**Data preprocessing**:

First we extract each of the time series data into two different variables.

Features Baseline and Features Follow up

features\_baseline = [

'interview\_age.baseline\_year\_1\_arm\_1',

'KSADSintern.baseline\_year\_1\_arm\_1',

'nihtbx\_cryst\_agecorrected.baseline\_year\_1\_arm\_1',

'ACEs.baseline\_year\_1\_arm\_1',

'avgPFCthick\_QA.baseline\_year\_1\_arm\_1',

'rsfmri\_c\_ngd\_cgc\_ngd\_cgc\_QA.baseline\_year\_1\_arm\_1',

'rsfmri\_c\_ngd\_dt\_ngd\_dt\_QA.baseline\_year\_1\_arm\_1'

]

features\_followup = [

'interview\_age.2\_year\_follow\_up\_y\_arm\_1',

'KSADSintern.2\_year\_follow\_up\_y\_arm\_1',

'nihtbx\_cryst\_agecorrected.2\_year\_follow\_up\_y\_arm\_1',

'ACEs.2\_year\_follow\_up\_y\_arm\_1',

'avgPFCthick\_QA.2\_year\_follow\_up\_y\_arm\_1',

'rsfmri\_c\_ngd\_cgc\_ngd\_cgc\_QA.2\_year\_follow\_up\_y\_arm\_1',

'rsfmri\_c\_ngd\_dt\_ngd\_dt\_QA.2\_year\_follow\_up\_y\_arm\_1',

]

Other features are defined as cross sectional data and put into a separate variable.

cross\_sectional\_features = [

'rel\_family\_id',

'demo\_sex\_v2',

'race\_ethnicity',

'acs\_raked\_propensity\_score',

'speechdelays',

'motordelays',

'fam\_history\_8\_yes\_no',

]

We drop any rows with missing labels or missing time series data as this is an important feature for any lstm model.

df.dropna(subset=features\_all\_time + ['group\_PDvLP\_3timepoint'], inplace=True)

We use KNN imputer (averages nearest neighbors from other data available from those neighbors) to fill in any missing data in the cross sectional. Note this is done aftering splitting training and test data to not cause any data leakage.

imputer = KNNImputer(n\_neighbors=5) #comment these two lines for non data imputation

X\_train\_cross = imputer.fit\_transform(X\_train\_cross)

X\_test\_cross = imputer.transform(X\_test\_cross)

**Hyperparameters**:

We do hyperparameter tuning on:

Alpha(used for focal loss)

LSTM Units (equivalent to nodes in deep neural network)

Dropout rate (regularization of LSTM (dropping out nodes)

Dense units (feed time series to LSTM - LSTM spits out vector - vector feeds into dense units with cross sectional data).

Batch size (back propagation number of instances i.e. how much of data is used before weights)

We did LOSO-CV tracked validation set accuracy, precision, recall, npv, specificity, and f1 and chose the hyperparameters with the highest validation set accuracy with some emphasis to precision.

ALPHA\_HP = 0.75

LSTM\_UNITS\_HP = 64

DROPOUT\_HP = 0.3

DENSE\_UNITS\_HP = 32

BATCH\_SIZE\_HP = 16

**Model Implementation**:

We scale the cross sectional and time series features first with standardscaler()

And concatenate the time series data into a stack to use as an input to the LSTM

seq = np.stack([baseline, followup])

X\_test\_ts.append(seq)

We build an LSTM model that processes the time series data and outputs a vector representation. This vector is then concatenated with cross-sectional data (static features). The combined features are passed through one dense layer with ReLu activation, culminating in a final dense layer that outputs the prediction.

# Time series input

input\_ts = Input(shape=(timesteps, ts\_features), name='time\_series\_input')

lstm\_out = LSTM(lstm\_units, return\_sequences=False)(input\_ts)

lstm\_out = Dropout(dropout\_rate)(lstm\_out)

# Cross-sectional input

input\_cross = Input(shape=(cross\_features,), name='cross\_sectional\_input')

# Concatenate LSTM output and cross-sectional data

concatenated = Concatenate()([lstm\_out, input\_cross])

# Dense layers

dense1 = Dense(dense\_units, activation='relu')(concatenated)

output = Dense(1, activation='sigmoid')(dense1)

Our loss function used is focal loss, which helps address class imbalance by putting more focus on minority class samples and on examples where the model is less confident in its predictions. We use the Adam optimizer for training.

model.compile(optimizer='adam', loss=focal\_loss(alpha=ALPHA\_HP), metrics=['accuracy']) #focal loss

**Training**:

We compiled the model and use early stopping with a validation set to prevent overfitting

early\_stop = EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

model.fit(

[X\_train\_ts\_scaled, X\_train\_cross\_scaled], y\_train,

epochs=20,

batch\_size=BATCH\_SIZE\_HP,

validation\_split=0.1,

callbacks=[early\_stop],

verbose=0

)

We train the model, and use a standard threshold of 0.5 to predict whether to classify or not.