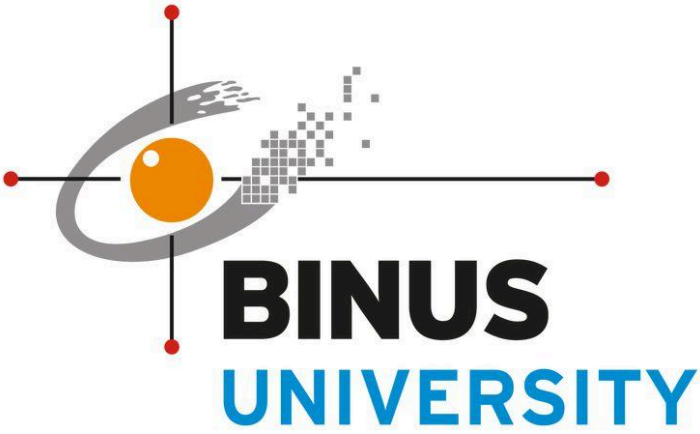
LAPORAN MODEL KLASIFIKASI TEXT MENTAL HEALTH DENGAN NAIVE BAYES



Disusun Oleh:

2702295695 – Joseph Christian Yusmita

2702261353 – Ronald Arya

2702260924 – Jason Manuel Wijaya

2702387941 - Dermagani Muktiasa

School of Computer Science

BINUS University

Jakarta, Indonesia

2025

1. Pendahuluan

Masalah kesehatan mental merupakan hal penting yang sering diabaikan dalam kehidupan sehari-hari. Di zaman sekarang, banyak orang membagikan isi pikirannya termasuk keluhan atau perasaan emosional melalui media sosial atau platform online dalam bentuk teks. Teks-teks ini bisa menjadi sumber informasi berharga untuk mengenali kondisi mental seseorang secara otomatis dengan bantuan machine learning.

Proyek ini bertujuan untuk mengklasifikasikan tulisan-tulisan tersebut ke dalam tiga kategori: kondisi normal (label 0), gangguan ringan (label 1), dan gangguan serius (label 2). Dengan melakukan klasifikasi ini, kita bisa membantu proses deteksi dini dan memberikan dukungan lebih cepat kepada individu yang membutuhkan.

Untuk membangun sistem klasifikasi ini, kami menggunakan algoritma **Naive Bayes**, sebuah metode statistik yang sederhana namun efektif dalam menangani teks. Naive Bayes bekerja berdasarkan prinsip probabilitas dan sering digunakan dalam tugas-tugas seperti filtering email atau analisis sentimen karena kecepatannya dan kemampuannya dalam bekerja dengan data berukuran besar.

Dataset yang digunakan berupa file CSV berisi teks dan label masing-masing. Proses pembangunan model meliputi pembersihan data, penyeimbangan distribusi kelas, pembagian data untuk pelatihan dan pengujian, serta evaluasi hasil klasifikasi. Penjelasan lebih lengkap mengenai tahapan dan hasil model Naive Bayes akan disampaikan dalam dokumentasi ini.

2. Literatur Review

2.1 Naive Bayes

Naive Bayes adalah algoritma klasifikasi yang didasarkan pada teorema Bayes dengan asumsi independen (naive) antar fitur (atribut). Algoritma ini banyak digunakan dalam masalah klasifikasi teks. Teorema Bayes menghitung probabilitas suatu hipotesis ***H*** berdasarkan data ***D***:

$$P(H|D) = \frac{P(D|H) \cdot P(H)}{P(D)}$$

Dengan keterangan :

- P(H | D) = Probabilitas hipotesis H setelah melihat data D.
- P(D | H) = Probabilitas data D jika hipotesis HHH benar.
- P(H) = Probabilitas awal hipotesis H.
- P(D) = Probabilitas data D secara keseluruhan.

Naive Bayes mengasumsikan bahwa fitur-fitur saling independen satu sama lain**,** sehingga:

$$P(X_1, X_2, ..., X_n|C) = P(X_1|C) \cdot P(X_2|C) \cdot ... \cdot P(X_n|C)$$

Dengan **C** sebagai kelas, dan $X_1, X_2, \ldots, X_n$ adalah fitur. Dalam pembuatan model klasifikasi teks, proses klasifikasi Bayes terdiri dari empat langkah utama. Pertama kita perlu menghitung *prior* (probabilitas awal) setiap kelas C

$$P(C|X) = \frac{Jumlah\ Data\ di\ kelas\ C}{Jumlah\ Total\ Data}$$

Menghitung *likelihood* untuk setiap fitur X_i terhadap kelas C. Nilai *likelihood* ini akan menjadi nilai “model” yang akan dibuat dan digunakan untuk membuat prediksi texts.

$$P(X_i|C)$$

Menghitung posterior (*frequency map*) untuk setiap kelas menggunakan teorema bayes

$$P(C|X) \propto P(C) \cdot \prod_{i=1}^{n} P(Xi|C)$$

Nilai posterior tertinggi adalah hasil prediksi dari model klasifikasi teks dengan naive bayes.

2.2 Laplace Smoothing

Laplace smoothing atau juga dikenal sebagai *add-one smoothing* digunakan untuk mengatasi masalah probabilitas nol ketika suatu fitur tidak pernah muncul dalam data pelatihan untuk suatu kelas tertentu. Ketika menghitung nilai posterior dari naive bayes maka akan digunakan rumus:

$$P(x|c) = \frac{Count(x,c)}{\Sigma w(Count(x', c) + \alpha)}$$

Nilai α merupakan laplace smoothing yang biasa memiliki nilai $\alpha = 1$. jika count(x, c) = 0, maka P(x|c)=0. Karena Naive Bayes mengalikan probabilitas kata-kata secara beruntun, satu saja probabilitas nol akan membuat keseluruhan P(x|c) menjadi nol mengakibatkan kehilangan data. Maka dari itu nilai α memastikan bahwa penyebut dari persamaan tersebut tidak akan 0.

2.3 Porter Stemmer Algorithm

Algoritma Porter Stemmer adalah salah satu metode untuk melakukan stemming (memotong imbuhan) pada kata berbahasa Inggris. Tujuannya mengubah kata turunan (misalnya “running”, “runner”, “ran”) menjadi bentuk akar (stem) yang umum (“run”). Meskipun dirancang untuk bahasa Inggris, konsep dasarnya berguna dipahami sebelum menerapkannya pada bahasa lain. Algoritma ini menjadi fondasi bagi banyak *natural language processing* (NLP) model. Dimana akan diaplikasikan pada tahap *preprocessing* sehingga “kosakata” model lebih sederhana dan memungkinkan prediksi yang lebih akurat.

3. Implementasi

Berikut adalah langkah - langkah yang dilakukan untuk membuat model yang dapat mengklasifikasikan text:

3.1 Pembacaan Data

Langkah pertama yang dilakukan adalah membaca file CSV menggunakan menggunakan function `read_csv()` dari library pandas dengan delimiter `;`. Karena pada data terdapat beberapa data yang *malformed* maka diperlukan opsi `on_bad_lines=”skip”` yang akan menghiraukan baris yang tidak sesuai.

```
path = f"{os.getcwd()}/data/training-data.csv"
data = pd.read_csv(path, on_bad_lines="skip", delimiter=';')
```

3.2 Preprocessing Data

Untuk memastikan hasil akhir dari model memiliki nilai akurasi yang baik, perlu dilakukan beberapa metode preprocessing untuk data yang akan digunakan. Pada model ini langkah - langkah preprocessing yang dilakukan adalah :

1. Pengubahan seluruh string menjadi lowercase :
   “They’re running towards us” → “they’re running towards us”

2. Menghilangkan karakter yang bukan alpha numeric menggunakan `re.sub()` :
   “they’re running towards us” → “theyre running towards us”

3. Menghilangkan *stop words* (bahasa konjungsi atau penghubung) menggunakan data dari library `nltk` :
   “theyre running towards us” → “theyre running towards”

4. Menggunakan algoritma *Porter Stemmer* untuk menyederhanakan kosakata:
   “theyre running towards” → “theyre run toward”

Hasil preprocessing diaplikasikan pada dataframe asli dengan menggunakan metode apply() untuk menghindari pembuatan variabel baru yang redundan

```
nltk.download('stopwords', quiet=True)
nltk.download('punkt_tab', quiet=True)

def preprocess_text(text):
  # Typecasts non string values into strings
  if not isinstance(text, str):
    text = str(text)

  # Converts text to lower case
  text = text.lower()

  # Remove special characters
  text = re.sub(r'[^\w\s]', '', text)

  # Tokenize string
  tokens = word_tokenize(text)

  # Remove stop words (words with low information value)
  stopwords_set = set(stopwords.words("english"))
  tokens = [word for word in tokens if word not in stopwords_set]

  # Apply porter stemmer algorithm to simplify vocabulary
  tokens = [PorterStemmer().stem(word) for word in tokens]
```

```
  return ' '.join(tokens)

data['text'] = data['text'].apply(preprocess_text)
```

Screenshot Hasil :

```
Before :
Every time I move my chair closer to talk to him, he keep fidgeting and moving away. Do I smell ?
After:
everi time move chair closer talk keep fidget move away smell
```

3.3 Pembagian Train, Validation dan Test Data

Untuk keperluan training, test dan validation data dibagi menjadi 70% untuk training, 20% untuk test dan 10% untuk validation menggunakan `train_test_split()`

```
# First Split: separate out test data (90% train + validation, 10% test)
train_validation_data, test_data = train_test_split(
  data,
  train_size=0.8,
  test_size=0.2,
  random_state=42
)

# Second Split: separate out train 70% and 20% validation
train_data, validation_data = train_test_split(
  train_validation_data,
  train_size=0.7,
  test_size=0.2,
  random_state=42
)
```

Screenshot Hasil :

```
Train: (1643, 3), Test: (588, 3), Validation: (470, 3)
```

3.4 Perhitungan Probabilitas Awal (Prior) Label

Menghitung probabilitas awal dari setiap label (0, 1, 2) `train_data` dengan mengaplikasikan rumus berikut pada seluruh label.

$$P(C|X) = \frac{Jumlah\ Data\ di\ kelas\ C}{Jumlah\ Total\ Data}$$

Hasil prior disimpan dalam sebuah numpy array `initial_probability`

```
labels = train_data.label
total_data_frequency = len(labels)
label_frequencies = labels.value_counts().to_numpy()
initial_probability = np.array(
  [frequency / total_data_frequency for frequency in label_frequencies]
)
```

Screenshot Hasil:
Nilai `initial_probability` atau prior

```
array([0.44978698, 0.28545344, 0.26475959])
```

3.5 Perhitungan Frekuensi Token (*likelihood*) per Label

Langkah berikutnya yaitu menghitung *likelihood* untuk setiap fitur X_i terhadap C dimana X_i adalah token ke-i dan C adalah label (0,1,2). Pada konteks ini, token merupakan seluruh kata - kata yang terdapat dalam dataset. Proses tersebut dilakukan dengan mengiterasi semua token di train_data kemudian menghitung total token per label dan frekuensi kemunculan token berdasarkan label.

```
token_freq_map = {}
token_freq_per_label = [0, 0 ,0]
for i, row in train_data.iterrows():
  label = row.label
  tokens = str(row.text).split()

  # Add the tokens count for the label
  token_freq_per_label[label] += len(tokens)
```

```
  for token in tokens:
    if token not in token_freq_map:
      # Initialize empty list for non existing tokens
      token_freq_map[token] = [0, 0, 0]

    token_freq_map[token][label] += 1
```

Pada setiap iterasi `train_data`, kalimat akan dipecah menggunakan `split()` menjadi token - token kata (feature extraction). Kemudian token tersebut yang akan dihitung kemunculannya pada setiap label dan disimpan dalam variabel `token_freq_map`. Nilai pada variabel tersebut merupakan **model parameter** yang akan dijadikan acuan untuk memprediksi `test_data`

Screenshot Hasil:
Contoh entry pada `token_freq_map` untuk beberapa kata.

```
'month': [5, 12, 16],
'win': [3, 1, 1],
'heart': [0, 5, 3],
'get': [26, 71, 81],
'defeat': [1, 1, 0],
```

3.6 Melakukan Prediksi

Langkah berikutnya adalah untuk mendapatkan model parameter kedua yaitu prediksi nilai peluang pada setiap token kata yang dilakukan menggunakan rumus berikut:

$$P(x|c) = \frac{Count(x,c)}{\Sigma w(Count(x', c) + \alpha)}$$

Implementasi dalam kode adalah sebagai berikut:

```
def calculate_probability(smoothing):
  token_probabilities = {}
  token_length = len(token_freq_map)

  for token, occurences in token_freq_map.items():
    token_probabilities[token] = []
    for label, occurence in enumerate(occurences):
```

```
      # p(w | L) = (Count(w,L) + a) / TotalWords(L) + a·V
      probability = (occurence + smoothing) / (token_freq_per_label[label] +
smoothing * token_length)
      token_probabilities[token].append(probability)

  return token_probabilities
```

Fungsi `calculate_probability()` akan menerapkan rumus di atas untuk setiap token kata yang terdapat dalam `token_freq_map` dengan nilai *laplace smoothing* α dijadikan sebagai parameter. Setelah dilakukan perhitungan probabilitas seluruh token kata data akan dikembalikan dalam bentuk dict seperti berikut

```
'someon': [0.0011761630946157868, 0.0044582437704004454, 0.0031178208073514933]
'who': [0.00026136957658128593, 0.0002388344877000239, 0.0003281916639317361]
'fight': [0.0003920543648719289, 0.0005572804713000557, 0.0004922874958976042]
'5': [0.0009147935180345008, 0.0003980574795000398, 0.0003281916639317361]
'month': [0.0007841087297438578, 0.0010349494467001034, 0.0013948145717098785]
```

Menggunakan nilai `token_probabilities`, model sudah dapat melakukan prediksi menggunakan rumus prediksi:

$$P(C|X) \propto P(C) \cdot \prod_{i=1}^{n} P(Xi|C)$$

Pada kode proses perhitungan dilakukan dalam function `classify()` yang menerima nilai `text` untuk diprediksi dan model parameter `token_probabilities` yang berasal dari fungsi `calculate_probability()` diatas.

```
def classify(text, token_probabilities):
  scores = initial_probability.copy()
  tokens = text.split()

  # Calculate probability for each label
  for label in range(len(scores)):
    for token in tokens:
      if token in token_probabilities:
```

```
        scores[label] *= token_probabilities[token][label]

  # Retrieve the largest guess score's index (AKA. the label)
  predicted_label = np.argmax(scores)
  return predicted_label
```

Berikut cara kerja kode diatas:

```
scores = initial_probability.copy()
```

Nilai `scores` merepresentasikan *P(C)*, nilai probabilitas untuk setiap kelas

```
for label in range(len(scores)):
  for token in tokens:
    if token in token_probabilities:
      scores[label] *= token_probabilities[token][label]
```

Kode diatas merupakan logika utama untuk melakukan komputasi:

$$P(C) \cdot \prod_{i=1}^{n} P(Xi|C)$$

Dimana untuk setiap `token` Xi dilakukan akumulasi perkalian pada kelas C (label) yang di akumulasi, menghasilkan nilai *posterior* yang disimpan dalam `scores`

```
predicted_label = np.argmax(scores)
```

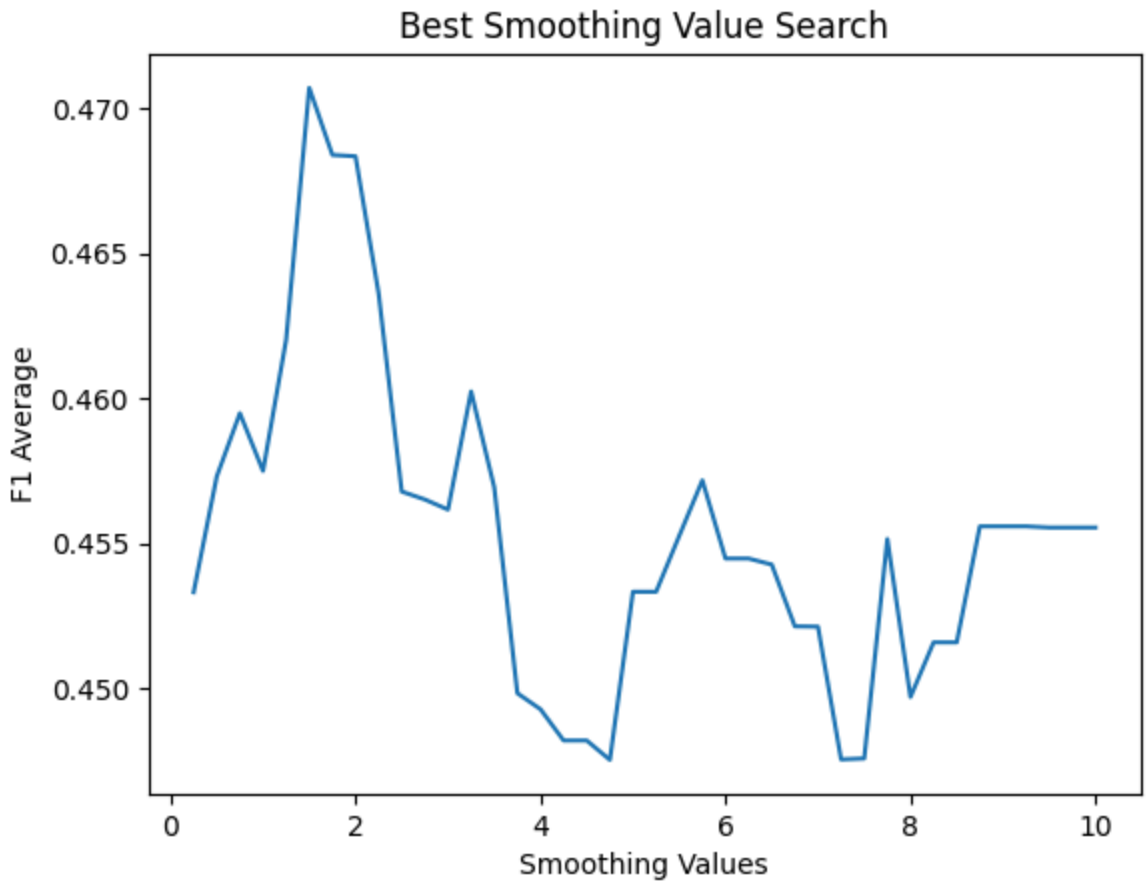
Terakhir, dari seluruh kemungkinan yang dapat terjadi, nilai *posterior* tertinggi diambil menggunakan function `np.argmax()`. Nilai tertinggi tersebut merupakan hasil prediksi dari model

3.7 Pencarian Nilai Laplace Smoothing Optimal

Untuk mengoptimasi akurasi model lebih lanjut, pencarian nilai laplace smoothing optimal dilakukan dengan menggunakan metode *grid-search* nilai 0 hingga 10 dengan step 0.25. Pada setiap iterasi nilai f1-score pada `validation_data` dibandingkan kemudian nilai terbaik disimpan dalam `best_smoothing`

```
MAX_SMOOTHING = 10
smoothing_values = np.arange(0, MAX_SMOOTHING + 0.1, 0.25)
best_smoothing = smoothing_values[0]
best_f1 = 0

history = []

for smoothing in smoothing_values:
  token_probabilities = calculate_probability(smoothing=smoothing)

  val_predictions = [
    classify(data.text, token_probabilities)
    for _, data in validation_data.iterrows()
  ]
  val_actual = [data.label for _, data in validation_data.iterrows()]

  # Get f1 score to compare
  report = classification_report(
    val_actual,
    val_predictions,
    labels=[0,1,2],
    output_dict=True
  )
  f1_average = report["weighted avg"]["f1-score"]

  history.append(f1_average)

  if best_f1 < f1_average:
    best_smoothing = smoothing
    best_f1 = f1_average
```

Setelah proses grid-search ditemukan bahwa nilai laplace smoothing optimal adalah pada angka `1.25` dengan nilai f-score 0.475 sesuai dengan trend yang ditunjukan pada grafik dibawah ini :



3.8 Evaluasi Performa Model

Langkah akhir dalam pembuatan model ini adalah dengan melakukan komparasi nilai *classification report* dan confusion matrix berdasarkan nilai `test_data`.

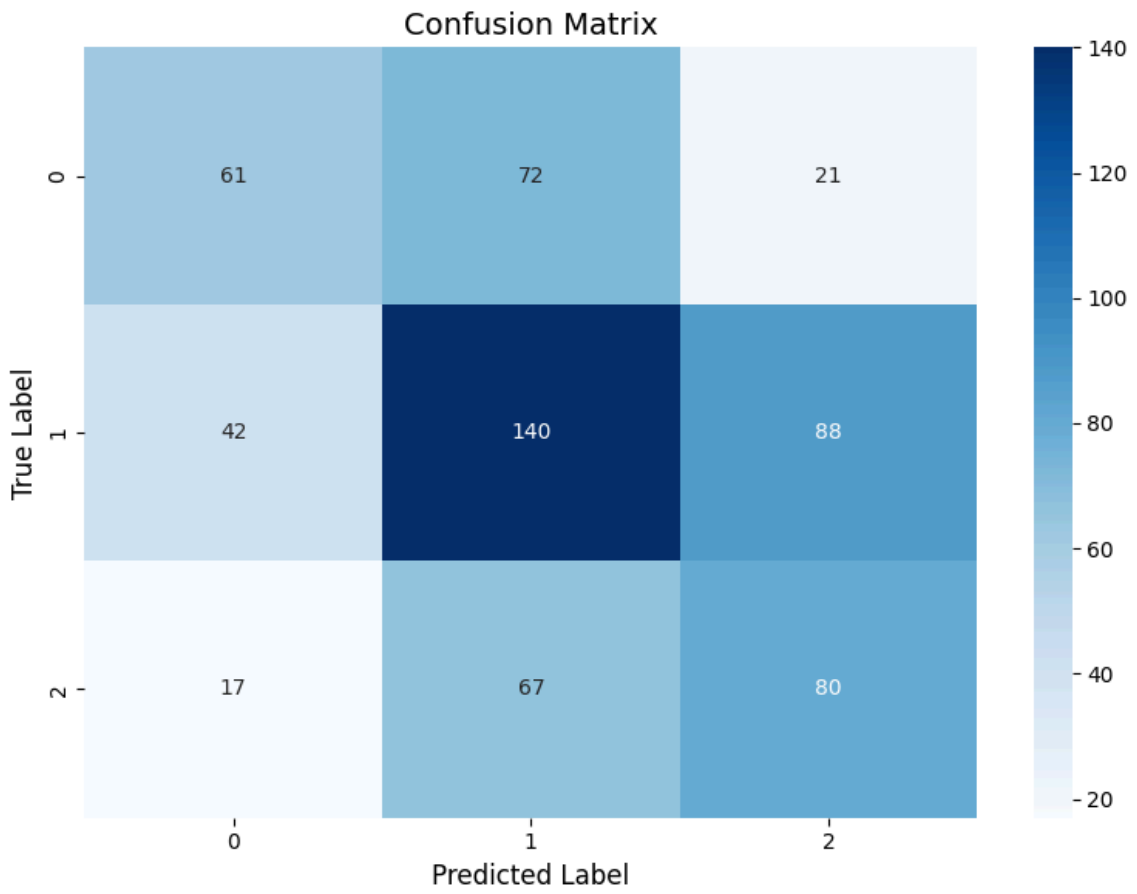
```
token_probabilities = calculate_probability(best_smoothing)
predictions = [classify(data.text, token_probabilities) for _, data in
test_data.iterrows()]
actual = [data.label for _, data in test_data.iterrows()]

print(classification_report(actual, predictions, labels=[0,1,2]))
```

Classification Report:

	precision	recall	f1-score	support
0	0.51	0.40	0.45	154
1	0.50	0.52	0.51	270
2	0.42	0.49	0.45	164
accuracy			0.48	588
macro avg	0.48	0.47	0.47	588
weighted avg	0.48	0.48	0.48	588

- **Accuracy (Akurasi)**: 0.48
  Hanya **48%** prediksi yang benar. Ini rendah, mengingat ini adalah klasifikasi 3 kelas (tebakan acak bisa dapat ~33%).
- **Macro average**:
  Rata-rata precision/recall/F1 dari semua kelas secara merata tanpa memperhatikan jumlah datanya. Menunjukkan bahwa model kesulitan di semua kelas, bukan hanya di kelas tertentu.
- **Weighted average**:
  Rata-rata yang memperhitungkan jumlah data per kelas. Nilainya sedikit lebih baik karena kelas 1 punya data lebih banyak dan performa lebih bagus.

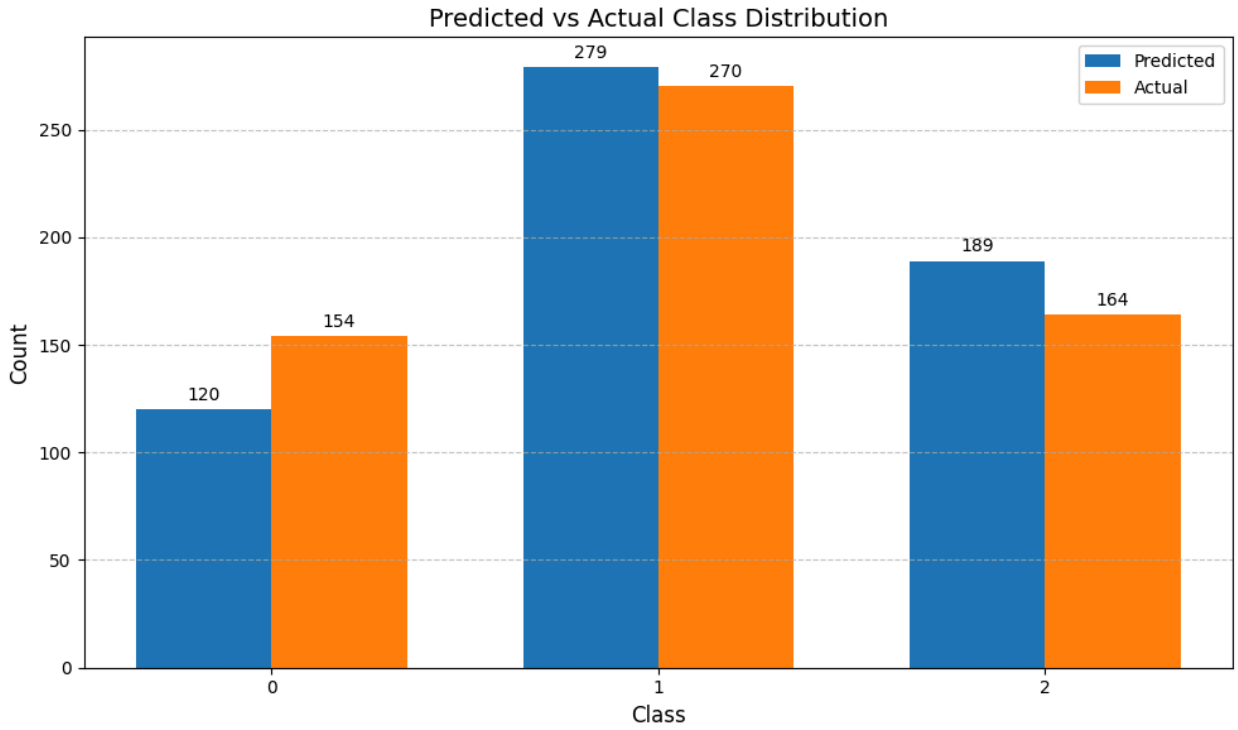
Confusion Matrix:



Confusion matrix diatas menunjukkan bahwa model masih kesulitan membedakan antar kelas, terutama antara kelas 0 dan 1, serta kelas 1 dan 2. Pada kelas 0, hanya 61 data yang terklasifikasi dengan benar, sedangkan 72 data diprediksi sebagai kelas 1 dan 21 sebagai kelas 2. Kelas 1 yang memiliki jumlah data terbanyak (270) hanya 140 yang benar, sementara 42 diprediksi sebagai kelas 0 dan 88 sebagai kelas 2. Untuk kelas 2, dari 164 data, hanya 80 yang benar, dan sebagian besar lainnya (67 data) justru diprediksi sebagai kelas 1.

Pola ini menunjukkan bahwa model masih sering salah dalam memetakan prediksi, terutama karena banyaknya kemiripan antar kelas. Hal ini konsisten dengan metrik seperti precision, recall, dan f1-score yang rendah dan tidak merata.

Class Distribution:



Grafik ini menunjukkan perbandingan jumlah data aktual dan prediksi model untuk kelas 0, 1, dan 2. Model kurang memprediksi kelas 0 (120 prediksi vs 154 asli), tapi *overpredicting* kelas 1 (279 vs 270) dan kelas 2 (189 vs 164). Ini menandakan model cenderung bias ke kelas 1, sehingga banyak data kelas 0 dan 2 salah diklasifikasikan sebagai kelas 1.

4. Kesimpulan dan Saran

Model Naive Bayes dasar memiliki performa sedang pada dataset ini, kemungkinan karena distribusi kosa kata yang mirip antar kelas dan jumlah dataset yang terbatas.

Saran Perbaikan:

- Terapkan teknik balancing (SMOTE / oversampling) jika imbalance label signifikan.
- Tambahkan fitur n-gram (bi-gram/trigram) untuk menangkap konteks.
- Gunakan teknik TF-IDF atau embedding (Word2Vec, FastText).
- Coba algoritma lain (SVM, Random Forest, atau neural network) dan bandingkan performa.