Tugas Besar 2 IF3170 Inteligensi Artifisial
Implementasi Algoritma Pembelajaran Mesin

Oleh:

13522121 - Jonathan Emmanuel Saragih

13522125 - Satriadhikara Panji Yudhistira

13522128 - Mohammad Andhika Fadillah

13522145 - Farrel Natha Saskoro

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

DAFTAR ISI

Bab 1

Deskripsi Persoalan

Pembelajaran mesin merupakan salah satu cabang dari kecerdasan buatan yang memungkinkan sistem untuk belajar dari data dan membuat prediksi atau keputusan tanpa diprogram secara eksplisit.

Bab 2

Penjelasan

2.1 Implementasi KNN

K-Nearest Neighbors (KNN) adalah salah satu algoritma klasifikasi non-parametrik yang digunakan untuk mengklasifikasikan data berdasarkan kedekatan jarak antara titik data. Algoritma ini bekerja dengan cara membandingkan jarak antara data yang ingin diklasifikasikan dengan data training dan menentukan kelas berdasarkan mayoritas tetangga terdekat.

KNN sering digunakan dalam berbagai aplikasi seperti pengenalan pola, sistem rekomendasi, klasifikasi teks, dan lain-lain. Keunggulan KNN terletak pada kesederhanaan implementasinya dan kemampuannya menangani data dengan distribusi yang tidak teratur.

$$dis\ (x_1, x_2) = \sqrt{\sum_{i=0}^{n} (x_{1i} - x_{2i})^2}$$

Teorema KNN adalah dasar dari algoritma ini, yang dinyatakan sebagai berikut:

- *x1* : Titik data yang ingin diklasifikasikan
- *X2* : Titik data training
- *n* : Jumlah fitur dari data

Deskirpsi K-Nearest Neighbors
def __init__(self, k=5, n_jobs=1, metric='minkowski', p=2, weights='uniform', verbose=True):
Fungsi ini digunakan untuk menginisialisasi atribut: - k: Jumlah tetangga terdekat yang digunakan. - n_jobs: Jumlah core CPU yang digunakan untuk proses paralel. - metric: Metrik jarak yang digunal ('manhattan', 'euclidean', atau 'minkowski'). - p: Parameter power untuk metrik Minkowski (hanya berlaku jika metric='minkowski'). - weights: Metode pembobotan ('uniform' untuk bobot sama atau 'distance' berdasarkan jarak). - verb True, akan mencetak informasi proses saat fitting atau prediksi.
def _compute_distances(self, test):
Fungsi ini digunakan untuk menghitung jarak antara satu data uji (test) dengan seluruh data latih. - test: np.ndarray, satu instance data uji. Output: - distances: np.ndarray, jarak antara data uji dengan setiap data latih.

def _get_nearest_neighbours(self, test):
Fungsi ini digunakan untuk mendapatkan k tetangga terdekat dari satu instance data uji. - test: np.ndarray, satu instance data uji. Output: - indices: np.ndarray, indeks k tetangga terdekat dalam data latih. - weights: np.ndarray, bobot untuk k tetangga (berdasarkan jarak jika weights='distance').
def fit(self, X_train, y_train):
Fungsi ini digunakan untuk menyimpan data latih yang akan digunakan untuk prediksi. - X_train: np.ndarray atau pd.DataFrame, data fitur latih. - y_train: np.ndarray atau pd.Series, label kelas untuk data latih.
def _predict_instance(self, row):
Fungsi ini digunakan untuk memprediksi kelas dari satu instance data uji. - row: np.ndarray, satu instance data uji. Output: - prediction: Prediksi kelas berdasarkan tetangga terdekat.
def predict(self, X_test):
Fungsi ini digunakan untuk memprediksi kelas dari semua instance data uji. - X_test: np.ndarray atau pd.DataFrame, data fitur uji. Output: - results: np.ndarray, hasil prediksi untuk semua data uji.
def save(self, path):
Fungsi ini digunakan untuk menyimpan model KNN ke file menggunakan pickle. - path: str, path file tempat menyimpan model.
@staticmethod def load(path):
Fungsi ini digunakan untuk memuat model KNN dari file yang disimpan sebelumnya. - path: str, path file tempat model disimpan. Output: - model: Objek KNN yang dimuat.

2.2 Implementasi Naive Bayes

Naive Bayes adalah salah satu algoritma klasifikasi sederhana namun sangat efektif yang didasarkan pada *teorema Bayes*. Algoritma ini digunakan dalam berbagai aplikasi seperti klasifikasi teks, analisis sentimen, deteksi spam, dan banyak lagi. *Naive Bayes* sering kali digunakan karena sifatnya yang cepat, ringan, dan mampu memberikan hasil yang baik pada data berskala besar.

Teorema Bayes

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$

Teorema Bayes adalah dasar dari algoritma ini, yang dinyatakan sebagai berikut:

- P(A | B): Probabilitas bahwa data B termasuk dalam kelas A (*posterior probability*).
- P(B | A): Probabilitas data B muncul dalam kelas A (*likelihood*).
- P(A): Probabilitas awal suatu kelas sebelum melihat data (*prior probability*).
- P(B): Probabilitas data B secara keseluruhan (*evidence*).

Deskripsi Kelas Naive Bayes
def __init__(self):
Fungsi ini digunakan untuk menginisialisasi atribut : ● class_probabilities: Menyimpan probabilitas setiap kelas. ● mean: Menyimpan rata-rata (mean) fitur untuk setiap kelas. ● variance: Menyimpan variansi fitur untuk setiap kelas. ● _epsilon: Angka kecil untuk menghindari pembagian dengan nol pada perhitungan variansi.
def _validate_input(self, X: np.ndarray, y: np.ndarray) -> None:
Fungsi ini bertujuan untuk memvalidasi data input sebelum digunakan dalam proses pelatihan atau prediksi. Fungsi mener parameter, yaitu X sebagai matriks fitur dan y sebagai label kelas. Validasi dilakukan dengan memastikan bahwa jumlah sa pada X dan y sama, sehingga tidak terjadi ketidaksesuaian data. Selain itu, fungsi juga memeriksa apakah matriks X meng nilai NaN (Not a Number) atau infinity, yang dapat menyebabkan error dalam perhitungan matematis. Jika ditemukan ketidaksesuaian jumlah sampel atau keberadaan nilai yang tidak valid, fungsi akan menghasilkan error berupa ValueError.
def fit(self, X: np.ndarray, y: np.ndarray) -> None:
Fungsi ini melatih model menggunakan data pelatihan dengan menerima parameter X sebagai matriks fitur berukuran (n_s n_features) dan y sebagai label kelas berukuran (n_samples,). Pertama, fungsi memvalidasi data input menggunakan fungs _validate_input. Selanjutnya, probabilitas setiap kelas dihitung berdasarkan frekuensi kemunculannya di y. Untuk setiap k fungsi menghitung rata-rata (mean) dan variansi (variance) dari setiap fitur secara terpisah. Hasil perhitungan disimpan da atribut internal, yaitu class_probabilities, mean, dan variance, yang akan digunakan untuk prediksi.
@lru_cache(maxsize=1024)e def _gaussian_probability(self, x: float, mean: float, variance: float) -> float:
Fungsi ini digunakan untuk menghitung probabilitas Gaussian dari suatu nilai fitur berdasarkan distribusi normal. Fungsi m tiga parameter: x sebagai nilai fitur, mean sebagai rata-rata fitur, dan variance sebagai variansi fitur. Perhitungan dilakukan menggunakan formula distribusi normal: $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$

Hasil perhitungan berupa probabilitas Gaussian dikembalikan sebagai output. Selain itu, fungsi ini menggunakan dekorato @lru_cache untuk menyimpan hasil perhitungan sebelumnya, sehingga mempercepat proses jika parameter yang sama dig kembali.
def _calculate_class_probability(self, features: np.ndarray, label: int) -> float:
Fungsi ini menghitung log-likelihood probabilitas fitur untuk kelas tertentu dengan menerima features sebagai array fitur d sebagai kelas. Fungsi mengambil rata-rata (mean) dan variansi (variance) fitur untuk kelas tersebut, lalu menghitung log-probabilitas Gaussian setiap fitur menggunakan formula distribusi normal. Log-probabilitas ini dijumlahkan untuk me log-likelihood total, kemudian ditambahkan dengan logaritma probabilitas kelas (log(class_probabilities)). Hasil akhirnya total log-likelihood yang dikembalikan sebagai output.
def predict_single(self, sample: np.ndarray) -> int:
Fungsi ini memprediksi label kelas untuk satu sampel dengan menghitung log-likelihood probabilitas untuk setiap kelas menggunakan _calculate_class_probability. Hasilnya disimpan dalam dictionary class_probs, dan kelas dengan probabilita dipilih menggunakan fungsi max. Fungsi mengembalikan label kelas dengan probabilitas tertinggi.
def predict(self, X: np.ndarray) -> np.ndarray:
Fungsi ini memprediksi label kelas untuk setiap sampel dalam dataset X, yang merupakan array dua dimensi. Fungsi meng menjadi tipe data float64 dan menginisialisasi array predictions untuk menyimpan hasilnya. Dengan menggunakan ThreadPoolExecutor, fungsi melakukan prediksi secara paralel untuk mempercepat proses. Setiap tugas prediksi (menggu predict_single) dikirim ke thread pool dan hasilnya disimpan dalam array predictions. Akhirnya, fungsi mengembalikan ar dimensi yang berisi prediksi label kelas untuk setiap sampel..

2.3 Implementasi ID3

ID3 (Iterative Dichotomiser 3) adalah salah satu algoritma klasifikasi yang digunakan untuk membangun pohon keputusan berdasarkan entropi dan perhitungan *Information Gain*. Algoritma ini bekerja dengan cara memilih atribut terbaik pada setiap tahap pemisahan untuk meminimalkan ketidakteraturan (*entropy*) dalam data. Pohon keputusan kemudian digunakan untuk mengklasifikasikan data baru.

Algoritma ID3 sering digunakan dalam berbagai aplikasi seperti sistem rekomendasi, klasifikasi data, dan pengambilan keputusan. Keunggulan ID3 terletak pada kemampuannya menghasilkan model yang mudah dipahami dan diterapkan.

Rumus utama dalam algoritma ID3 menggunakan entropy dan information gain sebagai dasar pemilihan atribut terbaik.

- Entropi

$$\text{Entropy}(S) = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

Entropi mengukur ketidakteraturan dalam sebuah dataset

- S: Dataset
- p_i: Probabilitas kemunculan kelas i

- Information Gain

$$\text{IG}(S, A) = \text{Entropy}(S) - \sum_{v \in \text{values}(A)} \frac{|S_v|}{|S|} \times \text{Entropy}(S_v)$$

Information Gain digunakan untuk menentukan atribut terbaik dengan mengukur pengurangan entropi setelah pemisahan berdasarkan atribut tersebut

- A: Atribut
- S_v: Subset data S setelah pemisahan berdasarkan A

Deskripsi Kelas ID3
def __init__(self, max_depth=None, min_samples_split=2):
Fungsi ini digunakan untuk menginisialisasi atribut: ● max_depth: Kedalaman maksimum pohon keputusan. ● min_samples_split: Jumlah minimum sampel untuk membagi simpul. ● root: Akar dari pohon keputusan yang akan dibangun.
def _entropy(self, y):
Fungsi ini menghitung nilai entropi untuk kumpulan label y berdasarkan rumus entropi. Menggunakan frekuensi kemunculan setiap label untuk menghitung probabilitas relatif.
def _information_gain(self, y, X_column, threshold):
Fungsi ini menghitung nilai Information Gain dari suatu pembagian data. - Membagi data berdasarkan threshold untuk fitur tertentu. - Menghitung entropi induk dan bobot entropi anak-anak. - Mengembalikan selisih antara entropi induk dan anak-anak sebagai Information Gain.
def _best_split(self, X, y):
Fungsi ini mencari pembagian (split) terbaik berdasarkan semua fitur. - Untuk setiap fitur, mencoba semua nilai unik sebagai threshold. - Menghitung Information Gain untuk setiap kombinasi fitur dan threshold. - Mengembalikan fitur terbaik dan nilai threshold terbaik.
def _build_tree(self, X, y, depth=0):
Fungsi ini membangun pohon keputusan secara rekursif menggunakan algoritma ID3. - Mengecek kondisi berhenti (kedalaman maksimum, jumlah kelas satu, atau data terlalu sedikit). - Membagi data menggunakan fitur dan threshold terbaik. - Membuat node anak untuk sisi kiri dan kanan.
def fit(self, X, y):
Fungsi ini melatih model ID3 dengan memanggil fungsi _build_tree. - X: Matriks fitur input. - y: Label kelas target.

- Mengembalikan model yang telah dilatih.
def _traverse_tree(self, x, node):
Fungsi ini digunakan untuk melakukan traversing (penelusuran) pohon dari akar ke daun. - Mengecek nilai fitur pada node dan memutuskan untuk ke kiri atau kanan. - Mengembalikan prediksi kelas pada node daun.
def predict(self, X):
Fungsi ini melakukan prediksi terhadap data input X. - Untuk setiap sampel dalam X, melakukan traversing pohon keputusan. - Mengembalikan hasil prediksi dalam bentuk array.

2.4 tahap cleaning dan preprocessing

2.4.1 PCA (Principal Component Analysis)

PCA digunakan untuk mengurangi dimensi dataset dengan mempertahankan informasi yang paling signifikan. Selain itu PCA juga digunakan untuk mencegah overfitting dan mempercepat komputasi.

Implementasi:

```
## dimensionality reduction
from sklearn.decomposition import PCA

class PCAImputer(BaseEstimator, TransformerMixin):

    def __init__(self, n_components=2):
        self.n_components = n_components

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

        return self

    def transform(self, X):
        X = PCA(n_components=self.n_components).fit_transform(X)

        return X

    def fit_transform(self, X, y=None):
        X = PCA(n_components=self.n_components).fit_transform(X)

        return X
```

2.4.2 MinMaxScaler

MinMaxScaler adalah metode untuk normalisasi data yang mengubah nilai fitur ke dalam skala tertentu, biasanya antara 0 dan 1. Metode ini menjaga distribusi data tetapi menyelaraskan nilai-nilai ke dalam rentang yang lebih kecil.

Implementasi:

```
## data normalization
from sklearn.preprocessing import MinMaxScaler

class MinMaxScalerImputer(BaseEstimator, TransformerMixin):

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

        return self

    def transform(self, X):
        X = MinMaxScaler().fit_transform(X)

        return X

    def fit_transform(self, X, y=None):
        X = MinMaxScaler().fit_transform(X)

        return X
```

2.4.3 SMOTE (Synthetic Minority Oversampling Technique)

SMOTE adalah teknik yang digunakan untuk menangani **imbalanced data** dalam masalah klasifikasi. Data tidak seimbang terjadi ketika satu kelas memiliki jumlah sampel yang jauh lebih banyak dibandingkan kelas lainnya, sehingga algoritma pembelajaran mesin cenderung bias terhadap kelas mayoritas.

Implementasi:

```
## handling imbalanced data
from imblearn.over_sampling import SMOTE

class SMOTEImputer(BaseEstimator, TransformerMixin):

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

        return self

    def transform(self, X):
        smote = SMOTE()
        X, y = smote.fit_resample(X, y)

        return X, y

    def fit_transform(self, X, y=None):
        smote = SMOTE()
        X, y = smote.fit_resample(X, y)

        return X, y
```

2.4.4 Encode (Encoding)

Encoding adalah proses mengubah data kategorikal menjadi bentuk numerik. Proses ini dilakukan agar data data dieksekusi oleh model.

Implementasi:

```
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.preprocessing import LabelEncoder

class FeatureEncoder(BaseEstimator, TransformerMixin):

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

        return self

    def transform(self, X):
        # Encode the categorical variables here

        cat_cols = X.select_dtypes(include=['object'])

        for col in cat_cols:

            # Get the column values
            col_values = X[col]

            # Find where the column values are not NaN
            not_nan = col_values.notna()

            # Apply the encoder only to the non-NaN values
            X.loc[not_nan, col] = LabelEncoder().fit_transform(col_values[not_nan])

            # Convert the column to integer datatype
            X[col] = X[col].astype('float64')

        return X
```

2.4.5 VarianceThreshold

VarianceThreshold adalah metode feature selection untuk menghapus fitur dengan variansi rendah. Tujuannya adalah untuk menghilangkan fitur yang hampir konstan atau tidak informatif.

Implementasi:

```
from sklearn.feature_selection import VarianceThreshold

class VarianceThresholdSelector(BaseEstimator, TransformerMixin):
    def __init__(self, threshold=0.1):
        self.threshold = threshold
        self.selector = VarianceThreshold(threshold=threshold)

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

    def transform(self, X):
        return self.selector.transform(X)
```

2.4.6 DropColumnsByCorrelation

DCC adalah metode untuk menghapus kolom yang memiliki korelasi tinggi. Hal ini bertujuan untuk mengurangi redundancy.

Implementasi:

```
class DropColumnsByCorrelation(BaseEstimator, TransformerMixin):
    def __init__(self, threshold=0.9):
        self.threshold = threshold
        self.columns_to_drop = None

    def fit(self, X, y=None):
        corr_matrix = pd.DataFrame(X).corr().abs()
        upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))
        self.columns_to_drop = [c for c in upper.columns if any(upper[c] > self.threshold)]
        return self

    def transform(self, X):
        X = pd.DataFrame(X)
        return X.drop(columns=self.columns_to_drop, errors='ignore')
```

2.4.7 Imputer

Imputer digunakan untuk menangani data yang hilang (missing values). Hal ini bertujuan untuk mengisi data yang hilang dengan sebuah nilai agar data dapat diproses.

Implementasi:

```
from miceforest import ImputationKernel
from sklearn.base import BaseEstimator, TransformerMixin

class FeatureImputer(BaseEstimator, TransformerMixin):

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

        return self

    def transform(self, X):
        mice_kernel = ImputationKernel(
        data = X,
        random_state = 2023)

        mice_kernel.mice(2)
        mice_imputation = mice_kernel.complete_data()
        X = mice_imputation

        return X
```

2.4.8 Impute outliers

Impute outliers adalah cara untuk mengganti data outliers dengan nilai. Hal ini bertujuan untuk menangani data yang berada di luar distribusi normal atau rentang yang wajar tanpa menghapusnya. Agar data menjadi tidak bias saat diproses. Dalam implementasinya outliers dijadikan NaN value lalu di-impute.

Implementasi:

```
def replace_outliers_with_nan(df, multiplier=1.5):
    # Calculate IQR
    Q1 = df.quantile(0.25)
    Q3 = df.quantile(0.75)
    IQR = Q3 - Q1

    # Identify outliers
    outliers = (df < (Q1 - multiplier * IQR)) | (df > (Q3 + multiplier * IQR))

    # Replace outliers with NaN
    df_out = df.copy()
    df_out[outliers] = np.nan

    return df_out

class OutlierTransformer(BaseEstimator, TransformerMixin):

    def __init__(self, multiplier=1.5):
        self.multiplier = multiplier

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

    def transform(self, X):
        X_out = X.apply(replace_outliers_with_nan, multiplier=self.multiplier)
        return X_out
```

Bab 3

Pembahasan

2.1 KNN

Menggunakan Library :

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Define the K-Nearest Neighbors (KNN) Model
knn = KNeighborsClassifier()

# Train the model
knn.fit(X_train, y_train)

# Predict on the validation set
knn_pred = knn.predict(X_val)

# Calculate and print the accuracy score
accuracy = accuracy_score(y_val, knn_pred)
print(f"Accuracy of KNN model: {accuracy}")
```

2] ✓ 2.5s

```
Accuracy of KNN model: 0.475124992871129
```

Menggunakan Algoritma From Scratch :

```
from Algorithm.KNearestN import KNN
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score

knn = KNN(k=5)

knn.fit(X_train, y_train)

knn_pred = knn.predict(X_val)

accuracy_score(y_val, knn_pred)
```

```
3]  ✓ 15m 4.5s

Using 1 core for predictions.
100%|██████████| 52603/52603 [15:00<00:00, 58.40it/s]
Prediction completed in 904.52 seconds.

0.4769499838412258
```

Berdasarkan hasil tersebut, dapat disimpulkan bahwa perbedaan akurasi antara kedua pendekatan sangat kecil (sekitar 0.04%). Hal ini menunjukkan bahwa implementasi manual dari algoritma KNN sudah benar dan sesuai dengan hasil yang dihasilkan oleh pustaka standar. Perbedaan kecil ini kemungkinan disebabkan oleh:

1. Presisi Perhitungan:
   Perbedaan akurasi dapat terjadi karena perbedaan presisi bilangan desimal pada perhitungan jarak dan proses pengambilan keputusan (voting).
2. Penanganan Ties (Nilai Jarak Sama):
   Pada beberapa kasus, ketika terdapat dua atau lebih tetangga terdekat dengan jarak yang sama, pustaka mungkin memiliki metode khusus untuk menangani kondisi ini, sedangkan implementasi manual bisa berbeda.
3. Optimasi Library:
   Library yang digunakan umumnya telah dioptimasi untuk menangani perhitungan sehingga lebih efisien, namun hasil akhirnya tetap mendekati atau sama dengan implementasi manual jika dilakukan dengan benar.

Insights :

1. Validasi Implementasi:
   Perbandingan ini menunjukkan bahwa algoritma yang diimplementasikan secara

manual memiliki konsistensi dengan pustaka standar. Hal ini membuktikan pemahaman yang baik terhadap logika KNN dan proses perhitungannya.

2. Evaluasi dengan Metrics Lain:
   Selain akurasi, untuk mendapatkan gambaran lebih jelas mengenai performa model, dapat dipertimbangkan metrics lain seperti Precision, Recall, atau F1-Score, terutama jika data bersifat imbalanced (jumlah kelas tidak seimbang).
3. Peningkatan Kinerja:
   - Melakukan tuning parameter k (jumlah tetangga) untuk melihat pengaruhnya terhadap performa model.
   - Menggunakan feature scaling seperti Min-Max Scaling atau Standardization agar perhitungan jarak menjadi lebih akurat, terutama jika fitur memiliki skala yang berbeda.

2.2 Naive Bayes

Menggunakan Library :

```
from sklearn.naive_bayes import GaussianNB

# Encode the target variable
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
y_val_encoded = label_encoder.transform(y_val)

nba = GaussianNB()
nba.fit(X_train, y_train_encoded)
nba_pred_encoded = nba.predict(X_val)
nba_pred = label_encoder.inverse_transform(nba_pred_encoded)

print(f"Accuracy: {accuracy_score(y_val, nba_pred)}")
✓ 0.1s
Accuracy: 0.6041100317472388
```

Menggunakan Algoritma From Scratch :

```
from Algorithm.NaiveBayes import NBayes

# Encode the target variable
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
y_val_encoded = label_encoder.transform(y_val)

nb = NBayes()
nb.fit(X_train, y_train_encoded)
nb_pred_encoded = nb.predict(X_val)
nb_pred = label_encoder.inverse_transform(nb_pred_encoded)

print(f"Accuracy: {accuracy_score(y_val, nb_pred)}")
✓ 10.2s
Accuracy: 0.6041100317472388
```

1. Perbandingan Waktu Eksekusi

- Library (Sklearn's GaussianNB): Waktu eksekusi hanya 0.1 detik.
- Algoritma From Scratch (NBayes): Waktu eksekusi jauh lebih lama, sekitar 10.25 detik.

  Insight:
  Menggunakan library yang sudah dioptimalkan seperti sklearn jauh lebih cepat dibandingkan implementasi manual algoritma. Hal ini dikarenakan:

  - Library memanfaatkan optimasi tingkat rendah dan pengelolaan memori yang lebih baik.
  - Algoritma manual mungkin kurang efisien dalam komputasi paralel atau memiliki overhead dalam iterasi perhitungan.

2. Akurasi Model

- Akurasi model menggunakan GaussianNB (Sklearn) dan NBayes (implementasi manual) sama, yaitu 0.6041.

  Insight:
  Hasil yang sama menunjukkan bahwa implementasi manual sudah benar dan sesuai dengan teori Naive Bayes. Namun, efisiensi waktu menjadi perbedaan signifikan.

3. Keuntungan Library dibandingkan Implementasi Manual

- Optimisasi: Library seperti Sklearn sudah sangat dioptimalkan untuk performa.
- Kemudahan Penggunaan: Hanya memerlukan beberapa baris kode untuk melatih dan memprediksi.
- Konsistensi: Library sudah diuji secara luas sehingga memiliki keandalan tinggi.

  Insight:
  Meskipun algoritma manual memberikan pemahaman mendalam tentang proses di balik Naive Bayes, menggunakan library lebih disarankan untuk aplikasi real-world yang membutuhkan kecepatan dan efisiensi.

4. Manfaat Implementasi Manual

- Memberikan pemahaman mendalam tentang bagaimana Naive Bayes bekerja, mulai dari perhitungan probabilitas Gaussian hingga prediksi akhir.
- Cocok untuk pembelajaran konsep dasar atau studi akademis.

  Insight:
  Implementasi manual ideal digunakan untuk eksplorasi, debugging, atau pembelajaran, tetapi kurang efisien untuk skala besar atau produksi.

2.3 ID3

Menggunakan Library:

Menggunakan Algoritma From Scratch:

Perbandingan antara implementasi manual ID3 dan penggunaan library ID3 (atau algoritma pohon keputusan serupa dari library seperti Scikit-learn) melibatkan beberapa aspek penting seperti akurasi, waktu komputasi, serta kemudahan penggunaan dan pemeliharaan kode. Berikut adalah penjelasan mendetail mengenai perbedaan keduanya serta insight masing-masing:

1. **Akurasi Model**
   - **Implementasi Manual**: Anda melaporkan bahwa implementasi manual menghasilkan akurasi sebesar **0.6231**
   - **Library**: Implementasi menggunakan library menghasilkan akurasi sebesar **0.4637**

   Insight:

   - **Kualitas Implementasi**: Akurasi yang lebih tinggi pada implementasi manual mungkin menunjukkan bahwa implementasi tersebut telah disesuaikan atau dioptimalkan untuk dataset tertentu. Namun, hal ini juga bisa disebabkan oleh adanya bug atau kesalahan dalam kode library yang menyebabkan performa menurun.
   - **Parameter dan Pengaturan**: Library seperti Scikit-learn menyediakan berbagai parameter dan opsi yang dapat mempengaruhi performa model, seperti max_depth, min_samples_split, criterion, dll. Jika pengaturan ini tidak disesuaikan dengan baik, bisa menyebabkan akurasi yang lebih rendah.
   - **Preprocessing Data**: Perbedaan dalam preprocessing data antara implementasi manual dan library juga dapat mempengaruhi akurasi. Misalnya, penanganan nilai yang hilang, encoding fitur kategorikal, atau skala fitur yang berbeda.

2. **Waktu Komputasi**
   - **Implementasi Manual**: Memerlukan waktu lebih dari **30 menit**.
   - **Library**: Hanya membutuhkan sekitar **0.4 detik**.

   Insight:

- **Optimasi Kinerja**: Library seperti Scikit-learn ditulis sebagian besar dalam bahasa pemrograman yang lebih cepat seperti C/C++ dan dioptimalkan untuk kinerja tinggi. Sedangkan implementasi manual, terutama yang ditulis sepenuhnya dalam Python tanpa optimasi khusus, cenderung jauh lebih lambat karena interpretasi bahasa yang kurang efisien.
- **Vectorization dan Paralelisasi**: Library sering menggunakan operasi vektorisasi dan paralelisasi yang memanfaatkan sepenuhnya kemampuan hardware modern, sehingga sangat mempercepat proses komputasi. Implementasi manual biasanya tidak memanfaatkan teknik-teknik ini secara optimal.
- **Algoritma yang Lebih Efisien**: Library mungkin menggunakan algoritma yang lebih efisien untuk pemilihan split, perhitungan entropi, dan optimasi lainnya yang mengurangi kompleksitas waktu.

Lampiran

Pembagian Tugas :

NIM	Tugas
13522121	KNN, Laporam
13522125	ID3, Laporan
13522128	NB, Laporan
13522145	EDA, Split training set and validation set, data cleaning and preprocessing, compile preprocessing pipeline, submission, laporan