Final 492 Report

Thomas Hegarty

ID: 40155703

Introduction

The primary objective of this notebook is to illustrate several simple techniques which improve the predictive performance of Parkinson's Disease progression (PD) on a population of PD patients. The progression of PD is measured using the PPMI database (https://ida.loni.usc.edu/login.jsp) which consists of UPDRS-III composite scores from 3,812 participants over 13 years. More specifically, we focus on the 3rd subtest of UPDRS-III, which is composed of in clinic motor examinations on patients and controls. This is widely considered (and some quick tests support this claim) to be the more statistically stable of the 4 sub-tests given its mechanical in-person nature. Our results show that a few simple techniques can significantly reduce model variance relative to the predicted outcome, without model selection, tuning, or parameterization. Additionally, we show another feasible approach using patient medication states alongside ideas for follow up research.

Methods

Approach and Logic

The two approaches we pursued during this research were:

1. An Interval Approach, which is briefly explained as an algorithm that buckets patient entries into 3 (or more) non-overlapping contiguous time intervals, each one greater than 6 months. The resulting intervals and their aggregated (or randomly selected values) are then used as the testing and training splits in k-fold cross validation
    a. Here we can apply a de-noising technique by taking the mean of entries within a given interval as the UPDRS-III score for that interval, this is compared to taking a random entry from that same interval.
    b. Here the primary data set is split into 3 sets based on which medication state the patient is on, ON, OFF, un-medicated (NaN), and the models are trained separately on each. So predicted results here will be conditional on which state the patient expects to be in for future visits.
2. A State Model Approach, who's objective is to use patients medication states, ON or OFF (medication at the time of examination), to improve predictive performance. Under this framework each of the patient's 3 randomly selected visits are tagged with as ON or OFF using one-hot encoding and passed into the same k-fold cross validation

Generally, both approaches outcomes are compared to the standard deviation of UPDRS-III scores in their respective original datasets, this acts as a pure prediction error baseline.

Model Selection

As a preamble, this research turned out to be less focused on model selection and parameterisation than initially expected. The models themselves are only as good as the data they train on (mostly), and here the discovery is that a clear framing of the PD progression problem as a time-series forecasting problem does a surprising amount of work. On that note, the models used here are placeholders. I selected Random-Forest-Regression (RFR) primarily because my supervisor (who's vastly more knowledgeable and skilled than I) recommended it. I decided to use Ridge-Regression as the inclusion of a linear model to compare to the more complex RFR could add some clues for future research on optimal model selection (though not a true comparison given the lack of a structure hypothesis test). Additionally, Ridge-Regression could be more robust to the amount of noise inherent in the UPDRS-III dataset due to the slope penalty vs. simple Linear-Regression.

Placeholder Models

- Random Forest Regression, default 5 nodes
- Ridge Regression

Interval Approach: Data Preparation Algorithm

The objective is to create rows in a wide dataframe with the following format: `t0 | u0 | t1 | u1 | ... | t_i | u_i`, where `t_i` is the date where score `u_i` was measured. Each row represents the `[date | score]` entries for a given patient. The following algorithm performs this task:

1. Given the non-uniform number of respective patient entries (some have 1, others more than 20) and to retain as much data as possible for our models we decided to bound the number of observations per patient to 3 (`i = 3`), which allowed us to keep patients with at least 3 and those with many more entries. So each patient's entries will be: `t0 | u0 | t1 | u1 | t_2 | u_2`
    a. This is done by using the individual patients entries to create a calendar, which is then segmented into 3 non-overlapping intervals, each greater than 6 months. The 6 month limit was chosen as it allows the disease to progress and therefore reduces overall measurement noise. The interval time delta is calculated as (end - start) / 3, where both end and start are drawn from the patients first and last visit INFODT (or date of examination). The patients individual calendar is then created by iteratively adding this time delta to the start date. This creates 4 dates, and thereby 3 intervals.
2. From here, the patients 3 entries, `u0 | u1| u2` are produced as follows (through the functions `process_mean_intervals` and `process_rand_intervals`):
    a. For the random baseline, each of the 3 entries are randomly drawn from it's respective interval [Note 1]
    b. The improved model takes the mean UPDRS-III score over the respective intervals as its entries
3. For both methods, the date attached to these intervals, `t_i`, is the median date of each interval, which is calculated by adding 1/2 the time delta to each of the intervals start dates.
4. Finally, the returned dataframes are converted to a wide format such that each row represents the 3 selected (or mean aggregated) INFODT-score combinations for a given PATNO.

Note 1: It's also possible to randomly select 3 sequential patient entries such that they are taken at least 6 months from each other. But this wouldn't allow us to best measure the improvment in model performance using the mean over intervals.

Note 2: (Rough) Function documentation for helper functions

- `median_date_calc` : returns the median date of a grouped object
- `interpolate_same_month` : Returns the specified computation (or random selection) on a grouped object. The grouping is done to filter same PATNO's and equivalent dates for the same PATNO, which for this analysis adds noise
- `pivot_wide` : Returns the passed dataframe in wide format, some pre-processing is done in the `process_mean_intervals` and `process_rand_intervals` functions

Interval Processing Code

```
# Data Processing and Helper Functions
def median_date_calc(date_group: pd.DataFrame, interval: pd.Timedelta) -> list:
    dates = date_group['INFODT'].tolist()
    date1 = dates[0] + interval / 2
    date2 = dates[1] + interval / 2
    date3 = dates[2] + interval / 2
    return [(date1-date1).days, (date2-date1).days, (date3-date1).days]
```

```
def interpolate_same_month(df: pd.DataFrame, method = 'rand') -> pd.DataFrame:
    # Select maximum score from same month measurements
    temp_df = df.copy()
    temp_df['YEAR_MONTH'] = temp_df['INFODT'].dt.to_period('M')

    #takes random selctions/maximum/minimum/mean of values which share same month and year
    if method == 'rand':
        sample = temp_df.groupby(['PATNO', 'YEAR_MONTH'])['NP3TOT'].apply(lambda x: x.sample(1)).reset_index()
        result = pd.merge(temp_df, sample, on=['PATNO', 'YEAR_MONTH', 'NP3TOT'])

        result.drop(columns=['YEAR_MONTH', 'level_2'], inplace=True)
        return result.drop_duplicates(subset=['PATNO', 'INFODT']).reset_index(drop=True)

    elif method == 'min':
        #result = pd.merge(temp_df, temp_df.groupby(['PATNO', 'YEAR_MONTH'])['NP3TOT'].min(), on=['PATNO', 'YEAR_MONTH', 'NP3T
OT'])
        temp_df['NP3TOT'] = temp_df.groupby(['PATNO', 'YEAR_MONTH'])['NP3TOT'].transform('min')
    elif method == 'max':
        #result = pd.merge(temp_df, temp_df.groupby(['PATNO', 'YEAR_MONTH'])['NP3TOT'].max(), on=['PATNO', 'YEAR_MONTH', 'NP3T
OT'])
        temp_df['NP3TOT'] = temp_df.groupby(['PATNO', 'YEAR_MONTH'])['NP3TOT'].transform('max')
    else:
        #result = pd.merge(temp_df, temp_df.groupby(['PATNO', 'YEAR_MONTH'])['NP3TOT'].mean(), on=['PATNO', 'YEAR_MONTH', 'NP3
TOT'])
        temp_df['NP3TOT'] = temp_df.groupby(['PATNO', 'YEAR_MONTH'])['NP3TOT'].transform('mean')


    temp_df.drop(columns=['YEAR_MONTH'], inplace=True)
    temp_df.reset_index(drop=True, inplace=True)
    return temp_df.drop_duplicates(subset=['PATNO', 'INFODT'])

def pivot_wide(df: pd.DataFrame, cols: int = 3) -> pd.DataFrame:

    df['time_index'] = (
            df.groupby('PATNO')['INFODT']
            .rank(method='first')
            .astype(int)-1
        )

    time_df = df[['PATNO', 'INFODT', 'time_index']]

    score_wide = df.pivot(
        index='PATNO',
        columns='time_index',
        values='NP3TOT'
    ).reset_index()

    time_wide = time_df.pivot(
        index='PATNO',
        columns="time_index",
        values='INFODT'
    ).reset_index()

    score_wide.columns = ['PATNO'] + [f'u{i}' for i in range(cols)] # resets score column names correctly
    time_wide.columns = ['PATNO'] + [f't{i}' for i in range(cols)] # resets time_index column names accordingly

    merged = pd.merge(score_wide, time_wide, on='PATNO')
    merged.drop(columns='PATNO', inplace=True)

    new_cols = []
    for i in range(cols): # re order columns for ML piplines
        new_cols.append(f't{i}')
        new_cols.append(f'u{i}')

    merged = merged[new_cols]

    return merged.drop(columns='t0')


# keep_small_intervals flag allows us to keep some entries with <6mo intervals
# assumes process_same_dates has already been calles
def process_mean_intervals(df: pd.DataFrame, blocks: int = 3, keep_small_intervals = True) -> pd.DataFrame:
    patnos = df['PATNO'].unique().tolist()
    result = pd.DataFrame(columns=['PATNO', 'INFODT', 'NP3TOT'])
    for patno in patnos:
        df_pat = df[df['PATNO'] == patno]
        start = df_pat['INFODT'].min()
        end = df_pat['INFODT'].max()
        diff = end - start

        if (diff < pd.Timedelta(days=180)) or (df_pat.index.size < 3):
            continue
```

```
        interval = diff // blocks

        # drop small intervals < 6mo
        if not keep_small_intervals and interval < pd.Timedelta(days=180):
            continue

        group_counts = df_pat.groupby(pd.Grouper(key='INFODT', freq= f'{interval.days+1}D'))['NP3TOT'].count()
        temp = df_pat.groupby(pd.Grouper(key='INFODT', freq= f'{interval.days+1}D'))['NP3TOT'].mean().reset_index()
        temp['group_counts'] = group_counts.values
        # checks for Failed Time Group Function
        if temp['NP3TOT'].isnull().values.any():
            continue

        dates = median_date_calc(df_pat.groupby(pd.Grouper(key='INFODT', freq= f'{interval.days}D'))['NP3TOT'].mean().reset_in
dex(), interval)

        # this logic will create some entries with < 6mo intervals, but this is only due to the abover grouper function using
a slightly different calendar
        temp['INFODT'] = dates
        temp['PATNO'] = patno

        result = pd.concat([result, temp], ignore_index=True)

    return result

def process_random_intervals(df: pd.DataFrame, blocks: int = 3, keep_small_intervals = True) -> pd.DataFrame:
    patnos = df['PATNO'].unique().tolist()
    patnos = df['PATNO'].unique().tolist()
    result = pd.DataFrame(columns=['PATNO', 'INFODT', 'NP3TOT'])
    for patno in patnos:
        df_pat = df[df['PATNO'] == patno]
        start = df_pat['INFODT'].min()
        end = df_pat['INFODT'].max()
        diff = end - start

        if (diff < pd.Timedelta(days=180)) or (df_pat.index.size < 3):
            continue

        interval = diff // blocks

        # drop small intervals < 6mo
        if not keep_small_intervals and interval < pd.Timedelta(days=180):
            continue

        temp = (
            df_pat.groupby(pd.Grouper(key='INFODT', freq=f'{interval.days+1}D'))['NP3TOT']
            .apply(lambda x: x.sample(n=1, random_state=1) if len(x) > 0 else None)
            .dropna()  # Drop groups where no sample was taken (i.e., empty groups)
            .reset_index()
        )

        # checks for Failed Time Group Function // when the groupby function can't return 3 groups (like it should)
        if temp.index.size != 3:
            #print("Failed Time Group Function")
            continue

        dates = median_date_calc(df_pat.groupby(pd.Grouper(key='INFODT', freq= f'{interval.days}D'))['NP3TOT'].mean().reset_in
dex(), interval)

        # this logic will create some entries with < 6mo intervals, but this is only due to the abover grouper function using
a slightly different calendar
        temp['INFODT'] = dates
        temp['PATNO'] = patno
        temp.drop(columns=['level_1'], inplace=True)

        result = pd.concat([result, temp], ignore_index=True)

    return result
```

▼ State Approach: Data Preparation Algorithm

The goal of this approach is to allow the models to train on patient medication state data. The original data set is filtered to only include patient visits with either an ON or OFF state (as in ON medication or OFF from being previously medicated). The resulting visits would then be fed to the models in a one-hot-encoding format.

The first step is simple

1. For each patient we randomly select 3 visits such that all are greater than 6 months of each other

Then we only need another iteration check for visits which occurred in the same month and year (as our dataset doesn't distinguish days)

2. For each patient, and their previously selected visits we check if the PATNO + INFODT visit combination is a duplicate)
    a. If so, then the compliment state and corresponding score replace the original state and score
    b. This is then combined to the other two visits, sorted, prepared for pivoting, and appended to a new temporary dataframe in the same format as the original from step 1 (Note 1)

3. After finishing, the new temp dataframe is pivoted then appended to the pivot version of the original from step 1.

▼ *Original Format to Pivot Format Visuals*

Original

PATNO	INFODT	score	PDSTATE	time_delta	time_index
0	3001	2015-04-01	39.0	OFF	0.000000
1	3001	2016-06-01	34.0	OFF	14.233333
2	3001	2018-03-01	36.0	OFF	35.500000
3	3002	2015-09-01	38.0	OFF	0.000000
4	3002	2016-04-01	30.0	ON	7.100000
5	3002	2019-03-01	35.0	OFF	42.566667

Pivot Format

t0	off_0	on_0	u0	t1	off_1	on_1	u1	t2	off_2	o
0	0.0	1	0	39.0	14.233333	1	0	34.0	35.500000	1
1	0.0	1	0	38.0	7.100000	0	1	30.0	42.566667	1
2	0.0	1	0	59.0	12.200000	0	1	39.0	24.366667	0
3	0.0	0	1	34.0	21.266667	0	1	22.0	80.100000	0
4	0.0	0	1	30.0	33.466667	0	1	38.0	58.833333	0
5	0.0	0	1	43.0	30.400000	0	1	29.0	41.566667	0

Note 1: Each new visit set is anonoymized from the original PATNO by adding an offset value to it, that way the pivoting function works properly (not an optimal approach but it works)

▼ **State Approach: Code**

```
UPDRS3 = "data/MDS-UPDRS_Part_III_10Jun2024.csv"
patient_status = "data/Participant_Status_03Jun2024.csv"

df3 = pd.read_csv(UPDRS3)
df_pat_stat = pd.read_csv(patient_status) #patient status data
df3 = df3.dropna(subset=['NP3TOT']).reset_index() # will keep for now, might need to include nans
df3['INFODT'] = pd.to_datetime(df3['INFODT'], format="%m/%Y") #reformat INFODT (Assesment Date) to date-time objects
df3['PDSTATE'] =  df3['PDSTATE'].fillna("None")
df3 = df3[["PATNO", "EVENT_ID", "INFODT", "PDSTATE", "PAG_NAME", "NP3TOT"]]

desired_cols_df_pat = {'PATNO', 'COHORT', 'ENROLL_STATUS'}
pat_filtered = df_pat_stat.drop(columns=set(df_pat_stat.columns) - desired_cols_df_pat)
df3_full = pd.merge(df3, pat_filtered, on="PATNO")
df3_full = df3_full[df3_full['ENROLL_STATUS'].isin(['Enrolled', 'Withdrew', 'Complete'])]
df3_full.drop(columns=['ENROLL_STATUS'], inplace=True)
df3_full = df3_full.sort_values(['PATNO', 'INFODT'])

# Partition our data sets
upd3_control = df3_full[df3_full['COHORT'] == 2]
upd3_PD = df3_full[df3_full['COHORT'] == 1]
# Data set of interest...
upd3_PD_off_on = upd3_PD[(upd3_PD['PDSTATE'] == 'ON') | (upd3_PD['PDSTATE'] == 'OFF') | (upd3_PD['PAG_NAME'] == 'NUPDR3OF') |
(upd3_PD['PAG_NAME'] == 'NUPDR3ON')].reset_index(drop=True)
```

```
def process_dates_caller(input: pd.DataFrame, entries: int = 3, same_month_flag = False) -> pd.DataFrame:
    # helps maintain random states of dataframes to compare to baselines, default is to assume input is already interpolated
    df = interpolate_same_month(input, 'rand') if same_month_flag else input

    patnos = df['PATNO'].unique().tolist()
    df = input.copy()
    limit = pd.Timedelta(6*30, unit='D') # 180 days ~ 6mo
    date_chunks = []

    for id in patnos:

        visits = df[df['PATNO'] == id]['INFODT'].tolist()
        soln = []
        lim = 0

        if len(visits) < entries:
            #print(f"Not Enough Entries for ID: {id}")
            continue

        sub_df = df[df['PATNO'] == id]
        random.seed(10)
        """
            Include same month check and processor here
        """

        while True:
            soln = np.random.choice(visits, entries, replace=False)
```

```
            if (soln[1] - soln[0]) > limit and (soln[2] - soln[1]) > limit:
                break
            if lim > 100:
                #print(f"No Possible Calendar Combination Found for ID: {id}")
                break
            lim += 1

        # add additional same_month flag here? then double append the two rows?
        if lim < 100:
            for i in range(entries):
                date_chunks.append([id, sub_df.loc[sub_df['INFODT'] == soln[i], 'INFODT'].values[0], sub_df.loc[sub_df['INFOD
T'] == soln[i], 'NP3TOT'].values[0], sub_df.loc[sub_df['INFODT'] == soln[i], 'PDSTATE'].values[0]])

    res = pd.DataFrame(date_chunks, columns=['PATNO', 'INFODT', 'score', 'PDSTATE'])

    # Prepare dataframe for pivoting

    res['time_delta'] = (
        res.groupby('PATNO')['INFODT']
        .transform(
            lambda x: (x - x.min()) / np.timedelta64(30, 'D')
        )
    )

    res['time_index'] = (
            res.groupby('PATNO')['INFODT']
            .rank(method='first')
            .astype(int)-1
        )

    return res



def pivot_wide_with_states(input: pd.DataFrame, cols: int = 3) -> pd.DataFrame:

    time_df = input[['PATNO', 'time_delta', 'time_index']]

    score_wide = input.pivot(
        index='PATNO',
        columns='time_index',
        values='score'
    ).reset_index()

    time_wide = time_df.pivot(
        index='PATNO',
        columns="time_index",
        values='time_delta'
    ).reset_index()

    off_wide = input.pivot(
        index='PATNO',
        columns='time_index',
        values='OFF'
    ).reset_index()

    on_wide = input.pivot(
        index='PATNO',
        columns='time_index',
        values='ON'
    ).reset_index()

    score_wide.columns = ['PATNO'] + [f'u{i}' for i in range(cols)] # resets score column names correctly
    time_wide.columns = ['PATNO'] + [f't{i}' for i in range(cols)] # resets time_index column names accordingly
    off_wide.columns = ['PATNO'] + [f'off_{i}' for i in range(cols)] # resets time_index column names accordingly
    on_wide.columns = ['PATNO'] + [f'on_{i}' for i in range(cols)] # resets time_index column names accordingly

    #merged = pd.merge(score_wide, time_wide, on='PATNO') # use PATNO to merge
    #print(merged)
    #merged.drop(columns='PATNO', inplace=True) # drop PATNO, no longer needed

    merged = pd.concat([score_wide, time_wide, off_wide, on_wide], axis=1)
    merged.drop(columns='PATNO', inplace=True)


    new_cols = []
    for i in range(cols): # re order columns for ML piplines
        new_cols.append(f't{i}')
        new_cols.append(f'off_{i}')
        new_cols.append(f'on_{i}')
        new_cols.append(f'u{i}')

    merged = merged[new_cols]
```

```
    return merged

def fill_same_month(og_df: pd.DataFrame, pre_proc: pd.DataFrame) -> pd.DataFrame:
    #filters on duplicate dates within patient ids
    same_mo_rows = og_df.groupby(['PATNO', 'INFODT']).filter(lambda x: len(x) > 1).index
    same_mo_df = og_df.iloc[same_mo_rows].copy()
    duplicate_patnos = same_mo_df['PATNO'].unique().tolist()
    final_res = pd.DataFrame(columns=['PATNO', 'INFODT', 'score', 'PDSTATE', 'time_delta', 'time_index'])

    offset = 0
    for id in duplicate_patnos:

        selected_dates = pre_proc[pre_proc['PATNO'] == id]['INFODT'].tolist()
        duplicate_dates = same_mo_df[same_mo_df['PATNO'] == id]['INFODT'].tolist()

        res_slice = pre_proc[pre_proc['PATNO'] == id].copy()
        dup_slice = same_mo_df[same_mo_df['PATNO'] == id].copy()
        temp = []
        for date in selected_dates:
            if date in duplicate_dates:
                #selected date = date
                selected_state = res_slice['PDSTATE'].values[0]

                temp.append([
                    id,
                    date,
                    dup_slice[(dup_slice['INFODT'] == date) & (dup_slice['PDSTATE'] != selected_state)]['NP3TOT'].values[0],
                    dup_slice[(dup_slice['INFODT'] == date) & (dup_slice['PDSTATE'] != selected_state)]['PDSTATE'].values[0]
                    ])

                #print(temp)

        if len(temp) > 0:
            for entry in temp:
                #print(res_slice[res_slice['INFODT'] != entry[1]][['PATNO', 'INFODT', 'score', 'PDSTATE']], entry[1])
                temp_res = res_slice[res_slice['INFODT'] != entry[1]][['PATNO', 'INFODT', 'score', 'PDSTATE']].copy()
                temp_res.loc[3] = entry
                #sort by date to get back to the correct time index
                temp_res.sort_values('INFODT', inplace=True)

                #add in the necessary columns for future pivoting
                temp_res['time_delta'] = (
                temp_res.groupby('PATNO')['INFODT']
                .transform(
                    lambda x: (x - x.min()) / np.timedelta64(30, 'D')
                    )
                )

                temp_res['time_index'] = (
                        temp_res.groupby('PATNO')['INFODT']
                        .rank(method='first')
                        .astype(int)-1
                    )

                # needed to add this due to my own stupidity
                offset += 1 if len(temp) > 1 else offset

                temp_res['PATNO'] = offset

                #print(temp_res)

                final_res = pd.concat([final_res, temp_res], axis=0, join='outer', ignore_index=True)

    return final_res
```

▼ Results

▼ Interval Model Code

```
# Model testing function and pipeline

# Imports
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import Ridge
from sklearn.pipeline import Pipeline

# Pipline Data Sets
nan = interpolate_same_month(upd3_PD_nan, 'rand')
off = interpolate_same_month(upd3_PD_off, 'rand')
on = interpolate_same_month(upd3_PD_on, 'rand')
control = interpolate_same_month(upd3_control, 'rand')
```

```
#BASELINES RANDOM
base_nan_rand = np.std(nan['NP3TOT'])
base_off_rand = np.std(off['NP3TOT'])
base_on_rand  = np.std(on['NP3TOT'])
base_control_rand = np.std(control['NP3TOT'])

PD_nan_rand = pivot_wide(process_random_intervals(nan, 3, keep_small_intervals=True))
PD_off_rand = pivot_wide(process_random_intervals(off, 3, keep_small_intervals=True))
PD_on_rand = pivot_wide(process_random_intervals(on, 3, keep_small_intervals=True))
control_rand = pivot_wide(process_random_intervals(control, 3, keep_small_intervals=True))

#DE-NOISED
nan_mean = interpolate_same_month(upd3_PD_nan, 'mean')
off_mean = interpolate_same_month(upd3_PD_off, 'mean')
on_mean = interpolate_same_month(upd3_PD_on, 'mean')
control_mean = interpolate_same_month(upd3_control, 'mean')

base_nan_mean = np.std(nan_mean['NP3TOT'])
base_off_mean = np.std(off_mean['NP3TOT'])
base_on_mean  = np.std(on_mean['NP3TOT'])
base_control_mean = np.std(control_mean['NP3TOT'])

PD_nan_mean = pivot_wide(process_mean_intervals(nan_mean, 3, keep_small_intervals=True))
PD_off_mean = pivot_wide(process_mean_intervals(off_mean, 3, keep_small_intervals=True))
PD_on_mean = pivot_wide(process_mean_intervals(on_mean, 3, keep_small_intervals=True))
control_mean = pivot_wide(process_mean_intervals(control_mean, 3, keep_small_intervals=True))


# Model Caller Functions
def test_models(df:pd.DataFrame, base_line: float, identifier: str = "DEFAULT", folds = 5):
    X = df.iloc[: , : len(df.columns)-1]
    y = df.iloc[: , len(df.columns)-1 : ]

    pipe_rf = Pipeline(steps=[('model', RandomForestRegressor())])
    pipe_ridge = Pipeline(steps=[('model', Ridge())])
    # .values will give the values in a numpy array (shape: (n,1))
    # .ravel will convert that array shape to (n, ) (i.e. flatten it)
    score_test_rf = -1 * cross_val_score(pipe_rf, X, y.values.ravel(), cv=folds, scoring="neg_root_mean_squared_error")
    score_test_ridge = -1 * cross_val_score(pipe_ridge, X, y.values.ravel(), cv=folds, scoring="neg_root_mean_squared_error")

    print(f"Testing {identifier} Model")
    print("-----------------------------------------------------------------")
    print(f"STD(NP3TOT) {identifier}: {base_line}")
    print("-----------------------------------------------------------------")

    print(f"Startings models, Simple Cross Validation, k = {folds}, VS std(UPDRS): \n")
    print(" Ridge Regression RMSE by fold: ", score_test_ridge)
    print(" Random Forest RMSE by fold: ", score_test_rf, "\n")
    print(f" Ridge Regression mean RMSE: {score_test_ridge.mean()}", '\n', f"Random Forest, default 5-nodes, mean RMSE: {score
_test_rf.mean()}")

    return [score_test_rf.mean(), score_test_ridge.mean()]

def compare_models(df_mean: pd.DataFrame, df_rand: pd.DataFrame, base_mean: float, base_rand: float):
    mean = test_models(df_mean, base_mean, "MEAN")
    rand = test_models(df_rand, base_rand, "Random Selection")

    print("-----------------------------------------------------------------")
    print("Comparing Models")
    print("-----------------------------------------------------------------")
    print(f"Mean Model: {mean}")
    print(f"Random Model: {rand}")
```

▼ **Interval Method Summary Table**

RMSE on predicted variable

Model	Method	NaN	OFF	ON	Control
	Base STD	**10.367**	**14.300**	**12.366**	**3.658**
Ridge Regression	**Random Interval**	7.091	11.551	9.705	3.697
	Mean Interval	6.127	9.767	8.493	3.184
relative improvement		*13.59%*	*15.44%*	*12.49%*	*13.88%*
Random Forest	**Random Interval**	7.595	12.802	10.668	4.424
	Mean Interval	6.479	10.353	9.068	3.142
relative improvement		*14.69%*	*19.13%*	*15.00%*	*28.98%*

Mean Improvement across models	**1.037**	**2.090**	**1.393**	**0.811**

▼ State Model Code

```
test = process_dates_caller(upd3_PD_off_on, 3, True)
on_hot = pd.get_dummies(test['PDSTATE'], dtype=int)
on_hot.drop(columns=['None'], inplace=True)
PD_on_off_encoded = pd.merge(test, on_hot, left_index=True, right_index=True)
res_on_off = pivot_wide_with_states(PD_on_off_encoded, 3)

duplicates = fill_same_month(upd3_PD_off_on, test)
one_hot_dup = pd.get_dummies(duplicates['PDSTATE'], dtype=int)
one_hot_dup.drop(columns=['None'], inplace=True)
dup_encoded = pd.merge(duplicates, one_hot_dup, left_index=True, right_index=True)
dup_res = pivot_wide_with_states(dup_encoded, 3)
dup_res.dropna(inplace=True)

analytical_set = pd.concat([res_on_off, dup_res], axis=0, join='outer', ignore_index=True)
```

```
# Simple stratified Cross Validation testing
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import Ridge
from sklearn.pipeline import Pipeline

base_std = upd3_PD_off_on['NP3TOT'].std()
folds = 5
pipe_rf = Pipeline(steps=[('model', RandomForestRegressor())])
pipe_ridge = Pipeline(steps=[('model', Ridge())])

X = analytical_set.loc[:, analytical_set.columns != 'u2']
y = analytical_set['u2']
score_test_rf = -1 * cross_val_score(pipe_rf, X, y.values.ravel(), cv=folds, scoring="neg_root_mean_squared_error")
score_test_ridge = -1 * cross_val_score(pipe_ridge, X, y.values.ravel(), cv=folds, scoring="neg_root_mean_squared_error")

print(f"STD(NP3TOT) : {base_std}")
print(" Ridge Regression RMSE by fold: ", score_test_ridge)
print(" Random Forest RMSE by fold: ", score_test_rf, "\n")
print(f" Ridge Regression mean RMSE: {score_test_ridge.mean()}", '\n', f"Random Forest, default 5-nodes, mean RMSE: {score_tes
t_rf.mean()}")
```

▼ State Model Summary Table

RMSE on predicted variable

Model	Method	OFF & ON States
	Base STD	**13.651**
Ridge Regression	**Random Selection**	10.922
Random Forest	**Random Selection**	8.404

Mean Improvement over Base STD	**3.784**

▼ Conclusion

The results above show that predictive improvement is possible with simple approaches. Without complicated models and parameterization there's still many low hanging fruit in data structuring and de-noising techniques. Regardless, this project is still very much a preliminary step in the right direction as we only viewed a subset of the UPDRS-III scores, ignored medical imaging, and spent little time toying with models. There's still much work to do.

▼ Ideas for Further Research

1. Include selected features form the remaining 3 UPDRS-III sub-tests.
2. Include medical Imagery and medication progression data.
3. Mix the results from the Interval and State Model Approaches.
4. Perform rigorous model selection and paramterization process.
5. Include promising indicating biomarkers in study data

Will add updates to my personal git repo in the future, with Patient identifiers scrambled