```python
import os.path

import numpy as np
import pandas as pd
from operator import mul
from functools import reduce, partial

from scipy.integrate import trapz

from keras.models import Sequential, Model
from keras.layers import Dense, Dropout, Activation, Input, Masking, concatenate, Embedding, RepeatVector, Reshape
from keras.layers.recurrent import LSTM, GRU
from keras.callbacks import Callback, LambdaCallback, TensorBoard, ReduceLROnPlateau, EarlyStopping, ModelCheckpoint
from keras.optimizers import Adam, RMSprop
from keras.initializers import Constant, Zeros
from keras.constraints import non_neg, unit_norm, max_norm
from keras import regularizers
from keras import backend as K

from sklearn.model_selection import train_test_split, StratifiedShuffleSplit
from sklearn.metrics import mean_squared_error, accuracy_score, recall_score, roc_auc_score
from sklearn.decomposition import PCA
from lifelines.utils import concordance_index
from sklearn.gaussian_process.kernels import Matern, ConstantKernel
from bayes_opt import BayesianOptimization

from rmtpp_data import *

import sys
sys.path.insert(0, '../utils')
from plot_format import *
from seaborn import apionly as sns
from adjustText import adjust_text

seed = 42
np.random.seed(seed)

class Rmtpp:

    time_scale = .1

    def __init__(self, name, run, hidden_neurons=32, dense_neurons=32, n_sessions=32, w_scale=.15, predict_sequence=False):
        '''
        :name: used to label model on tensorboard
        :run: run number to identify model on tensorboard
        :n_sessions: number of sessions to include (at most)

        To run
        - initialise this class
        - run set_x_y
        - run set_model
        - run fit_model
        - run get_scores to evaluate
        '''

        self.w_scale = w_scale
        self.predict_sequence = predict_sequence
        self.hidden_neurons = hidden_neurons
        self.dense_neurons = dense_neurons
        self.n_sessions = n_sessions
        self.data = RmtppData.instance()
        self.set_x_y(n_sessions=n_sessions)
        self.set_model()
        self.name = '{:02d}_{}_hiddenNr{}_dnsnr{}_nsess{}_ws{}_ts_{}'.format(run, name, hidden_neurons, dense_neurons,
        n_sessions, self.w_scale, self.time_scale)
        self.run = run
        self.best_model_cp_file = '../../results/rmtpp_new/{}.hdf5'.format(self.name)
        self.best_model_cp = ModelCheckpoint(self.best_model_cp_file, monitor="val_loss",
                                             save_best_only=True, save_weights_only=False)
        self.embeddings=['device', 'dayOfMonth', 'dayOfWeek', 'hourOfDay']
        self.embeddings_layer_names = [e+'_emb' for e in self.embeddings]
        self.embeddings_metadata={'/home/georg/Workspace/fy_project/code/rnn/{}_metadata.tsv'.format(e) for e in self.
        embeddings}

    def set_x_y(self, min_n_sessions=0, n_sessions=100, preset='deltaNextDays_enc'):
        '''
        :preset: target value setting (presets defined in rmtpp_data). 'deltaNextDays_enc' sets target to return time in days with
        encoded values (opposed to one-hot) for all categorical features
        initialises train and test data
        '''
        self.x_train, ₩
        self.x_test, ₩
        self.x_train_unscaled, ₩
        self.x_test_unscaled, ₩
        self.y_train, ₩
        self.y_test, ₩
        self.features, ₩
```

```
        self.targets = self.data.get_xy(min_n_sessions=min_n_sessions, n_sessions=n_sessions, preset=preset, target_sequences=self
        .predict_sequence)

        if self.predict_sequence:
            self.y_train_churned = self.y_train[:,-1,self.targets.index('churned')].astype('bool')
            self.y_test_churned = self.y_test[:,-1,self.targets.index('churned')].astype('bool')
        else:
            self.y_train_churned = self.y_train[:,self.targets.index('churned')].astype('bool')
            self.y_test_churned = self.y_test[:,self.targets.index('churned')].astype('bool')

        train_train_i, train_val_i = self.train_i, self.test_i = next(StratifiedShuffleSplit(test_size=.2, random_state=42).split(self.x_train, self
        .y_train_churned))

        self.device_index = self.features.index('device_enc')
        self.dayOfMonth_index = self.features.index('dayOfMonth_enc')
        self.dayOfWeek_index = self.features.index('dayOfWeek_enc')
        self.hourOfDay_index = self.features.index('hourOfDay_enc')
        self.num_features = list(set(self.features) - set([self.device_index, self.dayOfMonth_index, self.dayOfWeek_index, self.
        hourOfDay_index]))
        self.num_indices = list(map(self.features.index, self.num_features))
        self.startTimeDaysIndex = self.features.index('startUserTimeDays')

        used_feature_indices = list(range(len(self.features)))

        self.x_train = self.x_train[:,:,used_feature_indices].astype('float32')
        self.x_test = self.x_test[:,:,used_feature_indices].astype('float32')
        self.x_train_train = self.x_train[train_train_i]
        self.x_train_val = self.x_train[train_val_i]
        self.x_train_train_unscaled = self.x_train_unscaled[train_train_i]
        self.x_train_val_unscaled = self.x_train_unscaled[train_val_i]

        self.y_train_train = self.y_train.T[[1,2]].T.astype('float32')[train_train_i]
        self.y_train_val = self.y_train.T[[1,2]].T.astype('float32')[train_val_i]

        if self.predict_sequence:
            self.y_train_train[:,:,0] *= self.time_scale
            self.y_train_val[:,:,0] *= self.time_scale
        else:
            self.y_train_train[:,0] *= self.time_scale
            self.y_train_val[:,0] *= self.time_scale

        self.y_train_train_churned = self.y_train_churned[train_train_i]
        self.y_train_val_churned = self.y_train_churned[train_val_i]

        self.x_train_train_ret = self.x_train_train[~self.y_train_train_churned]
        self.x_train_val_ret = self.x_train_val[~self.y_train_val_churned]
        self.y_train_train_ret = self.y_train_train[~self.y_train_train_churned]
        self.y_train_val_ret = self.y_train_val[~self.y_train_val_churned]

        self.y_train = self.y_train.T[[1,2]].T.astype('float32')
        self.y_test = self.y_test.T[[1,2]].T.astype('float32')

        if self.predict_sequence:
            self.y_train[:,:,0] *= self.time_scale
            self.y_test[:,:,0] *= self.time_scale
        else:
            self.y_train[:,0] *= self.time_scale
            self.y_test[:,0] *= self.time_scale

        # if self.predict_sequence:
        #     self.y_train_train = self.y_train_train.reshape(self.y_train_train.shape+(1,))
        #     self.y_train_val = self.y_train_val.reshape(self.y_train_val.shape+(1,))
        #     self.y_train_train_ret = self.y_train_train_ret.reshape(self.y_train_train_ret.shape+(1,))
        #     self.y_train_val_ret = self.y_train_val_ret.reshape(self.y_train_val_ret.shape+(1,))
        #     self.y_train = self.y_train.reshape(self.y_train.shape+(1,))
        #     self.y_test = self.y_test.reshape(self.y_test.shape+(1,))


    def load_best_weights(self):
        self.model.load_weights(self.best_model_cp_file)


    def set_model(self, lr=.001):
        '''
        defines the model used
        '''
        self.lr = lr
        len_seq = self.x_train.shape[1]
        num_num_features = len(self.num_features)
        num_devices = int(self.x_train[:,:,self.device_index].max())+1
        num_dayOfMonths = int(self.x_train[:,:,self.dayOfMonth_index].max()) + 1
        num_dayOfWeeks = int(self.x_train[:,:,self.dayOfWeek_index].max()) + 1
        num_hourOfDays = int(self.x_train[:,:,self.hourOfDay_index].max()) + 1

        lstm_neurons = self.hidden_neurons

        # embedding layers
```

```
        device_input = Input(shape=(len_seq,), dtype='int32', name='device_input')
        device_embedding = Embedding(output_dim=1, input_dim=num_devices, name='device_emb',
                                     input_length=len_seq, mask_zero=True,
                                     embeddings_constraint=unit_norm())(device_input)
        # dayOfMonth_input = Input(shape=(len_seq,), dtype='int32', name='dayOfMonth_input')
        # dayOfMonth_embedding = Embedding(output_dim=5, input_dim=num_dayOfMonths,
        #                                  input_length=len_seq, mask_zero=True,
        #                                  embeddings_constraint=unit_norm(),
        #                                  name='dayOfMonth_emb')(dayOfMonth_input)
        dayOfWeek_input = Input(shape=(len_seq,), dtype='int32', name='dayOfWeek_input')
        dayOfWeek_embedding = Embedding(output_dim=2, input_dim=num_dayOfWeeks,
                                        name='dayOfWeek_emb', embeddings_constraint=unit_norm(),
                                        input_length=len_seq, mask_zero=True)(dayOfWeek_input)
        hourOfDay_input = Input(shape=(len_seq,), dtype='int32', name='hourOfDay_input')
        hourOfDay_embedding = Embedding(output_dim=4, input_dim=num_hourOfDays,
                                        name='hourOfDay_emb', embeddings_constraint=unit_norm(),
                                        input_length=len_seq, mask_zero=True)(hourOfDay_input)

        # inputs for numerical features
        num_input = Input(shape=(len_seq, num_num_features), name='num_input')

        num_masking = Masking(mask_value=0.)(num_input)

        merge_inputs = concatenate([device_embedding, #dayOfMonth_embedding,
                                    dayOfWeek_embedding, hourOfDay_embedding,
                                    num_masking])

        # preprocessing layer
        merge_inputs = Dense(self.dense_neurons, activation='tanh')(merge_inputs)
        merge_inputs = Dropout(.2)(merge_inputs)

        # lstm_output = GRU(lstm_neurons,
        lstm_output = LSTM(lstm_neurons,
                           activation='tanh',
                           return_sequences=self.predict_sequence,
                           # kernel_regularizer=regularizers.l2(0.03),
                           # kernel_regularizer=max_norm(),
                           dropout=.2,
                           # activity_regularizer=max_norm()
                           )(merge_inputs)

        # lstm_output = Dense(32, activation='tanh')(lstm_output)
        # lstm_output = Dropout(.2)(lstm_output)

        predictions = Dense(1,
                            activation='linear',
                            name='predictions',
                            # bias_constraint=max_norm(0)
                            # bias_constraint=max_norm(0)
                            # kernel_regularizer=regularizers.l2(0.03)
                            )(lstm_output)

        model = Model(inputs=[device_input, dayOfWeek_input, hourOfDay_input, num_input], outputs=predictions)

        loss = self.neg_log_likelihood_cens_seq if self.predict_sequence else self.neg_log_likelihood_cens
        model.compile(loss=loss, optimizer=RMSprop(lr=lr))

        self.model = model
        return model


    def fit_model(self, initial_epoch=0):
        '''
        train the model

        :initail epoch: set if we continue training from specific epoch (to show up correctly on tensorboard)
        '''
        log_file = '{}_lr{}_inp{}'.format(self.name, self.lr, self.x_train.shape[2])

        self.model.fit([self.x_train[:,:,self.device_index].astype('int32'),
                        # self.x_train[:,:,self.dayOfMonth_index].astype('int32'),
                        self.x_train[:,:,self.dayOfWeek_index].astype('int32'),
                        self.x_train[:,:,self.hourOfDay_index].astype('int32'),
                        self.x_train[:,:,self.num_indices]],
                       self.y_train,
                       batch_size=1024,
                       epochs=2000,
                       validation_split=0.2,
                       verbose=0,
                       initial_epoch=initial_epoch,
                       callbacks=[TensorBoard(log_dir='../../logs/rmtpp_new/{}'.format(log_file),
                                              embeddings_freq=100,
                                              embeddings_layer_names=self.embeddings_layer_names,
                                              embeddings_metadata=self.embeddings_metadata,
                                              histogram_freq=100),
                                  EarlyStopping(monitor='val_loss',
                                                min_delta=0,
```

```
                                             patience=100,
                                             verbose=1,
                                             mode='auto'),
                                self.best_model_cp])


    def neg_log_likelihood(self, targets, output):
        """ Loss function for RMTPP model

        :targets: vector of: [t_(j+1) - t_j, mask]
        :output: rnn output = v_t * h_j + b_t
        """

        w = self.w_scale
        w_t = w
        cur_state = K.batch_flatten(output)
        delta_t = K.batch_flatten(targets)

        res = -cur_state - w_t*delta_t ₩
              - (1/w)*K.exp(cur_state) ₩
              + (1/w)*K.exp(cur_state + w_t*(delta_t))

        # return res
        return res


    def neg_log_likelihood_cens(self, targets, output):
        """ Loss function for RMTPP model

        :targets: vector of: [t_(j+1) - t_j, mask]
        :output: rnn output = v_t * h_j + b_t
        """
        ret_mask = K.batch_flatten(K.cast(K.equal(targets[:,1], 0), 'float32'))
        delta_t = K.batch_flatten(targets[:,0])
        w = self.w_scale
        w_t = w

        cur_state = K.batch_flatten(output)

        ret_term = -cur_state - w_t*delta_t
        ret_term = ret_mask * ret_term
        common_term = -(1/w)*K.exp(cur_state) + (1/w)*K.exp(cur_state + w_t*(delta_t))

        return ret_term + common_term

    def neg_log_likelihood_cens_seq(self, targets, output):
        """ Loss function for RMTPP model

        :targets: vector of: [t_(j+1) - t_j, mask]
        :output: rnn output = v_t * h_j + b_t
        """
        ret_mask = K.batch_flatten(K.cast(K.equal(targets[:,:,1], 0), 'float32'))
        delta_t = K.batch_flatten(targets[:,:,0])
        w = self.w_scale
        w_t = w

        cur_state = K.batch_flatten(output)

        ret_term = -cur_state - w_t*delta_t
        ret_term = ret_mask * ret_term
        common_term = -(1/w)*K.exp(cur_state) + (1/w)*K.exp(cur_state + w_t*(delta_t))

        return ret_term + common_term

    def neg_log_likelihood_seq(self, targets, output):
        """ Loss function for RMTPP model

        :targets: vector of: [t_(j+1) - t_j, mask]
        :output: rnn output = v_t * h_j + b_t
        """

        # w_t = self.w_scale
        mask = K.batch_flatten(targets[:,:,1])
        # w = K.batch_flatten(output[:,:,1])
        w = self.w_scale
        w_t = w
        # cur_state = K.batch_flatten(output[:,:,0])
        cur_state = K.batch_flatten(output)
        delta_t = K.batch_flatten(targets[:,:,0])

        res = -cur_state - w_t*delta_t ₩
              - (1/w)*K.exp(cur_state) ₩
              + (1/w)*K.exp(cur_state + w_t*(delta_t))

        # return res
        return res*mask
```

```python
    def get_predictions(self, dataset='val', include_recency=False):
        if include_recency:
            pred_next_starttime_vec = np.vectorize(self.pred_next_starttime_rec)
        else:
            pred_next_starttime_vec = np.vectorize(self.pred_next_starttime)

        if dataset=='test':
            x = [self.x_test[:,:,self.device_index].astype('int32'),
                 self.x_test[:,:,self.dayOfWeek_index].astype('int32'),
                 self.x_test[:,:,self.hourOfDay_index].astype('int32'),
                 self.x_test[:,:,self.num_indices]]
            y = self.y_test
            t_js = self.x_test_unscaled[:,:,self.startTimeDaysIndex]
        else:
            x = [self.x_train_val[:,:,self.device_index].astype('int32'),
                 # self.x_train_val[:,:,self.dayOfMonth_index].astype('int32'),
                 self.x_train_val[:,:,self.dayOfWeek_index].astype('int32'),
                 self.x_train_val[:,:,self.hourOfDay_index].astype('int32'),
                 self.x_train_val[:,:,self.num_indices]]
            y = self.y_train_val
            t_js = self.x_train_val_unscaled[:,:,self.startTimeDaysIndex]

        if not self.predict_sequence:
            t_js = t_js[:,-1]

        # t_js = np.log(t_js.astype('float') + 1) * self.time_scale
        t_js = t_js * self.time_scale

        pred = self.model.predict(x)
        cur_states = pred.reshape(pred.shape[:-1])
        t_pred = pred_next_starttime_vec(cur_states, t_js)

        if self.predict_sequence:
            t_true = y[:,:,0]
        else:
            t_true = y[:,0]


        # t_pred = np.exp(t_pred/self.time_scale) - 1
        # t_true = np.exp(t_true/self.time_scale) - 1
        # return t_pred, t_true
        return t_pred/self.time_scale, t_true/self.time_scale


    def pred_next_starttime(self, cur_state, t_j):
        ts = np.arange(t_j, 1000*self.time_scale, self.time_scale)
        delta_ts = ts - t_j
        samples = self._pt(delta_ts, cur_state)
        # samples = delta_ts * self._pt(delta_ts, cur_state)

        return trapz(samples, ts)


    def pred_next_starttime_rec(self, cur_state, t_j):
        absence_time = 365*self.time_scale - t_j
        s_ts = self._pt(absence_time, cur_state)

        ts = np.arange(t_j, 1000*self.time_scale, self.time_scale)
        delta_ts = ts - t_j
        samples = self._pt(delta_ts, cur_state)

        return (1/s_ts) * trapz(samples[ts>(365*self.time_scale)], ts[ts>(365*self.time_scale)]) + trapz(samples[ts<=(365*self.
        time_scale)], ts[ts<=(365*self.time_scale)])


    def _pt(self, delta_t, cur_state):
        w_t = self.w_scale
        w = self.w_scale
        return np.exp((1/w)*np.exp(cur_state) - (1/w)*np.exp(cur_state + w_t*(delta_t)))
        # w_t = self.w_scale
        # w = self.w_scale
        # return delta_t * np.exp(-(-cur_state - w_t*delta_t ₩
        #                        - (1/w)*np.exp(cur_state) ₩
        #                        + (1/w)*np.exp(cur_state + w_t*(delta_t))))

    def get_scores(self, dataset='val', include_recency=False):
        if dataset=='val':
            churned = self.y_train_val_churned
            unscaled = self.x_train_val_unscaled
        else:
            churned = self.y_test_churned
            unscaled = self.x_test_unscaled

        pred_0, y_0 = self.get_predictions(dataset, include_recency)
        # return pred_0, y_0
```

```
        if self.predict_sequence:
            mask = y_0 != 0
            churned_mask = mask[~churned]
            pred_last = pred_0[:,-1].ravel()
            y_last = y_0[:,-1].ravel()
        else:
            pred_last = pred_0.ravel()
            y_last = y_0.ravel()

        # return pred_0, mask
        rmse_days = np.sqrt(mean_squared_error(pred_last[~churned], y_last[~churned]))

        if self.predict_sequence:
            rmse_days_all = np.sqrt(mean_squared_error(pred_0[~churned][churned_mask].ravel(), y_0[~churned][churned_mask].
            ravel()))
        else:
            rmse_days_all = 0

        rtd_ind = self.features.index('startUserTimeDays')

        ret_time_days_pred = unscaled[:,-1,rtd_ind] + pred_last
        ret_time_days_true = unscaled[:,-1,rtd_ind] + y_last


        churned_pred = ret_time_days_pred >= churn_days
        churned_true = ret_time_days_true >= churn_days

        churn_acc = accuracy_score(churned_true, churned_pred)
        churn_recall = recall_score(churned_true, churned_pred)
        churn_auc = roc_auc_score(churned_true, pred_last)

        concordance = concordance_index(y_last, pred_last, ~churned)

        return {'rmse_days': rmse_days,
                'rmse_days_all': rmse_days_all,
                'churn_acc': churn_acc,
                'churn_auc': churn_auc,
                'churn_recall': churn_recall,
                'concordance': concordance}


def runBayesOpt():
    RESULT_PATH = '../../results/rmtpp_new/bayes_opt/'

    # bounds = {'hidden_neurons': (1, 100), 'dense_neurons': (1,100)}
    bounds = {'w_scale': (.0001, 1.)}
    n_iter = 20

    bOpt = BayesianOptimization(_evaluatePerformance, bounds)

    # bOpt.initialize({'targets': [-2762.34752, -6820.47174, -5671.95848, -4458.36495],
    #                  'w_scale': [0.3746, 0.9507, 0.7320, 0.5987]})
    # bOpt.initialize({'w_scale': np.array([0.3746, 0.9507, 0.7320, 0.5987, 0.0001, 0.5362, 0.6414, 0.8423, 0.8101, 0.8728, 0.8289,
    0.1884, 1.0000, 0.4590, 0.7801, 0.2844, 0.7915, 0.7659, 0.0978, 0.6882, 0.4983, 0.5691, 0.4209, 0.8553]),
    #                  'targets': -np.array([64.694491602844849,
    #                                        96.325984867170703,
    #                                        94.282091001473148,
    #                                        79.028932108368451,
    #                                        66.328999345455429,
    #                                        80.06554529345641,
    #                                        86.673739067258424,
    #                                        96.070839922337683,
    #                                        91.61875460161319,
    #                                        95.183281119114056,
    #                                        91.719864616161928,
    #                                        59.024560678731802,
    #                                        99.498293832305706,
    #                                        76.732316053755667,
    #                                        92.14056326130931,
    #                                        57.639273218402288,
    #                                        92.137971389647902,
    #                                        90.788065365827876,
    #                                        67.18060379333096,
    #                                        89.842938462274176,
    #                                        81.532174387971722,
    #                                        86.448177020462524,
    #                                        69.130375113932558,
    #                                        97.745466472895842])/100})

    bOpt.maximize(init_points=4, n_iter=n_iter)

    with open(RESULT_PATH+'bayes_opt_w_scale_ret_noseq_rmse_32_32_32.pkl', 'wb') as handle:
        pickle.dump(bOpt, handle, protocol=pickle.HIGHEST_PROTOCOL)

    return bOpt

def _evaluatePerformance(w_scale):
```

```
    # def __init__(self, name, run, hidden_neurons=32, n_sessions=100):
    K.clear_session()
    # hidden_neurons = np.floor(hidden_neurons).astype('int')
    # dense_neurons = np.floor(dense_neurons).astype('int')
    # print('hidden_neurons: {}, dense_neurons: {}'.format(hidden_neurons, dense_neurons))
    print('w_scale: {}'.format(w_scale))
    model = Rmtpp('bayes_opt', 39, w_scale=w_scale, predict_sequence=False)
    model.fit_model()
    model.load_best_weights()
    scores = model.get_scores()
    print(scores)
    return -scores['rmse_days']/100


predPeriod = {
    'start': pd.Timestamp('2016-02-01'),
    'end': pd.Timestamp('2016-06-01')
}
obsPeriod = {
    'start': pd.Timestamp('2015-02-01'),
    'end': pd.Timestamp('2016-02-01')
}
predPeriodHours = (predPeriod['end'] - predPeriod['start']) / np.timedelta64(1, 'h')
hours_year = np.timedelta64(pd.datetime(2017,2,1) - pd.datetime(2016,2,1)) / np.timedelta64(1,'h')
churn_days = (predPeriod['end'] - obsPeriod['start']) / np.timedelta64(24, 'h')


def showResidPlot_short_date(model, y_pred, dataset='val', width=1, height=None):
    startUserTimeDaysCol = model.features.index('startUserTimeDays')
    churned = model.y_train_val_churned

    df = pd.DataFrame()
    df['predicted (days)'] = y_pred[~churned]
    df['actual (days)'] = model.y_train_val[~churned, 0] / model.time_scale
    df['daysInObs'] = model.x_train_val_unscaled[~churned, -1, startUserTimeDaysCol] + df['actual (days)']
    df['date'] = df['daysInObs'] * np.timedelta64(24,'h') + obsPeriod['start']
    df['residual (days)'] = df['predicted (days)'] - df['actual (days)']

    grid = sns.JointGrid('daysInObs', 'residual (days)', data=df, size=figsize(.5,.5)[0], xlim=(0,3000), ylim=(-110,110))
    grid = grid.plot_marginals(sns.distplot, kde=False, color='k')#, shade=True)
    grid = grid.plot_joint(plt.scatter, alpha=.1, s=6, lw=0)
    grid.ax_joint.clear()

    retUnc = grid.ax_joint.scatter(df['daysInObs'], df['residual (days)'], alpha=.1, s=6, lw=0, color='C0', label='Ret. user (uncens.)')

    xDates = [pd.datetime(2016,i,1) for i in [2,4,6]]
    xDatesHours = [(d - obsPeriod['start']).to_timedelta64()/np.timedelta64(24,'h') for d in xDates]
    xDatesStr = [d.strftime('%Y-%m') for d in xDates]
    grid.ax_joint.set_xticks(xDatesHours)
    grid.ax_joint.set_xticklabels(xDatesStr)
    grid.ax_joint.set_xlabel('actual return date')
    grid.ax_joint.set_ylabel('residual (days)')

    grid.ax_joint.set_ylim((-200,200))
    plt.show()


def showResidPlot_short_days(model, y_pred, width=1, height=None):
    startUserTimeDaysCol = model.features.index('startUserTimeDays')
    churned = model.y_train_val_churned

    df = pd.DataFrame()
    df['predicted (days)'] = y_pred[~churned]
    df['actual (days)'] = model.y_train_val[~churned, 0] / model.time_scale
    df['daysInObs'] = model.x_train_val_unscaled[~churned, -1, startUserTimeDaysCol] + df['actual (days)']
    df['date'] = df['daysInObs'] * np.timedelta64(24,'h') + obsPeriod['start']
    df['residual (days)'] = df['predicted (days)'] - df['actual (days)']

    grid = sns.JointGrid('actual (days)', 'residual (days)', data=df, size=figsize(.5,.5)[0], xlim=(0,3000), ylim=(-110,110))
    grid = grid.plot_marginals(sns.distplot, kde=False, color='k')#, shade=True)
    grid = grid.plot_joint(plt.scatter, alpha=.1, s=6, lw=0)
    grid.ax_joint.clear()

    retUnc = grid.ax_joint.scatter(df['actual (days)'], df['residual (days)'], alpha=.1, s=6, lw=0, color='C0', label='Ret. user (uncens.)')

    grid.ax_joint.set_xlabel('actual return time (days)')
    grid.ax_joint.set_ylabel('residual (days)')
    grid.ax_joint.set_ylim((-110,110))

    plt.show()


def get_embeddings(r, embedding):
    layer = r.model.get_layer('{}_emb'.format(embedding))
    labels = pd.DataFrame.from_csv('../rnn/{}_metadata.tsv'.format(embedding), sep='₩t').index
    n = len(labels)
    values = K.eval(layer.call(np.arange(0,n)))
```

```
    return values

def calc_explained_variance(r, embedding):
    embeddings = get_embeddings(r, embedding)
    d = embeddings.shape[-1]
    var = np.diag(np.cov(embeddings.T).reshape((d,d))).sum()
    pca_vars = np.array(list(map(
        lambda n: np.diag(np.cov(PCA(n_components=n).fit_transform(embeddings).T).reshape((n,n))).sum(),
        np.arange(1, embeddings.shape[-1]+1))))
    return pca_vars / var

def plot_embeddings(r, embedding, width=1, height=None):
    labels = pd.DataFrame.from_csv('../rnn/{}_metadata.tsv'.format(embedding), sep='₩t').index
    values = get_embeddings(r, embedding)
    n_dims = values.shape[-1]

    if n_dims > 2:
        flat = PCA(n_components=2).fit_transform(values)
        explained_var  = calc_explained_variance(r, embedding)[1]
    elif n_dims == 1:
        flat = np.concatenate((values, np.zeros(values.shape)), 1)
        explained_var = 1
    else:
        flat = values
        explained_var = 1

    fig, ax = newfig(width, height)
    ax.scatter(flat[:,0], flat[:,1], s=2)

    texts = []
    for i,lbl in enumerate(labels):
        texts.append(ax.text(flat[i,0], flat[i,1], lbl, size=6))

    adjust_text(texts, lim=1000, arrowprops=dict(arrowstyle="-", color='k', lw=0.5))

    if n_dims==1:
        ax.set_yticks([])
    # elif n_dims > 2:
    #     ax.text(0.01, 0.01,
    #             'Reduced from {} to 2 dim. through PCA; total variance described: {:.1f}₩%'.format(n_dims, explained_var*100),
    #             verticalalignment='bottom', horizontalalignment='left', transform=ax.transAxes,
    #             fontsize=6)

    fig.tight_layout()
    fig.show()
```