# Tugas Besar 2

# IF3070 Dasar Inteligensi Artifisial

# Implementasi Algoritma Pembelajaran Mesin


Disusun oleh:

Kelompok 24

| | |
|---|---|
| Darren Mansyl | 18223001 |
| Timothy Marvine | 18223021 |
| Amudi Purba | 18223049 |

| | **Program Studi Sistem dan Teknologi Informasi STEI-ITB** | **Jumlah Halaman** |
|---|---|---|
| | | |

# DAFTAR ISI

# BAB 1

## *Decision Tree Learning*

Decision Tree Learning merupakan salah satu metode *machine learning* berbasis *tree* yang digunakan untuk pengambilan keputusan secara bertahap. Model ini terdiri dari simpul (node) dan cabang (branch), di mana setiap simpul merepresentasikan suatu atribut pada dataset, sedangkan cabang merepresentasikan hasil dari keputusan berdasarkan atribut tersebut. Proses pengambilan keputusan dimulai dari simpul akar (root node), dilanjutkan melalui simpul-simpul, dan berakhir pada simpul daun (leaf node) yang menghasilkan prediksi akhir.

Decision tree termasuk algoritma **supervised learning** dan bersifat **non-parametrik**, artinya model ini tidak mengasumsikan bentuk distribusi tertentu pada data. Decision tree bekerja dengan cara membagi dataset menjadi subset yang lebih kecil berdasarkan atribut yang paling informatif, dan proses ini dilakukan secara rekursif hingga kondisi penghentian tertentu tercapai, seperti kedalaman maksimum pohon atau jumlah data minimum pada suatu node.

Secara umum, algoritma decision tree terbagi menjadi tiga pendekatan utama:

- ID3, merupakan algoritma decision tree generasi awal yang menggunakan **Information Gain** berbasis entropy sebagai kriteria pemilihan atribut. ID3 hanya mendukung atribut kategorikal dan cenderung bias terhadap atribut dengan jumlah kategori yang banyak, karena Information Gain tidak memberikan penalti terhadap jumlah cabang.
- C4.5, merupakan pengembangan dari ID3 dan mengatasi kelemahan utama ID3. Algoritma ini mendukung **atribut numerik dan kategorikal, m**enggunakan **Gain Ratio** sebagai kriteria pemilihan atribut kategorikal untuk menghindari bias terhadap atribut dengan banyak nilai unik, dan menggunakan **threshold split** untuk atribut numerik. Pendekatan inilah yang digunakan sebagai algoritma decision tree yang diimplementasi pada tugas besar ini.
- CART menggunakan **Gini Index** sebagai ukuran impurity dan selalu menghasilkan **binary tree**. CART dapat digunakan untuk klasifikasi maupun regresi, tetapi tidak menggunakan Gain Ratio maupun entropy seperti ID3 dan C4.5.

Untuk cara kerja Logistic Regression secara umum melibatkan beberapa langkah diantaranya,

1. Menyiapkan Data
2. Menghitung Entropy
3. Menentukan Split Terbaik untuk Fitur Numerik

4. Menentukan Split Terbaik untuk Fitur Kategorikal
5. Memilih Atribut Terbaik
6. Proses Rekursif Pembentukan Pohon

Berikut merupakan rumus-rumus yang kami pakai dalam algoritma kami,

1. Entropy
    - Rumus ini digunakan untuk mengukur ketidakmurnian node. Entropy 0 berarti semua data satu kelas.
    - Rumus:

$$H(S) = -\sum_{c \in C} p(c) log_2 p(c)$$

2. Weighted Entropy
    - Rumus ini digunakan untuk menghitung *impurity* gabungan setelah pembagian
    - Rumus:

$$H_{child} = \frac{|S_L|}{|S|} H(S_L) + \frac{|S_R|}{|S|} H(S_R)$$

3. Information Gain (Numerical)
    - Rumus ini digunakan untuk mengukur seberapa besar penurunan entropy setelah split
    - Rumus:

$$IG(S, t) = H(S) - H_{child}$$

4. Multi-branch
    - Rumus ini digunakan untuk menghitung impurity gabungan untuk split multi-cabang berdasarkan setiap kategori.
    - Rumus:

$$H_{child} = \sum_{v \in Values(A)} \frac{|S_v|}{|S|} H(S_v)$$

5. Information Gain (Categorical)
    - Rumus ini digunakan untuk mengukur penurunan *entropy* akibat split oleh atribut *A*
    - Rumus:

$$IG(S, A) = H(S) - H_{child}$$

6. Split Information
    - Rumus ini digunakan untuk “penalti” untuk atribut dengan banyak cabang (banyak kategori), agar tidak bias memilih fitur yang terlalu banyak nilai unik.
    - Rumus:

$$SplitInfo(S, A) = -\sum_{v \in Values(A)} \frac{|S_v|}{|S|} log_2\left(\frac{|S_v|}{|S|}\right)$$

7. Gain Ratio
    - Rumus ini digunakan untuk memilih split kategorikal yang informatif, tetapi tidak curang karena banyak kategori
    - Rumus:

$$GainRation(S, A) = \frac{IG(S,A)}{SplitInfo(S,A)}$$

```
class Node:
    def __init__(self, feature=None, threshold=None, children=None,
                 leaf_value=None, majority=None, prob=None):
        self.feature = feature
        self.threshold = threshold
        self.children = children
        self.leaf_value = leaf_value
        self.majority = majority
        self.prob = prob

def entropy(y):
    counts = Counter(y)
    total = len(y)
    return -sum((c/total)*np.log2(c/total) for c in
counts.values())

def best_split_numeric(feature, y):
    df = pd.DataFrame({"x": feature, "y": y}).sort_values("x")
    thresholds = np.percentile(df["x"], range(0, 100, 5))

    parent_entropy = entropy(y)
    best_gain, best_thresh = -np.inf, None

    for i in range(1, len(thresholds)):
        thresh = (thresholds[i-1] + thresholds[i]) / 2
        left_y  = df[df["x"] <= thresh]["y"]
        right_y = df[df["x"] > thresh]["y"]

        if len(left_y) == 0 or len(right_y) == 0:
            continue

        child_entropy = (
            len(left_y)/len(df) * entropy(left_y) +
            len(right_y)/len(df) * entropy(right_y)
        )

        gain = parent_entropy - child_entropy
        if gain > best_gain:
```

```
            best_gain, best_thresh = gain, thresh

    return best_thresh, best_gain, "numeric"


def gain_ratio_categorical(feature, y):
    df = pd.DataFrame({"x": feature, "y": y})
    parent_entropy = entropy(y)

    child_entropy, split_info = 0, 0
    for val in df["x"].unique():
        subset = df[df["x"] == val]["y"]
        p = len(subset) / len(df)
        child_entropy += p * entropy(subset)
        split_info -= p * np.log2(p)

    if split_info == 0:
        return None, -np.inf, None

    return None, (parent_entropy - child_entropy) / split_info,
"categorical"


def best_attribute(X, y):
    best_attr, best_gain, best_thresh, best_type = None, -np.inf,
None, None

    for col in X.columns:
        if col in categorical_cols:
            thresh, gain, t = gain_ratio_categorical(X[col], y)
        else:
            thresh, gain, t = best_split_numeric(X[col], y)

        if gain > best_gain:
            best_attr, best_gain, best_thresh, best_type = col,
gain, thresh, t

    return best_attr, best_thresh, best_type

def build_tree(X, y, depth=0, max_depth=20, min_samples=200):

    fraud_prob = (y.sum() + 1) / (len(y) + 2)

    if len(set(y)) == 1:
        return Node(leaf_value=y.iloc[0], majority=y.iloc[0],
prob=fraud_prob)

    if depth >= max_depth or len(X) < min_samples:
        maj = Counter(y).most_common(1)[0][0]
        return Node(leaf_value=maj, majority=maj, prob=fraud_prob)
```

```
    feature, threshold, feature_type = best_attribute(X, y)

    if feature is None:
        maj = Counter(y).most_common(1)[0][0]
        return Node(leaf_value=maj, majority=maj, prob=fraud_prob)

    node = Node(
        feature=feature,
        threshold=threshold,
        children={},
        majority=Counter(y).most_common(1)[0][0],
        prob=fraud_prob
    )

    if feature_type == "numeric":
        left = X[feature] <= threshold
        right = X[feature] > threshold

        node.children["left"]  = build_tree(X[left],  y[left],
depth+1, max_depth, min_samples)
        node.children["right"] = build_tree(X[right], y[right],
depth+1, max_depth, min_samples)
    else:
        for val in X[feature].unique():
            mask = X[feature] == val
            node.children[val] = build_tree(X[mask], y[mask],
depth+1, max_depth, min_samples)

    return node

def predict_one_proba(x, node):
    if node.leaf_value is not None:
        return node.prob

    if node.threshold is not None:
        return predict_one_proba(
            x,
            node.children["left"] if x[node.feature] <=
node.threshold else node.children["right"]
        )

    return predict_one_proba(x, node.children.get(x[node.feature],
node))


def predict_proba(X, tree):
    return X.apply(lambda row: predict_one_proba(row, tree),
axis=1)


def predict(X, tree, threshold=0.5):
```

```
    return (predict_proba(X, tree) >= threshold).astype(int)

tree = build_tree(X_train_dtl, y_train)

y_val_proba = predict_proba(X_val_dtl, tree)
y_val_pred  = (y_val_proba >= 0.3).astype(int)

print("ROC-AUC:", roc_auc_score(y_val, y_val_proba))
print(classification_report(y_val, y_val_pred))
```

Beberapa parameter utama yang digunakan dalam algoritma decision tree ini adalah sebagai berikut:

- max_depth = 20

  Parameter ini membatasi kedalaman maksimum pohon untuk mencegah model menjadi terlalu kompleks dan mengalami overfitting. Pembatasan kedalaman membantu menjaga generalisasi model terhadap data yang belum pernah dilihat.

- min_samples = 200

  Node hanya akan di-split jika jumlah data pada node tersebut melebihi batas minimum ini. Parameter ini berfungsi untuk mencegah pembentukan node berdasarkan jumlah data yang terlalu kecil, yang berpotensi menghasilkan keputusan yang tidak stabil.

- Threshold numerik berbasis persentil

  Untuk fitur numerik, kandidat threshold tidak diambil dari seluruh nilai unik, melainkan dari persentil data (5–95). Pendekatan ini secara signifikan mengurangi kompleksitas komputasi sekaligus menjaga kualitas pemisahan data.

# BAB 2

## *Logistic Regression*

Logistic Regression merupakan algoritma supervised machine learning yang digunakan untuk permasalahan klasifikasi. Berbeda dengan linear regression yang digunakan untuk memprediksi nilai kontinu, Logistic Regression digunakan untuk memprediksi probabilitas suatu data termasuk ke dalam kelas tertentu. Untuk cara kerja Logistic Regression secara umum melibatkan beberapa langkah diantaranya,

1. **Menyiapkan Data**

   Data input disiapkan dalam bentuk fitur (*X*) dan label target (*y*). Pada Logistic Regression, label target bersifat biner, yaitu bernilai 0 atau 1.

2. **Inisialisasi Parameter**

   Bobot (*w*) dan bias (*b*) diinisialisasi dengan nilai awal, umumnya nol atau nilai kecil secara acak.

3. **Menghitung Nilai Linear**

   Dilakukan perhitungan kombinasi linear antara fitur dan bobot menggunakan persamaan:

   $$z = X \cdot w + b$$

4. **Menerapkan Fungsi Sigmoid**

   Nilai *z* kemudian dimasukkan ke dalam fungsi sigmoid untuk menghasilkan probabilitas prediksi:

   $$\hat{y} = \frac{1}{1+e^{-z}}$$

   Nilai $\hat{y}$ berada pada rentang 0 hingga 1.

5. **Menghitung Gradien**

   Gradien dari fungsi loss terhadap bobot dan bias dihitung untuk mengetahui arah dan besar perubahan parameter.

6. **Memperbarui Parameter**

   Bobot dan bias diperbarui menggunakan metode Gradient Descent dengan mengurangi gradien yang dikalikan dengan learning rate.

7. **Proses Iterasi**

   Langkah perhitungan nilai linear, fungsi sigmoid, perhitungan loss, dan pembaruan parameter diulang hingga mencapai jumlah iterasi tertentu atau konvergensi.

8. **Melakukan Prediksi**

   Setelah proses pelatihan selesai, model digunakan untuk memprediksi data baru dengan menghitung probabilitas kelas.

9. **Klasifikasi Berdasarkan Threshold**

Probabilitas hasil prediksi dikonversi menjadi kelas biner menggunakan nilai ambang (*threshold*), umumnya 0.5. Jika probabilitas ≥ 0.5 maka data diklasifikasikan sebagai kelas 1, jika tidak sebagai kelas 0.

Berikut merupakan rumus-rumus yang digunakan dalam perhitungan Logistic Regression,

1. Model Linear
    - Rumus ini digunakan untuk menghitung kombinasi linear antara fitur input dengan bobot model. Nilai $z$ merepresentasikan skor awal sebelum dikonversi menjadi probabilitas kelas.
    - Rumus:

$$z = X \cdot w + b$$

2. Fungsi Sigmoid
    - Rumus ini digunakan untuk mengubah nilai linear $z$ menjadi nilai probabilitas dalam rentang 0 hingga 1, sehingga hasil prediksi dapat diinterpretasikan sebagai peluang suatu data termasuk ke dalam kelas positif.
    - Rumus:

$$\hat{y} = \frac{1}{1+e^{-z}}$$

3. Probabilitas Prediksi
    - Rumus ini digunakan untuk merepresentasikan probabilitas hasil prediksi model terhadap kelas positif (kelas 1) untuk setiap data masukan.
    - Rumus:

$$\hat{y} = \sigma(z)$$

4. Gradien terhadap Bobot
    - Rumus ini digunakan untuk menentukan arah dan besar perubahan bobot yang diperlukan agar kesalahan prediksi model dapat diminimalkan pada proses pelatihan.
    - Rumus:

$$\frac{\partial L}{\partial w} = \frac{1}{n} x^T \left( \hat{y} - y \right)$$

5. Gradien terhadap Bias
    - Rumus ini digunakan untuk mengukur kontribusi bias terhadap kesalahan prediksi, sehingga bias dapat diperbarui secara optimal selama proses pelatihan.
    - Rumus:

$$\frac{\partial L}{\partial w} = \frac{1}{n} \sum_{i=1}^{n} \left( \hat{y}i - yi \right)$$

6. Gradient Descent
    - Rumus ini digunakan untuk memperbarui nilai bobot model secara bertahap ke arah yang meminimalkan kesalahan prediksi, dengan α sebagai *learning rate*.
    - Rumus:

$$w := w - \alpha \frac{\partial L}{\partial b}$$

7. Threshold Klasifikasi
    - Rumus ini digunakan untuk mengonversi nilai probabilitas hasil prediksi menjadi label kelas biner yang bersifat diskret.
    - Rumus:

$$\hat{y}_{class} = \begin{cases} 1, & \text{jika } \hat{y} \geq 0.5 \\ 0, & \text{jika } \hat{y} < 0.5 \end{cases}$$

Berikut merupakan pseudocode dari algoritma Logistic Regression,

```
Input: Training data X, label y, learning rate lr, iterations
n_iter
Output: Weight vector w, bias b

1   Initialize w ← 0, b ← 0
2   For iter = 1 to n_iter do
3          z ← X · w + b
4          y_hat ← sigmoid (z)
5          dw ← (1/n) · X^T · (y_hat - y)
6          db ← mean(y_hat - y)
7          w ← w - lr · dw
8          b ← b - lr · db
9    End for
10    Return w, b
```

Berdasarkan *pseudocode* tersebut, kemudian kami mengimplementasikan dalam bahasa Python dengan dukungan library NumPy untuk memudahkan perhitungan.

```
class LogisticRegressionScratch:
    def __init__(self, lr=0.01, n_iter=3000, l2=0.0):
        self.lr = lr
        self.n_iter = n_iter
        self.l2 = l2

    def sigmoid(self, z):
        z = np.clip(z, -50, 50)
        return 1 / (1 + np.exp(-z))
```

```
    def fit(self, X, y):
        X = np.asarray(X, dtype=float)
        y = np.asarray(y, dtype=float)

        n_samples, n_features = X.shape
        self.w = np.zeros(n_features)
        self.b = 0.0

        for _ in range(self.n_iter):
            z = X @ self.w + self.b
            y_hat = self.sigmoid(z)

            dw = (X.T @ (y_hat - y)) / n_samples + self.l2 * self.w
            db = np.mean(y_hat - y)

            self.w -= self.lr * dw
            self.b -= self.lr * db

        return self

    def predict_proba(self, X):
        X = np.asarray(X, dtype=float)
        return self.sigmoid(X @ self.w + self.b)

    def predict(self, X, threshold=0.5):
        return (self.predict_proba(X) >= threshold).astype(int)
```

# BAB 3

# KNN

K-Nearest Neighbors (KNN) merupakan salah satu algoritma machine learning yang bersifat non-parametrik dan supervised, digunakan untuk tugas klasifikasi maupun regresi. Konsep utama KNN adalah bahwa objek yang mirip cenderung berada berdekatan satu sama lain dalam ruang fitur (feature space). Oleh karena itu, prediksi kelas atau nilai suatu data baru dilakukan dengan melihat sejumlah k tetangga terdekat dari data tersebut. Untuk cara kerja KNN secara umum melibatkan beberapa langkah diantaranya,

1. **Menghitung jarak**

   Untuk setiap sampel uji (X_test), algoritma menghitung jarak terhadap seluruh sampel pelatihan (X_train). Beberapa *distance metrics* yang umum digunakan adalah:

   - Euclidean distance
   - Manhattan distance
   - Minkowski distance

   Pemilihan metrik jarak sangat bergantung pada karakteristik dataset.

2. **Menentukan k tetangga terdekat**

   Setelah jarak dihitung, algoritma memilih k sampel *training* dengan jarak paling kecil terhadap data uji.

3. **Melakukan pemungutan suara**

   Kelas dipilih berdasarkan label yang paling sering muncul dari k tetangga terdekat

4. **Menghasilkan prediksi**

   Data uji diberi label atau nilai berdasarkan hasil voting atau nilai berdasarkan hasil voting atau perhitungan dari tetangga terdekat tersebut

Berikut merupakan rumus pada tiap *distance metrics* yang kami gunakan,

1. *Euclidean Distance*
   - Rumus ini digunakan untuk mengukur jarak lurus atau jarak geometris antara dua titik dalam ruang multidimensi
   - Rumus:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (xi - yi)^2}$$

2. *Manhattan Distance*

- Rumus ini digunakan untuk mengukur jarak sebagai jumlah selisih absolut antar-dimensi. Sering disebut sebagai *city-block distance* karena mirip jarak antar kota
- Rumus:

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

3. *Minkowski Distance*
   - Rumus ini merupakan generalisasi dari Euclidean dan Manhattan
   - Menggunakan paramater $p$ untuk mengontrol sensitivitas jarak terhadap perbedaan nilai antar fitur
   - Rumus:

$$d(x, y) = \left( \sum_{i=1} |xi - yi|^{p} \right)^{1/p}$$

Pada bagian ini kami memaparkan proses implementasi algoritma *K-Nearest Neighbors* (KNN) yang dibangun secara *from scratch*. Implementasi tersebut disusun dengan mengacu pada konsep KNN yang juga digunakan pada pustaka scikit-learn serta penjelasan algoritma dalam buku *Introduction to Information Retrieval*. Sebagai dasar, berikut merupakan pseudocode algoritma KNN yang telah disesuaikan dari referensi tersebut.

```
Train-KNN (C,D)
1   D' ← Preprocess (D)
2   k   ← Select-K(C,D)
3   return D' ,k

Apply-KNN (C,D',k,d)
1   S_k ← COMPUTENEAARESTNEIGHBORS (D',k,d)
2   for each c_j ∈ C
3   do p_j ← |S_k ∩ c_j|/k
4   return arg max_j p_j
```

Berdasarkan *pseudocode* tersebut, kemudian kami mengimplementasikan dalam bahasa Python dengan dukungan library NumPy untuk memudahkan perhitungan.

```
class KNNClassifier:
    def __init__(self, k=3, metric="euclidean"):
        self.k = k
        self.metric = metric.lower()
```

```
    def fit(self, X, y):
        self.X_train = np.asarray(X, dtype=float)
        self.y_train = np.asarray(y, dtype=int)
        return self

    def predict_one_proba(self, x):
        x = np.asarray(x, dtype=float)

        diff = self.X_train - x
        if self.metric == "euclidean":
            distances = np.sqrt(np.sum(diff**2, axis=1))
        elif self.metric == "manhattan":
            distances = np.sum(np.abs(diff), axis=1)
        else:
            raise ValueError("Unknown metric")

        k_idx = np.argpartition(distances, self.k)[:self.k]
        k_labels = self.y_train[k_idx]

        return np.mean(k_labels)

    def predict_proba(self, X):
        X = np.asarray(X, dtype=float)
        return np.array([self.predict_one_proba(x) for x in X])

    def predict(self, X, threshold=0.5):
        return (self.predict_proba(X) >= threshold).astype(int)
```

Dengan implementasi tersebut, algoritma K-Nearest Neighbors berhasil dijalankan sesuai konsep dasarnya, mulai dari perhitungan jarak hingga proses pemilihan tetangga terdekat dan penentuan kelas. Hasil prediksi dari model from scratch ini kemudian dibandingkan dengan implementasi scikit-learn untuk mengevaluasi konsistensi dan akurasinya.

# BAB 4

# *Cleaning and Pre-Processing*

## 4.1 Data Cleaning

Tahap data cleaning bertujuan untuk memastikan kualitas data sebelum digunakan dalam proses pemodelan. Pada tahap ini dilakukan identifikasi dan penanganan permasalahan umum pada data, seperti nilai hilang (*missing values*), *outliers*, dan duplikasi data. Proses ini penting karena data yang tidak bersih dapat menyebabkan bias, menurunkan performa model, serta menghasilkan prediksi yang tidak stabil. Seluruh proses dilakukan secara sistematis menggunakan pendekatan berbasis transformer agar dapat diintegrasikan ke dalam *machine learning pipeline*.

### 4.1.1 Handling Missing Data

Tahap ini bertujuan untuk mengidentifikasi dan menangani nilai yang hilang pada dataset. Pertama, kode menghitung jumlah dan persentase missing values pada setiap fitur untuk mengetahui tingkat keparahan masalah data hilang. Selanjutnya, *class FeatureImputer* digunakan untuk mengisi nilai hilang dengan pendekatan yang sesuai, yaitu median untuk fitur numerik dan most frequent untuk fitur kategorikal. Pendekatan ini dipilih karena median relatif *robust* terhadap *outlier*, sedangkan nilai yang paling sering muncul menjaga distribusi kategori tetap representatif. Proses imputasi dilakukan melalui metode *fit* dan *transform* agar konsisten antara data latih, validasi, dan uji.

```
missing_values = df_train.isnull().sum()

missing_percentage = (missing_values / len(df_train)) * 100

missing_info = pd.DataFrame({'Missing Values': missing_values,
'Percentage': missing_percentage})
missing_info = missing_info[missing_info['Missing Values'] > 0]
print(missing_info)
```

Kode dimulai dengan memeriksa kondisi data untuk mengetahui apakah terdapat nilai yang hilang pada setiap fitur. Hal ini dilakukan dengan menghitung jumlah dan persentase *missing values*, sehingga peneliti dapat memahami seberapa serius permasalahan data yang dihadapi.

```
class FeatureImputer(BaseEstimator, TransformerMixin):
    def __init__(self, numerical_cols, categorical_cols):
        self.numerical_cols = numerical_cols
        self.categorical_cols = categorical_cols
        self.num_imputer = SimpleImputer(strategy="median")
        self.cat_imputer = SimpleImputer(strategy="most_frequent")
```

```
    def fit(self, X, y=None):
        self.num_cols_ = [
            c for c in self.numerical_cols
            if c in X.columns and
pd.api.types.is_numeric_dtype(X[c]) and c not in
self.categorical_cols
        ]

        self.cat_cols_ = [
            c for c in self.categorical_cols
            if c in X.columns
        ]

        if self.num_cols_:
            self.num_imputer.fit(X[self.num_cols_])

        if self.cat_cols_:
            self.cat_imputer.fit(X[self.cat_cols_])

        return self

    def transform(self, X):
        X = X.copy()

        if self.num_cols_:
            X[self.num_cols_] =
self.num_imputer.transform(X[self.num_cols_])

        if self.cat_cols_:
            X[self.cat_cols_] =
self.cat_imputer.transform(X[self.cat_cols_])

        return X
```

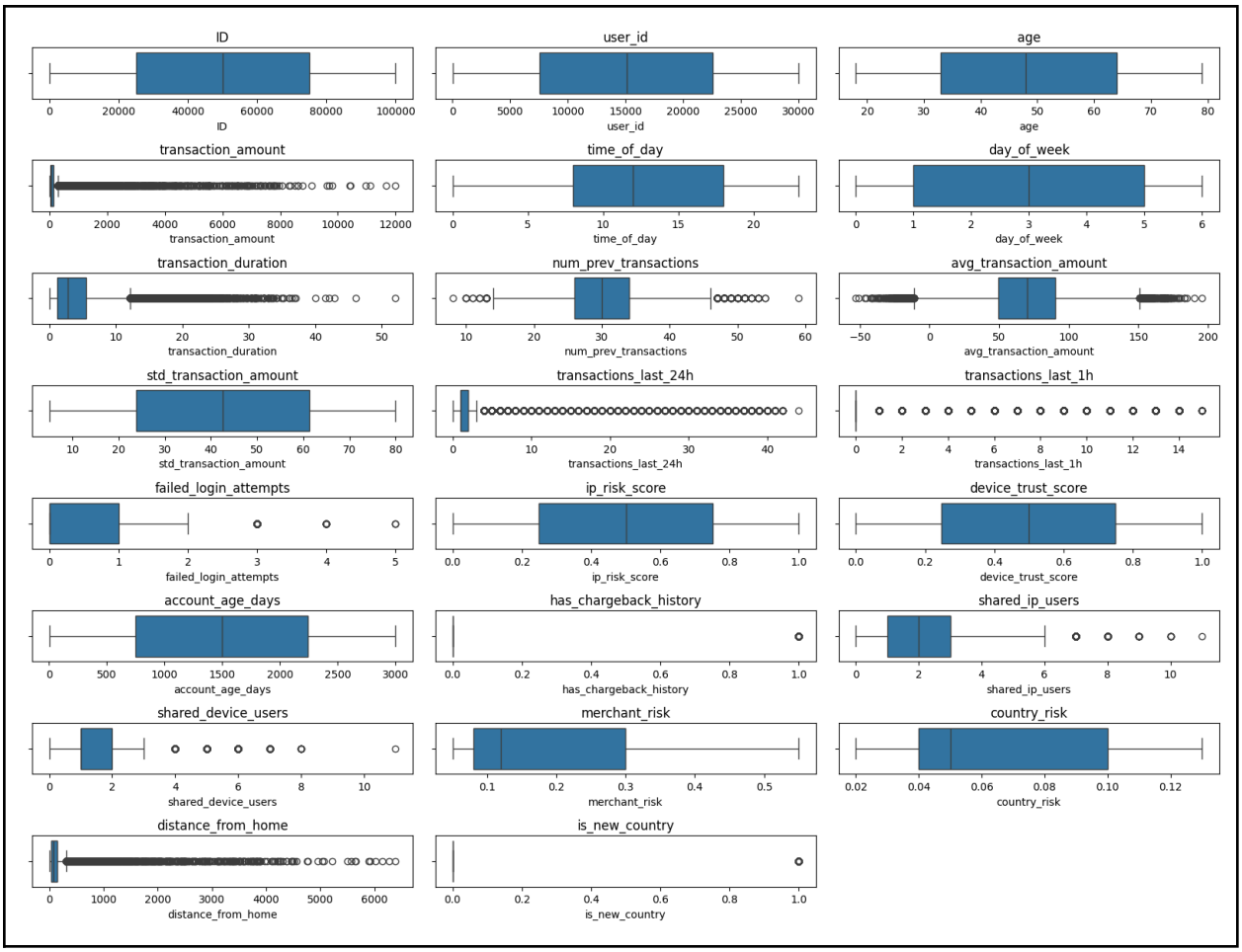
Setelah itu, *class FeatureImputer* berfungsi untuk mengisi nilai yang hilang pada data secara otomatis dengan cara yang sesuai dengan jenis fiturnya. Pada saat inisialisasi, algoritma menerima daftar fitur numerik dan kategorikal, lalu menyiapkan dua strategi pengisian yang berbeda, yaitu median untuk data numerik dan nilai yang paling sering muncul untuk data kategorikal. Ketika metode fit dijalankan, algoritma terlebih dahulu mengecek kolom mana saja yang benar-benar ada di dataset dan sesuai dengan tipe datanya, kemudian mempelajari nilai pengganti yang paling representatif dari data latih. Selanjutnya, pada tahap transform, nilai kosong pada setiap fitur akan digantikan menggunakan nilai yang telah dipelajari sebelumnya, sehingga data menjadi lengkap, konsisten, dan siap digunakan untuk proses pemodelan tanpa mengubah pola utama distribusi data..

4.1.2 Dealing with Outliers

Tahap penanganan outlier dilakukan untuk mengurangi pengaruh nilai ekstrem yang dapat mendistorsi proses pelatihan model. Visualisasi menggunakan *boxplot* digunakan untuk memberikan gambaran awal distribusi data numerik dan keberadaan outlier. Selanjutnya, kelas OutlierClipper menerapkan metode *Interquartile Range* (IQR) untuk menentukan batas bawah dan atas setiap fitur numerik. Nilai yang berada di luar batas tersebut tidak dihapus, tetapi diklip menggunakan fungsi np.clip agar tetap berada dalam rentang yang wajar. Pendekatan ini menjaga ukuran dataset tetap konsisten sekaligus mengurangi dampak ekstrem pada model.

```
df_train_non_cat = X_train.select_dtypes(include=["int64",
"float64"])

plt.figure(figsize=(16, 12))

for i, col in enumerate(df_train_non_cat.columns, 1):
    plt.subplot((len(df_train_non_cat.columns) + 2) // 3, 3, i)
    sns.boxplot(x=df_train_non_cat[col])
    plt.title(col)

plt.tight_layout()
plt.show()
```

Untuk menangani nilai-nilai ekstrem (*outliers*), boxplot digunakan untuk setiap fitur numerik agar dapat melihat sebaran data secara visual, terlampir sebagai berikut.




```
class OutlierClipper(BaseEstimator, TransformerMixin):
    def __init__(self, factor=1.5):
        self.factor = factor
        self.bounds = {}

    def fit(self, X, y=None):
        for col in X.columns:
            if pd.api.types.is_numeric_dtype(X[col]):
                Q1 = X[col].quantile(0.25)
                Q3 = X[col].quantile(0.75)
                IQR = Q3 - Q1
                lower = Q1 - self.factor * IQR
                upper = Q3 + self.factor * IQR
                self.bounds[col] = (lower, upper)

        return self

    def transform(self, X):
        X_out = X.copy()
```

```
        for col in self.bounds:
            lower, upper = self.bounds[col]
            X_out[col] = np.clip(X_out[col], lower, upper)

        return X_out
```

Selanjutnya, *class OutlierClipper* bekerja dengan mencari rentang nilai wajar menggunakan metode *Interquartile Range* (IQR). Pada tahap fit, algoritma menghitung batas bawah dan batas atas yang masih dianggap normal untuk setiap fitur. Ketika data ditransformasi, nilai yang berada di luar batas tersebut tidak dihapus, tetapi "dipaksa" masuk kembali ke batas terdekat. Dengan pendekatan ini, informasi data tetap dipertahankan, namun pengaruh nilai ekstrem terhadap model dapat diminimalkan.

4.1.3 Remove Duplicates

Penghapusan data duplikat dilakukan untuk memastikan bahwa setiap baris data merepresentasikan satu observasi yang unik. Kode terlebih dahulu menghitung jumlah data duplikat pada data latih, validasi, dan uji untuk memverifikasi keberadaan redundansi. Selanjutnya, metode *drop_duplicates()* digunakan untuk menghapus baris yang identik. Langkah ini penting karena data duplikat dapat menyebabkan *data leakage*, memperbesar bobot observasi tertentu, serta menghasilkan evaluasi model yang bias.

```
print("Duplicates in train set: ", train_set.duplicated().sum())
print("Duplicates in validation set: ", val_set.duplicated().sum())
print("Duplicates in test set: ", df_test.duplicated().sum())

# hapus duplikat
train_set = train_set.drop_duplicates()
val_set = val_set.drop_duplicates()
df_test = df_test.drop_duplicates()

print("Duplicates in train set after removal: ",
train_set.duplicated().sum())
print("Duplicates in validation set after removal: ",
val_set.duplicated().sum())
print("Duplicates in test set after removal: ",
df_test.duplicated().sum())
```

Algoritma terlebih dahulu menghitung berapa banyak data yang muncul lebih dari satu kali pada dataset pelatihan, validasi, dan pengujian. Jika ditemukan data duplikat, baris tersebut dihapus menggunakan fungsi *drop_duplicates()*. Langkah ini penting karena data yang sama jika muncul berulang dapat membuat model "belajar berlebihan" pada pola tertentu, sehingga hasil evaluasi menjadi tidak objektif.

4.1.4 Feature Engineering

*Feature engineering* dilakukan untuk meningkatkan kemampuan representasi data terhadap pola yang relevan dengan target. Pada tahap ini, fitur-fitur yang tidak informatif, bersifat unik, atau berpotensi menyebabkan noise dihapus dari dataset. Selanjutnya, *class FeatureEngineer* menghasilkan fitur baru berbasis domain knowledge, seperti rasio transaksi, risiko perangkat, anomali lokasi, dan kombinasi skor risiko. Fitur-fitur ini dirancang untuk menangkap hubungan non-linear dan perilaku mencurigakan yang tidak dapat direpresentasikan secara langsung oleh fitur asli. Seluruh operasi dilakukan dengan pengamanan numerik untuk mencegah pembagian nol dan nilai ekstrem.

```
# buang fitur yang tidak diperlukan, misalnya: yang unik, redundan,
terlalu banyak null
drop_cols = [
    "ID", "transaction_id", "user_id",
    "age", "gender", "country",
    "device_type", "device_os",
    "merchant_category", "transaction_type",
    "day_of_week", "time_of_day"
]

categorical_cols = [
    col for col in [
        "gender",
        "country",
        "device_type",
        "device_os",
        "transaction_type",
        "merchant_category",
        "day_of_week",
        "has_chargeback_history",
        "is_new_country"
    ] if col not in drop_cols
]



numerical_cols = [
    col for col in X_train.columns
    if col not in categorical_cols and col not in [
        "ID", "transaction_id", "user_id"
    ]
]
```

Pada bagian ini, data mulai diperkaya agar model lebih mudah mengenali pola penting. Pertama, fitur-fitur yang dianggap tidak relevan atau berpotensi mengganggu, seperti ID unik atau atribut deskriptif tertentu, dihapus dari dataset.

```
class FeatureEngineer(BaseEstimator, TransformerMixin):
```

```
    def __init__(self):
        pass

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

    def transform(self, X):
        X = X.copy()
        eps = 1e-6

        # amount_to_avg_ratio
        if "transaction_amount" in X.columns and
"avg_transaction_amount" in X.columns:
            denom = X["avg_transaction_amount"].abs() + 1 + eps
            X["amount_to_avg_ratio"] = (X["transaction_amount"] /
denom).clip(-1e6, 1e6)

        # ratio_1h_to24h
        if "transactions_last_1h" in X.columns and
"transactions_last_24h" in X.columns:
            denom = X["transactions_last_24h"].abs() + 1 + eps
            X["ratio_1h_to24h"] = ((X["transactions_last_1h"] + 1)
/ denom).clip(-1e6, 1e6)

        # device_shared_risk
        if "shared_device_users" in X.columns and
"device_trust_score" in X.columns:
            X["device_shared_risk"] = (
                X["shared_device_users"] * (1 -
X["device_trust_score"])
            ).clip(-1e6, 1e6)

        # location_anomaly
        if "distance_from_home_log" in X.columns and
"is_new_country" in X.columns:
            safe_dist =
X["distance_from_home_log"].replace([np.inf, -np.inf], 0)
            X["location_anomaly"] = (safe_dist *
X["is_new_country"]).clip(-1e6, 1e6)

        # combined_risk
        if {"ip_risk_score", "merchant_risk",
"country_risk"}.issubset(X.columns):
            X["combined_risk"] = (
                X["ip_risk_score"] *
                X["merchant_risk"] *
                X["country_risk"]
            ).clip(-1e6, 1e6)

        return X
```

Selanjutnya, kelas *FeatureEngineer* membuat fitur-fitur baru dengan menggabungkan beberapa informasi yang sudah ada, misalnya rasio nilai transaksi, tingkat risiko perangkat, hingga indikasi anomali lokasi. Setiap fitur baru dirancang untuk menangkap perilaku yang tidak bisa terlihat langsung dari satu kolom saja. Seluruh perhitungan dilengkapi dengan pengamanan agar tidak terjadi pembagian nol atau nilai yang terlalu besar.

### 4.2 Data Preprocessing

Tahap data preprocessing bertujuan untuk menyiapkan data agar kompatibel dengan kebutuhan algoritma machine learning yang digunakan. Proses ini mencakup penskalaan fitur, pengkodean data kategorikal, penanganan ketidakseimbangan kelas, normalisasi, serta reduksi dimensi. Setiap langkah dirancang untuk meningkatkan stabilitas pelatihan model, mempercepat konvergensi, dan menghindari dominasi fitur tertentu dalam proses pembelajaran.

#### 4.2.1 Feature Scaling

*Feature scaling* dilakukan untuk menyamakan skala antar fitur numerik agar tidak ada fitur yang mendominasi proses perhitungan jarak atau bobot. *Class FeatureScaler* menggunakan *StandardScaler* untuk mengubah distribusi data sehingga memiliki rata-rata nol dan standar deviasi satu. Pendekatan ini sangat penting untuk algoritma berbasis jarak seperti KNN. Proses scaling dilakukan hanya pada fitur numerik dan diterapkan secara konsisten melalui metode *fit* dan *transform*.

```
class DropColumns(BaseEstimator, TransformerMixin):
    def __init__(self, cols):
        self.cols = cols

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

    def transform(self, X):
        return X.drop(columns=[c for c in self.cols if c in
X.columns], errors="ignore")
```

Kolom tertentu yang dianggap tidak relevan atau tidak diperlukan dalam proses pemodelan akan dihapus terlebih dahulu. Algoritma menerima daftar nama kolom pada saat inisialisasi, lalu pada tahap transform kolom-kolom tersebut akan dihapus dari dataset jika memang tersedia. Penggunaan *errors="ignore"* memastikan proses tetap berjalan meskipun ada kolom yang tidak ditemukan, sehingga transformer ini aman digunakan pada berbagai dataset dengan struktur kolom yang sedikit berbeda.

```
class FeatureScaler(BaseEstimator, TransformerMixin):
    def __init__(self, num_features=None):
        self.num_features = num_features
        self.scaler = StandardScaler()

    def fit(self, X, y=None):
        if self.num_features is None:
            self.num_features = X.select_dtypes(include=['int64',
'float64']).columns.tolist()

        self.scaler.fit(X[self.num_features])
        return self

    def transform(self, X):
        X_scaled = X.copy()
        X_scaled[self.num_features] =
self.scaler.transform(X[self.num_features])
        return X_scaled
```

*Class FeatureEngineer* berguna untuk membentuk fitur-fitur baru yang lebih informatif dengan mengombinasikan beberapa fitur asli sehingga pola penting dalam data dapat lebih mudah dikenali oleh model. Pada tahap transform, data terlebih dahulu disalin agar tidak mengubah data asli, kemudian dilakukan perhitungan berbagai fitur turunan berbasis rasio dan risiko, seperti perbandingan nilai transaksi terhadap rata-rata transaksi, rasio aktivitas transaksi jangka pendek terhadap jangka panjang, tingkat risiko penggunaan perangkat bersama, indikasi anomali lokasi, serta kombinasi skor risiko dari berbagai sumber. Setiap fitur hanya dibuat jika kolom-kolom pendukungnya tersedia, dan seluruh perhitungan dilengkapi dengan penyesuaian numerik serta pembatasan nilai untuk mencegah pembagian nol dan nilai ekstrem. Dengan pendekatan ini, fitur yang dihasilkan mampu merepresentasikan perilaku yang lebih kompleks dan relevan dibandingkan fitur mentah, tanpa mengorbankan stabilitas data.

### 4.2.2 Feature Encoding

*Feature encoding* bertujuan untuk mengubah data kategorikal menjadi representasi numerik yang dapat diproses oleh model. Kelas *FeatureEncoder* menggunakan *LabelEncoder* untuk setiap fitur kategorikal dan menambahkan label khusus *unknown* untuk menangani kategori yang tidak muncul saat pelatihan. Pendekatan ini mencegah error pada tahap inferensi dan memastikan *robustness model* terhadap data baru. Proses *encoding* dilakukan secara terpisah per fitur untuk menjaga konsistensi *mapping* kategori.

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

```
        self.cat_cols = None

    def fit(self, X, y=None):
        self.cat_cols = X.select_dtypes(include=['object',
'category']).columns.tolist()

        for col in self.cat_cols:
            le = LabelEncoder()

            unique_values = X[col].astype(str).unique().tolist()

            if self.unknown_label not in unique_values:
                unique_values.append(self.unknown_label)

            le.fit(unique_values)
            self.encoders[col] = le

        return self

    def transform(self, X):
        X_transformed = X.copy()

        for col in self.cat_cols:
            le = self.encoders[col]

            X_transformed[col] = X_transformed[col].astype(str)

            X_transformed[col] = X_transformed[col].apply(
                lambda val: val if val in le.classes_ else
self.unknown_label
            )

            X_transformed[col] = le.transform(X_transformed[col])

        return X_transformed
```

Karena model tidak dapat memproses teks atau kategori secara langsung, fitur kategorikal harus diubah menjadi angka. Kelas *FeatureEncoder* memberi label numerik pada setiap kategori yang muncul di data latih. Selain itu, algoritma juga menyiapkan label khusus bernama *unknown* untuk mengantisipasi kategori baru yang mungkin muncul pada data validasi atau pengujian. Ketika data diproses, setiap kategori diubah menjadi angka sesuai *mapping* yang telah dipelajari, sehingga model dapat menerima data baru tanpa mengalami error.

### 4.2.3 Handling Imbalanced Dataset

Penanganan ketidakseimbangan kelas dilakukan untuk mengatasi dominasi kelas mayoritas yang dapat menyebabkan model bias. Kelas HandleImbalance menggunakan metode SMOTE (Synthetic

Minority Over-sampling Technique) untuk menghasilkan sampel sintetis pada kelas minoritas. Pendekatan ini meningkatkan kemampuan model dalam mengenali pola pada kelas minoritas tanpa menghapus data mayoritas. Proses *resampling* hanya diterapkan pada data latih untuk menghindari kebocoran informasi.

```
class HandleImbalance(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.smote = SMOTE(random_state=42)
        self.X_resampled = None
        self.y_resampled = None

    def fit(self, X, y):
        self.X_resampled, self.y_resampled =
self.smote.fit_resample(X, y)
        return self

    def transform(self, X):
        return X

    def get_resampled_data(self):
        return self.X_resampled, self.y_resampled
```

Pada dataset dengan distribusi kelas yang tidak seimbang, model cenderung hanya belajar dari kelas mayoritas. Untuk mengatasi hal ini, diterapkan SMOTE yang bekerja dengan membuat data sintetis pada kelas minoritas. Algoritma ini membentuk sampel baru berdasarkan pola yang ada. Proses ini hanya diterapkan pada data latih agar evaluasi model tetap adil dan tidak terjadi kebocoran informasi.

### 4.2.4 Data Normalization

Normalisasi dilakukan untuk mengubah vektor fitur menjadi skala unit tertentu menggunakan metode vector normalization. Kelas *FeatureNormalizer* menerapkan *Normalizer* dengan norma L2 agar setiap sampel memiliki panjang vektor yang seragam. Teknik ini berguna pada algoritma yang sensitif terhadap arah vektor dibandingkan besarannya. Hasil normalisasi dikembalikan dalam bentuk *DataFrame* agar struktur data tetap konsisten.

```
class FeatureNormalizer(BaseEstimator, TransformerMixin):

    def __init__(self, norm="l2"):
        self.norm = norm
        self.normalizer = Normalizer(norm=self.norm)

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

    def transform(self, X):
        X_norm = self.normalizer.transform(X)
```

```
        return pd.DataFrame(X_norm, columns=X.columns,
index=X.index)
```

Normalisasi dilakukan dengan cara mengubah setiap baris data menjadi vektor dengan panjang yang sama. Algoritma ini tidak mengubah arah data, hanya menyamakan skala keseluruhan tiap sampel. Proses ini berguna ketika model lebih memperhatikan pola relatif antar fitur dibandingkan besar nilainya. Hasil normalisasi kemudian dikembalikan dalam bentuk *DataFrame* agar mudah digunakan pada tahap selanjutnya.

4.2.5 Dimensionality Reduction

Reduksi dimensi bertujuan untuk mengurangi kompleksitas data sekaligus mempertahankan informasi utama. Kelas *DimensionalityReducer* menggunakan *Principal Component Analysis* (PCA) untuk memproyeksikan data ke ruang berdimensi lebih rendah. Pendekatan ini membantu mengurangi *curse of dimensionality*, mempercepat komputasi, dan mengurangi risiko *overfitting*. Jumlah komponen ditentukan sebagai parameter agar fleksibel terhadap kebutuhan model.

```
class DimensionalityReducer(BaseEstimator, TransformerMixin):
    def __init__(self, n_components=20):
        self.n_components = n_components
        self.pca = PCA(n_components=n_components)

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

    def transform(self, X):
        X_pca = self.pca.transform(X)
        return pd.DataFrame(
            X_pca,
            index=X.index
        )
```

Karena jumlah fitur yang banyak dapat membuat model menjadi lambat dan kompleks, dilakukan reduksi dimensi menggunakan PCA. Algoritma PCA bekerja dengan mencari kombinasi fitur baru yang mampu mewakili sebagian besar informasi dari data asli. Pada tahap fit, PCA mempelajari arah variasi terbesar pada data latih. Selanjutnya, data diproyeksikan ke ruang berdimensi lebih rendah, sehingga model dapat bekerja lebih efisien tanpa kehilangan informasi penting.

# BAB 5

## *Compile Preprocessing Pipeline*

Pada tahap ini, seluruh proses *data cleaning* dan *preprocessing* digabungkan ke dalam *machine learning pipeline* menggunakan *Pipeline* dari *scikit-learn*. *Pipeline* KNN mencakup imputasi, penanganan *outlier*, *feature engineering*, *encoding*, *scaling*, dan reduksi dimensi karena algoritma KNN sensitif terhadap skala dan dimensi data. Sementara itu, pipeline DTL tidak menggunakan *scaling* dan PCA karena *decision tree* tidak bergantung pada jarak antar fitur. Pendekatan *pipeline* memastikan konsistensi *preprocessing*, *mencegah data leakage*, dan mempermudah replikasi eksperimen pada data latih, validasi, dan uji.

```
knn_preprocess = Pipeline([
    ("imputer", FeatureImputer(numerical_cols, categorical_cols)),
    ("outlier", OutlierClipper()),
    ("engineer", FeatureEngineer()),
    ("drop", DropColumns(drop_cols)),
    ("encoder", FeatureEncoder()),
    ("scaler", FeatureScaler()),
    ("pca", DimensionalityReducer(n_components=5))
])

dtl_preprocess = Pipeline([
    ("imputer", FeatureImputer(numerical_cols, categorical_cols)),
    ("outlier", OutlierClipper()),
    ("engineer", FeatureEngineer()),
    ("drop", DropColumns(drop_cols)),
    ("encoder", FeatureEncoder())
])

logreg_preprocess = Pipeline([
    ("imputer", FeatureImputer(numerical_cols, categorical_cols)),
    ("outlier", OutlierClipper()),
    ("engineer", FeatureEngineer()),
    ("scaler", FeatureScaler(
        num_features=[
            c for c in numerical_cols
            if c not in categorical_cols and c not in drop_cols
        ]
    )),
    ("drop", DropColumns(drop_cols)),
    ("encoder", FeatureEncoder())
])
```

Pada algoritma, seluruh proses *preprocessing* disusun menjadi satu alur kerja menggunakan *pipeline*. *Pipeline* memastikan bahwa data selalu diproses dengan urutan yang sama, mulai dari imputasi hingga transformasi akhir.

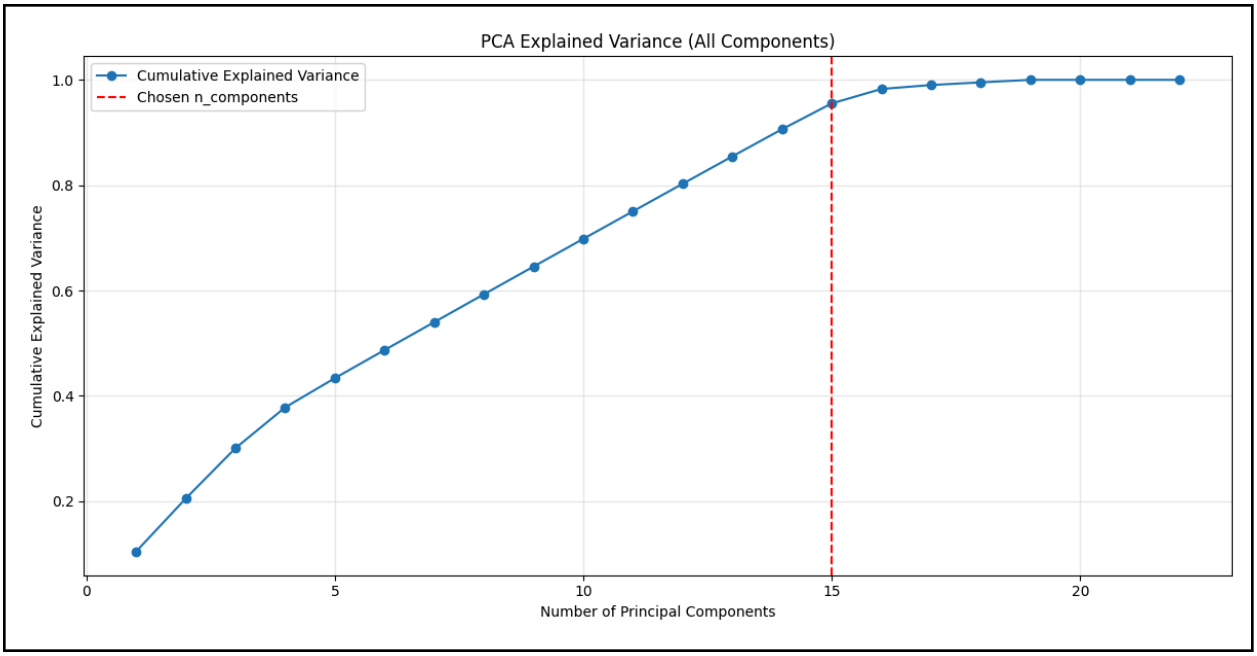
```
# KNN
X_train_knn = knn_preprocess.fit_transform(X_train)
X_val_knn   = knn_preprocess.transform(X_val)
X_test_knn  = knn_preprocess.transform(df_test)

# Balancing
smote = SMOTE(random_state=42)
X_train_knn_bal, y_train_knn_bal = smote.fit_resample(
    X_train_knn, y_train
)

# DTL
X_train_dtl = dtl_preprocess.fit_transform(X_train)
X_val_dtl   = dtl_preprocess.transform(X_val)
X_test_dtl  = dtl_preprocess.transform(df_test)

# Logistic Regression
X_train_logreg = logreg_preprocess.fit_transform(X_train)
X_val_logreg   = logreg_preprocess.transform(X_val)
X_test_logreg  = logreg_preprocess.transform(df_test)

X_train_logreg = X_train_logreg.values
X_val_logreg   = X_val_logreg.values
X_test_logreg  = X_test_logreg.values

y_train_np = y_train.values.astype(int)
y_val_np   = y_val.values.astype(int)
```

Untuk setiap algoritma, proses *preprocessing* disesuaikan dengan cara kerja model tersebut. Pada KNN, *pipeline* dibuat lebih lengkap karena algoritma ini menentukan prediksi berdasarkan jarak antar data, sehingga skala fitur dan jumlah dimensi sangat memengaruhi hasil perhitungan. Oleh karena itu, data perlu distandarisasi dan diringkas agar perbandingan jarak menjadi adil dan stabil. Sebaliknya, pada *Decision Tree*, *pipeline* disusun lebih sederhana karena model ini membuat keputusan berdasarkan aturan pemisahan nilai fitur, bukan perhitungan jarak. Dengan menyusun *preprocessing* dalam bentuk *pipeline*, seluruh tahapan dapat dijalankan secara berurutan dan konsisten, sehingga proses pelatihan dan pengujian model menjadi lebih terstruktur, transparan, dan mudah direproduksi.

```
X_train_ready = knn_preprocess.named_steps["scaler"].transform(
    knn_preprocess.named_steps["encoder"].transform(
        knn_preprocess.named_steps["drop"].transform(
            knn_preprocess.named_steps["engineer"].transform(
                knn_preprocess.named_steps["outlier"].transform(
```

```
knn_preprocess.named_steps["imputer"].fit_transform(X_train)
                )
            )
        )
    )
)


from sklearn.decomposition import PCA

pca_full = PCA()
pca_full.fit(X_train_ready)

explained_var = pca_full.explained_variance_ratio_
cumulative_var = np.cumsum(explained_var)

plt.figure(figsize=(12, 6))

plt.plot(
    range(1, len(cumulative_var) + 1),
    cumulative_var,
    marker="o",
    label="Cumulative Explained Variance"
)

plt.axvline(
    x=15,
    color="red",
    linestyle="--",
    label="Chosen n_components"
)

plt.xlabel("Number of Principal Components")
plt.ylabel("Cumulative Explained Variance")
plt.title("PCA Explained Variance (All Components)")
plt.legend()
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```

Algoritma ini digunakan untuk menentukan jumlah principal components PCA yang optimal secara objektif dengan terlebih dahulu menerapkan seluruh tahapan preprocessing yang sama seperti pada pipeline KNN (imputasi, penanganan *outlier*, *feature engineering*, *encoding*, dan *scaling*), namun tanpa langsung membatasi jumlah komponen PCA. Setelah data berada dalam bentuk numerik dan terstandarisasi, PCA diterapkan ke seluruh komponen untuk menghitung *explained variance ratio* dan *cumulative explained variance*, yang kemudian divisualisasikan dalam bentuk kurva.




Grafik menunjukkan hubungan antara jumlah principal components dan proporsi variansi data yang berhasil dijelaskan secara kumulatif, di mana kurva meningkat tajam pada komponen awal lalu mulai melandai seiring bertambahnya komponen. Garis putus-putus merah pada n = 15 menandai titik di mana *cumulative explained variance* telah mencapai sekitar 90–95%, yang berarti sebagian besar informasi penting dalam data sudah terwakili oleh 15 komponen pertama. Setelah titik ini, penambahan komponen hanya memberikan peningkatan variansi yang sangat kecil (*diminishing returns*), sehingga tidak sebanding dengan bertambahnya kompleksitas model. Oleh karena itu, n = 15 dipilih sebagai titik kompromi optimal antara mempertahankan informasi yang cukup representatif dan mengurangi dimensi data, sehingga PCA tetap efektif tanpa membawa noise atau dimensi yang kurang informatif.

# BAB 5

# PERBANDINGAN HASIL PREDIKSI

A. Perbandingan Implementasi *Scratch* dan Scikit Decision Tree Learning

Berikut merupakan hasil classification report dari Decision Tree Learning From Scratch dan KNN Scikit-learn,

1. Decision Tree Learning From Scratch

```
ROC-AUC: 0.5482255699150554
              precision    recall  f1-score   support

           0       0.86      0.95      0.91     17174
           1       0.22      0.08      0.12      2826

    accuracy                           0.83     20000
   macro avg       0.54      0.52      0.51     20000
weighted avg       0.77      0.83      0.79     20000
```

2. Decision Tree Learning Scikit-learn

```
              precision    recall  f1-score   support

           0       0.87      0.58      0.70     17174
           1       0.16      0.50      0.24      2826

    accuracy                           0.57     20000
   macro avg       0.52      0.54      0.47     20000
weighted avg       0.77      0.57      0.63     20000

AUC: 0.5561371820551005
```

Berdasarkan hasil evaluasi, Decision Tree Learning from scratch dan Decision Tree Scikit-learn menunjukkan perbedaan strategi prediksi yang cukup kontras, khususnya dalam menangani ketidakseimbangan kelas. Implementasi from scratch menghasilkan akurasi tinggi (0.83) dan ROC-AUC ≈ 0.548, namun memiliki recall kelas 1 yang rendah (0.08). Hal ini mengindikasikan bahwa model cenderung sangat bias terhadap kelas mayoritas dan lebih fokus memaksimalkan ketepatan prediksi kelas 0, sehingga banyak kasus kelas minoritas yang tidak terdeteksi.

Sebaliknya, Decision Tree Scikit-learn memiliki ROC-AUC yang sedikit lebih tinggi (≈ 0.556) dan recall kelas 1 yang jauh lebih baik (0.50), menunjukkan kemampuan yang lebih efektif dalam mengenali kelas minoritas. Namun, peningkatan sensitivitas ini dibayar dengan penurunan akurasi keseluruhan (0.57) serta precision kelas 1 yang tetap rendah, yang menunjukkan meningkatnya jumlah false positive. Perbedaan ini kemungkinan disebabkan oleh optimasi internal Scikit-learn seperti pemilihan split yang lebih matang,

penanganan impurity (Gini/Entropy), serta mekanisme pembentukan pohon yang lebih agresif dalam memisahkan kelas.

Secara keseluruhan, Decision Tree from scratch unggul dalam stabilitas dan akurasi global, sementara Decision Tree Scikit-learn lebih unggul dalam mendeteksi kelas minoritas. Hal ini menegaskan bahwa pemilihan implementasi perlu disesuaikan dengan tujuan analisis, apakah lebih mengutamakan akurasi total atau kemampuan mendeteksi kasus positif pada data yang tidak seimbang.

B. Perbandingan Implementasi *Scratch* dan Scikit Logistic Regression

Berikut merupakan hasil classification report dari KNN From Scratch dan KNN Scikit-learn,

3. Logistic Regression From Scratch

```
ROC-AUC (Validation): 0.563826732108997
              precision    recall  f1-score   support

           0       0.87      0.91      0.89     17174
           1       0.22      0.15      0.18      2826

    accuracy                           0.80     20000
   macro avg       0.54      0.53      0.53     20000
weighted avg       0.78      0.80      0.79     20000
```

4. Logistic Regression Scikit-learn

```
              precision    recall  f1-score   support

           0       0.88      0.57      0.69     17174
           1       0.17      0.54      0.26      2826

    accuracy                           0.56     20000
   macro avg       0.53      0.55      0.47     20000
weighted avg       0.78      0.56      0.63     20000

AUC: 0.5764936562461187
```

Berdasarkan hasil evaluasi, Logistic Regression from scratch dan Logistic Regression Scikit-learn menunjukkan perbedaan karakteristik prediksi yang cukup jelas, khususnya dalam mendeteksi kelas minoritas (label 1 / fraud). Model from scratch menghasilkan akurasi tinggi (0.80) dan ROC-AUC ≈ 0.564, dengan recall kelas 1 sebesar 0.15. Hal ini menunjukkan bahwa model cenderung lebih konservatif dan masih bias ke kelas mayoritas, namun tetap mampu menangkap sebagian pola kelas positif dengan tingkat stabilitas prediksi yang cukup baik.

Sebaliknya, Logistic Regression Scikit-learn menghasilkan ROC-AUC yang sedikit lebih tinggi (≈ 0.576) dan recall kelas 1 yang jauh lebih besar (0.54), menandakan kemampuan yang lebih baik dalam mendeteksi kasus positif. Namun, peningkatan recall tersebut diikuti oleh penurunan akurasi secara signifikan (0.56) serta precision kelas 1 yang rendah, yang mengindikasikan meningkatnya jumlah *false positive*. Perbedaan ini kemungkinan disebabkan oleh optimasi internal Scikit-learn yang lebih matang, serta perbedaan threshold keputusan yang mempengaruhi trade-off antara precision dan recall.

Secara keseluruhan, implementasi from scratch lebih unggul dalam stabilitas dan akurasi global, sementara implementasi Scikit-learn lebih efektif dalam menangkap kelas minoritas. Hal ini menegaskan bahwa pemilihan implementasi Logistic Regression perlu disesuaikan dengan tujuan analisis, apakah

lebih memprioritaskan akurasi keseluruhan atau sensitivitas terhadap kelas positif pada data yang tidak seimbang.

C. Perbandingan Implementasi *Scratch* dan Scikit KNN

Berikut merupakan hasil classification report dari KNN From Scratch dan KNN Scikit-learn,

5. KNN From Scratch

```
ROC-AUC: 0.5415673542792636
              precision    recall  f1-score   support

           0       0.86      0.99      0.92     17174
           1       0.27      0.03      0.06      2826

    accuracy                           0.85     20000
   macro avg       0.57      0.51      0.49     20000
weighted avg       0.78      0.85      0.80     20000
```

6. KNN Scikit-learn

```
              precision    recall  f1-score   support

           0       0.86      1.00      0.92     17174
           1       0.75      0.00      0.00      2826

    accuracy                           0.86     20000
   macro avg       0.80      0.50      0.46     20000
weighted avg       0.84      0.86      0.79     20000

AUC: 0.5412337408932395
```

Berdasarkan hasil evaluasi, KNN from scratch dan KNN Scikit-learn memiliki performa yang hampir setara dari sisi ROC-AUC (≈ 0.542 vs ≈ 0.541), yang menunjukkan bahwa kemampuan keduanya dalam memisahkan kelas secara umum relatif sama. Namun, terdapat perbedaan penting pada pola prediksi kelas minoritas (label 1).

Pada KNN from scratch, model masih mampu mendeteksi sebagian kecil data kelas 1 dengan recall sebesar 0.03, meskipun nilainya sangat rendah. Hal ini terjadi karena implementasi from scratch menggunakan pendekatan probabilistik sederhana berupa rata-rata label tetangga terdekat yang kemudian dibandingkan dengan threshold tertentu, sehingga masih memungkinkan prediksi positif meskipun data sangat tidak seimbang. Konsekuensinya, precision kelas 1 tetap rendah dan model tetap sangat bias ke kelas mayoritas.

Sementara itu, KNN Scikit-learn menghasilkan akurasi sedikit lebih tinggi (0.86), namun recall kelas 1 bernilai 0.00, yang berarti seluruh data hampir selalu diprediksi sebagai kelas mayoritas. Kondisi ini mengindikasikan bahwa kombinasi nilai k, distribusi data yang timpang, serta mekanisme voting mayoritas pada Scikit-learn membuat model menjadi sangat konservatif terhadap kelas minoritas, sehingga gagal mengenali kasus positif sama sekali.

Secara keseluruhan, perbedaan performa kedua model tidak berasal dari algoritma KNN itu sendiri, melainkan dari strategi pengambilan keputusan dan karakteristik data yang tidak seimbang. KNN from scratch sedikit lebih fleksibel dalam memprediksi kelas minoritas, sedangkan KNN Scikit-learn lebih stabil dalam memaksimalkan akurasi global, tetapi mengorbankan kemampuan deteksi kelas positif.

# PEMBAGIAN TUGAS

| Nama - NIM | Bagian yang dikerjakan |
|---|---|
| Darren Mansyl - 18223001 | - Algoritma Decision Tree Learning<br>- Algoritma Logistic Regression |
| Timothy Marvine - 18223021 | - Algoritma KNN<br>- Algoritma Logistic Regression |
| Amudi Purba - 18223049 | - Cleaning and Pre-processing<br>- Compile Processing Pipeline |

# REFERENSI


Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval*. Cambridge University Press.

DQLab. (n.d.). *Apa itu decision tree di machine learning model*. DQLab. https://dqlab.id/apa-itu-decision-tree-di-machine-learning-model

GeeksforGeeks. (n.d.). *Understanding logistic regression*. GeeksforGeeks. https://www.geeksforgeeks.org/machine-learning/understanding-logistic-regression/

GeeksforGeeks. *Iterative Dichotomiser 3 (ID3) Algorithm From Scratch*. GeeksforGeeks. Last updated 06 Aug, 2025. Available at: https://www.geeksforgeeks.org/machine-learning/iterative-dichotomiser-3-id3-algorithm-from-scratch/ GeeksforGeeks

GeeksforGeeks. *CART (Classification And Regression Tree) in Machine Learning*. GeeksforGeeks. Last updated 04 Dec, 2025. Available at: https://www.geeksforgeeks.org/machine-learning/cart-classification-and-regression-tree-in-machine-learning/ GeeksforGeeks

GeeksforGeeks. *Decision Tree Algorithms*. GeeksforGeeks. Last updated 08 Nov, 2025. Available at: https://www.geeksforgeeks.org/machine-learning/decision-tree-algorithms/ GeeksforGeeks

GeeksforGeeks. *Support Vector Machine (SVM) Algorithm*. GeeksforGeeks. Last updated 13 Nov, 2025. Available at: https://www.geeksforgeeks.org/machine-learning/support-vector-machine-algorithm/ GeeksforGeeks

GeeksforGeeks. *Understanding Logistic Regression*. GeeksforGeeks. (Date not shown in preview; typically the “Last Updated” date appears at the top of the page). Available at: https://www.geeksforgeeks.org/machine-learning/understanding-logistic-regression/ GeeksforGeeks