IF3170 INTELIGENSI ARTIFISIAL

Implementasi Algoritma Pembelajaran Mesin

Tugas Besar 2

Disusun untuk memenuhi tugas mata kuliah Inteligensi Artifisial
pada Semester 1 (satu) Tahun Akademik 2024/2025
Tugas Besar IF3170 Inteligensi Artifisial

Oleh

Shabrina Maharani	**13522134**
Muhammad Davis Adhipramana	**13522157**
Pradipta Rafa Mahesa	**13522162**
Valentino Chryslie Triadi	**13522164**

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2024

DAFTAR ISI

BAB 1 DESKRIPSI PERSOALAN

Pada Tugas Besar 2 IF3170 Inteligensi Buatan, penulis ditugaskan untuk mengimplementasikan tiga algoritma pembelajaran mesin—KNN, Gaussian Naive-Bayes, dan ID3—secara manual (*from scratch*) terhadap dataset UNSW-NB15, yang mencakup data lalu lintas jaringan dengan aktivitas normal dan serangan siber. Implementasi harus mencakup berbagai parameter dan metrik, seperti jumlah tetangga dan pilihan jarak pada KNN, serta pemrosesan data numerik

BAB 2 PENJELASAN SINGKAT IMPLEMENTASI KNN

k-Nearest Neighbor (KNN) adalah algoritma yang digunakan untuk mengklasifikasikan data berdasarkan data terdekat yang sudah diketahui sebelumnya. Ketika ada data baru yang belum diketahui kategorinya, KNN akan mencari beberapa data terdekat (disebut tetangga) dari data baru tersebut menggunakan perhitungan jarak, seperti jarak Euclidean untuk data numerik. Kategori data baru ditentukan berdasarkan mayoritas kategori dari tetangga-tetangga ini. Misalnya, jika tetangga terdekat dari data baru sebagian besar berasal dari kelas "A", maka data baru akan diklasifikasikan ke kelas "A". Namun, KNN memiliki kelemahan, seperti sensitif terhadap data outlier dan kurang efektif jika data tidak seimbang. Selain itu, KNN menjadi lambat jika data terlalu besar karena harus menghitung jarak ke semua data lainnya. Berikut adalah implementasi dari algoritma k-Nearest Neighbor (KNN).

```python
class KNN:
    '''
    methods: euclidean, minkowski, manhattan
    '''
    def __init__(self, n_neighbors=3, methods="euclidean"):
        self.n_neighbors = n_neighbors
        self.method = methods

    def fit(self, X, y):
        self.X_train = np.array(X)
        print(self.X_train.shape)
        self.y_train = np.array(y)
        print(self.y_train.shape)
        self.classes_ = np.unique(y)

    def predict(self, X):
        with ThreadPoolExecutor() as executor:
            predictions = list(executor.map(self._predict_single, X))
        return np.array(predictions)

    def predict_proba(self, X):
        X = np.array(X)
        probabilities = []
        with ThreadPoolExecutor() as executor:
            # Parallelize probability calculation for each sample
            probabilities = list(executor.map(self._predict_proba_single, X))
        return np.array(probabilities)

    def _predict_single(self, x):
        """Calculate the class label for a single test sample `x`."""
        distances = self._compute_distances(x)
        k_indices = np.argsort(distances)[:self.n_neighbors]
        k_labels = self.y_train[k_indices]
        most_common = Counter(k_labels).most_common(1)[0][0]
        return most_common

    def _predict_proba_single(self, x):
        """Calculate the class probabilities for a single test sample `x`."""
        distances = self._compute_distances(x)
        k_indices = np.argsort(distances)[:self.n_neighbors]
        k_labels = self.y_train[k_indices]
        counts = Counter(k_labels)
        total = sum(counts.values())
        prob = {label: counts.get(label, 0) / total for label in self.classes_}
        return [prob[label] for label in self.classes_]

    def _compute_distances(self, x):
        if self.method in ['euclidean', 'minkowski']:
            return cdist([x], self.X_train, metric=self.method).flatten()
        elif self.method == 'manhattan':
            return cdist([x], self.X_train, metric='cityblock').flatten()
        else:
            raise ValueError(f"Unknown distance metric '{self.method}'")
```

Algoritma ini bekerja dengan cara membandingkan setiap data uji dengan data latih berdasarkan jarak tertentu, seperti Euclidean, Manhattan, atau Minkowski, yang ditentukan melalui parameter methods. Saat memprediksi, algoritma menghitung jarak antara data uji dengan seluruh data latih, lalu memilih sejumlah tetangga terdekat berdasarkan parameter n_neighbors. Kategori data uji ditentukan dari kategori mayoritas tetangga tersebut. Selain prediksi kelas, metode ini juga menyediakan kemampuan untuk menghitung probabilitas setiap kelas melalui metode predict_proba, yang menentukan peluang data uji termasuk ke dalam masing-masing kategori. Untuk mempercepat proses prediksi, fungsi ini menggunakan pemrosesan paralel dengan ThreadPoolExecutor, sehingga setiap data uji dapat diproses secara bersamaan. Struktur kodenya dirancang modular, sehingga komponen seperti perhitungan jarak dan prediksi kelas dapat dipisah dengan rapi.

Algoritma untuk membandingkan data uji dapat dilihat sebagai berikut untuk masing-masing algoritma:

- Algoritma Euclidean

$$\text{distance}(x, X_i) = \sqrt{\sum_{j=1}^{d}(x_j - X_{i_j})^2]}$$

- Algoritma Manhattan

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

- Algoritma Minkowski

$$d(x, y) = \left(\sum_{i=1}^{n} (x_i - y_i)^p\right)^{\frac{1}{p}}$$

BAB 3 PENJELASAN SINGKAT IMPLEMENTASI GAUSSIAN NAIVE-BAYES

Algoritma Gaussian Naive Bayes merupakan salah satu dari berbagai teknik klasifikasi dalam machine learning. Algoritma ini berbasis pendekatan probabilitas dan menggunakan distribusi Gaussian, atau yang sering disebut distribusi normal. Dalam metode ini, setiap fitur diasumsikan tidak saling mempengaruhi satu sama lain dalam memprediksi kelas atau label output. Algoritma ini menghitung probabilitas suatu data termasuk ke dalam setiap kelas atau label. Kombinasi dari probabilitas yang dihitung untuk semua fitur menghasilkan prediksi akhir, yang menentukan kemungkinan data tersebut berada pada setiap kelas. Hasil klasifikasi akhir akan menetapkan data-data yang ada ke kelas dengan probabilitas tertinggi.

Distribusi normal adalah model statistik yang menggambarkan distribusi variabel acak kontinu dalam kehidupan nyata dan memiliki bentuk kurva seperti lonceng (bell-shaped curve). Dua karakteristik utama dari distribusi normal adalah rata-rata (μ) dan simpangan baku (σ).

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

Sebuah variabel (X) yang mengikuti distribusi normal memiliki nilai yang tersebar secara kontinu (variabel kontinu) dari $-\infty < X < +\infty$, dengan total luas di bawah kurva model sama dengan 1. Parameter μ dan σ pada distribusi ini diestimasi menggunakan metode *maximum likelihood*. Berikut adalah implementasi dari algoritma Gaussian Naive-Bayes.

```python
class GaussianNaiveBayes:
    def __init__(self):
        self.classes_ = None
        self.mean = None
        self.variance = None
        self.prior = None

    def fit(self, X, y):
        # Calculate mean, variance, and prior probabilities for each class
        print("THIS IS NAIVE BAYES")
        self.classes_ = np.unique(y)

        if(type(X) == tuple):
            print("X: ")
            print(X)
            print("y: ")
            print(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])

    def _gaussian_density(self, class_idx, x):
        mean = self.mean[class_idx]
        var = self.variance[class_idx] + 1e-9  # Avoid division by zero

        # Standard Gaussian PDF formula
        coeff = 1 / np.sqrt(2 * np.pi * var)
        exponent = np.exp(-((x - mean) ** 2) / (2 * var))
        return coeff * exponent

    def _log_gaussian_density(self, class_idx, x):
        mean = self.mean[class_idx]
        var = self.variance[class_idx] + 1e-9  # Avoid log(0)

        # Log Gaussian PDF formula
        log_coeff = -0.5 * np.log(2 * np.pi * var)
        log_exponent = -((x - mean) ** 2) / (2 * var)
        return log_coeff + log_exponent

    def _joint_log_likelihood(self, x):
        log_likelihood = []
        for idx, c in enumerate(self.classes_):
            log_prior = np.log(self.prior[idx])
            log_conditional = np.sum(self._log_gaussian_density(idx, x))
            log_likelihood.append(log_prior + log_conditional)
        return np.array(log_likelihood)

    def predict(self, X):
        log_likelihoods = np.array([self._joint_log_likelihood(x) for x in X])
        return self.classes_[np.argmax(log_likelihoods, axis=1)]

    def predict_proba(self, X):
        probabilities = []
        for x in X:
            log_likelihood = self._joint_log_likelihood(x)
            max_log = np.max(log_likelihood)
            probs = np.exp(log_likelihood - max_log)  # Stabilize exp
            normalized_probs = probs / np.sum(probs)  # Normalize to sum to 1
            probabilities.append(normalized_probs)
        return np.array(probabilities)
```

Implementasi Gaussian Naive Bayes di atas dimulai dengan model dilatih menggunakan fungsi fit, di mana pada fungsi ini dihitung rata-rata (mean), varians, dan probabilitas awal (prior probability) untuk setiap kelas berdasarkan data latih. Rata-rata dan varians digunakan untuk menggambarkan bagaimana fitur-fitur data didistribusikan dalam setiap kelas, sementara probabilitas awal menunjukkan seberapa sering setiap kelas muncul dalam data.

Untuk menghitung kemungkinan data termasuk ke dalam sebuah kelas, digunakan fungsi _gaussian_density dan _log_gaussian_density, yang menghitung *probability density (*ukuran yang menunjukkan seberapa besar kemungkinan suatu nilai tertentu muncul dalam distribusi probabilitas kontinu, seperti distribusi normal (Gaussian)) menggunakan distribusi Gaussian. Ketika Gaussian Naive Bayes menghitung probabilitas gabungan (joint probability) untuk setiap kelas, algoritma ini akan mengalikan probabilitas dari setiap fitur. Jika terdapat banyak fitur, hasil perkalian tersebut dapat menjadi sangat kecil hingga mendekati nol, yang menyebabkan masalah underflow pada komputer (angka terlalu kecil untuk direpresentasikan). Untuk mengatasi masalah ini, algoritma tidak langsung mengalikan probabilitas, tetapi mengubah terlebih dahulu menjadi logaritma. Fungsi _joint_log_likelihood kemudian menggabungkan peluang tersebut dengan prior probability, menghasilkan log likelihood gabungan untuk masing-masing kelas.

Ketika model digunakan untuk memprediksi, fungsi predict akan memilih kelas dengan log likelihood tertinggi untuk setiap data yang diuji. Sementara itu, fungsi predict_proba akan menghasilkan probabilitas untuk semua kelas, dengan menjumlahkan peluang hingga totalnya 100%. Secara keseluruhan, algoritma ini bekerja dengan menghitung parameter distribusi Gaussian pada data latih, lalu menggunakan informasi tersebut untuk membuat keputusan klasifikasi pada data baru.

BAB 3 PENJELASAN SINGKAT IMPLEMENTASI ID3

Algoritma ID3 (Iterative Dichotomiser 3) adalah metode yang digunakan untuk membangun pohon keputusan (decision tree) yang dikembangkan oleh J. Ross Quinlan pada tahun 1986. Algoritma ini bekerja secara top-down, dimulai dari node akar (root node) dan membagi data secara berulang ke dalam node cabang (internal nodes) hingga mencapai node daun (leaf nodes). Setiap pembagian data dilakukan dengan memilih atribut yang memberikan informasi terbaik untuk memisahkan kelas-kelas dalam dataset, sehingga menghasilkan struktur pohon yang sederhana dan efisien. ID3 menggunakan konsep entropy untuk mengukur tingkat ketidakteraturan atau ketidakpastian dalam data, serta information gain untuk menentukan seberapa besar pengurangan ketidakpastian ketika data dibagi berdasarkan atribut tertentu.

Proses pembentukan pohon keputusan diawali dengan menghitung entropy pada dataset untuk mengetahui seberapa teracak data tersebut. Kemudian, algoritma menghitung information gain untuk setiap atribut, memilih atribut dengan nilai information gain tertinggi sebagai pembagi pada node tersebut. Proses ini terus diulang pada setiap cabang hingga semua data terpisah sepenuhnya ke dalam kelas-kelas tertentu, atau tidak ada lagi atribut yang dapat digunakan untuk pemisahan. ID3 dikenal karena kesederhanaannya, tetapi dapat rentan terhadap overfitting, terutama jika data memiliki banyak atribut atau noise. Berikut adalah implementasi dari algoritma ID3.

```python

class ID3DecisionTree:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self.tree = None

    def fit(self, X, y):
        # Convert to numpy array for consistency
        X = np.array(X)
        y = np.array(y)

        feature_names = list(range(X.shape[1]))  # Use numeric indices for features if unnamed
        self.tree = self._build_tree(X, y, feature_names, depth=0)

    def predict(self, X):
        X = np.array(X)  # Ensure input is numpy array
        return np.array([self._predict_single(self.tree, row) for row in X])

    def _entropy(self, y):
        """Calculate entropy for a label distribution."""
        values, counts = np.unique(y, return_counts=True)
        probabilities = counts / len(y)
        return -np.sum(probabilities * np.log2(probabilities))

    def _information_gain(self, X, y, feature_index):
        """Calculate information gain for a feature."""
        parent_entropy = self._entropy(y)
        values, counts = np.unique(X[:, feature_index], return_counts=True)

        # Vectorized computation of weighted entropy
        weighted_entropy = np.sum(
            (counts / len(y)) * np.array([self._entropy(y[X[:, feature_index] == value]) for value in values])
        )
        return parent_entropy - weighted_entropy

    def _best_feature(self, X, y):
        """Determine the best feature to split on based on information gain."""
        with ThreadPoolExecutor() as executor:
            gains = list(executor.map(lambda i: self._information_gain(X, y, i), range(X.shape[1])))
        return np.argmax(gains), np.max(gains)

    def _build_tree(self, X, y, feature_names, depth):
        if len(np.unique(y)) == 1:
            return np.unique(y)[0]

        if len(feature_names) == 0 or (self.max_depth is not None and depth >= self.max_depth):
            return {"majority": np.bincount(y).argmax()}

        best_feature_index, best_gain = self._best_feature(X, y)
        if best_gain == 0:
            return {"majority": np.bincount(y).argmax()}

        tree = {
            "feature": feature_names[best_feature_index],
            "majority": np.bincount(y).argmax(),
            "subtrees": {}
        }

        values = np.unique(X[:, best_feature_index])
        for value in values:
            mask = X[:, best_feature_index] == value
            subtree = self._build_tree(
                X[mask],
                y[mask],
                feature_names,
                depth + 1
            )
            tree["subtrees"][value] = subtree

        return tree


    def _predict_single(self, tree, row):
        while isinstance(tree, dict):
            feature_name = tree["feature"]
            feature_value = row[feature_name]

            if feature_value in tree["subtrees"]:
                tree = tree["subtrees"][feature_value]
            else:
                return tree["majority"]

        return tree


    def print_tree(self, tree=None, indent=""):
        """Pretty-print the decision tree."""
        if tree is None:
            tree = self.tree

        if not isinstance(tree, dict):
            print(indent + f"Class: {tree}")
            return

        for feature, subtree in tree.items():
            print(indent + f"Feature: {feature}")
            for value, branch in subtree.items():
                print(indent + f"  Value: {value}")
                self.print_tree(branch, indent + "    ")

```

Implementasi ID3 di atas dimulai dengan model dilatih menggunakan fungsi fit, di mana disini kita langsung menjalankan _build_tree untuk membangun tree paling awal. Dalam tree tersebut akan tercipta subtree lainnya, pada tiap treenya akan dicari terlebih dahulu feature yang memiliki gain paling besar. Untuk fitur dengan nilai gain paling besar akan dipilih sebagai subtrees dari tree tersebut. Untuk perhitungan nilai gainnya, disini untuk mempercepat kalkulasi kita menggunakan concurrency untuk menghitung nilai gain dari seluruh fitur lalu mengambil nilai tertinggi saja. Ketika model digunakan untuk memprediksi, fungsi predict akan menghasilkan suatu array yang berisikan seluruh subtree dari tree yang sudah didapatkan pada saat fit dijalankan.

BAB 4 CLEANING DAN PREPROCESSING

Data Cleaning dan Preprocessing yang kami lakukan terbagi menjadi beberapa bagian, yakni:

1. **Handling Missing Data**

   Untuk penanganan data hilang pada tipe data numerik, kami menggunakan metode imputasi dengan strategi ‘median’. Untuk penanganan data hilang pada tipe data kategorikal, kami menggunakan metode imputasi dengan strategi ‘modus’. Alasan pemilihan metode tersebut adalah karena nilai missing values dalam data kurang dari 5% dari total data, distribusinya cukup tersebar, dan tidak terdapat korelasi signifikan antara data yang hilang, maka data imputation dipilih untuk mengatasi masalah ini. Selain itu, sebagian besar distribusi data bersifat skew (baik ke kiri maupun ke kanan), hanya ID yang memiliki distribusi normal. Oleh karena itu, metode imputasi yang digunakan adalah median imputation dengan menggunakan library Scikit-learn, karena median lebih cocok untuk data dengan distribusi yang skew. Selain itu, metode imputasi modus juga digunakan untuk menangani missing value pada data bertipe kategorikal

2. **Dealing with Outliers**

   Penanganan outlier dilakukan dengan berbagai metode sesuai dengan pola dan intensitas outlier di setiap kolom data. Untuk kolom dengan outlier yang tidak terlalu ekstrem, digunakan Winsorizing, yaitu metode clipping data ekstrem berdasarkan persentil tertentu. Jika data sangat skewed, transformasi seperti square root, logarithmic (log1p), atau power transformation digunakan untuk menormalisasi distribusi dan meredam pengaruh outliers. Beberapa kolom ada yang menggunakan kombinasi, seperti Winsorizing diikuti dengan log transform atau power transform, untuk mengelola distribusi yang kompleks. Kolom tanpa outlier tetap dipertahankan tanpa perubahan

3. **Removing Duplicates**

   Pada model kami, kami tidak melakukan penanganan data duplikat karena tidak ditemukan data yang duplikat

4. **Feature Engineering**

   Pada model kami, kami menambahkan fitur baru yakni ‘total_bytes’ yang didapat dari penjumlahan ‘sbytes’ dan ‘dbytes’. Kemudian, kami juga melakukan discretization atau mengelompokan data numerik pada fitur ‘dur’. Kami mengelompokan data pada fitur ‘dur’ menjadi beberapa kelompok, seperti "very_short" (0-1), "short" (1-5), "medium" (5-10), "long" (10-100), dan "very_long" (100-∞).

5. **Feature Scaling**

   Feature scaling adalah proses yang digunakan untuk memastikan semua fitur dalam dataset berada dalam skala yang sama, sehingga algoritma dapat bekerja lebih optimal. Dalam kelas FeatureScaler, proses scaling disesuaikan dengan distribusi data. Jika distribusi data tidak normal atau memiliki skewness yang signifikan, maka data akan dinormalisasi menggunakan MinMax scaling untuk mengubah nilainya ke skala tertentu, biasanya antara 0 dan 1. Pendekatan ini membantu menjaga proporsi nilai asli tanpa membuat asumsi distribusi data. Namun, jika data sudah berdistribusi normal atau

mendekati normal, maka proses standardization digunakan untuk mengubah data agar memiliki rata-rata nol dan standar deviasi satu.

6. **Feature Encoding**

   Feature encoding adalah proses konversi data-data yang bertipe kategorikal menjadi format numerik agar dapat diproses. Dalam kelas FeatureEncoder, terdapat metode seperti one-hot encoding dan ordinal encoding. One-hot encoding menghasilkan representasi biner untuk setiap kategori, memastikan bahwa tidak ada hierarki yang diimplikasikan antar kategori. Sebaliknya, ordinal encoding memberikan nilai numerik berdasarkan urutan kategori, cocok untuk data dengan hubungan hierarkis. Encoder ini juga dapat mendukung target encoding, di mana nilai kategori diubah berdasarkan hubungannya dengan variabel target, berguna untuk model berbasis pohon keputusan.

7. **Handling Imbalanced Dataset**

   Pada model kami, kami mengimplementasikan kelas untuk menangani data yang tidak seimbang. Namun, kami tidak menggunakannya dalam pipeline. Implementasi dari kelas ini adalah dengan melakukan resample terhadap data train. Kami membuat beberapa metode untuk melakukan resample, yakni dengan metode oversample (menambahkan data dengan men generate data baru), metode undersample (mengurangi data pada kelas mayoritas)

8. **Data Normalization**

   Pada model kami, kami mengimplementasikan kelas DataNormalizer dengan beberapa metode, seperti ‘log’, ‘sqrt’, ‘boxcox’, ‘minmax’, dan ‘yeojohnson’. DataNormalizer ini digunakan untuk mengecilkan range nilai dari sebuah fitur dan juga membuat data lebih stabil dan seimbang.

9. **Dimensionality Reduction**

   Pada model kami, kami mengimplementasikan kelas PCA sebagai kelas untuk mereduksi jumlah fitur. Reduksi jumlah fitur dapat berguna untuk mengurangi beban komputasi dan juga mengurangi overfitting. PCA bekerja dengan mencari kombinasi linear dari fitur asli (komponen utama) yang menjelaskan sebagian besar variabilitas dalam data.

BAB 5 PERBANDINGAN HASIL PREDIKSI

Berikut adalah analisis perbandingan hasil evaluasi menggunakan metrik evaluasi berupa f1-score weighted, f1-score macro, dan accuracy antara algoritma yang diimplementasikan secara manual (*from scratch*) dengan algoritma yang telah tersedia di *library* scikit-learn.

1. **KNN (k-Nearest Neighbor)**

   Berikut adalah metrik evaluasi hasil dari implementasi from scratch beserta akurasinya

	precision	recall	f1-score	support
Analysis	0.00	0.00	0.00	384
Backdoor	0.00	0.00	0.00	367
DoS	0.03	0.03	0.03	2459
Exploits	0.29	0.00	0.00	6636
Fuzzers	0.50	0.00	0.00	3637
Generic	0.52	0.98	0.68	7975
Normal	0.55	0.83	0.66	11254
Reconnaissance	0.00	0.00	0.00	2070
Shellcode	0.00	0.00	0.00	259
Worms	0.00	0.00	0.00	28
accuracy			0.49	35069
macro avg	0.19	0.18	0.14	35069
weighted avg	0.40	0.49	0.37	35069

```
Validation Set Accuracy: 0.4916
```

   Sedangkan, ini merupakan hasil dari library scikit-learn dengan proses cleaning dan preprocessor yang sama dengan implementasi manual.

	precision	recall	f1-score	support
Analysis	0.00	0.00	0.00	384
Backdoor	0.02	0.01	0.02	367
DoS	0.04	0.08	0.05	2459
Exploits	0.28	0.05	0.08	6636
Fuzzers	0.44	0.00	0.00	3637
Generic	0.52	0.98	0.68	7975
Normal	0.62	0.75	0.68	11254
Reconnaissance	0.00	0.00	0.00	2070
Shellcode	0.00	0.00	0.00	259
Worms	0.00	0.00	0.00	28
accuracy			0.48	35069
macro avg	0.19	0.19	0.15	35069
weighted avg	0.42	0.48	0.39	35069

```
Validation Set Accuracy: 0.4782
```

   Dapat dilihat bahwa akurasi dari kedua metrik evaluasi diatas berbeda. Hal tersebut dapat terjadi karena adanya perbedaan pendekatan dan metode dalam menghitung jarak antar data. Selain itu, perbedaan metode pengambilan neighbor dengan jarak yang sama juga mempengaruhi hasil akhir dari masing-masing prediksi.

2. **Gaussian Naive-Bayes**

   Berikut adalah matrik evaluasi hasil dari implementasi *from scratch.*

	precision	recall	f1-score	support
Analysis	0.00	0.00	0.00	384
Backdoor	0.00	0.00	0.00	367
DoS	0.34	0.33	0.34	2459
Exploits	0.05	0.05	0.05	6636
Fuzzers	0.15	0.00	0.00	3637
Generic	1.00	0.17	0.29	7975
Normal	0.57	0.91	0.70	11254
Reconnaissance	0.14	0.46	0.22	2070
Shellcode	0.00	0.00	0.00	259
Worms	0.00	0.00	0.00	28
accuracy			0.39	35069
macro avg	0.23	0.19	0.16	35069
weighted avg	0.47	0.39	0.34	35069

Sedangkan, ini merupakan hasil dari yang library scikit-learn dengan proses cleaning dan preprocessor yang sama dengan implementasi manual.

	precision	recall	f1-score	support
Analysis	0.00	0.00	0.00	384
Backdoor	0.00	0.00	0.00	367
DoS	0.34	0.33	0.34	2459
Exploits	0.05	0.05	0.05	6636
Fuzzers	0.15	0.00	0.00	3637
Generic	1.00	0.17	0.29	7975
Normal	0.57	0.91	0.70	11254
Reconnaissance	0.14	0.46	0.22	2070
Shellcode	0.00	0.00	0.00	259
Worms	0.00	0.00	0.00	28
accuracy			0.39	35069
macro avg	0.23	0.19	0.16	35069
weighted avg	0.47	0.39	0.34	35069

Validation Set Accuracy: 0.3929
Validation Set F1-Score (macro): 0.1609

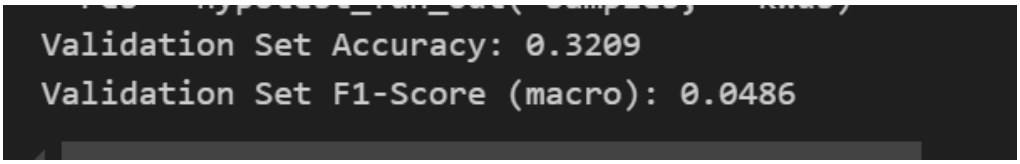
Perbandingan hasil antara algoritma Gaussian Naive Bayes yang diimplementasikan secara manual (GaussianNaiveBayes) dengan yang ada di *library* seperti Scikit-learn (GaussianNB) bisa dilihat melalui matrik evaluasi seperti precision, recall, F1-score, dan accuracy. Dari hasil yang diperoleh, terlihat tidak ada perbedaan antara metrik evaluasi antara hasil implementasi dengan library. Namun, terdapat beberapa perbedaan dalam proses menjalankannya? (berdasarkan dokumentasi dari scikit learn) yang mungkin disebabkan oleh beberapa hal, seperti stabilitas perhitungan, teknik smoothing, atau optimasi komputasi. Misalnya, Scikit-learn menggunakan metode smoothing dan cara yang lebih baik dalam menangani varians kecil, sehingga lebih stabil ketika menghadapi data dengan varians yang sangat rendah atau probabilitas yang kecil.

Perbedaan ini juga bisa terjadi karena pustaka Scikit-learn mengoptimalkan proses perhitungan dengan vektorisasi dan algoritma yang lebih efisien. Metrik seperti macro average menunjukkan rata-rata performa di semua kelas dengan cara yang sama, sementara weighted average lebih memperhitungkan jumlah data di setiap kelas,

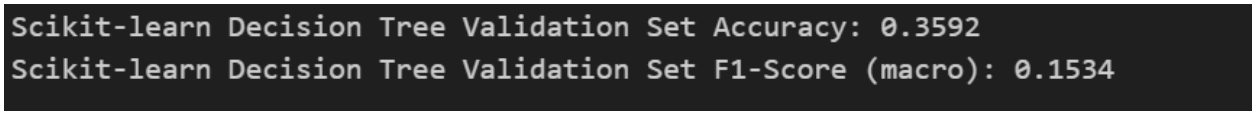
sehingga jika ada perbedaan besar antara kedua metrik ini, itu bisa menandakan bahwa model mempunyai performa yang lebih baik di kelas tertentu.

3. ID3

Berikut merupakan matrik evaluasi hasil implementasi from scratch



Sedangkan, berikut merupakan matrik evaluasi hasil implementasi dengan menggunakan library scikit



Kita menggunakan accuracy dan F1 score untuk perbandingannya. Kami mendapati bahwa hasil yang dimiliki untuk accuracy nya memiliki perbedaan 0.03 namun untuk test F1 scorenya, memiliki perbedaan nilai yang cukup significant. Perbedaan nilai ini mungkin terjadi di dalam pengimplementasian tree kami. Dalam tree yang kami buat parameter yang kami sediakan hanyalah max-depth saja sedangkan pada library scikit terdapat banyak parameter yang disediakan seperti splitter, min_samples_split, random_state dan sebagainya.

Perbedaan lainnya bisa terjadi karena Scikit-Tree juga sudah melakukan optimisasi lebih lanjut terhadap tree yang dibuat seperti splitting ataupun hal lainnya yang mampu mengoptimalkan tree yang ada.

Pembagian Tugas

Tugas	Anggota
Data Understanding	13522157, 13522164, 13522134
Data Insight	13522162
Data Cleaning	13522134
Data Preprocessing	13522157, 13522164, 13522134, 13522162
KNN	13522164
Gaussian Naive Bayes	13522134, 13522162
ID3	13522164, 13522157
Laporan	13522157, 13522164, 13522134, 13522162

Referensi

Russell, S., & Norvig, P. (2021). *Artificial intelligence: A modern approach (Global ed.)*. Pearson Higher Ed.

PPT Pembelajaran Mata Kuliah Inteligensi Artifisial Tahun 2024

Ha, Michelle. (2023). Gaussian Naive Bayes. Link : https://medium.com/@chellehdwjy/gaussian-naive-bayes-f05ec0b61d91

Géron, A. (2019). Hands-on machine learning with Scikit-Learn, Keras, and. TensorFlow: Concepts, tools, and techniques to build intelligent systems.