Modul Pelatnas IOAI Indonesia

Modul 5: Pengenalan Pengolahan Bahasa Alami (NLP)

Tim Pembina IOAI Indonesia
sc.ioai.id@gmail.com

Juli 2025

Modul Pelatnas IOAI Indonesia

Modul 5: Pengenalan Pengolahan Bahasa Alami (NLP)

Penyusun

Tim Pembina IOAI Indonesia

sc.ioai.id@gmail.com

Daftar Isi

Pengantar

Puji syukur kehadirat Tuhan YME atas berkat limpahan rahmat dan karnuia-Nya, Buku Modul Pelatnas IOAI ini telah berhasil kami selesaikan. Buku Modul ini kami susun sebagai salah satu referensi rangkaian pembinaan/pelatihan nasional bagi siswa peserta didik yang mengikuti Pelatnas dalam rangka membentuk tim yang akan mewakili Indonesia pada ajang International Olympiad in Artificial Intelligence (IOAI).

Terima kasih yang sebesar-besarnya kami sampaikan kepada para Pembina, Asisten Pembina, dan para Alumni ajang OSN bidang Informatika dan IOI (TOKI), serta semua pihak yang telah berkontribusi sehingga Buku Modul ini dapat terwujud. Kami menyadari masih banyak kekurangan dalam penulisan Buku Modul ini. Untuk itu kami mohon maaf dan kami sangat mengharapkan masukan untuk perbaikan dan penyempurnaan selanjutnya, sehingga keberadaan Buku Modul ini dapat memberikan manfaat yang sebesar-besarnya bagi semua pihak.

Semoga Buku Modul Pelatnas IOAI ini dapat digunakan sebaik-baiknya untuk menunjang kegiatan Pelatnas IOAI dan mampu membantu menghasilkan calon-calon talenta Indonesia di bidang AI yang mampu memberikan prestasi yang membanggakan di tingkat internasional.

BAB 1

Pengenalan Pengolahan Bahasa Alami (NLP)

1.1 Pendahuluan

Pengolahan Bahasa Alami atau Natural Language Processing (NLP) adalah cabang dari kecerdasan buatan (AI) yang berfokus pada bagaimana komputer dapat memahami, memproses, dan menghasilkan bahasa manusia. Tujuannya adalah membuat mesin mampu "mengerti" teks atau ucapan manusia secara otomatis, baik untuk dianalisis maupun diolah menjadi keluaran yang relevan.

Bahasa alami—seperti bahasa Indonesia atau Inggris—berbeda dari data numerik atau citra karena bersifat tidak terstruktur, ambigu, dan sangat bergantung pada konteks. Oleh karena itu, pemrosesan bahasa alami memerlukan pendekatan yang unik dan kompleks.

Perkembangan NLP telah berlangsung selama beberapa dekade. Pada awalnya, pendekatan yang digunakan bersifat simbolik dan berbasis aturan tata bahasa. Namun, metode tersebut sulit diskalakan untuk menangani keragaman bahasa manusia. Kemudian, NLP mulai menggunakan pendekatan statistik seperti n-gram, lalu bergeser ke pembelajaran mesin klasik (misalnya Naive Bayes dan Support Vector Machine), dan kini berkembang pesat dengan pendekatan deep learning yang memanfaatkan neural network, word embeddings, serta arsitektur modern seperti Transformer.

NLP kini menjadi salah satu pilar penting dalam banyak aplikasi AI modern. Teknologi seperti pencarian Google, asisten virtual (Siri, Alexa), penerjemahan otomatis (Google Translate), hingga sistem rekomendasi dan chatbot di layanan pelanggan, semuanya bergantung pada kemampuan mesin dalam memproses bahasa alami.

Dengan berkembangnya model bahasa besar (large language models) seperti BERT dan GPT, NLP telah mengalami lompatan besar dalam akurasi dan kemampuan. Namun, pemahaman dasar tentang bagaimana teks diolah dan direpresentasikan masih menjadi pondasi penting yang perlu dipahami sebelum menggunakan model-model canggih tersebut.

Dalam kehidupan sehari-hari, kita sering berinteraksi dengan aplikasi yang menggunakan NLP, seperti:

- Asisten virtual (Google Assistant, Siri)
- Chatbot layanan pelanggan
- Aplikasi penerjemah otomatis (Google Translate)

- Sistem klasifikasi ulasan produk (positif/negatif)
- Rekomendasi konten berdasarkan analisis teks

1.2 Tantangan dalam Pemrosesan Bahasa Alami

Bahasa manusia sangat kompleks dan ambigu. Bahkan kalimat sederhana dapat memiliki arti yang berbeda tergantung pada konteksnya. Beberapa tantangan utama dalam NLP antara lain:

- **Ambiguitas leksikal:** Kata yang sama dapat memiliki makna berbeda, misalnya kata *"bisa"* dalam konteks racun atau kemampuan.
- **Ambiguitas sintaktik:** Susunan kata bisa ditafsirkan dalam berbagai cara.
- **Variasi gaya dan struktur:** Bahasa informal di media sosial berbeda dengan bahasa formal di surat kabar.
- **Konteks:** Pemahaman makna sering kali bergantung pada konteks yang lebih luas, baik kalimat sebelumnya maupun dunia nyata.

Untuk mengatasi tantangan-tantangan ini, diperlukan pendekatan berbasis statistik, pembelajaran mesin, dan akhir-akhir ini—pendekatan berbasis deep learning seperti Transformer dan BERT.

1.3 Aplikasi NLP dalam Kehidupan Nyata

NLP telah diterapkan di berbagai bidang, di antaranya:

- **Pencarian informasi:** Mesin pencari seperti Google menggunakan NLP untuk memahami maksud pencarian pengguna dan memberikan hasil yang relevan.
- **Analisis sentimen:** Untuk mengetahui apakah suatu ulasan (review) bernada positif atau negatif.
- **Penerjemahan otomatis:** Mengubah teks dari satu bahasa ke bahasa lain.
- **Chatbot dan asisten virtual:** Memberikan respon otomatis terhadap pertanyaan pengguna.
- **Deteksi spam:** Mengklasifikasikan email sebagai spam atau bukan spam.

Teknologi NLP tidak hanya digunakan oleh perusahaan besar, tetapi juga menjadi bagian penting dalam pengembangan aplikasi edukatif, sosial, dan penelitian.

1.4 Large Language Models (LLM)

Dalam beberapa tahun terakhir, bidang NLP mengalami lompatan besar berkat pengembangan model bahasa berskala besar yang disebut sebagai *Large Language Models* (LLM).

Model ini dilatih menggunakan data teks dalam jumlah sangat besar dan arsitektur deep learning modern, terutama *Transformer*.

Contoh LLM yang populer meliputi:

- **BERT (Bidirectional Encoder Representations from Transformers)** — dikembangkan oleh Google, digunakan untuk memahami konteks dua arah dalam kalimat.
- **GPT (Generative Pretrained Transformer)** — dikembangkan oleh OpenAI, mampu menghasilkan teks panjang yang koheren dan realistis.
- **T5, RoBERTa, XLNet, ChatGPT, dan lain-lain** — varian atau model turunan dengan tujuan atau pendekatan pelatihan yang berbeda.

LLM mampu melakukan banyak tugas NLP seperti:

- Menjawab pertanyaan
- Menerjemahkan bahasa
- Meringkas teks
- Menyelesaikan kalimat
- Berpartisipasi dalam percakapan (chatbot)

Bagaimana LLM bekerja?

Secara umum, LLM belajar mengenali pola bahasa dari data dalam jumlah sangat besar. Mereka dibangun di atas arsitektur *Transformer* dan menggunakan mekanisme *attention* untuk memahami hubungan antar kata dalam konteks panjang.

Berbeda dengan model tradisional yang biasanya dilatih khusus untuk satu tugas (seperti klasifikasi sentimen), LLM dapat digunakan untuk berbagai tugas secara langsung (zero-shot), atau dilatih ulang sebagian (fine-tuning) untuk performa lebih baik.

Kenapa LLM penting?

Large Language Models mengubah paradigma pengembangan sistem NLP. Kini, banyak aplikasi dapat dibangun dengan memanfaatkan model pretrained seperti GPT-3 atau BERT tanpa perlu melatih ulang dari awal. Hal ini mempercepat inovasi dan membuat teknologi NLP lebih mudah diakses oleh banyak kalangan, termasuk pelajar dan pengembang pemula.

Namun, penting juga untuk memahami bahwa LLM bukan tanpa kelemahan. Mereka membutuhkan banyak data dan komputasi untuk dilatih, serta masih bisa menghasilkan keluaran yang tidak akurat atau bias jika tidak digunakan dengan hati-hati.

Modul ini akan memperkenalkan konsep dan praktik NLP dari dasar sebelum nantinya menyentuh penggunaan model pretrained seperti BERT dan GPT di bab-bab akhir.

1.5 Referensi dan Bahan Bacaan Lanjutan

- Jurafsky, D. and Martin, J.H. (2020). *Speech and Language Processing (3rd ed. draft)*. https://web.stanford.edu/~jurafsky/slp3/
- Natural Language Processing with Python (O'Reilly Media) — oleh Steven Bird, Ewan Klein, dan Edward Loper.
  Buku gratis tersedia di https://www.nltk.org/book/
- Course: *CS224N – Natural Language Processing with Deep Learning (Stanford)*: https://web.stanford.edu/class/cs224n/
- Artikel pengantar NLP oleh Huggingface:
  https://huggingface.co/learn/nlp-course/chapter1

BAB 2

Representasi Teks dan Pra-pemrosesan

2.1 Pendahuluan

Dalam NLP, teks mentah harus diubah menjadi representasi yang dapat diproses oleh algoritma komputasi. Proses ini disebut sebagai *pra-pemrosesan teks* (text preprocessing) dan merupakan langkah penting sebelum teks dapat dianalisis atau dimodelkan. Tujuannya adalah untuk menyederhanakan dan membersihkan data teks, sekaligus mengubahnya menjadi format numerik.

Beberapa tahapan umum dalam pra-pemrosesan teks antara lain:

- Tokenisasi
- Normalisasi teks (lowercase, menghapus tanda baca)
- Menghapus stopword
- Stemming atau lemmatization
- Representasi vektor (bag-of-words, TF-IDF, word embeddings)

Langkah-langkah ini membantu model NLP menjadi lebih efisien dan akurat.

Pada bab ini, kita akan membahas berbagai teknik dasar untuk merepresentasikan teks serta tahapan-tahapan penting dalam pra-pemrosesan, termasuk penggunaan *regular expressions*.

2.2 Pengolahan Teks dengan Regular Expressions (Regex)

Regular expressions atau *regex* adalah pola string yang digunakan untuk pencocokan (matching), pencarian, dan manipulasi teks. Regex sangat berguna dalam tahap pra-pemrosesan teks untuk mengekstraksi informasi tertentu atau membersihkan data dari karakter yang tidak diinginkan.

Contoh penggunaan regex:

- Menghapus semua karakter non-alfabet
- Menemukan semua angka dalam teks
- Menghapus tag HTML atau format markup

- Memisahkan kata dari kalimat dengan aturan tertentu

Berikut adalah contoh pola dasar dalam regex:

- `\d` : mencocokkan digit (angka 0-9)
- `\w` : mencocokkan karakter alfanumerik (huruf dan angka)
- `\s` : mencocokkan spasi/whitespace
- `.` : mencocokkan sembarang karakter
- `[...]` : mencocokkan salah satu karakter dalam daftar
- `^` dan `$` : masing-masing untuk awal dan akhir baris

Contoh Implementasi dengan Python

Berikut contoh implementasi sederhana menggunakan pustaka `re` di Python:

Listing 2.1. Penggunaan regex untuk pra-pemrosesan teks

```
import re

teks = "Halo dunia! Hari ini tanggal 24-07-2025, cuaca cerah."

# Menghapus semua angka
teks_bersih = re.sub(r"\d+", "", teks)
print(teks_bersih)

# Mengambil semua angka dari teks
angka = re.findall(r"\d+", teks)
print(angka)

# Menghapus tanda baca
teks_bersih = re.sub(r"[^\w\s]", "", teks)
print(teks_bersih)
```

Regex juga sering digunakan bersamaan dengan tokenisasi, pembersihan tanda baca, penghapusan tag HTML, atau validasi format tertentu (misal email, URL).

Meskipun tampak rumit di awal, regex sangat efisien dan fleksibel dalam mengolah data teks. Oleh karena itu, regex merupakan keterampilan penting dalam NLP praktis.

2.3 Tokenisasi

Tokenisasi adalah proses memecah teks mentah menjadi bagian-bagian kecil yang disebut *token*. Token bisa berupa kata, karakter, atau frasa tergantung pada jenis tokenisasi yang digunakan.

Tujuan tokenisasi adalah untuk mengubah teks yang panjang dan tidak terstruktur menjadi unit-unit kecil yang lebih mudah diproses oleh model NLP. Token biasanya

menjadi unit dasar yang digunakan dalam analisis teks, pembobotan (seperti TF-IDF), maupun input ke dalam model pembelajaran mesin.

Jenis-jenis Tokenisasi

- **Word Tokenization**: Memecah teks menjadi kata per kata.
- **Character Tokenization**: Memecah teks menjadi karakter individu.
- **Subword Tokenization**: Memecah kata menjadi unit yang lebih kecil (contoh: BPE, WordPiece), banyak digunakan dalam LLM seperti BERT dan GPT.
- **Sentence Tokenization**: Memecah teks panjang menjadi kalimat.

Contoh Implementasi Tokenisasi

1. Tokenisasi Sederhana dengan Python Biasa

Untuk kebutuhan awal, tokenisasi sederhana dapat dilakukan menggunakan metode `split()` atau `regex`.

Listing 2.2. Tokenisasi kata menggunakan split()

```
kalimat = "Saya␣suka␣belajar␣NLP!"
tokens = kalimat.split()
print(tokens)
```

Output:

```
['Saya', 'suka', 'belajar', 'NLP!']
```

Namun, pendekatan ini tidak menghapus tanda baca atau menangani struktur kalimat secara kompleks. Untuk itu, kita bisa menggunakan `regex`:

Listing 2.3. Tokenisasi kata dengan regex

```
import re

kalimat = "Halo␣dunia!␣Ini␣tahun␣2025."
tokens = re.findall(r'\b\w+\b', kalimat)
print(tokens)
```

Output:

```
['Halo', 'dunia', 'Ini', 'tahun', '2025']
```

2. Tokenisasi dengan spaCy

`spaCy` merupakan pustaka NLP modern dan efisien yang mendukung tokenisasi serta anotasi linguistik lainnya. Berikut contoh penggunaannya:

Listing 2.4. Tokenisasi dengan spaCy (Bahasa Inggris)

```
import spacy

nlp = spacy.load("en_core_web_sm")
```

```
kalimat = "Natural Language Processing is fun!"
doc = nlp(kalimat)
tokens = [token.text for token in doc]
print(tokens)
```

Output:

```
['Natural', 'Language', 'Processing', 'is', 'fun', '!']
```

Untuk Bahasa Indonesia, belum tersedia model resmi di spaCy, tetapi bisa digabungkan dengan tokenizer dari HuggingFace atau menggunakan tokenisasi berbasis aturan sederhana.

Catatan Praktis

- Tokenisasi yang baik akan membantu model memahami struktur bahasa dengan lebih akurat.
- Pada model pretrained seperti BERT atau GPT, tokenisasi subword lebih umum digunakan dan disediakan oleh tokenizer khusus (dibahas di bab selanjutnya).
- Tokenisasi dapat dipengaruhi oleh tanda baca, huruf kapital, dan struktur gramatikal.

2.4 Normalisasi Teks

Setelah teks dipecah menjadi token, langkah berikutnya adalah melakukan **normalisasi**. Tujuannya adalah menyederhanakan atau menyeragamkan bentuk kata sehingga dapat dianalisis secara lebih konsisten. Normalisasi sangat penting karena dalam bahasa alami, satu kata bisa ditulis dalam banyak variasi bentuk.

Langkah-Langkah Umum Normalisasi

1. **Lowercasing** (mengubah semua huruf menjadi kecil)
2. **Menghapus tanda baca dan angka (opsional)**
3. **Menghapus *stopwords*** (kata-kata umum seperti *the, and, is*)
4. **Stemming atau Lemmatization** (mengubah kata ke bentuk dasarnya)

Contoh Implementasi Normalisasi Teks

Normalisasi Dasar Menggunakan Python

Langkah awal adalah menghapus tanda baca dan huruf kapital:

Listing 2.5. Normalisasi sederhana menggunakan Python

```
import re

text = "The students are studying NLP techniques in 2025!"
# Lowercasing
```

```
text = text.lower()

# Remove punctuation and numbers
text = re.sub(r'[^a-z\s]', '', text)

print(text)
```

Output:

```
the students are studying nlp techniques in
```

Menghapus Stopwords dan Lemmatization dengan spaCy

Pustaka `spaCy` mendukung pemrosesan bahasa alami tingkat lanjut, termasuk lemmatization dan penghapusan stopwords:

Listing 2.6. Normalisasi dengan spaCy

```
import spacy

nlp = spacy.load("en_core_web_sm")
text = "The students are studying NLP techniques in 2025."
doc = nlp(text)

filtered_tokens = [token.lemma_ for token in doc
                   if not token.is_stop and token.is_alpha]

print(filtered_tokens)
```

Output:

```
['student', 'study', 'NLP', 'technique']
```

Penjelasan:

- `token.is_stop` mengecek apakah kata termasuk stopword
- `token.is_alpha` mengecek apakah token hanya huruf
- `token.lemma_` mengembalikan bentuk dasar dari kata

Catatan Praktis

- Lemmatization biasanya lebih akurat dari stemming karena mempertimbangkan konteks gramatikal kata.
- Proses normalisasi meningkatkan konsistensi data dan membantu mengurangi dimensi representasi teks.
- Tidak semua model butuh normalisasi eksplisit — model seperti BERT sudah menangani bentuk kata yang bervariasi secara internal.

2.5 Representasi Teks: Bag-of-Words dan TF-IDF

Setelah teks dibersihkan dan dinormalisasi, kita perlu merepresentasikan teks tersebut dalam bentuk yang bisa diproses oleh komputer, khususnya oleh model machine learning. Dua metode representasi yang paling umum dan sederhana adalah **Bag-of-Words (BoW)** dan **TF-IDF**.

Bag-of-Words (BoW)

Bag-of-Words merepresentasikan dokumen sebagai kumpulan kata unik (vocabulary) tanpa mempertimbangkan urutan kata. Setiap dokumen direpresentasikan sebagai vektor frekuensi kata dalam vocabulary.

Sebagai contoh, misalkan kita punya dua dokumen:

- Dokumen 1: "I love machine learning"
- Dokumen 2: "I love deep learning"

Vocabulary (gabungan semua kata unik): [I, love, machine, learning, deep]
Representasi vektor:

Dokumen	I	love	machine	learning	deep
1	1	1	1	1	0
2	1	1	0	1	1

Implementasi BoW (Python)

Listing 2.7. Representasi Bag-of-Words dengan scikit-learn

```
from sklearn.feature_extraction.text import CountVectorizer

docs = ["I love machine learning", "I love deep learning"]
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(docs)

print(vectorizer.get_feature_names_out())
print(X.toarray())
```

Output:

```
['deep' 'learning' 'love' 'machine']
[[0 1 1 1]
 [1 1 1 0]]
```

TF-IDF (Term Frequency-Inverse Document Frequency)

TF-IDF memperbaiki kekurangan BoW dengan tidak hanya menghitung frekuensi kata, tetapi juga mempertimbangkan seberapa **penting** suatu kata terhadap seluruh kumpulan dokumen.

Term Frequency (TF): seberapa sering sebuah kata muncul dalam sebuah dokumen.

$$TF\,(t,d) = \frac{\text{jumlah kemunculan kata } t \text{ dalam dokumen } d}{\text{jumlah total kata dalam } d}$$

Inverse Document Frequency (IDF): mengukur seberapa langka kata tersebut di seluruh dokumen.

$$IDF\,(t) = \log \frac{N}{df\,(t)}$$

TF-IDF adalah hasil kali dari keduanya:

$$\text{TF-IDF}\,(t,d) = TF\,(t,d) \times IDF\,(t)$$

Implementasi TF-IDF (Python)

Listing 2.8. Representasi TF-IDF dengan scikit-learn

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(docs)

print(vectorizer.get_feature_names_out())
print(X.toarray())
```

Catatan:

- TF-IDF cenderung memberi bobot lebih rendah ke kata-kata umum seperti "the", "and", dan lebih tinggi ke kata spesifik seperti "machine", "deep".
- Representasi ini cocok untuk digunakan dalam model klasifikasi teks maupun clustering.

BAB 3

Klasifikasi Teks Sederhana: Analisis Sentimen

Pendahuluan

Salah satu aplikasi paling umum dari Natural Language Processing (NLP) adalah **analisis sentimen**, yaitu mengklasifikasikan opini atau pernyataan dalam teks sebagai positif, negatif, atau netral. Contoh klasik adalah mengklasifikasikan ulasan produk atau film berdasarkan opini pengguna.

Untuk tugas klasifikasi seperti ini, kita dapat memanfaatkan model-model machine learning klasik seperti Naive Bayes dan Regresi Logistik, dengan representasi teks sederhana seperti Bag-of-Words atau TF-IDF.

3.1 Dataset dan Representasi

Dalam contoh ini, kita akan menggunakan dataset ulasan film IMDb yang sederhana, dengan dua kelas: positif dan negatif. Teks akan direpresentasikan menggunakan `TfidfVectorizer`.

Listing 3.1. Contoh Dataset Ulasan Sederhana

```
texts = [
    "I love this movie, it's fantastic!",
    "This film was terrible and boring.",
    "Amazing plot and characters!",
    "I hated the ending of this movie.",
    "What a great experience watching this!",
    "Not enjoyable at all."
]
labels = [1, 0, 1, 0, 1, 0]  # 1 = positif, 0 = negatif
```

Listing 3.2. TF-IDF Vectorization

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(texts)
```

3.2 Klasifikasi dengan Naive Bayes

Naive Bayes adalah model probabilistik yang sering digunakan untuk klasifikasi teks karena kesederhanaan dan performanya yang baik dalam kasus high-dimensional seperti teks.

Listing 3.3. Training Naive Bayes

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0

model_nb = MultinomialNB()
model_nb.fit(X_train, y_train)

y_pred = model_nb.predict(X_test)
print("Akurasi:", accuracy_score(y_test, y_pred))
```

3.3 Klasifikasi dengan Regresi Logistik

Selain Naive Bayes, regresi logistik juga umum digunakan untuk klasifikasi teks biner:

Listing 3.4. Training Logistic Regression

```
from sklearn.linear_model import LogisticRegression

model_lr = LogisticRegression()
model_lr.fit(X_train, y_train)

y_pred = model_lr.predict(X_test)
print("Akurasi:", accuracy_score(y_test, y_pred))
```

3.4 Diskusi dan Perbandingan

- **Naive Bayes** sangat cepat dan cocok untuk data teks dengan banyak fitur (kata).
- **Regresi Logistik** lebih fleksibel dalam menangani korelasi antar fitur.
- Dalam praktiknya, performa kedua model sangat tergantung pada preprocessing teks dan representasi fitur.

3.4.1 Studi Kasus: Analisis Sentimen dengan Data IMDb

Dalam bagian ini, kita akan mempelajari bagaimana melakukan analisis sentimen sederhana dari data ulasan film menggunakan teknik klasifikasi teks klasik. Dataset yang

digunakan adalah **IMDb Movie Review Dataset**, yang berisi ulasan-ulasan film dalam bahasa Inggris dan label sentimen (positif/negatif). Dataset ini tersedia di:
https://www.kaggle.com/datasets/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews

Untuk latihan, Anda dapat menggunakan subset kecil dari dataset ini (misalnya 2.000 baris saja). Berikut ini adalah langkah-langkah umum yang dapat dilakukan:

1. **Membaca dan Mengeksplorasi Data**
2. **Membersihkan Teks** (lowercase, hilangkan tanda baca, stopwords)
3. **Tokenisasi dan Vektorisasi** (menggunakan TF-IDF)
4. **Membagi data** menjadi data latih dan data uji
5. **Melatih model klasifikasi**, seperti Naive Bayes atau Logistic Regression
6. **Menguji dan mengevaluasi model** menggunakan akurasi atau confusion matrix

Berikut ini adalah contoh kode Python sederhana untuk melakukan klasifikasi:

Listing 3.5. Analisis Sentimen IMDb (klasik)

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report,
    confusion_matrix

# Load data (pastikan hanya subset kecil)
df = pd.read_csv("IMDB␣Dataset.csv")
df = df.sample(n=2000, random_state=42)
# Ambil subset 2000 baris

# Preprocessing
df['review'] = df['review'].str.lower()

# Label encoding
df['label'] = df['sentiment'].map({'positive': 1, 'negative': 0})

# TF-IDF vektorisasi
tfidf = TfidfVectorizer(max_features=5000)
X = tfidf.fit_transform(df['review']).toarray()
y = df['label']

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ra

# Latih model
model = LogisticRegression()
```

```
model.fit(X_train, y_train)

# Evaluasi
y_pred = model.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

Catatan:

- Pastikan telah menginstall `scikit-learn` dan `pandas`
- Untuk praproses teks lebih lanjut, siswa dapat mempelajari pustaka seperti `re` untuk regex atau `spaCy` untuk tokenisasi lanjutan

Referensi dan Bahan Bacaan Lanjutan

- https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html
- https://www.kaggle.com/datasets/lakshmi25npathi/imdb-dataset-of-50k-movie-review
- https://realpython.com/python-keras-text-classification/

Referensi dan Bahan Bacaan Lanjutan

- Scikit-learn Text Classification Guide: https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html
- "Text Mining with R" by Julia Silge and David Robinson (online book): https://www.tidytextmining.com/
- Dataset IMDb versi sederhana: https://www.kaggle.com/datasets/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews

BAB 4
Word Embedding

4.1 Representasi Kata dengan Word Embedding

4.1.1 Mengapa Representasi Kata dalam Bentuk Vektor Dibutuhkan?

Dalam pemrosesan bahasa alami (Natural Language Processing), model machine learning tidak dapat memahami kata-kata secara langsung. Kata-kata dalam bentuk teks harus direpresentasikan dalam bentuk numerik agar bisa diproses oleh algoritma.

Salah satu cara paling sederhana untuk merepresentasikan kata adalah dengan menggunakan **one-hot encoding**. Pada pendekatan ini, setiap kata dalam kosakata diwakili oleh vektor yang memiliki panjang sebesar jumlah total kata, dan hanya satu elemen yang bernilai 1 (posisi kata tersebut), sementara sisanya 0. Contoh:

- `["kucing", "anjing", "burung"]` ⇒
  - "kucing": [1, 0, 0]
  - "anjing": [0, 1, 0]
  - "burung": [0, 0, 1]

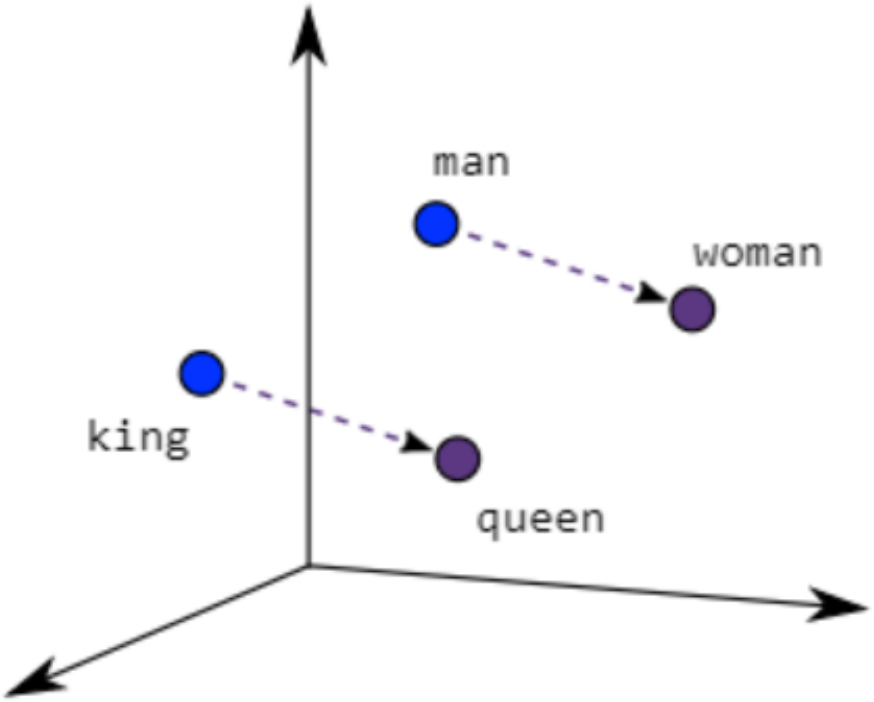
Namun, one-hot encoding memiliki beberapa kekurangan:

- Ukuran vektor sangat besar (sparse) jika kosakata banyak.
- Tidak menangkap hubungan atau kedekatan semantik antar kata.
- Semua kata dianggap berjarak sama (tidak ada struktur makna).

Untuk mengatasi masalah tersebut, dikembangkan teknik representasi kata dalam bentuk **dense vector**, yang kita kenal sebagai *word embedding*. Embedding memungkinkan kita untuk memetakan kata-kata ke dalam ruang vektor berdimensi rendah (misal 50, 100, atau 300) di mana kata-kata yang memiliki makna serupa akan memiliki vektor yang berdekatan. Gambar 4.1 mengilustrasikan hasil word embedding.

4.1.2 Word2Vec

Word2Vec adalah salah satu algoritma populer untuk menghasilkan word embedding, dikembangkan oleh tim Google pada tahun 2013. Inti dari Word2Vec adalah melatih



Gambar 4.1. Ilustrasi hasil word embedding

jaringan saraf kecil untuk mempelajari vektor representasi kata berdasarkan konteksnya dalam kalimat.

Word2Vec memiliki dua arsitektur utama:

1. **Continuous Bag of Words (CBOW):** memprediksi kata berdasarkan konteks sekitarnya.
2. **Skip-Gram:** memprediksi konteks berdasarkan kata pusat.

Contoh Skip-Gram: Kalimat: "Saya makan nasi goreng enak sekali" Kata target: "nasi", kontekstual: ["makan", "goreng"] Model akan belajar bahwa "nasi" sering muncul dekat dengan "makan" dan "goreng".

CBOW vs Skip-Gram:

- CBOW cenderung lebih cepat dan bekerja baik untuk dataset besar.
- Skip-Gram lebih baik untuk menangkap kata-kata yang jarang (rare words).

Contoh Implementasi Word2Vec dengan Gensim:

Listing 4.1. Pelatihan Word2Vec dengan Gensim

```
from gensim.models import Word2Vec

# Dataset kecil untuk contoh
sentences = [["saya", "suka", "makan", "nasi"],
             ["makan", "nasi", "goreng", "enak"],
             ["burung", "terbang", "tinggi"],
             ["kucing", "berlari", "cepat"]]

# Latih model Skip-Gram
model = Word2Vec(sentences, vector_size=50, window=2, sg=1, min_count=1)

# Ambil vektor dari kata
print(model.wv['makan'])

# Cek kata terdekat
print(model.wv.most_similar('makan'))
```

Keterangan parameter:

- `vector_size`: jumlah dimensi vektor (embedding size)
- `window`: jumlah kata konteks di sekitar target
- `sg=1`: gunakan skip-gram (jika 0 maka CBOW)
- `min_count`: frekuensi minimum kata agar disertakan dalam model

Setelah pelatihan, setiap kata dalam kosakata akan memiliki representasi vektor berdimensi tetap. Vektor-vektor ini dapat digunakan dalam berbagai tugas NLP seperti klasifikasi teks, analisis sentimen, dan lain-lain.

4.1.3 GloVe (Global Vectors for Word Representation)

GloVe merupakan metode representasi kata yang dikembangkan oleh tim Stanford. Berbeda dari Word2Vec yang bersifat prediktif (berbasis konteks lokal), GloVe menggunakan pendekatan **statistik global** dengan menghitung frekuensi ko-occurence antar kata di seluruh korpus.

Inti dari GloVe adalah membangun sebuah matriks co-occurrence X berukuran $V \times V$ (dengan V adalah ukuran kosakata), di mana X_{ij} menunjukkan seberapa sering kata j muncul dalam konteks kata i. Kemudian, dilakukan faktorisasi logaritmik dari matriks ini untuk mencari vektor representasi dari setiap kata.

Secara sederhana, GloVe meminimalkan fungsi kesalahan sebagai berikut:

$$J = \sum_{i,j=1}^{V} f\left(X_{ij}\right)\left(w_i^\top \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2$$

Di mana:

- w_i dan $\tilde{w}_j$ adalah vektor representasi kata dan konteksnya
- b_i dan $\tilde{b}_j$ adalah bias
- f adalah fungsi penimbang (weighting function)

Keunggulan GloVe:

- Lebih stabil dibanding Word2Vec pada dataset besar
- Mampu menangkap informasi semantik global antar kata

Menggunakan Pretrained GloVe Embedding

Kita bisa langsung menggunakan pretrained GloVe (contoh: 50D, 100D, 300D) yang tersedia secara publik:

Listing 4.2. Memuat GloVe Pretrained Embedding

```
import numpy as np

# Load pretrained GloVe (misal: glove.6B.100d.txt)
embedding_dict = {}
with open("glove.6B.100d.txt", 'r', encoding="utf-8") as f:
    for line in f:
        values = line.split()
        word = values[0]
        vector = np.asarray(values[1:], dtype='float32')
        embedding_dict[word] = vector

# Lihat vektor dari kata tertentu
print(embedding_dict["king"])
```

4.1.4 Visualisasi dan Aplikasi Word Embedding

Word embedding dapat divisualisasikan dengan mereduksi dimensi vektor (biasanya dari 100-300 dimensi) ke dalam 2 dimensi menggunakan teknik seperti PCA atau t-SNE.

Listing 4.3. Visualisasi Word Embedding dengan t-SNE

```
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

# Pilih kata-kata untuk divisualisasikan
words = ["king", "queen", "man", "woman", "apple", "banana", "fruit", "cat"
vectors = [embedding_dict[word] for word in words]

# Gunakan t-SNE untuk reduksi dimensi
tsne = TSNE(n_components=2, random_state=0)
X_tsne = tsne.fit_transform(vectors)

# Visualisasikan
plt.figure(figsize=(8, 6))
for i, word in enumerate(words):
    plt.scatter(X_tsne[i, 0], X_tsne[i, 1])
    plt.annotate(word, (X_tsne[i, 0], X_tsne[i, 1]))
plt.title("Visualisasi Word Embedding dengan t-SNE")
plt.show()
```

Aplikasi Word Embedding:

- Sebagai input layer untuk model neural network (RNN, LSTM, Transformer)
- Analogi semantik: `king - man + woman = queen`
- Mengukur kesamaan kata dengan cosine similarity

Contoh menghitung kemiripan kata:

Listing 4.4. Cosine Similarity Word Embedding

```
from numpy import dot
from numpy.linalg import norm

def cosine_similarity(v1, v2):
    return dot(v1, v2) / (norm(v1) * norm(v2))

print(cosine_similarity(embedding_dict["king"], embedding_dict["queen"]))
```

4.1.5 Pre-trained Embedding dan Integrasi dengan PyTorch

Word embedding yang telah dilatih sebelumnya (pre-trained embedding) seperti Word2Vec dan GloVe dapat digunakan langsung sebagai input untuk model berbasis deep learning, seperti RNN, LSTM, dan Transformer.

Dalam PyTorch, kita dapat menggunakan kelas `nn.Embedding` untuk menyimpan dan mengakses embedding kata. Untuk memanfaatkan embedding yang telah dilatih sebelumnya, kita dapat memuat vektor ke dalam layer `nn.Embedding` secara manual.

Berikut adalah contoh langkah-langkah memuat embedding GloVe ke dalam model PyTorch:

Listing 4.5. Menggunakan Pre-trained Embedding dalam PyTorch

```
import torch
import torch.nn as nn

# Misalnya vocab dan embedding_dict sudah dibuat dari sebelumnya
vocab = ["king", "queen", "man", "woman", "apple"]
embedding_dim = 100

# Buat matrix embedding sesuai urutan vocab
embedding_matrix = torch.zeros((len(vocab), embedding_dim))
for i, word in enumerate(vocab):
    embedding_matrix[i] = torch.tensor(embedding_dict[word])

# Buat layer embedding dan masukkan pre-trained weights
embedding_layer = nn.Embedding.from_pretrained(embedding_matrix)

# Gunakan untuk lookup kata ke indeks (misalnya "king" di index 0)
index = torch.tensor([0])
print(embedding_layer(index))  # hasil: tensor ukuran [1, 100]
```

Catatan:

- `from_pretrained(...)` membuat layer yang sudah berisi bobot embedding dari awal.
- Jika kita ingin layer embedding ini tidak dilatih ulang selama training, tambahkan parameter `freeze=True`.
- Jika kosakata terlalu besar, kita hanya perlu memuat kata-kata yang paling sering muncul saja.

Dengan teknik ini, kita bisa menghemat waktu pelatihan dan memperoleh performa yang lebih baik, terutama jika dataset pelatihan kecil.

BAB 5

Penerapan RNN dan LSTM dalam NLP

5.1 Pendahuluan dan Kaitan dengan NLP

Recurrent Neural Network (RNN) dan Long Short-Term Memory (LSTM) merupakan jenis arsitektur jaringan saraf tiruan yang dirancang untuk menangani data sekuensial. Dalam konteks Natural Language Processing (NLP), kedua model ini sangat berguna karena bahasa merupakan data sekuensial: urutan kata mempengaruhi makna.

Contoh aplikasi RNN dan LSTM dalam NLP meliputi:

- Analisis sentimen (memprediksi apakah sebuah ulasan bernada positif atau negatif)
- Pengenalan entitas (Named Entity Recognition)
- Penerjemahan mesin
- Pembuatan teks otomatis

Meskipun model seperti Transformer telah menjadi standar saat ini, RNN dan LSTM tetap relevan sebagai dasar yang kuat untuk memahami bagaimana model memproses sekuens teks.

5.2 Analisis Sentimen dengan LSTM

Pada bagian ini, kita akan membuat model klasifikasi sentimen sederhana menggunakan arsitektur LSTM untuk memproses ulasan film dari IMDb. Dataset IMDb berisi sekitar 50.000 ulasan film yang diberi label positif atau negatif.

5.2.1 Langkah-langkah Analisis Sentimen

1. **Pra-pemrosesan data:**

    - Tokenisasi dan padding
    - Membuat indeks kata
    - Membagi data menjadi train/test

2. **Membangun model LSTM:**
    - Embedding layer
    - LSTM layer
    - Fully connected (linear) + sigmoid
3. **Melatih dan menguji model**

5.2.2 Contoh Kode PyTorch

Listing 5.1. Model LSTM untuk Sentimen Analisis IMDb

```
import torch
import torch.nn as nn
from torchtext.legacy import data, datasets

# Setup
TEXT = data.Field(tokenize='spacy', tokenizer_language='en_core_web_sm', in
LABEL = data.LabelField(dtype=torch.float)
train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)

# Build vocab
TEXT.build_vocab(train_data, max_size=10000)
LABEL.build_vocab(train_data)

# DataLoader
train_iter, test_iter = data.BucketIterator.splits((train_data,
test_data), batch_size=64, sort_within_batch=True, device='cuda')

# Define model
class SentimentLSTM(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.lstm = nn.LSTM(embed_dim, hidden_dim)
        self.fc = nn.Linear(hidden_dim, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, text, text_lengths):
        embedded = self.embedding(text)
        packed = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths)
        _, (hidden, _) = self.lstm(packed)
        return self.sigmoid(self.fc(hidden[-1]))
```

```
# Instantiate and train
model = SentimentLSTM(len(TEXT.vocab), 100, 128).to('cuda')
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters())
```

Kode ini hanya menampilkan struktur awal. Untuk pelatihan penuh, kita perlu menambahkan loop `train()` dan `evaluate()` dengan epoch, akurasi, dan perhitungan loss.

5.2.3 Catatan Penting

- Perhatikan pemrosesan urutan panjang — LSTM sangat sensitif terhadap panjang sekuens.
- Pastikan padding dilakukan dengan benar dan panjang sekuens digunakan saat packing.
- Kita bisa menggunakan pretrained word embeddings seperti GloVe untuk hasil lebih baik.

5.3 Text Generation menggunakan RNN

Salah satu aplikasi menarik dari model RNN adalah pembuatan teks otomatis (text generation), di mana model dilatih untuk mempelajari pola dari urutan teks dan menghasilkan teks baru yang menyerupai data pelatihan.

5.3.1 Prinsip Dasar Text Generation

Model RNN untuk text generation dilatih untuk memprediksi karakter atau kata berikutnya berdasarkan konteks sebelumnya. Misalnya, jika diberikan awalan "Once upon a", model akan memprediksi karakter atau kata berikutnya untuk melanjutkan kalimat tersebut.

Dua pendekatan umum:

- **Character-level generation:** memprediksi karakter demi karakter.
- **Word-level generation:** memprediksi kata demi kata.

5.3.2 Langkah Umum

1. Ambil sebuah teks korpus dan lakukan tokenisasi.
2. Bentuk pasangan input-output untuk pelatihan (misal: input = "I am go", target = "am goi").
3. Bangun model RNN atau LSTM.
4. Lakukan pelatihan untuk meminimalkan kesalahan prediksi karakter/word berikutnya.
5. Gunakan model untuk menghasilkan teks secara autoregresif.

5.3.3 Contoh Implementasi PyTorch (Karakter-level)

Listing 5.2. Text Generation RNN dengan karakter

```
import torch, torch.nn as nn
import numpy as np

text = "hello world, this is a demo for text generation"
chars = sorted(set(text))
char2idx = {c: i for i, c in enumerate(chars)}
idx2char = {i: c for c, i in char2idx.items()}

# Encode text
encoded = [char2idx[c] for c in text]

# Hyperparameter
seq_length = 10
X = []
Y = []
for i in range(len(encoded) - seq_length):
    X.append(encoded[i:i+seq_length])
    Y.append(encoded[i+1:i+seq_length+1])

X = torch.tensor(X)
Y = torch.tensor(Y)

# Model
class CharRNN(nn.Module):
    def __init__(self, vocab_size, hidden_size):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, hidden_size)
        self.rnn = nn.RNN(hidden_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, hidden=None):
        x = self.embed(x)
        out, hidden = self.rnn(x, hidden)
        out = self.fc(out)
        return out, hidden

model = CharRNN(len(chars), 64)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# Training (1 epoch contoh)
for i in range(100):
    out, _ = model(X)
    loss = loss_fn(out.view(-1, len(chars)), Y.view(-1))
    optimizer.zero_grad()
```

```
loss.backward()
optimizer.step()
```

5.3.4 Hasil dan Eksperimen

Setelah pelatihan, kita bisa melakukan sampling dengan memulai dari karakter acuan (misalnya "t"), dan mengulang langkah memprediksi karakter berikutnya untuk membentuk teks baru.

Model ini bersifat sederhana, namun sudah mampu menunjukkan bagaimana teks dapat dihasilkan oleh mesin. Untuk hasil lebih realistis, LSTM dan korpus yang lebih besar (misalnya teks Shakespeare atau berita) bisa digunakan.

5.3.5 Catatan Tambahan

- Untuk word-level generation, kita bisa mengganti tokenisasi dari karakter menjadi kata.
- Temperature sampling dan beam search dapat digunakan untuk meningkatkan kualitas teks.

5.4 Model Many-to-One dan Many-to-Many

Dalam pemodelan sekuensial menggunakan RNN atau LSTM, struktur input-output dari data menentukan arsitektur yang digunakan. Dua arsitektur yang paling umum adalah **many-to-one** dan **many-to-many**.

5.4.1 Many-to-One

Model many-to-one adalah arsitektur di mana masukan berupa rangkaian token (kata atau karakter), namun hanya menghasilkan satu output. Ini umum digunakan dalam:

- **Klasifikasi teks:** seperti analisis sentimen.
- **Prediksi label tunggal:** seperti deteksi emosi dalam kalimat.

Contoh:

$$\texttt{[I am very happy today]} \rightarrow \texttt{Positive}$$

Pada implementasi, RNN/LSTM akan membaca seluruh input sequence, dan hidden state terakhir digunakan sebagai representasi untuk menghasilkan output.

5.4.2 Many-to-Many

Dalam model many-to-many, baik input maupun output berupa urutan. Ini digunakan dalam:

- **Named Entity Recognition (NER)**
- **Part-of-Speech (POS) tagging**

- **Machine Translation**

Contoh:

`[Saya tinggal di Jakarta]` → `[PRON VERB ADP PROPN]`

Pada implementasi, model memberikan prediksi untuk setiap elemen input, seringkali dengan satu layer `nn.Linear` yang mengubah hidden state menjadi output label/token.

5.4.3 Variasi Arsitektur Many-to-Many

- **Input dan output panjangnya sama:** Seperti dalam tagging.
- **Input dan output panjangnya berbeda:** Diperlukan mekanisme tambahan seperti *encoder-decoder* (misalnya dalam translation).

5.4.4 Contoh Kode PyTorch (Many-to-One)

Listing 5.3. Contoh struktur Many-to-One dengan PyTorch

```
class SentimentLSTM(nn.Module):
    def __init__(self, vocab_size, hidden_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, hidden_dim)
        self.lstm = nn.LSTM(hidden_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, 1)

    def forward(self, x):
        x = self.embedding(x)
        _, (hn, _) = self.lstm(x)      # ambil hidden state terakhir
        out = torch.sigmoid(self.fc(hn.squeeze(0)))
        return out
```

5.4.5 Catatan Tambahan

Pemahaman struktur input-output sangat penting dalam membangun model NLP yang sesuai dengan tugas yang ingin diselesaikan. Beberapa model modern seperti encoder-decoder Transformer juga mengikuti prinsip dasar ini.

5.5 Fine-tuning dan Pre-trained LSTM Models

Dalam banyak kasus, melatih model RNN atau LSTM dari awal memerlukan data dan waktu komputasi yang besar. Oleh karena itu, pendekatan yang umum digunakan adalah **memanfaatkan model yang telah dilatih sebelumnya (pre-trained)** dan melakukan **fine-tuning** agar model dapat menyesuaikan diri dengan data dan tugas baru.

5.5.1 Apa itu Pre-trained Model?

Pre-trained model adalah model neural network yang telah dilatih pada dataset besar dan umum, misalnya korpus Wikipedia, berita, atau koleksi teks lainnya. Model ini telah mempelajari representasi umum dari bahasa dan dapat digunakan kembali untuk berbagai tugas.

5.5.2 Fine-tuning

Fine-tuning adalah proses menyempurnakan bobot pre-trained model pada dataset spesifik. Dalam konteks LSTM untuk NLP, langkah-langkahnya biasanya sebagai berikut:

1. Ambil model LSTM yang telah dilatih sebelumnya.
2. Ganti atau tambahkan layer output akhir untuk disesuaikan dengan jumlah kelas pada tugas baru.
3. Lakukan pelatihan kembali dengan laju belajar yang lebih kecil.

Catatan: Fine-tuning berbeda dengan *feature extraction*, karena bobot dari model lama ikut diperbarui (bukan hanya digunakan sebagai fitur tetap).

5.5.3 Contoh Implementasi Fine-tuning

Sebagai ilustrasi, kita akan menggunakan model LSTM pre-trained untuk bahasa Inggris (misalnya dari repositori seperti `torchtext`), dan menyesuaikannya untuk tugas analisis sentimen.

Listing 5.4. Struktur Fine-tuning LSTM

```
class FineTuneLSTM(nn.Module):
    def __init__(self, pretrained_lstm, hidden_dim):
        super().__init__()
        self.embedding = pretrained_lstm.embedding
        self.lstm = pretrained_lstm.lstm
        self.fc = nn.Linear(hidden_dim, 1)  # disesuaikan dengan tugas baru

    def forward(self, x):
        x = self.embedding(x)
        _, (hn, _) = self.lstm(x)
        return torch.sigmoid(self.fc(hn.squeeze(0)))
```

5.5.4 Sumber Pre-trained LSTM

Beberapa sumber yang menyediakan pre-trained LSTM antara lain:

- TorchText – menyediakan embedding + LSTM siap pakai.
- GloVe – embedding pre-trained untuk kata, bisa digunakan bersama LSTM.
- Model-model open-source di Hugging Face (meskipun sebagian besar berbasis Transformer, ada juga versi RNN lama).

5.5.5 Kapan Fine-tuning Berguna?

- ◇ Saat dataset pelatihan sangat kecil.
- ◇ Saat ingin memanfaatkan pengetahuan linguistik umum dari model besar.
- ◇ Saat ingin menghemat waktu pelatihan.

5.5.6 Catatan Akhir

Dengan teknik fine-tuning, siswa dapat memanfaatkan kekuatan model besar tanpa harus melatih dari awal. Ini menjadi jembatan penting menuju penggunaan arsitektur modern seperti Transformer.

BAB 6

Transformer dan Arsitektur Modern dalam NLP

Transformer merupakan tonggak penting dalam perkembangan pemrosesan bahasa alami (NLP). Diperkenalkan oleh Vaswani et al. pada tahun 2017 dalam paper berjudul "Attention is All You Need", arsitektur ini menggantikan struktur berurutan (seperti RNN/LSTM) dengan pendekatan berbasis **self-attention**, sehingga memungkinkan pelatihan paralel yang lebih efisien dan performa yang lebih unggul.

6.1 Pendahuluan

Sebelum Transformer diperkenalkan, model RNN dan LSTM mendominasi bidang NLP. Namun, model-model tersebut memiliki keterbatasan:

- Sulit menangkap dependensi jangka panjang dalam teks.
- Latihan dilakukan secara sekuensial (tidak paralel).
- Waktu pelatihan lambat pada dataset besar.

Transformer mengatasi masalah ini dengan pendekatan *self-attention* yang dapat menghitung hubungan antara kata-kata secara langsung tanpa harus memproses secara berurutan.

6.2 Arsitektur Dasar Transformer

Arsitektur Transformer terdiri dari dua komponen utama:

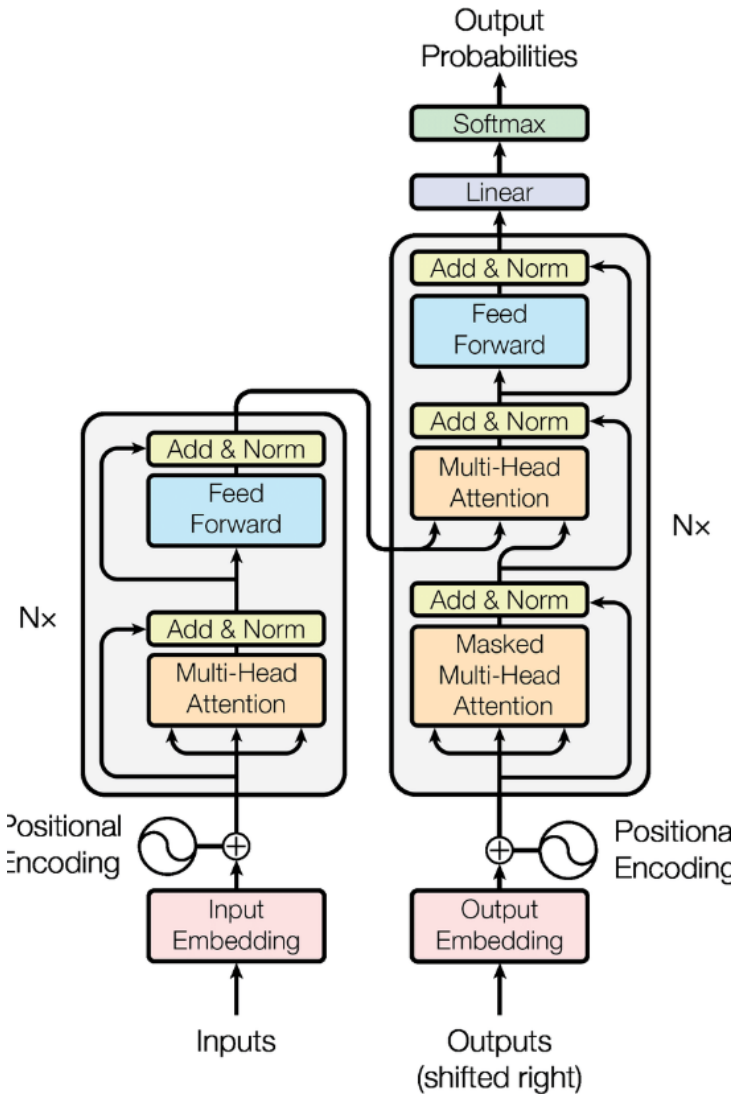
- **Encoder:** Mengubah input teks menjadi representasi yang kaya makna.
- **Decoder:** Menggunakan representasi ini untuk menghasilkan output teks (pada tugas seperti machine translation).

Setiap encoder dan decoder terdiri dari:

- **Multi-head Self-Attention Layer**
- **Feedforward Neural Network (FFN)**

- **Layer Normalization dan Residual Connection**

6.2.1 Skema Arsitektur



(Sumber: Vaswani et al., 2017)

6.2.2 Detail Proses

- **Input Embedding:** Setiap kata dikonversi menjadi vektor embedding.
- **Positional Encoding:** Karena tidak ada urutan dalam self-attention, informasi posisi kata ditambahkan.
- **Self-Attention:** Menentukan bobot antara setiap kata dan kata lain dalam kalimat.
- **FFN:** Layer neural biasa untuk memperkaya representasi.

6.2.3 Formulasi Self-Attention

Misalkan kita punya input vektor $x_1, x_2, ..., x_n$. Self-attention menghitung:

$$\text{Attention}\left(Q, K, V\right) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

dengan:

- $Q = XW^Q$ (Query)
- $K = XW^K$ (Key)
- $V = XW^V$ (Value)

Catatan: Semua kata saling "melihat" dan saling mempengaruhi berdasarkan skor kesamaan antara query dan key.

6.3 Self-Attention dan Multi-Head Attention

6.3.1 Self-Attention

Self-attention adalah mekanisme inti dari arsitektur Transformer. Tujuan utama dari self-attention adalah menangkap hubungan antara kata-kata dalam sebuah kalimat, tak peduli seberapa jauh jaraknya satu sama lain.

Misalnya pada kalimat: *"Dia meletakkan bukunya di atas meja karena itu sangat berat."* Kata "itu" mengacu pada "buku", dan hubungan ini bisa terlewat oleh model sekuensial tradisional. Dengan self-attention, model dapat memberikan bobot tinggi pada kata "buku" ketika memproses "itu".

Formulasi matematisnya:

$$\text{Attention}\left(Q, K, V\right) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

di mana:

- Q (query), K (key), dan V (value) adalah representasi dari input yang telah diproyeksikan melalui matriks pembelajaran.
- d_k adalah dimensi dari vektor key.
- Fungsi softmax digunakan untuk mendapatkan bobot distribusi perhatian terhadap seluruh token dalam input.

6.3.2 Multi-Head Attention

Satu perhatian (*attention head*) mungkin hanya menangkap satu jenis hubungan (misalnya, ketergantungan subjek-predikat). Untuk memperkaya representasi, Transformer menggunakan beberapa "kepala" perhatian secara paralel, disebut **multi-head attention**.

Secara umum:

$$\text{MultiHead}\left(Q, K, V\right) = \text{Concat}\left(\text{head}_1, ..., \text{head}_h\right)W^O$$
$$\text{di mana head}_i = \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right)$$

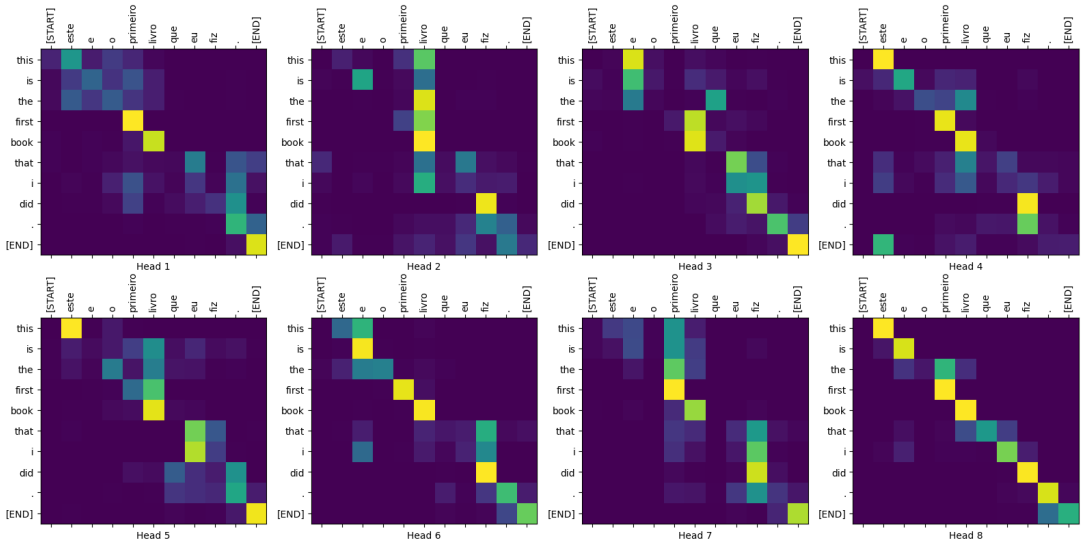
Setiap kepala belajar bobot perhatian secara berbeda, dan hasilnya digabungkan lalu diproyeksikan ulang.

6.3.3 Keunggulan Self-Attention

- **Non-sekuensial:** Semua kata dalam input dapat diproses secara paralel.
- **Efektif untuk dependensi panjang:** Tidak terpengaruh oleh jarak kata dalam kalimat.
- **Fleksibel dan adaptif:** Hubungan antar kata ditentukan oleh data.

6.3.4 Contoh Visualisasi Attention

Visualisasi attention sering ditampilkan sebagai heatmap, di mana setiap baris menunjukkan kata yang sedang diproses, dan setiap kolom menunjukkan bobot perhatian terhadap kata lain.



(Contoh visualisasi hubungan kata dalam kalimat menggunakan self-attention.)

6.4 Positional Encoding

6.4.1 Mengapa Positional Encoding Dibutuhkan?

Salah satu perbedaan utama antara arsitektur Transformer dan model sekuensial seperti RNN atau LSTM adalah bahwa Transformer **tidak memiliki urutan bawaan**. Artinya, Transformer tidak tahu urutan kata-kata dalam kalimat secara alami, karena semua token diproses secara paralel.

Untuk mengatasi hal ini, Transformer menggunakan **positional encoding** — yaitu informasi posisi yang ditambahkan ke embedding tiap token agar model tetap dapat memahami urutan kata dalam kalimat.

6.4.2 Formulasi Positional Encoding

Dalam makalah asli Transformer ("Attention is All You Need"), positional encoding diberikan secara deterministik dengan fungsi sinus dan cosinus:

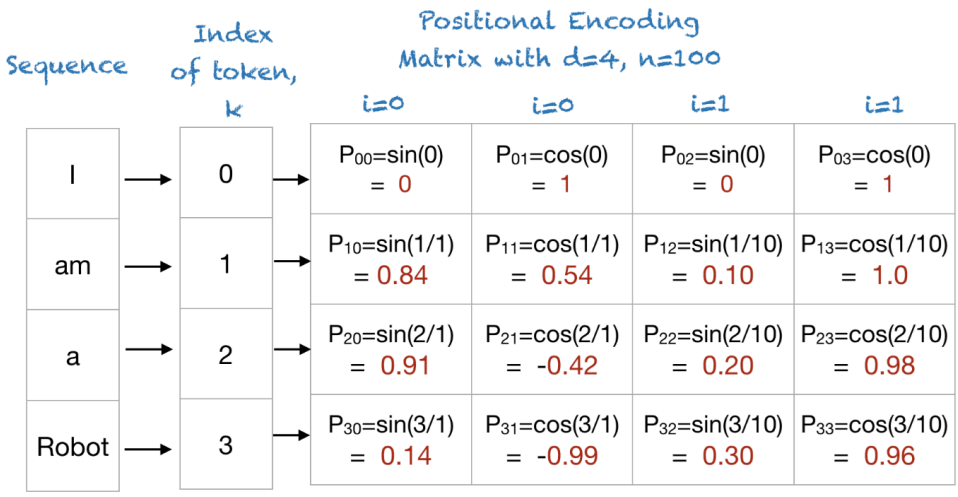
$$\text{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$
$$\text{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

di mana:

- pos adalah posisi token dalam urutan,
- i adalah dimensi embedding,
- d_{model} adalah dimensi total embedding.

Fungsi sinus dan cosinus digunakan karena sifat periodiknya dan kemampuannya untuk merepresentasikan berbagai skala posisi. Kombinasi nilai pada tiap dimensi ini memungkinkan model membedakan posisi satu kata dari yang lain.

6.4.3 Visualisasi Positional Encoding



Gambar di atas memperlihatkan representasi sinusoidal dari beberapa dimensi embedding untuk setiap posisi dalam urutan.

6.4.4 Alternatif Positional Encoding

Dalam praktik modern, banyak model besar (seperti BERT dan GPT) menggunakan **learnable positional embedding**, yaitu vektor posisi yang dipelajari selama pelatihan, bukan ditentukan secara tetap. Hal ini membuat model lebih fleksibel untuk data yang lebih bervariasi.

—

Catatan: Positional encoding ditambahkan secara langsung ke embedding input sebelum masuk ke layer self-attention.

"'latex

Contoh Implementasi Positional Encoding (Opsional)

Berikut ini contoh implementasi sederhana positional encoding sinusoidal dalam Python:

```
import numpy as np
import matplotlib.pyplot as plt

def positional_encoding(max_len, d_model):
    pe = np.zeros((max_len, d_model))
    for pos in range(max_len):
        for i in range(0, d_model, 2):
            angle = pos / np.power(10000, (2 * i)/d_model)
            pe[pos, i] = np.sin(angle)
            if i + 1 < d_model:
                pe[pos, i + 1] = np.cos(angle)
    return pe

pe = positional_encoding(50, 16)
plt.figure(figsize=(10, 6))
plt.imshow(pe, aspect='auto', cmap='coolwarm')
plt.xlabel("Embedding Dimensions")
plt.ylabel("Position")
plt.title("Positional Encoding Visualization")
plt.colorbar()
plt.show()
```

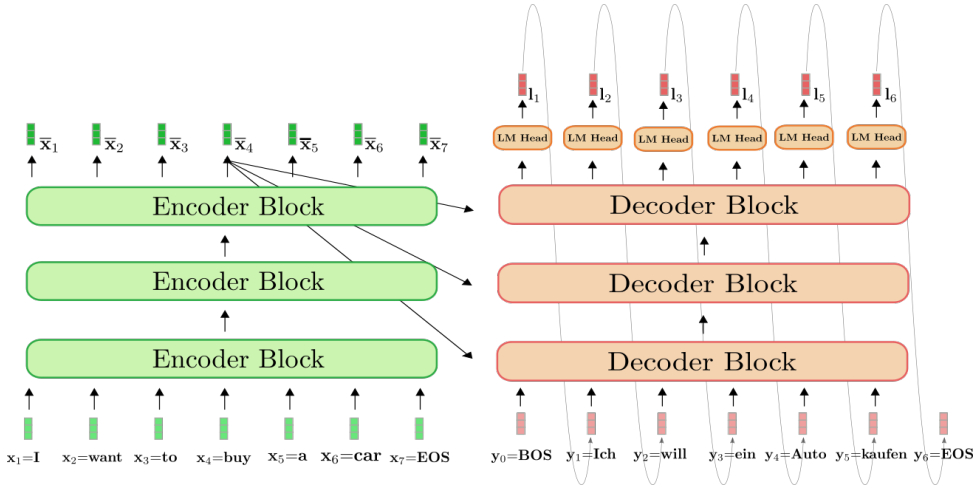
6.5 Arsitektur Encoder–Decoder Transformer

6.5.1 Struktur Umum

Arsitektur Transformer terdiri dari dua bagian utama:

- **Encoder**, yang memproses input sequence (misalnya kalimat dalam bahasa sumber).
- **Decoder**, yang menghasilkan output sequence (misalnya terjemahan ke bahasa target).

Setiap bagian terdiri dari beberapa layer identik (biasanya 6 atau lebih), dan masing-masing layer memiliki struktur modular.



Gambar 6.1. Struktur Encoder–Decoder Transformer (sumber: HunggingFace.com)

6.5.2 Komponen Encoder

Setiap layer **encoder** terdiri dari dua sublayer:

1. **Multi-head self-attention layer**: memungkinkan setiap posisi dalam input memperhatikan posisi lainnya.
2. **Feedforward layer**: jaringan saraf sederhana dua lapis yang diterapkan secara identik ke setiap posisi.

Setiap sublayer dibungkus dengan **residual connection** dan **layer normalization**.

6.5.3 Komponen Decoder

Setiap layer **decoder** terdiri dari tiga sublayer:

1. **Masked multi-head self-attention**: hanya memperhatikan token sebelumnya untuk menjaga sifat autoregresif.
2. **Multi-head attention terhadap output encoder**: decoder memperhatikan seluruh output encoder.
3. **Feedforward layer** seperti pada encoder.

6.5.4 Self-Attention dan Masking

Untuk menjaga agar output pada posisi t dalam decoder tidak dapat melihat kata-kata setelahnya (agar tidak "mencontek"), digunakan teknik **masking**, yaitu menyetel nilai attention ke $-\infty$ (dan setelah softmax menjadi 0) untuk token-token masa depan.

6.5.5 Output Model

Hasil akhir dari decoder diteruskan ke sebuah layer linear dan fungsi softmax untuk menghasilkan distribusi probabilitas atas kosakata.

6.5.6 Kelebihan Arsitektur Ini

Arsitektur Transformer unggul karena:

- Seluruh input dapat diproses secara paralel (tidak seperti RNN).
- Mampu menangkap hubungan jarak jauh antar token berkat mekanisme attention.
- Skalabilitas tinggi dan mudah diparalelkan pada hardware modern (GPU/TPU).

6.6 Pra-pelatihan dan Fine-Tuning Transformer

6.6.1 Pra-pelatihan (Pretraining)

Model Transformer modern biasanya dilatih melalui dua tahap utama:

1. **Pra-pelatihan**: model dilatih pada dataset besar tanpa label menggunakan tugas-tugas sederhana, seperti:
   - **Masked Language Modeling (MLM)** – seperti pada BERT: beberapa token disembunyikan dan model dilatih untuk memprediksi token yang hilang.
   - **Causal Language Modeling (CLM)** – seperti pada GPT: model memprediksi token berikutnya dari urutan sebelumnya.
2. Tujuannya adalah agar model belajar representasi bahasa umum yang dapat diterapkan ke berbagai tugas NLP.

6.6.2 Fine-Tuning

Setelah pra-pelatihan, model dilatih ulang (**fine-tuning**) pada dataset yang lebih kecil dan spesifik, seperti:

- Analisis sentimen
- Klasifikasi topik
- Named Entity Recognition
- Question answering

Fine-tuning dilakukan dengan menambahkan layer klasifikasi sederhana di atas output Transformer dan melatih seluruh model atau hanya sebagian layer.

6.6.3 Contoh Model Populer

- **BERT (Bidirectional Encoder Representations from Transformers)** – hanya menggunakan encoder; dilatih dengan MLM dan Next Sentence Prediction.
- **GPT (Generative Pretrained Transformer)** – hanya menggunakan decoder; dilatih dengan CLM.

- **RoBERTa, ALBERT, XLNet, DistilBERT** – berbagai varian dari BERT.
- **T5, BART** – encoder-decoder transformer yang cocok untuk generatif tasks seperti summarization dan translation.

6.6.4 Transfer Learning dalam NLP

Transformer mendorong revolusi dalam NLP karena pendekatan **transfer learning**:

- Model yang dilatih secara umum (pretrained) dapat digunakan kembali untuk berbagai tugas baru.
- Cukup sedikit data yang diperlukan untuk fine-tuning.

6.6.5 Contoh Implementasi Fine-Tuning dengan HuggingFace (Opsional)

```
from transformers import BertTokenizer, BertForSequenceClassification
from transformers import Trainer, TrainingArguments
from datasets import load_dataset

# Load dataset dan tokenizer
dataset = load_dataset("imdb", split='train[:5000]')
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

def tokenize(example):
    return tokenizer(example["text"],
    truncation=True, padding="max_length")

dataset = dataset.map(tokenize, batched=True)
dataset = dataset.rename_column("label", "labels")
dataset.set_format("torch", columns=["input_ids",
    "attention_mask", "labels"])

# Load model
model = BertForSequenceClassification.
    from_pretrained("bert-base-uncased")

# Training
args = TrainingArguments(output_dir="./results",
    per_device_train_batch_size=8, num_train_epochs=1)
trainer = Trainer(model=model, args=args, train_dataset=dataset)
trainer.train()
```

Catatan: Untuk pelatihan model skala besar, diperlukan GPU dan waktu pelatihan yang cukup lama. Namun, sebagian besar model sudah tersedia dalam bentuk pretrained dan dapat digunakan secara instan.

Referensi dan Bacaan Lanjutan

Berikut ini adalah sumber-sumber rujukan yang direkomendasikan untuk memperdalam pemahaman tentang pengolahan bahasa alami dan topik-topik lanjutan di bidang NLP:

1. **Jurafsky, D., & Martin, J. H.** (2023). *Speech and Language Processing (3rd Edition, Draft)*.
   Tersedia online: https://web.stanford.edu/~jurafsky/slp3/ Buku teks standar dunia untuk NLP klasik dan modern, mencakup topik-topik dasar hingga transformer dan pretrained models.
2. **Hugging Face Course (Free Online)** https://huggingface.co/learn/nlp-course Kursus praktis dengan banyak contoh kode Python menggunakan pustaka `transformers`, sangat cocok bagi pemula maupun lanjutan.
3. **Sebastian Ruder – NLP Summary Blog**
   https://www.ruder.io/tag/natural-language-processing/ Ringkasan panduan pembelajaran NLP dari awal, beserta artikel dan roadmap.
4. **The Illustrated Transformer** oleh Jay Alammar https://jalammar.github.io/illustrated-transformer/ Visualisasi intuitif dan penjelasan mendalam tentang cara kerja transformer.
5. **Google Colab Notebook for NLP Tasks** https://github.com/keras-team/keras-nlp Contoh-contoh penerapan NLP menggunakan `keras-nlp` dan `tensorflow`.
6. **Dataset NLP dan Kode Implementasi:** https://huggingface.co/datasets dan https://github.com/huggingface/transformers Menyediakan ribuan dataset NLP dan model pretrained yang siap digunakan.
7. **NLP-Tutorial** oleh graykode (GitHub) https://github.com/graykode/nlp-tutorial Implementasi dari berbagai metode NLP (word2vec, RNN, transformer, BERT) dalam kode Python minimalis.

Sumber-sumber ini dapat digunakan untuk eksplorasi lebih lanjut, baik untuk memahami teori maupun untuk eksperimen praktis. Beberapa di antaranya bersifat interaktif dan dapat langsung dijalankan di Google Colab tanpa instalasi lokal.

Bibliografi

[DBCR19] David M Diez, Christopher D Barr, and Mine Cetinkaya-Rundel, *Openintro statistics*, 3rd ed., OpenIntro, 2019.

[Dev23] Google Developers, *Machine learning crash course*, https://developers.google.com/machine-learning/crash-course, 2023.

[DFO20] Marc Peter Deisenroth, A Aldo Faisal, and Cheng Soon Ong, *Mathematics for machine learning*, Cambridge University Press, 2020.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*, MIT Press, 2016, Book in preparation for MIT Press.

[HTF01] Trevor Hastie, Robert Tibshirani, and Jerome Friedman, *The elements of statistical learning*, Springer Series in Statistics, Springer New York Inc., New York, NY, USA, 2001.

[ID17] Barbara Illowsky and Susan Dean, *Introductory statistics*, OpenStax, 2017.

[JWHT13] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, *An introduction to statistical learning*, Springer, 2013.

[Uni23] Duke University, *Data science math skills*, https://www.coursera.org/learn/datasciencemathskills, 2023.

[Was04] Larry Wasserman, *All of statistics: A concise course in statistical inference*, Springer, 2004.