

LAPORAN TUGAS BESAR

IF3170 INTELIGENSI ARTIFISIAL

Implementasi Algoritma Pembelajaran Mesin



Disusun oleh:
Kelompok 8

Ibrahim Ihsan Rasyid	13522018
Erdianti Wiga Putri Andini	13522053
Elijah Darrellshane Suryanegara	13522097
Muhammad Dava Fathurrahman	13522114

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

2024

DAFTAR ISI

DAFTAR ISI.....	2
BAB I	
DESKRIPSI PERSOALAN.....	3
BAB II	
PEMBAHASAN.....	5
2.1 Penjelasan Implementasi Algoritma Machine Learning.....	5
2.1.1 Implementasi KNN.....	5
2.1.2 Implementasi Gaussian Naive Bayes.....	8
2.1.3 Implementasi ID3.....	10
2.2 Penjelasan Tahap Cleaning.....	14
2.2.1 Handling Missing Data.....	14
2.2.2 Dealing with Outliers.....	15
2.2.3 Removing Duplicates.....	15
2.2.4 Feature Selection.....	16
2.3 Penjelasan Tahap Preprocessing.....	18
2.3.1 Feature Scaling.....	18
2.3.2 Feature Encoding.....	18
2.4 Hasil Prediksi dan Analisis.....	19
2.4.1 k-Nearest Neighbor.....	19
2.4.2 Gaussian Naive Bayes.....	20
2.4.3 ID3.....	20
BAB III	
KESIMPULAN DAN SARAN.....	22
3.1 Kesimpulan.....	22
3.2 Saran.....	22
BAB IV	
PEMBAGIAN TUGAS.....	23
LAMPIRAN.....	24

BAB I

DESKRIPSI PERSOALAN

Tugas besar ini berfokus pada penerapan algoritma pembelajaran mesin untuk mengklasifikasikan jenis serangan siber pada **dataset UNSW-NB15**. Dataset ini berisi data lalu lintas jaringan yang mencakup berbagai jenis aktivitas, baik yang normal maupun yang terkait dengan serangan siber. Dataset terdiri dari 10 kategori aktivitas, termasuk 9 jenis serangan seperti Fuzzers, Backdoors, DoS (Denial of Service), Exploits, Generic, Reconnaissance, Shellcode, dan Worms, serta 1 kategori aktivitas normal. Variabel target yang digunakan untuk klasifikasi adalah `attack_cat`, yang mewakili kategori jenis serangan.

Tugas ini mengharuskan implementasi algoritma pembelajaran mesin dari awal (*from scratch*) untuk tiga algoritma utama: K-Nearest Neighbors (KNN), Gaussian Naive-Bayes, dan ID3. Implementasi dari algoritma-algoritma tersebut harus mencakup pembuatan kelas-kelas untuk masing-masing algoritma (misalnya, class KNN, class GaussianNB, dan class ID3). Dalam implementasi KNN, algoritma ini harus mampu menerima dua input utama: jumlah tetangga dan metrik jarak antar data seperti Euclidean, Manhattan, dan Minkowski. Gaussian Naive-Bayes diharapkan dapat mengimplementasikan model probabilistik yang berbasis distribusi normal untuk setiap fitur. Sedangkan ID3 harus mencakup pemrosesan data numerik sesuai dengan prinsip-prinsip yang telah diajarkan dalam kuliah.

Selain implementasi *from scratch*, algoritma yang sama juga harus diterapkan menggunakan pustaka `scikit-learn` untuk membandingkan hasil yang diperoleh dari implementasi manual dan pustaka standar. Untuk ID3, implementasi menggunakan `DecisionTreeClassifier` dengan parameter `criterion='entropy'` akan digunakan sebagai pendekatan yang mirip dengan ID3.

Tugas ini juga mencakup berbagai tahap preprocessing, seperti data cleaning, untuk mengatasi masalah seperti nilai yang hilang, data duplikat, atau data yang tidak valid. Data transformation melibatkan encoding variabel kategori, normalisasi atau standarisasi fitur numerik, serta penanganan ketidakseimbangan data. Feature selection berfokus pada pemilihan fitur yang

relevan untuk meningkatkan performa model dan mengurangi kompleksitas. Jika diperlukan, dimensionality reduction seperti PCA (Principal Component Analysis) dapat diterapkan untuk mengurangi jumlah fitur tanpa mengurangi informasi yang penting. Setelah preprocessing, model yang dihasilkan diuji dan divalidasi menggunakan teknik seperti train-test split atau k-fold cross-validation untuk memastikan kinerja yang optimal dalam mengklasifikasikan jenis serangan.

BAB II

PEMBAHASAN

2.1 Penjelasan Implementasi Algoritma *Machine Learning*

2.1.1 Implementasi KNN

```
import numpy as np
from collections import Counter

class KNN:
    def __init__(self, k=3, metric='euclidean', batch_size=1000):
        self.k = k
        self.metric = metric.lower()
        self.batch_size = batch_size
        self.X_train = None
        self.y_train = None

    def fit(self, X, y):
        self.X_train = np.array(X, dtype=np.float32)
        self.y_train = np.array(y)
        self.classes_ = np.unique(y)
        return self

    def _compute_distances_batch(self, X_batch):
        if self.metric == 'euclidean':
            test_norm = np.sum(X_batch**2, axis=1)[:, np.newaxis]
            train_norm = np.sum(self.X_train**2, axis=1)

            distances = np.zeros((X_batch.shape[0],
self.X_train.shape[0]), dtype=np.float32)
            chunk_size = 1000

            for i in range(0, self.X_train.shape[0], chunk_size):
                end_idx = min(i + chunk_size, self.X_train.shape[0])
                distances[:, i:end_idx] = -2 * np.dot(X_batch,
self.X_train[i:end_idx].T)

                distances += test_norm + train_norm
            return np.sqrt(np.maximum(distances, 0))

        elif self.metric == 'manhattan':
            distances = np.zeros((X_batch.shape[0],
self.X_train.shape[0]), dtype=np.float32)
            chunk_size = 1000

            for i in range(0, self.X_train.shape[0], chunk_size):
                end_idx = min(i + chunk_size, self.X_train.shape[0])
                distances[:, i:end_idx] = np.sum(
                    np.abs(X_batch[:, np.newaxis] -
self.X_train[i:end_idx]),
```

```

        axis=2
    )
    return distances

def predict(self, X):
    X = np.array(X, dtype=np.float32)
    predictions = []

    for i in range(0, X.shape[0], self.batch_size):
        batch_end = min(i + self.batch_size, X.shape[0])
        X_batch = X[i:batch_end]

        distances = self._compute_distances_batch(X_batch)

        k_nearest_indices = np.argpartition(distances, self.k,
axis=1)[:self.k]
        k_nearest_labels = self.y_train[k_nearest_indices]

        batch_predictions = []
        for labels in k_nearest_labels:
            batch_predictions.append(Counter(labels).most_common(1)[0][0])

        predictions.extend(batch_predictions)

    return np.array(predictions)

def predict_proba(self, X):
    X = np.array(X, dtype=np.float32)
    all_probabilities = []

    for i in range(0, X.shape[0], self.batch_size):
        batch_end = min(i + self.batch_size, X.shape[0])
        X_batch = X[i:batch_end]

        distances = self._compute_distances_batch(X_batch)
        k_nearest_indices = np.argpartition(distances, self.k,
axis=1)[:self.k]
        k_nearest_labels = self.y_train[k_nearest_indices]

        batch_probabilities = []
        for labels in k_nearest_labels:
            counts = Counter(labels)
            probs = [counts.get(label, 0) / self.k for label in
self.classes_]
            batch_probabilities.append(probs)

        all_probabilities.extend(batch_probabilities)

    return np.array(all_probabilities)

def score(self, X, y):
    return np.mean(self.predict(X) == y)

```

Implementasi K-Nearest Neighbors (KNN) adalah dengan kelas KNN, yang diinisialisasi dengan tiga parameter utama: `k`, `metric`, dan `p`. Parameter `k` menentukan jumlah tetangga terdekat yang digunakan untuk membuat prediksi, sedangkan parameter `metric` memilih metrik jarak yang digunakan (Euclidean, Manhattan, atau Minkowski). Untuk metrik Minkowski, parameter `p` digunakan untuk menentukan nilai pangkat dalam rumus jaraknya. Kelas ini memiliki beberapa metode untuk menghitung jarak antar titik, termasuk `euclidean_distance` untuk menghitung jarak Euclidean, `manhattan_distance` untuk jarak Manhattan, dan `minkowski_distance` untuk jarak Minkowski. Metode `fit()` digunakan untuk menyimpan data training, sementara metode `_compute_distances_batch()` menjadi komponen kunci yang menghitung jarak antar titik data secara efisien menggunakan operasi matriks. Untuk jarak Euclidean, implementasi menggunakan $(a-b)^2 = a^2 + b^2 - 2ab$ untuk menghindari loop eksplisit.

Proses prediksi dilakukan melalui metode `predict()` yang menghitung jarak antara data baru dengan setiap titik data training, kemudian memilih `k` tetangga terdekat menggunakan `np.argsort` yang lebih efisien dibanding pengurutan penuh. Prediksi akhir ditentukan berdasarkan mayoritas label di antara `k` tetangga terdekat menggunakan Counter dari modul `collections`. Implementasi ini juga dilengkapi dengan metode `predict_proba()` untuk menghitung probabilitas kelas dan metode `score()` untuk mengukur akurasi model dengan membandingkan hasil prediksi dengan label sebenarnya. Fitur batch processing memungkinkan kode ini menangani dataset besar dengan lebih efisien dari segi penggunaan memori.

2.1.2 Implementasi Gaussian Naive Bayes

```
class NaiveBayes:
    def __init__(self):
        self.X_train = None
        self.y_train = None
        self.attack_cat = None
        self.prior = {}
        self.mean = {}
        self.variance = {}

    def fit(self, X, y):
        self.X_train = np.array(X)
        self.y_train = np.array(y)

        self.attack_cat = np.unique(y)

        # Calculate prior probabilities
        self.prior = {cat: np.sum(y == cat) / len(y) for cat in self.attack_cat}

        # Calculate mean and variance for each feature and category
        for cat in self.attack_cat:
            indices = np.where(y == cat)[0]
            self.mean[cat] = np.mean(self.X_train[indices], axis=0)
            self.variance[cat] = np.var(self.X_train[indices], axis=0)

        return self

    # Gaussian Probability Density Function
    def gaussian_pdf(self, x, mean, var):
        eps = 1e-9 # smoothing term to avoid divide by zero
        coeff = 1.0 / np.sqrt(2.0 * np.pi * (var + eps))
        exponent = -((x - mean) ** 2) / (2.0 * (var + eps))
        return coeff * np.exp(exponent)

    def predict(self, X):
        X = np.array(X)

        # Ensure the columns of X match the training set
        if X.shape[1] != self.X_train.shape[1]:
            raise ValueError("The feature dimensions of X and X_train do not match.")

        y_pred = []
        for x in X:
            posterior = {}
            for cat in self.attack_cat:
                # Start with log of prior probability
                posterior[cat] = np.log(self.prior[cat])
                # Add log likelihoods for each feature
                for i in range(self.X_train.shape[1]):
                    pdf_value = self.gaussian_pdf(x[i], self.mean[cat][i], self.variance[cat][i])
```



```

        posterior[cat] += np.log(pdf_value + 1e-9) # Add
        eps to avoid log(0)
        # Append the category with the highest posterior
        probability
        y_pred.append(max(posterior, key=posterior.get))

    return np.array(y_pred)

def score(self, X, y):
    y_pred = self.predict(X)
    return np.mean(y_pred == np.array(y))

```

Implementasi Gaussian Naive Bayes ini menggunakan pendekatan probabilistik untuk klasifikasi dengan asumsi bahwa fitur-fitur dalam data bersifat independen dan mengikuti distribusi Gaussian (normal). Kelas NaiveBayes memiliki atribut untuk menyimpan data train (X_{train} dan y_{train}), label kategori serangan (attack_cat), serta probabilitas prior, rata-rata, dan varians untuk setiap kategori. Pada metode `fit`, model mempelajari data dengan menghitung probabilitas prior untuk setiap kategori berdasarkan frekuensi kemunculan dalam data train. Selain itu, untuk setiap kategori, dihitung rata-rata dan varians dari fitur-fitur yang terkait dengan kategori tersebut, yang akan digunakan dalam perhitungan probabilitas untuk prediksi.

Metode `gaussian_pdf` digunakan untuk menghitung fungsi kepadatan probabilitas Gaussian untuk setiap fitur berdasarkan rata-rata dan varians yang dihitung sebelumnya. Dalam metode `predict`, model menghitung posterior probability untuk setiap kategori dengan mengalikan prior probability dengan kepadatan probabilitas Gaussian dari setiap fitur input, lalu mengambil logaritma dari hasilnya untuk menghindari *underflow*. Prediksi diberikan pada kategori dengan posterior probability tertinggi. Metode `score` mengukur akurasi model dengan membandingkan hasil prediksi dengan label yang sebenarnya dan menghitung persentase prediksi yang benar.

2.1.3 Implementasi ID3

```
import numpy as np
import pandas as pd
from collections import Counter
from sklearn.metrics import accuracy_score, classification_report

class ID3:
    def __init__(self):
        self.tree = {}

    def calc_total_entropy(self, train_data, label, class_list):
        total_row = train_data.shape[0]
        total_entr = 0

        for c in class_list:
            total_class_count = train_data[train_data[label] ==
c].shape[0]
            if total_class_count != 0:
                total_class_entr = - (total_class_count / total_row) *
np.log2(total_class_count / total_row)
                total_entr += total_class_entr

        return total_entr

    def calc_entropy(self, feature_value_data, label, class_list):
        class_count = feature_value_data.shape[0]
        entropy = 0

        for c in class_list:
            label_class_count =
feature_value_data[feature_value_data[label] == c].shape[0]
            if label_class_count != 0:
                probability_class = label_class_count / class_count
                entropy_class = - probability_class *
np.log2(probability_class)
                entropy += entropy_class

        return entropy

    def calc_info_gain(self, feature_name, train_data, label,
class_list):
        feature_value_list = train_data[feature_name].unique()
        total_row = train_data.shape[0]
        feature_info = 0.0

        for feature_value in feature_value_list:
            feature_value_data = train_data[train_data[feature_name]
== feature_value]
            feature_value_count = feature_value_data.shape[0]
            feature_value_entropy =
self.calc_entropy(feature_value_data, label, class_list)
            feature_value_probability = feature_value_count /
total_row
```

```

        feature_info += feature_value_probability *
feature_value_entropy

    return self.calc_total_entropy(train_data, label, class_list)
- feature_info

    def find_most_informative_feature(self, train_data, label,
class_list, max_features=None):
        feature_list = train_data.columns.drop(label)
        if max_features:
            feature_list = feature_list[:max_features] # Limit the
number of features to evaluate

        max_info_gain = -1
        max_info_feature = None

        for feature in feature_list:
            feature_info_gain = self.calc_info_gain(feature,
train_data, label, class_list)
            if max_info_gain < feature_info_gain:
                max_info_gain = feature_info_gain
                max_info_feature = feature

        return max_info_feature

    def generate_sub_tree(self, feature_name, train_data, label,
class_list):
        feature_value_count_dict =
train_data[feature_name].value_counts(sort=False)
        tree = {}

        for feature_value, count in feature_value_count_dict.items():
            feature_value_data = train_data[train_data[feature_name]
== feature_value]

            assigned_to_node = False
            for c in class_list:
                class_count =
feature_value_data[feature_value_data[label] == c].shape[0]

                if class_count == count:
                    tree[feature_value] = c
                    train_data = train_data[train_data[feature_name]
!= feature_value]
                    assigned_to_node = True

            if not assigned_to_node:
                tree[feature_value] = "?"

        return tree, train_data

    def make_tree(self, root, prev_feature_value, train_data, label,
class_list, max_depth, current_depth=0):
        print(f"Depth: {current_depth}, Data Shape:

```

```

{train_data.shape}")
    if current_depth >= max_depth or train_data.shape[0] == 0:
        print("Stopping recursion: Max depth reached or no data
left.")
        return

    max_info_feature =
self.find_most_informative_feature(train_data, label, class_list)
    tree, train_data = self.generate_sub_tree(max_info_feature,
train_data, label, class_list)
    next_root = None

    if prev_feature_value is not None:
        root[prev_feature_value] = {}
        root[prev_feature_value][max_info_feature] = tree
        next_root = root[prev_feature_value][max_info_feature]
    else:
        root[max_info_feature] = tree
        next_root = root[max_info_feature]

    for node, branch in list(next_root.items()):
        if branch == "?":
            feature_value_data =
train_data[train_data[max_info_feature] == node]
            self.make_tree(next_root, node, feature_value_data,
label, class_list, max_depth, current_depth + 1)

    def fit(self, train_data, label, max_depth=5):
        self.tree = {}
        class_list = train_data[label].unique()
        self.make_tree(self.tree, None, train_data, label, class_list,
max_depth)
        return self.tree

    def predict(self, instance):
        def _predict(tree, instance):
            if not isinstance(tree, dict):
                return tree
            else:
                root_node = next(iter(tree))
                feature_value = instance[root_node]
                if feature_value in tree[root_node]:
                    return _predict(tree[root_node][feature_value],
instance)
                else:
                    return None

        return _predict(self.tree, instance)

```

Implementasi ID3 disini membangun decision tree menggunakan pendekatan top-down yang berfokus pada pemilihan fitur terbaik untuk pemisahan

berdasarkan perhitungan information gain. Pada kelas ID3, setiap node pohon memiliki atribut seperti fitur yang digunakan untuk pemisahan (feature), nilai yang digunakan untuk pemisahan (value), dan hasil klasifikasi untuk node daun (results). Fungsi entropy digunakan untuk menghitung ketidakpastian atau kebingungannya data, sementara split_data membagi data berdasarkan nilai fitur tertentu. Metode build_tree bertanggung jawab untuk membangun decision tree dengan memilih fitur dan nilai yang memberikan information gain tertinggi pada setiap langkah rekursif. Proses ini berhenti ketika label data di node sudah homogen atau ketika kedalaman pohon mencapai batas maksimum (max_depth).

Pada bagian prediksi, metode predict digunakan untuk mengklasifikasikan sampel data dengan menelusuri decision tree yang telah dibangun. Fungsi _classify melakukan pencarian secara rekursif berdasarkan fitur yang dipilih di setiap node hingga mencapai daun pohon yang memberikan hasil klasifikasi. Untuk evaluasi kinerja model, metode performance_report menghitung akurasi dan memberikan laporan klasifikasi menggunakan metrik seperti precision, recall, dan F1-score. Model ini mengimplementasikan ID3 dari awal, tanpa menggunakan pustaka eksternal, dengan memanfaatkan konsep dasar teori informasi untuk pemilihan fitur yang optimal.

2.2 Penjelasan Tahap *Cleaning*

Data cleaning dilakukan dengan beberapa tahapan: *handling missing data*, *dealing with outliers*, dan *removing duplicates*.

2.2.1 *Handling Missing Data*

Terdapat beberapa metode dalam menangani kasus data yang hilang seperti *imputation* dan *deletion*. Pada dataset yang kami miliki, kami memilih untuk melakukan *data imputation*. Alasan kami memilih metode tersebut adalah karena pada dataset yang kami miliki, diketahui bahwa sekitar 5% datanya yang hilang, dan pada variabel target (*attack_cat*), tidak terdapat *missing value*. Persentase yang rendah tersebut menjadi alasan utama kami memilih untuk melakukan *imputation*. Selain itu, *missing values* tersebar cukup merata di banyak fitur, sehingga apabila dilakukan *deletion*, jumlah data akan banyak berkurang.

Implementasi *imputation* menggunakan strategi yang berbeda berdasarkan tipe data. Pada data kategorikal (*state*, *service*, *proto*) dan data biner (*is_sm_ips_ports*, *is_ftp_login*), digunakan *Most Frequent (Mode) Imputation*. Metode ini dipilih karena dapat menjaga distribusi kategori yang sudah ada dan cocok untuk data yang bersifat diskrit. Khusus untuk data biner, penggunaan nilai yang paling umum adalah pendekatan yang masuk akal karena hanya ada dua nilai.

Untuk data *counter* (*ct_state_ttl*, *ct_srv_src*), data *traffic network* (*bytes*, *packets*, *load*), dan data TCP (*window sizes*, *timing*), digunakan *Median Imputation*. Pemilihan median sebagai metode *imputation* untuk ketiga jenis data ini didasarkan pada karakteristik data yang cenderung memiliki outlier dan tidak terdistribusi normal.

2.2.2 *Dealing with Outliers*

Outlier merupakan nilai data yang memiliki perbedaan yang signifikan dari mayoritas data lain. Biasanya nilainya terlalu besar atau kecil dan tidak sesuai dengan pola dari data lainnya. Terdapat beberapa metode dalam menangani kasus outlier seperti *imputation*, *clipping*, *transformation*, dan *model-based*.

Pada fitur kategorikal, digunakan metode penanganan kategori *rare* di mana kategori yang memiliki proporsi $< 1\%$ menjadi kategori 'Other'. Tujuannya adalah untuk mengurangi dimensi data dan menghindari overfitting yang mungkin disebabkan oleh kategori yang sangat jarang muncul.

Pada fitur numerik, dilakukan pembagian menjadi beberapa kelompok berdasarkan karakteristiknya: fitur waktu, fitur byte, fitur paket, fitur hitungan, dan fitur TCP. Untuk fitur waktu seperti durasi (*dur*) dan jitter (*sjit*, *djit*), digunakan metode *percentile-based clipping* dengan rentang 1%-99% diikuti dengan transformasi logaritmik. Pendekatan ini tepat karena data waktu sering memiliki distribusi yang *skewed* dan nilai ekstrim yang sangat tinggi.

Pada fitur byte dan paket, digunakan metode IQR dengan batas 1,5 kali IQR dari Q1 dan Q3. Untuk fitur byte, dilakukan penambahan transformasi logaritmik setelah clipping, yang mana masuk akal mengingat data byte sering memiliki variasi nilai yang sangat besar. Hal ini juga memastikan nilai tidak negatif dengan menggunakan $\max(0, \text{lower})$ sebagai batas bawah.

Untuk fitur *count* (*ct_**), digunakan pendekatan yang mirip dengan IQR, namun menambahkan transformasi akar kuadrat. Transformasi ini membantu menstabilkan variansi. Sementara untuk fitur TCP, hanya dilakukan clipping pada nilai negatif karena fitur-fitur ini sudah memiliki rentang nilai yang relatif stabil. Fitur numerik sisanya ditangani dengan metode IQR standar.

2.2.3 *Removing Duplicates*

Penanganan data duplikat penting untuk dilakukan karena hal tersebut dapat menurunkan akurasi dari analisis. Data duplikat dapat menyebabkan bias pada

model *machine learning*, *overfitting*, dan mengurangi kemampuannya dalam menanggapi data yang baru. Selain itu, ukuran dataset juga meningkat secara tidak wajar.

Pada dataset yang kami miliki, kami menghapus semua data duplikat lalu mempertahankan kemunculan pertama dari setiap data. Hasil analisis kemudian disimpan dalam *dictionary* yang berisi informasi penting seperti jumlah baris awal, jumlah duplikat yang ditemukan, jumlah baris setelah penghapusan, total baris yang dihapus, dan contoh baris duplikat.

Dari aspek konsistensi data, mempertahankan entry pertama sering kali lebih aman karena data pertama cenderung memiliki kualitas yang lebih baik, belum mengalami proses duplikasi yang mungkin menimbulkan error atau modifikasi tidak diinginkan.

Sementara itu, penyimpanan informasi detail dalam dictionary membantu dalam dokumentasi proses pembersihan data, memudahkan dalam melakukan tracking perubahan yang terjadi pada dataset, dan memvalidasi bahwa proses pembersihan data berjalan sesuai dengan yang diharapkan

2.2.4 *Feature Selection*

Pada tahap ini, kami mengidentifikasi dan menghapus fitur yang tidak relevan atau tidak memberikan kontribusi signifikan terhadap model. Kami menggunakan visualisasi seperti violin plot dan analisis korelasi untuk mengevaluasi pengaruh masing-masing fitur terhadap variabel target. Violin plot membantu untuk memvisualisasikan distribusi data setiap fitur berdasarkan kategori target, sehingga dapat dengan mudah terlihat apakah suatu fitur memiliki distribusi yang jelas atau variabilitas yang signifikan. Fitur-fitur yang tampak tidak memberikan informasi yang cukup atau memiliki distribusi yang serupa di seluruh kategori target dihapus karena dianggap tidak relevan. Selain itu, analisis korelasi membantu untuk mengidentifikasi fitur yang memiliki

hubungan yang sangat kuat satu sama lain, yang berpotensi menyebabkan multikolinearitas meskipun terdapat asumsi data berdistribusi gaussian. *Feature* yang kami hapus adalah 'dur', 'sloss', 'sload', 'dload', 'dpkts', 'ct_srv_dst', 'ct_dst_ltm', 'ct_dst_sport_ltm', 'swin', 'dtepb', 'smean', 'djit', 'dinpkt', 'synack', 'ackdat'.

2.3 Penjelasan Tahap *Preprocessing*

Preprocessing dilakukan dalam beberapa tahapan: feature scaling dan feature encoding

2.3.1 *Feature Scaling*

Feature scaling adalah teknik preprocessing untuk menstandarkan rentang dari variabel bebas atau fitur pada data. Tujuan utama feature scaling adalah untuk memastikan setiap fitur berkontribusi yang setara pada proses training supaya algoritma machine learning dapat bekerja secara efektif terhadap data

Terdapat beberapa metode dalam melakukan feature scaling. Metode Min-max scaling menghasilkan skala pada rentang yang spesifik, biasanya $[0, 1]$. Metode Standardization menskalakan fitur agar memiliki nilai rata-rata (mean) 0 dari simpangan baku (standar deviasi) 1. Metode Robust Scaling menskalakan fitur berdasarkan rentang interkuartil dan kurang terpengaruh oleh outlier. Ketiga metode ini digunakan pada dataset kami sesuai karakteristik dari data, dan pada implementasinya, sebagian besar menggunakan metode Min-max dan Standardization.

2.3.2 *Feature Encoding*

Feature encoding adalah proses mengubah data kategorikal (non-numerik) menjadi format numerik sehingga dapat digunakan sebagai input untuk algoritma machine learning. Terdapat dua macam data kategorikal: nominal, yaitu kategori tanpa urutan tertentu, misal warna, nama negara, dan lain-lain; ordinal, yaitu kategori dengan urutan tertentu, misal tingkat pendidikan.

Terdapat tiga metode yang umum digunakan dalam feature encoding. Pertama adalah Label Encoding. Label Encoding memberikan sebuah bilangan unik untuk setiap kategori. Kedua, One-Hot Encoding. One-Hot Encoding mengkonversikan tiap kategori menjadi vektor biner. Ketiga, Target Encoding. Target Encoding menggantikan tiap kategori dengan rata-rata dari variabel target yang bersesuaian. Pada dataset ini, kami memanfaatkan masing-masing metode sesuai dengan karakteristik dari masing-masing data kategorikal

2.4 Hasil Prediksi dan Analisis

2.4.1 *k-Nearest Neighbor*

From Scratch	Sklearn
Training data shape: (113587, 41) Validation data shape: (30784, 41) Accuracy: 0.7490254677754677	Training data shape: (113587, 41) Validation data shape: (30784, 41) Accuracy: 0.7454196985446986

Hasil yang diperoleh dari kedua implementasi kNN menunjukkan bahwa akurasi algoritma kNN dari sklearn mencapai 74,54%, sementara implementasi from scratch mencatatkan akurasi 74,90%, dengan selisih yang sangat kecil, yakni hanya 0,3%. Perbedaan yang sangat kecil ini menunjukkan bahwa implementasi from scratch mampu mencapai performa yang hampir setara dengan implementasi sklearn. Hal ini mengindikasikan bahwa algoritma yang dibangun secara manual sudah cukup bagus dan dapat diandalkan. Hasil yang serupa ini juga mengonfirmasi bahwa implementasi from scratch telah mengikuti prinsip-prinsip dasar kNN dengan benar, termasuk dalam perhitungan jarak dan pemilihan tetangga terdekat.

Dengan ukuran dataset yang cukup besar (113.587 data train dan 30.784 data validasi), selisih akurasi yang hanya 0,3% menunjukkan bahwa implementasi from scratch dapat menangani dataset besar dengan baik dan menghasilkan hasil yang konsisten. Perbedaan akurasi yang sedikit lebih tinggi pada sklearn kemungkinan disebabkan oleh optimasi internal yang lebih baik dalam library tersebut, seperti penanganan kasus khusus atau algoritma pencarian kNN yang lebih efisien. Secara keseluruhan, implementasi kNN from scratch telah berhasil mendekati performa sklearn dengan sangat baik, membuktikan bahwa dasar-dasar algoritma telah diterapkan dengan tepat. Selisih kecil dalam akurasi bisa dimaklumi karena sklearn sudah sangat teroptimasi dan matang.

2.4.2 Gaussian Naive Bayes

From Scratch	Sklearn
Accuracy: 0.5233238045738046	Accuracy: 0.5323544698544699

Berdasarkan perbandingan antara implementasi Naive Bayes menggunakan sklearn dan yang dibuat dari scratch, terlihat bahwa kedua model menghasilkan akurasi yang hampir sama. Model Naive Bayes dari sklearn memperoleh akurasi 53,24%, sementara implementasi dari scratch mencapai 52,33%, dengan selisih yang sangat kecil, hanya 0,91%. Perbedaan akurasi yang begitu kecil menunjukkan bahwa implementasi dari scratch berhasil mengimplementasikan fungsi dasar algoritma Naive Bayes dengan baik, termasuk dalam perhitungan probabilitas prior dan likelihood.

Meski kedua implementasi menunjukkan hasil yang konsisten, akurasi yang relatif rendah (sekitar 50%) mengindikasikan bahwa Naive Bayes mungkin bukan model yang paling cocok untuk dataset ini. Hal ini bisa disebabkan oleh beberapa faktor, seperti asumsi independensi fitur yang tidak terpenuhi, distribusi data yang tidak sesuai dengan asumsi Gaussian, atau kompleksitas masalah yang membutuhkan model yang lebih canggih. Namun, dari sisi implementasi, kesamaan hasil antara kedua model membuktikan bahwa implementasi dari scratch telah mengikuti prinsip dasar Naive Bayes dengan benar. Perbedaan kecil dalam akurasi kemungkinan besar disebabkan oleh perbedaan dalam penanganan kasus khusus atau optimasi numerik yang ada di dalam library sklearn, misalnya penggunaan variabel *smoothing* dan ruang logaritmik.

2.4.3 ID3

From Scratch	Sklearn
Accuracy: 0.06084329521829522	Model trained successfully! Accuracy: 0.5811785343035343

Pada dasarnya, model ID3 from scratch yang telah dibuat masih gagal menangkap pola yang akurat dengan constraint yang diberikan saat ini (10000 training data dan $\text{max_depth} = 5$) dilihat dari perbedaan akurasi yang sangat signifikan.

Dengan demikian, rasanya diperlukan constraint yang lebih besar berupa menggunakan 113.587 data validasi untuk training model serta memvariasi max_depth yang lebih besar seperti $\text{max_depth} = 10$.

BAB III

KESIMPULAN DAN SARAN

3.1 Kesimpulan

Pada model yang kami buat, sejauh ini KNN menghasilkan hasil yang paling akurat disusul dengan Gaussian NB dan ID3. Namun, secara teori, seharusnya ID3 bisa menghasilkan performance yang jauh lebih baik jika diberikan waktu untuk memproses data yang lebih banyak untuk training beserta dengan max_depth yang lebih besar untuk iterasi rekursif yang lebih dalam.

3.2 Saran

Terdapat beberapa saran yang ingin kami sampaikan dalam pengerjaan tugas besar ini, yaitu:



- Mendalami metode-metode pre-processing dan hyperparameter tuning yang lebih efektif melihat akurasi yang masih jauh dari akurat
- Mengeksplorasi lebih banyak model seperti log regression yang mungkin saja dapat menghasilkan hasil yang lebih akurat untuk dataset ini
- Menguji model pada beberapa dataset berbeda selain daripada UNSW-15 jika ada waktu untuk memvalidasi keakuratan model

BAB IV

PEMBAGIAN TUGAS

NIM	Nama	Tugas
13522018	Ibrahim Ihsan Rasyid	Laporan
13522053	Erdianti Wiga Putri Andini	EDA, kNN, Preprocessing, Laporan
13522097	Ellijah Darrellshane Suryanegara	ID3, Laporan
13522114	Muhammad Dava Fathurrahman	Gaussian Naive Bayes, Preprocessing, Laporan

LAMPIRAN

- Link Github: https://github.com/wigaandini/Tubes2_AI_08
- Link Notebook:  AI_Tubes2_Kelompok 8
- Spesifikasi Tugas:  Spesifikasi Tugas Besar 2 IF3170 Inteligensi Artifisial 2024/2025