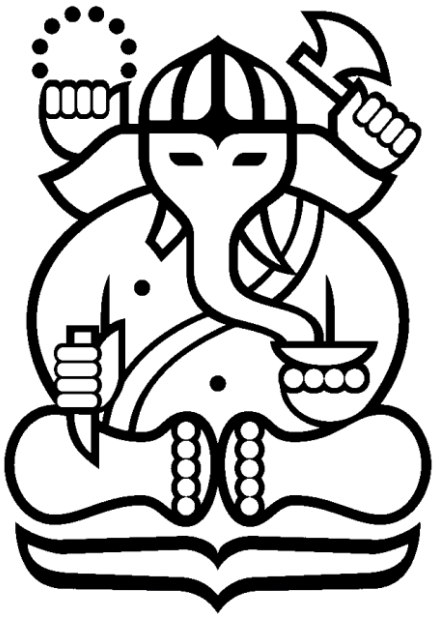
TUGAS BESAR 2
Implementasi Algoritma Pembelajaran Mesin

IF3070
Dasar Inteligensi Buatan



Disusun oleh:

Audra Zelvania Putri Harjanto / 18222106

Rizqi Andhika Pratama / 18222118

Sekar Anindita Nurjadini / 18222125

Khayla Belva Annandira / 18222138

Institut Teknologi Bandung

2024

Daftar Isi

Daftar Gambar

1. Data Cleaning & Pre-Processing

1.1 Handling Missing Data

Data yang hilang dapat menyebabkan bias atau kesalahan dalam analisis maupun model prediktif, sehingga penanganannya menjadi langkah penting dalam proses data cleaning. Dalam kode ini, penanganan data yang hilang dilakukan dengan membedakan antara data kategorik dan data numerik. Fitur kategorik, seperti FILENAME, URL, Domain, TLD, dan Title, diimputasi menggunakan modus, yaitu kategori yang paling sering muncul dalam kolom tersebut. Pendekatan ini dipilih karena modus dapat merepresentasikan nilai yang paling umum dalam data kategorik, sehingga menjaga konsistensi.

Sementara itu, fitur numerik diimputasi dengan nilai median. Pemilihan median dilakukan karena median tidak mudah terpengaruh oleh nilai ekstrem atau outlier dibandingkan dengan mean, yang dapat terpengaruh secara signifikan oleh nilai ekstrem. Dengan mengganti nilai yang hilang menggunakan median, distribusi data numerik tetap terjaga dan model lebih stabil dalam melakukan prediksi.

Setelah tahap imputasi, transformasi dilakukan untuk menggantikan nilai-nilai yang hilang dalam dataset dengan nilai yang telah dihitung sebelumnya.

1.2 Dealing with Outliers

Data outlier dapat mengganggu distribusi data, mengurangi kinerja model, dan memberikan hasil yang bias. Oleh karena itu, penanganan outlier menjadi langkah penting dalam proses preprocessing data. Dalam kode ini, outlier dideteksi menggunakan metode Interquartile Range (IQR), dengan batas bawah dan atas dihitung berdasarkan kuartil pertama (Q1) dan kuartil ketiga (Q3), dengan penyesuaian menggunakan parameter multiplier.

Pendekatan ini efektif untuk mendeteksi nilai-nilai yang secara signifikan berbeda dari mayoritas data.

Setelah outlier terdeteksi, terdapat beberapa strategi untuk menanganinya, seperti clipping, imputation dengan mean atau median, yang dapat dipilih berdasarkan kebutuhan analisis. Strategi clipping membatasi nilai outlier ke dalam batas atas dan bawah yang telah dihitung, sehingga distribusi data tetap berada dalam rentang yang lebih wajar. Alternatif lain, seperti imputasi dengan mean atau median, menggantikan nilai outlier dengan nilai yang lebih representatif dari distribusi data, dan menjaga konsistensi.

1.3 Remove Duplicates

Data duplikat dapat menyebabkan bias dalam analisis dan model prediktif karena entri yang sama dihitung lebih dari sekali, sehingga memengaruhi distribusi data dan hasil analisis. Dalam konteks pembelajaran mesin, data duplikat pada training set dapat mengarahkan model untuk belajar pola yang tidak representatif, yang berisiko menyebabkan overfitting. Hal ini mengurangi kemampuan model untuk melakukan generalisasi terhadap data baru, yang merupakan tujuan utama dari pelatihan model. Oleh karena itu, menghapus data duplikat pada training set menjadi langkah penting untuk memastikan model menerima informasi yang akurat dan tidak berulang.

Namun, data duplikat pada validation set sering kali dibiarkan apa adanya karena tujuan dari validation set adalah untuk mengevaluasi performa model pada data yang mendekati kondisi nyata. Langkah-langkah untuk menangani data duplikat, seperti yang diimplementasikan dalam kode, memastikan bahwa dataset menjadi lebih bersih, efisien, dan dapat diandalkan untuk analisis maupun pelatihan model. Hal ini penting untuk meningkatkan kualitas hasil analisis dan performa model secara keseluruhan.

1.4 Feature Engineering

Menambah atau memodifikasi fitur merupakan langkah penting dalam feature engineering untuk meningkatkan kemampuan model dalam memahami pola dan membuat prediksi yang lebih akurat. Pada implementasi ini, dilakukan penambahan empat kolom baru yang dirancang khusus untuk membantu model mendeteksi pola-pola yang relevan dengan phishing.

Kolom pertama, capital_ratio, dihitung berdasarkan proporsi huruf kapital dalam sebuah URL. Kolom kedua, contains_phishing_keywords, adalah fitur biner yang menunjukkan keberadaan kata kunci yang sering terkait dengan phishing, seperti "login," "secure," atau "verify." Dengan menambahkan informasi ini, model dapat lebih mudah mengenali URL yang mengandung elemen-elemen mencurigakan.

Kolom ketiga, url_segment_count, menghitung jumlah segmen pada URL berdasarkan jumlah tanda garis miring ('/'). Kolom terakhir, special_char_ratio, menghitung proporsi karakter khusus dalam URL. Kehadiran karakter khusus yang berlebihan sering menjadi ciri URL phishing, sehingga informasi ini dapat membantu model mengenali anomali pada pola URL.

1.5 Feature Scaling

Feature scaling adalah langkah penting dalam preprocessing data untuk memastikan bahwa semua fitur memiliki skala yang serupa, sehingga setiap fitur berkontribusi secara setara terhadap proses pelatihan model. Ketidakseimbangan skala fitur dapat menyebabkan algoritma tersebut tidak optimal untuk menemukan pola dalam data.

Salah satu metode yang umum digunakan adalah Min-Max Scaling, yaitu nilai fitur disesuaikan ke dalam rentang tertentu, biasanya antara 0 dan 1. Metode ini cocok untuk data dengan distribusi yang seragam. Alternatif

lainnya adalah Standardization atau Z-score Scaling, yang menyesuaikan fitur agar memiliki rata-rata nol dan standar deviasi satu. Pendekatan ini efektif ketika data memiliki distribusi Gaussian atau simetris.

Selain itu, Robust Scaling adalah pilihan yang ideal untuk data yang mengandung outlier, karena metode ini menggunakan IQR (Interquartile Range) daripada rata-rata dan standar deviasi, sehingga pengaruh outlier diminimalkan. Sementara itu, Log Transformation dapat digunakan untuk menstabilkan variansi dan membuat distribusi data lebih normal.

1.6 Feature Encoding

Feature encoding, atau pengkodean fitur, adalah proses mengubah data kategorikal (non-numerik) menjadi format numerik yang dapat digunakan oleh algoritma pembelajaran mesin. Model pembelajaran mesin umumnya memerlukan data numerik untuk dapat melakukan pemrosesan dan prediksi. Oleh karena itu, setiap kolom data kategorikal perlu diubah menjadi nilai numerik agar model dapat memanfaatkannya dengan baik.

Dalam kode yang diberikan, fitur kategorikal seperti 'FILENAME', 'URL', 'Domain', 'TLD', dan 'Title' diubah menjadi format numerik menggunakan Label Encoding. Label encoding mengonversi setiap kategori menjadi angka unik yang mewakili kategori tersebut. Misalnya, pada kolom 'URL' atau 'Domain', setiap nilai yang ada akan diberikan label numerik berdasarkan urutan kemunculannya dalam data. Jika ada kategori yang tidak ditemukan pada saat transformasi, kategori tersebut akan diberi label 'unknown' untuk menangani data yang tidak ada dalam dataset pelatihan.

1.7 Handling Imbalanced Dataset

Handling imbalanced dataset digunakan untuk mengatasi masalah ketika jumlah sampel dalam kelas-kelas pada dataset tidak merata. Kelas ImbalanceHandler bertujuan untuk menangani dataset yang tidak seimbang dengan menggunakan metode SMOTE (Synthetic Minority Over-sampling

Technique). SMOTE bekerja dengan menghasilkan sampel untuk kelas minoritas berdasarkan tetangga terdekatnya, bukan hanya menduplikasi sampel yang ada.

Kelas ImbalanceHandler mewarisi BaseEstimator dan TransformerMixin dari scikit-learn, memungkinkan penggunaannya dalam pipeline. Konstruktor __init__ menginisialisasi objek SMOTE dan menyimpan status apakah transformasi sudah diterapkan (_is_fitted). Metode fit tidak melakukan pelatihan, tetapi memastikan bahwa objek siap digunakan. Metode transform memeriksa apakah data sudah difit, kemudian menerapkan SMOTE untuk menghasilkan data sintetik untuk kelas minoritas jika diperlukan, dan mengembalikan data yang sudah di-resample. Jika label tidak tersedia, hanya data fitur yang dikembalikan.

2. Penjelasan Implementasi KNN

Algoritma K-Nearest Neighbors (KNN) merupakan metode *supervised learning* yang sederhana namun efektif, sering digunakan untuk tugas klasifikasi dan regresi dalam machine learning. Prinsip dasar dari algoritma ini adalah kedekatan. Data dengan karakteristik yang serupa akan berada dekat satu sama lain. Dalam konteks klasifikasi, KNN memberikan label kelas pada titik data dengan mengidentifikasi k tetangga terdekat dan menerapkan mekanisme voting mayoritas. Dengan kata lain, kelas yang paling sering muncul di antara tetangga tersebut akan menentukan klasifikasi akhir.

2.1 Keunggulan dan Karakteristik KNN

Untuk menentukan kedekatan antar data, KNN menggunakan metrik jarak seperti Euclidean, Manhattan, atau Minkowski, yang dipilih berdasarkan jenis data yang digunakan. Metrik ini berperan penting dalam membentuk keputusan yang memisahkan data ke dalam wilayah kelasnya.

2.2 Pemilihan Parameter K

Parameter k sangat berpengaruh besar dalam algoritma ini karena menentukan jumlah tetangga yang akan mempengaruhi keputusan klasifikasi. Nilai k yang kecil dapat membuat algoritma lebih sensitif terhadap noise, sedangkan nilai k yang besar bisa menghasilkan keputusan yang halus tetapi dapat mengabaikan perbedaan antar kelas. Oleh karena itu, pemilihan nilai k dan metrik jarak yang tepat sangat penting untuk mencapai hasil optimal dari KNN.

2.3 Proses Kerja KNN

Berikut adalah langkah-langkah utama dalam penerapan algoritma KNN:

1. Menentukan nilai k: Memilih jumlah tetangga terdekat yang akan dipertimbangkan dalam klasifikasi.
2. Menghitung jarak: Mengukur jarak antara titik data baru dan semua titik data dalam dataset training menggunakan metrik jarak yang sesuai.
3. Identifikasi tetangga terdekat: Menemukan k tetangga terdekat berdasarkan hasil perhitungan jarak.

Dengan penerapan ini, KNN dapat digunakan secara efektif dalam berbagai konteks meskipun penggunaan pada dataset besar dan kompleks dapat menjadi kurang efisien.

2.4 Implementasi Kode KNN

Berikut adalah implementasi yang kami terapkan untuk KNN from scratch.

```python
from scipy.spatial.distance import cdist
from collections import Counter
import numpy as np
import pickle
from concurrent.futures import ThreadPoolExecutor

class KNNClassifier:
    def __init__(self, k=3, metric="euclidean", batch_size=1000,
n_threads=4):k = k
        self.metric = metric
        self.batch_size = batch_size
        self.n_threads = n_threads
        self.train_data = None
        self.train_labels = None

    def fit(self, train_data, train_labels):
        self.train_data = np.asarray(train_data, dtype=np.float32)
        self.train_labels = np.asarray(train_labels)
        return self

    def _compute_batch_distances(self, test_batch):
        if self.metric == 'euclidean':
            distances = cdist(test_batch, self.train_data,
metric='euclidean')elif self.metric == 'manhattan':
            distances = cdist(test_batch, self.train_data,
metric='cityblock')elif self.metric == 'minkowski':
            distances = cdist(test_batch, self.train_data,
metric='minkowski')
            raise ValueError(f"Unsupported metric: {self.metric}")
        return distances

    def _predict_batch(self, test_batch):
        distances = self._compute_batch_distances(test_batch)
        batch_predictions = []

        for point_distances in distances:
            k_indices = np.argpartition(point_distances, self.k)[:self.k]
            k_labels = self.train_labels[k_indices]

            prediction = Counter(k_labels).most_common(1)[0][0]
            batch_predictions.append(prediction)

        return batch_predictions
```

```python
    def predict(self, test_points):
        if self.train_data is None:
            raise ValueError("Model has not been trained. Call 'fit'
first.")
        test_points = np.asarray(test_points, dtype=np.float32)
        predictions = []

        batches = [
            test_points[i:i + self.batch_size]
            for i in range(0, len(test_points), self.batch_size)
        ]

        with ThreadPoolExecutor(max_workers=self.n_threads) as executor:
            results = executor.map(self._predict_batch, batches)

        for result in results:
            predictions.extend(result)

        return np.array(predictions)

    def get_params(self, deep=True):
        return {
            'k': self.k,
            'metric': self.metric,
            'batch_size': self.batch_size,
            'n_threads': self.n_threads
        }

    def set_params(self, **params):
        if 'k' in params:
            self.k = params['k']
        if 'metric' in params:
            self.metric = params['metric']
        if 'batch_size' in params:
            self.batch_size = params['batch_size']
        if 'n_threads' in params:
            self.n_threads = params['n_threads']
        return self

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

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

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

Gambar 2.1 Implementasi Kode KKN from Scratch

Class KNNClassifier adalah implementasi algoritma k-Nearest Neighbors (KNN) yang dirancang untuk klasifikasi dengan fleksibilitas dan efisiensi. Algoritma ini bekerja dengan cara menghitung jarak antara titik data uji dan titik data pelatihan, kemudian menentukan label kelas berdasarkan mayoritas dari k tetangga terdekat. Parameter utama yang dapat disesuaikan termasuk jumlah tetangga (k), metrik jarak yang digunakan (seperti Euclidean, Manhattan, atau Minkowski), ukuran batch untuk memproses data, serta jumlah thread untuk memanfaatkan pemrosesan paralel.

Pada metode fit, model menyimpan data pelatihan dan labelnya, dan data tersebut diubah menjadi array numpy untuk memastikan komputasi yang efisien. Setelah model dilatih, prediksi dapat dilakukan melalui metode predict, yang membagi data uji ke dalam batch-batch dan menggunakan ThreadPoolExecutor untuk memproses prediksi secara paralel. Jarak antara titik data uji dan data pelatihan dihitung menggunakan metrik yang dipilih, dan tetangga terdekat ditentukan dengan menggunakan np.argpartition. Prediksi label dilakukan dengan mengambil mayoritas dari label tetangga terdekat.

Class ini juga menyediakan metode get_params dan set_params, yang memungkinkan pengguna untuk mengakses dan mengubah parameter model, mendukung integrasi dengan pipeline dan pencarian grid. Fungsi score digunakan untuk menghitung akurasi model, sementara metode save_model dan load_model memungkinkan penyimpanan dan pemuatan model dalam format pickle. Dengan dukungan pemrosesan batch dan paralel, class KNNClassifier menawarkan solusi yang efisien untuk menangani dataset besar dan meningkatkan fleksibilitas dalam pengaturan dan evaluasi model.

Berikut adalah implementasi yang kami terapkan untuk KNN dengan Scikit-learn.

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, roc_auc_score, recall_score, precision_score,
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from imblearn.over_sampling import SMOTE
import pandas as pd

# Preprocess data
X_train, X_test, y_train, y_test, _ = preprocess_data(train_df)

# Train the KNN model using sklearn
knn = KNeighborsClassifier(n_neighbors=3, metric='euclidean')
knn.fit(X_train, y_train)

# Generate predictions
predictions = knn.predict(X_test)

# Check if ROC AUC score is valid (requires probabilities for each class)
if len(knn.classes_) > 2:  # Multiclass case
    probabilities = knn.predict_proba(X_test)
    roc_auc = roc_auc_score(y_test, probabilities, multi_class='ovr', average='macro')
else:  # Binary case
    probabilities = knn.predict_proba(X_test)[:, 1]  # Select positive class probabilities
    roc_auc = roc_auc_score(y_test, probabilities)

# Calculate other evaluation metrics
accuracy = accuracy_score(y_test, predictions)
recall = recall_score(y_test, predictions, average='macro')
precision = precision_score(y_test, predictions, average='macro')
f1 = f1_score(y_test, predictions, average='macro')

# Print metrics
print(f"Accuracy: {accuracy}")
print(f"ROC AUC Score: {roc_auc}")
print(f"Recall: {recall}")
print(f"Precision: {precision}")
print(f"F1 Score: {f1}")
```

Gambar 2.2 Implementasi Kode KKN from Library

3. Penjelasan Implementasi Naive-Bayes

Naive Bayes adalah algoritma klasifikasi berbasis probabilitas yang digunakan dalam *supervised learning*. Algoritma ini memiliki prinsip dasar menghitung probabilitas kelas berdasarkan fitur yang ada pada data dan memilih kelas dengan probabilitas tertinggi.

3.1 Keunggulan dan Karakteristik Naive Bayes

Naive Bayes merupakan algoritma yang cepat dan efisien, terutama pada dataset besar. Algoritma ini menghitung probabilitas untuk setiap kelas berdasarkan fitur yang ada, dengan penyesuaian smoothing untuk menangani probabilitas nol. Naive Bayes cocok untuk digunakan dengan banyak fitur dan dapat menangani ketidakseimbangan kelas dengan efektif.

3.2 Proses Kerja Naive Bayes

Berikut adalah langkah-langkah utama dalam penerapan algoritma Naive-Bayes:

1. Inisialisasi Kelas NaiveBayes

   Kelas NaiveBayes diinisialisasi dengan beberapa parameter seperti *smoothing* untuk menangani probabilitas nol dan prior_adjustment untuk menyesuaikan probabilitas kelas jika diperlukan. Kelas NaiveBayes menyimpan berbagai informasi tentang probabilitas kelas dan fitur untuk digunakan dalam prediksi.
2. Fitting Model

   Fungsi fit bertugas untuk melatih model dengan menghitung probabilitas prior untuk setiap kelas dan probabilitas untuk setiap fitur dalam kelas.
3. Prediksi Probabilitas

   Fungsi predict_proba digunakan untuk menghitung probabilitas kelas yang dihitung setelah mempertimbangkan data yang ada.
4. Prediksi Kelas

   Fungsi predict mengubah probabilitas menjadi prediksi kelas dengan *threshold* tertentu, pada kode *threshold* yang digunakan adalah 0.005. Jika probabilitas untuk kelas tertentu lebih besar dari *threshold* ini, maka kelas tersebut akan dipilih sebagai hasil prediksi.
5. Evaluasi dan Penyimpanan Model

Evaluasi dilakukan dengan menggunakan *cross-validation* untuk memastikan model tidak terlalu *overfit* dan memiliki performa yang stabil di berbagai subset data.

3.3 Implementasi Kode Naive Bayes

Berikut adalah implementasi yang kami terapkan untuk NB from scratch.

```python
import numpy as np
import pandas as pd
import os
import pickle
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, LabelEncoder
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split, KFold
from sklearn.metrics import accuracy_score, classification_report

class NaiveBayes:
    # NaiveBayes class inisialization
    def __init__(self, smoothing=1e-3, prior_adjustment=None):
        self.smoothing = smoothing
        self.classes_ = None
        self.class_probabilities = {}
        self.feature_probabilities = {}
        self.class_counts = {}
        self.prior_adjustment = prior_adjustment

    # Trains the Naive Bayes model by calculating prior probabilities for each class
    def fit(self, X, y):
        X = np.array(X)
        y = np.array(y)

        self.classes_ = np.unique(y)
        n_samples, n_features = X.shape

        for cls in self.classes_:
            self.class_probabilities[cls] = np.sum(y == cls) / n_samples
            if self.prior_adjustment and cls in self.prior_adjustment:
                self.class_probabilities[cls] *= self.prior_adjustment[cls]

            self.class_counts[cls] = np.sum(y == cls)

        # Calculate the prior probability for each class
        self.feature_probabilities = {cls: [] for cls in self.classes_}
        for cls in self.classes_:
            X_cls = X[y == cls]
            for feature_idx in range(n_features):
                feature_vals = X_cls[:, feature_idx]
                unique_vals, counts = np.unique(feature_vals, return_counts=True)
                feature_prob = {
                    val: (count + self.smoothing) / (self.class_counts[cls] + self.smoothing *
len(unique_vals))
                    for val, count in zip(unique_vals, counts)
                }
                self.feature_probabilities[cls].append(feature_prob)

        return self
```

```python
# This method calculates the probability of each class for each sample in X
    def predict_proba(self, X):
        X = np.array(X)
        probabilities = []
        for sample in X:
            posteriors = []
            for cls in self.classes_:
                score = np.log(self.class_probabilities[cls] + self.smoothing)
                for feature_idx, feature_val in enumerate(sample):
                    feature_prob = self.feature_probabilities[cls][feature_idx].get(feature_val,
self.smoothing)
                    score += np.log(feature_prob + self.smoothing)
                posteriors.append(np.exp(score))
            probabilities.append(posteriors / np.sum(posteriors))
        return np.array(probabilities)

    # This method determines the predicted class for each sample in the input data
    def predict(self, X, threshold=0.005):
        probabilities = self.predict_proba(X)
        predictions = (probabilities[:, 1] >= threshold).astype(int)

        for i, prob in enumerate(probabilities):
            if prob[0] > prob[1] * 0.8:
                predictions[i] = 0

        return predictions

    # Saves a trained model to a file.
    def save_model(self, filename):
        with open(filename, 'wb') as file:
            pickle.dump(self, file)
        print(f"Model saved in {filename}.")

    # Loads a previously saved model from a file for reuse.
    @staticmethod
    def load_model(filename):
        with open(filename, 'rb') as file:
            model = pickle.load(file)
        print(f"Model loaded from {filename}.")
        return model
```

```python
X_train, X_test, y_train, y_test, X_test_final = preprocess_data(train_df, test_df)

# Train and evaluate model
nb = NaiveBayes(prior_adjustment={0: 50.0, 1: 1.0})
nb.fit(X_train, y_train)
nb.save_model('naive_bayes_model.pkl')

# Perform cross validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
cross_val_scores = []

X_train_np = np.array(X_train)
y_train_np = np.array(y_train)
```

```python
for train_idx, val_idx in kf.split(X_train_np):
    X_cv_train, X_cv_val = X_train_np[train_idx], X_train_np[val_idx]
    y_cv_train, y_cv_val = y_train_np[train_idx], y_train_np[val_idx]

    nb_cv = NaiveBayes(prior_adjustment={0: 50.0, 1: 1.0})
    nb_cv.fit(X_cv_train, y_cv_train)
    y_cv_pred = nb_cv.predict(X_cv_val)
    cross_val_scores.append(accuracy_score(y_cv_val, y_cv_pred))

print(f"IMPLEMENTATION FROM SCRATCH")
print(f"Cross-Validation Accuracy (Mean): {np.mean(cross_val_scores) * 100:.2f}%")
print(f"Cross-Validation Accuracy (Standard Deviation): {np.std(cross_val_scores) * 100:.2f}%")

# Evaluate the final model on the test set
y_pred = nb.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print(f"\nNaive Bayes kustom classification accuracy: {accuracy * 100:.2f}%\n")
print("Detailed Classification Report (Custom Naive Bayes):")
print(classification_report(y_test, y_pred))

# Save predictions to a CSV file
predictions = nb.predict(X_test_final)
submission_df = pd.DataFrame({
    "id": test_df["id"],
    "label": predictions
})
submission_file_path = 'submission-nb-scratch.csv'
submission_df.to_csv(submission_file_path, index=False)

print(f"Predictions saved to '{submission_file_path}'.")
```

Gambar 3.1 Implementasi NB from Scratch

Berikut adalah implementasi yang kami terapkan untuk NB dengan Scikit-learn.

```python
import numpy as np
import pandas as pd
import os
import pickle
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, LabelEncoder
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split, KFold
from sklearn.metrics import accuracy_score, classification_report
from sklearn.naive_bayes import GaussianNB
```

```python
X_train, X_test, y_train, y_test, X_test_final = preprocess_data(train_df, test_df)

# Train and evaluate model
model = GaussianNB()
model.fit(X_train.values, y_train.values)

# Perform cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
cross_val_scores = []

X_train_np = X_train.values
y_train_np = y_train.values

for train_idx, val_idx in kf.split(X_train_np):
    X_cv_train, X_cv_val = X_train_np[train_idx], X_train_np[val_idx]
    y_cv_train, y_cv_val = y_train_np[train_idx], y_train_np[val_idx]

    model.fit(X_cv_train, y_cv_train)
    y_cv_pred = model.predict(X_cv_val)
    cross_val_scores.append(accuracy_score(y_cv_val, y_cv_pred))

print(f"IMPLEMENTATION WITH SCIKIT-LEARN")
print(f"Cross-Validation Accuracy (Mean): {np.mean(cross_val_scores) * 100:.2f}%")
print(f"Cross-Validation Accuracy (Standard Deviation): {np.std(cross_val_scores) * 100:.2f}%")

# Evaluate the final model on the test set
y_pred = model.predict(X_test.values)
accuracy = accuracy_score(y_test, y_pred)

print(f"\nNaive Bayes classification accuracy: {accuracy * 100:.2f}%\n")
print("Detailed Classification Report:")
print(classification_report(y_test, y_pred))

# Save predictions to a CSV file
predictions = model.predict(X_test_final.values)
submission_df = pd.DataFrame({
    "id": test_df["id"],
    "label": predictions
})
submission_file_path = 'submission-nb-scikit-learn.csv'
submission_df.to_csv(submission_file_path, index=False)

print(f"Predictions saved to '{submission_file_path}'.")
```

Gambar 3.2 Implementasi NB menggunakan Scikit Learn

Kelas NaiveBayes mengimplementasikan algoritma klasifikasi Naive Bayes dengan pendekatan berbasis probabilitas. Algoritma ini bekerja dengan menghitung probabilitas untuk setiap kelas berdasarkan data yang ada, lalu memilih kelas dengan probabilitas tertinggi. Kelas NaiveBayes memiliki beberapa metode utama untuk menjalankan proses. Pada metode __init__, beberapa parameter diinisialisasi, seperti smoothing yang digunakan untuk menghindari probabilitas nol, serta class_probabilities, feature_probabilities, dan prior_adjustment untuk mengelola probabilitas kelas dan fitur.

Metode fit digunakan untuk melatih model dengan menghitung probabilitas prior untuk setiap kelas berdasarkan distribusi kelas dalam data, serta menghitung probabilitas fitur dalam setiap kelas. Untuk mengatasi masalah data yang hilang, metode ini menggunakan smoothing. Metode predict_proba menghitung probabilitas posterior setiap kelas untuk sampel yang diberikan, dengan cara menjumlahkan log dari probabilitas prior dan probabilitas fitur yang relevan, kemudian dinormalisasi.

Sementara itu, metode predict menentukan kelas mana yang akan diprediksi berdasarkan probabilitas posterior yang dihitung, dengan menggunakan threshold sebesar 0.005 untuk klasifikasi. Selain itu, terdapat metode save_model dan load_model yang digunakan untuk menyimpan model yang sudah dilatih ke file dan me-*load*nya kembali.

4. Perbandingan Hasil Prediksi

4.1 Analisis KNN

Perbandingan antara implementasi KNN dari scratch dan yang menggunakan library menunjukkan bahwa keduanya menghasilkan akurasi yang sangat baik, meskipun ada sedikit perbedaan. Implementasi dari scratch mencapai akurasi 0.9731, sementara implementasi menggunakan scikit-learn memperoleh akurasi 0.9705. Meskipun terdapat perbedaan kecil dalam akurasi, keduanya masih menunjukkan kemampuan yang sangat baik dalam mengenali pola dalam data. Hasil ini mengindikasikan bahwa fitur dasar algoritma KNN berfungsi dengan optimal, baik dengan implementasi manual dari scratch maupun menggunakan library seperti scikit-learn.

Namun, perbedaan yang signifikan muncul dalam hal waktu eksekusi. Implementasi dari scratch memerlukan waktu yang jauh lebih lama, yakni 8 menit, sementara implementasi dengan menggunakan scikit-learn hanya membutuhkan waktu 2 menit. Perbedaan waktu ini terjadi karena library seperti scikit-learn telah mengoptimalkan algoritma KNN dengan menggunakan implementasi berbasis C yang lebih efisien dalam hal komputasi. Sementara itu, implementasi dari scratch memerlukan proses yang lebih lama karena tidak memanfaatkan optimasi dan harus melalui lebih banyak langkah manual, termasuk dalam pengelolaan memori dan perhitungan jarak. Meskipun implementasi dari scratch memberikan fleksibilitas lebih dalam penyesuaian algoritma, library lebih unggul dalam hal kecepatan dan efisiensi.

```
Accuracy: 0.9731019419780638
ROC AUC Score: 0.86067797986059
Recall: 0.8606779798605901
Precision: 0.9361982705532068
F1 Score: 0.8941758627892029
Confusion Matrix:
[[ 2306   860]
 [  273 38683]]
Classification Report:
              precision    recall  f1-score   support

           0       0.89      0.73      0.80      3166
           1       0.98      0.99      0.99     38956

    accuracy                           0.97     42122
   macro avg       0.94      0.86      0.89     42122
weighted avg       0.97      0.97      0.97     42122
```

Gambar 4.1 Hasil Akurasi KNN from Scratch

Berdasarkan hasil evaluasi implementasi KNN dari scratch di atas, model menunjukkan Accuracy sebesar 97.31%, yang menunjukkan performa yang sangat baik dalam klasifikasi keseluruhan. Namun, analisis lebih rinci pada Confusion Matrix mengungkap bahwa kelas mayoritas (kelas 1) memiliki performa yang jauh lebih baik dibandingkan kelas minoritas (kelas 0).

Khusus untuk kelas 0, model memiliki Precision sebesar 0.89 dan Recall sebesar 0.73, mengindikasikan bahwa model cukup sering salah dalam menangkap semua sampel kelas 0, kemungkinan besar disebabkan oleh ketidakseimbangan jumlah data antar kelas. Sebaliknya, untuk kelas 1, Precision dan Recall masing-masing mencapai 0.98 dan 0.99, menunjukkan bahwa model hampir sempurna dalam mengidentifikasi sampel kelas 1.

ROC AUC Score sebesar 0.86 menunjukkan bahwa model cukup baik dalam membedakan antara kedua kelas secara keseluruhan, meskipun masih ada cara untuk meningkatkan kemampuan menerima kelas minoritas

(kelas 0). F1 Score makro sebesar 0.89 juga mencerminkan ketidakseimbangan performa antara kedua kelas.

Untuk meningkatkan performa, terutama pada kelas minoritas, langkah-langkah seperti oversampling menggunakan SMOTE, pengenalan weighted voting pada KNN, atau tuning lebih lanjut pada parameter k dapat diterapkan.

```
Accuracy: 0.9705854422866911
ROC AUC Score: 0.8888659684214084
Recall: 0.8728111187787742
Precision: 0.9080282247111293
F1 Score: 0.8894615202110798
Confusion Matrix:
[[ 2399   767]
 [  472 38484]]
Classification Report:
              precision    recall  f1-score   support

           0       0.84      0.76      0.79      3166
           1       0.98      0.99      0.98     38956

    accuracy                           0.97     42122
   macro avg       0.91      0.87      0.89     42122
weighted avg       0.97      0.97      0.97     42122
```

Gambar 4.2 Hasil Akurasi KNN from Library

Berdasarkan hasil evaluasi implementasi KNN menggunakan library di atas, model menunjukkan Accuracy sebesar 97.06% yang merupakan performa yang sangat baik dalam klasifikasi keseluruhan. Namun, analisis lebih rinci pada Confusion Matrix dan Classification Report mengungkap adanya perbedaan performa yang cukup signifikan antara kelas mayoritas (kelas 1) dan kelas minoritas (kelas 0).

Untuk kelas 0, model memiliki Precision sebesar 0.84 dan Recall sebesar 0.76, yang mengindikasikan bahwa meskipun prediksi untuk kelas 0 cukup

akurat, model kesulitan menerima semua sampel kelas 0, dengan 24% dari total sampel kelas 0 gagal dikenali. Sebaliknya, untuk kelas 1, Precision mencapai 0.98 dan Recall sebesar 0.99, menunjukkan bahwa model hampir sempurna dalam mengenali kelas mayoritas, dengan hanya sedikit kesalahan.

ROC AUC Score sebesar 0.89 menunjukkan bahwa model cukup baik dalam membedakan antara kedua kelas, meskipun masih ada cara untuk meningkatkan kemampuan model menangkap kelas minoritas. F1 Score Makro sebesar 0.89 juga mencerminkan adanya ketidakseimbangan performa antara kedua kelas, meskipun Weighted Average F1 Score sebesar 0.97 menunjukkan bahwa performa secara keseluruhan sangat baik, terutama karena kelas mayoritas mendominasi dataset.

Untuk meningkatkan performa pada kelas minoritas (kelas 0), seperti pada KNN scratch, langkah-langkah seperti oversampling menggunakan SMOTE, pengenalan weighted voting pada KNN, atau tuning lebih lanjut pada parameter k dapat dilakukan.

4.2 Analisis Naive Bayes

Implementasi *from scratch* menggunakan *class* bernama Naive Bayes di mana perhitungan dan penyimpanan model dilakukan secara manual. Implementasi dengan scikit-learn menggunakan *class* bawaan GaussianNB yang sudah mengimplementasikan semua fungsi Naive Bayes di dalamnya.

Waktu eksekusi implementasi dengan scikit-learn adalah 1.56 menit, sementara implementasi *from scratch* membutuhkan 2.26 menit. Hal ini terjadi karena implementasi *from scratch* melakukan perhitungan secara manual pada perhitungan probabilitas sehingga prosesnya lebih lambat dibandingkan implementasi dengan scikit-learn.

Dari segi akurasi, implementasi *from scratch* menunjukkan *Cross-Validation Accuracy (Mean)* sebesar 94.18% dan *Test Accuracy*

sebesar 98.31%. Sementara implementasi dengan scikit-learn menghasilkan *Cross-Validation Accuracy (Mean)* sebesar 92.62% dan *Test Accuracy* sebesar 98.87%. Meskipun implementasi *from scratch* memiliki *cross-validation* lebih tinggi, implementasi dengan scikit-learn menunjukkan hasil yang lebih baik pada data *test* karena GaussianNB lebih efisien dan optimal dalam menangani dataset.

Implementasi dengan scikit-learn mencatat *Recall* pada label 0 (*phishing*) sebesar 87%, lebih tinggi dibandingkan implementasi *from scratch* yang hanya mencapai 78%, hal ini terjadi karena implementasi menggunakan scikit learn memiliki keunggulan dalam mendeteksi data minoritas. Selain itu, implementasi dengan scikit-learn menunjukkan *Macro Average Call* sebesar 94% dan *Macro Average F1-Score* sebesar 95%, dibandingkan implementasi *from scratch* yang hanya mencatat 89% dan 93%.

```
Model saved in naive_bayes_model.pkl.
IMPLEMENTATION FROM SCRATCH
Cross-Validation Accuracy (Mean): 94.18%
Cross-Validation Accuracy (Standard Deviation): 0.08%

Naive Bayes custom classification accuracy: 98.31%

Detailed Classification Report (Custom Naive Bayes):
              precision    recall  f1-score   support

           0       1.00      0.78      0.87      3166
           1       0.98      1.00      0.99     38956

    accuracy                           0.98     42122
   macro avg       0.99      0.89      0.93     42122
weighted avg       0.98      0.98      0.98     42122

Predictions saved to 'submission-nb-scratch.csv'.
```

Gambar 4.3 Hasil Akurasi Naive Bayes from Scratch

```
IMPLEMENTATION WITH SCIKIT-LEARN
Cross-Validation Accuracy (Mean): 92.62%
Cross-Validation Accuracy (Standard Deviation): 0.11%

Naive Bayes classification accuracy: 98.77%

Detailed Classification Report:
              precision    recall  f1-score   support

           0       0.96      0.87      0.91      3166
           1       0.99      1.00      0.99     38956

    accuracy                           0.99     42122
   macro avg       0.97      0.94      0.95     42122
weighted avg       0.99      0.99      0.99     42122

Predictions saved to 'submission-nb-scikit-learn.csv'.
```

Gambar 4.4 Hasil Akurasi Naive Bayes from Library

5. Pembagian Tugas Tiap Anggota Kelompok

Berikut adalah tabel rincian mengenai pendistribusian kerja oleh anggota kelompok:

No	NIM Anggota	Nama Anggota	Deskripsi
1	18222106	Audra Zelvania Putri Harjanto	Data Cleaning and Processing, Naive Bayes Algorithm
2	18222118	Rizqi Andhika Pratama	Data Cleaning and Processing, KNN Algorithm
3	18222125	Sekar Anindita Nurjadini	Data Cleaning and Processing, Naive Bayes Algorithm
4	18222138	Khayla Belva Annandira	Data Cleaning and Processing, KNN Algorithm

6. Lampiran

Repository tempat penyimpanan program dapat diakses melalui **link** berikut.

Referensi

- https://archive.ics.uci.edu/dataset/967/phiusiil+phishing+url+dataset
- https://www.sciencedirect.com/science/article/abs/pii/S0167404823004558?via%3Dihub
- https://scikit-learn.org/1.5/modules/neighbors.html
- https://scikit-learn.org/1.5/modules/naive_bayes.html