Missing Data

Tommaso Romano - 941796

June 19, 2022

Contents

1 Introduzione

I dati del mondo reale sono disordinati e spesso contengono molti valori mancanti. Potrebbero esserci più ragioni per i valori mancanti, ma principalmente il motivo della mancanza può essere attribuito a:

- Dati non esistono.
- Dati non registrati per errore umano.
- Dati cancellati accidentalmente.

Di queste ragioni per cui i dati sono mancanti, se ne può aggiungere un'altra che à la causa principale della presenza di missing values nel dataset che andremo ad utilizzare:

- Dati non registrati per malfunzionamento

Infatti il dataset che useremo per questo progetto è relativo ad osservazioni di dati clinici che possono contenere dati mancanti, principalmente per mancata registrazione da parte dell'utente o malfunzionamento degli apparecchi di registrazione dei dati.

1.1 Pyspark

La tecnologia per gestire dataset, non sarà Pandas ma PySpark: PySpark è un'interfaccia per Apache Spark in Python. Non solo consente di scrivere applicazioni Spark usando le API Python, ma fornisce anche la shell PySpark per analizzare interattivamente i dati in un ambiente distribuito. PySpark supporta la maggior parte delle funzionalità di Spark come Spark SQL, DataFrame, Streaming, MLlib (Machine Learning) e Spark Core.

La scelta di utilizzare PySpark nel progetto è dovuta alla necessita di elaborare grosse quantità di dati, e velocemente. Tuttavia, per semplificare, useremo solamente una parte delle osservazione a disposizione, concentrandoci solamente su alcune condizioni cliniche.

Ecco come inizializzare PySpark e creare i DataFrame:

```
[198]  ✓ 0.5s
```

```
app_name = "name"
db_path = "db"
spark = SparkSession.builder.appName(app_name).enableHiveSupport().getOrCreate()
observations = spark.read.format("parquet").load(db_path+'/observations')
conditions = spark.read.format("parquet").load(db_path+'/conditions')
patients = spark.read.format("parquet").load(db_path+'/patients')
questionnaire = spark.read.format("parquet").load(db_path+'/questionnaire_responses')
```

PySpark offre la possibilità di visualizzare il corrente DataFrame come stringa tramite il comando *.show()*:

```
df = conditions
print(df.show())
```

[227] ✓ 1.6s

```
[Stage 565:>                                                        (0 + 1) / 1]

+---+--------------------+------------+------------------+--------------------+-----------------+-------------+--------------------+-------------------------+---
-------------+--------------------+--------------------+--------------------+-----------------------------+-----------------------------+--------------------
----------------+
| id|       onsetDateTime|resourceType| subject_reference|   meta_lastUpdated|      meta_source|meta_versionId|encounter_reference|clinicalStatus_coding_code|cli
code_coding_code| code_coding_display|  code_coding_system|
meta_profile|verificationStatus_coding_code|verificationStatus_coding_system|category_coding_code|category_coding_display|category_coding_system|
+---+--------------------+------------+------------------+--------------------+-----------------+-------------+--------------------+-------------------------+---
-------------+--------------------+--------------------+--------------------+-----------------------------+-----------------------------+--------------------
----------------+
| 23|2021-10-24 18:43:...|   Condition|Patient/1550736443|2021-10-24 18:43:39|#2TKhmJ1wzYgLnCA3|            1|       Encounter/2|                   active|
164971000119101|Diabetes type II ...|http://snomed.inf...|https://140.164.1...|                    confirmed|           http://terminolog...| encounter-diagnos
http://terminolog...|
| 23|2021-10-24 18:43:...|   Condition|Patient/1550736443|2021-10-24 18:43:39|#2TKhmJ1wzYgLnCA3|            1|       Encounter/2|                   active|
164971000119101|Diabetes type II ...|http://snomed.inf...|https://140.164.1...|                    confirmed|           http://terminolog...|           4394010
http://snomed.inf...|
| 24|2021-10-24 18:43:...|   Condition|Patient/1550736443|2021-10-24 18:43:39|#2TKhmJ1wzYgLnCA3|            1|       Encounter/2|                   active|
386806002|  Impaired cognition|http://snomed.inf...|https://140.164.1...|                    confirmed|           http://terminolog...|          439401001|
http://snomed.inf...|
| 24|2021-10-24 18:43:...|   Condition|Patient/1550736443|2021-10-24 18:43:39|#2TKhmJ1wzYgLnCA3|            1|       Encounter/2|                   active|
386806002|  Impaired cognition|http://snomed.inf...|https://140.164.1...|                    confirmed|           http://terminolog...| encounter-diagnosis|
http://terminolog...|
| 25|2021-10-24 18:43:...|   Condition|Patient/1550736443|2021-10-24 18:43:39|#2TKhmJ1wzYgLnCA3|            1|       Encounter/2|                   active|
197480006|    Anxiety disorder|http://snomed.inf...|https://140.164.1...|                    confirmed|           http://terminolog...|          439401001|
http://snomed.inf...|
| 25|2021-10-24 18:43:...|   Condition|Patient/1550736443|2021-10-24 18:43:39|#2TKhmJ1wzYgLnCA3|            1|       Encounter/2|                   active|
197480006|    Anxiety disorder|http://snomed.inf...|https://140.164.1...|                    confirmed|           http://terminolog...| encounter-diagnosis|
http://terminolog...|
```

Tuttavia dovuto alla quantità di colonne e l'utilizzo di jupyter nootebok, la visualizzazione tramite stringa, viene distorta ed a fatica comprensibile. Pertanto, dato il ridotto numero di osservazioni, trasformeremo il DataFrame di PySpark in Pandas per visualizzare meglio i risultati:

```
df = conditions
pd.DataFrame(df.toPandas()).head()
```

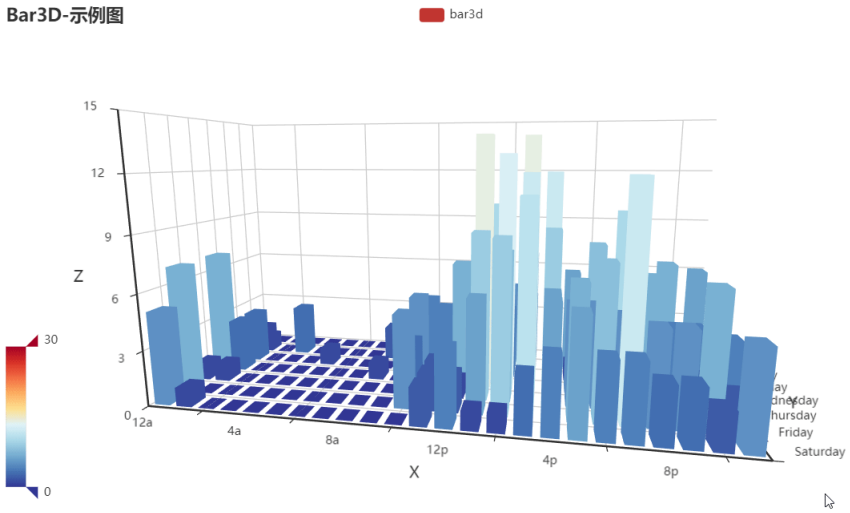
[228] ✓ 0.4s

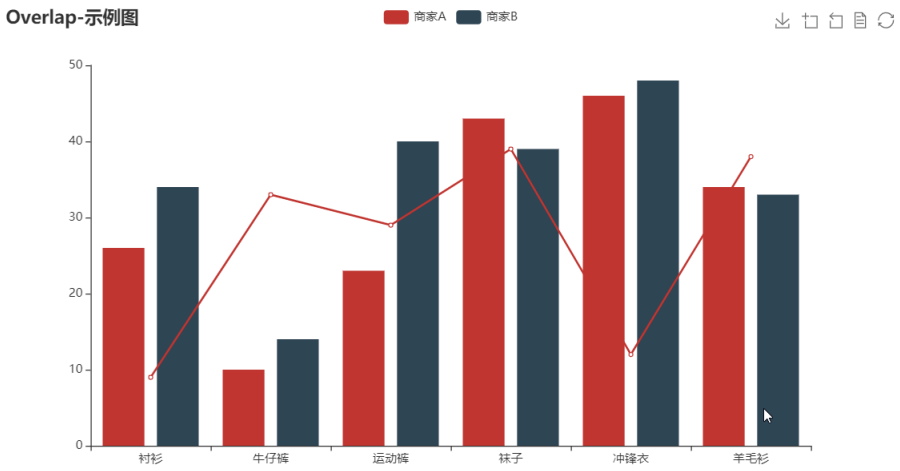
	id	onsetDateTime	resourceType	subject_reference	meta_lastUpdated	meta_source	meta_versionId	encounter_reference	clinicalStatus_coding_code	
0	23	2021-10-24 18:43:39.942	Condition	Patient/1550736443	2021-10-24 18:43:39	#2TKhmJ1wzYgLnCA3	1	Encounter/2	active	http://terminolog
1	23	2021-10-24 18:43:39.942	Condition	Patient/1550736443	2021-10-24 18:43:39	#2TKhmJ1wzYgLnCA3	1	Encounter/2	active	http://terminolog
2	24	2021-10-24 18:43:39.944	Condition	Patient/1550736443	2021-10-24 18:43:39	#2TKhmJ1wzYgLnCA3	1	Encounter/2	active	http://terminolog
3	24	2021-10-24 18:43:39.944	Condition	Patient/1550736443	2021-10-24 18:43:39	#2TKhmJ1wzYgLnCA3	1	Encounter/2	active	http://terminolog
4	25	2021-10-24 18:43:39.946	Condition	Patient/1550736443	2021-10-24 18:43:39	#2TKhmJ1wzYgLnCA3	1	Encounter/2	active	http://terminolog

1.2 ECharts

Un'altra tecnologia che useremo in questo progetto è ECharts di Apache: è uno strumento di visualizzazione JavaScript open source, che può essere eseguito in modo fluido su PC e dispositivi mobili. Infatti, a differenza dei grafici statici di matplot, ECharts offre una soluzione dinamica ed interattiva, oltre che ad una gamma più ampia e maggiormente personalizzabile di grafici.

Per funzionare su Python useremo la libreria *pyecharts*. Una galleria dinamica di esempio ufficiale di Apache si trova al seguente link, mentre delle gif di esempio di cui ho riportato sotto un fermo immagine, si trovano al seguente link.





1.3 Conditions

La prima cosa utile da fare, è utilizzare il dataset delle *conditions* per identificare il numero di pazienti che sono seguiti per una data condizione clinica. Il DataFrame contiene 19 colonne che sono *'id', 'onsetDateTime', 'resourceType', 'subject_reference', 'meta_lastUpdated', 'meta_source', 'meta_versionId', 'encounter_reference', 'clinicalStatus_coding_code', 'clinicalStatus_coding_system', 'code_coding_code', 'code_coding_display', 'code_coding_system', 'meta_profile', 'verificationStatus_coding_code', 'verificationStatus_coding_system', 'category_coding_code', 'category_coding_display','category_coding_system'* e ci serviranno solamente *clinicalStatus_coding_code, code_coding_display* in modo tale da filtrare per pazienti marcati come *active* e tramite PySpark, come per SQL raggrupparli tramite *.groupBy()* per nome della condizione, conteggiarli ed ordinarli per in modo no ascendente:

```
df = df.filter(df['clinicalStatus_coding_code'] == 'active')
df = df.groupBy('code_coding_display').count()
df = df.orderBy('count',ascending=False)
pd.DataFrame(df.toPandas()).head()
```

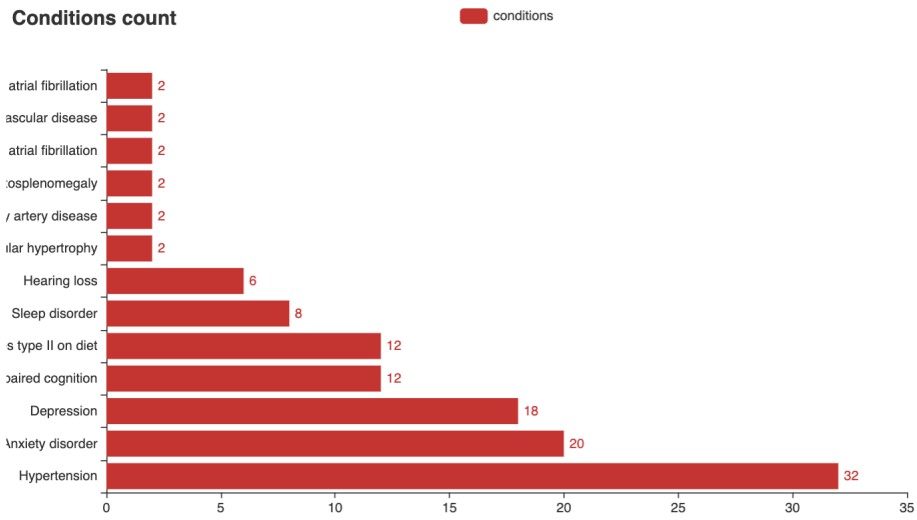
[229] ✓ 0.7s

	code_coding_display	count
0	Hypertension	32
1	Anxiety disorder	20
2	Depression	18
3	Impaired cognition	12
4	Diabetes type II on diet	12

Il secondo step è visualizzare i dati raccolti. Sicuramente il modo più facile per organizzare i dati categorici raccolti è tramite un grafico a barre. Tramite ECharts creo il grafico dinamico che è possibile scaricare al seguente link:

```
bar = Bar()
bar.add_xaxis(df_column_to_list(df,'code_coding_display'))
bar.add_yaxis('conditions', df_column_to_list(df,'count')
    # scale=True
    )
bar.reversal_axis()
bar.set_series_opts(label_opts=opts.LabelOpts(position='right'))
bar.set_global_opts(
        title_opts=opts.TitleOpts(title="Conditions count")
        )
bar.render_notebook()
```

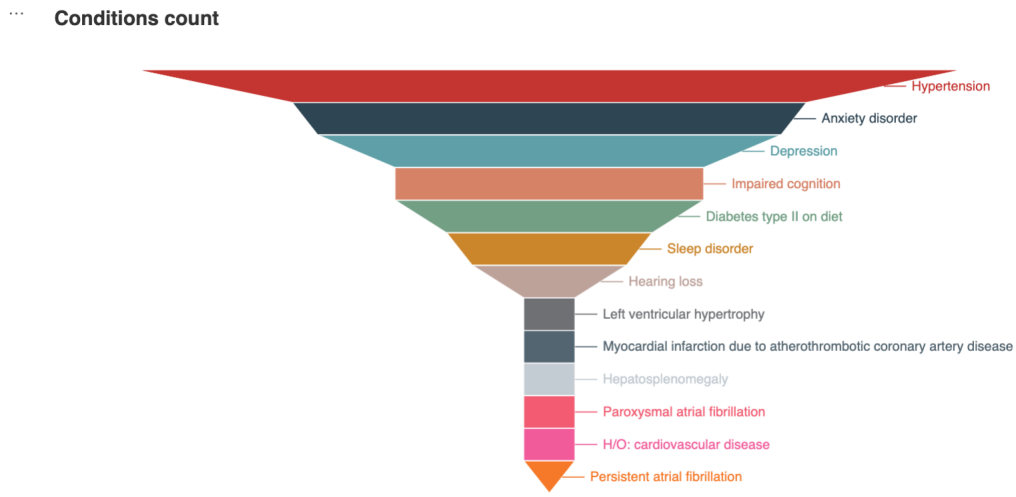
[406] ✓ 0.8s



Un altro modo per visualizzare e comparare i dati categorici, simile al grafico a barre ma più originale è il Funnel al seguente link:

```
df = pd.DataFrame(df.toPandas())

fu = Funnel()
fu.add("conditions", [list(z) for z in zip(df['code_coding_display'], df['count'])])
fu.set_global_opts(title_opts=opts.TitleOpts(title="Conditions count"),
    legend_opts=opts.LegendOpts(is_show=False))
fu.render_notebook()
```

[271] ✓ 0.6s



2 Missing data

Terminata la visualizzazione delle condizioni esistenti dei pazienti, ci occupiamo della visualizzazione delle osservazioni delle diverse condizioni cliniche. Il DatFrame che utilizzeremo contiene 63 colonne, ovvero: *effectiveDateTime', 'id', 'resourceType', 'status', 'valueInteger', 'valueQuantity_code', 'valueQuantity_system', 'valueQuantity_unit', 'valueQuantity_value', 'subject_reference', 'meta_lastUpdated', 'meta_source', 'meta_versionId', 'encounter_reference', 'bodySite_coding_code', 'bodySite_coding_display', 'bodySite_coding_system', 'code_coding_code', 'code_coding_display', 'code_coding_system', 'meta_profile', 'valueCodeableConcept_coding_code', 'valueCodeableConcept_coding_display', 'valueCodeableConcept_coding_system', 'component_valueQuantity_code', 'component_valueQuantity_system', 'component_valueQuantity_unit', 'component_valueQuantity_value', 'component_code_coding_code', 'component_code_coding_display', 'component_code_coding_system', 'valueBoolean', 'valueSampledData_data', 'valueSampledData_dimensions', 'valueSampledData_lowerLimit', 'valueSampledData_period', 'valueSampledData_upperLimit', 'valueSampledData_origin_code', 'valueSampledData_origin_system', 'valueSampledData_origin_unit', 'valueSampledData_origin_value', 'effectivePeriod_end', 'effectivePeriod_start', 'code_text', 'category_coding_code', 'categor_coding_display', 'category_coding_system', 'identifier_system', 'identifier_value', 'extension_url', 'extension_valueCodeableConcept_coding_code', 'extension_valueCodeableConcept_coding_display', 'extension_valueCodeableConcept_coding_system',*

'component_valueSampledData_data', 'component_valueSampledData_dimensions', 'component_valueSampledData_period', 'component_valueSampledData_origin_code', 'component_valueSampledData_or
'component_valueSampledData_origin_unit', 'component_code_text', 'component_code_coding_version',
'component_valueSampledData_extension_url', 'component_valueSampledData_extension_valueString'.

```
df = observations
pd.DataFrame(df.toPandas()).head()
```

[408] ✓ 15.4s

	effectiveDateTime	id	resourceType	status	valueInteger	valueQuantity_code	valueQuantity_system	valueQuantity_unit	valueQuantity_value	subject_reference	...	component_valueSampledDa
0	2022-02-12 09:24:14	27654	Observation	final	NaN	None	None	None	NaN	Patient/512815964	...	
1	2022-02-12 09:24:14	27654	Observation	final	NaN	None	None	None	NaN	Patient/512815964	...	
2	2022-02-12 09:24:14	27654	Observation	final	NaN	None	None	None	NaN	Patient/512815964	...	
3	2022-02-12 09:24:14	27654	Observation	final	NaN	None	None	None	NaN	Patient/512815964	...	72.48 69.96 67.13 62.16 68.7 65.9
4	2022-02-12 09:24:14	27654	Observation	final	NaN	None	None	None	NaN	Patient/512815964	...	3.23 3.23 4.35 2.63 3.70 2.17 3.

5 rows × 63 columns

Di questi 63 attributi, ne useremo solamente alcuni per determinati tipi di condizioni cliniche e pazienti. Infatti, molti di questi contengono molti se non tutti valori nulli o mancanti. Pertanto, una metodo alternativo e visivo per identificare missing values è tramite làutilizzo di grafici come quello a barre, delle matrici o heatmap.

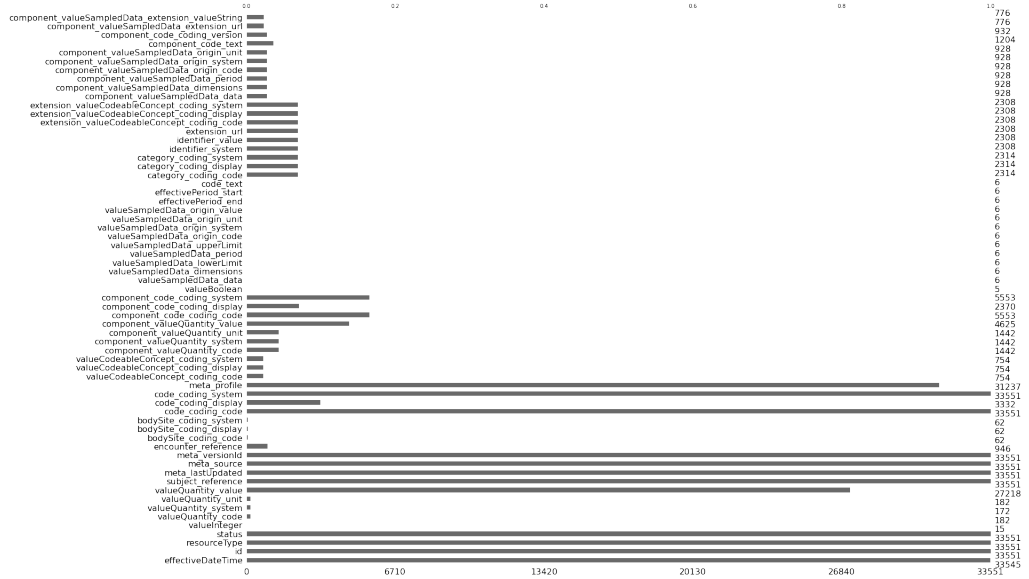
2.1 Missingno

Una libreria utile a visualizzare i dati macanti è sicurant *Missigno*. Infatti, contiene 3 grafici particolari (bars, matrix e heatmap) che consentono di visualizzare facilmente ed efficaciemente DataFrame con la presenza di dati mancanti, e capire i motivi per cui i dati non sono presenti.

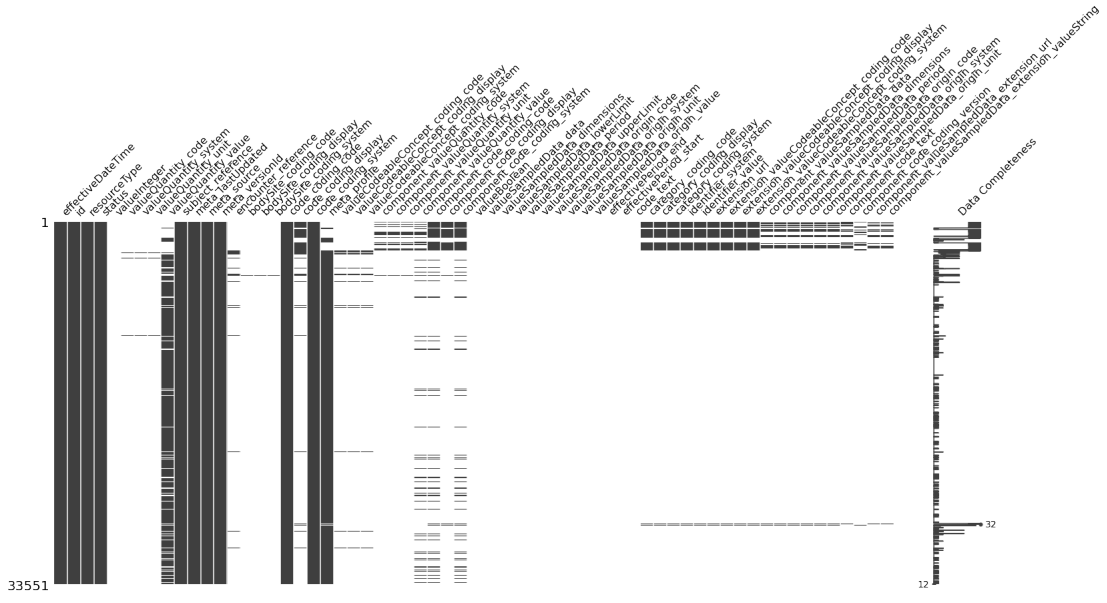
```
msno.bar(pd.DataFrame(observations.toPandas()))
msno.matrix(pd.DataFrame(observations.toPandas()),labels=True)
msno.heatmap(pd.DataFrame(observations.toPandas()))
```

[486] ✓ 12.4s

Il più facile grafico è sicuramente quello a barre che evidenzia per ogni attributo la quantità di dati presenti rispetto al totale delle osservazioni. Notare come nel DatFrame utilizzato, molti attributi contengano 0 valori e altri solamente una frazione delle osservazioni totali.

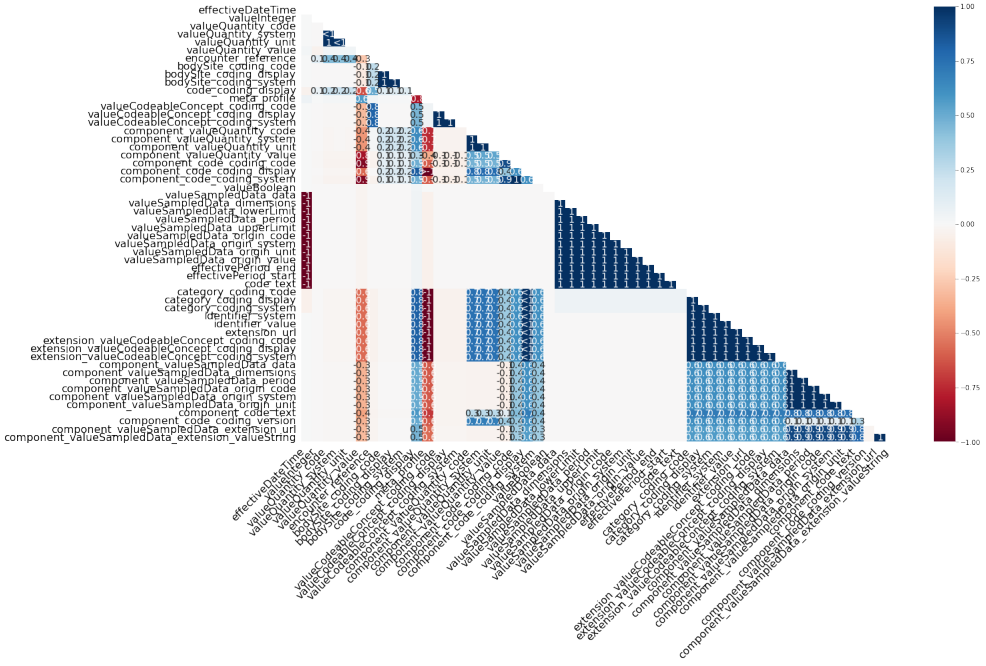


La matrice qui sotto invece, aggiunge informazioni molto importanti rispetto al grafico precedente: permette ti analizzare visivamente dove (ovvero in qual riga del DataFrame) i dati sono mancanti. Infatti, presenta un ulteriore strumento alla sua destra che è il *Data Completeness* che descrive in una data riga quanti sono i dati presenti rispetto a quelli mancanti.



Infine la heatmap permette di visualizzare un'altro elemento importante: la correlazione

tra i diversi attributi. In particolare, consente di identificare motivi di missing values relativi alla relazione tra due attributi.



A causa dell'elevato numero di attributi nel DataFrame, nei grafici precedenti, la visualizzazione risulta perlopiù confusionaria, occorre quindi ridurre il numero di colonne, eliminando ad esempio quelle che contengono zero valori, oppure quelle che ne contengono sotto una percentuale fornita dall'utente. Successivamente a questa modifica, i grafici saranno sicuramente più di facile comprensione.

2.2 Filter

A causa dell'elevato numero di attributi nel DataFrame, nei grafici precedenti, la visualizzazione risulta perlopiù confusionaria, occorre quindi ridurre il numero di colonne, eliminando ad esempio quelle contenenti zero osservazioni o eliminando quelle che contengono un numero di osservazioni sotto una certa soglia. Infatti molti attributi del dataset che stiamo utilizzando, sono stati inseriti per osservazioni future o per misurazioni non più di alcun utilizzo.

Pertanto, per l'analisi di dati che svilupperemo successivamente, utilizzeremo solamente 6 attributi: *subject_reference, effectiveDateTime, code_coding_code, component_code_coding_code, valueQuantity_value, component_valueQuantity_value.* Questi attributi corrispondono rispettivamente a: id del paziente esaminato, il timestamp della registrazione della misurazione, il

codice genitore dell'osservazione, il codice foglio dell'osservazione, il valore legato al codice padre dell'osservazione, il valore legato al codice figlio della misurazione.

Per filtrare efficacemente le osservazioni, creo queste due funzioni:

```
def filter(
    df:pyspark.sql.dataframe.DataFrame,
    **kwargs
)->pyspark.sql.dataframe.DataFrame:
    '''
    A Shortcut for filtering by values , null and None
    '''
    for k in kwargs.keys():
        if kwargs[k] == None:
            return df
        else:
            if kwargs[k] is not list:
                df = df.filter(F.col(k).contains(kwargs[k]))
            else:
                df = df.filter(F.col(k).isin(kwargs[k]))
    return df
```

[412] ✓ 0.1s

La funzione atomica/generale *filter (df, **kwargs)* permette di filtrare da un qualsiasi DataFrame PySpark in base ad una qualsiasi colonna ed un valore o una lista di valori.

```
def filter_observations(patients, codes, comp_codes):
    df = select_observations()
    if patients:
        if patients is not list:
            if '/' not in patients:
                patients = ('Patient/' + str(patients))
        else:
            new_p = []
            for p in patients:
                if '/' not in p:
                    new_p.append('Patient/' + str(p))
                else:
                    new_p.append(p)
            patients = new_p
    df = filter(df, subject_reference=patients)
    df = filter(df, code_coding_code=codes)
    df = filter(df, component_code_coding_code=comp_codes)
    return df
```

[413] ✓ 0.3s

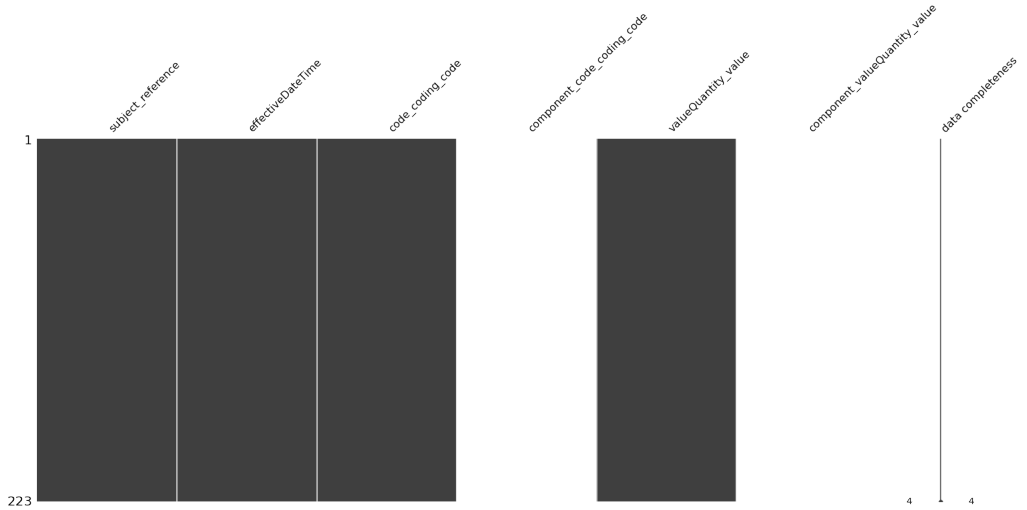
La funzioni *filter_observations (patients, codes, comp_codes)* è invece specifica per questo dataset per velocizzare molti pattern, ed utilizza la funzione atomica/generale filter insieme ad una di select delle colonne utili per le osservazioni che useremo.

```
df = filter_observations('621892226','55425-3',None)
pd.DataFrame(df.toPandas()).head()
```

[414] ✓ 0.6s

	subject_reference	effectiveDateTime	code_coding_code	component_code_coding_code	valueQuantity_value	component_valueQuantity_value
0	Patient/621892226	2022-01-06 01:00:00	55425-3	None	78.0	NaN
1	Patient/621892226	2022-01-05 01:00:00	55425-3	None	79.0	NaN
2	Patient/621892226	2022-01-04 01:00:00	55425-3	None	74.0	NaN
3	Patient/621892226	2022-01-03 01:00:00	55425-3	None	80.0	NaN
4	Patient/621892226	2022-01-02 01:00:00	55425-3	None	78.0	NaN

Grazie a queste funzioni possiamo ora visualizzare più efficacemente il grafico matrix per determinare la presenza o assenza di missing values per il paziente *621892226* e condizione *55425-3* (ovvero Heart Beats) che saranno i nostri dati di riferimento per le prossime osservazioni.



Come si può vedere dal grafico, i dati sono competi e non sono presenti dati mancanti. Infatti, i dati legati agli attributi component* non sono necessari per le osservazioni di Heart Beats. Ma siamo sicuri che non ci sono dati mancanti?

3 Imputation

Nelle precedenti analisi del DataFrame per identificare i missing values ci siamo occupati di setacciare solamente la relazione tra il codice della condizione con il codice di un paziente per trovare i valori. Tuttavia se volessimo creare un grafico che visualizza la media del battito cardiaco di giorno in giorno, potremmo ritrovare un'altra categoria di missing values che sono legati alla time sieries

3.1 Time Series

Utilizziamo il DataFrame creato nei punti precedenti ed ordiniamolo secondo il timestamp della rilevazione. Ci possiamo facilmente accorgere che tra il 2021-11-15 e 2021-12-10 non ci sono state rilevazioni del battito cardiaco. Infatti, quando ci si occupa di rilevazioni che non sono fatte automaticamente da un software automatico ma fidandosi della costanza di un utente, si possono trovare numerosi dati mancanti legati all agli archi temporali.

```
pd.DataFrame(df.toPandas()).sort_values(by=['effectiveDateTime']).head(10)
```
[415] ✓ 0.4s

	subject_reference	effectiveDateTime	code_coding_code	component_code_coding_code	valueQuantity_value	component_valueQuantity_value
104	Patient/621892226	2021-11-10 01:00:00	55425-3	None	88.0	NaN
108	Patient/621892226	2021-11-11 01:00:00	55425-3	None	88.0	NaN
107	Patient/621892226	2021-11-12 01:00:00	55425-3	None	88.0	NaN
109	Patient/621892226	2021-11-13 01:00:00	55425-3	None	88.0	NaN
106	Patient/621892226	2021-11-14 01:00:00	55425-3	None	88.0	NaN
105	Patient/621892226	2021-11-15 01:00:00	55425-3	None	88.0	NaN
115	Patient/621892226	2021-12-10 01:00:00	55425-3	None	91.0	NaN
110	Patient/621892226	2021-12-10 01:00:00	55425-3	None	77.0	NaN
114	Patient/621892226	2021-12-11 01:00:00	55425-3	None	83.0	NaN
111	Patient/621892226	2021-12-11 01:00:00	55425-3	None	83.0	NaN

Nel DataFrame precedente sono inoltre presenti più rilevazioni al giorno (ad esempio 2021-12-11) e se volessimo visualizzare un grafico giorno per giorno ci troveremmo giorni duplicati. Occorre quindi creare un funzione che raggruppa i dati temporali per una quantità che possiamo definire noi (come per ore, giorni, settimane o mesi). Utilizziamo una funzione già esistente di PySpark (*date_trunc*) e successivamente raggruppiamo per l'intervallo di tempo selezionato con quelle colonne che sono immutabili alle osservazioni. Infine processiamo i gruppi creati con una funzione semplice come la media.

```
def trunc_time(
    df:pyspark.sql.dataframe.DataFrame,
    time_col,
    groupby_cols:list,
    agg_dict,
    time='day'):
    '''
    - time: hour, day, week
    - agg_dict: { \'col1\' : \'avg\' }
    '''
    assert time in ['hour','day','week']
    new_time_col = 'new_' + time_col
    df = df.withColumn(new_time_col, F.date_trunc(time, time_col))
    if time_col not in groupby_cols: groupby_cols.append(new_time_col)
    df = df.groupBy(groupby_cols).agg(agg_dict)
    df = df.orderBy(F.col(new_time_col).asc())
    return df
```
[416] ✓ 0.5s

```
df1 = trunc_time(df, TIME, [CODE], {VALUE:'avg'})
pd.DataFrame(df1.toPandas()).head(10)
```
[578] ✓ 1.5s

	code_coding_code	new_effectiveDateTime	avg(valueQuantity_value)
0	55425-3	2021-11-10	88.0
1	55425-3	2021-11-11	88.0
2	55425-3	2021-11-12	88.0
3	55425-3	2021-11-13	88.0
4	55425-3	2021-11-14	88.0
5	55425-3	2021-11-15	88.0
6	55425-3	2021-12-10	84.0
7	55425-3	2021-12-11	83.0
8	55425-3	2021-12-12	90.0
9	55425-3	2021-12-13	85.0

Il DataFrame ottenuto non contiene intervalli di tempi duplicati per ora, giorno, settimana o mese. Tuttavia, non è ancora risolta la presenza di timestamp mancanti come

quelli tra 2021-11-15 e 2021-12-10. Creiamo quindi la funzione *fill_missing_time_values (df, ...)* che in base all'intervallo di tempo selezionato aggiunge righe di dati vuoti inserendo i timestamp mancanti. Nel dataset precedente quindi inserirebbe le date dal 2021-11-14 al 2021-12-09 con None or NaN a seconda dei tipi di attributi diversi dalle time values.

```
def fill_missing_time_values(
    df:pyspark.sql.dataframe.DataFrame,
    time_col,
    groupby_cols,
    agg_dict,
    time='day'):
    '''
    - time: hour, day, week
    '''
    df = trunc_time(df, time_col, groupby_cols, agg_dict, time)
    new_time_col = 'new_' + time_col
    new_time_col_indx = 0
    fill_row = []
    for i in range(0,len(df.columns)):
        fill_row.append(None)
        if df.columns[i] == new_time_col:
            new_time_col_indx = i
    tm = df_column_to_list(df, new_time_col)
    new_schema = df.schema
    for f in new_schema.fields:
        f.nullable = True
    df = spark.createDataFrame(df.collect(), schema=new_schema)
    if time == 'hours':
        plus = datetime.timedelta(hours=1)
    elif time == 'day':
        plus = datetime.timedelta(days=1)
    elif time == 'week':
        plus = datetime.timedelta(weeks=1)
    prev = tm[0]
    for t in tm:
        dif = t - prev
        if dif > plus:
            fix = prev
            while (t - fix) != plus:
                fix += plus
                row_to_add = fill_row.copy()
                row_to_add[new_time_col_indx] = datetime.datetime(fix.year, fix.month, fix.day,
                newRow = spark.createDataFrame([row_to_add], schema=new_schema)
                df = df.union(newRow)
            prev += dif
        elif dif == plus:
            prev += dif
    df = df.orderBy(F.col(new_time_col).asc())
    return df
```

[451] ✓ 0.3s

```
df2 = fill_missing_time_values(df, TIME, [CODE], {VALUE:'avg'})
pd.DataFrame(df2.toPandas()).head(10)
```

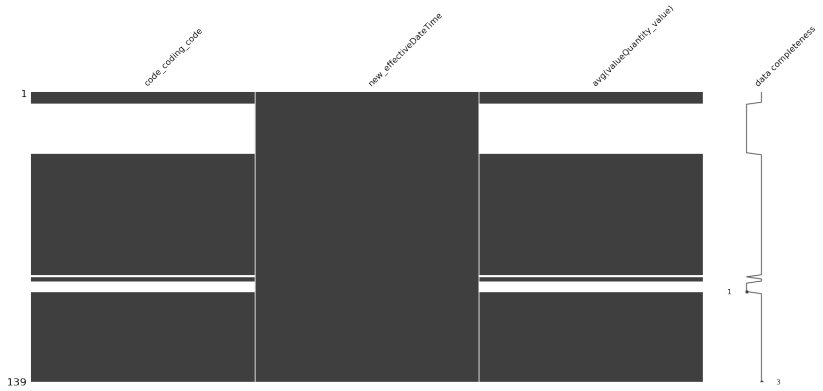
[612] ✓ 5.9s

...

</>

	code_coding_code	new_effectiveDateTime	avg(valueQuantity_value)
0	55425-3	2021-11-10	88.0
1	55425-3	2021-11-11	88.0
2	55425-3	2021-11-12	88.0
3	55425-3	2021-11-13	88.0
4	55425-3	2021-11-14	88.0
5	55425-3	2021-11-15	88.0
6	None	2021-11-16	NaN
7	None	2021-11-17	NaN
8	None	2021-11-18	NaN
9	None	2021-11-19	NaN

Il dataset ottenuto risolve il problema di timestamp mancanti aggiungendo inoltre valori nulli per tutti gli altri attributi. Se ora creiamo il grafico a matrice dei dati mancanti, otteniamo spazi vuoti in corrispondenza dei timestamp aggiunti dalle precedenti funzioni.



3.2 Simple Imputer

Per visualizzare il battito cardiaco di giorno per giorno o settimana per settimana è sicuramente opportuno usare un semplice grafico come quello a linea. Lo costruiamo tramite Echarts che permette di aggiungere altri indicatori come *MarkLine* e *MarkPoint* che possono essere usati per segnare dati sul grafico come massimo minimo e media.

```
def heartbeat_graph(x, y, title):
    hb_line = Line()
    hb_line.add_xaxis(x)
    hb_line.add_yaxis(title, y,
        label_opts=opts.LabelOpts(is_show=False),
        markline_opts=opts.MarkLineOpts(data=[opts.MarkLineItem(type_='average')]),
        markpoint_opts=opts.MarkPointOpts(data=[opts.MarkPointItem(type_='min'), opts.MarkPointItem(type_='max')])
        # scale=True
        )
    hb_line.set_series_opts()
    hb_line.set_global_opts(
            title_opts=opts.TitleOpts(title=title),
            datazoom_opts=opts.DataZoomOpts(),
            xaxis_opts=opts.AxisOpts(splitline_opts=opts.SplitLineOpts(is_show=False)),
            yaxis_opts=opts.AxisOpts(
                axistick_opts=opts.AxisTickOpts(is_show=True),
                splitline_opts=opts.SplitLineOpts(is_show=True),
                min_=70,
                max_=100
            )
            )
    return hb_line
```

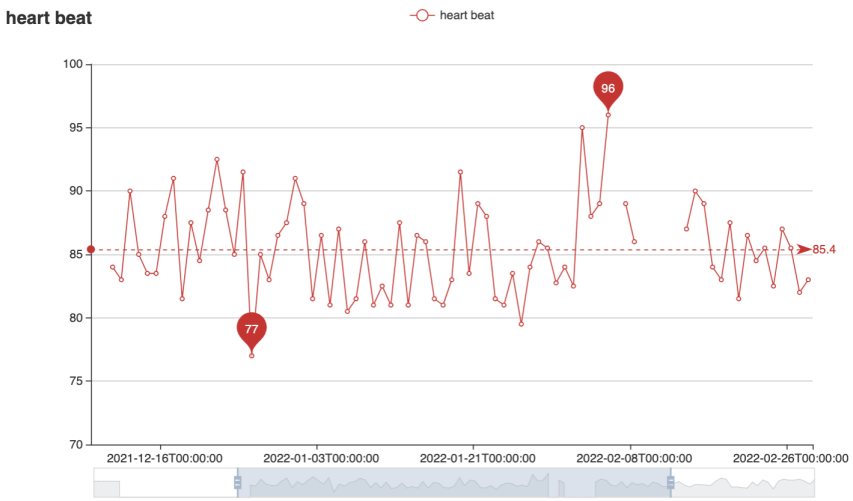
[619] ✓ 0.7s

```
df3 = filter_observations('621892226','55425-3',None)
df3 = fill_missing_time_values(df3, TIME, [CODE], {VALUE:'avg'})
x = df_column_to_list(df3, 'new_effectiveDateTime')
y = df_column_to_list(df3, 'avg('+VALUE+')')
bar = heartbeat_graph(x, y, 'heart beat')
bar.render_notebook()
```

[620] ✓ 9.7s

...

</>



Dal grafico ottenuto si possono facilmente vedere buchi legati alla mancanza dei dati. Ultimo passo è riempire i dati mancanti per i timestamp aggiunti in precedenza. Ci sono diverse tecniche di imputazione. Le più semplici sono sicuramente con il riempimento tramite costante, media, mediana.

```
from sklearn.impute import SimpleImputer

def __fill_missing_with_imputer__(df:pyspark.sql.dataframe.DataFrame,
    columns, pdf:pd.DataFrame, imputer)->pyspark.sql.dataframe.DataFrame:
    if type(columns) is str:
        pdf[columns] = imputer.fit_transform(pdf[[columns]])
    elif type(columns) is list:
        for c in columns:
            pdf[c] = imputer.fit_transform(pdf[[c]])
    return spark.createDataFrame(pdf, schema=df.schema)

def fill_missing_with_mean(df:pyspark.sql.dataframe.DataFrame,
    columns)->pyspark.sql.dataframe.DataFrame:
    '''
    - coloumns

    see https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html
    '''

    pdf = (pd.DataFrame(df.toPandas()))
    imputer = SimpleImputer(strategy='mean')
    return __fill_missing_with_imputer__(df,columns,pdf,imputer)
```

[621] ✓ 0.2s

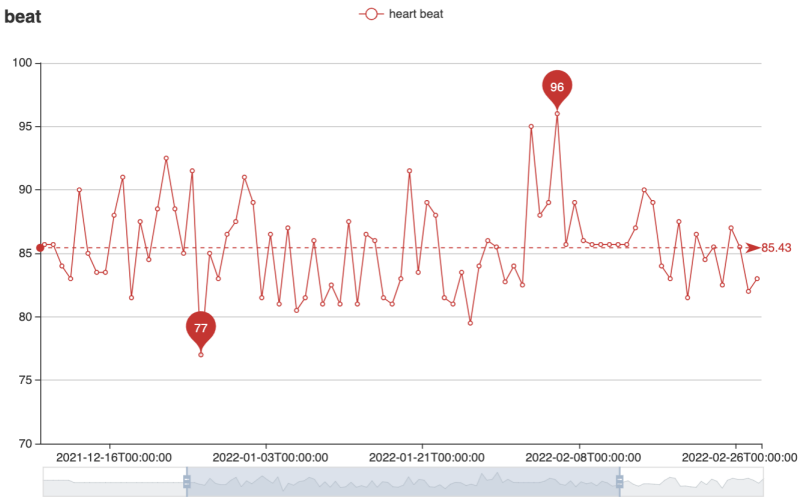
Abbiamo usato una libreria come quella di *Sklearn* che contiene funzionalità come

quella di *SimpleImputer* per creare una semplice funzione che riempie i dati mancanti di una qualsiasi colonna di un DataFrame PySpark.

```
df4 = fill_missing_with_mean(df3, 'avg('+VALUE+')')

x = df_column_to_list(df4, 'new_effectiveDateTime')
y = df_column_to_list(df4, 'avg('+VALUE+')')
bar = heartbeat_graph(x, y, 'heart beat')
bar.render_notebook()
```

[622] ✓ 5.4s

...



Il grafico risultante non contiene più valori mancanti e quindi spazi vuoti. Vi sono anche tecniche gradualmente più complesse per ricavare i dati mancanti partendo da quelli presenti, come Linear Regression, KNN, MICE, e altre.

3.3 Extensions

Grazie alle funzioni precedentemente descritte possiamo visualizzare l'andamento del battito cardiaco settimana per settimana e creare una gif animata della sua progressione nel tempo. Utilizzando una libreria come *imageio* raggruppiamo tutti i grafici dell'oscillazione per ogni settimana ottenendo una gif che mostra la progressione del battito cardiaco per un totale di 20 settimane.

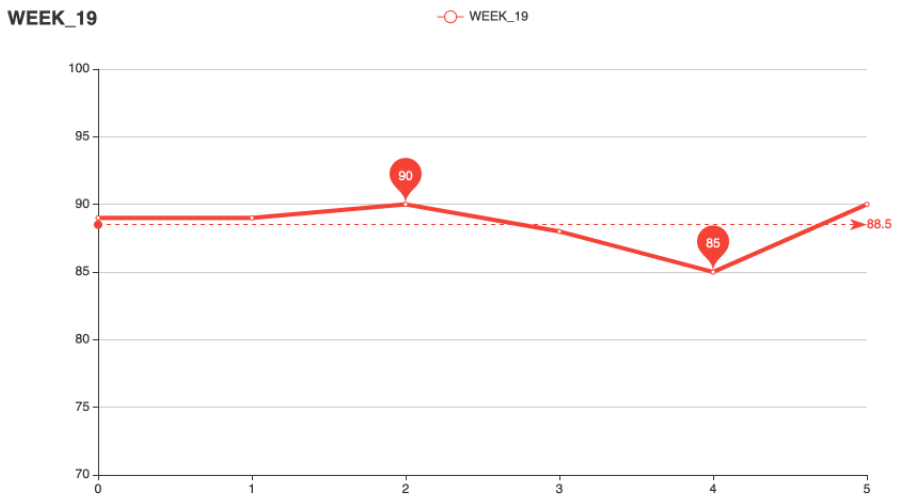
```python
import os
import imageio
from PIL import Image

png_dir = 'renders/weekly/'
images = []
for file_name in sorted(os.listdir(png_dir)):
    prev = None
    if file_name.endswith('.png'):
        file_path = os.path.join(png_dir, file_name)
        images.append(imageio.imread(file_path))
        prev = file_path

imageio.mimsave('renders/weekly/HB_MOVE.gif', images)
print("Done")
```

[45] ✓ 1.3s

```
Done
```

```python
for i in range(0, len(df4), 7):
    week = df4.iloc[i:i+7]
    x = range(0, len(week))
    y = week['avg('+VALUE+')']
    bar = heartbeat_graph(x, y, 'WEEK_'+str(int(i/7)))
    bar.render('renders/weekly/sum/week_'+str(int(i/7))+'.html')
```

[44] ✓ 0.2s



Un'altra estensione per questa visualizzazione animata può essere quella di salvare e visualizzare anche le settimane precedenti in modo tale da confrontarle mano a mano che il tempo passa. Il grafico ottenuto ad ogni frame conterrà in grigio sbiadito le settimane precedenti mentre in rilievo la settimana corrente.

```
prev = df4.iloc[0:7]
prevs = [prev['avg('+VALUE+')']]
for i in range(7, len(df4), 7):
    week = df4.iloc[i:i+7]
    x = range(0, len(week))
    prevs.append(week['avg('+VALUE+')'])
    y = prevs
    bar = heartbeat_graph(x, y, 'WEEK_'+str(int(i/7)))
    bar.render('renders/weekly/sum/week_'+str(int(i/7))+'.html')
```

[81] ✓ 0.2s