Datenpipeline zur Verarbeitung von Daten der Datenquelle HyStreet.com

Mathieu, Daniel, Kai

01-01-2023

Inhaltsverzeichnis

1 Einführung

Ziel dieser Arbeit ist die Erstellung einer Datenpipeline, die die Daten der Datenquelle HyStreet verarbeitet, speichert und anschließend analysiert sowie visualisiert. Die Datenquelle stellt Informationen zur Anzahl an Passanten pro Stunden zu verschiedenen Standorten zur Verfügung, Genaueres dazu in den nächsten Kapiteln. Zunächst soll geklärt werden, aus welcher Motivation und mit welchem Ziel die Daten analysiert wurden.

1.1 Motivation

— Motivation —

2 Datenpipeline

Im nachfolgenden Teil der Arbeit werden die einzelnen Elemente der erstellte Datenpipeline nun genauer besprochen und erläutert. Dazu wird in Abbildung 1 zunächst die Datenpipeline als Ganzen betrachtet.

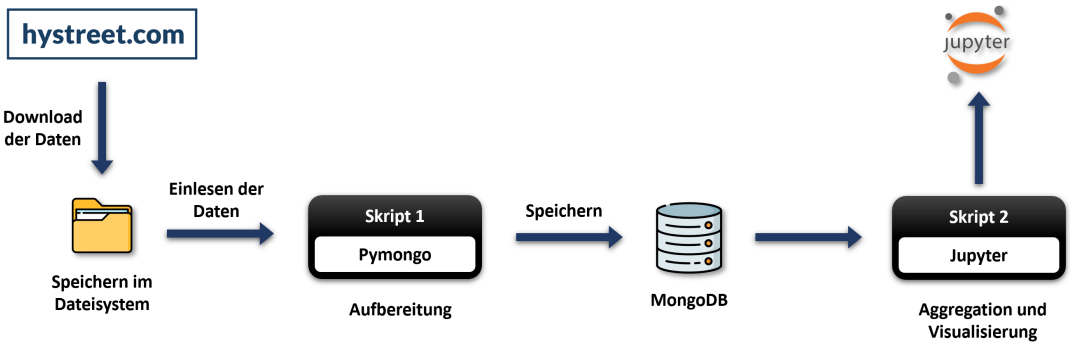

Figure 1: Datenpipeline zur Verarbeitung der HyStreet-Daten

Wie bereits erwähnt, stammen die Daten aus der Internetseite HyStreet.com. Die Webseite stellt somit auch den Startpunkt der Datenpipeline dar. Da die Daten nur kostenpflichtig automatisiert über eine API Schnittstelle heruntergeladen werden konnten, mussten wir uns für die Alternative entscheiden, die Daten manuell über deren Webseite herunterzuladen. Über den manuellen Download ist der Erhalt der Daten kostenlos. Zum genauen Vorgang des Downloads im folgenden Abschnitt Genaueres. Das Ergebnis dieses Schrittes ist eine Dateien im CSV-Format für den jeweiligen Standort. Der Aufbau dieser Dateien wurde bereits im einführenden Kapitel genauer beschrieben. Nach dem Herunterladen, werden die Dateien nun zunächst im lokalen Dateisystem gespeichert, um anschließend im Python Skript *Skript 1* verarbeitet werden zu können.

In diesem Skript werden die Daten aus allen heruntergeladenen Dateien mit Hilfe des Packages *Pandas* eingelesen und verarbeitet. Dabei werden diverse Spalten in ihre Einzelteile zerlegt und Spalten, die für unsere Zwecke nicht weiter erforderlich sind entfernt. Nach der Aufbereitung der Daten, werden diese schließlich in das JSON-Format umgewandelt, um im letzten Schritt in eine MongoDB eingelesen und gespeichert zu werden.

Nachdem die Daten nun in der MongoDB persistiert wurden, können diese nun im *Skript 2* analysiert und visualisiert werden. Sowohl die Analyse als auch die Visualisierung sind dabei in einem Jupyter Notebook integriert.

3 Datenaufbereiterung und Speicherung

In diesem Abschnitt betrachten wir den Teil der Datenpipeline, in dem die Daten heruntergeladen und anschließend im *Skript 1* aufbereitet und gespeichert werden.

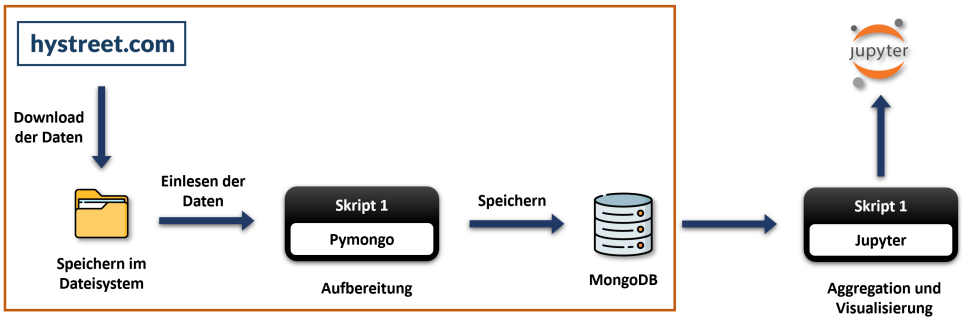

Figure 2: Datenpipeline - Einlesen und verarbeiten der Daten

3.1 Herunterladen der Daten

Wie bereits erwähnt stand für uns nur die kostenfreie Alternative zur Verfügung, die Daten manuell über die Webseite für jeden einzelnen Standort herunterzuladen. In Abbildung 3 zu sehen ist beispielhaft die Oberfläche der Webseite zum Herunterladen der Daten des Jungfernstegs in Hamburg. Auswählen ist Granularität der Daten, also, ob man die Angaben pro Stunde oder pro Tag möchte und der Zeitraum der gewünschten Daten ist auszuwählen.

Figure 3: Datenpipeline - Einlesen und verarbeiten der Daten

Dieses Prozedere für jeden gewünschten Standort durchgeführt, erhalten wir die jeweiligen CSV-Dateien. Diese werden im Ordner *input/* gespeichert, der Ordner, aus dem anschließend die Dateien eingelesen werden.

3.2 Aufbereiten der Daten

Die Daten werden nun eingelesen und anschließend so aufbereiten, dass sie später einfacher für die verschiedenen Analysen verwenden werden können. Das - und auch das Schreiben der Daten in die MongoDB - passiert im *Skript 1*, das ebenfalls in ein Jupyter Notebook integriert ist. Dafür benötigen wir die Packages *pymongo* (für die Verbindung zur MongoDB), *pandas* (zum Arbeiten mit den Daten), *json* (zum konvertieren von Daten in das JSON-Format) und *os* zum interagieren mit dem Dateisystem.

Nachdem wir die Packages importiert haben und die Datein im Ordner *input/* aufgelistet haben, lesen wir diese jeweils in ein eigenes Pandas-Dataframe ein und speichern sie gemeinsam in einer Liste.

[1]:
```
files = [ 'data/' + x for x in os.listdir('data') ]
data = [ pd.read_csv(x, delimiter=';') for x in files ]
```

Da wir ausschließlich die Daten der Jahre 2020 - 2022 betrachten werden, werd anschließend eine Funktion erstellt, die die Daten auf genau dieses Kriterium filtert und die gefilterten Daten zurückgibt.

[2]:
```
def filterData(data):
    '''Delete all rows where the year is not between 2020 and 2022'''
    for i in range(len(data)):
        data[i].drop(data[i][(data[i].year < 2020) | (data[i].year > 2022)].index,␣
↪inplace=True)
    return data
```

Wir definieren anschließend noch weitere Funktionen, um zusammengesetze Felder in den Daten aufzuteilen. Zu Erinnerung, wir startet mit Daten die wie folgt aussehen:

location	time of measuremnt	weekday	pedestrians count	temperature in c	weather condition	incidents

Ziel und Ergebnis der aufbereitung der Daten ist folgenden Schema:

address	weekday	pedestrians	temperature	weatherCondition	incidents	location	city	year	month	day	hour

Um dieses Ergebnis zu erreichen, definieren wir noch weitere Funktionen, die die einzelnen dafür benötigten Schritte ausführung und schließlich noch eine Funktion, die prüft, ob alle Daten in allen Städten vollständig sind. Hierbei wird geprüft, ob in allen Städten die gleiche Anzahl an Einträgen vorhanden ist.

Nach der Definition der einzelnen Funktionen, werden diese nur noch auf den ursprünglichen Datensatz angewendet und wir erhalten die aufbereiteten Daten mit dem passenden zuvor beschriebenen Schema.

[3]:
```
data = filterData(data)
data = separateLocation(data)
data = separateTime(data)
data = cleanData(data)
checkData (data)
```

3.3 Speichern der Daten

Der letzte Schritt im *Skript 1* besteht nun nur noch im Einlesen der Daten in die MongoDB. Dafür verwenden wir, wie erwähnt, das Package *pymongo*. Zunächst wird also eine Verbindung zum lokalen MongoDB-Client hergestellt und die Verbindung in einer Variable gespeichert. Anschließend wählen wir über den Client die Database *HyStreet* aus. Bevor wir die Daten nun einlesen, müssen diese noch in das JSON-Format überführt werden, was wir mit der Funktion *DATAFRAME.to_dict()*, zur Verfügung gestellt von *Pandas*, erreichen. Danach können die Daten schließlich in die Collection *HyStreetData* eingefügen werden. Damit die eingelesenen Dateien beim nächsten Durchlauf des Skripts nicht wiederholt eingelesen werden, werden die Dateien schließlich noch in einen anderen Ordner verschoben.

[4]:
```
client = pymongo.MongoClient('mongodb://localhost:27017')
db = client['HyStreet']

for i in range(len(data)):
    data[i] = data[i].to_dict(orient='records')
for i in range(len(data)):
    db.HyStreetData.insert_many(data[i])

for file in os.listdir('data'):
    shutil.move(f'data/{file}', f'geleseneDateien/eingelesen_{file}')
```

4 Analyse und Visualisierung

Betrachten wir nun *Skript 2*, in dem die aufbereiteten Daten innerhalb eines Jupyter Notebooks analysiert und visualisiert werden. Im Notebook werden zunächst die Daten aus der MongoDB eingelesen, einige Hilfsfunktionen definiert und die Standorte auf einer Karte mit Hilfe des Packages *Folium* angezeigt.

4.1 Vergleich Passantenanzahl pro Jahr

Um einen groben Überblick über die Daten zu erhalten, werden zunächst die Passantenanzahl pro Jahr für die einzelnen Standorte berechnet und die Standorte sowie Jahre miteinander verglichen. Für die Berechnung der Summe pro Jahr wird folgende Funktion definiert, die die gewünschten Daten in einem Pandas-Dataframe zurückgibt:

[1]:
```
def sumOfYear(data, year):
  result = []
  for address in addresses:
    filteredByYear = data.loc[data['address'] == address].loc[data['year'] == year]
    result.append((address, filteredByYear["pedestrians"].sum()))
  return result
```

Damit kann nun die absolute Passantenanzahl pro Jahr für jeden Standort ermittelt werden. Fügt man die Dataframes für die einzelnen Jahre nun zusammen, können die Standorte und Jahre mit dem Package *Plotly* und dessen Histogram-Plot einfach miteinander verglichen werden und liefern uns folgendes Ergebnis:

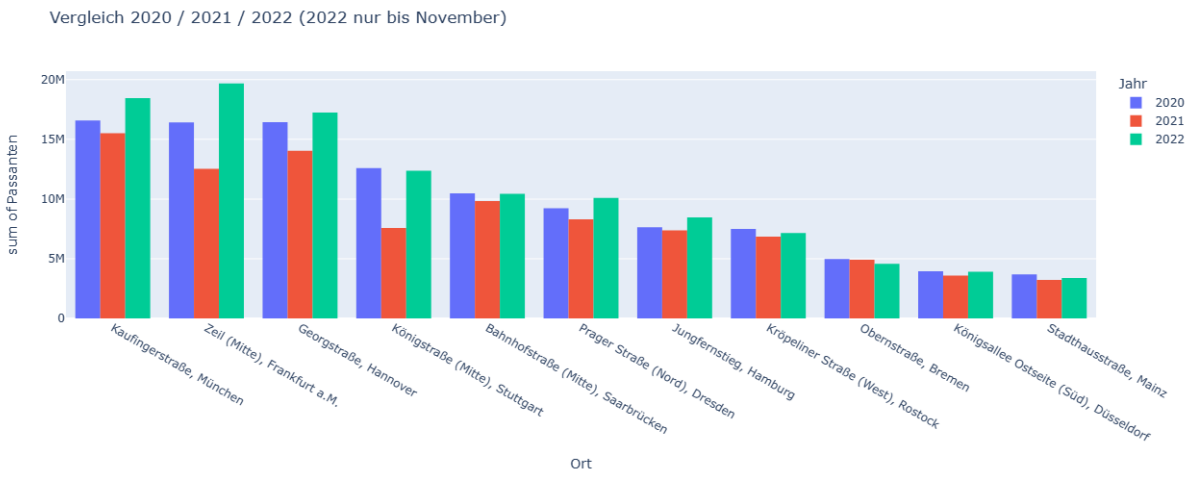

Figure 4: Vergleich der Standorte für die einzelnen Jahre

4.2 Uhrzeitenvergleich

Vor allem für Ladenbesitzer ist es nicht nur wichtig zu wissen, an welchem Wochentag die meisten Menschen an den jeweiligen Standorten unterwegs sind, sondern auch um welche Uhrzeit. Betrachtet werden hier alle Standorte gleichzeitig, um ein allgemeines Gefühl zu erhalten, welche Uhrzeiten besonders beliebt sind. Möchte man nur einen speziellen Standort betrachten, muss man die Eingabedaten zuvor nur auf den jeweiligen Standort filtern und die Funktion erzielt die gleichen Ergebnisse. In SQL übersetzt würde die Funktion folgendens berechnen, $SUM(pedestrians)GROUPBYMonth, Hour$, betrachtet also für alle Standorte gleichzeitig wie viele Passanten pro Monat pro jeweiliger Stunde gemessen wurde:

[2]:
```
def absPerHour(data, years):
  dataTime = data.loc[data['year'].isin(years)]
  dataTL = list()
  monate = ["Januar", "Februar", "März", "April", "..."]
  df = pd.DataFrame(columns=["Monat", "Stunde", "Passanten"])
  for month in range(12):
    tmp = dataTime.loc[dataTime['month'] == (month + 1)]
    tsum = tmp.groupby('hour')['pedestrians'].sum()
    for hour in range(0, 24):
      df = df.append({"Monat" : monate[month], "Stunde" : hour, "Passanten" : tsum.
↪iloc[hour] }, ignore_index=True)
  return df
absHours = absPerHour(data, [2020])
```

Das resultierende Pandas-Dataframe kann anschließend wieder mit Plotly und dessen Linien-Plot visualisiert werden. Zu erkennen ist, dass der Höhepunkt, unabhängig des Monats gegen ca. 16 Uhr erreicht wird.

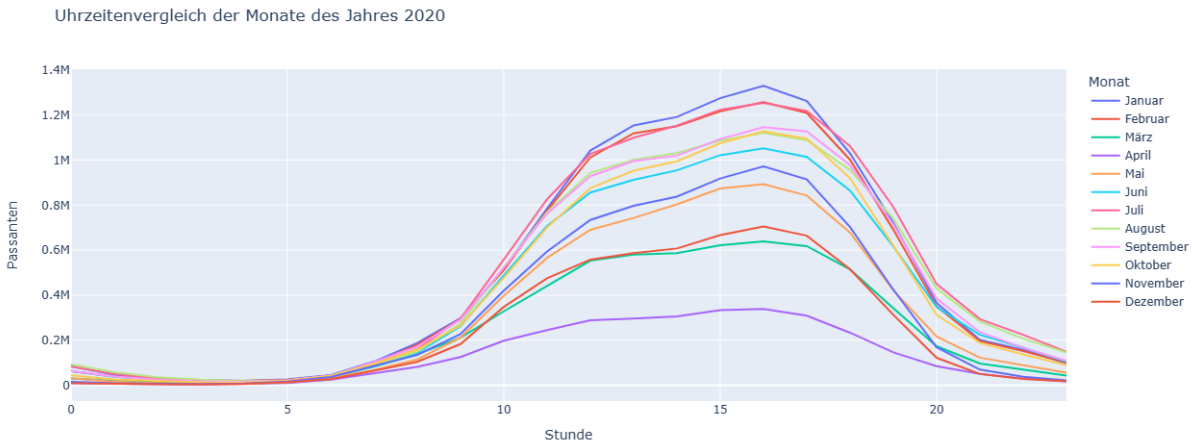

Figure 5: Vergleich der Passantenanzahl pro Monat pro Stunde

List of Figures

List of Tables