TUGAS BESAR 2 IF3170 INTELIGENSI BUATAN
Semester 5 Tahun 2023/2024
Implementasi Algoritma KNN dan Naive-Bayes

Disusun Oleh:

Angger Ilham Amanullah	13521001
Ditra Rizqa Amadia	13521019
Bernardus Willson	13521021
Raynard Tanadi	13521143

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

2023

BAB I
K-Nearest Neighbors

1.1. Dasar Teori

K-Nearest Neighbor atau KNN adalah sebuah algoritma *machine learning* yang bekerja berdasarkan prinsip bahwa objek yang nilai atribut-atributnya mirip cenderung berada dalam jarak yang dekat satu sama lain. Dengan kata lain, data yang memiliki karakteristik serupa akan cenderung saling bertetangga dalam ruang fitur (*feature space*). Algoritma K-Nearest Neighbor (KNN) juga memiliki karakteristik sebagai algoritma yang bersifat n*on-parametric* dan *lazy learning*.

Salah satu karakteristik utama KNN adalah sifat *non-parametric*-nya. Konsep *non-parametric* dalam KNN menggambarkan sifat algoritma ini yang tidak membuat asumsi tertentu tentang distribusi data yang digunakan. Dalam kata lain, KNN tidak memiliki parameter tertentu atau estimasi parameter yang harus diatur pada modelnya, terlepas dari seberapa banyak data yang digunakan. Ini menjadikan KNN sebagai algoritma yang sangat fleksibel dan mampu menangani berbagai jenis data.

KNN juga dikenal juga sebagai algoritma *lazy learning*. Ini berarti algoritma ini tidak melibatkan fase pelatihan yang signifikan seperti algoritma *machine learning* lainnya. Dalam KNN, hampir tidak ada pembentukan model yang dilakukan menggunakan data pelatihan. Sebaliknya, seluruh data pelatihan digunakan saat menguji atau melakukan klasifikasi data baru. Hal ini membuat proses pelatihan berjalan lebih cepat, tetapi proses pengujian memerlukan lebih banyak waktu dan sumber daya, termasuk penggunaan memori yang lebih besar.

Prinsip dasar algoritma KNN mengasumsikan bahwa objek yang mirip akan berada dalam jarak yang dekat satu sama lain. Dengan kata lain, data yang memiliki karakteristik serupa akan cenderung terletak berdekatan. KNN menggunakan seluruh data yang tersedia dalam pengambilan keputusan. Ketika ada data baru yang perlu diklasifikasikan, algoritma mengukur tingkat kemiripan atau fungsi jarak antara data baru tersebut dengan data yang sudah ada. Data baru kemudian ditempatkan dalam kelas yang paling banyak dimiliki oleh data tetangga terdekatnya.

1.2. Implementasi Algoritma

Berikut adalah implementasi algoritma KNN yang telah kami buat,

```
# KNN Class
class KNN:
```

```
        # constructor
        def __init__(self, k):
            self.k = k
            self.X_train = None
            self.y_train = None

        # calculate euclidean distance between two data points
        def euclidean_distance(self, row1, row2):
            return np.sqrt(np.sum((row1 - row2) ** 2))

        # get nearest data point
        def get_neighbors(self, X_test):
            distances = []

            for i in range(len(self.X_train)):
                dist = self.euclidean_distance(X_test, self.X_train[i])
                distances.append((dist, self.y_train[i]))

            distances.sort(key=lambda tup: tup[0])
            neighbors = distances[:self.k]

            return neighbors

        # assign local data point from data
        def fit(self, X_train, y_train):
            self.X_train = X_train.values
            self.y_train = y_train.values

        # calculate single point prediction value
        def _predict_single(self, x):
            neighbors = self.get_neighbors(x)
            output_values = [neighbor[1] for neighbor in neighbors]
            prediction = max(set(output_values), key=output_values.count)
            return prediction

        # calculate predictions value
        def predict(self, X_test):
            X_test = X_test.values

            # most likely not reachable
            # if X_test.ndim == 1:
```

```
            #     X_test = X_test.reshape(1, -1)

            predictions = [self._predict_single(x) for x in X_test]
            return predictions
```

1.3. Penjelasan Algoritma

Daftar fungsi yang telah kami buat adalah sebagai berikut,

- **euclidean_distance(self, row1, row2)**:
  Menghitung jarak euclidean antara dua nilai data, row1 dan row2. Menggunakan formula euclidean,

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

  untuk mengukur jarak antara dua nilai berbeda. Semakin mirip datanya, nilai *distance*-nya akan semakin kecil. Namun jika datanya semakin tidak mirip, maka nilai *distance*-nya akan semakin besar.

- **get_neighbors(self, X_test)**:
  Mendapatkan *list* K (sesuai nilai K yang telah ditentukan di awal) tetangga terdekat dari data uji 'X_test' dengan menghitung jarak antara 'X_test' dan semua titik dalam 'X_train'. Untuk perhitungan jarak menggunakan fungsi euclidean_distance.

- **fit(self, X_train, y_train)**:
  Melatih model dengan data 'X_train' dan label 'y_train'. Meng-*assign* 'X_train' dan 'y_train' dalam objek kelas KNN untuk digunakan dalam perhitungan *distance* dan prediksi.

- **_predict_single(self, x)**:
  Menghitung nilai prediksi untuk satu titik data x. Mengambil tetangga terdekat dari x menggunakan get_neighbors, kemudian melakukan prediksi dengan memilih nilai yang paling sering muncul dari tetangga terdekat tersebut.

- **predict(self, X_test)**:
  Mengembalikan daftar prediksi untuk setiap titik data dalam 'X_test' dengan memanggil _predict_single untuk setiap titik dalam 'X_test' dan mengumpulkan hasil prediksi ke dalam suatu *list*.

Berikut adalah langkah-langkah / alur dari algoritma yang telah dibuat,

1. ***Initiating***

   Pertama-tama kita perlu menginisiasi KNN dengan nilai K yang ingin diuji. Nilai K merupakan banyaknya *neighbor* terdekat yang ingin ditinjau.

2. **Iterasi Melalui Setiap Data Uji**

   Algoritma akan mengiterasi melalui setiap data poin dalam data uji (X_test). Untuk setiap data poin dalam ‘X_test’, algoritma akan memanggil metode _predict_single.

3. **Memanggil _predict_single**

   Metode _predict_single menerima satu data poin pada setiap pemanggilannya. Kemudian menggunakan metode get_neighbors untuk menemukan K tetangga terdekat dari data poin uji tersebut di dalam set *train*. Lalu memperoleh *distance* dari masing-masing K tetangga.

4. **Prediksi Nilai**

   Setelah mendapatkan label dari tetangga terdekat, algoritma menghitung nilai prediksi berdasarkan nilai mayoritas dari label-label ini. Label yang paling umum dari tetangga terdekat dipilih sebagai prediksi untuk data poin uji saat ini. Prediksi ini kemudian disimpan dalam sebuah list atau struktur data yang menyimpan prediksi untuk setiap data poin dalam data uji.

BAB II
Naive-Bayes

2.1. Dasar Teori

Naive Bayes Classifier atau NB adalah sekumpulan algoritma yang didasarkan pada Teorema Bayes. Dengan kata lain, algoritma ini bukan algoritma tunggal melainkan satu grup algoritma dimana masing-masing memiliki prinsip kerja yang mirip. Pengertian dan Contoh Algoritma Naive Bayes Classifier Algoritma ini bekerja berdasarkan prinsip probabilitas bersyarat, seperti yang diberikan oleh Teorema Bayes. Teorema Bayes menemukan probabilitas atau kemungkinan suatu peristiwa akan terjadi dengan memberikan probabilitas peristiwa lain yang telah terjadi. Dalam istilah yang lebih sederhana, Teorema Bayes adalah metode untuk menemukan probabilitas ketika kita mengetahui probabilitas tertentu lainnya. Teorema Bayes dinyatakan secara matematis dalam persamaan berikut,

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

dimana P(B) != 0

Pada dasarnya, kita mencoba mencari peluang kejadian A, apabila kejadian B bernilai benar. Kejadian B juga disebut sebagai bukti. P(A) adalah apriori dari A (probabilitas sebelumnya, yaitu probabilitas peristiwa sebelum bukti terlihat). Bukti adalah nilai atribut dari instance yang tidak diketahui (peristiwa B). P(A|B) adalah probabilitas posteriori dari B, yaitu probabilitas kejadian setelah bukti terlihat. Ciri utama dari algoritma Naive Bayes Classifier adalah adanya asumsi yg sangat kuat (naif) akan independensi dari masing-masing kondisi atau kejadian.

Namun pada kasus ini, kami menggunakan Gaussian Naive Bayes yang merupakan modifikasi dari Naive Bayes biasa. Naive Bayes biasa hanya cocok untuk data dengan fitur kategori atau data yang bisa diasumsikan sebagai distribusi diskrit seperti teks atau klasifikasi biner. Di sisi lain, Gaussian Naive Bayes lebih cocok untuk data dengan fitur-fitur numerik yang diasumsikan terdistribusi normal atau mendekati distribusi normal.

2.2. Implementasi Algoritma

Berikut adalah implementasi algoritma NB yang telah kami buat,

```
# Naive Bayes Class
class NaiveBayes:

    # constructor
```

```
    def __init__(self):
        self.class_probs = None
        self.feature_stats = None

    # calculate each class probability
    def calculate_class_probs(self, y_train):
        unique_classes, class_counts = np.unique(y_train, return_counts=True)
        total_samples = len(y_train)
        class_probs = dict(zip(unique_classes, class_counts / total_samples))
        return class_probs

    # calculate statistics of each features
    def calculate_feature_stats(self, X_train, y_train):
        unique_classes = np.unique(y_train)
        feature_stats = {}

        for class_label in unique_classes:
            class_mask = (y_train == class_label)
            class_samples = X_train[class_mask]
            feature_means = np.mean(class_samples, axis=0)
            feature_stds = np.std(class_samples, axis=0)
            feature_stats[class_label] = {'mean': feature_means, 'std':
feature_stds}

        return feature_stats

    # assign local data point from data
    def fit(self, X_train, y_train):
        self.class_probs = self.calculate_class_probs(y_train)
        self.feature_stats = self.calculate_feature_stats(X_train, y_train)

    # calculate likelihood of single feature
    def calculate_likelihood_gaussian(self, feat_val, mean, std):
        p_x_given_y = (1 / (np.sqrt(2 * np.pi) * std)) * np.exp(-((feat_val -
mean)**2 / (2 * std**2)))
        return p_x_given_y

    # calculate single class prediction value
    def _predict_single(self, x):
        class_probs = {}
```

```
        for class_label, class_prob in self.class_probs.items():
            feature_stats = self.feature_stats[class_label]['mean'],
self.feature_stats[class_label]['std']

            # Assuming features are continuous
            likelihood = np.prod(
                [self.calculate_likelihood_gaussian(x[i], feature_stats[0][i],
feature_stats[1][i]) for i in range(len(x))]
            )

            class_probs[class_label] = likelihood * class_prob

        prediction = max(class_probs, key=class_probs.get)
        return prediction

    # calculate prediction value
    def predict(self, X_test):
        # if X_test.ndim == 1:
        #     X_test = X_test.reshape(1, -1)

        predictions = [self._predict_single(x) for x in X_test]
        return predictions
```

2.3. Implementasi Algoritma

Daftar fungsi yang telah kami buat adalah sebagai berikut,

- **calculate_class_probs(self, y_train):**

  Menghitung probabilitas masing-masing kelas yang ada dalam y_train. Menggunakan distribusi kelas dan jumlah kemunculan kelas untuk menghitung probabilitas kelas.

- **calculate_feature_stats(self, X_train, y_train):**

  Menghitung mean dan standar deviasi untuk setiap class_label

- **fit(self, X_train, y_train):**

  Melatih model dengan data ‘X_train’ dan label ‘y_train’. Meng-*assign* ‘X_train’ dan ‘y_train’ dalam objek kelas NB untuk digunakan dalam apa cik.

- **calculate_likelihood_gaussian(self, feat_val, mean, std):**

  Menghitung dan mengembalikan nilai *likelihood Gaussian,* yaitu menghitung probabilitas kemunculan suatu nilai tertentu.

- **_predict_single(self, x)**:
  Menghitung nilai prediksi untuk satu titik data x. Akan mengembalikan ‘prediction’ yaitu variabel yang mengambil nilai prediksi dengan probabilitas terbesar.

- **predict(self, X_test):**
  Mengembalikan daftar prediksi untuk setiap titik data dalam ‘X_test’ dengan memanggil _predict_single untuk setiap titik dalam ‘X_test’ dan mengumpulkan hasil prediksi ke dalam suatu *list*.

Berikut adalah langkah-langkah / alur dari algoritma yang telah dibuat,

1. ***Initiating***
   Pertama tama kita menginisiasinya dengan memanggil NaiveBayes().

2. **Iterasi Melalui Setiap Data Uji**
   Algoritma akan mengiterasi melalui setiap data poin dalam data uji (X_test). Untuk setiap data poin dalam ‘X_test’, algoritma akan memanggil metode _predict_single.

3. **Memanggil _predict_single**
   Metode _predict_single menerima satu data poin pada setiap pemanggilnya. Kemudian melakukan *forloop* untuk menghitung probabilitas setiap nilai prediksi dengan memanfaatkan *likelihood gaussian* yang akan disimpan di ‘class_probs’.

4. **Prediksi Nilai**
   Pada metode _predict_single, terdapat variabel ‘prediction’ yang akan menyimpan nilai prediksi dengan probabilitas terbesar lalu dikembalikan ke metode predict. Pada metode predict terdapat *list* ‘predictions’ yang menyimpan semua hasil nilai prediksi.

BAB III

Perbandingan Scratch dengan Library

3.1. KNN Scratch

```
# predicting with KNN for train data

y_pred_train_KNN = knn_model.predict(X_train)
```

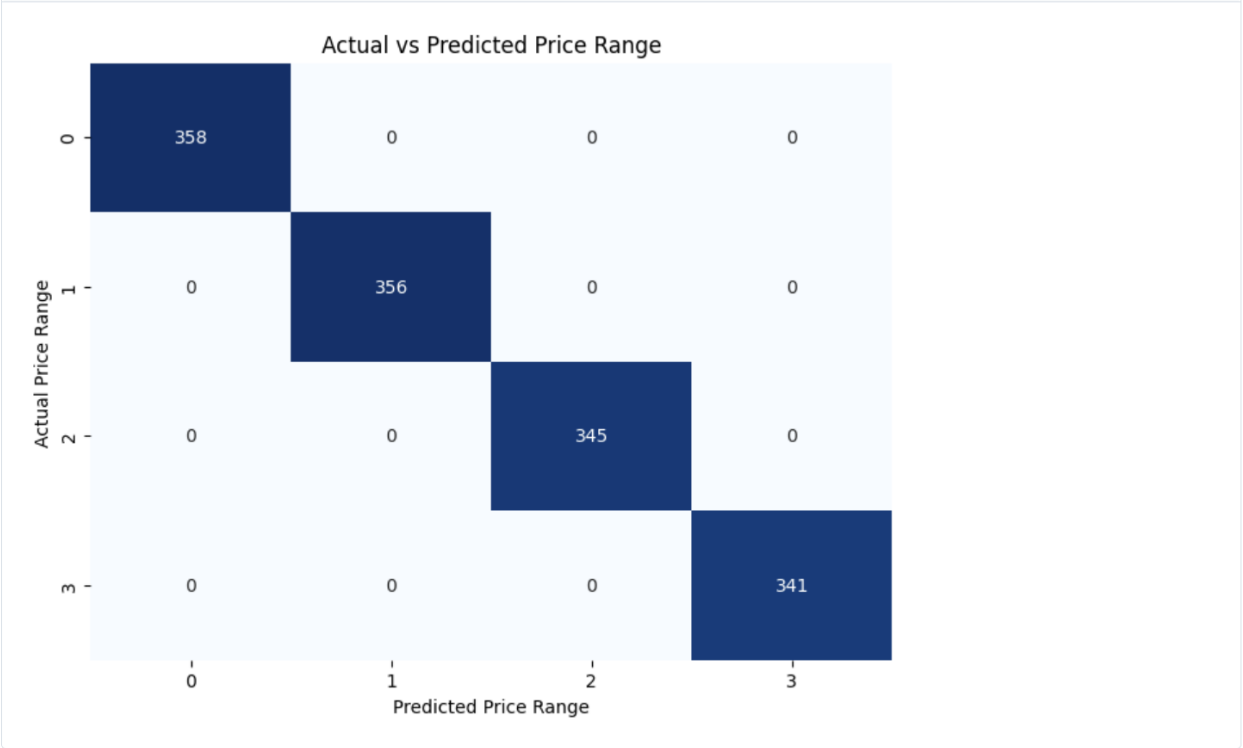
```
# evaluation for KNN training (1/2)

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

accuracy = accuracy_score(y_train, y_pred_train_KNN)
precision = precision_score(y_train, y_pred_train_KNN, average='weighted')
recall = recall_score(y_train, y_pred_train_KNN, average='weighted')
f1 = f1_score(y_train, y_pred_train_KNN, average='weighted')

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)

```

```
Accuracy: 1.0
Precision: 1.0
Recall: 1.0
F1 Score: 1.0
```

```
# predicting with KNN for validation data

y_pred_test_KNN = knn_model.predict(X_test)
```

```
# evaluation for KNN validation (1/2)

accuracy = accuracy_score(y_test, y_pred_test_KNN)
precision = precision_score(y_test, y_pred_test_KNN, average='weighted')
recall = recall_score(y_test, y_pred_test_KNN, average='weighted')
f1 = f1_score(y_test, y_pred_test_KNN, average='weighted')

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
```

```
Accuracy: 0.9183333333333333
Precision: 0.9179218265792964
Recall: 0.9183333333333333
F1 Score: 0.9179903781969241
```



3.2. KNN Library

```
# predicting with KNN scikit for training data

y_pred_train_KNN_sklearn = knn_model_sklearn.predict(X_train)
```

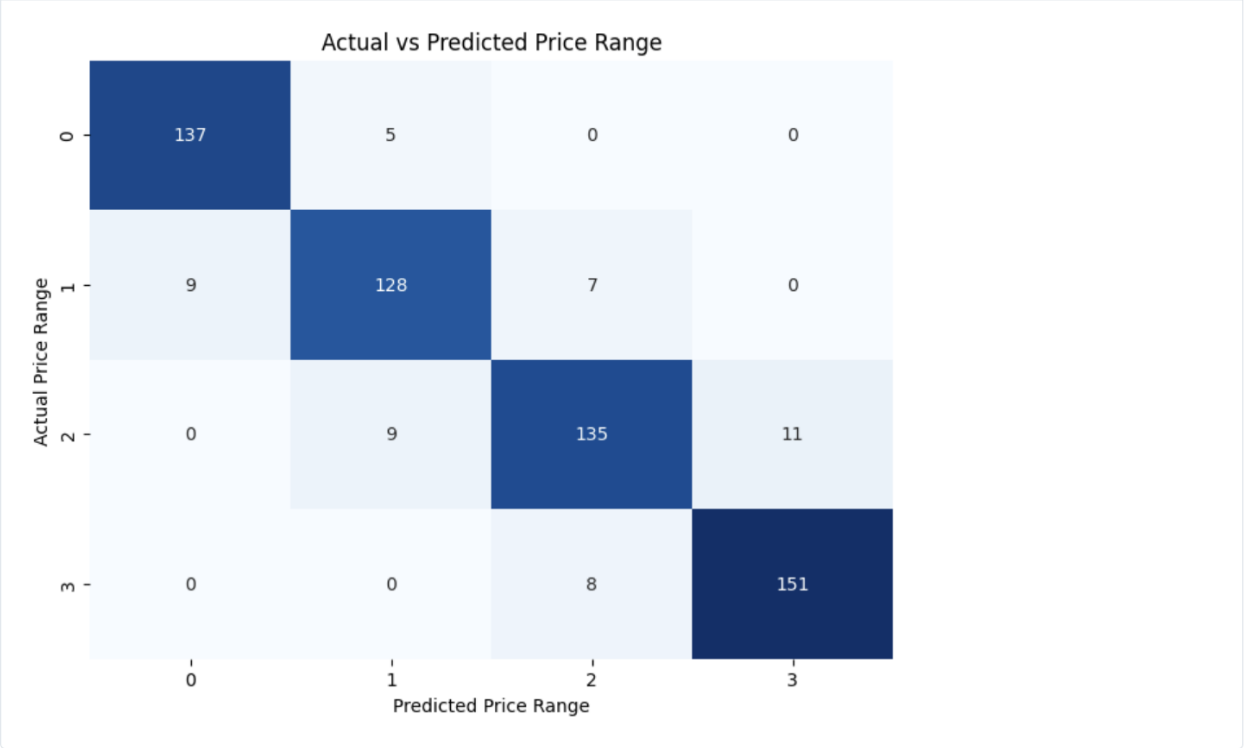
```
# evaluation for KNN scikit training (1/2)

accuracy = accuracy_score(y_train, y_pred_train_KNN_sklearn)
precision = precision_score(y_train, y_pred_train_KNN_sklearn, average='weighted')
recall = recall_score(y_train, y_pred_train_KNN_sklearn, average='weighted')
f1 = f1_score(y_train, y_pred_train_KNN_sklearn, average='weighted')

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
```

```
Accuracy: 1.0
Precision: 1.0
Recall: 1.0
F1 Score: 1.0
```

```
# predicting with KNN scikit for validation data

y_pred_test_KNN_sklearn = knn_model_sklearn.predict(X_test)
```

```
# evaluation for KNN scikit validation (1/2)

accuracy = accuracy_score(y_test, y_pred_test_KNN_sklearn)
precision = precision_score(y_test, y_pred_test_KNN_sklearn, average='weighted')
recall = recall_score(y_test, y_pred_test_KNN_sklearn, average='weighted')
f1 = f1_score(y_test, y_pred_test_KNN_sklearn, average='weighted')

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
```

```
Accuracy: 0.9183333333333333
Precision: 0.9179218265792964
Recall: 0.9183333333333333
F1 Score: 0.9179903781969241
```



3.3. Analisis KNN

Hasil dari algoritma KNN yang kami buat (*scratch*) dengan menggunakan *library* scikit-learn tampaknya konsisten dan sebanding dalam hal matrik evaluasi performa seperti *metrics precision*, *recall*, atau akurasi dan skor F1. Terlihat dari pengujian dengan menggunakan data *train* dan data validasi. Hal

ini menunjukkan bahwa algoritma KNN yang telah kami buat sudah benar dan sudah serupa dengan algoritma KNN dari *library*.

3.4. Naive-Bayes Scratch

```python
# predicting with NB for train data

# get encoded x_train as binary
y_train_encoded = pd.get_dummies(X_train)

# ensure all values are numeric
y_train_numeric = y_train_encoded.astype(float)

# use y_train_numeric for NB prediction
y_pred_train_NB = nb_model.predict(y_train_numeric.values)
```

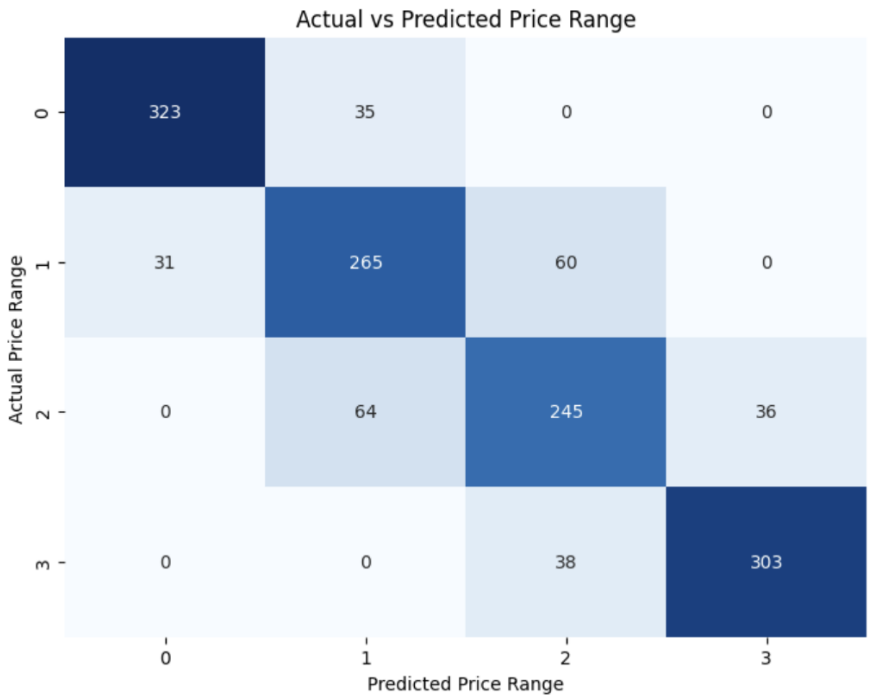
```python
# evaluation for NB training (1/2)

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

accuracy = accuracy_score(y_train, y_pred_train_NB)
precision = precision_score(y_train, y_pred_train_NB, average='weighted')
recall = recall_score(y_train, y_pred_train_NB, average='weighted')
f1 = f1_score(y_train, y_pred_train_NB, average='weighted')

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
```

```
Accuracy: 0.8114285714285714
Precision: 0.8121726598128494
Recall: 0.8114285714285714
F1 Score: 0.8117668200500303
```

```python
# predicting with KNN for validation data

# get encoded x_test as binary
y_test_encoded = pd.get_dummies(X_test)

# ensure all values are numeric
y_test_numeric = y_test_encoded.astype(float)

# use y_test_numeric for prediction
y_pred_test_NB = nb_model.predict(y_test_numeric.values)
```

```python
# evaluation for NB validation (1/2)

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

accuracy = accuracy_score(y_test, y_pred_test_NB)
precision = precision_score(y_test, y_pred_test_NB, average='weighted')
recall = recall_score(y_test, y_pred_test_NB, average='weighted')
f1 = f1_score(y_test, y_pred_test_NB, average='weighted')

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
```

```
Accuracy: 0.79
Precision: 0.7914451365746124
Recall: 0.79
F1 Score: 0.7903706433194465
```



3.5. Naive-Bayes Library

```
# predicting with NB scikit for training data

y_pred_train_NB_sklearn = nb_model_sklearn.predict(X_train)
```

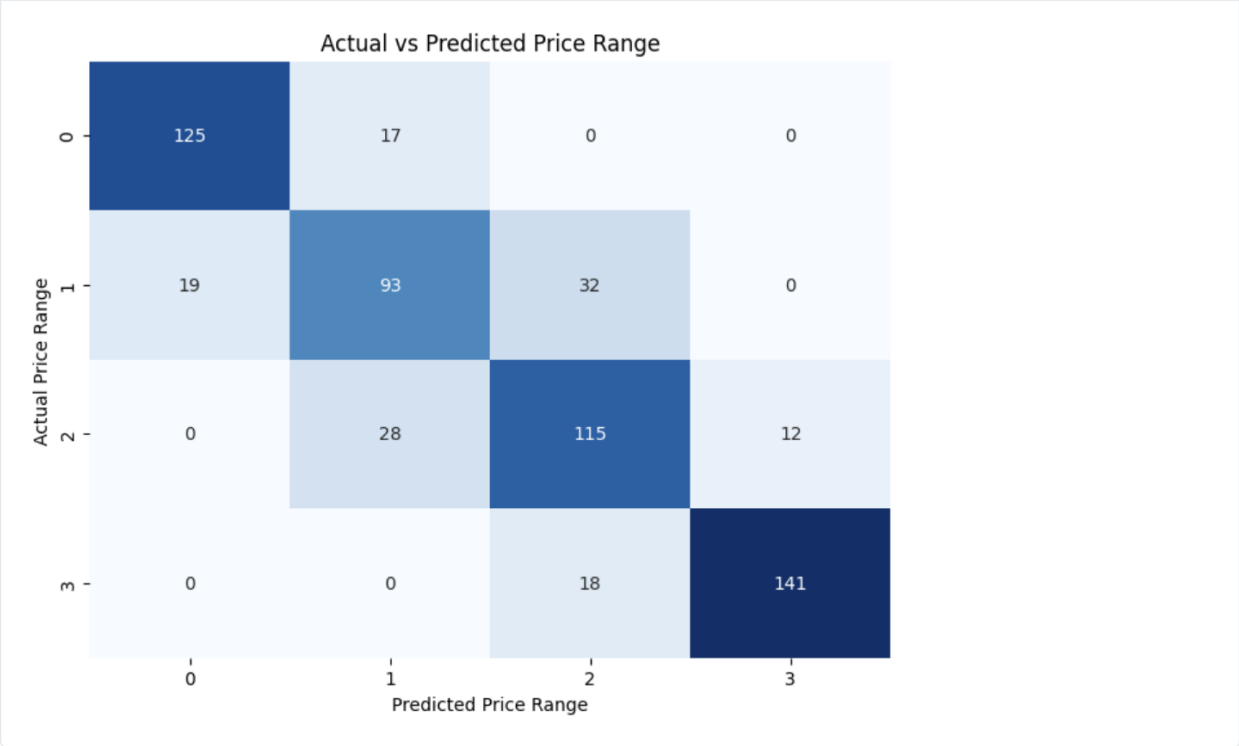
```
# evaluation for NB scikit training (1/2)

accuracy = accuracy_score(y_train, y_pred_train_NB_sklearn)
precision = precision_score(y_train, y_pred_train_NB_sklearn, average='weighted')
recall = recall_score(y_train, y_pred_train_NB_sklearn, average='weighted')
f1 = f1_score(y_train, y_pred_train_NB_sklearn, average='weighted')

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
```

```
Accuracy: 0.8114285714285714
Precision: 0.8121726598128494
Recall: 0.8114285714285714
F1 Score: 0.8117668200500303
```

```
# predicting with NB scikit for validation data

y_pred_test_NB_sklearn = nb_model_sklearn.predict(X_test)
```

```
# evaluation for NB scikit validation (1/2)

accuracy = accuracy_score(y_test, y_pred_test_NB_sklearn)
precision = precision_score(y_test, y_pred_test_NB_sklearn, average='weighted')
recall = recall_score(y_test, y_pred_test_NB_sklearn, average='weighted')
f1 = f1_score(y_test, y_pred_test_NB_sklearn, average='weighted')

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
```

```
Accuracy: 0.79
Precision: 0.7914451365746124
Recall: 0.79
F1 Score: 0.7903706433194465
```



3.6. Analisis Naive-Bayes

Hasil dari algoritma Naive-Bayes yang kami buat (*scratch*) dengan menggunakan *library* scikit-learn tampaknya konsisten dan sebanding dalam hal matrik evaluasi performa seperti *metrics precision*, *recall*, atau akurasi dan skor F1. Terlihat dari pengujian dengan menggunakan data *train* dan data validasi. Hal ini menunjukkan bahwa algoritma Naive-Bayes yang telah kami buat sudah benar dan sudah serupa dengan algoritma Naive-Bayes dari *library*.

BAB IV
Preprocessing

4.1. Data Cleaning

Pembersihan data atau *data cleaning* adalah proses mengidentifikasi dan mengoreksi kesalahan, konsistensi, dan akurasi data. Proses ini dilakukan dengan tujuan meningkatkan kualitas data. Kualitas data berpengaruh besar pada kinerja model. Terdapat beberapa teknik *data cleansing* yang kami gunakan, yaitu sebagai berikut.

- **Duplicate rows**

  Baris data duplikat atau duplicate rows adalah baris data yang memiliki nilai yang sama untuk setiap atributnya dengan baris lain. Hal tersebut menciptakan redundansi sehingga perlu dihilangkan. Namun dalam kasus tugas ini, tidak ditemukan data yang duplikat sehingga tidak perlu untuk melakukan penghapusan maupun penggantian nilai untuk data yang duplikat.

- **Invalid data**

  Invalid data adalah data yang nilainya tidak sesuai dengan batasan pada atributnya. Data invalid tersebut dapat diganti dengan median dari set data pada atribut yang sama. Terdapat beberapa data invalid pada test data yang kami gunakan. Contohnya adalah ‘px_height’ yang menggambarkan tinggi resolusi piksel pada layar ponsel dan ‘sc_w’ yang menggambarkan lebar layar ponsel dalam sentimeter (cm), dalam dunia nyata kedua atribut tersebut tidak mungkin bernilai 0. Maka dari itu, semua data yang bernilai 0 pada atribut tersebut diganti dengan median dari keseluruhan nilai masing-masing ‘px_height’ dan ‘sc_w’.

- **Missing value**

  Missing value adalah data yang nilainya tidak ada. Namun, pada test data yang kami gunakan tidak terdapat missing value sehingga tidak perlu untuk melakukan penghapusan maupun penggantian nilai untuk data yang hilang.

- **Outlier**

  Outlier adalah suatu data yang memiliki jarak abnormal dengan data lainnya. Indikator yang dapat digunakan yaitu apabila data tersebut melebihi batas atas outlier atau kurang dari batas bawah outlier. Batas-batas tersebut dihitung menggunakan persamaan

$$lower\_limit = Q1 - iqr\_threshold \times IQR,$$

$$upper\_limit = Q3 + iqr\_threshold \times IQR,$$

di mana Q1 adalah kuartil 1, Q3 adalah kuartil 3, IQR adalah jangkauan interkuartil, dan iqr_threshold adalah ambang IQR yaitu 1,5. Pada kasus ini kami mencari pencilan-pencilan dari data *train*, data *validation*, dan data *test*. Namun kami memutuskan untuk tidak mengganti nilai pencilan-pencilan tersebut karena korelasinya dari atribut-atribut yang terkena pencilan memiliki korelasi yang tinggi.

4.2. Preprocessing lainnya

Kami melakukan beberapa langkah-langkah yang menurut kami krusial untuk meningkatkan akurasi pada data *test*:

- Menggabungkan data *train* dengan data *validation* untuk *training* data saat akan memprediksi data test untuk submisi di akhir. Hal ini dilakukan karena menurut kami semakin banyak data yang dilatih, akurasi untuk memprediksi suatu data akan lebih meningkat lagi.
- Kami juga melakukan *drop* atribut-atribut yang menurut kami dapat mengganggu pemrosesan, yaitu atribut-atribut yang memiliki tingkat korelasi rendah.

```
# drop columns with bad correlation value

df_train = df_train.drop(columns=['blue', 'wifi', 'three_g', 'four_g',
'dual_sim', 'touch_screen', 'n_cores', 'pc', 'fc', 'talk_time', 'sc_w',
'm_dep'])
```