




# A3 Cohort Building

Welcome to A3! Please enter answers to the questions in the specified Markdown cells below, and complete the code snippets in the associated python files as specified. When you are done with the assignment, follow the instructions at the end of this assignment to submit.

## Learning Objective

In this assignment, you will gain experience extracting and transforming **clinical data** into datasets for downstream statistical analysis. You will practice using the **Python**  programming language and common time-saving tools in the **Pandas**  library that are ideally suited to these tasks.

## Resources

- Pandas Cheat Sheet : [https://pandas.pydata.org/Pandas\\_Cheat\\_Sheet.pdf](https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf)
- Relevant publications:
  - We will be following along with the cohort building process presented in "[A targeted real-time early warning score \(TREWScore\) for septic shock](#)" by Henry et al. published in Science Translational Medicine in 2015.
  - We will also be referring to "[Epidemiology of severe sepsis in the United States: Analysis of incidence, outcome, and associated costs of care](#)" by Angus et al.

## Environment Set-Up

To begin, we will need to set up an virtual environment with the necessary packages. A virtual environment is a self-contained directory that contains a Python interpreter (aka Python installation) and any additional packages/modules that are required for a specific project. It allows you to isolate your project's dependencies from other projects that may have different versions or requirements of the same packages.

For BIOMEDIN 215 Python assignments, we require that you utilize [Miniconda](#) to manage your virtual environments. Miniconda is a lightweight version of [Anaconda](#), a popular Python distribution that comes with many of the packages that are commonly used in data science.

## Instructions for setting up your environment using Miniconda:

1. If you do not already have Miniconda installed, download and install the latest version for your operating system from the following link:  
<https://docs.conda.io/en/latest/miniconda.html#latest-miniconda-installer-links>
2. Create a new virtual environment for this assignment by running the following command in your terminal:

```
conda env create -f environment.yml
```

This will create a new virtual environment called `biomedin215` with Python version 3.10

3. Activate your new virtual environment by running the following command in your terminal:

```
conda activate biomedin215
```

This will activate the virtual environment you created in the previous step.

4. Finally, ensure that your `ipynb` (this notebook)'s kernel is set to utilize the `biomedin215` virtual environment you created in the previous steps. Depending on which IDE you are using to run this notebook, the steps to do this may vary.

```
In [1]: # Run this cell:
# The lines below will instruct jupyter to reload imported modules before
# executing code cells. This enables you to quickly iterate and test revisio
# to your code without having to restart the kernel and reload all of your
# modules each time you make a code change in a separate python file.

%load_ext autoreload
%autoreload 2
```

```
In [2]: # Run this cell to ensure the environment is setup properly
import pandas as pd
import os

print("Sanity check: Success")
```

Sanity check: Success

---

## Data Description

We will be utilizing a subset of the [MIMIC III database](#), a publically available database of de-identified electronic health records from patients admitted to Intensive Care Units (ICUs) of the [Beth Israel Deaconess Medical Center](#) in Boston, Massachusetts. MIMIC III is widely used for research and benchmark evaluations in the field of clinical informatics.

You will analyze the available data to identify a cohort of patients that underwent septic shock during their admission to the ICU. **All of the data you need for this assignment is available on Canvas.**

Once you have downloaded and unzipped the data, you should see the following 6 csv files:

- `hypotension_labels.csv`
- `vitals_cohort_sirs.csv`
- `notes_small_cohort_v2.csv`
- `fluids_all.csv`
- `labs_cohort.csv`
- `diagnoses.csv`

**Specify the location of the folder containing the data in the following cells:**

```
In [3]: # Specify the path to the folder containing the data files
data_dir = "/Users/stevenang/Documents/stanford/biomedin215/assignments/A3/c
```

```
In [4]: # Run this cell to make sure all of the files are in the specified folder
expected_file_list = ["hypotension_labels.csv",
                      "vitals_cohort_sirs.csv",
                      "notes_small_cohort_v2.csv",
                      "fluids_all.csv",
                      "labs_cohort.csv",
                      "diagnoses.csv"]

for file in expected_file_list:
    assert os.path.exists(os.path.join(data_dir, file)), "Can't find file {}

print("Sanity check: Success")
```

Sanity check: Success

# 1. Building a Cohort Based on Inclusion

# Criteria and Defining Endpoints

## 1.1: (5 pts)

The first part of any patient-level study is to identify a cohort of patients who are relevant to the study, and at what point during their records they became eligible. Typically, this is done with a set of **inclusion criteria**, which, if met, qualify the patient for inclusion in the cohort. In this assignment, we will emulate the inclusion criteria used in the [TREWScore paper](#).

Read the first paragraph of the *Materials and Methods - Study Design* in the [TREWScore paper](#).

- **What criteria did the authors use to determine which patients should enter the study?**

The authors identified 16,234 distinct patient with the following criterias:

1. Age 15 and above
2. ICU Admission with at least one assessment each of:
  - GCS
  - BUN
  - Hemotocrit,
  - Heart rate record in EHR

## 1.2 (5 pts)

By looking through the [MIMIC documentation](#), you should be able to identify which tables in MIMIC contain the data that you need to identify patients that meet the inclusion criteria you described above. If you're stuck, try looking through the [mimic-code repository](#) provided by the MIMIC maintainers to get some ideas as to how to queries work on the database.

- **Report which table(s) in MIMIC you would query in order to identify patients that meet the inclusion criteria you identified above**

Here is the mapping of the MIMIC tables to the selection criterias:

Inclusion Criteria	MIMIC III Table	Comments
GCS	CHARTEVENTS + D_ITEMS	The CHARTEVENTS contains the actual reading of GCS related data and we also need D_ITEMS to find out itemid for GCS to do the filtering
BUN	LABEVENTS + D_LABITEMS	The LABEVENTS contains the actual reading of BUN and we also need D_LABITEMS to find out itemid for BUN to do the filtering
Hemotocrit	LABEVENTS + D_LABITEMS	The LABEVENTS contains the actual reading of Hemotocrit and we also need D_LABITEMS to find out itemid for BUN to do the filtering
Heart Rate	CHARTEVENTS + D_ITEMS	The CHARTEVENTS contains the actual reading of Heart Rate and we also need D_ITEMS to find out itemid for Heart Rate to do the filtering

## 1.3 (5 pts)

It can be tricky to develop the SQL queries necessary to extract the cohort of interest. Fortunately, the course staff ran the necessary query on the MIMIC III database to extract the identifiers of patients that meet the inclusion criteria discussed above. (There will be an in-class exercise where you can practice running queries later in the course!)

Read the vitals and labs data for our cohort stored in `vitals_cohort_sirs.csv` and `labs_cohort.csv` into dataframes by running the following code cell.

```
In [5]: # Run this cell to load the data from the CSV files into Pandas DataFrames
labs_cohort = pd.read_csv(os.path.join(data_dir, "labs_cohort.csv"))
vitals_cohort_sirs = pd.read_csv(os.path.join(data_dir, "vitals_cohort_sirs.csv"))
```

Run the following cells to examine the dataframes you just created.

```
In [6]: # display(labs_cohort.head())
# print(labs_cohort.info())
```

```
In [7]: # display(vitals_cohort_sirs.head())
# print(vitals_cohort_sirs.info())
```

**Briefly describe the contents of each dataframe.**

Both dataframe contains some common items like `subject_id`, `hadm_id`, `icustay_id`, and `charttime`. Where `subject_id` is unique to patient, `hadm_id` is unique to a patient hospital stay, `icustay_id` is unique to patient ICU stay, and `charttime` records the time at which an observation was charted or recorded.

Dataframe `"labs_cohort"` records all lab events happened to the patient. `lab_id` contains the name of the laboratory procedures and `valuenum` contains the actual result of that particular laboratory exam.

On the otherhand, `"vitals_cohort_sirs"` contains patient's vital reading like Heart Rate and Blood Pressure. Similar to `"labs_cohort"`, `"vitals_cohort_sirs"` has `vital_id` describing the vital sign of the record and `valuenum` field recording the actual value of the reading.

## 1.4 (5 pts)

While we are ultimately interested in utilizing all of the data from all of the patients that meet our inclusion criteria, it is good practice to work with a small *development* set of data for the purposes of developing our analytical pipeline, so that we may test ideas quickly without having to wait for code to execute on large datasets.

### 1.4.1 Coding Exercise: `get_dev_cohort_list`

Congratulations, you made it to the first coding exercise of the course! 🎉

In the `src` folder, you will find a file called `cohort.py` that contains a function called `get_dev_cohort_list`.

Complete the function following the instructions in the docstring. When you have completed the function, create a development cohort list from the vitals dataframe by running the following cell.

```
In [8]: # Run this cell after you have completed the get_dev_cohort_list in A3/src/c
from src.cohort import get_dev_cohort_list

cohort_list = get_dev_cohort_list(labs_cohort, num_subject_ids=1000)

# Sanity Check:
assert len(cohort_list) == 1000, f"Expected 1000 subject_ids, got {len(cohort_list)}"
assert cohort_list[0] == 3, f"Expected first subject_id to be 3, got {cohort_list[0]}"
assert cohort_list[-1] == 1390, f"Expected last subject_id to be 1390, got {cohort_list[-1]}"
```

```
print("Sanity check: Success")
```

Sanity check: Success

#### 1.4.2 Coding Exercise: `filter_df`

Now that we have our development cohort list, we can use it to filter the dataframes we created above to create development cohort versions of both dataframes.

Complete the function `filter_df` in `cohort.py` following the instructions in the docstring. When you have completed the function, create development versions of the `vitals_cohort_sirs` and `labs_cohort` dataframes by running the following cell.

```
In [9]: # Run this cell after you have completed the filter_df in A3/src/cohort.py
# NOTE: you do not need to modify the code in this cell
from src.cohort import filter_df

# Filter the DataFrame
dev_vitals = filter_df(vitals_cohort_sirs, "subject_id", cohort_list)
dev_labs = filter_df(labs_cohort, "subject_id", cohort_list)

# Sanity Check:
assert len(dev_vitals) == 599024, f"Expected 599024 rows, got {len(dev_vitals)}"
assert len(dev_labs) == 295845, f"Expected 295845 rows, got {len(dev_labs)}"
print("Sanity check: Success")
```

Sanity check: Success

## 1.5 (10 pts)

The **Systemic Inflammatory Response Syndrome (SIRS) criteria** has been an integral tool for the clinical definition of sepsis for the past several decades. If you are interested in learning more about SIRS and some associated controversies surrounding its continued use, read the following article: [Systemic Inflammatory Response Syndrome Criteria in Defining Severe Sepsis](#)

The next step in our process will be to assess whether patients satisfy each of the SIRS criteria at *each time step that vitals or lab data is available*. To this end, we would like to have a dataframe where each row corresponds to a unique combination of `subject_id`, `hadm_id`, `icustay_id`, and `charttime`, and with one column for each unique type of lab or vital that was measured at that time.

#### 1.5.1 Coding Exercise: `mean_summarization`

Some patients have multiple measurements of a given vital or lab that were taken at the same time. There may also have been multiple types of raw measurement that were mapped to the same measurement label.

Run the following cell to see an example of this:

```
In [10]: # Run this cell.
# NOTE: you do not need to modify the code in this cell

# This line displays all HEMATOCRIT labs for the patient with subject_id 34
# display(dev_labs[(dev_labs['subject_id'] == 34) & (dev_labs['lab_id'] == 'H
```

In the output of the above cell, there should be two `HEMATOCRIT` measurements recorded at the same time for the patient with `subject_id` 34.

This type of issue is very common in EHR data, as healthcare data is notoriously messy.

We need a single value to assess the SIRS criteria at each timepoint for each relevant lab, so we will need to write a function that summarizes the data for us.

There are many ways we could potentially do this. Some commonly used approaches include:

- Summarize duplicate measurements through an operation (such as taking the mean or median)
- Pick a single measurement at random from the set of duplicate measurements for each time point
- Pick the measurement with the highest (or lowest) value from the set of duplicate measurements for each time point

For this exercise, you will implement a function that summarizes values by calculating the **mean** value for each patient at a given time point.

Complete the function `summarize_by_mean` in `features.py` file following the instructions in the docstring. When you have completed the function, use it to summarize the labs and vitals data for each patient timepoint by running the following cell.

```
In [11]: # Run this cell after you have completed the summarize_by_mean in A3/src/features.py
# NOTE: you do not need to modify the code in this cell
from src.features import summarize_by_mean

# Summarize the labs by mean
```



```

dev_labs_mean = summarize_by_mean(dev_labs, columns_to_group_by=["subject_id", "hadm_id", "icustay_id", "charttime"])

# Summarize the vitals by mean
dev_vitals_mean = summarize_by_mean(dev_vitals, columns_to_group_by=["subject_id", "hadm_id", "icustay_id", "charttime"])

# Sanity Check:
assert len(dev_labs_mean) == 295769, f"Expected 295769 rows, got {len(dev_labs_mean)}"
assert len(dev_vitals_mean) == 579157, f"Expected 579157 rows, got {len(dev_vitals_mean)}"
print("Sanity check: Success")

```

Sanity check: Success

### 1.5.2 Coding Exercise: `pivot_wide`

Since we eventually want to assess whether patients satisfy the SIRS criteria at each time point, we would like to have a dataframe where each row corresponds to a unique combination of `subject_id`, `hadm_id`, `icustay_id`, and `charttime`, and contains one column for each unique type of lab or vital. Instead of having a `lab_id` (or `vital_id`) column to indicate what data we are working with and a `value_num` column to indicate the value, we want to have separate columns for each `lab_id` (or `vital_id`) that contain the corresponding value.

This is often referred to as a "wide" format of a table, since the dataframe becomes much wider by adding a column for each unique type of lab or vital. It is often advantageous to have data in a "wide" format for machine learning, as different weights and biases are associated with different features (which will be stored in the columns).

Implement the function `pivot_wide` in `features.py` file following the instructions in the docstring. When you have completed the function, create wide format versions of the mean-summarized vitals and labs data by running the following cell.

```

In [12]: # Run this cell after you have completed the pivot_wide in A3/src/features.py
# NOTE: you do not need to modify the code in this cell
from src.features import pivot_wide

# Pivot the labs
dev_labs_pivot = pivot_wide(dev_labs_mean, index_columns=["subject_id", "hadm_id", "icustay_id", "charttime"], columns_to_pivot=["lab_id", "value_num"])

# Pivot the vitals
dev_vitals_pivot = pivot_wide(dev_vitals_mean, index_columns=["subject_id", "hadm_id", "icustay_id", "charttime"], columns_to_pivot=["vital_id", "value_num"])

# Sanity Check: Number of columns
assert len(dev_labs_pivot.columns) == 24, f"Expected 24 columns in dev_labs_pivot"
assert len(dev_vitals_pivot.columns) == 8, f"Expected 8 columns in dev_vitals_pivot"
print("Sanity check: Success")

```

Sanity check: Success

## 1.6 (5 pts)

Since the measurement times for the vitals and labs may be different, the next step is to merge the vitals and labs dataframes together to get a complete timeline for each patient.

Implement the function `merge_dataframes` in `features.py` file following the instructions in the docstring. When you have completed the function, create a joined version of the wide format vitals and labs dataframes by running the following cell.

```
In [13]: # Run this cell after you have completed the pivot_wide in A3/src/features.py
# NOTE: you do not need to modify the code in this cell
from src.features import merge_dataframes

dev_merged = merge_dataframes(dev_labs_pivot, dev_vitals_pivot)

assert dev_merged.shape[0] == 229577, f"Your dev_merged dataframe should have 229577 rows"
assert dev_merged.shape[1] == 28, f"Your dev_merged dataframe should have 28 columns"

print("Sanity check: Success")
```

Sanity check: Success

## 1.7 (5 pts)

You will notice that the resulting dataframe contains a lot of "missing" values recorded as `NA`. There are many potential approaches for handling missing values that we could take to address this issue. In this case, we are going to use a last-value-carried-forward approach within an ICU stay to impute missing values.

**In a sentence or two, describe the potential benefits of using the last-value-carried-forward approach.**

This approach preserves the sample size and can be useful in time series analysis, specially when data is missing at random or when maintaining the most recent known value is a reasonable assumption

**In a sentence or two, describe the potential drawbacks of using the last-value-carried-forward approach.**

In case of the missing data is not random or long period of missing data, the last-value-

carried-forward approach assumes no changes occurs during these gaps. This assumption is very dangerous because it might introduce bias by artificially reducing variability of the feature and potentially overestimating stability in the data which may leads to inaccurate conclusion

Implement the function `impute_missing` in `features.py` file following the instructions in the docstring. When you have completed the function, impute missing values in the merged vitals and labs dataframe by running the next cell.

```
In [14]: # Run this cell after you have completed the impute_missing in A3/src/features.py
# NOTE: you do not need to modify the code in this cell
from src.features import impute_missing

# Impute missing values
dev_imputed = impute_missing(dev_merged)

# Assert there are no missing values as a sanity check
assert dev_imputed.isnull().sum().sum() == 565949 , "There is an incorrect r

print("Sanity check: Success")
```

Sanity check: Success

## 1.8 (5 pts)

In the TREWScore paper, the authors considered a patient to have sepsis **if at least two of the four SIRS criteria were simultaneously met** during an admission where a suspicion of infection was also present.

The four SIRS criteria are as follows:

1. A body temperature of under 36 °C or over 38 °C
2. Heart rate measured at over 90 beats per minute
3. Respiratory rate measured at over 20 breaths per minute OR a PaCO2 level of under 32 mmHg
4. A white blood cell count: over 12,000 cells/mm<sup>3</sup> OR under 4,000 cells/mm<sup>3</sup> OR greater than 10% immature forms (bands)

We currently have the following data columns for each example:

```
In [15]: # Run this cell to see the names of the columns in the DataFrame:
dev_imputed.columns
```

```
Out[15]: Index(['subject_id', 'hadm_id', 'icustay_id', 'charttime', 'ALBUMIN',
               'ANION GAP', 'BANDS', 'BICARBONATE', 'BILIRUBIN', 'BUN', 'CHLORIDE',
               'CREATININE', 'GLUCOSE', 'HEMATOCRIT', 'HEMOGLOBIN', 'INR', 'LACTAT
               E',
               'PLATELET', 'POTASSIUM', 'PT', 'PTT', 'PaCO2', 'SODIUM', 'WBC',
               'HeartRate', 'RespRate', 'SysBP', 'TempC'],
              dtype='object')
```

Implement the following functions in `sirs.py` using the the instructions in the docstrings:

- `summarize_sirs`
- `get_criteria_1`
- `get_criteria_2`
- `get_criteria_3`
- `get_criteria_4`

After implementing the functions, run the following cell to test your implementation.

```
In [16]: # Run this cell after you have completed the summarize_sirs in A3/src/sirs.py
# NOTE: you do not need to modify the code in this cell
from src.sirs import summarize_sirs

# Apply the summarize_sirs function to the DataFrame
dev_sirs = summarize_sirs(dev_imputed)

# Sanity check the correct number of columns
assert dev_sirs.shape[1] == 8, "The DataFrame has the wrong number of columns"
assert dev_sirs.shape[0] == dev_imputed.shape[0], "The DataFrame has the wrong number of rows"

print("Sanity check: Success")
```

Sanity check: Success

## 1.9 (20 pts)

At this point, we have computed the SIRS criteria for every patient in our cohort. To determine which patients meet the TREWScore definition of sepsis we now also need to determine which patients had suspicion of infection. In the TREWScore paper, the authors use a set of ICD9 codes to identify infection-related diagnoses.

We have extracted the entirety of the relevant table where ICD9 codes are stored in MIMIC and provide it to you in *diagnoses.csv*. Run the following cell to load the data into a dataframe.

```
In [17]: # Run to read in the diagnoses table from the MIMIC-III database, with all c
# NOTE: No modifications are needed in this cell
diagnoses = pd.read_csv(os.path.join(data_dir, "diagnoses.csv"), dtype=str).
diagnoses.columns = diagnoses.iloc[0]
diagnoses = diagnoses.iloc[1:].reset_index(drop=True)
diagnoses["subject_id"] = diagnoses["subject_id"].astype(int)
diagnoses["hadm_id"] = diagnoses["hadm_id"].astype(int)
```

```
In [18]: # Run the this cell to see what kind of data is in the diagnosis table.
# diagnoses.head()
```

Following the design of the TREWScore paper, we will use two methods to determine if a patient has an infection within a specific admission:

- A) An ICD-9 code indicating infection was present during the admission.
- B) The word `sepsis` or `septic` appears in a note taken during the admission.

### 1.9.1 (10 pts)

First, we will use ICD9 codes to determine which admissions indicate infection was present. Inside of `icd9_processing.py` we have provided lists of ICD9 code prefixes (See `INFECTION_ICD9_PREFIX` in `icd9_processing.py`). Using this reference information, implement the function `summarize_icd9` in `icd9_processing.py` to determine if a given admission has an ICD9 code associated with infection. Follow the instructions in the docstring to implement the function, and run the following cell when you are done to sanity check your implementation.

```
In [19]: # Run this cell after you have completed the summarize_sirs in A3/src/key_ic
# NOTE: you do not need to modify the code in this cell
from src.icd9_processing import summarize_icd9

# Apply the has_infection function to the DataFrame
icd9_infections = summarize_icd9(
    diagnoses,
    subject_ids=cohort_list,
    indicator_column_name="has_icd9_infection",
    icd9_prefix_list="infection")

# Sanity check the correct number of columns
assert icd9_infections.shape[1] == 3, "The DataFrame has the wrong number of
```

```
assert icd9_infections.shape[0] == 1378, "The DataFrame has the wrong number of rows"
print("Sanity check: Success")
```

Sanity check: Success

## 1.9.2 (10 pts)

In the paper, the authors also consider a patient to have infection during an admission if there is at least one mention of the terms `sepsis` or `septic` in a clinical note taken during their admission. The course staff has done the work of extracting the clinical notes for the 1000 patients we selected for our development set.

```
In [20]: notes = pd.read_csv(os.path.join(data_dir, "notes_small_cohort_v2.csv"), dtype=str)
notes["subject_id"] = notes["subject_id"].astype(int)
notes["hadm_id"] = notes["hadm_id"].astype(int)
```

```
In [21]: # Run this cell to see what the note summary looks like:
# notes.head()
```

Implement the function `summarize_notes` in `note_processing.py` following the instructions in the docstring. When you are done, run the following cell to sanity check your implementation.

```
In [22]: from src.note_processing import summarize_notes

note_infections = summarize_notes(notes, "has_note_infection")

# Sanity check
assert note_infections.shape[0] == 1373, "The DataFrame has the wrong number of rows"
assert note_infections.shape[1] == 3, "The DataFrame has the wrong number of columns"

print("Sanity check: Success")
```

Sanity check: Success

## 1.10 (5 pts)

Congrats! At this stage, we now have all the information we need to determine the times that patients meet the criteria for sepsis as defined in the TREWScore paper. Now we want to join the infection results with the SIRS criteria dataframe.

First, join the `note_infections` and `icd9_infections` by completing the function `join_infections` in `cohort.py`. Following the implementation instructions in the

docstring. Run the following cell to sanity check your implementation.

```
In [23]: from src.cohort import join_infections

# Join the infection tables
all_infections = join_infections(icd9_infections, note_infections)

# Sanity check: Ensure that Nan values are replaced with 0
assert all_infections.isnull().sum().sum() == 0, "There are missing values i

print("Sanity check: Success")
```

Sanity check: Success

Next, we want to merge `all_infections` with the `dev_sirs` dataframe. Implement the functions `summarize_sepsis` and `get_sepsis_status` in the file `trewscore.py`. Follow the instructions in the docstring of the function, and run the following cell to sanity check your implementation.

```
In [24]: from src.trewscore import summarize_sepsis

# Summarize Sepsis
dev_sepsis = summarize_sepsis(dev_sirs, all_infections)

assert dev_sepsis.shape[0] == 229577
assert dev_sepsis.shape[1] == 11

print("Sanity Check: Success")
```

Sanity Check: Success

We now have all of the sepsis labels! Almost done!

## 1.11 (10 pts)

In the TREWScore paper, the authors also identify patients with **severe sepsis** and **septic shock**. These are the last definitions we need to extract to prepare our cohort for analysis in the next assignment!

### 1.11.1 (5 pts)

Severe sepsis is defined as sepsis with **organ dysfunction**. Unfortunately, the criteria the authors use to define organ dysfunction is rather cumbersome. Instead of

implementing that criteria explicitly, we adopt a simpler approach. In the [Angus 2001 paper](#), the authors did just that by defining a set of ICD9 codes as a proxy for sepsis-related organ dysfunction. We have provided a list of ICD9 prefixes to pull the relevant codes.

Here we will reuse your implementation of the `summarize_icd9` function to pull the ICD9 information.

```
In [25]: # NOTE: The summarize_icd9 function will not be tested for this step
# question, but should work for any set of valid arguments.
organ_dys = summarize_icd9(diagnoses,
    subject_ids=cohort_list,
    indicator_column_name="has_organ_dysfunction",
    icd9_prefix_list="organ_disfunction")
```

Now that we have the `has_organ_dysfunction` column, we can extract the `severe_sepsis_status`. Implement `summarize_severe_sepsis` in `trewscore.py`. Run the following cell to sanity check your implementation.

```
In [26]: from src.trewscore import summarize_severe_sepsis

# Get the TREWScore Severe Sepsis status for each subject at each charttime
dev_severe_sepsis = summarize_severe_sepsis(dev_sepsis, organ_dys)

assert dev_severe_sepsis.shape[0] == 229577, "dev_severe_sepsis should have 229577 rows"
assert dev_severe_sepsis.shape[1] == 13, "dev_severe_sepsis should have 13 columns"

print("Sanity Check: Success")
```

Sanity Check: Success

### 1.11.2 (5 pts)

Finally, `septic shock` is defined as **severe sepsis**, **hypotension**, and **adequate fluid resuscitation** occurring at the same time. In order to determine which patients met the criteria for `septic shock` according to the TREWScore paper, we will first need to define the concepts **adequate fluid resuscitation** and **hypotension**.

The course staff have created a dataset with a variable indicating whether a patient had adequate fluid resuscitation or was hypotensive at each timepoint in their record. These data are stored in `fluids_all.csv`, `hypotension_labels.csv`.

```
In [27]: # Run this cell to load the data
```



```
# NOTE: No modifications are needed in this cell

# Load the data and filter to the cohort
hypotension_labels = pd.read_csv(os.path.join(data_dir, "hypotension_labels.csv"))
hypotension_labels = hypotension_labels[hypotension_labels["subject_id"].isin(cohort_list)]
fluids_all = pd.read_csv(os.path.join(data_dir, "fluids_all.csv"))
fluids_all = fluids_all[fluids_all["subject_id"].isin(cohort_list)]

# Drop columns that are not needed
fluids_all = fluids_all.drop(["amount_24h", "current_amount", "relative_amount"])
```

Implement the function `summarize_septic_shock` in the file `trewscore.py` following the instructions in the docstring. Run the cell below to sanity check your implementation.

```
In [28]: from src.trewscore import summarize_septic_shock

# Run this cell after you have completed the summarize_septic_shock in A3/src
dev_septic_shock = summarize_septic_shock(dev_severe_sepsis, hypotension_labels)


# Sanity Check:
assert dev_septic_shock.shape[0] == 239571, "dev_septic_shock should have 239571 rows"

print("Sanity Check: Success")
```

Sanity Check: Success

---

Congratulations! You've extracted a patient cohort from MIMIC and derived multiple sepsis-related endpoints.

You're done with assignment 3! 

---

## Feedback (0 points)

Please fill out the following [feedback form](#) so we can improve the course for future students!

---

## Submission Instructions

There are two files you must submit for this assignment:

1. A `PDF` of this notebook.
  - **Please clear any large cell outputs from executed code cells before creating the PDF.**
    - Including short printouts is fine, but please try to clear any large outputs such as dataframe printouts. This makes it easier for us to grade your assignments!
  - To export the notebook to PDF, you may need to first create an HTML version, and then convert it to PDF.
2. A `zip` file containing your code generated by the provided `create_submission_zip.py` script:
  - Open the `create_submission_zip.py` file and enter your SUNet ID where indicated.
  - Run the script via `python create_submission_zip.py` to generate a file titled `<your_SUNetID>_submission.zip` in the root project directory.