



# **Laporan Tugas Besar 2**

## **Implementasi Algoritma Pembelajaran Mesin**

Oleh:

Kelompok 10

Anggota:

1. Angelica Kierra Ninta Gurning - 13522048
2. Imanuel Sebastian Girsang - 13522058
3. Marzuli Suhada M - 13522070
4. Muhammad Neo Cicero Koda - 13522108

**IF3170 - Inteligensi Artifisial**

**2024**

## Daftar Isi

Daftar Isi.....	2
I. Penjelasan Implementasi KNN.....	3
II. Penjelasan Implementasi Naive-Bayes.....	5
III. Penjelasan Implementasi ID3.....	7
IV. Penjelasan Tahap Cleaning dan Preprocessing.....	8
V. Komparasi Hasil Prediksi antara Implementasi Manual dan Pustaka.....	15
VI. Kontribusi Anggota.....	15
VII. Referensi.....	16

## I. Penjelasan Implementasi KNN

Algoritma KNN (K-nearest Neighbor) merupakan algoritma *supervised learning* dimana hasil dari instance baru akan diberikan klasifikasi berdasarkan k-terdekat tetangga (berdasarkan jarak). Algoritma ini merupakan algoritma berbasis *lazy learning*. *Lazy learning* artinya, proses pembelajaran tidak ada sampai data *test* diberikan. Algoritma hanya menyimpan data *training* dan melakukan perhitungan ketika diminta untuk melakukan prediksi. KNN akan bekerja dengan membandingkan *training set* dengan *test set*. Prediksi dibuat berdasarkan data baru dan data *training*.

Pada implementasi KNN di tugas ini, terdapat tiga algoritma penentuan jarak, yaitu : Euclidean, Manhattan, dan Minkowski.

- a. Jarak Euclidean : merupakan cara untuk mengukur jarak terpendek (garis lurus) antara dua titik dalam ruang euclidean.

$$d(p, q) = \sqrt{\sum_{i=1}^n |p_i - q_i|^2}$$

- b. Jarak Manhattan : mengukur jarak berdasarkan garis horizontal dan vertikal.

$$d(p, q) = \sum_{i=1}^n |p_i - q_i|$$

- c. Jarak Minkowski : merupakan generalisasi dari jarak euclidean dan manhattan.

$$d(p, q) = \left( \sum_{i=1}^n |p_i - q_i|^p \right)^{1/p}$$

```
1 def euclidean_distance(x1, x2):
2     return np.sqrt(np.sum((x1 - x2) ** 2))
3
4 def manhattan_distance(x1, x2):
5     return np.sum(np.abs(x1 - x2))
6
7 def minkowski_distance(x1, x2, p=3):
8     return np.sum(np.abs(x1 - x2) ** p) ** (1 / p)
```

```

1 class KNN:
2     def __init__(self, k=3, metric='euclidean', p=3):
3         self.k = k
4         self.metric = metric
5         self.p = p
6         self.X_train = None
7         self.y_train = None
8
9     def fit(self, X, y):
10        self.X_train = X
11        self.y_train = y
12
13    def _calculate_distance(self, x1, x2):
14        if self.metric == 'euclidean':
15            return euclidean_distance(x1, x2)
16        elif self.metric == 'manhattan':
17            return manhattan_distance(x1, x2)
18        elif self.metric == 'minkowski':
19            return minkowski_distance(x1, x2, self.p)
20        else:
21            raise ValueError("Choose another metric, only euclidean, manhattan, and minkowski are supported")
22
23    def predict(self, X):
24        predictions = []
25        for x in X:
26            distances = [self._calculate_distance(x, x_train) for x_train in self.X_train]
27
28            k_indices = np.argsort(distances)[:self.k]
29
30            k_nearest_labels = [self.y_train[i] for i in k_indices]
31
32            unique_labels, counts = np.unique(k_nearest_labels, return_counts=True)
33            most_common = unique_labels[np.argmax(counts)]
34
35            predictions.append(most_common)
36        return np.array(predictions)
37
38    def score(self, X, y):
39        y_pred = self.predict(X)
40        accuracy = np.mean(y_pred == y)
41        return accuracy
42
43
44
45 import numpy as np
46 import concurrent.futures
47 from typing import Union, List
48
49 class KNN:
50     def __init__(self, n_neighbors: int = 5, metric: str = "euclidean", p: Union[int, None] = None, max_workers: int = None):
51         self.n_neighbors = n_neighbors
52         self.metric = metric
53         self.p = p
54         self.max_workers = max_workers
55
56         if not isinstance(n_neighbors, int):
57             raise TypeError(f"n_neighbors must be an integer, got {type(n_neighbors).__name__} instead.")
58
59         if self.metric == "manhattan" and self.p not in (None, 1):
60             raise ValueError("when metric='manhattan', p should be None or 1.")
61
62         if self.metric == "euclidean" and self.p not in (None, 2):
63             raise ValueError("when metric='euclidean', p should be None or 2.")
64
65         if metric == "minkowski" and p is None:
66             raise ValueError("For metric='minkowski', you must specify a value for p.")
67
68     def fit(self, X: np.ndarray, y: np.ndarray):
69         self.X = np.asarray(X)
70         self.y = np.asarray(y)
71         return self
72
73     def _calculate_distance(self, x: np.ndarray, training_point: np.ndarray) -> float:
74         if self.metric == "manhattan":
75             return np.sum(np.abs(training_point - x))
76         elif self.metric == "euclidean":
77             return np.linalg.norm(training_point - x)
78         elif self.metric == "minkowski":
79             return np.sum(np.abs(training_point - x) ** self.p) ** (1 / self.p)
80
81     def _predict_single(self, x: np.ndarray) -> int:
82         distances = [self._calculate_distance(x, point) for point in self.X]
83
84         nearest_indices = np.argsort(distances)[:self.n_neighbors]
85
86         nearest_labels = self.y[nearest_indices]
87
88         unique, counts = np.unique(nearest_labels, return_counts=True)
89
90         return unique[np.argmax(counts)]
91
92     def predict(self, X: np.ndarray) -> np.ndarray:
93         with concurrent.futures.ThreadPoolExecutor(max_workers=self.max_workers) as executor:
94             predictions = list(executor.map(self._predict_single, X))
95
96         return np.array(predictions)

```

Kelas KNN memiliki lima properti utama, yaitu `k`, `metric`, `p`, `x_train`, `y_train`. '`k`' merupakan jumlah nearest neighbor yang akan diambil. '`Metric`' merupakan cara yang digunakan untuk menghitung jarak. '`p`' akan digunakan jika metrik pencarian jarak yang digunakan adalah minkowski distance. Jika '`p`' yang dipilih adalah 2 maka minkowski distance akan menjadi Euclidean Distance, jika '`p`' yang dipilih adalah 1 maka minkowski distance akan menjadi Manhattan Distance. Berikut merupakan penjelasan dari tiap method yang ada:

1. Metode `fit`

Metode yang digunakan untuk melatih model dengan menyimpan *training data* dan labelnya

2. Metode `_calculate_distance`

Metode yang digunakan untuk menghitung jarak sesuai metrik yang dipilih

3. Metoda `predict`

Metode yang digunakan untuk menghitung prediksi label untuk data baru berdasarkan *training set*. Untuk tiap sampel uji  $x$  akan dihitung jarak dengan setiap sampel di `self.x_train`. Setelah itu jarak terkecil diurutkan dan dipilih sebanyak '`k`' buah. Selanjutnya mengambil label dari `y_train` sesuai dengan jarak yang terpilih. Kemudian akan dihitung kemunculan tiap label tetangga dan akan dipilih label dengan frekuensi tertinggi.

## II. Penjelasan Implementasi Naive-Bayes

Algoritma Naive Bayes merupakan algoritma yang didasarkan pada Teorema Bayes. Algoritma ini memiliki asumsi bahwa semua fitur merupakan saling independen satu sama lain (*conditionally independent*).

Pada Implementasi ini algoritma naive bayes yang digunakan adalah Gaussian Naive Bayes. Algoritma bayes ini mengasumsikan bahwa setiap fitur dalam dataset mengikuti distribusi Gaussian (normal) untuk tiap kelas. Dalam Gaussian Naive Bayes probabilitas likelihood dihitung menggunakan formula,

$$P(X|C) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(X-\mu)^2}{2\sigma^2}}$$

Untuk tiap fitur pada dataset, algoritma akan menghitung nilai mean dan variansi berdasarkan *training set* untuk tiap kelas. Selama proses prediksi, algoritma menghitung probabilitas untuk tiap kelas menggunakan Teorema Bayes dan memilih kelas dengan probabilitas tertinggi.



```

1 # Gaussian Naive Bayes from scratch
2 class GaussianNaiveBayes:
3     def __init__(self):
4         self.class_probs = None
5         self.feature_stats = None
6
7     def fit(self, X, y):
8         """
9         Fit the model using X as training data and y as labels
10        """
11        self.class_probs = self._compute_class_probs(y)
12        self.feature_stats = self._compute_feature_stats(X, y)
13
14    def _compute_class_probs(self, y):
15        """
16        Compute probabilities of each class
17        """
18        class_probs = defaultdict(lambda: 0)
19        total_samples = len(y)
20
21        for label in y:
22            class_probs[label] += 1
23
24        for label in class_probs:
25            class_probs[label] /= total_samples
26
27        return class_probs
28
29    def _compute_feature_stats(self, X, y):
30        """
31        Compute mean and variance of each feature ()
32        """
33        feature_stats = defaultdict(lambda: defaultdict(lambda: {'mean': 0, 'var': 0, 'count': 0}))
34
35        # Compute sum, squared sum, and count of each feature
36        for i in range(len(X)):
37            label = y.iloc[i]
38            for j in range(len(X[i])):
39                feature_value = X[i][j]
40                feature_stats[label][j]['mean'] += feature_value
41                feature_stats[label][j]['var'] += feature_value ** 2
42                feature_stats[label][j]['count'] += 1
43
44        # Compute mean and variance by dividing by count
45        for label in feature_stats:
46            for feature_index in feature_stats[label]:
47                count = feature_stats[label][feature_index]['count']
48                mean = feature_stats[label][feature_index]['mean'] / count
49                variance = (feature_stats[label][feature_index]['var'] / count) - mean ** 2
50                feature_stats[label][feature_index]['mean'] = mean
51                feature_stats[label][feature_index]['var'] = variance
52
53        return feature_stats
54
55    def _gaussian_pdf(self, x, mean, var):
56        """
57        The probability density function based on Gaussian distribution
58        """
59        if var == 0:
60            return 1 if x == mean else 0
61        coefficient = 1 / sqrt(2 * pi * var)
62        exponent = exp(-(x - mean) ** 2 / (2 * var))
63        return coefficient * exponent
64
65    def predict(self, X):
66        """
67        Predict class labels for input data X
68        """
69        predictions = []
70
71        for sample in X:
72            class_probs = self.class_probs.copy()
73            for label in self.class_probs:
74                prob = np.log(class_probs[label]) # Use log to avoid numerical underflow
75
76                for feature_index in range(len(sample)):
77                    mean = self.feature_stats[label][feature_index]['mean']
78                    var = self.feature_stats[label][feature_index]['var']
79                    feature_value = sample[feature_index]
80                    prob += np.log(self._gaussian_pdf(feature_value, mean, var))
81
82                class_probs[label] = prob
83
84            # Get the class with the highest log-probability
85            predicted_class = max(class_probs, key=class_probs.get)
86            predictions.append(predicted_class)
87
88        return np.array(predictions)

```

Kelas `GaussianNaiveBayes` memiliki dua properti utama, yaitu `self.class_probs` (probabilitas untuk tiap kelas) dan `self.feature_stats` (statistik untuk tiap fitur untuk tiap kelas). Penjelasan metode lainnya adalah sebagai berikut:

1. Metode `fit`

Merupakan metode yang digunakan untuk menghitung probabilitas kelas dan statistik fitur untuk tiap kelas

2. Metode `_compute_class_probs`

Merupakan probabilitas prior (jumlah sampel dari kelas tertentu dibagi dengan total sampel). Hasil probabilitas disimpan pada sebuah *dictionary*

3. Metode `_compute_feature_stats`

Merupakan metode untuk menghitung mean dan varians untuk tiap fitur pada tiap kelas

4. Metode `_gaussian_pdf`

Merupakan metode untuk menghitung probabilitas distribusi Gaussian.

5. Metode `predict`

Merupakan metode untuk memprediksi kelas untuk *testing data*. Untuk tiap kelas  $C$  akan dihitung  $\log(P(C))$ , selanjutnya akan ditambahkan  $\log(P(X|C))$ . Kelas akan dipilih berdasarkan log-probabilitas tertinggi. Probabilitas log digunakan untuk mencegah underflow numerik, terutama untuk kasus probabilitas yang sangat kecil

### III. Penjelasan Implementasi ID3

ID3 merupakan salah satu algoritma berbasis pohon keputusan yang memanfaatkan greedy search untuk melakukan klasifikasi. Algoritma ini membangun pohon keputusan dengan memilih atribut yang memberikan *information gain* tertinggi pada setiap langkah pembagian (*split*).

Entropi merupakan ukuran ketidakpastian atau keacakan dalam suatu kumpulan data. Semakin tinggi entropi, semakin tidak pasti atau acak distribusi data tersebut. Dalam algoritma ID3, entropi digunakan untuk mengukur "kebersihan" suatu pembagian data, dengan rumus sebagai berikut:

$$E(S) = - \sum_{i=1}^k p_i \log_2(p_i)$$



Dimana

1.  $S$  = Kumpulan data
2.  $K$  = Jumlah data unik di data  $S$
3.  $p_i$  = Banyaknya kelas ke -  $i$  di dataset  $S$

Information gain merupakan ukuran penurunan entropi yang diperoleh dari membagi data berdasarkan suatu atribut tertentu. Semakin tinggi information gain dari suatu atribut, semakin baik atribut tersebut dalam membagi data untuk meminimalkan ketidakpastian.

$$IG(S, A) = H(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} H(S_v)$$

Dimana

1.  $S$  = Kumpulan data
2.  $A$  = Atribut
3.  $H(S)$  = Entropi awal
4.  $S(V)$  = Subset data dari  $S$  untuk nilai  $V$  pada atribut  $A$

Untuk melakukan penanganan terhadap atribut kontinu, hal yang dilakukan adalah:

1. Melakukan *sorting* terhadap nilai-nilai atribut kontinu
2. Mencari *potential optimal breakpoint*, yaitu angka-angka pertemuan dimana terjadi perubahan atribut target
3. Mencari breakpoint terbaik dari setiap potensi tersebut dengan menghitung *information gain*.

Kelas ID3 memiliki dua properti utama yaitu `self.tree` (pohon keputusan yang terbentuk dari algoritma ini) serta `self.fallbackValue`. Penjelasan metode lainnya adalah sebagai berikut:

1. Metode `fit`

Merupakan metode yang digunakan untuk membentuk pohon keputusan dari data latih yang diberikan

2. Metode `_plurality_value`

Merupakan metode yang digunakan untuk menentukan label target paling *common* dari sekumpulan dataset.

3. Metode *entropy*

Merupakan metode untuk menghitung entropi dari suatu dataset dengan menggunakan rumus yang telah dijelaskan sebelumnya

4. Metode *information\_gain*

Merupakan metode untuk menghitung *information gain* dengan rumus yang telah dijelaskan sebelumnya.

5. Metode *build\_tree*

Merupakan metode untuk membangun pohon keputusan. Metode ini bekerja secara rekursif dengan memanggil dirinya sendiri hingga salah satu dari kasus basis yang ada dicapai. Terdapat 3 kasus basis:

1. Apabila *example* yang ada kosong, kembalikan *plurality value* dari *parentnya*
2. Apabila
3. Apabila atribut sudah habis, maka kembalikan *plurality value* dari *examples* saat ini.

6. Metode *predict*

Metode ini digunakan untuk melakukan prediksi terhadap data-data yang diberikan.

7. Metode *score*

Metode ini digunakan untuk melakukan penilaian secara akurasi dari hasil prediksi model yang telah dibuat

8. Metode *serialize\_to\_pickle*

Metode untuk melakukan *serializing* dari ID3 terkait kedalam bentuk binary agar bisa kemudian di *load*.

#### IV. Penjelasan Tahap Cleaning dan Preprocessing

Terdapat beberapa tahapan yang dilakukan untuk tahap *cleaning*, antara lain Handling Missing Data, Dealing with Outliers, Data Validation, Removing Duplicates, Feature Engineering. Berikut merupakan rincian dari tahap *cleaning*:

1. Handling Missing Data

Pada implementasi kali ini fitur numerical dan kategorikal akan dipisah. Selain itu akan digunakan SimpleImputer dari sklearn untuk menangani nilai-nilai yang kosong. Untuk fitur numerical, akan dipilih median sebagai pengganti data yang hilang. Untuk fitur kategorikal akan dipilih kategori yang paling banyak muncul sebagai pengganti data yang hilang.

```
1 from sklearn.impute import SimpleImputer
2
3 # Pisahkan kolom numerik dan kategorikal
4 numerical_cols = train_set.select_dtypes(include=['float64', 'int64']).columns
5 categorical_cols = train_set.select_dtypes(include=['object']).columns
6
7 # 1. Tangani missing values di kolom numerik dengan median
8 num_imputer = SimpleImputer(strategy='median')
9 train_set[numerical_cols] = num_imputer.fit_transform(train_set[numerical_cols])
10 val_set[numerical_cols] = num_imputer.transform(val_set[numerical_cols])
11
12 # 2. Tangani missing values di kolom kategorikal dengan modus
13 cat_imputer = SimpleImputer(strategy='most_frequent')
14 train_set[categorical_cols] = cat_imputer.fit_transform(train_set[categorical_cols])
15 val_set[categorical_cols] = cat_imputer.transform(val_set[categorical_cols])
```

Alasan menggunakan median digunakan pada fitur numerik karena lebih robust terhadap outlier dibandingkan rata-rata (mean). Sedangkan penggunaan modus cocok untuk fitur kategorikal karena menjaga distribusi data dengan memilih nilai yang paling umum.

## 2. Dealing with Outliers

Pada implementasi kali ini, tiap kolom akan dikelompokkan berdasarkan jumlah outlier pada data. Akan ada tiga jenis yaitu *high* (>20.000 outliers) , *medium* (10.000-20.000 outliers), dan *low* (< 10.000 outliers).

Untuk data dalam kategori *high*, akan digunakan teknik clipping, yaitu teknik untuk membatasi nilai dengan suatu upper atau lower bound. Untuk data dalam kategori *medium* , outlier akan diganti oleh median. Untuk data dalam kategori *low* maka akan dilakukan log transformation.

```

1 # Daftar kolom berdasarkan jumlah outlier
2 # Outlier > 20.000
3 high_outlier_cols = ['ct_dst_ltm', 'ct_src_dport_ltm', 'ct_dst_sport_ltm',
4                     'ct_dst_src_ltm', 'sbytes', 'dbytes', 'dloss', 'dload',
5                     'spkts', 'dpkts', 'smean', 'dmean', 'synack']
6
7 # Outlier 10.000 - 20.000
8 medium_outlier_cols = ['dur', 'sloss', 'sload', 'sjit', 'djit', 'sinpkt', 'dinpkt', 'tcprtt']
9
10 # Outlier < 10.000
11 low_outlier_cols = ['ct_state_ttl', 'is_sm_ips_ports', 'is_ftp_login', 'ct_ftp_cmd',
12                   'ct_srv_src', 'ct_srv_dst', 'response_body_len', 'ackdat']
13
14 # 1. Handling Outliers: High Outliers -> Clipping
15 for col in high_outlier_cols:
16     Q1 = train_set[col].quantile(0.25)
17     Q3 = train_set[col].quantile(0.75)
18     IQR = Q3 - Q1
19     lower_bound = Q1 - 1.5 * IQR
20     upper_bound = Q3 + 1.5 * IQR
21
22     # Clipping pada train_set
23     train_set[col] = train_set[col].clip(lower=lower_bound, upper=upper_bound)
24
25     # Clipping pada val_set (gunakan nilai dari train_set)
26     val_set[col] = val_set[col].clip(lower=lower_bound, upper=upper_bound)
27
28 # 2. Handling Outliers: Medium Outliers -> Replace with Median
29 for col in medium_outlier_cols:
30     Q1 = train_set[col].quantile(0.25)
31     Q3 = train_set[col].quantile(0.75)
32     IQR = Q3 - Q1
33     lower_bound = Q1 - 1.5 * IQR
34     upper_bound = Q3 + 1.5 * IQR
35
36     median = train_set[col].median()
37
38     # Imputation pada train_set
39     train_set[col] = train_set[col].apply(lambda x: median if x < lower_bound or x > upper_bound else x)
40
41     # Imputation pada val_set
42     val_set[col] = val_set[col].apply(lambda x: median if x < lower_bound or x > upper_bound else x)
43
44 # 3. Handling Outliers: Low Outliers -> Log Transformation
45 for col in low_outlier_cols:
46     train_set[col] = np.log1p(train_set[col]) # log1p untuk memastikan nilai nol aman
47     val_set[col] = np.log1p(val_set[col])    # lakukan hal yang sama pada val_set

```

Clipping membatasi pengaruh outlier ekstrim dengan mengatur batas nilai. Median digunakan untuk kategori *medium* karena tidak terpengaruh oleh outlier, sehingga tetap mewakili distribusi data. Log transformation digunakan untuk kategori *low* karena efektif mengatasi skala data yang sangat lebar, khususnya pada distribusi data yang miring (skewed).

### 3. Removing Duplicates

Untuk implementasi pada tugas ini, nilai yang duplikat akan dihapus dari dataset.

```

1 train_set = train_set.drop_duplicates().reset_index(drop=True)
2 val_set = val_set.drop_duplicates().reset_index(drop=True)

```

Duplikasi data dapat menyebabkan bias dalam model machine learning. Menghapus duplikat akan menjaga kualitas data dan meningkatkan performa model.

#### 4. Feature Engineering

Untuk feature engineering pada tugas ini dilakukan menggunakan berbasis model. Terdapat model XGBoost yang dilatih untuk mengambil *feature importance* berdasarkan nilai *gain*, sehingga dapat diketahui fitur mana yang paling berkontribusi pada pembuatan prediksi.

```
1 import xgboost as xgb
2 import matplotlib.pyplot as plt
3 from sklearn.preprocessing import LabelEncoder
4 import pandas as pd
5
6 # Select only numeric columns from the training set
7 train_set_numeric_fs = train_set.select_dtypes(include=["int64", "float64"])
8
9 # Encode the categorical target variable into numeric labels
10 label_encoder = LabelEncoder()
11 y_train_encoded_fs = label_encoder.fit_transform(y_train)
12
13 # Train XGBoost model using only numeric columns
14 model = xgb.XGBClassifier()
15 model.fit(train_set_numeric_fs, y_train_encoded_fs)
16
17 # Get feature importances
18 importances = model.get_booster().get_score(importance_type='gain')
19
20 # Convert importances to a DataFrame
21 importance_df = pd.DataFrame(importances.items(), columns=['Feature', 'Gain'])
22
23 # Sort the features by 'Gain' in descending order and select the top
24 top_numeric_df = importance_df.sort_values(by='Gain', ascending=False).head(10)
25
26 # Display the Top features
27 print("Top features by Gain:")
28 print(top_numeric_df)
29
30 # Save the top features in an array
31 top_numeric = top_numeric_df['Feature'].values
32
33 # Optionally, print the array
34 print("\nTop 10 features as an array:")
35 print(top_numeric)
```

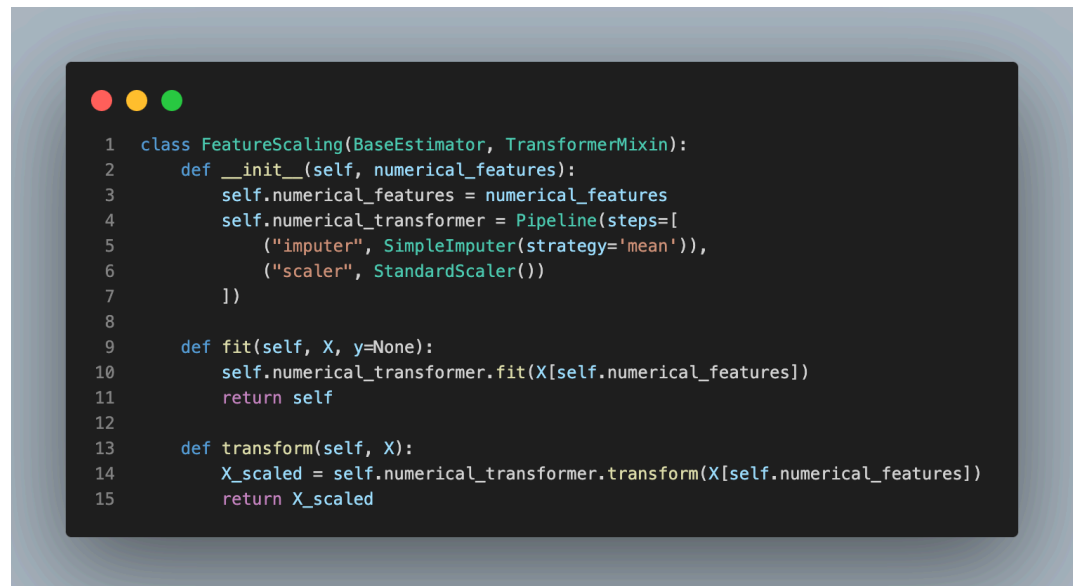
Feature importance membantu menyaring fitur yang relevan dan signifikan. Mengurangi fitur yang tidak penting dapat meningkatkan performa model dan mengurangi waktu komputasi.

Pada tahap *preprocessing*, dilakukan sejumlah tahapan seperti Feature Scaling, Feature Encoding, Handling Imbalanced Dataset, Data Normalization dan Dimensionality Reduction. Berikut merupakan penjelasan dari setiap tahapan pada tahap *preprocessing*:

##### 1. Feature Scaling

Feature Scaling bertujuan untuk menyelaraskan skala fitur numerik agar algoritma machine learning dapat bekerja dengan lebih baik. Beberapa algoritma seperti regresi logistik, SVM, atau algoritma berbasis jarak (misalnya KNN) sangat sensitif terhadap skala data.

Pada implementasi ini, pertama-tama dilakukan imputasi pada *missing value*. Data numerik yang memiliki nilai kosong akan diisi (imputasi) menggunakan rata-rata (*mean*) nilai dari kolom tersebut. Data kemudian distandarisasi menggunakan *StandardScaler*, yang mengubah data menjadi distribusi dengan rata-rata 0 dan standar deviasi 1.



```
1 class FeatureScaling(BaseEstimator, TransformerMixin):
2     def __init__(self, numerical_features):
3         self.numerical_features = numerical_features
4         self.numerical_transformer = Pipeline(steps=[
5             ("imputer", SimpleImputer(strategy='mean')),
6             ("scaler", StandardScaler())
7         ])
8
9     def fit(self, X, y=None):
10        self.numerical_transformer.fit(X[self.numerical_features])
11        return self
12
13    def transform(self, X):
14        X_scaled = self.numerical_transformer.transform(X[self.numerical_features])
15        return X_scaled
```

Standarisasi sangat penting untuk algoritma yang sensitif terhadap skala data, seperti SVM dan KNN. Menyamakan skala data mencegah fitur dengan skala besar mendominasi perhitungan model.

## 2. Feature Encoding

Mengubah data kategorikal menjadi representasi numerik agar dapat digunakan dalam algoritma machine learning. Pada implementasi ini, dilakukan imputasi *missing value*. Pada data kategorikal yang kosong diisi menggunakan nilai yang paling sering muncul (*modus*). Selanjutnya, dengan menggunakan

One-Hot Encoding yaitu mengubah setiap kategori menjadi kolom biner (0 atau 1).

```
1 class FeatureEncoding(BaseEstimator, TransformerMixin):
2     def __init__(self, categorical_features):
3         self.categorical_features = categorical_features
4         self.categorical_transformer = Pipeline(steps=[
5             ("imputer", SimpleImputer(strategy='most_frequent')),
6             ("encoder", OneHotEncoder(handle_unknown='ignore'))
7         ])
8
9     def fit(self, X, y=None):
10        self.categorical_transformer.fit(X[self.categorical_features])
11        return self
12
13    def transform(self, X):
14        X_encoded = self.categorical_transformer.transform(X[self.categorical_features])
15        return X_encoded
```

One-Hot Encoding cocok untuk data kategorikal karena tidak mengasumsikan urutan antar kategori. Memastikan data kategorikal dapat diolah oleh algoritma machine learning. Imputasi menggunakan modus menjaga distribusi kategori.

### 3. Handling Imbalanced Dataset

Ketidakseimbangan data terjadi ketika satu kelas dalam dataset jauh lebih dominan dibandingkan kelas lain (contoh: 90% kelas positif, 10% kelas negatif). Tahapan ini bertujuan untuk menangani masalah tersebut sehingga model tidak bias terhadap kelas mayoritas. Pada implementasi ini, dilakukan *Under-Sampling* yaitu mengurangi jumlah sampel dari kelas mayoritas untuk menyamakan proporsi dengan kelas minoritas.

```

1  class HandleImbalance(BaseEstimator, TransformerMixin):
2      def __init__(self, strategy="undersample"):
3          self.strategy = strategy
4          if self.strategy == "undersample":
5              self.sampler = RandomUnderSampler(sampling_strategy='auto')
6          elif self.strategy == "oversample":
7              self.sampler = SMOTE(sampling_strategy='auto')
8          else:
9              raise ValueError("Unknown sampling strategy")
10
11     def fit(self, X, y):
12         self.sampler.fit_resample(X, y)
13         return self
14
15     def transform(self, X, y=None):
16         X_resampled, y_resampled = self.sampler.fit_resample(X, y)
17         return X_resampled, y_resampled

```

Under-Sampling mengurangi bias model terhadap kelas mayoritas. Dengan dataset yang lebih seimbang, model memiliki performa yang lebih baik pada kelas minoritas.

#### 4. Data Normalization

Normalisasi mengubah data sehingga nilainya berada pada rentang tertentu (biasanya antara 0 dan 1). Tahapan ini dilakukan untuk memastikan skala data seragam. Pada implementasi ini, semua data fitur distandarisasi atau dinormalisasi ke rentang tertentu. Teknik yang digunakan adalah *StandardScaler* atau *MinMaxScaler*.



```
1 class DataNormalization(BaseEstimator, TransformerMixin):
2     def __init__(self):
3         self.scaler = StandardScaler()
4
5     def fit(self, X, y=None):
6         self.scaler.fit(X)
7         return self
8
9     def transform(self, X):
10        X_normalized = self.scaler.transform(X)
11        return X_normalized
```

Normalisasi sangat penting untuk algoritma berbasis jarak seperti KNN atau clustering. Menjaga skala data seragam membantu model bekerja lebih optimal.

## 5. Dimensionality Reduction

Mengurangi jumlah fitur dalam dataset tanpa kehilangan informasi yang signifikan. Hal ini dilakukan untuk mengurangi kompleksitas model dan meningkatkan performa dan waktu komputasi. Implementasi ini menggunakan metode *Truncated SVD* atau *PCA (Principal Component Analysis)* untuk mencari dimensi baru yang tetap mempertahankan sebagian besar variasi dalam data. Dimensi baru dipilih berdasarkan jumlah komponen (*n\_components*) yang ditentukan.

```

1  class DimensionalityReduction(BaseEstimator, TransformerMixin):
2      def __init__(self, n_components=10, method="SVD"):
3          self.n_components = n_components
4          if method == "SVD":
5              self.reducer = TruncatedSVD(n_components=self.n_components)
6          elif method == "PCA":
7              self.reducer = PCA(n_components=self.n_components)
8          else:
9              raise ValueError("Unknown dimensionality reduction method")
10
11     def fit(self, X, y=None):
12         self.reducer.fit(X)
13         return self
14
15     def transform(self, X):
16         X_reduced = self.reducer.transform(X)
17         return X_reduced

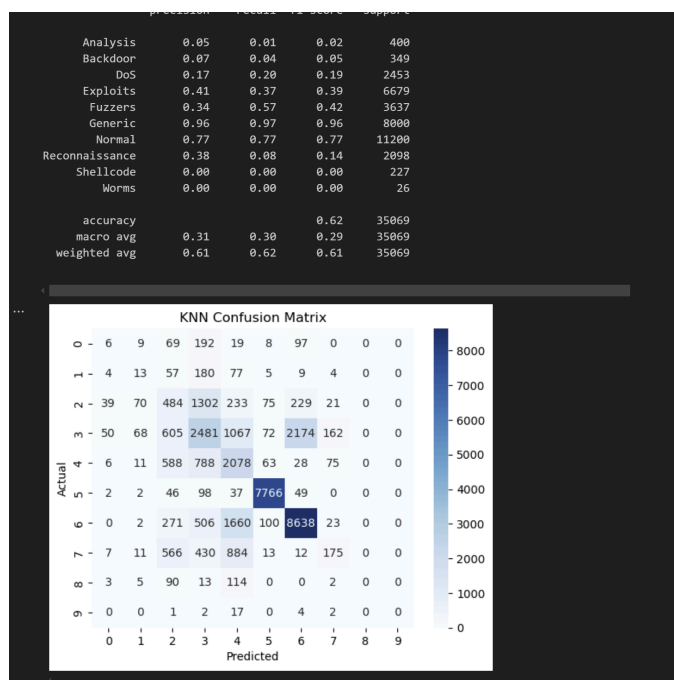
```

Mengurangi dimensi data mengatasi *curse of dimensionality* dan meningkatkan efisiensi komputasi. Teknik seperti PCA menjaga informasi penting dalam dataset sambil menghilangkan *noise* atau fitur redundan.

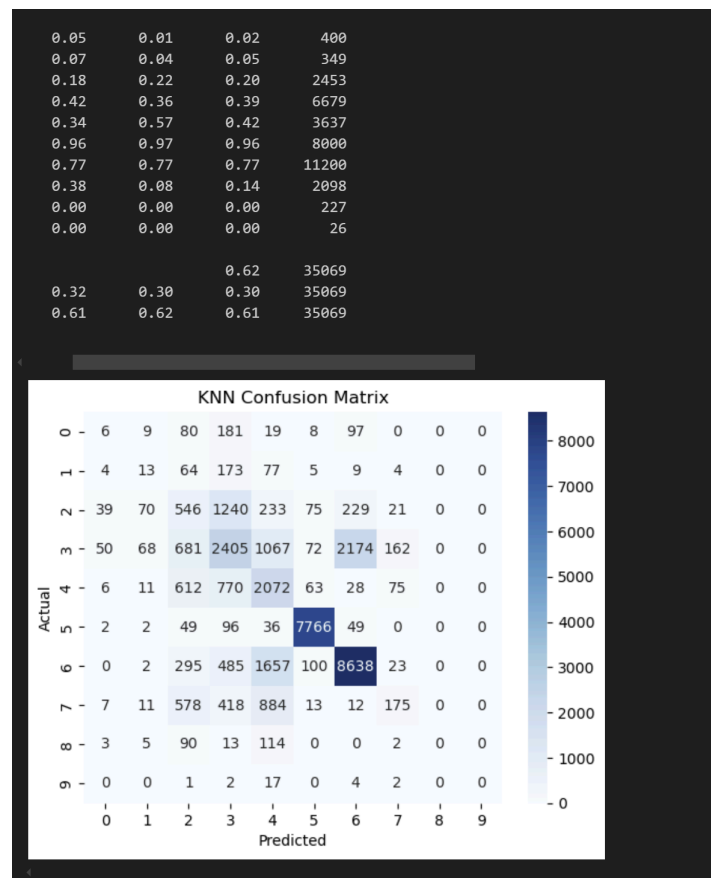
## V. Komparasi Hasil Prediksi antara Implementasi Manual dan Pustaka

### 1. KNN

Hasil *Confusion Matrix* menggunakan *scratch*:



Hasil *Confusion Matrix* menggunakan *scikit-learn*:



Berdasarkan gambar perbandingan keduanya, dapat dilihat bahwa implementasi baik dari *scratch* maupun dari pustaka mendapatkan hasil yang sama. Hanya terdapat perbedaan 0.01 di matriks macro avg. Hal ini mungkin disebabkan oleh perbedaan cara *handle* apabila ada jumlah KNN terdekat yang sama.

### 1. Akurasi Model

Baik pada implementasi *scratch* maupun pustaka *scikit-learn*, akurasi keseluruhan model adalah 62%. Ini menunjukkan bahwa implementasi dari *scratch* sudah sesuai dan menghasilkan hasil prediksi yang mirip dengan pustaka standar (*benchmark*).

### 2. Perbandingan Metrics (Precision, Recall, F1-score)

Nilai precision, recall, dan f1-score untuk tiap kelas menunjukkan kesamaan pada kedua implementasi. Tidak ada perbedaan signifikan dalam hasil perhitungan. Kelas Generic (kelas 6) memiliki performa tertinggi dengan f1-score mendekati 0.96 dan nilai precision serta recall yang tinggi. Hal ini menandakan bahwa model sangat baik dalam mengenali kelas ini. Sebaliknya, beberapa kelas seperti:

a. Shellcode (kelas 8), dan

b. Worms (kelas 9)

memiliki nilai  $f1\text{-score} = 0.00$ , yang berarti model kesulitan dalam mengenali data untuk kelas-kelas ini.

### 3. Distribusi Prediksi (Confusion Matrix)

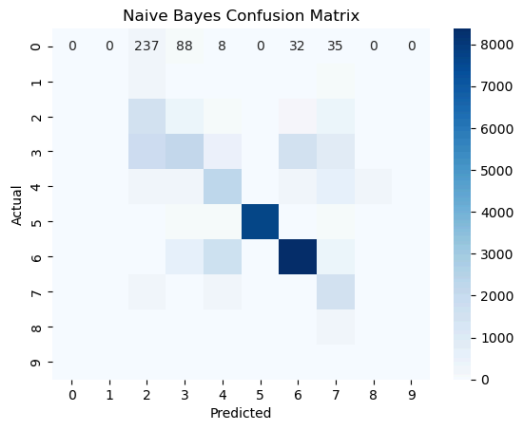
a. Distribusi prediksi pada Confusion Matrix dari kedua implementasi menunjukkan pola yang sama:

b. Sebagian besar kesalahan prediksi terjadi pada kelas dengan jumlah data kecil (contoh: kelas 0, 1, dan 9).

Kelas Generic dan Normal memiliki jumlah prediksi benar yang dominan, terlihat dari diagonal yang jelas pada kedua matriks.

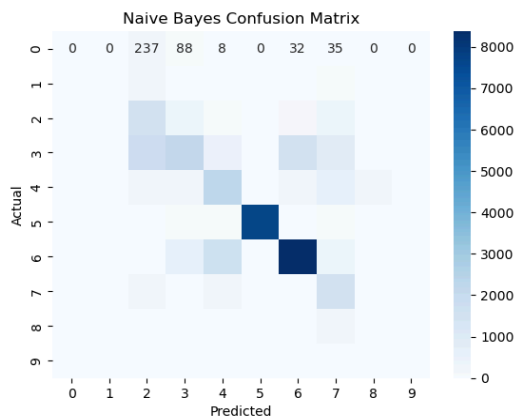
## 2. Naive Bayes

Hasil *Confusion Matrix* menggunakan *scratch*:



Naive Bayes Classification Report:				
	precision	recall	f1-score	support
Analysis	0.00	0.00	0.00	400
Backdoor	0.00	0.00	0.00	349
DoS	0.34	0.59	0.43	2453
Exploits	0.57	0.31	0.40	6679
Fuzzers	0.47	0.62	0.53	3637
Generic	1.00	0.96	0.98	8000
Normal	0.82	0.75	0.78	11200
Reconnaissance	0.38	0.76	0.51	2098
Shellcode	0.17	0.26	0.21	227
Worms	0.00	0.00	0.00	26
accuracy			0.67	35069
macro avg	0.38	0.42	0.38	35069
weighted avg	0.69	0.67	0.67	35069

Hasil *Confusion Matrix* menggunakan *scikit-learn*:



Naive Bayes Classification Report:				
	precision	recall	f1-score	support
Analysis	0.00	0.00	0.00	400
Backdoor	0.00	0.00	0.00	349
DoS	0.34	0.59	0.43	2453
Exploits	0.57	0.31	0.40	6679
Fuzzers	0.47	0.62	0.53	3637
Generic	1.00	0.96	0.98	8000
Normal	0.82	0.75	0.78	11200
Reconnaissance	0.38	0.76	0.51	2098
Shellcode	0.17	0.26	0.21	227
Worms	0.00	0.00	0.00	26
accuracy			0.67	35069
macro avg	0.38	0.42	0.38	35069
weighted avg	0.69	0.67	0.67	35069

Berdasarkan gambar perbandingan hasil *Confusion Matrix* dan *Classification Report* untuk model Naive Bayes yang diimplementasikan secara *scratch* dan menggunakan pustaka *scikit-learn*, dapat diambil beberapa *insight* sebagai berikut:

## 1. Akurasi Model

Baik pada implementasi *scratch* maupun pustaka *scikit-learn*, akurasi keseluruhan model adalah 67%. Ini menunjukkan bahwa implementasi dari *scratch* sudah sesuai dan menghasilkan hasil prediksi yang mirip dengan pustaka standar (*benchmark*).

## 2. Perbandingan Metrics (Precision, Recall, F1-score)

Nilai precision, recall, dan f1-score untuk tiap kelas menunjukkan kesamaan pada kedua implementasi. Tidak ada perbedaan signifikan dalam hasil perhitungan. Kelas Generic

(kelas 6) memiliki performa tertinggi dengan f1-score mendekati 0.98 dan nilai precision serta recall yang tinggi. Hal ini menandakan bahwa model sangat baik dalam mengenali kelas ini. Sebaliknya, beberapa kelas seperti:

- Analysis (kelas 0),
- Backdoor (kelas 1), dan
- Worms (kelas 9)

memiliki nilai f1-score = 0.00, yang berarti model kesulitan dalam mengenali data untuk kelas-kelas ini.

### 3. Distribusi Prediksi (Confusion Matrix)

Distribusi prediksi pada *Confusion Matrix* dari kedua implementasi menunjukkan pola yang sama:

- Sebagian besar kesalahan prediksi terjadi pada kelas dengan jumlah data kecil (contoh: kelas 0, 1, dan 9).
- Kelas Generic dan Normal memiliki jumlah prediksi benar yang dominan, terlihat dari diagonal yang jelas pada kedua matriks.

Hal ini menunjukkan bahwa kedua implementasi konsisten dalam menangkap tren yang sama, baik dalam mengenali kelas mayoritas maupun mengalami kesulitan dengan kelas minoritas.

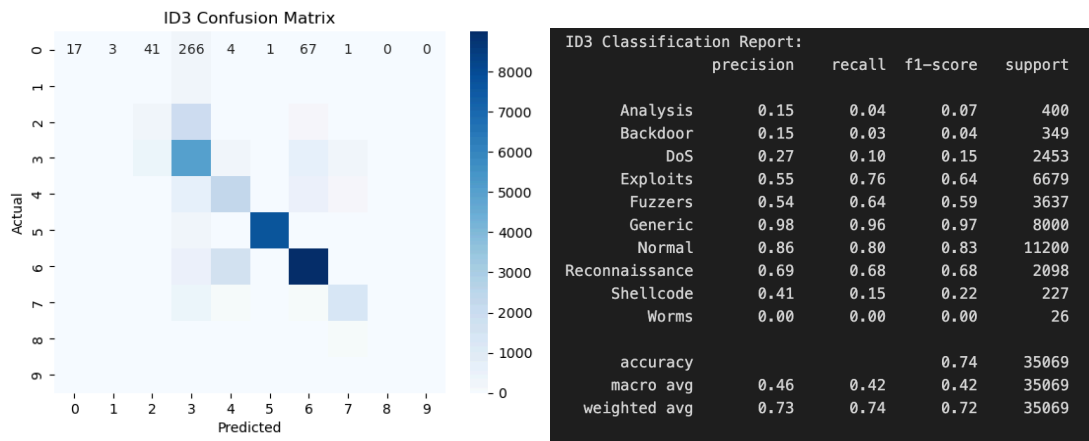
Implementasi model Naive Bayes secara scratch berhasil memvalidasi performa pustaka scikit-learn karena memberikan hasil yang identik. Tantangan utama dari model ini adalah:

- Ketidakseimbangan kelas: Kelas dengan jumlah data sedikit (seperti kelas Analysis, Backdoor, dan Worms) memiliki performa rendah.
- Kebutuhan untuk meningkatkan recall dan precision pada kelas minoritas, yang mungkin bisa diperbaiki dengan metode resampling atau algoritma yang lebih kompleks.

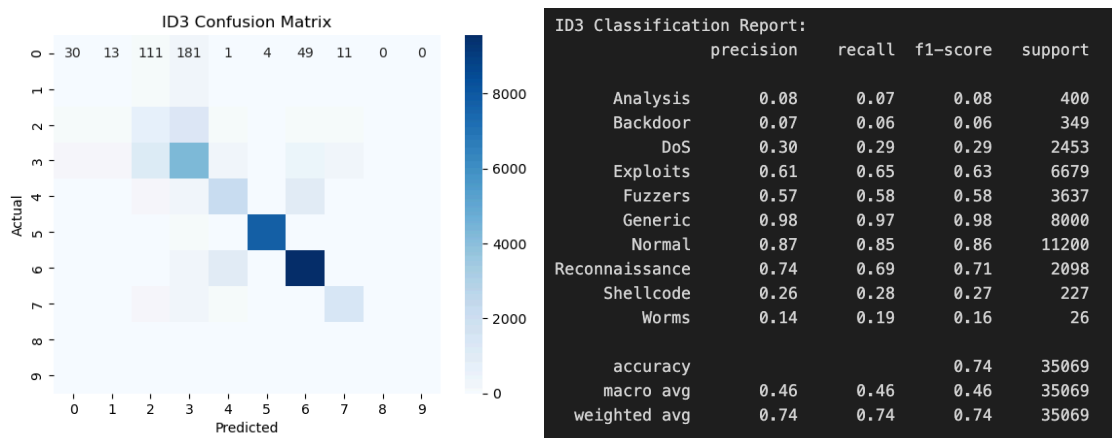
Namun, untuk kelas mayoritas seperti Generic dan Normal, performa model sudah sangat baik.

### 3. ID3

Hasil *Confusion Matrix* menggunakan *scratch*:



Hasil *Confusion Matrix* menggunakan *scikit-learn*:



Berdasarkan hasil perbandingan antara implementasi ID3 secara *scratch* dengan pustaka *scikit-learn*, berikut adalah beberapa *insight* yang dapat diperoleh:

#### 1. Akurasi Model

Akurasi kedua implementasi sama-sama mencapai 0.74 (74%), menunjukkan bahwa implementasi *scratch* sudah cukup akurat dan mendekati hasil dari pustaka *scikit-learn*. Ini menunjukkan bahwa logika pohon keputusan ID3 yang diimplementasikan *scratch* sudah sesuai dan valid.

## 2. Metrics (Precision, Recall, F1-score)

Hasil scikit-learn memiliki performa yang sedikit lebih baik dalam beberapa metrik dibandingkan implementasi *scratch*. Perbedaannya kecil, namun terlihat pada beberapa kelas:

- Precision pada kelas seperti Analysis (kelas 0) dan Backdoor (kelas 1) lebih rendah di implementasi *scratch* dibandingkan scikit-learn.
- Nilai F1-score untuk kelas Shellcode (kelas 8) dan Reconnaissance (kelas 7) juga sedikit lebih rendah pada implementasi *scratch*.

Kelas Generic (kelas 6) dan Normal (kelas 7) masih menjadi kelas dengan performa tertinggi pada kedua implementasi, dengan f1-score > 0.80.

## 3. Confusion Matrix

Polanya serupa pada kedua *Confusion Matrix*, di mana:

- Kelas mayoritas (Generic dan Normal) memiliki prediksi benar yang dominan, terlihat jelas di diagonal utama.
- Kelas minoritas seperti Analysis (kelas 0), Backdoor (kelas 1), dan Worms (kelas 9) sering salah diklasifikasikan sebagai kelas lain.

Namun, implementasi scikit-learn cenderung lebih stabil dalam mendistribusikan prediksi, sedangkan implementasi *scratch* masih menghasilkan sedikit *bias* terhadap kelas mayoritas.

## 4. Makna Perbedaan Hasil

Perbedaan kecil antara kedua implementasi kemungkinan disebabkan oleh:

- Optimasi algoritma yang lebih matang di pustaka scikit-learn.
- Potensi floating-point error atau sedikit perbedaan dalam logika *splitting* pohon keputusan pada implementasi *scratch*.

Meskipun demikian, hasil akhirnya konsisten dan mendukung performa model yang sama.

Implementasi ID3 secara *scratch* sudah berhasil memvalidasi pustaka scikit-learn dengan hasil yang mendekati serupa. Tantangan utama model ini adalah menangani



ketidakseimbangan kelas karena kelas minoritas masih sulit diprediksi. Rekomendasi perbaikan yang bisa dilakukan:

- Pertimbangkan menggunakan teknik pruning untuk menghindari *overfitting*.
- Lakukan resampling atau penanganan kelas minoritas dengan teknik seperti *SMOTE*.
- Eksplorasi alternatif seperti algoritma pohon keputusan C4.5 atau Random Forest untuk hasil lebih robust.

## VI. Kontribusi Anggota

Berikut adalah tabel rincian mengenai pendistribusian kerja oleh anggota kelompok:

No	NIM Anggota	Nama Anggota	Deskripsi
1	13522048	Angelica Kierra Ninta Gurning	Membuat Model KNN
2	13522058	Immanuel Sebastian Girsang	Membuat Model ID3
3	13522070	Marzuli Suhada M	Melakukan Data Cleaning & Preprocessing
4	13522108	Muhammad Neo Cicero Koda	Membuat Model Naive Bayes

## VII. Referensi

- [The UNSW-NB15 Dataset | UNSW Research](#)
- [UNSW-NB15: a comprehensive data set for network intrusion detection systems \(UNSW-NB15 network data set\) | IEEE Conference Publication | IEEE Xplore](#)
- [K-Nearest Neighbor\(KNN\) Algorithm - GeeksforGeeks](#)
- [What Are Naïve Bayes Classifiers? | IBM](#)
- [Decision Trees: ID3 Algorithm Explained | Towards Data Science](#)