LAPORAN TUGAS BESAR 1 IF3170 INTELEGENSI ARTIFISIAL
IMPLEMENTASI ALGORITMA PEMBELAJARAN MESIN

Diajukan sebagai pemenuhan tugas besar 2

Oleh:

Kelompok 22

1. 13522063 - Shazya Audrea Taufik
2. 13522072 - Ahmad Mudabbir Arif
3. 13522085 - Zahira Dina Amalia
4. 13522116 - Naufal Adnan

Dosen Pengampu :

1. Dr. Nur Ulfa Maulidevi, S.T, M.Sc.
2. Dr. Eng. Ayu Purwarianti, S.T., M.T

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2024

Daftar Isi

1. Implementasi KNN

Kelas KNN dibangun untuk memuat seluruh fungsi utama dari algoritma KNN, mulai dari inisialisasinya hingga fungsi-fungsi yang dibuat untuk menghitung jarak antar satu titik dengan titik lainnya dan mendapatkan tetangga terdekat. Kelas ini dibuat dengan menggunakan metode `__init__` tiga parameter, yaitu:

a. Parameter k: jumlah tetangga terdekat yang dipertimbangkan dalam prediksi.
b. Parameter metric: jenis matrik jarak yang digunakan (Euclidean, Manhattan, atau Minkowski).
c. Parameter p: digunakan untuk menentukan orde dalam perhitungan Minkowski distance.

```
class KNN:
  def __init__(self, k=5, metric='minkowski', p=2):
      self.k = k
      self.metric = metric

      if self.metric == 'manhattan':
          self.p = 1
      elif self.metric == 'euclidean':
          self.p = 2
      else:
          self.p = p

  def _get_nearest_neighbours(self, test):
      distances = np.linalg.norm(self.X_train - test, ord=self.p, axis=1)
      indices = np.argsort(distances)[:self.k]
      return indices

  def fit(self, X_train, y_train):
      if isinstance(X_train, pd.DataFrame):
          if X_train.columns.empty:
              self.X_train = X_train.values.astype(float)
          else:
              self.X_train = X_train.iloc[:, :-1].values.astype(float)
```

```
        else:
            self.X_train = X_train.astype(float)
        self.y_train = y_train

    def _predict_instance(self, row):
        indices = self._get_nearest_neighbours(row)
        labels = [self.y_train.iloc[neighbour] for neighbour in indices]
        prediction = max(set(labels), key=labels.count)
        return prediction

    def predict(self, X_test):
        if isinstance(X_test, pd.DataFrame):
            if X_test.columns.empty:
                X_test = X_test.values.astype(float)
            else:
                X_test = X_test.iloc[:, :-1].values.astype(float)
        else:
            X_test = X_test.astype(float)
        start_time = time.time()
        with concurrent.futures.ProcessPoolExecutor() as executor:
            results = list(tqdm(executor.map(self._predict_instance, X_test),
total=len(X_test)))

        elapsed_time = time.time() - start_time
        print(f"Prediction completed in {elapsed_time:.2f} seconds.")
        return np.array(results)

    def save(self, path):
        pickle.dump(self, open(path, 'wb'))

    @staticmethod
    def load(path):
        return pickle.load(open(path, 'rb'))
```

KNN ini diimplementasi dengan mendukung tiga jenis matrik jarak (Euclidean, Manhattan, dan Minkowski), mendukung paralelisasi untuk

meningkatkan efisiensi prediksi pada dataset besar, fleksibilitas dalam menerima format input data (DataFrame atau numpy array), serta adanya tempat untuk menyimpan dan memuat model untuk mendukung *reusability*. Berikut penjelasan dari setiap fungsi/metode yang ada pada kelas KNN yang telah dibuat.

Metode	Penjelasan
`_get_nearest_neighbours(self, test)`	Digunakan untuk menyimpan data latih (X_train) dan labelnya (y_train) sebagai referensi dalam proses prediksi dengan input berupa DataFrame atau array numpy.
`fit(self, X_train, y_train)`	Menghitung jarak antara satu titik data uji dengan semua data latih menggunakan metrik jarak yang dipilih. • *Euclidean Distance* untuk p=2, $\sqrt{\sum(x_i - y_i)^2}$ • *Manhattan Distance* untuk p=1, $\sum \lvert x_i - y_i \rvert$ • *Minkowski Distance*: $(\sum \lvert x_i - y_i \rvert^p)^{1/p}$ Memilih k tetangga terdekat berdasarkan nilai jarak terendah.
`_predict_instance(self, row)`	Mengambil tetangga terdekat dari data uji, kemudian menentukan label prediksi berdasarkan voting mayoritas dari label tetangga tersebut.
`predict(self, X_test)`	Menerapkan proses prediksi pada seluruh data uji dengan memanfaatkan `` `ProcessPoolExecutor` `` untuk mempercepat prediksi dengan paralelisasi. Kemudian mengembalikan array prediksi untuk seluruh data uji.
`save(self, path)`	Menyimpan model ke file .pkl menggunakan pickle.

`load(path)`	Memuat model yang telah disimpan. Fitur Utama:

Dengan fleksibilitas tinggi melalui parameterisasi jumlah tetangga dan metrik jarak, implementasi ini memungkinkan pengujian kinerja model dalam berbagai konfigurasi. Selain itu, pemanfaatan paralelisasi membuat algoritma ini lebih efisien dalam menangani dataset besar seperti UNSW-NB15 dibandingkan implementasi sekuensial. Namun, algoritma ini memiliki keterbatasan, yakni sensitif terhadap ukuran dataset latih karena setiap prediksi membutuhkan perhitungan jarak dengan seluruh data latih, yang dapat memperlambat kinerja pada dataset yang sangat besar.

2. Implementasi Naive-Bayes

Kelas NaiveBayes dirancang untuk mengimplementasikan algoritma Gaussian Naive Bayes secara from scratch, yang dapat digunakan untuk melakukan klasifikasi data berdasarkan asumsi independensi antar fitur. Implementasi ini mencakup perhitungan probabilitas kelas, mean, dan varians untuk setiap fitur berdasarkan label kelas, serta prediksi menggunakan formula distribusi Gaussian.

a. Parameter `class_probs`: menyimpan probabilitas prior dari masing-masing kelas dalam dataset berdasarkan distribusi data latih. Probabilitas prior ini digunakan untuk menghitung probabilitas posterior selama prediksi. Nilai awal berupa `None` dan akan diisi saat metode `fit` dijalankan.
b. Parameter `mean`: menyimpan rata-rata (mean) dari setiap fitur untuk setiap kelas dalam data latih. Mean ini digunakan dalam perhitungan probabilitas menggunakan distribusi Gaussian. Nilai awal berupa `None` dan akan diisi saat metode `fit` dijalankan.
c. Parameter `variance`: menyimpan varians dari setiap fitur untuk setiap kelas dalam data latih. Varians digunakan bersama dengan mean untuk menghitung probabilitas Gaussian. Nilai awal berupa `None` dan akan diisi saat metode `fit` dijalankan.

```
from collections import defaultdict


class NaiveBayes:
```

```
    def __init__(self):
        self.class_probs = None
        self.mean = None
        self.variance = None

    def fit(self, X, y):
        self.class_probs = self._calc_class_prob(y)
        self.mean, self.variance = self._calc_mean_var(X, y)

    def _calc_class_prob(self, y):
        count_cl = defaultdict(int)
        total = len(y)
        for label in y:
            count_cl[label] += 1
        class_probs = {label: count / total for label, count in
count_cl.items()}
        return class_probs

    def _calc_mean_var(self, X, y):
        unique = np.unique(y)
        mean = {}
        variance = {}
        for label in unique:
            data = X[y == label]
            mean[label] = np.mean(data, axis=0)
            variance[label] = np.var(data, axis=0)
        return mean, variance

    def _gauss(self, x, mean, variance):
        exp = np.exp(-((x - mean) ** 2) / (2 * variance))
        return (1 / (np.sqrt(2 * np.pi * variance))) * exp

    def _calc_feature_probs(self, features, label):
        class_prob = np.log(self.class_probs[label])

        for i, feature in enumerate(features):
```

```
            mean = self.mean[label][i]
            variance = self.variance[label][i]
            epsilon = 1e-10
            if variance < epsilon:
                variance = epsilon

            class_prob += np.log(
                self._gauss(feature, mean, variance)
            )
        return class_prob

    def predict(self, X):
        results = []
        for sample in X.values:
            class_probs = {
                label: self._calc_feature_probs(sample, label)
                for label in self.class_probs
            }
            predicted_class = max(class_probs, key=class_probs.get)
            results.append(predicted_class)

        return np.array(results)

    def save(self, path):
        pickle.dump(self, open(path, 'wb'))
    @staticmethod
    def load(path):
        return pickle.load(open(path, 'rb'))
```

Implementasi Naive Bayes ini fleksibel dalam menangani data multikelas dan didesain secara modular untuk kemudahan pemrosesan. Dengan pendekatan Gaussian, algoritma ini disesuaikan untuk data dengan fitur numerik. Selain itu, kelas ini memiliki kemampuan untuk menyimpan dan memuat model.

Metode	Penjelasan
`_calc_class_prob(self, y)`	Menghitung probabilitas setiap kelas dengan menghitung jumlah kemunculan label dan membaginya dengan total data.
`_calc_mean_var(self, X, y)`	Menghitung mean dan varians untuk setiap fitur berdasarkan label kelas yang sesuai.
`_gauss(self, x, mean, variance)`	Menghitung probabilitas distribusi Gaussian untuk sebuah fitur dengan mean dan varians tertentu.
`_calc_feature_probs(self, features, label)`	Menghitung probabilitas posterior untuk sebuah sampel data dengan menjumlahkan log-probabilitas setiap fitur.
`fit(self, X, y)`	Melatih model dengan menghitung probabilitas kelas, mean, dan varians untuk tiap fitur berdasarkan data latih (X) dan label (y).
`predict(self, X)`	Melakukan prediksi label untuk data uji (X) dengan memilih label dengan probabilitas posterior tertinggi.
`save(self, path)`	Menyimpan model ke file .pkl menggunakan pickle.
`load(path)`	Memuat model yang telah disimpan. Fitur Utama:

Naive Bayes ini sensitif terhadap asumsi independensi antar fitur, sehingga performa dapat menurun jika fitur dalam dataset memiliki korelasi yang kuat. Selain itu, jika terdapat varians yang sangat kecil atau nol pada suatu fitur, hasil prediksi dapat terpengaruh kecuali dilakukan penyesuaian (seperti penambahan epsilon).

3. Implementasi ID3

Kelas ID3DecisionTree dirancang untuk merepresentasikan algoritma ID3 (Iterative Dichotomiser 3) dalam membangun pohon keputusan berdasarkan konsep Information Gain dan Entropy. Kelas ini terdiri atas berbagai metode yang berfungsi untuk melatih model, membuat pohon keputusan, dan melakukan prediksi.

d. Parameter `tree`: struktur data untuk menyimpan pohon keputusan yang dihasilkan.
e. Parameter `default_class`: kelas default yang digunakan jika tidak ada keputusan spesifik yang bisa diambil.
f. Parameter `x, y`: matriks fitur (independen) dan label (dependen) yang digunakan untuk pelatihan dan prediksi.

```
class ID3DecisionTree:
  def __init__(self):
      self.tree = None
      self.default_class = None

  def entropy(self, y):
      class_counts = Counter(y)
      total = len(y)
      return -sum(
          (count / total) * np.log2(count / total)
          for count in class_counts.values()
          if count > 0
      )

  def information_gain(self, X_column, y):
      total_entropy = self.entropy(y)
      values, counts = np.unique(X_column, return_counts=True)
      weighted_entropy = sum(
          (counts[i] / len(y)) * self.entropy(y[X_column == value])
          for i, value in enumerate(values)
      )
      return total_entropy - weighted_entropy

  def best_split(self, X, y):
      best_gain = -1
      best_feature = None
      for feature in range(X.shape[1]):
          gain = self.information_gain(X[:, feature], y)
```

```
            if gain > best_gain:
                best_gain = gain
                best_feature = feature
        return best_feature

    def build_tree(self, X, y, features):
        if len(set(y)) == 1:
            return y[0]
        if len(features) == 0 or X.shape[1] == 0:
            return Counter(y).most_common(1)[0][0]

        best_feature = self.best_split(X, y)
        tree = {features[best_feature]: {}}

        feature_values = np.unique(X[:, best_feature])
        for value in feature_values:
            subset_indices = X[:, best_feature] == value
            subset_X = X[subset_indices]
            subset_y = y[subset_indices]
            new_features = np.delete(features, best_feature)
            new_X = np.delete(subset_X, best_feature, axis=1)
            tree[features[best_feature]][value] = self.build_tree(
                new_X, subset_y, new_features
            )
        return tree

    def fit(self, X, y):
        X = np.array(X)
        y = np.array(y)
        self.default_class = Counter(y).most_common(1)[0][0]
        features = np.array([f"Feature {i}" for i in range(X.shape[1])])
        self.tree = self.build_tree(X, y, features)

    def predict_one(self, x, tree):
        if not isinstance(tree, dict):
            return tree
```

```
        root_feature = next(iter(tree))
        feature_idx = int(root_feature.split()[-1])
        if feature_idx >= len(x):
            return self.default_class
        feature_value = x[feature_idx]
        subtree = tree[root_feature].get(feature_value, self.default_class)
        return self.predict_one(x, subtree)

    def predict(self, X):
        X = np.array(X)
        return np.array([self.predict_one(x, self.tree) for x in X])

    def save(self, path):
        pickle.dump(self, open(path, 'wb'))

    @staticmethod
    def load(path):
        return pickle.load(open(path, 'rb'))
```

Fitur utama dari implementasi ID3 Decision Tree adalah penggunaan Entropy dan Information Gain untuk mengukur kualitas pemisahan data, sehingga menghasilkan pohon keputusan yang optimal berdasarkan pemilihan fitur terbaik pada setiap simpul.

Metode	Penjelasan
`entropy(self, y)`	Fungsi ini menghitung nilai entropy, yang mengukur ketidakpastian atau keragaman dalam label data dengan rumus: $$H(y) = -\sum_i P_i \cdot \log_2(P_i)$$ di mana Pi adalah probabilitas kemunculan kelas i.
`information_gain(self, X_column, y)`	Menghitung Information Gain untuk setiap fitur, yaitu pengurangan entropy total setelah melakukan pemisahan berdasarkan nilai fitur tertentu, dengan rumus:

	$$IG = H(y) - \sum_v \frac{\lvert S_v \rvert}{\lvert S \rvert} \cdot H(S_v)$$ di mana H(Sv) adalah entropy subset v dari data.
`best_split(self, X, y)`	Memilih fitur terbaik berdasarkan nilai Information Gain tertinggi. Fitur ini akan menjadi akar atau simpul internal dari pohon dengan menggunakan metode iterasi untuk mengevaluasi setiap fitur pada dataset.
`build_tree(self, X, y, features)`	Fungsi rekursif untuk membangun pohon keputusan. Terdiri atas: ● Basis kasus: ○ Jika semua label dalam subset sama, kembalikan label tersebut. ○ Jika tidak ada fitur tersisa, kembalikan kelas mayoritas. ● Rekursi: ○ Pilih fitur terbaik. ○ Bagi data berdasarkan nilai fitur terbaik. ○ Buat subtree untuk setiap nilai fitur.
`fit(self, X, y)`	Menginisialisasi proses pelatihan dengan mengubah data menjadi array numpy, mengatur kelas default ke kelas yang paling sering muncul, dan memulai pembuatan pohon menggunakan `build_tree`.
`predict_one(self, x, tree)`	Memeriksa simpul pohon untuk setiap fitur, hingga mencapai daun dan mengembalikan prediksi.
`predict(self, X)`	Melakukan prediksi untuk setiap instance data dengan memanggil `predict_one` untuk seluruh data uji.
`save(self, path)`	Menyimpan model ke file .pkl menggunakan pickle.
`load(path)`	Memuat model yang telah disimpan. Fitur Utama:

Algoritma ini dirancang secara rekursif untuk membangun pohon hingga semua data dalam simpul homogen atau tidak ada fitur yang tersisa. Selain itu, implementasi ini dapat menangani dataset dengan fitur diskrit secara efektif dan menyediakan kemampuan untuk menyimpan serta memuat model, sehingga mendukung reusability dalam berbagai kasus. Namun,

algoritma ini memiliki beberapa kekurangan, terutama dalam menangani data dengan banyak fitur kontinu karena tidak adanya mekanisme pemisahan berbasis ambang batas (threshold splitting). Selain itu, kompleksitas pemisahan data pada setiap iterasi dapat menyebabkan penurunan performa pada dataset yang sangat besar.

4. Tahap Cleaning dan Preprocessing

Tahapan cleaning dan preprocessing pada kode ini bertujuan untuk mempersiapkan dataset agar siap digunakan dalam model pembelajaran mesin. Proses ini mencakup penanganan data hilang, pembuatan fitur baru, penghapusan fitur yang tidak relevan, normalisasi data, pengelompokan menggunakan clustering, dan reduksi dimensi dengan PCA.

1) Feature Imputation

   Preprocessing ini dilakukan pada kelas `FeatureImputer.` Kelas ini mengisi nilai hilang (*missing values*) pada fitur numerik dengan strategi tertentu seperti nilai tetap (*constant*) atau nilai rata-rata. Mengisi nilai hilang pada fitur kategorikal dengan nilai seperti *missing*. Melakukan encoding pada kolom kategorikal tertentu (attack_cat) menggunakan *LabelEncoder* untuk mengonversi kategori ke format numerik.

   Implementasi ini menggunakan *SimpleImputer* untuk menggantikan nilai hilang pada fitur numerik dan kategorikal. Ada pun untuk kolom id dan label dihapus karena tidak relevan dalam analisis.

```
class FeatureImputer(BaseEstimator, TransformerMixin):
  def __init__(self, num_strategy="constant", num_fill_value=0,
cat_strategy="constant", cat_fill_value="missing",
encode_attack_cat=True):
      self.num_strategy = num_strategy
      self.num_fill_value = num_fill_value
      self.cat_strategy = cat_strategy
      self.cat_fill_value = cat_fill_value
      self.encode_attack_cat = encode_attack_cat
      self.imputer = None
      self.label_encoder = None

  def fit(self, X, y=None):
      # Drop 'id' and 'label' columns
      self.columns_to_drop = ['id', 'label']
      X = X.drop(columns=self.columns_to_drop, errors='ignore')
```

```
      # Define imputers for numerical and categorical data
      self.imputer = ColumnTransformer(
          transformers=[
              ("num", SimpleImputer(strategy=self.num_strategy,
fill_value=self.num_fill_value), NUMERICALFEATURES),
              ("cat", SimpleImputer(strategy=self.cat_strategy,
fill_value=self.cat_fill_value), CATEGORICALFEATURES),
          ],
          remainder="passthrough"
      )

      # Fit the imputer
      self.imputer.fit(X)

      # Fit the LabelEncoder for 'attack_cat'
      if self.encode_attack_cat and "attack_cat" in X.columns:
          self.label_encoder = LabelEncoder()
          self.label_encoder.fit(X["attack_cat"])
          print("Labels in LabelEncoder:",
self.label_encoder.classes_)

      return self

  def transform(self, X):
      # Drop 'id' and 'label' columns
      X = X.drop(columns=self.columns_to_drop, errors='ignore')

      # Apply imputations
      X_transformed = pd.DataFrame(self.imputer.transform(X),
columns=self.imputer.get_feature_names_out())

      X_transformed.columns = [col.split("__")[-1] for col in
X_transformed.columns]

      if self.encode_attack_cat and "attack_cat" in X.columns:
          X_transformed["attack_cat"] =
self.label_encoder.transform(X["attack_cat"])

      return X_transformed
```

2) Duplicate Removal

```
initial_count = len(train_set)
df_no_duplicates = train_set.drop_duplicates()
dropped_count = initial_count - len(df_no_duplicates)

print(f"Number of dropped rows: {dropped_count}")
```

Hal ini dilakukan untuk menghapus baris duplikat dari dataset untuk memastikan integritas data dan mengurangi risiko bias dalam model akibat data yang berulang.

3) Feature Engineering

Hal ini dilakukan dengan menggunakan kelas `FeatureCreator` dengan fungsi untuk membuat fitur baru yang relevan berdasarkan kombinasi fitur yang ada, seperti:

- `total_bytes` = jumlah *bytes* yang dikirim dan diterima.
- `byte_ratio` = rasio *bytes* yang dikirim terhadap yang diterima.
- `src_to_dst_ratio` = rasio koneksi dari sumber ke tujuan.

Kita juga menghapus kolom kategorikal dari data untuk meningkatkan akurasi.

```
class FeatureCreator(BaseEstimator, TransformerMixin):
  def fit(self, X, y=None):
      self.fitted_cols = X.columns
      return self

  def transform(self, X):
      # Create new columns
      X['total_bytes'] = X['sbytes'] + X['dbytes']
      # X['packet_rate'] = X['total_bytes'] / (X['dur'] + 1e-9)
      X['byte_ratio'] = X['sbytes'] / (X['dbytes'] + 1e-9)
      X['src_to_dst_ratio'] = X['ct_srv_src'] / (X['ct_srv_dst'] +
1e-9)
      # X['port_usage_ratio'] = X['ct_dst_sport_ltm'] /
(X['ct_src_dport_ltm'] + 1e-9)

      return X

class FeatureDropper(BaseEstimator, TransformerMixin):
  def fit(self, X, y=None):
      return self

  def transform(self, X):
      return X.drop(columns=CAT_FEAT)
```

4) Feature Scaling

DIlakukan pada kelas `FeatureScaler` yang melakukan normalisasi data numerik ke rentang [0, 1] menggunakan `StandardScaler`. Skala yang konsisten memudahkan algoritma pembelajaran untuk bekerja dengan berbagai rentang nilai fitur.

```
class FeatureScaler(BaseEstimator, TransformerMixin):
  def __init__(self):
      self.scaler = None
      self.numerical_cols = None

  def fit(self, X, y=None):
```

```
      self.numerical_cols = NUMERICALFEATURES
      self.scaler = StandardScaler().fit(X[self.numerical_cols])
      return self

  def transform(self, X):
      X_scaled = X.copy()
      X_scaled[self.numerical_cols] =
self.scaler.transform(X[self.numerical_cols])

      return X_scaled
```

5) Feature Selection
   Dengan kelas `FeatureDropper`, dilakukan penghapusan fitur kategorikal dari dataset, yang mungkin tidak relevan atau sudah dikodekan di tahap sebelumnya.

```
class FeatureDropper(BaseEstimator, TransformerMixin):
  def fit(self, X, y=None):
      return self

  def transform(self, X):
      return X.drop(columns=CATEGORICALFEATURES)
```

6) Clustering
   Clustering dilakukan dengan mengelompokkan data menjadi beberapa klaster menggunakan algoritma K-Means Clustering. Kemudian ditambahkan label klaster sebagai fitur baru (cluster_label) untuk analisis lebih lanjut. Hal ini dapat dilihat pada fungsi `merge_features`

```
def merge_features(X_train, X_test, n_clusters=5, random_state=None):
    data = pd.concat([X_train, X_test], axis=0, ignore_index=True)
    kmeans = KMeans(n_clusters=n_clusters, random_state=random_state,
n_init='auto')
    data['cluster'] = kmeans.fit_predict(data)
    X_train_clustered, X_test_clustered = train_test_split(data,
test_size=len(X_test), random_state=random_state, shuffle=False)
    return X_train_clustered, X_test_clustered
```

7) Dimensionality Reduction
   Dengan kelas `pca_modified`, dimensi dataset numerik dikurangi dengan menggunakan *Principal Component Analysis* (PCA) untuk mengurangi redundansi dan meningkatkan efisiensi model. Kelas ini juga berfungsi dalam menambahkan komponen utama (*principal*

components) ke dataset untuk menggantikan fitur numerik asli dengan mengelompokkan data menjadi lebih sedikit kolom sesuai dengan kedekatan atau korelasi antar kolom.

```
class pca_modified(BaseEstimator, TransformerMixin):
    def __init__(self, n_components=2):
        self.pca = None
        self.num_features = None
        self.n_components = n_components

    def fit(self, X, y=None):
        self.num_features = NUMERICALFEATURES
        self.pca =
PCA(n_components=self.n_components).fit(X[self.num_features])
        return self

    def transform(self, X):
        X_pca = X.copy()
        principal_components =
self.pca.transform(X[self.num_features])

        for i in range(self.n_components):
            X_pca[f'PC{i + 1}'] = principal_components[:, i]

        X_pca = X_pca.drop(columns=[col for col in self.num_features
if col in X_pca.columns])

        return X_pca
```

8) Pipeline
   Untuk menggabungkan seluruh langkah *preprocessing* ke dalam satu alur kerja (pipeline). Hal ini dilakukan untuk memastikan transformasi dilakukan secara konsisten antara dataset pelatihan dan validasi.

```
pipeline = Pipeline([
    ("imputer", FeatureImputer(num_strategy="constant",
num_fill_value=0, cat_strategy="constant", cat_fill_value="missing")),
# Handle missing values
    ("featurecreator", FeatureCreator()),
    ("featurescaler", FeatureScaler()),
    ("dropper", FeatureDropper()),
    ("pca", CustomPCA(n_components=5))
])
```

9) Transformasi Data

Kemudian melatih pipeline menggunakan dataset pelatihan (train_set) dan menerapkan transformasi yang sama pada dataset validasi (val_set).

5. Perbandingan Hasil Prediksi

Perbandingan hasil prediksi dari algoritma yang diimplementasikan dengan hasil yang didapatkan dengan menggunakan pustaka. Jelaskan insight yang kalian dapatkan dari perbandingan tersebut.

1) Perbandingan Tiap Algoritma
   a. Perbandingan Algoritma KNN
      - Dengan algoritma *scratch*

Minkowski:

```
100%|██████████| 3506/3506 [02:26<00:00, 23.96it/s]
Prediction completed in 146.73 seconds.
F1 score for KNN on 10% of validation set: 0.6079
```

Manhattan: (full val set)

```
F1 Score on Validation Set: 0.3421
```

      - Dengan *library*

```
F1 score for KNeighborsClassifier on 10% of validation set: 0.5912
```

   b. Perbandingan Algoritma Naive-Bayes
      - Dengan algoritma *scratch*

```
F1 score for NaiveBayes on 10% of validation set: 0.0951
```

      - Dengan *library*

```
F1 score for GaussianNB on 10% of validation set: 0.0552
```

   c. Perbandingan Algoritma ID3
      - Dengan algoritma *scratch*

```
F1 score for ID3DecisionTree on 10% of validation set: 0.2114
```

      - Dengan *library*

```
F1 score for DecisionTreeClassifier on 10% of validation set: 0.1056
```

2) Insight yang didapatkan
   - Dalam algoritma KNN (K-Nearest Neighbors) dengan skor F1 pada *scratch* 0.6079 dan pada library 0.5912. Implementasi *scratch* memiliki performa yang lebih baik daripada implementasi pustaka KNeighborsClassifier. Hal ini bisa disebabkan oleh implementasi scratch yang lebih cocok dengan tipe data atau proses *data cleaning* dan *preprocessing* yang dibuat.
   - Dalam algoritma Naive-Bayes dengan skor F1 pada *scratch* 0.0951 dan pada library 0.0552. Implementasi *scratch* menghasilkan performa lebih baik dibandingkan pustaka GaussianNB. Perbedaan kecil ini bisa disebabkan oleh cara perhitungan probabilitas Gaussian atau penanganan varians mendekati nol dalam implementasi scratch. Namun, secara keseluruhan, skor F1 rendah menunjukkan keterbatasan Naive-Bayes dalam menangani dataset yang kompleks seperti UNSW-NB15.
   - Dalam algoritma ID3 (Decision Tree) dengan skor F1 pada *scratch* 0.2114 dan pada library 0.1056. Implementasi *scratch* memiliki skor F1 yang jauh lebih tinggi dibandingkan pustaka DecisionTreeClassifier dengan criterion='entropy'. Ini bisa disebabkan oleh perbedaan kriteria penghentian atau cara implementasi ID3 dari scratch yang lebih eksplisit dalam menangani atribut dan rekursi pohon. Library mungkin lebih sensitif terhadap data yang memiliki noise atau overfitting karena pohon lebih dalam.

3) Kesimpulan
   - **KNN** adalah algoritma dengan performa terbaik di antara ketiganya, baik pada implementasi *scratch* maupun pustaka, karena KNN cocok untuk data yang kompleks dengan pola jarak yang jelas.
   - **Naive-Bayes** memiliki performa paling rendah karena asumsi independensi antar fitur tidak sesuai dengan kompleksitas dataset UNSW-NB15.
   - **ID3** memberikan performa lebih baik dalam implementasi *scratch* dibandingkan pustaka, menunjukkan efektivitas implementasi

manual dalam menangani pohon keputusan dengan kontrol yang lebih eksplisit.

4) Saran
Fokus pada optimasi KNN dengan pemilihan parameter k dan metode jarak yang tepat. Untuk ID3, eksplorasi lebih lanjut terkait pruning atau kriteria penghentian pohon.

6. Kontribusi

NIM	Nama	Pembagian Kerja
13522063	Shazya Audrea Taufik	KNN, Laporan
13522072	Ahmad Mudabbir Arif	ID3, Laporan
13522085	Zahira Dina Amalia	Naive Bayes, Laporan
13522116	Naufal Adnan	KNN, Laporan

7. Referensi

- The UNSW-NB15 Dataset | UNSW Research
- UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set) | IEEE Conference Publication | IEEE Xplore
- K-Nearest Neighbor(KNN) Algorithm - GeeksforGeeks
- What Are Naïve Bayes Classifiers? | IBM
- Decision Trees: ID3 Algorithm Explained | Towards Data Science