Laporan Tugas Besar 1
IF3170 Inteligensi Artifisial
Implementasi Algoritma Pembelajaran Mesin
Semester I Tahun 2024/2025

Disusun Oleh:
Mohammad Nugraha Eka Prawira (13522001)
Muhammad Al Thariq Fairuz (13522027)
Randy Verdian (13522067)
Emery Fathan Zwageri (13522079)

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2024

Bab 1
Implementasi KNN

KNN.py

```python
import numpy as np
from sklearn.base import BaseEstimator, ClassifierMixin # Import these so we can use cross val score to evaluate our model, we take classifiermixin since we are building a classifier

class KNNeighbours(BaseEstimator, ClassifierMixin): # Inherit from BaseEstimator and ClassifierMixin
    """
    Create an instance of the K-Nearest Neighbors algorithm
    """

    def __init__(self, k: int =3, metrics: str = 'euclidean'):
        """
        Constructor for the K-Nearest Neighbors algorithm
        """
        self.k = k
        self.metrics = metrics

    def fit(self, X : np.ndarray, y : np.ndarray):
        """
        Fit the training data to the model
        """
        self.X_train = X
        self.y_train = y

    def euclidean_distance(self, x1 : np.ndarray, x2 : np.ndarray) -> float:
        """
        Compute the Euclidean distance between two points
        """
        return np.sqrt(np.sum((x1 - x2)**2))

    def manhattan_distance(self, x1 : np.ndarray, x2 : np.ndarray) -> float:
        """
        Compute the Manhattan distance between two points
        """
        return np.sum(np.abs(x1 - x2))

    def minkowski_distance(self, x1 : np.ndarray, x2 : np.ndarray, p : int) -> float    :
        """
        Compute the Minkowski distance between two points
        """
        return np.sum(np.abs(x1 - x2)**p)**(1/p)

    def _predict(self, x : np.ndarray, p : int = 2) -> int:
        """
        Predict the class label for a single sample
        """

        # Compute distances between x and all examples in the training set
        if self.metrics == 'euclidean':
            distances = [self.euclidean_distance(x, x_train) for x_train in self.X_train]
        elif self.metrics == 'manhattan':
            distances = [self.manhattan_distance(x, x_train) for x_train in self.X_train]
        elif self.metrics == 'minkowski':
            distances = [self.minkowski_distance(x, x_train, p) for x_train in self.X_train]
        else:
            raise ValueError('Invalid metrics')

        # Sort by distance and return indices of the first k neighbors
        k_indices = np.argsort(distances)[:self.k] # Return the indices of the k nearest neighbors from the lowest distance

        # Extract the labels of the k nearest neighbor training samples
        k_nearest_labels = [self.y_train[i] for i in k_indices]

        # Return the most common class label
        most_common, counts = np.unique(k_nearest_labels, return_counts=True)
        return most_common[np.argmax(counts)]

    def predict(self, X : np.ndarray, p : int = 2) -> np.ndarray:
        """
        Predict the class labels for a set of samples
        """
        y_pred = [self._predict(x, p) for x in X]
        return np.array(y_pred)
```

Snipped

Kelas KNNeighbours merupakan implementasi algoritma KNN dari *scratch* menggunakan python. Kelas ini menurunkan kelas dari BaseEstimator, ClassifierMixin agar kompatibel dengan cross_val_score dari scikit-learn sebagai fungsi untuk mengukur kemampuan model.

Kelas ini memiliki dua parameter, yaitu jumlah tetangga dan metrik yang digunakan, metrik yang tersedia ada euclidean, manhattan, dan minkowski. Algoritma ini bekerja dengan menghitung jarak antara data yang akan diprediksi dengan semua data yang ada pada *training* data dengan menggunakan metrik yang telah dipilih sebelumnya. Setelah perhitungan jarak dilakukan, akan diambil indeks dari k tetangga terdekat dan kemudian akan dihitung kelas terbanyak yang dimiliki oleh tetangga tersebut, kelas yang paling banyak muncul di antara tetangga-tetangga ini akan menjadi kelas yang diprediksi untuk data baru.

Bab 2
Implementasi Naive-Bayes

GNB.py

```python
import numpy as np
from sklearn.base import BaseEstimator, ClassifierMixin

class GaussianNaiveBayes(BaseEstimator, ClassifierMixin):
    """
    Create an instance of the Gaussian Naive Bayes algorithm
    """

    def fit(self, X : np.ndarray, y : np.ndarray):
        """
        Fit the training data to the model, calculate the mean, variance, and prior for each class
        """

        # Check for all unique values (classes) in the label
        self.classes = np.unique(y)
        self.epsilon = 1e-9   # Prevent division by zero

        # Store the mean, variance, and prior for each class
        self.mean = {}
        self.var = {}
        self.priors = {}

        # Calculate the mean, variance, and prior for each class
        for cls in self.classes:
            X_c = X[y == cls]
            self.mean[cls] = X_c.mean(axis=0)
            self.var[cls] = X_c.var(axis=0)
            self.priors[cls] = X_c.shape[0] / X.shape[0]

    # Calculate the probability of a feature vector x in a given class
    def _pdf(self, cls : int, x : np.ndarray) -> float:
        """
        Calculate the probability of a feature vector x in a given class
        """
        mean = self.mean[cls]
        var = self.var[cls] + self.epsilon
        numerator = np.exp(- (x - mean) ** 2 / (2 * var))
        denominator = np.sqrt(2 * np.pi * var)

        # Error handling: Add epsilon to denominator where it is zero
        denominator = np.where(denominator == 0, self.epsilon, denominator)

        return numerator / denominator

    def predict_instance(self, x : np.ndarray) -> int:
        """
        Predict the class label for a single sample
        """
        posteriors = []

        for cls in self.classes:
            prior = np.log(self.priors[cls])
            pdf_values = self._pdf(cls, x)

            # Adding epsilon before taking the log to avoid log(0)
            pdf_values = np.where(pdf_values == 0, self.epsilon, pdf_values)

            # Calculate the posterior and handle -inf values
            log_pdf_values = np.log(pdf_values)
            log_pdf_values[np.isneginf(log_pdf_values)] = -1e9  # Replace -inf with a very large negative number

            posterior = np.sum(log_pdf_values)
            posterior = prior + posterior
            posteriors.append(posterior)

        return self.classes[np.argmax(posteriors)]

    def predict(self, X : np.ndarray) -> np.ndarray:
        """
        Predict the class label for all of the samples
        """
        y_pred = [self.predict_instance(x) for x in X]
        return np.array(y_pred)
```

Snipped

Kelas GaussianNaiveBayes merupakan implementasi dari algoritma Naive Bayes dengan asumsi Gaussian (distribusi normal) yang dibuat dari awal menggunakan python. Kelas ini juga menurunkan kelas dari BaseEstimator dan ClassifierMixin untuk kompatibilitas dengan fungsi cross_val_score dari scikit-learn.

Algoritma ini pertama kali mengidentifikasi semua kelas unik yang ada dalam data Untuk setiap kelas, algoritma menghitung:

- Mean (rata-rata): Menghitung nilai rata-rata setiap fitur untuk data yang termasuk dalam kelas tersebut
- Variance (varians): Menghitung sebaran data untuk setiap fitur dalam kelas tersebut
- Prior: Menghitung probabilitas awal setiap kelas, yaitu jumlah data dalam kelas tersebut dibagi total jumlah data

Kemudian, akan dihitung probabilitas sebuah nilai fitur termasuk dalam suatu kelas menggunakan distribusi normal. Rumus yang digunakan adalah rumus distribusi normal:

$$(1/\sqrt{(2\pi\sigma^2)}) \ * \ e\text{^}(-\ (x\ -\ \mu)^2/2\sigma^2)$$

Dimana μ adalah mean,σ² adalah variance, epsilon (1e-9) ditambahkan untuk menghindari pembagian dengan nol

Untuk setiap kelas, akan dihitung posterior probability dengan rumus Bayes dan digunakan logaritma untuk menghindari underflow karena perkalian angka-angka kecil Prior probability diubah ke dalam bentuk logaritma dan PDF juga dihitung untuk setiap fitur dan dijumlahkan (dalam bentuk log). Kelas dengan posterior probability tertinggi dipilih sebagai hasil prediksi.

Bab 3
Implementasi ID3

DecisionTree.py

```python
import numpy as np
from sklearn.base import BaseEstimator, ClassifierMixin

class Node:
    """
    Create a node for the Decision Tree
    """

    def __init__(self, feature_index: int = None, threshold: float = None, left: 'Node' = None, right: 'Node' = None, info_gain: float = None, value: int = None):
        """
        Constructor for the Node class
        """
        self.feature_index = feature_index
        self.threshold = threshold
        self.left = left
        self.right = right
        self.info_gain = info_gain
        self.value = value

class DecisionTree(BaseEstimator, ClassifierMixin):
    """
    Create an instance of the Decision Tree algorithm
    """
    def __init__(self, min_samples_split: int = 4, max_depth: int = 4, mode: str = "gini"):
        """
        Constructor for the Decision Tree model
        """
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth
        self.root = None
        self.mode= mode

    def split(self, dataset: np.ndarray, feature_index: int, threshold: float) -> tuple[np.ndarray, np.ndarray]:
        """
        Split the dataset into two parts based on the feature and threshold
        """
        dataset_left= np.array([row for row in dataset if row[feature_index] <= threshold])
        dataset_right= np.array([row for row in dataset if row[feature_index] > threshold])
        return dataset_left, dataset_right

    def entropy(self, y: np.ndarray) -> float:
        """
        Calculate the entropy of the dataset
        """
        class_labels= np.unique(y)
        entropy= 0
        for cls in class_labels:
            p= len(y[y== cls]) / len(y)
            entropy += -p * np.log2(p)

        return entropy

    def gini_index(self, y: np.ndarray) -> float:
        """
        Calculate the Gini index of a tree node
        """
        class_labels= np.unique(y)
        gini= 0
        for cls in class_labels:
            p= len(y[y== cls]) / len(y)
            gini += p * (1 - p)

        return gini

    def calculate_leaf_value(self, y: np.ndarray) -> int:
        """
        Calculate the leaf node value
        """
        Y= list(y)
        return max(Y, key= Y.count)

    def information_gain(self, parent_node: np.ndarray, left_child: np.ndarray, right_child: np.ndarray, mode: str = "gini") -> float:
        """
        Calculate the information gain of a split
        """

        left_Weight= len(left_child) / len(parent_node)
        right_Weight= len(right_child) / len(parent_node)

        if mode== "gini":
            gain= self.gini_index(parent_node) - (left_Weight * self.gini_index(left_child) + right_Weight * self.gini_index(right_child))
        else:
            gain= self.entropy(parent_node) - (left_Weight * self.entropy(left_child) + right_Weight * self.entropy(right_child))

        return gain
```

Snipped

DecisionTree.py

```python
    def get_best_split(self, dataset: np.ndarray, num_features: int) -> dict:
        """
        Find the best split for the dataset based on the information gain
        """
        best_split= {}
        max_info_gain= -float('inf')

        for feature_index in range (num_features):
            feature_values= dataset[:, feature_index]
            possible_thresholds= np.unique(feature_values)

            for thresholds in possible_thresholds:
                dataset_left, dataset_right= self.split(dataset, feature_index, thresholds)

                if len(dataset_left) > 0 and len(dataset_right) > 0:
                   y, y_left, y_right= dataset[:, -1], dataset_left[:, -1], dataset_right[:, -1]

                   current_info_gain= self.information_gain(y, y_left, y_right, self.mode)

                   if current_info_gain > max_info_gain:
                       best_split['feature_index']= feature_index
                       best_split['threshold']= thresholds
                       best_split['data_left']= dataset_left
                       best_split['data_right']= dataset_right
                       best_split['info_gain']= current_info_gain
                       max_info_gain= current_info_gain

        return best_split

    def construct_tree(self, dataset: np.ndarray, current_depth: int = 0) -> 'Node':
        """
        Construct the decision tree recursively
        """
        X, Y= dataset[:,:-1], dataset[:,-1]
        num_samples, num_features= X.shape

        if num_samples >= self.min_samples_split and current_depth <= self.max_depth:
            best_split= self.get_best_split(dataset, num_features)

            if best_split['info_gain'] > 0:
                left_subtree= self.construct_tree(best_split['data_left'], current_depth+1)
                right_subtree= self.construct_tree(best_split['data_right'], current_depth+1)
                return Node(best_split['feature_index'], best_split['threshold'], left_subtree, right_subtree, best_split['info_gain'])

        # Calculate leaf node
        leaf_value= self.calculate_leaf_value(Y)

        return Node(value= leaf_value)

    # def print(self, tree= None, indent= " "):

    #     if tree is None:
    #         tree= self.root

    #     if tree.value is not None:
    #         print(tree.value)

    #     else:
    #         print(f"{tree.feature_index}, {tree.threshold} {tree.info_gain}")
    #         print(f"{indent}left: ", end= "")
    #         self.print(tree.left, indent + indent)
    #         print(f"{indent}right: ", end= "")
    #         self.print(tree.right, indent + indent)

    def fit(self, X: np.ndarray, Y: np.ndarray):
        """
        Fit the training data to the model
        """
        Y= np.reshape(Y, (-1, 1))
        dataset= np.concatenate((X, Y), axis= 1)
        self.root= self.construct_tree(dataset)

    def _predict(self, x: np.ndarray, tree: 'Node') -> int:
        """
        Predict the class label for a single sample
        """

        if tree.value is not None:
            return tree.value

        feature_val= x[tree.feature_index]

        if feature_val <= tree.threshold:
            return self._predict(x, tree.left)
        else:
            return self._predict(x, tree.right)

    def predict(self, X: np.ndarray) -> np.ndarray:
        """
        Predict the class labels for a set of samples
        """
        return [self._predict(x, self.root) for x in X]
```

Snipped

ID3 adalah algoritma yang membangun model prediksi dalam bentuk struktur pohon. Implementasi ini menggunakan pendekatan top-down (dari atas ke bawah) dengan dua metrik impurity yang bisa dipilih: Gini Index atau Entropy. Namun, karena yang diminta adalah ID3 model yang basisnya hanya menerima metrik impurity entropy, maka pada tugas besar kali ini, pelatihan dan pengujian model hanya akan menggunakan metrik impurity entropy saja. Model ini dibangung dengan meniru model CART yang ada pada *library* scikit-learn.

Implementasi ini mendefinisikan kelas Node yang merepresentasikan setiap titik keputusan dalam pohon. Kemudian, kelas DecisionTree: Kelas utama ini mewarisi dari BaseEstimator dan ClassifierMixin untuk kompatibilitas dengan cross_val_score scikit-learn. Parameter inisialisasinya meliputi:

- min_samples_split: Jumlah minimum sampel yang dibutuhkan untuk melakukan split
- max_depth: Kedalaman maksimum pohon
- mode: Metrik yang digunakan (hanya entropy yang akan digunakan pada tugas besar kali ini)

ID3 bekerja dengan membangun struktur pohon keputusan melalui proses *training* yang berurutan. Pertama, algoritma menerima data *training* yang terdiri dari fitur dan label kelaS. Proses pembangunan pohon dimulai dari akar (root) dan berlanjut secara rekursif ke bawah.

Pada setiap node, algoritma mencari cara terbaik untuk memisahkan data menjadi dua kelompok. Ini dilakukan dengan mengevaluasi setiap fitur dan mencoba berbagai nilai threshold untuk menemukan pemisahan yang menghasilkan information gain tertinggi. Information gain menunjukkan seberapa baik sebuah pemisahan dapat memurnikan data, semakin tinggi nilainya, semakin baik pemisahan tersebut.

Pengujian model ini menggunakan metrik impurity entropy, yaitu dengan cara mengukur ketidakpastian dalam data. Metrik ini membantu algoritma menentukan seberapa baik sebuah pemisahan dalam memisahkan kelas-kelas yang berbeda.

Proses pembuatan cabang baru akan terus berlanjut sampai salah satu kondisi pemberhentian terpenuhi:

- Jumlah sampel di node kurang dari minimum yang ditentukan (min_samples_split)
- Kedalaman pohon mencapai batas maksimum (max_depth)
- Tidak ada lagi pemisahan yang menghasilkan information gain positif
- Semua sampel di node tersebut sudah berasal dari kelas yang sama (homogen)

Ketika melakukan prediksi untuk data baru, algoritma mulai dari node akar dan mengikuti jalur ke bawah berdasarkan nilai-nilai fitur data tersebut. Pada setiap node internal (bukan daun), algoritma membandingkan nilai fitur dengan threshold node tersebut. Jika nilainya lebih kecil atau sama dengan threshold, data akan diarahkan ke cabang kiri; jika lebih besar, akan diarahkan ke cabang kanan.

Proses ini berlanjut sampai mencapai node daun (leaf node). Label kelas yang tersimpan di node daun tersebut menjadi prediksi untuk data input. Label ini ditentukan berdasarkan kelas mayoritas dari data *training* yang berakhir di node tersebut selama proses *training*.

Bab 4
Tahap Cleaning dan Preprocessing

4.1. Cleaning

Pada tahap cleaning kita melakukan imputasi pada *missing values* karena hasil dari EDA kami datanya sangat *noisy* dan kelas cenderung bernilai ‘normal’. Awalnya kami membuat beberapa imputer kami sendiri, tetapi tidak perform baik sehingga kami menggunakan imputer yang biasa saja. Ini beberapa Imputer yang kami gunakan:

1. SimpleImputer(Strategy=’most_frequent’)
   Imputer ini kami gunakan untuk impute missing value pada fitur kategorikal
2. SimpleImputer(Strategy=’median’)
   Imputer ini kami gunakan utnuk impute missing value pada fitur numerikal

4.2. Preprocessing

Pada tahap ini kami membuat pipeline dengan method Pipeline dan ColumnTransformer dari scikit learn. Kami mendefinisikan dua pipeline, satu untuk kolom numerik satu untuk kategorikal

```
num_cols= list(df_train.select_dtypes(include=np.number))

cat_cols = list(df_train.select_dtypes(include='object'))
all_cols = list(df_train.columns)



cat_transformer = Pipeline(steps = [
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('OHE', OneHotEncoder(handle_unknown='ignore',sparse_output = False)),

])
num_transformer = Pipeline(steps = [
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler()),
])

preprocessor = ColumnTransformer(
    transformers = [
    ('cat_transformer', cat_transformer,cat_cols),
    ('num_transformer', num_transformer, num_cols)
    ]
)
```

```
X_train_transformed = preprocessor.fit_transform(X_train)
X_val_transformed = preprocessor.transform(X_val)

[ ] le= LabelEncoder()
    y_train= le.fit_transform(y_train)
    y_val= le.transform(y_val)
```

Beberapa teknik feature encoder yang kami gunakan:

1. One Hot Encoder
   Kami menggunakan One Hot Encoder untuk encoding data kategorikal karena kebanyakan dari data kategorikal tidak memiliki arti urutan, karena itu lah kami menggunakan OHE ketimbang Ordinal Encoder. Sparse = false karena kita ingin keluarannya numpy array bukan sparse matrix, kekurangannya adalah lebih boros memory.
2. StandardScaler
   Kami menggunakan ini untuk meng scale datanya agar berada pada skala yang sama yaitu mean =0 variance= 1.
3. LabelEncoder
   Digunakan untuk encoding target menjadi nilai integer ordinal

Kami meng compile Pipeline tadi dengan menggunakan ColumnTransformer agar masing masing pipeline bisa diterapkan pada kolom yang sesuai. Setelah dicompile kita fit_transform pada data training dan transform pada data validasi.
Kami juga mengimplementasikan AutoEncoder dengan tensorflow dan juga menggunakan PCA, tetapi tidak memberikan hasil yang baik sehingga tidak kami gunakan untuk submisi.

```
class AutoEncoders(Model):
  def __init__(self, output_units):
    super().__init__()
    self.encoder = Sequential(
        [   Dense(156,activation = 'relu'),
            Dense(128, activation = 'relu'),
        ]
    )
    self.decoder = Sequential(
        [

          Dense(156,activation= "relu"),
          Dense(output_units, activation="tanh")
        ]
    )
  def call(self, inputs):
      encoded = self.encoder(inputs)
      decoded = self.decoder(encoded)
      return decoded
```

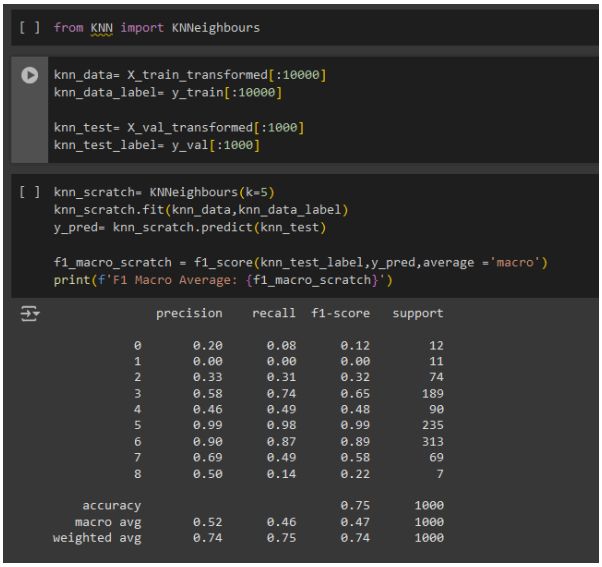
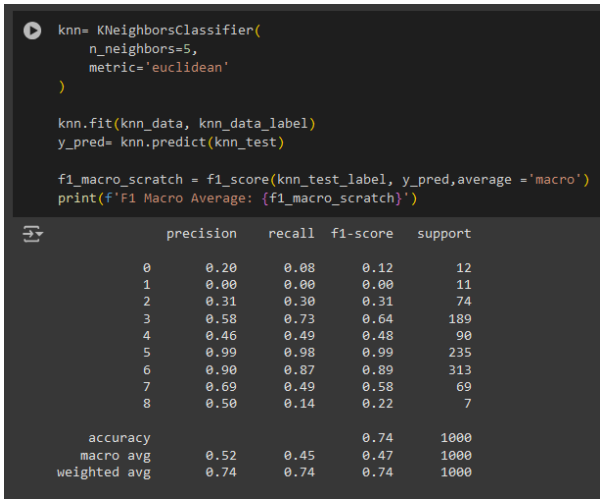
```
[ ]  ae = AutoEncoders(len(X_train_scaled.columns))
```

```
[ ]  ae.compile(
        loss='mae',
        metrics=['mae'],
        optimizer='adam'
    )

    history = ae.fit(
        X_train_scaled,
        X_train_scaled,
        epochs=10,
        batch_size=32,
        validation_data=(X_test_scaled, X_test_scaled)
    )
```

Bab 5
Perbandingan Hasil Prediksi

1. KNN

Scratch	Scikit-learn
	

Perbandingan antara KNN dari Scratch dan KNN dari Scikit-learn menunjukkan bahwa meskipun keduanya menggunakan konsep dasar yang sama, yaitu menghitung jarak dan memilih tetangga terdekat untuk prediksi, Scikit-learn lebih efisien dan cepat berkat optimasi yang sudah dilakukan, termasuk penggunaan multi-threading. Hal ini menghasilkan akurasi yang lebih tinggi dan waktu eksekusi yang lebih cepat pada Scikit-learn, terutama saat menangani dataset besar. Sementara itu, KNN dari Scratch memberikan pemahaman lebih mendalam tentang algoritma, namun memerlukan optimasi lebih lanjut untuk meningkatkan performa, seperti mengurangi waktu eksekusi yang lebih lama dibandingkan dengan implementasi built-in dari Scikit-learn.

2. Naive Bayes

Scratch	Scikit-learn

```
[ ] from GNB import GaussianNaiveBayes

gnb_scratch= GaussianNaiveBayes()
gnb_scratch.fit(X_train_transformed, y_train)
y_pred= gnb_scratch.predict(X_val_transformed)

f1_macro_scratch = f1_score(y_val,y_pred,average ='macro')
print(f'F1 Macro Average: {f1_macro_scratch}')

F1 Macro Average: 0.15689196303232086
```

```
gnb= GaussianNB()
gnb.fit(X_train_transformed, y_train)
y_pred= gnb.predict(X_val_transformed)

f1_macro_sklearn = f1_score(y_val,y_pred,average ='macro')
print(f'F1 Macro Average: {f1_macro_sklearn}')

F1 Macro Average: 0.21145610600630813
```

Keduanya memiliki hasil F1 Macro Average yang berbeda dimana scikit-learn memiliki skor yang lebih tinggi dibanding dari scratch yaitu 0.211 berbanding dengan 0.156. Perbandingan antara Gaussian Naïve Bayes (GNB) dari Scratch dan GNB dari Scikit-learn menunjukkan bahwa keduanya menggunakan pendekatan yang serupa dalam mengklasifikasikan data, yaitu dengan menghitung statistik penting untuk tiap kelas dan menggunakan distribusi Gaussian untuk memprediksi kelas dengan probabilitas tertinggi. Namun, GNB dari Scikit-learn memiliki keunggulan dalam hal akurasi karena menangani edge-case dengan lebih baik, seperti pembagian dengan nol menggunakan epsilon dan penanganan underflow pada probabilitas yang sangat kecil. Selain itu, GNB dari Scikit-learn lebih efisien dalam waktu eksekusi dan lebih stabil, terutama pada dataset besar dengan banyak fitur. Untuk meningkatkan model yang dibuat dari scratch, diperlukan optimasi, penanganan edge-case yang lebih baik, serta pemilihan teknik encoding yang lebih sesuai, seperti label encoding, untuk menghindari masalah curse of dimensionality yang menghambat pelatihan model.

3. ID3

Scratch

```
[ ] from DecisionTree import DecisionTree

dt_data= X_train_transformed[:1000]
dt_data_label= y_train[:1000]

dt_test= X_val_transformed[:100]
dt_test_label= y_val[:100]

[ ] dt_scratch= DecisionTree(mode='entropy')
dt_scratch.fit(dt_data, dt_data_label)
y_pred= dt_scratch.predict(dt_test)

f1_macro_scratch = f1_score(dt_test_label,y_pred,average ='macro')
print(f'F1 Macro Average: {f1_macro_scratch}')

F1 Macro Average: 0.4820763561551082
```

Scikit-learn

```
[ ] dt= DecisionTreeClassifier(criterion = "entropy", max_depth= None, min_samples_leaf=4)
dt.fit(dt_data, dt_data_label)
y_pred= dt.predict(dt_test)

f1_macro_sklearn = f1_score(dt_test_label,y_pred,average ='macro')
print(f'F1 Macro Average: {f1_macro_sklearn}')

F1 Macro Average: 0.4704355314133676
```

Kedua algoritma memiliki F1 Macro Average yang hampir sama yaitu Scratch

bernilai 0.48 sedangkan scikit-learn bernilai 0.47. Perbandingan antara Decision Tree dari Scratch dan Decision Tree dari Scikit-learn menunjukkan bahwa keduanya memiliki prinsip dasar yang sama. Namun, Decision Tree dari Scikit-learn memiliki keunggulan dalam hal akurasi karena optimisasi yang lebih baik (walaupun pada kasus ini scikit-learn memperoleh skor lebih kecil, kemungkinan besar karena hyperparameter default yang ada pada scikit-learn selain yang di-*define* menyebabkan model scikit learn *overfit*), termasuk penanganan kesalahan presisi floating-point dan pruning yang lebih efisien, serta manajemen hyperparameter yang lebih efektif. Selain itu, model dari Scikit-learn lebih efisien dalam waktu eksekusi, sementara model dari scratch cenderung lebih lambat dan kurang optimal.

Bab 6
Kontribusi Kelompok

Mohammad Nugraha Eka Prawira (13522001)	Pre Processing, Laporan
Muhammad Al Thariq Fairuz (13522027)	Model from scratch, pre processing, Laporan
Randy Verdian (13522067)	EDA, Pre processing, Laporan
Emery Fathan Zwageri (13522079)	AutoEncoders, preprocessing, kaggle gaming, Laporan

Bab 7
Referensi

KNN
GNB
DTL
KNeighborsClassifier — scikit-learn 1.6.0 documentation
Gaussian Naive Bayes - GeeksforGeeks
GaussianNB — scikit-learn 1.6.0 documentation
Decision Tree - GeeksforGeeks
DecisionTreeClassifier — scikit-learn 1.6.0 documentation
https://www.analyticsvidhya.com/blog/2021/06/dimensionality-reduction-using-autoencoders-in-python/
Autoencoders -Machine Learning - GeeksforGeeks