Tugas Besar 2 IF3070 Dasar Intelegensi Artifisial

Implementasi Algoritma Pembelajaran Mesin

Disusun Oleh:

Kelompok 13

Jonathan Wiguna	18222019
Naomi Pricilla Agustine	18222065
Harry Truman Suhalim	18222081
Micky Valentino	18222093

Program Studi Sistem dan Teknologi Informasi

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

DAFTAR ISI

BAB I
IMPLEMENTASI ALGORITMA

1.1 Implementasi Algoritma KNN

Algoritma KNN merupakan sebuah algoritma untuk klasifikasi dan regresi dengan memprediksi label suatu data berdasarkan jumlah tetangga terdekat dalam suatu dataset dan data yang serupa akan memiliki hasil yang juga serupa. KNN bekerja dengan cara menyimpan dataset *train* tanpa melakukan proses latihan, menghitung jarak antara dataset *train* dengan dataset yang *test*, dan menentukan label berdasarkan label mayoritas untuk klasifikasi atau berdasarkan rata-rata label untuk regresi dari tetangga terdekat.

Pada kelas KNNClassifier yang kami buat, kami mengimplementasikan algoritma KNN untuk memprediksi label dengan melakukan 2 metode yaitu *fit* dan *predict*/ Di awal, inisialisasi model dilakukan untuk menentukan jumlah tetangga yang akan dipertimbangkan, cara menghitung jarak, dan parameter yang akan digunakan untuk menghitung jarak dengan Minkowski. Selanjutnya, metode *fit* digunakan hanya bertujuan untuk menyimpan dataset *train* sebagai referensi untuk proses prediksi. Selain itu, kami juga membuat 3 fungsi untuk menghitung jarak antara data dalam *train* dan *test* yaitu _euclidean_distance, _manhattan_distance, dan _minkowski_distance yang digunakan dalam _calculate_distances. Setelah mendapatkan jarak, fungsi _get_neighbors akan digunakan untuk memilih sejumlah tetangga terdekat dan fungsi _get_majority_vote digunakan untuk menentukan label mayoritas dari sejumlah tetangga terdekat untuk klasifikasi. Setelah itu, prediksi akan dilakukan dengan memproses semua data *test* mulai dari menghitung jarak ke semua data *train*, mencari sejumlah tetangga terdekat, dan terakhir menentukan hasil prediksi berdasarkan label mayoritas tetangga.

Berikut merupakan hasil dari implementasi algoritma KNN yang telah kami buat.

```
class KNNClassifier(BaseEstimator):
    def __init__(self, k=3, metric='euclidean', p=2):
        self.k = k
        self.metric = metric
        self.p = p
```

```
    def fit(self, X, y):
        self.X_train = X.to_numpy()
        self.y_train = y.to_numpy()
        return self

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

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

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

    def _calculate_distances(self, x_test):
        distances = []
        for x_train in self.X_train:
            if self.metric == 'euclidean':
                dist = self._euclidean_distance(x_test, x_train)
            elif self.metric == 'manhattan':
                dist = self._manhattan_distance(x_test, x_train)
            elif self.metric == 'minkowski':
                dist = self._minkowski_distance(x_test, x_train, self.p)
            else:
                        raise ValueError("Invalid metric. Use 'euclidean',
'manhattan', or 'minkowski'.")
            distances.append(dist)
        return np.array(distances)

    def _get_neighbors(self, x_test):
        distances = self._calculate_distances(x_test)
        k_indices = np.argsort(distances)[:self.k]
        return self.y_train[k_indices]

    def _get_majority_vote(self, labels):
        labels = list(labels)
        unique_labels = list(set(labels))
        votes = [labels.count(label) for label in unique_labels]
        return unique_labels[np.argmax(votes)]
```

```
    def predict(self, X):
        X_test = X.to_numpy()
        predictions = []
        for x in X_test:
            neighbors = self._get_neighbors(x)
            majority_label = self._get_majority_vote(neighbors)
            predictions.append(majority_label)
        return np.array(predictions)
```

1.2 Implementasi Algoritma Naive-Bayes

Algoritma Naive Bayes merupakan sebuah algoritma klasifikasi berbasis probabilitas yang didasarkan pada Teorema Bayes. Pada kelas GaussianNaiveBayes yang kami buat, kami mengimplementasikan algoritma Naive Bayes untuk memprediksi label dengan melakukan metode fit dan predict. Metode fit bertujuan untuk melatih model dengan cara menghitung *mean* dan standar deviasi untuk setiap fitur di setiap nilai uniknya, serta peluang awal berdasarkan proporsi data terhadap nilai uniknya. Setelah itu, dilakukan perhitungan probabilitas menggunakan rumus distribusi Gaussian. Setelah melakukan fit, prediksi dilakukan dengan menghitung skor probabilitas setiap unique value dengan menambahkan peluang awal dan peluang dari fitur - fitur data. Unique value yang memiliki skor tertinggi dipilih sebagai prediksi. Terdapat juga beberapa algoritma tambahan untuk memberikan batas minimum agar mencegah nilai nol dan untuk menghindari kesalahan perhitungan pada angka yang sangat kecil.

Berikut merupakan hasil dari implementasi algoritma Naive Bayes yang telah kami buat.

```
class GaussianNaiveBayes(BaseEstimator):
    def fit(self, X, y):
        self.classes = np.unique(y)
        self.mean_std = {}
        self.prior = {}

        for c in self.classes:
            X_c = X[y == c]
            self.mean_std[c] = {
                'mean': np.mean(X_c, axis=0),
```

```
                        'std': np.where(np.std(X_c, axis=0) == 0, 1e-6,
np.std(X_c, axis=0))
            }
            self.prior[c] = len(X_c) / len(y)
        return self

    def _gaussian_probability(self, x, mean, std):
        epsilon = 1e-9
        exponent = np.exp(-((x - mean) ** 2) / (2 * std ** 2))
        prob = exponent / (np.sqrt(2 * np.pi) * std)
        return np.maximum(prob, epsilon)

    def _predict_sample(self, x):
        probabilities = {}
        for c in self.classes:
            prior = np.log(self.prior[c])
            mean = self.mean_std[c]['mean']
            std = self.mean_std[c]['std']
            likelihood = np.sum(np.log(self._gaussian_probability(x, mean,
std)))
            probabilities[c] = prior + likelihood
        return max(probabilities, key=probabilities.get)

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

BAB II
CLEANING DAN PREPROCESSING

2.1 *Data Cleaning*

Data Cleaning adalah langkah awal yang sangat penting dalam mempersiapkan *dataset* untuk *machine learning*. *Data* mentah yang dikumpulkan dari berbagai sumber seringkali berantakan dan mengandung kesalahan, nilai yang hilang, dan juga ketidakkonsistenan. *Data Cleaning* melibatkan langkah-langkah sebagai berikut.

2.1.1 *Handling Missing Data*

Handling Missing Data dilakukan pada tahap *data cleaning* untuk mengidentifikasi dan menangani nilai yang hilang pada *dataset*. Hal ini mencakup imputasi nilai yang hilang, menghapus baris atau kolom *data* yang hilang berlebihan, ataupun penggunaan teknik seperti interpolasi.

Terdapat beberapa cara untuk mengatasi *missing values* seperti menghapus baris atau kolom dengan *missing values*, mengisi *missing values* dengan mean, median, atau modus, mengisi dengan nilai tetap yang ditentukan, mengisi dengan interpolasi yang memperkirakan nilai berdasarkan tren *data* di sekitarnya, menggunakan *forward fill* atau *backward* fill untuk mengisi *missing values* berdasarkan nilai sebelumnya atau setelahnya dalam kolom yang sama, serta dapat dilakukan pengisian *missing values* dengan prediksi (imputasi lanjutan), yaitu menggunakan model untuk prediksi nilai yang hilang berdasarkan nilai lain dalam *dataset*.

Sebelum menangani *missing values*, kami melakukan identifikasi persentase *missing values* setiap kolom terlebih dahulu dengan menggunakan (train_set.isnull().sum() / len(train_set)) * 100 untuk menentukan cara yang tepat untuk mengatasi *missing values* tersebut.

Kami menggunakan 2 metode imputasi dalam penanganan *missing values.* Pertama, metode imputasi dengan menerapkan pengisian nilai menggunakan statistik

deskriptif, dimana nilai median digunakan untuk mengisi nilai pada kolom numerik dan *count-based features* yang tidak memiliki keterkaitan dengan kolom lain, sedangkan nilai modus digunakan untuk mengisi nilai pada kolom *boolean* dan kategorikal yang tidak memiliki keterkaitan dengan kolom lain. Kedua, metode imputasi yang digunakan untuk mengisi kolom yang memiliki keterkaitan dengan kolom lain adalah metode imputasi berdasarkan aturan yang memanfaatkan logis antar variabel dalam *dataset*. Seperti pengisian URL dengan memanfaatkan informasi Domain dan IsHTTPS yang tersedia. Kedua metode ini dipilih dengan mempertimbangkan karakteristik data dan keterkaitan antar variabel dalam *dataset*, yang memungkinkan pengisian *missing values* lebih kontekstual dan bermakna.

2.1.2 *Dealing with Outliers*

Di dataset, untuk tiap kolomnya akan ada *outliers* yaitu nilai-nilai yang secara signifikan berbeda dengan data lainnya. *Outliers* di tugas besar kali ini di-*handle* menggunakan IQR dan Z-Score, yang tidak kalah penting adalah bagaimana perlu di-*handle* secara terpisah antara yang memiliki label 1 dan yang memiliki label 0; Dikarenakan ditemukan ketika *debugging* bahwa apabila tidak dipisah, semua label 0 akan menjadi 1, hal ini kemungkinan besar disebabkan karena distribusi label yang sangat tidak rata (97% - 1 : 3% - 0). *Handling Outliers* ini diterapkan di *validation set* dan juga *training set*.

Data yang *skewness features*-nya tinggi (>1 atau < -1) akan di-*handle* menggunakan IQR dengan alasan metode tersebut cocok digunakan untuk menangani distribusi data yang tidak normal (Karena di perhitungannya menggunakan *percentile*). Sedangkan, untuk *skewness* yang cenderung normal akan digunakan z-score. (Karena menggunakan deviasi dan mean).

Hal yang paling menarik ketika melakukan *data cleaning* tahapan ini adalah keterkaitannya terhadap model. Ketika *outliers* tidak di-*handle Naive-Bayes* memiliki hasil yang lebih baik dibanding KNN, dan kebalikannya ketika di-*handle* KNN menjadi model yang lebih baik dibanding *Naive-Bayes*. Hal ini ternyata dikarenakan ketika

outliers ter-*handle* dengan baik, KNN memiliki konsep menghitung jarak antar poin, ketika ada *outliers* yang parah, kalkulasi jarak ini akan juga ikut rusak. Ketika *outliers*-nya sudah dihilangkan maka yang terjadi adalah kalkulasi jarak menjadi lebih *reliable.* Selanjutnya, untuk *Naive-Bayes* menggunakan konsep *distribution* dari sebuah *feature* sehingga ketika *outliers* dihilangkan justru yang terjadi adalah menghilangkan informasi terkait sebaran data-nya yang menyebabkan hasilnya menjadi lebih buruk.

2.1.3 *Remove Duplicates*

Remove Duplicates adalah metode yang dilakukan untuk mengidentifikasi dan menghilangkan baris yang duplikat. Penanganan duplikat ini sangat penting karena dapat membahayakan integritas data, yang dapat menyebabkan analisis menjadi tidak akurat. Duplikat dapat menyebabkan *overfitting* dan mengurangi kemampuan untuk generalisasi pada *data* baru. Selain itu, juga meningkatkan biaya komputasi dan waktu pemrosesannya. Menghapus duplikat dapat memastikan kualitas data untuk analisis yang akurat, efisien, dan konsisten.

Duplikat harus dihilangkan untuk menghindari distorsi statistik (tidak akurat), mengurangi *overfitting* dalam model, efisiensi pengolahan data, dan menghindari bias dalam hasil analisis. Kami menggunakan duplicated().sum() untuk mengidentifikasi baris yang duplikat pada *Data Frame* dan menjumlahkannya. Setelah itu, kami menghapus duplikat dengan menggunakan drop_duplicates() untuk menghapus semua baris yang duplikat dan hasilnya disimpan kembali ke *Data Frame*.

2.1.4 *Feature Engineering*

Feature Engineering adalah proses pembuatan fitur baru atau transformasi fitur yang ada untuk meningkatkan performa model *machine learning*. *Feature engineering* bertujuan untuk mempelajari pola dan membuat prediksi yang akurat dari data. *Feature engineering* memerlukan pemahaman mendalam tentang *data*, pengetahuan domain, dan eksperimen untuk menentukan fitur mana yang dapat meningkatkan kemampuan prediktif model.

Untuk penghapusan fitur dibagi menjadi beberapa kriteria, fitur yang memiliki korelasi tinggi dengan URLLength yang berarti memberikan informasi yang redundan, fitur-fitur yang memiliki nilai *variance* 0 yang berarti tidak memberikan informasi untuk model, *raw text* dan *derived features* dihapus karena *raw text* sudah diproses menjadi fitur lain dan *derived* sudah tercakup pada fitur lainnya, terakhir ada tingkat *missing values* yang tinggi dan *importance* yang rendah dihapus karena banyaknya nilai yang hilang dan berkontribusi kecil.

Kami membuat beberapa fitur baru berupa UIQualityScore untuk menentukan kualitas UI berdasarkan HasFavicon, HasSubmitButton, HasTitle, HasDescription, dan IsResponsive, FinancialRiskScore yang dinilai berdasarkan Bank, Crypto, Pay, dan IsHTTPS, DomainTrustScore yang dinilai skor domain berdasarkan TLDLegitmateProb, IsHTTPS, dan IsDomainIP, ContentComplexity yang mengukur kompleksitas kode berdasarkan LineOfCode dan LargestLineLength, serta RefPatternScore yang menilai rasio NoOfSelfRef terhadap total referensi (NoOfSelfRef + NoOfEmptyRef).

Lalu dilakukan analisis korelasi untuk tiap fiturnya dengan label, menampilkan statistik deskriptif fitur baru, dan juga mengecek distribusi label. Setelah itu dipilih 3 fitur yang memiliki korelasi paling tinggi karena dianggap paling relevan dan dilakukan penghapusan pada fitur yang memiliki korelasi rendah dengan label.

2.2 Data Preprocessing

Data preprocessing merupakan sebuah proses yang melakukan transformasi tambahan pada data untuk memastikan data siap digunakan oleh algoritma pembelajaran mesin. Terdapat beberapa langkah yang dilakukan dalam proses *data preprocessing* yaitu sebagai berikut.

2.2.1 *Feature Scaling*

Feature Scaling dilakukan dalam *data processing* dengan tujuan untuk mengatur rentang nilai fitur-fitur numerik dalam dataset agar seluruh fitur terdistribusi dengan seimbang dan tidak ada fitur dengan skala yang terlalu berbeda yang dapat memengaruhi hasil model algoritma *machine learning* menjadi tidak akurat.

Metode *feature scaling* yang kami coba adalah *standardization* dan *normalization*. *Standardization* dilakukan dalam *data preprocessing* dengan tujuan untuk mengubah data khususnya data numerik sehingga memiliki distribusi dengan mean = 0 dan standar deviasi = 1 agar memastikan seluruh fitur memiliki kontribusi yang seimbang sehingga hasil model algoritma *machine learning* menjadi lebih akurat. *Standardization* diterapkan menggunakan metode *Z-Score Scaling* dengan *class NumericStandardScaler* yang menggunakan *StandardScaler* dari *scikit-learn* untuk melakukan transformasi.

Normalization dilakukan dalam *data processing* dengan tujuan untuk mengubah data khususnya data numerik ke dalam skala tertentu yaitu dalam rentang antara 0 dan 1 sehingga data dapat terdistribusi dengan seragam dan tidak ada data dengan skala besar yang dapat mendominasi hasil model algoritma *machine learning*. *Normalization* diterapkan menggunakan metode *Min-Max Scaling* dengan *class NumericMinMaxScaler* yang menggunakan *MinMaxScaler* dari *scikit-learn* untuk melakukan transformasi.

Setelah mencoba kedua teknik tersebut, *normalization* lebih cocok digunakan pada dataset ini dibandingkan *standarization* dikarenakan *normalization* memastikan data berada dalam rentang yang sama, yaitu 0 hingga 1.

2.2.2 *Feature Encoding*

Metode *feature encoding* yang kami coba adalah *one hot encoding* dan *frequency encoding* dengan tujuan untuk mengubah data khususnya data objek atau kategorik yaitu TLD menjadi nilai numerik.

Pada *one hot encoding,* perubahan data dilakukan dengan setiap kategori direpresentasikan dengan satu elemen bernilai 1 dan sisanya 0. Teknik ini mencegah algoritma mengasumsikan adanya hubungan ordinal antar kategori.

Pada *frequency encoding*, perubahan data dilakukan berdasarkan berdasarkan pola distribusi frekuensi kemunculan kategori tersebut dalam dataset agar hasil model

algoritma *machine learning* menjadi lebih akurat dengan adanya informasi numerik dari data objek atau kategorikal yang dapat digunakan oleh model. Kategori yang muncul lebih sering dalam dataset akan mendapatkan hasil nilai dari *frequency encoding* yang lebih tinggi dibandingkan dengan kategori yang jarang muncul.

Berdasarkan percobaan tersebut, *frequency encoding* lebih cocok digunakan pada dataset ini dikarenakan pada kolom TLD yang dilakukan *encoding*, terdapat unique value yang banyak sehingga jika dilakukan *one hot encoding* maka kolom yang dihasilkan terlalu banyak.

2.2.3 *Handling Imbalanced Dataset*

Dataset bisa dikatakan *Imbalanced* apabila di sebuah dataset tersebut ada *majority class* dan *minority class*. Hal pertama yang harus dilakukan apakah sebuah data imbalanced atau tidak adalah dengan mencari terlebih dahulu distribusi saat ini.

```
Label Distribution BEFORE SMOTETomek:
       Count  Percentage (%)
label
0       5960           6.39
1      87305          93.61

Label Distribution BEFORE SMOTETomek:
       Count  Percentage (%)
label
0       2801           7.16
1      36305          92.84
```

Bisa dilihat bahwa distribusinya cukup ekstrem dimana label 0 memiliki jumlah yang sangat kecil (6-7%) dibandingkan dengan label 1 (92-94%). Karena distribusi tersebut label 0 adalah *minority class* di dataset *phishing* ini dan label 0 adalah *majority class*. Hal pertama yang terepikirkan oleh kami adalah untuk melakukan *oversampling*

dengan menggunakan SMOTE biasa, dimana akan dibentuk row baru/sintetis untuk label 0 dengan konsep KNN sampai jumlah label 0 sama dengan label 1. Tetapi setelah dipikir-dipikir pembuatan row yang sangat banyak ini sangat ekstrem dan kemungkinan akan menyebabkan model yang kurang baik. Oleh karena itu, setelah *browsing* kami menemukan ada alternatif lain dari SMOTE yaitu SMOTETomek dimana akan digabungkan konsep SMOTE dan konsep yang dinamakan Tomek Links. Penjelasan singkat terkait 2 konsep tersebut adalah sebagai berikut.

1. SMOTE : Model ML akan mengambil *nearest neighbour* dengan jumlah sesuai dengan *parameter* yang diinginkan (*default*-nya adalah 5) dari sebuah *node minority class* lalu diantara *node* yang diambil tersebut dan *neighbour*-nya akan dibuat sebuah dummy data
2. Tomek Links : Model ML akan mengambil sepasang *node minority class* dan *majority class* yang terdekat satu sama lain lalu menghapus *node majority class*.

Dari penjelasan tersebut bisa disimpulkan bahwa sebenarnya SMOTETomek ini melakukan 2 jenis sampling bersamaan yaitu *oversampling* menggunakan SMOTE dan undersampling menggunakan Tomek Links. Berikut adalah hasil setelah SMOTETomek dijalankan.

```
Label Distribution BEFORE SMOTETomek:
       Count  Percentage (%)
label
0       5960            6.39
1      87305           93.61

Label Distribution BEFORE SMOTETomek:
       Count  Percentage (%)
label
0       2801            7.16
1      36305           92.84
```

```
Label Distribution AFTER SMOTETomek:
       Count  Percentage (%)
label
0      16681           16.16
1      86525           83.84

Label Distribution AFTER SMOTETomek:
       Count  Percentage (%)
label
0       6815           15.97
1      35859           84.03
```

Persentase *rows* dengan label 0 meningkat cukup signifikan, yaitu bertambah sekitar 10%. Sebenarnya persentase ini merupakan salah satu parameter yang bisa di-*input* di fungsi SMOTETomek untuk menentukan seberapa besar distribusi dari

minority class di akhir. Selanjutnya bisa dilihat bahwa jumlah *rows* dengan label 1 juga berkurang walaupun tidak begitu signifikan, pengurangan ini adalah *undersampling* yang dilakukan dengan Tomek Links.

BAB III
EVALUASI HASIL

Evaluasi hasil ini didapatkan dari hasil yang di-*testing* secara lokal (tanpa test.csv) dan yang di-*submit* ke *Kaggle*(dengan test.csv). Pertama-tama yang paling *notable* dan pertama kali kami temukan perbedaannya adalah terkait peran *outliers* ke hasil akhir dari model. Seperti yang sebelumnya dijabarkan di bagian 2.1.2 : Dealing with Outliers kami menemukan bahwa setelah dilakukannya penanganan dari *outliers* tersebut hasil dari *model Naive Bayes* menjadi lebih buruk sedangkan model *K-Nearest-Neighbour* (KNN) menjadi sedikit lebih baik. Hal ini disebabkan oleh konsep dasar KNN yang menghitung jarak antar titik data. Ketika *outliers* tidak ditangani dengan baik, jarak yang dihitung menjadi tidak akurat karena adanya *outliers* yang ekstrem, yang dapat merusak kalkulasi jarak secara keseluruhan. Dengan menghilangkan *outliers*, kalkulasi jarak menjadi lebih baik dan lebih merepresentasikan hubungan antar data dengan lebih baik. Sebaliknya, pada algoritma *Naive Bayes* yang menggunakan konsep distribusi fitur, penghilangan *outliers* dapat menghilangkan informasi penting mengenai sebaran data. Hal ini menyebabkan model kehilangan pemahaman tentang variasi alami dari data, sehingga performa *Naive Bayes* dapat menurun setelah *outliers* dihilangkan. Berikut adalah data yang didapatkan ketika dites secara lokal :

Dengan *outlier handling* :

KNN Metrics:

- Accuracy: 0.961607430226677
- Precision: 0.962369039982895
- Recall: 0.9927763992280121
- F1 Score: 0.9773362646907147

Naive Bayes Metrics:

- Accuracy: 0.9024552853004736
- Precision: 0.9906547783190857
- Recall: 0.8914254204576785
- F1 Score: 0.9384242529786524

Tanpa *outlier handling* :

KNN Metrics:

- Accuracy: 0.9041559173585969
- Precision: 0.9634645072177234
- Recall: 0.9205092376960099
- F1 Score: 0.9414971747674311

Naive Bayes Metrics:

- Accuracy: 0.9465389034621805
- Precision: 0.9742803209060877
- Recall: 0.9615742897065673
- F1 Score: 0.9678856071261135

Namun, ketika melakukan *submission* ke *Kaggle*, dimana kami menggunakan *Naive Bayes* dan melakukan *handling outliers,* skor akhir yang didapatkan cukup buruk dengan sekitar 0.75. Selanjutnya, kami melakukan *submission* ulang dengan menggunakan model yang sama yaitu *Naive Bayes* tanpa melakukan *handling outliers* sama sekali dan mendapatkan nilai terbaik sekitar 0.92. Hal ini berarti terjadi *overfitting* yang cukup signifikan di bagian *Naive Bayes* yang dilakukan *handling outliers.*

Selanjutnya, penjelasan terkait keterkaitan antara parameter sampling_rate di SMOTETomek, di *submission* dengan nilai tertinggi sebelumnya, sampling_rate yang digunakan adalah 0.2 dimana berarti *minority class* yaitu label 0 akan dilakukan *oversampling* sampai distribusinya mencapai 0.2/20% dari keseluruhan distribusi data; untuk mengetes apakah sampling_rate ini berdampak cukup besar atau tidak ke hasil akhir, kami mencoba menggantinya menjadi 0.4 dimana menyebabkan pengurangan skor akhir *Kaggle* menjadi 0.90, hal ini kemungkinan besar disebabkan karena pembuatan *rows dummy* yang terlalu banyak sehingga prediksi yang dibuat menjadi kurang tepat.

Terkait *feature scaling*, proses tersebut dapat membantu beberapa performa model - model tertentu. Pada kasus KNN, *feature scaling* sangat membantu performa model karena model KNN berbasis jarak, sehingga *scaling* akan membuat jarak di setiap fitur menjadi sama.

Hal ini juga terlihat setelah melakukan testing dimana akurasi model yang mencapai 98% saat menggunakan *feature scaling*, sedangkan jika tidak menggunakan feature scaling, akurasi model hanya 79%. Berbeda hal dengan kasus *Naive Bayes*, feature scaling tidak terlalu membantu performa model. Hal ini dikarenakan model Naive Bayes tidak berbasis jarak atau magnitudo fitur . Setelah melakukan testing, akurasi model menggunakan feature scaling adalah 96% sedangkan tanpa feature scaling adalah 94%.

Terakhir, penjelasan terkait modeling from scratch dan scikit implementasi dimana *Naive Bayes* baik dari *scratch* maupun menggunakan *library* scikit-learn menunjukkan performa yang identik, yang memiliki arti bahwa implementasi from *scratch* sudah mengikuti prinsip algoritma Gaussian Naive Bayes (yang digunakan untuk *testing)* dengan benar. Perbedaan yang cukup menarik terlihat pada implementasi *K-Nearest Neighbors* (KNN), model yang dibuat dari *scratch* memerlukan waktu yang sangat lama (sekitar 1 jam) untuk mendapatkan hasil akhir dibandingkan dengan implementasi *scikit-learn* yang jauh lebih optimal (hanya perlu beberapa detik) . Hal ini menunjukkan bahwa library scikit-learn meningkatkan efisiensi dan akurasi model KNN melalui penggunaan struktur data yang efisien dan algoritma yang dioptimalkan, dimana hal ini akan sangat terasa ketika dataset yang digunakan besar.

BAB IV
PEMBAGIAN TUGAS

Nama	NIM	Tugas
Jonathan Wiguna	18222019	1. Kode : Feature Engineering, Outliers, Data Imbalance 2. Dokumen : Outliers, Data Imbalance, Hasil Evaluasi
Naomi Pricilla Agustine	18222065	1. Kode: Remove Duplicates, Feature Scaling (MinMaxScaler), KNN from Scratch & from Scikit-Learn, Save & Load Model 2. Dokumen: Algoritma KNN, Feature Scaling, Feature Encoding
Harry Truman Suhalim	18222081	1. Kode : Handling Missing Data, Feature Scaling (NumericStandardScaler), Error Analysis 2. Dokumen : Handling Missing Data, Remove Duplicates, Feature Engineering
Micky Valentino	18222093	1. Kode: Feature Encoding, Compile Preprocessing Pipeline, Cleaning & preprocessing test_df, Submission, Naive Bayes from Scratch & from Scikit-Learn 2. Dokumen: Algoritma Naive-Bayes

REFERENSI

Handling Missing Values in Pandas Dataframe | GeeksforGeeks

SMOTE: Oversampling for Class Imbalance

Tomek links Algorithm – Undersampling to handle Imbalanced data in machine learning b…

1. Solved Numerical Example of KNN Classifier to classify New Instance IRIS Example …

2. Solved Example KNN Classifier to classify New Instance Height and Weight Example …

https://www.geeksforgeeks.org/save-and-load-machine-learning-models-in-python-with-scikit-learn/

https://www.geeksforgeeks.org/information-gain-and-mutual-information-for-machine-learning/

https://kozodoi.me/blog/20230319/knn-from-scratch

https://machinelearningmastery.com/naive-bayes-classifier-scratch-python/