```
from tensorflow.random import set_seed
import numpy as np

set_seed(42)
np.random.seed(42)

import statsmodels.api as sm
from statsmodels.tsa.statespace.tools import diff
from statsmodels.tsa.stattools import acovf, acf, pacf, pacf_yw, pacf_ols
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import adfuller
from statsmodels.tools.eval_measures import mse,rmse
from pmdarima import auto_arima
import pandas as pd
from datetime import date
from datetime import datetime, timedelta
import pandas_datareader as pdr
import holidays

import matplotlib.pyplot as plt
from statsmodels.tsa.ar_model import AR, ARResults
from statsmodels.tsa.arima_model import ARMA, ARIMA, ARMAResults, ARIMAResults
from statsmodels.tsa.seasonal import seasonal_decompose as sd
from statsmodels.tsa.statespace.varmax import VARMAX, VARMAXResults
from statsmodels.tsa.statespace.sarimax import SARIMAX
from matplotlib.lines import Line2D

from sklearn.preprocessing import MinMaxScaler, StandardScaler
from keras.preprocessing.sequence import TimeseriesGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import LSTM

from pandas.tseries.holiday import USFederalHolidayCalendar as calendar

def adf_test(series, title=''):
    """
    Pass in a time series and an optional title, returns an ADF report
    series: series of data
        type: series or array

    returns an augmented dickey fuller test stating whether there is stationarity or not

    """
    print(f'Augmented Dickey-Fuller Test: {title}')
    result = adfuller(series.dropna(),autolag='AIC') # .dropna() handles differenced data

    labels = ['ADF test statistic','p-value','# lags used','# observations']
    out = pd.Series(result[0:4],index=labels)

    for key,val in result[4].items():
        out[f'critical value ({key})']=val

    print(out.to_string(),'\n\n\n')          # .to_string() removes the line "dtype: float64"

    if result[1] <= 0.05:
        print("Reject the null hypothesis",'\n')
        print("Data has no unit root and is stationary")
    else:
        print("Fail to reject the null hypothesis",'\n')
        print("Data has a unit root and is non-stationary")
```

```
def sort_and_clean_df(dataframe, target_columns, percent_data_threshold=1, US_df=False): # sort_df()
    """
    *purpose: Pass in dataframe and threshold percent as a decimal, returns a dataframe based on
that threshold
              - if threshold is not specified the function will use a threshold of 1 and keep all
columns

    *inputs:
    dataframe: dataframe
    target_column: target columns in a list, columns that you specifically want to keep.
        leave threshold at default 1 if you know which specific columns you wish to keep
    percent_data_missing_threshold: desired threshold expressed in decimal form

    *outputs: a new dataframe with no NaN values and desired columns.
    """

    # create dataframe with target_columns passed in list form
    dataframe = dataframe[target_columns]

    # calculate threshold as a percent of dataframe
    threshold_num = len(dataframe)*percent_data_threshold
    dataframe = dataframe.dropna(axis=1,thresh=len(dataframe)-threshold_num)
    dataframe = dataframe.fillna(0)
    dataframe = dataframe.sort_index()

    if US_df == True:
        dataframe.index.freq = 'D'

    return dataframe

def state_dataframe(dataframe, state_postal_code):
    '''
    *Notes: function assumes all state and US data are seasonal on a weekly basis, but can be
specified. if data has no seasonality, use return_arima_order()

    *inputs:
    dataframe: a dataframe of the state Covid data
        type = dataframe
    state_postal_code: state postal code to specify state
        type = str

    *outputs: returns state specific dataframe to work with
    '''
    # create dataframe based on state_postal_code
    dataframe = dataframe[dataframe['state']==state_postal_code]
    dataframe = pd.DataFrame(dataframe)

    # sort index, lowest index to oldest date
    dataframe = dataframe.sort_index()
    dataframe.index.freq = 'D'

    print(f"Successfully returned indexed dataframe for {state_postal_code}")

    return dataframe

def return_arima_order(dataframe, target_column, days_to_forecast=30, train_days=270, m_periods=1,
exogenous_column=None):
    '''
    Notes: returns stepwise_fit wrapper with arima order and seasonal arima orders
```

```
    *inputs:
    dataframe: a dataframe of Covid data
        type = dataframe
    target_column: target column in string format
        type = str
    m_periods: seasonality frequency. freq of DF is 1D, or 1 day intervals. set m to number of days
it takes to complete one cycle of seasonality, or leave to default
        type= int
    train_days: number of days you wish to train on
        type = int
    seasonal: if the data appears to be seasonal, set seasonal to True
        type = bool
    *exogenous_column: name of exogenous column data
        type = str

    *outputs: returns arima order and seasonal arima order
    '''
    # number of days to train on
    train_days = -train_days

    # specify seasonal data mased on m_periods value
    if m_periods >= 2:
        seasonal=True
    else:
        seasonal=False

    # create training data
    ts = dataframe.iloc[-train_days:-days_to_forecast][target_column]
    full_ts = dataframe.iloc[-train_days:][target_column]

    # run auto arima, seasonal determined by m_periods entered
    # stepwise_fit =
auto_arima(ts,start_P=0,start_Q=0,start_p=0,start_q=0,max_p=10,max_q=10,seasonal=seasonal,
method='lbfgs', n_jobs=-1,stepwise=True,m=m_periods)
    stepwise_full =
auto_arima(full_ts,start_P=0,start_Q=0,start_p=0,start_q=0,max_p=10,max_q=10,seasonal=seasonal,
method='lbfgs', n_jobs=-1,stepwise=True,m=m_periods)

    # print("ARIMA order is: ", stepwise_fit.order)

    # if seasonal is not None:
    #     print("Seasonal ARIMA order is: ", stepwise_fit.seasonal_order)
    # else:
    #     pass
    # print("Use ARIMA object stepwise_fit to store ARIMA and seasonal ARIMA orders in variables.")

    return stepwise_full

def arima_tune(dataframe, target_column, days_to_forecast=30, train_days=270, m_periods=1,
exogenous_column=None, verbose=False):
    '''
    Notes: function assumes all state and US data are seasonal on a weekly basis, but can be set to
None if the state data does not
    appear to be seasonal. Additionally, auto_ARIMA is calculating based on a trailing 6 month
period.

    *inputs:
    dataframe: a dataframe of Covid data
        type = dataframe
    target_column: target column in string format
        type = str
```

```
    days_to_forecast: number of days into the future you wish to forecast
        type = int
    train_days: number of days to train auto_arima on
        type = int
    m_periods: seasonality frequency (set m_periods to number of days per season since
index.freq='D'
        type = int
    seasonal: if the data appears to be seasonal, set seasonal to True
        type = bool
    *exogenous_column: name of exogenous column data
        type = str
    *verbose: bool, will return summary and qq plot if set to true

    *outputs: returns arima order and seasonal arima order and a corresponding model
    '''
    # create model fit, see summary
    if m_periods >= 2:
        seasonal=True
    else:
        seasonal=False

    # create training data
    ts = dataframe.iloc[-train_days:-days_to_forecast][target_column]
    full_ts = dataframe.iloc[-train_days:][target_column]

    if exogenous_column is None:
        exog=None
    else:
        exog=dataframe[exogenous_column]

    # run auto arima, seasonal determined by m_periods entered
    stepwise_fit =
auto_arima(ts,X=exog,start_P=0,start_Q=0,start_p=0,start_q=0,max_p=10,max_q=10,seasonal=seasonal,
method='lbfgs', n_jobs=-1,stepwise=True,m=m_periods)
    stepwise_fit_2 =
auto_arima(full_ts,X=exog,start_P=0,start_Q=0,start_p=0,start_q=0,max_p=10,max_q=10,seasonal=seasonal,
 method='lbfgs', n_jobs=-1,stepwise=True,m=m_periods)

    arima_order = stepwise_fit.order
    sarima_order = stepwise_fit.seasonal_order

    # print notes on arima order and sarima order
    print("ARIMA order is: ", stepwise_fit.order)
    if seasonal is not None:
        print("Seasonal ARIMA order is: ", stepwise_fit.seasonal_order)
    else:
        pass
    # further instructions
    print("Use ARIMA object stepwise_fit to store ARIMA and seasonal ARIMA orders in variables.")

    # create length variable and then train data. test data is not necessary here, but is defined
    # length = len(dataframe)-days_to_forecast
    # train_data = dataframe.iloc[:length]
    # test_data = dataframe.iloc[length:]

    # training data model
    if exogenous_column is None:
        model = SARIMAX(ts, order=stepwise_fit.order, seasonal_order=stepwise_fit.seasonal_order,
m=m_periods, enforce_invertibility=False, enforce_stationarity=False)
    else:
        model = SARIMAX(ts, exogenous=train_data[exogenous_column], order=stepwise_fit.order,
```

```
seasonal_order=stepwise_fit.seasonal_order, m=m_periods, enforce_invertibility=False,
enforce_stationarity=False)

    # full data model
    if exogenous_column is None:
        full_model = SARIMAX(full_ts, order=stepwise_fit.order,
seasonal_order=stepwise_fit_2.seasonal_order, m=m_periods, enforce_invertibility=False,
enforce_stationarity=False)
    else:
        full_model = SARIMAX(full_ts, exogenous=dataframe[exogenous_column],
order=stepwise_fit_2.order, seasonal_order=stepwise_fit.seasonal_order, m=m_periods,
enforce_invertibility=False, enforce_stationarity=False)
    results_full = full_model.fit()

    # instantiate fit model for train_data
    results = model.fit()

    if verbose:
        display(results.summary())
        results.plot_diagnostics()

    # return stepwise_fit and results (actual model)
    return stepwise_fit, stepwise_fit_2, results, results_full

def evaluate_predictions(model, dataframe, target_column, stepwise_fit, alpha, days_to_forecast=30,
train_days=270, exogenous_column=None):
    '''
    #purpose: creates a SARIMA or SARIMAX model based on datetime dataframe with any target column
    must specify arima_order at least, but seasonal_arima_order is optional

    #inputs:
    model: fitted model
    dataframe: a dataframe of the state Covid data
        type = dataframe
    target_column: column to forecast trend
        type = str
    days_to_forecast: number of days into the future you wish to forecast
        type = int
    stepwise_fit = arima order from stepwise_fit.order
        type = wrapper
    alpha: allows to set confidence interval
        type = float less than 1
    *exogenous_column: name of exogenous column data
        type = str

    #outputs: a graphic evaluation of the (actual) test vs predicted values of the model
    '''
    # create length variable and then train/test data
    length = train_days
    train_data = dataframe.iloc[-length:-days_to_forecast+1]
    test_data = dataframe.iloc[-days_to_forecast:] # fix to match train data from before

    # variables for start and end for predictions to evaluate against test data
    start = len(train_data)
    end = len(train_data) + len(test_data) - 1

    if exogenous_column is None:
        exog=None
    else:
        exog=dataframe[exogenous_column]
```

```
    predictions = model.get_prediction(start,end,typ='exogenous',exog=exog)

    # create plot_df for graphing
    upper_lower = predictions.conf_int(alpha=alpha)
    plot_df = predictions.conf_int(alpha=alpha)
    plot_df['Predictions'] = predictions.predicted_mean

    # create graph - {PLOT}
    ax = plot_df['Predictions'].plot(label='Model Prediction', figsize=(16,8))
    # plot_df['Predictions'].plot(ax=ax, label='Confidence Interval')
    train_data.iloc[-days_to_forecast-30:][target_column].plot(label=f'{target_column}');
    test_data[target_column].plot(label='Test Data');
    ax.fill_between(upper_lower.index,
                    upper_lower.iloc[:, 0],
                    upper_lower.iloc[:, 1], color='k', alpha=0.15)
    ax.set_xlabel('Date')
    ax.set_ylabel('Number of People')
    ax.set_title(f'Number of {target_column}')
    plt.legend()
    plt.show();

    return None

def do_not_allow_decrease(series, reached_max_value=None):
    '''
    purpose of function is to keep certain values from falling below zero.
    '''
    max_value=series.max()
    location_of_max=series.argmax()

    for idx in series.index:
        if series[idx]==max_value:
            new_i = max_value
            reached_max_value=1
        elif reached_max_value is not None:
            series[idx] = new_i
        else:
            pass

    return series

def build_SARIMAX_forecast(model, dataframe, target_column, stepwise_fit, alpha,
days_to_forecast=30, original_df=None, exogenous_column=None, state_postal_code=None):
    '''
    #purpose: creates a SARIMA or SARIMAX model based on datetime dataframe with any target column
    must specify arima_order at least, but seasonal_arima_order is optional

    #inputs:
    model: model
    dataframe: a dataframe of the state Covid data
        type = dataframe
    target_column: column to forecast trend
        type = str
    days_to_forecast: number of days into the future you wish to forecast
        type = int
    stepwise_fit = arima order from stepwise_fit.order
        type = tuple
    alpha: allows to set confidence interval
        type = float less than 1
    *exogenous_column: name of exogenous column data
```

```
    #outputs: two object outputs, a fit model called results_forecast and forecast object containing
predictions as well as a forecast graph
    '''
    # create appropriate length start and end for get_prediction below based on whether or not an
exogenous set of data is being used
    if original_df is None:
        start = len(dataframe)
        end = len(dataframe)+days_to_forecast
    elif original_df is not None:
        start = len(original_df)
        end = len(original_df)+days_to_forecast

    # build full dataframe model given exogenous data, or not
    if exogenous_column is None:
        exog=None
    else:
        exog=dataframe.iloc[-days_to_forecast:][exogenous_column]

    # create forecast object
    if exogenous_column is None:
        forecast_object = model.get_forecast(steps=days_to_forecast)
    else:
        forecast_object = model.get_forecast(steps=days_to_forecast, exog=exog)

    # build confidence intervals and predicted mean line in one df
    upper_lower = forecast_object.conf_int(alpha=alpha)
    upper_lower.columns = ['lower','upper']

    # wrote function to customize forecast and remove the possibility for a negative forecast in
deaths when death is the target forecast column
    if target_column == 'death':
        lower = pd.Series(upper_lower['lower'])
        lower = do_not_allow_decrease(lower)
        plot_df = forecast_object.conf_int(alpha=alpha)
        plot_df.columns = ['lower','upper']
        plot_df['lower'] = lower
        plot_df['Forecast'] = forecast_object.predicted_mean
        forecast = forecast_object.predicted_mean
    else:
        plot_df = forecast_object.conf_int(alpha=alpha)
        plot_df.columns = ['lower','upper']
        plot_df['Forecast'] = forecast_object.predicted_mean
        forecast = forecast_object.predicted_mean

    ax = plot_df['Forecast'].plot(label='Forecast', figsize=(16,8))
    dataframe.iloc[-200:][target_column].plot();
    ax.fill_between(upper_lower.index,
                    upper_lower.iloc[:, 0],
                    upper_lower.iloc[:, 1], color='k', alpha=0.15)
    ax.set_xlabel('Date')

    if state_postal_code is None:
        ax.set_ylabel(f'Number of People, {target_column}')
        ax.set_title(f'Covid-19 {target_column.upper()} Forecast')
        plt.legend()
        plt.show();
    elif state_postal_code is not None:
        ax.set_ylabel(f'Number of People, {target_column}')
        ax.set_title(f'Covid-19 {target_column.upper()} Forecast, {state_postal_code}')
        plt.legend()
        plt.show();
```

```
    return forecast, forecast_object # returns model forecast data

def get_exogenous_forecast_dataframe(dataframe, original_dataframe, exog_forecast, target_column,
exogenous_column, days_to_forecast=30, m_periods=1):
    '''
    #purpose: to create a forecast dataframe with forecasted exogenous data

    dataframe: a dataframe of the state Covid data
        type = dataframe
    original_dataframe: a reference dataframe that is the same as dataframe but will remain static
within function
        type = dataframe
    exog_forecast = predicted forecast for the exogenous variable
        type = pandas series
    target_column: column to forecast trend
        type = str
    *exogenous_column: name of exogenous column data
        type = str
    days: number of days into the future you wish to forecast
        type = int
    (see return_arima_order():)
    (see build_SARIMAX_forecast():)

    output: arima object and new dataframe that will go into build_SARIMAX_forecast():
    '''

    # create extended index for dataframe
    today = datetime.date(datetime.now())
    td = timedelta(days=days_to_forecast-1)
    future_date = today+td
    rng = pd.date_range(dataframe.index.min(),future_date,freq='D')

    # reindex and set index to range variable rng
    dataframe = dataframe.reindex(rng)
    dataframe = dataframe.set_index(rng)

    # fill exogenous column with forecast data
    dataframe[exogenous_column] = dataframe[exogenous_column].fillna(exog_forecast)

    stepwise_fit = return_arima_order(dataframe.iloc[0:len(original_dataframe)], target_column,
days_to_forecast, m_periods=m_periods)
    arima_order = stepwise_fit.order
    seasonal_order = stepwise_fit.seasonal_order

    return stepwise_fit, dataframe

def create_exog_forecast(dataframe, target_column, alpha=.05, days_to_forecast=30, train_days=270,
m_periods=1, state_postal_code=None, verbose=True):
    '''
    summary function that returns a new dataframe as well as a forecast that will become the
exogenous variable in the graph_exog_forecast function
    '''
    if state_postal_code is None:
        original_dataframe = dataframe
    else:
        dataframe = state_dataframe(dataframe, state_postal_code)
        original_dataframe = dataframe

    # return arima model 'results_full'
    stepwise_fit, stepwise_full, results, results_full = arima_tune(dataframe, target_column,
```

```
days_to_forecast=days_to_forecast, train_days=train_days, m_periods=m_periods, verbose=True)

    exog_forecast, results_forecast = build_SARIMAX_forecast(model=results_full,
                                                             dataframe=dataframe,
                                                             target_column=target_column,
                                                             stepwise_fit=stepwise_full,
                                                             alpha=alpha,
days_to_forecast=days_to_forecast,
                                                             original_df=None,
exogenous_column=None)

    return dataframe, exog_forecast

def graph_exog_forecast(dataframe, target_column, exog_forecast, df_ref, alpha=.05,
days_to_forecast=30, train_days=270, m_periods=1, exogenous_column=None, state_postal_code=None):
    '''
    summary function whose purpose is to graph a target_column's forecast
    '''
    if exogenous_column is not None:
        stepwise_fit, df_forecast = get_exogenous_forecast_dataframe(dataframe=dataframe,
                                                                 original_dataframe=df_ref,
                                                                 exog_forecast=exog_forecast,
                                                                 target_column=target_column,
                                                                 exogenous_column=exogenous_column,
                                                                 days_to_forecast=days_to_forecast,
                                                                 m_periods=m_periods)

    full_exog_model = SARIMAX(dataframe[target_column],dataframe[exogenous_column],
                              order=stepwise_fit.order,
                              seasonal_order=stepwise_fit.seasonal_order)

    model = full_exog_model.fit()

    exog_forecast, forecast_object = build_SARIMAX_forecast(model=model,
                                                             dataframe=df_forecast,
                                                             target_column=target_column,
                                                             stepwise_fit=stepwise_fit,
                                                             alpha=alpha,
                                                             days_to_forecast=days_to_forecast,
                                                             original_df=df_ref,
                                                             exogenous_column=exogenous_column,
                                                             state_postal_code=state_postal_code)

    return forecast_object

def create_NN_predict(df_states,state_postal_code,days,epochs):

    '''
    *purpose: creates a RNN model based on datetime dataframe with column 'death'
              and a state postal code under column 'state'

    *inputs:
    df_states: a dataframe of the state Covid data
    state_postal_code: state postal code to get state related death data
    days: number of days out you wish to forecast
    epochs: number of epochs you wish to run
    '''

    # create dataframe based on state_postal_code
    df_state = df_states[df_states['state']==state_postal_code]
```

```
    # sort index, lowest index to oldest date, drop na's in death column
    df_state = df_state.sort_index()
    df_state = df_state.dropna(subset=['death'])
    df_state_new = pd.DataFrame(df_state['death'])

    length = len(df_state_new)-days

    # create train/test split based on days forecasting
    train_data = df_state_new.iloc[:length]
    test_data = df_state_new.iloc[length:]

    # create scaler
    scaler = MinMaxScaler()

    # fit on the train data
    scaler.fit(train_data)

    # scale the train and test data
    scaled_train = scaler.transform(train_data)
    scaled_test = scaler.transform(test_data)

    # define time series generator
    days
    n_features = 1
    generator = TimeseriesGenerator(scaled_train, scaled_train,
                                    length=days, batch_size=1)

    # build LSTM model
    model = Sequential()
    model.add(LSTM(300, activation='relu',
                   input_shape=(days,n_features)))
    model.add(Dense(1))
    model.compile(optimizer='adam',loss='mse')

    # fit the model
    model.fit_generator(generator,epochs=epochs)

    # get data for loss values
    loss_per_epoch = model.history.history['loss']
    # plt.plot(range(len(loss_per_epoch)),loss_per_epoch);

    # evaluate the batch
    first_eval = scaled_train[-days:]
    first_eval = first_eval.reshape((1, days, n_features))

    scaler_predictions = []

    first_eval = scaled_train[-days:]
    current_batch = first_eval.reshape((1, days, n_features))

    # create test predictions
    for i in range(len(test_data)):
        current_pred = model.predict(current_batch)[0]
        scaler_predictions.append(current_pred)
        current_batch = np.append(current_batch[:,1:,:],[[current_pred]],axis=1)

    true_predictions = scaler.inverse_transform(scaler_predictions)
    test_data['Predictions'] = true_predictions

    legend_elements = [Line2D([0], [0], color='g', lw=4, label='Actual Deaths'),
                       Line2D([0], [0], color='#FFA500', lw=4, label=f'RNN {state_postal_code}
```

```
Predictions')]

    fig, ax = plt.subplots(figsize=(20,10));
    ax.plot(test_data)
    ax.plot(train_data);
    ax.grid(b=True,alpha=.5)
    plt.title(f'Test Data vs RNN, {state_postal_code}')
    ax.legend(handles=legend_elements)
    plt.xlabel('Date')
    plt.ylabel('Deaths')
    plt.show();

def multivariate_nn_forecast(df_states,days_to_train,days_to_forecast,epochs):

    '''
    *purpose: creates a multivariate RNN model and graph forecast
              based on datetime dataframe with column 'death'

    *inputs:
    df_states: a dataframe of the US Covid data
    days_to_train: number of past days to train on
    days_to_forecast: number of days out you wish to forecast
    epochs: number of epochs you wish to run
    '''

    # remove extra, unnecessary columns
    df_states = df_states.sort_index()
    df_states = df_states.drop(columns=['dateChecked','lastModified','hash',
                                        'pending','hospitalizedCumulative',
                                        'inIcuCumulative', 'onVentilatorCumulative',
                                        'recovered','total','deathIncrease',
                                        'hospitalized','hospitalizedIncrease',
                                        'negativeIncrease','posNeg','positiveIncrease',
                                        'states','totalTestResults','totalTestResultsIncrease',
                                        'negative'])

    # drop rows where at least one element is missing
    df_states = df_states.dropna()

    # move death to first index position
    df_states = df_states[['death','positive', 'hospitalizedCurrently', 'inIcuCurrently',
                           'onVentilatorCurrently']]

    # drop all but those currently on ventilators and percentage testing positive out of total test
pool
    df_states = df_states.drop(columns=['positive','inIcuCurrently','hospitalizedCurrently'])

    # where to specificy the columns to use in multivariate NN
    columns = list(df_states)[0:2]
    print(columns) # variables, x axis is time

    # extract x axis dates for plotting certain graphs
    X_axis_dates = pd.to_datetime(df_states.index)

    # create training df, ensure float data types
    df_training = df_states[columns].astype(float)

    # scale the dataset
    standard_scaler = StandardScaler()
    standard_scaler.fit(df_training)
    df_training_scaled = standard_scaler.transform(df_training)
```

```
    # create lists to append to
    X_train = []
    y_train = []

    # take in input arguments from function call
    future_days = 1
    past_days = days_to_train            # number of days to train the model on

    for i in range(past_days, len(df_training_scaled) - future_days + 1):
        X_train.append(df_training_scaled[i-past_days:i, 0:df_training.shape[1]])
        y_train.append(df_training_scaled[i+future_days-1:i+future_days,0])

    # set X_train and y_train data sets to numpy arrays
    X_train, y_train = np.array(X_train), np.array(y_train)

    # save shapes of numpy arrays as variables
    shapeX = X_train.shape
    shapey = y_train.shape

    def make_model():
        model = Sequential()
        model.add(LSTM(100, activation='relu', return_sequences=True,
                       input_shape=(shapeX[1],shapeX[2])))
        model.add(LSTM(50, activation='relu', return_sequences=False))
        model.add(Dense(shapey[1]))
        model.compile(optimizer='adam',loss='mse')
        return model

    # instantiate model (make_model function is in this .py file) and fit
    model = make_model()
    history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2, verbose=0)

    # create forecast data
    days = days_to_forecast
    forecast_dates = pd.date_range(list(X_axis_dates)[-1],periods=days,freq='D').tolist()
    forecast = model.predict(X_train[-days:])

    # create target future forecast data and inverse transform
    forecast_columns = np.repeat(forecast, df_training_scaled.shape[1],axis=-1)
    y_pred_future = standard_scaler.inverse_transform(forecast_columns)[:,0]

    # append dates back into new dataframe
    forecast_dates_array = []
    for time in forecast_dates:
        forecast_dates_array.append(time.date())

    # create final forecast dataframe
    df_fcast = []
    df_fcast = pd.DataFrame({'date':np.array(forecast_dates_array),'death':y_pred_future})
    df_fcast.index=pd.to_datetime(df_fcast['date'])

    # plot the data and the forecast data
    df_fcast['death'].plot(legend=True, figsize=(15,7));
    (df_states['death']).plot(legend=True);


def make_model():
    # initialize and build sequential model
    model = Sequential()
```

```
    model.add(LSTM(100, activation='relu', return_sequences=True,
                   input_shape=(shapeX[1],shapeX[2])))
    model.add(LSTM(50, activation='relu', return_sequences=False))
    model.add(Dense(shapey[1]))

    model.compile(optimizer='adam',loss='mse')
    model.summary()
    return model
```