NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

SUPPLEMENTAL TO THESIS
(1 of 1: Python and Matlab Scripts)

PREDICTIVE MAINTENANCE USING MACHINE LEARNING AND EXISTING DATA SOURCES

by

William J. Frazier

September 2022

Thesis Advisor: Neil C. Rowe
Co-Advisor: Ying Zhao

Approved for public release. Distribution is unlimited.

```python
# ------------------------------------- Models.py ----------------------------------------#
# Captain William Frazier
# NPS Thesis research
# This program performs the Cox Proportional Hazards model evaluation for the data generated.
# In its current state, running the program will generate flight data specific to the
# malfunction code being evaluated. When data is split into training and test data, the test
# data has not failed, because the split is performed on a specific date. Therefore, the
# SernoHistory and RemovalHistory programs need to re-run entirely, using the split date as
# the end date for flight data. This program will require several minutes to run for each
# malfunction code. Upon completion, all .csv files required to replicate the evaluation will
# already be saved into this directory.
#
# This program also has all of the Weibull analsis methods and comparison code used in the
# thesis work. However, it is not implemented in this program's current state. Any questions
# on replicating that methodology can be referred to the author.

from copy import deepcopy
import csv
from datetime import date, timedelta
import scipy.stats as s
from scipy.special import gammaln
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import weibull_min, exponweib, weibull_max, norm, chi2, beta, linregress, \
trim_mean
import math
from predictr import Analysis
import weibull
import reliability
import pandas as pd
from lifelines import KaplanMeierFitter
from lifelines import CoxPHFitter
from statistics import mean
import FlightHours
import RemovalHistory

### ------------------------- Steps for running this program ---------------------------- ###

## - Requires Access to NAVAIR DECKPLATE Query Studio
## - Requirements for FlightHours.py, RemovalHistory.py and SernoHistory.py to be satisfied.

class Models(object):

    def __init__(self):

        # Indicies for data columns
        self.TTF = 0
```

```
        self.MODE = 1
        self.MCN = 2
        self.RAW_MCN = 0
        self.RAW_SERNO = 1
        self.RAW_PN= 2

        # PCA Data from V22 FST Weibull Analysis
        self.models_input_filename = ''
        self.models_data_filename = ''
        self.models_test_filename = ''
        self.failures = []
        self.num_failures = 0
        self.data_header = ''
        self.failures_by_mal = {}
        self.modes_to_ignore = ['0', '813', '815']
        self.sernos = {}
        self.data_at_test_data = {}
        self.mal_codes_to_eval = []
        self.split_dates = []
        self.split = .80

        # Indicies
        self.ACTION_DATE = 0
        self.PART_NO = 1
        self.CAGE = 2
        self.SERNO = 3
        self.WUC = 4
        self.WEC_DESC = 5
        self.ORG = 6
        self.ORG_DESC = 7
        self.TEC = 8
        self.TMS = 9
        self.BUNO = 10
        self.REM_INS_FLAG = 11
        self.JCN = 12
        self.MCN = 13
        self.TYPE_MAINT = 14
        self.TRANS_CODE = 15
        self.ACT_TAKEN = 16
        self.MAL_CODE = 17
        self.COMP_DATE = 18
        self.TIME_1_CODE = 19
        self.TIME_1_CYCLE = 20
        self.TIME_2_CODE = 21
        self.TIME_2_CYCLE = 22
        self.TIME_3_CODE = 23
        self.TIME_3_CYCLE = 24
```

```
        self.HRS = 25
        self.LNDGs = 26
        self.TMR_1 = 27
        self.TMR_2 = 28
        self.TMR_3 = 29
        self.TMR_4 = 30
        self.TMR_5 = 31
        self.TMR_6 = 32
        self.TMR_7 = 33
        self.NON_AUST_LNDGS = 34
        self.AUST_LNDGS = 35
        self.SHIP_LNDGS = 36
        self.CENS = 37

        # Results tables
        self.results_header = ['Failure Mode','# Failures', 'MTTF', 'Eta','Beta','Rsqr',
                               'AbPval']
        self.results = [self.results_header]
        self.final_results = [self.results_header]

    def read_data(self):
        ''' Reads in the final data of TTF and Mode from a csv file. '''

        new_data = []
        with open(self.models_input_filename, 'r') as file:
            reader = csv.reader(file)
            for row in reader:
                new_data.append(row)

        self.data_header = new_data[0]
        new_data.pop(0)
        self.failures = new_data

    def write_data(self):
        ''' Connects the SerNo from FRC's Raw Data to the resultant data via MCN. '''

        with open(self.models_data_filename, 'w', newline='') as file:
            writer = csv.writer(file)
            writer.writerow(self.data_header)
            writer.writerows(self.failures)

    def remove_zeros(self):
        ''' Removes any TTFs == 0 which could cause NaN or Inf in weibull calc. '''

        without_zeros = [row for row in self.failures if (float(row[self.HRS]) != float(0))]

        self.failures = without_zeros
```

```
    def ensure_floats(self):
        ''' Ensures TTF in data is in proper format. '''

        for row in range(len(self.failures)):
            self.failures[row][self.HRS] = float(self.failures[row][self.HRS])

    def censor_flag(self):
        ''' Adds the censored boolean column for all TTF data for weibull module. '''

        self.data_header.append('Censor Flag')
        for row in range(len(self.failures)):
            self.failures[row].append(False)

    def sort_by_mal(self):
        ''' Sorts all failure data by malfunction code. '''

        failures = deepcopy(self.failures)
        for row in failures:

            if row[self.MAL_CODE] not in self.failures_by_mal:
                self.failures_by_mal[row[self.MAL_CODE]] = [row]
            else:
                self.failures_by_mal[row[self.MAL_CODE]].append(row)

    def get_split_dates(self, mal_code):
        ''' Gets all train/test split dates for mal codes to be evaluated. '''

        failures = deepcopy(self.failures_by_mal[mal_code])
        count = 0
        for row in failures:

            if count <= float(len(failures) * self.split):
                count += 1
            else:
                split_date = row[self.ACTION_DATE]
                mdy = split_date.split("-")
                return date(int(mdy[0]), int(mdy[1]), int(mdy[2]))

    def split_data_weibull(self, mal_code, num_failures, split_perc):

        train_failures = []
        train_suspensions = []
        test_failures = []
        num_train_failures = 0
        num_test_failures = 0
        num_train_suspensions = 0
```

```
        count = 0
        end_date = 0
        date_flag = 0

        for row in self.failures:

            if count <= float(num_failures * split_perc):

                if str(row[self.MAL_CODE]) == str(mal_code):
                    train_failures.append(row[self.HRS])
                    num_train_failures += 1
                    count += 1
                else:
                    train_suspensions.append(row[self.HRS])
                    num_train_suspensions += 1

            else:
                if date_flag == 0:
                    end_date = row[self.ACTION_DATE]
                    date_flag = 1

                if str(row[self.MAL_CODE]) == str(mal_code):
                    test_failures.append(row[self.HRS])
                    num_test_failures += 1


        return train_failures, num_train_failures, train_suspensions, num_train_suspensions, \
               test_failures, num_test_failures, end_date

    def split_data_CPH(self, mal_code, num_failures, split_perc):

        train_failures = []
        test_failures = []
        num_train_failures = 0
        num_test_failures = 0
        num_train_suspensions = 0
        count = 0
        end_date = 0
        date_flag = 0

        for row in self.failures:

            if count <= float(num_failures * split_perc):

                if str(row[self.MAL_CODE]) == str(mal_code):
                    row[self.CENS] = 1
                    train_failures.append(row)
```

```
                    num_train_failures += 1
                    count += 1
                else:
                    row[self.CENS] = 0
                    train_failures.append(row)
                    num_train_suspensions += 1

            else:
                if date_flag == 0:
                    end_date = row[self.ACTION_DATE]
                    date_flag = 1

                if str(row[self.MAL_CODE]) == str(mal_code):
                    row[self.CENS] = 1
                    test_failures.append(row)
                    num_test_failures += 1
                else:
                    row[self.CENS] = 0
                    test_failures.append(row)
                    num_train_suspensions += 1


        return train_failures, num_train_failures, num_train_suspensions, test_failures, \
               num_test_failures, end_date

    def reliability_RRY(self, train_failures, train_suspensions, m, conf_int, mode):

        model = reliability.Fitters.Fit_Weibull_2P(failures=train_failures,
                                                   right_censored=train_suspensions,
                                                   show_probability_plot=True,
                                                   print_results=True, CI=conf_int,
                                                   quantiles=None, CI_type='time',
                                                   method=m, optimizer=None, force_beta=None,
                                                   downsample_scatterplot=True)
        MTTF = model.alpha * math.exp( gammaln( 1 + (1 / model.beta)) )

        print('MTTF: ' + str(round(MTTF,2)) + '\n'
              'Eta: ' + str(round(model.alpha,2)) + '\n'
              'Beta: ' + str(round(model.beta,2)))

        return round(float(MTTF),2), round(float(model.loglik),2), round(float(model.AICc),2)

    def cumulative_probability_of_failure(self, beta, eta, mode):

        x_a = [100,200, 300, 400, 500, 1000, 1500, 2000]
        x_b = [x for x in range(0,5000)]
        prob = []
```

```
        f = []
        for hours_a in x_a:
            for hours_b in x_b:
                f.append(1 - math.exp( ((hours_a/eta)**beta) - ((hours_b/eta)**beta) ))
            prob.append(f)
            f = []

        plt.plot(x_b, prob[0], label = 'Current Hours = ' + str(x_a[0]))
        plt.plot(x_b, prob[1], label = 'Current Hours = ' + str(x_a[1]))
        plt.plot(x_b, prob[2], label = 'Current Hours = ' + str(x_a[2]))
        plt.plot(x_b, prob[3], label = 'Current Hours = ' + str(x_a[3]))
        plt.plot(x_b, prob[4], label = 'Current Hours = ' + str(x_a[4]))
        plt.plot(x_b, prob[5], label = 'Current Hours = ' + str(x_a[5]))
        plt.plot(x_b, prob[6], label = 'Current Hours = ' + str(x_a[6]))
        plt.plot(x_b, prob[7], label = 'Current Hours = ' + str(x_a[7]))
        plt.ylim([0,1])
        plt.xlabel('Flight Hours')
        plt.ylabel('Unreliability')
        plt.title('Conditional Probability of Failure Malfunction Code ' + str(mode))
        plt.legend()
        plt.show()

    def CoxPropHazard(self, mal_code, train_failures, a, p):

        test_failures = []
        cph_data = []
        for key in self.data_at_test_data.keys():
            if str(self.data_at_test_data[key][self.MAL_CODE]) == str(mal_code):
                test_failures.append(self.data_at_test_data[key])

        df_train = pd.DataFrame({
                'TTF': [row[self.HRS] for row in train_failures],
                'TMR_1 Hours': [row[self.TMR_1] for row in train_failures],
                'TMR_2 Hours': [row[self.TMR_2] for row in train_failures],
                'TMR_3 Hours': [row[self.TMR_3] for row in train_failures],
                'TMR_4 Hours': [row[self.TMR_4] for row in train_failures],
                'TMR_5 Hours': [row[self.TMR_5] for row in train_failures],
                'TMR_6 Hours': [row[self.TMR_6] for row in train_failures],
                'TMR_7 Hours': [row[self.TMR_7] for row in train_failures],
                'Non-Austere Landings': [row[self.NON_AUST_LNDGS] for row in train_failures],
                'Austere Landings': [row[self.AUST_LNDGS] for row in train_failures],
                'Ship Landings': [row[self.SHIP_LNDGS] for row in train_failures],
                'Fail': [row[self.CENS] for row in train_failures]
            })

        df_test = pd.DataFrame({
                'TTF': [row[self.HRS] for row in test_failures],
```

```
                'TMR_1 Hours': [row[self.TMR_1] for row in test_failures],
                'TMR_2 Hours': [row[self.TMR_2] for row in test_failures],
                'TMR_3 Hours': [row[self.TMR_3] for row in test_failures],
                'TMR_4 Hours': [row[self.TMR_4] for row in test_failures],
                'TMR_5 Hours': [row[self.TMR_5] for row in test_failures],
                'TMR_6 Hours': [row[self.TMR_6] for row in test_failures],
                'TMR_7 Hours': [row[self.TMR_7] for row in test_failures],
                'Non-Austere Landings': [row[self.NON_AUST_LNDGS] for row in test_failures],
                'Austere Landings': [row[self.AUST_LNDGS] for row in test_failures],
                'Ship Landings': [row[self.SHIP_LNDGS] for row in test_failures],
            })
        print(df_test)
        cph = CoxPHFitter(alpha=a, penalizer=p)
        cph.fit(df_train, duration_col="TTF", event_col="Fail")
        cph.print_summary()
        plt.figure()
        if mal_code == '   ':
            mal_code = 'Blank (Failed Technical Directive Inspection)'
        plt.title('Malfunction Code ' + str(mal_code) + ' Coefficient Ranges')
        cph.plot(hazard_ratios=True)
        plt.show()
        cph.check_assumptions(df_train,p_value_threshold=0.05,show_plots=True)
        plt.figure()
        test_rows = df_test.iloc[:,1:11]
        print(test_rows)
        cph.predict_survival_function(test_rows).plot()

        plt.title('Malfunction Code ' + str(mal_code) +
                  ' Survival Function for Test Data Failures')
        plt.ylabel('Survival (1 - Unreliability)')
        plt.xlabel('Flight Hours')

        plt.legend(labels=df_test['TTF'], loc = 'lower left')
        plt.show()

        cph.predict_median(test_rows)
        cph.predict_expectation(test_rows)

        return round(float(cph.log_likelihood_),2), round(float(cph.AIC_partial_),2)

    def v22_fst_weibull(self):

        TTF = 0
        MODE = 1
        MCN = 2
        diff = []
```

```
        v22_fst_data = []
        with open('v22_fst_results_PCA.csv', 'r') as file:
            reader = csv.reader(file)
            for row in reader:
                v22_fst_data.append(row)

        my_data = deepcopy(self.failures)

        for entry in range(len(v22_fst_data)):
            for entry_2 in my_data:
                if v22_fst_data[entry][MCN] == entry_2[self.MCN]:
                    v22_fst_data[entry].extend(entry_2[self.HRS:self.CENS+1])

        failure_modes = []
        for entry in v22_fst_data:
            if entry[MODE] not in failure_modes:
                failure_modes.append(entry[MODE])

        for mode in failure_modes:
            train_failures = []
            train_suspensions = []

            for row in v22_fst_data:
                if len(row) > 3:

                    if str(row[MODE]) == str(mode):
                        train_failures.append(row[3])
                        diff.append(float(row[0]) - float(row[3]))

                    else:
                        train_suspensions.append(row[3])
            print('Failure Mode: ' + str(mode) + ' reliability - RRY ' +
                  str(len(train_failures)) + ' failures, ' + str(len(train_suspensions)) + \
                  ' right censored\n')
            w_mttf, w_llh, w_aic = self.reliability_RRY(train_failures, train_suspensions,
                                                        'RRY', 0.95, mode)

        print(len(diff))
        print(mean(diff))

    def create_serno_removal_history(self):

        with open(self.models_test_filename, 'r') as file:
            reader = csv.reader(file)
            for row in reader:

                if row[self.SERNO] not in self.sernos.keys():
```

```
                    self.sernos[row[self.SERNO]] = [row]
                else:
                    self.sernos[row[self.SERNO]].append(row)

    def build_test_flight_data(self, test_data, end_date):

        for failure in test_data:
            serno = failure[self.SERNO]

            if serno not in self.data_at_test_data.keys():

                stopper = 0
                total_hrs = 0
                total_lndgs = 0
                total_TMR_1_hrs = 0
                total_TMR_2_hrs = 0
                total_TMR_3_hrs = 0
                total_TMR_4_hrs = 0
                total_TMR_5_hrs = 0
                total_TMR_6_hrs = 0
                total_TMR_7_hrs = 0
                total_NON_AUST_lndgs = 0
                total_AUST_lndgs = 0
                total_SHIP_lndgs = 0

                data = [str(total_hrs), str(total_lndgs), str(total_TMR_1_hrs),
                        str(total_TMR_2_hrs), str(total_TMR_3_hrs), str(total_TMR_4_hrs),
                        str(total_TMR_5_hrs), str(total_TMR_6_hrs), str(total_TMR_7_hrs),
                        str(total_NON_AUST_lndgs), str(total_AUST_lndgs),
                        str(total_SHIP_lndgs)]

                entry = 0
                while entry < len(self.sernos[serno]):

                    if stopper == 0:
                        if entry == 0:

                            self.sernos[serno][entry][self.HRS:self.CENS] = data

                        elif self.sernos[serno][entry][self.REM_INS_FLAG] == "I":
                            self.sernos[serno][entry][self.HRS:self.CENS] = data

                        else:
                            install = self.sernos[serno][entry-1][self.ACTION_DATE]
                            removal = self.sernos[serno][entry][self.ACTION_DATE]

                            mdy = install.split("-")
```

```
                    install = date(int(mdy[0]), int(mdy[1]), int(mdy[2]))

                    mdy = removal.split("-")
                    removal = date(int(mdy[0]), int(mdy[1]), int(mdy[2]))

                    mdy = str(end_date).split("-")
                    end_date = date(int(mdy[0]), int(mdy[1]), int(mdy[2]))

                    if removal > end_date:
                        removal = end_date
                        stopper = 1

                    buno = self.sernos[serno][entry][self.BUNO]

                    [flthrs, lndgs, TMR_1_hrs, TMR_2_hrs, TMR_3_hrs,
                     TMR_4_hrs, TMR_5_hrs, TMR_6_hrs, TMR_7_hrs,
                     NON_AUST_lndgs, AUST_lndgs, SHIP_lndgs]  = \
                        self.FlightHours.buno_query_all_data(buno, install, removal)

                    total_hrs += flthrs
                    total_lndgs += lndgs
                    total_TMR_1_hrs += TMR_1_hrs
                    total_TMR_2_hrs += TMR_2_hrs
                    total_TMR_3_hrs += TMR_3_hrs
                    total_TMR_4_hrs += TMR_4_hrs
                    total_TMR_5_hrs += TMR_5_hrs
                    total_TMR_6_hrs += TMR_6_hrs
                    total_TMR_7_hrs += TMR_7_hrs
                    total_NON_AUST_lndgs += NON_AUST_lndgs
                    total_AUST_lndgs += AUST_lndgs
                    total_SHIP_lndgs += SHIP_lndgs

                    data = [str(total_hrs), str(total_lndgs), str(total_TMR_1_hrs),
                            str(total_TMR_2_hrs), str(total_TMR_3_hrs),
                            str(total_TMR_4_hrs), str(total_TMR_5_hrs),
                            str(total_TMR_6_hrs), str(total_TMR_7_hrs),
                            str(total_NON_AUST_lndgs), str(total_AUST_lndgs),
                            str(total_SHIP_lndgs)]

                    self.sernos[serno][entry][self.HRS:self.CENS] = data
                    print(self.sernos[serno][entry])

                    if stopper == 1:
                        self.data_at_test_data[serno] = self.sernos[serno][entry]

                entry += 1
            entry += 1
```

```
    def build_test_flight_data_new(self, test_data):
        ''' Generates the test TTF data. '''
        for failure in test_data:
            serno = failure[self.SERNO]

            serno_history = deepcopy(self.sernos[serno])

            self.data_at_test_data[serno] = serno_history[-1]

# Main
# ----------------------------------------- Step 1 -----------------------------------------#
# Generate all removal data from the beginning of the program until today
# To perform a training/test split on a specific day, we need to extract that date

# Instantiate the Models Class
x = Models()

# Call the SernoRemovals class from RemovalHistory.py
# The cumulative flight data will be until today
end_date = date.today()
serno_history_save_filename = 'SERNO_HISTORY_WUC_27502X.csv'
removal_history_input_filename = serno_history_save_filename
removal_history_output_filename = 'REMOVAL_HISTORY_OUTPUT_WUC_27502X.csv'
y = RemovalHistory.SernoRemovals(end_date, removal_history_input_filename,
                                 removal_history_output_filename,
                                 serno_history_save_filename)

# ----------------------------------------- Step 2 -----------------------------------------#
# Determine the training/test split for each mal code evaluated
x.models_input_filename = removal_history_output_filename
x.models_test_filename = removal_history_output_filename
x.read_data()
x.remove_zeros()
x.ensure_floats()
x.censor_flag()
x.sort_by_mal()

for mal_code in x.failures_by_mal.keys():
    if len(x.failures_by_mal[mal_code]) >= 20 and str(mal_code) not in x.modes_to_ignore:
        x.mal_codes_to_eval.append(mal_code)
        x.split_dates.append(x.get_split_dates(mal_code))

# ----------------------------------------- Step 3 -----------------------------------------#
# Perform the CPH analysis for each mal code.

for mal_index in range(len(x.mal_codes_to_eval)):
```

```
    results = [['mal_code', 'method', 'CI', 'MTTF', 'Log Likelihood', 'AIC']]
    mal = x.mal_codes_to_eval[mal_index]
    mal_for_filename = mal
    if mal_for_filename == '   ':
        mal_for_filename ='Blank (TD)'

    print('-------------------------- MAL: ' + str(mal) + ' -------------------------------')

    end_date = x.split_dates[mal_index]
    serno_history_save_filename = 'SERNO_HISTORY_WUC_27502X_MAL_' + \
        str(mal_for_filename) + '.csv'
    removal_history_input_filename = serno_history_save_filename
    removal_history_output_filename = 'REMOVAL_HISTORY_OUTPUT_WUC_27502X_MAL_' + \
        str(mal_for_filename) + '.csv'
    y = RemovalHistory.SernoRemovals(end_date, removal_history_input_filename,
                                     removal_history_output_filename,
                                     serno_history_save_filename)

    x.models_input_filename = removal_history_output_filename
    x.models_test_filename = removal_history_output_filename
    x.models_data_filename = 'PCA_MODELS_MAL_' + str(mal_for_filename) + '.csv'
    x.read_data()
    x.remove_zeros()
    x.ensure_floats()
    x.censor_flag()
    x.sort_by_mal()
    x.write_data()
    x.create_serno_removal_history()

    num_failures = 0
    for row in x.failures:
        if str(row[x.MAL_CODE]) == str(mal):
            num_failures += 1

    # Split the data into train and test data
    train_failures, num_train_failures, num_train_suspensions, test_data, \
        num_test_failures, end_date = x.split_data_CPH(mal, num_failures, x.split)

    # based on the date the split occured, call RemovalHistory to get the flight data for the
    # test serial numbers at the time of split
    x.build_test_flight_data_new(test_data)

    alphas = [0.05]
    pens = [0]
    for alpha in alphas:
        for pen in pens:
```

```
            cph_llh, cph_aic = x.CoxPropHazard(mal, train_failures, alpha, pen)
            results.append([str(mal_for_filename), str(alpha), str(pen),
                            str(cph_llh), str(cph_aic)])
    print()
    print()

    with open('CPH_' + str(mal_for_filename) + '.csv', 'w', newline='') as file:
        writer = csv.writer(file)
        writer.writerows(results)
```

```python
# ---------------------------------- RemovalHistory.py -------------------------------------#
# Captain William Frazier
# NPS Thesis research
# This program takes the imputed and resolved removal data with cumulative flight data
# appended from SernoHistory.py and prepares it for reliability analysis. It removes
# duplicates of the same removal entry and removes entries based on a combination of
# malfunction code, transaction code, and action taken codes we do not wish to evaluate.
# The description of these entries are in the remove_admin_removals function.

import csv
from datetime import date, timedelta
import matplotlib.pyplot as plt
import numpy as np
from copy import deepcopy
import SernoHistory

### -------------------------- Steps for running this program --------------------------- ###

## - Requires SernoHistory.py to have run properly
## - Saves remaining removal data to a .csv

class SernoRemovals(object):

    def __init__(self, end_date, removal_history_input_filename,
                 removal_history_output_filename, serno_history_save_filename):

        self.data = []
        self.sernos = {}
        self.end_date = end_date
        self.removals_input_filename = removal_history_input_filename
        self.removals_output_filename = removal_history_output_filename
        self.header = ''
        self.sorted_filename = serno_history_save_filename

        # Indicies
        self.ACTION_DATE = 0
        self.PART_NO = 1
        self.CAGE = 2
        self.SERNO = 3
        self.WUC = 4
        self.WEC_DESC = 5
        self.ORG = 6
        self.ORG_DESC = 7
        self.TEC = 8
        self.TMS = 9
        self.BUNO = 10
        self.REM_INS_FLAG = 11
```

```
        self.JCN = 12
        self.MCN = 13
        self.TYPE_MAINT = 14
        self.TRANS_CODE = 15
        self.ACT_TAKEN = 16
        self.MAL_CODE = 17
        self.COMP_DATE = 18
        self.TIME_1_CODE = 19
        self.TIME_1_CYCLE = 20
        self.TIME_2_CODE = 21
        self.TIME_2_CYCLE = 22
        self.TIME_3_CODE = 23
        self.TIME_3_CYCLE = 24
        self.HRS = 25
        self.LNDGs = 26
        self.TMR_1 = 27
        self.TMR_2 = 28
        self.TMR_3 = 29
        self.TMR_4 = 30
        self.TMR_5 = 31
        self.TMR_6 = 32
        self.TMR_7 = 33
        self.NON_AUST_LNDGS = 34
        self.AUST_LNDGS = 35
        self.SHIP_LNDGS = 36

        # Instatiate Called by Models.py
        self.SernoHistory = SernoHistory.SernoHistory(self.end_date, self.sorted_filename)
        self.read_data()
        self.format_date()
        self.build_sernos()
        self.remove_duplicates()
        self.remove_admin_removals()
        self.write_data()

    def read_data(self):
        ''' Reads in the resolved DP-0025 report data in the csv generated from
        SernoHistory.py '''

        with open(self.removals_input_filename, 'r') as file:
            reader = csv.reader(file)

            for row in reader:
                self.data.append(row)

            # save the header row and pop it off
            self.header = self.data[0]
```

```
            self.data.pop(0)

    def format_date(self):
        ''' Converts maintenance action date string from .csv into datetime data type.
        Assumes .csv date is in MM/DD/YYYY and needs to be [YYYY, MM, DD]'''

        for row in range(0,len(self.data)):

            # Action Taken Date
            mdy = self.data[row][self.ACTION_DATE].split("/")
            if len(mdy) == 3:
                self.data[row][self.ACTION_DATE] = date(int(mdy[2]), int(mdy[0]), int(mdy[1]))
            else:
                mdy = self.data[row][self.ACTION_DATE].split("-")
                if len(mdy) == 3:
                    self.data[row][self.ACTION_DATE] = date(int(mdy[0]), int(mdy[1]),
                                                            int(mdy[2]))

            # Action Completion Date
            mdy = self.data[row][self.COMP_DATE].split("/")
            if len(mdy) == 3:
                self.data[row][self.COMP_DATE] = date(int(mdy[2]), int(mdy[0]), int(mdy[1]))
            else:
                mdy = self.data[row][self.COMP_DATE].split("-")
                if len(mdy) == 3:
                    self.data[row][self.COMP_DATE] = date(int(mdy[0]), int(mdy[1]),
                                                          int(mdy[2]))

    def sort_by_date(self, history):
        ''' Sorts an input list of historical R/I records by their action date value. '''

        history.sort(key=lambda x: x[self.ACTION_DATE])

        return history

    def display_records(self, records):
        ''' Prints the gathered R/I action records with proper formatting. '''

        max_lens = [len(str(max(i, key=lambda x: len(str(x))))) for i in zip(*records)]
        print('\n'.join('  '.join(item[i].ljust(max_lens[i]) for i in range(len(max_lens)))
                        for item in records))
        print()

    def build_sernos(self):
        ''' Generates master dictionary of sernos:[R/I actions]. If the SerNo is not yet a key
        in the dict, it is added and its value is the corresponding row of data i.e. the
        oldest action completed from the data. Subsequent rows with that same SerNo will
```

```
        have that data appended to the value list. '''

        for row in range(0,len(self.data)):
            key = str(self.data[row][self.SERNO])
            if key not in self.sernos:
                self.sernos[key] = [self.data[row]]
            else:
                self.sernos[key].extend([self.data[row]])

    def SerNo_query_short(self, history_list):
        ''' Returns the R/I history of a SerNo '''

        history_list.insert(0,self.header)

        # Convert dates to strings for formating
        for row in range(1,len(history_list)):
            history_list[row][self.ACTION_DATE] = \
                history_list[row][self.ACTION_DATE].strftime('%m/%d/%Y')
            history_list[row][self.COMP_DATE] = \
                history_list[row][self.COMP_DATE].strftime('%m/%d/%Y')

        # only the essential columns of data
        #indicies = [0,1,3,7,10,11,12,13,22,24]
        indicies = [self.ACTION_DATE,
                    self.PART_NO,
                    self.SERNO,
                    self.ORG_DESC,
                    self.BUNO,
                    self.TRANS_CODE,
                    self.ACT_TAKEN,
                    self.MAL_CODE,
                    self.HRS,
                    self.LNDGs,
                    self.TMR_1,
                    self.TMR_2,
                    self.TMR_3,
                    self.TMR_4,
                    self.TMR_5,
                    self.TMR_6,
                    self.TMR_7,
                    self.NON_AUST_LNDGS,
                    self.AUST_LNDGS,
                    self.SHIP_LNDGS]

        short_list = []

        for row in history_list:
```

```
            short_list.append([row[index] for index in indicies])

        # Print using formatting
        self.display_records(short_list)

    def get_history(self, serno):
        ''' Generates the list of R/I actions based on a serno and adds the header. '''

        history_list = deepcopy(self.sernos[serno])

        history_list = self.sort_by_date(history_list)

        return history_list

    def remove_duplicates(self):

        for serno in self.sernos.keys():

            new_history = []
            temp_history = deepcopy(self.sernos[serno])

            for removal in temp_history:

                if removal not in new_history:
                    new_history.append(removal)

            self.sernos[serno] = new_history

    def remove_admin_removals(self):
        ''' Removes any removal MAFs that are administrative in nature.
        Determined by transaction, action taken, and malfunction codes.
        Comments are from the NAMP 4790.2D Appendix E. '''

        for serno in self.sernos.keys():

            temp_history = deepcopy(self.sernos[serno])

            row = 0
            while row < len(temp_history):

                trans_code = temp_history[row][self.TRANS_CODE]
                mal_code = temp_history[row][self.MAL_CODE]
                act_code = temp_history[row][self.ACT_TAKEN]

                if str(trans_code) == '11':
                    # Could not duplicate or found within tolerances
                    if act_code == "A":
```

```
                temp_history.pop(row)
                row -= 1

            elif str(trans_code) == '16':
                # No Defect Removal
                temp_history.pop(row)
                row -= 1

            elif str(trans_code) == '17':
                # Installation, should not have a removal serno
                temp_history.pop(row)
                row -= 1

            elif str(trans_code) == '18':
                # Cannibalization or removal for defect
                pass

            elif str(trans_code) == '20':
                # Cannibalization
                pass

            elif str(trans_code) == '21':
                # repairable component is removed
                pass

            elif str(trans_code) == '23':
                # Removal and replacement of a defective, suspected defective, or
                # scheduled maintenance of a repairable component from an end item
                pass

            elif str(trans_code) == '30':

                # I Level test and check, component already removed
                temp_history.pop(row)
                row -= 1

            elif str(trans_code) == '31':

                # I Level maintenance, component already removed
                temp_history.pop(row)
                row -= 1

            elif str(trans_code) == '47':

                # TDs not resulting in a removal
                if act_code == "C":
                    temp_history.pop(row)
```

```
                        row -= 1

                row += 1

            row = 0
            self.sernos[serno] = temp_history

    def write_data(self):

        with open(self.removals_output_filename, 'w', newline='') as file:
            writer = csv.writer(file)
            writer.writerow(self.header)
            for serno in self.sernos.keys():
                writer.writerows(self.sernos[serno])
```

```python
# ------------------------------------ SernoHistory.py ---------------------------------------#
# Captain William Frazier
# NPS Thesis research
# This program takes component installation and removal data from the DP-0025 serial number
# tracking report from NAVAIR Deckplate, imputes missing removal or installation entries based
# on assumptions, removes any duplicates, and resolves entries when more than one component is
# installed in the same WUC position for a Buno. The results are deconflicted installation and
# removal entries which reflect what dates each serno was installed on a buno. Then, Flight
# Data from NAVFLIRs are then attributed to serialized components based on the flights
# recorded for bunos they were installed on. The results are written to a .csv for the
# RemovalHistory.py program.

import csv
from datetime import date, timedelta
import matplotlib.pyplot as plt
from copy import deepcopy

import FlightHours

### -------------------------- Steps for running this program ---------------------------- ###

## - Requires Access to NAVAIR DECKPLATE Query Studio
## - Run the DP-0025 report* with the following parameters:
##   - TEC: AYNE
##   - Part Number: %
##   - Serial Number: %
##   - Start Date: March 1, 2000 (known first 27502X entry is March 12, 2000...
##   - End Date: Do not use today's date, can choose any date up to yesterday
##     * Recommend submitting as a job, takes several minutes
## - Save the report as a .csv
## - Open the .csv
## - Select all, filter:
## - Add a custom filter to the WUC column: "Begins with" 27502
## - Custom sort:
##   - Rmvl Inst Equip SerNo - Smallest to Largest
##   - Action Date - Oldest to Newest
##   - SerNo (this is actually BuNo) - Smallest to Largest
##   - R/I - Z to A
## - Run Models.py
## - Requires NAVFLIR Data, see FlightHours.py for report generation steps

class SernoHistory(object):
    ''' Dictionary of Sernos whose key:value pair is serno:[list of historical flight data].
    This class has methods that will map out by calendar date which BUNO the serno was
    installed on throughout its service. It will then utilize methods from the FlightHours
    class to populate data from flight records for that buno during a date range subsequently
    providing the flight data of a serno during a date range. '''
```

```python
    def __init__(self, end_date, save_filename):

        self.data = []
        self.final_v22_PCA_data = []
        self.sernos = {}
        self.beginning = date(1997, 4, 3)
        self.end = end_date
        self.day_delta = (self.end - self.beginning).days
        self.raw_data_filename = 'DP-0025_WUC_27502X.csv'
        self.save_data_filename = save_filename
        self.header = ''
        self.header_extended = ''
        self.serno_flight_data = {}
        self.sorted_data = []

        # Indicies of the DP-0025 .csv report
        self.ACTION_DATE = 0
        self.PART_NO = 1
        self.CAGE = 2
        self.SERNO = 3
        self.WUC = 4
        self.WEC_DESC = 5
        self.ORG = 6
        self.ORG_DESC = 7
        self.TEC = 8
        self.TMS = 9
        self.BUNO = 10
        self.REM_INS_FLAG = 11
        self.JCN = 12
        self.MCN = 13
        self.TYPE_MAINT = 14
        self.TRANS_CODE = 15
        self.ACT_TAKEN = 16
        self.MAL_CODE = 17
        self.COMP_DATE = 18
        self.TIME_1_CODE = 19
        self.TIME_1_CYCLE = 20
        self.TIME_2_CODE = 21
        self.TIME_2_CYCLE = 22
        self.TIME_3_CODE = 23
        self.TIME_3_CYCLE = 24

        # Indicies for flight data
        self.HRS = 25
        self.LNDGs = 26
        self.TMR_1 = 27
```

```
        self.TMR_2 = 28
        self.TMR_3 = 29
        self.TMR_4 = 30
        self.TMR_5 = 31
        self.TMR_6 = 32
        self.TMR_7 = 33
        self.NON_AUST_LNDGS = 34
        self.AUST_LNDGS = 35
        self.SHIP_LNDGS = 36

        # WUC Criteria
        self.WUC_criteria = ['275021', '275020', '27502015', '27502115']

        self.FlightHours = FlightHours.FlightHours()
        self.FlightHours.read_data()
        self.FlightHours.format_date()
        self.FlightHours.build_bunos()
        self.FlightHours.fill_flthrs()
        self.FlightHours.fill_landings()

        # Instantiated when called by RemovalHistory.py
        self.read_data()
        self.format_date()
        self.build_sernos()
        self.evaluate_sernos()
        self.remove_duplicates()
        self.resolve_wuc_mistmatches()
        self.deconflict_multiple_installs()
        self.resort_installs()
        self.add_flight_data()
        self.write_data()


    def read_data(self):
        ''' Reads in the DP-0025 report .csv file generated from Deckplate.
        The DP-0025 report has a header for features, extract that. '''

        with open(self.raw_data_filename, 'r') as file:
            reader = csv.reader(file)
            temp_data = []
            for row in reader:
                temp_data.append(row)

            # save the header row and pop it off
            self.header = temp_data[0]
            self.header_extended = temp_data[0]
```

```
        # add the flight data column headers for later
        self.header_extended.extend(['total_flthrs', 'total_lndgs', 'total_TMR_1_hrs',
                                     'total_TMR_2_hrs', 'total_TMR_3_hrs',
                                     'total_TMR_4_hrs', 'total_TMR_5_hrs',
                                     'total_TMR_6_hrs', 'total_TMR_7_hrs',
                                     'total_NON_AUST_lndgs', 'total_AUST_lndgs',
                                     'total_SHIP_lndgs'])

        # save the raw data minus the header
        temp_data.pop(0)
        for row in temp_data:
            row[self.COMP_DATE] = ""
            self.data.append(row)

    def format_date(self):
        ''' Converts a maintenance action date string from .csv into datetime data type.
        Assumes .csv date is in MM/DD/YYYY and needs to be [YYYY, MM, DD]'''

        for row in range(0,len(self.data)):
            # Action Taken Date
            mdy = self.data[row][self.ACTION_DATE].split("/")
            if len(mdy) == 3:
                self.data[row][self.ACTION_DATE] = date(int(mdy[2]), int(mdy[0]), int(mdy[1]))
            else:
                mdy = self.data[row][self.ACTION_DATE].split("-")
                if len(mdy) == 3:
                    self.data[row][self.ACTION_DATE] = date(int(mdy[0]), int(mdy[1]),
                                                            int(mdy[2]))
            # Action Completion Date
            mdy = self.data[row][self.COMP_DATE].split("/")
            if len(mdy) == 3:
                self.data[row][self.COMP_DATE] = date(int(mdy[2]), int(mdy[0]), int(mdy[1]))
            else:
                mdy = self.data[row][self.COMP_DATE].split("-")
                if len(mdy) == 3:
                    self.data[row][self.COMP_DATE] = date(int(mdy[0]), int(mdy[1]),
                                                          int(mdy[2]))

    def write_data(self):
        ''' Saves the resultant removal entries to a csv file.
        Omits installation entries as the removals are assumed
        to be the failure events. '''

        with open(self.save_data_filename, 'w', newline="") as file:

            writer = csv.writer(file)
            writer.writerow(self.header_extended)
```

```
            for key in self.sernos.keys():
                entries = list(filter(lambda x:(x[self.REM_INS_FLAG]=="R"),self.sernos[key]))
                writer.writerows(entries)

    def sort_data(self, history):
        ''' Sorts an input list of historical R/I records by their action date value.
        Achieves the same custom sort applied in the .csv in the program steps above,
        except we want Intalls before Removals which was resolved in the imputations. '''
        func1 = lambda x: (x[self.SERNO])
        func2 = lambda x: (x[self.ACTION_DATE])
        func3 = lambda x: (x[self.BUNO])
        func4 = lambda x: (x[self.REM_INS_FLAG])

        sorted(sorted(sorted(sorted(history,key=func4),key=func3),key=func2),key=func1)

        return history

    def build_sernos(self):
        ''' Generates master dictionary of sernos:[R/I actions]. If the SerNo is not yet a key
        in the dict, it is added and its value is the corresponding row of data i.e. the
        oldest action completed from the data. Subsequent rows with that same SerNo will
        have that data appended to the value list. '''

        for row in range(0,len(self.data)):

            # Ensures only entries whose WUC begins with the WUC_criteria
            if str(self.data[row][self.WUC]) in self.WUC_criteria:

                key = str(self.data[row][self.SERNO])
                if key not in self.sernos:
                    self.sernos[key] = [self.data[row]]
                else:
                    self.sernos[key].extend([self.data[row]])

    def get_halfway_date(self, first_date, last_date):
        ''' In the event there is a missing entry, provides the date halfway
        between the two entries before and after the missing entry. '''

        day_delta = (last_date - first_date).days

        return first_date + timedelta(day_delta//2)

    def sort_serno_history(self, serno_history):
        ''' Applies the logic to the serno:[history] dict key:value pairs that sorts
        the R/I actions in the correct chronological sequence. We need every removal to
        be preceeded by an installation. This rearrangement sets the conditions to begin
        recording the ranges of dates for which a SerNo was installed on a BUNO. '''
```

```
        new_history = []
        entry = 0
        while entry < len(serno_history):

            # Get records
            current_record = deepcopy(serno_history[entry])
            previous_record = None
            next_record = None

            # If this isnt the first entry, get the previous record
            if entry != 0:
                previous_record = deepcopy(serno_history[entry-1])

            # If this isnt the last entry, get the next record
            if entry < len(serno_history) - 1:
                next_record = deepcopy(serno_history[entry+1])

            # Evaluating the first chronological entry
            if entry == 0:

                # First Entry is an Install
                if current_record[self.REM_INS_FLAG] == "I":

                    new_history.append(current_record)

                # First Entry is Removal, install is missing
                else:
                    ''' Assumed component has been installed from aircraft's delivery.'''
                    imputed_record = deepcopy(current_record)
                    imputed_record[self.ACTION_DATE] = self.beginning
                    imputed_record[self.REM_INS_FLAG] = "I"
                    new_history.append(imputed_record)
                    new_history.append(current_record)

            # Middle/Last entry
            else:

                # Entry is Install, Next is also Install, missing removal, impute
                if current_record[self.REM_INS_FLAG] == "I" and \
                   previous_record[self.REM_INS_FLAG] == "I":
                    ''' This is a point of assumption. The serno was installed on two
                    consecutive entries but never removed. We can't assume it was removed the
                    same day of the first installation, and we can't assume it was immediately
                    removed and installed on the second installation date. There is likely
                    maintenance required on the component but we aren't sure. Instead, assume
                    it was removed halfway between the two installation entries. We also can't
```

```
                assume the reason for removal, leave it blank but use the last buno it was
                installed on. '''

                imputed_record = deepcopy(previous_record)
                imputed_record[self.REM_INS_FLAG] = "R"
                #imputed_record[self.MAL_CODE] = "missing"
                imputed_date = self.get_halfway_date(previous_record[self.ACTION_DATE], \
                                                     current_record[self.ACTION_DATE])
                imputed_record[self.ACTION_DATE] = imputed_date
                new_history.append(imputed_record)

            # Entry is Removal, Next is also Removal, missing install, impute
            elif current_record[self.REM_INS_FLAG] == "R" and \
                 previous_record[self.REM_INS_FLAG] == "R":
                ''' Same assumption, use the halfway date to impute the missing
                installation. However, we are now missing the install to the next buno not
                the current'''

                imputed_record = deepcopy(current_record)
                imputed_record[self.REM_INS_FLAG] = "I"
                imputed_date = self.get_halfway_date(previous_record[self.ACTION_DATE], \
                                                     current_record[self.ACTION_DATE])
                imputed_record[self.ACTION_DATE] = imputed_date
                new_history.append(imputed_record)

            # Entry is Removal and previous was install
            elif current_record[self.REM_INS_FLAG] == "R" and \
                 previous_record[self.REM_INS_FLAG] == "I":

                # Not the same buno
                if current_record[self.BUNO] != previous_record[self.BUNO]:
                    ''' Missing two entries, remove from current entry buno and install
                    for next buno. Assume removed and installed halfway between.
                    Don't assume mal code. '''

                    imputed_record = deepcopy(previous_record)
                    imputed_record[self.REM_INS_FLAG] = "R"
                    #imputed_record[self.MAL_CODE] = "missing"
                    imputed_date = \
                        self.get_halfway_date(previous_record[self.ACTION_DATE], \
                                              current_record[self.ACTION_DATE])
                    imputed_record[self.ACTION_DATE] = imputed_date
                    new_history.append(imputed_record)

                    imputed_record = deepcopy(current_record)
                    imputed_record[self.REM_INS_FLAG] = "I"
                    imputed_record[self.ACTION_DATE] = imputed_date
```

```
                        new_history.append(imputed_record)

                new_history.append(current_record)

            entry += 1

        return new_history

    def add_flight_data(self):
        ''' Adds flight data from a buno to the serno for the dates it was installed. '''

        for serno in self.sernos.keys():

            total_hrs = 0
            total_lndgs = 0
            total_TMR_1_hrs = 0
            total_TMR_2_hrs = 0
            total_TMR_3_hrs = 0
            total_TMR_4_hrs = 0
            total_TMR_5_hrs = 0
            total_TMR_6_hrs = 0
            total_TMR_7_hrs = 0
            total_NON_AUST_lndgs = 0
            total_AUST_lndgs = 0
            total_SHIP_lndgs = 0

            data = [str(total_hrs), str(total_lndgs), str(total_TMR_1_hrs),
                    str(total_TMR_2_hrs), str(total_TMR_3_hrs), str(total_TMR_4_hrs),
                    str(total_TMR_5_hrs), str(total_TMR_6_hrs), str(total_TMR_7_hrs),
                    str(total_NON_AUST_lndgs), str(total_AUST_lndgs), str(total_SHIP_lndgs)]

            entry = 0
            while entry < len(self.sernos[serno]):

                if entry == 0:
                    self.sernos[serno][entry].extend(data)

                # Flight data is the same as its last removed, since it wasn't installed
                elif self.sernos[serno][entry][self.REM_INS_FLAG] == "I":
                    self.sernos[serno][entry].extend(data)

                elif self.sernos[serno][entry-1][self.REM_INS_FLAG] == "I" and \
                     self.sernos[serno][entry][self.REM_INS_FLAG] == "R":
                    if self.sernos[serno][entry-1][self.BUNO] == \
                       self.sernos[serno][entry][self.BUNO]:

                        install_date = self.sernos[serno][entry-1][self.ACTION_DATE]
```

```
                        removal_date = self.sernos[serno][entry][self.ACTION_DATE]
                        buno = self.sernos[serno][entry][self.BUNO]

                        [flthrs, lndgs, TMR_1_hrs, TMR_2_hrs, TMR_3_hrs,
                         TMR_4_hrs, TMR_5_hrs, TMR_6_hrs, TMR_7_hrs,
                         NON_AUST_lndgs, AUST_lndgs, SHIP_lndgs]  = \
                            self.FlightHours.buno_query_all_data(buno,
                                                                 install_date, removal_date)

                        total_hrs += flthrs
                        total_lndgs += lndgs
                        total_TMR_1_hrs += TMR_1_hrs
                        total_TMR_2_hrs += TMR_2_hrs
                        total_TMR_3_hrs += TMR_3_hrs
                        total_TMR_4_hrs += TMR_4_hrs
                        total_TMR_5_hrs += TMR_5_hrs
                        total_TMR_6_hrs += TMR_6_hrs
                        total_TMR_7_hrs += TMR_7_hrs
                        total_NON_AUST_lndgs += NON_AUST_lndgs
                        total_AUST_lndgs += AUST_lndgs
                        total_SHIP_lndgs += SHIP_lndgs

                        data = [str(total_hrs), str(total_lndgs), str(total_TMR_1_hrs),
                                str(total_TMR_2_hrs), str(total_TMR_3_hrs),
                                str(total_TMR_4_hrs), str(total_TMR_5_hrs),
                                str(total_TMR_6_hrs), str(total_TMR_7_hrs),
                                str(total_NON_AUST_lndgs), str(total_AUST_lndgs),
                                str(total_SHIP_lndgs)]

                        self.sernos[serno][entry].extend(data)

                entry += 1

    def evaluate_sernos(self):
        ''' For each serno in the dictionary, perform the evaluation of R/I history. Prints an
        error if one occurs. Writes the clean data to a new .csv by calling write_data(). '''

        try:

            for serno in self.sernos.keys():
                self.resolve_serno(serno)

        except Exception:
            print("Error evaluating serno " + str(serno) + "\n")

    def resolve_serno(self, serno):
        ''' Retrieves all R/I records for the serno provided.
```

```
        Calls sort_serno_history to sort and impute missing entries.
        Prints an error if one occurs. '''

        try:
            temp_results = []
            temp_serno_history = deepcopy(self.sernos[serno])

            for row in temp_serno_history:
                temp_results.append(row)

            # Properly sort/impute R/I entries and replace the old history with the new
            new_history = self.sort_serno_history(self.get_history(serno))
            self.sernos[serno] = new_history

        except Exception as e:
            print("Error resolving serno " + str(serno) + " " + e + "\n")

    def get_history(self, serno):
        ''' Generates the list of R/I actions based on a serno. '''

        history_list = deepcopy(self.sernos[serno])

        return history_list

    def query_buno(self, buno):
        ''' Returns all serno R/I entries for a single BUNO.
        Used for deconflicting imputations that may have multiple
        sernos installed on the same WUC. '''

        try:
            # Iterate through all serno/part_num key values in the dict
            temp_results = []

            for key in self.sernos.keys():

                # make a copy of the dict key
                temp_key = deepcopy(key)
                temp_value = deepcopy(self.sernos[temp_key])

                for row in temp_value:
                    if row[self.BUNO] == buno:
                        temp_results.append(row)

            # sort the results by the custom criteria
            sorted_results = self.sort_data(temp_results)
            results = deepcopy(sorted_results)
```

```
            return results

        except Exception:
            print("Error retrieving R/I history when deconflicting buno ", str(buno))


    def remove_duplicates(self):
        ''' Once all missing R/I data has been imputed, evaluates each I/R pair of entries to
        see if the serno has more than one of the same pair and remove the duplicates. '''

        try:

            # Returns all R/I entries for a buno
            for serno in self.sernos.keys():
                temp_history = self.get_history(serno)
                new_history = []

                # While there are still entries in the temp list
                while len(temp_history) > 0:

                    # If there is only one entry left
                    if len(temp_history) == 1:
                        new_history.append(temp_history.pop(0))

                    # Pop the install/removal pair from the temp list and add to new list
                    else:
                        install = temp_history.pop(0)
                        removal = temp_history.pop(0)
                        new_history.append(install)
                        new_history.append(removal)

                        current_pair = [install, removal]

                        # If there is only one entry remaining
                        if len(temp_history) == 1:
                            new_history.append(temp_history.pop(0))

                        else:
                            index = 0
                            while index < len(temp_history)-1:
                                next_pair = [temp_history[index], temp_history[index+1]]

                                if current_pair == next_pair:
                                    temp_history.pop(index)
                                    temp_history.pop(index)
                                else:
                                    index += 2
```

```
                self.sernos[serno] = new_history

        except Exception:
            print("Error removing duplicates for serno " + str(serno))


    def deconflict_multiple_installs(self):
        ''' Once all missing R/I data has been imputed, deconflicts multiple SerNos that were
        assumed to have been installed during production that occupy and aircrafts left or
        right hand WUC. This is only operational if the component being evaluated has 2
        positions on the aircraft. '''

        try:

            # Returns all serno R/I entries for a buno
            for buno in self.FlightHours.bunos.keys():
                buno_history = self.query_buno(buno)

                # Split the removals and installs into seperate lists
                right_installs = []
                right_removals = []
                left_installs = []
                left_removals = []

                for entry in buno_history:
                    # Right
                    if entry[self.WUC][:6] == "275020" and entry[self.REM_INS_FLAG] == "I":
                        right_installs.append(entry)
                    elif entry[self.WUC][:6] == "275020" and entry[self.REM_INS_FLAG] == "R":
                        right_removals.append(entry)
                    # Left
                    elif entry[self.WUC][:6] == "275021" and entry[self.REM_INS_FLAG] == "I":
                        left_installs.append(entry)
                    elif entry[self.WUC][:6] == "275021" and entry[self.REM_INS_FLAG] == "R":
                        left_removals.append(entry)

                # sort each list by action date
                right_installs.sort(key=lambda x:x[self.ACTION_DATE])
                right_removals.sort(key=lambda x:x[self.ACTION_DATE])
                left_installs.sort(key=lambda x:x[self.ACTION_DATE])
                left_removals.sort(key=lambda x:x[self.ACTION_DATE])

                # Right
                i = 0
                prev_rem_date = None
                # Iterate over each removal
```

```
            while i < len(right_removals):
                j = 0
                # Iterate over each install
                while j < len(right_installs):
                    # If they are the same Serno
                    if right_removals[i][self.SERNO] == right_installs[j][self.SERNO]:
                        # And they are the same Buno
                        if right_removals[i][self.BUNO] == right_installs[j][self.BUNO]:
                            # And the install was before or on the removal date
                            if right_removals[i][self.ACTION_DATE] >= \
                               right_installs[j][self.ACTION_DATE]:
                                # Check if there is a previous removal yet
                                # if the installation date was before the last removal,
                                # change the install date
                                if prev_rem_date != None and \
                                   right_installs[j][self.ACTION_DATE] < prev_rem_date:
                                    for index in range(len(
                                        self.sernos[right_installs[j][self.SERNO]])):
                                        # Find the matching entry in the master serno
                                        # history dictionary
                                        if self.sernos[right_installs[j] \
                                                       [self.SERNO]][index] == \
                                           right_installs[j]:
                                            # Change the date officially
                                            self.sernos[right_installs[j] \
                                                        [self.SERNO]][index] \
                                                [self.ACTION_DATE] = prev_rem_date
                                            break
                                # An install and removal have been successfully evaluated,
                                # pop them from the lists
                                prev_rem_date = right_removals[i][self.ACTION_DATE]
                                right_installs.pop(j)
                                break
                    j += 1
                i += 1

            # Left
            i = 0
            prev_rem_date = None
            # Iterate over each removal
            while i < len(left_removals):
                j = 0
                # Iterate over each install
                while j < len(left_installs):
                    # If they are the same Serno
                    if left_removals[i][self.SERNO] == left_installs[j][self.SERNO]:
                        # And they are the same Buno
```

```
                            if left_removals[i][self.BUNO] == left_installs[j][self.BUNO]:
                                # And the install was before or on the removal date
                                if left_removals[i][self.ACTION_DATE] >= \
                                   left_installs[j][self.ACTION_DATE]:
                                    # Check if there is a previous removal yet
                                    # if the installation date was before the last removal,
                                    # change the install date
                                    if prev_rem_date != None and \
                                       left_installs[j][self.ACTION_DATE] < prev_rem_date:
                                        for index in range(len(
                                            self.sernos[left_installs[j][self.SERNO]])):
                                            # Find the matching entry in the master serno
                                            # history dictionary
                                            if self.sernos[left_installs[j] \
                                                           [self.SERNO]][index] == \
                                               left_installs[j]:
                                                # Change the date officially
                                                self.sernos[left_installs[j] \
                                                            [self.SERNO]][index] \
                                                    [self.ACTION_DATE] = prev_rem_date
                                                break
                                    # An install and removal have been successfully evaluated,
                                    # pop them from the lists
                                    prev_rem_date = left_removals[i][self.ACTION_DATE]
                                    left_installs.pop(j)
                                    break
                        j += 1
                    i += 1

        except Exception:
            print("Error deconflicting multiple installs for buno ", str(buno))

    def resolve_wuc_mistmatches(self):
        ''' Before any sorting was performed by this program, it was identified that the
        DP-0025 data reflects accompanying installation and removal MAFs for the same serno
        sometimes have erroneous WUCs. For example, a serno is installed on the left side but
        removed from the right. This function replaces the installation WUC if they do not
        match. This is another assumption that may effect flight data. '''

        for serno in self.sernos.keys():
            temp_history = self.get_history(serno)

            i = 0
            while i < len(temp_history)-1:
                if temp_history[i][self.REM_INS_FLAG] == "I" and \
                   temp_history[i+1][self.REM_INS_FLAG] == "R" and \
                   temp_history[i][self.SERNO] == temp_history[i+1][self.SERNO]:
```

```
                if temp_history[i][self.WUC] != temp_history[i+1][self.WUC]:
                    temp_history[i][self.WUC] = temp_history[i+1][self.WUC]
            i += 1

        self.sernos[serno] = temp_history

    def resort_installs(self):
        ''' Once duplicates are deconflicted, some install entry dates changed.
        This function re-sorts the serno data. '''

        for serno in self.sernos.keys():
            temp_history = self.get_history(serno)

            # Split the removals and installs into seperate lists
            installs = []
            removals = []
            new_history = []

            for entry in temp_history:
                if entry[self.REM_INS_FLAG] == "I":
                    installs.append(entry)
                elif entry[self.REM_INS_FLAG] == "R":
                    removals.append(entry)

            # sort each list by action date
            installs.sort(key=lambda x:x[self.ACTION_DATE])
            removals.sort(key=lambda x:x[self.ACTION_DATE])

            i = 0
            # Iterate over each removal
            while i < len(removals):
                j = 0
                # Iterate over each install
                while j < len(installs):
                    # If they are the same Serno
                    if removals[i][self.SERNO] == installs[j][self.SERNO]:
                        # And they are the same Buno
                        if removals[i][self.BUNO] == installs[j][self.BUNO]:
                            # And the install was before or on the removal date
                            if removals[i][self.ACTION_DATE] >= installs[j][self.ACTION_DATE]:
                                # And the same WUC
                                if removals[i][self.WUC] == installs[j][self.WUC]:
                                    new_history.append(installs[j])
                                    new_history.append(removals[i])
                                    installs.pop(j)
                                    removals.pop(i)
```

```
                                i = 0
                                break
                        j += 1

            self.sernos[serno] = new_history
```

```python
# ------------------------------------ FlightHours.py --------------------------------------#
# Captain William Frazier
# NPS Thesis research
# This program takes Naval Flight Record (NAVFLIR) data from the Buno
#   Flight Hours Between Dates report from NAVAIR Deckplate, builds a
#   Python dictionary for each Buno and stores cumulative flight data
#   by calendar date. The results is for any Buno and Date, the
#   cumulative flight data is available. Therefore, the data flown
#   between two dates a serialized component was installed on the
#   aircraft is available. This class is imported by the SernoHistory.py
#   program to generate cumulative flight data at the time of a
#   component's removal (failure).

import csv
from datetime import date, timedelta
import matplotlib.pyplot as plt

### ------------------------- Steps for running this program ---------------------------- ###

## - Requires Access to NAVAIR DECKPLATE Query Studio
## - The Buno Flight Hours Between Dates report is a custom report
## - In the Query Studio access the Flight Summary environment
## - Add the following features in this orders (left to right)
##   - TEC: AYNE
##   - Buno
##   - TMR Hours
##   - TMR Code
##   - TMR Desc
##   - Total Lndgs
##   - Type Lndg Code
##   - Type Lndg Desc
##   - Depart Date
## Sort the report by Depart Date: Ascending order
##     * Recommend submitting as a job, takes several minutes
## - Save the report as a .csv
## Called by SernoHistory.py and Models.py


class FlightHours(object):
    ''' Dictionary of BUNOs:calendar{date:flight data} '''

    def __init__(self):

        self.data = []
        self.bunos = {}
        self.beginning = date(1997, 4, 3)
        self.end = date.today()
```

```
        self.day_delta = (self.end - self.beginning).days
        self.input_filename = 'Buno Flight Hours Between Dates_22_Aug_22.csv'
        self.output_filename = 'Buno Flight Data Between Dates_22_Aug_22.csv'
        self.temp_output = []

        # Indicies
        self.TEC = 0
        self.BUNO = 1
        self.HRS = 2
        self.TMR_CODE = 3
        self.TMR_DESC = 4
        self.TTL_LNDG = 5
        self.LNDG_TYPE = 6
        self.LNDG_DESC = 7
        self.DATE = 8

        self.NON_AUST_LNDG_CODES = ['0','5','8','9','E','H','J','K','Y','Z']
        self.AUST_LNDG_CODES = ['6','7','F','G','L','M','P','Q','T','W']
        self.SHIP_LNG_CODES = ['1','2','3','4','A','B','C','D','N']

        self.BUNO_HRS = 0
        self.BUNO_LNDG = 1
        self.TMR_1 = 2
        self.TMR_2 = 3
        self.TMR_3 = 4
        self.TMR_4 = 5
        self.TMR_5 = 6
        self.TMR_6 = 7
        self.TMR_7 = 8
        self.NON_AUST_LNDGS = 9
        self.AUST_LNDGS = 10
        self.SHIP_LNDGS = 11

    def read_data(self):
        ''' Reads in NAVFLIR data from the DECKPLATE saved csv file. '''
        with open(self.input_filename, 'r') as file:
            reader = csv.reader(file)

            for row in reader:
                self.data.append(row)

    def format_date(self):
        ''' Converts flight hour date string from .csv into datetime data type.
        Assumes .csv date is in MM/DD/YYYY and needs to be [YYYY, MM, DD]'''
        for row in range(1,len(self.data)):
            mdy = self.data[row][self.DATE].split("/")
            self.data[row][self.DATE] = date(int(mdy[2]), int(mdy[0]), int(mdy[1]))
```

```
    def build_calendar(self):
        ''' Fills the dictionary with the key:value pair of date:[0]
        The date range is from self.beginning to date.today()
        '''
        new_calendar = {}
        # Each day has a list of the flight data:
        # [Total flthrs, Total Landings, TMR_1 flthrs, TMR_2 flthrs, TMR_3 flthrs,
        #  TMR_4 flthrs, TMR_5 flthrs, TMR_6 flthrs, TMR_7 flthrs,
        #  NON_AUST_LNDGS, AUST_LNDGS, SHIP_LNDGS]

        for day in range(self.day_delta):
            new_calendar[self.beginning + timedelta(day)] = [0,0,0,0,0,0,0,0,0,0,0,0]

        return new_calendar

    def build_bunos(self):
        ''' Generates master dictionary of BUNOs:calendar{} which also
        calls build_calendar to generate the value which is a dictionary as well. '''
        for row in range(1,len(self.data)):
            if self.data[row][self.BUNO] not in self.bunos:
                self.bunos[self.data[row][self.BUNO]] = self.build_calendar()

    def fill_flthrs(self):
        ''' Fills the BUNO's calendar with flighthours using flight summary .csv data
        The date column is already in datetime format which matches dictionary key'''

        # Step 1: Fill flight record hours into the dates flown
        for row in range(1,len(self.data)):

            TMR_1_hrs = 0
            TMR_2_hrs = 0
            TMR_3_hrs = 0
            TMR_4_hrs = 0
            TMR_5_hrs = 0
            TMR_6_hrs = 0
            TMR_7_hrs = 0

            # Column values from csv
            data = self.data[row]
            # Key for self.bunos dictionary
            current_buno = self.data[row][self.BUNO]
            # Key for buno's calendar dictionary
            current_date = self.data[row][self.DATE]
            # Value to increment for buno's calendar dictionary
            flt_hrs_flown = self.data[row][self.HRS]
            tmr_code = self.data[row][self.TMR_CODE]
```

```
            if tmr_code != '':
                tmr_code = str(tmr_code)[0]

                if tmr_code == "1":
                    TMR_1_hrs = flt_hrs_flown
                elif tmr_code == "2":
                    TMR_2_hrs = flt_hrs_flown
                elif tmr_code == "3":
                    TMR_3_hrs = flt_hrs_flown
                elif tmr_code == "4":
                    TMR_4_hrs = flt_hrs_flown
                elif tmr_code == "5":
                    TMR_5_hrs = flt_hrs_flown
                elif tmr_code == "6":
                    TMR_6_hrs = flt_hrs_flown
                elif tmr_code == "7":
                    TMR_7_hrs = flt_hrs_flown

            if flt_hrs_flown != '':
                current_hrs = self.bunos[current_buno][current_date][self.BUNO_HRS]
                current_TMR_1_hrs = self.bunos[current_buno][current_date][self.TMR_1]
                current_TMR_2_hrs = self.bunos[current_buno][current_date][self.TMR_2]
                current_TMR_3_hrs = self.bunos[current_buno][current_date][self.TMR_3]
                current_TMR_4_hrs = self.bunos[current_buno][current_date][self.TMR_4]
                current_TMR_5_hrs = self.bunos[current_buno][current_date][self.TMR_5]
                current_TMR_6_hrs = self.bunos[current_buno][current_date][self.TMR_6]
                current_TMR_7_hrs = self.bunos[current_buno][current_date][self.TMR_7]

                # Increment date's flthrs for that BUNO
                # Allows for multiple flights documented for one day
                self.bunos[current_buno][current_date][self.BUNO_HRS] = \
                    round(float(flt_hrs_flown) + float(current_hrs), 2)
                self.bunos[current_buno][current_date][self.TMR_1] = \
                    round(float(TMR_1_hrs) + float(current_TMR_1_hrs), 2)
                self.bunos[current_buno][current_date][self.TMR_2] = \
                    round(float(TMR_2_hrs) + float(current_TMR_2_hrs), 2)
                self.bunos[current_buno][current_date][self.TMR_3] = \
                    round(float(TMR_3_hrs) + float(current_TMR_3_hrs), 2)
                self.bunos[current_buno][current_date][self.TMR_4] = \
                    round(float(TMR_4_hrs) + float(current_TMR_4_hrs), 2)
                self.bunos[current_buno][current_date][self.TMR_5] = \
                    round(float(TMR_5_hrs) + float(current_TMR_5_hrs), 2)
                self.bunos[current_buno][current_date][self.TMR_6] = \
                    round(float(TMR_6_hrs) + float(current_TMR_6_hrs), 2)
                self.bunos[current_buno][current_date][self.TMR_7] = \
                    round(float(TMR_7_hrs) + float(current_TMR_7_hrs), 2)
```

```
        # Step 2: Iterate through each date in the calendar and sum current with previous
        for current_buno in self.bunos.keys():
            for day in range(1, self.day_delta):
                current_day = self.beginning + timedelta(day)
                prev_day = current_day - timedelta(1)

                current_day_hrs = self.bunos[current_buno][current_day][self.BUNO_HRS]
                current_day_TMR_1_hrs = self.bunos[current_buno][current_day][self.TMR_1]
                current_day_TMR_2_hrs = self.bunos[current_buno][current_day][self.TMR_2]
                current_day_TMR_3_hrs = self.bunos[current_buno][current_day][self.TMR_3]
                current_day_TMR_4_hrs = self.bunos[current_buno][current_day][self.TMR_4]
                current_day_TMR_5_hrs = self.bunos[current_buno][current_day][self.TMR_5]
                current_day_TMR_6_hrs = self.bunos[current_buno][current_day][self.TMR_6]
                current_day_TMR_7_hrs = self.bunos[current_buno][current_day][self.TMR_7]

                prev_day_hrs = self.bunos[current_buno][prev_day][self.BUNO_HRS]
                prev_day_TMR_1_hrs = self.bunos[current_buno][prev_day][self.TMR_1]
                prev_day_TMR_2_hrs = self.bunos[current_buno][prev_day][self.TMR_2]
                prev_day_TMR_3_hrs = self.bunos[current_buno][prev_day][self.TMR_3]
                prev_day_TMR_4_hrs = self.bunos[current_buno][prev_day][self.TMR_4]
                prev_day_TMR_5_hrs = self.bunos[current_buno][prev_day][self.TMR_5]
                prev_day_TMR_6_hrs = self.bunos[current_buno][prev_day][self.TMR_6]
                prev_day_TMR_7_hrs = self.bunos[current_buno][prev_day][self.TMR_7]

                self.bunos[current_buno][current_day][self.BUNO_HRS] = \
                    round(prev_day_hrs + current_day_hrs,2)
                self.bunos[current_buno][current_day][self.TMR_1] = \
                    round(prev_day_TMR_1_hrs + current_day_TMR_1_hrs,2)
                self.bunos[current_buno][current_day][self.TMR_2] = \
                    round(prev_day_TMR_2_hrs + current_day_TMR_2_hrs,2)
                self.bunos[current_buno][current_day][self.TMR_3] = \
                    round(prev_day_TMR_3_hrs + current_day_TMR_3_hrs,2)
                self.bunos[current_buno][current_day][self.TMR_4] = \
                    round(prev_day_TMR_4_hrs + current_day_TMR_4_hrs,2)
                self.bunos[current_buno][current_day][self.TMR_5] = \
                    round(prev_day_TMR_5_hrs + current_day_TMR_5_hrs,2)
                self.bunos[current_buno][current_day][self.TMR_6] = \
                    round(prev_day_TMR_6_hrs + current_day_TMR_6_hrs,2)
                self.bunos[current_buno][current_day][self.TMR_7] = \
                    round(prev_day_TMR_7_hrs + current_day_TMR_7_hrs,2)

    def fill_landings(self):
        ''' Fills the BUNO's calendar with number of landings and type. '''
        # Step 1: Fill number of landings into the dates flown
        for row in range(1,len(self.data)):
```

```
            non_aust_lndgs = 0
            aust_lndgs = 0
            ship_lndgs = 0

            # Column values from csv
            # Key for self.bunos dictionary
            current_buno = self.data[row][self.BUNO]
            # Key for buno's calendar dictionary
            current_date = self.data[row][self.DATE]
            # Value to increment for buno's calendar dictionary
            num_landings = self.data[row][self.TTL_LNDG]
            lndg_code = self.data[row][self.LNDG_TYPE]

            if lndg_code != '':
                lndg_code = str(lndg_code)

                if lndg_code in self.NON_AUST_LNDG_CODES:
                    non_aust_lndgs = num_landings
                elif lndg_code in self.AUST_LNDG_CODES:
                    aust_lndgs = num_landings
                elif lndg_code in self.SHIP_LNG_CODES:
                    ship_lndgs = num_landings

            if num_landings != '':
                current_landings = self.bunos[current_buno][current_date][self.BUNO_LNDG]
                current_non_aust_lndgs = \
                    self.bunos[current_buno][current_date][self.NON_AUST_LNDGS]
                current_aust_lndgs = self.bunos[current_buno][current_date][self.AUST_LNDGS]
                current_ship_lndgs = self.bunos[current_buno][current_date][self.SHIP_LNDGS]

                # Increment date's flthrs for that BUNO
                # Allows for multiple flights documented for one day
                self.bunos[current_buno][current_date][self.BUNO_LNDG] = \
                    round(float(num_landings) + float(current_landings), 2)
                self.bunos[current_buno][current_date][self.NON_AUST_LNDGS] = \
                    round(float(non_aust_lndgs) + float(current_non_aust_lndgs), 2)
                self.bunos[current_buno][current_date][self.AUST_LNDGS] = \
                    round(float(aust_lndgs) + float(current_aust_lndgs), 2)
                self.bunos[current_buno][current_date][self.SHIP_LNDGS] = \
                    round(float(ship_lndgs) + float(current_ship_lndgs), 2)

        # Step 2: Iterate through each date in the calendar and sum current with previous
        for current_buno in self.bunos.keys():
            for day in range(1, self.day_delta):
                current_day = self.beginning + timedelta(day)
                prev_day = current_day - timedelta(1)
```

```
            current_day_lndgs = self.bunos[current_buno][current_day][self.BUNO_LNDG]
            current_day_non_aust_lndgs = \
                self.bunos[current_buno][current_day][self.NON_AUST_LNDGS]
            current_day_aust_lndgs = \
                self.bunos[current_buno][current_day][self.AUST_LNDGS]
            current_day_ship_lndgs = \
                self.bunos[current_buno][current_day][self.SHIP_LNDGS]

            prev_day_lndgs = self.bunos[current_buno][prev_day][self.BUNO_LNDG]
            prev_day_non_aust_lndgs = \
                self.bunos[current_buno][prev_day][self.NON_AUST_LNDGS]
            prev_day_aust_lndgs = self.bunos[current_buno][prev_day][self.AUST_LNDGS]
            prev_day_ship_lndgs = self.bunos[current_buno][prev_day][self.SHIP_LNDGS]

            self.bunos[current_buno][current_day][self.BUNO_LNDG] = \
                round(prev_day_lndgs + current_day_lndgs,2)
            self.bunos[current_buno][current_day][self.NON_AUST_LNDGS] = \
                round(prev_day_non_aust_lndgs + current_day_non_aust_lndgs,2)
            self.bunos[current_buno][current_day][self.AUST_LNDGS] = \
                round(prev_day_aust_lndgs + current_day_aust_lndgs,2)
            self.bunos[current_buno][current_day][self.SHIP_LNDGS] = \
                round(prev_day_ship_lndgs + current_day_ship_lndgs,2)

    def buno_query_all_data(self, buno, start_date, end_date):
        ''' Returns the tracked numbers flown by a buno between two dates.
        Called by SernoHistory.py class. '''

        if buno in self.bunos.keys():

            flthrs = round(self.bunos[buno][end_date][self.BUNO_HRS] -
                           self.bunos[buno][start_date][self.BUNO_HRS] , 2)
            lndgs = round(self.bunos[buno][end_date][self.BUNO_LNDG] -
                          self.bunos[buno][start_date][self.BUNO_LNDG] , 2)
            TMR_1_hrs = round(self.bunos[buno][end_date][self.TMR_1] -
                              self.bunos[buno][start_date][self.TMR_1] , 2)
            TMR_2_hrs = round(self.bunos[buno][end_date][self.TMR_2] -
                              self.bunos[buno][start_date][self.TMR_2] , 2)
            TMR_3_hrs = round(self.bunos[buno][end_date][self.TMR_3] -
                              self.bunos[buno][start_date][self.TMR_3] , 2)
            TMR_4_hrs = round(self.bunos[buno][end_date][self.TMR_4] -
                              self.bunos[buno][start_date][self.TMR_4] , 2)
            TMR_5_hrs = round(self.bunos[buno][end_date][self.TMR_5] -
                              self.bunos[buno][start_date][self.TMR_5] , 2)
            TMR_6_hrs = round(self.bunos[buno][end_date][self.TMR_6] -
                              self.bunos[buno][start_date][self.TMR_6] , 2)
            TMR_7_hrs = round(self.bunos[buno][end_date][self.TMR_7] -
                              self.bunos[buno][start_date][self.TMR_7] , 2)
```

```
            NON_AUST_lndgs = round(self.bunos[buno][end_date][self.NON_AUST_LNDGS] -
                                   self.bunos[buno][start_date][self.NON_AUST_LNDGS] , 2)
            AUST_lndgs = round(self.bunos[buno][end_date][self.AUST_LNDGS] -
                               self.bunos[buno][start_date][self.AUST_LNDGS] , 2)
            SHIP_lndgs = round(self.bunos[buno][end_date][self.SHIP_LNDGS] -
                               self.bunos[buno][start_date][self.SHIP_LNDGS] , 2)

        else:
            return [0,0,0,0,0,0,0,0,0,0,0,0]

        return [flthrs, lndgs, TMR_1_hrs, TMR_2_hrs, TMR_3_hrs, TMR_4_hrs, TMR_5_hrs,
                TMR_6_hrs, TMR_7_hrs, NON_AUST_lndgs, AUST_lndgs, SHIP_lndgs]
```

```
%% NASA Turbofan LSTM Net
% Captain William Frazier
% NPS Thesis research
% Predicting RUL of NASA Turbofan Dataset
% https://www.kaggle.com/c/predictive-maintenance
clear
clc
labels = ["id","cycle","op1","op2","op3","sensor1","sensor2",...
  "sensor3","sensor4","sensor5","sensor6","sensor7","sensor8",...
  "sensor9","sensor10","sensor11","sensor12","sensor13","sensor14",...
  "sensor15","sensor16","sensor17","sensor18","sensor19","sensor20",...
  "sensor21","RUL","TTF_Window"];
% Load first set of training data
% train_FD001_new.csv is the updated version of the original train_FD001.csv
% it includes two new columns at the end which were created in orange for
% RUL and a TTF window
train_FD001 =
readtable('C:\Users\willi\OneDrive\Desktop\Thesis\Kaggle\NASA\train_FD001_new.csv');
train_FD001.Properties.VariableNames = labels;
X_all_train = table2array(train_FD001(:,1:26));
Y_train = table2array(train_FD001(:,27));
% Load first set of test data
test_FD001 =
readtable('C:\Users\willi\OneDrive\Desktop\Thesis\Kaggle\NASA\test_FD001_new.csv');
test_FD001.Properties.VariableNames = labels;
X_all_test = table2array(test_FD001(:,1:26));
Y_test = table2array(test_FD001(:,27));
% Grab the number of engines in the training/test sets
num_units_train = X_all_train(length(X_all_train),1);
num_units_test = X_all_test(length(X_all_test),1);
results = table();
% Declare how many input features to keep
for numFeatures = 3:16
  % Remove constant features
  [X_train, X_labels] = removeConstants(X_all_train, labels);
  [X_test, X_labels] = removeConstants(X_all_test, labels);
  % Identify any missing data
  missing = imputeMissing(X_all_train, Y_train, X_all_test, Y_test);
  % Calculate VIF of training inputs
  [VIF, VIFTable] = getVIF(X_train(:,(3:18)), X_labels(:,(3:18)));
  % Retain only engine ID, cycle, and the numFeatures least correlated
  [X_train, X_labels, X_test] = removeCorrInputs(X_train, X_labels, ...
    X_test, VIF, numFeatures);
  % Normalize data
  [X_train, RUL_train, X_test, RUL_test] = normalize(X_train, Y_train, ...
    X_test, Y_test);
  [LSTM_input_train, LSTM_RUL_train] = createCells(num_units_train, X_train,
RUL_train);
  [LSTM_input_test, LSTM_RUL_test] = createCells(num_units_test, X_test, RUL_test);

  % Healthy State
  health_index = 150;
  LSTM_RUL_train = adjustHealth(LSTM_RUL_train, health_index);
  % Split training into train/val
  train_perc = .75;
```

```
  val_perc = .25;
  % Padding
  [LSTM_input_train, LSTM_RUL_train] = padding(LSTM_input_train, LSTM_RUL_train);
  % Hyperparameters
  numHiddenUnits = 200;
  FCnumNodes = 50;
  dropoutrate = 0.2;
  maxEpochs = 1000;
  miniBatchSize = 49;
  InitialLearnRate = 0.01;
  GradientThreshold = 1;
  % No splitting
  [net,info] = LSTM(LSTM_input_train, LSTM_RUL_train, numHiddenUnits, ...
    FCnumNodes, dropoutrate, maxEpochs, miniBatchSize, ...
    InitialLearnRate, GradientThreshold);
  % Predict
  % Apply the over-padding prevention by setting minibatchsize to 1
  Y_pred_test = predict(net, LSTM_input_test,'MiniBatchSize',1);
  results.(numFeatures - 2) = [info.TrainingRMSE(end); getError(LSTM_RUL_test,
Y_pred_test)];
end
results.Properties.Description = ["Loss by number of input features used"];
results.Properties.VariableNames =
["3","4","5","6","7","8","9","10","11","12","13","14","15","16"];
results.Properties.RowNames = ["Final Training RMSE"; "Test RMSE"];
results
%% Remove constant features
function [X_new, X_labels] = removeConstants(X, labels)
  % Sensors 1, 5, 6, 10, 16, 18, 19, Operation Setting 3 are all constant
  X_new = X(:,[1:4,7:9,12:14,16:20,22,25:26]);
  X_labels = labels([1:4,7:9,12:14,16:20,22,25:26]);
end
%% Find missing data
function missing = imputeMissing(X_all_train, Y_train, X_all_test, Y_test)
  missing = [];
  check_for_missing = isnan(X_all_train);
  [rowmissing, colmissing] = find(check_for_missing);
  check_for_missing = isnan(Y_train);
  [rowmissing, colmissing] = find(check_for_missing);
  check_for_missing = isnan(X_all_test);
  [rowmissing, colmissing] = find(check_for_missing);
  check_for_missing = isnan(Y_test);
  [rowmissing, colmissing] = find(check_for_missing);
end
%% Calculate VIF
% Looks at multicolinearity between input features
function [VIF, VIFTable] = getVIF(X_train, X_labels)
  R = corrcoef(X_train);
  rowNames = X_labels;
  VIF = diag(inv(R));
  colNames = ["VIF"];
  VIFTable = array2table(VIF,'RowNames',rowNames,'VariableNames',colNames);
end
%% Calculate PCA
%% Remove highly correlated input features
```

```
function [X_train_new, X_labels_new, X_test_new] = ...
  removeCorrInputs(X_train, X_labels, X_test, VIF, numFeatures)
% Initialize output datasets with engine and cycle columns
X_train_new = X_train(:,1:2);
X_labels_new = X_labels(1:2);
X_test_new = X_test(:,1:2);
for i = 3:2+numFeatures
  [val, pos] = min(VIF);
  VIF(pos) = [];
  X_train_new(:,i) = X_train(:,pos+2);
  X_labels_new(i) = X_labels(pos+2);
  X_labels(pos+2) = [];
  X_test_new(:,i) = X_test(:,pos+2);
end
end
%% Normalize/Standardize
% Transpose for MatLab mapminmax(); by default normalizes between -1 and 1
function [X_train, Y_train, X_test, Y_test] = normalize(X_train, Y_train, ...
  X_test, Y_test)
% Transpose for MatLab mapminmax(); by default normalizes between -1 and 1
X_train = X_train';
X_test = X_test';
% Normalize the training data and apply that mapping to the test data
% Exclude engine id and cycle
[X_train_scaled, PS] = mapminmax(X_train(3:size(X_train,1),:));
X_test_scaled = mapminmax('apply',X_test(3:size(X_test,1),:),PS);
% Rejoin with engine and cycle
X_train(3:size(X_train,1),:) = X_train_scaled;
X_test(3:size(X_train,1),:) = X_test_scaled;
% Don't need to transpose back, keep for Cell creation
% Transpose the target though, to match
Y_train = Y_train';
Y_test = Y_test';
end
%% Create LSTM Network Training Input and Output Cells
% MatLab LSTM expects a cell
% Dimensions of the cell are n_samples x 1
% Each element in the cell is a unique engine id's sequential data
function [input_cell, target_cell] = createCells(num_units, X, Y)
% Initialize the input/target cells for MatLab LSTM expectations
% Number of units is equal to the last cell's engine id
input_cell = cell(num_units,1);
target_cell = cell(num_units,1);
% Counters so we can grab all the rows of data for an engine unit and
% insert that as one element into the cell
unit_id = 1;
row_index = 1;
cell_index = 1;
first_cycle = 1;

% Index from 1 to the length of all rows in x data
while row_index <= length(X)
  % Insert the rows of data for each unique engine ID
  % If the current row's engine ID is not = unit
  % Fill from first index to i-1
```

```
  % update first = i, unit and cell index increment by 1
  if X(1,row_index) ~= unit_id
    input_cell{cell_index} = X(3:size(X,1),[first_cycle:row_index-1]);
    target_cell{cell_index} = Y(:,[first_cycle:row_index-1]);
    cell_index = cell_index + 1;
    first_cycle = row_index;
    unit_id = unit_id + 1;
  end
  % If this is the last row, there are no more engines. fill from first
  if row_index == length(X)
    input_cell{cell_index} = X(3:size(X,1),[first_cycle:row_index]);
    target_cell{cell_index} = Y(:,[first_cycle:row_index]);
  end
  row_index = row_index + 1;
end
end
%% Splitting
function [LSTM_input_train, LSTM_RUL_train, LSTM_input_val, LSTM_RUL_val] = ...
  dataSplit(X_train, Y_train, test_perc, val_perc, num_units)
  LSTM_input_train = cell(num_units*test_perc,1);
  LSTM_RUL_train = cell(num_units*test_perc,1);
  LSTM_input_val = cell(num_units*val_perc,1);
  LSTM_RUL_val = cell(num_units*val_perc,1);
  a = randperm(numel(X_train));
  for i = 1:num_units*test_perc
    LSTM_input_train(i) = X_train(a(i));
    LSTM_RUL_train(i) = Y_train(a(i));
  end
  j = i;
  for i = 1:num_units*val_perc
   LSTM_input_val(i) = X_train(a(i+j));
   LSTM_RUL_val(i) = Y_train(a(i+j));
  end
end
%% Padding
% Because each engine has variable length samples/cycles, padding can be
% really aggressive for an engine that only has 10 cycles recorded if it
% is mini-batched with one with 150 cycles. To avoid this, we are going
% to sort our sequential data in descending order by number of cycles.
% We will also be sure to set our network options to never shuffle.
function [LSTM_input, LSTM_target] = padding(X, Y)
  % map/sort for training cell
  for i=1:numel(X)
    unit_id = X{i};
    num_cycles(i) = size(unit_id,2);
  end
  [num_cycles,index] = sort(num_cycles,'descend');
  LSTM_input = X(index);
  LSTM_target = Y(index);
end
%% How far back do we look?
% We've been discussing how far back to look in historical data before
% the data is no longer useful. Set a healthy RUL threshold and anything
% above that RUL value is set equal to that threshold
% Treat any RUL over 150 as 150
```

```
function Y_train = adjustHealth(Y_train, health_index)
  for i = 1:numel(Y_train)
    Y_train{i}(Y_train{i} > health_index) = health_index;
  end
end
%% RMSE Function
function RMSE = getError(NN_target_test, Y_pred_test)
  for i = 1:numel(NN_target_test)
    YTestLast(i) = NN_target_test{i}(end);
    YPredLast(i) = Y_pred_test{i}(end);
  end
  RMSE = sqrt(mean((YPredLast - YTestLast).^2));
end
%% Function for LSTM
function [net, info] = LSTM(LSTM_input_train, LSTM_RUL_train, ...
  numHiddenUnits, FCnumNodes, dropoutrate, maxEpochs, miniBatchSize, ...
  InitialLearnRate, GradientThreshold)
numFeatures = size(LSTM_input_train{1},1);
numResponses = size(LSTM_RUL_train{1},1);
% Hyperparameters / Model Architecture
layers = [ ...
  sequenceInputLayer(numFeatures)
  lstmLayer(numHiddenUnits,'Name', 'LSTM_1', 'OutputMode','sequence')
  fullyConnectedLayer(FCnumNodes, 'Name', 'FC_1')
  dropoutLayer(dropoutrate, 'Name', 'Dropout_1')
  fullyConnectedLayer(numResponses)
  regressionLayer];
% Training Options
options = trainingOptions('adam', ...
  'MaxEpochs',maxEpochs, ...
  'MiniBatchSize',miniBatchSize, ...
  'InitialLearnRate',InitialLearnRate, ...
  'GradientThreshold',GradientThreshold, ...
  'Shuffle','never', ...
  'Plots','none', ...
  'Verbose',0);

[net,info] = trainNetwork(LSTM_input_train, LSTM_RUL_train, layers, options);
end
```

```
%% NASA Turbofan CNN
%  Captain William Frazier
%  NPS Thesis practice/research
%  Predicting RUL of NASA Turbofan Dataset
%  https://www.kaggle.com/c/predictive-maintenance
clear
clc
labels = ["id","cycle","op1","op2","op3","sensor1","sensor2",...
    "sensor3","sensor4","sensor5","sensor6","sensor7","sensor8",...
    "sensor9","sensor10","sensor11","sensor12","sensor13","sensor14",...
    "sensor15","sensor16","sensor17","sensor18","sensor19","sensor20",...
    "sensor21","RUL","TTF_Window"];
% Load first set of training data
% train_FD001_new.csv is the updated version of the original train_FD001.csv
% it includes two new columns at the end which were created in orange for
% RUL and a TTF window
train_FD001 =
readtable('C:\Users\willi\OneDrive\Desktop\Thesis\Orange\NASA\MatLab\train_FD001_new.
csv');
train_FD001.Properties.VariableNames = labels;
X_all_train = table2array(train_FD001(:,1:26));
Y_train = table2array(train_FD001(:,27));
% Load first set of test data
test_FD001 =
readtable('C:\Users\willi\OneDrive\Desktop\Thesis\Orange\NASA\MatLab\test_FD001_new.c
sv');
test_FD001.Properties.VariableNames = labels;
X_all_test = table2array(test_FD001(:,1:26));
Y_test = table2array(test_FD001(:,27));
% Grab the number of engines in the training/test sets
num_units_train = X_all_train(length(X_all_train),1);
num_units_test = X_all_test(length(X_all_test),1);
%% Remove constant features
% Sensors 1, 5, 6, 10, 16, 18, 19, Operation Setting 3 are all constant
X_train = X_all_train(:,[1:4,7:9,12:14,16:20,22,25:26]);
x_labels = labels([1:4,7:9,12:14,16:20,22,25:26]);
%% Calculate Coeff Correlation and VIF of training data
R = corrcoef(X_train);
rowNames = x_labels;
colNames = x_labels;
RTable = array2table(R,'RowNames',rowNames,'VariableNames',colNames);
VIF = diag(inv(R));
rowNames = x_labels;
colNames = ["VIF"];
VIFTable = array2table(VIF,'RowNames',rowNames,'VariableNames',colNames);
%% Remove extremely highly correlated and recalculate VIF
X_train = X_all_train(:,[3:4,7:9,12,13,17,18,20,22,25,26]);
x_labels = labels([3:4,7:9,12,13,17,18,20,22,25,26]);
R = corrcoef(X_train);
rowNames = x_labels;
colNames = x_labels;
RTable = array2table(R,'RowNames',rowNames,'VariableNames',colNames);

VIF = diag(inv(R));
rowNames = x_labels;
```

```
colNames = ["VIF"];
VIFTable = array2table(VIF,'RowNames',rowNames,'VariableNames',colNames);
% Apply the reduction to the test data as well
X_test = X_all_test(:,[3:4,7:9,12,13,17,18,20,22,25,26]);
%% Normalize/Standardize
% Transpose for MatLab mapminmax(); by default normalizes between -1 and 1
X_train = X_train';
X_test = X_test';
% Normalize the training data and apply that mapping to the test data
[X_train_scaled, PS] = mapminmax(X_train);
X_test_scaled = mapminmax('apply',X_test,PS);
X_train = X_train';
X_test = X_test';
X_train_scaled = X_train_scaled';
X_test_scaled = X_test_scaled';
%% OPTIONAL
% rejoin with cycle/engine ID for train/validation splitting
X_train_pre_split = cat(2,X_all_train(:,1:2),X_train_scaled);
X_test_pre_split = cat(2,X_all_test(:,1:2),X_test_scaled);
%% How far back do we look?
healthy = 150;
for i = 1:numel(Y_train)
    if Y_train(i) > healthy
        Y_train(i) = healthy;
    end
end
%% This takes the training data set and creates a new NN input data set
window_size = 30;
NN_input_train = [];
NN_RUL_train = [];
unit_id = 1;
row_index = 1;
cell_index = 1;
first_cycle = 1;
while row_index <= length(X_all_train) - window_size
    if row_index == 1
        NN_input_train(:,:,:,row_index) =
X_train_scaled(row_index:row_index+window_size-1,:);
        NN_RUL_train(:,row_index) = Y_train(row_index:row_index+window_size-1,:);
        row_index = row_index+1;
    end
        % Ensure there are at least window_size cycles recorded before failure
    if Y_train(row_index,1) >= window_size
        % While unit number is still the same, insert samples into final
        while X_all_train(row_index+window_size-1,1) == unit_id
            % Insert rows i to i+window_size, all columns
            NN_input_train(:,:,:,cell_index) =
X_train_scaled(row_index:row_index+window_size-1,:);
            NN_RUL_train(:,cell_index) = Y_train(row_index:row_index+window_size-
1,:);
            cell_index = cell_index + 1;
            row_index = row_index+1;
            if row_index > length(X_train_scaled) - window_size
                return
            end
```

```
        end
        % Once the unit number is changed, need to 'hop' down, change unit
        row_index = row_index + window_size-1;
        unit_id = X_all_train(row_index,1);
    % If there aren't enough, move index to the new unit
    else
        while row_index < length(X_all_train)
            if X_all_train(row_index,1) == unit_id
                row_index = row_index+1;

            else
                unit_id = X_all_train(row_index,1);
                break
            end
        end
    end
end
%% This takes the test data set and creates a new NN input data set
NN_input_test = [];
NN_RUL_test = [];
unit_id = 1;
row_index = 1;
cell_index = 1;
first_cycle = 1;
while row_index <= length(X_all_test) - window_size
    if row_index == 1
        NN_input_test(:,:,:,row_index) =
X_test_scaled(row_index:row_index+window_size-1,:);
        NN_RUL_test(:,row_index) = Y_test(row_index:row_index+window_size-1,:);
        row_index = row_index+1;
    end
    % Ensure there are at least window_size cycles recorded before failure
    if Y_test(row_index,1) >= window_size
        % While unit number is still the same, insert samples into final
        while X_all_test(row_index+window_size-1,1) == unit_id
            % Insert rows i to i+window_size, all columns
            NN_input_test(:,:,:,cell_index) =
X_test_scaled(row_index:row_index+window_size-1,:);
            NN_RUL_test(:,cell_index) = Y_test(row_index:row_index+window_size-1,:);
            cell_index = cell_index + 1;
            row_index = row_index+1;
            if row_index > length(X_test_scaled) - window_size
                return
            end
        end
        % Once the unit number is changed, need to 'hop' down, change unit
        row_index = row_index + window_size-1;
        unit_id = X_all_test(row_index,1);
    % If there aren't enough, move index to the new unit
    else
        while row_index < length(X_all_test)
            if X_all_test(row_index,1) == unit_id
                row_index = row_index+1;
            else
                unit_id = X_all_test(row_index,1);
```

```
                break
            end
        end
    end
end
%% OPTIONAL
% attempt to split 75 – 25. ensure the inputs and target are in the same cell indexes
% create an array from 1 to 100 (number of engines) psuedo randomly
% without repeating or omitting any numbers
test_perc = .75;
val_perc = .25;
a = randperm(num_units_train);
CNN_input_train = [];
CNN_RUL_train = [];
index = 1;
% for the first 75% of engine ids
for i = 1:num_units_test*test_perc
    for j = 1:length(X_train_pre_split)
        if X_train_pre_split(j,1) == a(i)
            CNN_input_train(index,:) = X_train_pre_split(j,:);
            CNN_RUL_train(index,:) = Y_train(j,:);
            index = index + 1;
        end
    end
end
k = i;
CNN_input_val = [];
CNN_RUL_val = [];
index = 1;
for i = 1:num_units_test*val_perc
    % iterate through all
    for j = 1:length(X_train_pre_split)
        if X_train_pre_split(j,1) == a(i+k)
            CNN_input_val(index,:) = X_train_pre_split(j,:);
            CNN_RUL_val(index,:) = Y_train(j,:);
            index = index + 1;
        end
    end
end
%% This takes the training data set and creates a new NN input data set (75 %)
window_size = 30;
NN_input_train = [];
NN_RUL_train = [];
unit_id = 1;
row_index = 1;
cell_index = 1;
first_cycle = 1;
while row_index <= length(CNN_input_train) - window_size
    if row_index == 1
        NN_input_train(:,:,:,row_index) =
CNN_input_train(row_index:row_index+window_size-1,3:15);
        NN_RUL_train(:,row_index) = CNN_RUL_train(row_index:row_index+window_size-
1,:);
        row_index = row_index+1;
    end
```

```
    if CNN_RUL_train(row_index,1) >= window_size
        while CNN_input_train(row_index+window_size-1,1) == unit_id
            NN_input_train(:,:,:,cell_index) =
CNN_input_train(row_index:row_index+window_size-1,3:15);
            NN_RUL_train(:,cell_index) =
CNN_RUL_train(row_index:row_index+window_size-1,:);
            cell_index = cell_index + 1;
            row_index = row_index+1;
            if row_index > length(CNN_input_train) - window_size
                return
            end
        end
        % Once the unit number is changed, need to 'hop' down, change unit
        row_index = row_index + window_size-1;
        unit_id = CNN_input_train(row_index,1);
    % If there aren't enough, move index to the new unit
    else
        while row_index < length(CNN_input_train)
            if CNN_input_train(row_index,1) == unit_id
                row_index = row_index+1;

            else
                unit_id = CNN_input_train(row_index,1);
                break
            end
        end
    end
end
%% This takes the test data set and creates a new NN validation data set (25 %)
window_size = 30;
NN_input_val = [];
NN_RUL_val = [];
unit_id = 1;
row_index = 1;
cell_index = 1;
first_cycle = 1;
while row_index <= length(CNN_input_val) - window_size
    if row_index == 1
        NN_input_val(:,:,:,row_index) =
CNN_input_val(row_index:row_index+window_size-1,3:15);
        NN_RUL_val(:,row_index) = CNN_RUL_val(row_index:row_index+window_size-1,:);
        row_index = row_index+1;
    end

    if CNN_RUL_val(row_index,1) >= window_size
        while CNN_input_val(row_index+window_size-1,1) == unit_id
            NN_input_val(:,:,:,cell_index) =
CNN_input_val(row_index:row_index+window_size-1,3:15);
            NN_RUL_val(:,cell_index) = CNN_RUL_val(row_index:row_index+window_size-
1,:);
            cell_index = cell_index + 1;
            row_index = row_index+1;
            if row_index > length(CNN_input_val) - window_size
                return
            end
```

```
        end
        % Once the unit number is changed, need to 'hop' down, change unit
        row_index = row_index + window_size-1;
        unit_id = CNN_input_val(row_index,1);
    % If there aren't enough, move index to the new unit
    else
        while row_index < length(CNN_input_val)
            if CNN_input_val(row_index,1) == unit_id
                row_index = row_index+1;

            else
                unit_id = CNN_input_val(row_index,1);
                break
            end
        end
    end
end
%% Neural Net Model Building and Training Function
% Architecture Hyperparameters/Training Hyperparameters
dropoutrate = 0.50;
maxEpochs = 600;
miniBatchSize = 512;
numFeatures = [30 13 1];
numHiddenNodes = 100;
numResponses = 30;
InitialLearnRate = 0.01;
GradientThreshold = 1;
% CNN inputs are 2D: time_window x n_features
% Four convolutional layers with 10 filters and filter size 10x1
% numResponses should be a 30x1 RUL prediction
lgraph = layerGraph;
lgraph = addLayers(lgraph, [sequenceInputLayer(numFeatures,"Name","input");
    sequenceFoldingLayer("Name", "fold")]);
lgraph = addLayers(lgraph, [convolution2dLayer([10
1],10,"Padding","Same","Name","conv_1");
    tanhLayer("Name","tanh_1")]);
lgraph = addLayers(lgraph, [convolution2dLayer([10
1],10,"Padding","Same","Name","conv_2");
    tanhLayer("Name","tanh_2")]);
lgraph = addLayers(lgraph, [convolution2dLayer([10
1],10,"Padding","Same","Name","conv_3");
    tanhLayer("Name","tanh_3")]);

lgraph = addLayers(lgraph, [convolution2dLayer([10
1],10,"Padding","Same","Name","conv_4");
    tanhLayer("Name","tanh_4")]);
lgraph = addLayers(lgraph, [convolution2dLayer([3
1],1,"Padding","Same","Name","conv_5");
    tanhLayer("Name","tanh_5")]);
lgraph = addLayers(lgraph, [sequenceUnfoldingLayer("Name", "unfold")
    flattenLayer("Name", "flatten")
    dropoutLayer(dropoutrate, "Name", "dropout")]);
lgraph = addLayers(lgraph, [fullyConnectedLayer(numHiddenNodes, "Name", "FC_1");
    tanhLayer("Name","tanh_6")]);
lgraph = addLayers(lgraph, [fullyConnectedLayer(1, "Name", "FC_2");
```

```
    regressionLayer("Name", "output")]);
lgraph = connectLayers(lgraph, "fold/out", "conv_1");
lgraph = connectLayers(lgraph, "tanh_1", "conv_2");
lgraph = connectLayers(lgraph, "tanh_2", "conv_3");
lgraph = connectLayers(lgraph, "tanh_3", "conv_4");
lgraph = connectLayers(lgraph, "tanh_4", "conv_5");
lgraph = connectLayers(lgraph, "tanh_5", "unfold/in");
lgraph = connectLayers(lgraph, "dropout", "FC_1");
lgraph = connectLayers(lgraph, "fold/miniBatchSize", "unfold/miniBatchSize");
lgraph = connectLayers(lgraph, "tanh_6", "FC_2");
% Training Options
options = trainingOptions('adam', ...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'InitialLearnRate',InitialLearnRate, ...
    'GradientThreshold',GradientThreshold, ...
    'Shuffle','never', ...
    'Plots','training-progress',...
    'ValidationData', {NN_input_val, NN_RUL_val_new}, ...
    "OutputNetwork", "best-validation-loss",... % added to keep best model
    'Verbose',0);
%%
for i=1:size(NN_RUL_train,2)
    NN_RUL_train_new(i) = NN_RUL_train(30,i);
end
%%
for i=1:size(NN_RUL_test,2)
    NN_RUL_test_new(i) = NN_RUL_test(30,i);
end
%%
for i=1:size(NN_RUL_val,2)
    NN_RUL_val_new(i) = NN_RUL_val(30,i);
end
%%
net = trainNetwork(NN_input_train, NN_RUL_train_new,lgraph,options);
%%
t_hat_test = predict(net, NN_input_test);
%RMSE_train = {sqrt(mean((Y_train - t_hat_train).^2))}
RMSE_test = {sqrt(mean((NN_RUL_test_new - t_hat_test).^2))};
%%
for i = 1:width(NN_RUL_test)
    YTestLast(i) = NN_RUL_test(30,i);
end
%end
figure()
plot(t_hat_test, YTestLast, 'X')
hold on
title("LSMT Predictions for FD001 Test. RMSE = ", RMSE_test)
xlabel("Predicted RUL")
ylabel("True RUL")

hold off
```