JURNAL FASILKOM
Volume 16 No. 2 | Agustus 2026: 445-455

P-ISSN : 2089-3353
E-ISSN : 2808-9162

Pengembangan Model Klasifikasi *Code Smells* Pada *Backend Python* Menggunakan Algoritma *Random Forest* (Studi Kasus Proyek *Open Source Github*)

Aqilla Ar-Hammar[1], Mira Maisura[2]
[1,2]Program Studi Pendidikan Teknologi Informasi, Fakultas Tarbiah dan Keguruan, Universitas Islam Negri Ar-Raniry
[1]220212077@student.ar-raniry.ac.id *, [2]mira.maisura@ar-raniry.ac.id

Abstract

Deadline pressure in software development often drives coding shortcuts, leading to internal quality degradation known as code smells. These structural anomalies contribute to technical debt accumulation and complicate system maintenance over time. This study develops an automated classification model to detect code smell contamination in the Python backend ecosystem. The methodology uses the Random Forest ensemble algorithm integrated with the Synthetic Minority Over-sampling Technique (SMOTE) for class balancing. Data mining on GitHub with high-reputation criteria extracted 137,728 code samples from 7 large-scale repositories using the Radon multi-metric tool. To simulate human error in real-world scenarios, 5% random noise was inserted into the labeling data. Testing using the confusion matrix shows the proposed model achieves highly stable and balanced performance, with average precision, recall, and f1-score of 0.95 in both macro and weighted averages. Ablation study analysis proves that SMOTE intervention effectively maintains detection consistency in minority class categories. Feature importance ranking identifies the Logical Lines of Code (LLOC) metric as the most crucial indicator with 37.65% influence weight, followed by LOC and Blank metrics. This research provides an automated quality assurance system for developers to detect code refactoring opportunities at an early stage.

Keywords: backend ecosystem, code smells, ensemble learning, random forest, SMOTE

Abstrak

Tekanan tenggat waktu dalam pengembangan perangkat lunak sering mendorong pengembang mengambil jalan pintas penulisan kode, yang memicu munculnya degradasi kualitas struktural yang dikenal sebagai *code smells*. Penumpukan anomali struktural ini berkontribusi langsung pada pembengkakan utang teknis (*technical debt*) dan mempersulit proses pemeliharaan sistem dalam jangka panjang. Penelitian ini bertujuan mengembangkan model klasifikasi otomatis berbasis kecerdasan buatan untuk mendeteksi kontaminasi *code smells* pada arsitektur *backend Python*. Metode yang digunakan adalah eksperimen kuantitatif data sains dengan menerapkan algoritma *ensemble learning* Random Forest yang diintegrasikan dengan teknik penyeimbangan kelas SMOTE (*Synthetic Minority Over-sampling Technique*). Data dikumpulkan dari platform GitHub melalui kriteria inklusi reputasi tinggi, berhasil mengekstrak 137.728 entitas sampel kode dari 7 repositori skala besar menggunakan instrumen multi-metrik Radon. Untuk menyimulasikan kesalahan manusia, gangguan acak (*random noise*) sebesar 5% disisipkan ke dalam data pelabelan. Hasil pengujian menunjukkan model usulan mencapai performa klasifikasi yang stabil dan seimbang dengan rata-rata presisi, *recall*, dan *f1-score* sebesar 0.95 pada evaluasi *macro average* dan *weighted average*. Melalui studi ablasi, SMOTE terbukti secara empiris menjaga konsistensi deteksi pada kelas minoritas. Analisis kontribusi fitur mengidentifikasi metrik *Logical Lines of Code* (LLOC) sebagai indikator paling krusial dengan bobot pengaruh 37,65%. Penelitian ini memberikan kontribusi praktis berupa sistem penjaminan kualitas otomatis bagi pengembang untuk mendeteksi lokasi refaktorisasi kode sejak dini, sehingga mampu menekan pembengkakan utang teknis dan mempermudah pemeliharaan sistem.

Kata kunci: *backend ecosystem, code smells, ensemble learning, random forest,* SMOTE

©This work is licensed under a Creative Commons Attribution -ShareAlike 4.0 International License

1. Pendahuluan

Dalam era transformasi digital yang masif, arsitektur pendukung belakang (*backend ecosystem*) memegang peranan krusial sebagai mesin utama pengendali logika bisnis, keamanan, dan integrasi basis data dari suatu aplikasi perangkat lunak skala besar. Mengingat signifikansi fungsinya, menjaga kualitas internal dari kode sumber (*source code*) pada lapisan *backend* menjadi sebuah kebutuhan mutlak demi menjamin reliabilitas dan skalabilitas sistem dalam jangka panjang [1]. Namun, tekanan tenggat waktu peluncuran produk dan perubahan kebutuhan pengguna yang dinamis sering kali memaksa pengembang mengambil jalan pintas dalam penulisan kode, yang secara bertahap memicu munculnya degradasi kualitas struktural internal yang dikenal dengan istilah *code smells* [2].

Code smells adalah gejala pada kode sumber yang mengindikasikan adanya masalah pada struktur desain internal perangkat lunak [3]. Keberadaan *code smells* dalam jangka panjang menyebabkan penumpukan utang teknis (*technical debt*) yang menghambat kecepatan tim dalam membangun fitur baru dan meningkatkan biaya pemeliharaan sistem secara signifikan [4]. Pada arsitektur *backend*, dua jenis *code smells* yang paling dominan ditemukan adalah *God Class* kelas yang memiliki terlalu banyak tanggung

P-ISSN : 2089-3353
E-ISSN : 2808-9162

jawab fungsional serta *Long Method* metode dengan baris kode yang terlalu panjang [5].

Secara tradisional, identifikasi *code smells* dilakukan melalui peninjauan kode secara manual (*manual code review*) atau penggunaan perangkat bantu berbasis ambang batas statis (*threshold-based tools*). Kedua pendekatan ini memiliki batasan yang signifikan. Peninjauan manual sangat bergantung pada subjektivitas, pengalaman, dan keahlian penilai, sehingga hasilnya cenderung tidak konsisten antar individu. Selain itu, metode ini tidak efektif untuk diterapkan pada proyek modern berskala *enterprise* yang memiliki ratusan ribu hingga jutaan baris kode. Di sisi lain, pendekatan berbasis ambang batas statis sering menghasilkan tingkat kesalahan deteksi (*false positive*) yang tinggi karena tidak mampu beradaptasi secara fleksibel terhadap karakteristik unik setiap proyek perangkat lunak [6]. Akibat dari keterbatasan ini, banyak *code smells* yang lolos dari deteksi (*false negative*) atau sebaliknya, kode yang sebenarnya baik justru ditandai sebagai bermasalah. Dampak jangka panjangnya adalah pembengkakan *technical debt* yang semakin sulit dikelola, peningkatan biaya pemeliharaan dan penurunan produktivitas tim pengembang.

Beberapa penelitian terdahulu telah menerapkan algoritma pembelajaran mesin untuk klasifikasi kualitas kode, dan secara umum menunjukkan hasil yang menjanjikan. A. Abdou and N. Darwish [5] melakukan eksperimen pada 74 sistem dengan 16 algoritma, dan menemukan bahwa J48 serta *Random Forest* memberikan performa terbaik dengan akurasi hingga 99,02%. Di Nucci dkk. [7], melanjutkan eksperimen serupa dengan menguji 32 algoritma dan melaporkan bahwa *Random Forest* kembali menjadi salah satu algoritma terbaik dengan akurasi hingga 96%. Studi oleh Rajwant Singh Rao dkk. [8], menunjukkan bahwa penerapan lima algoritma *ensemble learning* (seperti AdaBoost dan Bagging) yang dikombinasikan dengan SMOTE mampu mencapai akurasi 100% pada beberapa *dataset code smells*. Sementara itu, penelitian Mhawish dan Gupta [9] melaporkan bahwa *Random Forest* mencapai akurasi 99,74% untuk mendeteksi *Data Class*.

Namun, sebagian besar riset tersebut masih menghadapi kendala serius terkait fenomena ketidakseimbangan kelas (*class imbalance*) pada data mentah, di mana sampel kode bersih jauh lebih dominan daripada sampel yang mengandung *code smells* [8]. Kondisi ini sering kali diabaikan, padahal jika model dilatih dengan data yang tidak seimbang, model cenderung mengalami bias dan hanya optimal dalam mengenali kelas mayoritas [10]. Sebagai contoh, penelitian Alazba dan Aljamaan [12] menunjukkan bahwa model yang dilatih tanpa penanganan *class imbalance* menghasilkan performa yang lebih rendah. Selain itu, aspek ketangguhan (*robustness*) model terhadap gangguan pelabelan data akibat kesalahan manusia juga masih jarang dievaluasi secara mendalam [1]. Lewowski dan Madeyski [11] juga menyoroti bahwa deteksi *code smells* secara manual sangat bergantung pada subjektivitas dan sulit untuk direproduksi, yang menjadi tantangan tersendiri dalam penelitian di bidang ini..

Guna mengatasi kesenjangan tersebut, penelitian ini mengusulkan pengembangan model klasifikasi otomatis *code smells* pada ekosistem *backend* Python melalui integrasi *ensemble learning* Random Forest dan SMOTE. Random Forest dipilih karena arsitekturnya yang berbasis *bagging* (*bootstrap aggregating*) terbukti efektif dalam mengurangi varians prediksi dengan membangun banyak pohon keputusan independen dari subset data yang diambil secara acak, sehingga menghasilkan model yang lebih stabil dan tidak mudah *overfitting* terhadap data latih [5][7][8]. Kemampuan ini sangat krusial untuk menangani kompleksitas metrik kode sumber yang memiliki variasi antar proyek *open source*. Sementara itu, SMOTE dipilih untuk mengatasi permasalahan ketidakseimbangan kelas (*class imbalance*) yang menjadi kendala utama pada penelitian terdahulu [8][9][12]. Pada penelitian oleh Dewangan dkk. [8], dataset *code smells* memiliki distribusi kelas yang tidak seimbang di mana sampel kode bersih jauh lebih dominan dibandingkan sampel *smells*. Kondisi serupa juga ditemukan pada studi Alazba dan Aljamaan [12] yang melaporkan bahwa model yang tidak menerapkan teknik penyeimbangan data mengalami penurunan akurasi pada kelas minoritas. Dengan demikian, integrasi SMOTE diharapkan mampu menjaga konsistensi deteksi pada kelas *smells* yang jumlah sampelnya lebih sedikit.

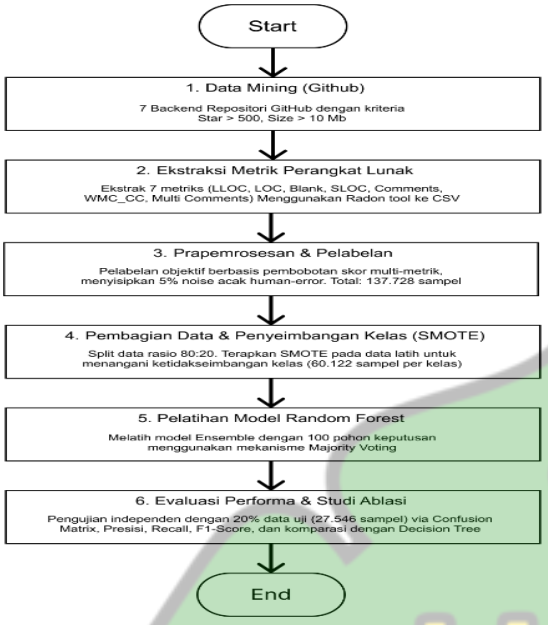
Untuk itu, penelitian ini bertujuan mengembangkan model klasifikasi otomati *code smells* pada ekosistem *backend* Python dengan mengintegrasikan algoritma *ensemble learning* Random Forest dan teknik penyeimbangan kelas SMOTE. Model yang dikembangkan akan dievaluasi menggunakan metrik standar seperti akurasi, presisi, recall, dan F1-score untuk mengukur kemampuannya dalam membedakan kode bersih dan kode berindikasi *smells*. Selain itu, penelitian ini juga melakukan pengujian komparatif melalui studi ablasi untuk menganalisis pengaruh SMOTE terhadap stabilitas deteksi pada kelas minoritas, serta pemetaan peringkat metrik kode melalui analisis *feature importance* sebagai panduan refaktorisasi bagi pengembang dalam memprioritaskan perbaikan kode.

2. Metode Penelitian

Penelitian ini menggunakan pendekatan kuantitatif dengan metode eksperimen berbasis *machine learning* untuk membangun model klasifikasi otomatis dalam mendeteksi kontaminasi *code smells* pada komponen arsitektur *backend* Python. Metodologi penelitian mencakup berbagai tahapan yang sistematis, mulai dari pengumpulan data, ekstraksi metrik, pra-pemrosesan data, pengembangan model, hingga evaluasi performa

P-ISSN : 2089-3353
E-ISSN : 2808-9162

[1]. Seluruh tahapan penelitian divisualisasikan pada Gambar 1.



Gambar 1. Diagram *Flowchart* Alur Eksperimen Sistem

2.1. Pengumpulan Data

Pengumpulan data dilakukan melalui penambangan repositori *open-source* di platform *GitHub*. Berdasarkan kriteria inklusi yang ketat proyek berbasis Python, memiliki bintang (*stars*) di atas 500, ukuran di atas 10.000 KB, dan difungsikan sebagai *backend enterprise* diperoleh 7 proyek utama sebagai subjek observasi [5]. Dari seluruh repositori tersebut, hanya berkas kode yang menerapkan struktur kelas atau *Object-Oriented Programming* (OOP) yang diambil guna mempertahankan relevansi metrik kualitas internal [2]. Ketujuh repositori yang dipilih di antara lain adalah: *Dash-FastAPI-Admin, django-vue3-admin, minpy, mealie, openvino-ai-plugins-gimp, core, dan ansible.*

2.2. Ekstraksi Metrik Perangkat Lunak

Ekstraksi metrik kode dilakukan menggunakan perangkat Radon secara otomatis. Radon adalah alat analisis statis sumber terbuka (*open-source*) yang dikembangkan khusus untuk bahasa pemrograman Python [2]. Alat ini banyak digunakan dalam praktik rekayasa perangkat lunak untuk mengukur berbagai aspek kualitas kode secara kuantitatif. Radon mampu menghitung metrik kode pada tiga tingkatan utama: modul, kelas, dan fungsi. Metrik yang dihasilkan mencakup ukuran fisik kode, kompleksitas logika, dan dokumentasi, sehingga memberikan gambaran menyeluruh tentang struktur internal perangkat lunak yang dianalisis [8].

Untuk memperoleh data kuantitatif dari ketujuh repositori yang telah dikumpulkan, Radon dijalankan secara rekursif pada seluruh berkas kode Python dalam masing-masing proyek. Proses ekstraksi ini menghasilkan tujuh metrik kuantitatif yang dikelompokkan berdasarkan jenisnya, meliputi metrik ukuran fisik, dokumentasi, dan kompleksitas internal [4]. Metrik-metrik tersebut beserta singkatan dan deskripsinya dirangkum pada Tabel 1.

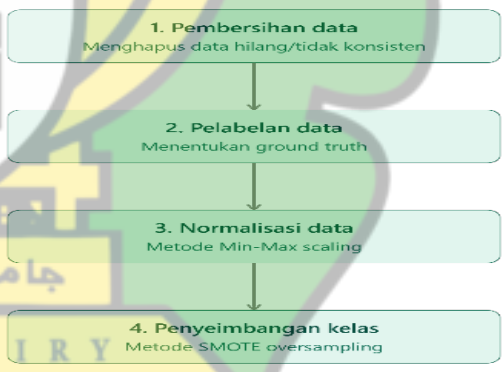
Tabel 1. Metrik Kode yang Diekstrak dengan Radon

Simbol Metrik	Kepanjangan
LOC	*Lines Of Code*
LLOC	*Logical Lines Of Code*
SLOC	*Source Line Of Code*
Comments	*Comments*
Multi	*Multi-line Comments*
Blank	*Blank Lines*
WMC_CC	*Weighted Methods per Class – Cyclomatic Complexity*

Metrik-metrik ini berfungsi sebagai variabel masukan (*features*) yang dianalisis oleh algoritma untuk menentukan apakah suatu bagian kode tergolong *code smell* atau tidak [8][9]. Seluruh data fitur yang telah diekstrak kemudian disimpan ke dalam format tabulasi CSV (*Comma-Separated Values*) untuk memudahkan proses pra-pemrosesan data dan pelatihan model pada tahap selanjutnya.

2.3. Pra-pemrosesan Data

Tahap prapemrosesan data (*preprocessing*) merupakan fase awal sebelum data dapat digunakan untuk pelatihan model. Pada penelitian ini, tahapan *preprocessing* yang dilalui ditampilkan pada Gambar 2.



Gambar 2. Flowchart Alur Eksperimen Sistem (*Preprocessing*)

Berdasarkan diagram alur diatas, tahapan pertama adalah pembersihan data (*data cleaning*) untuk memastikan tidak ada nilai yang hilang atau tidak konsisten. Tahapan kedua adalah pelabelan data (*labeling*) untuk menentukan *ground truth* setiap entitas kode. Proses pelabelan dilakukan secara otomatis oleh sistem berdasarkan skor multi-metrik yang dihitung dari kombinasi indikator LLOC, kompleksitas metode, dan rasio dokumentasi, tanpa melibatkan intervensi manusia secara langsung [6][11]. Tahap ketiga adalah normalisasi data (*data normalization*) menggunakan metode Min-Max untuk menyamakan skala nilai metrik. Tahap keempat

P-ISSN : 2089-3353
E-ISSN : 2808-9162

penyeimbangan kelas menggunakan SMOTE untuk mengatasi ketidakseimbangan distribusi kelas [6][11].

Proses pelabelan data (*ground truth*) dilakukan secara objektif melalui skema skor multi-metrik yang dikembangkan berdasarkan kombinasi dari tiga indikator utama baris logika aktif (LLOC), kompleksitas metode (WMC_CC), dan rasio dokumentasi (rasio antara baris komentar dengan total baris kode). Skor akhir setiap entitas ditentukan melalui pembobotan ketiga indikator tersebut, di mana entitas dengan skor di atas ambang batas tertentu dikategorikan sebagai *Smells* (1) dan di bawah ambang batas sebagai Normal (0). Pelabelan dilakukan secara otomatis oleh sistem berdasarkan skor yang dihasilkan, tanpa melibatkan intervensi manusia secara langsung, sehingga konsistensi label dapat terjaga [6][11]. Untuk menyimulasikan margin kesalahan manusia (*human error*) yang natural, gangguan acak (*random noise*) sebesar 5% disisipkan ke dalam data pelabelan [1].

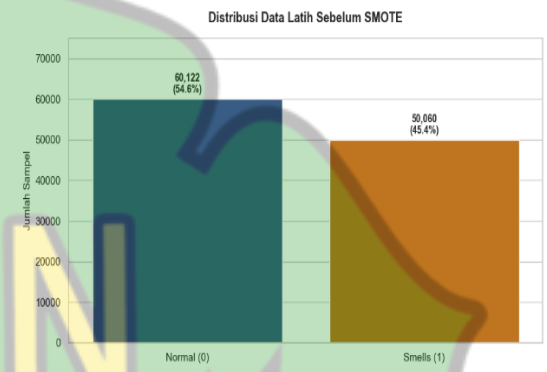
2.4. Pembagian Data dan Penyeimbangan Kelas

Penelitian ini menggunakan dua kategori kelas dalam proses klasifikasi, yaitu kelas Normal (0) untuk entitas kode yang tidak mengandung *code smells* dan kelas *Smells* (1) untuk entitas kode yang terindikasi mengandung *code smells*. Komposisi kelas asli sebelum dilakukan penyeimbangan menunjukkan sifat ketidakseimbangan kelas (*class imbalance*), dengan rincian kelas Normal (0) sebanyak 75.153 sampel dan kelas *Smells* (1) sebanyak 62.575 sampel. Ketimpangan ini berisiko mengakibatkan terjadinya bias prediksi ketika menggunakan algoritma dengan klasifikasi konvensional [8].

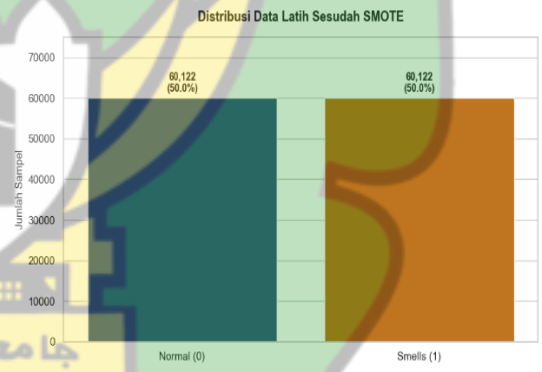
Untuk mengatasi hal tersebut, penelitian ini menerapkan teknik *Synthetic Minority Over-sampling Technique* (SMOTE). SMOTE bekerja dengan membuat sampel sintetis untuk kelas minoritas (*Smells*) berdasarkan interpolasi antara sampel yang sudah ada [9]. Secara teknis, untuk setiap sampel dalam kelas minoritas, SMOTE mencari tetangga terdekatnya (*k-nearest neighbors*) dalam ruang fitur yang sama, lalu menciptakan sampel baru dengan melakukan interpolasi linier antara sampel tersebut dan salah satu tetangganya. Proses ini diulang hingga jumlah sampel kelas minoritas setara dengan kelas mayoritas.

Tahapan pemisahan data (*data splitting*) menggunakan rasio standar 80% untuk data latih (*training set*) dan 20% untuk data uji (*testing set*) [12]. Setelah menerapkan SMOTE pada porsi data latih, distribusi kelas berhasil diselaraskan secara sempurna dengan jumlah seimbang yaitu 60.122 sampel untuk kelas Normal (0) dan 60.122 sampel untuk kelas *Smells* (1).

Penerapan SMOTE dalam penelitian ini dilakukan setelah proses pemisahan data (*data splitting*) pada data latih, bukan sebelum. Pertimbangan utamanya adalah untuk menjaga kemurnian data uji agar tetap merepresentasikan distribusi asli data di lapangan (*real-world distribution*), sehingga evaluasi model lebih objektif dan tidak *over*-optimis. Dengan demikian, langkah yang dilakukan adalah membagi data menjadi data latih (80%) dan data uji (20%) menggunakan rasio standar [12], kemudian menerapkan SMOTE hanya pada data latih untuk menyeimbangkan distribusi kelas, serta mempertahankan data uji dalam kondisi aslinya (tidak diseimbangkan) untuk evaluasi yang realistis. Setelah menerapkan SMOTE pada porsi data latih, distribusi kelas berhasil diselaraskan secara sempurna dengan jumlah seimbang yaitu 60.122 sampel untuk kelas Normal (0) dan 60.122 sampel untuk kelas *Smells* (1). Perbandingan distribusi kelas sebelum dan sesudah penerapan SMOTE disajikan pada Gambar 3 dan Gambar 4.



Gambar 3. Distribusi Kelas Sebelum SMOTE



Gambar 4. Distribusi Kelas Sesudah SMOTE

Model kecerdasan buatan dibangun menggunakan algoritma Random Forest, sebuah arsitektur *ensemble learning* yang menerapkan prinsip pembelajaran *Bagging* (*Bootstrap Aggregating*) [8]. Dalam proses pelatihan, model mengonstruksi koleksi 100 pohon keputusan (*decision trees*) independen secara acak dari subset data latih. Keputusan akhir klasifikasi untuk memisahkan entitas kode berkategori Normal (0) atau mengandung *Smells* (1) ditentukan melalui mekanisme pengambilan suara terbanyak (*majority voting*) dari seluruh pohon keputusan tersebut [9].

Parameter utama yang digunakan dalam model Random Forest adalah *n_estimators* (jumlah pohon keputusan) yang ditetapkan sebanyak 100, *max_depth* (kedalaman maksimum pohon) yang tidak dibatasi untuk memungkinkan pertumbuhan pohon secara

P-ISSN : 2089-3353
E-ISSN : 2808-9162

penuh, serta *min_samples_split* (jumlah minimum sampel untuk melakukan split) yang diset pada nilai default [2][9][10]. Selain itu, parameter *max_features* yang menentukan jumlah fitur yang dipertimbangkan pada setiap split diatur menggunakan nilai default "sqrt" (akar kuadrat dari total fitur), sementara *bootstrap* diaktifkan untuk memastikan setiap pohon dibangun dari sampel data yang berbeda melalui teknik *bootstrap sampling* [8].

Selain model usulan *Random Forest* + SMOTE, penelitian ini juga mengembangkan tiga skenario pembanding untuk keperluan studi ablasi, yaitu: *Random Forest* tanpa SMOTE, *Decision Tree* + SMOTE, dan *Decision Tree* tanpa SMOTE [7]. Seluruh model yang dikembangkan menggunakan bahasa pemrograman Python dengan *library Scikit-Learn* [10]. Proses *hyperparameter tuning* dilakukan dengan pendekatan *grid search* untuk mendapatkan konfigurasi model yang optimal, di mana kombinasi parameter diuji secara sistematis dan dipilih berdasarkan performa terbaik pada data validasi.

2.5. Evaluasi Model

Pengujian keandalan model divalidasi menggunakan porsi data uji mandiri yang diukur secara kuantitatif melalui instrumen pengujian matriks kebingungan (*confusion matrix*) untuk menghasilkan nilai presisi (*precision*), *recall*, dan F1-score [5]. Metrik evaluasi yang digunakan adalah:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \quad (1)$$

$$Precision = \frac{TP}{TP+FP} \quad (2)$$

$$Recall = \frac{TP}{TP+FN} \quad (3)$$

$$F1-Score = 2 \times \frac{Precision \times Recall}{Precision+Recall} \quad (4)$$

dengan TP (*True Positive*) adalah jumlah data *smells* yang terdeteksi benar, TN (*True Negative*) adalah jumlah data normal yang terdeteksi benar, FP (*False Positive*) adalah data normal yang salah terdeteksi sebagai *smells*, dan FN (*False Negative*) adalah data *smells* yang tidak terdeteksi [8].

Selain itu, pengujian studi ablasi diintegrasikan secara penuh untuk mengevaluasi efisiensi parameter model serta pengaruh kehadiran algoritma SMOTE terhadap stabilitas pengklasifikasi saat dibandingkan dengan model tunggal tradisional Decision Tree [7]. Analisis kontribusi fitur (*feature importance*) juga dilakukan untuk mengidentifikasi metrik kode mana yang paling berpengaruh terhadap klasifikasi *code smells* [4].

2.6. Studi Ablasi

Studi ablasi (*ablation study*) merupakan pengujian komparatif yang dirancang untuk mengevaluasi kontribusi masing-masing komponen model terhadap performa keseluruhan [7]. Dalam penelitian ini, studi ablasi dilakukan dengan membandingkan performa model usulan (*Random Forest* + SMOTE) terhadap tiga skenario pembanding, yaitu *Random Forest* tanpa SMOTE, *Decision Tree* + SMOTE, dan *Decision Tree* tanpa SMOTE [7]. Tujuan dari pengujian ini adalah untuk mengukur secara kuantitatif seberapa besar pengaruh SMOTE dalam menjaga stabilitas deteksi pada kelas minoritas (*Smells*), serta membandingkan efektivitas arsitektur *ensemble learning* Random Forest terhadap model tunggal Decision Tree.

Seluruh skenario model dilatih dan dievaluasi menggunakan data uji yang sama, yaitu 20% dari total data yang tidak dilibatkan dalam proses pelatihan [12]. Performa setiap skenario diukur berdasarkan metrik akurasi global (*global accuracy*) dan F1-score tertimbang (*weighted F1-score*) untuk mendapatkan gambaran yang komprehensif tentang kemampuan klasifikasi masing-masing model [8]. Hasil dari studi ablasi ini akan menunjukkan apakah penambahan SMOTE memberikan peningkatan performa yang signifikan, serta apakah *Random Forest* sebagai model *ensemble* lebih unggul dibandingkan *Decision Tree* dalam menangani variasi data metrik kode yang kompleks [7][8].

3. Hasil dan Pembahasan

Pada bab ini dipaparkan seluruh rangkaian hasil pengujian empiris dari implementasi model klasifikasi cerdas untuk deteksi *code smells* pada ekosistem aplikasi *backend* berbasis bahasa pemrograman Python. Penelitian ini memproses total 137.728 entitas sampel kode yang diekstraksi dari 7 repositori *open-source* GitHub. Data yang digunakan berbentuk tabulasi numerik dalam format CSV yang terdiri dari tujuh metrik kuantitatif, yaitu LOC, LLOC, SLOC, *Comments*, *Multi*, *Blank*, dan WMC_CC. Hasil ekstraksi metrik menunjukkan bahwa setiap entitas kode memiliki nilai pada ketujuh metrik tersebut yang kemudian menjadi fitur masukan (*input features*) bagi model klasifikasi.

Proses pemrosesan dan pelabelan data dilakukan secara objektif melalui skema skor multi-metrik yang mengkombinasikan kepadatan baris logika aktif, kompleksitas metode, dan rasio dokumentasi. Dari total 137.728 sampel, komposisi kelas awal menunjukkan ketidakseimbangan dengan kelas Normal (0) sebanyak 75.153 sampel dan kelas *Smells* (1) sebanyak 62.575 sampel. Untuk mengatasi hal tersebut, data dibagi dengan rasio 80:20, di mana 80% data (110.182 sampel) digunakan sebagai data latih dan 20% data (27.546 sampel) sebagai data uji [12]. SMOTE kemudian diterapkan hanya pada data latih, menghasilkan distribusi kelas yang seimbang yaitu 60.122 sampel untuk kelas Normal (0) dan 60.122 sampel untuk kelas *Smells* (1). Sementara itu, data uji dipertahankan dalam kondisi aslinya (tidak diseimbangkan) untuk evaluasi yang realistis [9]. Pengujian dirancang untuk memvalidasi performa algoritma Random Forest yang dikombinasikan dengan

P-ISSN : 2089-3353
E-ISSN : 2808-9162

teknik penyeimbangan kelas SMOTE dibandingkan dengan algoritma tradisional lainnya dalam skenario studi ablasi.

3.1. Karakteristik Eksperimen dan Profil Dataset

Proses penambangan data (*data mining*) dalam penelitian ini dilakukan secara terencana dan terarah pada platform GitHub dengan menetapkan kriteria inklusi yang ketat, meliputi proyek berbasis Python, memiliki bintang (*stars*) di atas 500, ukuran repositori melebihi 10.000 KB, dan difungsikan sebagai *backend enterprise* [13]. Berdasarkan kriteria tersebut, dipilih 7 proyek *open-source* sebagai subjek observasi, yaitu: *Dash-FastAPI-Admin, django-vue3-admin, minpy, mealie, openvino-ai-plugins-gimp, core, dan ansible.* Karakteristik struktural internal serta distribusi total seluruh entitas sampel data yang berhasil diekstraksi dari ketujuh repositori tersebut dirangkum pada Tabel 2.

Tabel 2. Dataset *Open-Source* GitHub

Repositori *Github*	Karakteristik *Backend*	Bahasa
Dash-FastAPI-Admin	*Dashboard & Admin System*	*Python*
Django-vue3-admin	*Web Application Framework Backend*	*Python*
Minpy	*NumPy-compliant Complex Engine*	*Python*
Mealie	*Recipe & Nutrient Graph API Service*	*Python*
Openvino-ai-plugins-gimp	*AI Infrastructure Plugins Extension*	*Python*
Core	*Home Assistant Automation Architecture*	*Python*
Ansible	*Configuration Management & Deployment*	*Python*

Seluruh sampel diekstraksi secara kuantitatif menggunakan perangkat Radon untuk memperoleh nilai indikator ukuran kode dan kompleksitas struktur logika internal. Melalui mekanisme pelabelan objektif berbasis pembobotan skor multi-metrik serta penyisipan unsur *noise* acak sebesar 5% untuk menyimulasikan faktor kesalahan manusia, diperoleh distribusi kelas awal yang sangat organik. Komposisi kelas asli menunjukkan sifat ketidakseimbangan kelas (*class imbalance*), dengan rincian kelas Normal (0) sebanyak 75.153 sampel dan kelas *Smells* (1) sebanyak 62.575 sampel.

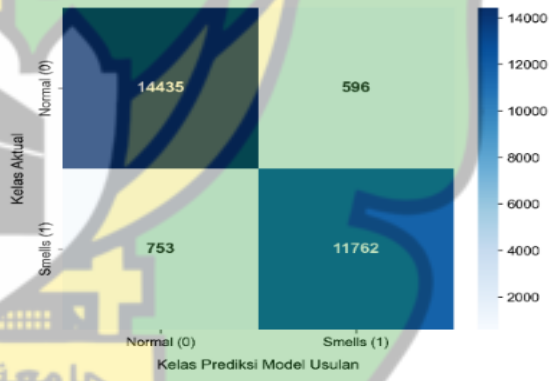
Tahapan pemisahan data (*data splitting*) menggunakan rasio standar 80% untuk data latih dan 20% untuk data uji. Setelah menerapkan SMOTE pada porsi data latih, distribusi kelas berhasil diselaraskan secara sempurna dengan jumlah seimbang yaitu 60.122 sampel untuk kelas Normal (0) dan 60.122 sampel untuk kelas *Smells* (1).

3.2. Evaluasi Kuantitatif Performa Model Klasifikasi Usulan

Pengujian model usulan (*Random Forest* + SMOTE) dilakukan menggunakan data uji independen sebanyak 27.546 sampel, yang di mana data tersebut sama sekali tidak dilibatkan dalam proses pelatihan model sebelumnya. Berdasarkan hasil pengujian yang dilakukan menggunakan matriks kebingungan (*confusion matrix*), Model yang diusulkan dalam penelitian ini terbukti mampu memprediksi secara tepat sebanyak 14.435 sampel yang diklasifikasikan sebagai kelas Normal (*True Negative*), di sisi lain model ini juga tercatat hanya menghasilkan 596 kesalahan prediksi tipe I (*False Positive*). Dalam proses identifikasi entitas kode yang mengalami kontaminasi cacat, model berhasil mengklasifikasikan 11.762 sampel secara tepat ke dalam kelas *Smells* (*True Positive*) dengan tingkat luput (*False Negative*) sebanyak 753 sampel. Pola distribusi klasifikasi tersebut ditampilkan pada Tabel 3 serta divisualisasikan dalam Gambar 5.

Tabel 3. *Confusion Matrix*

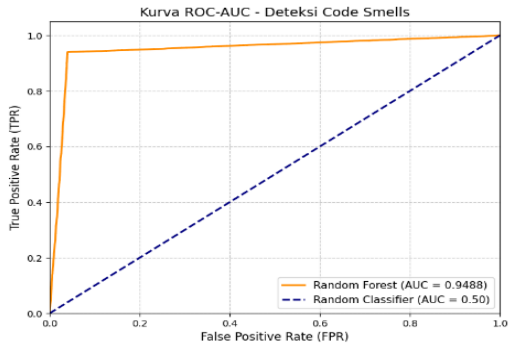
Kelas Aktual (*Actual Class*)	Kelas Prediksi Model (Normal)	Kelas Prediksi Model (*Smells*)
Normal (0)	14.435	596
Smells (1)	753	11.762



Gambar 5. *Heatmap Confusion Matrix*

Selain *confusion matrix*, performa model juga dievaluasi menggunakan kurva ROC-AUC (*Receiver Operating Characteristic - Area Under the Curve*) yang ditampilkan pada Gambar 6. Kurva ROC menggambarkan kemampuan model dalam membedakan antara kelas Normal dan *Smells* pada berbagai tingkat ambang batas (*threshold*). Nilai AUC yang mendekati 1 menunjukkan bahwa model memiliki kemampuan diskriminasi yang sangat baik. Kurva ROC-AUC pada Gambar 6 memperlihatkan bahwa model usulan mencapai nilai AUC yang tinggi, yang mengindikasikan bahwa model mampu memisahkan kedua kelas secara efektif.

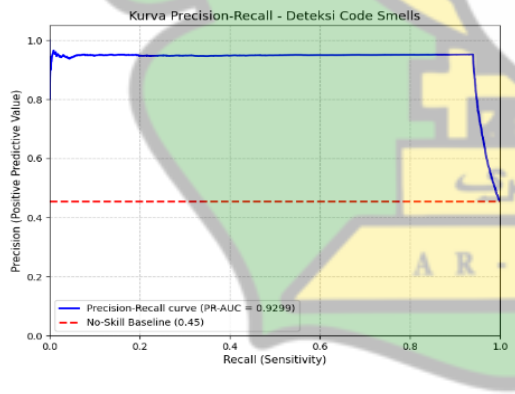
P-ISSN : 2089-3353
E-ISSN : 2808-9162



Gambar 6. Kurva ROC-AUC (*Receiver Operating Characteristic*)

Gambar 6 menunjukkan kurva ROC-AUC dengan nilai AUC sebesar 0.9488, yang mengindikasikan bahwa model Random Forest dengan SMOTE memiliki kemampuan sangat baik dalam membedakan antara kode Normal dan kode berindikasi *Smells*. Nilai AUC yang mendekati 1.0 ini memperkuat temuan pada Tabel 3 bahwa model memiliki performa klasifikasi yang stabil dan seimbang.

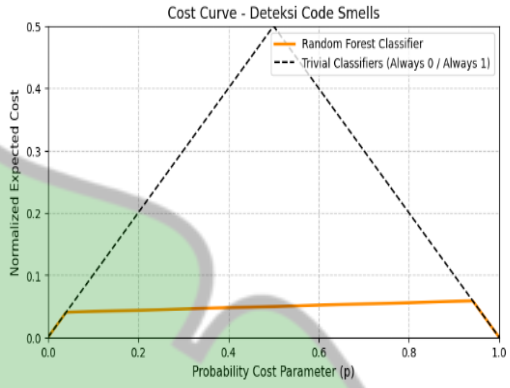
Selanjutnya, karena data uji dipertahankan dalam kondisi aslinya yang tidak seimbang (*imbalanced*) untuk evaluasi yang realistis, kurva *Precision-Recall* juga disajikan pada Gambar 7. Kurva *Precision-Recall* lebih informatif daripada kurva ROC ketika menghadapi data dengan distribusi kelas yang tidak seimbang, karena kurva ini lebih sensitif terhadap perubahan performa pada kelas minoritas (*Smells*) [9]. Kurva *Precision-Recall* pada Gambar 7 menunjukkan bahwa model tetap memiliki performa yang baik dalam mendeteksi kelas minoritas, dengan nilai presisi yang tinggi seiring dengan peningkatan *recall*.



Gambar 7. Kurva *Precision-Recall*

Gambar 7 menunjukkan kurva Precision-Recall dengan nilai PR-AUC sebesar 0.9299, yang berada jauh di atas garis *no-skill baseline* (0.45). Hal ini menegaskan bahwa meskipun data uji dipertahankan dalam kondisi tidak seimbang (*imbalanced*), model tetap mampu mendeteksi kelas minoritas (*Smells*) dengan presisi dan recall yang tinggi.

Evaluasi performa model juga diperkuat melalui analisis kurva biaya (*cost curve*) yang disajikan pada Gambar 8. Kurva biaya menggambarkan rata-rata biaya yang dinormalisasi (*normalized expected cost*) dari model pada berbagai tingkat probabilitas kelas positif (*probability cost parameter*, *p*) [9]. Kurva ini sangat berguna untuk mengevaluasi ketangguhan model terhadap berbagai skenario biaya kesalahan yang berbeda, terutama ketika kesalahan pada kelas minoritas (*Smells*) memiliki konsekuensi yang lebih serius daripada kesalahan pada kelas mayoritas (Normal).



Gambar 8. *Cost Curve*

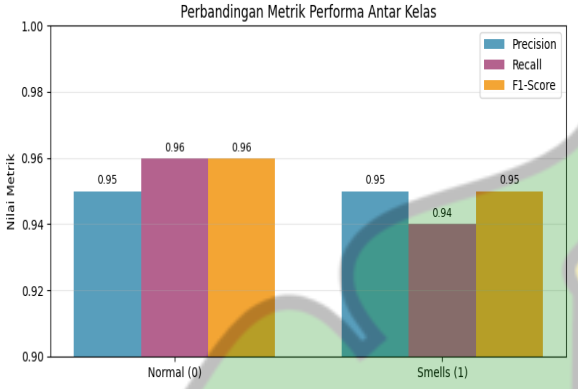
Pada Gambar 8, kurva berwarna oranye menunjukkan performa model Random Forest yang diusulkan, sementara garis putus-putus berwarna biru dan hijau masing-masing merepresentasikan pengklasifikasi trivial (*trivial classifiers*) yang selalu memprediksi kelas 0 (Normal) atau kelas 1 (*Smells*). Model yang baik ditandai dengan kurva biaya yang berada di bawah garis pengklasifikasi trivial pada seluruh rentang nilai *p*. Berdasarkan Gambar 8, terlihat bahwa kurva biaya model usulan berada jauh di bawah kedua garis acuan tersebut, yang mengindikasikan bahwa model memiliki kinerja yang stabil dan tidak sensitif terhadap perubahan biaya kesalahan. Temuan ini memperkuat hasil evaluasi sebelumnya bahwa model Random Forest + SMOTE mampu mempertahankan performa tinggi meskipun dihadapkan pada berbagai skenario ketidakseimbangan biaya klasifikasi.

Hasil pengolahan data pengujian matriks kebingungan menunjukkan bahwa performa model sangat stabil, konsisten, serta seimbang di antara kedua kategori kelas yang diuji. Model klasifikasi berbasis ensemble learning yang dikembangkan mampu mencatatkan nilai rata-rata presisi (*precision*), *recall*, dan *F1-score* yang mencapai angka 0.95 secara merata di seluruh metrik tanpa mengalami bias. Keunggulan stabilitas ini terkonfirmasi melalui metode perhitungan *macro average* maupun *weighted average*, yang menegaskan bahwa arsitektur pengklasifikasi ini memiliki tingkat generalisasi yang tinggi serta tangguh meskipun dilatih di bawah kondisi distribusi dataset yang memiliki variasi kompleksitas struktural serta paparan *noise* pelabelan alami. Rincian struktur persebaran performa pada masing-masing kategori kelas dirangkum pada Tabel 4 dan Gambar 9.

P-ISSN : 2089-3353
E-ISSN : 2808-9162

Tabel 4. Metrik Akurasi Nilai Klasifikasi Deteksi Kategori Kelas

Kategori Kelas	*Precision*	*Recall*	*F1-Score*	*Support*
Normal *(0)*	0.95	0.96	0.96	15.031
Smells (1)	0.95	0.94	0.95	12.515
Accuracy	-	-	0.95	27.546
Macro Avg	0.95	0.95	0.95	27.546
Weighted Avg	0.95	0.95	0.95	27.546



Gambar 9. Perbandingan Metrik Performa Antar Kelas

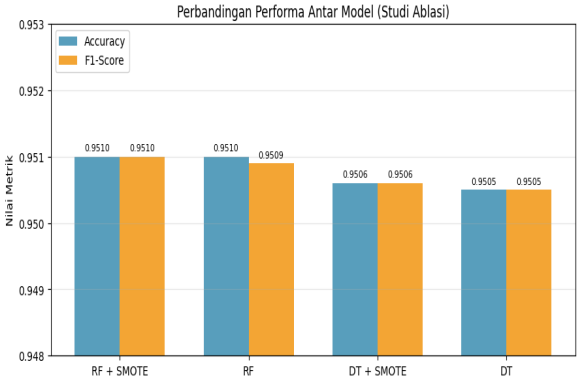
Gambar 9 menyajikan perbandingan nilai presisi, *recall*, dan *F1-score* antara kelas Normal (0) dan *Smells* (1). Kedua kelas menunjukkan nilai metrik yang hampir identik di angka 0.95, yang mengindikasikan bahwa model klasifikasi tidak mengalami bias terhadap salah satu kelas dan memiliki performa yang seimbang.

3.3. Studi Ablasi dan Perbandingan Komparatif Antar-Model

Untuk membuktikan kontribusi signifikan dari penambahan SMOTE serta memvalidasi keunggulan struktural *Random Forest,* sebuah pengujian komparatif berskala penuh dirancang sebagai bagian dari analisis studi ablasi. Skenario eksperimen ini membandingkan performa model usulan langsung terhadap arsitektur pengklasifikasi tunggal berupa algoritma *Decision Tree*. Selain membandingkan kinerja antar-algoritma secara langsung, studi ablasi juga berfungsi untuk memetakan pengaruh hilangnya komponen SMOTE terhadap proses pembelajaran masing-masing algoritma, terutama ketika dihadapkan pada kondisi ketidakseimbangan kelas yang bersifat organik dan disertai gangguan *noise* alami yang tidak terhindarkan. Seluruh hasil rekapitulasi data komparasi performa final dari keempat skenario model pengujian dirangkum dalam Tabel 5 dan Gambar 10.

Tabel 5. Perbandingan Performa Komparatif Antar-Model

Skenario Pengujian	*Accuracy*	*F1-Score*
Random Forest + SMOTE	0.9510	0.9510
Random Forest	0.9510	0.9509
Decision Tree + SMOTE	0.9506	0.9506
Decision Tree	0.9505	0.9505



Gambar 10. Perbandingan Metrik Performa Antar Kelas

Gambar 10 menyajikan perbandingan akurasi dan *F1-score* dari keempat skenario model yang diuji. Model usulan *Random Forest* + SMOTE menunjukkan performa tertinggi (0.9510) dibandingkan dengan tiga skenario lainnya. Visualisasi ini memperjelas kontribusi positif SMOTE dalam meningkatkan performa model, baik pada *Random Forest* maupun *Decision Tree.*

Analisis teoretis terhadap hasil komparasi performa pada Tabel 5 menyingkap beberapa temuan ilmiah yang krusial. Temuan pertama memperlihatkan supremasi arsitektur usulan, di mana kombinasi *Random Forest* dengan SMOTE berhasil menduduki urutan performa puncak dengan capaian akurasi global 0.9510 dan *F1-score* tertimbang 0.9510. Fakta empiris ini membuktikan bahwa penggabungan kekuatan penanganan varians tinggi dari metode *Bagging* yang tertanam di dalam *Random Forest* berkolaborasi secara optimal dengan minimalisasi bias kelas dari SMOTE, sehingga mampu menghasilkan batas keputusan (*decision boundary*) yang paling akurat.

Selanjutnya, hasil studi ablasi menunjukkan dampak positif penerapan SMOTE terhadap stabilitas model. Ketika komponen SMOTE dihilangkan dari *Random Forest,* nilai *F1-score* tertimbang menyusut dari 0.9510 menjadi 0.9509. Meskipun tipis akibat masifnya volume data asli, penurunan ini menegaskan bahwa penyeimbangan data sintetis berkontribusi positif dalam menjaga konsistensi deteksi pada kelas minoritas (*smells*). Pola serupa terjadi pada *Decision Tree,* di mana intervensi SMOTE mampu mendongkrak nilai akurasi dari 0.9505 menjadi 0.9506.

Di samping itu, ketangguhan model (*robustness*) terhadap gangguan menjadi poin evaluasi yang sangat menonjol dalam penelitian ini. Untuk menguji sejauh mana model klasifikasi mampu bertahan terhadap kondisi dunia nyata yang tidak ideal, *dataset* eksperimen sengaja dikotori dengan penyisipan *noise* pelabelan acak (*random label noise*) sebesar 5%. Tingkat *noise* ini dipilih secara khusus untuk merepresentasikan tingkat kesalahan subjektivitas manusia (*human error*) yang wajar terjadi dalam praktik pelabelan data di lapangan, mengingat penentuan suatu kode tergolong *code smell* atau tidak

P-ISSN : 2089-3353
E-ISSN : 2808-9162

sering kali bersifat interpretatif dan bergantung pada sudut pandang pengembang.

Meskipun gangguan ini berpotensi mengacaukan proses pembelajaran mesin dengan memberikan sinyal yang keliru, hasil pengujian menunjukkan bahwa nilai akurasi global dan F1-score seluruh varian model baik Random Forest maupun Decision Tree, dengan dan tanpa SMOTE tetap mampu bertahan stabil pada kisaran 95%. Stabilitas performa di bawah tekanan *noise* ini mengindikasikan bahwa algoritma berbasis pohon keputusan kolektif memiliki kemampuan generalisasi yang sangat baik dan tidak mudah terpengaruh oleh anomali data dalam jumlah kecil.

Fenomena ini membuktikan tingkat reliabilitas sistem yang luar biasa, di mana algoritma cerdas berbasis pohon keputusan kolektif terbukti mampu mengekstrak karakteristik struktural internal kode secara mandiri seperti pola kepadatan baris logika aktif, kompleksitas metode, dan rasio dokumentasi sekaligus secara adaptif mengabaikan distorsi anomali data yang tidak mencerminkan kondisi sebenarnya. Dengan kata lain, model tidak sekadar menghafal pola pelabelan yang diberikan, melainkan benar-benar memahami fitur-fitur mendasar yang membedakan kode bersih dari kode yang mengandung *smells*. Kemampuan ini sangat krusial untuk penerapan di dunia industri, di mana data pelatihan jarang sekali sempurna dan sering kali mengandung kesalahan pelabelan akibat keterbatasan waktu, perbedaan interpretasi, atau kurangnya dokumentasi standar.

3.4. Analisis Kontribusi Indikator Kontaminasi (*Feature Importance*)

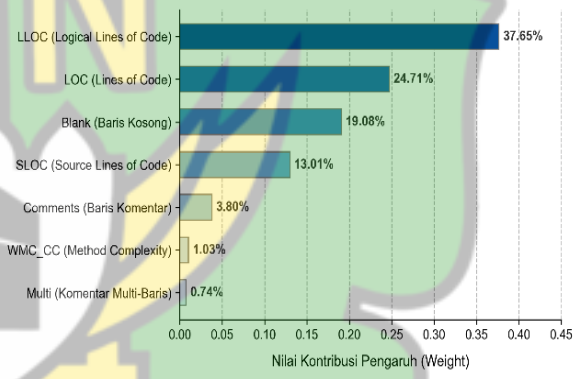
Untuk memberikan kontribusi praktis bagi komunitas rekayasa perangkat lunak, penelitian ini melakukan ekstraksi nilai penting fitur (*feature importance*) dari arsitektur pohon keputusan kolektif Random Forest. Analisis ini bertujuan mengidentifikasi indikator internal kode mana yang paling sensitif memicu degradasi kualitas berupa *code smells* pada arsitektur *backend* Python, sehingga pengembang dapat memprioritaskan upaya *refactoring* pada aspek-aspek kode yang paling berisiko. Pemahaman mengenai kontribusi relatif setiap metrik sangat berharga karena tidak semua indikator kode memiliki pengaruh yang sama terhadap kemunculan anomali struktural; dengan mengetahui metrik mana yang paling dominan, tim pengembang dapat menyusun strategi penulisan kode yang lebih disiplin dan terarah.

Proses ekstraksi tingkat kepentingan fitur pada algoritma *Random Forest* dilakukan dengan menghitung seberapa besar kontribusi setiap metrik dalam mengurangi tingkat ketidakmurnian (*impurity*) pada setiap percabangan yang ada di pohon keputusan, yang selanjutnya dirata-ratakan di seluruh pohon yang terdapat dalam hutan. Apabila nilai kontribusi suatu fitur semakin tinggi, maka semakin besar pula peran metrik tersebut dalam proses pemisahan antara entitas kode yang bersih yang mengandung *smells* selama proses klasifikasi berlangsung. Pendekatan kuantitatif yang digunakan ini menawarkan keunggulan tersendiri jika dibandingkan dengan metode tradisional yang masih bergantung pada intuisi maupun pengalaman subjektif dari pengembang. Hal ini dikarenakan hasil yang diperoleh melalui pendekatan ini sepenuhnya berbasis pada data dan dapat dipertanggungjawabkan secara statistik. Distribusi kontribusi dari ketujuh metrik yang diuji dipetakan secara terperinci pada Tabel 6.

Tabel 6. Peringkat Bobot Nilai Kontribusi Fitur Indikator Kode

Simbol Metrik Kode	Kontribusi (*Weight*)	Dampak (*%*)
LLOC	0.376451	37.65%
LOC	0.247064	24.71%
Blank	0.190799	19.08%
SLOC	0.130054	13.01%
Comments	0.038011	3.80%
WMC_CC	0.010263	1.03%
Multi (*Comments*)	0.007358	0.74%

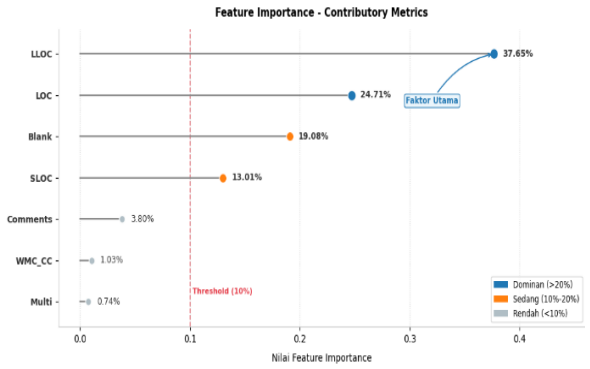
Sebaran nilai kontribusi pengaruh ini divisualisasikan dalam bentuk grafik batang horizontal pada Gambar 11.



Gambar 11. Grafik Batang *Feature Importance*

Berdasarkan urutan tingkat pengaruh pada Tabel 6, indikator LLOC (*Logical Lines of Code*) keluar sebagai faktor determinan paling krusial dengan nilai bobot pengaruh mencapai 37,65%. LLOC mengukur jumlah baris kode yang benar-benar mengeksekusi instruksi logika aktif, tidak termasuk baris kosong, komentar, atau deklarasi sederhana. Temuan ini menegaskan sifat khas dari pengembangan *backend* Python berbasis OOP, di mana penumpukan baris instruksi yang menjalankan komputasi aktif jauh lebih dominan memicu timbulnya *code smells* ketimbang sekadar ukuran fisik atau panjang kotor baris teks kode. Hal ini menunjukkan bahwa kompleksitas fungsional, bukan sekadar volume kode, yang menjadi pemicu utama degradasi struktural; semakin banyak logika yang dipadatkan dalam satu kelas atau metode, semakin besar kemungkinan terjadinya pelanggaran prinsip desain seperti *Single Responsibility Principle* atau *Open-Closed Principle*.

P-ISSN : 2089-3353
E-ISSN : 2808-9162



Gambar 12. *Lollipop Chart Feature Importance*

Gambar 12 menyajikan *lollipop chart* yang menampilkan kontribusi setiap metrik dengan cara yang lebih ringkas dan informatif. Pada *lollipop chart*, setiap metrik direpresentasikan sebagai titik (*lollipop*) dengan ukuran yang proporsional terhadap bobot kontribusinya, serta dilengkapi dengan nilai persentase di samping masing-masing titik. Metrik dengan kontribusi >20% (LLOC, LOC) dikategorikan sebagai faktor dominan, kontribusi 10-20% (*Blank*, SLOC) sebagai faktor sedang, dan kontribusi <10% (*Comments*, WMC_CC, *Multi*) sebagai faktor rendah. Garis *threshold* 10% pada Gambar 12 memisahkan secara jelas metrik yang memiliki pengaruh substansial terhadap keputusan klasifikasi model. Visualisasi ini memperkuat temuan bahwa LLOC merupakan indikator paling krusial dengan kontribusi 37,65%, diikuti oleh LOC (24,71%), Blank (19,08%), dan SLOC (13,01%).

Indikator LOC (*Lines of Code*) berada di peringkat kedua dengan pengaruh 24,71%. Meskipun LOC mengukur total baris fisik kode tanpa membedakan antara logika aktif, deklarasi, atau dokumentasi, metrik ini tetap memiliki korelasi yang cukup kuat terhadap munculnya *code smells* karena secara kasar mencerminkan ukuran absolut suatu entitas kode. Namun, bobotnya yang lebih rendah dibandingkan LLOC mengindikasikan bahwa ukuran fisik saja tidak cukup untuk menjelaskan risiko struktural; sebuah kelas dapat memiliki banyak baris kode karena dokumentasi yang panjang atau deklarasi atribut yang banyak, tanpa serta-merta mengalami penurunan kualitas desain yang signifikan.

Indikator baris kosong (*Blank*) menempati posisi ketiga dengan pengaruh 19,08%, yang merepresentasikan kebiasaan pengembang dalam memberikan pembatas visual antarbagian kode. Fenomena ini menarik karena secara tidak langsung mencerminkan ukuran internal suatu fungsi atau kelas; semakin banyak baris kosong yang digunakan, semakin besar kemungkinan struktur internalnya telah berkembang menjadi terlalu gemuk dan mulai kehilangan kohesi. Dengan kata lain, baris kosong berfungsi sebagai penanda visual dari pembagian logika yang seharusnya dipecah menjadi entitas-entitas yang lebih kecil dan terfokus, sebagaimana dianjurkan oleh prinsip tanggung jawab tunggal (*Single Responsibility Principle*).

Sebaliknya, metrik dokumentasi seperti *Comments* dan *Multi Comments* mencatatkan pengaruh yang sangat rendah, masing-masing di bawah 4%. Temuan ini mengindikasikan bahwa kehadiran teks penjelas dalam kode tidak memiliki korelasi struktural langsung terhadap pembentukan anomali logika sistem. Meskipun dokumentasi sangat penting untuk keterbacaan dan pemeliharaan kode dari perspektif manusia, metrik ini tidak cukup sensitif untuk digunakan sebagai indikator prediktif terhadap keberadaan *code smell*. Dengan demikian, pengembang tidak perlu mengkhawatirkan bahwa penambahan komentar akan meningkatkan Risiko *code smells*, namun juga tidak dapat mengandalkan dokumentasi sebagai tolak ukur kualitas struktural kode.

4. Kesimpulan

Berdasarkan seluruh rangkaian eksperimen dalam pengembangan model klasifikasi otomatis *code smells* pada ekosistem *backend* Python, dapat disimpulkan bahwa integrasi algoritma Random Forest dan SMOTE berhasil menghasilkan model klasifikasi yang unggul dan adaptif, dengan capaian rata-rata *precision*, *recall*, dan F1-score sebesar 0.95 secara merata pada evaluasi *macro* maupun *weighted average* melalui pengujian terhadap 27.546 sampel data uji.

Studi ablasi membuktikan bahwa SMOTE mampu menjaga konsistensi klasifikasi pada kelas minoritas, sementara model menunjukkan ketangguhan dengan tetap mempertahankan akurasi tinggi di kisaran 95% meskipun diuji di bawah pengaruh *random noise* 5%. Analisis *feature importance* mengidentifikasi LLOC sebagai faktor determinan paling krusial dengan bobot pengaruh 37,65%, disusul LOC (24,71%) dan Blank (19,08%), yang menegaskan bahwa pada arsitektur *backend* Python OOP, penumpukan baris instruksi logika aktif jauh lebih sensitif menurunkan kualitas kode dibandingkan metrik dokumentasi. Implikasi praktis penelitian ini memberikan kontribusi bagi pengembang dalam mendeteksi lokasi *refactoring* sejak dini, menekan pembengkakan *technical debt*, dan mempermudah pemeliharaan sistem jangka panjang. Untuk penelitian selanjutnya, disarankan untuk menguji model pada bahasa pemrograman lain seperti Java atau C++, mengeksplorasi algoritma *deep learning* seperti LSTM atau CNN, serta menambahkan variabel metrik tambahan untuk meningkatkan akurasi prediksi.

Daftar Rujukan

[1] T. Lewowski and L. Madeyski, "How far are we from reproducible research on code smell detection? A systematic literature review," *Inf. Softw. Technol.*, vol. 144, no. November 2021, p. 106783, 2022, doi: 10.1016/j.infsof.2021.106783.

[2] P. S. Yadav, R. S. Rao, A. Mishra, and M. Gupta, "Machine Learning-Based Methods for Code Smell Detection: A Survey," *Appl. Sci.*, vol. 14, no. 14, 2024, doi:

P-ISSN : 2089-3353
E-ISSN : 2808-9162

10.3390/app14146149.

[3] M. Hadj-Kacem and N. Bouassida, "Application of Deep Learning for Code Smell Detection: Challenges and Opportunities," SN Comput. Sci., vol. 5, p. 553, 2024, doi: 10.1007/s42979-024-02956-5.

[4] P. S. Thakur, S. S. Chouhan, and S. S. Rathore, "Systematic literature review on software code smell detection approaches," J. Syst. Softw., vol. 222, p. 112345, 2026, doi: 10.1016/j.jss.2026.112784.

[5] A. Abdou and N. Darwish, "Severity classification of software code smells using machine learning techniques: A comparative study," J. Softw. Evol. Process, vol. 36, no. 1, p. e2454, 2024, doi: 10.1002/smr.2454.

[6] A. Alazba and H. Aljamaan, "Code smell detection using feature selection and stacking ensemble: An empirical investigation," *Inf. Softw. Technol.*, vol. 138, no. May, p. 106648, 2021, doi: 10.1016/j.infsof.2021.106648.

[7] S. Dewangan, R. S. Rao, A. Mishra, and M. Gupta, "A novel approach for code smell detection: An empirical study," *IEEE Access*, vol. 9, pp. 162869–162883, 2021, doi: 10.1109/ACCESS.2021.3133810.

[8] S. Dewangan, R. S. Rao, A. Mishra, and M. Gupta, "Code Smell Detection Using Ensemble Machine Learning Algorithms," Appl. Sci., vol. 12, no. 20, p. 10321, 2022, doi: 10.3390/app122010321.

[9] R. S. Rao, S. Dewangan, A. Mishra, and M. Gupta, "A study of dealing class imbalance problem with machine learning methods for code smell severity detection using PCA-based feature selection technique," *Sci. Rep.*, vol. 13, no. 1, pp. 1–18, 2023, doi: 10.1038/s41598-023-43380-8.

[10] T. Guggulothu and S. A. Moiz, "Code smell detection using multi-label classification approach," *Softw. Qual. J.*, vol. 28, no. 3, pp. 1063–1086, 2020, doi: 10.1007/s11219-020-09498-y.

[11] F. do R. Santos and R. Choren, "Data preprocessing for machine learning based code smell detection: A systematic literature review," *Inf. Softw. Technol.*, vol. 184, p. 107752, 2025, doi: 10.1016/j.infsof.2025.107752.

[12] K. Alkharabsheh, S. Alawadi, Y. Crespo, and J. A. Taboada, "Exploring the role of project status information in effective code smell detection," *Cluster Comput.*, vol. 28, no. 1, 2025, doi: 10.1007/s10586-024-04724-9.

[13] M. Zakeri-Nasrabadi, S. Parsa, E. Esmaili, and F. Palomba, "A Systematic Literature Review on the Code Smells Datasets and Validation Mechanisms," *ACM Comput. Surv.*, vol. 55, no. 13 s, 2023, doi: 10.1145/3596908.