```
import numpy as np
import pandas as pd
import patsy as pt
import os.path
import pickle
from functools import partial
from multiprocessing import Pool
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split, StratifiedShuffleSplit

obsPeriod = {
    'start': pd.Timestamp('2015-02-01'),
    'end': pd.Timestamp('2016-02-01')
}

actPeriod = {
    'start': pd.Timestamp('2015-10-01'),
    'end': pd.Timestamp('2016-02-01')
}

predPeriod = {
    'start': pd.Timestamp('2016-02-01'),
    'end': pd.Timestamp('2016-06-01')
}


class RmtppData:
    PATH = '../../data/rmtpp/'

    def instance():
        if os.path.isfile(RmtppData.PATH+'rmtpp_data.pkl'):
            return pickle.load(open(RmtppData.PATH+'rmtpp_data.pkl', 'rb'))
        else:
            d = RmtppData()
            d._initialise()
            return d

    def get_xy(self, min_n_sessions=10, n_sessions=10, target_sequences=False, preset='startUserTimeHours'):
        x_train, x_test, y_train, y_test = self.x_train, self.x_test, self.y_train_unscaled, self.y_test_unscaled
        x_train_unscaled, x_test_unscaled = self.x_train_unscaled, self.x_test_unscaled
        feature_indices = self.presets[preset]['feature_indices']
        features = self.presets[preset]['features']
        target_indices = self.presets[preset]['target_indices']
        targets = self.presets[preset]['target']

        # if encode_devices:
        #     feature_indices = [self.deviceEncIndex] + feature_indices
        #     features = ['device'] + features
        # else:
        #     feature_indices = self.deviceIndices + feature_indices
        #     features = self.devices + features

        y_train = y_train.apply(lambda x: x.T[target_indices].T)
        y_test = y_test.apply(lambda x: x.T[target_indices].T)
        x_train = x_train.apply(lambda x: x.T[feature_indices].T)
        x_test = x_test.apply(lambda x: x.T[feature_indices].T)
        x_train_unscaled = x_train_unscaled.apply(lambda x: x.T[feature_indices].T)
        x_test_unscaled = x_test_unscaled.apply(lambda x: x.T[feature_indices].T)

        # if not include_churned:
        #     x_train = x_train[~x_train.index.isin(self.churned_cust)]
        #     # x_test = x_test[~x_test.index.isin(self.churned_cust)]
        #     x_train_unscaled = x_train_unscaled[~x_train_unscaled.index.isin(self.churned_cust)]
        #     # x_test_unscaled = x_test_unscaled[~x_test_unscaled.index.isin(self.churned_cust)]
        #     y_train = y_train[~y_train.index.isin(self.churned_cust)]
        #     # y_test = y_test[~y_test.index.isin(self.churned_cust)]

        if min_n_sessions > 1:
            cust = self.num_sessions[self.num_sessions > min_n_sessions].index
            x_train = x_train[x_train.index.isin(cust)]
            x_test = x_test[x_test.index.isin(cust)]
            x_train_unscaled = x_train_unscaled[x_train_unscaled.index.isin(cust)]
            x_test_unscaled = x_test_unscaled[x_test_unscaled.index.isin(cust)]
            y_train = y_train[y_train.index.isin(cust)]
            y_test = y_test[y_test.index.isin(cust)]

        if n_sessions == -1:
            n_sessions = self.num_sessions.max()

        if target_sequences:
            y_train = _pad_x(y_train, n_sessions)
            y_test = _pad_x(y_test, n_sessions)
        else:
            y_train = np.array(y_train.apply(lambda x: x[-1]).tolist())
            y_test = np.array(y_test.apply(lambda x: x[-1]).tolist())

        x_train = _pad_x(x_train, n_sessions)
```

```
        x_test = _pad_x(x_test, n_sessions)
        x_train_unscaled = _pad_x(x_train_unscaled, n_sessions)
        x_test_unscaled = _pad_x(x_test_unscaled, n_sessions)

        return x_train, x_test, x_train_unscaled, x_test_unscaled, y_train, y_test, features, targets

    def _initialise(self):
        df_0 = self.df_0 = pd.read_pickle('../../data/rnn/first/rnn_stage2_df.pkl')
        churned = self.churned = df_0.groupby('customerId').last().churned
        self.churned_cust = churned.index[churned].values
        self.num_sessions = self.df_0.groupby('customerId').customerId.count()

        # encoded features (in range 1-..)
        self.deviceEncoder = LabelEncoder()
        df_0['device_enc'] = self.deviceEncoder.fit_transform(df_0.device) + 1
        df_0['hourOfDay_enc'] = df_0.hourOfDay + 1
        df_0['dayOfMonth_enc'] = df_0.dayOfMonth
        df_0['dayOfWeek_enc'] = df_0.dayOfWeek + 1

        # get times in days
        df_0['startUserDate'] = df_0.startUserTime.dt.date.apply(pd.Timestamp)
        df_0['startUserTimeDays'] = (df_0.startUserDate - obsPeriod['start']) / np.timedelta64(24, 'h')
        df_0['deltaNextDays'] = df_0.deltaNextHours / 24
        df_0['deltaPrevDays'] = df_0.deltaPrevHours / 24
        df_0['logDeltaNextDays'] = np.log(df_0.deltaNextDays + 1)
        df_0['logDeltaPrevDays'] = np.log(df_0.deltaPrevDays + 1)

        # add nextStartUserTimeHours
        df_0['nextStartUserTimeHours'] = df_0.startUserTimeHours + df_0.deltaNextHours
        df_0['nextStartUserTimeDays'] = df_0.startUserTimeDays + df_0.deltaNextDays

        # train/test split, stratify by churn
        train_i, test_i = self.train_i, self.test_i = next(StratifiedShuffleSplit(test_size=.2, random_state=42).split(churned, churned.values))
        train_df_unscaled = self.train_df_unscaled = df_0[df_0.customerId.isin(churned.index[train_i])]
        test_df_unscaled = self.test_df_unscaled = df_0[df_0.customerId.isin(churned.index[test_i])]

        train_features = self.train_features = sorted(list(set(df_0.columns) - set(['customerId','startUserTime', 'startUserDate'])))
        target_features = self.target_features = ['nextStartUserTimeDays', 'deltaNextDays', 'nextStartUserTimeHours', 'deltaNextHours',
        'churned', 'logDeltaNextDays']
        enc_features = [f for f in train_features if f.endswith('_enc')]
        self.deviceEncIndex = train_features.index('device_enc')
        self.devices = [x for x in train_features if x.startswith('device[')]
        self.deviceIndices = list(map(train_features.index, self.devices))

        # scaling
        features_numeric = self.features_numeric = sorted(list(set(df_0.columns) - set(['customerId','startUserTime', 'startUserDate',
        'device_enc', 'device', 'hourOfDay_enc', 'dayOfMonth_enc', 'dayOfWeek_enc'] + self.devices)))
        train_df_scaled = self.train_df_scaled = train_df_unscaled.copy()
        test_df_scaled = test_df_unscaled.copy()
        scaler = self.scaler = StandardScaler()
        train_df_scaled[features_numeric] = scaler.fit_transform(train_df_unscaled[features_numeric])
        test_df_scaled[features_numeric] = scaler.transform(test_df_unscaled[features_numeric])
        self.train_df_scaled = train_df_scaled
        self.test_df_scaled = test_df_scaled

        # storing feature/target combinations as features for quick access
        # format: predict/mainFeature
        self.presets = {
            'deltaNextDays': {
                'features': sorted(list(set(self.train_features) - ₩
                                set(['deltaNextHours', 'startUserTimeHours',
                                    'deltaNextDays', 'deltaPrevHours', 'churned',
                                    'device_enc', 'device', 'nextStartUserTimeHours',
                                    'nextStartUserTimeDays',
                                    'logDeltaNextDays', 'logDeltaPrevDays'] + enc_features))),
                'target': ['nextStartUserTimeDays', 'deltaNextDays', 'churned'] },

            'deltaNextDays_enc': {
                'features': sorted(list(set(self.train_features) - ₩
                                set(['deltaNextHours', 'startUserTimeHours',
                                    'deltaNextDays', 'deltaPrevHours', 'churned',
                                    'device', 'nextStartUserTimeHours',
                                    'nextStartUserTimeDays', 'hourOfDay', 'dayOfWeek',
                                    'dayOfMonth', 'logDeltaNextDays',
                                    'logDeltaPrevDays'] + self.devices))),
                'target': ['nextStartUserTimeDays', 'deltaNextDays', 'churned'] },

            'logDeltaNextDays_enc': {
                'features': sorted(list(set(self.train_features) - ₩
                                set(['deltaNextHours', 'startUserTimeHours',
                                    'deltaNextDays', 'deltaPrevHours', 'churned',
                                    'device', 'nextStartUserTimeHours',
                                    'nextStartUserTimeDays', 'hourOfDay', 'dayOfWeek',
                                    'dayOfMonth', 'logDeltaNextDays',
                                    'deltaPrevDays'] + self.devices))),
                'target': ['nextStartUserTimeDays', 'logDeltaNextDays', 'churned'] },
```

```
            'nextStartUserTimeDays': {
                'features': sorted(list(set(self.train_features) - ₩
                                set(['deltaNextHours', 'startUserTimeHours',
                                    'deltaNextDays', 'deltaPrevHours', 'churned',
                                    'device_enc', 'device', 'nextStartUserTimeHours',
                                    'nextStartUserTimeDays',
                                    'logDeltaNextDays', 'logDeltaPrevDays'] + enc_features))),
                'target': ['nextStartUserTimeDays', 'deltaNextDays', 'churned'] },

            'nextStartUserTimeDays_enc': {
                'features': sorted(list(set(self.train_features) - ₩
                                set(['deltaNextHours', 'startUserTimeHours',
                                    'deltaNextDays', 'deltaPrevHours', 'churned',
                                    'device', 'nextStartUserTimeHours',
                                    'nextStartUserTimeDays', 'hourOfDay', 'dayOfWeek',
                                    'dayOfMonth', 'logDeltaNextDays',
                                    'logDeltaPrevDays'] + self.devices))),
                'target': ['nextStartUserTimeDays', 'deltaNextDays', 'churned'] }}

        for preset in self.presets:
            self.presets[preset]['feature_indices'] = list(map(self.train_features.index, self.presets[preset]['features']))
            self.presets[preset]['target_indices'] = list(map(self.target_features.index, self.presets[preset]['target']))

        # convert to array
        self.x_train, self.y_train = _df_to_xy_array(train_df_scaled, train_features, target_features)
        self.x_train_unscaled, self.y_train_unscaled = _df_to_xy_array(train_df_unscaled, train_features, target_features)
        self.x_test, self.y_test = _df_to_xy_array(test_df_scaled, train_features, target_features)
        self.x_test_unscaled, self.y_test_unscaled = _df_to_xy_array(test_df_unscaled, train_features, target_features)

        # store customer ids
        train_cust = self.train_cust = self.y_train.index.values
        test_cust = self.test_cust = self.y_test.index.values

        with open(self.PATH+'rmtpp_data.pkl', 'wb') as handle:
            pickle.dump(self, handle, protocol=pickle.HIGHEST_PROTOCOL)


def _pad_x(x, n):
    return np.array(list(map(lambda _x: np.pad(_x[-n:], ((max(n-len(_x),0),0),(0,0)), 'constant'), x.values)))

def _pad_y(x, n):
    return np.array(list(map(lambda _x: np.pad(_x[-n:], (max(n-len(_x),0), 0), 'constant'), x.values)))

def _df_to_xy_array(df, train_features, target_features):
    grouped = df.groupby('customerId')
    x = grouped.apply(lambda g: g[train_features].as_matrix())
    y = grouped.apply(lambda g: g[target_features].as_matrix())

    return x, y
```