Systemy Sztucznej Inteligencji

dokumentacja projektu systemu przewidywania pogody

Hubert Bojda, Dawid Gala, grupa 4/7

17 czerwca 2024

Opis programu

System sztucznej inteligencji który klasyfikuje za pomocą klasyfikatora KNN dane pobrane z pogodowego API. Dane są odpowiednio przygotowane poprzez zredukowanie ilości klas abstrakcji do najliczniejszych czyli 'Foggy', 'Overcast', 'Clear'. Zdecydowaliśmy się na takie klasy, ponieważ są one dominujące w naszej bazie danych. Następnie dane zostają przetasowane oraz podzielone na zbiór danych testowych i treningowych. W następnym kroku dane są przetwarzane przez klasyfikator.

Instrukcja obsługi

Nasz program należy uruchomić z wykorzysatniem środowiska Jupyter. Nasz projekt został utworzony w programie PyCharm, który zawiera wbudowany serwer Jupyter.

Opis działania

Klasyfikator KNN, czyli algorytm k najbliższych sąsiadów, służy do klasyfikacji i prognozowania wartości na podstawie zmiennej określonej w kolumnie decyzyjnej w bazie danych. Algorytm porównuje wartości w kolumnach objaśniających dane zjawisko z wartościami zmiennych, które są zawarte w zbiorze uczącym. Zawiera on informacje o k najbliższych obserwacjach ze zbioru uczącego. Ważnym aspektem przy tworzeniu klasyfikatora jest dobór odpowiedniej metryki obliczającej dystans między obserwacjami zbioru uczącego a zbioru treningowego. Najpopularniejsze metryki to: euklidesowa, mińkowskiego lub manhattan. Dzięki kolejnym iteracjom podział danych jest poprawiany względem zadanej metryki. Algorytm przenosi dane pomiędzy klasami tak, aby wariancja wewnątrz każdej klasy była jak najmniejsza.

Wzory

1. **Obliczanie odległości euklidesowej**:

$$d(x_i, x_j) = \sqrt{\sum_{k=1}^{n} (x_{i,k} - x_{j,k})^2} \tag{1}$$

gdzie:

- x_i i x_j to dwa punkty w przestrzeni n-wymiarowej,
- $x_{i,k}$ i $x_{j,k}$ to k-te współrzędne tych punktów.

2. **Wybór k najbliższych sąsiadów**:

Po obliczeniu odległości euklidesowych dla wszystkich punktów w zbiorze treningowym, wybieramy k najbliższych sąsiadów, czyli tych z najmniejszymi wartościami odległości.

3. **Głosowanie większościowe**:

Klasa przypisywana nowemu punktowi jest najczęściej występującą klasą wśród k najbliższych sąsiadów. Głosowanie większościowe można opisać wzorem:

$$y = \arg\max_{c \in C} \sum_{i=1}^{k} I(y_i = c) \tag{2}$$

gdzie:

- y to przewidywana klasa dla nowego punktu,
- C to zbiór wszystkich możliwych klas,
- I to funkcja wskaźnikowa, która przyjmuje wartość 1, gdy $y_i = c$, w przeciwnym razie 0,
- y_i to klasa przypisana do i-tego najbliższego sąsiada.

Algorytm

Poniżej przedstawiony jest algorytm dzielenia zbioru danych na zbiór danych testowych i treningowych:

Algorithm 1: Pseudokod funkcji my_train_test_split

Require: X, y, *test_size* (domyślnie 0.3), *random_state* (domyślnie *None*)
1: **if** *random_state* nie jest *None* **then**
2: Ustaw ziarno losowości na *random_state*
3: **end if**
4: *test_size* = **int**(*test_size* * **len**(X))
5: *shuffled_indices* = permutacja indeksów X
6: *indices_train* = *shuffled_indices*[*test_size*:]
7: *indices_test* = *shuffled_indices*[:*test_size*]
8: $X\_train = X$.iloc[*indices_train*]
9: $X\_test = X$.iloc[*indices_test*]
10: $y\_train = y$.iloc[*indices_train*]
11: $y\_test = y$.iloc[*indices_test*]
12: **return** X_train, X_test, y_train, y_test

Poniżej przedstawione są algorytmy klasyfikatora KNN:

Algorithm 2: Opis algorytmu KNN

Require: $X\_train, y\_train, X\_test, k\_neighbors$
1: $predictions \leftarrow []$
2: **for** x **in** każdy punkt w X_test **do**
3: Oblicz odległości między punktem x a danymi treningowymi X_train.
4: Posortuj indeksy odległości w celu znalezienia k najbliższych sąsiadów.
5: Wybierz etykiety odpowiadające indeksom k najbliższych sąsiadów.
6: Znajdź najczęściej występującą etykietę (głosowanie większościowe).
7: Dodaj przewidywaną etykietę do listy przewidywań.
8: **end for**
9: **return** $predictions$

Algorithm 3: Funkcja obliczająca dokładność

Require: $X\_train, y\_train, X\_test, y\_test, k\_neighbors$
1: $predictions \leftarrow$ wywołaj funkcję predict$(X_train, y_train, X_test, k_neighbors)$
2: Oblicz dokładność przewidywań poprzez porównanie przewidywanych etykiet z rzeczywistymi etykietami y_test.
3: **return** dokładność

Zbiór danych

Zbiór danych weatherHistory.csv pochodzi ze stony Kaggle.com.

Plik posiada 96453 rekordów i 12 kolumn. Jej schemat jest przedstawiony poniżej.

Tabela 1: Dane pogodowe z pliku CSV

$FormattedDate$	$Summary$	$PrecipType$	$Temperature(C)$
2006-04-18 07:00:00.000 +0200	Clear	rain	8.6889
2006-04-12 04:00:00.000 +0200	Overcast	rain	6.6222
2006-04-11 19:00:00.000 +0200	Foggy	rain	8.8

Tabela 2: Dane pogodowe z pliku CSV

$ApparentTemperature(C)$	$Humidity$	$WindSpeed(km/h)$	$WindBearing(degrees)$
8.6889	0.93	1.4329	290.0
2.6167	0.93	25.0355	0.0
5.2944	0.99	26.5006	339.0

Tabela 3: Dane pogodowe z pliku CSV

$Visibility(km)$	$Pressure(millibars)$	$DailySummary$
5.8443	1012.96	Partly cloudy until night.
6.118	1003.68	Foggy overnight and breezy in the morning.
2.6565	1004.99	Foggy in the evening.

Przygotowanie danych

Przed rozpoczęciem tasowania danych i ich normalizacji trzeba było odpowiednio dopasować zbiór danych do klasyfikatora. W pierwszej kolejności zostały usunięte kolumny z wartościami typu String, tego typu kolumn nie da sie przetworzyć w sposób matematyczny. Jedynie została kolumna *Summary*, która jest naszą kolumną decyzyjną

Listing 1: Usuwanie kolumn

```
del df['Formatted Date']
del df['Apparent Temperature (C)']
del df['Precip Type']
del df['Loud Cover']
del df['Daily Summary']
del df['Humidity']
del df['Wind Bearing (degrees)']
```

Aby zredukować ilość klas abstrakcji zamienilismy wszystkie rekordy z podsumowaniem wskazującym na "overcast, clear, foggy", jednak z lekko innym opisem. Dzięki temu udało się zredukować dane do trzech najbardziej według nas znaczących klas abstrakcji.

Usuwanie rekordów w których występują opisy które nie pasują do klas abstrakcji.

Listing 2: Usuwanie rekordów

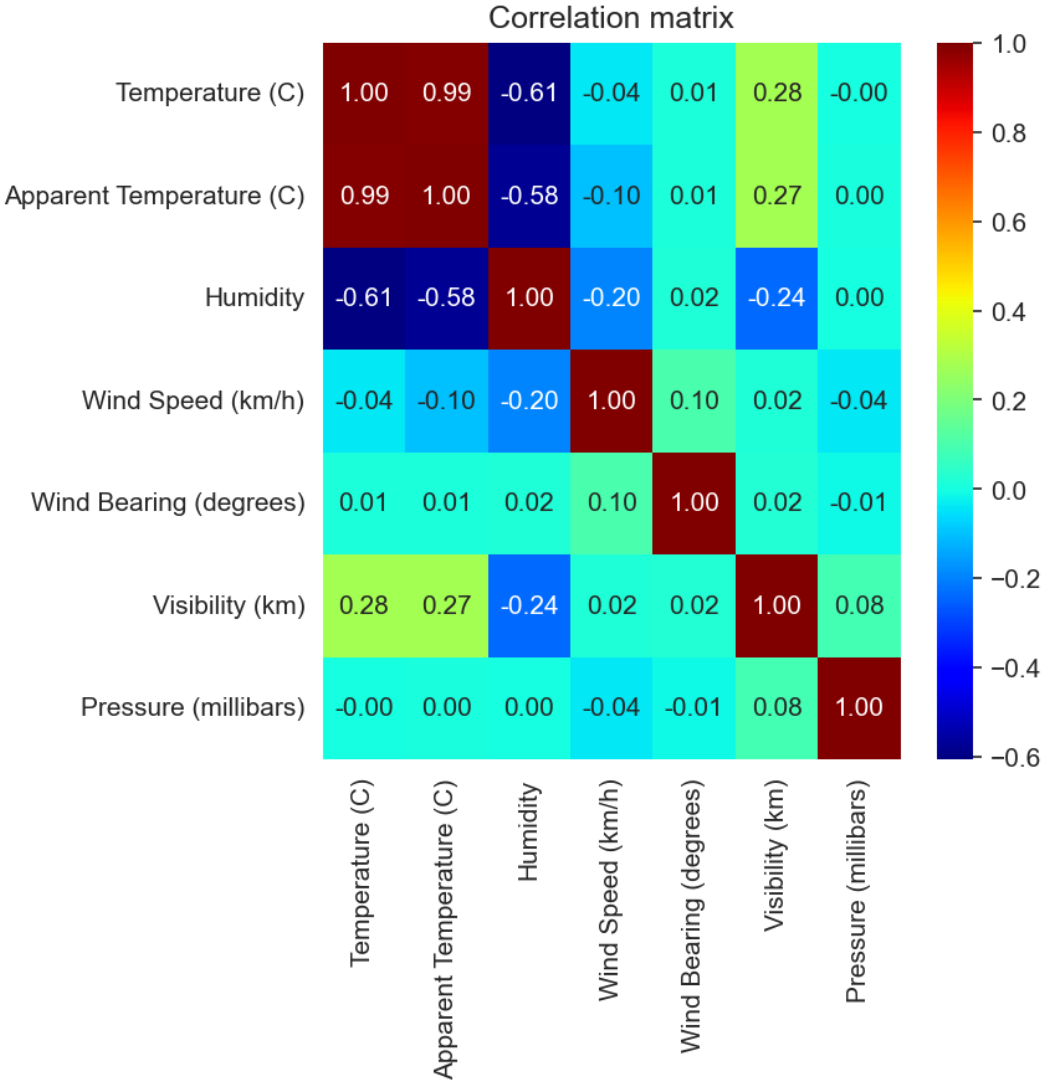
```
df = df.drop(df[df['Summary'] == 'Breezy and Mostly Cloudy'].index)
df = df.drop(df[df['Summary'] == 'Breezy and Partly Cloudy'].index)
df = df.drop(df[df['Summary'] == 'Humid and Mostly Cloudy'].index)
df = df.drop(df[df['Summary'] == 'Breezy and Overcast'].index)
df = df.drop(df[df['Summary'] == 'Humid and Partly Cloudy'].index)
df = df.drop(df[df['Summary'] == 'Windy and Foggy'].index)
df = df.drop(df[df['Summary'] == 'Windy and Overcast'].index)
```

Reszta usuniętych rekordów znajduje się w sekcji z pełnym kodem.

Dzięki tym zmianom, w naszej bazie zostały najważniejsze dla nas dane, a sama baza skurczyła się do 34635 wierszy, co znacznie skróci nasze czasy obliczeń.

Macierz korelacji

Macierz korelacji - jest to macierz która reprezentuje korelacje pomiędzy danymi w naszym zbiorze danych. Wartości w każdej komórce pochodzą z przedziału <-1,1>. Jeśli wartości są bliskie 1 to oznacza że wartości w kolumnach są silnie skorelowane dodatnio np. gdy jedna zmienna wzrasta to druga też. Natomiast gdy wartości są bliskie -1 to oznacza że wartości są silnie skorelowane ujemnie np. gdy jedna zmienna maleje to druga rośnie. Gdy wartość jest bliska 0 to zmienne nie są w ogóle skorelowane.



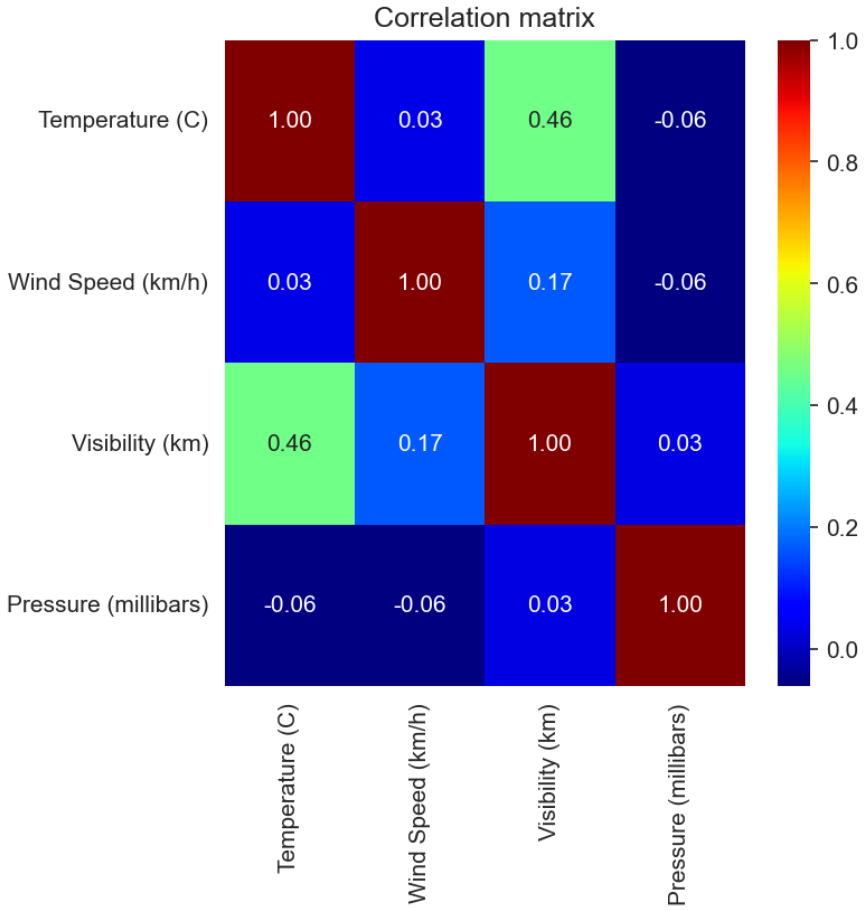
Rysunek 1: Macierz korelacji pełnego zbioru danych

Analiza macierzy korelacji

Powyższa macierz przedstawia zależność pomiędzy zmiennymi dla całej bazy danych liczącej 96453 rekordów.

- Kolumna $Temperature(C)$ oraz $ApparentTemperature(C)$ są silnie dodatnio skorelowane. Nie jest to zaskoczeniem ponieważ kiedy rzeczywista temperatura rośnie lub maleje to odczuwalna również.
- Ciekawym zjawiskiem jest zestawienie kolumn $Temperature(C)$ i $ApparentTemperature(C)$ wraz z kolumną $Humidity$. Na $Rysunek1$ wartość korelacji wynosi odpowiednio -0.61 i -0.58. Co oznacza, że gdy temperatura rośnie to wilgotność spada.
- Podsumowując, analizę można w matematyczny sposób przedstawić wpływ różnych zmiennych w naszym obserwowanym zjawisku. Tego typu wizualizacje pomagają wyobrazić i zrozumieć jak czynniki mają na siebie wpływ w obserwowanym doświadczeniu a które nie.

Poniżej znajduje się macierz korelacji dla zredukowanego zbioru danych, który zawiera 34635 rekordów.



Rysunek 2: Macierz korelacji zbioru danych liczący 34635 rekordów

Implementacja

Projekt składa się z jednego pliku Jupyter o nazwie *projekt.ipynb*. Środowisko Jupyter jest popularne dla analityków danych i inżynierów wykorzystujących uczenie maszynowe. Jupyter pozwala na wykonywanie kodu krok po kroku poprzez podział na bloki. Ta funkcjonalność umożliwia analizowanie kodu i jego prototypowanie. Wspiera bibilioteki typu Matplotlib lub Seaborn ktore pozwalają na interaktywną wizualizację danych.

Tasowanie i normalizcja danych

Listing 3: Podział danych na zbiór testowy i treningowy

```
y = df.pop("Summary")
X = df

def my_train_test_split(X, y, test_size=0.3, random_state=None):

    if random_state is not None:
        np.random.seed(random_state)

    test_size = int(test_size * len(X))

    shuffled_indices = np.random.permutation(X.index)
    indices_train = shuffled_indices[test_size:]
    indices_test = shuffled_indices[:test_size]

    X_train = X.iloc[indices_train]
    X_test = X.iloc[indices_test]
    y_train = y.iloc[indices_train]
    y_test = y.iloc[indices_test]

    return X_train, X_test, y_train, y_test

X_train, X_test, y_train, y_test = my_train_test_split(X, y, test_size
    =0.3)
```

Powyższy kod odpowiada za podział zbioru danych na zbiory testowe i treningowe

- Na początku zbiór danych jest dzielony na X - cechy i y - etykiety
- Funkcja *my_traing_test_split* przyjmuje jako argument zbiór cech, etykiet oraz opcjonalnie parametr rozmiaru zbioru testowego i ziarno losowości.
- Zwracane są 2 zbiory cech i 2 zbiory etykiet określone jako testowe i treningowe.

Listing 4: Klasyfikator KNN

```
class KNN:
    def __init__(self, k_neighbors=3):
        self.k_neighbors = k_neighbors

    def __distance_metric(self, point, data):
        return np.sqrt(np.sum((data - point) ** 2, axis=1))

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train

    def predict(self, X_test):
        predictions = []

        for x in X_test.values:

            distances = self.__distance_metric(x, self.X_train)

            indices = np.argsort(distances)[:self.k_neighbors]

            labels = [self.y_train.iloc[i] for i in indices]
            result = max(set(labels), key=labels.count)

            predictions.append(result)

        return predictions

    def score(self, X_test, y_test):
        y_pred = self.predict(X_test)
        accuracy = sum(y_test == y_pred) / len(y_test)
        return accuracy
```

Powyższy kod przedstawia implementacje klasyfikatora KNN

- Klasyfikator jest zaimplemetnowany jako klasa.
- Konstruktor pobiera parametr K najbliższych sąsiadów, domyślnie przypisanych jest 3 sąsiadów.
- Metoda *distance_metric* za pomocą metryki euklidesowej wyznacza odległość pomiędzy wektorami zmiennych.
- Metoda fit pobiera treningowy zbiór cech i etykiet i przypisuje do pól w klasie KNN.
- Metoda *predict* jest odpowiedzialna za wyznaczenie odległości pomiędzy danymi zbioru X_test i X_train i odpowiednie przewidywanie danych. Metoda zwraca etykietę, która najczęściej występowała wśród najbliższych sąsiadów.
- Metoda *score* pobiera zbiory testowe cech i etykiet a następnie porównuje przewidywane etkiety z tymi rzeczywistymi. Metoda zwraca stosunek poprawnie sklasyfikowanych punktów.

Testy

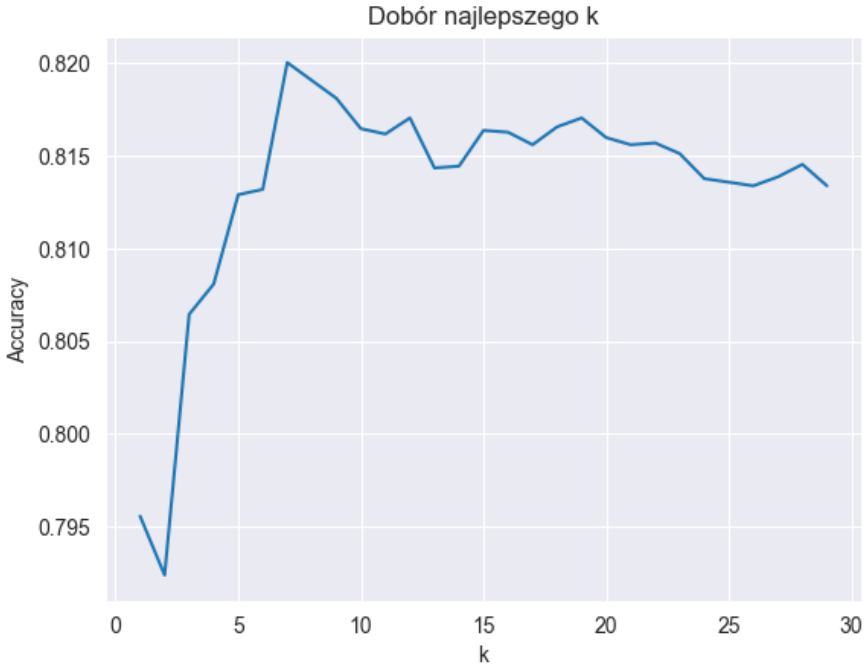
Pierwsza faza testów polegała na odpowiednim zredukowaniu ilości klas w zbiorze danych na potrzeby projektu. Nasza pierwotna baza zawiera sporą ilość rekordów oraz klas co powoduje wysoką złożoność obliczeniową modelu.

Po przygotowniu zbioru danych można było zacząć testy modelu.

Listing 5: Testowanie

```
1     list_of_accuracies = []
2     for i in range(1,30):
3         my_knn = KNN(i)
4         my_knn.fit(X_train, y_train)
5         list_of_accuracies.append(my_knn.score(X_test, y_test))
```

Powyższy kod jest odpowiedzialny za utowrzenie tablicy w której będą umieszczane wyniki dokładności naszego modelu w zależności od liczby k sąsiadów. W powyższym kodzie pobierzemy wyniki dokładności z zakresu $[1, 30)$ Dzięki wykorzystanym danym możemy zbadać dla jakiego k nasz model radzi sobie najlepiej. Z pomocą posłuży nam bliblioteka Matplotlib, która rysuje wykresy.

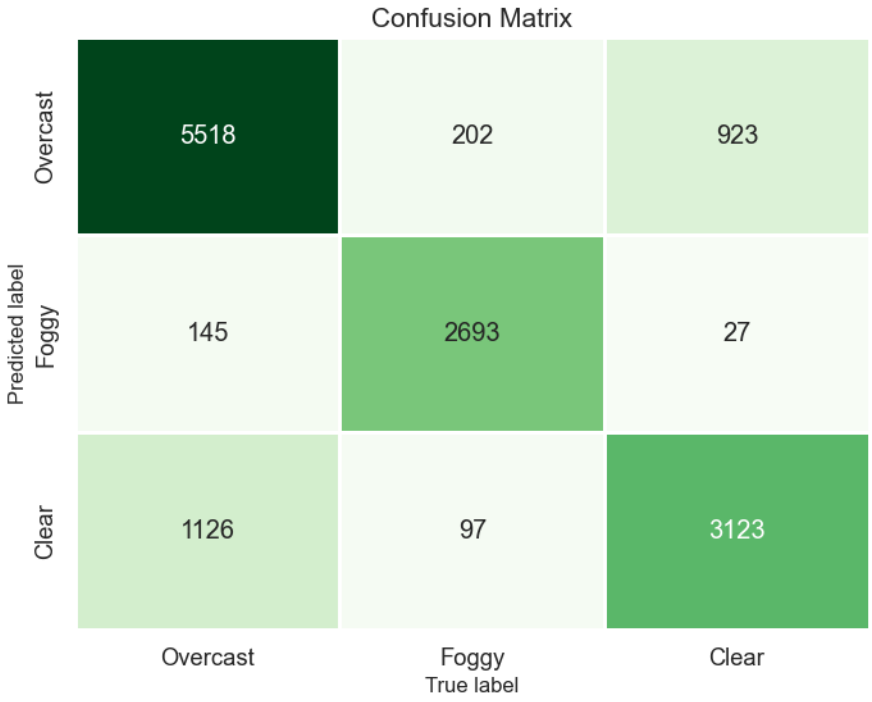


Rysunek 3: Wykres, który przedstawia zależność k sąsiadów od accuracy

Z powyższego wykresu wynika, że nasz model radzi sobie najlepiej z $k = 7$. Jednakże w przedziale $[1, 10]$ wartości accuracy znacząco rosną i dopiero w przedziale $(10, 30]$ wykres stablizuje się.
Natomiast wszystkie wartości dokładności oscylują w przedziale $(0.792, 0.82]$. Pokazuje to, że powyższy model oraz baza danych dobrze się zachowuje podczas klasyfikacji.

Tablica pomyłek

Tablica pomyłek jest to wizualizacja stosowana przy ocenie jakości klasyfikacji. Wiersze tablicy odpowiadają poprawnym klasom decyzyjnym a kolumny przedstawiają przewidywania klasyfikatora. Poniżej znajduje się tablica dla naszego modelu którego k wynosi 13.



Rysunek 4: Tablica pomyłek dla $k = 13$

Podsumowanie

Analizując powyższe wykresy, można wysunąć kilka wniosków na temat klasyfikatora KNN. Widać, że algorytm ten doskonale radzi sobie z danymi numerycznymi, co znajduje potwierdzenie w przypadku bazy pogodowej. Wyniki dokładności oscylują w przedziale między 0.792 a 0.82, co jasno wskazuje na skuteczność działania tego klasyfikatora. Jego zdolność do dokładnego przewidywania warunków pogodowych na podstawie historycznych danych jest zatem bardzo obiecująca.

Pełen kod aplikacji

Listing 6: Pełen kod aplikacji

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

df = pd.read_csv('weatherHistoryWithRoundValues.csv')

del df['Formatted Date']
del df['Apparent Temperature (C)']
del df['Precip Type']
del df['Loud Cover']
del df['Daily Summary']
del df['Humidity']
del df['Wind Bearing (degrees)']

df = df.drop(df[df['Summary'] == 'Breezy and Mostly Cloudy'].index)
df = df.drop(df[df['Summary'] == 'Breezy and Partly Cloudy'].index)
df = df.drop(df[df['Summary'] == 'Humid and Mostly Cloudy'].index)
df = df.drop(df[df['Summary'] == 'Breezy and Overcast'].index)
df = df.drop(df[df['Summary'] == 'Humid and Partly Cloudy'].index)
df = df.drop(df[df['Summary'] == 'Windy and Foggy'].index)
df = df.drop(df[df['Summary'] == 'Windy and Overcast'].index)
df = df.drop(df[df['Summary'] == 'Breezy and Foggy'].index)
df = df.drop(df[df['Summary'] == 'Windy and Partly Cloudy'].index)
df = df.drop(df[df['Summary'] == 'Breezy and Dry'].index)
df = df.drop(df[df['Summary'] == 'Dry and Mostly Cloudy'].index)
df = df.drop(df[df['Summary'] == 'Windy and Dry'].index)
df = df.drop(df[df['Summary'] == 'Humid and Overcast'].index)
df = df.drop(df[df['Summary'] == 'Dangerously Windy and Partly Cloudy'].
    index)
df = df.drop(df[df['Summary'] == 'Windy and Mostly Cloudy'].index)
df = df.drop(df[df['Summary'] == 'Dry and Partly Cloudy'].index)
df = df.drop(df[df['Summary'] == 'Windy'].index)
df = df.drop(df[df['Summary'] == 'Breezy'].index)
df = df.drop(df[df['Summary'] == 'Rain'].index)
df = df.drop(df[df['Summary'] == 'Partly Cloudy'].index)
df = df.drop(df[df['Summary'] == 'Mostly Cloudy'].index)
df = df.drop(df[df['Summary'] == 'Dry'].index)
df = df.drop(df[df['Summary'] == 'Light Rain'].index)
df = df.drop(df[df['Summary'] == 'Drizzle'].index)

df = df.reset_index(drop=True)

listofvalues = df['Summary'].unique().tolist()
for value in listofvalues:
    count = df['Summary'].str.contains(value).sum()
    print(value, " wystepuje: ", count)

plt.figure(figsize=(5, 5), dpi=150)
```

```
sns.heatmap(df.drop("Summary", axis=1).corr(), annot=True, cmap="jet", fmt=".2f")
plt.title("Correlation matrix")
plt.show()

y = df.pop("Summary")
X = df


def my_train_test_split(X, y, test_size=0.3, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)

    test_size = int(test_size * len(X))

    shuffled_indices = np.random.permutation(X.index)
    indices_train = shuffled_indices[test_size:]
    indices_test = shuffled_indices[:test_size]

    X_train = X.iloc[indices_train]
    X_test = X.iloc[indices_test]
    y_train = y.iloc[indices_train]
    y_test = y.iloc[indices_test]

    return X_train, X_test, y_train, y_test


X_train, X_test, y_train, y_test = my_train_test_split(X, y, test_size=0.3)

class KNN:
    def __init__(self, k_neighbors=3):
        self.k_neighbors = k_neighbors

    def __distance_metric(self, point, data):
        return np.sqrt(np.sum((data - point) ** 2, axis=1))

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train

    def predict(self, X_test):
        predictions = []

        for x in X_test.values:
            distances = self.__distance_metric(x, self.X_train)

            indices = np.argsort(distances)[:self.k_neighbors]

            labels = [self.y_train.iloc[i] for i in indices]
            result = max(set(labels), key=labels.count)

            predictions.append(result)

        return predictions
```

```
103
104     def score(self, X_test, y_test):
105         y_pred = self.predict(X_test)
106         accuracy = sum(y_test == y_pred) / len(y_test)
107         return accuracy
108
109
110 list_of_accuracies = []
111 for i in range(1, 30):
112     my_knn = KNN(i)
113     my_knn.fit(X_train, y_train)
114     list_of_accuracies.append(my_knn.score(X_test, y_test))
115
116 x_data = np.arange(1, 30, 1)
117 plt.plot(x_data, list_of_accuracies)
118 plt.xlabel('k')
119 plt.ylabel('Accuracy')
120 plt.title('Choosing the best k')
121 plt.grid(True)
122 plt.show()
123
124 my_knn = KNN(13)
125 knn_predicitons = my_knn.predict(X_test)
126 cm = confusion_matrix(y_test, knn_predicitons, labels=listofvalues)
127 cm_df = pd.DataFrame(cm, index=listofvalues, columns=listofvalues)
128 plt.figure(figsize=(8, 6))
129 sns.heatmap(cm_df, annot=True, cmap='Greens', fmt='d', linewidths=1,
        linecolor='white', cbar=False)
130 plt.title('Confusion Matrix', fontsize=15)
131 plt.ylabel('Predicted label', fontsize=12)
132 plt.xlabel('True label', fontsize=12)
133
134 plt.show()
```