akan disesuaikan dengan letak k-th *nearest neighbour simplex* tersebut, dengan nilai k berupa ukuran sampel untuk menghitung *local distance* (McInnes, 2020). Kemudian diimplementasikan *fuzzy topology* dalam radius *patch* sehingga hubungan antar data direpresentasikan dengan nilai antara 0 hingga 1, bukan menggunakan bilangan biner 0 atau 1 (McInnes, 2020). *Output* dari proses ini yaitu *weighted graph* yang memiliki *edge weights* sebagai representasi seberapa dekat letak kedua titik tersebut terhubung (Coenen dan Pearce, 2019).

Representasi data ke dimensi yang lebih rendah dilakukan dengan mencari kembali struktur *fuzzy topological* dari data, dengan *manifold Euclidean space* dimensi rendah. Selanjutnya dilakukan optimasi untuk menemukan representasi dimensi rendah terbaik dengan mencari hasil struktur *fuzzy topological* dari dimensi rendah yang paling mendekati struktur *fuzzy topological* dimensi tinggi (McInnes, 2020).

Optimasi dihitung menggunakan algoritma *force directed graph layout* melalui *cross-entropy* Stochastic Gradient Descent dan *negative sampling*-nya, dengan rumus sebagai berikut (McInnes, 2020):

$$\sum_{e \in E} w_h(e) \log\left(\frac{w_h(e)}{w_l(e)}\right) + \left(1 - w_h(e)\right) \log\left(\frac{1 - w_h(e)}{1 - w_l(e)}\right) \quad \text{...(2.2)}$$

dengan E merupakan set dari seluruh 1-simplices, $w_h(e)$ adalah *weight* dari 1-*simplex* e dalam dimensi tinggi, dan $w_l(e)$ adalah *weight* dari *simplex* e dalam dimensi rendah. $w_h(e) \log(\frac{w_h(e)}{w_l(e)})$ berperan sebagai *attractive force* antar e setiap terdapat *weight* dalam dimensi tinggi yang bernilai besar, sementara $\left(1 - w_h(e)\right) \log(\frac{1 - w_h(e)}{1 - w_l(e)})$ berperan sebagai *repulsive force* antar verteks dari e setiap *weight* dalam dimensi tinggi bernilai kecil.

Dua parameter utama dari UMAP adalah n_neighbors dan min_dist. Parameter n_neighbors menentukan seberapa seimbangnya struktur data lokal dan global (McInnes, 2020). Semakin rendah nilai n_neighbors, maka model akan lebih berfokus pada struktur data lokal sehingga jarak antar *cluster* cenderung diabaikan, begitu pula sebaliknya (Coenen dan Pearce, 2019). Sementara parameter min_dist menentukan jarak minimum dari data dalam dimensi rendah. Semakin besar nilai min_dist, maka representasi data menjadi berjauhan dan lebih berfokus pada struktur topologi yang luas (McInnes, 2020).

2.4 Synthetic Minority Over-sampling Technique (SMOTE)

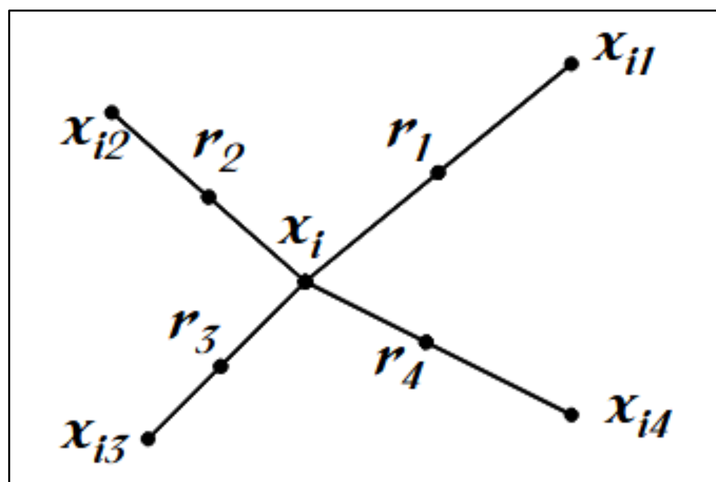
SMOTE adalah teknik *oversampling* yang membuat data sintesis dari kelas minoritas (Blagus dan Lusa, 2013). Sampel yang dihasilkan merupakan interpolasi antara data dari kelas minoritas yang berdekatan. Oleh sebab itu, algoritma ini lebih berfokus pada nilai dan hubungan antar fitur (Fernández, dkk., 2018).

Cara algoritma membuat data baru diawali dengan menentukan jumlah data yang akan dibuat (N), kemudian data dari kelas minoritas dipilih secara *random* dan diambil *k-nearest neighbour*-nya, dengan *default* nilai k adalah 5 (Fernández, dkk., 2018). Kemudian data baru akan dibuat pada garis yang sejajar dengan hubungan antara data dengan *nearest neighbour* data. Letak data baru ini ditentukan secara *random* dengan mengalikan jarak dari data ke *nearest neighbour* dengan nilai antara 0 hingga 1. Perhitungan dilakukan dengan rumus berikut (Douzas dan Bacao, 2017):

$$x_{gen} = x + \propto . (x' - x) \quad \ldots(2.3)$$

dengan x_{gen} adalah data yang dibuat, x adalah data pada kelas minoritas, x' adalah data *nearest neighbour* dari *x,* dan $\propto$ adalah nilai *random* antara 0 hingga 1.

Berikut merupakan ilustrasi terkait, dengan x_i adalah data asli, x_{i1-4} adalah data *nearest neighbour*, sehingga *k* dalam Gambar 2.5 adalah 4, serta *r* adalah data sintesis yang dihasilkan oleh SMOTE (Douzas dan Bacao, 2017).



Gambar 2.5 Pembuatan Data Sintesis SMOTE

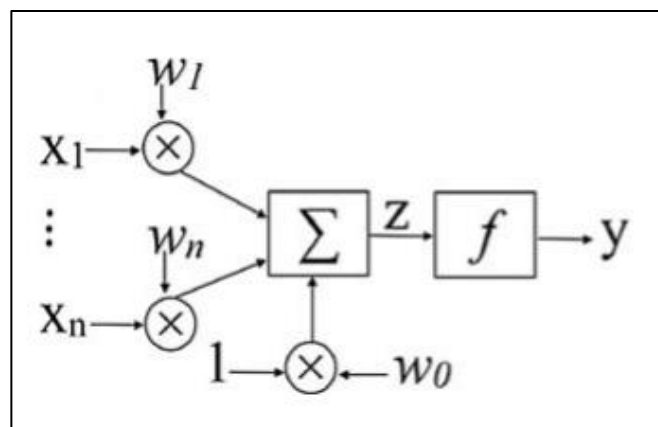
Sumber: (Fernández, dkk., 2018).

Nilai k untuk menentukan *nearest neighbour* harus ditentukan dengan seksama karena memiliki pengaruh yang besar terhadap data sintesis yang dibuat. Semakin besar nilai *k*, maka kemungkinan membuat data sintesis yang *noisy* akan semakin tinggi, namun di sisi lain, nilai *k* yang semakin kecil akan menyebabkan data sintesis hanya berkumpul pada beberapa titik tertentu saja (Douzas dan Bacao, 2017).

2.5 Multilayer Perceptron (MLP)

MLP adalah model *Artificial Neural Network feed forward* (Jotheeswaran dan Koteeswaran, 2015). MLP dapat memprediksi *output* melalui *input* yang diberikan dengan menggunakan fungsi model nonlinear (Taud dan Mas, 2018). Dalam MLP, seluruh *nodes* dari *layer* sebelumnya terhubung dengan seluruh *nodes* dalam *layer* selanjutnya (Witten, dkk., 2016).

MLP adalah perkembangan dari *single layer perceptron.* Sebelum membahas MLP, berikut merupakan ilustrasi dari s*ingle layer perceptron*:



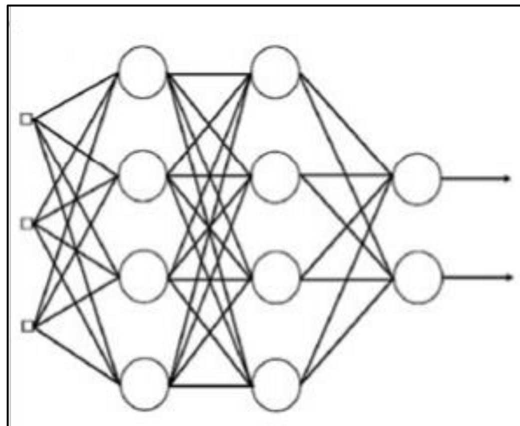
Gambar 2.6 Single Layer Perceptron

Sumber: (Taud dan Mas, 2018)

Dengan x adalah *input*, w adalah *weight*, w_o adalah *bias*, Σ adalah *weighted sum*, f adalah *activation function*, dan y adalah *output* (Taud dan Mas, 2018). *Output* dapat diperoleh melalui rumus:

$$y = f(z) \text{ dan } z = \sum_{i=0}^{n} w_i x_i \quad \text{...(2.4)}$$

MLP minimal terdiri atas 3 *layer* utama, yaitu *input layer* (layer pertama), *output layer* (layer terakhir), dan *hidden layer* (*layer* antara *input* dan *output layer*) (Jotheeswaran dan Koteeswaran, 2015). *Hidden layer* disebut demikian dengan alasan *layer* tersebut tidak memiliki koneksi dengan *environment*, atau dengan kata lain, pengguna tidak melihat *layer* tersebut (Witten, dkk., 2016).



Gambar 2.7 Multilayer Perceptron

Sumber: (Taud dan Mas, 2018)

Setiap *input* dari *neuron* memiliki *weight*-nya masing-masing. Setiap *neuron* yang berada dalam *layer* yang sama memiliki *activation function* yang sama, dengan *default ReLU* jika menggunakan MLPClasssifier dari sklearn (Taud dan Mas, 2018). Pencarian *output* yang optimal sangat dipengaruhi oleh nilai *weight* dari setiap *input*. Optimalisasi *weight* ini dapat dilakukan dengan *backpropagation* (Witten dan Frank, 2002). *Backpropagation* mencari nilai optimal dengan

memperhitungkan *error* dari layer paling akhir ke layer paling awal (Taud dan Mas, 2018). Perhitungan nilai *error* diperoleh menggunakan rumus (Witten, dkk., 2016):

$$E = \frac{1}{2}\,(y - f(x))^2 \qquad \ldots(2.5)$$

dengan *f(x)* adalah *output* yang diperoleh, y adalah *output* yang diinginkan, dan nilai $\frac{1}{2}$ digunakan untuk menyeimbangkan pangkat ketika melakukan perhitungan dengan turunan.

Sementara untuk memperbarui nilai *weight,* diawali dengan mengetahui nilai *error* dari *weight* tersebut menggunakan *chain*, kemudian menghitung *weight* baru dengan rumus (Jotheeswaran dan Koteeswaran, 2015):

$$w_{new} = w_{old} - \lambda\ E_i \qquad \ldots(2.6)$$

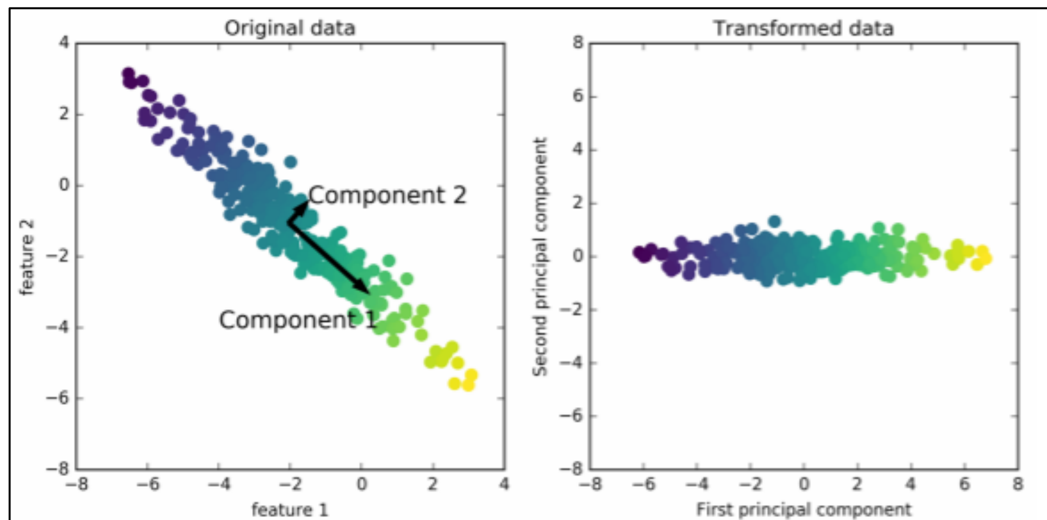
dengan λ adalah *learning rate* dan E_i adalah nilai *error* dari *weight* terkait.

Performa algoritma ini tergantung pada jumlah *hidden layer*, jumlah *nodes*, *learning rate,* jumlah iterasi, dan perubahan *weight* (Taud dan Mas, 2018). Semakin banyak jumlah *hidden layer*, maka kemungkinan *overfitting* akan semakin besar, namun jumlah *hidden layer* yang terlalu sedikit akan mengakibatkan performa model kurang baik (Taud dan Mas, 2018).

2.6 Principal Component Analysis (PCA)

Principal Component Analysis secara umum digunakan untuk mereduksi dimensi dengan tujuan akhir memvisualisasikan data atau melakukan *feature extraction* (Müller dan Guido, 2016). Algoritma ini bekerja dengan membuat sistem koordinat baru sesuai dengan data sampel yang diberikan (Witten, dkk., 2016). Axis pertama adalah garis yang memiliki nilai *variance* maksimum, artinya garis ini mengandung informasi terbanyak dibandingkan dengan arah lainnya. Kemudian

algoritma akan mencari axis selanjutnya yang ortogonal terhadap garis pertama namun tetap memiliki informasi terbanyak, dan seterusnya (Witten, dkk., 2016). Setiap axis yang ditemukan tersebut kemudian disebut sebagai *principal components*. Visualisasi proses ini dapat dilihat pada Gambar 2.8.



Gambar 2.8 Visualisasi Pencarian *Principal Components* dengan PCA

Sumber: (Müller dan Guido, 2016)

Jika dataset direduksi menjadi 3 dimensi, maka diambil 3 *principal components* yang dicari menggunakan cara awal, sehingga hasil reduksi tidak terdiri atas 3 fitur utama saja, namun hasil kombinasi dari banyak fitur dengan informasi tertinggi (Müller dan Guido, 2016). Setiap *principal components* merepresentasikan nilai *variance* dari dataset. Jika jumlah *principal components* sama dengan dimensi asli, maka nilai *explained variance* yang diperoleh adalah 100%, sehingga semakin sedikit *principal components*, maka nilai *explained variance* akan semakin kecil (Witten, dkk., 2016).

Pencarian *principal components* diawali dengan membuat *covariance matrix* (Σ) dari sampel dengan menggunakan rumus sebagai berikut (Žižka, dkk., 2019).

$$Cov(x_i, x_j) = \sum_{k=1}^{n} (x_{ik} - \bar{x}_i)^2 (x_{jk} - \bar{x}_j)^2 \quad \ldots(2.7)$$

$$\Sigma = \begin{bmatrix} Cov(x_1, x_1) & \cdots & Cov(x_1, x_N) \\ \vdots & \ddots & \vdots \\ Cov(x_N, x_1) & \cdots & Cov(x_N, x_N) \end{bmatrix} \quad \ldots(2.8)$$

dengan i dan j adalah urutan data input dan k adalah urutan fitur yang dimiliki data terkait. Tahap selanjutnya adalah menghitung *eigenvalues* (λ) dan *eigenvectors* (v) dari metrik berdasarkan korelasi antar ketiganya (Žižka, dkk., 2019).

$$\sum v_i^T = \lambda_i v_i^T \quad \ldots(2.9)$$

Eigenvalues dicari terlebih dahulu, lalu diurutkan secara *descending* mulai dari nilai tertinggi. Jika ingin melakukan reduksi dataset menjadi 2 dimensi, maka harus mencari 2 nilai tertinggi dari *eigenvalues* dan mencari *eigenvectors* untuk masing-masing *eigenvalues*. *Eigenvectors* pertama merepresentasikan *principal components* pertama, sementara *eigenvectors* kedua merepresentasikan *principal components* kedua.

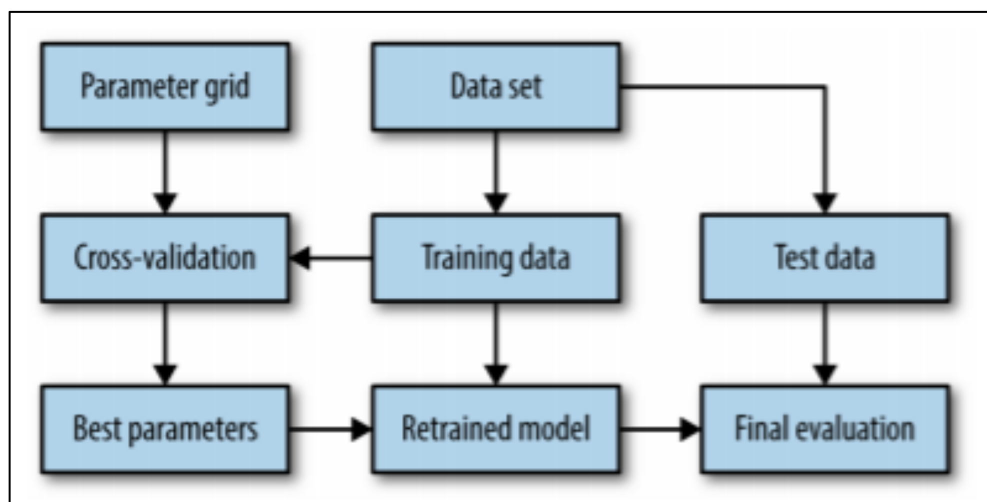
2.7 Grid Search

Grid Search adalah metode yang digunakan untuk mencari parameter yang tepat untuk meningkatkan performa model dengan mencoba seluruh kombinasi *hyperparameter* yang diberikan (Müller dan Guido, 2016). Grid Search juga menyediakan fitur untuk melakukan *cross validation*. Proses ini bertujuan untuk mengevaluasi performa model secara lebih umum karena adanya ketergantungan performa dengan pembagian data. Jumlah *fold* yang ingin dilakukan dapat diberikan pada parameter cv ketika melakukan inisiasi GridSearchCV dari sklearn. Apabila *value* tidak diberikan, maka secara *default* akan menjalankan 5-*fold cross validation*.

Total percobaan yang dilakukan sama dengan hasil kali dari jumlah *fold cross validation* dengan jumlah *value* dari masing-masing parameter model terkait.

Apabila jumlah *fold* adalah 5, dengan banyak *value* untuk 3 parameter adalah 5, 10, dan 3, maka Grid Search akan melakukan percobaan sebanyak 750 kali (Witten, dkk., 2016). Proses yang memakan waktu lama ini dapat dipersingkat dengan menjalankan percobaan secara paralel melalui parameter n_jobs dari GridSearchCV. Nilai n_jobs secara default adalah 1, namun jika ingin memaksimalkan penggunaan prosesor, dapat memberikan nilai -1.

Setelah seluruh percobaan dilakukan, Grid Search akan menentukan *best parameters*, yakni kombinasi parameter yang memiliki rata-rata tertinggi atas *f-measure* seluruh *fold* untuk setiap kombinasi parameter yang diuji (Müller dan Guido, 2016). Alur Grid Search secara umum dapat dilihat pada Gambar 2.9.



Gambar 2.9 Alur proses Grid Search

Sumber: (Müller dan Guido, 2016).

2.8 Metrik Evaluasi

Hasil analisis sentimen akan dievaluasi dan diketahui tingkat *precision*, *recall,* dan *f-measure*-nya menggunakan *confusion matrix* atau *contingency matrix* (Müller dan Guido, 2016). Hasil dari *confusion matrix* berupa *array* dengan ukuran 2x2 yang digambarkan seperti tabel berikut:

Tabel 2.1 Confusion Matrix

Prediction / Actual	Positive	Negative
Positive	True Positive (TP)	False Negative (FN)
Negative	False Positive (FP)	True Negative (TN)

Sumber: (Müller dan Guido, 2016)

TP dan TN merupakan data yang diklasifikasi dengan tepat, sementara FN dan FP merupakan data yang diklasifikasi dengan tidak tepat (Witten, dkk., 2016). Dalam konteks analisa sentimen, nilai FP bertambah apabila terdapat data yang secara aktual bernilai negatif, namun diprediksi sebagai positif, sementara FN merupakan kebalikan dari FP, yaitu data yang secara aktual bernilai positif, namun diprediksi sebagai negatif (Žižka, dkk., 2019).

Berdasarkan keempat nilai tersebut, dapat dihitung nilai *precision, recall,* dan *f-measure. Precision* digunakan untuk membatasi nilai FP jika FP memiliki peran yang penting dalam penelitian ini (Müller dan Guido, 2016). *Precision* menghitung berapa persentase prediksi positif bernilai benar dengan rumus berikut (Müller dan Guido, 2016).

$$Precision = \frac{TP}{TP+FP} \quad \ldots(2.10)$$

Sementara nilai *recall* adalah kebalikan dari *precision*. Nilai ini digunakan untuk membatasi nilai FN dengan menghitung berapa banyak data positif yang diprediksi sebagai nilai positif (Müller dan Guido, 2016).

$$Recall = \frac{TP}{TP+FN} \quad \ldots(2.11)$$

Berdasarkan rumus *precision* dan *recall*, terdapat *trade-off* antara kedua nilai tersebut. Oleh sebab itu, perlu menghitung nilai *f-measure* untuk memperoleh

gambaran umum dari data. *F-measure* atau *f-score* adalah rata-rata harmonik dari *precision* dan *recall*, sehingga memberikan perhitungan yang lebih baik dibandingkan nilai akurasi dalam mengklasifikasikan *imbalance dataset*. *F-measure* dapat diperoleh melalui rumus berikut (Müller dan Guido, 2016).

$$F - measure = 2 \,.\frac{precision \,.recall}{precision + recall} \qquad \ldots(2.12)$$