

LAPORAN TUGAS BESAR 2
IF3170 INTELIGENSI ARTIFISIAL
Implementasi Algoritma Pembelajaran Mesin



Disusun Oleh:

13522069	Nabila Shikoofa Muida
13522096	Novelya Putri Ramadhani
13522102	Hayya Zuhailii Kinasih
13522104	Diana Tri Handayani

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

2024

DAFTAR ISI

BAB I

IMPLEMENTASI ALGORITMA.....	3
1.1. Implementasi KNN.....	3
1.1.1. Inisialisasi Model.....	3
1.1.2. Training Model.....	3
1.1.3. Menghitung Jarak.....	3
1.1.4. Proses Prediksi.....	4
1.1.5. Parameter Model.....	4
1.2. Implementasi Gaussian Naive-Bayes.....	5
1.2.1. Inisialisasi Model.....	5
1.2.2. Training Model.....	5
1.2.3. Menghitung Likelihood Gaussian.....	6
1.2.4. Menghitung Posterior.....	6
1.2.5. Proses Prediksi.....	6
1.2.6. Parameter Model.....	6
1.3. Implementasi ID3.....	7
1.3.1 Inisialisasi ID3.....	7
1.3.2 Perhitungan Entropy.....	7
1.3.3 Perhitungan Information Gain.....	7
1.3.4 Penentuan Plurality Value.....	8
1.3.5 Pemilihan Atribut Sebagai Node.....	8
1.3.6 Discretization.....	8
1.3.7 Training Model.....	9
1.3.8 Proses Prediksi.....	11

BAB II

DATA CLEANING AND PREPROCESSING.....	12
2.1. Data Cleaning.....	12
2.1.1. Handling Missing Values.....	12
2.1.2. Dealing with Outliers.....	13
2.1.3. Remove Duplicates.....	13
2.1.4. Feature Engineering.....	13
2.2. Data Preprocessing.....	14
2.2.1. Feature Scaling.....	14
2.2.2. Feature Encoding.....	15
2.2.3. Handling Imbalanced Classes.....	16

2.2.4. Data Normalization.....	16
2.2.5. Dimensionality Reduction.....	17
2.3. Pipelining.....	17
BAB III	
PERBANDINGAN HASIL PREDIKSI.....	19
3.1. Hasil KNN.....	19
3.2. Hasil Naive-Bayes.....	21
3.3. Hasil ID3.....	23
BAB IV	
KESIMPULAN.....	25
LAMPIRAN.....	26
REFERENSI.....	27

BAB I

IMPLEMENTASI ALGORITMA

1.1. Implementasi KNN

K-Nearest Neighbors (KNN) adalah algoritma supervised learning yang menentukan kelas atau nilai sebuah data berdasarkan sejumlah tetangga terdekat (k-nearest neighbors) di data latih.

1.1.1. Inisialisasi Model

Kelas KNN dibuat dengan dua parameter utama:

- **k**: jumlah tetangga terdekat.
- **distance_metric**: jenis metrik jarak yang digunakan (euclidean, manhattan, atau minkowski) yang menentukan bagaimana jarak antara data uji dan data latih dihitung.

```
def __init__(self, k=3, distance_metric="euclidean"):
    self.k = k
    self.distance_metric = distance_metric
```

1.1.2. Training Model

Metode `fit` digunakan untuk menyimpan data latih (`X_train`) dan labelnya (`y_train`). Data ini nantinya digunakan dalam proses prediksi.

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

1.1.3. Menghitung Jarak

Algoritma KNN menggunakan beberapa metrik untuk menghitung jarak antara dua data:

1. Euclidean Distance

```
def _euclidean_distance(self, x1, x2):
    return np.sqrt(np.sum((x1 - x2) ** 2))
```

2. Manhattan Distance

```
def _manhattan_distance(self, x1, x2):
    return np.sum(np.abs(x1 - x2))
```

3. Minkowski Distance

```
def _minkowski_distance(self, x1, x2, p=3):  
    return np.sum(np.abs(x1 - x2) ** p) ** (1/p)
```

Metode `_compute_distance` digunakan untuk memilih jenis metrik jarak yang sesuai dengan parameter.

```
def _compute_distance(self, x1, x2):  
    if self.distance_metric == "euclidean":  
        return self._euclidean_distance(x1, x2)  
    elif self.distance_metric == "manhattan":  
        return self._manhattan_distance(x1, x2)  
    elif self.distance_metric == "minkowski":  
        return self._minkowski_distance(x1, x2)  
    else:  
        raise ValueError("Invalid distance metric:  
{0}".format(self.distance_metric))
```

1.1.4. Proses Prediksi

Metode `predict` digunakan untuk memprediksi label data uji. Langkah-langkahnya adalah:

- Menghitung jarak antara data uji dengan seluruh data latih.
- Menentukan k tetangga terdekat berdasarkan jarak.
- Memilih kelas yang paling sering muncul di antara k tetangga tersebut.

```
def predict(self, X):  
    prediction = []  
    for _, x in X.iterrows():  
        distances = [self._compute_distance(x.values, x_train) for  
x_train in self.X_train]  
        k_indices = np.argsort(distances)[:self.k]  
        k_nearest_labels = [self.y_train[i] for i in k_indices]  
        most_common = Counter(k_nearest_labels).most_common(1)  
        prediction.append(most_common[0][0])  
    return np.array(prediction)
```

1.1.5. Parameter Model

Model ini menyediakan metode `get_params` untuk mendapatkan parameter `k` dan `distance_metric` yang berguna untuk tuning atau evaluasi model.

```
def get_params(self, deep=True):  
    return {"k": self.k, "distance_metric": self.distance_metric}
```

1.2. Implementasi Gaussian Naive-Bayes

Gaussian Naive-Bayes (GNB) adalah algoritma berbasis probabilitas yang mengasumsikan bahwa setiap fitur mengikuti distribusi Gaussian (normal distribution), dengan asumsi naive bahwa semua fitur bersifat independen.

1.2.1. Inisialisasi Model

Kelas `GaussianNaiveBayes` dibuat dengan dua parameter utama:

- `classes`: menyimpan daftar kelas unik dari data.
- `mean`: rata-rata (mean) fitur untuk setiap kelas.
- `var`: variansi (variance) fitur untuk setiap kelas.
- `priors`: probabilitas awal (prior probability) untuk setiap kelas.

```
def __init__(self):
    self.classes = None
    self.mean = {}
    self.var = {}
    self.priors = {}
```

1.2.2. Training Model

Metode `fit` digunakan untuk menghitung parameter statistik dari data latih, yaitu:

- Rata-rata
- Variansi
- Prior (probabilitas suatu kelas berdasarkan proporsi data di kelas tersebut).

Data dipecah berdasarkan kelas untuk menghitung nilai statistiknya. Hasilnya disimpan dalam dictionary `mean`, `var`, dan `priors`.

```
def fit(self, X, y):
    self.classes = np.unique(y)
    for cls in self.classes:
        X_cls = X[y == cls]
        self.mean[cls] = np.mean(X_cls, axis=0)
        self.var[cls] = np.var(X_cls, axis=0)
        self.priors[cls] = X_cls.shape[0] / X.shape[0]
```

1.2.3. Menghitung Likelihood Gaussian

Untuk setiap kelas, metode `_gaussian_likelihood` menghitung probabilitas (likelihood) data berdasarkan distribusi Gaussian.

```
def _gaussian_likelihood(self, class_idx, x):  
    mean = self.mean[class_idx]  
    var = self.var[class_idx] + 1e-9  
    numerator = np.exp(-(x - mean)**2 / (2 * var))  
    denominator = np.sqrt(2 * np.pi * var)  
    return numerator / denominator
```

1.2.4. Menghitung Posterior

Metode `_posterior` digunakan untuk menghitung probabilitas posterior menggunakan Teorema Bayes. Logaritma digunakan untuk mencegah underflow numerik dan posterior dihitung sebagai jumlah log prior dan log likelihood.

```
def _posterior(self, x):  
    posteriors = []  
    for cls in self.classes:  
        prior = np.log(self.priors[cls])  
        likelihood = np.sum(np.log(self._gaussian_likelihood(cls,  
x)))  
        posterior = prior + likelihood  
        posteriors.append(posterior)  
    return self.classes[np.argmax(posteriors)]
```

1.2.5. Proses Prediksi

Metode `predict` digunakan untuk memprediksi label data uji berdasarkan posterior probabilitas tertinggi untuk setiap data. Proses prediksi dilakukan dengan menghitung posterior untuk setiap kelas, kemudian memilih kelas dengan probabilitas tertinggi.

```
def predict(self, X):  
    y_pred = [self._posterior(x) for x in X.values]  
    return np.array(y_pred)
```

1.2.6. Parameter Model

Metode `get_params` disediakan untuk memudahkan akses terhadap parameter model, meskipun dalam implementasi ini tidak ada parameter eksternal yang perlu disesuaikan.

```
def get_params(self, deep=True):  
    return {}
```

1.3. Implementasi ID3

ID3 adalah algoritma supervised learning yang membentuk sebuah *decision tree* berdasarkan data yang disediakan dan membuat prediksi berdasarkan *decision tree* tersebut.

1.3.1 Inisialisasi ID3

Algoritma ini menerima parameter *random_seed* untuk memastikan hasil *training* yang konsisten jika dijalankan berulang kali. Kelas ID3 menyimpan variabel berupa *start* (yang merupakan *root* dari pohon), *colVal* (daftar value unik yang dimiliki suatu kolom), dan *thresholds* (*threshold value* untuk kolom non kategorikal).

```
def __init__(self, random_seed: int = 0):  
    self.start : Node = Node('', False)  
    self.colVal: dict[str, list[str]] = {}  
    self.thresholds : dict[str, float] = {}  
    self.random_seed = random_seed
```

1.3.2 Perhitungan Entropy

Entropy adalah tingkat ketidakpastian dalam informasi yang tersedia. Perhitungannya mengikuti persamaan berikut:

$$H(X) := - \sum_{x \in \mathcal{X}} p(x) \log p(x),$$

Implementasinya memanfaatkan *library* numpy untuk membuat perhitungan lebih efisien.

```
def __entropy(self, s: pd.Series) -> float:  
    vcount = s.value_counts(normalize=True, sort=False)  
    return -(vcount*np.log2(vcount)).sum()
```

1.3.3 Perhitungan Information Gain

Information gain adalah besarnya informasi yang diperoleh tentang suatu variabel dengan mengamati nilai suatu variabel lain. Nilai itu dapat dihitung dengan persamaan berikut, dengan $H(T|a)$ adalah entropy variabel T , *given* variabel a :

$$IG(T, a) = H(T) - H(T|a),$$

$$H(T|a) = \sum_{v \in \text{vals}(a)} \frac{|S_a(v)|}{|T|} \cdot H(S_a(v))$$

Pada pengimplementasian, fungsinya hanya menghitung dan membandingkan nilai $H(T|a)$ untuk menghemat komputasi.

```
def __information_gain(self, x: pd.Series, y: pd.Series) -> float:
    vcount = x.value_counts(normalize=True, sort=False)
    ent = [self.__entropy(y[x[x == val].index]) for val in
vcount.index]
    return (vcount*ent).sum()
```

1.3.4 Penentuan Plurality Value

Plurality Value adalah nilai yang paling sering muncul dalam suatu *series*. Pada fungsi ini, *random_seed* digunakan ketika ditemukan bahwa *series* memiliki lebih dari satu modus.

```
def __plurality_value(self, y: pd.Series):
    modes = y.mode()
    return modes[self.random_seed % len(modes)]
```

1.3.5 Pemilihan Atribut Sebagai Node

Atribut yang dipilih untuk menjadi node selanjutnya dalam *decision tree* adalah atribut yang memberikan hasil information gain paling rendah.

```
def __best_attr(self, x: pd.DataFrame, y: pd.Series) -> str:
    return x.columns[np.argmin([self.__information_gain(x[col], y)
for col in x.columns])]
```

1.3.6 Discretization

Discretization adalah proses pengelompokkan kolom bertipe kontinu agar menjadi diskrit. Hal ini dilakukan karena atribut kontinu tidak cocok untuk digunakan dalam *decision tree*. Dalam materi perkuliahan, metode untuk diskritisasi adalah dengan menyortir kolom agar

terurut ke atas, kemudian mengambil nilai tengah antara dua baris yang nilai targetnya berbeda. Dari nilai-nilai itu, dipilih nilai yang memberikan information gain paling baik sebagai *threshold*.

Pada program ini, proses pengambilan nilai tengah berbeda. Program pertama mengelompokkan nilai-nilai suatu kolom berdasarkan nilai kolom targetnya, kemudian diambil nilai rata-rata per kelompoknya. Nilai-nilai tersebut disortir menaik, kemudian dihitung nilai tengah antara dua nilai yang bersebelahan. Terakhir, dipilih nilai yang memberikan information gain paling baik sebagai threshold.

Setelah *testing*, ditemukan bahwa hasil dari metode kedua cukup setara dengan metode pertama, namun metode kedua memakan waktu lebih sedikit.

```
def __discretize(self, x: pd.Series, y: pd.Series):
    d = pd.concat([x, y], axis=1)
    d = d.sort_values(by=[x.name], ascending=True)
    avg_of_y = [d[d[y.name] == val][x.name].mean() for val in
y.unique()]
    avg_between_y = [(avg_of_y[i]+avg_of_y[i+1])/2 for i in range
(len(avg_of_y)-1)]

    best_gain = 9999
    best_split = 0
    for val in avg_between_y:
        gain = self.__information_gain(x >= val, y)
        if gain < best_gain:
            best_gain = gain
            best_split = val

    return best_split
```

1.3.7 Training Model

Proses training dimulai dengan mencatat nilai-nilai unik setiap kolom, sembari melakukan diskritisasi pada kolom yang bersifat kontinu. Kemudian, dibangun *decison tree*.

```
def fit(self, X: pd.DataFrame, y: pd.Series):
    x_train = X.copy()
    for c in x_train.columns:
        if (x_train[c].dtype != 'object') and
```

```
(x_train[c].nunique() > 2):
    split = self.__discretize(x_train[c], y)
    self.thresholds.update({c: split})
    x_train[c] = x_train[c] >= split

    self.colVal.update({c: x_train[c].unique()})
    self.start = self.__decision_tree_learning(x_train, y, None)
```

Pembangunan pohon dilakukan secara rekursif. Jika sudah tidak ada baris dalam suatu *subset*, maka dikembalikan plurality value dari *superset* sebagai *leaf* dalam *tree*. Jika nilai semua target dalam suatu *subset* adalah sama, maka dikembalikan nilai tersebut sebagai *leaf* dalam *tree*. Jika suatu *subset* sudah tidak memiliki kolom lagi, maka dikembalikan plurality value dari *subset* itu sebagai *leaf* dalam *tree*.

Jika ketiga kondisi tersebut tidak dipenuhi, maka dipilih kolom yang memberikan information gain terbaik untuk menjadi *node* dalam *tree*. Untuk setiap nilai unik dalam kolom tersebut, dibuat *branch* yang menuju ke *node* selanjutnya. *Node* selanjutnya dilakukan dengan memanggil fungsi ini kembali secara rekursif dengan *subset* yang sesuai.

```
def __decision_tree_learning(self, x: pd.DataFrame, y: pd.Series,
                             yParent: pd.Series | None):
    if len(y) == 0:
        return Node(self.__plurality_value(yParent), True)
    elif (y.nunique() == 1):
        return Node(y.head(1).values[0], True)
    elif x.empty:
        return Node(self.__plurality_value(y), True)
    else:
        col = self.__best_attr(x, y)
        tree = Node(col, False)
        for val in self.colVal[col]:
            exs = x[x[col] == val].drop(columns=[col])
            subtree = self.__decision_tree_learning(exs,
            y[exs.index], y)
            tree.children[val] = subtree
        return tree
```

1.3.8 Proses Prediksi

Prediksi dimulai dengan melakukan diskritisasi pada kolom dengan atribut kontinu. Kemudian, dilakukan iterasi terhadap semua baris dalam tabel yang tersedia. Prediksi dilakukan dengan menelusuri *decision tree* berdasarkan nilai dalam baris hingga mencapai *leaf node*. Nilai dalam *leaf node* itu adalah prediksi yang dibuat untuk baris itu.

```
def predict(self, X: pd.DataFrame):
    xc = X.copy()
    for c in self.thresholds:
        xc[c] = xc[c] >= self.thresholds[c]
    result = []
    for row in xc.iterrows():
        node = self.start
        while not node.isResult:
            node = node.children.get(row[1][node.name])
        result.append(node.name)
    return result
```

BAB II

DATA CLEANING AND PREPROCESSING

2.1. Data Cleaning

Data cleaning bertujuan untuk memastikan integritas dataset dengan mengidentifikasi dan menangani masalah seperti nilai yang hilang (missing values), data duplikat, dan inkonsistensi data. Fokus utama pada tahap ini adalah:

2.1.1. Handling Missing Values

Kami menggunakan metode imputasi dengan menggunakan `SimpleImputer` dari library `scikit-learn` untuk menangani missing values.

- Untuk fitur numerik, nilai yang hilang diisi menggunakan rata-rata (*mean*) dari kolom terkait. Mean imputation mempertahankan distribusi data asli dan cocok dipakai karena data yang hilang bersifat acak (*missing at random*).
- Untuk fitur kategorikal, nilai yang hilang diisi dengan kategori yang paling sering muncul (*mode*). Mode imputation menjaga pola distribusi kategori dalam dataset tanpa mengganggu hubungan antar fitur, sehingga lebih sesuai untuk fitur kategorikal seperti `state` dan `service`, yang memiliki nilai dengan makna logis tertentu.

```
# for numerical features
num_imputer = SimpleImputer(strategy='mean')
train_set[non_categorical_feat] =
num_imputer.fit_transform(train_set[non_categorical_feat])
val_set[non_categorical_feat] =
num_imputer.transform(val_set[non_categorical_feat])

# for categorical features
cat_imputer = SimpleImputer(strategy='most_frequent')
train_set[categorical_feat] =
cat_imputer.fit_transform(train_set[categorical_feat])
val_set[categorical_feat] =
cat_imputer.transform(val_set[categorical_feat])
```

2.1.2. Dealing with Outliers

Untuk menangani outliers, kami menggunakan metode Interquartile Range (IQR) Capping karena metode ini efektif dalam mengurangi pengaruh nilai ekstrem tanpa menghilangkan data secara langsung.. IQR adalah rentang antara kuartil pertama (Q1) dan kuartil ketiga (Q3) dalam distribusi data. Nilai di luar rentang yang ditentukan akan dianggap sebagai outlier. Pendekatan ini cocok untuk dataset ini, yang mengandung banyak fitur numerik dengan distribusi skewed, di mana outliers dapat memengaruhi rata-rata dan kinerja model. Dengan menggunakan IQR Capping, distribusi data tetap terjaga, dan model machine learning dapat mempelajari pola tanpa terganggu oleh nilai ekstrem.

```
def iqr_capping(data, feature):  
    Q1 = data[feature].quantile(0.25)  
    Q3 = data[feature].quantile(0.75)  
    IQR = Q3 - Q1  
    lower_limit = Q1 - 1.5 * IQR  
    upper_limit = Q3 + 1.5 * IQR  
    return np.clip(data[feature], lower_limit, upper_limit)
```

2.1.3. Remove Duplicates

Penanganan nilai duplikat sangat penting karena dapat mengganggu integritas data dan memengaruhi hasil analisis. Keberadaan data duplikat dapat menyebabkan bias pada model machine learning, meningkatkan risiko overfitting, dan mengurangi kemampuan model untuk melakukan generalisasi terhadap data baru. Selain itu, data duplikat memperbesar ukuran dataset, sehingga meningkatkan biaya komputasi dan waktu pemrosesan. Oleh karena itu, nilai duplikat perlu dihapus untuk menjaga kualitas data dan agar analisis akurat, efisien, dan konsisten.

```
train_set = train_set.drop_duplicates()  
val_set = val_set.drop_duplicates()
```

2.1.4. Feature Engineering

Pada bagian ini, kami menerapkan Feature Selection untuk menghapus fitur yang tidak relevan atau redundan untuk menyederhanakan dataset dan mencegah risiko overfitting. Kami melakukannya dengan menganalisis fitur-fitur yang berkorelasi tinggi yang berpotensi memberikan informasi berlebih dan menyebabkan multikolinearitas, sehingga perlu dihapus

dari dataset. Penghapusan fitur ini membantu mengurangi dimensi dataset, mempercepat proses training, dan meningkatkan efisiensi model.

```
for c in corr.columns:
    corr_c = corr[c]
    corr_c = corr_c[((corr_c > 0.75) & (corr_c < 1)) | ((corr_c <
-0.75) & (corr_c > -1))].dropna()
    if (not corr_c.empty):
        print(corr_c)

dropped_cols = ['spkts', 'sloss', 'dpkts', 'dloss', 'dwin',
'stcpb', 'dtcpb', 'synack', 'ackdat', 'ct_srv_dst', 'ct_dst_ltm',
'ct_src_ltm', 'ct_src_dport_ltm', 'ct_dst_sport_ltm',
'ct_dst_src_ltm']
train_set = train_set.drop(columns=dropped_cols)
val_set = val_set.drop(columns=dropped_cols)
```

Selain itu, kami juga menemukan nilai tidak valid pada fitur `state` dan `service` yang kemudian kami ganti menjadi `'-'`. Penyesuaian ini memastikan semua nilai kategori konsisten dan dapat diproses oleh model machine learning tanpa menyebabkan error atau hasil yang tidak akurat.

```
states = ["ACC", "CLO", "CON", "ECO", "ECR", "FIN", "INT", "MAS",
"PAR", "REQ", "RST", "TST", "TXD", "URH", "URN", "-"]
services = ["http", "ftp", "smtp", "ssh", "dns", "ftp-data", "irc",
"-"]

train_set.loc[~train_set.state.isin(states), 'state'] = '-'
train_set.loc[~train_set.service.isin(services), 'service'] = '-'
val_set.loc[~val_set.state.isin(states), 'state'] = '-'
val_set.loc[~val_set.service.isin(services), 'service'] = '-'
```

2.2. Data Preprocessing

Data Preprocessing adalah proses yang bertujuan untuk membuat data lebih cocok untuk *training*.

2.2.1. Feature Scaling

Feature Scaling adalah tahapan menstandarisasi jangkauan data suatu atribut. Hal itu dilakukan agar semua atribut seimbang dan dapat bekerja lebih baik dalam proses *training*.

Algoritma *scaling* yang digunakan pada program ini adalah standardisasi. Algoritma tersebut melakukan *scaling* berdasarkan *z-score*, yang pada akhirnya akan menghasilkan data dengan rata-rata 0 dan berdistribusi normal. Alasannya adalah karena beberapa model *machine learning* bekerja lebih baik dengan data yang berdistribusi normal. *Outlier* pada data juga sudah diproses pada tahap sebelumnya, sehingga hal itu tidak akan mempengaruhi proses *scaling*.

```
class FeatureScaler(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.scaler = StandardScaler()

    def fit(self, X, y=None):
        self.scaler.fit(X)
        return self

    def transform(self, X):
        return pd.DataFrame(self.scaler.transform(X),
                             columns=X.columns)
```

2.2.2. Feature Encoding

Feature Encoding adalah proses mentransformasi kolom kategorik menjadi numerik, karena *machine learning model* pada umumnya hanya menerima kolom bertipe numerik. Metode *encoding* yang dipilih untuk program ini adalah Label Encoding yang akan memberikan label numerik pada setiap kategori unik dalam atribut. Metode tersebut dipilih untuk menjaga *dimensionality* dataset serendah mungkin.

```
class FeatureEncoder(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.encoders = {}

    def fit(self, X, y=None):
        for col in X.select_dtypes(include=['object']).columns:
            le = LabelEncoder()
            self.encoders[col] = le.fit(X[col])
        return self

    def transform(self, X):
        X_encoded = X.copy()
        for col, encoder in self.encoders.items():
```



```
X_encoded[col] = encoder.transform(X[col])  
return X_encoded
```

2.2.3. Handling Imbalanced Classes

Imbalanced Classes adalah kasus ketika suatu kelas target memiliki representasi yang jauh lebih besar daripada suatu kelas lainnya dalam dataset. Hal itu dapat menyebabkan *bias* dalam beberapa algoritma *machine learning*. Maka, salah satu metode untuk menangani hal itu adalah dengan menambahkan populasi dataset agar semua kelas terepresentasi secara seimbang.

```
class HandleImbalance(BaseEstimator, TransformerMixin):  
    def __init__(self):  
        self.smote = SMOTE(random_state=42)  
  
    def fit(self, X, y=None):  
        return self  
  
    def transform(self, X, y=None):  
        if y is None:  
            raise ValueError("Target variable (y) is required for  
SMOTE.")  
        X_resampled, y_resampled = self.smote.fit_resample(X, y)  
        return X_resampled, y_resampled
```

2.2.4. Data Normalization

Seperti yang sudah disinggung pada bagian Feature Scaling, beberapa *machine learning model* bekerja lebih baik dengan data yang berdistribusi normal. Tahap ini dilakukan untuk memastikan hal itu. Pada tahap ini, atribut yang memiliki kemiringan (*skew*) yang tinggi ditransformasi dengan Log Transformation agar *variance*-nya lebih stabil.

```
class DataNormalizer(BaseEstimator, TransformerMixin):  
    def fit(self, X, y=None):  
        return self  
  
    def transform(self, X):  
        X_normalized = X.copy()  
        for col in X.columns:
```

```
        X_normalized[col] = np.log1p(X[col]) # log(1 + x)
for numerical stability
    return X_normalized
```

2.2.5. Dimensionality Reduction

Dimensionality Reduction dilakukan untuk mengecilkan jumlah fitur yang digunakan untuk *training*. Hal itu dapat dilakukan dengan Feature Selection ataupun Feature Extraction. Feature Extraction adalah proses pengelompokkan dan penggabungan data menjadi kolom-kolom baru yang tetap merepresentasikan kolom-kolom lama. Metode yang dipilih untuk Feature Extraction adalah PCA, yang membuat kolom-kolom yang berkorelasi rendah satu sama lain.

```
class DimensionalityReduction(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        self.pca = PCA(n_components=4, random_state=42)
        self.pca.fit(X)
        return self

    def transform(self, X):
        X = self.pca.transform(X)
        return X
```

2.3. Pipelining

Program ini mengimplementasi model KNN, Gaussian-Naive Bayes, dan ID3 untuk *training* dan prediksi. Kebutuhan dari ketiga model tersebut berbeda juga, sehingga ada beberapa tahap *preprocessing* yang cocok untuk satu model namun tidak yang lain.

```
knn_pipeline = Pipeline([
    ('encoding', FeatureEncoder()),
    ('scaling', FeatureScaler()),
    ('oversampling', HandleImbalance())
])

gnb_pipeline = Pipeline([
    ('encoding', FeatureEncoder()),
    ('oversampling', HandleImbalance())
])
```

```
])

id3_pipeline = Pipeline([
    ('encoding', FeatureEncoder()),
])

for col in categorical_feat:
    unseen_values = ~val_set[col].isin(train_set[col].unique())
    if unseen_values.any():
        # Replace unseen values with 'unknown'
        val_set[col] =
val_set[col].where(val_set[col].isin(train_set[col].unique()),
'unknown')

# Process train and validation sets
X_train_knn =
knn_pipeline.fit_transform(train_set.drop(columns=['attack_cat']),
train_set['attack_cat'])
X_val_knn =
knn_pipeline.transform(val_set.drop(columns=['attack_cat']))
y_train_knn = train_set['attack_cat']
y_val_knn = val_set['attack_cat']

X_train_gnb =
gnb_pipeline.fit_transform(train_set.drop(columns=['attack_cat']),
train_set['attack_cat'])
X_val_gnb =
gnb_pipeline.transform(val_set.drop(columns=['attack_cat']))
y_train_gnb = train_set['attack_cat']
y_val_gnb = val_set['attack_cat']

X_train_id3 =
id3_pipeline.fit_transform(train_set.drop(columns=['attack_cat']),
train_set['attack_cat'])
X_val_id3 =
id3_pipeline.transform(val_set.drop(columns=['attack_cat']))
y_train_id3 = train_set['attack_cat']
y_val_id3 = val_set['attack_cat']
```

BAB III

PERBANDINGAN HASIL PREDIKSI

3.1. Hasil KNN



Metric	Scratch KNN	Library KNN
Accuracy	0.65	0.64
Macro F1-Score	0.38	0.39
Weighted F1-Score	0.67	0.67
Reconnaissance Recall	0.85	0.84
DoS Precision	0.95	0.93
Shellcode F1-Score	0.47	0.50
Worms Recall	0.75	0.75

Pada implementasi KNN (scratch) dan KNN (library), performa model menunjukkan hasil yang hampir serupa, baik dalam metrik akurasi maupun F1-score. Akurasi yang diperoleh adalah 65% untuk implementasi manual dan 64% untuk implementasi library, dengan rata-rata macro F1-score masing-masing sebesar 0,3831 dan 0,3861. Perbedaan kecil ini menunjukkan

bahwa kedua implementasi berhasil melakukan klasifikasi dengan efektivitas yang hampir setara.

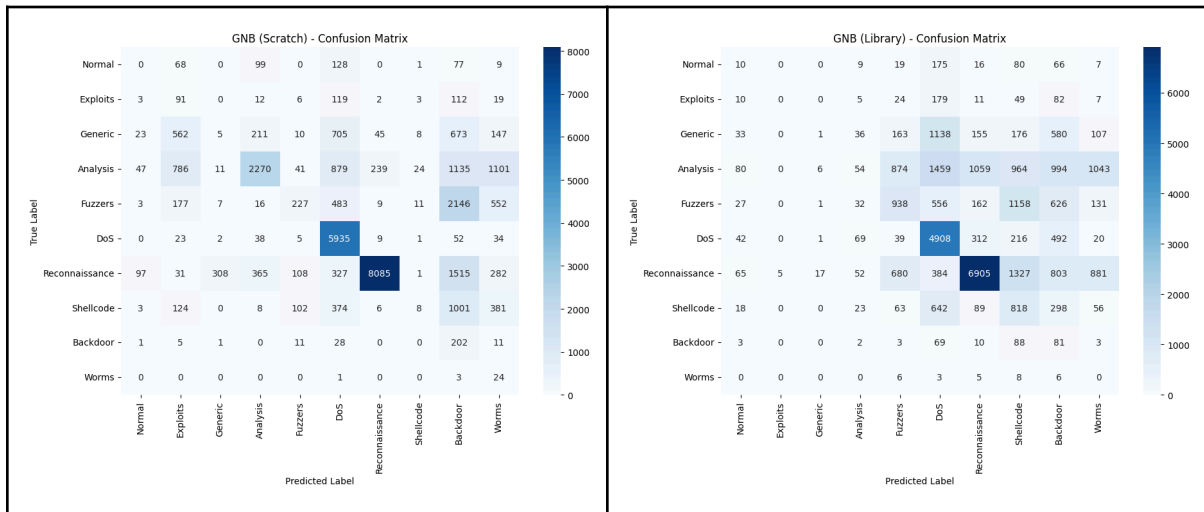
Perbedaan hasil antara keduanya terutama disebabkan oleh optimasi algoritma. KNN (scratch) memiliki keterbatasan dalam efisiensi pemrosesan dataset besar dan perhitungan jarak antar sampel, sedangkan KNN (library) yang menggunakan optimasi bawaan Scikit-Learn mampu memproses data lebih efisien, sehingga performanya lebih stabil, terutama pada kelas mayoritas seperti Reconnaissance dan DoS.

Pada KNN (scratch), recall untuk kelas minoritas seperti Worms cukup tinggi meskipun precision rendah, menunjukkan kecenderungan model untuk memprediksi lebih banyak sampel ke dalam kelas tersebut, meskipun banyak yang keliru. Sebaliknya, KNN (library) memberikan hasil yang lebih seimbang, meskipun performa tetap rendah pada kelas dengan jumlah data yang sangat sedikit, seperti Worms dan Backdoor.

Dari confusion matrix, terlihat bahwa sebagian besar kesalahan klasifikasi terjadi pada kelas dengan fitur yang mirip atau data yang tidak merata. Normal, Exploits, dan Generic sering diprediksi salah sebagai kelas lain, sementara kelas mayoritas seperti Reconnaissance dan DoS memiliki prediksi yang lebih baik karena distribusi data dan fitur yang lebih jelas.

Kesamaan performa kedua implementasi menunjukkan bahwa KNN cocok untuk menangani dataset ini, meskipun keterbatasannya dalam menangani class imbalance masih terlihat. Penggunaan library tidak memberikan peningkatan signifikan pada metrik evaluasi, tetapi membantu mempercepat komputasi, yang menjadi keuntungan utama untuk dataset yang besar.

3.2. Hasil Naive-Bayes



Metric	Scratch GNB	Library GNB
Accuracy	0.51	0.42
Macro F1-Score	0.2369	0.1938
Weighted F1-Score	0.54	0.40
DoS Precision	0.66	0.52
Reconnaissance Precision	0.96	0.79
Worms Recall	0.86	0.00
DoS Recall	0.97	0.80

Hasil evaluasi algoritma Gaussian Naive Bayes (GNB) menunjukkan bahwa implementasi dari scratch memiliki performa lebih baik dibandingkan implementasi berbasis library. Berdasarkan metrik *accuracy*, *macro F1-score*, *precision*, dan *recall*, implementasi dari scratch mencapai akurasi 51% dengan *macro F1-score* sebesar 0,2369. Model menunjukkan performa terbaik pada kelas DoS (recall 0,97) dan Reconnaissance (precision 0,96), mengindikasikan kemampuan model dalam menangkap pola distribusi meskipun dihadapkan pada *class imbalance* yang signifikan.

Sebaliknya, implementasi menggunakan library hanya mencapai akurasi 42% dengan *macro F1-score* sebesar 0,1938. Model ini gagal memprediksi beberapa kelas, terutama Worms, dengan recall 0,00 yang menunjukkan tidak ada instance yang berhasil terklasifikasi. Berdasarkan *confusion matrix*, sebagian besar instance kelas minoritas diklasifikasikan ke kelas mayoritas seperti Reconnaissance dan DoS. Meskipun recall pada kelas mayoritas DoS masih cukup baik (0,80), performanya tetap lebih rendah dibandingkan implementasi dari scratch.

Perbedaan performa ini disebabkan oleh fleksibilitas yang lebih besar pada implementasi dari scratch dalam menyesuaikan parameter model dengan distribusi dataset. Model dari scratch memungkinkan estimasi prior yang lebih sesuai dengan distribusi kelas serta optimasi manual pada parameter seperti mean dan variance, sehingga mampu menangkap pola data lebih akurat, khususnya pada kelas dengan jumlah instance kecil. Sebaliknya, implementasi berbasis library cenderung generik dan kurang optimal untuk dataset yang tidak seimbang, sehingga gagal menangani kelas minoritas seperti Worms dengan baik.

Secara keseluruhan, implementasi dari scratch unggul dalam menyesuaikan parameter model dengan dataset yang kompleks dan tidak seimbang. Model ini mampu memprediksi kelas mayoritas seperti DoS dan Reconnaissance dengan lebih baik, sambil mempertahankan performa pada kelas lainnya. Sementara itu, implementasi berbasis library, meskipun lebih cepat dan efisien, membutuhkan optimasi tambahan untuk meningkatkan performa pada kelas minoritas.

3.3. Hasil ID3



Metric	Scratch ID3	Library ID3
Accuracy	0.63	0.68
Macro F1-Score	0.3515	0.4242
Weighted F1-Score	0.63	0.69
DoS Precision	0.99	0.98
Reconnaissance Precision	0.89	0.85
Worms Recall	0.82	0.89
Normal Precision	0.08	0.08

Hasil evaluasi algoritma Decision Tree ID3 dianalisis menggunakan metrik *precision*, *recall*, *F1-score*, dan *accuracy*, dengan membandingkan implementasi dari scratch dan menggunakan library. Secara keseluruhan, implementasi menggunakan library menunjukkan performa yang lebih baik.

Pada implementasi dari scratch, akurasi yang diperoleh sebesar 63% dengan rata-rata *macro F1-score* sebesar 0,3515. Model memiliki performa terbaik pada kelas DoS dan Reconnaissance (precision dan recall mendekati 0,99 dan 0,89). Namun, model kesulitan

memprediksi kelas minoritas seperti Worms, Backdoor, dan Exploits, yang terlihat dari rendahnya nilai *precision* dan *recall*. Dari *confusion matrix*, banyak instance kelas minoritas diklasifikasikan keliru ke kelas mayoritas, seperti DoS dan Reconnaissance, sehingga menghasilkan prediksi yang tidak seimbang.

Implementasi menggunakan library menunjukkan akurasi sebesar 68% dan rata-rata *macro F1-score* sebesar 0,4242, dengan distribusi prediksi yang lebih merata di seluruh kelas. Precision dan recall pada kelas minoritas, seperti Backdoor dan Worms, juga meningkat, meskipun kesalahan prediksi pada kelas minoritas masih terjadi.

Peningkatan performa pada implementasi berbasis library disebabkan oleh optimasi algoritma pada library seperti scikit-learn, yang lebih efisien secara komputasi dan stabil dalam menangani dataset besar serta permasalahan *class imbalance*. Sebaliknya, implementasi dari scratch memiliki keterbatasan dalam hal optimasi teknis, seperti pengelolaan memori dan pemilihan threshold terbaik. Meskipun demikian, tantangan utama kedua implementasi tetap sama, yaitu kesulitan memprediksi kelas minoritas akibat keterbatasan jumlah sampel, yang berdampak pada kemampuan model untuk generalisasi.

BAB IV

KESIMPULAN

Performa model klasifikasi Decision Tree ID3, K-Nearest Neighbors (KNN), dan Gaussian Naive Bayes (GNB) bergantung pada karakteristik algoritma dan kondisi dataset yang tidak seimbang. Secara keseluruhan, ID3 berbasis library menunjukkan performa terbaik dengan *macro F1-score* tertinggi (0,4242), berkat kemampuannya memanfaatkan fitur kategorikal secara efektif. Sebaliknya, GNB memiliki performa terendah, baik dari implementasi library (0,1938) maupun scratch (0,2369), karena keterbatasannya dalam menangani distribusi data yang kompleks dan ketidakseimbangan kelas.

Model KNN memberikan hasil yang cukup konsisten dengan *macro F1-score* sekitar 0,38 untuk implementasi scratch maupun library. Keunggulan KNN terletak pada penggunaan metrik jarak yang efektif setelah preprocessing, meskipun masih kurang dalam memprediksi kelas minoritas seperti Worms dan Backdoor. Sementara itu, implementasi dari scratch untuk GNB dan ID3 menunjukkan hasil lebih baik dibandingkan library, karena memungkinkan penyesuaian manual yang telah dilakukan terhadap data tidak seimbang.

Evaluasi menggunakan *confusion matrix* memperlihatkan bahwa model lebih akurat dalam memprediksi kelas mayoritas seperti DoS dan Reconnaissance, sementara kelas minoritas sering diprediksi dengan buruk. Meskipun SMOTE membantu menyeimbangkan distribusi data, efektivitasnya tetap terbatas pada model yang kurang fleksibel dalam belajar dari variasi kelas.


Secara keseluruhan, ID3 adalah model terbaik untuk dataset ini karena akurasi yang konsisten dan mampu menangani ketidakseimbangan data. Di sisi lain, GNB memiliki performa terendah akibat keterbatasannya dalam menangkap distribusi probabilitas pada dataset kompleks.

LAMPIRAN

Tabel pembagian tugas:

NIM	Nama	Tugas
13522069	Nabila Shikoofa Muida	EDA, Implementasi Library, Laporan
13522096	Novelya Putri Ramadhani	Data Preprocessing, Implementasi KNN, Validation, Laporan
13522102	Hayya Zuhailii Kinasih	Data Preprocessing, Implementasi ID3, Laporan
13522104	Diana Tri Handayani	EDA, Implementasi GNB, Laporan

REFERENSI

-  Spesifikasi Tugas Besar 2 IF3170 Inteligensi Artifisial 2024/2025
- [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](#)
- [Entropy \(information theory\)](#)
- [Information gain \(information theory\)](#)