Fachhochschule
Nordwestschweiz

IRIS Centroid Browser Challenge

Bericht

Christopher Frame, Eugen Cuic, Lukas Reber

Brugg, 21.12.2020

Inhaltsverzeichnis

1 Einleitung

Das vorliegende Dokument beschreibt die Umsetzung der Challenge «IRIS Centroid Browser», welche im HS20 im Studiengang BSc Data Science an der FHNW von Christopher Frame, Eugen Cuic und Lukas Reber erarbeitet wurde.

Die Arbeit dokumentiert den Verlauf sowie die Resultate der Projektarbeit innerhalb der Challenge.

1.1 Aufgabe

Die Aufgabenstellung ist in DS Spaces unter folgendem Link dokumentiert: https://ds-spaces.technik.fhnw.ch/iris-centroid-browser/

Mittels der Aufgabenstellung und ersten Gesprächen mit den Stakeholdern wurden die unter Kapitel 1.2 aufgeführten Requirements definiert, nach welchen das Projektteam die Challenge umgesetzt hat.

1.1.1 Vorgehen / Entwicklungsprozess

Um eine speditive Vorgehensweise zu ermöglichen und die Interdependenzen gering zu halten, wurde versucht, die Aufgaben bestmöglich aufzuteilen. Dies ermöglichte es, dass jedes Teammitglied auch an anderen Projekten arbeiten konnte, ohne den Fortschritt des Projekts negativ zu beeinflussen.

Der Fortschritt wurde an einem gemeinsamen Meeting jeden Donnerstag kommuniziert. Dadurch war es möglich, allfällige Abhängigkeiten und Anpassungen an Arbeiten anderer Teammitglieder frühzeitig zu erkennen und entsprechend zu priorisieren.

Die Entwicklung des Projekts wurde in folgender Sequenz durchgeführt:

1. Das Verständnis von Daten und Anforderungen an Server
2. Aufsetzen von Server
3. Erstellung der CI / CD Pipeline
4. Erstellung des Datenmodell
5. Befüllung der Datenbank
6. Erstellung des Grundgerüstes der Webapplikation
7. Verbindung von Webapplikation mit Datenbank
8. Fertigstellung V1 ohne Interaktivität
9. Anpassung der Datenbank für V2
10. Erstellung Grundgerüst der Webapplikation für V2
11. Update von CI / CD und Testfällen

1.1.2 Organisation

Die drei grössten Aufgabengebiete wurden innerhalb des Projektteams aufgeteilt, wobei für jedes Aufgabengebiet eine verantwortliche Person definiert wurde:

- Webtechnologie: Eugen Cuic
- Datenbank / Datenwrangling: Christopher Frame
- Softwarekonstruktion: Lukas Reber

1.2 Requirements

Die Requirements wurden zu Beginn mit den Stakeholdern (Challenge Owner/innen und Fachexperten / Fachexpertinnen) vereinbart und wie folgt festgelegt:

Produkt

Das Endprodukt (die Webseite) muss folgende Anforderungen erfüllen: (V1)

- Anzeige einer Auswahlliste von Centroids (hierbei kann der/die Anwender/in einen Wert zwischen 1-53 wählen (ohne 52)
- Nach der Selektion eines bestimmten Centroid erhält der/die Anwender/in eine Auswahl an Beobachtungen angezeigt, welche den entsprechenden Centroid beinhalten.
- Durch die Auswahl einer dieser Beobachtungen wird dem/der Benutzer/in detaillierte Informationen zum entsprechenden Centroid, respektive der entsprechenden Beobachtung angezeigt. Dies ist:
    - Anzeige eines Diagramms mit der Anzahl Vorkommnisse des Centroids pro Zeiteinheit
    - Optional: Zusätzliche Informationen wie Anzahl Steps, Zeitpunkt der Beobachtung und weitere Informationen falls vorhanden (V2)
- In der detaillierten Ansicht der Beobachtung kann zusätzlich pro Step das entsprechenden Slit-Jaw Image (SJI) mit den visualisierten Y-Streifen angezeigt werden.

Datenbank / Data Wrangling

- Alle nötigen Informationen sind in der Datenbank vorhanden und abrufbar.
- Das Datenbankmodell ist so konzipiert, dass die Abfragen, welche über die Webseite aufgerufen werden, in angemessener Zeit bearbeitet werden können.
- Das Datenbankmodell ist so aufgebaut, dass die Daten möglichst speichereffizient abgelegt sind.

Anforderung Pipeline

- Die Daten sollen automatisch von der Datenbank auf dem Server0090 ausgelesen und direkt in eine Datenbank auf der virtuellen Maschine geschrieben werden.
- Die Pipeline soll die Daten möglichst effizient in die Datenbank schreiben.
- Der Zeitraum, um Daten auszulesen und in die Datenbank zu schreiben, soll einfach anpassbar sein.

Softwarekonstruktion

- Versionskontrolle mit Git
- Pflege des kompletten Source Codes (Datenbank, Backend, Frontend) in einem gemeinsamen Applikationsrepository
- Verwendung eines definierten Branching Workflows in der Entwicklung
- Continuous Integration aller Branches
- Continuous Delivery Pipelines für das Deployment aller Applikationsbestandteile auf die Produktivumgebung
- Unit Tests der funktionalen Methoden im Backend
- Kontinuierliche Auslieferung von funktionalen Erweiterungen während der Entwicklung der Applikation
- Sauber strukturierter und kommentierter Code

Webtechnologie

- Verständnis wie eine Webapplikation aufgebaut ist
- Die Webapplikation soll durch verschiedene Tests evaluiert werden
- Die durchgeführten Tests sollen in einem Testbericht ersichtlich sein
- Die Webapplikation soll beim Aufbau so wenig wiederholende Elemente aufweisen wie möglich

2 Architektur

2.1 Software-Architektur

Die Architektur der gesamten Applikationsumgebung besteht aus den folgenden Komponenten:

- Webserver (IP: 213.136.68.142):
  Darauf wird die lauffähige Version der Django Applikation deployed

- Datenbankserver (IP: 213.136.68.142):
  Postgres Datenbank, welche von der Webseite verwendet wird

- Repository (https://github.com/eugencuic/FHNW_HS20_IRIS_Centroid_Browser):
  Code Repository für die Webapplikation sowie die weiteren Codefragmente, welche im Laufe des Projekt erstellt wurden

- IRIS Server (server0090.cs.technik.fhnw.ch):
  Source Server der IRIS Daten welche für die Challenge verwendet wurden.

- Clients: Entwicklung der Applikation und Zugriff auf den IRIS Server

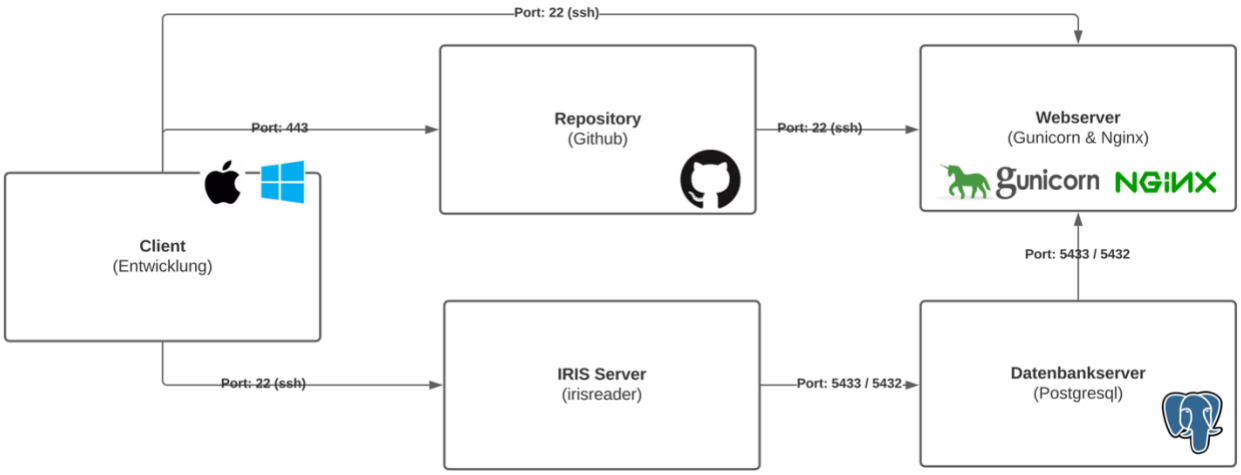

Abbildung 1: Architekturübersicht

Die Entwicklung der Webapplikation findet lokal auf den Clients der Projektmitarbeitenden statt. Sämtlicher Code, welcher für die Webapplikation oder die Datenpipeline erstellt wurde, ist im Repository auf Github gespeichert.

Die Webapplikation wird mittels dem Python Webframework Django in der Version 3.1.2 erstellt. Django bietet einen integrierten Webserver, welcher während der Entwicklung der Website lokal auf dem Client gestartet werden kann. Für den Betrieb der Webseite auf der produktiven Umgebung werden Gunicorn und Nginx eingesetzt. Da es sich bei Django um eine Python Applikation handelt, werden ein WSGI Server (Web Server Gateway Interface), welcher den dynamischen Teil der Webseite aufbereitet, sowie ein Webserver benötigt, welcher den statischen Teil der Webseite (Bilder, CSS Dateien, JS Dateien) bereitstellt.

2.1.1 Schnittstellen

Die in Abbildung 1 dargestellte Architekturübersicht entspricht dem geplanten Soll-Zustand des Projekts. Aufgrund von Netzwerkrestriktionen innerhalb des FHNW Netzwerks konnten nicht alle Schnittstellen wie abgebildet implementiert werden.

Der Webserver sowie der Datenbankserver laufen nun auf einem Ubuntu Server ausserhalb des FHNW Netzwerks, da der Zugriff von Github mittels ssh auf das interne FHNW Netzwerk nicht möglich ist. Zudem ist der Zugriff vom IRIS Server über Port 5433/5432 auf einen Server ausserhalb des FHNW Netzes auch nicht möglich. Aus diesem Grund werden die Daten zuerst auf einem Datenbankserver innerhalb des FHNW Netzes aufbereitet und anschliessend manuell auf den von der Webapplikation verwendeten Datenbankserver migriert.

2.2 Ausgangslage der Daten

Die Daten für den IRIS Centroid Browser sind auf dem server0090.cs.technik.fhnw.ch, welcher über das FHNW Netzwerkes erreichbar ist, in einer hierarchischen Struktur, geordnet nach Datum, abgelegt. Die Daten beinhalten Beobachtungen der äusseren Bereiche der Sonnenatmosphäre für den Zeitraum von 2013 bis 2019. Da die Sonne im Jahr 2014 besonders aktiv war, gibt die Aufgabenstellung vor, nur die Daten aus diesem Jahr zu berücksichtigen.

Für den IRIS Centroid Browser wurden zwei verschiedene Arten von Daten verwendet. Für jede Beobachtung existiert eine Serie von Bildern mit unterschiedlichen Filtern, welche als Slit Jaw Images (kurz SJI) bezeichnet werden. Diese SJI haben in der Mitte eine vertikale Linie an schwarzen Pixeln, da dort Messungen der elektromagnetischen Strahlung gemacht wurden. Diese Messungen werden als Raster bezeichnet und wurden als Spektrogramme gespeichert. In einem weiteren Schritt wurden diese Raster analysiert und in 53 verschiedene Gruppen unterteilt, sogenannte Centroids.

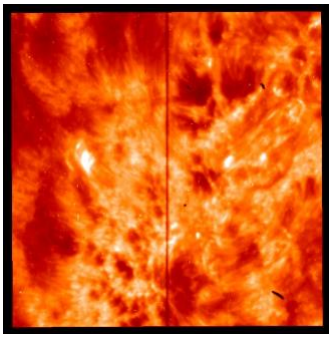

Abbildung 2: Beispiel eines SJI

Die Daten enthalten Beobachtungen an verschiedenen Tagen, die ihrerseits in der Frequenz stark variieren: Während an einigen Tagen keine Beobachtungen durchgeführt wurden, existieren für andere Tage gleich mehrere Messungen. Jede Beobachtung besteht ihrerseits aus mehreren Steps. Diese Steps sind einzelne Messungen, bei denen die Spektralgruppen der Sonne gemessen und ein SJI erstellt wurde. Die Messungen der Spektralgruppen und die SJI wurden nicht zur selben Zeit aufgenommen und sind von uns zugeordnet worden.

3 Datenbank-Architektur

3.1 Aufbau in zwei Schritten

Der Auftrag der Challenge war es, die Web-App in zwei verschieden Versionen zu erstellen. Mit der ersten Version werden nach Auswahl des Centroids alle Beobachtung aufgeführt, in welcher dieser vorkam. Nach der Auswahl einer Beobachtung wird ein Diagramm angezeigt, das die Häufigkeit der Vorkommnisse des Centroids in dieser Beobachtung visualisiert.

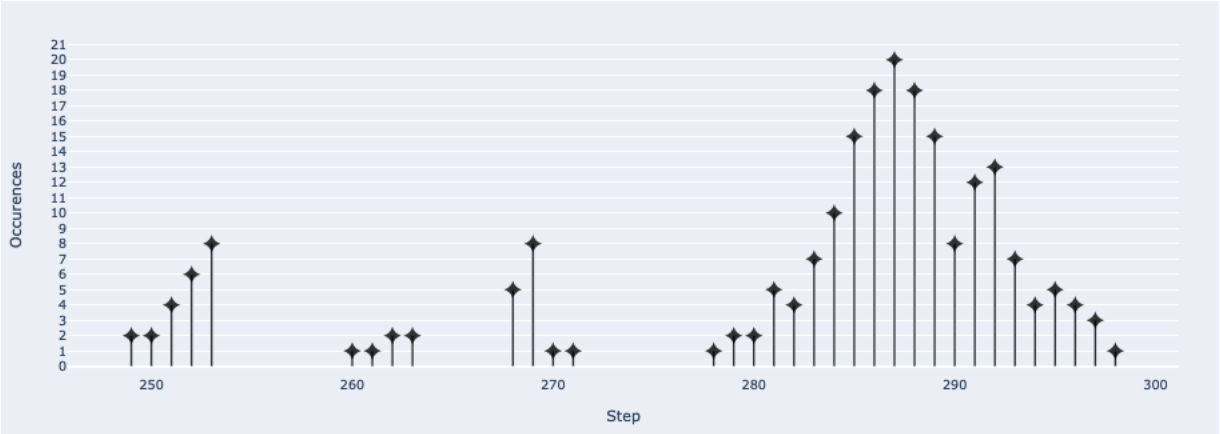

Abbildung 3: Beispieldiagramm der Website Version 1 (Centroid 1 mit Beobachtung 20140101_063241_3840257196)

Für die zweite Version sollen auch die SJI angezeigt werden. Bei den SJI wird zusätzlich eine Linie eingezeichnet, die aufzeigt, wo genau dieser Centroid gemessen wurde.

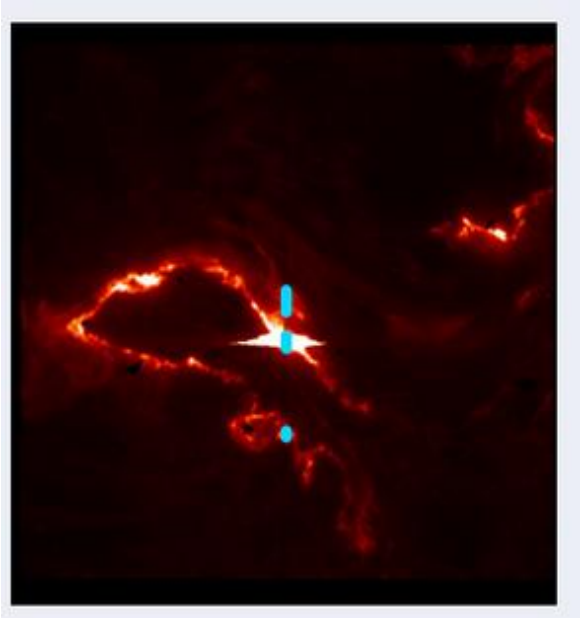

Abbildung 4: SLJ mit visualisertem Pixel (Centroid 1, Beobachtung: 20140910_112825_3860259453, Filter: 1400)

Für die zweite Version wurde eine separate Datenbank erstellt, wobei die Datenbank der Version eins als Grundlage für die Datenbank der Version zwei diente. Da die Webapplikation

mit dem Django Framework erstellt wurde, war eine zwingende Anforderung an die Datenbank, dass sie mit Django kompatibel ist. Weitere Anforderungen an die Datenbank waren eine geringe Datenredundanz sowie ein einfaches Modell für eine langzeitige Speicherung der Daten. Deshalb ergab sich als passender Datenbanktyp für diese Verwendung eine relationale Datenbank. Die Entscheidung ist dabei auf eine Postgres Datenbank gefallen, da diese alle Anforderungen erfüllt und auch bei grösseren Datensätzen performant bleibt.

3.2 Datenbank Version 1

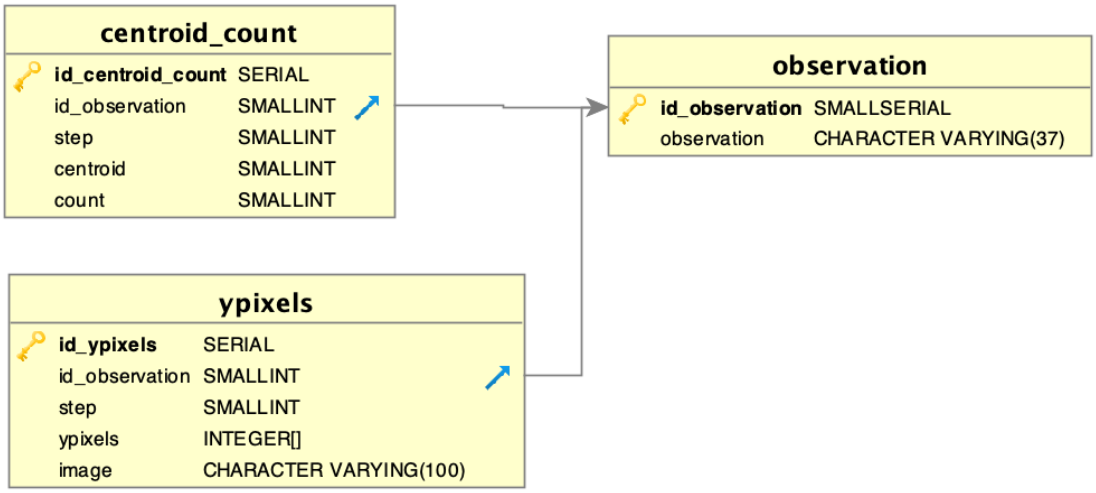

Abbildung 5: ERD Datenbank Version 1

Die Datenbank besteht aus drei Tabellen, wobei mit dem Schlüssel immer der Primarykey markiert ist. Die Primarykeys wurden als Serial spezifiziert, das heisst, dass sich die ID pro neue Zeile um eins erhöht. Somit kann sichergestellt werden, dass nicht zwei Einträge dieselbe ID aufweisen. Diese Werte werden intern als Integer abgespeichert und können nur positive Werte annehmen. Die Pfeile geben an, welches Foreignkeys sind und auf welchen Primarykey sie referenzieren.

3.2.1 Erklärung der einzelnen Tabellen:

Tabelle observation

- id_observation: Eindeutig Angabe der Observation (zur Normalisierung, Primarykey)
- Observation: Observationsname

Tabelle centroid_count

- id_centroid_count: Eindeutig Angabe der Zeile (Primarykey)
- id_observation: Bezeichnung der Observation
- step: Messung der Beobachtung
- centroid: Welcher Centroid gezählt wurde
- count: Wie oft der Centroid in einem Step vorkam

Tabelle ypixels (Die Tabelle ypixels dient als Vorbereitung für die Version 2)

- id_ypixels: Eindeutig Angabe der Zeile (Primarykey)
- id_observation: Bezeichnung der Observation
- step: Messung der Beobachtung
- ypixels: Die vorgekommenen centroids als Array
- image: Hier war geplant, der Pfad zum entsprechenden Bild zu speichern

3.3 Datenbank Version 2

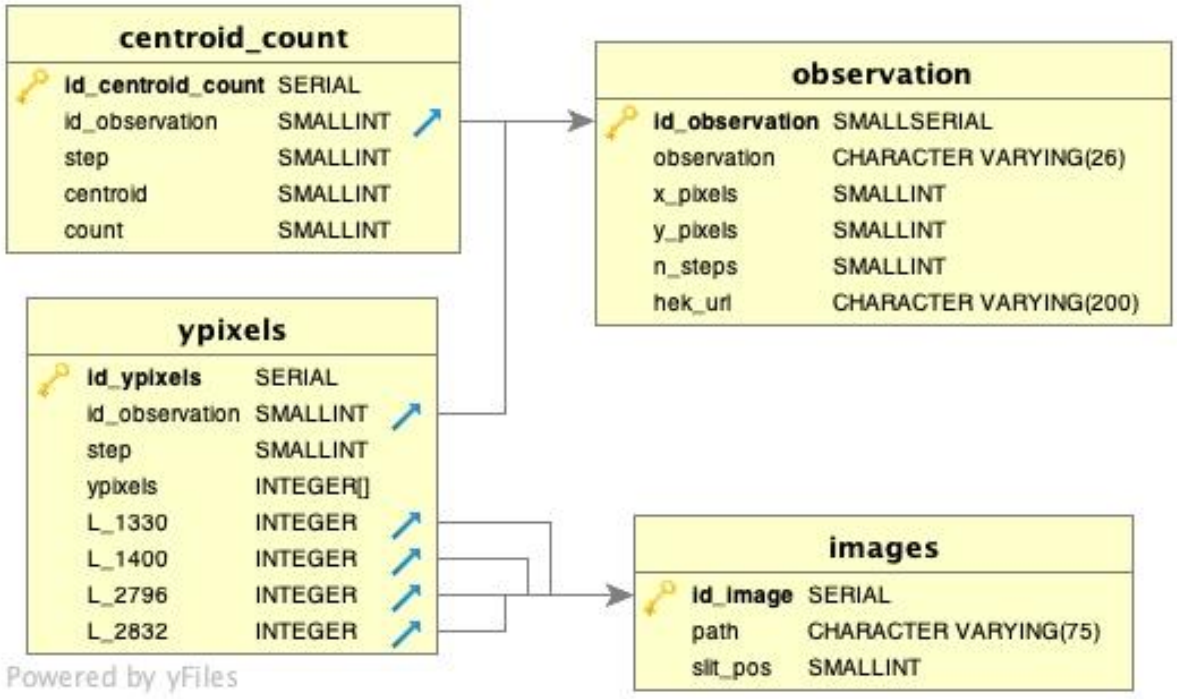

Abbildung 6: ERD Datenbank Version 2

Für die zweite Version der Web-App konnte die Tabelle centroid_count so belassen werden, wie sie für Version 1 erstellt wurde. In der Tabelle Observation sind zusätzliche Informationen über die Bilder sowie die URL zur Beobachtung in der Heliophysics Events Knowledgebase hinzugefügt worden. Die Information im Attribut slit_pos ist für jedes Bild anders, deshalb muss sie in der Tabelle images, welche deutlich mehr Einträge als die Tabelle observation beinhaltet, gespeichert werden.

Die Bilder, welche auf der Website dargestellt werden, sind ausserhalb der Datenbank in einer Ordnerstruktur abgespeichert worden. Um den Pfad zu den Bildern korrekt abzuspeichern, wurde eine neue Tabelle images hinzugefügt. Weiter wurden vier Spalten in der Tabelle ypixels eingefügt, welche zu den Bildern in der Tabelle images verweisen. Diese Spalten beinhalten die Information, welche Bilder zeitlich am nächsten zu den Rastern aufgenommen wurden.

3.3.1 Indexieren der Tabellen

Datenbankaufrufe in der ersten Version der Web-App dauerten sehr lange. Dies wurde in der zweiten Version verbessert, indem zusätzlich zu den indexierten Primarykeys weitere Indizes erstellt wurden. Die indexierten Tabellen sind folgende:

Tabelle centroid_count:

- id_observation
- centroid

Tabelle ypixels:

- id_observation
- step

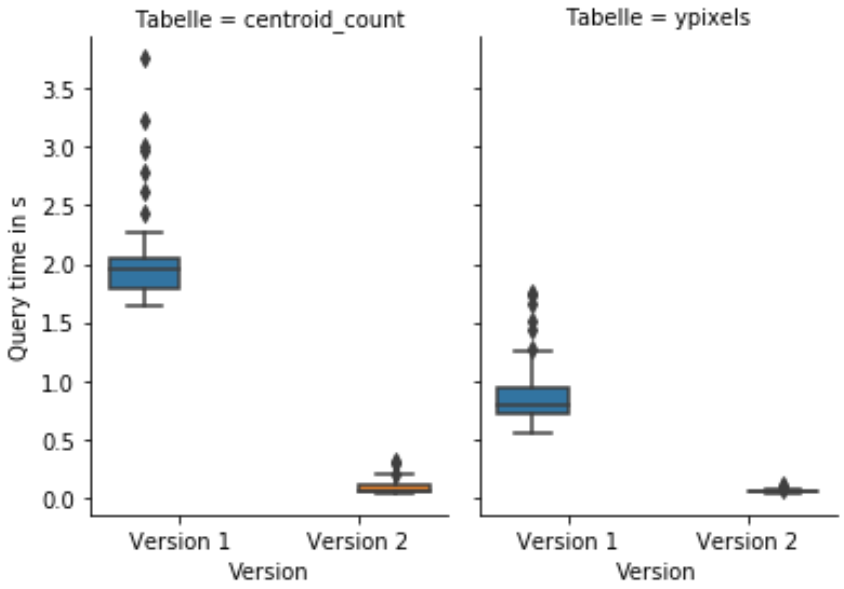

Abbildung 7: Vergleich der Query Time mit und ohne Indizes

Um den Effekt der Indizierung zu messen, wurden die Datenbanken der Version eins und zwei verglichen. Dabei war nur die Version zwei indexiert. Hierzu wurde ein Programm erstellt, welches 50 Select Queries mit einem Filter für die id_observation und dem Centroid, respektive dem Step, an die Tabellen stellten. Beim Filter wurde für jede Query die gesuchte Observation verändert. Im Diagramm ist deutlich zu erkennen, dass die indexierte Version zwei schneller als die nicht indexierte Version eins ist.

3.3.2 Anpassungen mit SQL

Nach dem Ausführen des Skripts, welches die Datenbank erstellt und befüllt, wurde festgestellt, dass für die Entwicklung der Web-App noch zwei Anpassungen an der Datenbank notwendig sind. Zum einen wurden die Namen der Spalten 1330, 1400, 2796 und 2832 in der Tabelle ypixels mit dem Präfix «L_» ergänzten. Dies, da die verwendeten Queries oft als Integer und nicht als String interpretiert wurden und somit fehlschlugen. Weiter wurden in den genannten

Spalten die Werte ersetzt, welche einen Nullwert anstatt der Zahl Null beinhalten. Dies vereinfacht das Anzeigen der Bilder in der Web-App.

4 ETL Pipeline

Im folgenden Kapitel ist die Pipeline, welche die Rohdaten konvertiert und in der richtigen Struktur in die Postgres Datenbank speichert, dokumentiert.

4.1 ETL Pipeline Version eins

```
#check if database exists
connection = None
try:
    connection = psycopg2.connect("VM_DB_adress")
    print("connected to Database")

except:
    print('Database not connected or does not exist.')

if connection is None:
    if input("Do you want to try to create the Database iris? (y/n)") == ("y"):
        try:
            #create Database called iris with the role postgres
            engine = create_engine(VM_DB_adress[:-5]) #slice -5 becaue iris is not yet created
            con = engine.connect()
            con.execute("commit")
            con.execute("CREATE DATABASE iris")
            con.close()
            print("Database was created.")
        except:
            print("error while trying to create the Database")
            exit()
    else:
        exit()

#create engine for DB
engine = create_engine(VM_DB_adress)
con = engine.connect()

if input('''type "drop" to drop tabels ypixels, cetroid_count and observation and after recreate them''') == ("drop"):
    drop_tables(con)


try:
    con.execute('''SELECT "id_observation" FROM "observation" LIMIT 1;
        SELECT "id_observation" FROM "centroid_count" LIMIT 1;
        SELECT "id_observation" FROM "ypixels" LIMIT 1;
        ''', con) #to check it Tabels exist
except:
    create_tables(con)
    print("tables were created")


print("starting to extract data and write into database")
fill_DB(startdate, enddate, con)
```

Abbildung 8: Zugriff auf den Server & Erstellen der Datenbank

Damit das Skript funktioniert, muss Postgres auf dem Server, der die Datenbank hosten soll, installiert sein. Im Skript müssen weiter die korrekten Angaben zur Verbindung eingefügt werden. Danach sucht das Skript nach der Datenbank Iris. Falls die Datenbank noch nicht besteht, wird der Benutzer gefragt, ob er diese erstellen will. Als Nächstes muss der Nutzer entscheiden, ob er die bereits vorhanden Daten in der Datenbank löschen will. Dies um sicherzugehen, dass nicht dieselben Daten mehrmals in der Datenbank gespeichert werden. Falls die Relationen der Datenbank noch nicht bestehen, werden sie erstellt.

4.2 Extrahieren und Transformieren

Um die Daten auszulesen, wird das IRISreader-Paket verwendet. Dies zum einen, um durch die Daten zu iterieren und zum anderen, um sie korrekt anzuzeigen.

```
for date in daterange:
    obsit = obs_iterator("/data2/iris/20{}".format(date.strftime("%y/%m/%d")), read_v4=False) # iterate through dates
    print("starting:", date.strftime("%y/%m/%d"))
    #create empty variables for table centroid_count
    col_names =  ['id_observation', 'step', 'centroid', 'count']
    centroid_count_oneday  = pd.DataFrame(columns = col_names) #panda DF for table centroid_count
    n_obs = 0

    #create empty variables for table observation
    obs_path = [] #list that will append the observation path


    #create empty variables for table ypixels
    col_names =  ['id_observation', 'step', 'ypixels', 'image']
    ypixels_oneday  = pd.DataFrame(columns = col_names)



    for obs in obsit: #iterate through each observation
        if obs.raster.has_line("Mg II k"):
            n_obs +=1 #used to for indexing observation
            obs_path.append(obs.path) # appends list with all observations path
            raster = obs.raster("Mg II k")
            steps = raster.n_steps

            for step in range( steps ): #iterate through each step
                img = raster.get_interpolated_image_step( step, LAMBDA_MIN, LAMBDA_MAX, n_breaks=216 )
                centroids = assign_mg2k_centroids( img ) #cetroid is a np.array which lists the centroids of each step

                (unique, counts) = np.unique(np.array(centroids), return_counts=True) #two lists with the frequency and count

                centroid_step_data = {'id_observation': n_obs, 'step': step, 'centroid': unique, 'count': counts}
                centroid_step = pd.DataFrame(centroid_step_data)
                centroid_count_oneday = centroid_count_oneday.append(centroid_step) #append each step

                ypixels_step_data = {'id_observation': n_obs, 'step': step, 'ypixels': [centroids]}
                ypixels_step = pd.DataFrame(ypixels_step_data)
                ypixels_oneday = ypixels_oneday.append(ypixels_step)#append each step
```

Abbildung 9: Auslesen der Daten

Das Auslesen der Daten geschieht in drei for Schlaufen. Die erste Schlaufe iteriert durch alle Tage im gewählten Zeitraum. Die zweite Schlaufe wird verwendet, um an diesem Tag alle Beobachtungen aufzurufen; in dieser for Schalufe wird auch der Name der Observation in einer Liste abgespeichert, um danach ein Pandas Dataframe zu erstellen. In der dritten for Schlaufe wird jeder einzelne Messschritt einzeln aufgerufen.
Mit numpy unique wird im array centroid nach den Centroids gesucht, die in einem Raster aufgetreten sind. Danach wird für jede Tabelle ein Pandas Dataframe erstellt, in welchem die Anzahl der vorhanden Spektralgruppen gespeichert wird. Anschliessend werden zwei Pandas Dataframes erstellt, um die Daten pro Raster abzuspeichern. Das Dataframe pro Raster wird darauf in einem Dataframe mit dem Suffix «_oneday» gespeichert. Dies, damit die Daten pro Tag in die Datenbank geschrieben werden können.

4.3 Laden

```
        if n_obs > 0:
            observation = pd.DataFrame({'observation': obs_path}) #creates observation panda DF to write to DB

            #get the las id_observation and add 1
            highest_entry = con.execute('SELECT "id_observation" FROM "observation" ORDER BY "id_observation" DESC LIMIT 1', con)
#last obdervation id
            try: highest_entry = [row[0] for row in highest_entry][0] #new id for observation
            except: highest_entry = 0 #new id for observation

            #add to dataframe for correct id_observation
            centroid_count_oneday['id_observation'] = centroid_count_oneday['id_observation'] + highest_entry
            ypixels_oneday['id_observation'] = ypixels_oneday['id_observation'] + highest_entry


            #observation to DB
            observation['observation'] = observation['observation'].str.slice(start=12)
            observation.to_sql('observation', con, index=False, if_exists='append')

            #centroid_count to DB
            centroid_count_oneday.to_sql('centroid_count', con, index=False, if_exists='append')

            #ypixels to DB
            ypixels_oneday['ypixels'] = ypixels_oneday['ypixels'].apply(lambda x: x.tolist()) #psycopg2 can't handle np.ndarry
            ypixels_oneday.to_sql('ypixels', con, index=False, if_exists='append')
        else:
            print("NO observations in directory on {}".format(date.strftime("%y/%m/%d")))

    endtime = datetime.datetime.now()
    runtime = endtime -  starttime
    print("Runtime to fill DB: {} hours".format(runtime.total_seconds()/3600))
```

Abbildung 10: Schreiben der Daten in die Datenbank

Falls an einem Tag Beobachtungen durchgeführt wurden, sind die Daten pro Tag mittels pandas.to_sql in die Datenbank geschrieben worden. Hierbei musste beachtet werden, dass die Regeln der Datenbank mit den Primary- und Foreignkeys zu keinem Zeitpunkt verletzt werden.

Die Daten konnten nicht direkt auf unsere Zieldatenbank geschrieben werden, da der entsprechende Port für den Zugriff nicht geöffnet war. Deshalb wurden die Daten zuerst auf den server2064.cs.technik.fhnw.ch geschrieben. Im Anschluss wurde ein Datenbankdump erstellt und manuell auf den externen Zielserver kopiert, von wo aus die fertige Datenbank aus dem Datendump erneut erstellt wurde.

4.4 ETL Pipeline Version zwei

Bei der Version zwei blieb die Grobstruktur des Skripts bestehen. Neu hinzu kam, dass am Anfang eine CSV-Datei geladen wird. Aus dieser Datei werden die Informationen zu den SJI ausgelesen. Weiter lässt sich aus den Informationen in der CSV-Datei der Pfad der Ordnerstruktur herleiten.

```
if len(images_obs_1330) > 0:
    row = images_obs_1330.index.get_loc(timestamp, method='nearest')
    image_1330 = images_obs_1330.iloc[row]['id_image']
else:
    image_1330 = 0
```

Abbildung 11: Matchen der Raster auf die SJI

Um nicht bei jedem Messschritt der Beobachtung alle Daten der CSV-Datei zu durchsuchen, wurden für jede Beobachtung vier Dataframes mit den entsprechenden Filtern (1330, 1400, 2796 und 2832) erstellt. Diese Dataframes verwenden den Timestamp als Index. Dadurch konnte mit der methode «nearest» der Funktion «get_loc» das zeitlich am nächsten liegende SJI zum Timestamp des Rasters gefunden werden.

4.5 Learning

Wir bemerkten, dass die Tabelle images mit über 2`000`000 Zeilen zum Schreiben etwa 20 Minuten brauchte. Wir erkannten im Nachhinein, dass dies durch den Befehl pandas.to_sql zustande kam, da dieser jede Zeile einzeln geschrieben hatte. Dies hätte durch Definieren der Batchsize verhindert werden können.

5 Umsetzung

Wie im Kapitel 1.2 aufgezeigt, wurde die Webapplikation mittels Django entwickelt. In den nächsten Kapiteln wird auf folgende Punkte in Bezug auf die Django Applikation eingegangen:

- Struktur einer Django Applikation und das Framework
- Das Backend einer Django Applikation in unserem Projekt
- Das Frontend einer Django Applikation in unserem Projekt
- Die Businesslogik, die zwischen Front- und Backend platziert ist in unserem Projekt

5.1 Struktur und Framework der Django Applikation

Das Django Framwork ist ein “looslely coupled” Framework. Unter diesem Begriff wird verstanden, dass die einzelnen Bestandteile des Frameworks eine tiefe Abhängigkeit zueinander haben, wobei Änderungen in einzelnen Bausteinen lediglich eine lokale Auswirkung haben und sich damit die Änderungen global einfacher umsetzen lassen können.

Aus der High-Level Perspektive kann das Django Framework folgendermassen unterteilt werden:

- Backend = Models.py
- Frontend = HTML Templates
- Businesslogik = Views.py

Zur Veranschaulichung wie die einzelnen Elemente zueinander stehen, dient untenstehende Grafik:

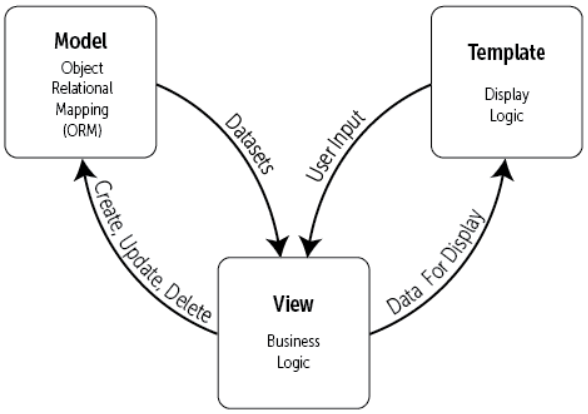

Abbildung 12: Django Framework Quelle: www.djangobook.com

5.2 Backend

Django liefert im Backend (Model) verschiedene Tools, die es ermöglichen, mit der Datenbank zu kommunizieren, mit den Daten zu arbeiten und diese als Datasets der Businesslogik (View) zur Verfügung zu stellen. In diesem Modul befinden sich die Informationen über die Datensets

und wie damit umgegangen werden muss. In der Applikation werden die Verbindungsdetails der Datenbank in den Settings hinterlegt

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'irisv2',
        'USER': 'postgres',
        'PASSWORD': 'lukas4president',
        'HOST': '213.136.68.142',
        'PORT':'5432',
    }
}
```

Abbildung 13: Datenbankzugriff

Jedes Modell widerspiegelt eine Tabelle aus der Datenbank und wird als Python-Klasse definiert. Um die Informationen aus der bereits erstellten Datenbank abzurufen und die Modelle automatisch zu generieren, bietet Django die Möglichkeit ein legacy System mittels integrierter Funktionen nachzubilden. Dazu wurde in der Arbeitsumgebung der Applikation folgender Befehl ausgeführt:

```
python /iris/manage.py inspectdb > iris/models.py
```

Nachdem die Modelle die Datenbank repliziert haben und sich diese im Backend System befinden, wurden lediglich gewisse Voreinstellungen für den Datenabruf definiert. Django bietet die Möglichkeit z.B. die Ordnung der abgerufenen Objekte im Voraus zu definieren oder eine String-Methode zur Beschreibung wie die Objekte im Queryset dargestellt werden.

- Stringmethode für die Ypixel Tabelle:

```
def __str__(self):
    return 'Centroid ID: %s, Step: %s, 1330: %s, 1400:%s, 2796: %s, 2832: %s' % (self.id_observation,
                                                                                 self.step,
                                                                                 self.l_1330,
                                                                                 self.l_1400,
                                                                                 self.l_2796,
                                                                                 self.l_2832
                                                                                 )
```

- Sortiermethode für die CentroidCount Tabelle:

```
class Meta:
    managed = False
    db_table = 'centroid_count'
    ordering = ['id_observation', 'step', 'centroid']
```

5.3 Verbindung von Backend und Frontend mittels View

Als Verbindungsstück zwischen Model und Template ist die View dafür verantwortlich die HTTP Requests der User aus dem Template entgegenzunehmen, zu verarbeiten, wenn nötig Daten aus dem Modell abzurufen und dem User als HTTP Response zurücksenden.

Bei der Verbindung zwischen View und Model kann die View Daten in der Datenbank erstellen, updaten, löschen oder wie im Falle der Iris Applikation rauslesen. Da in der Applikation lediglich das Lesen der Daten verwendet wurde, sind die Abfragen dementsprechend erstellt und eingebettet.

Als User Input gibt es folgende Möglichkeiten:

- Die Nummer des Centroids als <centroid>
- Die ID der Observation als <observation>
- Die Art des Bildes als <image_choice>
- Der Step einer Observation

Die Inputs werden über die URL weitergegeben und als Pattern im File urls.py der Applikation abgespeichert.

```
urlpatterns = [
    path('', views.index, name='index'),
    path('observations/<int:centroid>&<int:observation>&<int:image_choice>&<int:step>', views.list_view, name='list_view'),
]
```

Abbildung 14: URL Patterns

Als Beispiel wird hier der Abruf der Observation List dargestellt.

Um alle Observations eines Centroids zu erhalten wird mittels mehrere Queries eine kombinierte Liste erstellt, welche eine interne ID für die Nachverfolgung in der Datenbank und einen externen Key beinhaltet.

```
## Initial Querrysets in order to load Plots and Graphics
observation_list = CentroidCount.objects.filter(centroid=centroid).order_by('id_observation').values_list('id_observation', flat=True).distinct()
key_list = Observation.objects.filter(id_observation__in=[observation_list]).values_list('observation', flat=True)
zipped_list = zip(observation_list, key_list)
```

Abbildung 15: Querysets für Key und Observation ID

In diesem Query wird im ersten Schritt eine observation_list erstellt, welche alle IDs der Observations umfasst, die den Centroid enthalten. Die Keys der einzelnen Observationen werden über eine zweite Tabelle mit einem Abgleich der vorhandenen Observation List erstellt. Eine Analogie zur SQL Sprache wäre der innere Join der zwei Tabellen mit dem vorgefilterten Attribut der Observation IDs.

Im letzten Schritt werden Tupel als Liste erstellt, so dass diese als Variable an das Template weitergegeben werden kann.

Neben diesem Aufruf gibt es noch einige weitere, dazu gehören:

- x_values : Alle steps wo der Centroid vorkommt
- y_values: Die Häufigkeit, wie oft der Centroid in einem Step vorkommt
- hek_url: Der Verweis auf die HEK Webseite der entsprechenden Observation
- plot: beide Plots werden mittels der mitgegebenen Variablen in der View erstellt
- x_max: als maximum Wert der X-Achse des detaillierten Pots
- key_observation: um den Key für die jeweilige Beobachtung einzeln zu erhalten
- nx, ny: Variabeln um die Images and den Plot anzupassen

5.4 Frontend, Django Template Language

Das Frontend, die so genannten Templates, sind leere Textfiles, die verschiedene, textbasierte Formate generieren können (HTML, XML, CSV etc.). Das Django Framework nutzt dabei eine eigens entwickelte Templatesprache, welche für HTML-erfahrene einfach zu beherrschen sein

soll (für eine detaillierte technische Dokumentation der Sprache kann auf folgendem Link nachgelesen werden).

Für die Applikation wurden folgende Elemente verwendet:

```
context={
        'zipped_list':zipped_list,
        'centroid':centroid,
        'observation':observation,
        'image_choice':image_choice,
        'step':step,
        'plot_graph':plot_graph,
        'key_observation':key_observation,
        'hek_url':hek_url,
        }
```

Abbildung 16: Kontextvariabeln der View

Variablen werden mit doppelt geschweiften Klammern im Template benutzt. Die Variablen werden in der View mittels Kontextvariablen als HTTP Response weitergegeben.

Wie im vorherigen Kapitel erklärt, gibt es Variablen, die in der View berechnet werden, andere werden mittels Input des Users definiert und mitgegeben. Im Gegensatz zu den berechneten Variaben in der View, gibt es Variablen, die über die URL einen neuen Wert zugewiesen erhalten.

Beim Aufrufen der Homepage erhält der User mittels Klicken auf eines der Bilder die Möglichkeit, einen der 52 Centroide auszuwählen.

Abbildung 17: Auswahlmöglichkeiten Homepage

Mittels vordefinierter Liste wird jedem Bild eine Nummer zugewiesen, die für die Konstruktion der URL im href Tag des HTML mitgegeben werden kann

```
{% load static %}
<div class = 'container-fluid'>
{% for num in images_num %}

  {% if forloop.first %}<div class="row">{% endif %}
    <div class="col-sm">
      <a href="/centroid_webapp/observations/{{num}}&0&0&0">
      <img src="{% static 'images/' "" %}{{num}}.png" class='img-fluid' alt='Responsive image'>
      </a>
  </div>
    {% if forloop.counter|divisibleby:4 %}</div><br><div class="row">{% endif %}
    {% if forloop.last %}</div>{% endif %}
  {% endfor %}
```

Abbildung 18: Konstruktion der Gruppenbilder auf der Homepage

Wird beispielsweise auf. die Gruppe eins geklickt, wird folgende URL konstruiert:

http://213.136.68.142/centroid_webapp/observations/1&0&0&0

Für Observation, Image_Choice und Step wird eine Null mitgegeben, da diese noch nicht definiert sind.

Im nächsten Schritt wird der User auf die Liste mit allen möglichen Observations geführt, wo eine Observation ausgewählt werden sollte.

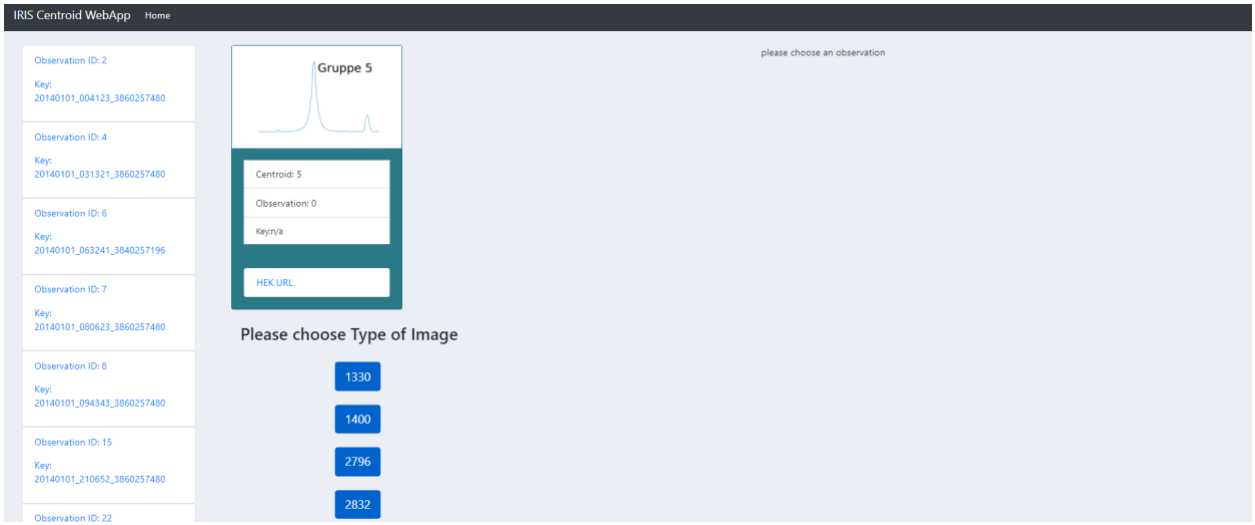

Abbildung 19: Startpunkt in der Observation Liste

Auf nachfolgender Abbildung sind aufgrund fehlender Auswahl noch keine Plots geladen. Sobald auf eine der Observation geklickt wird, wird auch der entsprechende Plot geladen, welcher zum Centroid gehört

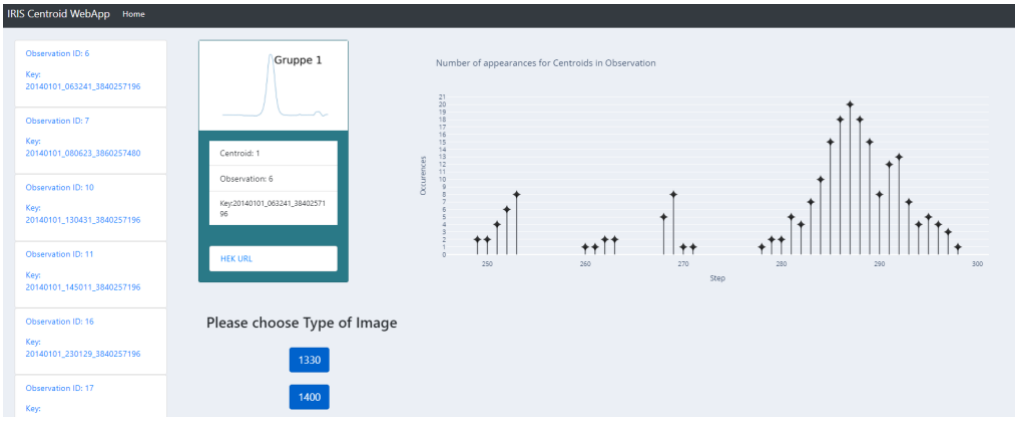

Abbildung 20: Centroid High Level Observation

Für den User ist es nun möglich, einen spezifischen Image Type zu wählen und diesen genauer zu untersuchen. In diesem Beispiel ist der Type 1400 ausgewählt worden, welcher dazu führen soll, den Centroid 1 mit der Observation 6 und dem Image Typ 1400 darzustellen:

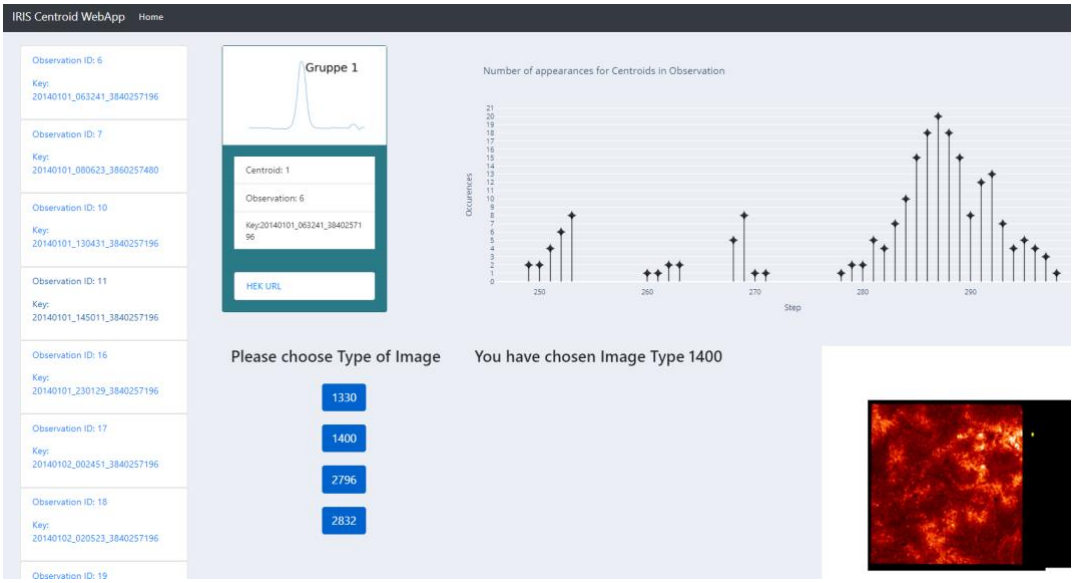

Abbildung 21: Detailplot Centroid1, Observation 6, Type 1400

Damit der Code überschaubar bleibt und der User sauber durch die Applikation geführt werden kann, sind als weitere Elemente der Django Templatesprache Loops und Fallunterscheidungen mittels IF / ELSE eingebaut worden. Als erklärendes Beispiel wird der Prozess herangezogen wie die Observation und deren ID und Key zusammengestellt wird.

```
{% for item in zipped_list %}
  {% if forloop.first %}
    <div class="row">
  {% endif %}
    <div class='col-sm'>
      <a href="{{centroid}}&{{item.0}}&{{image_choice}}"  class='list-group-item'> <p> Observation ID: {{item.0}} </p> <p>Key: {{item.1}}</p></a>
    </div>
    {% if forloop.counter|divisibleby:1 %}
      </div><div class="row">
    {% endif %}
    {% if forloop.last %}
      </div>
    {% endif %}
  {% endfor %}
```

Abbildung 22: Erstellung Observation List in Template

Die <zipped_list> wurde im Query des vorhergehenden Kapitels bereits erklärt und erstellt und wird hier in einem For Schleife verwendet, um alle Observations zu laden, die zum entsprechenden Centroid und dem Image Type gehören. Weiter wird für jedes Item der zipped_list eine neue Zeile kreiert; die entsprechenden Variablen werden ihrerseits als Link hinterlegt. Beim Anwählen des Links werden die entsprechenden Plots geladen.

Schwierigkeiten bei der Erstellung der Plots

Ziel dieses Projekts ist es, die Observations so darzustellen, dass eine Möglichkeit besteht, durch die erstellten Bilder des Satelliten zu iterieren und zu markieren, wo die ausgewählte Observation (nach Step und Centroid) zu finden ist. Für die Erstellung der detaillierten Grafiken wird mit Plotly gearbeitet; dieses stellt eine API zur Verfügung, um mehrschichte Plots zu erstellen und mittels Slider auszuwählen.

Die Logik hinter den detaillierten Plots ist es, für jeden verfügbaren Step, der im Centroid enthalten ist, die entsprechende Grafik zu laden und den Scatterplot zu erstellen.

Die ursprünglich angedachte Lösung dies auch mittels Plotly umzusetzen war jedoch nicht erfolgreich. Die SJI und Ypixel konnten nicht sauber übereinander geplottet werden. Zudem war die Performance der Webseite stark beinträchtigt.

Um die Ladezeiten der Webseite nicht zu stark zu beeinträchtigen, wurde eine zusätzlich Navigation eingebaut, um den entsprechenden Step welcher visualisiert werden soll zu laden. Zudem wurde der Plot mittel Matplotlib generiert.

6 Continuous Integration / Continous Delivery

Die Entwicklung der Webseite wird mittels der folgenden Git Repository getrackt:
https://github.com/eugencuic/FHNW_HS20_IRIS_Centroid_Browser

Für jede Person, welche am Projekt arbeitet, wurde ein eigener Branch erstellt, in welchem er seine Änderungen commited. Demzufolge sieht die Struktur des Repository wie folgt aus:

```
% git branch -a -r
  origin/HEAD -> origin/master
  origin/chris_frame
  origin/eugen_cuic
  origin/lukas_reber
  origin/master
```

Commits in den persönlichen Branches können ohne Approval vorgenommen werden. Um den Master Branch zu aktualisieren, wird anschliessend ein Pull Request für den Master mit dem entsprechend Branch erstellt. Das Repository wurde so konfiguriert, dass sämtliche Pull Requests von zwei Personen (nicht dem Request Owner) approved werden müssen, erst anschliessend ist ein Merge in den Master Branch möglich. Dies hat den Vorteil, dass immer alle drei Projektmitglieder über den aktuellen Stand im Master Branch informiert sind.

Mit Ausnahme der Daten welche sich in der Datenbank befinden sowie den SJI, befinden sich alle Daten im Repository.

6.1 CI

Die Continous Integration (CI) Pipeline wurde mittels GitHub Actions umgesetzt. GitHub Actions erlaubt es, Scripts aufgrund von Events, welche auf dem Repository stattfinden, zu starten.

Die Github Actions für die CI Pipeline beinhaltet die folgenden Schritte:

- Starten von Ubuntu Instanzen mit den Python Versionen 3.6, 3.7, 3.8:
    - Wir arbeiten im Projekt grundsätzlich mit Python 3.8. Damit ist sichergestellt, dass die Webapp auch unter legacy Python Installationen lauffähig ist.

- Für jede Instanz werden alle nötigen Dependencies installiert
    - Die Dependencies befinden sich in der Datei requirements.txt.

- Für jede Instanz wird die Code Formatierung (Linting) mittels Flake8 geprüft
    - Flake8 wird mit den folgenden Optionen ausgeführt:
        - count: Zählen der Anzahl Fehler

        - select: Es werden die folgenden Error/ Violation Codes geprüft: E9,F63,F7,F82
        - show-source: Gibt den fehlerhaften Sourcecode aus
        - statistics: Gibt eine abschliessende Statistik über die Anzahl Fehler pro Fehlercode zurück

- Für jede Instanz werden die definierten Tests mittels pytest durchgeführt.
    - Die Tests sind im Repository unter iris/tests abgelegt
    - Die Tests sind aufgeteilt nach Tests der Views und Tests spezifischer Funktionen
    - Auf eine Berechnung der Testcoverage wurde verzichtet. Da wir Django als Framework verwenden, wurde ein grosser Teil des Codes nicht innerhalb dieses Projekts entwickelt und muss dementsprechend auch nicht hier getestet werden.

Die GitHub Actions Konfiguration für CI ist im Repository unter .github/workflows/ci.yml hinterlegt.

6.2 CD

Analog zur CI Pipeline wurde auch die Continous Delivery Pipeline in GitHub Actions umgesetzt. Die Pipeline wird gestartet, wenn ein Pull Request in den Master freigegeben wurde.

Die GitHub Actions für die CD Pipeline beinhaltet die folgenden Schritte:

- Verbinden mit SSH von GitHub auf den Server, auf welchem die Webseite deployed wird.
    - Hier wurden die entsprechenden Verbindungsdetails (Servername & Port) sowie die nötigen Credentials (Username & Passwort) als Secrets in Repository hinterlegt

- Ausführen eines Bash Scripts, welches auf dem Server abgelegt ist. Das Script übernimmt anschliessend das Deployment. Hierfür wurden die folgenden Schritte ausgeführt:

    - Aktivieren des Virtual Environments, unter welchem die Applikation läuft
    - Mittels git pull die aktuellste Version vom Repository herunterladen
    - Aktuelle Requirements aus requirements.txt installieren
    - Mittels des Befehls collectstatic prüfen, ob neue statische Dateien im Release vorhanden sind und falls ja, diese in das konfigurierte Verzeichnis kopieren
    - Gunicorn und nginx neu starten

Das Script führt folgenden Code aus:

```
#!/bin/bash
```

```
# Activate Virtualenv
. iris_env/bin/activate

# Change into the Django app directory
cd website/FHNW_HS20_IRIS_Centroid_Browser/

# Pull changes from git
git pull

# Apply requirements
pip3 install -r requirements.txt

# update static folder
python3 iris/manage.py collectstatic --noinput

# Restart nginx
systemctl restart nginx

# Restart supervisor which runs gunicorn
supervisorctl restart iris
```

Die GitHub Actions Konfiguration für CI ist im Repository unter .github/workflows/cd.yml hinterlegt.

7 Anhang

7.1 Abbildungsverzeichnis