TUGAS BESAR 2

IF3070 Dasar Inteligensi Artifisial

Implementasi K-*Nearest Neighbor* dan Gaussian *Naive-Bayes* dalam Model Klasifikasi *Supervised Learning*

Disusun Oleh Kelompok Seblak Dondon:

18222008	Abel Apriliani
18222036	Olivia Christy Lismanto
18222044	Khansa Adilla Reva
18221062	Nafisha Virgin

Dosen Pengampu :

Dr. Nur Ulfa Maulidevi, S.T., M.Sc.

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2024

DAFTAR ISI

BAB I
IMPLEMENTASI MODEL

1.1. K-Nearest Neighbor (KNN)

1.1.1. Penjelasan KNN

Algoritma K-Nearest Neighbor (KNN) adalah algoritma *machine learning* yang bersifat non-*parametric* dan *lazy learning*. Metode yang bersifat non-*parametric* memiliki makna bahwa metode tersebut tidak membuat asumsi apa pun tentang distribusi data yang mendasarinya. Dengan kata lain, tidak ada jumlah parameter atau estimasi parameter yang tetap dalam model, terlepas data tersebut berukuran kecil ataupun besar.

KNN bekerja berdasarkan prinsip bahwa setiap titik data yang berdekatan satu sama lain akan berada di kelas yang sama. Dengan kata lain, KNN mengklasifikasikan titik data baru berdasarkan kemiripan.

Adapun alur cara kerja algoritma KNN adalah sebagai berikut:

- Langkah-1: Pilih nilai banyaknya tetangga K.
- Langkah-2: Hitung jarak dari jumlah tetangga K.
- Langkah-3: Ambil tetangga terdekat K sesuai jarak yang dihitung.
- Langkah-4: Di antara tetangga K ini, hitung jumlah titik data di setiap kategori.
- Langkah-5: Tetapkan titik data baru ke kategori yang jumlah tetangganya paling banyak.
- Langkah-6: Model sudah siap.

1.1.2. Inisialisasi

```
def __init__(self, k):
        self.k = k
```

Algoritma KNN diinisialisasi dengan nilai k yang merepresentasikan jumlah tetangga terdekat. Semakin kecil nilai k, prediksi akan lebih sensitif terhadap tetangga terdekat. Sebaliknya, nilai k yang besar cenderung memberikan hasil yang lebih stabil, tetapi berisiko mengurangi sensitivitas terhadap pola lokal. Inisialisasi k dalam kelas ini bertujuan untuk menyimpan nilai tersebut agar digunakan saat proses klasifikasi atau regresi.

1.1.3. Fungsi Fit

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

Fungsi fit berguna untuk menyimpan data pelatihan. X merupakan data berisi fitur dan y data berisi label. Data disimpan dalam *array* NumPy untuk memudahkan proses pengolahan data.

1.1.4. Fungsi Menghitung Jarak

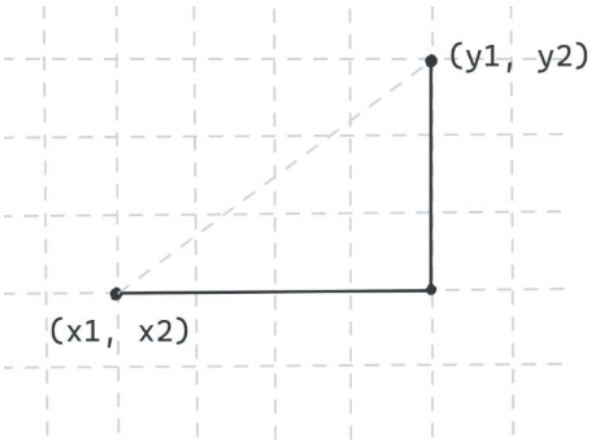
```
def euclidean_distance(self, x1, x2):
     return np.sqrt(np.sum((x1 - x2) ** 2))

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

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

Terdapat 3 fungsi menghitung jarak yang tersedia, yaitu :

1. Jarak Euclidean adalah jarak terpendek antara dua titik di ruang metrik

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

2. Jarak Manhattan



Manhattan Distance in 2D Plane

d(x,y) = |x1 - y1| + |x2 - y2|

3. Jarak Minkowski

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

1.1.5. Fungsi Predict

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

Fungsi predict merupakan fungsi untuk memprediksi seluruh label dari data yang ada berdasarkan model KNN yang sudah dilatih. Parameter X merupakan data yang akan diprediksi labelnya dan parameter dist_metric merupakan matriks perhitungan jarak dari tetangga. Prediksi dicari melalui fungsi _predict. Lalu, hasil prediksi disimpan di dalam array y_pred.

1.1.6. Fungsi y_predict

```
def _predict(self, x, dist_metric):
        x = np.array(x)
        if dist_metric == "Euclidean":
            distances = [self.euclidean_distance(x, X_train) for
X_train in self.X_train]
        elif dist_metric == "Manhattan":
            distances = [self.manhattan_distance(x, X_train) for
X_train in self.X_train]
        elif dist_metric == "Minkowski":
            distances = [self.minkowski_distance(x, X_train) for
X_train in self.X_train]

        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        most_common = np.bincount(k_nearest_labels).argmax()
        return most_common
```

Fungsi y_predict merupakan fungsi untuk memprediksi label berdasarkan model KNN yang sudah dilatih. Parameter x merupakan *instance* dari data yang labelnya akan diprediksi dan dist_metric merupakan matriks perhitungan jarak yang dipilih. Lalu, *instance* tersebut disimpan dalam bentuk *array* NumPy untuk mempermudah pengolahan data. Lalu, dilakukan perhitungan jarak berdasarkan matriks perhitungan yang dipilih. Hasil perhitungan jarak akan disimpan di dalam

variabel distances. Kemudian, distances akan disort dengan menggunakan fungsi np.argsort(). Fungsi ini mengurutkan indeks berdasarkan nilai distances dari terkecil ke terbesar. Hasil dari fungsi tersebut disimpan dalam variabel k_indices. Lalu, label dari k tetangga terdekat didapatkan dengan mengecek data y_train sesuai indeks di k_indices. Kemudian, label mayoritas dari k tetangga terdekatlah yang menjadi hasil dari prediksi.

1.2 Gaussian Naive-Bayes (GNB)

1.2.1. Penjelasan Gaussian Naive-Bayes

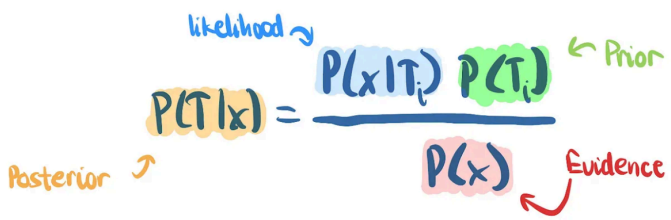
Gaussian Naive Bayes merupakan sebuah teknik klasifikasi yang digunakan dalam machine learning dengan menggunakan metode probability dan Distribusi Gaussian atau Distribusi Normal. Algoritma ini mengasumsikan bahwa setiap fitur secara independen mempengaruhi variabel target. Konsep utama yang digunakan dalam GNB adalah Teorema Bayes, yang digunakan untuk menghitung probabilitas masing-masing kelas berdasarkan fitur yang diberikan. Dengan mengasumsikan bahwa fitur-fitur mengikuti distribusi Gaussian, GNB menyederhanakan perhitungan probabilitas dengan hanya memerlukan dua parameter, yaitu rata-rata dan deviasi standar untuk setiap fitur. Hal ini menjadikannya lebih efisien secara komputasi.

Bagian "*naive*" pada namanya ini berasal dari asumsi bahwa fitur-fitur dalam data saling independen. Penyederhanaan ini memungkinkan GNB untuk menghitung probabilitas suatu kelas berdasarkan sejumlah fitur dengan menggunakan rumus berikut.

$$P(class|features) = \frac{P(class) \times P(features|class)}{P(features)}$$

Keterangan:

- P(class | features) adalah *posterior probability of class* dengan fitur tertentu.
- P(class) adalah *prior probability of class.*
- P(features | class) adalah *likelihood* (kemungkinan fitur-fitur terjadi pada kelas tertentu) yang diasumsikan mengikuti Distribusi Gaussian.
- P(features) adalah *prior probability of features*.



Model Gaussian Naive-Bayes menghitung kemungkinan setiap fitur berdasarkan kelas, lalu mengalikan probabilitas-probabilitas tersebut untuk

menentukan kelas yang paling mungkin untuk sebuah data. Kelas dengan probabilitas tertinggi akan dipilih sebagai hasil prediksi.

Keunggulan utama dari GNB adalah efisiensinya, terutama dalam dataset yang berdimensi tinggi. Algoritma ini mampu menangani data kontinu dengan baik dan sangat berguna ketika distribusi fitur-fitur tersebut mendekati distribusi Gaussian. Akan tetapi, GNB memiliki beberapa keterbatasan, seperti asumsi independensi fitur yang sering kali sebenarnya tidak berlaku dalam data dunia nyata, sensitivitas terhadap outlier, dan kesulitan dalam menangani dataset yang tidak seimbang.

1.2.2. Inisialisasi

```
def __init__(self):
    self.classes = None
    self.mean = None
    self.var = None
    self.priors = None
```

Fungsi ini digunakan untuk menyiapkan atribut dasar yang dibutuhkan untuk model.

Algoritma GNB diinisialisasi dengan empat atribut, yaitu:

- self.classes: menyimpan label dataset
- self.mean: menyimpan rata-rata tiap fitur di setiap kelas
- self.var: menyimpan varians tiap fitur di setiap kelas
- self.priors: menyimpan probabilitas sebelumnya dari setiap kelas, yaitu seberapa sering setiap kelas muncul dalam dataset.

1.2.3. Fungsi Fit

```
def fit(self, X, y):
    n_samples, features = X.shape
    self.classes = np.unique(y)
    n_classes = len(self.classes)
    self.mean = np.zeros((n_classes, features))
    self.var = np.zeros((n_classes, features))
    self.priors = np.zeros(n_classes)


    for idx, cls in enumerate(self.classes):
        X_cls = X[y == cls]
        self.mean[idx, :] = np.mean(X_cls, axis=0)
        self.var[idx, :] = np.var(X_cls, axis=0)
        self.priors[idx] = X_cls.shape[0] / n_samples
```

Fungsi fit digunakan untuk menyimpan data pelatihan. Pada fungsi ini, akan dihitung nilai rata-rata dan varians dari tiap fitur untuk setiap kelas lalu disimpan. Fungsi ini juga menghitung *prior probability* untuk setiap kelas, yang merupakan

proporsi sampel yang termasuk di dalam masing-masing kelas. Hal ini penting untuk mengklasifikasikan *instance* baru.

1.2.4. Fungsi Menghitung *Likelihood*

```
    def _gaussian_likelihood(self, class_idx, x):
        mean = self.mean[class_idx]
        var = self.var[class_idx] + 1e-9
        numerator = np.exp(-(x - mean) ** 2 / (2 * var))
        denominator = np.sqrt(2 * np.pi * var)
        return numerator / denominator
```

Fungsi ini digunakan untuk menghitung kemungkinan (*likelihood*) terkait suatu data x berasal dari kelas tertentu berdasarkan Distribusi Gaussian. Dalam algoritma ini, x merupakan nilai fitur yang ingin dihitung probabilitasnya berdasarkan distribusi Gaussian kelas tersebut. Fungsi ini akan menggunakan rata-rata dan varians yang telah dihitung sebelumnya.

Pada fungsi ini, dihitung probabilitas berdasarkan rumus distribusi normal (Gaussian). Perhitungan dilakukan dengan mengukur seberapa jauh nilai x dari rata-rata kelas tersebut, kemudian dinormalisasi dengan varians. Penambahan angka kecil pada varians dilakukan untuk mencegah pembagian dengan nol.

1.2.5. Fungsi Menghitung Posterior *Probability*

```
    def _posterior(self, x):
        posteriors = []
        for idx, cls in enumerate(self.classes):
            prior = np.log(self.priors[idx])
            likelihood = np.sum(np.log(self._gaussian_likelihood(idx,
x)))
            posterior = prior + likelihood
            posteriors.append(posterior)
        return self.classes[np.argmax(posteriors)]
```

Fungsi ini bertujuan untuk menghitung probabilitas posterior untuk setiap kelas berdasarkan vektor fitur x. *Core* dari klasifikasi *Naive Bayes* adalah *posterior probability* ini yang dihitung menggunakan Teorema Bayes. Untuk setiap kelas, fungsi ini akan menghitung logaritma *prior probability* dan logaritma *likelihood* fitur. Kelas dengan probabilitas posterior tertinggi akan dipilih sebagai kelas yang diprediksi (*predicted class*). Pada dasarnya, fungsi ini akan menentukan kelas yang paling mungkin untuk sebuah data x yang diberikan menggunakan pengetahuan yang diperoleh model selama *training*.

1.2.6. Fungsi Predict

```
    def predict(self, X):
        if not isinstance(X, np.ndarray):
            X = np.array(X)
        X = X.astype(float)
        y_pred = [self._posterior(x) for x in X]
        return np.array(y_pred)
```

Fungsi predict merupakan fungsi untuk memprediksi seluruh label dari data yang ada berdasarkan model GNB yang sudah dilatih. Setelah model dilatih, model dapat digunakan untuk mengklasifikasikan data baru. Fungsi ini akan mengambil data fitur baru dan menerapkan kalkulasi probabilitas posterior yang dipelajari untuk memprediksi kelas untuk setiap *instance* baru.

BAB 2
CLEANING AND PREPROCESSING DATA

2.1. Data Cleaning

Data cleaning adalah proses memastikan keakuratan, konsistensi, dan kegunaan data dalam kumpulan data. Prosesnya yang dilakukan adalah mendeteksi kesalahan data atau data yang *corrupt* dan memperbaiki atau menghapus data sesuai kebutuhan.

2.1.1. Handling Missing Data

Data yang hilang akan menurunkan akurasi dari prediksi model. Bahkan, akan mengakibatkan *error* pada model KNN. Oleh karena itu, data yang hilang harus diatasi.

```
      def handle_missing_data(df, non_cat_features,
cat_features) -> pd.DataFrame:

          missing_before = df.isnull().sum()

          num_imputer = SimpleImputer(strategy='mean')
          df[non_cat_features] =
num_imputer.fit_transform(df[non_cat_features])

          cat_imputer =
SimpleImputer(strategy='most_frequent')
          df[cat_features] =
cat_imputer.fit_transform(df[cat_features])
```

Data yang hilang ditangani dengan menggunakan metode data imputation. Imputation pada data numerik adalah dengan menggunakan *mean* dan pada data kategorik menggunakan modus karena efektif dan sederhana. Metode ini dianggap dapat mempertahankan distribusi dataset tanpa menambah kerumitan. Pendekatan ini menghindari asumsi sembarangan (seperti pada imputasi dengan nilai tetap) dan mencegah hilangnya data (seperti pada metode penghapusan).

2.1.2 *Dealing with Outliers*

Mengatasi *outliers* diperlukan karena *outlier* yang banyak bisa mempengaruhi distribusi data dan menurunkan akurasi model, terutama model yang berbasis statistik seperti *Naive Bayes*.

```
capping_features_train = []
capping_features_val = []
mean_imputation_features_train = []
mean_imputation_features_val = []
isolation_forest_features_train = []
isolation_forest_features_val = []
```

```
for stats in outlier_stats:
    feature = stats['feature']
    train_percentage = stats['train_percentage']
    val_percentage = stats['val_percentage']

    if feature in train_set.columns:
        if train_percentage > 5:
            capping_features_train.append(feature)
        elif 1 <= train_percentage <= 5:
            mean_imputation_features_train.append(feature)
        elif train_percentage < 1:
            isolation_forest_features_train.append(feature)

    if feature in val_set.columns:
        if val_percentage > 5:
            capping_features_val.append(feature)
        elif 1 <= val_percentage <= 5:
            mean_imputation_features_val.append(feature)
        elif val_percentage < 1:
            isolation_forest_features_val.append(feature)



def handle_outliers(train_set, val_set):
    """
    Handle outliers using pre-identified lists for capping,
mean imputation, and Isolation Forest.
    """

    for feature in capping_features_train:
        print(f"Clipping outliers for feature (Train Set):
{feature}")
        _, _, _, lower_limit_train, upper_limit_train =
iqr(train_set, feature)
        train_set[feature] = np.clip(train_set[feature],
lower_limit_train, upper_limit_train)

    for feature in capping_features_val:
        print(f"Clipping outliers for feature (Validation
Set): {feature}")
        _, _, _, lower_limit_val, upper_limit_val =
iqr(val_set, feature)
        val_set[feature] = np.clip(val_set[feature],
lower_limit_val, upper_limit_val)

    for feature in mean_imputation_features_train:
        print(f"Imputing outliers with mean for feature
(Train Set): {feature}")
        _, _, _, lower_limit_train, upper_limit_train =
iqr(train_set, feature)
        mean_value = train_set[feature].mean()
        train_set[feature] = train_set[feature].apply(
            lambda x: mean_value if x < lower_limit_train or
x > upper_limit_train else x
        )

    for feature in mean_imputation_features_val:
        print(f"Imputing outliers with mean for feature
(Validation Set): {feature}")
        _, _, _, lower_limit_val, upper_limit_val =
```

```
iqr(val_set, feature)
        mean_value = val_set[feature].mean()
        val_set[feature] = val_set[feature].apply(
            lambda x: mean_value if x < lower_limit_val or x
> upper_limit_val else x
        )

    for feature in isolation_forest_features_train +
isolation_forest_features_val:
        print(f"Applying Isolation Forest for feature:
{feature}")
        iso_forest = IsolationForest(contamination=0.01,
random_state=42, n_estimators=300)
        combined_data = pd.concat([train_set[feature],
val_set[feature]], axis=0).to_frame()
        outlier_predictions =
iso_forest.fit_predict(combined_data)
        outlier_mask = outlier_predictions == -1

        combined_data = combined_data[~outlier_mask]
        train_set =
train_set.loc[train_set.index.intersection(combined_data.ind
ex)]
        val_set =
val_set.loc[val_set.index.intersection(combined_data.index)]

        print(f"Isolation Forest removed
{outlier_mask.sum()} outliers for feature: {feature}")

   return train_set, val_set

train_set, val_set = handle_outliers(train_set, val_set)
```

Kode ini menangani outlier dengan tiga cara berdasarkan tingkat keparahan outlier pada dataset.

- Capping: Mengatasi outlier dengan membatasi nilai ke rentang IQR apabila persentase outliernya lebih dari 5%
- Mean Imputation: Menggantikan outlier dengan nilai rata-rata apabila persentase outliernya antara 1% sampai 5%
- Isolation Forest: Menghapus outlier ekstrem yang jarang terjadi (< 1%).

2.1.3. *Remove Duplicates*

Mengatasi data duplikat sangat penting karena dapat memengaruhi keakuratan hasil analisis. Data yang ganda berisiko menyebabkan bias pada model *machine learning*, mengakibatkan *overfitting*, dan menyulitkan model untuk memahami data baru. Selain itu, keberadaan data duplikat memperbesar ukuran dataset secara tidak perlu, yang pada

akhirnya meningkatkan waktu dan biaya pemrosesan. Data duplikat juga dapat menyebabkan hasil statistik menjadi tidak konsisten, sehingga mempengaruhi keandalan keputusan berbasis data. Oleh karena itu, membersihkan data dari duplikat adalah langkah penting untuk memastikan analisis yang akurat, efisien, dan terpercaya.

```
print("Before drop duplicates in training set:",
train_set.duplicated().sum())
print("Before drop duplicates in validation set:",
val_set.duplicated().sum())
```

Output:

```
Before drop duplicates in training set: 0
Before drop duplicates in validation set: 0
```

Karena tidak ada yang *duplicate* maka proses *remove duplicates* tidak perlu dilakukan.

2.1.4. *Feature Engineering*

Feature engineering merupakan proses yang dilakukan untuk menciptakan fitur baru atau memodifikasi fitur yang sudah ada dengan tujuan peningkatan kinerja model *machine learning*. Proses ini bertujuan untuk membantu model dalam mengenali pola dan menghasilkan prediksi yang lebih akurat.

Dalam feature engineering, biasanya dilakukan pencarian terhadap fitur yang memiliki korelasi tinggi, lalu menghapus salah satu fitur tersebut agar data menjadi lebih efisien dan mengurangi risiko overfitting.

Dilakukan perhitungan korelasi antar fitur numerik dalam dataset, kemudian menampilkan *heatmap* untuk visualisasi hubungan antar fitur dengan menggunakan warna, serta mencetak matriks korelasi tersebut.

```
corr =  train_set_non_cat.corr()
plt.figure(figsize=(15, 12))

sns.heatmap(corr, annot=False, cmap='coolwarm', center=0,
linewidths=0.5, fmt=".2f")

plt.title("Heatmap of Numerical Feature Correlations")
plt.show()

print(corr)
```

Dilakukan pencarian korelasi antar fitur dalam matriks korelasi, mencetak fitur-fitur yang memiliki korelasi lebih dari 0.75 atau kurang dari -0.75 dengan fitur lain, dan menampilkan hasilnya jika ada.

```
for c in corr.columns:
    corr_c = corr[c]
    corr_c = corr_c[((corr_c > 0.75) & (corr_c < 1)) |
((corr_c < -0.75) & (corr_c > -1))].dropna()
    if not corr_c.empty:
        print(f"\nCorrelation for Feature '{c}':")
        print(corr_c.sort_values(ascending=False))
```

Dilakukan pencarian fitur yang memiliki korelasi tinggi dengan fitur lain :

```
print("Highly Correlated Features (Positive and Negative):")
high_corr_pairs = []
for col in corr.columns:
    corr_c = corr[col]

    corr_c = corr_c[((corr_c > 0.75) & (corr_c < 1)) |
((corr_c < -0.75) & (corr_c > -1))].dropna()
    for idx in corr_c.index:

        if (idx, col) not in high_corr_pairs and (col, idx)
not in high_corr_pairs:
            high_corr_pairs.append((col, idx))
            print(f"{col} - {idx}: {corr_c[idx]:.2f}")

if not high_corr_pairs:
    print("No features with high positive or negative
correlation found.")
```

Lalu, dilakukan *dropping* fitur yang memiliki korelasi tinggi dengan fitur lainnya.

```
features_to_drop = [
    'NoOfLettersInURL',
    'NoOfOtherSpecialCharsInURL',
    'NoOfObfuscatedChar',
    'NoOfDegitsInURL',
    'LetterRatioInURL',
    'SpacialCharRatioInURL',
    'URLTitleMatchScore'
]

train_set = train_set.drop(columns=features_to_drop)
val_set = val_set.drop(columns=features_to_drop)

print("Dropped features:", features_to_drop)
```

2.2. Data Preprocessing

Data preprocessing merupakan proses mempersiapkan data agar lebih siap untuk diolah lebih lanjut dalam rangka ekstraksi pengetahuan. Prosesnya yang dilakukan adalah dengan mentransformasikan data menjadi sebuah format yang *clean* dan terstruktur. Tujuan dari data preprocessing ini adalah untuk meningkatkan kualitas data dan menjadikan lebih sesuai dengan algoritma *machine learning*.

2.2.1. *Feature Scaling*

Feature scaling merupakan proses yang dilakukan untuk memastikan bahwa tiap fitur numerikal memiliki skala yang seragam. Hal ini bertujuan untuk mencegah fitur dengan skala yang lebih besar mendominasi model, terutama pada algoritma yang sensitif terhadap penskalaan, seperti algoritma k-nearest neighbors (KNN). Dengan mentransformasikan data ke skala yang seragam, proses ini akan meningkatkan *performance* model dan menghindari ketidakstabilan numerik dalam kalkulasi.

```
      class FeatureScaler(BaseEstimator,
TransformerMixin):
          def __init__(self):
              self.standard_scaler = StandardScaler()

          def fit(self, X, y=None):
              self.standard_scaler.fit(X)
              return self

          def transform(self, X):
              X_scaled =
self.standard_scaler.transform(X)
              X_scaled_df = pd.DataFrame(X_scaled,
columns=X.columns, index=X.index)
              return X_scaled_df
```

Feature scaling pada dataset ini dilakukan dengan menggunakan metode standarisasi. Dengan metode ini, nilai setiap fitur akan disesuaikan sehingga memiliki nilai rata-rata 0 dan standar deviasi 1, sehingga semua fitur dipastikan akan berada pada skala yang sebanding. Metode ini dianggap lebih sesuai dengan dataset yang memiliki distribusi data yang tidak seragam, sehingga dapat meningkatkan *performance* model yang sensitif terkait skala fitur, seperti KNN dengan lebih baik.

2.2.2. *Feature Encoding*

Feature encoding merupakan proses mengonversi data kategorikal (non-numerik) menjadi data dalam format numerik. Hal ini penting dilakukan sebab sebagian besar algoritma *machine learning* memerlukan input dalam bentuk numerikal. Data kategorikal dapat berupa nominal atau ordinal. Proses *encoding* memastikan bahwa data kategorikal tetap dapat bernilai dalam proses kalkulasi.

```
      class FeatureEncoder(BaseEstimator,
TransformerMixin):
          def __init__(self, unknown_label="unknown"):
              self.encoders = {}
              self.unknown_label = unknown_label

          def fit(self, X, y=None):
              for feature in
X.select_dtypes(include=['object']).columns:
                  unique_values = set(X[feature])
                  if self.unknown_label not in
unique_values:

unique_values.add(self.unknown_label)

                  encoder = LabelEncoder()
                  encoder.fit(list(unique_values))
                  self.encoders[feature] = encoder
              return self

          def transform(self, X):
              transformed_X = X.copy()

              for feature in self.encoders:
                  if feature in transformed_X.columns:
                      transformed_X[feature] =
transformed_X[feature].map(
                          lambda val: val if val in
self.encoders[feature].classes_ else self.unknown_label
                      )
                      transformed_X[feature] =
self.encoders[feature].transform(transformed_X[feature])

              return transformed_X
```

Feature encoding pada dataset ini dilakukan dengan menggunakan metode *label encoding*. Dengan metode ini, tiap kategori pada suatu fitur akan diberikan sebuah *unique integer*. Jika terdapat kategori baru yang tidak ada di dalam data *training*, data tersebut akan diberi label khusus, seperti *unknown.* Metode ini dipilih karena lebih sederhana dan efisien dibandingkan metode lainnya.

2.2.3. Handling Imbalanced Dataset

Menangani dataset yang tidak seimbang sangat penting untuk meningkatkan *performance* model *machine learning* sebab ketidakseimbangan dataset dapat menyebabkan hasil yang bias dan prediksi yang kurang *reliable*. Ketika suatu kelas mendominasi, model cenderung akan jauh lebih fokus ke kelas *majority* yang akan berdampak pada kesalahan metrik akurasi yang dihasilkan. Selain itu, model yang dilatih dengan dataset yang *imbalance* juga kurang dapat menggeneralisasi data baru dengan baik.

Untuk menangani dataset yang tidak seimbang ini, terdapat beberapa metode yang dapat diterapkan. Teknik *resampling* seperti *oversampling* (menambah jumlah data kelas minoritas) atau *undersampling* (mengurangi jumlah data kelas mayoritas) dapat membantu untuk mengurangi ketidakseimbangan tersebut. Selain itu, penggunaan *evaluation metrics* yang lebih informatif, seperti presisi, *recall*, F1-*score*, dan *confusion matrix* dibandingkan hanya akurasi sangat disarankan karena dapat menggambarkan *performance* model dengan lebih baik. Dari sisi pendekatan algoritma, pilih algoritma yang dirancang untuk menangani data tidak seimbang serta sesuaikan bobot kelas untuk memberikan perhatian lebih pada kelas minoritas, sehingga model dapat lebih akurat dalam memprediksi kelas.

```
target_samples = {
    '1': 3500,
    '0': 780
}

sampled_data = []

train_set['label'] = train_set['label'].astype(str)

for category in train_set['label'].unique():
    category_data = train_set[train_set['label'] ==
category]
    target = target_samples.get(category,
len(category_data))

    if target >= len(category_data):
        sampled_data.append(category_data)
    else:
        sampled_data.append(category_data.sample(n=target,
random_state=42))

train_set_reduced = pd.concat(sampled_data, axis=0)

print("Distribution after reduction:")
print(train_set_reduced['label'].value_counts())

train_set = train_set_reduced
```

```
class HandleImbalance(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.smote = SMOTE(random_state=42)

    def fit(self, X, y=None):
        self.X_balanced, self.y_balanced =
self.smote.fit_resample(X, y)
        return self

    def transform(self, X, y=None):
        return X
```

Proses *handling imbalanced dataset* dilakukan dengan melakukan *resampling* dan SMOTE (S*ynthetic Minority Over-sampling Technique*). *Resampling* dilakukan dengan mengubah distribusi data pada kelas target dengan mengurangi jumlah data pada kelas *majority* dan menambah data pada kelas *minority* menggunakan *random sampling*. Hal ini bertujuan untuk meningkatkan keseimbangan kelas tanpa harus mengorbankan representasi dari data.

Selanjutnya, SMOTE digunakan untuk mensintesis sampel baru dari kelas minoritas untuk menyeimbangkan dataset. Penggunaan random_state=42 memastikan bahwa proses *resampling* dapat diulang dan menghasilkan hasil yang konsisten setiap kali dijalankan, sehingga model *machine learning* yang dilatih tidak bias dan dapat memprediksi kedua kelas dengan lebih baik.

2.2.4. Pipelining

Pipelining merupakan suatu teknik yang digunakan untuk melakukan penyusunan langkah-langkah *preprocessing data* dan *model training* dalam urutan yang terstruktur dan terotomatisasi. Tujuannya adalah untuk memastikan konsistensi pelaksanaan setiap tahap *preprocessing*, seperti *encoding, scaling*, dan penanganan *imbalance data*, yang meminimalkan risiko kesalahan dan memastikan alur kerja yang efisien. *Pipeline* memastikan transformasi data yang sama diterapkan pada data pelatihan dan data uji.

Pada proyek ini, digunakan model KNN (*K-Nearest Neighbor*) dan GNB (*Gaussian -Naive Bayes*) untuk melakukan *training* dan prediksi dataset. Dalam hal ini, terdapat perbedaan kebutuhan antara kedua model sehingga perumusan *pipeline* dilakukan secara terpisah yang menghasilkan dua *pipeline* yang berbeda.

```
knn_model_pipeline = Pipeline([

    ('categorical_encoding', FeatureEncoder()),
    ('feature_scaling', FeatureScaler()),
    ('smote_resampling', HandleImbalance())
])

gnb_model_pipeline = Pipeline([
    ('categorical_encoding', FeatureEncoder()),
    ('smote_resampling', HandleImbalance())
])
```

Nilai variabel val_set yang tidak ada di train_set diganti dengan label 'unknown'. Hal ini ditujukan untuk menangani kategori yang belum terlihat di data pelatihan agar tidak menyebabkan kesalahan saat pemodelan.

```
for feature in cat_cols:
                                val_set[feature]       =
val_set[feature].where(val_set[feature].isin(train_set[fe
```

```
   ature].unique()), 'unknown')
```

Dilakukan pembagian dataset ke dalam variabel-variabel khusus secara terpisah untuk memudahkan proses *modeling* dan *data training*. Variabel X_train_knn_model dan X_val_knn_model berisi fitur dari data pelatihan (train_set) dan data validasi (val_set) yang diproses dengan pipeline model KNN, sementara variabel y_train_knn_model dan y_val_knn_model berisi label target dari data pelatihan dan validasi yang digunakan untuk pelatihan dan evaluasi model. X_train dan X_val adalah fitur, sedangkan y_train dan y_val adalah label target.

```
X_train_knn_model =
knn_model_pipeline.fit(train_set.drop(columns=['label']),
train_set['label']).transform(train_set.drop(columns=['label
']))
X_val_knn_model =
knn_model_pipeline.transform(val_set.drop(columns=['label'])
)
y_train_knn_model, y_val_knn_model = train_set['label'],
val_set['label']
```

BAB 3
PERBANDINGAN HASIL PREDIKSI

3.1. KNN

	precision	recall	f1-score	support
0	0.88	0.33	0.48	1981
1	0.95	1.00	0.97	25566
accuracy			0.95	27547
macro avg	0.91	0.67	0.73	27547
weighted avg	0.95	0.95	0.94	27547

KNN from scratch

	precision	recall	f1-score	support
0	0.88	0.33	0.48	1981
1	0.95	1.00	0.97	25566
accuracy			0.95	27547
macro avg	0.91	0.67	0.73	27547
weighted avg	0.95	0.95	0.94	27547

KNN *scikit-learn*

Seperti yang terlihat pada gambar, hasil implementasi KNN menggunakan metode from scratch dan menggunakan scikit-learn menghasilkan performa yang sama persis, baik dalam hal metrik seperti precision, recall, F1-score, maupun akurasi. Hal ini menunjukkan bahwa algoritma yang diimplementasikan secara manual (from scratch) sudah sesuai dengan metode yang digunakan dalam library scikit-learn. Namun, terdapat perbedaan yang sangat signifikan dalam waktu pemrosesan antara kedua pendekatan ini. Saat menggunakan KNN from scratch, proses penghitungan membutuhkan waktu belasan menit, yang kemungkinan besar disebabkan oleh efisiensi algoritma dan cara pengolahan data yang lebih sederhana. Sebaliknya, implementasi KNN dengan scikit-learn hanya membutuhkan waktu beberapa detik saja untuk menyelesaikan proses yang sama.

Perbedaan ini muncul karena scikit-learn menggunakan optimasi algoritma dan teknik pengolahan data yang lebih efisien, seperti penggunaan struktur data yang dioptimalkan (contoh: KD-Tree atau Ball-Tree) dan pemanfaatan komputasi paralel. Di sisi lain, implementasi from scratch dilakukan dengan algoritma dasar tanpa optimasi tambahan, sehingga lebih lambat.

Performa KNN sangat baik untuk label 1 tetapi kurang optimal untuk label 0. Hal ini kemungkinan besar disebabkan oleh ketidakseimbangan jumlah data antar label, di mana label 1 memiliki jumlah data yang jauh lebih besar dibandingkan label 0. Akibatnya, model cenderung bias terhadap label 1, sehingga recall untuk label 0 menjadi rendah.

3.2. Gaussian Naive-Bayes

	precision	recall	f1-score	support
0	1.00	0.09	0.16	1981
1	0.93	1.00	0.97	25566
accuracy			0.93	27547
macro avg	0.97	0.54	0.56	27547
weighted avg	0.94	0.93	0.91	27547

GNB from scratch

	precision	recall	f1-score	support
0	1.00	0.04	0.08	1981
1	0.93	1.00	0.96	25566
accuracy			0.93	27547
macro avg	0.97	0.52	0.52	27547
weighted avg	0.94	0.93	0.90	27547

GNB *scikit-learn*

Seperti yang terlihat pada gambar, hasil implementasi GNB menggunakan metode from scratch dan menggunakan scikit-learn menghasilkan performa yang sangat mirip dan hanya terdapat sedikit perbedaan dalam hal metrik seperti precision, recall, F1-score, maupun akurasi.

Akurasi dengan *scratch* dan dengan scikit-learn menunjukkan hasil yang serupa. Akan tetapi, meskipun akurasi keseluruhan terlihat tinggi, skor *recall* dan f1-*score* untuk label 0 relatif rendah untuk kedua metode. Hal ini menunjukkan bahwa model tidak memiliki performa yang cukup baik di kelas minoritas ini. Model lebih sering memprediksi label 1 dan kurang sensitif terhadap label 0. *Recall* yang rendah mengindikasikan kedua model relatif gagal untuk mengidentifikasi sebagian besar *instance* label 0 dengan baik. Skor f1-*score* yang juga relatif rendah semakin memperkuat bahwa model tidak terlalu mampu untuk merepresentasikan kelas label 0 dengan benar.

Sementara itu, nilai seluruh *accuration matrix* untuk label 1 hampir identik dalam kedua implementasi. Hal ini menunjukkan bahwa kedua metode memiliki performa yang baik untuk kelas mayoritas.

Terdapat sedikit perbedaan pada waktu proses antara metode *scratch* dan scikit-learn, di mana pemrosesan metode *scratch* berlangsung sedikit lebih lama dibandingkan scikit-learn. Hal ini dapat disebabkan optimasi yang dimiliki oleh scikit-learn, seperti penggunaan struktur data dan algoritma yang lebih efisien. Meskipun demikian, meskipun ada sedikit perbedaan, kedua metode menunjukkan performa yang sangat mirip dalam hal akurasi (sedikit berbeda jika tingkat ketelitian akurasinya diperbesar). Hal ini menunjukkan bahwa implementasi GNB dengan *scratch* dapat memberikan hasil yang cukup setara dengan metode yang lebih terstruktur seperti scikit-learn.

Performa GNB sangat baik untuk label 1, tetapi kurang optimal untuk label 0. Hal ini kemungkinan besar disebabkan oleh ketidakseimbangan jumlah data antar tabel, di mana label 1 memiliki jumlah data yang jauh lebih besar dibandingkan label 0. Akibatnya, model cenderung bias terhadap kelas mayoritas (label 1) dan recall untuk label 0 pun menjadi rendah.

Model GNB memang cukup sensitif terhadap data yang tidak seimbang. Hal ini disebabkan asumsi independensi antar fitur dalam model ini akan lebih menguntungkan jika data relatif seimbang. Ketika data tidak seimbang, model cenderung mengabaikan kelas yang lebih minor sehingga menyebabkan bias dalam prediksi.

BAB 4
KESIMPULAN

Dalam proyek ini, praktik *preprocessing* dan evaluasi data yang strategis diterapkan untuk mengatasi tantangan dan meningkatkan kinerja model. Penerapan pipeline modular untuk memastikan konsistensi dalam transformasi data di seluruh dataset, sementara solusi-target seperti SMOTE dan label encoding secara efektif menangani ketidakseimbangan kelas dan label yang tidak terduga. Proses *cleaning data* yang komprehensif meningkatkan kualitas dataset dengan mengelola nilai yang hilang, *outlier*, dan fitur yang tidak relevan, memastikan data yang digunakan dalam pelatihan model lebih bersih dan siap pakai.

Selain itu, penyesuaian ukuran dataset dan alur kerja yang lebih efisien membantu mengurangi waktu pelatihan yang lama, memungkinkan pengembangan model yang lebih cepat dan efektif. Proses evaluasi model dilakukan dengan menggunakan berbagai metrik, yakni *precision, recall,* F1-score, maupun akurasi, yang memberikan gambaran menyeluruh tentang kinerja model dan menjadi dasar untuk perbaikan lebih lanjut. Melalui evaluasi ini, kami memperoleh wawasan yang lebih dalam mengenai kekuatan dan kelemahan masing-masing model yang menjadi landasan untuk tahap pelatihan berikutnya. Upaya-upaya ini secara keseluruhan membangun fondasi yang kokoh untuk pemodelan yang dapat diandalkan dan prediksi yang akurat, menunjukkan betapa pentingnya *preprocessing* yang cermat dan validasi yang teliti dalam mencapai hasil yang sukses.

PEMBAGIAN TUGAS

NIM	Nama	Pembagian Tugas
18222008	Abel Apriliani	Notebook initialization, EDA, Data Cleaning, Pipelining, Validation, Submission, Report (Help Data Cleaning)
18222036	Olivia Christy L.	Data Preprocessing, Pipelining, Submission, Error Analysis, Report (Help Data Cleaning, Data Preprocessing, dan Kesimpulan)
18222044	Khansa Adilla Reva	KNN Implementation, Submission, Report (Data Cleaning & KNN)
18222062	Nafisha Virgin	Naive Bayes Implementation, Report (Data Preprocessing & GNB)

REFERENSI

Arwan, dkk. (2018, 8 Juni). *Synthetic Minority Over-sampling Technique (SMOTE) Algorithm For Handling Imbalanced Data*. Diakses dari https://mti.binus.ac.id/2018/06/08/synthetic-minority-over-sampling-technique-smote-algorithm-for-handling-imbalanced-data/

GeeksforGeeks. (2024, 12 September). *Data Preprocessing in Data Mining*. Diakses dari https://www.geeksforgeeks.org/data-preprocessing-in-data-mining/

GeeksforGeeks. (n.d.). *Implementation of k-Nearest Neighbors from scratch using Python*. Diakses dari https://www.geeksforgeeks.org/implementation-of-k-nearest-neighbors-from-scratch-using-python/

Kashidafe. (2024, 23 Maret). *Gaussian Naive Bayes: Understanding the Basics and Applications*. Diakses dari https://medium.com/@kashishdafe0410/gaussian-naive-bayes-understanding-the-basics-and-applications-52098087b963

Kassem. (2024, 26 Agustus). *Building a Naive Bayes Classifier from Scratch: A Step-by-Step Guide*. https://elcaiseri.medium.com/building-a-naive-bayes-classifier-from-scratch-a-step-by-step-guide-557638e2d878

Universitas Medan Area. (2023, 16 Februari). *Algoritma k-Nearest Neighbors (KNN): Pengertian dan Penerapan*. Diakses dari https://lp2m.uma.ac.id/2023/02/16/algoritma-k-nearest-neighbors-knn-pengertian-dan-penerapan/