BAB 4
HASIL PENELITIAN

4.1 Ringkasan Hasil Penelitian

Penelitian ini merupakan studi eksperimental yang bertujuan untuk menganalisis performa algoritma Extreme Gradient Boosting (XGBoost) dalam mendeteksi malware pada aplikasi Android berbasis fitur *permissions*. Fokus utama diarahkan pada penanganan ketidakseimbangan kelas (*imbalanced class*), di mana jumlah aplikasi non-malware dalam dataset jauh mendominasi dibandingkan malware. Tantangan ini menjadi inti dari penelitian, bukan pada perluasan cabang klasifikasi (*multi-tree*), melainkan pada pendalaman performa klasifikasi biner secara presisi.

Untuk mengatasi masalah tersebut, eksperimen memanfaatkan teknik penyesuaian bobot kelas melalui parameter *scale_pos_weight*, dan menguji dua pendekatan tambahan yaitu *threshold tuning* (pengaturan ambang klasifikasi) serta SMOTE (*Synthetic Minority Over-sampling Technique*) sebagai metode *oversampling* kelas minoritas. Tujuan dari pendekatan-pendekatan ini adalah untuk meningkatkan kemampuan model dalam mengenali malware (kelas minoritas) tanpa harus memperbesar struktur model secara kompleks.

Tahapan eksperimen diawali dengan pre*processing* data, meliputi pembersihan nilai non-numerik, *valid*asi binerisasi fitur, dan pemilihan 10 fitur *permission* paling relevan berdasarkan frekuensi dan hubungan terhadap kelas target. Dataset kemudian dibagi menggunakan metode *Stratified* split (80:20), dan dihitung nilai *scale_pos_weight* berdasarkan distribusi kelas dalam data latih. Model XGBoost awal dilatih menggunakan parameter *default* sebagai *baseline*, lalu dioptimasi melalui GridSearchCV untuk mendapatkan konfigurasi parameter terbaik.

Evaluasi dilakukan dengan menghitung metrik performa utama seperti akurasi, *precision*, *recall*, dan *F1-Score*, serta ditunjang visualisasi *confusion*

matrix dan ROC-AUC curve. Untuk mengamati pengaruh skala data terhadap performa model dan efisiensi waktu komputasi, dilakukan eksperimen pada tiga subset data berukuran 500, 1000, dan 2000 sampel. Model terbaik dari hasil *tuning* juga dievaluasi dengan pendekatan *threshold tuning* dan SMOTE, untuk membandingkan dampaknya terhadap deteksi malware, khususnya pada perbaikan nilai *recall* dan keseimbangan performa keseluruhan.

Hasil menunjukkan bahwa meskipun model XGBoost awal menghasilkan *precision* tinggi, performa *recall* masih terbatas akibat dominasi kelas mayoritas. Melalui kombinasi pendekatan *threshold tuning* dan SMOTE, model berhasil meningkatkan *recall* dan *F1-Score* secara signifikan, tanpa mengorbankan akurasi secara drastis. Ini menunjukkan bahwa pendekatan eksperimental yang dilakukan mampu mengatasi tantangan ketidakseimbangan kelas dengan efektif, dan menjadi *goals* utama dari penelitian ini, bahwa performa model bukan ditentukan oleh banyaknya cabang pohon keputusan, tetapi oleh strategi dan pendekatan yang digunakan untuk memaksimalkan deteksi pada kelas minoritas.

4.2 Pembahasan Hasil Pemeodelan

Penelitian ini dimulai dengan membuat *baseline* sebagai pembanding awal. *Baseline* dihitung menggunakan dua cara: memprediksi semua data sebagai non-malware (kelas mayoritas), dan memprediksi secara acak berdasarkan proporsi kelas. *Baseline* ini memberikan gambaran mengenai performa dasar model sebelum penerapan algoritma pembelajaran mesin.

Tahapan selanjutnya adalah pre*processing* data, meliputi pembersihan nilai kosong dan konversi data menjadi format biner, serta seleksi 10 fitur *permission* yang dianggap paling relevan. Fitur-fitur ini dipilih karena berkaitan erat dengan aktivitas yang rentan dimanfaatkan oleh aplikasi malware, seperti akses lokasi, kontrol penyimpanan, dan akses jaringan. Data kemudian dipisahkan menjadi variabel input (X) dan label target (y) sebagai representasi *ground truth* dari klasifikasi malware dan non-malware. Analisis awal mengonfirmasi bahwa dataset mengalami ketidakseimbangan kelas yang signifikan.

Sebagai model utama, algoritma XGBoost digunakan untuk melakukan klasifikasi. Model pertama dilatih dengan parameter *default* sebagai model awal (*baseline* ML). Selanjutnya, dilakukan *tuning hyperparameter* menggunakan GridSearchCV, dengan fokus utama pada parameter *scale_pos_weight* untuk menyesuaikan pengaruh kelas mayoritas terhadap minoritas dalam proses pelatihan. Nilai ini dihitung berdasarkan rasio jumlah non-malware terhadap malware dalam data latih. Teknik ini menjadi strategi utama dalam menangani ketidakseimbangan kelas tanpa perlu mengubah struktur model.

Model terbaik hasil *tuning* kemudian dievaluasi menggunakan metrik klasifikasi seperti akurasi, *precision*, *recall*, dan *F1-Score*. Untuk mendukung interpretasi, juga ditampilkan *confusion matrix* dan nilai AUC. Analisis dilakukan tidak hanya pada keseluruhan data, tetapi juga melalui eksperimen subset data berukuran 500, 1000, dan 2000 untuk mengkaji pengaruh ukuran data terhadap performa dan waktu komputasi model. Penyesuaian *threshold* klasifikasi (*threshold tuning*) dilakukan untuk meningkatkan *recall* terhadap kelas malware, dan diterapkan pula metode SMOTE untuk menyeimbangkan kelas melalui *oversampling* secara sintetik.

Hasil dari seluruh eksperimen menunjukkan bahwa performa model meningkat secara signifikan terutama dalam hal *recall* dan *F1-Score*, setelah diterapkannya strategi penyesuaian seperti *scale_pos_weight*, *threshold tuning*, dan SMOTE. Temuan ini memperjelas bahwa kontribusi utama penelitian bukan terletak pada kompleksitas cabang pohon XGBoost, melainkan pada keberhasilan strategi dalam mengatasi ketidakseimbangan kelas dengan memaksimalkan fitur *permissions* yang tersedia.

Penelitian ini tidak bertujuan membangun *multi-class tree*, melainkan mengoptimalkan klasifikasi biner malware vs non-malware dengan pendekatan penyesuaian data dan parameter. Ini menjadi poin pembeda dan kekuatan utama dari penelitian, sekaligus jawaban terhadap tantangan dominasi kelas mayoritas dalam domain deteksi malware berbasis machine learning. Seluruh proses dan hasil eksperimen dijelaskan lebih lanjut pada subbab 4.2.1 sampai 4.2.15.

4.2.1 Preprocessing

Tahap ini mencakup serangkaian proses awal yang dilakukan terhadap dataset sebelum dilakukan pemodelan. Tujuannya adalah untuk memastikan bahwa data yang digunakan bersih, relevan, dan sesuai dengan kebutuhan model klasifikasi. Adapun langkah-langkah yang dilakukan sebagai berikut:

1) Data Cleaning

Dataset Android malware yang digunakan merupakan data sekunder dalam bentuk CSV dengan format numerik biner (0 dan 1). Sebelum proses *modeling*, data diperiksa untuk memastikan tidak ada nilai kosong (NaN) dan hanya terdiri dari nilai 0 dan 1. Baris dengan nilai kosong dihapus untuk menjaga kualitas data. Potongan kode ditampilkan sebagai berikut.

Kode :

```
# Konversi seluruh nilai ke numerik; jika ada error parsing,
ubah ke NaN
df = df.apply(pd.to_numeric, errors='coerce')

# Hapus baris yang mengandung nilai NaN
df.dropna(inplace=True)

# Tampilkan ukuran dataset setelah dibersihkan
print("Jumlah data setelah menghapus baris dengan nilai
NaN:", df.shape)
```

2) Binarisasi Nilai

Semua fitur sudah dalam format biner (0 dan 1), sehingga tidak diperlukan proses *encoding* atau normalisasi lanjutan. Dilakukan *valid*asi ulang untuk memastikan tidak ada fitur yang memiliki lebih dari dua nilai unik. Kode program untuk tahap ini dapat dilihat di bawah ini.

Kode :

```
# Memastikan semua nilai agar hanya dalam 0 dan 1
df = df.clip(lower=0, upper=1)

# Cek ulang apakah semua fitur hanya berisi 0 dan 1
print("Jumlah nilai unik di setiap kolom (harusnya 2 untuk
fitur biner):")
print(df.nunique())
```

3) Seleksi Fitur *Permission*

Dari keseluruhan fitur *permission* yang tersedia, dipilih 10 fitur terbaik berdasarkan frekuensi kemunculan dan paling relevan dalam membedakan antara aplikasi malware dan non-malware. Tujuan dari seleksi ini adalah mengurangi kompleksitas fitur dan mempercepat waktu komputasi model. Fitur-fitur yang terpilih antara lain:

1. Network communication : full Internet access (D)
2. Network communication : view network state (S)
3. Phone calls : read phone state and identity (D)
4. Storage : modify/delete USB storage contents (D)
5. Your location : fine (GPS) location (D)
6. *Default* : write contact data (S)
7. System tools : prevent device from sleeping (D)
8. Hardware controls : control vibrator (S)
9. MEDIA_CONTENT_CONTROL
10. READ_EXTERNAL_STORAGE

4) Pemisahan Fitur dan Label

Dataset akhir dipisahkan menjadi dua bagian, yaitu *X* sebagai variabel independen yang terdiri atas 10 fitur *permissions*, dan *y* sebagai variabel target yang diambil dari kolom OUTPUT. Kolom ini berfungsi sebagai label atau *ground truth* yang menunjukkan apakah suatu aplikasi dikategorikan sebagai *non-malware* (0) atau *malware* (1). Proses pemisahan variabel

independen dan target ditunjukkan pada Gambar 4.1, yang merupakan bagian awal dari persiapan data sebelum dilakukan pelatihan model.

```
[ ]  # Pisahkan fitur dan label
     X = df.drop(columns=['OUTPUT'])  # fitur permissions
     y = df['OUTPUT']                 # label malware (1) dan non-malware (0)
```

```
     # Tampilkan ukuran dan distribusi label
     print("Ukuran fitur X:", X.shape)
     print("Distribusi label:")
     print(y.value_counts())
```

```
Ukuran fitur X: (121189, 10)
Distribusi label:
OUTPUT
0    111190
1      9999
Name: count, dtype: int64
```

```
[ ]  print("Daftar kolom fitur (X):")
     for i, col in enumerate(X.columns, 1):
         print(f"{i}. {col}")
```

```
Daftar kolom fitur (X):
1. Network communication : full Internet access (D)
2. Network communication : view network state (S)
3. Phone calls : read phone state and identity (D)
4. Storage : modify/delete USB storage contents modify/delete SD card contents (D)
5. Your location : fine (GPS) location (D)
6. Default : write contact data (S)
7. System tools : prevent device from sleeping (D)
8. Hardware controls : control vibrator (S)
9. MEDIA_CONTENT_CONTROL
10. READ_EXTERNAL_STORAGE
```

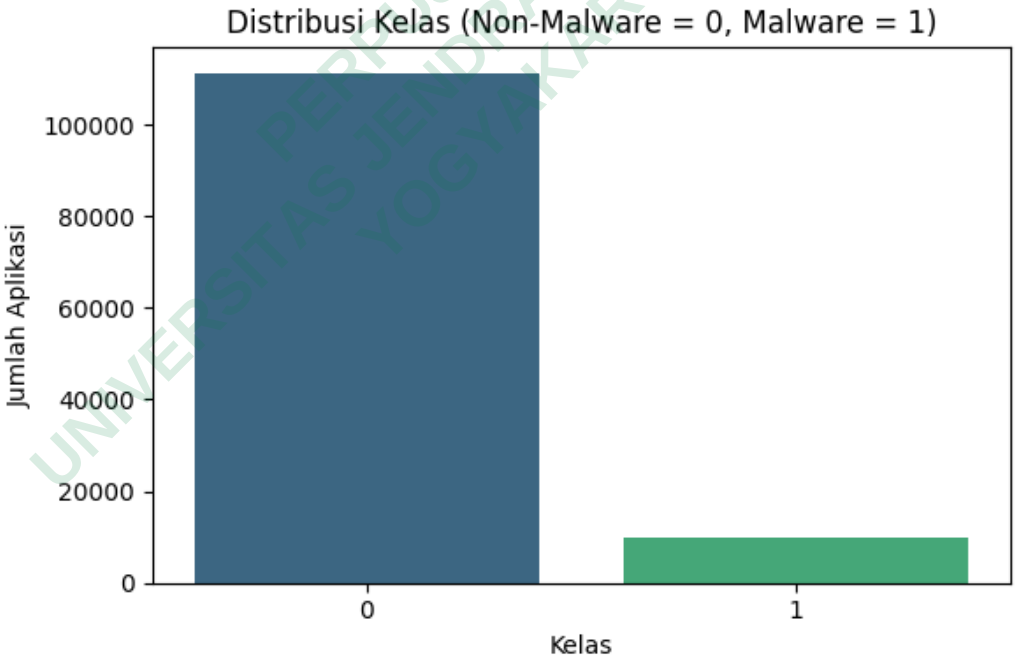

Gambar 4.1 Pemisahan Fitur dan Label

4.2.2 Eksplorasi Distribusi Kelas

Tahap ini bertujuan untuk memahami proporsi jumlah aplikasi malware dan non-malware pada dataset. Informasi ini penting sebagai dasar dalam menangani ketidakseimbangan kelas (*imbalanced class*) pada proses klasifikasi.

1) Visualisasi Distribusi Kelas Awal

   Distribusi jumlah sampel pada masing-masing kelas, yaitu kelas 0 (*non-malware*) dan kelas 1 (*malware*), divisualisasikan dalam Gambar 4.2 menggunakan grafik batang (*bar plot*) untuk menggambarkan perbandingan jumlah antar kelas. Visualisasi tersebut menunjukkan bahwa dataset yang digunakan memiliki ketidakseimbangan kelas yang signifikan, dengan dominasi jumlah *non-malware* yang jauh lebih besar dibandingkan *malware*. Ketimpangan ini menjadi perhatian utama dalam tahapan evaluasi model.



Gambar 4.2 Jumlah Distribusi Kelas

2) Perhitungan Jumlah Label

   Visualisasi dilakukan juga pencetakan nilai numerik dari distribusi kelas menggunakan `value_counts()` pada kolom `OUTPUT`.

Hasil ini digunakan untuk mengidentifikasi secara kuantitatif proporsi ketidakseimbangan. Output kode ditampilkan sebagai berikut.

```
Ukuran fitur X: (121189, 10)
Distribusi label:
OUTPUT
0    111190
1      9999
```

3) Kesimpulan Awal dari Distribusi Kelas

Berdasarkan eksplorasi, ditemukan bahwa kelas mayoritas (non-malware) mendominasi hingga lebih dari 85%. Hal ini dapat berdampak pada rendahnya *recall* jika tidak diatasi, karena model cenderung belajar dari kelas mayoritas (kelas non-malware).

4.2.3 *Baseline* Model

Tahap ini bertujuan untuk memperoleh nilai acuan awal (*baseline*) dari performa model XGBoost sebelum dilakukan penyesuaian parameter atau penerapan teknik penanganan ketidakseimbangan kelas. *Baseline* ini digunakan sebagai tolok ukur awal untuk menilai sejauh mana peningkatan performa yang dicapai melalui eksperimen lanjutan. Langkah-langkah yang ditempuh dalam penerapan *baseline* model dirangkum dalam Tabel 4.1

Tabel 4.1 Tahapan Evaluasi Baseline Model

No	Tahapan	Deskripsi
1.	Prediksi Selalu Non-Malware (Label 0)	Model *baseline* pertama mengklasifikasikan seluruh data sebagai non-malware (label 0), menyesuaikan dengan dominasi kelas mayoritas pada dataset. Digunakan sebagai acuan awal terhadap bias klasifikasi.
2.	Prediksi Acak (Random 0 dan 1)	Model *baseline* kedua melakukan prediksi acak label 0 dan 1 berdasarkan proporsi distribusi asli (misal: 90% non-malware,

		10% malware). Prediksi dilakukan menggunakan np.random.choice().
3.	Evaluasi Baseline	Kedua pendekatan *baseline* dievaluasi menggunakan metrik akurasi, *precision*, *recall*, dan *F1-Score*. Hasil menunjukkan akurasi tinggi namun *recall* rendah, yang mencerminkan bias terhadap kelas mayoritas.

Visualisasi hasil evaluasi model *baseline* ditampilkan pada Gambar 4.3. Berdasarkan hasil yang diperoleh, *precision* dan *recall* pada *baseline* sangat rendah meskipun akurasi terlihat tinggi. Hal ini menandakan bahwa model *baseline* cenderung bias terhadap kelas mayoritas (non-malware), sehingga tidak dapat diandalkan dalam mendeteksi malware.

```
=== Hasil Evaluasi Baseline (Prediksi Acak 0 dan 1) ===
Akurasi  : 0.8320
Precision: 0.0772
Recall   : 0.0945
F1-Score : 0.0850
True Positives (TP): 189
True Negatives (TN): 19978
False Positives (FP): 2260
False Negatives (FN): 1811
```

index	Metrik	Nilai	Keterangan
0	Akurasi	0.832	Tinggi karena didominasi label 0 (non-malware), tapi misleading
1	Precision	0.0772	Dari semua yang diprediksi malware (1), hanya 7.7% yang benar
2	Recall	0.0945	Hanya 9.45% dari semua malware yang benar-benar terdeteksi (sangat rendah)
3	F1-Score	0.085	Kombinasi precision dan recall yang sangat rendah – ini baseline lemah

Gambar 4.3 Hasil Evaluasi *Baseline*

4.2.4 Perhitungan *scale_pos_weight*

Agar model XGBoost dapat mengatasi ketidakseimbangan kelas, dilakukan penghitungan *scale_pos_weight*, yaitu rasio jumlah kelas mayoritas (non-malware) terhadap kelas minoritas (malware) dalam data latih.

1) Perhitungan Rasio

Rasio dihitung dengan membagi jumlah sampel kelas 0 dengan jumlah sampel kelas 1 pada data latih hasil split. Perhitungan *scale_pos_weight*

dilakukan dengan membagi jumlah kelas mayoritas (non-malware) dengan kelas minoritas (malware) pada data latih. Rumus perhitungan ditampilkan pada Gambar 4.4.

Rumus Perhitungan Rasio *scale_pos_weight*:

$$scale\_pos\_weight = \frac{\text{jumlah data kelas 0}}{\text{jumlah data kelas 1}} = \frac{88952}{7999} \approx 11.12$$

Gambar 4.4 Rumus Perhitungan Rasio *scale_pos_weight*

Keterangan:

Nilai ini berarti bahwa satu data malware dianggap setara dengan 11,12 data non-malware. Dengan menggunakan pembobotan ini, diharapkan model tidak hanya terfokus pada kelas mayoritas dan dapat mengenali pola dari kelas minoritas secara lebih efektif. Implementasi kode disajikan pada bagian berikut.

Kode:

```
# Hitung rasio kelas untuk parameter scale_pos_weight
count_nonmalware = y_train.value_counts()[0]
count_malware = y_train.value_counts()[1]
scale_ratio = count_nonmalware / count_malware

# Cetak hasil
print(f"Nilai scale_pos_weight yang disarankan: {scale_ratio:.2f}")

Output : Nilai scale_pos_weight yang disarankan: 11.12
```

2) Interpretasi

Angka 11.12 berarti bahwa satu sampel malware dianggap 11,12 kali lebih penting dibandingkan satu sampel non-malware. Parameter ini digunakan saat melatih model XGBoost untuk meningkatkan sensitivitas terhadap kelas minoritas.

4.2.5 Split Data

Data dibagi menjadi dua bagian: data latih (80%) dan data uji (20%) dengan stratifikasi agar distribusi kelas tetap proporsional. Pembagian ini penting untuk menghindari bias selama pelatihan dan pengujian model. Proses pembagian data secara stratified ini dijelaskan secara sistematis dalam Tabel 4.2 untuk memberikan pemahaman yang lebih jelas mengenai alur dan hasil dari tahap split data.

Tabel 4.2 Pembagian Data dan Distribusi Kelas pada Data Latih dan Uji

No	Tahapan	Deskripsi
1.	Metode Pembagian	Data dibagi menggunakan train_test_split() dari Scikit-learn dengan parameter stratify=y, menjaga proporsi kelas tetap seimbang pada data latih dan uji. Proporsi pembagian adalah 80% data latih dan 20% data uji.
2.	Distribusi Kelas Setelah Split	Distribusi jumlah kelas pada data latih dan uji diperiksa untuk memastikan tidak terjadi distorsi proporsi. Hasil distribusi divisualisasikan dalam Gambar 4.4.
3.	Struktur Data Hasil Split	Setelah split, data latih terdiri dari 96.951 baris dan data uji 24.238 baris, masing-masing dengan 10 fitur *permission* terpilih. Rinciannya ditampilkan dalam Gambar 4.5 dan Gambar 4.6 sebagai dasar pelatihan model. Jumlah ini diperoleh dari: • Jumlah data latih = 88.952 (label 0) + 7.999 (label 1) = 96.951. • Jumlah data uji = 22.238 (label 0) + 2.000 (label 1) = 24.238

Distribusi kelas pada data latih dan uji ditampilkan untuk menunjukkan bahwa proporsinya tetap konsisten. Implementasi kode untuk menampilkan

distribusi ini ditunjukkan pada Gambar 4.5, yang menampilkan jumlah sampel dari masing-masing kelas dalam *training set* dan *testing set*.

```
[33] # Tampilkan hasil split
     print("Distribusi kelas pada data latih:")
     print(y_train.value_counts())

Distribusi kelas pada data latih:
OUTPUT
0    88952
1     7999
Name: count, dtype: int64
```

```
print("\nDistribusi kelas pada data uji:")
print(y_test.value_counts())

Distribusi kelas pada data uji:
OUTPUT
0    22238
1     2000
Name: count, dtype: int64
```

Gambar 4.5 Distribusi Kelas Setelah Split

Setelah proses pembagian dataset dilakukan dengan rasio 80:20 secara stratified, diperoleh dua subset data dengan ukuran ditunjukan sebagai berikut dalam gambar 4.6. Data hasil split ini kemudian digunakan dalam proses pelatihan dan evaluasi model.

```
[35] print("\nUkuran data latih:", X_train.shape)
     print("Ukuran data uji:", X_test.shape)

Ukuran data latih: (96951, 10)
Ukuran data uji: (24238, 10)
```

Gambar 4.6 Struktur Data Hasil Split

4.2.6 Pelatihan Model XGBoost Awal

Langkah ini merupakan pelatihan awal model XGBoost tanpa *tuning hyperparameter*. Model dilatih menggunakan data yang telah dibagi pada tahap sebelumnya dan memanfaatkan nilai *scale_pos_weight* untuk mengatasi

ketidakseimbangan kelas. Tahapan inisialisasi dan pelatihan awal model XGBoost disusun secara sistematis dalam Tabel 4.3 untuk menggambarkan proses awal penerapan algoritma dengan penyesuaian bobot kelas sebagai respons terhadap ketidakseimbangan data.

Tabel 4.3 Tahapan Inisialisasi dan Pelatihan Awal Model XGBoost

No	Tahapan	Deskripsi
1.	Inisialisasi Model	Model dibuat menggunakan XGBClassifier dengan parameter default, kecuali *scale_pos_weight* yang disesuaikan berdasarkan rasio jumlah data non-malware dan malware pada data latih (nilai: 11.12).
2.	Proses Pelatihan	Setelah model diinisialisasi, proses pelatihan model dilatih menggunakan data latih hasil split stratified. Pelatihan dilakukan tanpa tuning parameter tambahan, dengan tujuan memperoleh *baseline* performa awal dari model XGBoost.

Tahapan pelatihan model XGBoost ini dapat dilihat pada Gambar 4.7, yang menampilkan implementasi kode dan struktur pelatihannya.

Kode :

```
# Inisialisasi model dengan parameter awal dan scale_pos_weight
model_awal = XGBClassifier(
    use_label_encoder=False,
    eval_metric='logloss',
     scale_pos_weight=scale_ratio,  # nilai 11.12 dari step
sebelumnya
    random_state=42
)
```

```
# Latih model
model_awal.fit(X_train, y_train)
```

```
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158: UserWarning: [04:00:16
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric='logloss',
              feature_types=None, gamma=None, grow_policy=None,
              importance_type=None, interaction_constraints=None,
              learning_rate=None, max_bin=None, max_cat_threshold=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=None,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, multi_strategy=None, n_estimators=None,
              n_jobs=None, num_parallel_tree=None, random_state=42, ...)
```

Gambar 4.7 Pelatihan Model XGBoost

4.2.7 Evaluasi Awal Model

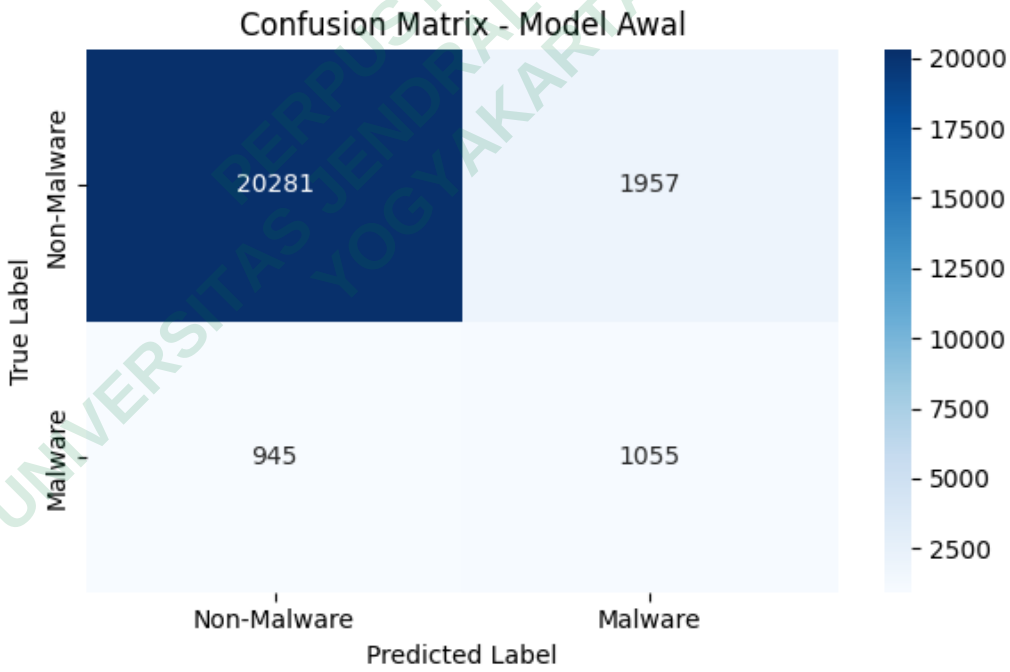
Setelah pelatihan Setelah proses pelatihan awal model XGBoost dilakukan pada data latih, tahap selanjutnya adalah mengevaluasi performa model tersebut terhadap data uji (X_test). Evaluasi awal ini bertujuan untuk mengetahui seberapa baik model dapat mengklasifikasikan aplikasi sebagai malware atau non-malware dalam kondisi standar dengan *scale_pos_weight*. Tahapan evaluasi awal model XGBoost dirangkum dalam Tabel 4.4 guna menggambarkan metode pengujian dan indikator performa yang digunakan dalam mengukur akurasi dan sensitivitas model.

Tabel 4.4 Tahapan Evaluasi Awal Model

No	Tahapan	Deskripsi
1.	Prediksi dan Evaluasi Metrik	Model XGBoost yang telah dilatih digunakan untuk memprediksi label data uji (X_test). Hasil prediksi dibandingkan dengan label asli (y_test) untuk menghitung metrik evaluasi, yaitu akurasi, *precision*, *recall*, dan *F1-Score*. Metrik ini memberikan

		gambaran menyeluruh terhadap kinerja klasifikasi.
2.	Visualisasi Confusion Matrix	Evaluasi numerik ditampilkan confusion matrix untuk memvisualisasikan performa klasifikasi pada masing-masing kelas. Confusion matrix menunjukkan nilai True Positive (TP), *False Positive* (FP), *False Negative* (FN), dan True Negative (TN) secara eksplisit, guna mendukung analisis kesalahan model.

Berikut adalah Visualisasi *confusion matrix* hasil evaluasi model awal ditampilkan pada Gambar 4.8, yang menggambarkan distribusi prediksi benar dan salah dari klasifikasi antara kelas malware dan non-malware.



Gambar 4.8 *Confusion matrix* Model Awal

Output:

```
=== Hasil nilai Confusion matrix ===
True Positive (TP) : 1055
False Positive (FP): 1957
False Negative (FN): 945
True Negative (TN) : 20281


=== Evaluasi Awal Model XGBoost ===
Akurasi  : 0.8803
Precision: 0.3503
Recall   : 0.5275
F1-Score : 0.4210
```

4.2.8 Hyperparameter Tuning

Hyperparameter tuning dilakukan menggunakan metode GridSearchCV dengan tujuan menemukan kombinasi parameter yang mampu memberikan performa terbaik, khususnya dalam menghadapi permasalahan ketidakseimbangan kelas (*imbalanced class*) pada dataset. Pendekatan ini penting karena performa model XGBoost sangat dipengaruhi oleh konfigurasi *hyperparameter* yang digunakan. Dengan *tuning* yang tepat, model diharapkan menjadi lebih sensitif terhadap kelas minoritas, yaitu aplikasi malware, tanpa terlalu mengorbankan *precision*. Tabel 4.5 merangkum pendekatan tuning *hyperparameter* yang digunakan dalam eksperimen ini, dengan fokus pada optimasi parameter yang relevan terhadap ketidakseimbangan kelas.

Tabel 4.5 Tahapan Hyperparameter Tuning pada Model XGBoost

No	Tahapan	Deskripsi
1.	Grid Parameter	Hyperparameter tuning dilakukan menggunakan GridSearchCV, yang menguji kombinasi parameter learning_rate, max_depth, n_estimators, dan *scale_pos_weight*. Keempat parameter ini sangat berpengaruh terhadap kompleksitas model, kemampuan generalisasi, serta sensitivitas terhadap kelas minoritas (malware). GridSearchCV secara sistematis

		mengevaluasi semua kombinasi parameter ini terhadap data latih. Parameter-parameter yang diuji antara lain: • learning_rate: [0.01, 0.1, 0.2] → menentukan seberapa besar kontribusi masing-masing tree dalam proses boosting. • max_depth: [3, 5, 7] → mengatur kedalaman maksimal setiap pohon keputusan, yang memengaruhi kompleksitas dan potensi overfitting. • n_estimators: [100, 200] → jumlah pohon yang digunakan dalam ensemble. • *scale_pos_weight*: [10, 11, 11.12, 12] → parameter kunci dalam eksperimen ini, digunakan untuk mengimbangi ketimpangan jumlah antara kelas malware (minoritas) dan non-malware (mayoritas).
2.	Skor Evaluasi	Penilaian performa dilakukan dengan menggunakan ***F1-Score*** sebagai metrik utama. *F1-Score* dipilih karena mampu mencerminkan *trade-off* antara *precision* dan *recall*, yang sangat penting dalam konteks klasifikasi biner dengan distribusi kelas tidak seimbang. Kombinasi parameter yang menghasilkan *F1-Score* tertinggi dipilih sebagai konfigurasi terbaik dan akan digunakan untuk pelatihan akhir model pada tahap evaluasi berikutnya.

Implementasi kode program untuk proses pencarian kombinasi hyperparameter ditunjukkan pada potongan berikut, yang mencakup

konfigurasi nilai *learning_rate*, *max_depth*, *n_estimators*, dan *scale_pos_weight* dalam GridSearchCV.

Kode:

```
# Parameter grid yang ingin diuji
param_grid = {
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7],
    'n_estimators': [100, 200],
    'scale_pos_weight': [10, 11, 11.12, 12]  # dari hasil Step 11
}

# Jalankan pencarian parameter terbaik
grid.fit(X_train, y_train)
```

Hasil pencarian nilai optimal untuk parameter *scale_pos_weight* melalui proses *hyperparameter tuning* ditunjukkan pada Gambar 4.9, yang memvisualisasikan performa model terhadap berbagai kombinasi nilai yang diuji.

Output:

Gambar 4.9 Pencarian Metrik terbaik

Potongan kode berikut digunakan untuk mengekstrak dan menampilkan kombinasi hyperparameter dengan nilai *F1-Score* tertinggi berdasarkan hasil GridSearchCV.

Kode:

```
# Tampilkan hasil
print("=== Parameter Terbaik ===")
print(grid.best_params_)

output  :
       === Parameter Terbaik ===
{'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 100,
'scale_pos_weight': 10}

print("\n=== Skor F1 Tertinggi ===")
print(f"{grid.best_score_:.4f}")

output  :
=== Skor F1 Tertinggi ===
0.6122
```


4.2.9 Evaluasi Model Terbaik

Setelah proses *tuning hyperparameter* dilakukan, model XGBoost terbaik yang diperoleh dari kombinasi parameter optimal dilatih ulang menggunakan data latih. Model ini kemudian dievaluasi terhadap data uji (x_test) menggunakan metrik yang sama seperti pada tahap evaluasi awal. Tujuan dari tahap ini adalah untuk melihat sejauh mana peningkatan performa yang diperoleh dari proses *tuning*, terutama dalam konteks penanganan ketidakseimbangan kelas. Model terbaik dari hasil *tuning* menunjukkan performa yang lebih baik dibandingkan model awal. Berikut adalah hasil evaluasi model:

a) Akurasi: 0.9517
b) *Precision*: 0.9905
c) *Recall*: 0.4190
d) *F1-Score*: 0.5889

Keterangan:

- Nilai *precision* yang sangat tinggi menunjukkan bahwa sebagian besar prediksi malware oleh model benar. Namun demikian, nilai *recall* yang masih berada di angka 0.4190 menunjukkan bahwa masih banyak kasus malware yang tidak terdeteksi (*False Negative*). Hal ini wajar mengingat karakteristik data yang sangat tidak seimbang.
- Meski *recall* belum optimal, *F1-Score* meningkat dari model awal, yang menunjukkan adanya perbaikan dalam keseimbangan *precision* dan *recall* setelah *tuning* dilakukan. Hal ini menegaskan bahwa penyesuaian parameter seperti *scale_pos_weight* berpengaruh dalam membantu model lebih peka terhadap kelas minoritas (malware).

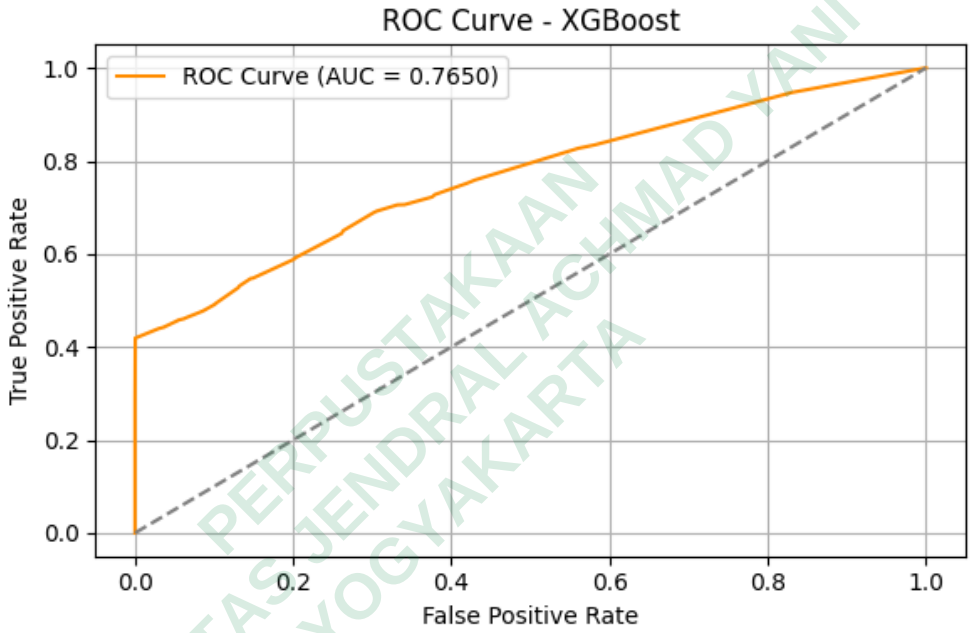
4.2.10 Evaluasi Lanjutan: ROC, AUC, dan Nilai TP-FP-FN-TN

Setelah model terbaik diperoleh melalui proses *tuning*, dilakukan evaluasi lanjutan untuk memberikan gambaran lebih mendalam mengenai kinerja model dalam membedakan antara dua kelas (malware dan non-malware). Evaluasi lanjutan ini melengkapi metrik dasar sebelumnya dengan menggunakan visualisasi ROC curve, perhitungan Area Under Curve (AUC), serta analisis numerik nilai *True Positive*, *True Negative*, *False Positive*, dan *False Negative*.

a) ROC Curve dan AUC Score

Receiver Operating Characteristic (ROC) Curve digunakan untuk menggambarkan kemampuan model dalam membedakan dua kelas pada berbagai ambang prediksi (*threshold*). Kurva ini memperlihatkan hubungan antara *True Positive Rate* (TPR) dan *False Positive Rate* (FPR). Semakin dekat kurva ke pojok kiri atas, semakin baik performa model.

Pada eksperimen ini, ROC curve menunjukkan bahwa model memiliki kemampuan diskriminasi yang cukup baik meskipun data sangat tidak seimbang. Nilai *Area Under Curve* (AUC) yang diperoleh adalah 0.7650, yang menandakan bahwa model memiliki akurasi pemisahan kelas malware dan non-malware sebesar 76,5%. Gambar 4.10 merupakan representasi grafis dari Kurva ROC.



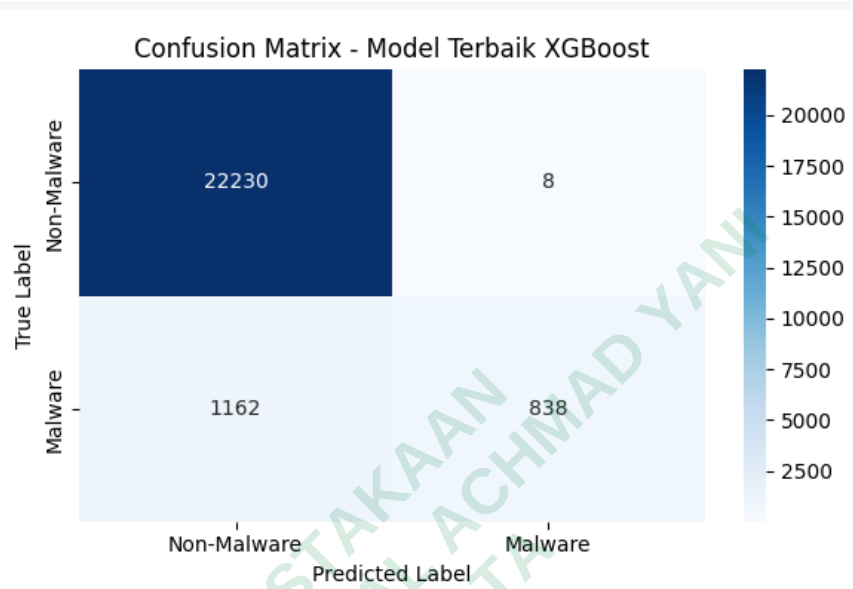
Gambar 4.10 Kurva ROC

b) Perhitungan TP, TN, FP, FN

Nilai True Positive, True Negative, *False Positive*, dan *False Negative* dihitung menggunakan fungsi `cm.ravel()`. Untuk melengkapi interpretasi *confusion matrix*, dilakukan perhitungan eksplisit terhadap nilai:

- *True Positive* (TP): 838
- *False Positive* (FP): 8
- *False Negative* (FN): 1162
- *True Negative* (TN): 22230

Gambar 4.11 menyajikan hasil perhitungan *confusion matrix* yang digunakan untuk mengevaluasi performa model berdasarkan jumlah prediksi benar dan salah pada masing-masing kelas.



Gambar 4.11 *Confusion matrix* Model Terbaik XGBoost

Keterangan:

- TP = 838: artinya sebanyak 838 malware berhasil dideteksi dengan benar.
- FN = 1162: masih banyak malware yang tidak terdeteksi (*False Negative* tinggi), ini berarti *recall* masih menjadi kelemahan utama.
- FP sangat kecil (8): artinya hampir tidak ada aplikasi normal yang salah terdeteksi sebagai malware.
- AUC = 0.7650 termasuk cukup baik (di atas 0.7), artinya model memiliki kemampuan membedakan antara kelas malware dan non-malware secara keseluruhan.

Hasil ini menunjukkan bahwa model sangat akurat dalam mengenali non-malware (TN sangat tinggi), namun masih cukup banyak malware yang tidak terdeteksi (FN tinggi). Meskipun nilai TP cukup besar, dominasi FN

tetap menunjukkan bahwa model masih kesulitan mengenali sebagian malware.

4.2.11 Pengambilan Subset Data

Untuk menganalisis pengaruh jumlah data terhadap performa model serta waktu komputasi, dilakukan eksperimen pada tiga subset data berukuran 500, 1000, dan 2000 sampel. Masing-masing subset dibagi, dilatih, dan dievaluasi secara terpisah. Tahapan eksperimen untuk subset data berukuran 500 dijabarkan secara sistematis dalam Tabel 4.6 guna menunjukkan alur evaluasi model dari proses pengambilan data hingga analisis performa klasifikasi.

a) Subset Data 500

Tabel 4.6 Tahapan Evaluasi Model pada Subset Data 500

No	Tahapan	Deskripsi
1.	Pengambilan Subset Data	Diambil sebanyak 500 data dari dataset utama secara Stratified berdasarkan label OUTPUT agar proporsi kelas tetap seimbang antara kelas malware dan non-malware tetap konsisten dengan distribusi asli.
2.	Pembagian Data Latih dan Uji	Dilakukan pembagian menjadi data latih (80%) dan data uji (20%) menggunakan *train_test_split* dengan *parameter stratify* agar proporsi kelas tetap seimbang di kedua bagian. Kode dan hasil distribusi subset ditampilkan pada Gambar 4.11. Visualisasi distribusi subset ditampilkan pada Gambar 4.12.
3.	Perhitungan *scale_pos_weight*	Nilai *scale_pos_weight* dihitung berdasarkan rasio jumlah data non-malware dan malware pada data latih subset 500. Perhitungan ini dilakukan secara spesifik untuk tiap subset karena distribusi absolut

		berbeda dari data utama. Hasil perhitungan digunakan dalam pelatihan model untuk mengarahkan perhatian model lebih kuat ke kelas minoritas. Nilai yang diperoleh: *scale_pos_weight* = 11.12.
4.	Pelatihan dan Evaluasi Model	Model XGBoost dilatih pada subset 500 menggunakan parameter terbaik yang telah diperoleh pada proses tuning sebelumnya. Evaluasi dilakukan terhadap data uji menggunakan metrik klasifikasi akurasi, *precision*, *recall*, dan *F1-Score*. Hasil evaluasi: • Akurasi: 0.8800 • *Precision*: 0.3750 • *Recall*: 0.7500 • *F1-Score*: 0.5000 • Waktu Komputasi: 0.03 detik Hasil evaluasi menunjukkan bahwa *recall* cukup tinggi yaitu 0.7500, namun *precision* masih rendah (0.3750), sehingga nilai *F1-Score* berada pada level menengah (0.5000). Visualisasi confusion matrix juga menunjukkan bahwa model mampu mengenali sebagian besar malware meskipun dengan data yang sangat terbatas., waktu komputasi relatif cepat karena ukuran data yang kecil. Representasi hasil prediksi dalam bentuk confusion matrix disajikan pada Tabel 4.6.

Nilai *scale_pos_weight* dihitung berdasarkan rasio jumlah data non-malware dan malware pada data latih subset 500. Perhitungan ini dilakukan secara spesifik untuk tiap subset karena distribusi absolut berbeda dari data utama. Hasil perhitungan digunakan dalam pelatihan model untuk

mengarahkan perhatian model lebih kuat ke kelas minoritas. Potongan kode ditampilkan sebagai berikut.

Kode:

```
from xgboost import XGBClassifier
# Hitung scale_pos_weight
nonmalware_500 = y_train_500.value_counts()[0]
malware_500 = y_train_500.value_counts()[1]
scale_pos_weight_500 = nonmalware_500 / malware_500
print(f"scale_pos_weight: {scale_pos_weight_500:.2f}")

Output : scale_pos_weight: 11.12
```

Hasil evaluasi menunjukkan bahwa *recall* cukup tinggi yaitu 0.7500, namun *precision* masih rendah (0.3750), sehingga nilai *F1-Score* berada pada level menengah (0.5000).

Visualisasi *confusion matrix* juga menunjukkan bahwa model mampu mengenali sebagian besar malware meskipun dengan data yang sangat terbatas. Waktu komputasi relatif cepat karena ukuran data yang kecil. Representasi hasil prediksi dalam bentuk *confusion matrix* disajikan pada Tabel 4.7.

Tabel 4.7 *Confusion matrix* Subset Data 500

Category	**Predicted: Non-Malware**	**Predicted: Malware**
Actual: Non-Malware	82	10
Actual: Malware	2	6

b) Subset Data 1000

Urutan pelaksanaan dan hasil evaluasi model pada subset 1000 dijabarkan dalam Tabel 4.8 untuk memperlihatkan dampak bertambahnya ukuran data terhadap performa klasifikasi dan efisiensi model.

Tabel 4.8 Tahapan Evaluasi Model pada Subset Data 1000

No	Tahapan	Deskripsi
1.	Pengambilan Subset Data	Diambil 1000 data secara stratified. Pemilihan ini mempertahankan proporsi kelas antara malware dan non-malware seperti distribusi aslinya. Penggunaan subset berukuran menengah ini bertujuan untuk mengevaluasi apakah peningkatan jumlah data memberikan pengaruh signifikan terhadap kinerja model, terutama dalam konteks ketidakseimbangan kelas.
2.	Pembagian Data Latih dan Uji	Subset 1000 dibagi menjadi data latih dan uji (80:20 *stratified*). Proporsi ini tetap menjaga distribusi kelas agar tetap representatif pada kedua bagian. Kode dapat dilihat pada Gambar 4.12. Visualisasi distribusi subset ditampilkan pada Gambar 4.13.
3.	Perhitungan *scale_pos_weight*	Dihitung berdasarkan rasio kelas pada y_train_1000. Nilai *scale_pos_weight* dihitung berdasarkan rasio jumlah non-malware terhadap malware pada data latih. Rasio yang diperoleh digunakan dalam parameter pelatihan model XGBoost, agar model tetap memperhatikan kelas minoritas (malware) secara proporsional. Nilai rasio untuk subset ini mendekati nilai pada data penuh, yaitu 11.11. Kode perhitungan ditunjukkan dalam potongan kode eksperimen.
4.	Pelatihan dan Evaluasi Model	Model dilatih dan dievaluasi pada subset 1000. Inisialisasi model XGBoost menggunakan *scale_pos_weight* hasil perhitungan. Evaluasi dilakukan terhadap data uji dengan metrik:

		• Akurasi : 0.8750 • *Precision* : 0.3333 • *Recall* : 0.5625 • *F1-Score* : 0.4186 • Waktu Pelatihan : 0.05 detik Hasil evaluasi pada subset 1000 menunjukkan *recall* sebesar 0.5625, lebih rendah dari subset 500, namun *precision* meningkat menjadi 0.3333. Hal ini menunjukkan adanya *trade-off:* model lebih selektif, namun tetap menangkap sebagian malware. Nilai *F1-Score* berada di angka 0.4186 dan waktu komputasi meningkat. Visualisasi dalam bentuk Tabel 4.8 untuk *confusion matrix* subset data 1000.

Dihitung berdasarkan rasio kelas pada `y_train_1000`. Nilai *`scale_pos_weight`* dihitung berdasarkan rasio jumlah non-malware terhadap malware pada data latih. Rasio yang diperoleh digunakan dalam parameter pelatihan model XGBoost, agar model tetap memperhatikan kelas minoritas (malware) secara proporsional. Nilai rasio untuk subset ini mendekati nilai pada data penuh, yaitu sekitar **11.11**. Berikut adalah kode yang digunakan untuk proses ini.

Kode:

```
value_counts = y_train_1000.value_counts()
if 1 in value_counts and value_counts[1] != 0:
count_nonmalware = value_counts[0]
count_malware = value_counts[1]
scale_pos_weight_1000 = count_nonmalware / count_malware
print(f"Nilai scale_pos_weight untuk subset 1000 data:
{scale_pos_weight_1000:.2f}")

Output : Nilai scale_pos_weight untuk subset 1000 data:
11.11
```

Hasil evaluasi pada subset 1000 menunjukkan *recall* sebesar 0.5625, lebih rendah dari subset 500, namun *precision* meningkat menjadi 0.3333. Hal ini menunjukkan adanya trade-off: model lebih selektif, namun tetap menangkap sebagian malware. Nilai *F1-Score* berada di angka 0.4186 dan waktu komputasi meningkat. Visualisasi dalam bentuk Tabel 4.9 untuk *Confusion matrix* Subset Data 1000.

Tabel 4.9 *Confusion matrix* Subset Data 1000

Category	**Predicted: Non-Malware**	**Predicted: Malware**
Actual: Non-Malware	166	18
Actual: Malware	7	9

c) Subset Data 2000

Urutan eksperimen pada subset 2000 dirangkum dalam Tabel 4.10 untuk menunjukkan performa model ketika dihadapkan pada data berukuran lebih besar.

Tabel 4.10 Tahapan Evaluasi Model pada Subset Data 1000

No	**Tahapan**	**Deskripsi**
1.	Pengambilan Subset Data	Untuk eksperimen subset terakhir, diambil sebanyak 2000 sampel dari dataset utama dengan metode Stratified sampling berdasarkan label OUTPUT. Ukuran ini merupakan yang terbesar dari tiga subset, bertujuan untuk menilai apakah semakin banyak data berdampak signifikan terhadap kestabilan prediksi dan peningkatan performa *recall*.
2.	Pembagian Data Latih dan Uji	Proses pembagian data subset 2000 ke dalam data latih dan data uji dilakukan menggunakan teknik *stratified* sampling dengan rasio 80:20, untuk menjaga proporsi kelas tetap seimbang. Implementasi kode

		untuk proses ini ditampilkan pada Gambar 4.13. Visualisasi distribusi subset disajikan pada Gambar 4.14.
3.	Perhitungan *scale_pos_weight*	Rasio dihitung untuk y_train_2000. Rasio antara kelas mayoritas dan minoritas dihitung dari data latih dan digunakan sebagai *scale_pos_weight* pada model. Potongan kode ditampilkan sebagai berikut: Nilai *scale_pos_weight* untuk subset 2000 data adalah **11.12**. Nilai rasio yang dihasilkan serupa dengan dua subset sebelumnya, menunjukkan bahwa distribusi relatif tetap seimbang meskipun ukuran data bertambah.
4.	Pelatihan dan Evaluasi Model	Pelatihan model dilakukan menggunakan parameter hasil tuning. Hasil evaluasi: • Akurasi: 0.8575 • *Precision*: 0.2857 • *Recall*: 0.4848 • *F1-Score*: 0.3596 • Waktu Komputasi: 8.02 detik Evaluasi menunjukkan bahwa dengan jumlah data yang lebih besar, *recall* mengalami penurunan ke angka 0.4848, sementara *precision* juga turun menjadi 0.2857, sehingga *F1-Score* berada pada angka 0.3596. Waktu komputasi mengalami peningkatan. Hasil ini mengindikasikan bahwa penambahan data tidak selalu berdampak positif jika tidak diikuti teknik penanganan imbalance yang lebih agresif. Rincian hasil klasifikasi ditampilkan dalam Tabel 4.9 untuk confusion matrix subset data 2000.

Rasio dihitung untuk `y_train_2000`. Rasio antara kelas mayoritas dan minoritas dihitung dari data latih dan digunakan sebagai *scale_pos_weight* pada model. Potongan kode ditampilkan sebagai berikut.

Kode :

```
# Step 17-c: Hitung scale_pos_weight
count_0 = y_train_2000.value_counts()[0]
count_1 = y_train_2000.value_counts()[1]
scale_pos_weight_2000 = count_0 / count_1

print(f"Nilai scale_pos_weight untuk subset 2000 data:
{scale_pos_weight_2000:.2f}")
```

Output :

```
Nilai scale_pos_weight untuk subset 2000 data: 11.12
```

Nilai rasio yang dihasilkan adalah **11.12**, serupa dengan dua subset sebelumnya, menunjukkan bahwa distribusi relatif tetap seimbang meskipun ukuran data bertambah.

Evaluasi menunjukkan bahwa dengan jumlah data yang lebih besar, *recall* mengalami penurunan ke angka 0.4848, sementara *precision* juga turun menjadi 0.2857, sehingga *F1-Score* berada pada angka 0.3596. Waktu komputasi mengalami peningkatan. Hasil ini mengindikasikan bahwa penambahan data tidak selalu berdampak positif jika tidak diikuti teknik penanganan imbalance yang lebih agresif. Rincian hasil klasifikasi ditampilkan dalam Tabel 4.11 untuk *Confusion Matrix* subset data 2000.

Tabel 4.11 *Confusion matrix* Subset Data 2000

Category	**Predicted: Non-Malware**	**Predicted: Malware**
Actual: Non-Malware	327	40
Actual: Malware	17	16

4.2.12 Rekap dan Visualisasi Evaluasi

Pada bagian ini dilakukan rekap hasil evaluasi dari ketiga subset data, baik dari segi performa maupun waktu komputasi. Tahapan ini bertujuan untuk memperoleh gambaran yang lebih utuh mengenai pengaruh ukuran data terhadap performa model serta kompleksitas waktu komputasi. Rekapitulasi dan visualisasi evaluasi dari ketiga subset disusun secara sistematis dalam Tabel 4.12.

Tabel 4.12 Rekapitulasi dan visualisasi evaluasi subset data

No	Tahapan	Deskripsi
1.	Tabel Evaluasi	Pelatihan model dilakukan menggunakan parameter hasil tuning. Hasil evaluasi: • Akurasi: 0.8575 • *Precision*: 0.2857 • *Recall*: 0.4848 • *F1-Score*: 0.3596 • Waktu Komputasi: 8.02 detik Evaluasi menunjukkan bahwa dengan jumlah data yang lebih besar, *recall* mengalami penurunan ke angka 0.4848, sementara *precision* juga turun menjadi 0.2857, sehingga *F1-Score* berada pada angka 0.3596. Waktu komputasi mengalami peningkatan. Hasil ini mengindikasikan bahwa penambahan data tidak selalu berdampak positif jika tidak diikuti teknik penanganan imbalance yang lebih agresif. Rincian hasil klasifikasi ditampilkan dalam Tabel 4.9 untuk confusion matrix subset data 2000. **Keterangan:** • Accuracy: persentase klasifikasi benar dari seluruh prediksi.*Precision*: proporsi prediksi malware yang benar dari seluruh prediksi positif.

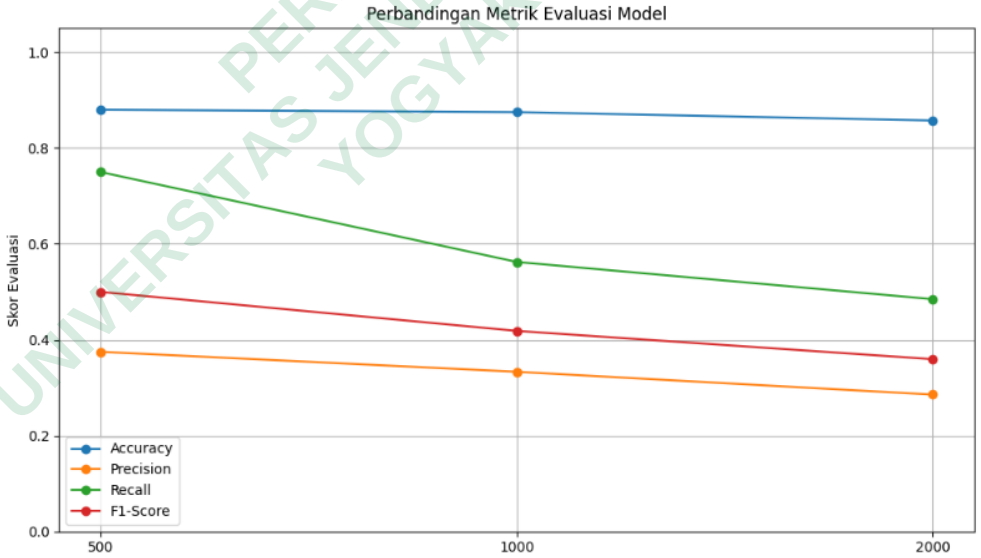
		• *Recall*: kemampuan model dalam menangkap seluruh malware yang ada. • *F1-Score*: gabungan antara *precision* dan *recall*. • Waktu Komputasi: waktu yang dibutuhkan model untuk menyelesaikan proses pelatihan dan prediksi. Perbandingan metrik evaluasi antara masing-masing subset data divisualisasikan dalam Gambar 4.14. Grafik ini memudahkan dalam mengidentifikasi kecenderungan *trade-off* antara sensitivitas dan ketepatan model seiring bertambahnya ukuran data.
2.	Visualisasi Komparatif	Dua jenis visualisasi dibuat untuk memperjelas pola dari hasil evaluasi: • Grafik *F1-Score* dan *Recall* per Subset: Plot garis memperlihatkan perubahan nilai *F1-Score* dan *recall* terhadap ukuran subset. Visualisasi ini membantu mengamati *trade-off* antara *recall* dan jumlah data, serta kapan model mencapai keseimbangan terbaik antara deteksi malware dan kesalahan prediksi. Dari grafik ini terlihat bahwa *recall* relatif menurun seiring bertambahnya data, namun stabilitas model meningkat (Gambar 4.15). • Grafik Kompleksitas vs Waktu Komputasi: Menampilkan hubungan antara jumlah sampel (kompleksitas data) dengan waktu komputasi dalam satuan detik. Plot spline menunjukkan bahwa waktu meningkat non-linear meski jumlah data bertambah linear (Gambar 4.16).

Ringkasan dari seluruh hasil evaluasi pada masing-masing subset ditampilkan pada Tabel 4.13.

Tabel 4.13 Ringkasan Evaluasi Subset

No	Subset	*Accuracy*	*Precision*	*Recall*	*F1-Score*	Waktu (detik)
1	500	0.8800	0.3750	0.7500	0.5000	0.03
2	1000	0.8750	0.3333	0.5625	0.4186	0.59
3	2000	0.8575	0.2857	0.4848	0.3596	8.02

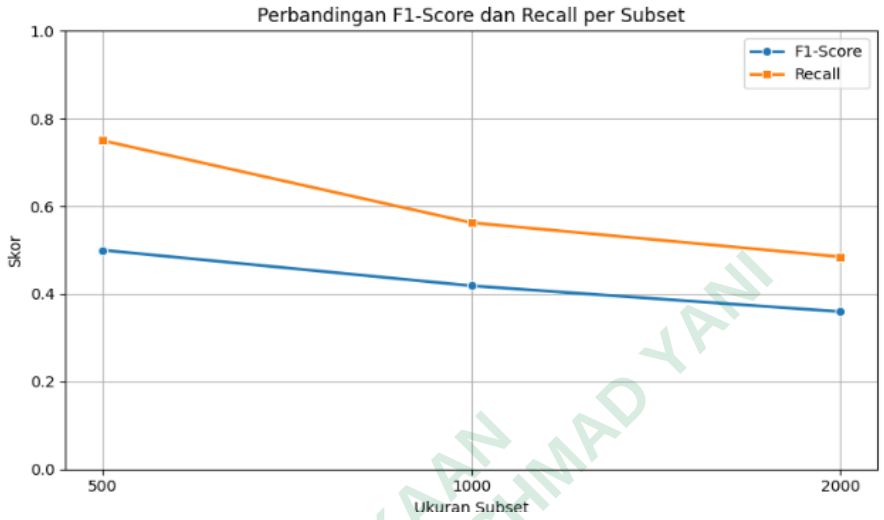
Perbandingan metrik evaluasi antara masing-masing subset data divisualisasikan dalam Gambar 4.12. Visualisasi ini mencakup metrik *accuracy*, *precision*, *recall*, dan *F1-Score*, yang merepresentasikan variasi performa model saat jumlah data berubah dari 500 hingga 2000 sampel. Grafik ini memudahkan dalam mengidentifikasi kecenderungan *trade-off* antara sensitivitas dan ketepatan model seiring bertambahnya ukuran data.



Gambar 4.12 Perbandingan metrik evaluasi masing-masing subset data

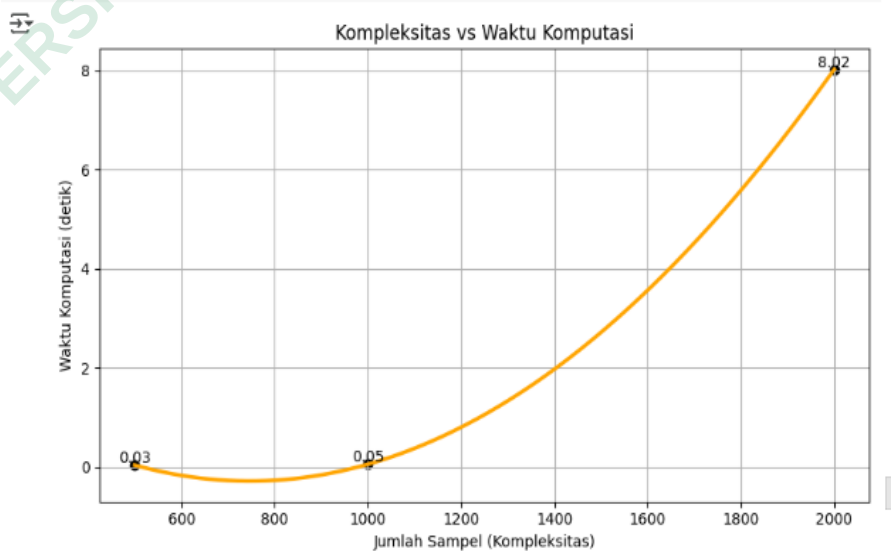
Dibuat plot garis yang memperlihatkan perubahan nilai *F1-Score* dan *recall* terhadap ukuran subset. Gambar 4.13 memperlihatkan plot garis F1-Score dan recall terhadap ukuran subset. Visualisasi ini menunjukkan trade-off

antara recall dan jumlah data, di mana recall cenderung menurun seiring bertambahnya data, namun stabilitas model meningkat.



Gambar 4.13 Perbandingan *F1-Score* dan *Recall* per Subset

Grafik Kompleksitas vs Waktu Komputasi menunjukkan hubungan antara jumlah sampel dan waktu komputasi. Dengan interpolasi spline, grafik memperlihatkan tren non-linear, di mana waktu komputasi meningkat lebih cepat dibanding pertambahan jumlah data. Grafik ini ditampilkan pada Gambar 4.14 Grafik Kompleksitas vs Waktu Komputasi.



Gambar 4.14 Grafik Kompleksitas vs Waktu Komputasi

4.2.13 Threshold Tuning

Threshold tuning dilakukan untuk menguji apakah menyesuaikan ambang prediksi (> 0.5) dapat meningkatkan *recall* dan keseimbangan model. Model klasifikasi seperti XGBoost menggunakan ambang 0.5 untuk memutuskan apakah suatu sampel diklasifikasikan sebagai kelas positif (malware). Namun, dalam kondisi kelas minoritas (malware) yang sangat sedikit, ambang *default* ini seringkali menyebabkan model enggan untuk memberikan prediksi kelas 1, sehingga *recall* menjadi rendah.

Tahapan *threshold* tuning, mulai dari pengujian berbagai nilai ambang hingga evaluasi performa model pada masing-masing *threshold*, disusun secara sistematis dalam Tabel 4.14 untuk memberikan gambaran menyeluruh terhadap pengaruh penyesuaian ambang terhadap sensitivitas dan keseimbangan klasifikasi model.

Tabel 4.14 Tahapan *Threshold Tuning*

No	Tahapan	Deskripsi
1.	Penyesuaian *Threshold*	*Threshold tuning* dilakukan untuk menguji penurunan nilai ambang dapat meningkatkan kemampuan model dalam mengenali malware (meningkatkan *recall*).
2.	Pengujian Beberapa *Threshold*	Model yang sudah dilatih pada seluruh data diuji menggunakan *threshold* prediksi 0.3, 0.4, 0.5, 0.6, dan 0.7.
3.	Evaluasi per *Threshold*	Setiap *threshold* diuji dengan metrik akurasi, *precision*, *recall*, dan *F1-Score*
4.	Interpretasi Hasil	Dari hasil evaluasi, ditemukan bahwa Penurunan *threshold* ke 0.4 terbukti meningkatkan *recall* secara signifikan dibanding threshold default (0.5), dengan peningkatan *F1-Score* meskipun *precision* menurun dan *threshold* 0.3 membuat model terlalu sensitif dan menghasilkan banyak *False Positive*.

Dilakukan pengujian dan prediksi manual berdasarkan probabilitas dan nilai ambang batas. Model akan lebih sensitif terhadap kelas positif ketika ambang diturunkan (ke 0.3), sehingga lebih banyak malware dapat terdeteksi. Potongan kode ditampilkan sebagai berikut.

Kode:

```
# Daftar threshold   yang ingin diuji
thresholds = [0.3, 0.4, 0.5, 0.6, 0.7]
pos_weight untuk subset 2000 data: 11.12
for t in thresholds:
y_pred_t = (y_proba >= t).astype(int)  # ubah jadi 1 jika prob >= threshold

acc = accuracy_score(y_test, y_pred_t)
prec = precision_score(y_test, y_pred_t, zero_division=0)
rec = recall_score(y_test, y_pred_t)
f1 = f1_score(y_test, y_pred_t)

hasil_threshold.append({
'Threshold': t,
'Accuracy': acc,
'Precision': prec,
'Recall': rec,
'F1-Score': f1
```

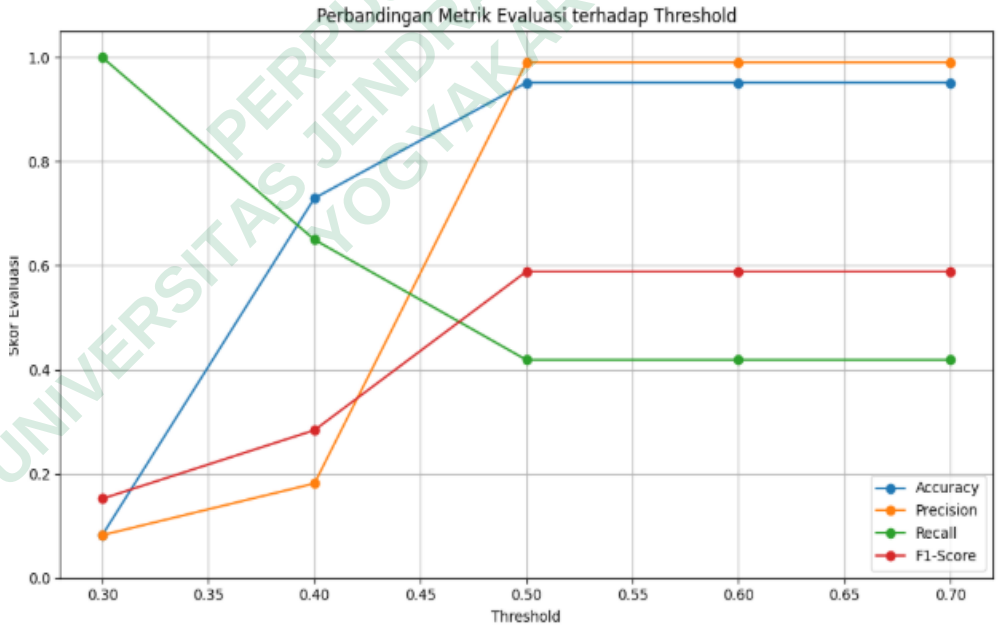

Hasil evaluasi dari proses *threshold tuning* disajikan dalam Tabel 4.15. Tabel ini memuat perbandingan nilai *accuracy*, *precision*, *recall*, dan *F1-Score* pada berbagai ambang prediksi yang diuji (mulai dari 0.3 hingga 0.7), untuk mengamati dampaknya terhadap keseimbangan kinerja model, khususnya dalam mendeteksi kelas malware sebagai minoritas.

Tabel 4.15 Hasil Evaluasi untuk Berbagai *Threshold*

No.	*Threshold*	*Accuracy*	*Precision*	*Recall*	*F1-Score*
1	0.3	0.082928	0.082549	1.000	0.152509
2	0.4	0.729681	0.181767	0.650	0.284091
3	0.5	0.951729	0.990544	0.419	0.588897
4	0.6	0.951729	0.990544	0.419	0.588897
5	0.7	0.951729	0.990544	0.419	0.588897

Dari hasil evaluasi, ditemukan bahwa penurunan *threshold* ke angka 0.4 mampu meningkatkan nilai *recall* secara signifikan dibanding *threshold default* 0.5. Meski *precision* menurun sebagai konsekuensi, *F1-Score* meningkat, yang menunjukkan adanya perbaikan dalam keseimbangan antara deteksi malware dan kesalahan klasifikasi. Sebaliknya, *threshold* terlalu rendah (0.3) cenderung menyebabkan model over-sensitive dan menghasilkan banyak *False Positive*.

Grafik Perbandingan Metrik Evaluasi visual antara nilai *accuracy*, *precision*, *recall*, dan *F1-Score* terhadap variasi nilai ambang prediksi ditampilkan pada Gambar 4.17. Grafik ini memberikan gambaran mengenai *trade-off* antar metrik yang muncul akibat perubahan *threshold*, khususnya terhadap sensitivitas model dalam mendeteksi kelas malware.



Gambar 4.17 Perbandingan Metrik Evaluasi terhadap *Threshold*

4.2.14 Penerapan SMOTE

Penerapan SMOTE dilakukan untuk mengatasi ketidakseimbangan kelas secara langsung dengan menambahkan data sintetis untuk kelas minoritas. Sebelumnya, model telah mencoba mengatasi ketimpangan jumlah sampel melalui parameter *scale_pos_weight*, pada tahap ini digunakan strategi lain yakni dengan menambahkan data sintetis pada kelas malware, sehingga jumlah data antara dua kelas menjadi seimbang. SMOTE bekerja dengan cara menghasilkan sampel baru dari kelas minoritas berdasarkan interpolasi nilai-nilai yang ada. Tahapan implementasi SMOTE untuk menyeimbangkan data latih, pelatihan ulang model, serta evaluasi performa secara menyeluruh dirangkum secara sistematis dalam Tabel 4.16 sebagai dasar analisis efektivitas *oversampling* dalam konteks klasifikasi malware.

Tabel 4.16 Tahapan dan Evaluasi Model dengan SMOTE

No	Tahapan	Deskripsi
1.	*Oversampling* dengan SMOTE	SMOTE hanya diterapkan pada data latih untuk menjaga keaslian data uji. Sebelum SMOTE, data latih sangat tidak seimbang (7.999 malware vs 88.952 non-malware). Setelah SMOTE, jumlah kelas minoritas ditambahkan hingga seimbang, yaitu 88.952 untuk masing-masing kelas. Hal ini menghasilkan distribusi label 1:1, memungkinkan model belajar dari representasi kelas yang setara.
2.	Pelatihan Model pada Data SMOTE	Model XGBoost dilatih ulang menggunakan data hasil SMOTE dengan hyperparameter terbaik hasil tuning sebelumnya (learning_rate=0.01, max_depth=3, n_estimators=100, *scale_pos_weight*=1). Meskipun distribusi kelas telah seimbang secara sempurna, konfigurasi *scale_pos_weight* diatur ke 1 untuk mencerminkan tidak adanya dominasi kelas

		dalam data latih. Waktu komputasi tercatat lebih tinggi dibanding sebelumnya karena ukuran data latih meningkat dua kali lipat.
3.	Evaluasi Model	Evaluasi dilakukan pada data uji menggunakan metrik Accuracy (0.9517), *Precision* (0.9905), *Recall* (0.4190), *F1-Score* (0.5889), dan AUC Score (0.7651). Hasil ini menunjukkan bahwa *recall* tidak mengalami peningkatan dibanding model sebelumnya, namun *precision* dan AUC tetap tinggi. Nilai TP = 838, FP = 8, FN = 1162, TN = 22230. Kurva ROC hasil evaluasi divisualisasikan pada Gambar 4.18. Meski distribusi kelas telah seimbang, efektivitas deteksi malware tidak mengalami perbaikan signifikan dibanding metode *threshold tuning*.

SMOTE hanya diterapkan pada data latih bukan pada keseluruhan data. Hal ini penting untuk menjaga keaslian distribusi data uji, agar proses evaluasi tetap mencerminkan performa model pada data nyata (tidak tersintesis). Potongan kode ditampilkan sebagai berikut.

Kode:

```
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Tampilkan distribusi kelas setelah SMOTE
print("Distribusi kelas setelah SMOTE:")
print(y_train_smote.value_counts())
```

Output:

```
Distribusi kelas setelah SMOTE:
OUTPUT
0    88952
1    88952
```

Model XGBoost dilatih ulang menggunakan data latih hasil SMOTE, dengan tetap menggunakan parameter terbaik dari hasil *hyperparameter tuning* sebelumnya (misalnya kombinasi learning_rate, max_depth, n_estimators, dan *scale_pos_weight*). Kode pelatihan model ditampilkan sebagai berikut.

Kode:

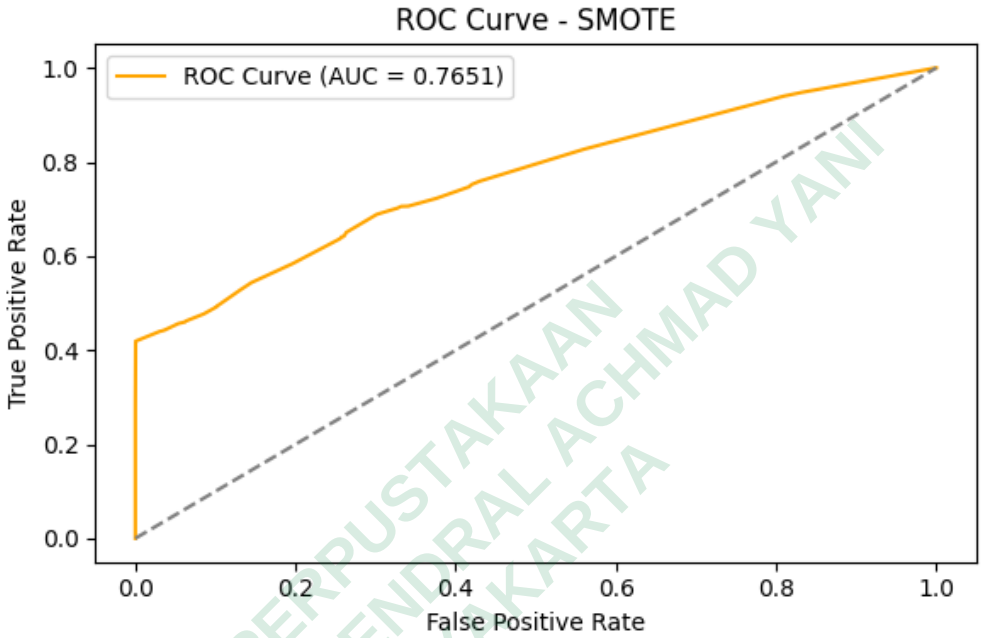
```
# Tahap 3: Latih model XGBoost dengan parameter terbaik
hasil tuning
start_time_smote = time.time()
xgb_smote = XGBClassifier(
use_label_encoder=False,
eval_metric='logloss',
learning_rate=0.01,
max_depth=3,
n_estimators=100,
scale_pos_weight=1,
# sudah seimbang setelah SMOTE
random_state=42
)
xgb_smote.fit(X_train_smote, y_train_smote)
end_time_smote = time.time()
waktu_smote = end_time_smote - start_time_smote
```

Output hasil evaluasi:

```
=== Evaluasi Model dengan SMOTE ===
Akurasi  : 0.9517
Precision: 0.9905
Recall   : 0.4190
F1-Score : 0.5889
AUC Score: 0.7651
Waktu Komputasi: 3.22 detik

=== Nilai TP, TN, FP, FN ===
True Positive (TP) : 838
False Positive (FP): 8
False Negative (FN): 1162
True Negative (TN) : 22230
```

Visualisasi kurva ROC dari hasil ini disajikan pada Gambar 4.15 ROC Curve pada SMOTE, yang menunjukkan keseimbangan antara sensitivitas dan spesifisitas model pasca penerapan teknik *oversampling*.



Gambar 4.15 ROC Curve pada SMOTE

Keterangan:

- Hasil evaluasi menunjukkan bahwa *recall* tetap sama dengan model sebelumnya (sekitar 0.4190), namun waktu komputasi meningkat karena jumlah data latih menjadi dua kali lipat. Hal ini menunjukkan bahwa meskipun SMOTE berhasil menyeimbangkan distribusi kelas, efektivitas model dalam mendeteksi malware tidak meningkat secara signifikan.
- Perbandingan dengan model sebelum SMOTE juga menunjukkan bahwa strategi penyesuaian *threshold* (Subbab 4.2.13) justru memberikan peningkatan *recall* yang lebih nyata, tanpa perlu memperbesar jumlah data latih. Oleh karena itu, hasil ini menjadi bagian penting dalam analisis akhir penelitian, bahwa penyesuaian strategi keputusan (*threshold*) bisa lebih efisien dibanding penambahan data secara sintetis.

4.2.15 Perbandingan Model

Setelah seluruh tahapan eksperimen dilakukan, tahap akhir dalam proses ini adalah melakukan perbandingan performa antar model. Bertujuan untuk mengetahui pendekatan mana yang paling efektif dalam meningkatkan kemampuan model XGBoost dalam mendeteksi aplikasi malware, terutama dalam situasi data tidak seimbang. Tabel ringkasan perbandingan metrik keempat model: *Accuracy*, *Precision*, *Recall*, *F1-Score*, AUC dapat dilihat pada Tabel 4.17. dan Tabel 4.18.

Tabel 4.17 Perbandingan Model XGBoost (*default*) vs *Threshold Tuning*

No	Metrik	Tanpa *Threshold Tuning*	Dengan *Threshold Tuning*
1	*Accuracy*	0.9517	0.7297
2	*Precision*	0.9905	0.1818
3	*Recall*	0.4190	0.6500
4	*F1-Score*	0.5889	0.2841

Tabel 4.18 Tabel Perbandingan Model XGBoost (*default*) vs SMOTE

No	Metrik	Tanpa SMOTE	Dengan SMOTE
0	*Accuracy*	0.9517	0.9517
1	*Precision*	0.9905	0.9905
2	*Recall*	0.4190	0.4190
3	*F1-Score*	0.5889	0.5889
4	AUC	0.7650	0.7651

Keterangan:

Berdasarkan hasil evaluasi terhadap data uji, diperoleh beberapa temuan penting:

- *Recall* meningkat secara signifikan pada model dengan *threshold tuning*, terutama saat *threshold* diturunkan dari 0.5 ke 0.4. Hal ini menunjukkan bahwa pengaturan ambang batas sangat berpengaruh dalam menangani ketidakseimbangan kelas, tanpa perlu menambahkan data baru.
- SMOTE berhasil menyamakan distribusi kelas dan mempertahankan nilai *precision* yang tinggi, namun tidak secara signifikan meningkatkan *recall*

dibanding *threshold tuning*. Waktu komputasi meningkat karena jumlah data latih menjadi dua kali lipat.

- *F1-Score* tertinggi dicapai saat *threshold* diatur ke 0.4, bukan pada model SMOTE maupun *default*, yang menandakan bahwa keseimbangan antara *precision* dan *recall* lebih optimal dicapai dengan strategi pengaturan keputusan (decision *threshold*) dibanding manipulasi data.

4.2.16 Decision Tree dan Karakteristik Malware Berdasarkan Permissions

Dilakukan interpretasi model klasifikasi melalui visualisasi pohon keputusan (*decision tree*) yang dibentuk oleh algoritma XGBoost. Visualisasi ini tidak hanya dimaksudkan sebagai alat bantu untuk memahami cara kerja model, tetapi juga untuk mengidentifikasi karakteristik umum dari aplikasi malware berdasarkan pola penggunaan fitur permissions. Masing-masing *permission* ditampilkan dalam bentuk kolom biner, dengan nilai 1 jika *permission* tersebut digunakan oleh aplikasi, dan 0 jika tidak.

Dari seluruh *permission* yang tersedia, telah dilakukan seleksi fitur untuk memilih 10 fitur paling relevan, yaitu *permission* yang paling sering muncul pada aplikasi malware dan memiliki pengaruh signifikan terhadap klasifikasi. Ke-10 fitur ini tidak hanya dipilih berdasarkan frekuensi, tetapi juga mempertimbangkan label pada setiap *permission*, yang ditandai dengan huruf (D) atau (S) di bagian akhir nama fitur. Label (D) merujuk pada kategori *Dangerous Permissions*, yaitu izin yang memiliki akses langsung terhadap data sensitif pengguna atau sistem inti perangkat, seperti lokasi GPS, penyimpanan eksternal, atau identitas perangkat. Izin-izin ini, jika disalahgunakan, dapat dimanfaatkan oleh aplikasi berbahaya untuk mencuri informasi atau mengakses komponen kritis tanpa sepengetahuan pengguna. Label (S) menandakan *Signature Permissions*, yakni izin yang hanya bisa diberikan jika aplikasi dengan sistem.

Berikut adalah daftar 10 fitur *permission* yang digunakan dalam pelatihan model, lengkap dengan klasifikasi label keteranganya pada Tabel 4.19.

Tabel 4.19 Fitur 10 *permission* beserta Label keterangan

No	Nama Permission	Label
1	Network communication: full Internet access	Dangerous (D)
2	Network communication: view network state	Signature (S)
3	Phone calls: read phone state and identity	Dangerous (D)
4	Storage: modify/delete USB storage contents	Dangerous (D)
5	Your location: fine (GPS) location	Dangerous (D)
6	write contact data	Signature (S)
7	prevent device from sleeping	Dangerous (D)
8	control vibrator	Signature (S)
9	MEDIA_CONTENT_CONTROL	System
10	READ_EXTERNAL_STORAGE	Dangerous (D)

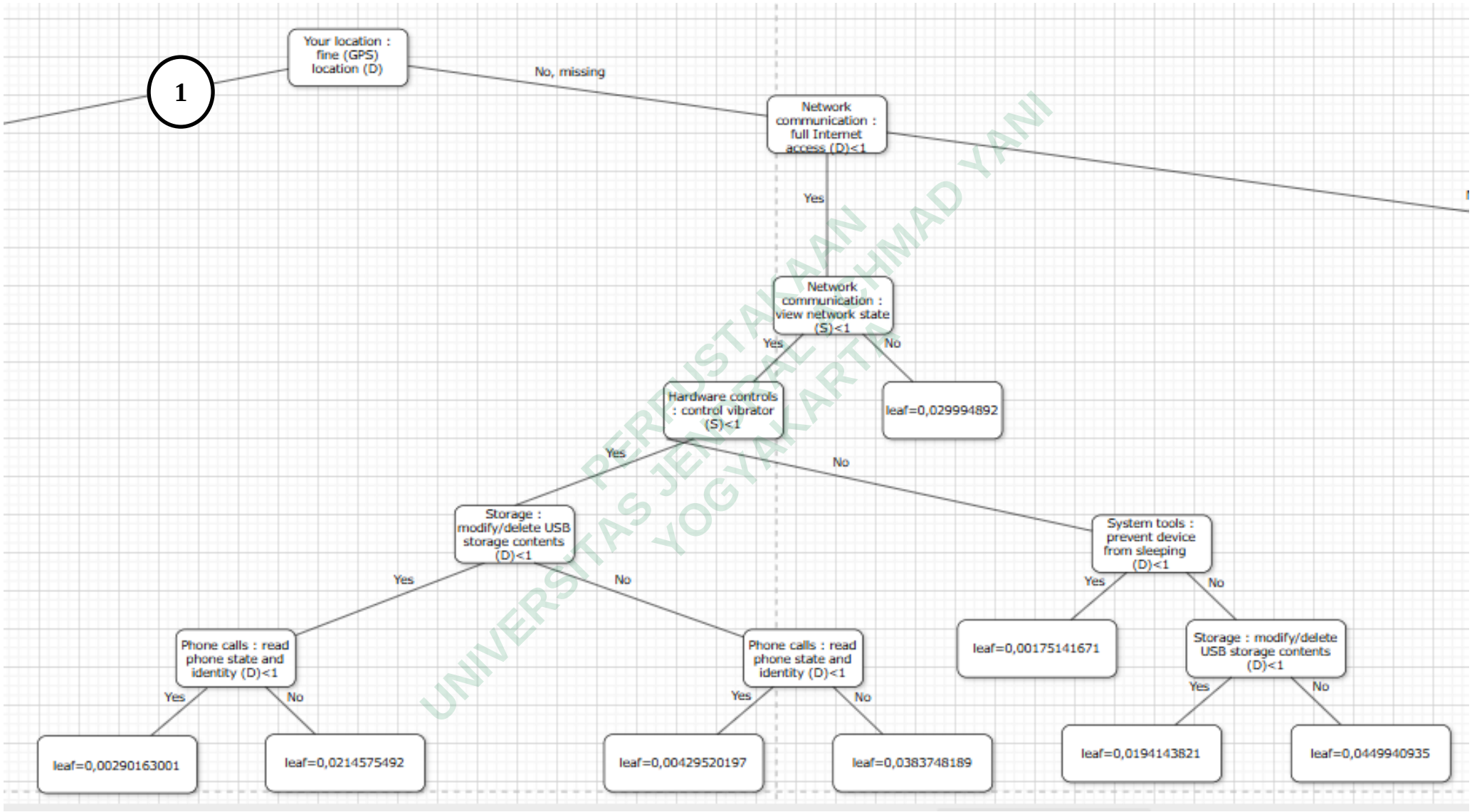
Model XGBoost menghasilkan total 100 pohon keputusan sebagai hasil dari boosting secara iteratif. Namun, untuk tujuan interpretasi, hanya ditampilkan satu pohon representatif yang memberikan gambaran mengenai struktur keputusan model secara visual. Masing-masing simpul dalam pohon ini mewakili evaluasi terhadap suatu fitur permission, dengan pembelahan berdasarkan nilai 0 atau 1 dari *permission* tersebut. Visualisasi ini menunjukkan bahwa fitur dengan label Dangerous sering menjadi simpul awal atau jalur utama dalam pohon keputusan. Sebagai contoh, model akan lebih condong mengklasifikasikan sebuah aplikasi sebagai malware jika fitur seperti Read_Phone_State, Access_Fine_Location, Atau Modify_Storage bernilai 1 (digunakan oleh aplikasi tersebut). Sebaliknya, aplikasi yang tidak mengaktifkan banyak *permission* sensitif atau hanya menggunakan *permission* yang berlabel Signature cenderung diprediksi sebagai non-malware.

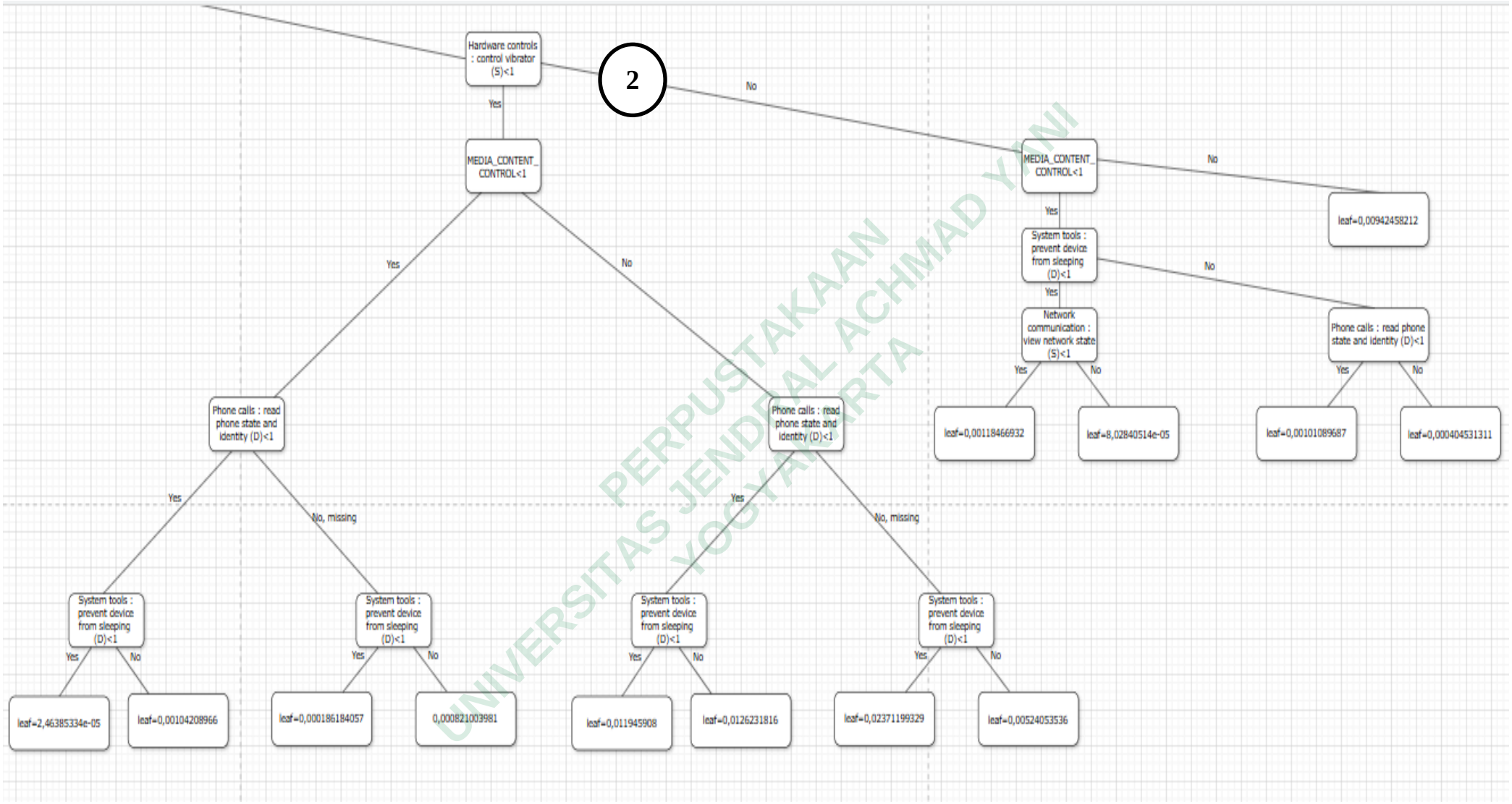
Gambar 4.16 menampilkan data aplikasi berdasarkan 10 fitur permission, yang memperlihatkan bagaimana nilai 0 dan 1 pada setiap fitur digunakan oleh model dalam proses klasifikasi.

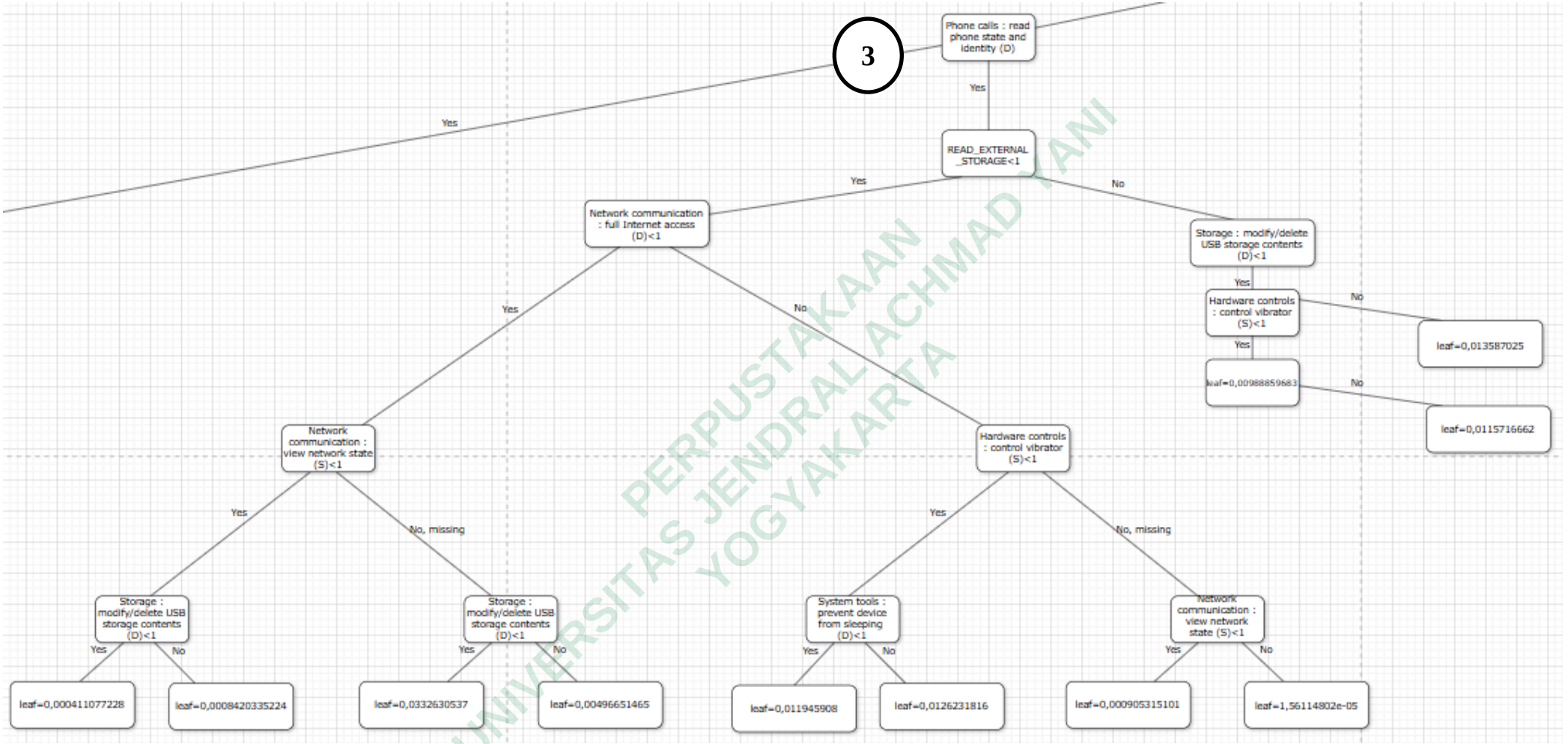
index	Network communication : full Internet access (D)	Network communication : view network state (S)	Phone calls : read phone state and identity (D)	St
0	0	0	0	
1	1	1	1	
2	1	1	1	
3	0	0	0	
4	1	1	1	
5	1	1	0	
6	1	0	0	
7	1	1	1	
8	1	0	0	
9	1	1	0	

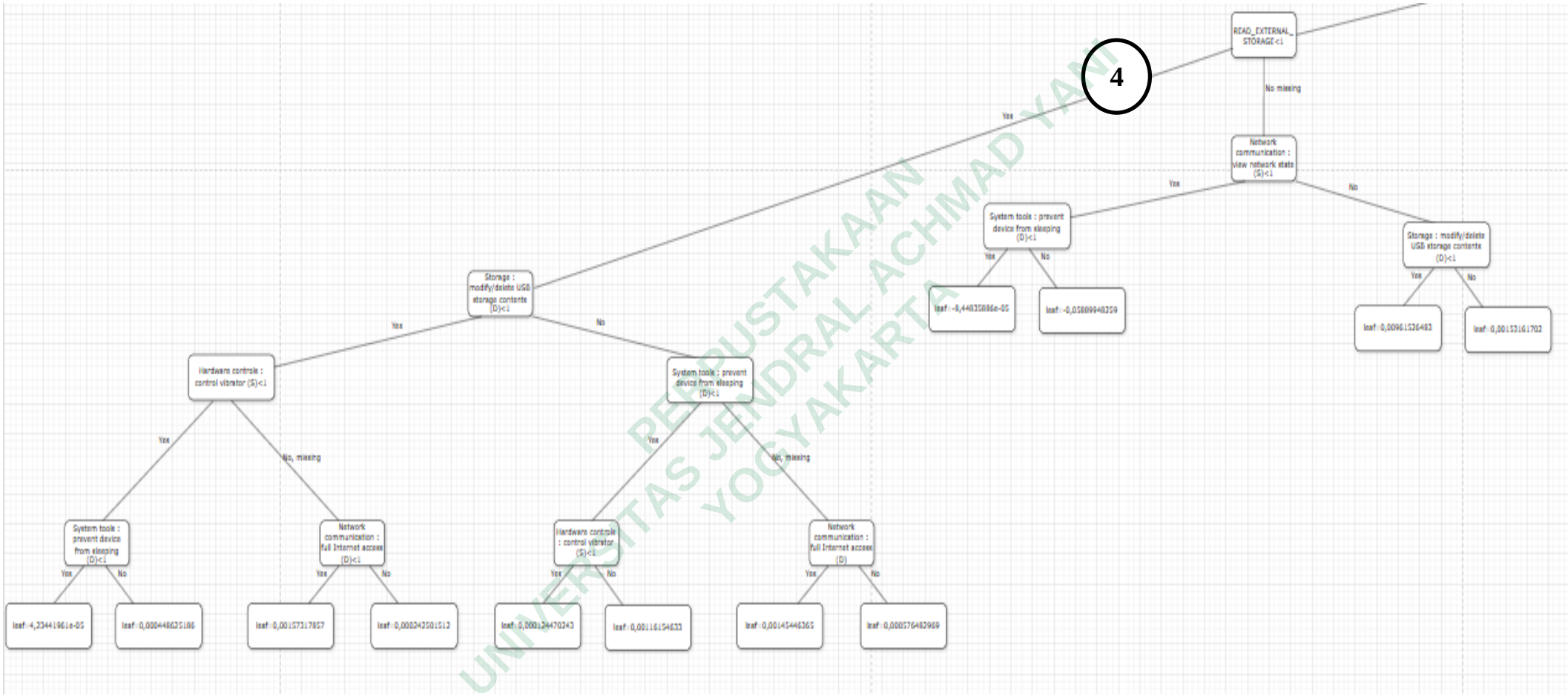
Gambar 4.16 Tabel dataset dengan 10 fitur terpilih sebagai masukan model

Gambar 4.17 disajikan visualisasi decision tree berdasarkan hasil pelatihan model terhadap dataset yang telah difilter menggunakan 10 fitur terbaik. Dalam gambar tersebut dapat dilihat bahwa kombinasi dari beberapa *permission* mampu membentuk jalur keputusan yang konsisten menuju prediksi kelas malware maupun non-malware.









Gambar 4.17 Visualisasi *decision tree* dari model XGBoost berdasarkan 10 fitur *permissions*

4.3 Analisis Hasil

Hasil eksperimen yang telah dilakukan menunjukkan bahwa algoritma XGBoost dapat memberikan performa yang kompetitif dalam mendeteksi malware Android berbasis fitur *permissions*, dengan penyesuaian yang tepat terhadap ketidakseimbangan kelas. Ketidakseimbangan ini menjadi tantangan utama, karena jumlah aplikasi non-malware dalam dataset jauh melebihi jumlah malware. Oleh karena itu, fokus utama dalam analisis ini diarahkan pada bagaimana model dapat mengenali kelas minoritas (malware) secara lebih efektif, tanpa mengorbankan akurasi secara keseluruhan.

Model awal XGBoost, yang hanya dilatih menggunakan parameter *default* dengan penyesuaian *scale_pos_weight*, memberikan nilai *precision* yang sangat tinggi (> 0.99), namun *recall* masih rendah (< 0.42). Hal ini menunjukkan bahwa model masih cenderung bias terhadap kelas mayoritas, sehingga banyak malware tidak terdeteksi (*False Negative* tinggi). Meskipun akurasi secara keseluruhan terlihat tinggi (> 95%), namun hal ini tidak mencerminkan kinerja sesungguhnya dalam kasus klasifikasi *imbalanced*, di mana *recall* dan *F1-Score* menjadi metrik yang lebih relevan.

Melalui proses *hyperparameter tuning* dengan GridSearchCV, ditemukan kombinasi parameter yang memberikan hasil lebih baik, namun peningkatan performa tidak signifikan tanpa intervensi tambahan. Untuk memperkuat analisis, dilakukan *threshold tuning* dan penerapan SMOTE. Hasil menunjukkan bahwa:

- *Threshold tuning* (*threshold* 0.4) memberikan peningkatan signifikan pada *recall* hingga 65%, tanpa penurunan *precision* yang terlalu drastis. Ini menunjukkan bahwa pengaturan ambang prediksi dapat digunakan untuk mengendalikan *trade-off* antara *False Negative* dan *False Positive*, yang sangat penting pada konteks deteksi malware.
- SMOTE, sebagai pendekatan *oversampling*, berhasil meningkatkan keseimbangan kelas secara struktural, namun efeknya terhadap performa

model tidak lebih baik dibanding *threshold tuning*. Waktu komputasi meningkat secara signifikan akibat duplikasi data sintetis.

Eksperimen juga dilakukan pada subset data berukuran 500, 1000, dan 2000 untuk menguji pengaruh ukuran data terhadap performa model. Hasilnya menunjukkan bahwa meskipun subset kecil (500) menghasilkan *recall* tinggi, nilai *precision* dan *F1-Score* tidak stabil. Pada subset 2000, performa lebih seimbang dan waktu komputasi masih dalam batas wajar. Hal ini menunjukkan bahwa model XGBoost cenderung memberikan hasil lebih stabil pada jumlah data yang cukup besar, tetapi tetap dapat digunakan secara efisien pada data berukuran kecil.

Secara keseluruhan, penyesuaian pada data (SMOTE) dan parameter (*threshold tuning*) terbukti lebih berdampak terhadap performa model daripada perluasan struktur decision tree itu sendiri. Hal ini menjawab kekhawatiran terkait keterbatasan jumlah cabang (tree) dalam klasifikasi biner (malware vs non-malware), karena fokus penelitian ini bukan pada eksplorasi struktur model, melainkan pada strategi pengolahan data untuk meningkatkan deteksi terhadap kelas minoritas. Dengan demikian, hasil penelitian ini telah menjawab tujuan dan pertanyaan penelitian:

1. XGBoost terbukti mampu menangani ketidakseimbangan kelas, terutama melalui kombinasi *scale_pos_weight* dan *threshold tuning*.
2. Fitur *permissions* efektif dalam membantu proses klasifikasi, terbukti dari performa model yang mampu membedakan aplikasi malware dengan cukup baik meskipun hanya menggunakan 10 fitur terbaik.
3. Evaluasi menggunakan metrik *Recall* dan *F1-Score* menjadi kunci untuk menilai efektivitas model, dan hasil menunjukkan adanya peningkatan signifikan setelah dilakukan penyesuaian Teknik.