Training Data for Machine Learning to Enhance PCOR Data Infrastructure

Implementation Guidance

- 6.1 Data Understanding
- 6.2 Overall Training Dataset
  - 6.2.0 Deidentify the Data
  - 6.2.1 Overview of Cohort Creation
  - 6.2.2 Connect to Postgres Database
  - 6.2.3 Convert Data to CSV
  - 6.2.4 Create Patients Table
  - 6.2.5 Create Medevid Table
  - 6.2.6 Join Patients to Medevid
  - 6.2.7 Create Transplant Waitlist Features
  - 6.2.8 Create Partition Data
  - 6.2.9 Join patients_medevid_waitlist Table to the Partition Index
    - 6.2.9.1 Calculate Demographic Subtotals Per Partition
  - 6.2.10 Get Pre-ESRD Claims Data
  - 6.2.11 Create Claims Tables
  - 6.2.12 Map Diagnosis Codes (drg_cd) to Primary Diagnosis Codes (pdgns_cd)
  - 6.2.13 Get pre-2011 pre-ESRD Claims Data
  - 6.2.14 Diagnosis Groupings
  - 6.2.15 Aggregate pre-ESRD Claims Data
  - 6.2.16 Join the preesrdfeatures Tables to the Partition Index
  - 6.2.17 Map ICD-9 to ICD-10
  - 6.2.18 Prepare Data for Modeling
  - 6.2.19 Impute Missing Values
  - 6.2.20 Utility Files
    - dx_mappings_ucsf.csv
    - 2017_I9gem_map.txt
    - icd10_ccs_codes.R
    - icd10_dx_codes.txt
    - icd9_ccs_codes.R
    - icd9_dx_2014.txt
    - imputation_rules.xlsx
    - pre_esrd_ip_claim_variables.R
    - pre_esrd_hh_claim_variables.R
    - pre_esrd_hs_claim_variables.R
    - pre_esrd_op_claim_variables.R
    - pre_esrd_sn_claim_variables.R
    - pre_esrd_pre2011_claim_variables.R
    - setfieldtypes.R
  - 6.2.21 Documentation of the Training Dataset

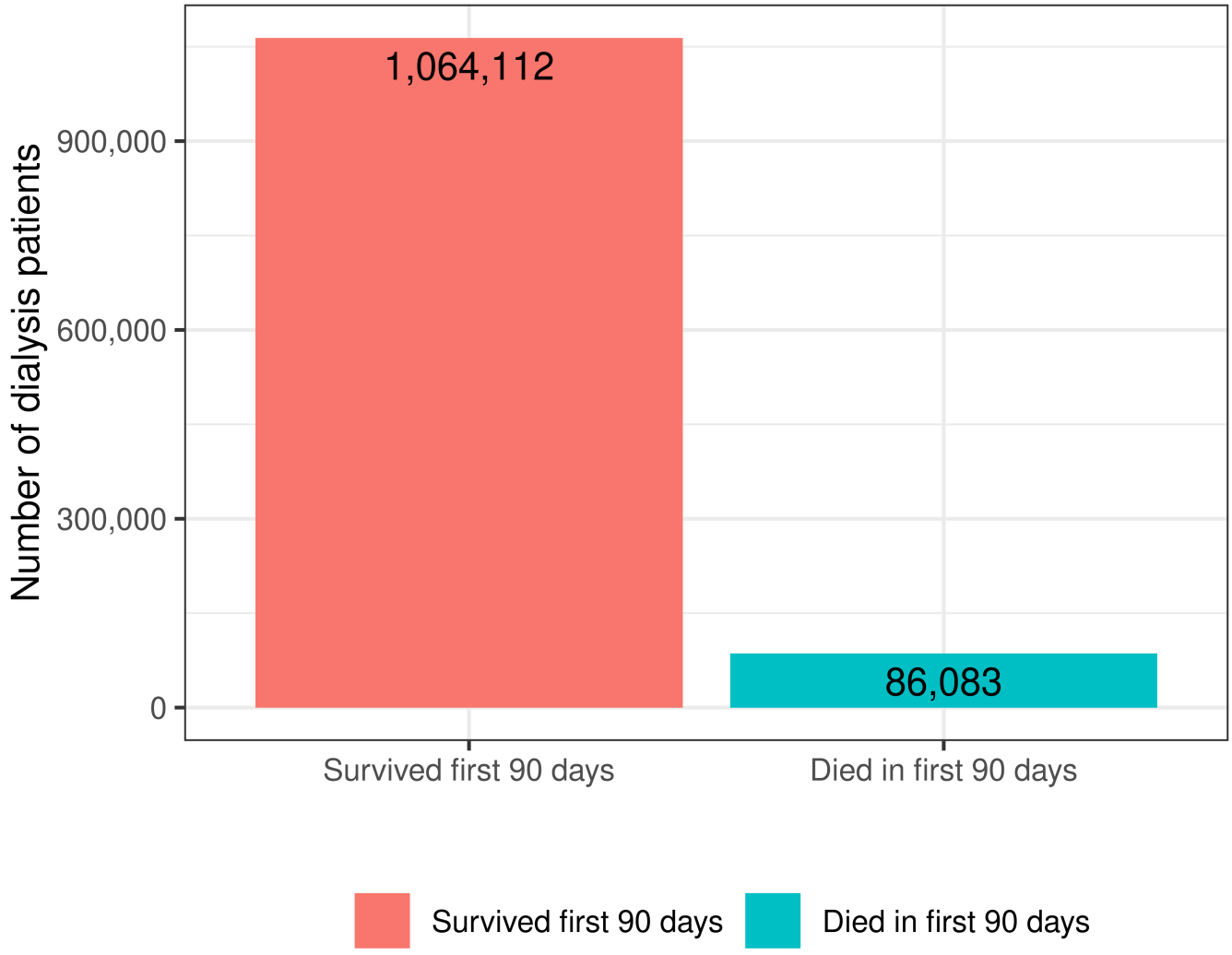
- 6.3 ML Modeling and Evaluation
  - 6.3.1 Non-Imputed XGBoost Model
    - 6.3.1.1 Pre-processing the training dataset
    - 6.3.1.2 Hyperparameter tuning for the non-imputed dataset
    - 6.3.1.3 Final XGBoost model for the non-imputed dataset
    - 6.3.1.4 Calibration
    - 6.3.1.5 Plotting calibrated results
    - 6.3.1.6 Saving data for the fairness assessment
    - 6.3.1.7 Fairness assessment
    - 6.3.1.8 Risk assessment
  - 6.3.2 Imputed XGBoost Model
    - 6.3.2.1 Pre-processing the training dataset
    - 6.3.2.2 Hyperparameter tuning for each imputed dataset
    - 6.3.2.3 Pooled hyperparameter tuning
    - 6.3.2.4 Final imputed XGBoost model
    - 6.3.2.5 Calibration
    - 6.3.2.6 Plotting calibrated results
    - 6.3.2.7 Saving data for the fairness assessment
    - 6.3.2.8 Fairness assessment
    - 6.3.2.9 Risk assessment
  - 6.3.3 Logistic Regression (LR) Model
    - 6.3.3.1 Pre-processing the training dataset
    - 6.3.3.2 Hyperparameter tuning and final logistic regression model
    - 6.3.3.3 Pool results
    - 6.3.3.4 Plot results
    - 6.3.3.5 Feature importance
    - 6.3.3.6 Fairness assessment
    - 6.3.3.7 Risk assessment
  - 6.3.4 Artificial Neural Network--Multilayer Perceptron (MLP) Model
    - 6.3.4.1 Run docker container (optional)
    - 6.3.4.2 Run on a server (i.e. AWS)
    - 6.3.4.3 Pre-processing the data
    - 6.3.4.4 Hyperparameter tuning
    - 6.3.4.5 Building layers and compiling the model
    - 6.3.4.6 Final MLP model
    - 6.3.4.7 Pool results
    - 6.3.4.8 Plot results
    - 6.3.4.9 Fairness assessment
    - 6.3.4.10 Risk assessment

6.1 Data Understanding

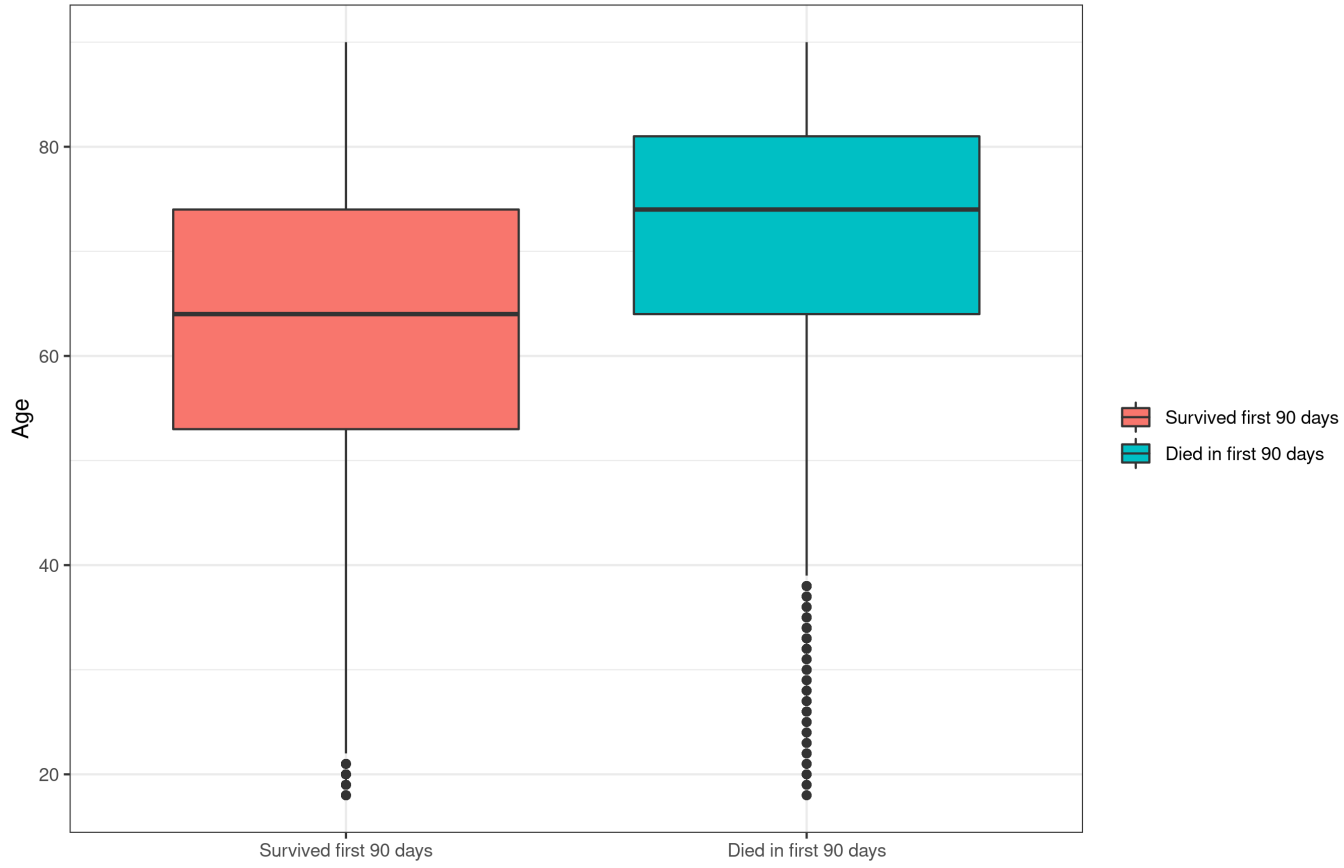
The source data for building the overall training dataset was obtained from the United States Renal Data System (USRDS), the national data registry developed from resources initiated by the Centers for Medicare & Medicaid Services (CMS) and its funded end-stage kidney disease (ESKD) networks and subsequently maintained by the National Institute for Diabetes and Digestive and Kidney Diseases (NIDDK). USRDS stores

and distributes data on the outcomes and treatments of chronic kidney disease (CKD) and ESKD population in the U.S. *(Note: to be consistent with USRDS terminology for data tables, this document uses end stage renal disease - ESRD - instead of ESKD.)* To better understand the data, data profiling was performed on the demographic variables and the outcome variable of interest (mortality in the first 90 days of dialysis). Information on constructing the outcome variable can be found in Section 6.2.4 Create Patients Table.

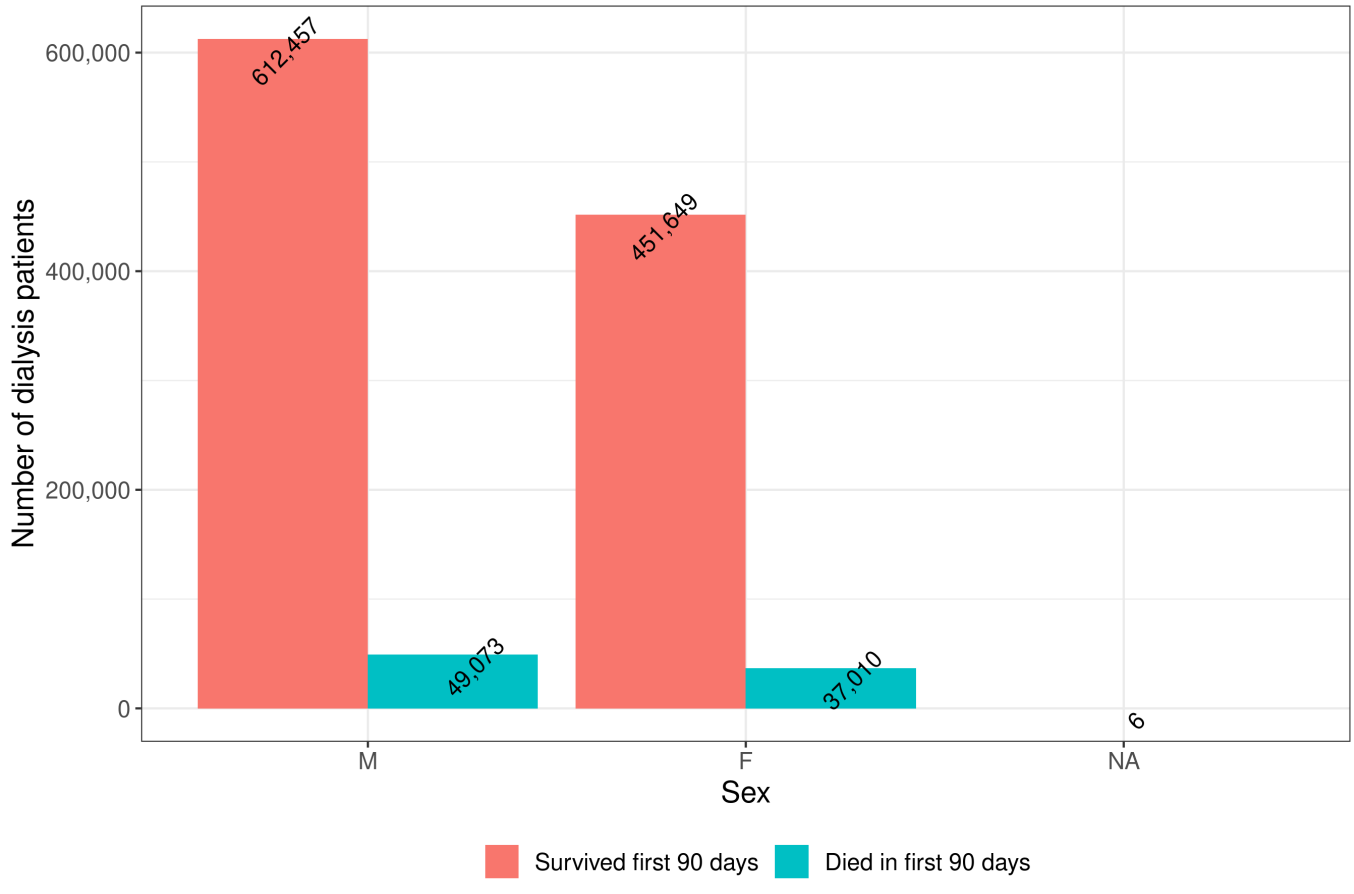
The distribution of patients in the cohort who survived versus died in the first 90 days after dialysis initiation:



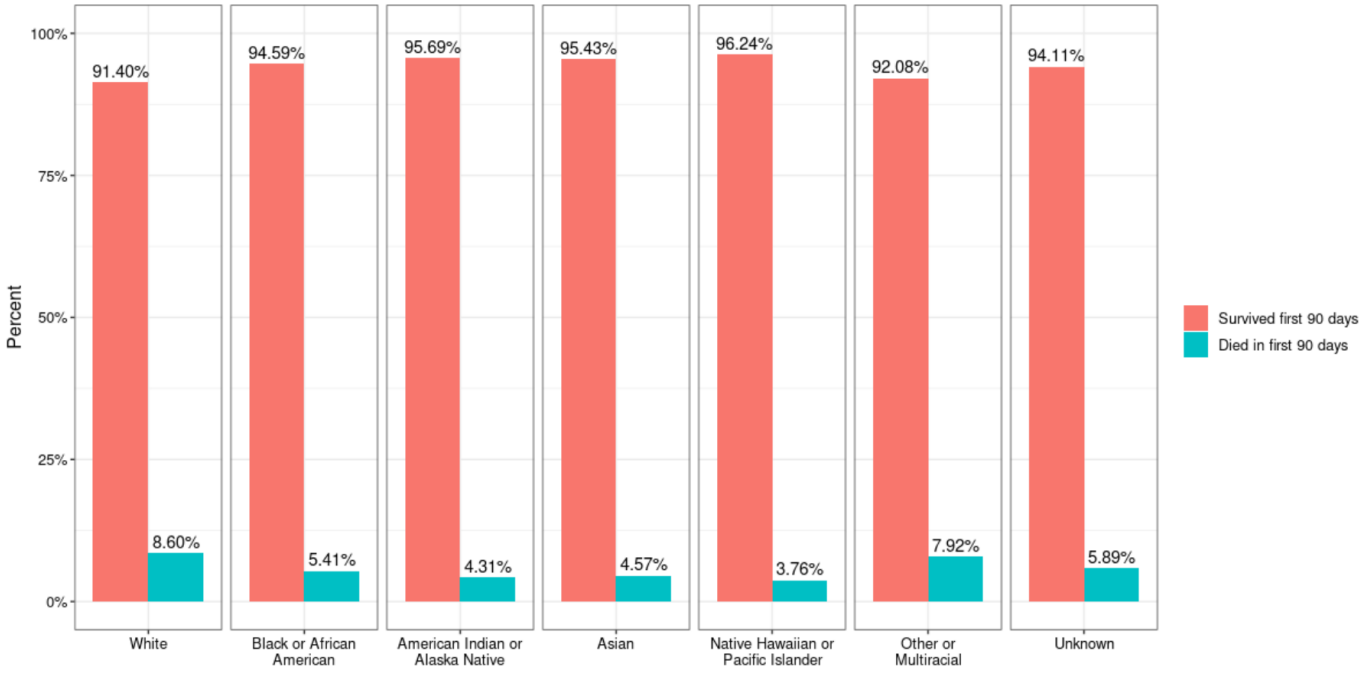
The age distribution of patients who survived versus died in the first 90 days after dialysis initiation:



The sex distribution of patients who survived versus died in the first 90 days after dialysis initiation:



The distribution by race of patients who survived versus died in the first 90 days after dialysis initiation:



6.2 Overall Training Dataset

Section 6.2 details the methodology used to create the overall training dataset. A high level overview of the tables used for the training dataset can be found in Section 6.2.1 Overview of Cohort Creation and results in a final dataset with 1,150,195 observations and 188 variables. The final dataset used for modeling is stored in PostgreSQL (Postgres) tables called `medxpreesrd`for the non-imputed variables and `micecomplete_pmm`for the imputed variables (5 sets of imputations were generated; more information on imputations can be found in Section 6.2.19 Impute Missing Values).

The construction of `medxpreesrd` involves using more than 20 USRDS data tables, as well as publicly available data, for mapping diagnosis codes to groupings.

All scripts are located in the DataSet/ directory on GitHub.

Two types of files are involved in constructing `medxpreesrd`:

1. Sequential scripts - these have the prefix **S0-, "S1-"**, etc. to indicate the sequence in which they are run
2. Utility scripts - these create the data used by the sequential scripts

Other resources that could be helpful to users include:

- USRDS Researcher's Guide
- USRDS Researcher's Guide Appendix A
- USRDS Researcher's Guide Appendix B
- USRDS Researcher's Guide Appendix C
- USRDS Researcher's Guide Appendix D

6.2.0 Deidentify the Data

The data received from USRDS was de-identified before use to comply with the approved University of San Francisco (UCSF) institutional review board (IRB) study plan. As per the Health Insurance Portability and Accountability Act (HIPAA) guidance, the following are identifiers:

- All elements of dates (except year) for dates directly related to an individual, including birth date, admission date, discharge date, date of death
- All ages over 89 and all elements of dates (including year) indicative of such age, except that such ages and elements may be aggregated into a single category of age 90 or older
- All geographic subdivisions smaller than a state, including street address, city, county, precinct, zip code, and their equivalent geocodes

The date variables in USRDS were de-identified by offsetting each date by a randomly chosen number specific to each patient. For example, if first ESRD service date is April 5, 2016 and the random offset is 60 days, then first ESRD service day is transformed to April 5, 2016 plus 60 days (or June 5, 2016); for the same patient, if the date of birth is Sept 1, 1950, then date of birth gets transformed to Sept 1, 1950 plus 60 days (or Nov 1, 1950). The ages of patients were de-identified by setting the age of all patients over the age of 90 to 90. The location variables were de-identified by removing all location variables (zip code, etc.) from the dataset.

Points to consider

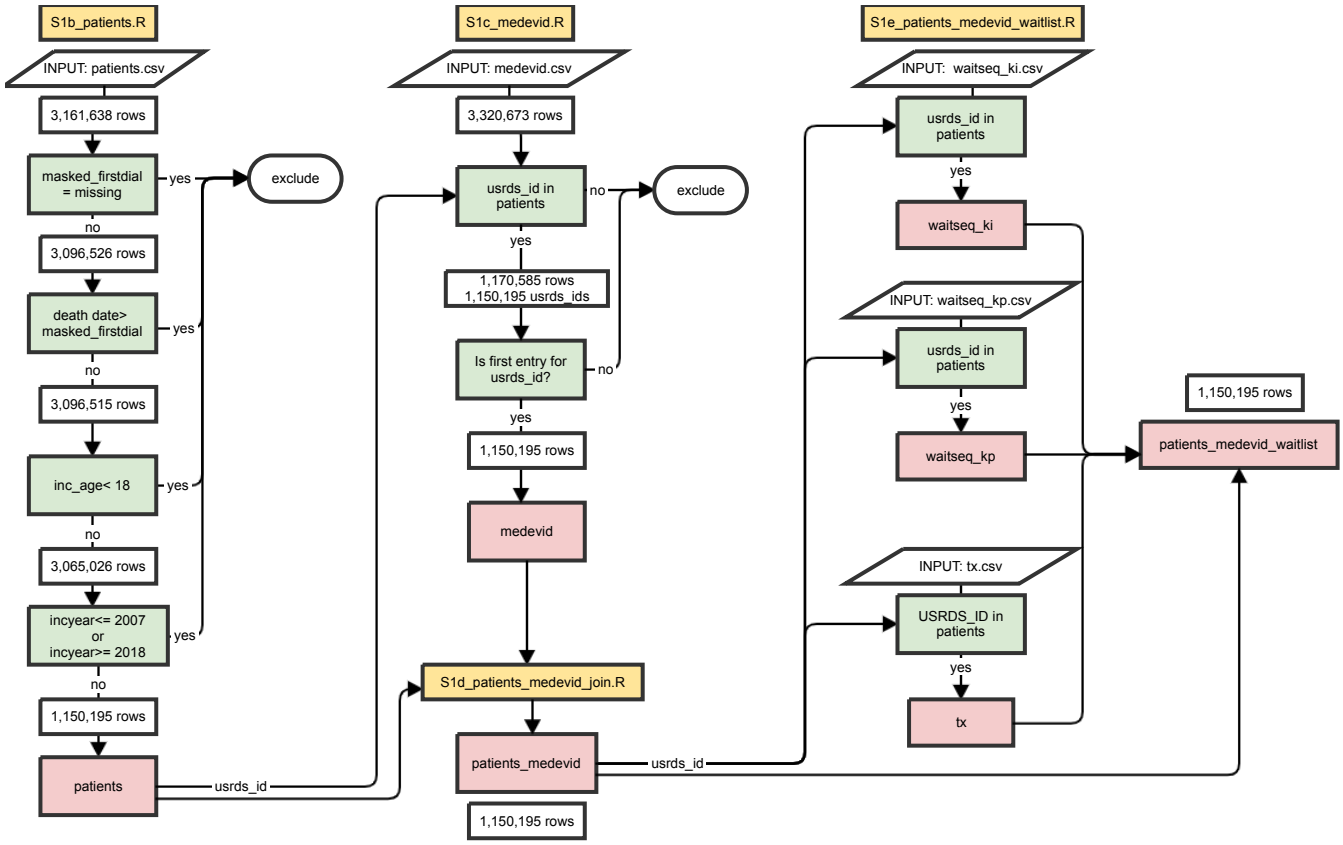
1. Other methods can be used to de-identify locations without completely deleting the variables, such as by combining all zip codes with the same three initial digits to form geographic units containing more than 20,000 people according to the current publicly available data from the Bureau of the Census. For all such geographic units containing 20,000 for fewer people, the initial three digits should be changed to 000.

2. Complete de-identification of the datasets obtained from USRDS was performed to comply with UCSF IRB requirements. Not all IRBs may require that PII/PHI be de-identified prior to use in a project. Future researchers may consider working with their IRB to ensure that relevant identifier variables for a specific use case are retained in the source dataset used for building the training datasets and ML models.

6.2.1 Overview of Cohort Creation

This diagram is a high level view of the tables used to create the cohort for the dataset. The number of rows and/or patients is listed at each stage of the cohort selection. Each of these R scripts is detailed below in the guide. The colors represent the following items:

- Yellow = R scripts
- Pink = Tables in the PostgreSQL database
- Green = Inclusion criteria



6.2.2 Connect to Postgres Database

Steps for running the S0-connectToPostgres.R script

This script creates and calls a function to 1) create a connection to a Postgres database and 2) drop a Postgres table if it exists. Both of these functions are used heavily throughout the dataset creation and are typically loaded into the top of each script. Each user will have their own details for connecting to Postgres.

Environment:

The Postgres Database for this project was hosted in an Amazon Web Services (AWS) environment with the following specifications:

```
Name: t2.large
vCPU: 2
GPU: 0
Architecture: x86_64
Memory: 8 GB
Storage: 500 GB
Operating System: Windows
Network Performance: low to moderate
Zone: US govcloud west
```

The overall training dataset set was created using R (version 3.6.3 (2020-02-29) running on x86_64 Linux Ubuntu 20.04.1 LTS) and a PostgreSQL database (PostgreSQL 12.3, compiled by gcc (GCC) 4.8.3 20140911 (Red Hat 4.8.3-9), 64-bit). The specific R libraries and versions are shown in the table below:

R library	Version
RPostgres	1.3.1
DBI	1.1.1
stringr	1.4.0
haven	2.4.0
readr	1.4.0
lubridate	1.7.9.2
dplyr	1.0.4
magrittr	1.5
tidyr	1.1.2
sqldf	0.4-11
RSQLite	2.2.3
gsubfn	0.7
proto	1.0.0
readxl	1.3.1
plyr	1.8.6
mice	3.13.0

Input:

```
Name of the database
database port
user name
user password
```

Step 1. Create function for connecting to Postgres.

```
dbConnect(
    RPostgres::Postgres(),
    dbname = cred$dbname,
    host=cred$host,
    port=cred$port,
    user=cred$user,
    password=cred$password
    )
- Output:
    - An object called "con" that can be used in database queries.
```

```
- Example:
con = getConnection()
```

Step 2. Create function for dropping a Postgres table if it exists.

```
drop_table_function <- function(con, tablename) {
  if (isTRUE(dbExistsTable(con, tablename)))   {
    print(str_glue("existing {tablename} table dropped"))
    dbRemoveTable(con, tablename)
  }
  else {
    print(str_glue("{tablename} table does not exist"))
  }
}
```

6.2.3 Convert Data to CSV

Steps for running the S1a_convertSAStoCSV.R script

This script reads in a list of files from the raw source data (**.sas7bdat** files in USRDS data) and saves them as a .csv file. The reason for this conversion to CSV before loading into R is that the SAS files contain more information than is needed/able to store in Postgres or an R table, such as categorical variable encodings that are also documents in the USRDS Researchers Guide Appendix B and C.

Input:

- source_data_dir string path to the raw data file
- file_name string name of the file without the extension
- output_data_dir string path to the output directory for csv files

Ouput:

- A .csv version of the file with the same file_name

Step 1. Convert and combine the raw source data from .sas7bdat files to .csv files

```
convert_to_csv = function(source_data_dir, file_name, output_data_dir) {
  raw_file_path = haven::read_sas(str_glue("{source_data_dir}
{file_name}.sas7bdat"))
  csv_path = str_glue("{output_data_dir}{file_name}.csv")
  write_csv(raw_file_path, csv_path)
}
```

- Example:

```
convert_to_csv("/home/sas_data_usrds/", "patients",
"/home/csv_data_usrds/")}
```

6.2.4 Create Patients Table

Steps for running the S1b_patients.R script

This script creates the table `patients`in the Postgres database, after filtering on the criteria to create the study cohort and the dependent variable (died in the first 90 days of dialysis).

Input: csv files are produced in script S1a-convertSAStoCSV.R

```
patients.csv
```

Output: Postgres table

```
patients
```

Step 1. Import patients and apply exclusion criteria

```
cohort_patients = read_csv(file.path(data_dir, "patients.csv"), col_types
= cols(
      CDTYPE = "c"))

names(cohort_patients) = tolower(names(cohort_patients))
```

Connect to the Postgres database using the S0-connectToPostgres.R script, which results in the variable **con** that is used in the queries to Postgres. This also imports the **drop_table_function** used when creating a Postgres table. These functions are used in almost every script and will be imported at the top in the code.

```
source(file.path(source_dir, "S0-connectToPostgres.R"))
```

Store the raw cohort_patients data as "patients" table in Postgres (after dropping the table if it exists).

```
fields = names(cohort_patients)
drop_table_function(con, "patients")
   dbCreateTable(
      con,
      name = "patients",
      fields = cohort_patients,
      row.names = NULL
   )
dbWriteTable(
```

```
      con,
      name = "patients",
      value = cohort_patients,
      row.names = FALSE,
      append = TRUE
    )
```

The `patients` table holds the data for the 1,150,195 patients (rows) in the cohort which is created by excluding the following rows:

- age less than 18
- incident year before 2008
- incident year after 2017
- first dialysis date is missing
- death date is before first dialysis date (one patient)

The script does each exclusion criteria separately to calculate the number of rows/patients at each stage, but the exclusion can be done more simply with one SQL query shown below (and commented out in the script).

```
exclude_patients = str_glue(
                              "DELETE FROM patients
                              WHERE inc_age<18
                              OR incyear>2017
                              OR incyear<2008
                              OR masked_firstdial IS NULL
                              OR masked_died<masked_firstdial"
    )
dbSendStatement(con, exclude_patients)
```

Step 2. Using the **create_dependent_var** function, check that set all **inc_age** greater than 90 to be 90.

This step doubles checks that all patients with age > 90 are set to 90.

```
patients_dependent_var = cohort_patients %>%
  mutate(inc_age=ifelse(inc_age>90, 90, inc_age),
```

Step 3. Create the **days_on_dial** variable by converting the **masked_died** and **masked_firstdial** to a date and by calculating the number of days between them.

```
  masked_firstdial = as_date(masked_firstdial, origin = "1960-01-01"),
  masked_died = as_date(masked_died, origin = "1960-01-01"),
  days_on_dial = as.double(difftime(masked_died,
                                    masked_firstdial,
                                    units = "days")),
```

Step 4. Create the dependent variable (outcome variable) **died_in_90** by setting all patients with a value for **days_on_dial** to 1 (died) and patients with no value for **days_on_dial** to 0 (survived).

```
  died_in_90 = ifelse(is.na(days_on_dial), 0, ifelse(days_on_dial <= 90,
1, 0)),
```

Step 5. Convert data variables to dates that are used to calculate waitlist and transplant status.

```
  masked_first_se = as_date(masked_first_se, origin = "1960-01-01"),
```

can_first_listing_dt is the first date patient is ever waitlisted.

```
  masked_can_first_listing_dt = as_date(masked_can_first_listing_dt,
origin = "1960-01-01"),
```

can_rem_dt is the date patient was removed from the waitlist the first time.

```
  masked_can_rem_dt = as_date(masked_can_rem_dt, origin = "1960-01-01"),
  masked_tx1date = as_date(masked_tx1date, origin = "1960-01-01"),
  masked_tx1fail = as_date(masked_tx1fail, origin = "1960-01-01")
```

Step 6. Save this table to Postgres as `patients` after dropping the initial `patients` table.

```
drop_table_function(con, "patients")
   dbCreateTable(
      con,
      name = "patients",
      fields = patients_dependent_var,
      row.names = NULL
   )
   dbWriteTable(
      con,
      name = "patients",
      value = patients_dependent_var,
      row.names = FALSE,
      append = TRUE
   )
```

Points to consider

1. The study cohort as well as the outcome variable for the use case should be driven by a strong clinical understanding of the data and defined with clinician input. For example, kidney transplant

events occurring within the first 90 days of initiating dialysis were evaluated as a potential competing outcome for death. However, since only less than 1% of the patient cohort recieved kidney transplants, retaining the patients in the study cohort would likely have only a small effect on modeling. After consultating with clinicans and the Technical Expert Panel, it was determined that patients with kidney transplants should be retained in the patient cohort for the purposes of this analysis.

2. The origin date when converting data variables to readable dates is 1960-01-01.

6.2.5 Create Medevid Table

The Medevid table contains the data from CMS Form 2728 -- Medical Evidence Form, a form required to be completed and submitted when a patient is diagnosed with ESRD and receives their first chronic dialysis treatment(s) or receives a transplant. Medevid contains data on patient demographics, insurance status, comorbid conditions, primary cause of kidney failure, and laboratory values at the time of ESRD diagnosis as well as prior nephrology care, dietician care, and patient education.

Steps to running the 4.1.5 S1c_medevid.R script

This script creates the `medevid` table in Postgres database. The `medevid` table is filtered based on the following:

- Keep only rows with a matching **usrds_id** from our `patients` table
- For **usrds_ids** with multiple entries, select the first entry (in the table order) for each **usrds_id**
- The final table results in 1,150,195 patients/rows

Input: csv files are produced in script S1a-convertSAStoCSV.R

```
patients
medevid.csv
```

Output: Postgres table

```
medevid
```

Step 1. Import `medevid` data

```
raw_medevid = read_csv(file.path(data_dir, filename), col_types = cols(
      CDTYPE = "c",
      masked_UREADT = "c",
      ALGCON = "c",
      PATNOTINFORMEDREASON = "c",
      RACEC = "c",
      RACE_SUB_CODE = "c"))

   names(raw_medevid) = tolower(names(raw_medevid))
```

Step 2. Import the **usrds_ids** from the `patients` table

```
patients_filtered = dbGetQuery(
      con,
      str_glue(
         "
         SELECT *
         FROM {table_name_pt}
         "))
```

Step 3. Remove unused columns

Columns with the comorbidities that were only collected in the 1995 version of the Medical Evidence Form were removed from the training dataset as patients with ESRD incident years between 2008-2017 do not have this data collected.

```
medevid_ids_filtered = raw_medevid %>%
      select(-c(
                  "como_ihd",
                  "como_mi",
                  "como_cararr",
                  "como_dysrhyt",
                  "como_pericar",
                  "como_diabprim",
                  "como_hiv",
                  "como_aids")) %>%
```

Step 4. Filter on **usrds_ids** from the `patients` table

```
      filter(usrds_id %in% patients_filtered$usrds_id)
```

Step 5. Keep first row of `medevid` data if a **usrds_id** has more than one, per the USRDS Researcher's Guide for de-duplicating the `medevid` table

```
medevid_filtered = medevid_ids_filtered %>%
      distinct(usrds_id, .keep_all = TRUE) %>%
```

Step 6. Calculate the dialysis train time in days

```
      mutate(
         masked_trnend = as_date(masked_trnend, origin = "1960-01-01"),
         masked_trstdat = as_date(masked_trstdat, origin = "1960-01-01"),
```

```
          dial_train_time = as.double(difftime(masked_trnend,
                                               masked_trstdat,
                                               units = "days"))
      )
```

Points to consider

1. It is important not to sort or alter the order of (or import into a SQL database) the `medevid` table before selecting the first entry per **usrds_id** for the `medevid` table. This order of the entries from the `medevid` table is curated as per the USRDS Researcher's Guide, which advises users to selected the first `medevid` entry for analysis. For other use cases, especially those requiring a longtitudinal dataset, the multiple MEDEVID records per patient may need to be retained. Decisions on how to handle the duplicated data should be made with the proposed use case in mind.

2. Constructed features should be validated with clinicians to ensure that they are meaningful. For example, several preliminary features were initially constructed from the variables in the `medevid` table to serve as a proxy for prior nephrology care, such as whether a physician signature was present on the Medical Evidence form, the time between a patient signature and physician signature on the Medical Evidence Form, etc. After discussing these features with the clinicians on the team, it was determined that since a nephrologist's signature is required when a patient is referred for dialysis treatment, the signature date field is not a good proxy measure.

6.2.6 Join Patients to Medevid

Steps to running the S1d_patients_medevid_join.R script

This script creates the `patients_medevid` table in Postgres database. The `patients_medevid` table consists of the `patients` table with the data from the `medevid` table added via left join on **usrds_id**.

Input: Postgres tables

```
patients
medevid
```

Output: Postgres table

```
patients_medevid
```

The result of this script produces the same 1,150,195 rows as we have in the `patients` table.

Step 1. Left join the `patients` table and `medevid` table on **usrds_id**

```
patients_medevid = left_join(
      patients_filtered %>% select(-c( "cdtype")),
      medevid_filtered %>% select(-c("randomoffsetindays", "disgrpc",
"network", "inc_age",
```

```
                                          "pdis", "sex", "race",
"masked_died")),
      by = "usrds_id"
   )
```

The **cdtype** column is kept from the `medevid` table, the other duplicate colums are from the `patients` table

Step 2. Populate any missing values for **sex** and **pdis** variables in `patients` with values from `medevid` (otherwise keep any duplicate columns from `patients`)

```
patients_medevid = patients_medevid %>%
         mutate(
            sex = ifelse(is.na(sex), sex_med, sex),
            pdis = ifelse(is.na(pdis), pdis_med, pdis)
            ) %>%
         select(-c(sex_med, pdis_med))
```

6.2.7 Create Transplant Waitlist Features

The transplant data tables contains kidney transplant information from United Network for Organ Sharing (UNOS), such as information on transplant elibigility and transplant status. These features are created as transplant waitlist status and the amount of time on the transplant waitlist are indicative of a patients overall health status. Patients who are on the transplant waitlist are generally much healthier than those who are not listed.

Steps to running the S1e_patients_medevid_waitlist.R script

This script creates the `waitseq_ki`, `waitseq_kp`, `tx`, `tx_waitlist_vars` and `patients_medevid_waitlist` tables in the Postgres database from the .csv files and the `patients_medevid` table.

Input: csv files are produced in script S1a-convertSAStoCSV.R

```
patients_medevid
tx.csv
waitseq_kp.csv
waitseq_ki.csv
```

Output: Postgres tables

```
waitseq_ki
waitseq_kp
tx
tx_waitlist_vars
patients_medevid_waitlist
```

`patients_medevid_waitlist` is the full cohort that should be used from this point forward.

The result of this script is the calculation of the following variables added to the `patients_medevid` table which will be saved as the `patients_medevid_waitlist` table.

```
days_on_waitlist (number of days in transplant waitlist)
waitlist_status (active, transplanted, removed, never)
```

Step 1. Import the `patients` table

```
pat = dbGetQuery(con,
                      "SELECT *
                      FROM patients_medevid")
```

Step 2. Import the **waitseq_ki.csv** file.

```
waitseq_ki = read_csv(file.path(data_dir,"waitseq_ki.csv"), col_types =
cols(
    USRDS_ID = col_double(),
    randomOffsetInDays = col_double(),
    PROVUSRD = col_double(),
    PID = col_double(),
    masked_BEGIN = col_double(),
    masked_ENDING = col_double()
  ))
```

Step 3. Transform column names to lowercase

```
names(waitseq_ki) = tolower(names(waitseq_ki))
```

Step 4. Filter on rows with usrds_id in cohort.

```
waitseq_ki = waitseq_ki %>%
  filter(usrds_id %in% pat$usrds_id) %>%
```

Step 5. Set **masked_begin** and **masked_ending** as dates.

```
  mutate(ws_list_dt = as_date(masked_begin, origin = "1960-01-01"),
         ws_end_dt = as_date(masked_ending, origin = "1960-01-01"),
         source = "ki") %>%
```

Step 6. Keep only the following columns:

- ws_list_dt = New Waiting Period Starting Date
- ws_end_dt = New Waiting Period Ending Date
- provusrd = USRDS Assigned Facility ID
- source = 'ki'
- usrds_id
- pid

```
  select(usrds_id, pid, provusrd, ws_list_dt, ws_end_dt, source)
```

Step 7. Save as the `waitseq_ki` table in Postgres

```
fields = names(waitseq_ki)
drop_table_function(con, "waitseq_ki")
print(str_glue("create waitseq_ki in postgres"))
dbCreateTable(
  con,
  name = "waitseq_ki",
  fields = waitseq_ki,
  row.names = NULL
)
dbWriteTable(
  con,
  name = "waitseq_ki",
  value = waitseq_ki,
  row.names = FALSE,
  append = TRUE
)
```

Step 8. Import **waitseq_kp.csv**

```
waitseq_kp = read_csv(file.path(data_dir,"waitseq_kp.csv"), col_types =
cols(
  USRDS_ID = col_double(),
  randomOffsetInDays = col_double(),
  PROVUSRD = col_double(),
  PID = col_double(),
  masked_BEGIN = col_double(),
  masked_ENDING = col_double()
))
names(waitseq_kp) = tolower(names(waitseq_kp))
```

Step 9. Filter on rows with usrds_id in cohort.

```
waitseq_kp = waitseq_kp %>%
  filter(usrds_id %in% pat$usrds_id) %>%
```

Step 10. Set **masked_begin** and **masked_ending** as dates and save with new names **ws_list_dt** and **ws_end_dt**.

```
  mutate(ws_list_dt = as_date(masked_begin, origin = "1960-01-01"),
         ws_end_dt = as_date(masked_ending, origin = "1960-01-01"),
         source = "kp") %>%
```

Step 11. Keep only the following columns:

- ws_list_dt = New Waiting Period Starting Date
- ws_end_dt = New Waiting Period Ending Date
- provusrd = USRDS Assigned Facility ID
- source = 'kp'
- usrds_id
- pid

```
  select(usrds_id, pid, provusrd, ws_list_dt, ws_end_dt, source)
```

Step 12. Save as the `waitseq_kp` table in Postgres.

```
fields = names(waitseq_kp)
drop_table_function(con, "waitseq_kp")
print(str_glue("create waitseq_kp in postgres"))
dbCreateTable(
  con,
  name = "waitseq_kp",
  fields = waitseq_kp,
  row.names = NULL
)
dbWriteTable(
  con,
  name = "waitseq_kp",
  value = waitseq_kp,
  row.names = FALSE,
  append = TRUE
)
```

Step 13. Concatenate waitseq_ki and waitseq_kp.

```

```

```
waitseq = bind_rows(waitseq_ki, waitseq_kp) %>%
  arrange(usrds_id, ws_list_dt)
```

Step 14. Join the new waitseq table to `patients`.

```
pat_waitseq = left_join(
  pat %>% select(usrds_id, masked_first_se, masked_firstdial,
                 masked_can_first_listing_dt, masked_can_rem_dt,
                 masked_tx1date, masked_died, can_rem_cd, masked_tx1fail),
  waitseq,
  by = "usrds_id") %>%
  arrange(usrds_id, ws_list_dt)
```

Step 15. Label patients as ACTIVE on the waitlist and calculate the days on the transplant waitlist for ACTIVE patients

Patients are labeled as "active" (those who are considered active on the transplant waitlist) if they meet one of the following criteria:

1. If list_date is before dial_date and end_date is on or after dial_date
2. Status is ACTIVE on the first day of dialysis

First, check if earliest listing date from `waitseq` matches first listing date from `patients`.

```
first_list = pat_waitseq %>% group_by(usrds_id) %>%
  arrange(usrds_id, ws_list_dt) %>%
  distinct(usrds_id, .keep_all = TRUE) %>%
  ungroup(usrds_id)
```

If **list_date** is before **dial_date** and **end_date** is on or after **dial_date**, OR if **list_dt** < **dial_dt** and **end_dt** == NA: status is ACTIVE.

```
pat_waitseq = pat_waitseq %>%
  mutate(active = ifelse(
    (ws_list_dt < masked_firstdial & ws_end_dt >= masked_firstdial) |
(ws_list_dt < masked_firstdial & is.na(ws_end_dt)), 1, 0))

active = pat_waitseq %>%
  filter((ws_list_dt < masked_firstdial & ws_end_dt >= masked_firstdial) |
(ws_list_dt < masked_firstdial & is.na(ws_end_dt)))
```

Calculate the days on the transplant waitlist for ACTIVE patients using **dial_dt** - **ws_list_dt**.

```
active = active %>%
  mutate(
    days_on_waitlist = as.double(difftime(masked_firstdial,
                                          ws_list_dt,
                                          units = "days"))
  )
```

Step 16. Remove ACTIVE patients from `pat_waitseq`

Sort by **usrds_id** and **ws_list_dt** and keep the row with the earliest **ws_list_dt**.

```
active = active %>% group_by(usrds_id) %>%
  arrange(usrds_id, ws_list_dt) %>%
  distinct(usrds_id, .keep_all = TRUE) %>%
  ungroup(usrds_id)
```

Remove active patients from `pat_waitseq`. Get unique **usrds_ids** in active dataframe.

```
active_id = unique(active$usrds_id)
```

Filter out rows from `pat_waitseq` where **usrds_id** is in the list of active USRDS IDs (**active_id**).

```
pat_waitseq_not_act = pat_waitseq %>%
  filter(!usrds_id %in% active_id)
```

Step 17. Import the transplant dataset and process the data

Import **tx.csv**.

```
tx = read_csv(file.path(data_dir,"tx.csv"), col_types = cols(
  DHISP = "c",
  DSEX = "c",
  RHISP = "c",
  RSEX = "c"
))

names(tx) = tolower(names(tx))
```

Filter on rows with **usrds_id** in cohort.

```
tx = tx %>%
  filter(usrds_id %in% pat$usrds_id) %>%
```

Transform **masked_tdate** to a date and save as **t_tx_dt**.

```
  mutate(t_tx_dt = as_date(masked_tdate, origin = "1960-01-01"),
```

Transform **masked_faildate** to a date and save as **t_fail_dt**.

```
         t_fail_dt = as_date(masked_faildate, origin = "1960-01-01")) %>%
```

Keep only the following columns:

- **t_tx_dt** = transplant date
- **t_fail_dt** = Transplant Failure Date
- **provusrd** = USRDS Assigned Facility ID
- **tottx** = Patient Total Number of Transplants
- **tx_srce** = Source of Transplant Record
- **usrds_id**

```
  select(usrds_id, provusrd, t_tx_dt, t_fail_dt, tottx, tx_srce) %>%
  arrange(usrds_id, t_tx_dt)
```

Step 18. Save as `tx` in Postgres database

```
fields = names(tx)
drop_table_function(con, "tx")
print(str_glue("create tx in postgres"))
dbCreateTable(
  con,
  name = "tx",
  fields = tx,
  row.names = NULL
)
dbWriteTable(
  con,
  name = "tx",
  value = tx,
  row.names = FALSE,
  append = TRUE
)
```

Step 19. Construct TRANSPLANTED status

Subset rows where LISTING DATE and LIST END DATE are both **BEFORE** DIAL START DATE Subset rows with ws_list_dt (listing date) & ws_end_date (list end date) both **BEFORE** patient masked_firstdial (dialysis

start date).

```
list_before_dial = pat_waitseq_not_act %>%
  filter(ws_list_dt < masked_firstdial & ws_end_dt < masked_firstdial)
```

Group by **usrds_id**, sort by largest to smallest end_date, and keep the maximum **end_date** for each **usrds_id**.

```
closest_end_dt_to_dial = list_before_dial %>% group_by(usrds_id) %>%
  arrange(usrds_id, desc(ws_end_dt)) %>%
  distinct(usrds_id, .keep_all = TRUE) %>%
  ungroup(usrds_id)
```

Left join `closest_end_dt_to_dial` and `tx` on **usrds_id**. This has effect of filtering `tx` dataset and keeping rows where **usrds_id** is in `closest_end_dt_to_dial`.

If the maximum end date (**max_end_dt**) is equal to the transplant date (**t_tx_dt**), then the status is TRANSPLANTED.

```
max_end_dt = left_join(
  closest_end_dt_to_dial %>% select(-pid, -provusrd),
  tx %>% select(usrds_id, t_tx_dt, t_fail_dt),
  by = "usrds_id"
)

max_end_dt = max_end_dt %>%
  mutate(transplanted = if_else(is.na(t_tx_dt), 0,
                                if_else(ws_end_dt == t_tx_dt, 1, 0)))
transplanted = max_end_dt %>%
  filter(ws_end_dt == t_tx_dt)
```

Days on waitlist for TRANSPLANTED patients is **t_tx_dt** - **ws_list_dt**.

```
transplanted = transplanted %>%
  mutate(
    days_on_waitlist = as.double(difftime(t_tx_dt,
                                          ws_list_dt,
                                          units = "days"))
  )
```

Step 20. Construct REMOVED status

Remove rows from **max_end_dt** where **usrds_id** is in `transplanted`.

Get the unique **usrds_ids** in transplanted dataframe.

```
transplanted_id = unique(transplanted$usrds_id)
```

Filter out rows from **max_end_dt** where **usrds_id** is in **transplanted_id**.

```
no_act_or_trans = max_end_dt %>%
  filter(!usrds_id %in% transplanted_id)
```

The remaining IDs should have REMOVED status. Check that all rows meet the removed criteria.

```
num_no_act_tx = nrow(no_act_or_trans %>%
       filter(ws_end_dt != t_tx_dt | is.na(t_tx_dt)))
```

Create a REMOVED column and set **removed** = 1 if **ws_end_dt** != **t_tx_dt** or **t_tx_dt** = NA.

```
no_act_or_trans = no_act_or_trans %>%
  mutate(removed = if_else(ws_end_dt != t_tx_dt | is.na(t_tx_dt), 1, 0))

removed = no_act_or_trans %>%
  filter(ws_end_dt != t_tx_dt | is.na(t_tx_dt))
```

Days on waitlist for REMOVED patients is **ws_end_dt** - **ws_list_dt**.

```
removed = removed %>%
  mutate(
    days_on_waitlist = as.double(difftime(ws_end_dt,
                                          ws_list_dt,
                                          units = "days"))
  )
```

Note: REMOVED only has duplicates because the `tx` table has duplicate rows for some patients, but the waitseq start and end dates are the same for both rows of each **usrds_id**, so only the first record is kept.

```
removed = removed %>% group_by(usrds_id) %>%
  arrange(usrds_id, ws_list_dt) %>%
  distinct(usrds_id, .keep_all = TRUE) %>%
  ungroup(usrds_id)
```

Get unique **usrds_ids** in the `removed` dataframe.

```
removed_id = unique(removed$usrds_id)
```

Step 21. Merge **days_on_waitlist** with **usrds_id** from active, transplanted, and removed.

```
days = bind_rows(active %>% select(usrds_id, days_on_waitlist),
                 transplanted %>% select(usrds_id, days_on_waitlist),
                 removed %>% select(usrds_id, days_on_waitlist))
days = days %>% arrange(usrds_id)
```

Add ACTIVE patients to the `patients` table by setting all rows in `pat` where **usrds_id** is in **active_id** to ACTIVE = 1.

```
pat = pat %>%
  mutate(active = if_else(usrds_id %in% active_id, 1, 0)) %>%
  select(usrds_id, active, masked_first_se, masked_firstdial,
masked_can_first_listing_dt,
         masked_can_rem_dt, masked_tx1date, masked_died, can_rem_cd,
masked_tx1fail)
```

Add TRANSPLANTED patients to the `patients` table by setting all rows in pat where **usrds_id** is in **transplanted_id** to TRANSPLANTED = 1.

```
pat = pat %>%
  mutate(transplanted = if_else(usrds_id %in% transplanted_id, 1, 0))

n_both = nrow(pat %>% filter(active == 1 & transplanted == 1))
if (n_both!=0){
  print("WARNING! rows exist where active and transplanted are both == 1")
}
```

Add REMOVED patients to the `patients` table by setting all rows in pat where **usrds_id** is in **removed_id** to REMOVED = 1.

```
pat = pat %>%
  mutate(removed = if_else(usrds_id %in% removed_id, 1, 0))
```

Step 22. Construct the NEVER status

Set all rows where active, transplanted, and removed are all 0 to NEVER = 1.

```
pat = pat %>%
  mutate(never = if_else(active == 0 & transplanted == 0 & removed == 0,
```

```
1, 0))
```

Calculate the time on the waitlist. Join **days_on_waitlist** onto `patients` table.

```
pat = left_join(
  pat,
  days,
  by = "usrds_id"
)
```

When **never** is 0, set **days_on_waitlist** to 0.

```
pat = pat %>%
  mutate(days_on_waitlist = replace_na(days_on_waitlist, 0))
```

Step 23. Reshape into long form with one **waitlist_status** variable.

```
pat2 = pat %>%
  mutate(waitlist_status = names(
    pat %>% select(
      active, transplanted, removed,never))[
        max.col(pat %>% select(active, transplanted, removed, never))])
```

Step 24. Save waitlist variables to Postgres.

```
tx_waitlist_vars = pat2 %>%
  select(usrds_id, waitlist_status, days_on_waitlist) %>%
  arrange(usrds_id)

csv_path = str_glue("{data_dir}tx_waitlist_vars.csv")
write_csv(tx_waitlist_vars,csv_path)
drop_table_function(con, "tx_waitlist_vars")
dbWriteTable(
  con,
  name = "tx_waitlist_vars",
  value = tx_waitlist_vars,
  row.names = FALSE,
  append = TRUE)
```

Step 25. Merge with `patients_medevid` and save to Postgres by adding the waitlist and transplant features to the `patient_medevid` table.

```
patients_med = dbGetQuery(con,
                 "SELECT *
                  FROM patients_medevid")

patients_med_waitlist = inner_join(
  patients_med,
  tx_waitlist_vars,
  by="usrds_id"
)
fields = names(patients_med_waitlist)
print(str_glue("create patients_medevid_waitlist in postgres"))
drop_table_function(con, "patients_medevid_waitlist")
dbCreateTable(
  con,
  name = "patients_medevid_waitlist",
  fields = patients_med_waitlist,
  row.names = NULL
)
dbWriteTable(
  con,
  name = "patients_medevid_waitlist",
  value = patients_med_waitlist,
  row.names = FALSE,
  append = TRUE
)
```

6.2.8 Create Partition Data

Steps for Running S2a_partitionData.R

This script creates the `partition_10` table in Postgres which consists of **usrds_id** and **subset** and adds this **subset** column to the `patients_medevid_waitlist` table. This **subset** column is the result of partitioning the number of rows (1,150,195) into 10 random subsets (numbered 0, 1, ..., 9) and assigning a patient identifier (**usrds_id**) to each subset. The purpose of partitioning the data is three-fold:

1. to ensure that there is no leakage between the training and test datasets (correctly stratify the classes)
2. to manage performance of imputation code (larger datasets require longer run times)
3. to ensure that the machine learning models are reproducible for any users (as opposed to setting the seed and using a library like caret to partition)

Note: Each subset is *approximately* 10% because it is constructed completely at random.

Input: `patients_medevid_waitlist` table from Postgres

Output: `partition_10` table in Postgres

Step 1. Define function to create **num_partitions** (10) indexed in a column named **subset** and save to Postgres as **partition_10**

```
partition_data <- function(con,
                           usrds_id,
                           num_partitions,
                           data_tablename,
                           seed_value) {

  set.seed(2539)

  randvalue = runif(
    length(usrds_id),
    min = 0,
    max = num_partitions
    )

  universe = cbind(
    usrds_id,
    floor(randvalue)) %>%
    as.data.frame()

  names(universe) = c("usrds_id", "subset")

  tblname = str_glue("partition_{num_partitions}")
  drop_table_function(con, tblname)
  dbWriteTable(
    con,
    tblname,
    universe,
    append = FALSE,
    row.names = FALSE
    )
}
```

Step 2. Import the **usrds_ids** from Postgres.

```
data_tbl = "patients_medevid_waitlist"

usrds_id = dbGetQuery(
  con,
  str_glue(
    "
    SELECT usrds_id
    FROM {data_tbl}
    ORDER BY usrds_id
    "))
usrds_id = usrds_id$usrds_id
```

Call the function defined above to create the 10 partitions.

```
partition_data(
                con,
                usrds_id,
                num_partitions = 10,
                data_tablename = data_tbl
              )
```

6.2.9 Join `patients_medevid_waitlist` Table to the Partition Index

Steps to running the S2b_join_partition_data.R script

This script joins the `patients_medevid_waitlist` table to the partition index.

Input: `patients_medevid_waitlist`

Output: `patients_medevid_waitlist`

Step 1. Define a function to import and alter the `patients_medevid_waitlist` table by adding the **subset** column, and save to Postgres.

```
join_data_partitions <- function(con,

data_tablename="patients_medevid_waitlist",
                                 num_partitions=10){

  dbSendStatement(con, str_glue(
    "
     ALTER TABLE {data_tablename}
     ADD subset integer
     "), n = -1)

  dbSendStatement(
    con,
    str_glue(
      "
        UPDATE {data_tablename} d
        SET subset = p.subset
        FROM partition_{num_partitions} p
        WHERE d.usrds_id = p.usrds_id
        "), n = -1)
}
```

Step 2. Execute the function

```
data_tbl = "patients_medevid_waitlist"

join_data_partitions(
  con,
  data_tablename = data_tbl,
```

```
  num_partitions = 10
)
```

6.2.9.1 Calculate Demographic Subtotals Per Partition

Steps to running the S2c_calculate_partition_totals.R script

This script creates a table with counts of select categories for each data partition to ensure that the partitions are representative of the entire dataset.

Input: `patients_medevid_waitlist`

Output: ./partition_totals_rev_method.csv

Step 1. Pull the data from the database and count the number of patients per partition for the following variables:

- sex = male
- race = white
- number of missing hemoglobin values
- number of missing serum creatine values
- number of missing albumin values
- number of patients who died in the first 90 days (outcome variable)

```
df = dbGetQuery(
  con,
  "
  SELECT *
  FROM patients_medevid_waitlist
  "
)

subsets_totals = df %>%
  select(subset) %>%
  group_by(subset) %>%
  count()

subsets_totals = rename(subsets_totals, c("total_pts"=n))

subsets_male = df %>%
  filter(sex==1) %>%
  select(sex, subset) %>%
  group_by(sex, subset) %>%
  count()
subsets_male <- rename(subsets_male, c("total_males"=n))

subsets_white = df %>%
  filter(race==1) %>%
  select(subset, race) %>%
  group_by(subset,race) %>%
  count()
```

```
subsets_white <- rename(subsets_white, c("total_white"=n))

subsets_heme = df %>%
  filter(is.na(heglb)==TRUE) %>%
  select(subset,heglb) %>%
  group_by(heglb, subset) %>%
  count()
subsets_heme <- rename(subsets_heme, c("total_heme_na"=n))

subsets_sercr = df %>%
  filter(is.na(sercr)==TRUE) %>%
  select(subset,sercr) %>%
  group_by(sercr, subset) %>%
  count()
subsets_sercr <- rename(subsets_sercr, c("total_sercr_na"=n))

subsets_album = df %>%
  filter(is.na(album)==TRUE) %>%
  select(subset,album) %>%
  group_by(album, subset) %>%
  count()
subsets_album <- rename(subsets_album, c("total_album_na"=n))

subsets_outcome = df %>%
  filter(died_in_90==1) %>%
  select(subset,died_in_90) %>%
  group_by(died_in_90,subset) %>%
  count()
subsets_outcome <- rename(subsets_outcome, c("total_died"=n))

dd =
  left_join(
    subsets_totals,
    subsets_outcome,
    by='subset'
)

dd = left_join(
  dd,
  subsets_male,
  by='subset'
)

dd = left_join(
  dd,
  subsets_white,
  by='subset'
)

dd = left_join(  dd,
  subsets_heme,
  by='subset'
)
```

```
dd = left_join(  dd,
  subsets_album,
  by='subset'
)

dd = left_join(  dd,
  subsets_sercr,
  by='subset'
)
write_csv(dd, "partition_totals_rev_method.csv")
```

Table: Counts of select categories for each data partition

Sub-set	Number of Males	Number of Race Group (White)	Number of Missing Hemoglobin Values	Number of Missing Serum Creatinine Values	Number of Missing Albumin Values	Total Number of Patients	Number of Patients who Died
0	65,981	76,535	17,248	2,055	35,925	114,824	8,529
1	66,131	76,864	17,108	2,051	35,129	115,050	8,773
2	66,137	76,773	17,240	2,043	35,428	115,044	8,669
3	66,031	76,846	17,406	1,937	35,100	115,027	8,426
4	66,282	76,788	16,971	1,917	34,933	114,802	8,549
5	66,042	76,652	17,285	2,008	35,138	114,936	8,671
6	66,579	77,002	17,266	1,976	35,219	115,207	8,728
7	66,332	77,221	17,266	2,035	35,019	115,557	8,695
8	66,982	76,605	17,027	2,014	34,797	114,925	8,478
9	66,033	76,751	16,847	1,936	34,973	114,823	8,565

6.2.10 Get Pre-ESRD Claims Data

The pre-ESRD claims tables in USRDS contains Medicare pre-ESRD Parts A and B, which are used to construct features for health care received prior to ESRD diagnosis.

Steps for running S3a_esrd_claims.R

This script extracts, filters, and stores the pre-ESRD claims tables from 2011-2017 for the cohort. This script uses the create_claim_table.R functions detailed in the next section.

Input: csv files are produced in script S1a-convertSAStoCSV.R

- create_claim_table.R
- pre_esrd_ip_claim_variables.R
- pre_esrd_hs_claim_variables.R

- pre_esrd_hh_claim_variables.R
- pre_esrd_op_claim_variables.R
- pre_esrd_sn_claim_variables.R

Output: The Postgres tables

```
preesrd5y_ip_clm_inc
preesrd5y_hs_clm_inc
preesrd5y_hh_clm_inc
preesrd5y_op_clm_inc
preesrd5y_sn_clm_inc
```

Step 1. Import the input file names and column types from the **pre_esrd_{xx}_claim_variables.R** scripts

The types of claims include:

- Inpatient (IP)
- Outpatient (OP)
- Home health (HH)
- Hospice (HS)
- Skilled Nursing Unit (SN)

```
source('CreateDataSet/create_claim_table.R')

claim_types = c(
  "ip",
  "hs",
  "hh",
  "op",
  "sn"
)
```

Step 2. Import and run the **create_claim_table** function for each claim type for years 2011-2017.

```
for (typ in claim_types) {
  source(str_glue("CreateDataSet/pre_esrd_{typ}_claim_variables.R"))

  create_claim_table(
    data_dir,
    con,
    filenames_esrd,
    fieldnames_esrd,
    columns_esrd,
    columns_esrd_2015,
    table_name_pt='patients_medevid_waitlist'
    )
}
```

Points to consider

Pre-ESRD claims data includes clinical as well as administrative information. Clinicians should be engaged to identify the variables in the claims data with predictive value.

6.2.11 Create Claims Tables

Steps to running the create_claim_table.R script

This script contains the functions used in **S3a_esrd_claims.R** to create the pre-ESRD claims tables. The schema for the tables changes from year to year. For example, there is no ***cdtype*** field prior to 2014, since all diagnosis codes were ICD9 prior to 2014. The script handles these year-to-year changes in schema.

Step 1. Define **create_claim_table** function

```
create_claim_table <- function(
  data_dir,
  con,
  filenames,
  fieldnames,
  column_type,
  column_type_2015,
  table_name_pt) {
  # send information to insert each year of claims data into the same
Postgres table

  fieldnames = tolower(fieldnames)
  for (filename in filenames) {
    incident_year =
      substr(filename, str_length(filename) - 3, str_length(filename))

    if (incident_year < 2015) {
      # claims prior to 2015 are all icd9, so we set cdtype to I for those
years
      csvfile = read_csv(file.path(data_dir, str_glue("{filename}.csv")),
col_types = column_type_2015)
      csvfile = csvfile %>%
        mutate(cdtype =  "I")
    }
    else {
      csvfile = read_csv(file.path(data_dir, str_glue("{filename}.csv")),
col_types = column_type)
    }

    tblname = str_remove(filename, incident_year)
    names(csvfile) = tolower(names(csvfile))
    fields = names(csvfile)

    patients = dbGetQuery(
      con,
      str_glue(
```

```
        "SELECT usrds_id
            FROM {table_name_pt}")
    )

    df = patients %>%
      inner_join(
        csvfile,
        by = "usrds_id") %>%
      mutate(
        incident_year = incident_year)

    df$pdgns_cd = df$pdgns_cd %>%
          trimws() %>%
          str_pad(.,
                  width = 7,
                  side = "right",
                  pad = "0")

    if (grepl('_ip_', tblname)){
      df = createIP_CLM(df, incident_year)
    }
    else {
      df <- df %>%
        filter(!is.na(masked_clm_from) & (masked_clm_from != ""))
  }
  # Append every set, except '2012' which will be the first table to
import.
  # this is b/c 2012 has the format that we want to use to create the
table
  # and append the other years since the format changes between 2011 and
2012-2017

    if (incident_year==2012){
      drop_table_function(con, tblname)
      print(str_glue("creating {tblname} claims using {incident_year}=
{nrow(df)}
                       patients={nrow(df %>% distinct(usrds_id,
keep_all=FALSE))}"))

      dbWriteTable(
        con,
        tblname,
        df[, fieldnames],
        append = FALSE,
        row.names = FALSE)
    }
    else {
      print(str_glue("adding {incident_year} to {tblname}={nrow(df)}
                     patients={nrow(df %>% distinct(usrds_id,
keep_all=FALSE))}"))
      dbWriteTable(
        con,
        tblname,
        df[, fieldnames],
```

```
          append = TRUE,
          row.names = FALSE)
      }
    }
  }
```

Step 2. Create a separate function **createIP_CLM** to handle the inpatient (IP) claims differently. This filters out rows with missing data.

```
createIP_CLM = function(df, incident_year) {
  # filtering for table named "preesrd5y_ip_clm"
  print(str_glue("filtering IP claims {incident_year}"))

  df = df %>%
    filter(
      !is.na(masked_clm_from) &
      (masked_clm_from != "") &
      !is.na(drg_cd) &
      (drg_cd != "")
      )
  return(df)
}
```

6.2.12 Map Diagnosis Codes (drg_cd) to Primary Diagnosis Codes (pdgns_cd)

More information about the primary diagnosis codes and aggregate categories can be found in Section 6.2.14 Diagnosis Groupings.

Steps to running the S3b_mapDrgCdToPdgnsCd.R script

Prior to 2011, there is no **pdgns_cd** (primary diagnosis code) in the USRDS pre-ESRD data. This is an issue, because we need the **pdgns_cd** in order to map a claim to one of the 12 aggregate categories. This script addresses the issue by mapping the **drg_cd** (which *is* available prior to 2011) to a **pdgns_cd**. The mapping is not one-to-one. This script therefore constructs a probability distribution for the mapping, and the **pdgns_cd** is subsequently constructed based on this probability distribution.

Input: Postgres table

```
preesrd5y_ip_clm_inc
```

Output: Postgres table

```
drg_cd_mapping
```

The script `S3a-esrd_claims.R` must be run in order to generate the data used by this script. The in-patient claims have both **drg_cd** and **pdgns_cd**. These are used as the source data for mapping **drg_cd** to **pdgns_cd**.

Step 1. Import data from the `preesrd5y_ip_clm_inc` table

```
res = dbGetQuery(
  con,
  "WITH pre_drg_pdgn AS (
                        SELECT drg_cd, pdgns_cd, COUNT(*) AS nmbr
                        FROM preesrd5y_ip_clm_inc
                        WHERE cdtype='I'
                        GROUP BY drg_cd, pdgns_cd),
          drg_cd_tbl AS (
                        SELECT drg_cd, pdgns_cd, nmbr,
                        row_number() OVER (PARTITION BY drg_cd
                                           ORDER BY nmbr DESC)
                        FROM pre_drg_pdgn
                        )
  SELECT a.drg_cd, a.pdgns_cd, a.nmbr, a.row_number, SUM(b.nmbr) AS cum
  FROM drg_cd_tbl a
    INNER JOIN drg_cd_tbl b
    ON a.drg_cd=b.drg_cd
    AND a.row_number<=b.row_number
    GROUP BY a.drg_cd, a.pdgns_cd, a.nmbr, a.row_number
    ORDER BY a.drg_cd, a.row_number"
)
```

Step 2. Aggregate table by **drg_cd**

```
bydrgcd = res %>%
  group_by(drg_cd) %>%
  dplyr::summarise(
    total = sum(as.numeric(nmbr)))
res = res %>%
  inner_join(
    bydrgcd,
    by = "drg_cd")
res = res %>%
  mutate(
        cum0 = as.numeric(cum - nmbr),
        cum = as.numeric(cum),
        lb = cum0 / total,
        ub = cum / total
)
```

Step 3. Select the columns to save.

```
drg_cd_mapping = res %>%
  select(
          drg_cd,
          pdgns_cd,
          lb,
          ub)
```

Step 4. Save to Postgres as `drg_cd_mapping`

```
drg_tblname = "drg_cd_mapping"
drop_table_function(con, drg_tblname)
dbWriteTable(con,
             drg_tblname,
             drg_cd_mapping,
             append = F,
             row.names = FALSE)
```

6.2.13 Get pre-2011 pre-ESRD Claims Data

Steps to running the S3c_esrd_claims_pre_2011.R script

Before 2011, pre-ESRD claims are stored in the files inc2008.csv, inc2009.csv, inc2010.csv. The files are organized differently from the other pre-ESRD files: the type of claim is not part of the file name (instead, it is identified in the file's contents in a field called "hcfasaf"); and the contents of the file can differ from year to year. Also, the **pdgns_cd** is not available prior to 2012. This script constructs a **pdgns_cd** from the **drg_cd** which is available prior to 2011.

Input: .csv files are produced in script S1a-convertSAStoCSV.R

- pre_esrd_pre2011_claim_variables.R

```
inc2008.csv
inc2009.csv
inc2010.csv
drg_cd_mapping
```

Output: Rows of pre-2011 claims for the cohort added to the following Postgres tables

```
preesrd5y_ip_clm_inc
preesrd5y_hh_clm_inc
preesrd5y_hs_clm_inc
preesrd5y_op_clm_inc
preesrd5y_sn_clm_inc
```

File names and column types are defined in pre_esrd_pre2011_claim_variables.R

```
source('CreateDataSet/pre_esrd_pre2011_claim_variables.R')
```

Step 1. Import the pre-2011 claims and filter on **usrds_ids** in the cohort and features in the post-2011 claims.

- Set cdtype = "I" to indicate ICD-9
- Set any missing drg_cd=000.

```
create_pre_2011 <- function(
    data_dir,
    filename,
    tblname,
    append_flag,
    table_name_pt,
    newIn2010,
    column_types){

    inc20xx = read_csv(file.path(data_dir, str_glue("{filename}.csv")),
col_types=column_types)
    incident_year =
        substr(filename, str_length(filename) - 3, str_length(filename))
    names(inc20xx) = tolower(names(inc20xx))

    patients = dbGetQuery(
        con,
        str_glue(
            "SELECT usrds_id
            FROM {table_name_pt}")
    )

    # filter on ids from the patient table
    inc20xx = inc20xx %>%
        filter(
            usrds_id %in% patients$usrds_id) %>%
        mutate(
            incident_year = incident_year,
            cdtype = "I",
            drg_cd = ifelse(
                    is.na(drg_cd), "000", drg_cd),
            drg_cd = ifelse(
                    drg_cd == "", "000", drg_cd)) %>%
        mutate(
            drg_cd = as.numeric(drg_cd))

    sortednm = names(inc20xx) %>% sort()
    inc20xx = inc20xx[, sortednm]

    if (append_flag==FALSE){
        inc20xx[, newIn2010] = NA
        drop_table_function(con, tblname)
```

```
    }
    print(nrow(inc20xx))
    dbWriteTable(
        con,
        tblname,
        inc20xx,
        append = append_flag,
        row.names = FALSE)
}
```

Step 2: For each claim type (home health - hh, hospice - hs, inpatient - ip, skilled nursing unit - sn, outpatient - op)

- Generate a uniform random number for each record in pre2011 claims, and look up **pdgns_cd** from **drg_cd_mapping** based on this random number, which will produce a **pdgns_cd** reflecting the underlying joint distribution of (**drg_cd**, **pdgns_cd**) in the data

```
get_claim_type_x <- function(claim_type, table_nm) {
    print(str_glue("get {claim_type}"))
    df = dbGetQuery(
        con,
        str_glue(
            "
            SELECT *
            FROM {table_nm}
            WHERE hcfasaf='{claim_type}'
            "))
    return(df)
}
get_distribution <- function(df){
    # Generate a uniform rv for each record in df, and look up pdgns_cd
from drg_cd_mapping
    # based on this rv, which will produce a pdgns_cd reflecting the
underlying
    # joint distribution of (drg_cd,pdgns_cd) in the data

    print("get distribution of drg_cd, pdgns_cd")
    set.seed(597)

    df$rv = runif(
        dim(df)[1]
        )
    temptablename = "temp_df"

    drop_table_function(con, temptablename)

    dbWriteTable(
        con,
        temptablename,
        df,
        temporary = TRUE
```

```
    )
    dg = dbGetQuery(
        con,
        str_glue(
            "
            SELECT a.*, b.pdgns_cd
            FROM {temptablename} a
                LEFT JOIN drg_cd_mapping b
                ON a.drg_cd = b.drg_cd
                AND a.rv <= b.ub
                AND a.rv > b.lb
            "))
    return(dg)
}
```

Step 3: Insert these rows into the main Postgres table for this claim type.

```
insert_claim_rows <- function(claim_type, pre2011_data) {
    #Get the field names to be inserted into the pre-esrd data,
    # in the correct order
    print(str_glue("intert pre 2011 {claim_type} rows into table
{nrow(pre2011_data)}"))
    main_fieldnames = names(
        dbGetQuery(
            con,
            str_glue(
                "
                SELECT *
                FROM preesrd5y_{claim_type}_clm_inc
                LIMIT 10
                ")
            )
    )

    #Set fields in main claims fieldnames that do not appear in the
pre2011 data = nan
    pre2011_data[, setdiff(main_fieldnames, names(pre2011_data))] = NA

    # Include only fields also in main_fieldnames, in the proper order
    pre2011_data = pre2011_data[, main_fieldnames]

    # append pre2011 rows to the main claims table
    main_tblname = str_glue("preesrd5y_{claim_type}_clm_inc")
    dbWriteTable(
        con,
        main_tblname,
        pre2011_data,
        append = TRUE,
        row.names = FALSE)
    }
```

Step 4. Define the wrapper function to separate into each year and claim type and save to Postgres tables.

```
source_pre_2011 <- function(data_dir, tblname, column_types) {

    newIn2010 = c(
        "dpoadmin",
        "dpodose",
        "hgb",
        "dpocash",
        "attending_phys",
        "operating_phys",
        "other_phys"
    )

    create_pre_2011(data_dir,
                    "inc2010",
                    tblname,
                    append_flag=FALSE,
                    table_name_pt = "patients_medevid_waitlist",
                    newIn2010,
                    column_types)

    create_pre_2011(data_dir,
                    "inc2009",
                    tblname,
                    append_flag=TRUE,
                    table_name_pt = "patients_medevid_waitlist",
                    newIn2010,
                    column_types)

    create_pre_2011(data_dir,
                    "inc2008",
                    tblname,
                    append_flag=TRUE,
                    table_name_pt = "patients_medevid_waitlist",
                    newIn2010,
                    column_types)

    ########BEGIN HOME HEALTH#######
    df = get_claim_type_x("H",tblname)
    dg = get_distribution(df)
    insert_claim_rows("hh", dg)
    rm(df,dg)

    ####BEGIN HOSPICE##########
    df = get_claim_type_x("S", tblname)
    dg = get_distribution(df)
    insert_claim_rows("hs", dg)
    rm(df,dg)

    ####BEGIN INPATIENT#######
    df = get_claim_type_x("I", tblname)
    dg = get_distribution(df)
```

```
    insert_claim_rows("ip", dg)
    rm(df,dg)

    ###BEGIN SKILLED NURSING####
    df = get_claim_type_x("N", tblname)
    dg = get_distribution(df)
    insert_claim_rows("sn", dg)
    rm(df,dg)

    ####BEGIN OUTPATIENT####
    df = get_claim_type_x("O", tblname)

    # Step 2: Generate a uniform rv for each record in df, and look up
pdgns_cd from drg_cd_mapping
    # based on this rv, which will produce a pdgns_cd reflecting the
underlying
    # joint distribution of (drg_cd, pdgns_cd) in the data
    set.seed(597)
    df$rv = runif(
        dim(df)[1]
        )
    temptablename = "temp_df"
    drop_table_function(con, temptablename)
    dbWriteTable(
        con,
        temptablename,
        df,
        temporary = TRUE
        )

    make_query <- function(dg_vals, temptablename){
        dg = str_glue(
                "WITH w as (
                            SELECT *
                            FROM {temptablename}
                            WHERE MOD(CAST(usrds_id AS NUMERIC),10) IN
({dg_vals})
                            )
                SELECT a.*, b.pdgns_cd
                FROM w a
                LEFT JOIN drg_cd_mapping b
                    ON a.drg_cd = b.drg_cd
                    AND a.rv <= b.ub
                    AND a.rv > b.lb"
        )
        return(dg)
    }
    dg_1 = dbGetQuery(con, make_query("0,1", temptablename))

    dg_2 = dbGetQuery(con, make_query("2,3", temptablename))

    dg_3 = dbGetQuery(con, make_query("4,5", temptablename))

    dg_4 = dbGetQuery(con, make_query("6,7", temptablename))
```

```
    dg_5 = dbGetQuery(con, make_query("8,9", temptablename))

    dg = rbind(dg_1, dg_2)
    dg = dg %>%
        rbind(dg_3) %>%
        rbind(dg_4) %>%
        rbind(dg_5)

    #step 3 append rows to main table
    insert_claim_rows("op", dg)
}
```

Step 5. Execute all functions defined above

```
source_pre_2011(data_dir,"pre_esrd_2011", columns_esrd_2015)
```

6.2.14 Diagnosis Groupings

There are several thousand primary diagnosis codes in pre-ESRD claims data, which need to be meaningfully categorized in order to create useful features. 12 major disease groups that were determined by the clinicians on the project include: diabetes, hypertension, heart failure, cardiovascular arterial disease, cerebrovascular disease, peripheral arterial disease, pneumonia, kidney failure, malignant neoplasm, smoking, alcohol dependence, and drug dependence.

Steps for running S3d_dxCodeGrouping.R

This script maps each valule in pdgns_cd column in the pre-ESRD data to one of 12 aggregated diagnosis groupings, and stores the mapping in the `dxmap` Postgres table. Two sources of input are used for the groupings: CCS (Clinical Classification System) and UCSF physician expertise.

Input:

- icd9_ccs_codes.R (for CCS groupings)
- icd10_ccs_codes.R (for CCS groupings)
- icd9_dx_2014.txt (for the icd9 pdgsn_cd)
- icd10_dx_codes.txt (for the icd10 pdgsn_cd)
- dx_mappings_ucsf.csv (for UCSF-advised categorizations of diagnosis codes)

Output:

```
dxmap
```

Step 1. Define functions

```

```

```
read_icd9 <- function(directory, filename) {
  #READ IN ICD9 SOURCE DATA
  lines = readLines(file.path(directory,filename))
  lines =
    iconv(lines[2:length(lines)],
          from = "latin1",
          to = "ASCII",
          sub = ""
          )

  #Convert utf-8 to ASCII and remove special characters like umlauts and
accents
  pdgns_cd = substr(lines, 1, 6) %>%
             trimws() %>%
             str_pad(.,
                     width = 7,
                     side = "right",
                     pad = "0"
                     )
  description = substr(lines, 7, 130)

  df9 = as.data.frame(cbind(pdgns_cd, description))
  df9$cdtype = "I"
  return(df9)
}
read_icd10 <- function(directory, filename){
  lines = readLines(file.path(directory, filename))
  lines <-
    iconv(lines[2:length(lines)],
          from = "latin1",
          to = "ASCII",
          sub = ""
          )
  pdgns_cd = substr(lines, 1, 7) %>%
             trimws() %>%
             str_pad(.,
                     width = 7,
                     side = "right",
                     pad = "0"
                     )
  description = substr(lines, 11, 130)
  df10 = as.data.frame(cbind(pdgns_cd, description), stringsAsFactors = F)
  df10 = df10 %>% filter(pdgns_cd != '0000000')
  #There may be multiple entries with the same pdgns_cd for icd10, so
choose one
  df10 = sqldf(
             "
             SELECT pdgns_cd, MAX(description) AS description
             FROM df10
             GROUP BY pdgns_cd"
             )
  df10$cdtype = "D"
  return(df10)
```

```
}
map_pdgns = function(df9, df10){
  # join icd9 and icd10
  df <- as.data.frame(rbind(df9, df10)) %>%
    mutate_at(
      vars('cdtype', 'pdgns_cd', 'description'),
      as.character
    )
  df = df %>%
    mutate(
      dx_neo = as.integer(
        grepl("malignant neoplasm", tolower(df$description)) &
          grepl("family history", tolower(df$description))
      ),
      # dx_poi=as.integer(grepl("poisoning",tolower(df$description))),
      dx_smo = as.integer((
        cdtype == 'D' & pdgns_cd %in% smo_10
      ) |(
        cdtype == 'I' & pdgns_cd %in% smo_9)
      ),
      dx_alc = as.integer((
        cdtype == 'D' & pdgns_cd %in% alc_10
      ) | (
        cdtype == 'I' & pdgns_cd %in% alc_9)
      ),
      dx_drg = as.integer((
        cdtype == 'D' & pdgns_cd %in% drg_10
      ) | (
        cdtype == 'I' & pdgns_cd %in% drg_9)
      ),
      dx_pne = as.integer((
        cdtype == 'D' & pdgns_cd %in% pne_10
      ) | (
        cdtype == 'I' & pdgns_cd %in% pne_9)
      ),
      dx_kid = as.integer((
        cdtype == 'D' & pdgns_cd %in% kid_10
      ) | (
        cdtype == 'I' & pdgns_cd %in% kid_9)
      )
    )
  return(df)
}
getComorbids <- function(directory, filename, df, colname, prefix = 'dx_')
{
  ucsf_mappings = read.csv(file.path(directory, filename),
stringsAsFactors = FALSE)
  dg = sqldf(
    "SELECT df.*, b.label
    FROM df
    LEFT JOIN ucsf_mappings b
    ON df.pdgns_cd>=b.lb
    AND df.pdgns_cd<=b.ub",
    method = "raw"
```

```
  )
  values = unique(dg[, colname]) %>% setdiff(NA)
  for (v in values) {
    dg[, paste0(prefix, v)] = (as.integer(dg[, colname] == v))
    dg[, paste0(prefix, v)] = replace_na(dg[, paste0(prefix, v)], 0)
  }
  dg$label = NULL
  return(dg)
}
```

Step 2. Execute Functions

```
df9 = read_icd9(source_dir, "icd9_dx_2014.txt")
df10 = read_icd9(source_dir, "icd10_dx_codes.txt")
mapped9_10 = map_pdgns(df9, df10)
dh = getComorbids(source_dir, "dx_mappings_ucsf.csv", df=mapped9_10,
colname = "label")
```

Step 3. Save to Postgres database as `dxmap`

```
drop_table_function(con, "dxmap")
tblname = "dxmap"
dbWriteTable(
  con,
  tblname,
  dh,
  append = FALSE,
  row.names = FALSE
  )
```

Points to consider

The primary diagnosis codes in the pre-ESRD claims should be converted with clinician's input into relevant disease groupings that can be used to create features with predictive value. It is difficult find a one-size-fits-all method for mapping diagnosis codes to meaningful categories as the categories are highly dependent on the use case. Future researchers may want to consider alternative disease groupings that are informed by clinicians and other health-care researchers.

6.2.15 Aggregate Pre-ESRD Claims Data

Steps for running S4a_pre_esrd_full.R

USRDS data have multiple pre-ESRD claims per patient. This script aggregates the data for each patient through the following steps:

1. Merge the pre-ESRD claims tables
2. Construct counts of claims grouped by type of claim and diagnosis code

3. Create one record per patient, with all pre-ESRD summary statistics aggregated for each patient
4. Create binary variables to indicate the presence or absence of pre-ESRD claims and of each type of claim (IP, HH, HS, OP, SN)

The record includes total number of claims and total length of stay, grouped by:

1. Type of claim (IP, HH, HS, OP, SN) and
2. The aggregated diagnosis grouping.

Input:

- setfieldtypes.R

```
preesrd5y_ip_clm_inc
preesrd5y_hs_clm_inc
preesrd5y_hh_clm_inc
preesrd5y_op_clm_inc
preesrd5y_sn_clm_inc
patients_medevid_waitlist
```

Output:

```
preesrdfeatures
```

Table: Number of unique patients with each type of Medicare Pre-ESRD claims

	Inpatient (IP)	**Outpatient (OP)**	**Skilled Nursing Unit (SN)**	**Home Health (HH)**	**Hospice (HS)**
Number of Unique Patients	553,704	514,926	140,417	224,272	12,482
Total Number of Claims	2,496,683	15,222,280	592,970	939,751	50,200

Step 1. Define functions for SQL queries to get claim information for 3 types of aggregations and join to the `dxmap` table.

```
prepareQuery = function(dxcols, tablename, qryAggType = 1, testMode = 0) {

  qry_pt1=paste0("b.", dxcols$column_name, collapse=",")

  if (qryAggType == 1) {
    vec1 = paste0("SUM(stay*", dxcols$column_name, ")")
    vec2 = paste0(" AS stay", substr(dxcols$column_name, 3, 6))
    qry_pt5 = paste0(vec1, vec2, collapse = ", ")

  } else if (qryAggType == 2) {
```

```
    vec1 = paste0("SUM(", dxcols$column_name, ")")
    vec2 = paste0(" AS clms", substr(dxcols$column_name, 3, 6))
    qry_pt5 = paste0(vec1, vec2, collapse = ", ")

  }  else if (qryAggType == 3) {
    vec1 = paste0("MAX(", dxcols$column_name, ")")
    vec2 = paste0(" AS has", substr(dxcols$column_name, 3, 6))
    qry_pt5 = paste0(vec1, vec2, collapse = ", ")
  }

    qry_main = str_glue("WITH w AS (
                                  SELECT a.usrds_id,
                                    a.pdgns_cd,
                                    a.masked_clm_thru-a.masked_clm_from AS
stay,
                                    a.cdtype,
                                    a.hgb,
                                    a.hcrit,
                                    {qry_pt1}
                                   FROM {tablename} a
                                   LEFT JOIN dxmap b
                                    ON a.cdtype=b.cdtype
                                    AND a.pdgns_cd=b.pdgns_cd
                                )
                      SELECT usrds_id, {qry_pt5}
                      FROM w
                      GROUP BY usrds_id"
                      )
  return(qry_main)
}
```

Step 2. Get column names

```
dxcols = names(dbGetQuery(
  con,
  "
  SELECT *
  FROM dxmap
  LIMIT 5
  "))

dxcols = dxcols[4:length(dxcols)] %>% as.data.frame()
names(dxcols) = "column_name"
```

Step 3. Send a SQL query for each type of claim and aggregation.

```
ip1 = dbGetQuery(con,prepareQuery(
                                  dxcols,
                                  "preesrd5y_ip_clm_inc",
```

```
                                    qryAggType = 1,
                                    testMode = 0
                  ))

ip2 = dbGetQuery(con,prepareQuery(
                                    dxcols,
                                    "preesrd5y_ip_clm_inc",
                                    qryAggType = 2,
                                    testMode = 0
                  ))

ip3 = dbGetQuery(con,prepareQuery(
                                    dxcols,
                                    "preesrd5y_ip_clm_inc",
                                    qryAggType = 3,
                                    testMode = 0
                  ))

op1 = dbGetQuery(con,prepareQuery(
                                    dxcols,
                                    "preesrd5y_op_clm_inc",
                                    qryAggType = 1,
                                    testMode = 0
                  ))

op2 = dbGetQuery(con,prepareQuery(
                                    dxcols,
                                    "preesrd5y_op_clm_inc",
                                    qryAggType = 2,
                                    testMode = 0
                  ))

op3 = dbGetQuery(con,prepareQuery(
                                    dxcols,
                                    "preesrd5y_op_clm_inc",
                                    qryAggType = 3,
                                    testMode = 0
                  ))

sn1 = dbGetQuery(con, prepareQuery(
                                      dxcols,
                                      "preesrd5y_sn_clm_inc",
                                      qryAggType = 1,
                                      testMode = 0
                  ))

sn2 = dbGetQuery(con, prepareQuery(
                                      dxcols,
                                      "preesrd5y_sn_clm_inc",
                                      qryAggType = 2,
                                      testMode = 0
                  ))

sn3 = dbGetQuery(con, prepareQuery(
```

```
                                    dxcols,
                                    "preesrd5y_sn_clm_inc",
                                    qryAggType = 3,
                                    testMode = 0
                  ))
```

Step 4. Calculate MAX(masked_clm_thru)-MIN(masked_clm_from) as the time range of claims for each patient.

```
prepareAggQuery = function(clm_type) {
  qry_main = str_glue("SELECT usrds_id,
                        SUM(masked_clm_thru-masked_clm_from) AS stay,
                        MAX(masked_clm_thru)-MIN(masked_clm_from) AS
range,
                        MIN(masked_clm_from) AS earliest_clm,
                        MAX(masked_clm_thru) AS latest_clm,
                        COUNT(*) AS claims
                      FROM preesrd5y_{clm_type}_clm_inc
                      GROUP BY usrds_id"
                     )
  return(qry_main)
}

hha = dbGetQuery(con, prepareAggQuery("hh"))
ipa = dbGetQuery(con, prepareAggQuery("ip"))
opa = dbGetQuery(con, prepareAggQuery("op"))
sna = dbGetQuery(con, prepareAggQuery("sn"))
hsa = dbGetQuery(con, prepareAggQuery("hs"))
```

Note: A large amount of code devoted to creating queries is not included in this guide. See the code for details.

Step 5. Get claims_range

```
df = dbGetQuery(con, qry)

earliest_cols = names(df)[grepl("earliest_clm", names(df))]
latest_cols = names(df)[grepl("latest_clm", names(df))]
for (c in earliest_cols) {
  df[, c] = ifelse(is.na(df[, c]), 500000, df[, c])
}
for (c in latest_cols) {
  df[, c] = ifelse(is.na(df[, c]), -500000, df[, c])
}

earliest_claim_date = apply(df[, earliest_cols], 1, "min")
latest_claim_date = apply(df[, latest_cols], 1, "max")
df$claims_range = latest_claim_date - earliest_claim_date
```

```
cols_to_delete = union(earliest_cols, latest_cols)
df[, cols_to_delete] = NULL
```

Out of the individual columns named "has_dx_claimtype" (e.g., "has_neo_ip") create a single column "has_dx"

```
has_cols = names(df)[grepl("has_", names(df))]
```

Step 6. Create a list of diagnosis groupings

```
dxs = unique(
  substr(
    has_cols, 5, 7))
```

Step 7. Create a binary result to yield 1 if the patients has any present, 0 if not, na if all are nans

- Example 1: has_dia_ip=NA, has_dia_op=0, has_dia_sn=1. So x=c(NA,0,1). Then returns 1
- Example 2: x=c(NA,NA,NA). Then returns NA
- Example 3: x=c(NA,0,NA). Then returns 0

```
mymax = function(x) {
  p_sum = sum(x > 0, na.rm = T) #number of positive elements
  z_sum = sum(x == 0, na.rm = T) #number of zero elements
  return(ifelse(p_sum > 0, 1, ifelse(z_sum > 0, 0, NA)))
}
```

Use this so we end up with NA if a vector is all NA

```
safe.max = function(invector) {
  na.pct = sum(is.na(invector))/length(invector)
  if (na.pct == 1) {
    return(NA) }
  else {
    return(max(invector,na.rm=TRUE))
  }
}
```

For each diagnosis grouping

```
for (c in dxs) {
  hasdxcols = has_cols[grepl(c, has_cols)]
  df[,paste0("has_",c)]=apply(
    df[,hasdxcols],
```

```
    1,
    function(x) safe.max(as.numeric(x))
    )
}

hasvars = names(df)[grepl("has_", names(df))]
hasvarsettings = hasvars[grepl("_ip$|_op$|_sn$|_hh$|_hs$", hasvars)]
df[, hasvarsettings] = NULL #remove variables like "has_neo_ip", keeping
in "has_neo"
df$claims_range = ifelse(df$claims_range < 0, NA, df$claims_range)
```

Step 8. Create a binary feature for each claim type. These are used in the parametric models instead of the detailed claim numbers.

```
df$prior_hh_care = as.integer(df$claims_hh > 0 &
                                !is.na(df$claims_hh))
df$prior_hs_care = as.integer(df$claims_hs > 0 &
                                !is.na(df$claims_hs))
df$prior_ip_care = as.integer(df$claims_ip > 0 &
                                !is.na(df$claims_ip))
df$prior_op_care = as.integer(df$claims_op > 0 &
                                !is.na(df$claims_op))
df$prior_sn_care = as.integer(df$claims_sn > 0 &
                                !is.na(df$claims_sn))
priorvars = names(df)[grepl("prior_", names(df))]

df$has_preesrd_claim = apply(
  df[, priorvars],
  1,
  function(x) safe.max(as.numeric(x))
)
```

Step 9. Save to Postgres as `preesrdfeatures`

```
drop_table_function(con, pre_esrd_tblname)
dbWriteTable(
  con,
  pre_esrd_tblname,
  df,
  field.types = myfieldtypes,
  append = FALSE,
  row.names = FALSE
)
```

6.2.16 Join the `preesrdfeatures` Tables to the Partition Index

Steps to running the S4b_join_to_partition_data.R script

Join the `preesrdfeatures` tables to our partition index.

Input:

```
preesrdfeatures
partition_10
```

Output:

```
preesrdfeatures
```

Step 1. Define a function to import and alter the `preesrdfeatures` table by adding the **subset** column, and save to Postgres.

```
join_data_partitions <- function(con,
                                 data_tablename="preesrdfeatures",
                                 num_partitions=10){

  dbSendStatement(con, str_glue(
    "
     ALTER TABLE {data_tablename}
     ADD subset integer
     "), n = -1)

  dbSendStatement(
    con,
    str_glue(
      "
        UPDATE {data_tablename} d
        SET subset = p.subset
        FROM partition_{num_partitions} p
        WHERE d.usrds_id = p.usrds_id
        "), n = -1)
}
```

Step 2. Execute the function

```
data_tbl = "preesrdfeatures"

join_data_partitions(
  con,
  data_tablename = data_tbl,
  num_partitions = 10
)
```

6.2.17 Map ICD-9 to ICD-10

Steps for S5_pdis_mapping.R

This script maps ICD-9 to ICD-10 codes and creates a table named `pdis_recode_map` which is used in *S6-Prepare Data Set*, for assigning **pdis** to a numeric value called **pdis_recode**. The 2017_I9gem_map.txt is used for this purpose.

PDIS (primary disease causing renal failure) is either the ICD-9 or ICD-10 code for the primary cause of renal failure depending on the year of the claim. Claims prior to 2015 contain ICD-9 codes whereas claims post-2016 contain ICD-10 codes. Claims from 2015 and 2016 can be either ICD-9 or ICD-10. The format for the **pdis** variable in the USRDS data is as a character variable. The ICD-9 codes were mapped to their ICD-10 equivalents to preserve the original meaning of the character variable in the numeric encoding. It was re-coded by

- Mapping all codes to their ICD-10 equivalent
- Converting them to a factor
- Typing them to a numeric

Input: The **pdis** column from `patients_medevid_waitlist` (originally comes from the `patients` table)

```
2017_I9gem_map.txt
patients_medevid_waitlist
```

Output:

```
pdis_recode_map
```

Step 1. Import the data from Postgres.

```
df1 = dbGetQuery(con,
                 "SELECT *
                  FROM patients_medevid_waitlist")
```

Whether the code is ICD-9 or ICD-10 is determined by the column **cdtype**. In order to create a map, we must get each unique **pdis** value for each **cdtype** where **cdtype** is NOT missing (NULL). Exclude entries where **cdtype** is missing (NULL). (There are 20,003 patients where **cdtype** is missing.)

```
pdis_occurrences = dbGetQuery(con,
  "SELECT cdtype, pdis, COUNT(*) AS nmbr
    FROM patients_medevid_waitlist
    WHERE cdtype IS NOT NULL
    GROUP BY pdis, cdtype"
)
```

Step 2. Standardize the format so that we can match with another pdis file.

```
pdis_occurrences$pdis = pdis_occurrences$pdis %>%
                            trimws() %>%
                            str_pad(.,
                                    width = 7,
                                    side = "right",
                                    pad = "0"
                                    )
```

Step 3. Import 2017_I9gem_map.txt

```
map_icd_9_to_10 = read.table(file = file.path(source_dir,
"2017_I9gem_map.txt"),
                             header = TRUE) %>%
                             select(icd9, icd10)
```

Step 4. Format columns

```
map_icd_9_to_10 = map_icd_9_to_10 %>%
  mutate(icd9 = icd9 %>%
                trimws() %>%
                str_pad(.,
                        width = 7,
                        side = "right",
                        pad = "0"
                        ),
          icd10 = icd10 %>%
                  trimws() %>%
                  str_pad(.,
                          width = 7,
                          side = "right",
                          pad = "0"
                        )
)
```

ICD-10: The character-level **pdis_recode** is same as **pdis** when **cdtype** equals "D" (indicating ICD-10)

```
pdis_occurrences_D = pdis_occurrences %>%
  filter(cdtype == "D") %>%
  mutate(pdis_recode_char = pdis)
```

Step 5. Use the crosswalk to map the ICD-9 codes to ICD-10

```
pdis_occurrences_I = sqldf(
  "SELECT a.*, b.icd10 AS pdis_recode_char
   FROM pdis_occurrences a
   LEFT JOIN map_icd_9_to_10 b
     ON a.pdis=b.icd9
     WHERE a.cdtype='I'",
  method = "raw"
)
```

Step 6. Concatenate the 2 maps

```
pdis_recode_map = union(pdis_occurrences_D, pdis_occurrences_I)
pdis_recode_map = pdis_recode_map %>%
  mutate(pdis_recode = as.factor(pdis_recode_char) %>% as.numeric())
```

Step 7. Calculate the sum of each recode value when recode is not missing (NaN)

```
pdis_recode_agg = pdis_recode_map %>%
  group_by(pdis_recode) %>%
  dplyr::summarise(pdis_recode_nmbr = sum(nmbr)) %>%
  as.data.frame()

pdis_recode_map = pdis_recode_map %>% left_join(pdis_recode_agg, by =
"pdis_recode")
```

Step 8. Save to Postgres as `pdis_recode_map`

```
tblname = "pdis_recode_map"
drop_table_function(con, tblname)
dbWriteTable(con,
             tblname,
             pdis_recode_map,
             append = FALSE,
             row.names = FALSE)
```

6.2.18 Prepare Data for Modeling

Steps for running S6-prepareDataSet.R script

This script creates `medxpreesrd` and uses the full dataset `patients_medevid_waitlist` and `preesrdfeatures` to construct the table `medxpreesrd` by:

- creating binary variables to indicate whether imputed values are missing or out of bounds for a given patient
- encoding character values to numeric

- counting the number of value types for como_* columns
- incorporating **pdis_recode** column
- deleting features not used for modeling

Input:

```
patients_medevid_waitlist
preesrdfeatures
pdis_recode_map
dxmap
imputation_rules.xlsx
```

Output:

```
medxpreesrd
```

Step 1. Define variables

```
subsets = "0, 1"
tablename = "patients_medevid_waitlist"
table_preesrd = "preesrdfeatures"
medex_tblname = "medxpreesrd"
```

Step 2. Import 2 partitions of data (subsets = "0, 1") from `patients_medevid_waitlist`. This code should be run 5 times (for the 10 partitions/subsets), but this example will be for 1 loop. The code contains the details to run the functions for the remaining partitions.

```
qry = str_glue(
                  "SELECT *
                  FROM {tablename}
                  WHERE subset IN ({subsets})"
                )
data_subset = dbGetQuery(con, qry)
```

For each variable in the list vars=(c("height","weight","bmi","sercr","album","gfr_epi","heglb"), introduce a binary variable for whether the variable is NA (which means "missing") and a separate binary variable for whether it is out of bounds (that is, not missing but below the clinically plausible min or above the clinically plausible max as determined by UCSF clinicians). These boundaries are imported from imputation_rules.xlsx. The function **valueExceptions** returns a data frame with usrds_id (the key field) and binary values to indicate whether or not each column in vars is NA.

```
valueExceptions = function(df, vars) {
  bounds = read_excel(str_glue("{source_dir}imputation_rules.xlsx"), sheet
```

```
=
                        "Bounds") %>% as.data.frame()
  isnavars = c()
  for (v in vars) {
    newv = str_glue("wasna_{v}")
    df[, newv] = as.integer(is.na(df[, v]))
    isnavars = c(isnavars, newv)
  }
  outofbndsvars = c()
  for (v in vars) {
    newv = str_glue("outofbnds_{v}")
    df[, newv] = as.integer(!is.na(df[, v]) &
                            !(df[, v] >= bounds[1, v] &
                                df[, v] <= bounds[2, v]))
    outofbndsvars = c(outofbndsvars, newv)
  }
  return(df[, c("usrds_id", isnavars, outofbndsvars)])
}
```

Step 3. Execute **valueExceptions**

```
labvars=c("height","weight","bmi","sercr","album","gfr_epi","heglb")
ve=valueExceptions(data_subset,labvars)
```

Join the out of bounds binary indicators for each patient to the data.

```
df=data_subset %>%
    left_join(
      ve,
      by="usrds_id")
```

Step 4. Create a function to set the values to NA if it is out of bounds.

```
setOutOfBoundsToNA=function(df,vars) {
  for (v in vars) {
    df[,v]=ifelse(
      df[,paste0("outofbnds_",v)] == 1,
      NA,
      df[,v])
  }
  return(df)
}
```

Step 5. Execute the above function on the data

```
df=setOutOfBoundsToNA(df,labvars)
oobvars = setdiff(names(df),names(data_subset))
```

Step 6. Get the list of categorical features.

```
getCategoryVars <- function(dataset){
  pattern1 = "^MEDCOV|^PATTXOP|^PATINFORMED$|^DIET|^NEPHCARE|^EPO" %>%
tolower()
  pattern2 = "^DIAL|^TYPTRN|^AVGMATURING|^AVFMATURING" %>% tolower()
  pattern3 = "^ACCESSTYPE|^TRCERT|^CDTYPE" %>% tolower()
  pattern4 = "^EMPCUR|^EMPPREV|^pdis$|^hispanic$|^COMO_" %>% tolower()
  categoryVars = names(dataset)[grepl(pattern1, names(dataset))]
  categoryVars = union(categoryVars, names(dataset)[grepl(pattern2,
names(dataset))])
  categoryVars = union(categoryVars, names(dataset)[grepl(pattern3,
names(dataset))])
  categoryVars = union(categoryVars, names(dataset)[grepl(pattern4,
names(dataset))])
  return(categoryVars)
}

categoryVars = getCategoryVars(df)
```

Step 7. Get the list of continuous features

```
getContinuousVars <- function(dataset){
  pattern_continuous =
"^GFR_EPI|^SERCR|^ALBUM|^HEGLB|^HBA1C|^BMI$|^HEIGHT|^WEIGHT" %>% tolower()
  continuousVars = names(dataset)[grepl(pattern_continuous,
names(dataset))]
  return(continuousVars)
}
continuousVars = getContinuousVars(df)
df = df[, c("usrds_id",
              "subset",
              "comorbid",
              "inc_age",
              "race",
              "sex",
              "disgrpc",
              "waitlist_status",
              "days_on_waitlist",
              "died_in_90",
              oobvars,
              categoryVars,
              continuousVars)]
```

Step 8. Get non numeric features

```
getNonNumericCols = function(dx) {
  cols = c()
  for (v in names(dx)) {
    if (!is.numeric(dx[, v])) {
      cols = c(cols, v)
    }
  }
  return(cols)
}
nonNumCols = setdiff(getNonNumericCols(df), c("pdis", "comorbid",
"cdtype","hispanic","waitlist_status"))
```

ML models typically require numeric values instead of characters or factors. The function **replaceCharacterVals** ensures that character values are replaced with a number.

```
replaceCharacterVals = function(dx,
                                vars,
                                sourceValue = c("N", "Y", "M", "F", "U",
"C", "X", "D", "I", "A", "R"),
                                sinkValue = c("2", "1", "12", "13", "9",
"15", "16", "17", "18", "20", "21"))
{
  for (v in vars) {
    print(v)
    dx[, v] = mapvalues(pull(dx, v), sourceValue, sinkValue)
    dx[, v] = as.integer(pull(dx, v))
  }
  return(dx)
}
df = replaceCharacterVals(df, nonNumCols)
```

pdis must be encoded as a number prior to ML model training.

```
recodePdis = function(df, con) {
  df$pdis = df$pdis %>%
    trimws() %>% str_pad(.,
                         width = 7,
                         side = "right",
                         pad = "0") #Format pdis with the same padding as
in pdis_recode_map
  pdis_map = dbGetQuery(
    con, "
    SELECT pdis, cdtype, pdis_recode
    FROM pdis_recode_map")

  pdis_map = pdis_map %>%
```

```
    group_by(pdis, cdtype) %>%
    dplyr::summarise(pdis_recode = min(pdis_recode))

  df = df %>%
    left_join(
      pdis_map,
      by = c("cdtype", "pdis")) %>%
    mutate(
      pdis_recode = ifelse(is.na(pdis_recode), 9999, pdis_recode)
      )

  return(df)
}
df = recodePdis(df, con)
```

Step 9. Count value types in como_* variables for each ID

```
comoEncode <- function(dataset){
    como_names = names(dataset)[grepl("^como_", names(dataset))]

    dataset$num_como_nas = apply(
      dataset[, como_names],
      1,
      function(xx)
        sum(is.na(xx))
      )
    dataset$num_como_Ns = apply(
      dataset[, como_names],
      1,
      function(xx)
        sum(xx == 2, na.rm = TRUE)
      )
    dataset$num_como_Ys = apply(
      dataset[, como_names],
      1,
      function(xx)
        sum(xx == 1, na.rm = TRUE)
      )
    dataset$num_como_Us = apply(
      dataset[, como_names],
      1,
      function(xx)
        sum(xx == 9, na.rm = TRUE)
      )
    return(dataset)
}
df = comoEncode(df)
```

Step 10. Remove features from the `medevid` table that are not needed for the ML models

- All comorbidities that are only captured on the 1995 Medical Evidence Form (and therefore before our cohort ESRD inicident years of 2008-2017), such as **como_cararr**, **como__hiv**, etc.
- All laboratory variables that have greater than 40% missingness, such as **albumin** and **hba1c**
- All year variables and masked dates, such as **incyear**, **masked_died**, etc.
- All **pdis** (primary disease causing ESRD) re-formatted variables, such as **pdis_count**, **pdis_mortality**, etc.

```
  varsToDelete = c(
    "albumlm",
    "como_ihd",
    "como_mi",
    "como_cararr",
    "como_dysrhyt",
    "como_pericar",
    "como_diabprim",
    "como_hiv",
    "como_aids",
    "comorbid_count",
    "comorbid_mortality",
    "comorbid_se",
    "comorbid",
    "ethn",
    "hba1c",
    "incyear",
    "masked_died",
    "masked_tx1fail",
    "masked_txactdt",
    "masked_txlstdt",
    "masked_txinitdt",
    "masked_remdate",
    "masked_unossdt",
    "masked_mefdate",
    "masked_ctdate",
    "masked_tdate",
    "masked_patsign",
    "masked_trstdat",
    "masked_trnend",
    "pdis_count",
    "pdis_mortality",
    "pdis_se",
    "pdis",
    "recnum",
    "tottx"
  )
df[, varsToDelete] = NULL
```

Step 11. Get the `preesrdfeatures` for the 2 subsets of data.

```
qry = str_glue(
        "SELECT *
```

```
        FROM {table_preesrd}
        WHERE subset in ({subsets})"
        )
preesrd = dbGetQuery(con, qry)
```

Step 12. Join the data with columns from `preesrdfeatures`

```
full_data = df %>%
    left_join(
      preesrd,
      by = c("usrds_id","subset")
      )
```

Step 13. Save to Postgres as `medxpreesrd`

```
dbWriteTable(
    con,
    "medxpreesrd",
    full_data,
    row.names = FALSE
          )
```

6.2.19 Impute Missing Values

Missing data have the potential to introduce bias and loss of information, which can result in invalid conclusions. Multiple imputation was chosen as the method to impute missing values over single imputation methods because it addresses the uncertainty about missing data by creating several plausible imputed datasets. For this project, multiple imputation was performed on the clinical and laboratory values (height, weight, BMI , serum creatinine, serum albumin, hemoglobin, and GFR-EPI) using the MICE (multiple imputations by chained equations) library in R (version 3.13.0), the predictive mean matching methodology for the imputations, and 5 imputations (5 datasets) to achieve 95% relative efficiency.

Steps for running S7_makeImputations.R

This script contains the code to create 5 imputations for missing data for each of the laboratory variables

```
weight
height
gfr_epi
sercr
album
```

and saves the data to Postgres as the `micecomplete_pmm` table.

The table `micecomplete_pmm` has 5 rows per **usrds_id**, for each of the imputed columns. There is one row per imputation, hence 5 rows per **usrds_id**. A modeler who wants to use imputed values would use both `medxpreesrd` and `micecomplete_pmm`, replacing weight, height, bmi, sercr, etc. in `medxpreesrd` with the imputed values in `micecomplete_pmm`. This is shown in the modeling steps.

Input:

- imputation_rules.xlsx

```
medxpreesrd
```

Output:

```
micecomplete_pmm
```

Step 1. Define the function to import the data and impute the missing values and save to Postgres as a new table. The function:

- Sets out-of-bounds values to NA so that they will be imputed
- Lists the variables to impute
- Lists the variables used to inform the imputation
- Imputes the missing values
- Calculates BMI and GFR as they are derived from other imputed variables

```
makeImputations <-
  function(con, subset, bounds, impseed, data_tablename) {
    df = dbGetQuery(
      con,
      str_glue(
        "SELECT *
        FROM {data_tablename}
        WHERE subset={subset}"
      ))

    varstoimpute = names(bounds)[2:length(names(bounds))]

    varstoimpute = c(
      "height",
      "weight",
      "bmi",
      "sercr",
      "album",
      "gfr_epi",
      "heglb"
      )

    varstouse = c(
```

```
      "inc_age",
      "race",
      "sex",
      "hispanic",
      "num_como_nas",
      "num_como_Ns",
      "num_como_Ys",
      "num_como_Us",
      "sercr",
      "height",
      "weight",
      "album",
      "heglb"
    )

    dg = df[, c("usrds_id", union(varstoimpute, varstouse))]
    dh = df[, c("usrds_id", "wasna_gfr_epi")]
    dg = dg %>%
      mutate(
        hispanic = as.factor(hispanic),
        race = as.factor(race),
        sex = ifelse(is.na(sex), 0, sex) %>%
          as.factor()
    )

    imp <- mice(dg, seed = impseed, maxit = 0)
    predictorMatrixDf = imp$predictorMatrix
    #An entry of 1 means the column variable was used to impute the row
variable
    meth = imp$method

    #row_imputed indexes the row (variable to be imputed);
    #c indexes the column (variable to use as an independent variable to
impute row_imputed)
    for (row_imputed in colnames(predictorMatrixDf)) {
      predictorMatrixDf[,row_imputed ] = 0
    }

    for (col_imputed in varstoimpute) {
      for (impute_by in varstouse) {
        if (col_imputed != impute_by)
          predictorMatrixDf[col_imputed, impute_by] = 1
      }
  }

  # bmi is arithmetically related to weight and height
  # so it needs to be handled with a separate model
  predictorMatrixDf["bmi", "height"] = 1
  predictorMatrixDf["bmi", "weight"] = 1

  for (to_use in c("usrds_id", varstouse)) {
    meth[to_use] = ""
  }
  for (to_impute in varstoimpute) {
```

```
    meth[to_impute] = "pmm"
  }
  meth["bmi"] = "~ I(weight/(.01*height)^2)"
  #Model the arithmetic relationship among bmi, weight, and height

  miceimp <-
    mice(
      dg,
      m = 5,
      maxit = 20,
      threshold = .99999,
      seed = impseed,
      predictorMatrix = predictorMatrixDf,
      method = meth,
      print =  FALSE
    )

  writeImputations(
    con,
    miceimp,
    varstoimpute,
    dh,
    subset
    )
  return(0)

}
```

Step 2. Set the seeds for each subset, import the boundary data.

```
seeds = c(2397, 3289, 4323, 4732, 691, 2388, 2688, 176, 1521, 461)
source_dir = file.path("CreateDataSet")
bounds = read_excel(file.path(source_dir, "imputation_rules.xlsx"), sheet
="Bounds"
                    ) %>% as.data.frame()
```

Step 3. Execute the makeImputations function for each subset of the data (according to the 10 partitions).

```
for (s in 0:9) {
  makeImputations(
    con,
    subset = s,
    bounds,
    impseed = seeds[s],
    data_tablename="medxpreesrd"
    )
}
```

Points to consider

1. Curating clinical and laboratory variables requires input from clinicians to remove outlier values and to properly identify relevant variables to retain in the training dataset. For example:

- Both hemoglobin and hematocrit are included in the USRDS data and are used by clinicians to identify anemia. Clinicans identified hemoglobin as the more accurate variable so hematocrit was removed from the training dataset since keeping both features results in redunant data.
- GFR-EPI is kept in the training data as it is perferred in clinical practice over GFR MDRD.

2. For smaller datasets, all features in the training dataset can be used to inform the imputation. With a dataset of over 1 million observations, using all features is time prohibitive. Variables that are rarely missing and correlate with the variables to be imputed (age, race, sex, ethnicity, number of comorbidities, and the clinical/laboratory values) were used in the imputation model for this project.

3. Variables like BMI and GFR should be passively imputed since they are derived from other imputed variables (BMI: height and weight, GFR: serum creatinine, along with age, sex, and race)

4. There are various types of imputation methods that can be selected from the 'mice' package, such as 'norm', 'pmmm', etc. Running a goodness of the imputations test by masking and then imputing known values as well as comparing the runtimes for each method will help the user select the appropriate imputation method.

5. Only the features that were missing at random (MAR) with a percent missingness <40% were imputed (i.e., clinical and laboratory values of height, weight, BMI , serum creatinine, serum albumin, hemoglobin, and GFR-EPI). Future researchers could improve the imputations by imputing features that are missing not at random (MNAR) with a more complex imputation model. Additionally other multiple imputation packages, such as the multiple imputation (MI) and Amelias packages, could also be used for the imputations.

6. Storing imputations in a database table separate from the table storing medevid, patients, and pre-ESRD data prevents the rest of the training dataset from being stored five times; it also reduces the amount of storage required for the training dataset.

6.2.20 Utility files

dx_mappings_ucsf.csv

Utility file: Used by the script S3d_dxCodeGrouping.R to produce the table `dxMap`

UCSF advised the categorizations of diagnosis codes.

2017_I9gem_map.txt

Utility file: Used by the script S5_pdis_mapping.R to produce the table `pdis_recode_map` for mapping pdis codes between ICD-9 and ICD-10.

icd10_ccs_codes.R

Utility file: Used by the script S3d_dxCodeGrouping.R to produce the table `dxMap`

Group codes related to alcohol abuse, drug abuse, pulmonary disorders, and renal failure based on the Clinical classification system (CCS) rules for grouping ICD-10 diagnosis codes.

See https://www.hcup-us.ahrq.gov/toolssoftware/ccs/ccs.jsp#download

icd10_dx_codes.txt

Utility file: Used by the script S3d_dxCodeGrouping.R to produce the table `dxMap`

Group codes related to alcohol abuse, drug abuse, pulmonary disorders, and renal failure based on the Clinical classification system (CCS) rules for grouping ICD-10 diagnosis codes.

See https://www.hcup-us.ahrq.gov/toolssoftware/ccs/ccs.jsp#download

icd9_ccs_codes.R

Utility file: Used by the script S3d_dxCodeGrouping.R to produce the table `dxMap`

Group codes related to alcohol abuse, drug abuse, pulmonary disorders, and renal failure based on the Clinical classification system (CCS) rules for grouping ICD-9 diagnosis codes.

See https://www.hcup-us.ahrq.gov/toolssoftware/ccs/ccs.jsp#download

icd9_dx_2014.txt

Utility file: Used by the script S3d_dxCodeGrouping.R to produce the table `dxMap`

Group codes related to alcohol abuse, drug abuse, pulmonary disorders, and renal failure based on the Clinical classification system (CCS) rules for grouping ICD9 diagnosis codes.

See https://www.hcup-us.ahrq.gov/toolssoftware/ccs/ccs.jsp#download

imputation_rules.xlsx

Utility used by script S7-makeImputations.R and S6-prepareDataSet.R

Contains the upper and lower bounds for clinical and laboratory variables.

Table: Upper and lower bounds for clinical and laboratory variables

Variable	Lower bound	Upper bound
Height (cm)	76	243
Weight (kg)	20	250
BMI (kg/m2)	13	75
Serum Creatinine (mg/dL)	0.5	50
Serum Albumin (g/dL)	0.5	8
GFR EPI	1	30
Hemoglobin (g/dL)	2	18

pre_esrd_ip_claim_variables.R

Utility: used by S3a_esrd_claims.R

Filenames and column types to input into S3a_esrd_claims.R to create the table for the inpatient (IP) pre-ESRD claims from 2011-2017. Pre-ESRD claims in USRDS are kept in files specific to the year. The order of these filenames is very important as the 2012 table needs to be created first in the function. The .csv files named here are produced in script S1a-convertSAStoCSV.R

Filenames:

```
"preesrd5y_ip_clm_inc2012"
"preesrd5y_ip_clm_inc2013"
"preesrd5y_ip_clm_inc2014"
"preesrd5y_ip_clm_inc2015"
"preesrd5y_ip_clm_inc2016"
"preesrd5y_ip_clm_inc2017"
"preesrd5y_ip_clm_inc2011"
```

pre_esrd_hh_claim_variables.R

Utility: used by S3a_esrd_claims.R

File names and column types to input into S3a_esrd_claims.R to create the table for the home health (HH) pre-ESRD claims from 2011-2017. Pre-ESRD claims in USRDS are kept in files specific to the year. The order of these filenames is very important as the 2012 table needs to be created first in the function. The .csv files named here are produced in script S1a-convertSAStoCSV.R

Filenames:

```
"preesrd5y_hh_clm_inc2012"
"preesrd5y_hh_clm_inc2013"
"preesrd5y_hh_clm_inc2014"
"preesrd5y_hh_clm_inc2015"
"preesrd5y_hh_clm_inc2016"
"preesrd5y_hh_clm_inc2017"
"preesrd5y_hh_clm_inc2011"
```

pre_esrd_hs_claim_variables.R

Utility: used by S3a_esrd_claims.R

Filenames and column types to input into S3a_esrd_claims.R to create the table for the hospice (HS) pre-ESRD claims from 2011-2017. Pre-ESRD claims in USRDS are kept in files specific to the year. The order of these filenames is very important as the 2012 table needs to be created first in the function. The .csv files named here are produced in script S1a-convertSAStoCSV.R

Filenames:

```
"preesrd5y_hs_clm_inc2012"
"preesrd5y_hs_clm_inc2013"
"preesrd5y_hs_clm_inc2014"
"preesrd5y_hs_clm_inc2015"
"preesrd5y_hs_clm_inc2016"
"preesrd5y_hs_clm_inc2017"
"preesrd5y_hs_clm_inc2011"
```

pre_esrd_op_claim_variables.R

Utility script: used by S3a_esrd_claims.R

Filenames and column types to input into S3a_esrd_claims.R to create the table for the outpatient (OP) pre-ESRD claims from 2011-2017. Pre-ESRD claims in USRDS are kept in files specific to the year. The order of these filenames is very important as the 2012 table needs to be created first in the function. The .csv files named here are produced in script S1a-convertSAStoCSV.R

Filenames:

```
"preesrd5y_op_clm_inc2012"
"preesrd5y_op_clm_inc2013"
"preesrd5y_op_clm_inc2014"
"preesrd5y_op_clm_inc2015"
"preesrd5y_op_clm_inc2016"
"preesrd5y_op_clm_inc2017"
"preesrd5y_op_clm_inc2011"
```

pre_esrd_sn_claim_variables.R

Utility: used by S3a_esrd_claims.R

Filenames and column types to input into S3a_esrd_claims.R to create the table for the skilled nursing (SN) pre-ESRD claims from 2011-2017. Pre-ESRD claims in USRDS are kept in files specific to the year. The order of these filenames is very important as the 2012 table needs to be created first in the function. The .csv files named here are produced in script S1a-convertSAStoCSV.R.

Filenames:

```
"preesrd5y_sn_clm_inc2012"
"preesrd5y_sn_clm_inc2013"
"preesrd5y_sn_clm_inc2014"
"preesrd5y_sn_clm_inc2015"
"preesrd5y_sn_clm_inc2016"
"preesrd5y_sn_clm_inc2017"
"preesrd5y_sn_clm_inc2011"
```

pre_esrd_pre2011_claim_variables.R

Utility: used by S3c_esrd_claims_pre_2011.R

Filenames and column types to input into S3c_esrd_claims_pre_2011.R to create the tables for the pre-ESRD claims from 2008-2010. Pre-2011 pre-ESRD claims in USRDS are kept in files specific to the year but not claim type. The .csv files named here are produced in script S1a-convertSAStoCSV.R.

Filenames:

```
inc2010
inc2009
inc2008
```

setfieldtypes.R

Used by the script S4a_pre_esrd_full.R

Utility file: The "setfieldtypes" utility is used in order to cast these column types explicitly, thereby avoiding auto assignment of "integer64" as the column data type.

6.2.21 Documentation of the Training Dataset

All features included in the training dataset are documented in the data dictionary. Features fall into two main categories:

- Features taken directly from USRDS
- Features constructed from the data in USRDS

Additionally, all features in the training dataset are labeled as operational or not operational to identify and flag "nuisance variables" in the training dataset to ensure the that the ML models do not learn on noise.

6.3 ML Modeling and Evaluation

Three ML algorithms were selected to provisionally test the training dataset: eXtreme gradient boosting (XGBoost), logistic regression (LR), and a neural network implementation--multilayer perceptron (MLP). Some of the general considerations for selecting an algorithm include characteristics of the training dataset (tabular data vs image data, number of features, etc.), algorithms that have performed well in a specific domain area (kidney disease/clinical use cases), and available computational resources (for example, deep learning algorithms require intense compute resources). The algorithms that were selected are a mixture of non-parametric (XGBoost) and parametric (LR and MLP) models.

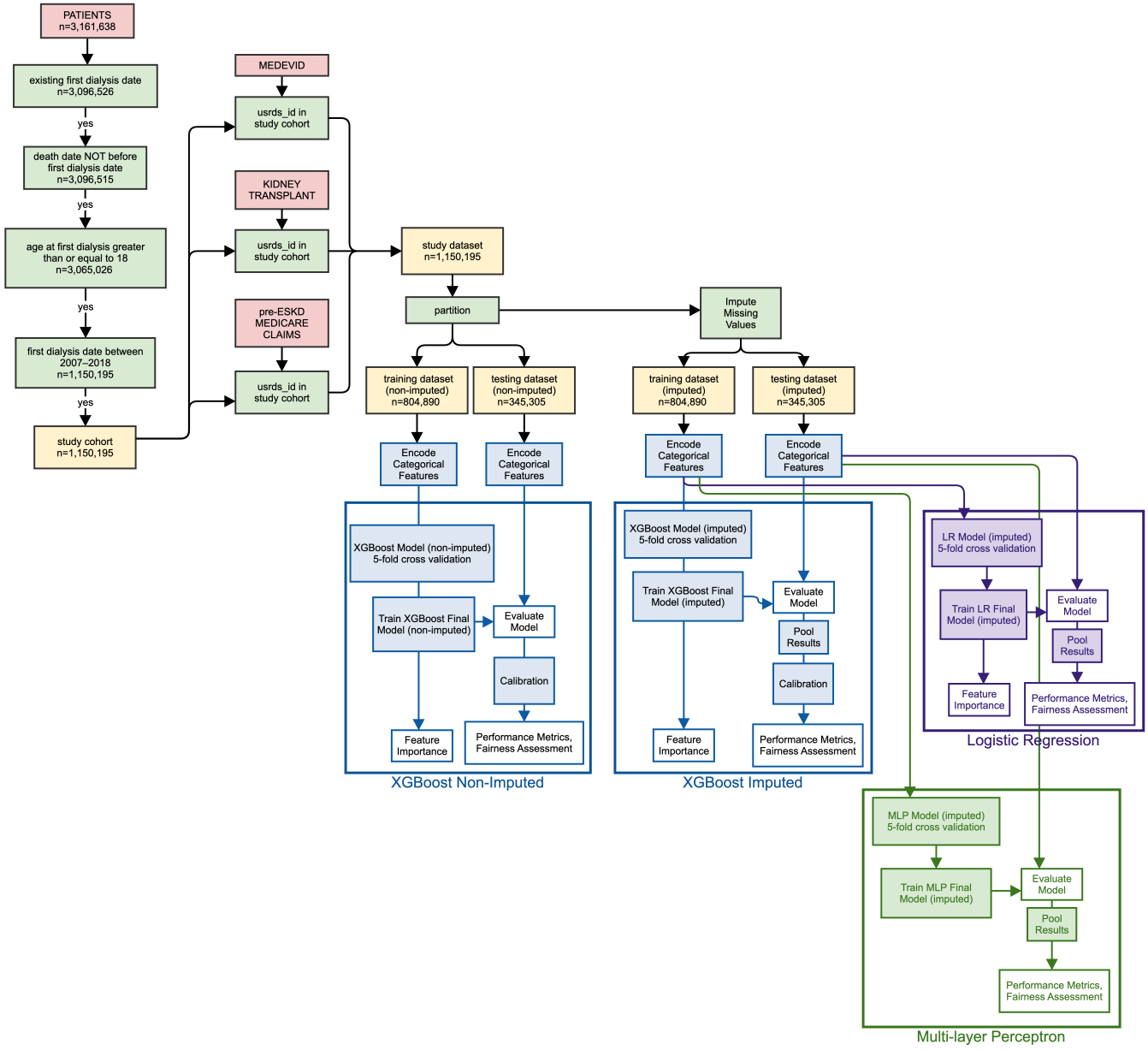
- XGBoost is a popular implementation of gradient boosted decision trees because it performs especially well for tabular data, can be applied to a wide array of use cases, data types, and desired prediction outcomes (regression vs classification), and can handle missing value natively which allows for a comparison between models run on non-imputed data versus models run on imputed data.

- LR is a classic categorization model that can be used to examine the association of (categorical or continuous) independent variable(s) with one binary dependent variable. However, it requires that the input dataset have no missing values.
- MLP is a class of hierarchical artificial neural network (ANN) that consists of at least three layers of nodes – an input layer, a hidden layer and an output layer – to carry out the process of machine learning. They are used for tabular datasets and classification prediction problems.

The training dataset derived from the raw USRDS dataset was developed by building features relevant to the use case – predicting mortality in the first 90 days of dialysis. Each feature captures information known about a patient on or prior to the date of dialysis initiation. The final structure of the training dataset, which was used to train and test the ML models, consists of approximately 200 features, and has one record per patient. Two sets of features were included in the training dataset: features taken directly from the USRDS datasets and features that were constructed. The training dataset with the full set of features were split into train and test at approximately a 70/30 ratio.

Each section below contains the code for pre-processing the data, hyperparameter tuning, final modeling, calibration, fairness assessment, and risk assessment for each model. All models were evaluated using area under the receiver operating characteristic curve (AUC ROC) and confusion matrix (true positives, true

negatives, false positives, and false negatives) evaluated at thresholds from 0.1 to 0.5.



Points to Consider

The 90-day mortality outcome was predicted using USRDS data available from patients on or prior to being diagnosed with ESRD, who progressed to ESRD. This means that the ML models predicted an outcome conditional on ESRD (e.g., the models are applicable only to those having ESRD). Future extensions of this work could merge USRDS data with EHR data to be able to predict progression to ESRD or incorporate patient-centered features from EHR data to better predict mortality in the first 90 days after dialysis initiation

6.3.1 Non-Imputed XGBoost Model

All results for the non-imputed XGBoost model are located in the /roc_auc/ directory.

Environment

The environment used for the non-imputed XGBoost model was purchased on Amazon Web Services (AWS):

```
Name: m5.4xlarge
vCPU: 16
GPU: 0
Cores: 8
Threads per core: 2
Architecture: x86_64
Memory: 64 GB
Operating System: Linux (Ubuntu 20.04.1 Focal Fossa)
Network Performance: 10 GB or less
Zone: US govcloud west
```

All sections of code for the non-imputed XGBoost model takes approximately 2 days to run.

The XGBoost models were created using R (version 3.6.3 (2020-02-29)) and the following libraries:

R library	Version
RPostgres	1.3.1
DBI	1.1.1
stringr	1.4.0
haven	2.4.0
readr	1.4.0
lubridate	1.7.9.2
dplyr	1.0.4
magrittr	1.5
tidyr	1.1.2
sqldf	0.4-11
RSQLite	2.2.3
gsubfn	0.7
proto	1.0.0
readxl	1.3.1
plyr	1.8.6
skimr	2.1.2
data.table	1.14.0
mltools	0.3.5

R library	Version
here	1.0.1
rgenoud	5.8-3.0
DiceKriging	1.5.8
purrr	0.3.4
mlrMBO	1.1.5
mlr	2.18.0
smoof	1.6.0.2
checkmate	2.0.0
ParamHelpers	1.14
xgboost	1.3.2.1
Matrix	1.2-18
rBayesianOptimization	1.1.0
rsample	0.0.9
pROC	1.17.0.1
openxlsx	4.2.3

6.3.1.1 Pre-processing the training dataset

XGBoost can only handle numeric values as inputs to the model and can natively handle missing values - so, all categorical variables were one-hot encoded into dummy variables that are binary indicators of each factor in the categorical features (e.g., the sex feature will be turned into 3 columns: sex_1 (male), sex_2 (female), sex_3 (unknown)).

Steps for running the 0_xgb_nonimputed_preprocess.R script

- Inputs:
  - `medexpressesrd`table from the Postgres database
  - category_variables.R
- Outputs:
  - universe.RData (data ready for modeling)

Step 1. Load the libraries

```
library(RPostgres) #Interface to PostgreSQL
library(DBI) #R database interface
library(dplyr)
library(tidyr)
library(skimr) #Summarizing databases
library(data.table)
```

```
library(mltools) #data.table and mltools are needed for the "one_hot"
function
library(readr)  #read rds
```

Step 2. Load the list of categorical variables

```
source(file.path("~","ONC_xgboost","category_variables.R"))
```

Step 3. Connect to the Postgres database and load the `medexpressesrd` table as the variable `universe`

The credentials required to connect to the database should be inserted in the following snippet of code below:

```
con <- dbConnect(
  RPostgres::Postgres(),
  dbname = '',
  host = '',
  port = '',
  user = '',
  password = '')
```

The data from the database is loaded into R as the variable `universe`.

```
universe=dbGetQuery(
  con,
  "SELECT *
  FROM medxpreesrd")
```

Step 4. Set numeric features as numeric type

```
num_vars = setdiff(names(universe) , categoryVars)

continuous_vars = c("height", "weight", "bmi", "sercr", "album",
"gfr_epi", "heglb")

num_vars = setdiff(num_vars, continuous_vars)
for (cc in num_vars) {
  universe[,cc]=as.numeric(universe[,cc])
}
```

Step 5. Separate categorical features from continuous and numeric to one-hot encode

```

```

```
for (c in categoryVars) {
  universe[,c]=as.factor(universe[,c])
}
universe=data.table(universe)
universe=one_hot(as.data.table(universe), naCols=TRUE, dropUnusedLevels =
TRUE)
```

Step 6. Save the pre-processed data

```
save(universe, file="universe.RData")
```

Points to consider

One-hot encoding the categorical variables is preferable to numeric encoding (casting categorical encodings as numeric) as it is a better numeric representation of ordinal variables. However, one-hot encoding increases the number of variables in the training dataset which increases run time.

6.3.1.2 Hyperparameter tuning for the non-imputed dataset

Hyperparameters were tuned on the training partitions for the non-imputed dataset with a Bayesian optimization and 5-fold cross validation to identify the optimal hyperparameters for the model. Bayesian optimization is the preferred method for hyperparameter tuning (versus grid search or random search) because it is able to find a set of hyperparameters that result in model performance equivalent to what the model performance would have been if the optimal hyperparameters had been found through exhaustive grid search. Exhaustive grid search can require testing tens of thousands of hyperparameter sets, which could be computationally infeasible or take an extraordinary amount of time. Because Bayesian optimization tests the combinations in an informed appoach, it is often able to find an optimal set of hyperparameters in only 50-100 iterations.

The best performing model was evaluated by the selecting the hyperparameter combination with the highest AUC ROC.

Steps for running the 1_xgb_nonimputed_cv.R script

- Input:
  - universe.RData
- Output:
  - 2021_xgb_cv_results_nonimputed.RData

Step 1. Load the libraries

```
library(RPostgres)
library(DBI)
library(xgboost)
library(dplyr)
library(tidyr)
library(magrittr)
```

```
library(smoof)
library(mlrMBO)  # for bayesian optimisation
library(skimr) # for summarizing databases
library(purrr) # to evaluate the loglikelihood of each parameter set in
the random grid search
library(DiceKriging)
library(rgenoud)
library(here)
library(data.table)
library(mltools) #data.table and mltools are needed for "one_hot" function
library(readr)  #read rds
```

Step 2. Load the one hot encoded data and keep only the training subsets (1-6) and separate dependent variable and non-feature columns used for identification or subsetting

```
load('universe.RData')

depvar = "died_in_90"
trainsubsets = c(0,1,2,3,4,5,6)

rhscols = setdiff(names(universe), c("usrds_id", "subset", "died_in_90"))

train_onc=universe %>% filter(subset %in% trainsubsets) %>%
as.data.frame()
```

Step 3. Generate the list of indices for 5-fold cross validation

```
# creating 5 fold validation
cv_folds = rBayesianOptimization::KFold(train_onc[, depvar],
                                        nfolds= 5,
                                        stratified = TRUE,
                                        seed= 0)
```

Step 4. Prepare the training dataset as a matrix

```
dtrain <-xgb.DMatrix(as.matrix(train_onc[, rhscols]), label = train_onc[,
depvar])
```

Step 5. Define the parameters for hyperparameter tuning

The hyperparameters that were selected for tuning and the ranges that were tuned were:

Parameter	Range
Eta	0.001 - 0.8

Parameter	Range
Gamma	0 - 9
Lambda	1 -9
Alpha	0 - 9
Max Depth	2 - 10
Minimum Child Weight	1 - 5
Number of Rounds	10 - 500
Subsample	0.2 - 1
Column Sample by Tree	0.3 - 1
Maxiumum Bin	255 - 1023

Additional information on these hyperparameters can be found at https://xgboost.readthedocs.io/en/latest/.

Parameters that were set include:

- scale_pos_weight as 3.5, which is the square root of the ratio of the negative class (survived the first 90 days of dialysis) and the positive class (died in the first 90 days of dialysis). This parameter handles the class imbalance by weighting the minority class (died in the first 90 days of dialysis).
- Number of iterations as 100. Bayesian optimization will run through 100 iterations to identify the optimal hyperparameters.
- Early stopping rounds to 15, as evaluated using the highest AUC ROC. This parameter ends model training if the AUC ROC has not increased in 15 iterations.

```
# Tune parameters ----------------------------------------------------
obj.fun  <- smoof::makeSingleObjectiveFunction(
  name = "xgb_cv_bayes",
  fn =   function(x){
    set.seed(12345)
    cv <- xgb.cv(params = list(
      booster          = "gbtree",
      scale_pos_weight = sqrt(12),
      eta              = x["eta"],
      max_depth        = x["max_depth"],
      min_child_weight = x["min_child_weight"],
      gamma            = x["gamma"],
      lambda           = x["lambda"],
      alpha            = x["alpha"],
      subsample        = x["subsample"],
      colsample_bytree = x["colsample_bytree"],
      max_bin          = x["max_bin"],
      objective        = 'binary:logistic',
      eval_metric     = "auc",
      tree_method     = "hist"),
      data=dtrain,
      nrounds = x["nround"],
```

```
      folds =  cv_folds,
      prediction = FALSE,
      showsd = TRUE,
      early_stopping_rounds = 15,
      verbose = 1)

    cv$evaluation_log[, max(test_auc_mean)]
  },
  par.set = makeParamSet(
    makeNumericParam("eta",              lower = 0.001, upper = 0.8),
    makeNumericParam("gamma",            lower = 0,     upper = 9),
    makeNumericParam("lambda",           lower = 1,     upper = 9),
    makeNumericParam("alpha",            lower = 0,     upper = 9),
    makeIntegerParam("max_depth",        lower = 2,      upper = 10),
    makeIntegerParam("min_child_weight", lower = 1,      upper = 5),
    makeIntegerParam("nround",           lower = 10,      upper = 500),
    makeNumericParam("subsample",        lower = 0.2,   upper = 1),
    makeNumericParam("colsample_bytree", lower = 0.3,   upper = 1),
    makeIntegerParam("max_bin",          lower = 255,     upper = 1023)
  ),
  minimize = FALSE
)

des = generateDesign(n=length(getParamSet(obj.fun)$pars)+1, # the number
of experiments cannot equal the number of variables therefore to increase
computation time, we are adding 1 to the total number of hyperparameters.
                     par.set = getParamSet(obj.fun),
                     fun = lhs::randomLHS)  ## . If no design is given by
the user, mlrMBO will generate a maximin Latin Hypercube Design of size 4
times the number of the black-box function's parameters.

control = makeMBOControl()
control = setMBOControlTermination(control, iters = 100) # number of
Bayesian iterations
```

Step 6. Tune the hyperparameters with Bayesian optimization and 5-fold cross-validation on the training data

```
results = mbo(fun = obj.fun,
              design = des,
              control = control,
              show.info = TRUE)
```

Step 7. Save the results to the output file

```
save(results, file = "2021_xgb_cv_results_nonimputed.RData")
```

Points to consider

1. Benchmark tests should be run on a fewer number of iterations to gauge the run-time per iteration. Hyperparameter tuning in a model with a large hyperparameter space, such as for gradient boosted decision trees, can be computationally and time intensive. This approach allows the user to estimate the time to completion for the hyperparameter tuning script.

2. Different AUC evaluation metrics can be chosen to determine the optimal set of hyperparameters, such as optimizing on precision-recall (PR) AUC or model calibration. The decision for the metric on which to optimize should be made in conjunction with clinical experts and depend on the goals of the model or study.

6.3.1.3 Final XGBoost model for the non-imputed dataset

Steps for running the 2_xgb_nonimputed_final_model.R script

The final model is trained on the training subsets (0-6) of all 5 sets of non-imputed data using the optimal hyperparameters from hyperparameter tuning. The final model is evaluated on the testing subsets (7-9) of all 5 sets of non-imputed data using the ROC AUC as well as on the confusion matrix (true positives, false positives, true negatives, and false negatives) and associated model evaluation metrics (sensitivity, specificity, positive predictive value, positive likelihood ratio, and F1 score) at 0.1-0.5 thresholds.

- Input:

```
universe.RData
2021_xgb_cv_results_nonimputed.RData
```

- Output:

```
[date]_xgbResults_onehot_nonimp.csv
2021_xgb_nonimputed_feature_importance.RData
2021_xgb_nonimputed_y_proba.csv
2021_nonimputed_predictions.xlsx
2021_xgbResults_nonimputed.csv
```

Step 1. Load the libraries

```
library(xgboost)
library(sqldf)
library(dplyr)
library(tidyr)
library(magrittr)
library(smoof)
library(mlrMBO)  # for bayesian optimisation
library(skimr) # for summarising databases
library(purrr) # to evaluate the loglikelihood of each parameter set in
the random grid search
library(DiceKriging)
library(rgenoud)
```

```
library(here)
library(data.table)
library(mltools) #data.table and mltools are needed for "one_hot" function
library(readr)  #read rds
```

Step 2. Load the one-hot encoded data and split the training subsets (0-6) from the test subsets (7-9) and separate dependent variable and non-feature columns used for identification or subsetting.

```
load("universe.RData")

depvar = "died_in_90"

trainsubsets = c(0,1,2,3,4,5,6)
testsubsets = c(7,8,9)

rhscols = setdiff(names(universe), c("usrds_id", "subset", "died_in_90"))

train_onc=universe %>% filter(subset %in% trainsubsets) %>%
as.data.frame()
train_onc = train_onc[order(train_onc$usrds_id),]

test_onc=universe %>% filter(subset %in% testsubsets) %>% as.data.frame()
test_onc = test_onc[order(test_onc$usrds_id),]
```

Step 3. Set the train and test datasets in the matrix format

```
dtrain <-xgb.DMatrix(as.matrix(train_onc[, rhscols]), label = train_onc[,
depvar])
dtest <-xgb.DMatrix(as.matrix(test_onc[, rhscols]), label = test_onc[,
depvar])
```

Step 4. Set the seed and load in the optimal hyperparameters identified during hyperparameter tuning

```
set.seed(297)

load("./roc_auc/2021_xgb_cv_results_nonimputed.RData")

xeta= results$x[['eta']]
xgamma= results$x[['gamma']]
xlambda= results$x[['lambda']]
xalpha= results$x[['alpha']]
xmax_depth= results$x[['max_depth']]
xmin_child_weight= results$x[['min_child_weight']]
xnround=results$x[['nround']]
xsubsample= results$x[['subsample']]
xcolsample_bytree= results$x[['colsample_bytree']]
```

```
xmax_bin=results$x[['max_bin']]

scenarios = as.data.frame(
  rbind(
    c(xalpha, xcolsample_bytree, xeta, xgamma, xlambda, xmax_bin,
xmax_depth, xmin_child_weight, xnround, xsubsample)
  ))
names(scenarios)=c("alpha","colsample_bytree","eta","gamma","lambda","max_
bin","max_depth",
                   "min_child_weight","rounds","subsample")

scenarios$inx = 1:dim(scenarios)[1]

watchlist <- list(eval = dtest, train = dtrain)

attr(dtrain, 'label') <- getinfo(dtrain, 'label')
dy = NULL

for (i in scenarios$inx) {
  s = scenarios[scenarios$inx == i, ]

  param <-
    list(
      max_depth = s$max_depth,
      eta = s$eta,
      nthread = 16,
      verbosity = 0,
      gamma = s$gamma,
      lambda = s$lambda,
      alpha = s$alpha,
      maximize = TRUE,
      tree_method = "hist",
      max_bin = s$max_bin,
      min_child_weight=s$min_child_weight,
      eval_metric = "auc",
      colsample_bytree=s$colsample_bytree,
      subsample=s$subsample,
      scale_pos_weight=sqrt(12),
      objective = "binary:logistic"
    )
```

Step 5. Set the seed and run the final non-imputed XGBooost model

```
set.seed(297)
starttime = proc.time()[3]
fit <-
  xgb.train(
    param,
    dtrain,
    s$rounds,
    # nthread=16,
    watchlist,
```

```
    maximize = TRUE,
    early_stopping_rounds = 15,
    verbose = 1
  )
```

Step 6: Obtain feature importance and save the file

```
feature_imp = xgb.importance(fit$feature_names,
                             model = fit)

save(feature_imp, file =
"./roc_auc/2021_xgb_nonimputed_feature_importance.RData")
```

Step 7. Save the predictions

```
dx = as.data.frame(cbind(predict(fit, newdata = dtest),
as.vector(getinfo(dtest, "label"))))
names(dx)[1:2] = c("score", "y")
dx$usrds_id = test_onc$usrds_id

write.csv(dx,file="./roc_auc/2021_xgb_nonimputed_y_proba.csv")

openxlsx::write.xlsx(as.data.frame(dx), file =
"./roc_auc/2021_nonimputed_predictions.xlsx",
                     sheetName='Sheet1', row.names=FALSE,showNA = F)
```

Step 8. Calculate the confusion matrix by each threshold value

```
outdata = as.data.frame(seq(0, .99, .01))
names(outdata) = "bin"

above_thresh = sqldf(
  "select a.bin as threshold, sum(b.y) as tp, count(b.y) as detections
  from outdata a
  left join dx b on a.bin<=b.score
  group by a.bin
  order by a.bin desc"
)

below_thresh = sqldf(
  "select a.bin as threshold, sum(b.y) as fn, count(b.y) as nondetections
  from outdata a
  left join dx b on a.bin>b.score
  group by a.bin
  order by a.bin desc"
)
```

```
perfdata = above_thresh %>% left_join(below_thresh, by = c("threshold"))
perfdata$tp = replace_na(perfdata$tp, 0)
perfdata$fn = replace_na(perfdata$fn, 0)

perfdata = perfdata %>% mutate(
  fp = detections - tp,
  tn = nondetections - fn,
  sensitivity = tp / (tp + fn),
  specificity = tn / (fp + tn),
  fpr = 1 - specificity,
  tpr = sensitivity,
  LR = sensitivity / (1 - specificity),
  ppv = tp / detections,
  npv = tn / (tn + fn)
)

perfdata$iter = i

perfdata$durationinsecs = durationinsecs

replace = T) > sample(neg.scores, 7000, replace = T))
perfdata$auc_xgb_test = max(fit$evaluation_log$eval_auc)
perfdata$auc_xgb_train = max(fit$evaluation_log$train_auc)

dy = as.data.frame(rbind(dy, perfdata))

print(paste0("Finished iteration ", i, " auc_tim_test: ",
max(perfdata$auc_xgb_test, " Duration ", durationinsecs)))
}

dy = dy %>% mutate(
accuracy = (tp + tn) / (tp + tn + fp + fn),
f1_score = 2 * ppv * sensitivity / (ppv + sensitivity)
)
```

Step 9. Save the results file

```
write.csv(dy,file="./roc_auc/2021_xgbResults_nonimputed.csv")
```

6.3.1.4 Calibration

The calibration curve shows the reliability of the model by each prediction score category, the number of patients that fall into each category, and the proportion of patients in each category who actually died in the first 90 days following dialysis initiation.

Steps for running the 3_xgb_nonimputed_calibration.ipynb script

- Input:

```
2021_xgb_nonimputed_y_proba.csv
medexpreesrd
```

- Output:

```
xgb_nonimputed_calibrated.pickle
y_calibrated_xgb_nonimputed.pickle
```

Step 1. Import libraries

```
import pandas as pd
import numpy as np
import pickle

import sys
#path to the functions file
sys.path.append('/ML-AI-PCOR-2021/onc_functions/')

# import custom functions
from plot_functions import onc_plot_calibration_curve
from calibrate_onc import calibrate_onc

#connect to posgres database
import psycopg2
import sqlalchemy
from sqlalchemy import create_engine

con = create_engine('postgresql://username:password@location/dbname')
```

Step 2. Load results from the XGBoost non-imputed model

```
pred_df = pd.read_csv('./roc_auc/2021_xgb_nonimputed_y_proba.csv')
```

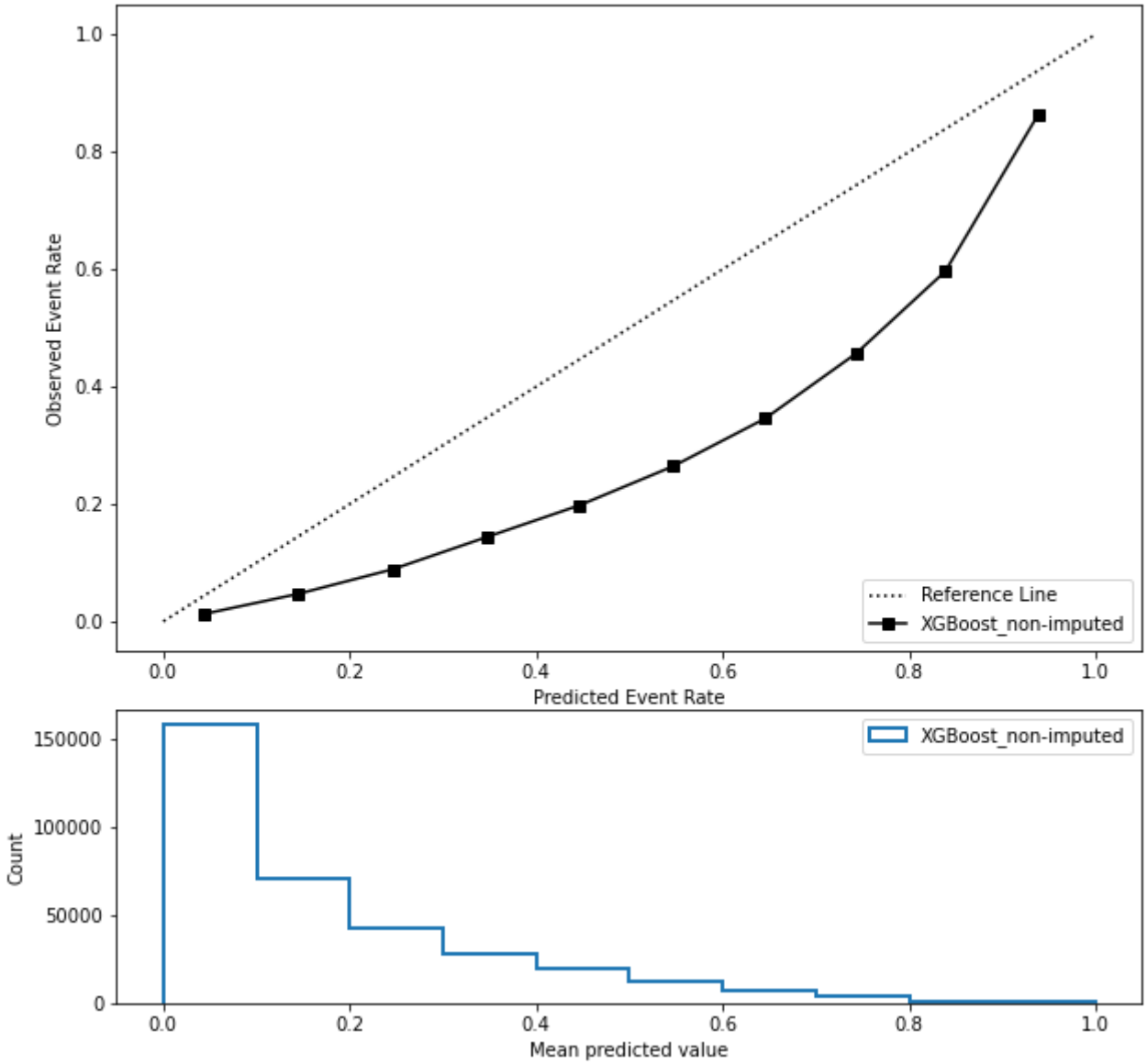
Step 3. Plot the calibration of the original model. This function **onc_plot_calibration_curve** is located in the /onc_functions/plot_functions.py file.

```
def onc_plot_calibration_curve(y_true, y_proba, label, filename):

    #calculate numbers to plot
    clf_score = brier_score_loss(y_true, y_proba, pos_label=1)
    fraction_of_positives, mean_predicted_value = \
                calibration_curve(y_true, y_proba, n_bins=10)
    # set up plot
    fig1 = plt.figure(1, figsize=(10,10))#,dpi=400)
```

```
    ax1 = plt.subplot2grid((3, 1), (0, 0), rowspan=2)
    ax2 = plt.subplot2grid((3, 1), (2, 0))

    #plot the reference for a prefectly calibrated model
    ax1.plot([0, 1], [0, 1], "k:", label="Reference Line")

    # plot the calibration curve
    ax1.plot(mean_predicted_value, fraction_of_positives, "ks-",
                    label=label)

    # plot histogram of predicted values
    ax2.hist(y_proba, range=(0, 1), bins=10, label=label,
                 histtype="step", lw=2)

    # set axes and other figure parameters
    ax1.set_ylabel("Observed Event Rate")
    ax1.set_xlabel("Predicted Event Rate")
    ax1.set_ylim([-0.05, 1.05])
    ax1.legend(loc="lower right")

    ax2.set_xlabel("Mean predicted value")
    ax2.set_ylabel("Count")
    ax2.legend(loc="upper right", ncol=1)

    plt.rc('axes', labelsize=22)    # fontsize of the x and y labels
    plt.rc('xtick', labelsize=15)    # fontsize of the tick labels
    plt.rc('ytick', labelsize=15)    # fontsize of the tick labels
    plt.rc('legend', fontsize=20)    # legend fontsize
    #save figure resolution
    plt.savefig(filename + ".png",  dpi=400,  transparent=True)
    plt.show()
```

Run the function above.

```
onc_plot_calibration_curve(
                           y_true=data.y,
                           y_proba=data.score,
                           label='XGBoost non-imputed',

filename='./roc_auc/xgb_nonimputed_orig_calibration')
```



The XGBoost model can be calibrated by training an isotonic regression on a portion of the testing set. (Model calibration is performed as probabilities of death in the first 90 days are more informative and useful for clinicians than a simple binary prediction. In order to produce valid probability estimates, predicted events rates should track observed rates across the full range of predicted risk.)

Step 4. Load the subset for each ID:

```
df = pd.read_sql_query('''
SELECT   usrds_id, subset FROM medxpreesrd;''', con)
```

Merge the subset details with the predictions.

```
data = pd.merge(pred_df, df, how="left", on="usrds_id")
```

The next steps are inside function **calibrate_onc** located in the /onc_functions/calibrate_onc.py file.

Step 5. Split the predictions from the test set (how we evaluated the model) into a test/train for the calibration (isotonic regression classifier). Split test data (subsets 7-9) into new train (7-8)/test (9) sets

```
calibration_train_set = data[((data.subset==7)|(data.subset==8))].copy()
calibration_test_set = data[data.subset==9].copy()
```

Step 6. Define the calibration model

```
ir = IsotonicRegression(out_of_bounds="clip")
```

Step 7. Fit the model to the XGBoost predictions from the (new) training set

```
ir.fit(calibration_train_set.score, calibration_train_set.y)
```

Step 8. Evaluate the model using the (new) test set

```
p_calibrated = ir.transform(calibration_test_set.score)
calibration_test_set['p_calibrated'] = p_calibrated
```

Step 9. Save

```
with open(path + 'model_calibrated_' + model_name + '.pickle', 'wb') as
picklefile:
            pickle.dump(ir,picklefile)

with open(path + 'y_calibrated_' + model_name + '.pickle', 'wb') as
picklefile:
            pickle.dump(calibration_test_set, picklefile)
```

Step 10. Print the scores from the original and calibrated model. The function **print_calibrated_results** is found in the /onc_functions/calibrate_onc.py file.

```
def print_calibrated_results(y_true, y_pred, y_calibrated):
    '''print scores for pre and post calibration'''

    acc = accuracy_score(y_true, np.round(y_pred))
    acc_calibrated = accuracy_score(y_true, np.round(y_calibrated ))
    print ("accuracy - original/calibrated:", acc, "/", acc_calibrated)

    auc = roc_auc_score(y_true, y_pred)
    auc_calibrated = roc_auc_score(y_true, y_calibrated)
```

```
    print ("ROC AUC - original/calibrated:     ", auc, "/",
auc_calibrated)

    pr = average_precision_score(y_true, y_pred)
    pr_calibrated = average_precision_score(y_true, y_calibrated )
    print ("avg precision - original/calibrated:", pr, "/", pr_calibrated)

    clf_score = brier_score_loss(y_true, y_calibrated, pos_label=1)
    print("\tBrier: %1.3f" % (clf_score))
```

Run these 2 calibration functions

```
calibrated_results = calibrate_onc(data, path='./roc_auc/',
model_name='xgb_nonimputed')
```

6.3.1.5 Plotting calibrated results

Steps for running the 4_xgb_nonimputed_calibrated_plots.ipynb script

- Input:

```
y_calibrated_xgb_nonimputed.pickle
```

- Output:

```
xgb_nonimputed_calibration.png
xgb_nonimputed_mortality_bar.png
xgb_nonimputed_roc_auc_bw.png
2021_xgb_nonimputed_calibrated_confusion_matrix.csv
```

Step 1. Import libraries

```
import pandas as pd
import numpy as np
import pickle

import sys
#add the absolute path to the onc_functions directory
sys.path.append('/ML-AI-PCOR-2021/onc_functions/')


#import custom plotting functions
from plot_functions import (onc_plot_calibration_curve,
                            onc_calc_cm,
                            onc_plot_roc,
```

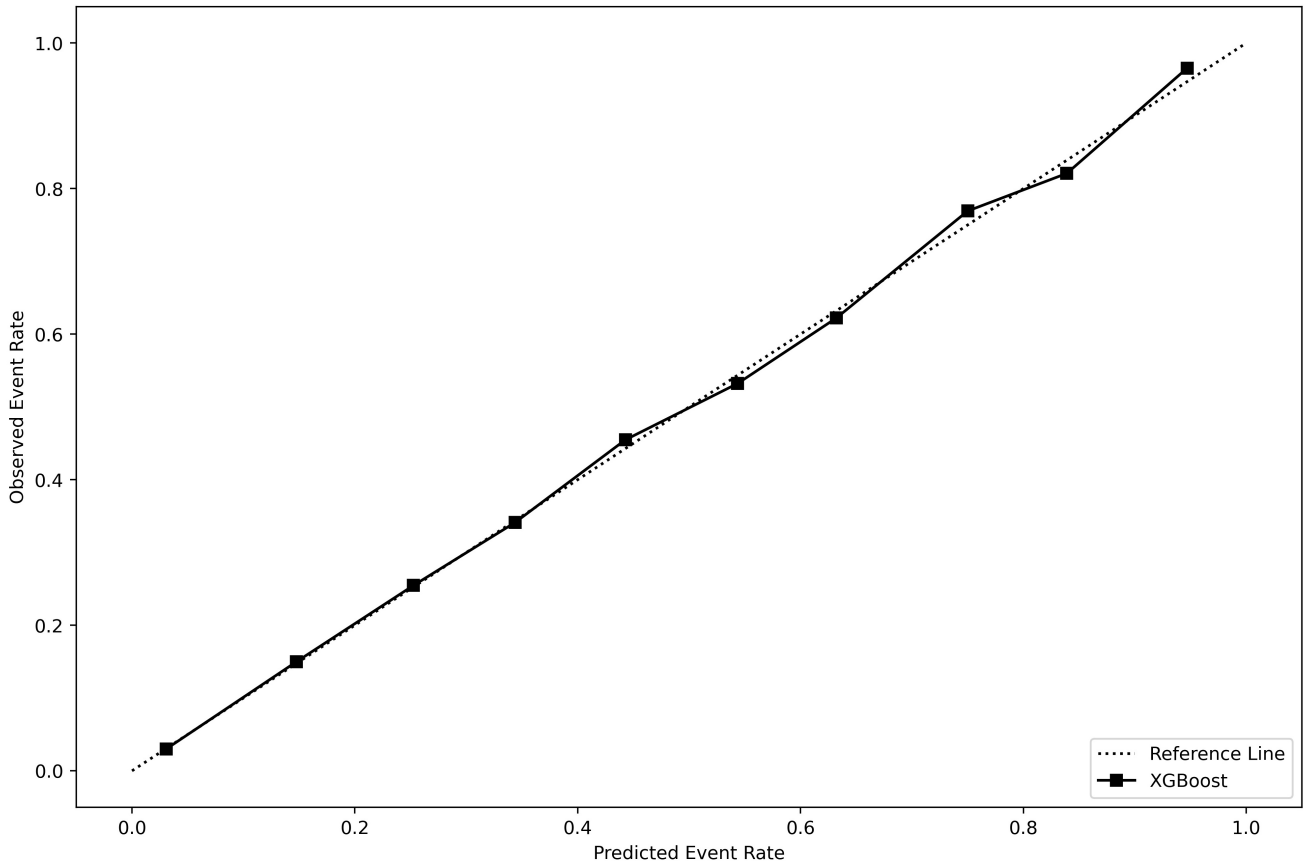
```
                                onc_plot_precision_recall,
                                onc_plot_risk,
                                onc_plot_roc_no_threshold)
```

Step 2. Load results from the calibrated model

```
with open('./roc_auc/y_calibrated_xgb_nonimputed.pickle, 'rb') as
picklefile:
            calibrated_results = pickle.load(picklefile)
```

Step 3. Plot the calibration curve of the calibrated model using the same **onc_plot_calibration_curve** function from /onc_functions/plot_functions.py

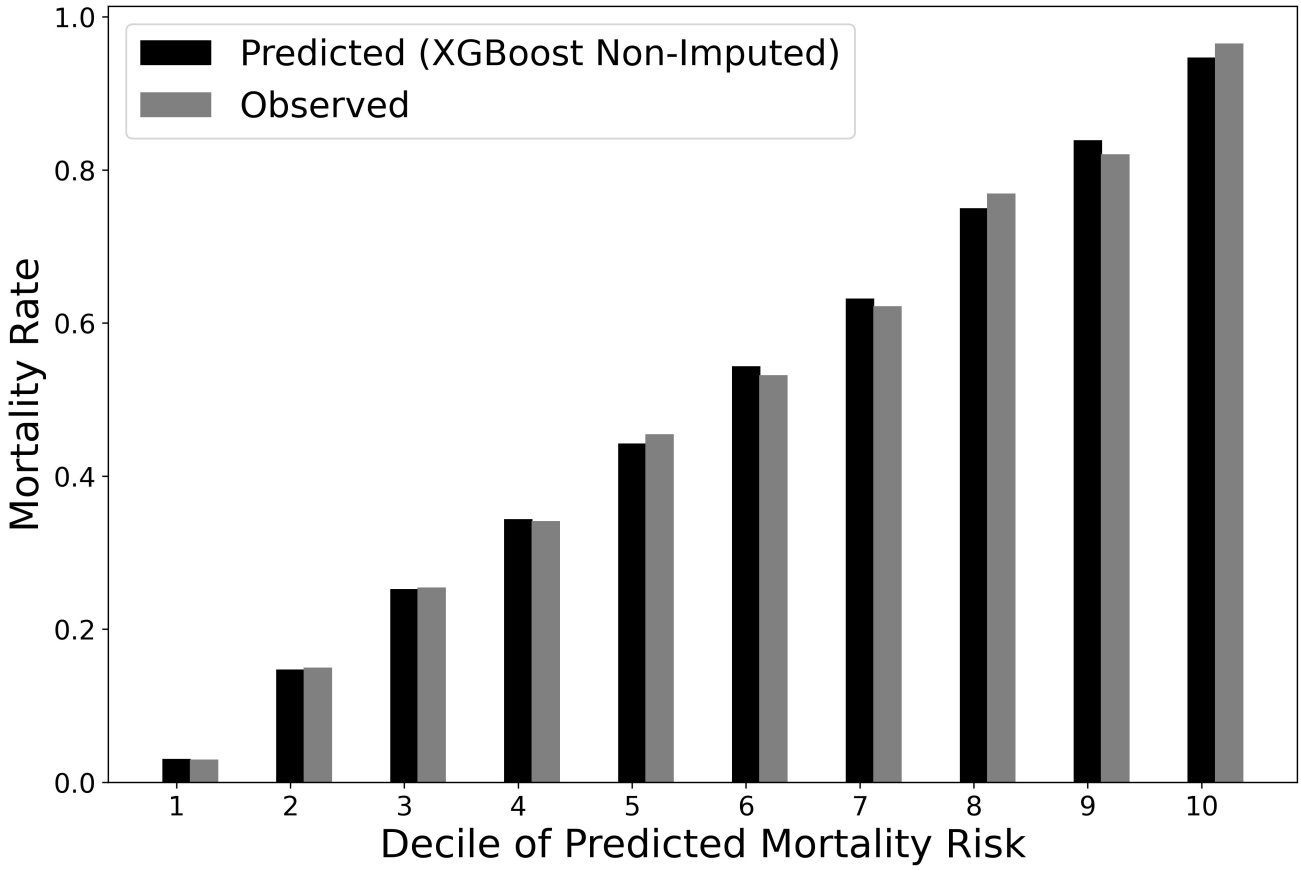
```
onc_plot_calibration_curve(
                y_true=calibrated_results.y,
                y_proba=calibrated_results.p_calibrated,
                label='XGBoost_non-imputed calibrated',
                filename='./roc_auc/xgb_nonimputed_calibrated')
```



Step 4. Plot the Risk of the calibrated model. This function **onc_plot_risk** is located and imported from /onc_functions/plot_functions.py

```
def onc_plot_risk(y_true, y_proba, label, filename):
    # calculate values for plot
    fraction_of_positives, mean_predicted_value = \
                calibration_curve(y_true, y_proba, n_bins=10)

    # set up figure params
    fig1 = plt.figure(1, figsize=(12,30),dpi=400)
    ax1 = plt.subplot2grid((7, 1), (0, 0), rowspan=2)

    # bar plot
    xs = np.arange(len(fraction_of_positives))
    ax1.bar(xs, mean_predicted_value, color='k', width = 0.25,
label=label)
    ax1.bar(xs+.25, fraction_of_positives, color='gray', width = 0.25,
label='Observed')

    #more figure settings
    plt.xticks(xs, np.arange(1, len(xs)+1, 1))
    ax1.set_ylabel("Mortality Rate")
    ax1.set_xlabel("Decile of Predicted Mortality Risk")
    ax1.legend(loc="upper left")
    plt.rc('axes', labelsize=22)    # fontsize of the x and y labels
    plt.rc('xtick', labelsize=15)    # fontsize of the tick labels
    plt.rc('ytick', labelsize=15)    # fontsize of the tick labels
    plt.rc('legend', fontsize=20)    # legend fontsize
    #save plot
    plt.savefig(filename + ".png",  dpi=400,  transparent=True)
```

```
onc_plot_risk(
  calibrated_results.y,
  calibrated_results.p_calibrated,
  label='Predicted (XGBoost Non-Imputed)',
  path='./roc_auc/',
  filename='xgb_nonimputed_mortality_bar')
```

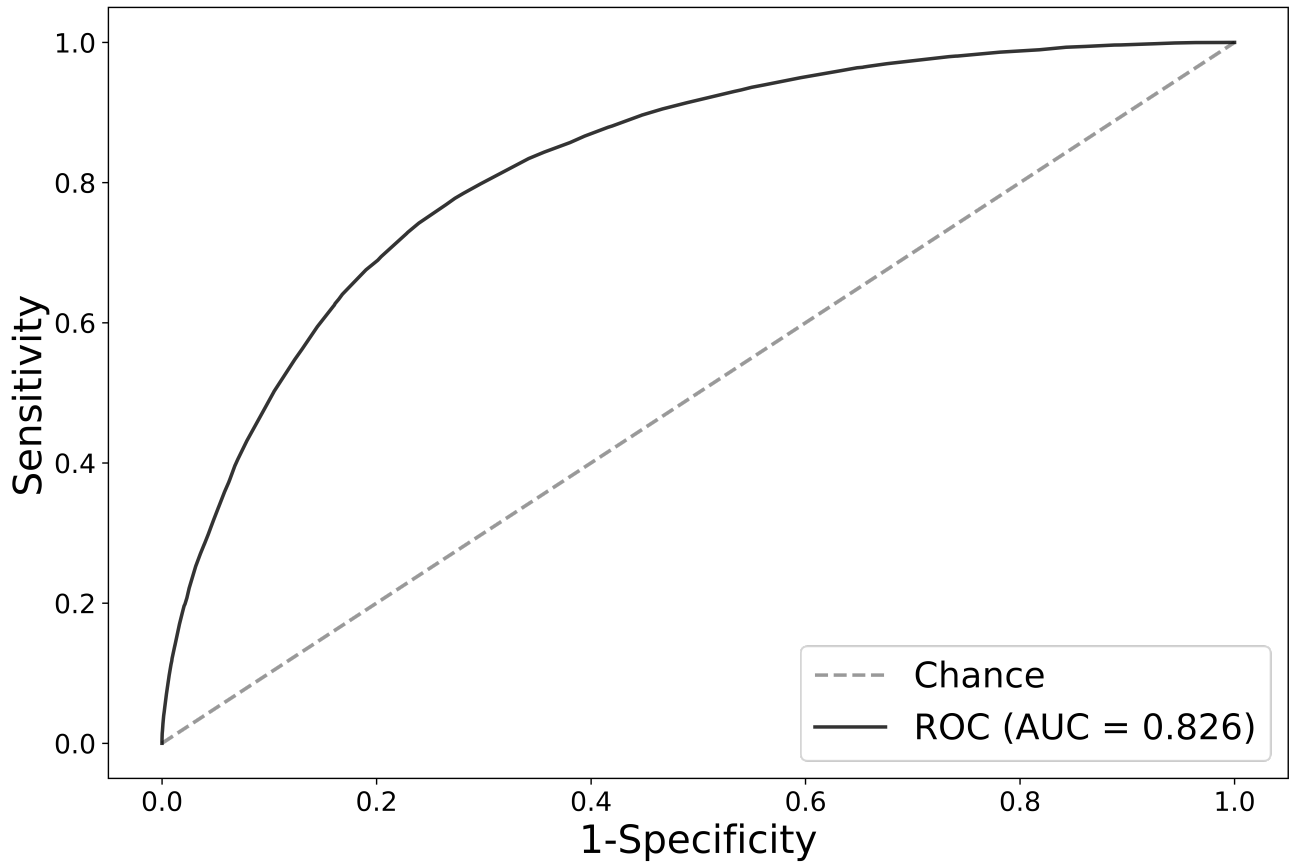


Step 5. Plot the ROC AUC of the calibrated model. This function **onc_plot_roc** is located and imported from /onc_functions/plot_functions.py

```
def onc_plot_roc(y_true, y_pred, model_name, **kwargs):
    '''
    Plot the ROC AUC and return the test ROC AUC results.
    INPUT: y_true, y_pred, model_name, **kwargs
    '''

    #calc values for plot
    false_positives, true_positives, threshold = roc_curve(y_true, y_pred)
    c_roc_auc_score = auc(false_positives, true_positives)

    #set figure params
    fig1 = plt.figure(1, figsize=(12,30),dpi=400)
    ax1 = plt.subplot2grid((7, 1), (0, 0), rowspan=2)

    #plot reference line for chance
    ax1.plot([0, 1], [0, 1], linestyle='--', lw=2, color='gray',
        label='Chance', alpha=.8)

    # plot AUC ROC
    ax1.plot(false_positives, true_positives,
        label=r'ROC (AUC = %0.3f)' % (c_roc_auc_score),
        lw=2, alpha=.8, color = 'k')

    # additional figure params
    ax1.set(xlim=[-0.05, 1.05], ylim=[-0.05, 1.05],)
```

```
    ax1.legend(loc="lower right")
    plt.xlabel('1-Specificity')
    plt.ylabel('Sensitivity')
    plt.rc('axes', labelsize=22)    # fontsize of the x and y labels
    plt.rc('xtick', labelsize=15)    # fontsize of the tick labels
    plt.rc('ytick', labelsize=15)    # fontsize of the tick labels
    plt.rc('legend', fontsize=20)    # legend fontsize
    # save plot
    plt.savefig(model_name + "_calibrated_roc_auc_bw.png",  dpi=400,
transparent=True)
    plt.show()
```

```
onc_plot_roc(
          calibrated_results.y,
          calibrated_results.p_calibrated,
          model_name='xgb_nonimputed');
```



Step 6. Save the performance metrics at multiple thresholds. The following function is imported from /onc_functions/plot_functions.py

```
def onc_calc_cm(y_true, y_predictions, range_probas=[0.1,0.5]):
    '''
    Plot the confusion matrix and scores for multiple thresholds
    '''
```

```
    df = pd.DataFrame(index = range_probas,
                      columns=['threshold','sensitivity','specificity',

'likelihood_ratio_neg','likelihood_ratio_pos',

'tp','fp','tn','fn','total_survived','total_deceased',])
    for proba_threshold in range_probas:

        cm = confusion_matrix(y_true, y_predictions > proba_threshold)
        tn = cm[0][0]
        fp = cm[0][1]

        sensitivity = recall_score(y_true, y_predictions >
proba_threshold)
        specificity = tn / (tn + fp)

        df.loc[proba_threshold, "threshold"] = proba_threshold
        df.loc[proba_threshold,"sensitivity"] = sensitivity
        df.loc[proba_threshold, "specificity"] = specificity
        df.loc[proba_threshold, "likelihood_ratio_neg"] = (1-
sensitivity)/specificity
        df.loc[proba_threshold, "likelihood_ratio_pos"] = sensitivity/(1-
specificity)
        df.loc[proba_threshold, "tp"] = cm[1][1]
        df.loc[proba_threshold, "fp"] = fp
        df.loc[proba_threshold, "tn"] = tn
        df.loc[proba_threshold, "fn"] = cm[1][0]
        df.loc[proba_threshold, "total_survived"] = np.sum(cm[0])
        df.loc[proba_threshold, "total_deceased"] = np.sum(cm[1])
    return df
```

```
cm = onc_calc_cm(
              calibrated_results.y,
              calibrated_results.p_calibrated,
              range_probas=[.10,.20, .30, .40, .50])
cm.to_csv('./roc_auc/2021_xgb_nonimputed_calibrated_confusion_matrix.csv')
```

```
cm
```

	threshold	sensitivity	specificity	likelihood_ratio_neg	likelihood_ratio_pos	tp	fp	tn	fn
0.1	0.1	0.694337	0.795667	0.384159	3.39807	5947	21712	84546	2618
0.2	0.2	0.396264	0.931967	0.647808	5.82462	3394	7229	99029	5171
0.3	0.3	0.199533	0.978364	0.818169	9.22226	1709	2299	103959	6856
0.4	0.4	0.120957	0.990589	0.887394	12.8527	1036	1000	105258	7529
0.5	0.5	0.0463514	0.997798	0.955753	21.0479	397	234	106024	8168

6.3.1.6 Saving data for the fairness assessment

Steps for running the 5_xgb_fairness_assess_get_data.ipynb script

Get the columns of data required to compute fairness assessment and save the file.

```
inc_age = age
sex
dialtyp=type of dialysis
race
hispanic
```

- Input:

```
`medexpressesrd` table from Postgres
```

- Output:

```
complete_nonimputed.pickle
```

Step 1. Import the libraries

```
import psycopg2
import sqlalchemy
from sqlalchemy import create_engine

import numpy as np
import pandas as pd
import sys
import pickle
```

Step 2. Connect to the Postgres database

The credentials required to connect to the database should be inserted in the following snippet of code below:

```
con = create_engine('postgresql://username:password@location/dbname')
```

Step 3. Import the columns required for the fairness assessment from the database

```
df = pd.read_sql_query('''SELECT   usrds_id, died_in_90, inc_age, sex,
dialtyp, race, hispanic, subset FROM medxpreesrd;''', con)
```

Step 4. Save the file

```
with open('complete_fairness_data.pickle', 'wb') as picklefile:
    pickle.dump(df, picklefile)
```

6.3.1.7 Fairness assessment

ML models can perform differently for different categories of patients, so the non-imputed XGBoost model was assessed for fairness, or how well the model performs for each category of interest (demographics—sex, race, and age—as well as initial dialysis modality). Age were binned into the following categories based on clinician input and an example in literature: 18-25, 26-35, 36-45, 46-55, 56-65, 66-75, 76-85, 86+. The USRDS predefined categories for race, sex, and dialysis modality were used for the fairness assessment.

Steps for running the 6_xgb_nonimputed_fairness.ipynb script

Calculations for specific groups of patients to assess the fairness of the final model for all patients in the test subsets. Note: Fairness assessment is run on the non-calibrated model results.

- Input:

```
complete_nonimputed.pickle
2021_xgb_nonimputed_y_proba.csv
```

- Output:

```
2021_xgb_nonimputed_fairness.csv
```

Step 1. Import libraries

```
import numpy as np
import pandas as pd
import pickle

import datetime
dte = datetime.datetime.now()
dte = dte.strftime("%Y%m%d")

# import custom functions
import sys
#path to the functions directory
sys.path.append('/ML-AI-PCOR-2021/onc_functions/')


from fairness import get_fairness_assessment
```

Step 2. Write the function that calculates AUC and the confusion matrix from the model prediction scores. This function is located and imported from the /onc_functions/fairness.py file.

```
def get_fairness_assessment(df, y_proba_col_name, y_true_col_name):

    #turn the continuous age variable into age categories
    df['agegroup'] = pd.cut(df.inc_age,
                            bins=[17, 25, 35, 45, 55, 65, 75, 85, 90],
                            labels=[1, 2, 3, 4, 5, 6, 7, 8])

    df = df.drop(columns=['inc_age'])

    #replace NaNs with a large number that does not appear in the data,
effectively creating another category for missing values
    df.loc[:,['race','dialtyp','hispanic']] = df.loc[:,
['race','dialtyp','hispanic']].fillna(100.0, axis=1).copy()

    #Identify the cols for the fairness assessment
    fairness_cols = ['agegroup', 'sex','dialtyp', 'race','hispanic']

    #loop through all categories and values to get counts, auc, and
confusion matrix
    rows_list = []
    for col in fairness_cols:
        for name, c in df.groupby(col):
            fairness_dict = {}
            fairness_dict['Feature'] = col
            fairness_dict['Value'] = name
            fairness_dict['Count'] = c.shape[0]

            fairness_dict['AUC'] = roc_auc_score(c[y_true_col_name],
c[y_proba_col_name])
            tn, fp, fn, tp = confusion_matrix(y_true = c[y_true_col_name],
                                              y_pred =
np.where(c[y_proba_col_name] >= 0.5, 1, 0)).ravel()
            fairness_dict['TN'] = tn
            fairness_dict['FP'] = fp
            fairness_dict['FN'] = fn
            fairness_dict['TP'] = tp
            rows_list.append(fairness_dict)

    #convert results from a list to a dataframe
    df_fairness = pd.DataFrame(rows_list)
    return df_fairness
```

Step 3. Load results from the model and fairness details

```
pred_df = pd.read_csv('./roc_auc/2021_xgb_nonimputed_y_proba.csv')

with open('../complete_fairness_data.pickle', 'rb') as f:
```

```
        dataset = pickle.load(f)

# merge model results with fairness details
data = pred_df.merge(dataset, how='left', on=['usrds_id'])
```

Step 4. Calculate fairness assessment

```
fairness = get_fairness_assessment(data,
                                   y_proba_col_name='score',
                                   y_true_col_name='died_in_90')
```

Step 5. Save results

```
fairness.to_csv('./roc_results/' + str(dte) +
'_xgb_nonimputed_fairness.csv')
```

	Feature	Value	Count	AUC	TN	FP	FN	TP
0	agegroup	1.0	4340	0.859782	4289	5	45	1
1	agegroup	2.0	12774	0.844446	12523	39	188	24
2	agegroup	3.0	26120	0.848271	25361	178	487	94
3	agegroup	4.0	53564	0.818192	51089	660	1548	267
4	agegroup	5.0	85076	0.799289	78955	1797	3508	816
5	agegroup	6.0	86140	0.785491	74353	4263	5370	2154
6	agegroup	7.0	62193	0.764716	46951	6974	4626	3642
7	agegroup	8.0	15098	0.748486	9194	2936	1235	1733
8	sex	1.0	198347	0.830416	173954	9746	9456	5191
9	sex	2.0	146957	0.818450	128760	7106	7551	3540
10	dialtyp	1.0	310415	0.816646	270848	15496	16115	7956
11	dialtyp	2.0	15082	0.850065	14758	44	248	32
12	dialtyp	3.0	13295	0.858981	12988	36	245	26
13	dialtyp	4.0	77	0.965753	70	3	1	3

14	dialtyp	100.0	6436	0.779859	4051	1273	398	714
15	race	1.0	230577	0.817986	196977	13823	12509	7268
16	race	2.0	93560	0.826123	85998	2552	3760	1250
17	race	3.0	3225	0.819874	3044	53	98	30
18	race	4.0	12965	0.845486	12063	325	436	141
19	race	5.0	3776	0.833047	3566	42	142	26
20	race	6.0	881	0.808297	772	48	46	15
21	race	9.0	321	0.789957	295	9	16	1
22	hispanic	1.0	51021	0.843191	47324	1198	1852	647
23	hispanic	2.0	292532	0.820216	254208	15364	15037	7923
24	hispanic	9.0	1752	0.790421	1183	290	118	161

Points to consider

Performing the fairness assessment on the categories of interest gives additional insight into how the model performs by different patient categories of interest (by demographics, etc.). Future researchers should perform fairness assessments to better evaluate model performance, especially for models that may be deployed in a clinical setting. Other methods of assessing fairness include evaluating true positives, sensitivity, positive predictive value, etc. at various threshold across the different groups of interest, which would allow selection of a threshold that balances model performance across the groups of interest.

6.3.1.8 Risk assessment

Steps for running the 7_xgb_nonimputed_risk_categories.ipynb script

Note: Risk assessment is run on the non-calibrated model results

- Input:

```
complete_fairness_data.pickle
2021_xgb_nonimputed_y_proba.csv
```

- Output:

```
2021_xgb_nonimputed_risk_cat.csv
```

Step 1. Import libraries

```
import numpy as np
import pandas as pd
import pickle

import sys
#path to the functions directory
sys.path.append('/ML-AI-PCOR-2021/onc_functions/')

from risk import get_risk_categories

print('python-' + sys.version)
import datetime
dte = datetime.datetime.now()
dte = dte.strftime("%Y%m%d")
```

Step 2. Import the details from the fairness assessment

```
with open('../complete_fairness_data.pickle', 'rb') as f:
        dataset = pickle.load(f)
```

Step 3. Import the pooled results from the model

```
pred_df = pd.read_csv('./roc_auc/2021_xgb_nonimputed_y_proba.csv')
```

Step 4. Merge the details with the results

```
data = pred_df.merge(dataset, on=['usrds_id'])
```

Step 5. Calculate risk. The function **get_risk_categories** is imported from the /onc_functions/risk.py file.

```
def get_risk_categories(dataset, y_proba_col_name, y_true_col_name):

    test_x_pd = dataset[dataset.subset > 6].copy().sort_values(by =
'usrds_id')
    del dataset

    df = test_x_pd.loc[:,[y_true_col_name,y_proba_col_name]]

    #construct the risk categories from the predicted score
    df['risk_categories'] = pd.cut(df[y_proba_col_name],
                                   bins=[-0.1, 0.09, 0.19, 0.29, 0.39,
0.49, 0.59, 0.69, 0.79, 0.89, 0.99],
                                   labels=['0-0.09', '0.1-0.19', '0.2-
0.29', '0.3-0.39', '0.4-0.49',
```

```
                                                '0.5-0.59','0.6-0.69','0.7-
0.79','0.8-0.89','0.9-0.99'])

    #loop through all the categories to get the predicted score
    risk_list = []
    for name, c in df.groupby('risk_categories'):
        risk_dict = {}
        risk_dict['Risk Category'] = name
        risk_dict['Count'] = c[y_true_col_name].shape[0]
        risk_dict['Count Died in 90'] = c[y_true_col_name].sum()
        risk_dict['Count Survived'] = c[y_true_col_name].shape[0]-
c[y_true_col_name].sum()
        risk_dict['Percent Died in 90'] =
c[y_true_col_name].sum()/c[y_true_col_name].shape[0]

        risk_list.append(risk_dict)

    df_risk = pd.DataFrame(risk_list)
    return df_risk
```

Run the function above

```
risk_cat = get_risk_categories(data,
                               y_proba_col_name='score',
                               y_true_col_name='died_in_90')
```

Step 6. Save

```
risk_cat.to_csv('./results/' + str(dte) + '_xgb_nonimputed_risk_cat.csv')
```

	Risk Category	Count	Count Died in 90	Count Survived	Percent Died in 90
0	0-0.09	148693	1702	146991	0.011446
1	0.1-0.19	75410	3256	72154	0.043177
2	0.2-0.29	44315	3722	40593	0.083990
3	0.3-0.39	29430	4064	25366	0.138090
4	0.4-0.49	20283	3893	16390	0.191934
5	0.5-0.59	13228	3399	9829	0.256955
6	0.6-0.69	7967	2679	5288	0.336262
7	0.7-0.79	4098	1827	2271	0.445827
8	0.8-0.89	1457	838	619	0.575154
9	0.9-0.99	417	351	66	0.841727

6.3.2 Imputed XGBoost Model

All results for the imputed XGBoost model are located in the /roc_results/ directory.

Environment

The environment used for the Imputed XGBoost model was purchased on Amazon Web Services (AWS):

```
Name: m5.12xlarge
vCPU: 48
GPU: 0
Cores: 24
Threads per core: 2
Architecture: x86_64
Memory: 192 GB
Operating System: Linux (Ubuntu 20.04 Focal Fossa)
Network Performance: 10 GB
Zone: US govcloud west
```

Points to consider

Hyperparameter tuning for the imputed XGBoost model required an instance with more than 65 GB of memory. When purchasing more memory we decided to upgrade the number of cores which helped to improve the computation time. The model utilizes parallel processing which used all available cores/CPUs. It took approximately 5 days to run the entire code for the imputed XGBoost model.

6.3.2.1 Pre-processing the training dataset

The data pre-processing for the imputed XGBoost model is similar to the steps for the non-imputed XGBoost model--all categorical variables were one-hot encoded into dummy variables that are binary indicators of each factor in the categorical features (e.g., the sex feature will be turned into 3 columns: sex_1 (male), sex_2 (female), sex_3 (unknown)). An additional step is required to read in the 5 imputed datasets `micecomplete_pmm` and merge with the `medxpreesrd` data.

Steps for running the 0_xgb_imputed_preprocess.R script

1. Load `medexpressesrd` table from Postgres and imputed data `micecomplete_pmm`.
2. Merge to create our 5 datasets. Left join `medexpreesrd` and the first set of imputations, keeping imputed cols from imp1, not `medxpreesrd`.
3. Categorical features get one-hot encoded.

- Input: `medexpressesrd` and `micecomplete_pmm` tables from Postgres
- Output: universe.RData (data ready for modeling)

Step 1. Load the libraries

```
library(RPostgres) #Interface to PostgreSQL
library(DBI) #R database interface
library(dplyr)
library(tidyr)
library(skimr) #Summarizing databases
library(data.table)
library(mltools) #data.table and mltools are needed for the "one_hot"
function
library(readr)  #read rds
```

Step 2. Load the list of categorical variables

```
source(file.path("~","ONC_xgboost","category_variables.R"))
```

Step 3. Connect to the Postgres database and load the `medexpressesrd` table as the variable `universe`. The credentials required to connect to the database should be inserted in the following snippet of code below:

```
con <- dbConnect(
  RPostgres::Postgres(),
  dbname = '',
  host = '',
  port = '',
  user = '',
  password = '')
```

The data from the database is loaded into R as the variable `universe`.

```
universe=dbGetQuery(
  con,
  "SELECT *
  FROM medxpreesrd")
```

Step 4. Load the `micecomplete_pmm` table as the variable `imputations_pmm`

```
imputations_pmm = dbGetQuery(
  con,
  "
  SELECT *, row_number() OVER(PARTITION BY usrds_id) AS impnum
  FROM micecomplete_pmm
  ")
```

Step 5. Left join the `medxpreesrd` and `imputations_pmm` tables and keep only the imputed columns from `imputations_pmm`

```
universe = left_join(
  medxpreesrd %>%
    select(-c("height", "weight", "bmi", "sercr", "album", "gfr_epi",
"heglb", "cdtype")),
  imputations_pmm,
  by = c("usrds_id", "subset")
)
```

Step 6. Set numeric features as numeric type

```
num_vars = setdiff(names(universe) , categoryVars)

continuous_vars = c("height", "weight", "bmi", "sercr", "album",
"gfr_epi", "heglb")

num_vars = setdiff(num_vars, continuous_vars)
for (cc in num_vars) {
  universe[,cc]=as.numeric(universe[,cc])
}
```

Step 7. Separate categorical features from continuous and numeric to one-hot encode

```
for (c in categoryVars) {
  universe[,c]=as.factor(universe[,c])
}
universe=data.table(universe)
```

```
universe=one_hot(as.data.table(universe), naCols=TRUE, dropUnusedLevels =
TRUE)
```

Step 6. Save the pre-processed data

```
save(universe, file="universe.RData")
```

Points to consider

One-hot encoding the categorical variables is preferable to numeric encoding (casting categorical encodings as numeric) as it is a better numeric representation of ordinal variables. However, one-hot encoding increases the number of variables in the training dataset which increases run time and features with more than 5 categories are typically not one-hot encoded for this reason.

6.3.2.2 Hyperparameter tuning for each imputed dataset

This section tunes the hyperparameters using Bayesian optimization and 5-fold cross validation on each imputed dataset and returns the optimal hyperparameters for each imputed dataset.

Steps for running the 1_xgb_imputed_get_hyperparams.R script

This file will run 100 Bayesian models that will:

1. Result in a new range of hyperparameters.
2. Set the dependent variable to died_in_90.
3. Drop non-feature and dependent variable cols ("usrds_id", "subset","died_in_90")
4. Create the train and test sets based on the following "subset" values

```
trainsubsets = c(0,1,2,3,4,5,6)
```

- Input:

```
universe.RData
```

- Output:

```
[date]_xgb_results_imputed_1.RData
```

Step 1. Load the libraries

```
library(xgboost)
library(dplyr)
```

```
library(tidyr)
library(magrittr)
library(smoof)
library(mlrMBO)  # for bayesian optimisation
library(skimr) # for summarising databases
library(purrr) # to evaluate the loglikelihood of each parameter set in
the random grid search
library(DiceKriging)
library(rgenoud)
library(data.table)
library(mltools) #data.table and mltools are needed for "one_hot" function
library(readr)  #read rds
library(rBayesianOptimization)
library(Matrix)
```

Step 2. Load the one hot encoded data and keep only the training subsets (1-6).

```
load('universe.RData')

depvar = "died_in_90"
trainsubsets = c(0,1,2,3,4,5,6)
```

Step 3. Initiate a list to hold the hyperparameter tuning results from each of the 5 imputed datasets

```
model_results <-list()
```

Step 4. Loop through each imputation

```
for(i in 1:5){
____
}
```

For each imputation perform the following steps:

Step 5. Set the training dataset

```
  train=universe %>%
    filter(subset %in% trainsubsets, impnum == i) %>% as.data.frame()
```

Step 6. Generate the list of indices for 5-fold cross validation

```
cv_folds = rBayesianOptimization::KFold(train[, depvar], # creating 5 fold
validation
```

```
                                                nfolds= 5,
                                                stratified = TRUE,
                                                seed = 0)
```

Step 7. Set the training dataset as numeric and prepare the training dataset as a matrix

```
train[] <- lapply(train, as.numeric) #force to numeric columns

options(na.action='na.pass')
trainm <- sparse.model.matrix(died_in_90 ~ ., data = train[, rhscols])
dtrain <- xgb.DMatrix(data = trainm, label=train[, depvar])
```

Step 8. Define the parameters for hyperparameter tuning

The hyperparameters that were selected for tuning and the ranges that were tuned were:

Parameter	**Range**
Eta	0.001 - 0.8
Gamma	0 - 9
Lambda	1 -9
Alpha	0 - 9
Max Depth	2 - 10
Minimum Child Weight	1 - 5
Number of Rounds	10 - 500
Subsample	0.2 - 1
Column Sample by Tree	0.3 - 1
Maxiumum Bin	255 - 1023

Additional hyperparameters can be found at https://xgboost.readthedocs.io/en/latest/.

Parameters that were set include:

- Scale_pos_weight as 3.5, which is the square root of the ratio of the negative class (survived the first 90 days of dialysis) and the positive class (died in the first 90 days of dialysis). This parameter handles the class imbalance by weighting the minority class (died in the first 90 days of dialysis).
- Number of iterations as 100. Bayesian optimization will run through 100 iterations to identify the optimal hyperparameters.
- Early stopping rounds to 15, as evaluated using the highest ROC AUC. This parameter ends model training if the ROC AUC has not increased in 15 iterations.

```
# Tune parameters ----------------------------------------------------
obj.fun  <- smoof::makeSingleObjectiveFunction(
  name = "xgb_cv_bayes",
  fn =   function(x){
    set.seed(12345)
    cv <- xgb.cv(params = list(
      booster          = "gbtree",
      scale_pos_weight = sqrt(12),
      eta              = x["eta"],
      max_depth        = x["max_depth"],
      min_child_weight = x["min_child_weight"],
      gamma            = x["gamma"],
      lambda           = x["lambda"],
      alpha            = x["alpha"],
      subsample        = x["subsample"],
      colsample_bytree = x["colsample_bytree"],
      max_bin          = x["max_bin"],
      objective        = 'binary:logistic',
      eval_metric     = "auc",
      tree_method     = "hist"),
      data=dtrain,
      nrounds = x["nround"],
      folds =  cv_folds,
      prediction = FALSE,
      showsd = TRUE,
      early_stopping_rounds = 15,
      verbose = 1)

    cv$evaluation_log[, max(test_auc_mean)]
  },
  par.set = makeParamSet(
    makeNumericParam("eta",              lower = 0.001, upper = 0.8),
    makeNumericParam("gamma",            lower = 0,     upper = 9),
    makeNumericParam("lambda",           lower = 1,     upper = 9),
    makeNumericParam("alpha",            lower = 0,     upper = 9),
    makeIntegerParam("max_depth",        lower = 2,      upper = 10),
    makeIntegerParam("min_child_weight", lower = 1,      upper = 5),
    makeIntegerParam("nround",           lower = 10,      upper = 500),
    makeNumericParam("subsample",        lower = 0.2,   upper = 1),
    makeNumericParam("colsample_bytree", lower = 0.3,   upper = 1),
    makeIntegerParam("max_bin",          lower = 255,     upper = 1023)
  ),
  minimize = FALSE
)

des = generateDesign(n=length(getParamSet(obj.fun)$pars)+1, # the number
of experiments cannot equal the number of variables therefore to increase
computation time, we are adding 1 to the total number of hyperparameters.
                     par.set = getParamSet(obj.fun),
                     fun = lhs::randomLHS)  ## . If no design is given by
the user, mlrMBO will generate a maximin Latin Hypercube Design of size 4
times the number of the black-box function's parameters.
```

```
control = makeMBOControl()
control = setMBOControlTermination(control, iters = 100) # number of
Bayesian iterations
```

Step 8. Tune the hyperparameters with Bayesian optimization and 5-fold cross-validation on the training data

```
results = mbo(fun = obj.fun,
              design = des,
              control = control,
              show.info = TRUE)
```

Step 9. Save the results to the list `model_results`

```
model_results[[i]] <- results
```

This is the end of the loop. To return the best hyperparameters for model i, use the following line of code

```
model_results[[i]]$x
```

Step 10. Save the output to a file

```
save(model_results, file = "2021_xgb_results_imputed_1.RData")
```

Points to consider

Bayesian optimization was used to narrow down the hyperparameter ranges to consider for a pooled approach to hyperparameter tuning. Using Bayesian optimization to limit the hyperparameter space reduced the time required to run a pooled approach for hyperparameter tuning for the imputed dataset.

6.3.2.3 Pooled Hyperparameter Tuning

This script runs a random cross validated search on this new range using a pooled approach for all 5 imputed datasets and takes the single set of hyperparameters from the best AUC and run it with the validation data. The "best" single combination of hyperparameters resulting from this script was fed into 5 individual models for each imputed dataset resulting in 5 predictions averaged from each imputed dataset that was used to compute AUC.

Steps for running the 2_xgb_imputed_gridsearch_cv.R script

- Input:

```
universe.RData
```

- Output:

```
2021_pooling_sample.RData
2021_final_hp_results_random_grid_imputed.xlsx
2021_final_hp_results_random_grid_imputed.RData
```

Step 1. Load the libraries

```
library(pROC)
library(rsample)
library(xgboost)
library(sqldf)
library(dplyr)
library(tidyr)
library(magrittr)
library(smoof)
library(mlrMBO)  # for bayesian optimisation
library(skimr) # for summarising databases
library(purrr) # to evaluate the loglikelihood of each parameter set in
the random grid search
library(DiceKriging)
library(rgenoud)
library(data.table)
library(mltools) #data.table and mltools are needed for "one_hot" function
library(readr)  #read rds
library(rBayesianOptimization)
library(openxlsx)
library(Matrix)
```

Step 2. Load the one hot encoded data and keep only the training subsets (1-6).

```
load("~/universe.RData")
depvar = "died_in_90"
rhscols = setdiff(names(universe), c("usrds_id", "subset", "cdtype"))

trainsubsets = c(0,1,2,3,4,5,6)
```

Step 3. Set the seed and the hyperparameter grid from the narrowed ranges from Step 2

Set the seed.

```
set.seed(123)
```

25 iterations will be run for the pooled hyperparameter tuning approach.

```
how_many_models <- 25
```

Set the updated ranges for each hyperparameter based on the results from Bayesian optimization and randonly generate 25 values for each hyperparameter.

```
eta <-              data.frame(eta = runif(how_many_models,min =
0.04852942, max = 0.08619335))
gamma <-            data.frame(gamma = runif(how_many_models,min =
0.766442, max = 6.013658))
lambda <-           data.frame(lambda = runif(how_many_models,min =
5.845102, max = 8.751962))
alpha <-            data.frame(alpha = runif(how_many_models,min =
6.516213, max = 8.719468))
max_depth <-        data.frame(max_depth = sample(6:7, how_many_models,
replace=TRUE))
min_child_weight <- data.frame(min_child_weight = sample(1:4,
how_many_models, replace=TRUE))
nround <-          data.frame(nround = sample(419:499, how_many_models,
replace=TRUE))
subsample <-        data.frame(subsample = runif(how_many_models,min =
0.7314413, max = 0.8471972))
colsample_bytree <- data.frame(colsample_bytree =
runif(how_many_models,min = 0.5921707, max = 0.8566342))
max_bin <-          data.frame(max_bin = sample(529:972, how_many_models,
replace=TRUE))
```

Initiate the hyperparameter grid.

```
random_grid <-eta %>%
  bind_cols(gamma) %>%
  bind_cols(lambda) %>%
  bind_cols(alpha) %>%
  bind_cols(max_depth) %>%
  bind_cols(min_child_weight) %>%
  bind_cols(nround)  %>%
  bind_cols(subsample) %>%
  bind_cols(colsample_bytree) %>%
  bind_cols(max_bin) %>%as_tibble()


df.params <- bind_rows(random_grid) %>%
  mutate(rownum = row_number(),
         model = row_number())
list_of_param_sets <- df.params %>% nest(-rownum)

colnames(list_of_param_sets) <- c("model","hyperparamters")
```

Step 4. Prepare the training dataset

```
  filter(subset <=6 ) %>% as.data.frame()
```

Step 5. Loop through each imputation

```
for(i in 1:5){
----
}
```

For each imputation perform the following steps:

Step 6. Set the training dataset

```
train_onc=train_full %>%
  filter(impnum == i) %>% as.data.frame()
```

Step 7. Sort the training dataset by `usrds_id`

```
train_onc = train_onc[order(train_onc$usrds_id),] #We sort the data to
make sure an usrsd_id will always end up in the training or validation
regardless of which imputed dataset we are using
```

Step 8. Remove columns containing all NAs

```
all_na <- function(x) any(!is.na(x)) #creating function that removes
columns containing all NAs
train_onc <- train_onc %>% select_if(all_na) #removing the columns
containing all NAs
```

Step 9. Set all columns to numeric and set the seed

```
train_onc[] <- lapply(train_onc, as.numeric) #force to numeric columns

set.seed(2369)
```

Step 10. Set the train/test split (70% for training and 30% for validation)

```
tr_te_split <- rsample::initial_split(train_onc, prop = 7/10) #70% for
training, 30% for validation/test
train_onc <- rsample::training(tr_te_split) %>% as.data.frame()
test_onc  <- rsample::testing(tr_te_split) %>% as.data.frame()
```

Step 11. Prepare training and validation/test set for modeling

```
#per https://stackoverflow.com/questions/48805977/r-missing-data-causes-
error-with-xgboost-sparse-model-matrix
options(na.action='na.pass')
trainm <- sparse.model.matrix(died_in_90 ~ ., data = train_onc[,
c(rhscols,"died_in_90")])
dtrain <- xgb.DMatrix(data = trainm, label=train_onc[, depvar])

testm <- sparse.model.matrix(died_in_90 ~ ., data = test_onc[,
c(rhscols,"died_in_90")])
dtest <- xgb.DMatrix(data = testm, label=test_onc[, depvar])
watchlist <- list(train = dtrain, eval = dtest)
```

Step 12. Write function to loop through all 25 hyperparameter combinations and run the model

```
random_grid_results <- list_of_param_sets %>%
  mutate(results = map(hyperparamters, function(x){

    message(paste0("model #",       x$model,
                   " eta = ",              x$eta,
                   " max.depth = ",        x$max_depth,
                   " min_child_weigth = ", x$min_child_weight,
                   " subsample = ",        x$subsample,
                   " colsample_bytree = ", x$colsample_bytree,
                   " gamma = ",            x$gamma,
                   " nrounds = ",          x$nround))

    set.seed(12345)
    singleModel <- xgb.train(params = list(
      booster          = "gbtree",
      scale_pos_weight = sqrt(12),
      eta              = x$eta,
      max_depth        = x$max_depth,
      min_child_weight = x$min_child_weight,
      gamma            = x$gamma,
      lambda           = x$lambda,
      alpha            = x$alpha,
      subsample        = x$subsample,
      colsample_bytree = x$colsample_bytree,
      max_bin          = x$max_bin,
      objective        = 'binary:logistic',
      eval_metric     = "auc"),
      data=dtrain,
      nrounds = x$nround,
      prediction = FALSE,
      watchlist = watchlist,
      showsd = TRUE,
      early_stopping_rounds = 15,
```

```
      verbose = 2)

    output <- list(score = predict(singleModel, dtest),
                   id = test_onc$usrds_id
    )
    return(output)
    }))
```

Step 12: Add all results to a list `all[[i]] <- random_grid_results` This is the end of the loop.

Step 13: Loop through each set of the 25 hyperparameters to pool the 5 prediction scores from each imputation together and calculate AUC

```
for(i in 1:how_many_models){

  one <- as.data.frame(data.table::transpose(all[[1]]$results[[i]]))[1,]
%>%
    tidyr::gather(key = "usrds_id", value = "score") %>%
    mutate(usrds_id = all[[1]]$results[[i]]$id)

  two <- as.data.frame(data.table::transpose(all[[2]]$results[[i]]))[1,]
%>%
    tidyr::gather(key = "usrds_id", value = "score") %>%
    mutate(usrds_id = all[[2]]$results[[i]]$id)

  third <- as.data.frame(data.table::transpose(all[[3]]$results[[i]]))[1,]
%>%
    tidyr::gather(key = "usrds_id", value = "score") %>%
    mutate(usrds_id = all[[3]]$results[[i]]$id)

  fourth <- as.data.frame(data.table::transpose(all[[4]]$results[[i]]))
[1,] %>%
    tidyr::gather(key = "usrds_id", value = "score") %>%
    mutate(usrds_id = all[[4]]$results[[i]]$id)

  fifth <- as.data.frame(data.table::transpose(all[[5]]$results[[i]]))[1,]
%>%
    tidyr::gather(key = "usrds_id", value = "score") %>%
    mutate(usrds_id = all[[5]]$results[[i]]$id)

  pooling  = one %>%
    inner_join(two, by = "usrds_id") %>%
    inner_join(third, by = "usrds_id") %>%
    inner_join(fourth, by = "usrds_id") %>%
    inner_join(fifth, by = "usrds_id")

  pooling$averaged  <- apply(pooling[2:ncol(pooling)], 1, mean) #averaging
scores

  pooling <- left_join(pooling, test_onc %>%
select("usrds_id","died_in_90"), by = "usrds_id")
```

```
  #calculate AUC
  auc <- pROC::auc(pooling$died_in_90, pooling$averaged) #compute AUC
  }
  # Add the results to a dataframe
  toAdd <- data.frame(hyper = all[[1]]$hyperparamters[[i]],
                      auc = auc)

  final_hp_results <- rbind(final_hp_results,toAdd)
```

Step 14. Save the output file

```
save(final_hp_results, file =
"2021_final_hp_results_random_grid_imputed.RData")

openxlsx::write.xlsx(as.data.frame(final_hp_results), file =
"2021_final_hp_results_random_grid_imputed.xlsx",
                     sheetName='Sheet1', row.names=FALSE,showNA = F)
```

Points to consider

Using a random grid search on the narrowed hyperparameter ranges allowed for the pooling of the model prediction scores from each imputed dataset on each hyperparameter combination to produce one AUC, which resulted in a much shorter compute time to identify the optimal hyperparameters.

6.3.2.4 Final Imputed XGBoost Model

Steps for running the 3_xgb_imputed_final_hyperparams.R script

Run final model using the best combination of hyperparameters from the previous step on each of the 5 imputed datasets. Pool results by averaging across samples to get the the ROC AUC, confusion matrix and feature importance

- Input:

```
universe.RData
```

- Output:

```
2021_final_hp_results_single_imputed_xgb.xlsx
2021_final_hp_results_single_imputed_xgb.RData
2021_conf_matrix.RData
2021_myplot_xgb.RData
2021_Rplots.pdf
2021_all_features.RData
2021_averaged_feature_importance_xgb.RData
2021_xgb_pooling_results_final_roc.csv
```

Step 1. Load the libraries

```
library(pROC)
library(rsample)
library(RPostgres)
library(DBI)
library(xgboost)
library(sqldf)
library(dplyr)
library(tidyr)
library(magrittr)
library(smoof)
library(mlrMBO)  # for bayesian optimisation
library(skimr) # for summarising databases
library(purrr) # to evaluate the loglikelihood of each parameter set in
the random grid search
library(DiceKriging)
library(rgenoud)
library(data.table)
library(mltools) #data.table and mltools are needed for "one_hot" function
library(readr)  #read rds
library(rBayesianOptimization)
library(openxlsx)
library(Matrix)
library(stringr)
```

Step 2. Load the data and set the seed

```
load('~/universe.RData')
depvar = "died_in_90"

rhscols = setdiff(names(universe), c("usrds_id", "subset", "died_in_90"))

trainsubsets = c(0,1,2,3,4,5,6)
testsubsets = c(7,8,9)

set.seed(123)
```

Step 3. Set the number of models (1) and set the optimal hyperparameters from the previous section

```
how_many_models <- 1
eta <-              data.frame(eta = 0.0501135)
gamma <-            data.frame(gamma = 2.937342)
lambda <-           data.frame(lambda = 8.20660)
alpha <-            data.frame(alpha = 7.27306)
max_depth <-        data.frame(max_depth = 7)
min_child_weight <- data.frame(min_child_weight = 2)
```

```
nround <-          data.frame(nround = 493)
subsample <-        data.frame(subsample = 0.7513711)
colsample_bytree <- data.frame(colsample_bytree = 0.6611578)
max_bin <-          data.frame(max_bin = 935)
```

Step 4. Initiate the hyperparameter grid

```
random_grid <-eta %>%
  bind_cols(gamma) %>%
  bind_cols(lambda) %>%
  bind_cols(alpha) %>%
  bind_cols(max_depth) %>%
  bind_cols(min_child_weight) %>%
  bind_cols(nround)  %>%
  bind_cols(subsample) %>%
  bind_cols(colsample_bytree) %>%
  bind_cols(max_bin) %>%as_tibble()


df.params <- bind_rows(random_grid) %>%
  mutate(rownum = row_number(),
         model = row_number())
list_of_param_sets <- df.params %>% nest(-rownum)

colnames(list_of_param_sets) <- c("model","hyperparamters")
```

Step 5. Set the training and test datasets

```
train_full = universe %>%
  filter(subset <=6 ) %>% as.data.frame()

test_full = universe %>%
  filter(subset > 6 ) %>% as.data.frame()
```

Step 6. Loop through each imputation

```
for(i in 1:5){
----
}
```

For each imputation perform the following steps:

Step 7. Prepare the training and the test data for modeling

```
train_onc=train_full %>%
  filter(impnum == i) %>% as.data.frame()

train_onc = train_onc[order(train_onc$usrds_id),]

rownames(train_onc) <- train_onc$usrds_id #preserving usrds_id as rownames
because usrds_id will be removed in next line

train_onc <- train_onc[, c(rhscols,"died_in_90")] #selecting variables

all_na <- function(x) any(!is.na(x)) #creating function that removes
columns containing all NAs
train_onc <- train_onc %>% select_if(all_na) #removing the columns
containing all NAs

train_onc[] <- lapply(train_onc, as.numeric) #force to numeric columns

print(paste("dimensions for train_onc:",dim(train_onc)))

#per https://stackoverflow.com/questions/48805977/r-missing-data-causes-
error-with-xgboost-sparse-model-matrix
options(na.action='na.pass')
trainm <- sparse.model.matrix(died_in_90 ~ ., data = train_onc)
dtrain <- xgb.DMatrix(data = trainm, label=train_onc[, depvar])
rm(trainm)
rm(train_onc)
gc()

#### test pre-processing

test_onc=test_full %>%
  filter(impnum == i) %>% as.data.frame()

test_onc = test_onc[order(test_onc$usrds_id),]

test_ids <- test_onc$usrds_id #preserving usrds_id
rownames(test_onc) <- test_onc$usrds_id #preserving usrds_id as rownames
because usrds_id will be removed in next line

test_onc <- test_onc[, c(rhscols,"died_in_90")] #selecting variables

test_onc <- test_onc %>% select_if(all_na) #removing the columns
containing all NAs

test_onc[] <- lapply(test_onc, as.numeric) #force to numeric columns

print(paste("dimensions for test_onc:",dim(test_onc)))

options(na.action='na.pass')
testm <- sparse.model.matrix(died_in_90 ~ ., data = test_onc)
dtest <- xgb.DMatrix(data = testm, label=test_onc[, depvar])
rm(testm)
gc()
```

Step 8. Write function to run the model

```
watchlist <- list(train = dtrain, eval = dtest)

random_grid_results <- list_of_param_sets %>%
  mutate(results = map(hyperparamters, function(x){

    message(paste0("model #",       x$model,
                   " eta = ",              x$eta,
                   " max.depth = ",        x$max_depth,
                   " min_child_weigth = ", x$min_child_weight,
                   " subsample = ",        x$subsample,
                   " colsample_bytree = ", x$colsample_bytree,
                   " gamma = ",            x$gamma,
                   " nrounds = ",          x$nround))

    set.seed(12345)
    singleModel <- xgb.train(params = list(
      booster          = "gbtree",
      scale_pos_weight = sqrt(12),
      eta              = x$eta,
      max_depth        = x$max_depth,
      min_child_weight = x$min_child_weight,
      gamma            = x$gamma,
      lambda           = x$lambda,
      alpha            = x$alpha,
      subsample        = x$subsample,
      colsample_bytree = x$colsample_bytree,
      max_bin          = x$max_bin,
      objective        = 'binary:logistic',
      eval_metric     = "auc"),
      data=dtrain,
      nrounds = x$nround,
      prediction = FALSE,
      watchlist = watchlist,
      showsd = TRUE,
      early_stopping_rounds = 15,
      verbose = 0)
```

Step 9. Obtain feature importance and save prediction scores and USRDS IDs to a list

```
      feature_imp <- xgb.importance(singleModel$feature_names,
                                    model = singleModel)
      all_features[[i]] <- feature_imp # add feature_imp to list

      output <- list(score = predict(singleModel, dtest),
                     id = test_ids
```

Step 10. Add modeling results to a list `all[[i]] <- random_grid_results` This is the end of the loop.

Step 11: Pool the 5 prediction scores from each imputation together and calculate AUC

```
final_hp_results_single <- data.frame()

for(i in 1:how_many_models){

  one <- as.data.frame(data.table::transpose(all[[1]]$results[[i]]))[1,]
%>%
    tidyr::gather(key = "usrds_id", value = "score") %>%
    mutate(usrds_id = all[[1]]$results[[i]]$id)

  two <- as.data.frame(data.table::transpose(all[[2]]$results[[i]]))[1,]
%>%
    tidyr::gather(key = "usrds_id", value = "score") %>%
    mutate(usrds_id = all[[2]]$results[[i]]$id)

  third <- as.data.frame(data.table::transpose(all[[3]]$results[[i]]))[1,]
%>%
    tidyr::gather(key = "usrds_id", value = "score") %>%
    mutate(usrds_id = all[[3]]$results[[i]]$id)

  fourth <- as.data.frame(data.table::transpose(all[[4]]$results[[i]]))
[1,] %>%
    tidyr::gather(key = "usrds_id", value = "score") %>%
    mutate(usrds_id = all[[4]]$results[[i]]$id)

  fifth <- as.data.frame(data.table::transpose(all[[5]]$results[[i]]))[1,]
%>%
    tidyr::gather(key = "usrds_id", value = "score") %>%
    mutate(usrds_id = all[[5]]$results[[i]]$id)

  pooling  = one %>%
    inner_join(two, by = "usrds_id") %>%
    inner_join(third, by = "usrds_id") %>%
    inner_join(fourth, by = "usrds_id") %>%
    inner_join(fifth, by = "usrds_id")

  pooling$averaged  <- apply(pooling[2:ncol(pooling)], 1, mean) #averaging
scores
  pooling$usrds_id <- as.character(pooling$usrds_id)

  test_onc$usrds_id <- as.character(rownames(test_onc))

  pooling <- left_join(pooling, test_onc %>%
select("usrds_id","died_in_90"), by = "usrds_id")

  pooling$predicted <- ifelse(pooling$averaged > 0.5, 1,0)

  print("pooling summary after left_join():")
  summary(pooling)
```

```
  print("conf matrix:")
  table(pooling$predicted, pooling$died_in_90)
  conf_matrix <- table(pooling$predicted, pooling$died_in_90)

  save(conf_matrix, file = "2021_conf_matrix.RData")
```

Step 12. For each imputation, calculate the confusion matrix and model evaluation metrics at a threshold of 0.5

```
  tp <- conf_matrix[2,2]
  fp <- conf_matrix[2,1]
  fn <- conf_matrix[1,2]
  tn <- conf_matrix[1,1]

  sensitivity = tp / (tp + fn)
  specificity = tn / (fp + tn)
  fpr = 1 - specificity
  tpr = sensitivity
  LR = sensitivity / (1 - specificity)
  ppv = tp / (tp + fp)
  npv = tn / (tn + fn)
  f1_score = 2 * ppv * sensitivity / (ppv + sensitivity)

  accuracy <- mean(pooling$predicted == pooling$died_in_90)
```

Step 13. Plot ROC AUC

```
  myplot <- pROC::plot.roc(pooling$died_in_90, pooling$averaged)

  save(myplot, file = "2021_myplot_xgb.RData")
```

Step 14. Save final non-imputed XGBoost modeling results

```
  write.csv(pooling, '2021_xgb_pooling_results_final_roc.csv')

  final_hp_results_single <- rbind(final_hp_results_single,toAdd)

  save(final_hp_results_single, file =
"2021_final_hp_results_single_imputed_xgb.RData")
  openxlsx::write.xlsx(as.data.frame(final_hp_results_single), file =
"2021_final_hp_results_single_imputed_xgb.xlsx",
                       sheetName='Sheet1', row.names=FALSE,showNA = F)
}

print("saving the feature importance")
save(all_features, file = "2021_all_features.RData")
```

Step 15: Average feature importance across imputations and save file

```
#averging the feature importance
averaged <- all_features %>% reduce(inner_join, by = "Feature") %>%
as.data.frame()

rownames(averaged) <- averaged$Feature
averaged = averaged %>% select(contains("Gain"))

averaged$average = as.data.frame(apply(averaged, 1, mean)) #compute
average

averaged$feature = rownames(averaged)

save(averaged, file = "2021_averaged_feature_importance_xgb.RData")
```

6.3.2.5 Calibration

The calibration curve shows the reliability of the model by each prediction score category, the number of patients that fall into each category, and the proportion of patients in each category who actually died in the first 90 days following dialysis initiation.

Steps to running the 4_xgb_imputed_calibration.ipynb script

- Input:

```
2021_xgb_pooling_results_final_roc.csv
```

- Output:

```
model_calibrated_xgb_imputed.pickle
y_calibrated_xgb_imputed.pickle
```

Step 1. Import libraries

```
import pandas as pd
import numpy as np
import pickle

import sys
#path to the functions directory
sys.path.append('/ML-AI-PCOR-2021/onc_functions/')

# import custom functions
from plot_functions import onc_plot_calibration_curve
from calibrate_onc import calibrate_onc
```

```
#connect to posgres database
import psycopg2
import sqlalchemy
from sqlalchemy import create_engine

con = create_engine('postgresql://username:password@location/dbname')
```

Step 2. Load results from the XGBoost Imputed model

```
# load results from the aucroc evaluated model)
pred_df =
pd.read_csv('./roc_results/2021_xgb_pooling_results_final_roc.csv')
pred_df = pred_df.loc[:,['averaged','died_in_90','usrds_id']]
```

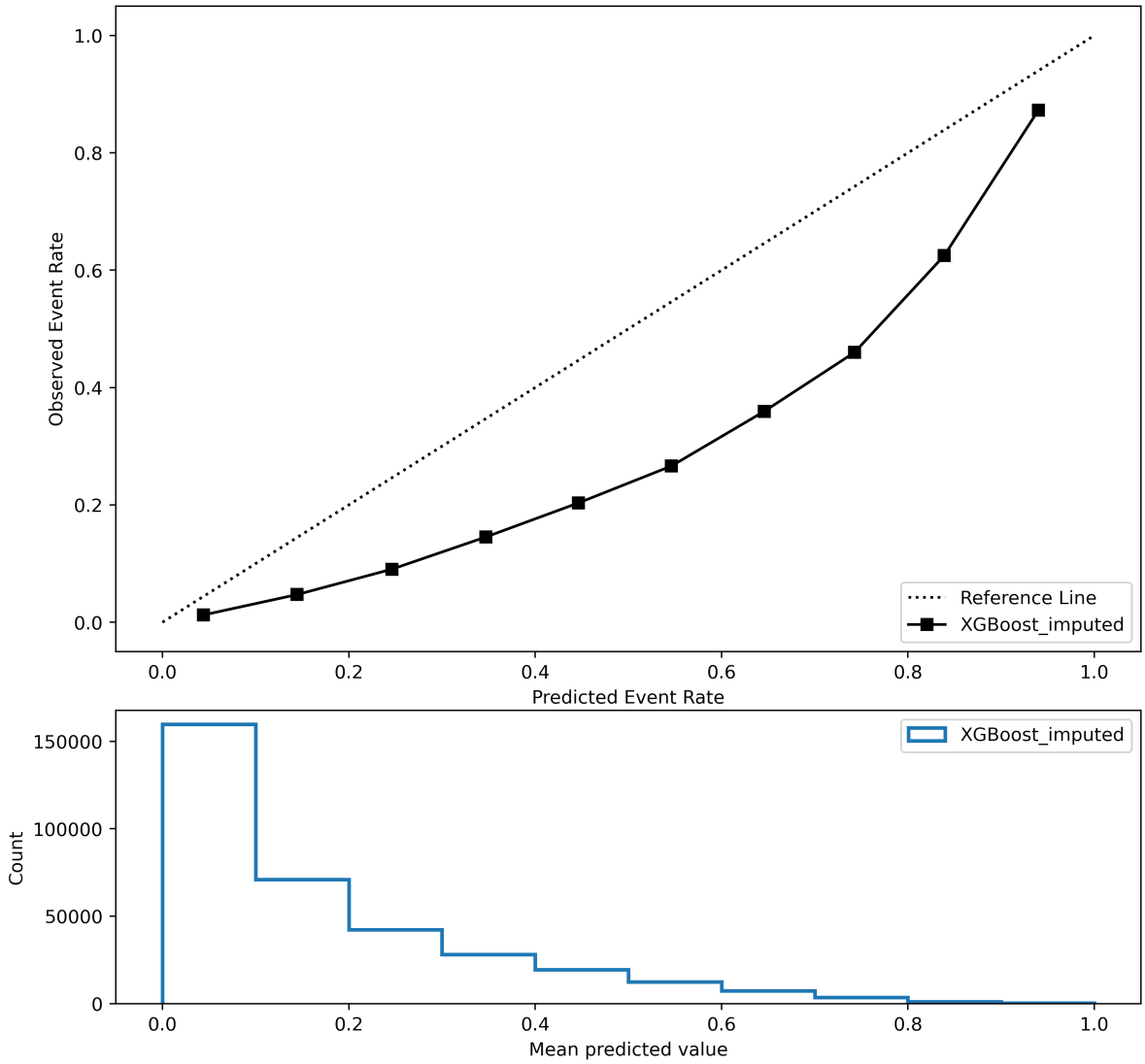
Step 3. Plot the original model's calibration curve. This function **onc_plot_calibration_curve** is located in the /onc_functions/plot_functions.py file.

```
def onc_plot_calibration_curve(y_true, y_proba, label, filename):

    #calculate numbers to plot
    clf_score = brier_score_loss(y_true, y_proba, pos_label=1)
    fraction_of_positives, mean_predicted_value = \
                calibration_curve(y_true, y_proba, n_bins=10)
    # set up plot
    fig1 = plt.figure(1, figsize=(10,10))#,dpi=400)
    ax1 = plt.subplot2grid((3, 1), (0, 0), rowspan=2)
    ax2 = plt.subplot2grid((3, 1), (2, 0))

    #plot the reference for a prefectly calibrated model
    ax1.plot([0, 1], [0, 1], "k:", label="Reference Line")

    # plot the calibration curve
    ax1.plot(mean_predicted_value, fraction_of_positives, "ks-",
                    label=label)

    # plot histogram of predicted values
    ax2.hist(y_proba, range=(0, 1), bins=10, label=label,
                 histtype="step", lw=2)

    # set axes and other figure parameters
    ax1.set_ylabel("Observed Event Rate")
    ax1.set_xlabel("Predicted Event Rate")
    ax1.set_ylim([-0.05, 1.05])
    ax1.legend(loc="lower right")

    ax2.set_xlabel("Mean predicted value")
    ax2.set_ylabel("Count")
    ax2.legend(loc="upper right", ncol=1)
```

```
    plt.rc('axes', labelsize=22)    # fontsize of the x and y labels
    plt.rc('xtick', labelsize=15)    # fontsize of the tick labels
    plt.rc('ytick', labelsize=15)    # fontsize of the tick labels
    plt.rc('legend', fontsize=20)    # legend fontsize
    #save figure resolution
    plt.savefig(filename + ".png",  dpi=400,  transparent=True)
    plt.show()
```

Run the function above.

```
onc_plot_calibration_curve(
                          y_true=pred_df.died_in_90,
                          y_proba=pred_df.averaged,
                          label='XGBoost_imputed',

filename='./roc_results/xgb_imputed_orig_calibration')
```



The XGBoost model can be calibrated by training an isotonic regression on a portion of the testing set. (Model calibration is performed as probabilities of death in the first 90 days are more informative and useful for clinicians than a simple binary prediction. In order to produce valid probability estimates, predicted events rates should track observed rates across the full range of predicted risk.)

Step 4. Load the subset for each ID:

```
df = pd.read_sql_query('''
SELECT   usrds_id, subset FROM medxpreesrd;''', con)
```

Merge the subset details with the predictions

```
data = pd.merge(pred_df, df, how="left", on="usrds_id")
```

The next steps are inside function **calibrate_onc** located in the /onc_functions/calibrate_onc.py file.

Step 5. Split the predictions from the test set (how we evaluated the model) into a test/train for the calibration (isotonic regression classifier). Split test data (subsets 7-9) into new train (7-8)/test (9) sets

```
calibration_train_set = data[((data.subset==7)|(data.subset==8))].copy()
calibration_test_set = data[data.subset==9].copy()
```

Step 6. Define the calibration model

```
ir = IsotonicRegression(out_of_bounds="clip")
```

Step 7. Fit the model to the XGBoost predictions from the (new) training set

```
ir.fit(calibration_train_set.score, calibration_train_set.y )
```

Step 8. Evaluate the model using the (new) test set

```
p_calibrated = ir.transform(calibration_test_set.score)
calibration_test_set['p_calibrated'] = p_calibrated
```

Step 9. Save

```
with open(path + 'model_calibrated_' + model_name + '.pickle', 'wb') as
picklefile:
            pickle.dump(ir,picklefile)

with open(path + 'y_calibrated_' + model_name + '.pickle', 'wb') as
picklefile:
            pickle.dump(calibration_test_set, picklefile)
```

Step 10. Print the scores from the original and calibrated model. The function **print_calibrated_results** is found in the /onc_functions/calibrate_onc.py file.

```
def print_calibrated_results(y_true, y_pred, y_calibrated):
    '''print scores for pre and post calibration'''

    acc = accuracy_score(y_true, np.round(y_pred))
    acc_calibrated = accuracy_score(y_true, np.round(y_calibrated ))
    print ("accuracy - original/calibrated:", acc, "/", acc_calibrated)
```

```
    auc = roc_auc_score(y_true, y_pred)
    auc_calibrated = roc_auc_score(y_true, y_calibrated)
    print ("ROC AUC - original/calibrated:     ", auc, "/",
auc_calibrated)

    pr = average_precision_score(y_true, y_pred)
    pr_calibrated = average_precision_score(y_true, y_calibrated )
    print ("avg precision - original/calibrated:", pr, "/", pr_calibrated)

    clf_score = brier_score_loss(y_true, y_calibrated, pos_label=1)
    print("\tBrier: %1.3f" % (clf_score))
```

Run these 2 calibration functions.

```
calibrated_results = calibrate_onc(data,
path='./roc_results/',model_name='xgb_imputed')
```

6.3.2.6 Plotting calibrated results

Steps to running the 5_xgb_imputed_calibrated_plots.ipynb script

- Input:

```
y_calibrated_xgb_imputed.pickle
```

- Output:

```
xgb_imputed_calibration.jpeg
xgb_imputed_mortality_bar.jpeg
xgb_imputed_roc_auc_bw.jpeg
2021_xgb_imputed_calibrated_confusion_matrix.csv
```

Step 1. Import libraries

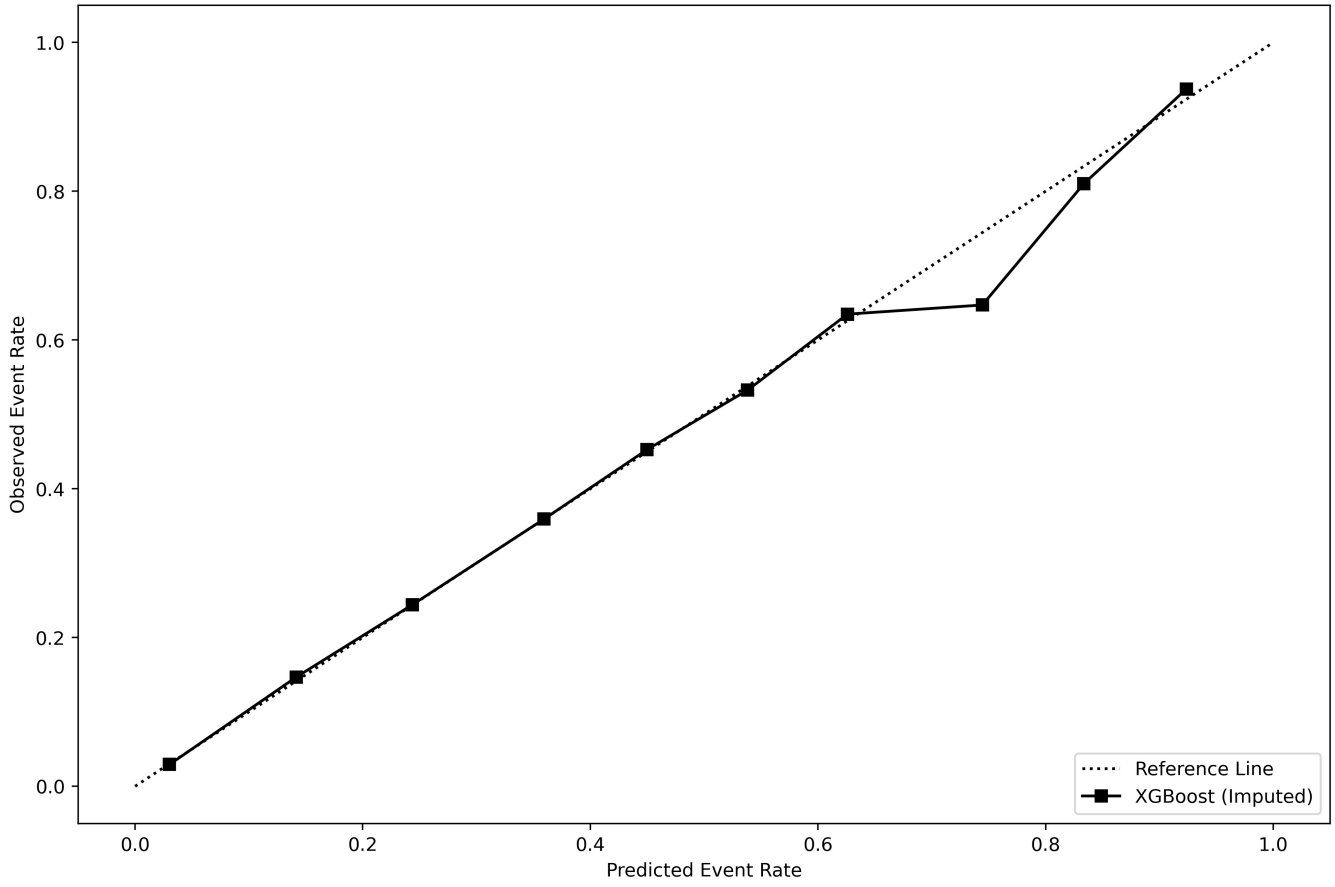
```
import pandas as pd
import numpy as np
import pickle

#import custom plotting functions
from plot_functions import onc_plot_calibration_curve, onc_calc_cm,
onc_plot_roc, onc_plot_precision_recall, onc_plot_risk
```

Step 2. Load results from the calibrated model

```
with open('./roc_results/y_calibrated_xgb_imputed.pickle', 'rb') as
picklefile:
            calibrated_results = pickle.load(picklefile)
```

Step 3. Plot the calibration curve of the calibrated model using the same **onc_plot_calibration_curve** function from /onc_functions/plot_functions.py

```
onc_plot_calibration_curve(
                y_true=calibrated_results.y,
                y_proba=calibrated_results.p_calibrated,
                label='XGBoost imputed calibrated',
                filename='./roc_results/xgb_imputed_calibrated')
```

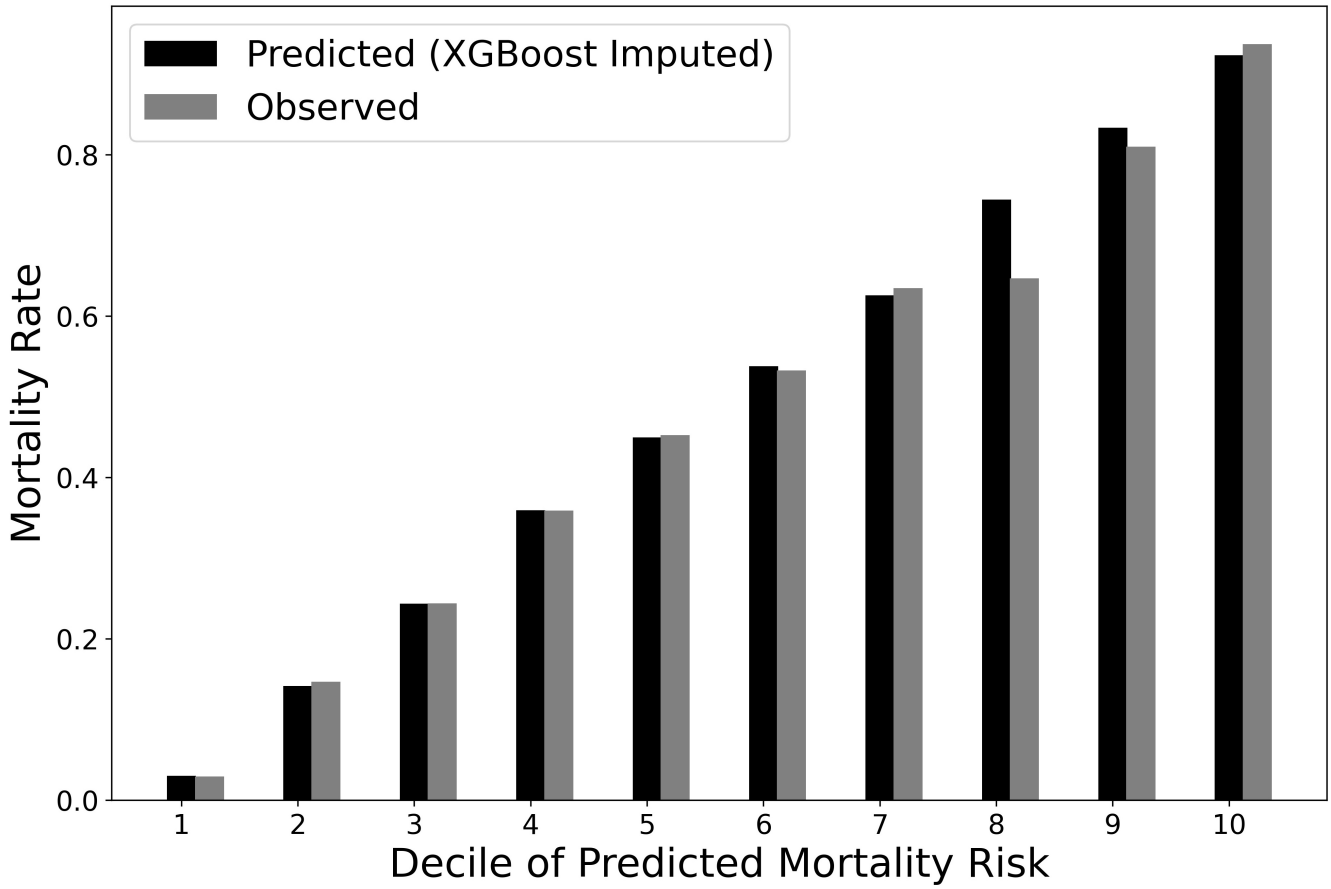


Step 4. Plot the Risk of the calibrated model. This function **onc_plot_risk** is located and imported from plot_functions.py

```
def onc_plot_risk(y_true, y_proba, label, filename):
    # calculate values for plot
    fraction_of_positives, mean_predicted_value = \
                calibration_curve(y_true, y_proba, n_bins=10)

    # set up figure params
```

```
    fig1 = plt.figure(1, figsize=(12,30),dpi=400)
    ax1 = plt.subplot2grid((7, 1), (0, 0), rowspan=2)

    # bar plot
    xs = np.arange(len(fraction_of_positives))
    ax1.bar(xs, mean_predicted_value, color='k', width = 0.25,
label=label)
    ax1.bar(xs+.25, fraction_of_positives, color='gray', width = 0.25,
label='Observed')

    #more figure settings
    plt.xticks(xs, np.arange(1, len(xs)+1, 1))
    ax1.set_ylabel("Mortality Rate")
    ax1.set_xlabel("Decile of Predicted Mortality Risk")
    ax1.legend(loc="upper left")
    plt.rc('axes', labelsize=22)    # fontsize of the x and y labels
    plt.rc('xtick', labelsize=15)    # fontsize of the tick labels
    plt.rc('ytick', labelsize=15)    # fontsize of the tick labels
    plt.rc('legend', fontsize=20)    # legend fontsize
    #save plot
    plt.savefig(filename + ".png",  dpi=400,  transparent=True)
```

```
onc_plot_risk(
    y_true=calibrated_results.y,
    y_proba=calibrated_results.p_calibrated,
    label='Predicted (XGBoost Imputed)',
    filename='xgb_imputed_mortality_bar')
```

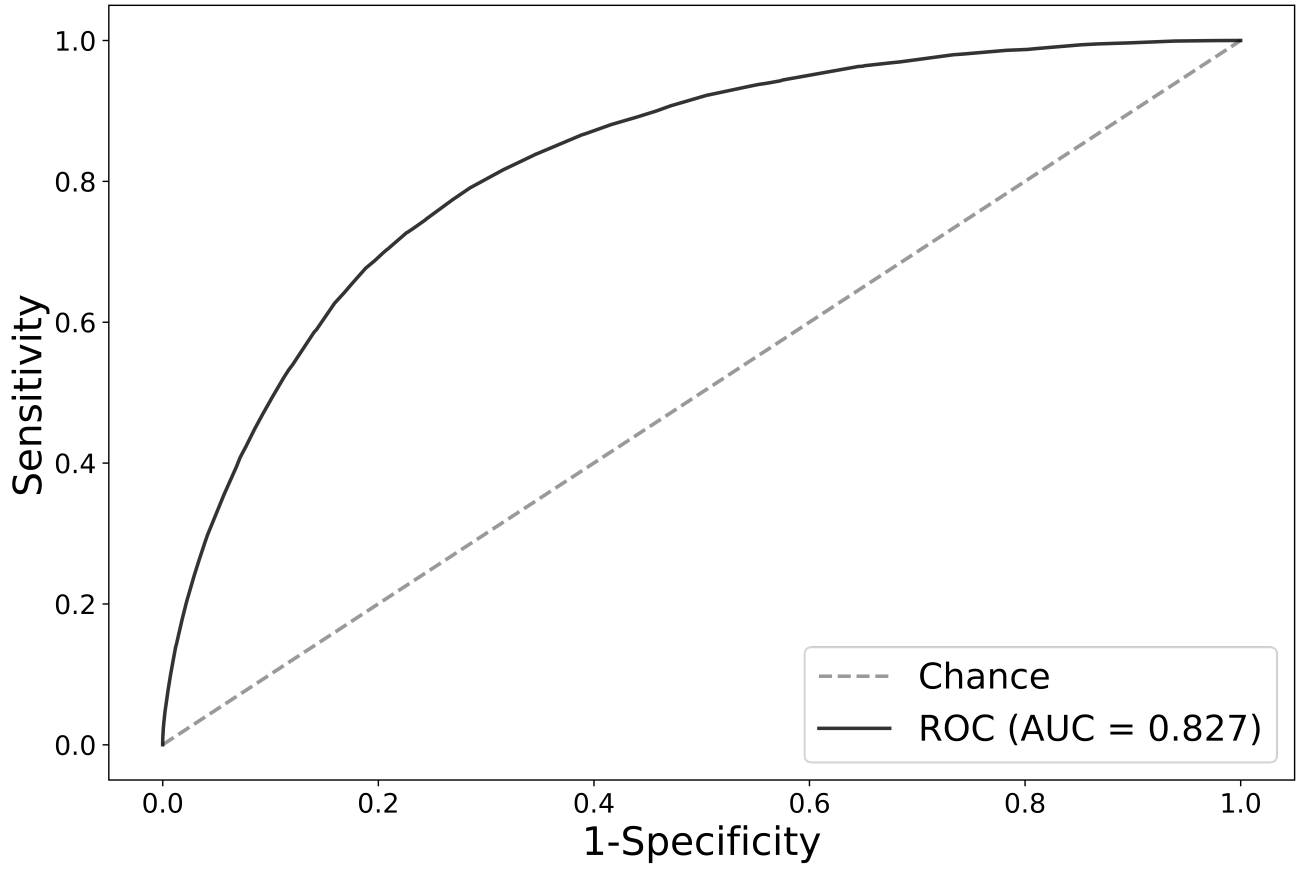


Step 5. Plot the ROC AUC of the calibrated model. This function **onc_plot_roc** is located and imported from plot_functions.py

```
def onc_plot_roc(y_true, y_pred, model_name, **kwargs):
    '''
    Plot the ROC AUC and return the test ROC AUC results.
    INPUT: y_true, y_pred, model_name, **kwargs
    '''

    #calc values for plot
    false_positives, true_positives, threshold = roc_curve(y_true, y_pred)
    c_roc_auc_score = auc(false_positives, true_positives)

    #set figure params
    fig1 = plt.figure(1, figsize=(12,30),dpi=400)
    ax1 = plt.subplot2grid((7, 1), (0, 0), rowspan=2)

    #plot reference line for chance
    ax1.plot([0, 1], [0, 1], linestyle='--', lw=2, color='gray',
        label='Chance', alpha=.8)

    # plot AUC ROC
    ax1.plot(false_positives, true_positives,
        label=r'ROC (AUC = %0.3f)' % (c_roc_auc_score),
        lw=2, alpha=.8, color = 'k')

    # additional figure params
```

```
    ax1.set(xlim=[-0.05, 1.05], ylim=[-0.05, 1.05],)
    ax1.legend(loc="lower right")
    plt.xlabel('1-Specificity')
    plt.ylabel('Sensitivity')
    plt.rc('axes', labelsize=22)    # fontsize of the x and y labels
    plt.rc('xtick', labelsize=15)    # fontsize of the tick labels
    plt.rc('ytick', labelsize=15)    # fontsize of the tick labels
    plt.rc('legend', fontsize=20)    # legend fontsize
    # save plot
    plt.savefig(model_name + "_calibrated_roc_auc_bw.png",  dpi=400,
transparent=True)
    plt.show()
```

```
onc_plot_roc(
            y_true=calibrated_results.y,
            y_pred=calibrated_results.p_calibrated,
            model_name='xgb_imputed');
```



Step 6. Save the performance metrics at multiple thresholds. The following function is imported from /onc_functions/plot_functions.py

```
def onc_calc_cm(y_true, y_predictions, range_probas=[0.1,0.5]):
    '''
    Plot the confusion matrix and scores for multiple thresholds
```

```
    '''
    df = pd.DataFrame(index = range_probas,
                      columns=['threshold','sensitivity','specificity',

'likelihood_ratio_neg','likelihood_ratio_pos',

'tp','fp','tn','fn','total_survived','total_deceased',])
    for proba_threshold in range_probas:

        cm = confusion_matrix(y_true, y_predictions > proba_threshold)
        tn = cm[0][0]
        fp = cm[0][1]

        sensitivity = recall_score(y_true, y_predictions >
proba_threshold)
        specificity = tn / (tn + fp)

        df.loc[proba_threshold, "threshold"] = proba_threshold
        df.loc[proba_threshold,"sensitivity"] = sensitivity
        df.loc[proba_threshold, "specificity"] = specificity
        df.loc[proba_threshold, "likelihood_ratio_neg"] = (1-
sensitivity)/specificity
        df.loc[proba_threshold, "likelihood_ratio_pos"] = sensitivity/(1-
specificity)
        df.loc[proba_threshold, "tp"] = cm[1][1]
        df.loc[proba_threshold, "fp"] = fp
        df.loc[proba_threshold, "tn"] = tn
        df.loc[proba_threshold, "fn"] = cm[1][0]
        df.loc[proba_threshold, "total_survived"] = np.sum(cm[0])
        df.loc[proba_threshold, "total_deceased"] = np.sum(cm[1])
    return df
```

```
cm = onc_calc_cm(
              calibrated_results.y,
              calibrated_results.p_calibrated,
              range_probas=[.10,.20, .30, .40, .50])
cm.to_csv('./roc_results/2021_xgb_imputed_calibrated_confusion_matrix.csv'
)
cm
```

threshold	sensitivity	specificity	likelihood_ratio_neg	likelihood_ratio_pos	tp	fp	tn	fn
0.1	0.703327	0.791696	0.37473	3.37644	6024	22134	84124	2541
0.2	0.423234	0.922829	0.624997	5.48439	3625	8200	98058	4940
0.3	0.202919	0.977818	0.815163	9.14796	1738	2357	103901	6827
0.4	0.100409	0.992556	0.906338	13.4883	860	791	105467	7705
0.5	0.0451839	0.997939	0.956788	21.9231	387	219	106039	8178

6.3.2.7 Saving data for the fairness assessment

Steps to running the 6_xgb_fairness_assess_get_data.ipynb script

Get the columns of data required to compute fairness assessment and save

```
inc_age = age
sex
dialtyp=type of dialysis
race
```

- Input:`medexpressesrd` table from Postgres
- Output:

```
complete_fairness_data.pickle
```

Step 1. Import the libraries

```
import psycopg2
import sqlalchemy
from sqlalchemy import create_engine

import numpy as np
import pandas as pd
import sys
import pickle
```

Step 2. Connect to the Postgres database

The credentials required to connect to the database should be inserted in the following snippet of code below:

```
con = create_engine('postgresql://username:password@location/dbname')
```

Step 3. Import the columns required for the fairness assessment from the database

```
df = pd.read_sql_query('''SELECT   usrds_id, died_in_90, inc_age, sex,
dialtyp, race, hispanic, subset FROM medxpreesrd;''', con)
```

Step 4. Save the files

```
with open('complete_fairness_data.pickle', 'wb') as picklefile:
    pickle.dump(df, picklefile)
```

6.3.2.8 Fairness assessment

ML models can perform differently for different categories of patients, so the imputed XGBoost model was assessed for fairness, or how well the model performs for each category of interest (demographics—sex, race, and age—as well as initial dialysis modality). Age is binned into the following categories based on UCSF clinician input and an example in literature: 18-25, 26-35, 36-45, 46-55, 56-65, 66-75, 76-85, 86+. The USRDS predefined categories for race, sex, and dialysis modality were used for the fairness assessment.

Steps for running the 7_xgb_imputed_fairness.ipynb script

Calculations for specific groups of patients to assess the fairness of the final model for all patients in the test subsets. For the fairness assessment for the imputed XGBoost model, all results are for imputation #5 for the non-calibrated model.

- Input:

```
2021_xgb_pooling_results_final_roc.csv
complete_fairness_data.pickle
```

- Output:

```
2021_xgb_imputed_fairness.csv
```

Step 1. Import libraries

```
import numpy as np
import pandas as pd

import pickle
import datetime
import sys
#path to the functions directory
sys.path.append('/ML-AI-PCOR-2021/onc_functions/')

# import custom function
from fairness import get_fairness_assessment
```

Step 2. Write the function that calculates AUC and the confusion matrix from the model prediction scores. This function is located and imported from the /onc_functions/fairness.py file.

```
def get_fairness_assessment(df, y_proba_col_name, y_true_col_name):

    #turn the continuous age variable into age categories
```

```
    df['agegroup'] = pd.cut(df.inc_age,
                            bins=[17, 25, 35, 45, 55, 65, 75, 85, 90],
                            labels=[1, 2, 3, 4, 5, 6, 7, 8])

    df = df.drop(columns=['inc_age'])

    #replace NaNs with a large number that does not appear in the data,
effectively creating another category for missing values
    df.loc[:,['race','dialtyp','hispanic']] = df.loc[:,
['race','dialtyp','hispanic']].fillna(100.0, axis=1).copy()

    #Identify the cols for the fairness assessment
    fairness_cols = ['agegroup', 'sex','dialtyp', 'race','hispanic']

    #loop through all categories and values to get counts, auc, and
confusion matrix
    rows_list = []
    for col in fairness_cols:
        for name, c in df.groupby(col):
            fairness_dict = {}
            fairness_dict['Feature'] = col
            fairness_dict['Value'] = name
            fairness_dict['Count'] = c.shape[0]

            fairness_dict['AUC'] = roc_auc_score(c[y_true_col_name],
c[y_proba_col_name])
            tn, fp, fn, tp = confusion_matrix(y_true = c[y_true_col_name],
                                              y_pred =
np.where(c[y_proba_col_name] >= 0.5, 1, 0)).ravel()
            fairness_dict['TN'] = tn
            fairness_dict['FP'] = fp
            fairness_dict['FN'] = fn
            fairness_dict['TP'] = tp
            rows_list.append(fairness_dict)

    #convert results from a list to a dataframe
    df_fairness = pd.DataFrame(rows_list)
    return df_fairness
```

Step 3. Load results from the model and fairness details

```
pred_df =
pd.read_csv('./roc_results/2021_xgb_pooling_results_final_roc.csv')

with open('../complete_fairness_data.pickle', 'rb') as f:
        dataset = pickle.load(f)

# merge model results with fairness details
data = pred_df.merge(dataset, how='left', on=['usrds_id','died_in_90'])
```

Step 4. Calculate fairness assessment

```
fairness = get_fairness_assessment(data,
                                   y_proba_col_name='averaged',
                                   y_true_col_name='died_in_90')
```

Step 5. Save results

```
fairness.to_csv('./roc_results/' + str(dte) + '_xgb_imputed_fairness.csv')
```

	Feature	Value	Count	AUC	TN	FP	FN	TP
0	agegroup	1.0	4340	0.868198	4288	6	42	4
1	agegroup	2.0	12774	0.845767	12527	35	186	26
2	agegroup	3.0	26120	0.847524	25362	177	483	98
3	agegroup	4.0	53564	0.818923	51122	627	1555	260
4	agegroup	5.0	85076	0.800299	79018	1734	3540	784
5	agegroup	6.0	86140	0.785300	74566	4050	5413	2111
6	agegroup	7.0	62193	0.765894	47085	6840	4697	3571
7	agegroup	8.0	15098	0.750794	9313	2817	1268	1700
8	sex	1.0	198347	0.830897	174200	9500	9565	5082
9	sex	2.0	146957	0.819263	129080	6786	7619	3472
10	dialtyp	1.0	310415	0.817315	271340	15004	16289	7782
11	dialtyp	2.0	15082	0.851288	14766	36	249	31
12	dialtyp	3.0	13295	0.859224	12995	29	241	30
13	dialtyp	4.0	77	0.969178	70	3	1	3
14	dialtyp	100.0	6436	0.781018	4110	1214	404	708
15	race	1.0	230577	0.818587	197403	13397	12658	7119
16	race	2.0	93560	0.826826	86110	2440	3801	1209
17	race	3.0	3225	0.827648	3051	46	96	32

18	race	4.0	12965	0.846709	12082	306	431	146
19	race	5.0	3776	0.830553	3563	45	137	31
20	race	6.0	881	0.811355	776	44	46	15
21	race	9.0	321	0.770124	296	8	15	2
22	hispanic	1.0	51021	0.843355	47378	1144	1876	623
23	hispanic	2.0	292532	0.820910	254706	14866	15187	7773
24	hispanic	9.0	1752	0.790733	1197	276	121	158

Points to consider

Performing the fairness assessment on the categories of interest gives additional insight into how the model performs by different patient categories of interest (by demographics, etc.). Future researchers should perform fairness assessments to better evaluate model performance, especially for models that may be deployed in a clinical setting. Other methods of assessing fairness include evaluating true positives, sensitivity, positive predictive value, etc. at various threshold across the different groups of interest, which would allow selection of a threshold that balances model performance across the groups of interest.

6.3.2.9 Risk assessment

Steps for running the 8_xgb_imputed_risk_categories.ipynb script

Note: Risk categorization is run on the non-calibrated model results.

- Input:

```
complete_fairness_data.pickle
2021_xgb_pooling_results_final_roc.csv
```

- Output:

```
2021_xgb_imputed_risk_cat.csv
```

Step 1. Import libraries

```
import numpy as np
import pandas as pd
import pickle

import sys
```

```
#path to the functions directory
sys.path.append('/ML-AI-PCOR-2021/onc_functions/')

from risk import get_risk_categories

print('python-' + sys.version)
import datetime
dte = datetime.datetime.now()
dte = dte.strftime("%Y%m%d")
```

Step 2. Import the details from the fairness assessment

```
with open('../complete_fairness_data.pickle', 'rb') as f:
        dataset = pickle.load(f)
```

Step 3. Import the pooled results from the model

```
pred_df =
pd.read_csv('./roc_results/2021_xgb_pooling_results_final_roc.csv')
```

Step 4. Merge the details with the results

```
data = pred_df.merge(dataset, on=['usrds_id','died_in_90'])
```

Step 5. Calculate risk. The function **get_risk_categories** is imported from the /onc_functions/risk.py file.

```
def get_risk_categories(dataset, y_proba_col_name, y_true_col_name):

    test_x_pd = dataset[dataset.subset > 6].copy().sort_values(by =
'usrds_id')
    del dataset

    df = test_x_pd.loc[:,[y_true_col_name,y_proba_col_name]]

    #construct the risk categories from the predicted score
    df['risk_categories'] = pd.cut(df[y_proba_col_name],
                                   bins=[-0.1, 0.09, 0.19, 0.29, 0.39,
0.49, 0.59, 0.69, 0.79, 0.89, 0.99],
                                   labels=['0-0.09', '0.1-0.19', '0.2-
0.29', '0.3-0.39', '0.4-0.49',
                                           '0.5-0.59','0.6-0.69','0.7-
0.79','0.8-0.89','0.9-0.99'])

    #loop through all the categories to get the predicted score
    risk_list = []
```

```
    for name, c in df.groupby('risk_categories'):
        risk_dict = {}
        risk_dict['Risk Category'] = name
        risk_dict['Count'] = c[y_true_col_name].shape[0]
        risk_dict['Count Died in 90'] = c[y_true_col_name].sum()
        risk_dict['Count Survived'] = c[y_true_col_name].shape[0]-
c[y_true_col_name].sum()
        risk_dict['Percent Died in 90'] =
c[y_true_col_name].sum()/c[y_true_col_name].shape[0]

        risk_list.append(risk_dict)

    df_risk = pd.DataFrame(risk_list)
    return df_risk
```

Run the function above

```
risk_cat = get_risk_categories(data,
                               y_proba_col_name='score',
                               y_true_col_name='died_in_90')
```

Step 6. Save

```
risk_cat.to_csv('./results/' + str(dte) + '_xgb_imputed_risk_cat.csv')
```

	Risk Category	Count	Count Died in 90	Count Survived	Percent Died in 90
0	0-0.09	149202	1681	147521	0.011267
1	0.1-0.19	76283	3315	72968	0.043457
2	0.2-0.29	44122	3784	40338	0.085762
3	0.3-0.39	29109	4082	25027	0.140232
4	0.4-0.49	20156	3946	16210	0.195773
5	0.5-0.59	13145	3424	9721	0.260479
6	0.6-0.69	7775	2699	5076	0.347138
7	0.7-0.79	3919	1753	2166	0.447308
8	0.8-0.89	1246	755	491	0.605939
9	0.9-0.99	343	294	49	0.857143

6.3.3 Logistic Regression (LR) Model

LR is a classic categorization model that can be used to examine the association of (categorical or continuous) independent variable(s) with one binary dependent variable.

Environment

The environment used for the LR model was purchased on Amazon Web Services (AWS):

```
Name: m5.4xlarge
vCPU: 16
GPU: 0
Cores: 8
Threads per core: 2
Architecture: x86_64
Memory: 64 GB
Operating System: Linux (Ubuntu 20.04 Focal Fossa)
Network Performance: 10 GB or less
Zone: US govcloud west
```

The LR model takes less than 1 day to run each section of code if using the above environment.

The LR model and cross validation methods from the Python (version 3.6.9) library scikit learn (version 0.24.1) were utilized along with the following libraries:

Python Library	Version
scikit-learn	0.24.1
numpy	1.19.5
pandas	1.1.5
matplotlib	3.3.3
seaborn	0.11.1

Points to consider

The use of parallel processing significantly decreased the amount of time it took to run the model.

6.3.3.1 Pre-processing the training dataset

The preprocessing included one hot encoding of categorical features and removal of features with missing values.

Steps for running the 1_lr_preprocessing.ipynb script

- Input:

```
medexpressesrd
micecomplete_pmm
numeric_columns.pickle
```

- Output:

```
complete1.pickle
complete2.pickle
complete3.pickle
complete4.pickle
complete5.pickle
```

Step 1. Install/import libraries

```
import psycopg2
import sqlalchemy
from sqlalchemy import create_engine

# other libraries
import numpy as np
import pandas as pd
import sys
import pickle
import seaborn as sns

#plotting
import matplotlib.pyplot as plt
%matplotlib inline
```

Step 2. Connect to the Postgres database.

The credentials for the Postgres database will be inserted here.

```
con = create_engine('postgresql://username:password@address/dbname')
```

Step 3. Get data

Load the full non-imputed data found in the `medexpressesrd` table from Postgres database.

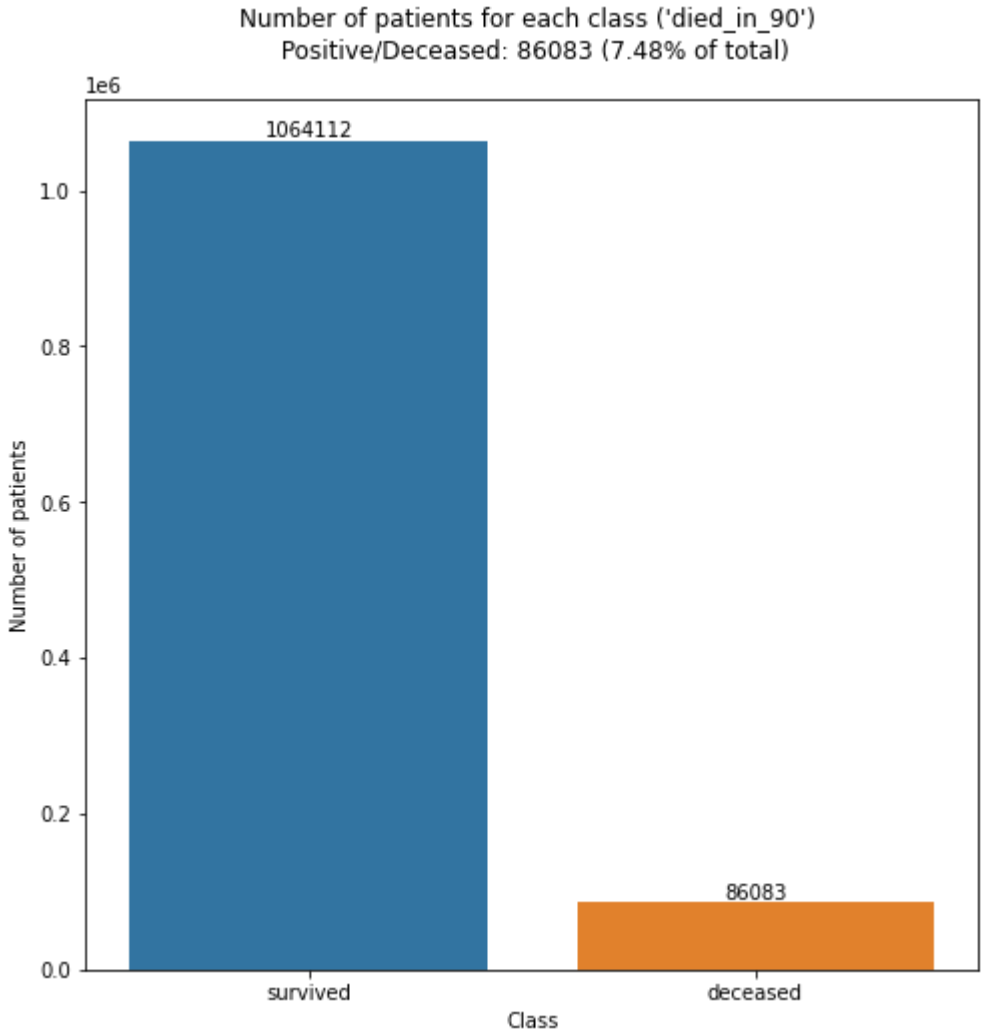
```
df = pd.read_sql_query('''SELECT * FROM medxpreesrd;''', con)
```

Get counts for each class. (This gets used later when we train the model.)

```
neg_class_count, positive_class_count = np.bincount(df['died_in_90'])
```

The labels are 2 integers, 0 (survived) or 1 (deceased). These correspond to the class. Note that we have a class imbalance with deceased being the minority class.

```
Label   Class   Count
0   survived    1064112
1   deceased    86083
```



Step 4. Remove missing data

Logistic regression models cannot account for missing values. The columns to remove are loaded at the top of the notebook in the variable **vars_to_remove** and were chosen by the clinical experts who are part of the project team.

For this dataset, the columns of pre-esrd claims data that have missing values (claim counts, etc) were removed, keeping the binary features from the Medicare pre-ESRD claims, which include indicators for claims in each care setting, indicator for pre-ESRD claims, indicators for each diagnosis group.

```
df.drop(columns=vars_to_remove,inplace=True)
```

Step 5. Encode categorical features

One variable **dial_train_time** was created from taking the difference between 2 dates in the medevid table. This feature was the only non-claims-related feature to have a large number of missing values, but instead of dropping, it was encoded as follows:

- a number greater than zero=1
- 0=0
- missing=3 Thus, the feature is turned into a categorical rather than numeric variable to retain some (though not all) information.

```
df.dial_train_time = df.dial_train_time.fillna(-1)
df.dial_train_time=df.dial_train_time.astype(int).clip(lower=-1,upper=1)
df.dial_train_time=df.dial_train_time.astype(str).replace("-1","na")
```

Use dummy variables for categorical variables (loaded at the top of the notebook) which is the method used for one-hot encoding.

Get the list of categorical variables that have more then 2 levels, then encode using pandas get_dummies function.

```
dummy_list = []
for col in categoryVars:
    u = len(df[col].unique())
    if u>2:
        dummy_list.append(col)

df = pd.concat([df,
pd.get_dummies(df.loc[:,dummy_list].astype('str'))],axis=1).drop(columns=d
ummy_list,axis=1)
```

Step 6. Load imputed data Import imputed data `micecomplete_pmm` table from Postgres.

```
imp = pd.read_sql_query('''SELECT *, row_number()
 OVER(PARTITION BY usrds_id) AS impnum
 FROM micecomplete_pmm
''', con)
```

Step 7. Remove the 5 (imputed) columns from the original data

```
df.drop(columns=["height", "weight", "bmi", "sercr", "album", "gfr_epi",
"heglb"],inplace=True)
df.shape = (1150195, 290)
```

Step 8. Separate the imputed data into 5 data frames

This makes it easier to store, load, and compute.

```
imp1 = imp[imp.impnum==1]
imp2 = imp[imp.impnum==2]
imp3 = imp[imp.impnum==3]
imp4 = imp[imp.impnum==4]
imp5 = imp[imp.impnum==5]
```

Step 9. Merge the encoded data with each of the 5 imputed datasets.

This is a left merge on the non-imputed data based on the **usrds_id** and the **subset** number.

```
complete1 = pd.merge(df, imp1, how='left', on=["usrds_id","subset"])
complete2 = pd.merge(df, imp2, how='left', on=["usrds_id","subset"])
complete3 = pd.merge(df, imp3, how='left', on=["usrds_id","subset"])
complete4 = pd.merge(df, imp4, how='left', on=["usrds_id","subset"])
complete5 = pd.merge(df, imp5, how='left', on=["usrds_id","subset"])

complete5.shape
(1150195, 298)
```

Step 10. Save

Save each set to the current directory as a pickle file (a file type that works well for pandas dataframes).

```
with open('complete1.pickle', 'wb') as picklefile:
    pickle.dump(complete1, picklefile)
with open('complete2.pickle', 'wb') as picklefile:
    pickle.dump(complete2, picklefile)
with open('complete3.pickle', 'wb') as picklefile:
    pickle.dump(complete3, picklefile)
with open('complete4.pickle', 'wb') as picklefile:
    pickle.dump(complete4, picklefile)
with open('complete5.pickle', 'wb') as picklefile:
    pickle.dump(complete5, picklefile)
```

Points to consider

The approach used to handle missing values is dependent on the dataset and the features in the dataset. Clinical expertise is crucial in understanding the impact of missing values and whether or not they should be imputed, removed, or replaced.

6.3.3.2 Hyperparameter tuning and final logistic regression model

This script computes the 5-fold cross-validation gridsearch on each set of the complete imputed data to find the best hyperparameters for the logistic regression model. LR has few parameters to set, therefore, hyperparameter tuning is simpler for this model. The data was split into test and train sets (the same

~70/30 split used in the other models). The training data was used to run the cross-validation model. The following parameters were tested:

- regularization strength
- regularization type (penalty)
- max iterations for convergence
- class weight

The cross validation was run once per each imputed dataset and results were pooled (averaged).

Steps for running the 2_logistic_regression.ipynb script

- Input:

```
complete1.pickle
complete2.pickle
complete3.pickle
complete4.pickle
complete5.pickle
```

- Output:

```
2021_LR_cv_clf_imp_1.pickle
2021_LR_cv_clf_imp_2.pickle
2021_LR_cv_clf_imp_3.pickle
2021_LR_cv_clf_imp_4.pickle
2021_LR_cv_clf_imp_5.pickle
2021_final_LR_model_test_pred_proba_imp_x.pickle
2021_final_model_LR_fpr_all.pickle
2021_final_model_LR_tpr_all.pickle
2021_final_model_LR_auc_all.pickle
2021_final_LR_model.pickle
```

Step 1. Install/import libraries

```
import pandas as pd
import numpy as np
import os
import pickle
import sklearn.metrics as metrics
from sklearn.metrics import auc, plot_confusion_matrix, roc_curve,
plot_roc_curve, accuracy_score, roc_auc_score, classification_report,
confusion_matrix, PrecisionRecallDisplay, precision_recall_curve,
RocCurveDisplay

from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.model_selection import KFold, StratifiedKFold,
train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegressionCV, LogisticRegression
from sklearn.dummy import DummyRegressor, DummyClassifier

import seaborn as sns
import matplotlib.pyplot as plt
```

Step 2. Load numeric column list

```
with open('numeric_columns.pickle', 'rb') as f:
    nu_cols = pickle.load(f)
```

Step 3. Import a set of data and scale numeric columns

Each imputed set should be run separately, so only import one set at a time. Here, we import imputation 5.

```
    with open('complete5.pickle', 'rb') as f:
        dataset = pickle.load(f)
```

Keep only the training data subsets (1-6) since we are only running a cross validation to obtain the optimal parameters for the model.

```
    X_train =  dataset[dataset.subset <= 6].copy().sort_values(by =
'usrds_id')
```

Step 4. Separate the labels (typically denoted as 'y') and save as an array.

```
    y_train = np.array(X_train.pop('died_in_90'))
```

Step 6. Scale only the numeric columns with an sklearn library **StandardScaler**.

```
    scaler = StandardScaler()
    X_train[nu_cols] = scaler.fit_transform(X_train[nu_cols])
```

Step 5. Save the training data as an array (rather than a pandas dataframe) and drop the non-feature columns.

```
    X_train = np.array(X_train.drop(columns=
['subset','usrds_id','impnum']))
```

Step 6. Create the grid for the grid search

```
param_grid = [{
        'penalty':['l1','l2','elasticnet'],
        'C': np.logspace(-3, 3, 10, 20),
        'max_iter': [500, 1000, 1500],
        'class_weight' :['balanced']
        }]
```

Step 7. Instantiate the model

```
lr_model = LogisticRegression()
clf = GridSearchCV(
    lr_model,
    param_grid=param_grid,
    cv=5,
    verbose=True,
    n_jobs=-1,
    scoring='average_precision'
)
```

Step 8. Fit the grid search 5-fold cross-validated logistic regression model

```
best_clf = clf.fit(X_train, y_train)
```

Save the model.

```
with open('2021_LR_cv_clf_imp_'+str(imp)+'.pickle', 'wb') as picklefile:
        pickle.dump(clf,picklefile)
```

Step 9. Calculate predictions on the test set

```
pred_proba_onc_train = clf.predict_proba(X_train)[:,1]
```

Step 10. Metrics and results Calculate and plot the roc_auc (area under the receiver operating characteristic curve) for each fold.

```
train_score = roc_auc_score(y_train, pred_proba_onc_train)
```

Step 11. Import data and scale numeric cols

Train and evaluate the final model based on the best parameters from the cross-validation step. This must be done for each of the 5 imputed datasets.

Load a set of data.

```
with open('./complete' + str(imp) + '.pickle', 'rb') as f:
        dataset = pickle.load(f)
```

Separate the training set (1-6)

```
train_x =  dataset[dataset.subset <= 6].copy().sort_values(by =
'usrds_id')
```

from the test set (7-9).

```
test_x = dataset[dataset.subset > 6].copy().sort_values(by = 'usrds_id')
```

Separate the labels.

```
train_y = np.array(train_x.pop('died_in_90'))
test_y = np.array(test_x.pop('died_in_90'))
```

Scale the numeric columns by training the model on the training set and then using this to transform the test set. Also remove the non-feature columns used to identify patients, imputations or subsets.

```
scaler = StandardScaler()
train_x[nu_cols] = scaler.fit_transform(train_x[nu_cols])
train_x = np.array(train_x.drop(columns=['subset','usrds_id','impnum']))

test_x[nu_cols] = scaler.transform(test_x[nu_cols])
test_x = np.array(test_x.drop(columns=['subset','usrds_id','impnum']))
```

Step 12. Instantiate the final logistic regression model

Use the best hyperparameters from the cross-validation for the final model.

```
lr_model_final = LogisticRegression(C=0.1,
                                    penalty='l2',
                                    max_iter=1000,
```

```
                                        solver='saga',
                                        class_weight='balanced',
                                        n_jobs=-1,
                                        verbose=1,
                                        random_state=499)
```

Step 13. Train the model

Fit the model on the training data subsets.

```
logistic_model_final = lr_model_final.fit(train_x, train_y)
```

Step 14. Evaluate the model

Evaluate the model by predicting on the test set and plot the outcome for each imputation.

```
pred_proba_onc_test = logistic_model_final.predict_proba(test_x)
```

Step 15. Save the model

```
with open('2021_final_LR_model_test_pred_proba_imp_' + str(imp) +
'.pickle', 'wb') as picklefile:
        pickle.dump(pred_proba_onc_test, picklefile)
```

Points to consider

1. Standardization allows for comparison of multiple features in different units and the penalty (i.e., L1) will be applied more equally across the features. The model will learn the importance of features better and faster when it isn't overwhelmed by a feature with a much larger range than the others.
2. Logistic regression models do not perform well when the outcome variable is imbalanced (or heavily skewed towards one outcome). The outcome variables (survived, died_in_90) in the training dataset was balanced through weighting (edit the weight parameter in the model to give more weight to the minority class and less to the majority class). Balancing the data ensures that the models have sufficient data from both of the outcome classes (died vs survived) on which to train. This results in a better balance between sensitivity and specificity, which is important for this dataset where mortality is predicted.
3. Due to the small set of hyperparameters to tune, this model does not require a GPU or even multiple CPUs to run the cross-validation.
4. It is important to keep the test set separate when scaling, otherwise we are peeking at the test set which will cause an invalid evaluation of the model.

6.3.3.3 Pool Results

Steps for running the 3_pool_results_from_imputations.ipynb script

This script pools the probability predicions for each row/patient from all 5 imputations by averaging the scores.

- Input:

```
2021_final_LR_model_test_pred_proba_imp_1.pickle
2021_final_LR_model_test_pred_proba_imp_2.pickle
2021_final_LR_model_test_pred_proba_imp_3.pickle
2021_final_LR_model_test_pred_proba_imp_4.pickle
2021_final_LR_model_test_pred_proba_imp_5.pickle
```

- Output:

```
2021_final_LR_model_test_pred_proba_pooled.pickle
```

Step 1. Libraries

```
import pickle
import numpy as np
import pandas as pd

import psycopg2
import sqlalchemy
from sqlalchemy import create_engine
```

Step 2. Import results from each imputation

```
with
open('./results/2021_final_LR_model_test_pred_proba_imp_1.pickle','rb') as
f:
    imp1_pred = pickle.load(f)
with
open('./results/2021_final_LR_model_test_pred_proba_imp_2.pickle','rb') as
f:
    imp2_pred = pickle.load(f)
with
open('./results/2021_final_LR_model_test_pred_proba_imp_3.pickle','rb') as
f:
    imp3_pred = pickle.load(f)
with
open('./results/2021_final_LR_model_test_pred_proba_imp_4.pickle','rb') as
f:
    imp4_pred = pickle.load(f)
with
```

```
open('./results/2021_final_LR_model_test_pred_proba_imp_5.pickle','rb') as
f:
    imp5_pred = pickle.load(f)
```

Step 3. Keep only the predictions from the positive class

```
pooled = pd.DataFrame()
pooled['imp1']=imp1_pred[:,1]
pooled['imp2']=imp2_pred[:,1]
pooled['imp3']=imp3_pred[:,1]
pooled['imp4']=imp4_pred[:,1]
pooled['imp5']=imp5_pred[:,1]
```

Step 4. Calculate the mean and standard deviation of the predicted probability for the positive class (died_in_90) for each patient/row.

```
pooled['score'] = pooled.mean(axis=1)
pooled['std_'] = pooled.std(axis=1)
```

	imp1	imp2	imp3	imp4	imp5	score	std_
0	0.925393	0.914949	0.901626	0.929962	0.921700	0.918726	0.009861
1	0.778438	0.800176	0.776335	0.791798	0.777974	0.784944	0.009424
2	0.449145	0.417679	0.474578	0.594807	0.488599	0.484961	0.059995
3	0.636240	0.627138	0.649013	0.598810	0.635832	0.629407	0.016815
4	0.168010	0.186585	0.171646	0.276738	0.167967	0.194189	0.041841

Step 5. Import details from the medexpreesrd table

```
con = create_engine('postgresql://username:password@location/dbname')

dataset = pd.read_sql_query('''SELECT  usrds_id, died_in_90, subset FROM
medxpreesrd;''', con)
```

Step 6. Sort the values so they are in the same order as when the LR was calculated and keep only the test set.

```
dataset = dataset[dataset.subset > 6].copy().sort_values(by =
'usrds_id').reset_index(drop=True)
```

Step 7. Merge the details with the pooled predictions.

```
pooled = pooled.merge(dataset, left_index=True, right_index=True)
```

Step 8. Save

```
with open('./results/2021_final_LR_model_test_pred_proba_pooled.pickle',
'wb') as picklefile:
        pickle.dump(pooled, picklefile)
```

6.3.3.4 Plot Results

Steps for running the 4_plot_lr_roc_auc.ipynb script

- Input:

```
2021_final_LR_model_test_pred_proba_pooled.pickle
```

- Output:

```
lr_roc_auc_bw.png
2021_lr_confusion_matrix.csv
```

Step 1. Import libraries

```
import pandas as pd
import numpy as np
import pickle

import sys
#path to the functions directory
sys.path.append('/ML-AI-PCOR-2021/onc_functions/')

#import custom plotting functions
from plot_functions import onc_calc_cm, onc_plot_roc
```

Step 2. Load the pooled model results

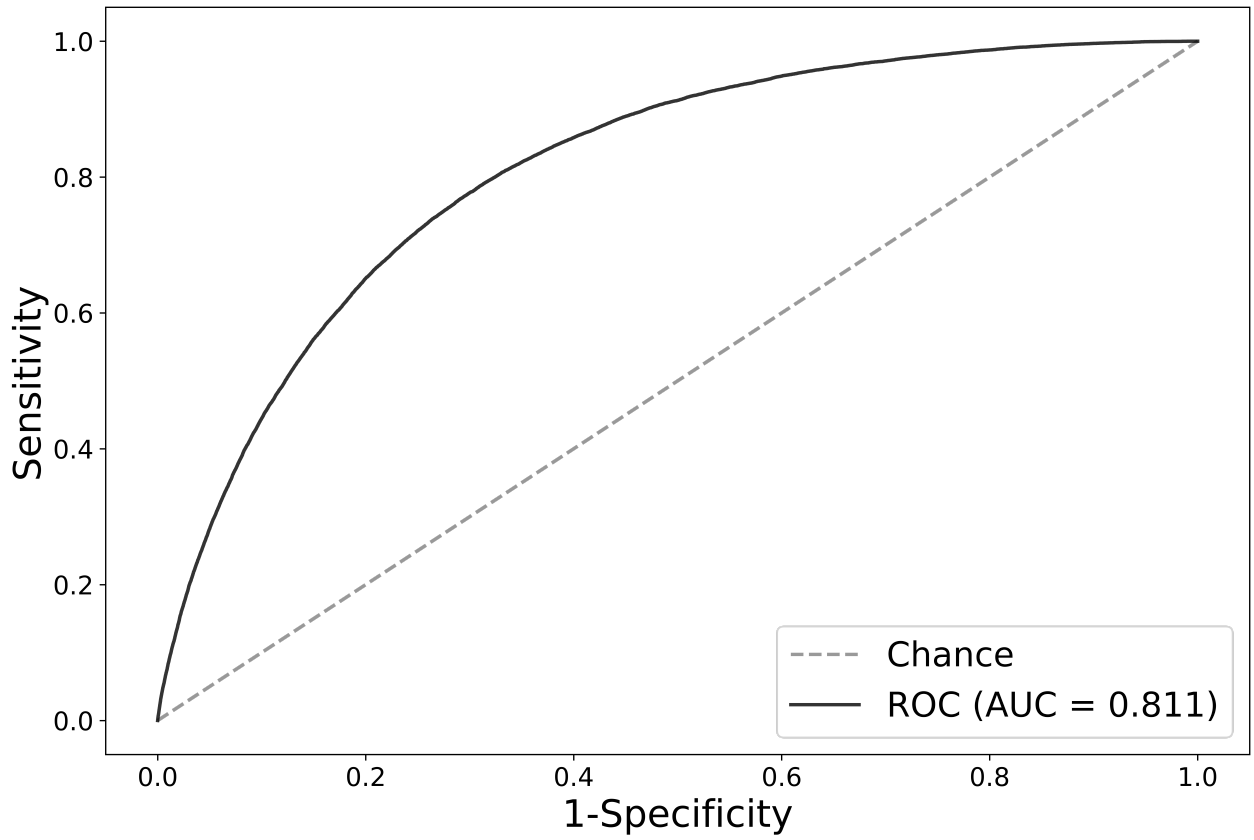
```
with
open('./results/2021_final_LR_model_test_pred_proba_pooled.pickle','rb')
as f:
    results = pickle.load(f)
```

```
results = results.loc[:,['score','died_in_90','subset','usrds_id']]
results = results.rename(columns={'died_in_90':'y'})
```

Step 3. Plot the ROC AUC. This function **onc_plot_roc** is located and imported from /onc_functions/plot_functions.py

```
def onc_plot_roc(y_true, y_pred, model_name, **kwargs):
    '''
    Plot the ROC AUC and return the test ROC AUC results.
    INPUT: y_true, y_pred, model_name, **kwargs
    '''

    #calc values for plot
    false_positives, true_positives, threshold = roc_curve(y_true, y_pred)
    c_roc_auc_score = auc(false_positives, true_positives)

    #set figure params
    fig1 = plt.figure(1, figsize=(12,30),dpi=400)
    ax1 = plt.subplot2grid((7, 1), (0, 0), rowspan=2)

    #plot reference line for chance
    ax1.plot([0, 1], [0, 1], linestyle='--', lw=2, color='gray',
        label='Chance', alpha=.8)

    # plot AUC ROC
    ax1.plot(false_positives, true_positives,
        label=r'ROC (AUC = %0.3f)' % (c_roc_auc_score),
        lw=2, alpha=.8, color = 'k')

    # additional figure params
    ax1.set(xlim=[-0.05, 1.05], ylim=[-0.05, 1.05],)
    ax1.legend(loc="lower right")
    plt.xlabel('1-Specificity')
    plt.ylabel('Sensitivity')
    plt.rc('axes', labelsize=22)    # fontsize of the x and y labels
    plt.rc('xtick', labelsize=15)    # fontsize of the tick labels
    plt.rc('ytick', labelsize=15)    # fontsize of the tick labels
    plt.rc('legend', fontsize=20)    # legend fontsize
    # save plot
    plt.savefig(model_name + "_roc_auc_bw.png",  dpi=400,
transparent=True)
    plt.show()
```

```
onc_plot_roc(
            y_true=results.y,
            y_pred=results.score,
            model_name='lr');
```



Step 4. Print and save the performance metrics at multiple thresholds

```
def onc_calc_cm(y_true, y_predictions, range_probas=[0.1,0.5]):
    '''
    Plot the confusion matrix and scores for multiple thresholds
    '''
    df = pd.DataFrame(index = range_probas,
                      columns=['threshold','sensitivity','specificity',

'likelihood_ratio_neg','likelihood_ratio_pos',

'tp','fp','tn','fn','total_survived','total_deceased',])
    for proba_threshold in range_probas:

        cm = confusion_matrix(y_true, y_predictions > proba_threshold)
        tn = cm[0][0]
        fp = cm[0][1]

        sensitivity = recall_score(y_true, y_predictions >
proba_threshold)
        specificity = tn / (tn + fp)

        df.loc[proba_threshold, "threshold"] = proba_threshold
        df.loc[proba_threshold,"sensitivity"] = sensitivity
        df.loc[proba_threshold, "specificity"] = specificity
        df.loc[proba_threshold, "likelihood_ratio_neg"] = (1-
sensitivity)/specificity
```

```
        df.loc[proba_threshold, "likelihood_ratio_pos"] = sensitivity/(1-
specificity)
        df.loc[proba_threshold, "tp"] = cm[1][1]
        df.loc[proba_threshold, "fp"] = fp
        df.loc[proba_threshold, "tn"] = tn
        df.loc[proba_threshold, "fn"] = cm[1][0]
        df.loc[proba_threshold, "total_survived"] = np.sum(cm[0])
        df.loc[proba_threshold, "total_deceased"] = np.sum(cm[1])
    return df
```

```
cm = onc_calc_cm(
    results.y,
    results.score,
    range_probas=[.10,.20, .30, .40, .50])
cm.to_csv('./results/2021_lr_confusion_matrix.csv')
cm
```

threshold	sensitivity	specificity	likelihood_ratio_neg	likelihood_ratio_pos	tp	fp	tn	fn
0.1	0.992696	0.15179	0.0481217	1.17034	25550	271060	48507	188
0.2	0.966781	0.324821	0.10227	1.43189	24883	215765	103802	855
0.3	0.921206	0.48065	0.163932	1.77377	23710	165967	153600	2028
0.4	0.850183	0.612116	0.244753	2.19185	21882	123955	195612	3856
0.5	0.752661	0.722268	0.342447	2.71003	19372	88754	230813	6366

6.3.3.5 Feature Importance

Steps for running the 5_lr_feature_importance.ipynb script

Plot the feature importance according to the final logistic regression model.

- Input:

```
complete5.pickle
2021_final_LR_model
```

- Output:

```
2021_top_bottom_plot.svg
2021_top_log_regression_coef_20.csv
```

Step 1. Load the final model

```

```

```
with open('./results/2021_final_LR_model.pickle', 'rb') as picklefile:
        logistic_model_final = pickle.load(picklefile)
```

Step 2. Import feature names Import one imputation of the data to get feature names.

```
with open('./complete5.pickle', 'rb') as f:
        dataset = pickle.load(f)
feats = dataset.iloc[0:1,:]
```

Drop the columns that were not used as features in the model.

```
ff = feats.drop(columns=
['usrds_id','subset','died_in_90','impnum']).copy()
ff = ff.columns
ff = np.array(ff)
```

Step 3. Sort features by coeffecient score

Create a function to sort the features by their highest (tops) or lowest (bottom) coeffecient to see which features were most important for the model to determine which class a patient was in. Here we get the strongest scores regardless of whether they are positive or negative because we are interested in the magnitude (difference from zero).

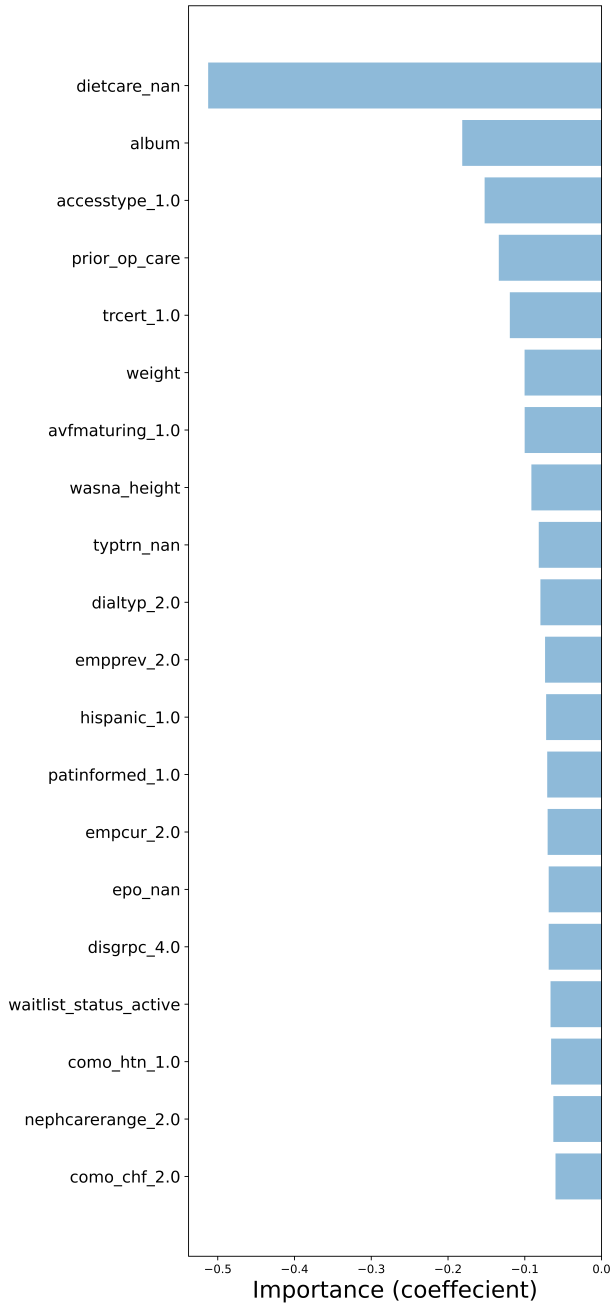
```
def get_most_important_features(r, model, n=5):

    classes ={}
    for class_index in range(model.coef_.shape[0]):
        word_importances = [(el, r[i]) for i, el in
enumerate(model.coef_[class_index])]
        sorted_coeff = sorted(word_importances, key = lambda x : x[0],
reverse=True)
        tops = sorted(sorted_coeff[:n], key = lambda x : x[0])
        bottom = sorted_coeff[-n:]
        classes[class_index] = {
            'tops':tops,
            'bottom':bottom
        }
    return classes
```

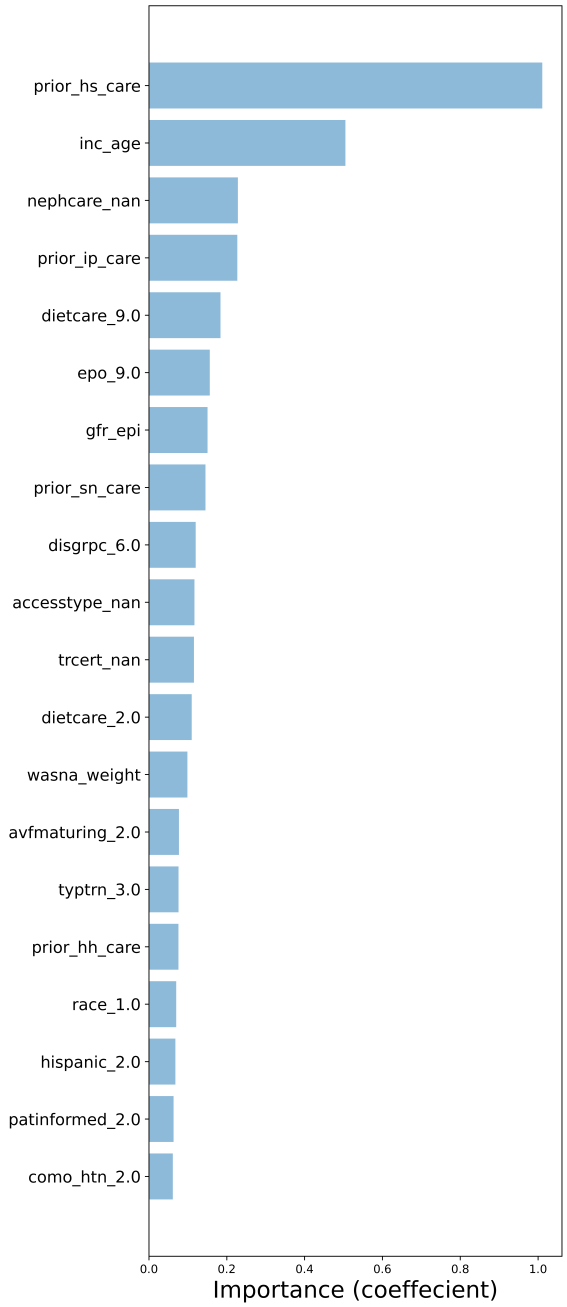
Call function for the top and bottom 20 features.

```
importance = get_most_important_features(ff, logistic_model_final, 20)
```

Save the strongest scores (most negative and most positive).

```
top_scores = [a[0] for a in importance[0]['tops']]
top_words = [a[1] for a in importance[0]['tops']]
bottom_scores = [a[0] for a in importance[0]['bottom']]
bottom_words = [a[1] for a in importance[0]['bottom']]
top_coef = pd.DataFrame(columns=['vocab','coef'])
top_coef['vocab'] = top_words + bottom_words
top_coef['coef'] = top_scores + bottom_scores
top_coef = top_coef.sort_values(by='coef',axis=0,ascending=False)
top_coef.to_csv('./results/2021_top_log_regression_coef_20.csv')
```





6.3.3.6 Fairness assessment

ML models can perform differently for different categories of patients, so the LR model was assessed for fairness, or how well the model performs for each category of interest (demographics—sex, race, and age—as well as initial dialysis modality). Age is binned into the following categories based on clinician input and an example in literature: 18-25, 26-35, 36-45, 46-55, 56-65, 66-75, 76-85, 86+. The USRDS predefined categories for race, sex, and dialysis modality were used for the fairness assessment.

Steps for running the 6_logistic_regression_fairness.ipynb script

This script calculates the ROC AUC for specific groups of patients to assess the fairness of the final model.

- Input:

```
medexpreesrd
2021_final_LR_model_test_pred_proba_pooled.pickle
```

- Output:

```
complete_fair1.pickle
2021_lr_fairness.csv
```

Step 1. Import libraries

```
import numpy as np
import pandas as pd
import sys
import pickle

#path to the functions directory
sys.path.append('/ML-AI-PCOR-2021/onc_functions/')

from fairness import get_fairness_assessment

import datetime
dte = datetime.datetime.now()
dte = dte.strftime("%Y")

print('python-' + sys.version)
```

Step 2. Get the columns of data required to compute fairness assessment and save them.

```
con = create_engine('postgresql://username:password@location/dbname')
df = pd.read_sql_query('''SELECT usrds_id, died_in_90, inc_age, sex,
dialtyp, race, hispanic, subset FROM medxpreesrd;''', con)
```

Step 3. Save

```
with open('complete_fair1.pickle', 'wb') as picklefile:
    pickle.dump(df, picklefile)
```

Step 4. Import the pooled results from the LR model

```
with open('./results/2021_final_LR_model_test_pred_proba_pooled.pickle',
'rb') as f:
        y_pred = pickle.load(f)
```

Step 5. Merge the fairness details with the results

```
data = df.merge(y_pred, on=['usrds_id','died_in_90','subset'])
```

Step 6. Calculate fairness. The function **get_fairness_assessment** is imported from the /onc_functions/fairness.py file. This function calculates AUC and the confusion matrix from the model prediction scores for specific groups.

```
def get_fairness_assessment(data,
                            y_proba_col_name,
                            y_true_col_name):

    #turn the continuous age variable into age categories
    df['agegroup'] = pd.cut(df.inc_age,
                            bins=[17, 25, 35, 45, 55, 65, 75, 85, 90],
                            labels=[1, 2, 3, 4, 5, 6, 7, 8])

    df = df.drop(columns=['inc_age'])

    #replace NaNs with a large number that does not appear in the data,
effectively creating another category for missing values
    df.loc[:,['race','dialtyp','hispanic']] = df.loc[:,
['race','dialtyp','hispanic']].fillna(100.0, axis=1).copy()

    #Identify the cols for the fairness assessment
    fairness_cols = ['agegroup', 'sex','dialtyp', 'race','hispanic']

    #loop through all categories and values to get counts, auc, and
confusion matrix
    rows_list = []
    for col in fairness_cols:
        for name, c in df.groupby(col):
            fairness_dict = {}
            fairness_dict['Feature'] = col
            fairness_dict['Value'] = name
            fairness_dict['Count'] = c.shape[0]
```

```
            fairness_dict['AUC'] = roc_auc_score(c[y_true_col_name],
c[y_proba_col_name])
            tn, fp, fn, tp = confusion_matrix(y_true = c[y_true_col_name],
                                              y_pred =
np.where(c[y_proba_col_name] >= 0.5, 1, 0)).ravel()
            fairness_dict['TN'] = tn
            fairness_dict['FP'] = fp
            fairness_dict['FN'] = fn
            fairness_dict['TP'] = tp
            rows_list.append(fairness_dict)

    #convert results from a list to a dataframe
    df_fairness = pd.DataFrame(rows_list)
    return df_fairness
```

Step 7. Calculate the assessment and save the results.

```
fairness = get_fairness_assessment(data,
                                   y_proba_col_name='score',
                                   y_true_col_name='died_in_90')
fairness.to_csv('./results/2021_lr_fairness.csv')
```

	Feature	Value	Count	AUC	TN	FP	FN	TP
0	agegroup	1.0	4340	0.829859	4261	33	40	6
1	agegroup	2.0	12774	0.825866	12382	180	177	35
2	agegroup	3.0	26120	0.828693	24568	971	396	185
3	agegroup	4.0	53564	0.803657	46993	4756	1013	802
4	agegroup	5.0	85076	0.788509	66144	14608	1803	2521
5	agegroup	6.0	86140	0.768390	50356	28260	1810	5714
6	agegroup	7.0	62193	0.740362	23076	30849	991	7277
7	agegroup	8.0	15098	0.724801	3033	9097	136	2832
8	sex	1.0	198347	0.816653	134009	49691	3614	11033
9	sex	2.0	146957	0.803552	96803	39063	2752	8339
10	dialtyp	1.0	310415	0.802400	202318	84026	5928	18143
11	dialtyp	2.0	15082	0.831254	14448	354	202	78
12	dialtyp	3.0	13295	0.844102	12748	276	195	76

13	dialtyp	4.0	77	0.976027	55	18	0	4
14	dialtyp	100.0	6436	0.741123	1244	4080	41	1071
15	race	1.0	230577	0.802574	141142	69658	4160	15617
16	race	2.0	93560	0.812754	72489	16061	1826	3184
17	race	3.0	3225	0.804476	2672	425	57	71
18	race	4.0	12965	0.837901	10499	1889	219	358
19	race	5.0	3776	0.809844	3245	363	86	82
20	race	6.0	881	0.776969	556	264	13	48
21	race	9.0	321	0.756385	210	94	5	12
22	hispanic	1.0	51021	0.831624	41485	7037	953	1546
23	hispanic	2.0	292532	0.805458	188814	80758	5402	17558
24	hispanic	9.0	1752	0.760173	514	959	11	268

Points to consider

Performing fairness assessment on the categories of interest gives additional insight into how the model performs by different patient categories of interest (by demographics, etc.). Future researchers should perform fairness assessments to better evaluate model performance, especially for models that may be deployed in a clinical setting. Other methods of assessing fairness include evaluating true positives, sensitivity, positive predictive value, etc. at various threshold across the different groups of interest, which would allow selection of a threshold that balances model performance across the groups of interest.

6.3.3.7 Risk Assessment

Steps for running the 7_logistic_regression_risk.ipynb script

- Input:

```
complete_fair1.pickle
2021_final_LR_model_test_pred_proba_pooled.pickle
```

- Output:

```
2021_lr_risk_cat.csv
```

Step 1. Import libraries

```
import numpy as np
import pandas as pd
import pickle

import sys
#path to the functions directory
sys.path.append('/ML-AI-PCOR-2021/onc_functions/')

from risk import get_risk_categories

print('python-' + sys.version)
import datetime
dte = datetime.datetime.now()
dte = dte.strftime("%Y%m%d")
```

Step 2. Import the details from the fairness assessment

```
with open('complete_fair1.pickle','rb') as f:
    details = pickle.load(f)
```

Step 3. Import the pooled results from the LR model

```
with open('./results/2021_final_LR_model_test_pred_proba_pooled.pickle',
'rb') as f:
        y_pred = pickle.load(f)
```

Step 4. Merge the details with the results

```
data = df.merge(y_pred, on=['usrds_id','died_in_90','subset'])
```

Step 5. Calculate risk. The function **get_risk_categories** is imported from the /onc_functions/risk.py file.

```
def get_risk_categories(dataset, y_proba_col_name, y_true_col_name):

    test_x_pd = dataset[dataset.subset > 6].copy().sort_values(by =
'usrds_id')
    del dataset

    df = test_x_pd.loc[:,[y_true_col_name,y_proba_col_name]]

    #construct the risk categories from the predicted score
    df['risk_categories'] = pd.cut(df[y_proba_col_name],
```

```
                                    bins=[-0.1, 0.09, 0.19, 0.29, 0.39,
0.49, 0.59, 0.69, 0.79, 0.89, 0.99],
                                    labels=['0-0.09', '0.1-0.19', '0.2-
0.29', '0.3-0.39', '0.4-0.49',
                                            '0.5-0.59','0.6-0.69','0.7-
0.79','0.8-0.89','0.9-0.99'])

    #loop through all the categories to get the predicted score
    risk_list = []
    for name, c in df.groupby('risk_categories'):
        risk_dict = {}
        risk_dict['Risk Category'] = name
        risk_dict['Count'] = c[y_true_col_name].shape[0]
        risk_dict['Count Died in 90'] = c[y_true_col_name].sum()
        risk_dict['Count Survived'] = c[y_true_col_name].shape[0]-
c[y_true_col_name].sum()
        risk_dict['Percent Died in 90'] =
c[y_true_col_name].sum()/c[y_true_col_name].shape[0]

        risk_list.append(risk_dict)

    df_risk = pd.DataFrame(risk_list)
    return df_risk
```

Run the function above

```
risk_cat = get_risk_categories(data,
                               y_proba_col_name='score',
                               y_true_col_name='died_in_90')
```

Step 6. Save

```
risk_cat.to_csv('./results/' + str(dte) + '_lr_risk_cat.csv')
```

	Risk Category	Count	Count Died in 90	Count Survived	Percent Died in 90
0	0-0.09	43117	145.0	42972.0	0.003363
1	0.1-0.19	56041	633.0	55408.0	0.011295
2	0.2-0.29	51733	1103.0	50630.0	0.021321
3	0.3-0.39	44494	1770.0	42724.0	0.039781
4	0.4-0.49	38156	2412.0	35744.0	0.063214
5	0.5-0.59	33358	3208.0	30150.0	0.096169
6	0.6-0.69	29338	4074.0	25264.0	0.138864
7	0.7-0.79	24342	4765.0	19577.0	0.195752
8	0.8-0.89	17561	4878.0	12683.0	0.277775
9	0.9-0.99	7141	2740.0	4401.0	0.383700

6.3.4 Artificial Neural Network--Multilayer Perceptron (MLP) Model

This section containts the code for the artificial neural net model - multilayer perceptron: MLP. The repository contains ipython notebooks to train an artificial neural net to classify patients as `survived` or `died_in_90`. Tensorflow is the library used to create and train the neural network. For a more detailed explanation of neural networks or tensorflow, we recommend the tutorials at https://www.tensorflow.org/tutorials.

Environment

These ipython notebooks can easily be run in a plain tensorflow docker container or used on their own in a conda or other ipython environment.

The environment used for the MLP model was purchased on Amazon Web Services (AWS):

```
Name: p2.16xlarge
vCPU: 64
Cores: 32
Threads per core: 2
Architecture: x86_64
Memory: 732 GB
GPU: 16
GPU Memory: 12 GB
GPU Manufacturer: NVIDIA
GPU Name: K80
Operating System: Linux (Ubuntu 20.04 Focal Fossa)
Network Performance: 25 GB
Zone: US govcloud west
```

This code takes approximately 3 days to run all the sections.

The MLP model was trained using the python (version 3.6.9) tensorflow (version 2.4.1) library and cross validation methods were from the Python (version 3.6.9) library scikit learn (version 0.24.1) and the following libraries:

Python library	Version
tensorflow	2.4.1
scikit-learn	0.24.1
numpy	1.19.5
pandas	1.1.5
matplotlib	3.3.3

Points to consider

An instance with GPUs can be utilized for tuning a large number of hyperparameters, as done here. If multiple cores or a GPU are not available, recommend choosing only a few hyperparameters to tune at one time.

6.3.4.1 Run Docker container (optional)

Run the following command in the same directory where you store the jupyter notebook (the repository). This will first pull the container, then instantiate it and mount the current directory to allow you to access the notebooks and also save any changes.

Step 1. Use the following code to run the container on your local machine:

```
sudo docker run -it -v "$PWD":/tf/notebooks --name onc_tf_1
tensorflow/tensorflow:latest-jupyter
```

Step 2. Then open a browser and go to the default port mapping

```
localhost:8888
```

Step 3. There you will see this directory of notebooks that you can open and run.

6.3.4.2 Run on a server (i.e. AWS)

Ports will need to be mapped when running the container. In our case 8080 is the port that is open and we can access it from the local machine at that port if connected to the server

```
sudo docker run -it -p 8080:8888 -v "$PWD":/tf/notebooks --name onc_tf_1
tensorflow/tensorflow:latest-jupyter
```

To utilize a GPU (which can be useful for the cross-validation step).

```
sudo docker run -it -p 8080:8888 -v "$PWD":/tf/notebooks --name onc_tf_gpu
tensorflow/tensorflow:latest-gpu-jupyter
```

Then you can access the current directory and notebooks at https://[my_aws_ec2_address]:8080

Points to consider

To run the cross-validation hyperparameter tuning with a GPU, run it from a python file (a conda environment worked) not the ipython jupyter notebook.

6.3.4.3 Pre-processing the data

The preprocessing is the same as for the logistic regression model and includes one hot encoding of categorical features and removal of features with missing values in lieu of the binary features.

Steps for running the 1_mlp_preprocessing.ipynb script

- Input:

```
medexpressesrd
micecomplete_pmm
numeric_columns.pickle
```

- Output:

```
complete1.pickle
complete2.pickle
complete3.pickle
complete4.pickle
complete5.pickle
```

Step 1. Install and import libraries

Install the python packages into the docker container (if running in a container) and import them into the notebook.

```
!pip install --upgrade pip
!pip install --upgrade scikit-learn
!pip install pandas
!pip install psycopg2-binary
!pip install sqlalchemy
!pip install seaborn
```

```
import psycopg2
import sqlalchemy
from sqlalchemy import create_engine

# other libraries
import numpy as np
import pandas as pd
import sys
import pickle

#plotting
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

Step 2. Connect to the Postgres database

The credentials for the Postgres database will be inserted here.

```
con = create_engine('postgresql://username:password@address/dbname')
```

Step 3. Get data

Load the full non-imputed data found in the `medexpressesrd` table from Postgres database.

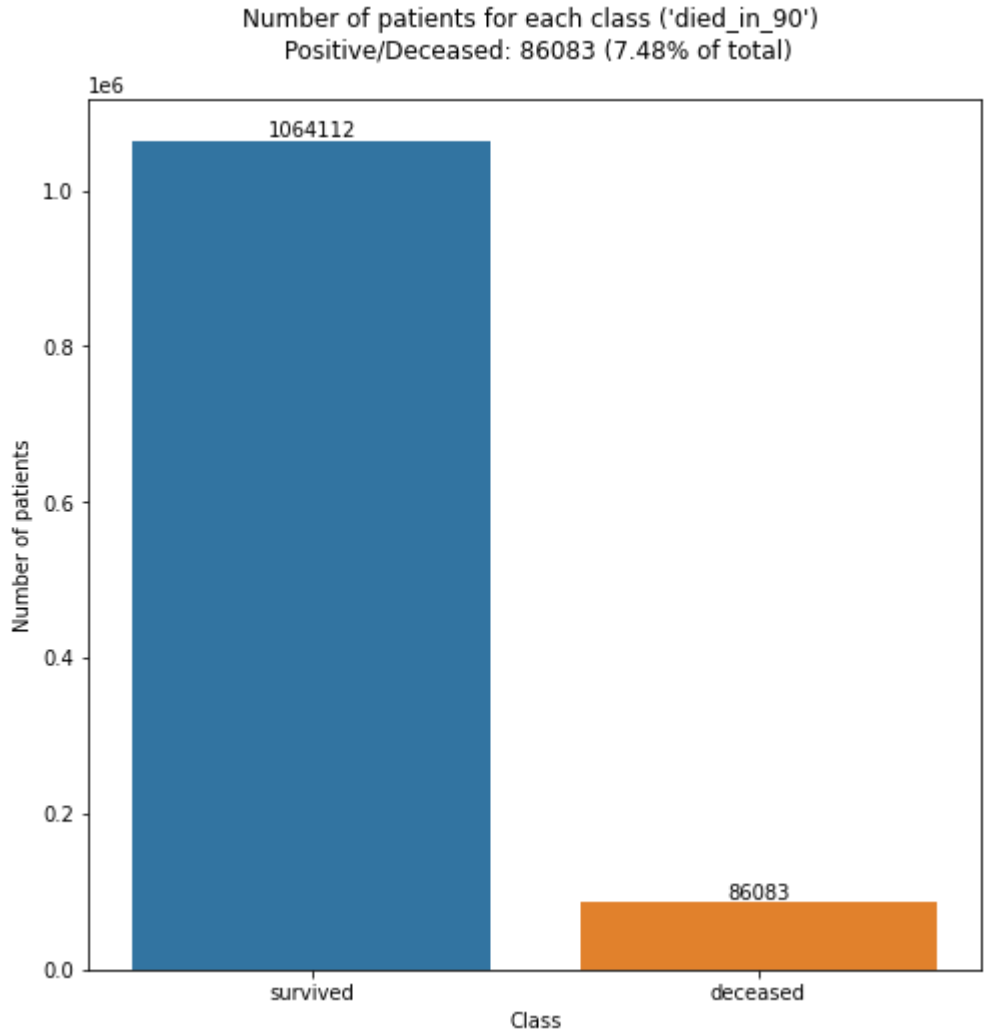
```
df = pd.read_sql_query('''SELECT * FROM medxpreesrd;''', con)
```

Get counts for each class (this gets used lated when we train the model)

```
neg_class_count, positive_class_count = np.bincount(df['died_in_90'])
```

The labels are 2 integers, 0 (survived) or 1 (deceased). These correspond to the class. Note that we have a class imbalance with deceased being the minority class.

```
Label   Class   Count
0   survived    1064112
1   deceased    86083
```



Step 4. Remove missing data

Neural networks models cannot handle missing values. The columns to remove are loaded at the top of the notebook in the variable vars_to_remove and were chosen based on clinician input.

For this dataset, the columns of pre-ESRD claims data that have missing values (claim counts, etc) were removed but kept the binary features from the Medicare pre-ESRD claims, which include indicators for claims in each setting, indicator for pre-ESRD claims, and indicators for each diagnosis group.

```
df.drop(columns=vars_to_remove,inplace=True)
```

Step 5. Encode categorical features

One variable **dial_train_time** was created from taking the difference between 2 dates in the medevid table. This feature was the only non-claims-related feature to have a large number of missing values, but we did not want to drop it so we encoded it in the following way

- a number greater than zero=1
- 0=0
- missing=3

Thus, the feature is turned into a categorical rather than numeric variable to retain some (though not all) information.

```
df.dial_train_time = df.dial_train_time.fillna(-1)
df.dial_train_time=df.dial_train_time.astype(int).clip(lower=-1,upper=1)
df.dial_train_time=df.dial_train_time.astype(str).replace("-1","na")
```

Use dummy variables for categorical variables (loaded at the top of the notebook).

Get the list of categorical variables that have more then 2 levels, then encode using pandas get_dummies function.

```
dummy_list = []
for col in categoryVars:
    u = len(df[col].unique())
    if u>2:
        dummy_list.append(col)

df = pd.concat([df,
pd.get_dummies(df.loc[:,dummy_list].astype('str'))],axis=1).drop(columns=d
ummy_list,axis=1)
```

Step 6. Load imputed data

Import imputed data `micecomplete_pmm` table from Postgres.

```
imp = pd.read_sql_query('''SELECT *, row_number()
 OVER(PARTITION BY usrds_id) AS impnum
 FROM micecomplete_pmm
''', con)
```

Step 7. Remove the 5 columns from the original data

```
df.drop(columns=["height", "weight", "bmi", "sercr", "album", "gfr_epi",
"heglb"],inplace=True)
df.shape = (1150195, 290)
```

Step 8. Separate the imputed data into 5 dataframes

Separating the 5 imputed datasets into different dataframes makes it easier to store, load, and compute.

```
imp1 = imp[imp.impnum==1]
imp2 = imp[imp.impnum==2]
imp3 = imp[imp.impnum==3]
```

```
imp4 = imp[imp.impnum==4]
imp5 = imp[imp.impnum==5]
```

Step 9. Merge the encoded data with each of the 5 imputed datasets

This is a left merge on the non-imputed data based on the **usrds_id** and the **subset** number.

```
complete1 = pd.merge(df, imp1, how='left', on=["usrds_id","subset"])
complete2 = pd.merge(df, imp2, how='left', on=["usrds_id","subset"])
complete3 = pd.merge(df, imp3, how='left', on=["usrds_id","subset"])
complete4 = pd.merge(df, imp4, how='left', on=["usrds_id","subset"])
complete5 = pd.merge(df, imp5, how='left', on=["usrds_id","subset"])

complete5.shape
(1150195, 298)
```

Step 10. Save the data

Save each set to the current directory as a pickle file (a file type that works well for pandas dataframes).

```
with open('complete1.pickle', 'wb') as picklefile:
    pickle.dump(complete1, picklefile)
with open('complete2.pickle', 'wb') as picklefile:
    pickle.dump(complete2, picklefile)
with open('complete3.pickle', 'wb') as picklefile:
    pickle.dump(complete3, picklefile)
with open('complete4.pickle', 'wb') as picklefile:
    pickle.dump(complete4, picklefile)
with open('complete5.pickle', 'wb') as picklefile:
    pickle.dump(complete5, picklefile)
```

Points to consider

The approach used to handle missing values is dependent on the dataset and the features in the dataset. Clinical expertise is crucial in understanding the impact of missing values and whether or not they should be imputed, removed, or replaced.

6.3.4.4 Hyperparameter tuning

This script computes the 5-fold cross-validation gridsearch to find the best hyperparameters for the MLP model. Keras (tensorflow) is used to create the layers of the neural net and take advantage of multiple GPUs (if available). The gridsearch cross-validation is computed using with sci-kit learn GridSearchCV. The class imbalance is handled by using the class_weights parameter. This hyperparameter cannot be tuned as part of the grid search and the function mlp_cv must be run for each set of weights.

Steps for running the 2_mlp_cross_val.ipynb script

- Input:

```
build_mlp.py
complete1.pickle
complete2.pickle
complete3.pickle
complete4.pickle
complete5.pickle
numeric_columns.pickle
```

-Output: x = [1,2,3,4,5]

```
2021_grid_best_params_imp_x_weight_m.pickle
2021_grid_best_auc_imp_x_weight_m.pickle
2021_grid_cv_results_imp_x_weight_m.pickle

2021_grid_best_params_imp_x_weight_5.pickle
2021_grid_best_auc_imp_x_weight_5.pickle
2021_grid_cv_results_imp_x_weight_5.pickle

2021_grid_best_params_imp_x_weight_10.pickle
2021_grid_best_auc_imp_x_weight_10.pickle
2021_grid_cv_results_imp_x_weight_10.pickle

2021_grid_best_params_imp_x_weight_20.pickle
2021_grid_best_auc_imp_x_weight_20.pickle
2021_grid_cv_results_imp_x_weight_20.pickle
```

Step 1. Import packages

Tensorflow is a popular framework for training a neural network and keras is a high-level API used for ease of access to the tensorflow functions.

```
!pip install --upgrade scikit-learn
!pip install pandas
!pip install pyyaml h5py
!pip install seaborn

# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
from tensorflow.keras.optimizers import Adam, SGD, Adamax

import sklearn.metrics as metrics
from sklearn.metrics import auc, plot_roc_curve, roc_curve,
mean_squared_error, accuracy_score, roc_auc_score, classification_report,
confusion_matrix, log_loss
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler
```

```
import sklearn

# load custom function for building the NN
from build_mlp import build_mlp

# other libraries
import numpy as np
import pandas as pd
import sys
import pickle

#plotting
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

Step 2. Import list of numeric columns

The following code snippet obtains the list of numeric columns to be scaled

```
with open('numeric_columns.pickle', 'rb') as f:
    nu_cols = pickle.load(f)
```

Step 3. Import a set of data and scale numeric columns

Each imputed set should be run separately, so only import one set at a time. The example below imports imputation 5:

```
with open('complete5.pickle', 'rb') as f:
        dataset = pickle.load(f)
```

Keep only the training data subsets (1-6) since we are only running a cross validation to obtain the optimal parameters for the model.

```
X_train =  dataset[dataset.subset <= 6].copy().sort_values(by =
'usrds_id')
```

Separate the labels (typically denoted as 'y') and save as an array.

```
y_train = np.array(X_train.pop('died_in_90'))
```

Scale only the numeric columns with an sklearn library **StandardScaler**.

```
scaler = StandardScaler()
X_train[nu_cols] = scaler.fit_transform(X_train[nu_cols])
```

Save the training data as an array (rather than a pandas dataframe) and drop the non-feature columns.

```
    X_train = np.array(X_train.drop(columns=
['subset','usrds_id','impnum']))
```

Step 4. Create the grid for the grid search

Insert the different parameters you want to test when doing the cross validation. For a detailed explanation of each parameter, see the section on the *Build_mlp.py* script below.

```
        neurons = [16, 32, 64, 128]
        layers = [1, 2]
        kernel_regularizer = ['l2']
        dropout_rate = [ 0.1, 0.2, 0.4, 0.5, 0.6]
        learn_rate = [.001, .0001, .0002]
        activation = ['relu', 'sigmoid', 'tanh']
        optimizer = ['Adam']
        output_bias = [None]
        epochs = [10, 20] # 1mill/256=4000 steps for one pass thru dataset
        batches = [512, 256]
```

Step 5. Set parameters for early stopping (to optimize time)

Early stopping for the epochs based on the auc_pr (area under the precision-recall curve).

```
early_stopping = tf.keras.callbacks.EarlyStopping(
                            monitor='auc_pr' ,
                            verbose=1,
                            patience=10,
                            mode='max',
                            restore_best_weights=True)
```

Step 6. Build layers

Use the Keras wrapper for scikitlearn and our custom build_mlp function imported from the custom python script build_mlp.py

```
 weighted_model_skl = KerasClassifier(build_fn=build_mlp,
                                      verbose=0)
```

Points to consider

Standardization and scaling of numeric features allows for comparison of multiple features in different units. The model will learn the importance of features better and faster when it is not overwhelmed by a feature with a much larger range than the others.

6.3.4.5 Building layers and compiling the model

This script is used to create the neural network by building the layers and then compiling the model. This model will consist of a few simple layers (Dense and Dropout) to illustrate the concepts and give an example.

Wrapper function

In the actual notebook, the full code is wrapped in a function (**mlp_cv**) for ease of running the cross-validation with the different weightings and each imputation dataset. An example of the code for imputation 5 is included below:

```
mlp_cv(class_weight_20, weight_name=20, imputation=5)
```

Steps for running the build_mlp.py script

Step 1. Import tensorflow packages to create the layers of the network.

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.layers.experimental import preprocessing
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.constraints import max_norm
from tensorflow.keras.metrics import AUC
```

Step 2. Select metrics to measure the loss and the accuracy of the model.

Multiple metrics can be reported as we have done here.

```
METRICS = [
      tf.keras.metrics.TruePositives(name='tp'),
      tf.keras.metrics.FalsePositives(name='fp'),
      tf.keras.metrics.TrueNegatives(name='tn'),
      tf.keras.metrics.FalseNegatives(name='fn'),
      tf.keras.metrics.BinaryAccuracy(name='accuracy'),
      tf.keras.metrics.Precision(name='precision'),
      tf.keras.metrics.Recall(name='recall'),
      tf.keras.metrics.AUC(name='auc'),
      tf.keras.metrics.AUC(name='auc_pr',
            num_thresholds=200,
            curve="PR",
```

```
            summation_method="interpolation",
            dtype=None,
            thresholds=None,
            multi_label=False,
            label_weights=None)
]
```

Step 3. Define a function to take in the parameters for building the network.

```
def build_mlp(
    layers=2,
    neurons=16,
    output_bias=None,
    optimizer='Adam',
    activation='relu',
    learn_rate=.0002,
    dropout_rate=0.2,
    kernel_regularizer='l2',
    metrics=METRICS
):
    if output_bias is not None:
        output_bias = tf.keras.initializers.Constant(output_bias)
    model = tf.keras.Sequential()
```

This loop allows for us to test one or more identical dense layers based on the **layers** argument. Dense or fully connected layers consist of the number of neurons(nodes) defined in the **neurons** argument, an activation function defined by **activation**, an input shape of 294 (this is specific to our data since we always have 294 features or columns. If a feature is added or removed, this number needs to reflect that), and a kernel regularizer defined by the **kernel_regularizer** argument.

```
for i in range(layers):
        model.add(Dense(
                        neurons,
                        activation=activation,
                        input_shape=(294,),
                        kernel_regularizer=kernel_regularizer))
```

A Dropout layer is used to avoid overfitting when testing the model. The parameter **dropout_rate** determines the rate.

```
model.add(Dropout(dropout_rate))
```

The final layer is a dense layer with a sigmoid activation function to give the probablility of each class for each sample.

```
model.add(Dense(
                1,
                activation='sigmoid',
                bias_initializer=output_bias))
```

Step 4. Set the final arguments for compiling the model.

- Optimizer —This is how the model is updated based on the data it sees and its loss function.
- Loss function —This measures how accurate the model is during training. You want to minimize this function to "steer" the model in the right direction. BinaryCrossentropy() is used because we are training a binary classification model.
- Metrics —Used to monitor the training and testing steps. Here we use multiple metrics that are printed out.

```
    opt = optimizer(lr=learn_rate)

    model.compile(
      optimizer=opt,
      loss=tf.keras.losses.BinaryCrossentropy(),
      metrics=metrics)

    return model
```

Step 5. Instantiate the grid search cross validation model

Evaluate our list of parameters using the sci-kit learn package **GridSearchCV** to run a 5-fold cross validation. This will result in the best combination of these parameters according to our score (which is set to **average_precision** to optimize precision and recall). Multiple processors can be taken advantage of by setting n_jobs=-1. Training many different parameters will take a significant amount of time that depends on the number of processors and size of the data.

```
grid = GridSearchCV(
                weighted_model_skl,
                param_grid=params,
                cv=5,
                scoring='average_precision',
                return_train_score=True,
                n_jobs=-1
            )
```

Step 6. Class imbalance and fit the model to the data

When fitting the model to the training data, the class imbalanced can be set using the **class_weight** parameter.

```
grid_result = grid.fit(
                X_train,
                y_train,
                class_weight=selected_class_weight,
                callbacks=[early_stopping]
            )
```

Note that when testing multiple class weights, this must be done by fitting a new cross-validated model (running the entire script) for each different set of class weights to be tested, rather than as a hyperparameter in the model. Multiple class weights were tested on the data.

- Classes: 0=survived (negative class), 1=died_in_90 (positive class).

```
total = 1150195
positive_class_count =  86083      #(7.48% of total)
neg_class_count = 1064112     #(92.52% of total)
# Scaling by total/2 helps keep the loss to a similar magnitude.
# The sum of the weights of all examples stays the same.
weight_for_0 = (1 / neg_class_count)*(total)/2.0
weight_for_1 = (1 / positive_class_count)*(total)/2.0

class_weight_m = {0: weight_for_0, 1: weight_for_1}
```

- A range of stronger weights the minority class.

```
class_weight_5 = {0: 1, 1: 5}
class_weight_10 = {0: 1, 1: 10}
class_weight_20 = {0: 1, 1: 20}
```

Step 7. Summarize and print results

```
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']

for mean, stdev, param in zip(means, stds, params):
                    print("%f (%f) with: %r" % (mean, stdev, param))
print("Best: %f using %s" % (grid_result.best_score_,
grid_result.best_params_))
```

Step 8. Save results

Save the result for each set of class weight parameters (to select the one with the best score) and for each dataset (each imputation) and review the best_params_ from each one to check for similarity.

```
    with open('./results/2021_grid_best_params_imp_' + str(imputation) +
'_weight_' + str(weight_name) + '.pickle', 'wb') as f:
                    pickle.dump(grid_result.best_params_, f)

    with open('./results/2021_grid_best_auc_imp_' + str(imputation) +
'_weight_' + str(weight_name)  + '.pickle','wb') as f:
                    pickle.dump(grid_result.best_score_, f)

    with open('./results/2021_grid_cv_results_imp_' + str(imputation) +
'_weight_' + str(weight_name) + '.pickle','wb') as f:
                    pickle.dump(grid_result.cv_results_, f)
```

Points to consider

1. One imputed set of data was used to tune hyperparameters which reduced computational time while effectively tuning the parameters.

2. Due to the severe class imbalance in the dataset, the AUC ROC tends to be high while recall is low. It is well-known that precision-recall plots are more informative than AUC ROC plots when training a binary classification model on severely imbalanced data; therefore, the average precision metric from sklearn (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html#sklearn.metrics.average_precision_score) was used to tune the MLP model.

sklearn.metrics.average_precision_score

sklearn.metrics. **average_precision_score**(*y_true*, *y_score*, ***, *average='macro'*, *pos_label=1*, *sample_weight=None*) [source]

Compute average precision (AP) from prediction scores.

AP summarizes a precision-recall curve as the weighted mean of precisions achieved at each threshold, with the increase in recall from the previous threshold used as the weight:

$$\text{AP} = \sum_n (R_n - R_{n-1}) P_n$$

where P_n and R_n are the precision and recall at the nth threshold [1]. This implementation is not interpolated and is different from computing the area under the precision-recall curve with the trapezoidal rule, which uses linear interpolation and can be too optimistic.

6.3.4.6 Final MLP Model

Steps for running the 3_mlp_final_model.ipynb script

This script trains and evaluates the final MLP model based on the best parameters from the hyperparameter tuning/cross-validation step.

- Input:

```
complete1.pickle
complete2.pickle
complete3.pickle
complete4.pickle
complete5.pickle
numeric_columns.pickle
```

- Output: x=[1-5]

```
2021_MLP_final_results_imp_x.pickle
2021_MLP_final_model_imp_x.h5
2021_MLP_final_eval_imp_x.pickle

2021_MLP_final_fpr.pickle
2021_MLP_final_tpr.pickle
2021_MLP_final_auc.pickle

2021_MLP_final_ytest_all.pickle
2021_MLP_final_ypred_all.pickle

2021_MLP_final_prec.pickle
2021_MLP_final_recall.pickle
2021_MLP_final_avgprec_thresh.pickle
2021_MLP_final_avgprec_score.pickle
```

Step 1. Install and import libraries

```
!pip install --upgrade scikit-learn
!pip install pandas
!pip install pyyaml h5py
!pip install seaborn

# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam

import sklearn
from sklearn.metrics import auc, average_precision_score, roc_curve,
precision_recall_curve, roc_auc_score, confusion_matrix
from sklearn.preprocessing import StandardScaler

# other libraries
import numpy as np
import pandas as pd
import sys
import pickle

#plotting
```

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

Step 2. Import list of numeric columns

Get list of columns to be scaled

```
with open('numeric_columns.pickle', 'rb') as f:
    nu_cols = pickle.load(f)
```

Step 3. Import data

Import the data from a single imputation.

```
imp=5
with open('complete' + str(imp) + '.pickle', 'rb') as f:
        dataset = pickle.load(f)
```

Separate the training set (subsets 0-6).

```
X_train =  dataset[dataset.subset <= 4].copy().sort_values(by =
'usrds_id')
```

Separate the labels.

```
y_train = np.array(X_train.pop('died_in_90'))
```

Separate the validation set (subsets 5-6). This validation set gets used as the 'test' set of data while the model is being trained to avoid ever 'looking' at the test set until we evaluate the model.

```
X_val = dataset[(dataset.subset == 6) | (dataset.subset ==
5)].copy().sort_values(by = 'usrds_id')
```

Separate labels.

```
y_val = np.array(X_val.pop('died_in_90'))
```

Separate the test set (subsets 7 8 9) for evaluating the model to see how well it performs on new data. Sorting by usrds_id is important so that we can calculate the fairness (or you could just run the predictions again and save the order).

```
X_test =  dataset[dataset.subset > 6].copy().sort_values(by = 'usrds_id')
```

Separate labels.

```
y_test = np.array(X_test.pop('died_in_90'))
```

Scale only the numeric columns with an sklearn library **StandardScaler** by fitting the scaler model on the training set, and use this model and transform on the validation and test sets. Then drop the non-feature columns and save as an array.

```
scaler = StandardScaler()
X_train[nu_cols] = scaler.fit_transform(X_train[nu_cols])
X_train = np.array(X_train.drop(columns=['subset','usrds_id','impnum']))

X_val[nu_cols] = scaler.transform(X_val[nu_cols])
X_val = np.array(X_val.drop(columns=['subset','usrds_id','impnum']))

X_test[nu_cols] = scaler.transform(X_test[nu_cols])
X_test = np.array(X_test.drop(columns=['subset','usrds_id','impnum']))
```

Step 3. Build final model

Build the final model based on the best hyperparameters from the previous cross-validation step. First set the metric(s) to report.

```
METRICS = [
      tf.keras.metrics.TruePositives(name='tp'),
      tf.keras.metrics.FalsePositives(name='fp'),
      tf.keras.metrics.TrueNegatives(name='tn'),
      tf.keras.metrics.FalseNegatives(name='fn'),
      tf.keras.metrics.BinaryAccuracy(name='accuracy'),
      tf.keras.metrics.Precision(name='precision'),
      tf.keras.metrics.Recall(name='recall'),
      tf.keras.metrics.AUC(name='auc'),
      tf.keras.metrics.AUC(name='auc_pr',
            num_thresholds=200,
            curve="PR",
            summation_method="interpolation",
            dtype=None,
            thresholds=None,
            multi_label=False,
```

```
            label_weights=None)
]
```

Define a function to create the model layers and complie the model. Insert the best hyperparameters from the cross validation tuning. For details on each line, see the above explanation of the **build_mlp.py** script.

```
def final_build_mlp(
    layers=2,
    neurons=16,
    output_bias=None,
    optimizer='Adam',
    activation='relu',
    learn_rate=.0002,
    dropout_rate=0.2,
    kernel_regularizer='l2',
    metrics=METRICS
):

    model = tf.keras.Sequential()

    #add 2 dense layers
    for i in range(layers):
        model.add(Dense(
                        neurons,
                        activation=activation,
                        input_shape=(294,),
                        kernel_regularizer=kernel_regularizer)
        )
    model.add(Dropout(dropout_rate))
    model.add(Dense(
                    1,
                    activation='sigmoid',
                    bias_initializer=output_bias))
    opt = Adam(lr=learn_rate)
    model.compile(
      optimizer=opt,
      loss=tf.keras.losses.BinaryCrossentropy(),
      metrics=metrics)

    return model
```

Step 4. Instantiate the final model

```
final_model = final_build_mlp()
```

Step 5. Train (fit) the final model

Set the optimal parameters for fitting the model based on results from the cross-validation.

```
class_weight_10 = {0: 1, 1: 10}
epochs_final = 10
batches = 256
```

Fit the model to the training and validation data.

```
final_history = final_model.fit(
        X_train,
        y_train,
        batch_size=batches,
        epochs=epochs_final,
        validation_data=(X_val, y_val),
        class_weight=class_weight_10)
```

Step 6. Get predictions on the training set from the model

```
train_predictions_final = final_model.predict(
                                            X_train,
                                            batch_size=batches
    )
```

Step 7. Evaluate the final model

Evaluate the final model that has been trained on the new data (test set).

```
test_predictions_final = final_model.predict(
                                            X_test,
                                            batch_size=batches
                                              )
final_eval = final_model.evaluate(
                                  X_test,
                                  y_test,
                                  batch_size=batches,
                                  verbose=1
                                    )
```

Print results of test set.

```
    res = {}
    for name, value in zip(final_model.metrics_names, final_eval):
        print(name, ': ', value)
        res = {name : value}
```

Points to consider

Standardization and scaling of numeric features allows for comparison of multiple features in different units. The model will learn the importance of features better and faster when it isn't overwhelmed by a feature with a much larger range than the others. Scaling and standardization should be done separately for training and test sets so that we are not looking at the test set.

Neural network models are computationally expensive and there are many parameters to tune and decisions to make. It is beneficial to consult with other experts and test the code before running the full set of data. Multiple CPUs or GPUs can speed up computation time for the cross validation hyperparameter tuning step.

6.3.4.7 Pool Results

Steps for running 4_pool_results_from_imputations.ipynb

- Input:

```
medexpreesrd
2021_MLP_final_ytest_all.pickle
2021_MLP_final_ypred_all.pickle
```

- Output:

```
2021_final_MLP_model_test_pred_proba_pooled.pickle
```

Step 1. Import libraries

```
import pickle
import numpy as np
import pandas as pd

import psycopg2
import sqlalchemy
from sqlalchemy import create_engine
```

Step 2. Import results from the MLP model

```
with open('./results/2021_MLP_final_ytest_all.pickle', 'rb') as f:
    y_true_mlp = pickle.load(f)

with open('./results/2021_MLP_final_ypred_all.pickle', 'rb') as
picklefile:
    y_pred_proba_mlp = pickle.load(picklefile)
```

Step 3. Reformat into a pandas dataframe. This makes the data easier to work with.

The index for the values is zero here b/c we only saved it for the positive class, otherwise it would be 1

```
pooled = pd.DataFrame()
pooled['imp1']=y_pred_proba_mlp[0][:,0]
pooled['imp2']=y_pred_proba_mlp[1][:,0]
pooled['imp3']=y_pred_proba_mlp[2][:,0]
pooled['imp4']=y_pred_proba_mlp[3][:,0]
pooled['imp5']=y_pred_proba_mlp[4][:,0]
```

Step 4. Calculate the mean and standard deviation of the predicted probability for the positive class (died_in_90) for each patient/row.

```
pooled['score'] = pooled.mean(axis=1)
pooled['std_'] = pooled.std(axis=1)
```

Step 5. Import the details from the original data

Enter your credentials for your postgres database.

```
con = create_engine('postgresql://username:password@location/dbname')

dataset = pd.read_sql_query('''SELECT  usrds_id, died_in_90, subset FROM
medxpreesrd;''', con)
```

Step 6. Sort the values so they are in the same order as when the MLP model was calculated and keep only the test set.

```
dataset = dataset[dataset.subset > 6].copy().sort_values(by =
'usrds_id').reset_index(drop=True)
```

Step 7. Merge the details with the pooled predictions.

```
pooled = pooled.merge(dataset, left_index=True, right_index=True)
```

Step 8. Save

```
with open('./results/2021_final_MLP_model_test_pred_proba_pooled.pickle',
'wb') as picklefile:
        pickle.dump(pooled, picklefile)
```

6.3.4.8 Plot Results

Steps for running the 5_plot_mlp_roc_auc.ipynb script

- Input:

```
2021_final_MLP_model_test_pred_proba_pooled.pickle
```

- Output:

```
mlp_roc_auc_bw.png
2021_mlp_confusion_matrix.csv
```

Step 1. Import libraries

```
import pandas as pd
import numpy as np
import pickle

import sys
#path to the functions directory
sys.path.append('/ML-AI-PCOR-2021/onc_functions/')

#import custom plotting functions
from plot_functions import onc_calc_cm, onc_plot_roc,
onc_plot_precision_recall
```

Step 2. Load results from the pooled model results

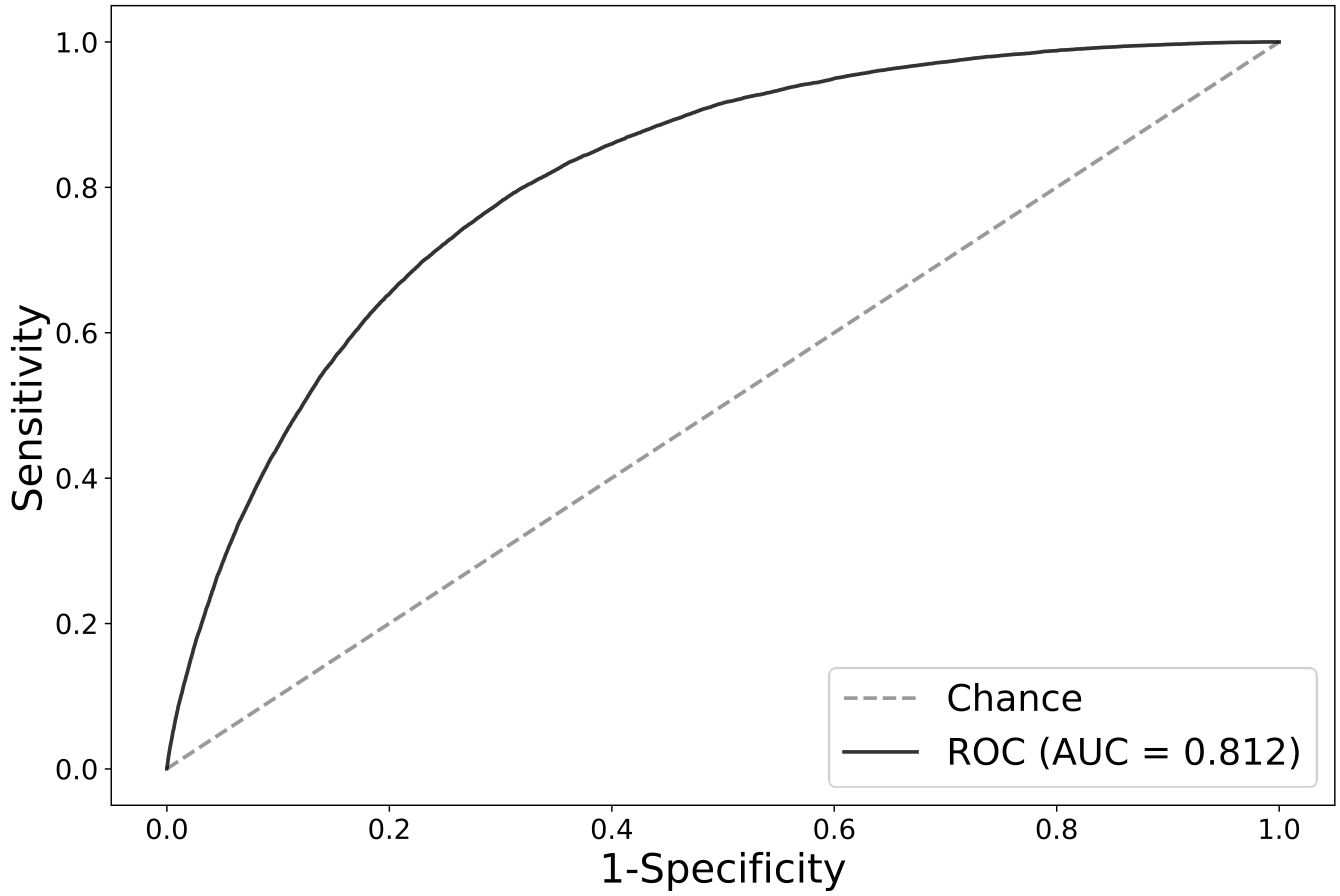
```
with open('./results/2021_final_MLP_model_test_pred_proba_pooled.pickle',
'rb') as f:
        results = pickle.load(f)

results = results.loc[:,['score','died_in_90','subset','usrds_id']]
results = results.rename(columns={'died_in_90':'y'})
```

Step 3. Plot the ROC AUC. This function **onc_plot_roc** is located and imported from /onc_functions/plot_functions.py

```
def onc_plot_roc(y_true, y_pred, model_name, **kwargs):
    '''
    Plot the ROC AUC and return the test ROC AUC results.
    INPUT: y_true, y_pred, model_name, **kwargs
```

```
    '''

    #calc values for plot
    false_positives, true_positives, threshold = roc_curve(y_true, y_pred)
    c_roc_auc_score = auc(false_positives, true_positives)

    #set figure params
    fig1 = plt.figure(1, figsize=(12,30),dpi=400)
    ax1 = plt.subplot2grid((7, 1), (0, 0), rowspan=2)

    #plot reference line for chance
    ax1.plot([0, 1], [0, 1], linestyle='--', lw=2, color='gray',
        label='Chance', alpha=.8)

    # plot AUC ROC
    ax1.plot(false_positives, true_positives,
        label=r'ROC (AUC = %0.3f)' % (c_roc_auc_score),
        lw=2, alpha=.8, color = 'k')

    # additional figure params
    ax1.set(xlim=[-0.05, 1.05], ylim=[-0.05, 1.05],)
    ax1.legend(loc="lower right")
    plt.xlabel('1-Specificity')
    plt.ylabel('Sensitivity')
    plt.rc('axes', labelsize=22)    # fontsize of the x and y labels
    plt.rc('xtick', labelsize=15)    # fontsize of the tick labels
    plt.rc('ytick', labelsize=15)    # fontsize of the tick labels
    plt.rc('legend', fontsize=20)    # legend fontsize
    # save plot
    plt.savefig(model_name + "_roc_auc_bw.png",  dpi=400,
transparent=True)
    plt.show()
```

```
onc_plot_roc(
            y_true=results.y,
            y_pred=results.score,
            model_name='mlp');
```



Step 4. Print and save the performance metrics at multiple thresholds

```
def onc_calc_cm(y_true, y_predictions, range_probas=[0.1,0.5]):
    '''
    Plot the confusion matrix and scores for multiple thresholds
    '''
    df = pd.DataFrame(index = range_probas,
                      columns=['threshold','sensitivity','specificity',

'likelihood_ratio_neg','likelihood_ratio_pos',

'tp','fp','tn','fn','total_survived','total_deceased',])
    for proba_threshold in range_probas:

        cm = confusion_matrix(y_true, y_predictions > proba_threshold)
        tn = cm[0][0]
        fp = cm[0][1]

        sensitivity = recall_score(y_true, y_predictions >
proba_threshold)
        specificity = tn / (tn + fp)

        df.loc[proba_threshold, "threshold"] = proba_threshold
        df.loc[proba_threshold,"sensitivity"] = sensitivity
        df.loc[proba_threshold, "specificity"] = specificity
        df.loc[proba_threshold, "likelihood_ratio_neg"] = (1-
```

```
sensitivity)/specificity
        df.loc[proba_threshold, "likelihood_ratio_pos"] = sensitivity/(1-
specificity)
        df.loc[proba_threshold, "tp"] = cm[1][1]
        df.loc[proba_threshold, "fp"] = fp
        df.loc[proba_threshold, "tn"] = tn
        df.loc[proba_threshold, "fn"] = cm[1][0]
        df.loc[proba_threshold, "total_survived"] = np.sum(cm[0])
        df.loc[proba_threshold, "total_deceased"] = np.sum(cm[1])
    return df
```

```
cm = onc_calc_cm(
    results.y,
    results.score,
    range_probas=[.10,.20, .30, .40, .50])
cm.to_csv('./results/2021_mlp_confusion_matrix.csv')
cm
```

threshold	sensitivity	specificity	likelihood_ratio_neg	likelihood_ratio_pos	tp	fp	tn	fn
0.1	0.987839	0.201204	0.0604413	1.23666	25425	255269	64298	313
0.2	0.949763	0.399753	0.12567	1.58229	24445	191819	127748	1293
0.3	0.894631	0.540735	0.194864	1.94796	23026	146766	172801	2712
0.4	0.822985	0.651134	0.271856	2.35903	21182	111486	208081	4556
0.5	0.726824	0.746807	0.365792	2.87064	18707	80912	238655	7031

6.3.4.9 Fairness assessment

ML models can perform differently for different categories of patients, so the MLP model was assessed for fairness, or how well the model performs for each category of interest (demographics—sex, race, and age—as well as initial dialysis modality). Age is binned into the following categories based on UCSF clinician input and an example in literature: 18-25, 26-35, 36-45, 46-55, 56-65, 66-75, 76-85, 86+. The USRDS predefined categories for race, sex, and dialysis modality were used for the fairness assessment.

Steps to running the 6_mlp_fairness_assessment.ipynb script

This script calculates the ROC AUC for specific groups of patients to assess the fairness of the final model.

- Input:

```
medexpreesrd
2021_final_MLP_model_test_pred_proba_pooled.pickle
```

- Output:

```
complete_fair1.pickle
2021_mlp_fairness.csv
```

Step 1. Import libraries

```
import numpy as np
import pandas as pd
import pickle
import sys
#path to the functions directory
sys.path.append('/ML-AI-PCOR-2021/onc_functions/')

import psycopg2
import sqlalchemy
from sqlalchemy import create_engine

from fairness import get_fairness_assessment
```

Step 2. Get the columns of data required to compute fairness assessment and save them

```
con = create_engine('postgresql://username:password@location/dbname')
df = pd.read_sql_query('''SELECT usrds_id, died_in_90, inc_age, sex,
dialtyp, race, hispanic, subset FROM medxpreesrd;''', con)
```

Step 3. Save

```
with open('complete_fair1.pickle', 'wb') as picklefile:
    pickle.dump(df, picklefile)
```

Step 4. Import the pooled results from the LR model

```
with open('./results/2021_final_MLP_model_test_pred_proba_pooled.pickle',
'rb') as f:
        proba = pickle.load(f)
```

Step 5. Merge the fairness details with the results

```
data = df.merge(proba, on=['usrds_id','subset','died_in_90'])
```

Step 6. Calculate fairness. The function **get_fairness_assessment** is imported from the /onc_functions/fairness.py file. This function calculates AUC and the confusion matrix from the model

prediction scores for specific groups.

```
def get_fairness_assessment(data,
                            y_proba_col_name,
                            y_true_col_name):

    #turn the continuous age variable into age categories
    df['agegroup'] = pd.cut(df.inc_age,
                           bins=[17, 25, 35, 45, 55, 65, 75, 85, 90],
                           labels=[1, 2, 3, 4, 5, 6, 7, 8])

    df = df.drop(columns=['inc_age'])

    #replace NaNs with a large number that does not appear in the data,
effectively creating another category for missing values
    df.loc[:,['race','dialtyp','hispanic']] = df.loc[:,
['race','dialtyp','hispanic']].fillna(100.0, axis=1).copy()

    #Identify the cols for the fairness assessment
    fairness_cols = ['agegroup', 'sex','dialtyp', 'race','hispanic']

    #loop through all categories and values to get counts, auc, and
confusion matrix
    rows_list = []
    for col in fairness_cols:
        for name, c in df.groupby(col):
            fairness_dict = {}
            fairness_dict['Feature'] = col
            fairness_dict['Value'] = name
            fairness_dict['Count'] = c.shape[0]

            fairness_dict['AUC'] = roc_auc_score(c[y_true_col_name],
c[y_proba_col_name])
            tn, fp, fn, tp = confusion_matrix(y_true = c[y_true_col_name],
                                              y_pred =
np.where(c[y_proba_col_name] >= 0.5, 1, 0)).ravel()
            fairness_dict['TN'] = tn
            fairness_dict['FP'] = fp
            fairness_dict['FN'] = fn
            fairness_dict['TP'] = tp
            rows_list.append(fairness_dict)

    #convert results from a list to a dataframe
    df_fairness = pd.DataFrame(rows_list)
    return df_fairness
```

Step 7. Calculate the assessment and save the results.

```
fairness = get_fairness_assessment(data,
                                   y_proba_col_name='score',
```

```
                                    y_true_col_name='died_in_90'
)
fairness.to_csv('./results/2021_mlp_fairness.csv')
```

	Feature	Value	Count	AUC	TN	FP	FN	TP
0	agegroup	1.0	4340	0.851780	4236	58	35	11
1	agegroup	2.0	12774	0.834839	12325	237	159	53
2	agegroup	3.0	26120	0.833624	24592	947	378	203
3	agegroup	4.0	53564	0.810153	47642	4107	1055	760
4	agegroup	5.0	85076	0.790628	68047	12705	1961	2363
5	agegroup	6.0	86140	0.768169	52807	25809	2032	5492
6	agegroup	7.0	62193	0.740282	25405	28520	1219	7049
7	agegroup	8.0	15098	0.720045	3601	8529	192	2776
8	sex	1.0	198347	0.818053	137712	45988	3930	10717
9	sex	2.0	146957	0.804818	100942	34924	3101	7990
10	dialtyp	1.0	310415	0.803432	208694	77650	6484	17587
11	dialtyp	2.0	15082	0.837490	14460	342	208	72
12	dialtyp	3.0	13295	0.849120	12739	285	187	84
13	dialtyp	4.0	77	0.989726	69	4	0	4
14	dialtyp	100.0	6436	0.753809	2693	2631	152	960
15	race	1.0	230577	0.803584	146754	64046	4636	15141
16	race	2.0	93560	0.815380	74295	14255	1975	3035
17	race	3.0	3225	0.807649	2744	353	60	68
18	race	4.0	12965	0.837525	10681	1707	240	337
19	race	5.0	3776	0.819267	3290	318	86	82
20	race	6.0	881	0.782887	614	206	21	40
21	race	9.0	321	0.766060	277	27	13	4
22	hispanic	1.0	51021	0.834179	42020	6502	992	1507
23	hispanic	2.0	292532	0.806653	195829	73743	6003	16957
24	hispanic	9.0	1752	0.766108	806	667	36	243

Points to consider

Performing the fairness assessment on the categories of interest gives additional insight into how the model performs by different patient categories of interest (by demographics, etc.). Future researchers should perform fairness assessments to better evaluate model performance, especially for models that may be deployed in a clinical setting. Other methods of assessing fairness include evaluating true positives, sensitivity, positive predictive value, etc. at various threshold across the different groups of interest, which would allow selection of a threshold that balances model performance across the groups of interest.

6.3.4.10 Risk assessment

Steps for running the 7_mlp_risk.ipynb script

- Input:

```
complete_fair1.pickle
2021_final_MLP_model_test_pred_proba_pooled.pickle
```

- Output:

```
2021_mlp_risk_cat.csv
```

Step 1. Import libraries

```
import numpy as np
import pandas as pd
import pickle

import sys
#path to the functions directory
sys.path.append('/ML-AI-PCOR-2021/onc_functions/')

from risk import get_risk_categories

print('python-' + sys.version)
import datetime
dte = datetime.datetime.now()
dte = dte.strftime("%Y%m%d")
```

Step 2. Import the details from the fairness assessment

```
with open('complete_fair1.pickle','rb') as f:
    details = pickle.load(f)
```

Step 3. Import the pooled results from the model

```
with open('./results/2021_final_MLP_model_test_pred_proba_pooled.pickle',
'rb') as f:
        y_pred = pickle.load(f)
```

Step 4. Merge the details with the results

```
data = df.merge(y_pred, on=['usrds_id','died_in_90','subset'])
```

Step 5. Calculate risk. The function **get_risk_categories** is imported from the /onc_functions/risk.py file.

```
def get_risk_categories(dataset, y_proba_col_name, y_true_col_name):

    test_x_pd = dataset[dataset.subset > 6].copy().sort_values(by =
'usrds_id')
    del dataset

    df = test_x_pd.loc[:,[y_true_col_name,y_proba_col_name]]

    #construct the risk categories from the predicted score
    df['risk_categories'] = pd.cut(df[y_proba_col_name],
                                   bins=[-0.1, 0.09, 0.19, 0.29, 0.39,
0.49, 0.59, 0.69, 0.79, 0.89, 0.99],
                                   labels=['0-0.09', '0.1-0.19', '0.2-
0.29', '0.3-0.39', '0.4-0.49',
                                           '0.5-0.59','0.6-0.69','0.7-
0.79','0.8-0.89','0.9-0.99'])

    #loop through all the categories to get the predicted score
    risk_list = []
    for name, c in df.groupby('risk_categories'):
        risk_dict = {}
        risk_dict['Risk Category'] = name
        risk_dict['Count'] = c[y_true_col_name].shape[0]
        risk_dict['Count Died in 90'] = c[y_true_col_name].sum()
        risk_dict['Count Survived'] = c[y_true_col_name].shape[0]-
c[y_true_col_name].sum()
        risk_dict['Percent Died in 90'] =
c[y_true_col_name].sum()/c[y_true_col_name].shape[0]

        risk_list.append(risk_dict)

    df_risk = pd.DataFrame(risk_list)
    return df_risk
```

Run the function above

```
risk_cat = get_risk_categories(data,
                               y_proba_col_name='score',
                               y_true_col_name='died_in_90')
```

Step 6. Save

```
risk_cat.to_csv('./results/' + str(dte) + '_mlp_risk_cat.csv')
```

	Risk Category	Count	Count Died in 90	Count Survived	Percent Died in 90
0	0-0.09	57014	245.0	56769.0	0.004297
1	0.1-0.19	66582	928.0	65654.0	0.013938
2	0.2-0.29	47796	1359.0	46437.0	0.028433
3	0.3-0.39	37752	1791.0	35961.0	0.047441
4	0.4-0.49	33388	2423.0	30965.0	0.072571
5	0.5-0.59	30980	3234.0	27746.0	0.104390
6	0.6-0.69	30237	4652.0	25585.0	0.153851
7	0.7-0.79	29631	6861.0	22770.0	0.231548
8	0.8-0.89	11472	4036.0	7436.0	0.351813
9	0.9-0.99	453	209.0	244.0	0.461369