IF3170 Inteligensi Artifisial
Implementasi Algoritma Pembelajaran Mesin

Tugas Besar 2

Disusun untuk memenuhi tugas mata kuliah Inteligensi Artifisial untuk
Semester 5 tahun ajaran 2024 / 2025

Oleh

Aland Mulia Pratama	**13522124**
Rizqika Mulia Pratama	**13522126**
Christian Justin Hendrawan	**13522135**
Auralea Alvinia Syaikha	**13522148**

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG

2024

DAFTAR ISI

BAB I
DESKRIPSI MASALAH

Gambar 1. Gambaran Dataset untuk Implementasi Algoritma Pembelajaran Mesin

Pembelajaran mesin merupakan cabang dari kecerdasan buatan yang memungkinkan sistem untuk belajar dari data guna membuat prediksi atau keputusan tanpa memerlukan program eksplisit. Dalam tugas ini, dilakukan implementasi algoritma pembelajaran mesin menggunakan dataset UNSW-NB15, yang berisi data lalu lintas jaringan mencakup berbagai jenis serangan siber serta aktivitas normal.

Tugas besar ini melibatkan implementasi algoritma **K-Nearest Neighbors (KNN)**, **Gaussian Naive-Bayes**, dan **ID3 (Iterative Dichotomiser 3)** dari awal (from scratch), berdasarkan spesifikasi berikut:

1. **KNN** from scratch:
    - Menerima minimal dua parameter: jumlah tetangga dan metrik jarak.
    - Mendukung tiga jenis metrik jarak: Euclidean, Manhattan, dan Minkowski.
2. **Gaussian Naive-Bayes** from scratch.
3. **ID3 from scratch**, termasuk pengolahan data numerik sesuai materi kuliah.

Selain implementasi dari awal, algoritma tersebut juga harus diimplementasikan menggunakan pustaka scikit-learn. Untuk algoritma ID3 dalam scikit-learn, digunakan DecisionTreeClassifier dengan parameter criterion='entropy' karena mendekati ID3. Hasil dari implementasi from scratch dan scikit-learn akan dibandingkan.

Setiap model yang dibuat harus dapat disimpan dan dimuat kembali menggunakan format penyimpanan tertentu (misalnya, .txt atau .pkl). Selain itu, hasil prediksi dapat dikirimkan ke platform Kaggle untuk evaluasi dengan bonus diberikan berdasarkan peringkat leaderboard.

Tahapan dalam menyelesaikan tugas ini meliputi:

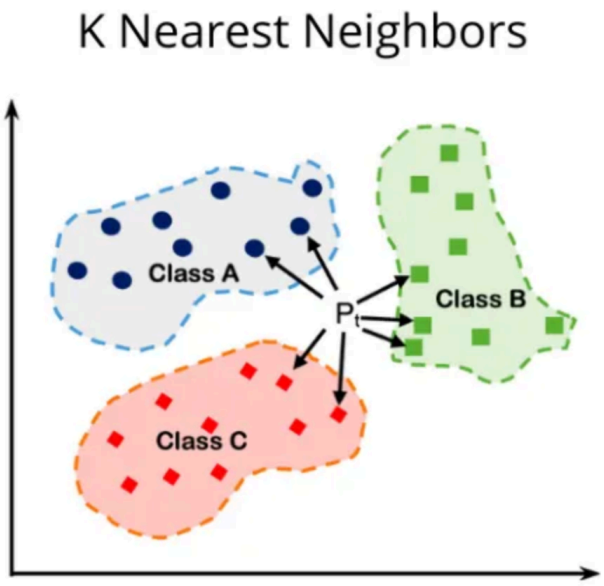
1. **Pembersihan Data:** Membersihkan data dari nilai hilang, duplikasi, atau entri tidak valid.

2. **Transformasi Data:** Melibatkan proses seperti encoding variabel kategori, normalisasi atau standarisasi fitur numerik, dan penanganan ketidakseimbangan data.
3. **Pemilihan Fitur:** Mengidentifikasi fitur relevan untuk meningkatkan kinerja model.
4. **Reduksi Dimensi:** Mengurangi jumlah fitur tanpa kehilangan informasi penting, misalnya menggunakan PCA.
5. **Modeling dan Validasi:** Melatih dan memvalidasi model untuk mengklasifikasi kategori serangan (attack_cat), menggunakan metode validasi seperti train-test split atau k-fold cross-validation.

Notebook yang dikumpulkan harus dapat mereproduksi hasil prediksi yang konsisten dengan submisi Kaggle, di mana hanya model KNN, Naive Bayes, dan ID3 hasil implementasi *from scratch* yang diperbolehkan digunakan.

BAB II
PEMBAHASAN

2.1. Implementasi Algoritma KNN



Gambar 2. Algoritma KNN

Algoritma K-Nearest Neighbors (KNN) adalah algoritma pembelajaran mesin supervised yang digunakan untuk klasifikasi dan regresi. KNN bekerja dengan cara mencari titik data yang paling dekat (nearest neighbors) dengan data uji (test data) berdasarkan suatu ukuran jarak, seperti Euclidean atau Manhattan, lalu mengklasifikasikan data uji berdasarkan kelas atau nilai mayoritas dari tetangga terdekat tersebut.

1. **Inisialisasi Kelas KNN (__init__)**

```
    def __init__(self, k=5, n_jobs=1, metric='minkowski', p=2,
weights='uniform', verbose=True):
        if k < 1 or not isinstance(k, int):
            raise ValueError("k must be an integer greater than 0.")
        if metric not in ['manhattan', 'euclidean', 'minkowski']:
            raise ValueError("Invalid metric. Choose from: 'manhattan',
'euclidean', 'minkowski'.")
        if weights not in ['uniform', 'distance']:
            raise ValueError("weights must be either 'uniform' or
'distance'.")
        if n_jobs < 1 and n_jobs != -1:
            raise ValueError("n_jobs must be greater than 0 or -1 to
use all available cores.")

        self.k = k
        self.metric = metric
        self.p = p
        self.weights = weights
        self.verbose = verbose
```

```
        self.n_jobs = cpu_count() if n_jobs == -1 else n_jobs
```

Pada bagian **inisialisasi parameter** dari algoritma KNN, terdapat beberapa parameter yang penting untuk mengkonfigurasi model. k adalah jumlah **tetangga terdekat** yang akan dipertimbangkan untuk melakukan prediksi, dengan nilai default 5. n_jobs mengatur berapa banyak core CPU yang akan digunakan untuk komputasi paralel. Jika nilainya -1, maka algoritma akan memanfaatkan semua core yang tersedia di mesin. *Metric* menentukan ukuran jarak yang digunakan untuk mengukur kedekatan antara data pelatihan dan data uji. Terdapat tiga opsi yang tersedia, yaitu 'euclidean', 'manhattan', dan 'minkowski'. Jika menggunakan *minkowski*, parameter *p* menentukan eksponen dalam perhitungan jarak (dengan nilai default 2, yang berarti menggunakan jarak Euclidean). weights mengatur cara pemberian bobot pada tetangga terdekat: 'uniform' memberikan bobot yang sama rata untuk semua tetangga, sedangkan 'distance' memberikan bobot yang lebih besar pada tetangga yang lebih dekat. Terakhir, parameter verbose memungkinkan untuk menampilkan informasi proses yang berjalan, seperti jumlah core yang digunakan atau waktu yang diperlukan untuk melakukan prediksi.

2. Menghitung Jarak antara Titik Uji dan Data Pelatihan (_compute_distances)

```
 def _compute_distances(self, test_instance):
        """
        Compute distances between the test instance and all training
data points.
        """
        if self.metric == 'euclidean':
            distances = np.sqrt(np.sum((self.X_train - test_instance)
** 2, axis=1))
        elif self.metric == 'manhattan':
            distances = np.sum(np.abs(self.X_train - test_instance),
axis=1)
        elif self.metric == 'minkowski':
            distances = np.sum(np.abs(self.X_train - test_instance) **
self.p, axis=1) ** (1 / self.p)
        else:
            raise ValueError("Unsupported distance metric.")
        return distances
```

Metode **_compute_distances** digunakan untuk menghitung jarak antara setiap titik uji (**test_instance**) dan seluruh data pelatihan (**X_train**) dengan menggunakan salah satu dari tiga **metrik jarak** yang tersedia. **Jarak Euclidean** dihitung berdasarkan **rumus Pythagoras**, yaitu akar kuadrat dari jumlah kuadrat perbedaan antara koordinat titik uji dan titik pelatihan. Ini adalah metrik yang paling umum digunakan untuk mengukur kedekatan antar titik dalam ruang multidimensi. **Jarak Manhattan**, di sisi lain, dihitung dengan **menjumlahkan selisih mutlak** antara koordinat-koordinat titik uji dan titik pelatihan, yang sering disebut juga sebagai "jarak kota blok" karena cara perhitungannya mirip dengan pergerakan di jalan-jalan kota yang membentuk grid. Sementara itu, **Jarak Minkowski** adalah **generalisasi** dari kedua metrik tersebut, yang melibatkan parameter **p**. Jika nilai **p = 2**, maka jarak Minkowski akan setara dengan jarak Euclidean, sedangkan

jika **p = 1**, maka hasilnya akan menjadi jarak Manhattan. Parameter **p** memberikan fleksibilitas untuk menyesuaikan jenis pengukuran jarak yang lebih sesuai dengan karakteristik data yang digunakan.

3. **Mencari K-Tetangga Terdekat (_get_k_nearest_neighbors)**

```
    def _get_k_nearest_neighbors(self, distances):
        """
        Find the k-nearest neighbors and their corresponding indices
and weights.
        """
        k_indices = np.argpartition(distances, self.k)[:self.k]
        k_distances = distances[k_indices]

        if self.weights == 'distance':
            # Avoid division by zero by replacing zero distances with a
small value
            k_weights = 1 / (k_distances + 1e-10)
        else:
            k_weights = np.ones_like(k_distances)

        return k_indices, k_weights
```

Setelah jarak antara titik uji dan data pelatihan dihitung, metode ini memilih **k tetangga terdekat** berdasarkan urutan jarak terpendek. Untuk menentukan k tetangga terdekat, digunakan fungsi *np.argpartition*, yang mengurutkan indeks jarak dan memilih k indeks teratas dengan jarak terpendek. Proses ini memastikan bahwa hanya tetangga yang paling dekat yang dipertimbangkan dalam prediksi. Terkait dengan **bobot tetangga** (weights), terdapat dua opsi yang dapat dipilih: 'uniform' dan 'distance'. Jika menggunakan 'uniform', semua tetangga terdekat akan diberi bobot yang sama rata (yaitu, bobot 1 untuk setiap tetangga), sehingga semua tetangga dianggap setara dalam pengaruhnya terhadap prediksi. Di sisi lain, jika menggunakan 'distance', tetangga yang lebih dekat akan diberikan bobot yang lebih besar, dengan bobot yang dihitung sebagai **invers jarak** (semakin kecil jarak, semakin besar bobotnya), sehingga memprioritaskan pengaruh tetangga yang lebih dekat dalam proses klasifikasi atau regresi.

4. **Fungsi Fit**

```
    def fit(self, X_train, y_train):
        """
        Store the training data and labels.
        """
        self.X_train = np.array(X_train, dtype=np.float32)
        self.y_train = np.array(y_train)
```

Pada metode fit, data pelatihan (X_train dan y_train) disimpan di objek model. Data pelatihan ini akan digunakan nantinya saat melakukan prediksi.

5. **Prediksi untuk Satu Titik Uji (_predict_instance)**

```
    def _predict_instance(self, test_instance):
        """
```

```
        Predict the class label for a single test instance.
        """
        distances = self._compute_distances(test_instance)
        k_indices, k_weights = self._get_k_nearest_neighbors(distances)

        k_labels = self.y_train[k_indices]
        if self.weights == 'uniform':
            # Majority voting
            unique_labels, counts = np.unique(k_labels,
return_counts=True)
            prediction = unique_labels[np.argmax(counts)]
        else:  # 'distance'
            # Weighted voting
            label_weights = {}
            for label, weight in zip(k_labels, k_weights):
                label_weights[label] = label_weights.get(label, 0) +
weight
            prediction = max(label_weights, key=label_weights.get)

        return prediction
```

Pertama, dihitung jarak antara titik uji dan setiap titik pelatihan. Kemudian, ditemukan k tetangga terdekat menggunakan jarak yang dihitung. Untuk setiap tetangga, dilakukan pemungutan suara untuk menentukan label kelas yang paling sering muncul. Jika bobot 'distance' digunakan, pemungutan suara dilakukan dengan mempertimbangkan bobot, sehingga tetangga yang lebih dekat memiliki pengaruh lebih besar.

6. Prediksi untuk Banyak Titik Uji (predict)

```
    def predict(self, X_test):
        """
        Predict class labels for all test instances in parallel.
        """
        X_test = np.array(X_test, dtype=np.float32)

        if self.verbose:
            print(f"Making predictions using {self.n_jobs} {'core' if
self.n_jobs == 1 else 'cores'}.")

        start_time = time.time()

        with
concurrent.futures.ThreadPoolExecutor(max_workers=self.n_jobs) as
executor:
            if self.verbose:
                results =
list(tqdm(executor.map(self._predict_instance, X_test),
total=len(X_test)))
            else:
                results = list(executor.map(self._predict_instance,
X_test))

        elapsed_time = time.time() - start_time
        if self.verbose:
            print(f"Predictions completed in {elapsed_time:.2f}
seconds.")

        return np.array(results)
```

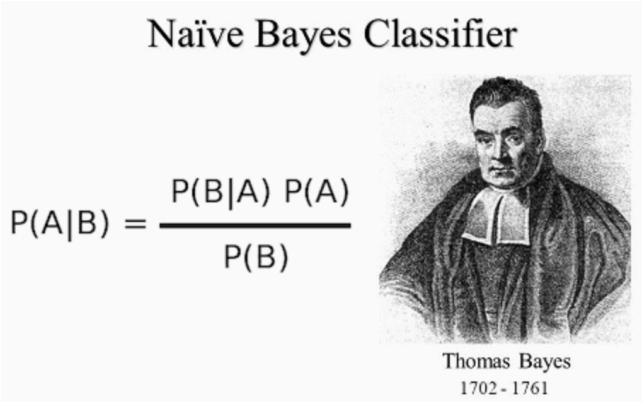
Metode ini memungkinkan prediksi untuk banyak titik uji secara paralel, memanfaatkan ThreadPoolExecutor untuk melakukan komputasi di banyak core CPU (tergantung pada parameter n_jobs). Jika verbose=True, maka proses prediksi akan dibarengi dengan tampilan progres yang menunjukkan berapa banyak prediksi yang sudah selesai menggunakan tqdm.

7. Menyimpan dan Memuat Model

```
def save(self, path):
    """
    Save the trained KNN model to a file.
    """
    with open(path, 'wb') as f:
        pickle.dump(self, f)

@staticmethod
def load(path):
    """
    Load a trained KNN model from a file.
    """
    with open(path, 'rb') as f:
        return pickle.load(f)
```

Model yang telah dilatih dapat disimpan ke file menggunakan metode save. Metode ini menggunakan pickle untuk menyimpan seluruh objek model KNN. Metode load memungkinkan untuk memuat model yang sudah dilatih dari file yang disimpan.

2.2. Implementasi Algoritma Naive-Bayes



Gambar 3. Algoritma Naive Bayes Classifier

Algoritma Gaussian Naive Bayes (GNB) adalah salah satu varian dari Naive Bayes yang mengasumsikan bahwa fitur-fitur dalam dataset mengikuti distribusi normal (Gaussian). GNB biasanya digunakan untuk klasifikasi, di mana kita mencoba untuk memprediksi kelas suatu sampel berdasarkan nilai fitur-fitur yang diberikan. Dalam implementasi ini, distribusi setiap fitur untuk setiap kelas diasumsikan mengikuti distribusi normal (Gaussian), dengan parameter distribusi tersebut adalah rata-rata (mean) dan varians (variance) dari fitur-fitur tersebut.

1. **Fungsi fit**

```
def fit(self, X, y):
    # hitung mean, variance, dan prior probability untuk setiap kelas
    self.classes = np.unique(y)
    n_features = X.shape[1]
    n_classes = len(self.classes)

    self.mean = np.zeros((n_classes, n_features), dtype=np.float64)
    self.variance = np.zeros((n_classes, n_features), dtype=np.float64)
    self.prior = np.zeros(n_classes, dtype=np.float64)

    for idx, c in enumerate(self.classes):
        X_c = X[y == c]
        self.mean[idx, :] = X_c.mean(axis=0)
        self.variance[idx, :] = X_c.var(axis=0)
        self.prior[idx] = X_c.shape[0] / float(X.shape[0])
```

Fungsi fit digunakan untuk melatih model dengan data pelatihan X (fitur) dan y (label kelas). Proses pelatihan terdiri dari beberapa langkah:

- Menentukan kelas-kelas unik dalam dataset target y menggunakan np.unique(y).
- Menghitung mean, variance, dan prior probability untuk setiap kelas:
  - ❖ Mean dihitung sebagai rata-rata fitur untuk setiap kelas.
  - ❖ Variance dihitung sebagai varians fitur untuk setiap kelas.
  - ❖ Prior probability dihitung sebagai proporsi data yang termasuk dalam kelas tertentu.
  - ❖ Setelah pelatihan, model akan memiliki informasi tentang distribusi fitur untuk setiap kelas, serta prior probabilitas kelas tersebut.

2. **Fungsi _gaussian_density**

```
def _gaussian_density(self, class_idx, x):
    # hitung Gaussian probability density function
    mean = self.mean[class_idx]
    var = self.variance[class_idx]
    numerator = np.exp(- (x - mean) ** 2 / (2 * var))
    denominator = np.sqrt(2 * np.pi * var)
    return numerator / denominator
```

Fungsi ini digunakan untuk menghitung probabilitas densitas fungsi Gaussian untuk suatu titik data pada fitur tertentu. Fungsi Gaussian adalah:

$$P(x) = \frac{1}{\sqrt{2\pi\sigma^2}} exp(-\frac{(x-\mu)^2}{2\sigma^2})$$

Di mana:

- x adalah nilai fitur yang sedang dipertimbangkan.
- μ adalah rata-rata (mean) dari fitur untuk kelas tertentu.
- σ^2 adalah varians dari fitur untuk kelas tersebut.

Fungsi ini menghitung probabilitas untuk setiap fitur pada kelas tertentu dengan menggunakan rumus di atas.

3. **Fungsi _posterior**

```
def _posterior(self, x):
    # hitung probabilitas posterior untuk setiap kelas
    posteriors = []
    for idx, c in enumerate(self.classes):
        prior = np.log(self.prior[idx])
        class_conditional = np.sum(np.log(self._gaussian_density(idx, x)))
        posterior = prior + class_conditional
        posteriors.append(posterior)
    return self.classes[np.argmax(posteriors)]
```

Fungsi ini menghitung probabilitas posterior untuk suatu titik data dengan membandingkan probabilitas dari masing-masing kelas. Berdasarkan Teorema Bayes, probabilitas posterior untuk kelas C_k diberikan oleh:

$$P(C_k|x) = \frac{P(x|C_k)P(C_k)}{P(x)}$$

Di sini, $P(x|C_k)$ adalah likelihood (probabilitas fitur diberikan kelas), dan $P(C_k)$ adalah probabilitas prior dari kelas. Fungsi ini menghitung logaritma dari prior dan likelihood (karena log lebih mudah dihitung dan menghindari masalah underflow) dan memilih kelas dengan probabilitas posterior tertinggi.

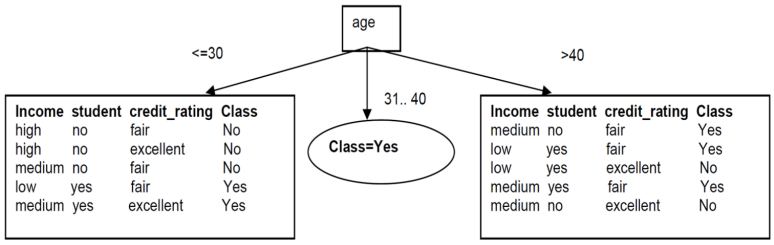
4. **Fungsi predict**

Fungsi predict digunakan untuk memprediksi kelas dari suatu set data X (fitur). Fungsi ini memanggil fungsi _posterior untuk setiap sampel dalam X, dan mengembalikan kelas yang memiliki probabilitas posterior tertinggi.

5. **Fungsi predict_proba**

Fungsi ini mengembalikan probabilitas kelas untuk setiap sampel dalam X. Ia menghitung distribusi Gaussian untuk setiap kelas dan mengalikan dengan prior kelas untuk menghasilkan probabilitas masing-masing kelas. Namun, dalam implementasi ini, ada kesalahan kecil karena fungsi ini tidak mengembalikan probabilitas yang ter-normalisasi, yang akan lebih berguna dalam beberapa aplikasi.

2.3.Implementasi ID3



Gambar 4. Algoritma Decision Tree Learning

Algoritma ID3 adalah salah satu algoritma pembelajaran mesin yang digunakan untuk membuat pohon keputusan. Pohon keputusan adalah struktur seperti pohon yang digunakan untuk

membuat keputusan berdasarkan serangkaian aturan yang berasal dari data pelatihan. Algoritma ID3 membangun pohon keputusan dengan menggunakan konsep entropi dan informasi gain untuk memilih fitur terbaik yang memisahkan data pada setiap simpul pohon.

1. **Kelas Node**

```python
class Node:
    def __init__(self, feature=None, threshold=None, left=None, right=None, value=None):
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value
```

Kelas Node digunakan untuk merepresentasikan simpul dalam pohon keputusan. Setiap simpul dalam pohon keputusan dapat berupa simpul internal atau simpul daun. Berikut adalah atribut-atribut kelas Node :

a. **feature :** Indeks fitur yang digunakan untuk membagi data pada simpul ini. Jika simpul ini adalah simpul daun, maka feature bernilai None.
b. **threshold** : Nilai ambang batas untuk membagi data berdasarkan fitur yang dipilih. Data yang nilainya kurang dari atau sama dengan threshold akan pergi ke simpul kiri (left), sedangkan yang lebih besar akan pergi ke simpul kanan (right).
c. **left** : Referensi ke simpul anak kiri. Ini adalah hasil pembagian data yang memenuhi kondisi feature <= threshold
d. **right** : Referensi ke simpul anak kanan. Ini adalah hasil pembagian data yang tidak memenuhi kondisi feature <= threshold.
e. **value** : Nilai prediksi pada simpul daun. Jika simpul ini adalah simpul daun, maka value berisi nilai prediksi. Jika simpul ini adalah simpul internal, maka value bernilai None.

2. **Kelas CustomLabelEncoder**

```python
class CustomLabelEncoder:
    def __init__(self):
        self.classes_ = None
        self.class_to_idx = {}

    def fit_transform(self, y):
        unique_classes = list(set(y))
        self.classes_ = unique_classes
        self.class_to_idx = {c: i for i, c in enumerate(unique_classes)}
        return np.array([self.class_to_idx[yi] for yi in y])

    def inverse_transform(self, y):
        return np.array([self.classes_[yi] for yi in y])
```

Kelas CustomLabelEncoder adalah sebuah kelas kustom yang digunakan untuk mengubah label kategori menjadi label numerik dan sebaliknya. Kelas ini mirip dengan LabelEncoder yang disediakan oleh scikit-learn, namun diimplementasikan secara

manual untuk kebutuhan khusus. Berikut adalah penjelasan detail dari setiap bagian kelas ini:

a. **fit_transform** : Metode ini mengubah label kategori menjadi label numerik. Pertama, metode ini menemukan kelas-kelas unik dalam y dan menyimpannya dalam atribut classes_. Kemudian, metode ini membuat kamus class_to_idx yang memetakan setiap kelas ke indeks numeriknya. Akhirnya, metode ini mengembalikan array dari label numerik yang sesuai dengan label kategori dalam y.
b. **inverse_transform** : Metode ini mengubah label numerik kembali menjadi label kategori. Metode ini menggunakan atribut classes_ untuk menemukan nama kelas yang sesuai dengan setiap indeks numerik dalam y.

3. Kelas ID3DecisionTree

```
class ID3DecisionTree:
    def __init__(self, max_depth=10, min_samples_split=2):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.root = None
        self.label_encoder = CustomLabelEncoder()
```

Kelas ID3DecisionTree digunakan untuk mengimplementasikan algoritma ID3 dalam membangun pohon keputusan. Berikut adalah atribut-atribut dalam kelas ini:

a. **max_depth**: Kedalaman maksimum pohon keputusan. Ini membatasi seberapa dalam pohon dapat tumbuh. Misalnya, jika max_depth diatur ke 10, pohon tidak akan memiliki lebih dari 10 tingkat simpul.
b. **min_samples_split**: Jumlah minimum sampel yang diperlukan untuk membagi simpul. Jika jumlah sampel di simpul kurang dari nilai ini, simpul tersebut tidak akan dibagi lebih lanjut.
c. **root**: Akar dari pohon keputusan. Ini adalah simpul pertama dari pohon yang akan dibangun. Awalnya, root bernilai None dan akan diatur saat pohon dibangun.
d. **label_encoder**: Encoder yang digunakan untuk mengubah label kategorikal menjadi numerik. Ini membantu dalam menangani label yang berbentuk teks sehingga dapat digunakan dalam perhitungan.

4. Fungsi entropy

```
def entropy(self, y):
    _, counts = np.unique(y, return_counts=True)
    ps = counts / len(y)
    return -(ps * np.log2(ps)).sum()
```

Fungsi entropy digunakan untuk menghitung entropi dari label y. Entropi adalah ukuran ketidakpastian atau ketidakteraturan dalam data. Fungsi ini pertama-tama menghitung jumlah kemunculan unik dalam y, kemudian menghitung probabilitas setiap nilai unik.

5. Fungsi find_best_split

```python
def find_best_split(self, X, y):
    best_gain = -1
    best_feature = None
    best_threshold = None

    n_features = X.shape[1]

    # Vectorized computation for each feature
    for feature_idx in range(n_features):
        thresholds = np.percentile(X[:, feature_idx], [25, 50, 75])
        for threshold in thresholds:
            left_mask = X[:, feature_idx] <= threshold
            if left_mask.sum() < self.min_samples_split or (~left_mask).sum() < self.min_samples_split:
                continue

            parent_entropy = self.entropy(y)
            left_entropy = self.entropy(y[left_mask])
            right_entropy = self.entropy(y[~left_mask])

            n = len(y)
            n_l, n_r = left_mask.sum(), (~left_mask).sum()
            gain = parent_entropy - (n_l/n * left_entropy + n_r/n * right_entropy)

            if gain > best_gain:
                best_gain = gain
                best_feature = feature_idx
                best_threshold = threshold

    return best_feature, best_threshold
```

Fungsi find_best_split digunakan untuk menemukan fitur dan threshold terbaik untuk membagi data. Fungsi ini menginisialisasi variabel untuk menyimpan informasi gain terbaik, fitur terbaik, dan threshold terbaik. Kemudian, fungsi ini melakukan iterasi melalui setiap fitur dan menentukan threshold potensial

6. **Fungsi build_tree**

```
def build_tree(self, X, y, depth=0):
    n_samples, _ = X.shape

    if (self.max_depth is not None and depth >= self.max_depth) or \
       n_samples < self.min_samples_split or \
       len(np.unique(y)) == 1:
        return Node(value=Counter(y).most_common(1)[0][0])

    feature, threshold = self.find_best_split(X, y)

    if feature is None:
        return Node(value=Counter(y).most_common(1)[0][0])

    left_mask = X[:, feature] <= threshold

    left = self.build_tree(X[left_mask], y[left_mask], depth + 1)
    right = self.build_tree(X[~left_mask], y[~left_mask], depth + 1)

    return Node(feature, threshold, left, right)
```

Fungsi build_tree digunakan untuk membangun pohon keputusan secara rekursif. Fungsi ini memeriksa kondisi penghentian seperti kedalaman maksimum, jumlah sampel minimum, atau label seragam. Jika kondisi penghentian terpenuhi, fungsi ini mengembalikan simpul daun dengan nilai prediksi yang paling umum. Jika tidak, fungsi ini menemukan fitur dan threshold terbaik untuk membagi data, membagi data menjadi dua bagian berdasarkan threshold, dan membangun simpul anak kiri dan kanan secara rekursif.

7. **Fungsi fit**

```
def fit(self, X, y):
    X = np.array(X)
    y = self.label_encoder.fit_transform(y)
    self.root = self.build_tree(X, y)
```

Fungsi fit digunakan untuk melatih model dengan data pelatihan. Fungsi ini mengubah data X menjadi array numpy dan mengencode label y menjadi numerik menggunakan LabelEncoder. Setelah itu, fungsi ini memanggil build_tree untuk membangun pohon keputusan berdasarkan data pelatihan. Akar pohon keputusan disimpan dalam atribut root.

8. **Fungsi _traverse_tree**

```
def _traverse_tree(self, x, node):
    if node.value is not None:
        return node.value
    if x[node.feature] <= node.threshold:
        return self._traverse_tree(x, node.left)
    return self._traverse_tree(x, node.right)
```

Fungsi _traverse_tree digunakan untuk melakukan traversal pohon dari akar ke daun berdasarkan fitur data input. Fungsi ini memeriksa apakah simpul saat ini adalah simpul daun. Jika ya, fungsi ini mengembalikan nilai prediksi. Jika tidak, fungsi ini memeriksa nilai fitur pada data input. Jika nilai fitur kurang dari atau sama dengan threshold, fungsi ini melanjutkan traversal ke simpul kiri. Jika tidak, fungsi ini melanjutkan traversal ke simpul kanan.

9. **Fungsi predict**

```
def predict(self, X):
    X = np.array(X)
    predictions = np.array([self._traverse_tree(x, self.root) for x in X])
    return self.label_encoder.inverse_transform(predictions)
```

Fungsi predict digunakan untuk memprediksi label dari data baru. Fungsi ini mengubah data X menjadi array numpy dan melakukan traversal pohon untuk setiap data input menggunakan fungsi _traverse_tree. Prediksi yang dihasilkan kemudian di-decode menjadi label asli menggunakan LabelEncoder.

2.4. Implementasi Tahap Cleaning

Tahap cleaning data merupakan langkah awal yang sangat penting dalam proses analisis dan pemodelan data. Proses ini memastikan bahwa data yang digunakan bebas dari masalah-masalah yang dapat mempengaruhi kualitas dan keakuratan hasil analisis atau prediksi. Berikut adalah beberapa teknik utama yang digunakan dalam tahap cleaning data:

Handle Missing Data

Fungsi *handle_missing_data* dirancang untuk menangani nilai yang hilang (missing values) dalam sebuah DataFrame. Fungsi ini memulai dengan mencetak jumlah nilai yang hilang di setiap kolom sebelum penanganan. Kemudian, DataFrame asli disalin untuk memastikan data asli tidak dimodifikasi secara langsung. Selanjutnya, kolom-kolom dipisahkan berdasarkan tipe datanya menjadi numerik (kolom bertipe *float64* atau *int64*) dan kategorikal (kolom bertipe object).

Untuk kolom numerik yang memiliki nilai hilang, metode imputasi menggunakan median digunakan. Median dipilih karena lebih tahan terhadap outlier dibandingkan rata-rata (mean). Sedangkan untuk kolom kategorikal, nilai hilang diisi menggunakan modus (nilai yang paling sering muncul dalam kolom tersebut). Setelah semua nilai hilang diatasi, fungsi mencetak

kembali jumlah nilai hilang yang tersisa untuk memverifikasi bahwa semua nilai hilang telah ditangani. Akhirnya, DataFrame yang telah bersih dikembalikan untuk digunakan lebih lanjut.

Handle Outliers

Proses **handling outliers** yang dilakukan pada data bertujuan untuk menangani nilai ekstrem di setiap fitur numerik. Pertama, hanya kolom numerik yang dipilih dari DataFrame menggunakan metode *select_dtypes*, memastikan bahwa transformasi hanya diterapkan pada fitur bertipe *float64* atau *int64*. Selanjutnya, semua kolom numerik diterapkan transformasi log menggunakan fungsi np.log1p. Transformasi ini mengambil logaritma alami dari setiap nilai yang ditambahkan 1, sehingga memastikan bahwa nilai nol tetap valid dalam transformasi.

Remove Duplicates

Proses **penghapusan duplikat** dalam dataset dilakukan menggunakan metode *drop_duplicates()* untuk memastikan bahwa setiap baris dalam dataset adalah unik. Langkah ini bertujuan untuk menghilangkan baris-baris yang memiliki nilai identik di seluruh kolom, sehingga mencegah data terduplikasi yang dapat menyebabkan bias dalam analisis atau pelatihan model. Hasilnya disimpan dalam DataFrame baru bernama *train_df_unique*, yang memungkinkan data asli tetap utuh untuk referensi jika diperlukan. Setelah penghapusan duplikat, bentuk dataset baru diverifikasi menggunakan *head()* untuk memastikan hasilnya sesuai dengan yang diharapkan. Langkah ini penting untuk menjaga kualitas dan integritas data, sekaligus menghindari redundansi yang dapat mempengaruhi hasil analisis atau performa model secara negatif.

Feature Engineering

Fungsi *enhanced_network_features* dirancang untuk melakukan rekayasa fitur (feature engineering) yang komprehensif pada dataset jaringan. Fungsi ini dimulai dengan membuat salinan dataset asli untuk menghindari modifikasi langsung. Pertama, fungsi ini menggunakan SelectKBest dengan statistik **f_classif** untuk memilih 20 fitur numerik paling penting dalam dataset berdasarkan hubungan antara fitur dan label target. Fitur numerik diekstraksi dari kolom dataset bertipe *float64* atau *int64*, dengan mengabaikan kolom *label* dari proses seleksi jika ada. Setelah seleksi, hanya fitur-fitur terpilih, bersama kolom *label* dan *attack_cat*, yang dipertahankan dalam dataset hasil. Fungsi ini mencetak jumlah fitur yang dipilih untuk memberikan gambaran ringkas mengenai proses rekayasa fitur. Akhirnya, dataset yang telah diolah (*train_df_engineered*) beserta daftar fitur terpilih dikembalikan untuk digunakan dalam analisis atau pelatihan model lebih lanjut. Pendekatan ini memastikan bahwa dataset lebih terfokus pada fitur yang paling relevan, meningkatkan efisiensi dan potensi performa model.

2.5. Implementasi Tahap Splitting Data

Proses **splitting data** ini dilakukan untuk membagi dataset menjadi dua bagian: training set (80%) dan validation set (20%). Pembagian ini dilakukan menggunakan fungsi *train_test_split* dari *sklearn*, dengan parameter *stratify* yang memastikan distribusi kategori target (*attack_cat*) tetap konsisten antara kedua subset. Dengan menjaga proporsi kategori, model yang dilatih akan lebih representatif terhadap distribusi data aslinya.

Parameter *random_state=42* digunakan untuk memastikan hasil pembagian data bersifat reproduktif, sehingga eksperimen dapat diulang dengan hasil yang sama. Setelah pembagian, bentuk masing-masing subset dicetak untuk memverifikasi ukuran data, dan distribusi kelas dalam kedua subset dihitung untuk memastikan stratifikasi telah berhasil.

Langkah terakhir adalah memisahkan kolom fitur dan target label. Kolom target (*attack_cat*) disimpan dalam variabel *y_train* dan *y_val*, sementara kolom fitur disimpan dalam *X_train* dan *X_val* dengan menghapus kolom *attack_cat* dan *label*. Proses ini mempersiapkan data untuk pelatihan model dan evaluasi secara terpisah, memastikan hasil evaluasi yang lebih akurat.

2.6. Implementasi Tahap Preprocessing

Tahap preprocessing data adalah bagian krusial dalam mempersiapkan dataset untuk analisis lebih lanjut atau pelatihan model. Pada tahap ini, data akan dipersiapkan dan disiapkan dengan berbagai teknik untuk memastikan kualitas dan efektivitas model yang akan digunakan. Berikut adalah beberapa langkah dalam implementasi tahap preprocessing:

Handle Missing Data

Kelas *HandleMissingData* digunakan untuk mengatasi data yang hilang. Kelas ini mengimputasi nilai yang hilang dengan cara yang berbeda untuk kolom numerik dan kategorikal. Kolom numerik diimputasi dengan nilai median, sedangkan kolom kategorikal diimputasi dengan nilai yang paling sering muncul (modus). Proses imputasi ini sangat penting untuk menjaga konsistensi dan menghindari kesalahan selama pelatihan model.

Categorical Encoding

EncodeCategorical digunakan untuk mengonversi kolom-kolom yang bersifat kategorikal menjadi representasi numerik. Proses ini menggunakan *OrdinalEncoder* yang mengubah kategori menjadi angka, dengan nilai yang tidak dikenal diatur ke -1. Pengkodean ini sangat berguna agar algoritma pembelajaran mesin yang berbasis numerik dapat memproses data tersebut.

Feature Scaler

Pada tahap berikutnya, semua fitur numerik diskalakan menggunakan *FeatureScaler* dengan *MinMaxScaler*. Ini mengubah setiap nilai fitur ke dalam rentang 0 hingga 1, yang membantu algoritma yang peka terhadap skala data, seperti regresi logistik atau KNN, untuk bekerja lebih efektif. Selain itu, data numerik juga bisa di-scale dalam pipeline *NetworkDimensionalityReducer* yang menggunakan *StandardScaler* untuk memastikan fitur memiliki distribusi dengan mean 0 dan deviasi standar 1 sebelum dilakukan reduksi dimensi.

Reduksi Dimensi

Untuk mengurangi kompleksitas data, *NetworkDimensionalityReducerPipeline* menggunakan PCA (Principal Component Analysis). PCA mengurangi jumlah fitur dengan mengkombinasikan fitur asli ke dalam komponen utama yang menangkap sebagian besar varians data. Ini membantu untuk mengurangi risiko overfitting dan meningkatkan kinerja model dengan mengurangi redundansi dan noise dalam data.

Pemilihan Fitur

FeatureSelector memilih fitur terbaik berdasarkan uji statistik ANOVA (*f_classif*). Di sini, fitur yang memiliki hubungan paling signifikan dengan target dipilih. Proses pemilihan fitur ini membantu mengurangi dimensi data dan fokus pada fitur yang paling relevan, yang dapat meningkatkan akurasi model dengan mengurangi overfitting.

Imbalance Class

Untuk mengatasi ketidakseimbangan kelas dalam dataset, kode ini menggunakan **SMOTE (Synthetic Minority Over-sampling Technique)**. SMOTE menggeneralisasi kelas minoritas dengan cara menghasilkan data sintetis untuk kelas tersebut, sehingga jumlah sampel antar kelas menjadi lebih seimbang. Ini penting untuk menghindari bias pada model yang lebih condong ke kelas mayoritas.

Remove Duplicates

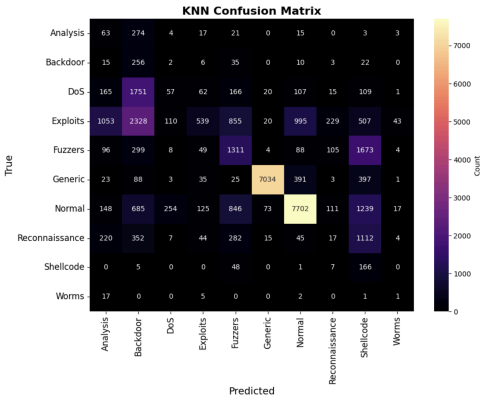
RemoveDuplicates digunakan untuk menghapus baris yang memiliki duplikasi, memastikan bahwa data yang digunakan untuk pelatihan tidak terkontaminasi oleh informasi yang sama berulang kali.

Pipeline

Semua proses preprocessing diorganisir dalam sebuah pipeline yang membuatnya lebih efisien dan mudah untuk diterapkan pada dataset baru. Pipeline ini pertama-tama menangani missing values, mengonversi data kategorikal, normalisasi fitur numerik, memilih fitur terbaik, serta melakukan oversampling pada data yang tidak seimbang. Seluruh proses ini disusun dalam urutan yang tepat untuk memastikan bahwa setiap tahapan dilakukan secara otomatis dan sesuai urutan yang benar.

BAB III
HASIL DAN ANALISIS

3.1. Hasil Prediksi dari Algoritma KNN



KNN Accuracy: 0.7398842282357638
KNN F1 Score: 0.41903406018764117

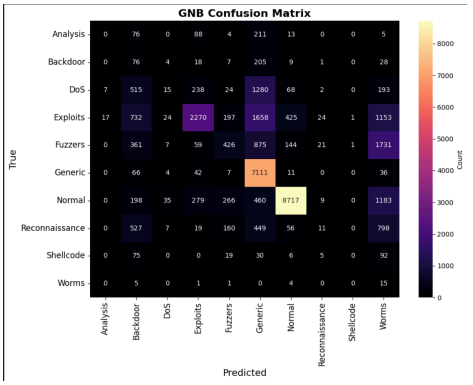
	precision	recall	f1-score	support
Analysis	0.09	0.06	0.07	400
Backdoor	0.08	0.01	0.02	349
DoS	0.31	0.31	0.31	2453
Exploits	0.58	0.69	0.63	6679
Fuzzers	0.49	0.62	0.54	3637
Generic	1.00	0.98	0.99	8000
Normal	0.91	0.85	0.88	11200
Reconnaissance	0.72	0.45	0.56	2098
Shellcode	0.37	0.14	0.20	227
Worms	0.00	0.00	0.00	26
accuracy			0.74	35069
macro avg	0.45	0.41	0.42	35069
weighted avg	0.75	0.74	0.74	35069

Hasil prediksi menggunakan algoritma **K-Nearest Neighbors (KNN)** pada dataset menunjukkan performa yang cukup beragam. Berdasarkan metrik evaluasi, model mampu memprediksi kelas **"Generic"** dan **"Normal"** dengan sangat baik. Kelas "Generic" memiliki precision sebesar 1.00 dan recall sebesar 0.98, menghasilkan F1-score yang sangat tinggi, yaitu 0.99. Demikian pula, kelas "Normal" menunjukkan performa yang solid dengan precision 0.91, recall 0.85, dan F1-score 0.88. Hal ini menunjukkan bahwa model dapat mengenali pola pada kelas mayoritas dengan baik.

Namun, performa model pada kelas-kelas lain sangat buruk. Contohnya, kelas **"Analysis," "Backdoor,"** dan **"Worms"** memiliki nilai precision, recall, dan F1-score yang sangat rendah. Kelas "Worms," misalnya, memiliki precision, recall, dan F1-score yang semuanya mendekati nol, menunjukkan bahwa model gagal total dalam mengenali pola pada kelas ini. Hal ini kemungkinan besar disebabkan oleh distribusi data yang tidak seimbang, di mana kelas-kelas minoritas memiliki jumlah data yang jauh lebih kecil dibandingkan dengan kelas mayoritas seperti "Generic" dan "Normal."

Secara keseluruhan, **akurasi model tercatat sebesar 73.98%**, sedangkan **F1-score rata-rata makro hanya mencapai 0.42**. Ini mencerminkan ketidakmampuan model untuk memberikan prediksi yang seimbang di seluruh kelas. Salah satu kelemahan utama dari algoritma KNN adalah ketergantungannya pada distribusi data dan sensitivitasnya terhadap dataset yang tidak seimbang. Selain itu, performa model KNN juga sangat bergantung pada parameter seperti jumlah tetangga (k) dan metrik jarak yang digunakan.

3.2. Hasil Prediksi dari Algoritma Naive Bayes



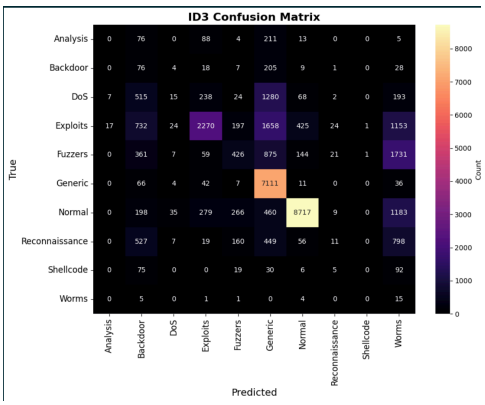

```
GNB-Scratch Accuracy: 0.5496063920747708
GNB-Scratch F1-Score: 0.2310
                precision    recall  f1-score   support

      Analysis       0.00      0.00      0.00       397
      Backdoor       0.03      0.22      0.05       348
           DoS       0.16      0.01      0.01      2342
      Exploits       0.75      0.35      0.48      6501
      Fuzzers        0.38      0.12      0.18      3625
      Generic        0.58      0.98      0.73      7277
       Normal        0.92      0.78      0.85     11147
Reconnaissance       0.15      0.01      0.01      2027
    Shellcode        0.00      0.00      0.00       227
        Worms        0.00      0.58      0.01        26

     accuracy                           0.55     33917
    macro avg        0.30      0.30      0.23     33917
 weighted avg        0.63      0.55      0.55     33917
```

Hasil prediksi menggunakan algoritma Naive Bayes pada dataset menunjukkan **performa yang cukup beragam**. Berdasarkan confusion matrix, model mampu memprediksi **kelas "Generic" dan "Normal" dengan cukup baik**. Kelas "Generic" memiliki **precision 0.58** dan **recall yang sangat tinggi, yaitu 0.98**, menghasilkan **F1-score sebesar 0.73**. Demikian pula, kelas "Normal" menunjukkan performa yang solid dengan precision 0.92, recall 0.78, dan F1-score 0.85. Hal ini menunjukkan bahwa model dapat mengenali pola pada **kelas mayoritas dengan cukup baik**.

Namun, performa model pada kelas-kelas lain sangat buruk. Contohnya, **kelas "Analysis," "Shellcode," dan "Worms"** memiliki nilai precision, recall, dan F1-score mendekati nol. Hal ini menunjukkan bahwa model gagal total untuk mengenali pola pada kelas-kelas ini. Salah satu penyebabnya adalah distribusi data yang tidak seimbang, di mana kelas-kelas minoritas memiliki jumlah data yang sangat kecil dibandingkan kelas mayoritas seperti "Normal" dan "Generic." Algoritma Naive Bayes cenderung bias terhadap kelas mayoritas dalam situasi seperti ini.

Secara keseluruhan, **akurasi model tercatat sebesar 55%**, sedangkan **F1-score rata-rata makro hanya mencapai 0.231**. Ini mencerminkan ketidakmampuan model untuk memberikan prediksi yang seimbang di seluruh kelas. Salah satu kelemahan utama dari algoritma Naive Bayes adalah asumsinya yang sederhana bahwa semua fitur saling independen, yang tidak selalu sesuai dengan data dunia nyata, terutama dalam konteks keamanan jaringan dimana hubungan antar fitur bisa sangat kompleks.

3.3. Hasil Prediksi dari Algoritma ID3




```
Predicting: 100%|██████████| 33917/33917 [00:00<00:00, 582526.48sample/s]
ID3-Scratch Accuracy: 0.6171831235073857
ID3-Scratch F1-Score: 0.3574
                precision    recall  f1-score   support

      Analysis       0.07      0.28      0.11       397
      Backdoor       0.04      0.29      0.08       348
           DoS       0.28      0.25      0.26      2342
      Exploits       0.82      0.36      0.50      6501
      Fuzzers        0.45      0.52      0.48      3625
      Generic        0.98      0.94      0.95      7277
       Normal        0.99      0.77      0.87     11147
Reconnaissance       0.29      0.18      0.23      2027
    Shellcode        0.04      0.48      0.08       227
        Worms        0.01      0.31      0.01        26

     accuracy                           0.62     33917
    macro avg        0.40      0.44      0.36     33917
 weighted avg        0.78      0.62      0.67     33917
```

Berdasarkan confusion matrix, model dapat memprediksi kelas mayoritas seperti **"Generic"** dan **"Normal"** dengan sangat baik. Kelas **"Generic"** memiliki precision 0.98, recall 0.94, dan F1-score 0.95, sedangkan kelas "Normal" mencatat precision 0.99, recall 0.77, dan F1-score 0.87. Hal ini menunjukkan bahwa model dapat mengenali pola pada kelas dengan data yang lebih melimpah secara efektif. Namun, untuk kelas minoritas seperti **"Worms"** dan **"Shellcode,"** performa model sangat buruk, dengan precision, recall, dan F1-score yang hampir nol. Hasil ini mengindikasikan bahwa model kesulitan menangkap pola pada kelas dengan jumlah data yang sangat kecil.

Secara keseluruhan, akurasi model tercatat sebesar 61.7%, yang lebih tinggi dibandingkan Naive Bayes. Namun, F1-score rata-rata makro hanya mencapai 0.3574, mencerminkan performa yang buruk pada kelas minoritas. Model cenderung bias terhadap kelas mayoritas seperti **"Generic"** dan **"Normal,"** sementara kelas minoritas seperti "Analysis," "Reconnaissance," dan terutama "Worms" tidak terdeteksi dengan baik. Penyebab utama dari hasil ini adalah ketidakseimbangan data yang signifikan, di mana kelas minoritas memiliki jumlah data yang jauh lebih kecil dibandingkan kelas mayoritas. Selain itu, overfitting menjadi masalah yang mungkin terjadi pada algoritma Decision Tree, terutama jika tidak dilakukan pruning atau tuning terhadap hyperparameter.

3.4. Perbandingan hasil prediksi dari KNN, Naive Bayes, dan ID3

Kinerja pada Kelas Mayoritas

Pada kelas mayoritas seperti **"Generic"** dan **"Normal,"** ketiga algoritma—KNN, Naive Bayes, dan ID3—menunjukkan kinerja yang solid, meskipun ada perbedaan signifikan dalam detail metrik evaluasi. KNN mencatatkan hasil yang sangat baik pada kelas "Generic," dengan precision sempurna (1.00) dan F1-score yang sangat tinggi (0.99). Untuk kelas "Normal," meskipun sedikit lebih rendah dengan precision 0.91 dan F1-score 0.88, model ini tetap efektif dalam mengenali pola pada kelas mayoritas. Naive Bayes juga menunjukkan hasil baik pada kelas mayoritas, dengan precision 0.92 untuk "Normal" dan recall yang sangat tinggi (0.98) pada kelas "Generic." Namun, Naive Bayes mengalami penurunan pada precision kelas "Generic," yang lebih rendah dibandingkan KNN, meskipun recall tetap tinggi.

ID3, di sisi lain, memberikan hasil yang sangat solid untuk kelas mayoritas, dengan precision tinggi pada "Generic" (0.98) dan "Normal" (0.99). Namun, recall untuk "Normal" sedikit lebih rendah (0.77) dibandingkan KNN dan Naive Bayes, yang menunjukkan bahwa ID3 kurang efektif dalam mengenali seluruh data pada kelas ini. Meskipun begitu, ID3 tetap memiliki keunggulan dalam hal interpretabilitas, berkat struktur pohon keputusan yang mudah dipahami.

Kelebihan dan Kekurangan Algoritma

KNN memiliki kelebihan utama dalam kesederhanaan dan kemampuannya untuk menangani data dengan baik, terutama untuk kelas mayoritas. Algoritma ini juga cukup fleksibel dalam penyesuaian parameter seperti jumlah tetangga (k) dan metrik jarak. Namun, KNN sangat sensitif terhadap data yang tidak seimbang, yang dapat menyebabkan kinerja buruk pada kelas minoritas. Selain itu, **KNN** membutuhkan banyak waktu dan memori saat bekerja dengan dataset besar, karena harus menghitung jarak untuk setiap prediksi.

Naive Bayes sangat cepat dalam hal pelatihan dan prediksi, serta menunjukkan performa baik pada kelas mayoritas, terutama dengan data yang terdistribusi dengan baik. Namun, model ini mengandalkan asumsi independensi fitur yang tidak selalu akurat pada data dunia nyata, yang dapat memengaruhi akurasi dalam kasus di mana fitur-fitur tersebut saling terkait. Selain itu, Naive Bayes sering kali bias terhadap kelas mayoritas, menghasilkan kinerja buruk pada kelas minoritas.

ID3 memiliki kelebihan dalam hal interpretabilitas, karena menghasilkan pohon keputusan yang jelas dan mudah dipahami. Algoritma ini efektif pada data terstruktur dan bisa menangani fitur kategorikal dengan baik. Namun, **ID3** cenderung bias terhadap kelas mayoritas dan kesulitan dalam mengenali pola pada kelas minoritas, terutama jika data sangat tidak seimbang. Selain itu, **ID3** rentan terhadap masalah overfitting jika tidak diterapkan pruning atau tuning hyperparameter dengan baik.

BAB IV
KESIMPULAN DAN SARAN

4.1. Kesimpulan

Berdasarkan analisis yang telah dilakukan, beberapa hal penting terkait dengan kinerja model dapat disimpulkan. Pertama, model **KNN (k-Nearest Neighbors)** menunjukkan performa yang lebih baik dibandingkan dengan model lainnya seperti **Gaussian Naive Bayes** dan **ID3**. KNN mengungguli model-model lainnya karena sifatnya yang berbasis jarak, di mana ia efektif dalam menangani data dengan cluster yang jelas terpisah. Selain itu, KNN juga dapat memanfaatkan skala fitur dengan lebih baik melalui proses *feature scaling*, yang mencegah dominasi fitur dengan skala yang lebih besar. Sebaliknya, model seperti Naive Bayes, meskipun cepat dan sederhana, cenderung gagal dalam menangani hubungan yang kompleks jika asumsi tentang independensi fitur atau distribusi Gaussian tidak terpenuhi. Begitu juga dengan **ID3**, yang meskipun unggul dalam menangani hubungan non-linear, bisa rentan terhadap overfitting jika tidak dipangkas dengan baik.

Selanjutnya, hasil dari analisis menggunakan **confusion matrix** menunjukkan bahwa adanya **ketidakseimbangan kelas** dapat mempengaruhi kinerja model. Ketidakseimbangan ini menyebabkan model lebih cenderung memprediksi kelas mayoritas, yang mengarah pada **false negatives** yang lebih tinggi untuk kelas minoritas. Oleh karena itu, teknik seperti **oversampling** dan **undersampling** sangat membantu dalam meningkatkan akurasi untuk kelas yang kurang terwakili. Penanganan terhadap data yang hilang (missing data) juga memainkan peran penting. Menggunakan teknik imputasi untuk mengisi nilai yang hilang terbukti lebih baik daripada menghapus data tersebut, karena imputasi mempertahankan ukuran dataset dan mencegah hilangnya informasi penting.

Proses *feature scaling* terbukti memberikan dampak positif terhadap kinerja model, terutama pada algoritma berbasis jarak seperti KNN. Dengan menskalakan fitur-fitur, model dapat menghindari dominasi fitur dengan rentang nilai yang lebih besar, sehingga setiap fitur memiliki kontribusi yang seimbang dalam proses pembelajaran. Selain itu, penggunaan teknik reduksi dimensi seperti **PCA (Principal Component Analysis)** juga terbukti efektif, karena dapat mengurangi dimensi data dengan menghilangkan fitur yang redundan, yang pada gilirannya mempercepat pelatihan model dan meningkatkan akurasi prediksi.

4.2. Saran

Model KNN dalam situasi di mana data memiliki struktur klaster yang jelas dan dapat dipengaruhi oleh jarak antar titik, terutama setelah melakukan **feature scaling** untuk memastikan kontribusi fitur yang lebih seimbang. Untuk mengatasi ketidakseimbangan kelas, sebaiknya menggunakan teknik seperti **oversampling** atau **undersampling** agar model dapat lebih memperhatikan kelas minoritas dan mengurangi false negatives. Selain itu, untuk menangani **data yang hilang**, **teknik imputasi** lebih disarankan dibandingkan penghapusan data, karena imputasi membantu mempertahankan ukuran dataset dan informasi penting. Penggunaan **reduksi dimensi** melalui **PCA** juga dapat meningkatkan efisiensi dan akurasi model, terutama jika data mengandung banyak fitur redundan. Dengan pendekatan-pendekatan ini, diharapkan kinerja model dapat lebih optimal dan mampu menangani kompleksitas serta ketidakseimbangan data dengan lebih baik.

BAB V
Referensi

[1] Dokumen Tucil2_13522124_13522135
[2] Dokumen 13522124_13522135_K3_Deadline2.ipynb
[3] Tim Pengajar IF3170. 2024/2025. Slide perkuliahan Introduction to DS.
[4] Tim Pengajar IF3170. 2024/2025. Slide perkuliahan Data Understanding.
[5] Tim Pengajar IF3170. 2024/2025. Slide perkuliahan Data Preparation.

Lampiran

Pembagian Kerja Kelompok

NIM	Nama	Pekerjaan
13522124	Aland Mulia Pratama	Implementasi Algoritma KNN dan Naive Bayes. Modifikasi Preprocessing dan Fungsi Confusion Matrix. Modifikasi Algoritma KNN (performa waktu). Mengerjakan Error Analysis. Pengerjaan laporan BAB III dan BAB IV.
13522126	Rizqika Mulia Pratama	Implementasi Algoritma KNN dan Gaussian Naive Bayes, Data Preprocessing (Handling Outliers, Data Normalization, Feature Encoding, Feature Scaling, Feature Selection), Pipelining, Modeling, Validation, Model Save, Model Load.
13522135	Christian Justin Hendrawan	Mengerjakan bagian cleaning data : Handle Missing Value dan Feature Engineering, mengimplementasikan Network Dimensionality Reducer, dan mengimplementasikan ID3. Pengerjaan Laporan BAB I dan BAB II.
13522148	Auralea Alvinia Syaikha	Mengerjakan bagian cleaning data (handle missing values, outlier, handling duplicates), bagian preprocessing (feature encoding, feature scaling, handle imbalance dataset), serta memodif pipeline. Mengerjakan laporan bagian analisis prediksi KNN.