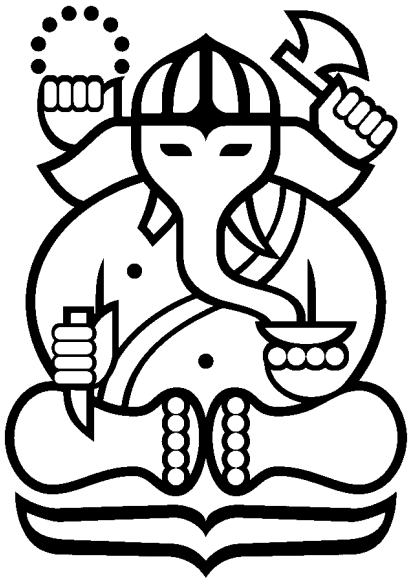
LAPORAN TUGAS BESAR 2

IF3170 Intelegensi Artifisial

Implementasi Algoritma Pembelajaran Mesin



Disusun oleh:

Aurelius Justin Philo Fanjaya (13522020)

Bagas Sambega Rosyada (13522071)

Fedrianz Dharma (13522090)

Raden Francisco Trianto B. (13522091)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

BANDUNG

2024

DAFTAR ISI

BAB I
IMPLEMENTASI ALGORITMA

Algoritma-algoritma yang kami implementasikan terdiri dari K-Nearest Neighbor (KNN), Naive-Bayes, dan ID3. Berikut adalah penjelasan dari masing-masing algoritma:

1. Implementasi KNN

K-Nearest Neighbor adalah algoritma Machine Learning yang termasuk ke dalam Supervised Learning, menyimpan semua data latih, dan lazy learner. Algoritma KNN tidak memiliki hipotesis yang artinya kelas dari data baru akan diprediksi menggunakan data latih yang tersimpan. Algoritma KNN akan menghitung 'distance' dari data baru ke semua data latih yang tersimpan. Setelah itu, KNN akan mencari *k* data yang paling mirip atau memiliki 'distance' terkecil (*k nearest neighbor*). Beberapa contoh fungsi 'distance' yang dapat digunakan adalah Euclidean, Manhattan, dan Minkowski. Setelah itu, algoritma KNN akan mencari mayoritas kelas dari *k nearest neighbor*. Mayoritas kelas itulah yang akan menjadi hasil prediksi kelas untuk data baru.

Kelebihan dan Kekurangan dari KNN:

- Kelebihan
  - Menggunakan aproksimasi sehingga lebih sederhana untuk target yang membutuhkan fungsi yang kompleks.
- Kekurangan
  - Biaya untuk mengklasifikasikan data baru sangat mahal.
  - Menggunakan semua fitur → fungsi target hanya membutuhkan beberapa fitur.

Kelas KNN

- Constructor

```
class KNN:
    def __init__(self, k, metric, p = 3):
        self.k = k
        self.metric = metric
        self.p = p
```

Method untuk membuat kelas KNN dengan atribut:

- k → jumlah *k neighbor*
- metric → Fungsi 'distance' yang ingin digunakan
- p → pangkat yang digunakan untuk fungsi 'distance' Minkowski

- Method fit

```
def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train
```

Method untuk menyimpan data latih dengan parameter:

  - X_train → fitur dari data latih
  - y_train → target dari data latih

- Fungsi ‘distance’

```
def euclidean_distance(self, a, b):
    return np.sqrt(np.sum((a-b)**2))
def manhattan_distance(self, a, b):
    return np.sum(np.abs(a-b))
def minkowski_distance(self, a, b):
    return np.sum(np.abs(a-b)**self.p)**(1/self.p)
```

Beberapa fungsi yang dapat digunakan untuk mengevaluasi ‘distance’ antara data latih dengan data baru.

- Method predict_single

```
def predict_single(self, X):
    if self.metric == 'euclidean':
        distances = [self.euclidean_distance(x_train, X) for x_train in self.X_train]
    elif self.metric == 'manhattan':
        distances = [self.manhattan_distance(x_train, X) for x_train in self.X_train]
    elif self.metric == 'minkowski':
        distances = [self.minkowski_distance(x_train, X) for x_train in self.X_train]
    else:
        raise ValueError('Invalid metric')

    # sort by distance
    distance_sorted = np.argsort(distances)

    # ambil k elemen
    index_of_k_element = distance_sorted[:self.k]
    k_element_labels = [self.y_train[i] for i in index_of_k_element]

    # cari mayoritas label pada k elemen
    most_common = Counter(k_element_labels).most_common(1)
    return most_common[0][0]
```

Method untuk menghitung ‘distance’ antara data baru dengan semua data latih dengan menggunakan salah satu fungsi ‘distance’ yang ada. Setelah itu akan

dihitung k baris data yang memiliki 'distance' yang paling kecil. Kemudian method akan mengembalikan mayoritas dari kelas dari *k baris* data tersebut.

- Method predict

```
def predict(self, X):
    predictions = [self.predict_single(x) for x in X]
    return np.array(predictions)
```

Method ini digunakan untuk memprediksi kelas dari semua data baru (X) dengan menggunakan bantuan method *predict_single*. Method akan mengembalikan hasil array hasil prediksi kelas semua data baru.

2. Implementasi Naive-Bayes

Model Naive Bayes merupakan algoritma perhitungan probabilitas bersyarat dari Naive Bayes. Algoritma ini termasuk ke dalam algoritma Supervised Learning, karena kelas target sudah terdefinisi dan digunakan sebagai label. Tidak seperti algoritma KNN, algoritma Naive Bayes dapat dilatih sekali, kemudian model dapat digunakan berulang kali setelahnya.

Algoritma ini menghitung probabilitas setiap kelas target, dan peluang bersyarat fitur jika diberikan kelas target. Kemudian, jika diberikan sebuah data yang akan diuji, maka program akan mengalikan peluang kelas target, dengan seluruh fitur yang bersesuaian dengan kelas tersebut. Jika terdapat 5 kelas target, maka akan ditemukan total 5 nilai peluang, dan kelas hasil prediksi yang akan diambil adalah kelas dengan nilai probabilitas tertinggi. Secara matematis, fungsi predict dirumuskan sebagai:

$$P(target \mid f1, f2, \ldots, fn) = P(target) \cdot P(f1 \mid target) \cdot P(f2 \mid target) \cdot \ldots \cdot P(fn \mid target)$$

Kelebihan dari algoritma Naive-Bayes sebagai berikut,

- Ukuran model kecil dan dapat digunakan kembali
- Sangat bagus untuk data yang didominasi oleh data kategorikal, karena penghitungan frekuensi data unik menjadi lebih akurat
- Mudah untuk dianalisis karena menggunakan nilai probabilitas bersyarat

Kekurangan dari algoritma Naive-Bayes adalah,

- Kurang bagus untuk data numerik, karena data harus dibuat menjadi kategorikal dengan *encoding* atau *binning*

- Penghitungan berdasarkan probabilitas, yang berarti jika suatu nilai data mendominasi, maka nilai data tersebut akan mendominasi
- Tidak mempertimbangkan korelasi antar-fitur dan bobot atau *information gain* dari fitur-fiturnya

Implementasi kode Kelas Naive-Bayes

- Constructor

```
class NaiveBayes:
    def __init__(self, target_column: str, bin_number: int = 20):
        """
        :param target_column: column dataframe target. assume only one column
        :param bin_number: number of bins for numerical columns
        """
        self.data_train: pd.DataFrame = None
        self.classes_ = None
        self.categorical_columns: List[str] = []
        self.numerical_columns: List[str] = []
        self.data_train_freq: Dict[str, Dict[str, Dict[
            str, int]]] = {}  # [key: column name, value: [key: target unique values, value: [key: unique value of column, value: frequency of unique value for each column]]]
        self.data_train_prob: Dict[str, Dict[str, Dict[
            str, float]]] = {}  # [key: column name, value: [key: target unique values, value: [key: unique value of column, value: probability of unique value for each column]]]
        self.target_column: str = target_column
        self.target_freq: Dict[str, int] = {}  # [key: target unique values, value: frequency of target unique values]
        self.target_prob: Dict[
            str, float] = {}  # [key: target unique values, value: probability of target unique values]
        self.bin_number: int = bin_number
```

- Calculate_frequency_column
  Menghitung frekuensi data untuk setiap nilai unik atau frekuensi setiap *binning* untuk fitur numerikal.

```
def calculate_frequency_column(self):
        for col in self.categorical_columns:
            self.data_train_freq[col] = {}
            for target in self.data_train[self.target_column].unique():
                freq = {}
                for value in self.data_train[col].unique():
                    freq[value] = self.data_train[
                        (self.data_train[self.target_column] == target) & (self.data_train[col] == value)].shape[0]
                self.data_train_freq[col][target] = freq

        for col in self.numerical_columns:
            self.data_train_freq[col] = {}
            # Create binned column
            self.data_train[f'{col}_binned'] = pd.qcut(self.data_train[col], q=self.bin_number, duplicates='drop')

            for target in self.data_train[self.target_column].unique():
                freq = {}
                for value in self.data_train[f'{col}_binned'].unique():
                    freq[str(value)] = self.data_train[(self.data_train[self.target_column] == target) & (
                            self.data_train[f'{col}_binned'] == value)].shape[0]
                self.data_train_freq[col][target] = freq

        for target in self.data_train[self.target_column].unique():
            self.target_freq[target] = self.data_train[self.data_train[self.target_column] == target].shape[0]
```

- Calculate_probability_column
  Menghitung probabilitas nilai unik setiap fitur atau *range binned* untuk fitur numerikal

```
def calculate_probability_column(self):
        """
        Calculate probability based on frequency of each column, relative to the target column
        """
        for col in self.categorical_columns:
            self.data_train_prob[col] = {}
            for target in self.unique_target:
                prob = {}
                for value in self.data_train[col].unique():
                    prob[value] = self.data_train_freq[col][target][value] / self.number_of_rows
                self.data_train_prob[col][target] = prob

        for col in self.numerical_columns:
            self.data_train_prob[col] = {}
            for target in self.unique_target:
                prob = {}
                for value in self.data_train[f'{col}_binned'].unique():
                    prob[str(value)] = self.data_train_freq[col][target][str(value)] / self.number_of_rows
                self.data_train_prob[col][target] = prob

        for target in self.unique_target:
            self.target_prob[target] = self.target_freq[target] / self.number_of_rows
```

- Fit

  Melakukan pelatihan berupa penghitungan frekuensi dan probabilitas setiap nilai unik fitur dan nilai unik target. Fungsi ini juga menentukan kolom mana yang merupakan kolom kategorikal dan kolom numerikal

```
def fit(self, X_train, y):
        """
        Train the model
        :param X_train: training data
        :param y: target of training data
        """
        # Set classes, fix X_train
        self.classes_ = np.unique(y)
        if not isinstance(X_train, pd.DataFrame):
            X_train = pd.DataFrame(X_train)

        # Fix y datatype
        if isinstance(y, np.ndarray):
            y = pd.Series(y, name=self.target_column)
        elif not isinstance(y, pd.Series):
            raise TypeError("`y` must be a NumPy array or pandas Series")

        # Combine data
        self.data_train = pd.concat([X_train, y], axis=1)
        self.define_column_type()
        self.unique_target = self.data_train[self.target_column].unique()
        self.calculate_frequency_column()
        self.calculate_probability_column()
        return self
```

- Predict

  Fungsi untuk menghasilkan prediksi dari data tes yang diberikan. Fungsi ini menghitung probabilitas yang bersesuaian dengan nilai fitur yang diberikan, dan

mengalikannya dengan nilai probabilitas target, kemudian mencari nilai probabilitas terbesar di antara semua probabilitas target tersebut.

```python
def predict_single(self, data_test: pd.DataFrame):
        """
        Predict the data. Input is just one row.
        """
        result = {}
        for target in self.data_train[self.target_column].unique():
            prob = self.target_prob[target]  # Start with the prior probability of the target class

            for col in self.categorical_columns:
                column_probabilities = self.data_train_prob[col][target]
                value = data_test[col]

                prob *= column_probabilities.get(value, 1e-6)

            for col in self.numerical_columns:
                value = data_test[col]
                column_probabilities = self.data_train_prob[col][target]

                # Find the correct interval
                matching_interval = None
                for interval_str, interval_prob in column_probabilities.items():
                    if interval_str == 'nan':
                        continue

                    # Parse interval string
                    interval_str_num = interval_str[1:-1]
                    left, right = map(float, interval_str_num.split(','))

                    # Check if value fits in this interval
                    if left < value <= right:
                        matching_interval = interval_str
                        prob *= interval_prob
                        break

                # Use a fallback probability if no matching interval is found
                if matching_interval is None:
                    prob *= min(column_probabilities.values(), default=1e-6)

            result[target] = prob

        # Return the class with the highest probability
        return max(result, key=result.get)

    def predict(self, data_test: Union[pd.DataFrame, np.array]):
        """
        Predict the data. Input can be a single row or multiple rows.
        :param data_test: data to predict
        :return: numpy array of predicted values
        """
        if not isinstance(data_test, pd.DataFrame):
            data_test = pd.DataFrame(data_test)
        if len(data_test.shape) == 1 or data_test.shape[0] == 1:
            # If single row
            return np.array([self.predict_single(data_test)])
        else:
            # For multiple rows, iterate for each row
            return np.array([self.predict_single(row) for _, row in data_test.iterrows()])
```

- Save_model dan load_model
  Fungsi save_model berfungsi untuk menyimpan model dalam format .pkl dan fungsi load_model digunakan untuk membuat ulang model dari file model yang disimpan.

```
def save_model(self, filename: str = 'nb_model.pkl'):
        """
        Save the model to a file
        """
        joblib.dump(self, filename)
        print(f"Model saved as {filename}")

    @staticmethod
    def load_model(filename: str = 'nb_model.pkl'):
        """
        Load the model from a file
        """
        return joblib.load(filename)
```

3. Implementasi ID3

Algoritma ID3 yang digunakan mengikuti implementasi ID3 pada slide perkuliahan dengan penjelasan sebagai berikut.

- Kelas ID3

```
class ID3:
    def __init__(self):
        self.tree = None
        self.discretization_thresholds = {}
        self.label_encoder = {}
        self.label_decoder = {}
```

self.tree adalah decision tree yang akan dibangun. Self.discretization_thresholds adalah *threshold* untuk nilai-nilai numerik yang akan dilakukan discretization. self.label_encoder dan self.label_decoder digunakan untuk menghandle label yang non-numerik.

- Discretize_continuous_attribute

```python
def discretize_continuous_attribute(self, examples, A):
    sorted_indices = np.argsort(examples[:, A])
    X_sorted = examples[sorted_indices]
    y_sorted = examples[:, -1][sorted_indices]

    breakpoints = []
    for i in range(1, len(X_sorted)):
        if y_sorted[i] != y_sorted[i - 1]:
            midpoint = (X_sorted[i, A] + X_sorted[i - 1, A]) / 2
            breakpoints.append(midpoint)

    best_gain = -1
    best_threshold = None
    for threshold in breakpoints:
        left_mask = X_sorted[:, A] < threshold
        right_mask = ~left_mask

        y_left = y_sorted[left_mask]
        y_right = y_sorted[right_mask]

        left_examples = X_sorted[left_mask]
        right_examples = X_sorted[right_mask]
        left_full = np.column_stack([left_examples, y_left])
        right_full = np.column_stack([right_examples, y_right])

        gain = self.information_gain_for_split(y_sorted, left_full, right_full)
        if gain > best_gain:
            best_gain = gain
            best_threshold = threshold

    return best_threshold
```

method ini digunakan untuk melakukan discretization (mencari threshold) terhadap atribut *continuous* dengan cara melakukan sorting terhadap atribut tersebut, kemudian mencari threshold terbaik dengan mengambil threshold dengan nilai information_gain terbaik dari breakpoint (titik di mana label dari 2 row bersebelahan berbeda) yang ditemukan.

- Fungsi-fungsi helper (sesuai slide perkuliahan)

```python
def plurality_value(self, examples):
    most_common_encoded = Counter(examples[:, -1]).most_common(1)[0][0]
    return self.label_decoder[most_common_encoded]

def check_all_example_same(self, examples):
    return np.all(examples[:, -1] == examples[0, -1])

def entropy(self, S):
    y = S[:, -1]
    _, counts = np.unique(y, return_counts=True)
    probabilities = counts / len(y)
    return np.sum([-p * np.log2(p) for p in probabilities])

def information_gain(self, S, A):
    # Hitung Entropy(S)
    entropy_S = self.entropy(S)

    # Hitung information gain
    Xa = S[:, A]
    values_A, len_Sv = np.unique(Xa, return_counts=True)
    len_S = len(S)

    return entropy_S - sum(
        (len_Sv[i] / len_S) * self.entropy(S[Xa == v]) for i, v in enumerate(values_A)
    )

def split_information(self, S, A):
    Xa = S[:, A]
    Si_values, len_Si = np.unique(Xa, return_counts=True)
    len_S = len(S)
    ratio_Si_S = len_Si / len_S
    return -np.sum(ratio * np.log2(ratio) for ratio in ratio_Si_S)

def gain_ratio(self, S, A):
    return self.information_gain(S, A) / self.split_information(S, A)
```

```python
def importance(self, a, examples, metric):
    if metric == "information_gain":
        return self.information_gain(examples, a)
    elif metric == "gain_ratio":
        return self.gain_ratio(examples, a)
    return -1

def argmax(self, examples, attributes):
    best_value = -1
    best_attribute = None
    for a in attributes:
        metric = "information_gain"
        if a in self.continuous_attributes:
            if a not in self.discretization_thresholds:
                self.discretization_thresholds[a] = self.discretize_continuous_attribute(examples, a)

            threshold = self.discretization_thresholds[a]
            binary_examples = examples.copy()
            binary_examples[:, a] = (binary_examples[:, a] < threshold).astype(int)
            value = self.importance(a, binary_examples, metric)
        else:
            value = self.importance(a, examples, metric)

        if value > best_value:
            best_value = value
            best_attribute = a
    return best_attribute
```

- decision_tree_learning

```
def decision_tree_learning(self, examples, attributes, parent_examples):
    if len(examples) == 0:
        return self.plurality_value(parent_examples)
    elif self.check_all_example_same(examples):
        return self.label_decoder[examples[0, -1]]
    elif len(attributes) == 0:
        return self.plurality_value(examples)
    else:
        A = self.argmax(examples, attributes)
        tree = {A: {}}

        if A in self.continuous_attributes:
            threshold = self.discretization_thresholds[A]
            XA = (examples[:, A] < threshold).astype(int)
        else:
            XA = examples[:, A]

        for vk in np.unique(XA):
            if A in self.continuous_attributes:
                exs = examples[XA == vk]
            else:
                exs = examples[XA == vk]

            subtree = self.decision_tree_learning(exs, [a for a in attributes if a != A], examples)
            tree[A][vk] = subtree
    return tree
```

method ini adalah fungsi utama rekursif pembentuk tree sesuai dengan slide perkuliahan, hanya saja sudah dimodifikasi untuk handle discretization. Fungsi ini akan dipanggil pada proses fitting.

- fit

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

    encoded_labels = self._encode_labels(y_train)

    self.continuous_attributes = [
        i for i in range(X_train.shape[1])
        if np.issubdtype(X_train[:, i].dtype, np.number)
    ]

    self.discretization_thresholds = {}

    examples = np.column_stack([X_train, encoded_labels])

    for attr in self.continuous_attributes:
        self.discretization_thresholds[attr] = self.discretize_continuous_attribute(examples, attr)

    self.default_class = Counter(encoded_labels).most_common(1)[0][0]

    attributes = list(range(X_train.shape[1]))

    self.tree = self.decision_tree_learning(examples, attributes, examples)
    return self
```

fungsi ini melakukan encoding terlebih dahulu terhadap label agar dapat diproses. Kemudian, dilakukan pencarian threshold untuk discretization atribut-atribut numerik. Setelah itu, dihitung default class sebagai fallback untuk untuk kasus-kasus khusus. Setelah itu baru dibentuk tree secara rekursif dengan decision_tree_learning(examples, attributes, examples).

- predict & predict_single

```
def predict_single(self, X, tree=None):
    if tree is None:
        tree = self.tree

    # Base case: Leaf (label)
    if not isinstance(tree, dict):
        return tree

    attribute = next(iter(tree))

    if attribute in self.continuous_attributes:
        threshold = self.discretization_thresholds[attribute]
        value = int(X[attribute] < threshold)
    else:
        value = X[attribute]

    if value not in tree[attribute]:
        return self.label_decoder[self.default_class]

    return self.predict_single(X, tree[attribute][value])

def predict(self, X):
    X = np.array(X)
    predictions = [self.predict_single(x) for x in X]
    return np.array(predictions)
```

Method predict_single melakukan penelusuran decision tree yang sudah dibuat untuk melakukan prediction terhadap sebuah baris data. Method predict melakukan predict_single untuk setiap baris data pada masukan X.

- save_model dan load_model

```python
def save_model(self, filepath):
    with open(filepath, 'wb') as f:
        pickle.dump(self, f)
    print(f"Model saved to {filepath}")

@classmethod
def load_model(cls, filepath):
    with open(filepath, 'rb') as f:
        model = pickle.load(f)
    print(f"Model loaded from {filepath}")
    return model
```

Fungsi untuk melakukan save (dump) model dan load model. Menggunakan library pickle.

BAB II
DATA CLEANING AND PREPROCESSING

1. Handling Missing Data

Jumlah *missing data* pada tiap *feature* adalah > 5% dari total data sehingga tidak memiliki dampak yang signifikan. Terdapat kolom yang memang sengaja memiliki nilai kosong yaitu state dan service dimana memang salah satu nilainya dapat bernilai kosong.

Namun untuk *feature* yang lain kita harus mengatasinya. Jika *missing data* tersebut dibuang, kita dapat kehilangan informasi yang mungkin saja berharga. Oleh karena itu, untuk menangani *missing data*, kita menggunakan Data Imputation (pengisian data) dengan *library* SimpleImputer. *Missing data* untuk feature *numerical* akan *diimpute* dengan nilai Mean, sedangkan *missing data* untuk feature categorical akan *diimpute* dengan nilai Mode.

2. Dealing with Outliers

Untuk menangani outliers, kita menggunakan Isolation Forest untuk fitur numerikal. Isolation Forest adalah *tree-based model* yang melakukan deteksi anomali. Isolation Forest ini memang terkenal untuk mengatasi data hasil network. Isolation Forest akan memilih *feature numerical* secara random dan melakukan *partitioning* pada *feature* tersebut dengan memilih suatu nilai di antara nilai maksimum dan minimum pada feature tersebut. Jumlah partitioning yang dibutuhkan akan dirata-ratakan sebagai nilai normal. Data *outlier* akan menghasilkan jumlah *partitioning* yang lebih sedikit sehingga kita dapat memisahkan data normal dan data outlier.

Pada fitur kategorikal akan dicari frekuensi dari tiap kategori pada feature tersebut. Kategori yang kurang dari suatu nilai *threshold* akan dianggap sebagai outlier dan dihapus.

3. Remove Duplicates

Data duplikat dapat mempengaruhi data integrity yang dapat menyebabkan analisis yang tidak akurat, bias, dan overfitting. Selain itu, data duplikat juga akan meningkatkan jumlah resources yang harus digunakan untuk melatih model. Data yang duplikat akan dihapus dengan menggunakan fungsi *drop_duplicates* yang tersedia pada Pandas DataFrame.

4. Data Validation

Melakukan pengecekan pada feature-feature apakah nilai-nilai yang ada sudah sesuai dengan range nilai yang valid untuk feature tersebut. Pada feature ‘is_ftp_login’ dan ‘is_sm_ips_ports’ dilakukan pengecekan untuk memastikan bahwa nilai pada feature tersebut hanyalah 0 atau 1. Untuk feature yang lain tidak kami validasi. Hal ini dikarenakan kami masih kurang mengenal bidang security dengan baik, sehingga kami tidak dapat membataskan nilai apa saja yang dapat dibilang valid untuk feature tersebut.

5. Feature Engineering

Melakukan feature engineering dengan menghapus data dengan *low variance feature*. Karena data dengan *low variance feature* hanya memiliki dampak yang kecil terhadap model. Selain itu juga, akan dilakukan penghapusan pada feature yang memiliki korelasi yang tinggi terhadap feature lain. Karena feature-feature dengan korelasi tinggi dapat menyebabkan multicollinearity yang dapat memperburuk performa model.

6. Feature Scaling

Melakukan *scaling* atau normalisasi *range* data sehingga tidak ada data yang terlalu dominan atau melebihi skala tertentu. Feature scaling bertujuan untuk memastikan bahwa fitur dengan skala yang berbeda tidak mendominasi mesin modelling yang sensitif terhadap ukuran fitur, sehingga meningkatkan performa dan efisiensi model.

Feature scaling digunakan terutama pada fitur numerikal, karena data kontinu seringkali memiliki standar deviasi yang tinggi sehingga kepadatan/bobot data tidak terdistribusi secara baik. Untuk mengatasi masalah ini, feature scaling yang digunakan adalah MinMaxScaler, yaitu algoritma untuk mengubah data nilai menjadi data dengan *range* tertentu. Metode ini digunakan karena skala setiap fitur berbeda-beda, sehingga perlu dinormalisasi untuk menghindari hasil perhitungan setiap fitur yang berbeda.

7. Feature Encoding

Proses mengubah data kategori (categorical data) menjadi representasi numerik agar dapat digunakan oleh algoritma pembelajaran mesin yang mengandalkan perhitungan data numerik, misalnya KNN. Pada program ini, encoding yang digunakan adalah One Hot Encoding, yaitu merepresentasikan setiap *unique values* di data kategorikal menjadi vektor biner 0 dan 1. One Hot Encoding digunakan karena tidak ada korelasi yang terlalu berpengaruh antar-fitur dan tidak ada pembobotan untuk setiap fiturnya.

8. Handling Imbalanced Dataset

Untuk mengatasi dataset yang tidak terdistribusi merata, program menggunakan Stratified K-Fold. Stratified K-Fold memastikan bahwa proporsi kelas target di setiap fold serupa dengan proporsi kelas pada dataset asli.

Dalam proses ini, dataset dibagi menjadi k-bagian secara dan memastikan bahwa distribusi kelas target di setiap *fold* mencerminkan distribusi kelas dataset. Penanganan dataset yang tidak merata langsung dilakukan ketika evaluasi model dilakukan di dalam *pipeline* untuk meningkatkan akurasi setiap model dan memastikan setiap model menggunakan jumlah *fold* yang tidak terlalu kecil atau terlalu besar.

BAB III
HASIL & ANALISIS

A. KNN

Hasil eksekusi algoritma KNN adalah sebagai berikut:

- Sklearn KNeighborsClassifier

```
# KNN Pipeline
knn_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('to_dense', DenseTransformer()),
    ('classifier', KNeighborsClassifier(n_neighbors=10))
])
validate(knn_pipeline, X, Y, 3)
```

✓ 0.2s

```
Scores: [0.36655123 0.37452229 0.39017233]
Mean: 0.3770819517764205
```

- KNN from scratch

```
from knn import KNN
knn_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('to_dense', DenseTransformer()),
    ('classifier', KNN(10, 'minskowski', p=2))
])
validate(knn_pipeline, X, Y, 3)
```

✓ 1m 28.1s

```
Scores: [0.37321011 0.37193023 0.38722178]
Mean: 0.3774540414978302
```

Dari hasil eksekusi tersebut, dapat dilihat bahwa hasil scoring f1_macro dari model Sklearn KNeighborsClassifier tidak jauh berbeda dengan hasil scoring f1_macro dari model KNN yang kami buat dari scratch. Hasil dari model KNN yang kami buat dari scratch sedikit lebih baik dengan score rata-rata 0.3774

dibandingkan dengan model Sklearn KNeighborsClassifier dengan score rata-rata 0.3770.

Kedua model tersebut tidak menghasil score yang jauh berbeda dikarenakan algoritma KNN tidak menghasil hipotesis dan hanya mencari kelas dari data yang tersimpan pada model berdasarkan perhitungan jarak. Oleh karena itu, jika kedua model menggunakan algoritma jarak yang sama, yaitu euclidean, maka hasil scoring kedua model tidak akan jauh berbeda.

B. Naive Bayes

Dari prediksi yang dieksekusi menggunakan Naive Bayes, didapat hasil sebagai berikut,

- SKLearn GaussianNaiveBayes

```
gnb_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('to_dense', DenseTransformer()),
    ('classifier', GaussianNB())
])
validate(gnb_pipeline, X, Y, 2)
```

```
Scores: [0.23806796 0.24594297]
Mean: 0.24200546556152236
```

- NaiveBayes from scratch

```
from NaiveBayes import NaiveBayes

nb_numerical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', MinMaxScaler())
])

# Fit categorical
nb_categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent'))
])

# Call categorical and numerical
nb_preprocessor = ColumnTransformer(
    transformers=[
        ('num', nb_numerical_transformer, numerical_features),
        ('cat', nb_categorical_transformer, categorical_features)
    ]
)

nb_pipeline_2 = Pipeline(steps=[
    ('preprocessor', nb_preprocessor),
    ('classifier', NaiveBayes(target_column=target_col[0], bin_number=30))
])
validate(nb_pipeline_2, X, Y, 2)
```

```
Scores: [0.35824446 0.36421559]
Mean: 0.361230024906681
```

Implementasi *pipeline* NaiveBayes berbeda dengan ID3 dan KNN karena NaiveBayes berjalan lebih baik dengan menggunakan data kategorikal, sehingga *pipeline* NaiveBayes tidak menggunakan *encoding* untuk data kategorikal. Skor NaiveBayes yang diimplementasikan dari *scratch* memiliki performa yang lebih baik, yaitu 0.3612 dibandingkan GaussianNaiveBayes dengan 0.24201.

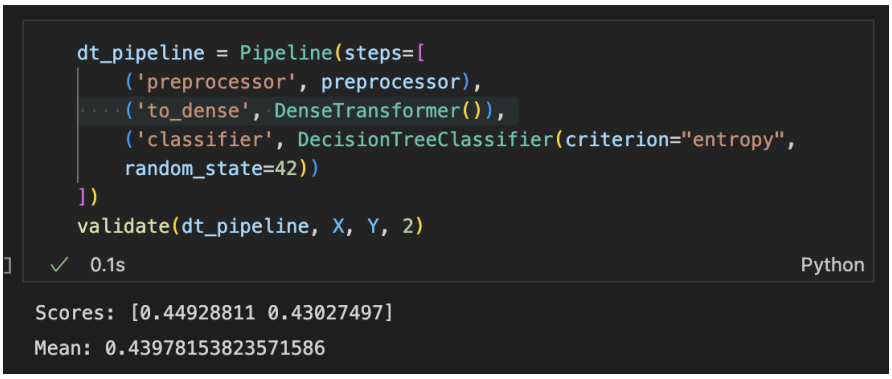
Perbedaan skor ini dikarenakan Gaussian Naive Bayes baik digunakan untuk data kontinu (numerikal), dan karenanya *pipeline* Gaussian Naive Bayes memakai *preprocessor* yang menggunakan *encoder* dan Dense Transformer untuk membentuk *sparse matrix*.

Proses *one-hot encoding* pada Gaussian Naive Bayes untuk mengubah data kategorikal menjadi numerikal mengurangi akurasi program untuk menghitung frekuensi dan probabilitas *unique values* dari fitur kategorikal. Selain itu, proses *binning* data numerikal pada Naive Bayes *from scratch* diatur menjadi 30 *binning* saja, untuk menghindari *underfitting* maupun *overfitting*.

C. ID3

Hasil dari eksekusi yang dilakukan adalah sebagai berikut:

- Sklearn DecisionTreeClassifier

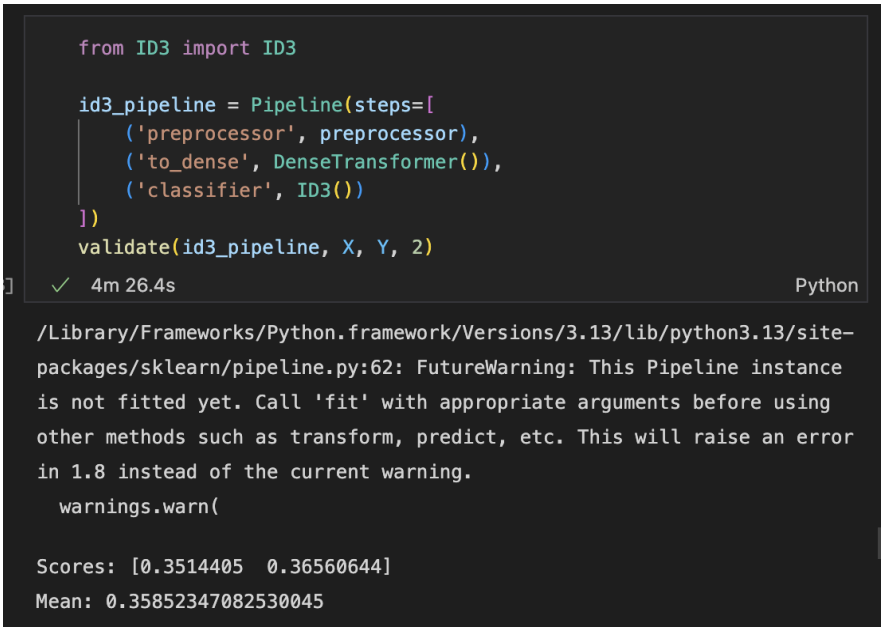


```
dt_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('to_dense', DenseTransformer()),
    ('classifier', DecisionTreeClassifier(criterion="entropy",
    random_state=42))
])
validate(dt_pipeline, X, Y, 2)
```

✓ 0.1s Python

```
Scores: [0.44928811 0.43027497]
Mean: 0.43978153823571586
```

- ID3 From Scratch



```
from ID3 import ID3

id3_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('to_dense', DenseTransformer()),
    ('classifier', ID3())
])
validate(id3_pipeline, X, Y, 2)
```

✓ 4m 26.4s Python

```
/Library/Frameworks/Python.framework/Versions/3.13/lib/python3.13/site-
packages/sklearn/pipeline.py:62: FutureWarning: This Pipeline instance
is not fitted yet. Call 'fit' with appropriate arguments before using
other methods such as transform, predict, etc. This will raise an error
in 1.8 instead of the current warning.
  warnings.warn(

Scores: [0.3514405  0.36560644]
Mean: 0.35852347082530045
```

Dari hasil tersebut, dapat dilihat bahwa hasil scoring f1_macro dari model Sklearn DecisionTreeClassifier secara signifikan lebih baik dibandingkan model ID3 yang kami buat from scratch dalam tugas ini. Score DTL sklearn mendapatkan rata-rata 0.439 sedangkan pada model yang kami kembangkan, kami mendapatkan score rata-rata 0.358.

Hal ini dapat terjadi karena beberapa perbedaan antara implementasi model ID3 yang kami buat dan implementasi DecisionTreeClassifier sklearn, meskipun menggunakan criterion=”entropy”. Perbedaan-perbedaan tersebut antara lain:

1. Discretization

    a. ID3 yang kami implementasikan melakukan discretization dengan menentukan threshold terbaik tiap atribut numerik di awal (saat awal fitting sebelum membentuk tree). Penentuan threshold dilakukan sesuai dengan PPT kuliah, yaitu dengan mencari threshold terbaik (information gain terbaik) dari kandidat-kandidat threshold, yaitu nilai atribut tersebut yang berada di antara 2 label yang berbeda setelah di sorting.
    b. Algoritma DecisionTreeLearning sklearn melakukan discretization dengan splitting secara dinamis ketika pembentukan tree, sehingga dihasilkan hasil yang lebih baik.
2. CART
    a. Model DecisionTreeLearning sklearn menggunakan algoritma CART, bukan ID3, meskipun menggunakan criterion=”entropy”.
    b. Model implementasi kami menggunakan algoritma ID3 biasa.
3. Feature Importance
    a. Model DecisionTreeLearning sklearn mengimplementasikan feature importance, yang mengatur weight atribut ketika splitting.
    b. Model ID3 kami tidak melakukan feature importance sama sekali.

Perbedaan-perbedaan tersebut membuat Model DecisionTreeLearning sklearn lebih teroptimasi, yang dapat dilihat dari *score* f1_macro yang lebih baik.

PEMBAGIAN TUGAS

NIM	Nama	Pembagian Tugas
13522020	Aurelius Justin Philo Fanjaya	● Implementasi Algoritma ID3 from Scratch ● Preprocessing ● Modelling & pipeline
13522071	Bagas Sambega Rosyada	● Implementasi Algoritma Naive Bayes from Scratch ● Pipelining
13522090	Fedrianz Dharma	● Implementasi Algoritma KNN from Scratch ● Data validation dan modelling
13522091	Raden Francisco Trianto B.	● Inisialisasi notebook ● Exploratory Data Analysis ● Preprocessing ● Data validation dan pipelining

REFERENSI

- Slide Perkuliahan IF3170 Intelegensi Artifisial
- Artificial Intelligence: A Modern Approach. Peter Norvig, Stuart Russel.
- Scikit-learn documentation: https://scikit-learn.org/stable/api/index.html