TUGAS BESAR 2

IF3070 – DASAR INTELIGENSI ARTIFISIAL

Implementasi Algoritma Pembelajaran Mesin

Disusun Oleh Kelompok 10:

Clement Nathanael Lim / 18222032 (K02)

Mattheuw Suciadi Wijaya / 18222048 (K02)

Hartanto Luwis / 18222064 (K02)

Farah Aulia / 18222096 (K02)

PROGRAM STUDI SISTEM DAN TEKNOLOGI INFORMASI

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2024

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

BAB I
IMPLEMENTASI MODEL ALGORITMA

1.1. Implementasi Algoritma K – *Nearest Neighbor* (KNN)

K – *Nearest Neighbor* merupakan algoritma *supervised learning* yang digunakan untuk membuat prediksi dengan mengklasifikasikan titik data berdasarkan kesamaan tertentu dari titik data lain yang berdekatan. Algoritma ini akan mengidentifikasi persamaan antara data lama dan data baru kemudian mengkategorikan data baru berdasarkan kategori yang paling mirip dari data yang sudah dibuat sebelumnya. Algoritma ini memungkinkan pemilihan jarak yang digunakan dalam perhitungan kedekatan antar titik data.

Berikut merupakan langkah-langkah dalam melakukan implementasi algoritma KNN sebagai berikut.

1. **Persiapan Data**

   Tahapan awal dilakukan dengan metode persiapan data, dimana pada tahapan ini akan dilakukan pembersihan data yang bertujuan untuk mengatasi data yang hilang (*missing value*), *handling outlier*, serta *handling redundant data*. Selanjutnya, akan dilakukan *normalization* terhadap *feature* yang akan digunakan, dimana KNN akan sangat bergantung pada metrik jarak tersebut. Dilakukan normalisasi bertujuan untuk menyelaraskan skala setiap fitur, sehingga fitur dengan rentang nilai yang lebih besar tidak akan mendominasi perhitungan jarak nantinya. Penerapan normalisasi fitur nantinya akan menggunakan teknik *Min-Max Scaling*. Lalu, akan dipersiapkan sebuah data latih (*training*) dan data uji (*testing*) dengan tujuan agar performa algoritma dapat dilakukan evaluasi pada pengujian model nantinya.

2. **Perhitungan Jarak**

   Langkah selanjutnya yang akan dilakukan setelah persiapan data adalah perhitungan jarak, dimana KNN akan menggunakan jarak antara titik data *training* maupun data uji untuk menentukan hasil prediksi dari kesamaan berikut.

Terdapat 3 jenis metrik jarak yang kami gunakan untuk menguji prediksi model KNN tersebut antara lain.

a. Jarak *Euclidean*

Jarak *Euclidean* merupakan jarak untuk menghitung jarak lurus antara dua titik di ruang multidimensi yang dapat dihitung menggunakan rumus sebagai berikut.

$$d(x, y) = \sqrt{\sum_{i=1}^{n} \left(x_i - y_i\right)^2}$$

Keunggulan dari penggunaan jarak *euclidean* adalah merupakan rumus yang cocok untuk digunakan terhadap data yang memiliki korelasi linear. Namun, penggunaan rumus euclidean sangat sensitif terhadap skala yang dimiliki oleh masing-masing fitur pada setiap implementasi model yang dibuat.

b. Jarak *Manhattan*

Jarak *Manhattan* merupakan jarak yang digunakan untuk mengukur total perbedaan absolut antara koordinat dua titik yang dapat dihitung menggunakan rumus sebagai berikut.

$$d(x, y) = \sum_{i=1}^{n} \left|x_i - y_i\right|$$

Keunggulan dari penggunaan jarak manhattan sendiri ialah cocok untuk data yang distribusi grid atau fitur yang tidak mempunyai korelasi linear. Namun, penggunaan rumus ini bisa saja tidak akurat apabila terdapat suatu dimensi yang masih memiliki banyak outlier yang belum ditangani.

c. Jarak *Minkowski*

Jarak *Minkowski* merupakan hasil generalisasi dari rumus *euclidean* dan *manhattan* yang dapat dihitung menggunakan rumus sebagai berikut.

$$d(x,y) = \left(\sum_{i=1}^{n} \left| x_i - y_i \right|^p \right)^{\frac{1}{p}}$$

Jika nilai p = 2, maka akan menjadi perhitungan yang sama dengan rumus *euclidean*. Apabila p = 1, maka perhitungan akan sama dengan rumus *manhattan*.

3. **Menentukan Tetangga Terdekat**

   Setelah jarak antara titik data uji dan seluruh titik pada data latih telah berhasil dihitung, maka langkah selanjutnya adalah menentukan tetangga terdekat berdasarkan hasil perhitungan rumus yang digunakan oleh algoritma KNN sesuai dengan nilai K. Nilai K diambil berdasarkan titik yang paling terdekat sesuai dengan nilai K yang telah ditentukan sebelumnya. Pemilihan nilai K tetap harus disesuaikan dengan hasil pengujian yang telah dilakukan.

4. **Klasifikasi**

   Tahapan terakhir dalam langkah-langkah pembuatan model KNN adalah klasifikasi, dimana klasifikasi dilakukan dengan menentukan kelas dari hasil data uji yang didapatkan berdasarkan nilai K tetangga terdekat tersebut. Pada tahap ini, akan dilakukan penentuan kelas data uji berdasarkan mayoritas *voting* dimana setiap tetangga akan memiliki nilai label masing-masing dan label yang paling sering muncul di antara K tetangga terdekat akan menjadi prediksi untuk titik data uji.

Berdasarkan hasil penerapan langkah-langkah pembuatan model KNN yang telah dilakukan sesuai dengan tahapan di atas, maka dapat dilakukan implementasi dalam algoritma sebagai berikut.

a. **Fungsi `def__init__(self, k=3, metric= “euclidean”, p=3)`**

```python
# Mendefinisikan perhitungan jarak
import numpy as np
from collections import Counter

# Mendefinisikan perhitungan jarak
class KNNSearch:

    def __init__(self, k=3, metric="euclidean", p=3):
        self.k = k
        self.metric = metric
        self.p = p  # Default p for Minkowski distance
```

Gambar 1.1.1. Fungsi *_init_* pada KNN

Fungsi di atas berperan sebagai inisialisasi awal yang berisikan parameter input awal yakni variabel "k" yang berperan sebagai jumlah tetangga terdekat yang akan digunakan dalam proses klasifikasi nantinya, dimana pada kondisi ini kami mengambil k = 3. Selain itu, terdapat variabel `self.metric` yang berperan dalam menentukan jenis metrik jarak yang akan digunakan untuk menghitung jarak antar titik data yang nantinya akan digunakan dalam percobaan menghitung jarak antar titik data. Variabel p nantinya akan digunakan dalam perhitungan jarak antar titik data menggunakan rumus *manhattan*.

b. Fungsi `Euclidean_dist(self, x1, x2)`

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

Gambar 1.1.2. Fungsi *euclidean_dist* pada KNN

Fungsi di atas merupakan penerapan rumus menghitung jarak *euclidean* sesuai dengan penerapan rumus yang telah dipaparkan sebelumnya pada penjelasan langkah-langkah implementasi model KNN.

c. Fungsi `manhattan_dist(self, x1, x2)`

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

Gambar 1.1.3. Fungsi *manhattan_dist* pada KNN

Fungsi di atas merupakan penerapan rumus menghitung jarak *manhattan* sesuai dengan penerapan rumus yang telah dipaparkan sebelumnya pada penjelasan langkah-langkah implementasi model KNN.

d. Fungsi `minkowski_dist(self, x1, x2)`

```
def minkowski_dist(self, x1, x2):
    return np.power(np.sum(np.abs(x1 - x2) ** self.p), 1 / self.p)
```

Gambar 1.1.4. Fungsi *minkowski_dist* pada KNN

Fungsi di atas merupakan penerapan rumus menghitung jarak *minkowski* sesuai dengan penerapan rumus yang telah dipaparkan sebelumnya pada penjelasan langkah-langkah implementasi model KNN.

e. Fungsi `fit(self, X_train, y_train)`

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

Gambar 1.1.5. Fungsi *fit* pada KNN

Fungsi di atas berperan untuk melatih model dengan melakukan penyimpanan terhadap data training **X (*features*)** dan **y (*label*)** ke dalam atribut *internal class*. **`X_train`** merupakan data latih berupa fitur-fitur yang akan digunakan sebagai prediksi nantinya, sedangkan **`y_train`** berperan sebagai label atau target yang sesuai untuk sampel dalam **`X_train`**.

f. Fungsi `hitung_jarak(self, x1, x2)`

```
def hitung_jarak(self, x1, x2):
    if self.metric == 'euclidean':
        return self.Euclidean_dist(x1, x2)
    elif self.metric == 'manhattan':
        return self.manhattan_dist(x1, x2)
    elif self.metric == 'minkowski':
        return self.minkowski_dist(x1, x2)
    else:
        raise ValueError(f"Metrik tak valid: {self.metric}")
```

Gambar 1.1.6. Fungsi hitung_jarak pada KNN

Fungsi di atas berperan untuk menghitung jarak antara dua titik data, yakni **`x1`** dan **`x2`**. Fungsi **`hitung_jarak`** akan memanggil fungsi-fungsi sebelumnya seperti *euclidean*, *manhattan*, ataupun *minkowski* yang akan digunakan sebagai

metode pengukuran metrik jarak. Pemilihan metrik tersebut nantinya akan dilakukan berdasarkan parameter **`self.metric`** yang akan diatur sata inisialisasi objek *class*.

g. Fungsi `_get_neighbors(self, x_test)`

```python
def _get_neighbors(self, x_test):
    distances = []
    for i, x_train in enumerate(self.X_train):
        distance = self.hitung_jarak(x_train, x_test)
        distances.append((distance, self.y_train[i]))

    # Urutkan berdasarkan jarak dan ambil k tetangga terdekat
    distances.sort(key=lambda x: x[0])
    neighbors = [label for _, label in distances[:self.k]]
    return neighbors
```

Gambar 1.1.7. Fungsi *get_neighbors* pada KNN

Fungsi di atas berperan untuk melakukan prediksi dan menemukan nilai K sebagai tetangga terdekat dari sampel data uji (**`x_test`**) berdasarkan metrik jarak yang telah ditentukan sebelumnya pada fungsi **`hitung_jarak`**. Variabel **`x_test`** akan berperan sebagai sebuah array atau vektor yang mewakili sampel data uji tersebut dimana function **`_get_neighbors`** akan menghitung jarak antara **`x_test`** dan setiap sampe dalam data latih **`self.X_train`**. Hasil jarak yang telah dihitung akan disimpan dalam bentuk pasangan (**`distance`**, **`label`**) ke dalam daftar **`distances`** (jarak antara **`x_train`** dan **`x_test`**) yang telah dibuat. Variabel **`self.y_train[i]`** berperan sebagai label dari titik data latih yang bersangkutan tersebut. Setelah jarak dihitung untuk seluruh data *training*, maka daftar **`distances`** tersebut akan diurutkan berdasarkan nilai jarak (**`distance`**) dari nilai terkecil hingga nilai terbesar. Lalu, setelah jarak diurutkan maka fungsi tersebut akan mengambil nilai K tetangga terdekat pada elemen pertama atau terkecil dari hasil pengurutan **`distances`** yang telah dilakukan.

h. Fungsi `predict(self, X_test)`

```
def predict(self, X_test):
    predictions = []
    for x_test in X_test:
        neighbors = self._get_neighbors(x_test)
        most_common = Counter(neighbors).most_common(1)
        predictions.append(most_common[0][0])
    return np.array(predictions)
```

Gambar 1.1.8. Fungsi *predict* pada KNN

Fungsi di atas berperan dalam melakukan prediksi terhadap data uji (**`X_test`**) berdasarkan data latih yang telah disimpan sebelumnya. Fungsi **`predict()`** akan menentukan label atau nilai target untuk setiap sampel data uji berdasarkan nilai k tetangga terdekat yang telah ditemukan. Mula-mula, dilakukan inisialisasi daftar prediksi dimana variabel **`predictions`** yang merupakan daftar kosong yang akan diisi dengan label prediksi untuk setiap sampel data uji. Lalu, untuk setiap data sampel uji dalam **`X_test`** akan dilakukan pengecekan untuk menemukan nilai k tetangga terdekat dengan memanggil fungsi sebelumnya, yakni **`_get_neighbors(x_test)`**.

`Counter(neighbors).mos_common(1)` digunakan untuk menghitung frekuensi setiap label di antara k tetangga terdekat. Hasilnya nanti akan dikembalikan ke dalam variabel **`most_common(1)`** dimana variabel tersebut akan mengembalikan elemen yang paling sering muncul beserta jumlah kemunculannya dalam format [(**`label`**, **`count`**)]. Label yang paling umum kemudian diambil dengan nilai frekuensi tertinggi. Setelah proses iterasi selesai, maka hasil dari **`predictions`** diubah menjadi array NumPy [**`np.array(predictions)`**].

i. Fungsi `score(self, X_test, y_test)`

```
def score(self, X_test, y_test):
    y_pred = self.predict(X_test)
    return np.mean(y_pred == y_test)
```

Gambar 1.1.9. Fungsi *score* pada KNN

Fungsi di atas akan berfungsi untuk mengukur akurasi nilai hasil prediksi yang telah dilakukan oleh model KNN terhadap data uji tersebut. Fungsi ini akan membandingkan hasil prediksi model dengan nilai label sebenarnya dari data uji dan menghitung proporsi prediksi yang benar nantinya. Awalnya fungsi akan memanggil `self.predict(X_test)` untuk mendapatkan label prediksi model (y_pred) berdasarkan data uji tersebut. Kemudian akan dilakukan perbandingan prediksi dengan label sebenarnya melalui ekspresi `y_pred == y_test` yang akan menghasilkan *array boolean*. Fungsi `np.mean(y_pred == y_test)` digunakan untuk menghitung rata-rata dari *array boolean* tersebut yang setara dengan proporsi prediksi yang benar (akurasi).

1.2. Implementasi Algoritma *Gaussian Naive Bayes*

Gaussian Naive Bayes adalah algoritma klasifikasi yang menggunakan prinsip probabilitas dan distribusi *Gaussian* dengan menggunakan Teorema Bayes dalam membuat klasifikasi dari data. Selain itu, antara suatu fitur terhadap fitur lainnya bersifat independen. Oleh karena itu, algoritma ini cocok digunakan untuk data yang bersifat kontinu. *Gaussian Naive Bayes* mengasumsikan bahwa setiap parameter, yang juga disebut fitur atau prediktor, memiliki kapasitas independen untuk memprediksi variabel keluaran.

Berikut merupakan langkah-langkah yang akan dilakukan untuk menerapkan implementasi algoritma *Gaussian Naive Bayes*:

1. **Persiapan data**

   Tahapan ini dilakukan kurang lebih sama dengan tahapan persiapan pada algoritma KNN.

2. **Melakukan perhitungan probabilitas *prior* untuk setiap kelas**

   Perhitungan probabilitas *prior* dilakukan dengan melakukan pembagian jumlah sampel dalam kelas tersebut terhadap total jumlah sampel. Selain itu, pada tahapan ini juga akan dilakukan perhitungan mean dan variansi untuk setiap kelas yang akan digunakan pada perhitungan *Likelihood*.

3. **Melakukan perhitungan *likelihood***

   Perhitungan ini dilakukan terhadap semua fitur dalam data uji untuk dihitung probabilitas kondisionalnya dengan menggunakan fungsi distribusi normal, di mana μk adalah rata-rata dan $\sigma^2 k$ adalah variansi dari fitur x_i pada kelas Ck.

$$p(x_i|y_j) = \frac{1}{\sqrt{2\pi\sigma_j^2}} e^{-\frac{(x_i-\mu_j)^2}{2\sigma_j^2}}$$

4. **Melakukan perhitungan posterior**

   Menghitung probabilitas *posterior* untuk setiap kelas dengan menjumlahkan prior dengan semua likelihood yang dihitung sebelumnya.

5. **Klasifikasi**

   Tentukan kelas dengan probabilitas posterior tertinggi sebagai prediksi untuk data uji.

Berdasarkan hasil penerapan langkah-langkah pembuatan model *Gaussian Naive Bayes* yang telah dilakukan sesuai dengan tahapan di atas, maka dapat dilakukan implementasi dalam algoritma sebagai berikut.

1. **Fungsi `__init__(self)`**

```
class GaussianNaiveBayes:
    def __init__(self):
        self.classes = None
        self.mean = None
        self.var = None
        self.priors = None
        self.i = 1  # Untuk mencatat urutan prediksi seperti pada KNN
```

Gambar 1.2.1. Fungsi *init* pada *naive bayes*

Tahapan awal dari algoritma ini adalah inisiasi variabel yang akan digunakan pada algoritma *Gaussian Naive Bayes* ini. Tahapan ini merupakan

tahapan untuk setiap variabel didefinisikan dan diberi nilai awal *none*. Variabel tersebut meliputi:

- `self.classses` yang menyimpan daftar kelas unik,
- `self.mean` yang menyimpan rata-rata setiap fitur untuk setiap kelas,
- `self.var` yang menyimpan variansi setiap fitur untuk setiap kelas,
- `self.priors` yang menyimpan probabilitas prior untuk setiap kelas,
- `self.i` yang menyimpan urutan prediksi seperti pada KNN.

2. **Fungsi `fit(self,X,y)`**

```python
def fit(self, X, y):
    # Identifikasi kelas unik
    self.classes = np.unique(y)
    # Inisialisasi mean, variansi, dan prior
    self.mean = np.zeros((len(self.classes), X.shape[1]), dtype=np.float64)
    self.var = np.zeros((len(self.classes), X.shape[1]), dtype=np.float64)
    self.priors = np.zeros(len(self.classes), dtype=np.float64)

    for idx, c in enumerate(self.classes):
        X_c = X[y == c]
        self.mean[idx, :] = X_c.mean(axis=0)
        # Variansi diberi smoothing untuk mencegah pembagian nol
        self.var[idx, :] = np.maximum(X_c.var(axis=0), 1e-9)
        self.priors[idx] = X_c.shape[0] / float(X.shape[0])
```

Gambar 1.2.2. Fungsi *fit* pada *naive bayes*

Pada awalnya dilakukan identifikasi kelas unik yang ada di y dengan menggunakan `np.unique(y)`. Setelah itu, inisiasi *mean*, *varian*, dan *prior* sesuai dengan jumlah kelas dan fitur. Setelah itu, lakukan iterasi untuk setiap kelas dengan mengambil semua nilai pada kelas c, lalu hitung *mean* dan *variansi* untuk setiap fitur. Selain itu, untuk menghindari pembagian dengan nol maka dilakukan *smoothing* pada variansi.

3. **Fungsi `_calculate_likelihood(self, class_idx, x)`**

Perhitungan *Likelihood* dilakukan berdasarkan rumus berikut :

$$p(x_i|y_j) = \frac{1}{\sqrt{2\pi\sigma_j^2}} e^{-\frac{(x_i-\mu_j)^2}{2\sigma_j^2}}$$

```python
def _calculate_likelihood(self, class_idx, x):
    mean = self.mean[class_idx]
    var = self.var[class_idx]
    # Rumus Gaussian
    numerator = np.exp(-0.5 * ((x - mean) ** 2) / var)
    denominator = np.sqrt(2 * np.pi * var)
    return numerator / denominator
```

Gambar 1.2.3. Fungsi *calculate_likelihood* pada *naive bayes*

Fungsi ini digunakan untuk melakukan perhitungan terhadap probabilitas kondisional atau *likelihood* untuk masing-masing kelas. Fungsi terlebih dahulu mengambil nilai rata-rata dan variansi dari kelas tertentu yang telah dihitung pada fungsi fit. Setelah itu dilakukan perhitungan secara terpisah untuk bagian numerator atau bagian eksponensial dan bagian denominator $\sqrt{2\pi\sigma^2}$. Setelah itu melakukan pembagian untuk kedua nilai tersebut yang akan menghasilkan probabilitas dari $P(x \mid c)$.

4. **Fungsi `_calculate_posterior(self, x)`**

```python
def _calculate_posterior(self, x):
    posteriors = []
    for idx, c in enumerate(self.classes):
        prior = np.log(self.priors[idx])
        likelihoods = np.sum(np.log(self._calculate_likelihood(idx, x)))
        posterior = prior + likelihoods
        posteriors.append(posterior)
    return self.classes[np.argmax(posteriors)]
```

Gambar 1.2.4. Fungsi *calculate_posterior* pada *naive bayes*

Fungsi ini digunakan untuk menghitung probabilitas posterior untuk setiap kelas. Fungsi akan melakukan iterasi terhadap setiap kelas untuk menghitung log prior atau logP(c) ditambah log *likelihood*. Fungsi akan memilih nilai posterior yang paling besar.

5. **Fungsi `_predict(self, x)`**

```
def _predict(self, x):
    return self._calculate_posterior(x)
```

Gambar 1.2.5. Fungsi *predict* pada *naive bayes*

Fungsi ini akan melakukan prediksi kelas untuk satu sampel dengan menggunakan fungsi **`_calculate_posterior(x)`** yang akan berperan untuk mengembalikan prediksi untuk sampel data (x) berdasarkan perhitungan posterior yang telah dilakukan sebelumnya.

6. **Fungsi `predict(self, x)`**

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

Gambar 1.2.6. Fungsi *predict* pada *naive bayes*

Fungsi di atas akan berperan untuk melakukan prediksi terhadap label atau nilai target untuk seluruh data uji (*X*) dimana fungsi ini akan memanggil fungsi sebelumnya, yakni **`_predict(x)`** untuk melakukan perhitungan terhadap prediksi pada setiap sampel individu dalam *dataset* uji. Pada akhirnya, fungsi ini akan mengembalikan Array NumPy yang nantinya akan berisi hasil prediksi yang telah didapat untuk semua sampel dalam X.

BAB II
CLEANING & PREPROCESSING

2.1. Tahap *Cleaning*

Berikut merupakan penjelasan mengenai tahapan proses *cleaning* yang akan dilakukan dalam implementasi algoritma pemodelan, mulai dari penanganan data yang hilang, data *outlier*, redundansi data, hingga penerapan *feature engineering* yang dipilih untuk pengembangan implementasi model algoritma yang akan digunakan nantinya.

2.1.1. *Handling Missing Data*

Kami menggunakan `simpleImputer` dari `sklearn.impute` , serta `iterativeImputer` untuk menangani *missing values,* Untuk menangani data numerikal kami, kami menggunakan *mean* untuk mengisi *missing values.* Sedangkan untuk data kategorikal, kami menggunakan strategi *most frequent.*

2.1.2. *Handling Outlier*

Kami melakukan *handling outliers* dengan mengubah data yang memiliki *outliers* dengan median. Penggunaan median karena media tidak terpengaruh oleh *outliers* untuk menjaga konsistensi.

2.1.3. *Remove Duplication*

Pada tahap ini, kami membuat fungsi untuk melakukan penghapusan untuk duplikasi. Akan tetapi, dari data set yang digunakan, diketahui bahwa dataset yang kita gunakan tidak memiliki duplikasi. Oleh karena itu, tidak ada duplikasi yang dihapus.

2.1.4. *Feature Engineering*

Pada tahap ini, kami melakukan beberapa implementasi fungsi untuk melakukan *feature selection* dalam proses *feature engineering*,

dimana seleksi fitur dilakukan untuk memilih fitur yang paling relevan terhadap target berdasarkan 2 jenis analisis utama, yakni korelasi untuk fitur numerik dan *Cramer's V* untuk fitur kategorikal.

Hasil analisis yang dilakukan bertujuan untuk membantu kami dalam mempertimbangkan fitur – fitur apa saja yang akan dipilih. Setelah dilakukan pertimbangan seleksi fitur numerik ataupun kategorikal tersebut, kami melakukan penggabungan terhadap fitur yang ada dengan tujuan untuk mempermudah proses prediksi. Kami juga melakukan pembatasan terhadap pemilihan fitur yang dilakukan. Berdasarkan *trial and error*, kami memutuskan untuk menggunakan fitur numerikal saja pada *model* yang dibangun, disebabkan adanya peningkatan akurasi dalam model yang dibangun.

2.2. Tahap *Preprocessing*

Berikut merupakan penjelasan mengenai tahapan proses *preprocessing* yang akan dilakukan dalam implementasi algoritma pemodelan nantinya, mulai dari penerapan *feature scaling*, *feature encoding*, hingga proses penanganan *imbalanced dataset* yang ada.

2.2.1. *Feature Scaling*

Untuk melakukan Scaling, kami menggunakan fungsi yang ada pada *library* `sklearn.preprocessing` yaitu `StandardScaler`. Scaler ini melakukan standarisasi fitur numerikal. Tujuan dilakukannya *feature scaling* ini untuk menjaga konsistensi skala fitur dan meningkatkan kinerja model.

2.2.2. *Feature Encoding*

Feature Encoding kami lakukan dengan menggunakan *library* `sklearn.preprocessing` yaitu *one hot encoder.* Alasan penggunaannya karena berdasarkan *feature selection,* fitur kategorikal yang kami pilih semuanya bertipe biner dan *one hot encoder* mengonversi

data kategorikal menjadi numerik dalam bentuk biner (0 untuk *false,* 1 untuk *true*). Oleh karena itu, *one hot encoder* cocok digunakan.

2.2.3. *Handling Imbalanced Dataset*

Pada tahap ini, kami tidak melakukan proses *handling imbalanced dataset*, sebab ketika dilakukan uji coba terhadap *imbalanced data* hasil prediksi yang didapatkan tidak lebih baik terhadap hasil prediksi yang dihasilkan oleh pemodelan algoritma sebelum dilakukan *handling imbalanced dataset* tersebut.

BAB III
ANALISIS & PEMBAHASAN

3.1. Analisis Hasil Prediksi Algoritma K – *Nearest Neighbor From Scratch & Library Scikit – learn*

3.1.1. Algoritma KNN dengan Jarak *Euclidean*

Berikut merupakan hasil analisis terhadap pengujian implementasi algoritma KNN menggunakan perhitungan jarak euclidean dengan nilai k = 5.

Tabel 3.1.1 Hasil Prediksi Algoritma KNN *Euclidean Scratch & Scikit – learn*

Scratch	*Scikit – learn*
Akurasi KNN dari scratch: 0.9857752489331437 precision recall f1-score support 0 0.98 0.83 0.90 53 1 0.99 1.00 0.99 650 accuracy 0.99 703 macro avg 0.98 0.91 0.95 703 weighted avg 0.99 0.99 0.99 703 F1 Score Macro: 0.9451569618673157	F1 Score Euclidean: 0.9451569618673157 Classification Report Euclidean: precision recall f1-score support 0 0.98 0.83 0.90 53 1 0.99 1.00 0.99 650 accuracy 0.99 703 macro avg 0.98 0.91 0.95 703 weighted avg 0.99 0.99 0.99 703

Berdasarkan hasil analisis pada tabel 3.1.1, perbandingan analisis prediksi algoritma KNN dengan jarak *Euclidean From Scratch & Scikit-learn,* ditunjukkan bahwa nilai akurasi yang dihasilkan oleh kedua model tersebut bernilai sama, yakni sebesar 98%. Namun, nilai F1 score (Macro Average) yang dihasilkan terdapat perbedaan, dimana nilai F1 score macro pada implementasi scratch sebesar 0.94, sedangkan pada implementasi library scikit-learn dihasilkan sebesar 0.94. Hal ini sudah sama antara scratch dan scikit learn.

3.1.2. Algoritma KNN dengan Jarak *Manhattan*

Berikut ini merupakan hasil analisis terhadap pengujian implementasi algoritma KNN menggunakan perhitungan jarak *manhattan* dengan nilai k = 5.

Tabel 3.1.2 Hasil Prediksi Algoritma KNN *Manhattan Scratch & Scikit – learn*

Scratch	*Scikit – learn*
Akurasi KNN dari scratch: 0.9914651493598862 precision recall f1-score support 0 1.00 0.89 0.94 53 1 0.99 1.00 1.00 650 accuracy 0.99 703 macro avg 1.00 0.94 0.97 703 weighted avg 0.99 0.99 0.99 703 F1 Score Macro: 0.9677029096477794	F1 Score Manhattan: 0.9677029096477794 Classification Report Manhattan: precision recall f1-score support 0 1.00 0.89 0.94 53 1 0.99 1.00 1.00 650 accuracy 0.99 703 macro avg 1.00 0.94 0.97 703 weighted avg 0.99 0.99 0.99 703

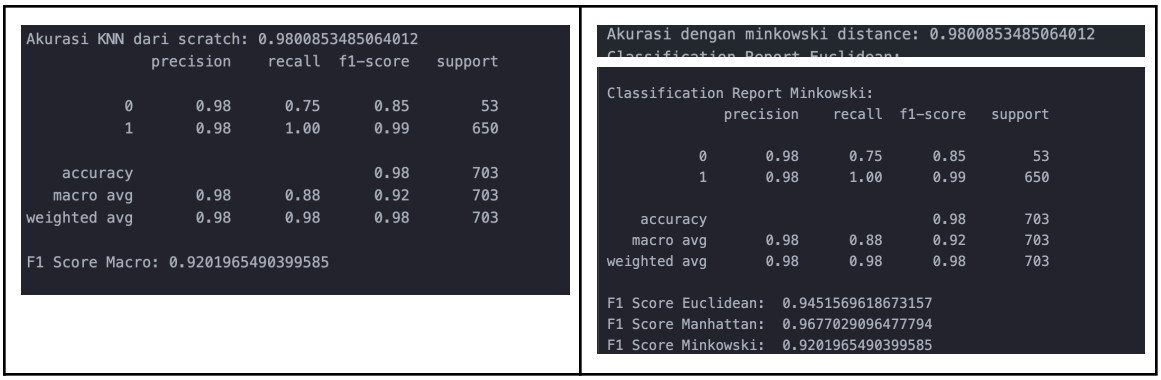
Berdasarkan hasil analisis pada tabel 3.1.2, perbandingan analisis prediksi algoritma KNN dengan jarak *manhattan from scratch & scikit – learn*, ditunjukkan bahwa nilai akurasi yang dihasilkan oleh kedua model tersebut berbeda cukup jauh, yakni pada algoritma *manhattan scratch* memiliki nilai 96.77%, sedangkan algoritma dari *scikit – learn* mendapat nilai 96.77%. Hal ini disebabkan bahwa algoritma KNN – *Manhattan* dinilai cukup efisien dibandingkan dengan algoritma KNN yang lain. Algoritma KNN *Manhattan* menghitung jarak antara 2 titik dengan menjumlahkan perbedaan absolut, sedangkan algoritma yang lain mengukur jarak menggunakan rumus *pythagoras* (untuk algoritma *euclidean*) dan algoritma *minkowski* yang cukup kompleks, sehingga dapat disimpulkan bahwa algoritma KNN – *Manhattan scikit - learn* cukup baik dibandingkan dengan algoritma KNN yang lain.

3.1.3. Algoritma KNN dengan Jarak *Minkowski*

Berikut merupakan hasil analisis terhadap pengujian implementasi algoritma KNN menggunakan perhitungan jarak *minkowski* dengan nilai k = 5.

Tabel 3.1.3 Hasil Prediksi Algoritma KNN *Minkowski Scratch & Scikit – learn*

Scratch	*Scikit – learn*



```
Akurasi KNN dari scratch: 0.9800853485064012
              precision    recall  f1-score   support

           0       0.98      0.75      0.85        53
           1       0.98      1.00      0.99       650

    accuracy                           0.98       703
   macro avg       0.98      0.88      0.92       703
weighted avg       0.98      0.98      0.98       703

F1 Score Macro: 0.9201965490399585
```

```
Akurasi dengan minkowski distance: 0.9800853485064012
Classification Report Minkowski:
              precision    recall  f1-score   support

           0       0.98      0.75      0.85        53
           1       0.98      1.00      0.99       650

    accuracy                           0.98       703
   macro avg       0.98      0.88      0.92       703
weighted avg       0.98      0.98      0.98       703

F1 Score Euclidean:  0.9451569618673157
F1 Score Manhattan:  0.9677029096477794
F1 Score Minkowski:  0.9201965490399585
```

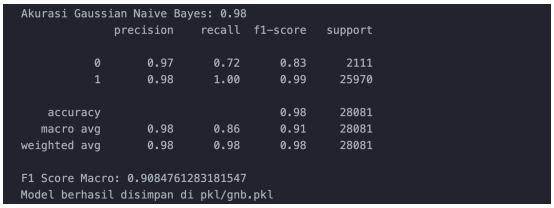
Berdasarkan hasil analisis pada tabel 3.3.1 perbandingan analisis prediksi algoritma KNN dengan jarak *Minkowski From Scratch & Scikit-learn,* ditunjukkan bahwa nilai akurasi yang dihasilkan oleh kedua model tersebut bernilai hampir mendekati, dimana pada implementasi algoritma KNN from *scratch* didapatkan sebesar 98% sedangkan pada implementasi *library scikit-lear*n didapatkan juga 98%. Nilai akurasi yang hampir mendekati tersebut juga menunjukkan bahwa performa kedua model tersebut konsisten terhadap hasil prediksi yang dibuat. Perbedaan kecil yang dihasilkan juga dapat disebabkan oleh adanya kesalahan teknis kecil, baik pada penerapan implementasi algoritma KNN *from scratch* ataupun penggunaan *library scikit-learn* tersebut. Namun, nilai F1 *score* (*Macro Average*) yang dihasilkan terdapat perbedaan, dimana nilai F1 *score macro* pada implementasi *scratch* sebesar 0.92, sedangkan pada implementasi *library scikit-learn* dihasilkan sebesar 0.92.

3.2. Analisis Hasil Prediksi Algoritma *Gaussian Naive Bayes From Scratch & Library Scikit – learn*

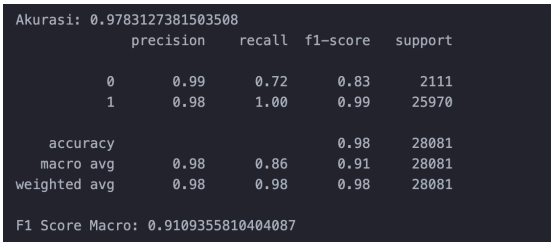
Berikut merupakan hasil analisis terhadap pengujian implementasi algoritma *Gaussian Naive Bayes*.

Tabel 3.2.1 Hasil Prediksi Algoritma KNN *Naive Bayes Scratch & Scikit – learn*

Scratch	***Scikit – learn***



```
Akurasi Gaussian Naive Bayes: 0.98
              precision    recall  f1-score   support

           0       0.97      0.72      0.83      2111
           1       0.98      1.00      0.99     25970

    accuracy                           0.98     28081
   macro avg       0.98      0.86      0.91     28081
weighted avg       0.98      0.98      0.98     28081

F1 Score Macro: 0.9084761283181547
Model berhasil disimpan di pkl/gnb.pkl
```



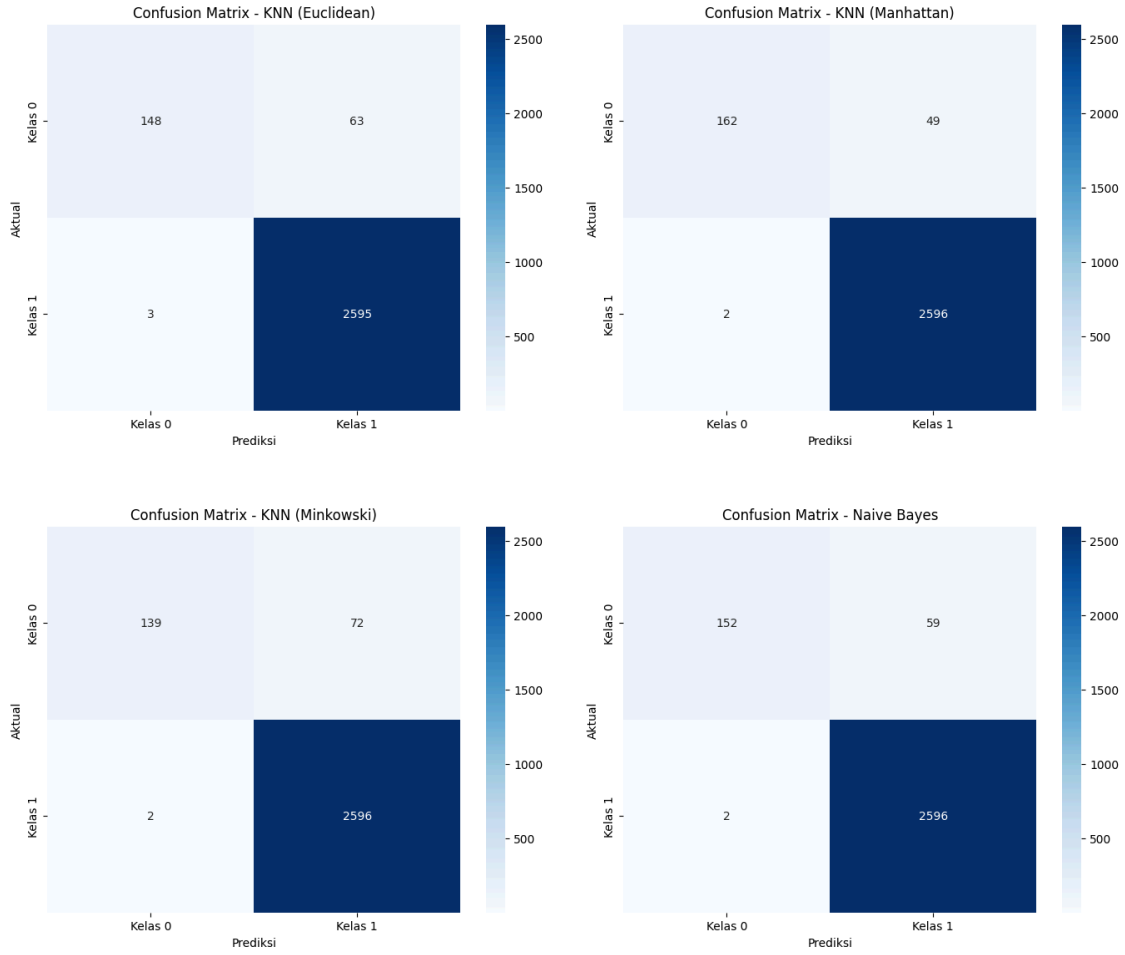
```
Akurasi: 0.9783127381503508
              precision    recall  f1-score   support

           0       0.99      0.72      0.83      2111
           1       0.98      1.00      0.99     25970

    accuracy                           0.98     28081
   macro avg       0.98      0.86      0.91     28081
weighted avg       0.98      0.98      0.98     28081

F1 Score Macro: 0.9109355810404087
```

Berdasarkan hasil analisis pada tabel 3.2.1 perbandingan analisis prediksi algoritma GNB, ditunjukkan bahwa nilai akurasi yang dihasilkan oleh kedua model tersebut bernilai sedikit berbeda, yakni sebesar 98% dan 98% kalau dibulatkan. Namun, nilai F1 *score* (Macro Average) yang dihasilkan terdapat perbedaan, dimana nilai F1 score macro pada implementasi scratch sebesar 0.9084, sedangkan pada implementasi library *scikit-learn* dihasilkan sebesar 0.9109. Hal tersebut menunjukkan adanya sedikit perbedaan akan tetapi tetap menunjukkan nilai prediksi yang hampir sama atau mendekati dari hasil perbandingan kedua implementasi algoritma GNB tersebut. Nilai akurasi yang hampir mendekati tersebut juga menunjukkan bahwa performa kedua model tersebut konsisten terhadap hasil prediksi yang dibuat. Perbedaan kecil yang dihasilkan juga dapat disebabkan oleh adanya kesalahan teknis kecil dan perbedaan dalam pengelolaan data.

BAB IV

ERROR ANALYSIS & IMPROVEMENTS

4.1. Analisis Kesalahan dan *Improvements*



Kami melakukan analisis performa model klasifikasi dengan *confusion matrix*. Di sini kami menguji model KNN dengan metrik *Euclidean*, *Manhattan*, *Minkowski*, dan *Gaussian Naive Bayes* (GNB). Hasilnya menunjukkan bahwa model KNN dengan metrik *Manhattan* memberikan performa terbaik dengan *True Negative* (TN) sebanyak 162 dan *True Positive* (TP) sebanyak 2596, serta nilai *False Positive* (FP) dan *False Negative* (FN) yang paling rendah, yaitu 49 dan 2. Hal ini menunjukkan bahwa model tersebut lebih akurat dan konsisten dalam memprediksi kelas positif dan negatif.

Walaupun demikian nilai *False Positive* pada model masih tergolong tinggi untuk setiap model, mengindikasikan kecenderungan model tersebut untuk keliru

mengklasifikasikan data negatif sebagai positif. Untuk meningkatkan kinerja model di masa depan, kami merekomendasikan implementasi teknik handling imbalance seperti SMOTE, penambahan data latih, dan penggunaan *ensemble model* yang dapat menggabungkan kekuatan berbagai metode klasifikasi untuk menghasilkan prediksi yang lebih akurat.

PEMBAGIAN TUGAS ANGGOTA KELOMPOK

NIM	Nama	Tugas
18222032	Clement Nathanael Lim	1. Membuat algoritma KNN *from scratch*. 2. Menguji KNN dengan *scikit – learn*. 3. Mengerjakan laporan.
18222048	Mattheuw Suciadi Wijaya	1. Membuat implementasi *feature engineering*. 2. Membuat implementasi algoritma *Gaussian Naive-Bayes from scratch.* 3. Mengerjakan laporan.
18222064	Hartanto Luwis	1. Membuat *data preprocess*. 2. Men–*debug* untuk meningkatkan akurasi. 3. Membuat algoritma *Gaussian Naive-Baye*s from *scikit-learn* 4. Mengerjakan laporan.
18222096	Farah Aulia	1. Membuat *pre-processing* data *pipeline* data. 2. Menguji knn dengan *scikit – learn*. 3. Membuat *error analysis* dan *improvements*. 4. Mengerjakan laporan.

REFERENSI

1.6. Nearest Neighbors. (n.d.). Scikit-learn. https://scikit-learn.org/1.5/modules/neighbors.html

1.9. Naive Bayes. (n.d.). Scikit-learn. https://scikit-learn.org/1.5/modules/naive_bayes.html

Prashant. (2020, August 28). *Naive Bayes Classifier in Python*. Kaggle. https://www.kaggle.com/code/prashant111/naive-bayes-classifier-in-python#Gaussian-Na%C3%AFve-Bayes-algorithm

Prasad, A., & Chandra, S. (2023). PhiUSIIL: A diverse security profile empowered phishing URL detection framework based on similarity index and incremental learning. *Computers & Security*, *136*, 103545. https://doi.org/10.1016/j.cose.2023.103545

UCI Machine Learning Repository. (n.d.). https://archive.ics.uci.edu/dataset/967/phiusiil+phishing+url+dataset