In [1]:
```
import warnings
warnings.filterwarnings("ignore")
```

CQF Final Projects

candidate: Zhenyang Lu, zzhenyanglu@gmail.com (mailto:zzhenyanglu@gmail.com)

Aug 2021

0. Data Access

Data Source:

Disclaimer: trading data used in this project is from Alpaca trading platform, because it's paper trading account, only 1000 periods of bar data would be available, which poses a bit of challenge to statistical analysis because of the amount of data. I've included the client secret to communicate with alpaca server.

Instrument:

After trials on many pairs of ETF, and stocks that test if we can set up a mean reverting trading system, pair of GLD and GDX emerges as a promising candidate (not probably even the best). Below are profiles:

GLD: https://finance.yahoo.com/quote/GLD/profile?p=GLD
GDX: https://finance.yahoo.com/quote/GLD/profile?p=GDX

from an economic point of view, they are both tracking retail industry, and their statistical property still makes sense as a pair of trading instrument, we could set up a weak stationary system between them (as you would see below in this project)

raw_data

contains bar data of GLD and GDX as it retrieves data directly from Alpaca

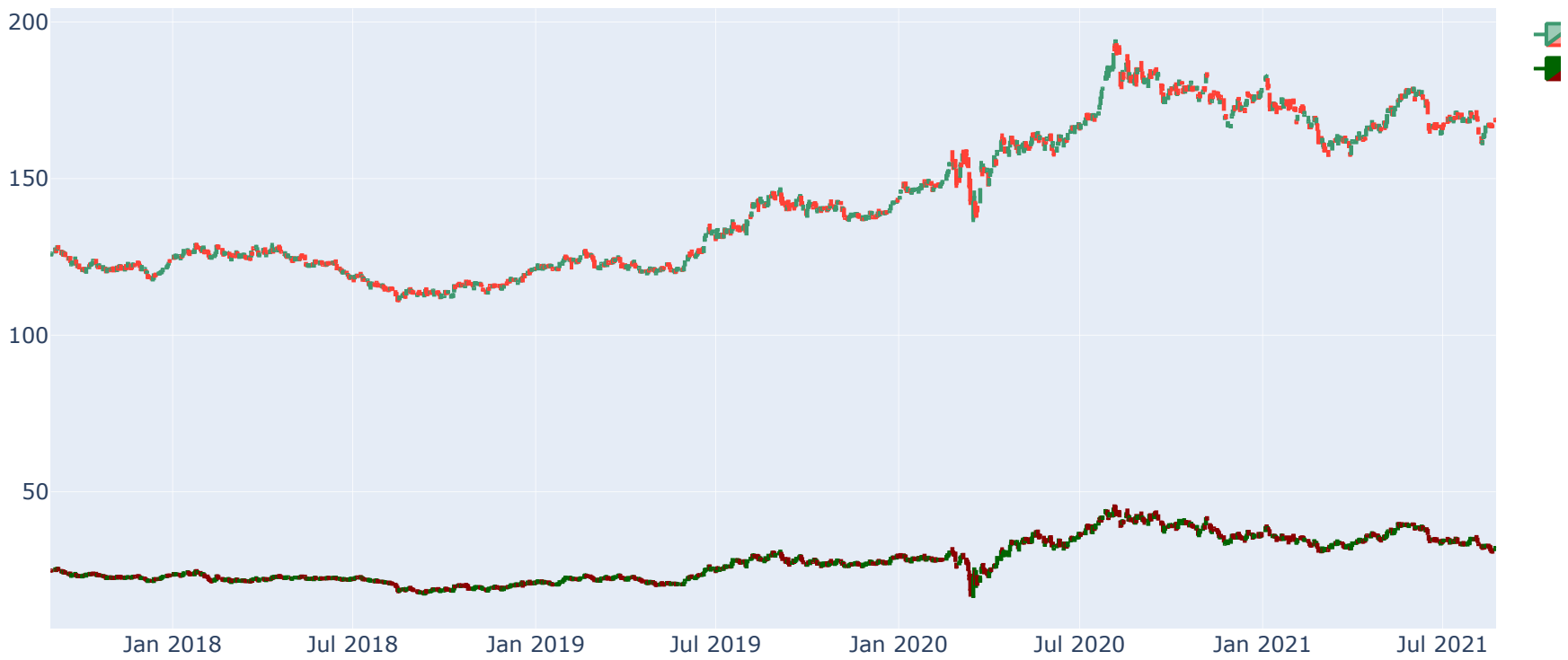
In [2]:
```
import yaml, os
import alpaca_trade_api as tradeapi
import pandas as pd

with open("alpaca.yaml") as f:
    data = yaml.load(f)
    os.environ["APCA_API_KEY_ID"] = data["key_id"]
    os.environ["APCA_API_SECRET_KEY"] = data["secret"]
    os.environ["APCA_API_BASE_URL"] = data["base_url"]
    api = tradeapi.REST(api_version='v2')

symbols = ['GLD','GDX']

## Get historical data
raw_data = api.get_barset(symbols, 'day', 1000).df
raw_data = raw_data[~raw_data[symbols[0]]['volume'].isnull() & ~raw_data[symbols[1]]['volume'].isnull()]

raw_data.head()
```

Out[2]:

	GDX					GLD				
	open	high	low	close	volume	open	high	low	close	volume
time										
2017-09-01 00:00:00-04:00	24.84	24.87	24.49	24.790	24826510	126.01	126.17	125.12	126.06	8334772
2017-09-05 00:00:00-04:00	25.00	25.33	24.99	25.290	50424402	126.65	127.78	126.57	127.46	8791531
2017-09-06 00:00:00-04:00	25.19	25.29	24.78	24.930	37648691	127.34	127.44	126.55	126.82	6829598
2017-09-07 00:00:00-04:00	25.20	25.58	25.15	25.475	41087910	127.57	128.32	127.39	128.14	7390006
2017-09-08 00:00:00-04:00	25.42	25.48	25.08	25.260	34811858	128.20	128.30	127.59	127.98	6040239

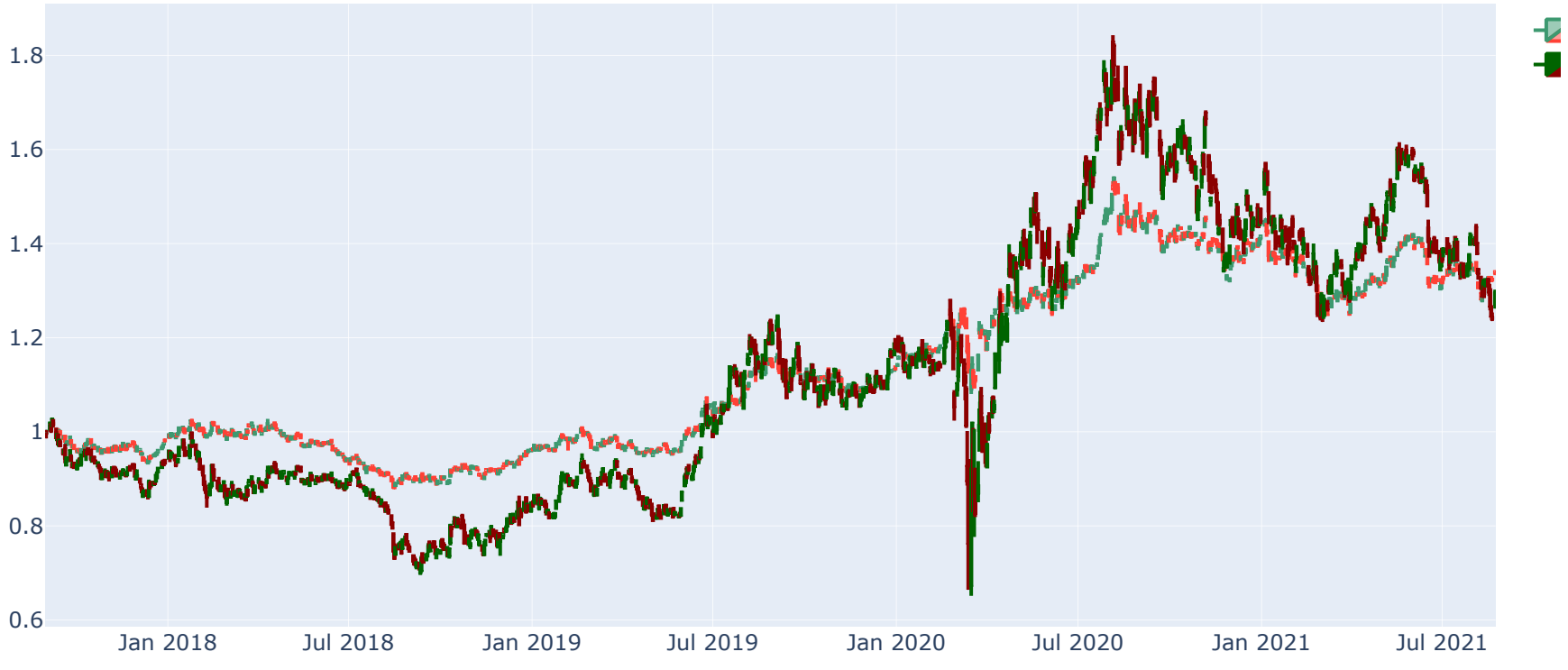
Raw Bar Chart

In [3]:
```
import plotly.graph_objects as go
from plotly.subplots import make_subplots

fig = go.Figure(data=[go.Candlestick(x=raw_data.index,
                open=raw_data[symbols[0]]['open'],
                high=raw_data[symbols[0]]['high'],
                low=raw_data[symbols[0]]['low'],
                close=raw_data[symbols[0]]['close'], name = symbols[0]),

                go.Candlestick(x=raw_data.index,
                open=raw_data[symbols[1]]['open'],
                high=raw_data[symbols[1]]['high'],
                low=raw_data[symbols[1]]['low'],
                close=raw_data[symbols[1]]['close'], name = symbols[1])],
               )

fig.update_layout(xaxis_rangeslider_visible=False)

fig.data[1].increasing.fillcolor = 'darkgreen'
fig.data[1].decreasing.fillcolor = 'darkred'
fig.data[1].increasing.line.color = 'darkgreen'
fig.data[1].decreasing.line.color = 'darkred'

fig.show()
```



Normalized Bar Chart

In [4]:
```
import plotly.graph_objects as go
from plotly.subplots import make_subplots

# normalize data

for symbol in symbols:
    for col in ['open','high','low','close']:
        raw_data.loc[:, (symbol, f'{col}_normalized')] = raw_data.loc[:, (symbol, f'{col}')] / raw_data.iloc[0][symb

fig = go.Figure(data=[go.Candlestick(x=raw_data.index,
                open=raw_data[symbols[0]]['open_normalized'],
                high=raw_data[symbols[0]]['high_normalized'],
                low=raw_data[symbols[0]]['low_normalized'],
                close=raw_data[symbols[0]]['close_normalized'], name = symbols[0]),

                go.Candlestick(x=raw_data.index,
                open=raw_data[symbols[1]]['open_normalized'],
                high=raw_data[symbols[1]]['high_normalized'],
                low=raw_data[symbols[1]]['low_normalized'],
                close=raw_data[symbols[1]]['close_normalized'], name = symbols[1])],
               )

fig.update_layout(xaxis_rangeslider_visible=False)
fig.data[1].increasing.fillcolor = 'darkgreen'
fig.data[1].decreasing.fillcolor = 'darkred'
fig.data[1].increasing.line.color = 'darkgreen'
fig.data[1].decreasing.line.color = 'darkred'
fig.show()
```



1. A not so fancy handcrafted linear regression analysis package

Math:

The following function $linear\_regression$ is based on the following formulae:

coefficients: $(X^T X)^{-1} X^T Y$

Residual covariance matrix: $\widehat{\Sigma} = \widehat{\varepsilon}^T \widehat{\varepsilon} \big/ m$

Covariance matrix of regression coefficients: $\widehat{CoV} = \widehat{\Sigma} \otimes (X^T X)^{-1}$

Standard error of regression coefficient: $\left(\mathrm{Var}\left(\widehat{\beta}_{kj}\right)\right)^{1/2} = \left(\frac{m}{m-\ell}\mathrm{diag}(\widehat{CoV})_{kj}\right)^{1/2}$

t-test statistic for whether regression coefficient is zero: $\mathrm{tstat}_{kj} = \widehat{\beta}_{kj} \Big/ \left(\mathrm{Var}\left(\widehat{\beta}_{kj}\right)\right)^{1/2}$

Akaike information criterion: $AIC = \log\left|\widehat{\Sigma}\right| + 2\ell n \big/ m$

Bayesian information criterion: $BIC = \log\left|\widehat{\Sigma}\right| + \log(m)\ell n \big/ m$

where

Number of observations:	m
Number of regressands:	n
Number of regressors:	ℓ
Independent variable array:	Y
Independent variable array:	X
Residuals:	$\widehat{\varepsilon} = Y - X\widehat{B}$

In [5]:

```python
# z is independent variables
# x is dependent variables
import numpy as np

def linear_regression(z: np.array,x: np.array) -> dict:
    """
    arguments:
    z: independent variables
    x: dependent variable

    return values:
    1. a dictitionary that contains the following stats:
        coefficients of model
        t-score
        coefficients standard error
        AIC
        BIC
        residual covariance
        mean squared error(mse)
        sum of squares total(sse)
        standard error

    2. residual array
    """

    coefficient = np.dot(np.dot(np.linalg.inv(np.dot(z.transpose(),z)),z.transpose()),x)
    m =  z.shape[0]  # # of obs
    l =  1           # # of dependent variables
    n =  z.shape[1]  # # of independent variables
    z_t_z_inv = np.linalg.inv(np.dot(z.T,z))
    x_t_z_z_t_z_inv_z_t=np.dot(np.dot(x.T,z),z_t_z_inv)

    sse = np.dot(x.T,x)-np.dot(np.dot(x_t_z_z_t_z_inv_z_t,z.T),x)
    mse = sse/(z.shape[0]-z.shape[1])
    stderr = mse*np.linalg.inv(np.dot(z.T,z))[0][0]

    try:
        residual = x.iloc[:,0] - (coefficient*z).sum(axis=1)
    except:
        residual = x - (coefficient*z).sum(axis=1)
    residual_covar = np.dot(residual.T, residual)/x.shape[0]
    aic = np.log(np.abs(residual_covar)) + 2*l*n/m
    bic = np.log(np.abs(residual_covar)) + np.log(m)*l*n/m
    coefficients_covar = np.linalg.inv(np.dot(z.transpose(),z))*residual_covar
    coefficients_stderr = np.sqrt(m/(m-l) * np.diag(coefficients_covar))
    t = coefficient/coefficients_stderr

    result = {'coefficients': coefficient, 't':t, 'coefficients_stderr': coefficients_stderr, 'AIC': aic, 'BIC': bic

    return result, residual
```

1.1 a quick test - cross validation between statsmodel

we do a regress normalized open price of GLD on that of GDX

$$Price_{GLD} = \beta * Price_{GDX} + constant$$

observations:

1. generally speaking the result looks fine. model coefficients as well as coefficient standard errors, t-score, training residuals matches the result from statsmodel as shown below
2. But AIC and BIC scores don't match.. could not figure out why.. AIC/BIC from statsmodel package look so large to me.

Below is he result of my own linear regression tool with that of python's statsmodels.

This is statsmodel regression analysis summary:

In [6]:
```
from statsmodels.api import OLS, add_constant

symbol_0 = raw_data[(symbols[0],'open_normalized')]
symbol_1 = raw_data[(symbols[1],'open_normalized')]
symbol_1 = add_constant(symbol_1)

ols_model = OLS(symbol_0, symbol_1).fit()
ols_model.summary()
```

Out[6]:

OLS Regression Results

Dep. Variable:	('GLD', 'open_normalized')	**R-squared:**	0.951
Model:	OLS	**Adj. R-squared:**	0.951
Method:	Least Squares	**F-statistic:**	1.945e+04
Date:	Mon, 23 Aug 2021	**Prob (F-statistic):**	0.00
Time:	20:02:37	**Log-Likelihood:**	1797.1
No. Observations:	1000	**AIC:**	-3590.
Df Residuals:	998	**BIC:**	-3580.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.4150	0.005	78.236	0.000	0.405	0.425
('GDX', 'open_normalized')	0.6397	0.005	139.469	0.000	0.631	0.649

Omnibus:	298.178	**Durbin-Watson:**	0.080
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	1094.262
Skew:	1.399	**Prob(JB):**	2.42e-238
Kurtosis:	7.294	**Cond. No.**	8.32

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

This is result from my own regression analysis function:

In [7]:
```
from statsmodels.api import OLS, add_constant

symbol_0 = raw_data[(symbols[0],'open_normalized')]
symbol_1 = raw_data[(symbols[1],'open_normalized')]
symbol_1 = add_constant(symbol_1)

results, residual = linear_regression(symbol_1, symbol_0)

print('\tconst          : ', results['coefficients'][0], ' std error: ', results['coefficients_stderr'][0], " t scor
print('\topen_normalized: ', results['coefficients'][1], ' std error: ', results['coefficients_stderr'][0], " t scor

## assert that the two residual arrays output by two models are same
assert pd.testing.assert_series_equal(residual, ols_model.resid) == None, "Residuals output by two models are differ
print( "Residuals output by two models are same")
```

```
        const          :  0.41498051458829327  std error:  0.005301534961749411  t score:  78.2755404957958
        open_normalized:  0.6397290244963131  std error:  0.005301534961749411  t score:  139.53913988800377
Residuals output by two models are same
```

2.0 Equilibrium Correction Model (ECM)

Below we are performing $Engle - GrangerAnalysis$

2.1 Regression analysis on absolute prices of GLD and GDX

Section 1.1 serves as a cross validation as well as a regression analysis, from which we can see:

$$Price_{GLD} = 0.637 * Price_{GDX} + 0.4150,$$

the coefficients are significant as the t-score shows: 78.27 and 139.54, which are far exceeding 1% threshold.

To validate vice versa, we perform regres GDX on GLD

Based on the linear analysis below this model looks like:

$$Price_{GDX} = 1.4869 * Price_{GLD} - 0.5622$$

both coefficients have a high t-score, which indicates they are all significant.

we'll proceed to test if residual of this model is weak stationary

In [8]:
```
from statsmodels.api import OLS, add_constant

symbol_0 = raw_data[(symbols[0],'open_normalized')]
symbol_1 = raw_data[(symbols[1],'open_normalized')]
symbol_0 = add_constant(symbol_0)

results, stationary_residual = linear_regression(symbol_0, symbol_1)

print('\tconst          : ', results['coefficients'][0], ' std error: ', results['coefficients_stderr'][0], " t scor
print('\topen_normalized: ', results['coefficients'][1], ' std error: ', results['coefficients_stderr'][0], " t scor
```

```
const          :  -0.5622304254391975  std error:  0.012229427304274954  t score:  -45.97356944447126
open_normalized:  1.4868752957421312  std error:  0.012229427304274954  t score:  139.53913988800318
```

2.2 Weak Stationarity of residual

As previously mentioned, the regression model we have thus far is:

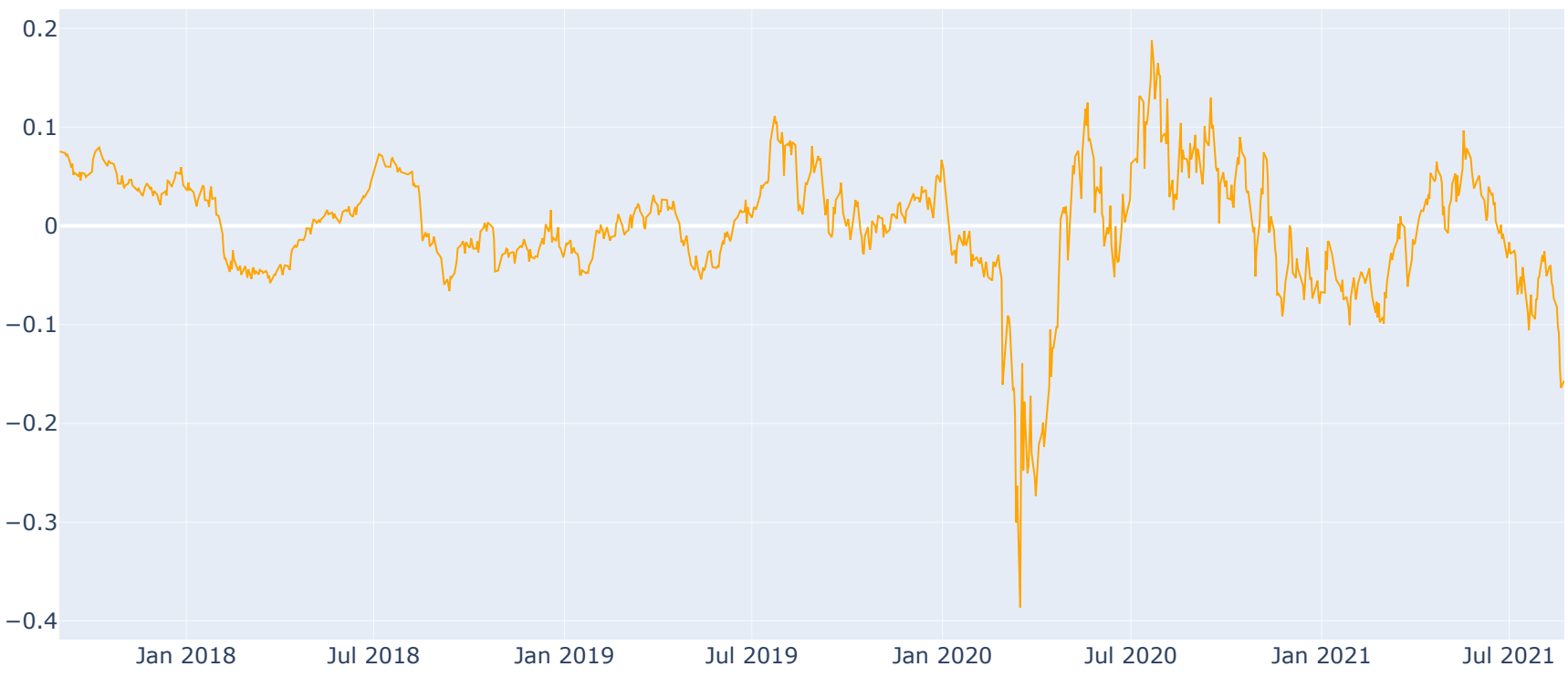
$$Price_{GDX} = 1.4869 * Price_{GLD} - 0.5622 + e^t$$

Hence the residual would be:

$$e^t = Price^t_{GDX} - 1.4869 * Price^t_{GLD} + 0.5622$$

Below is a time plot of the stationary residual:

In [9]:
```
import plotly.graph_objects as go
from plotly.subplots import make_subplots

go.Figure(data=[
    go.Scatter(x=stationary_residual.index, y=stationary_residual, line=dict(color='orange', width=1)),
]).show()
```



Augmented Dickey-Fuller Test on Residual

Below is lag 1 ADF test on the residual, and the p value is smaller than 1%, thus we can reject the null hypothesis:

H_0 : a unit root is present in the residual

Thus the residual is weak stationary

In [10]:
```
from statsmodels.tsa.stattools import adfuller

lags = 1

result = adfuller(stationary_residual, maxlag=lags)
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
for key, value in result[4].items():
    print('\t%s: %.3f' % (key, value))
```

```
ADF Statistic: -3.700351
p-value: 0.004108
Critical Values:
        1%: -3.437
        5%: -2.864
        10%: -2.568
```

2.3 plug the error correction term into linear regression model and test statistical significance

we proceed with step 2 of Engle-Granger Procedure: Error Correction, before pulling any statistical test, we'll wrangle the data a bit to produce $\Delta Price_{GLD}, \Delta Price_{GDX}$ and shift the residual term by 1 period, which is below:

lagged_residual, (GLD, open_normalized_change) and (GDX, open_normalized_change) columns are what we are interested in below

In [11]:
```
# data preparing for EC Model

# shift residual by 1 and join with raw prices DF
if 'lagged_residual' in raw_data.columns:
    raw_data.drop('lagged_residual', inplace=True, axis=1)

raw_data = raw_data.merge(
    pd.DataFrame(stationary_residual.shift(1),
                 columns=['lagged_residual']),
    left_index=True,
    right_index=True,
    how='left'
)

# create price delta of GLD and GDX
if 'open_normalized_diff' not in raw_data.columns:
    raw_data[(symbols[0],'open_normalized_change')] = raw_data[(symbols[0],'open_normalized')].diff(1)
    raw_data[(symbols[1],'open_normalized_change')] = raw_data[(symbols[1],'open_normalized')].diff(1)
    raw_data = raw_data[~raw_data[(symbols[0],'open_normalized_change')].isnull() & ~raw_data[(symbols[1],'open_norm


# sample GLD and GDX price delta and EC Model correction data (the weak stationary residual)
raw_data[[(symbols[0],'open_normalized_change'),
          (symbols[0],'open_normalized'),
          (symbols[1],'open_normalized_change'),
          (symbols[1],'open_normalized'),
          ('lagged_residual')]].head()
```

Out[11]:

time	(GLD, open_normalized_change)	(GLD, open_normalized)	(GDX, open_normalized_change)	(GDX, open_normalized)	lagged_residual
2017-09-05 00:00:00-04:00	0.005079	1.005079	0.006441	1.006441	0.075355
2017-09-06 00:00:00-04:00	0.005476	1.010555	0.007649	1.014090	0.074245
2017-09-07 00:00:00-04:00	0.001825	1.012380	0.000403	1.014493	0.073752
2017-09-08 00:00:00-04:00	0.005000	1.017380	0.008857	1.023349	0.071440
2017-09-11 00:00:00-04:00	-0.011586	1.005793	-0.026167	0.997182	0.072863

We are now in position to perform **the 2nd step of Engle-Granger** procedure which is to estimate the Equilibrium Correction Model, or error correction equations:

$$\Delta P_t^{GLD} = \varphi \Delta P_t^{GDX} - (1-\alpha)\widehat{e}_{t-1} + \varepsilon_t$$

$$\Delta \mathrm{GDX}_t = \varphi \Delta \mathrm{GLD}_t - (1-\alpha)\mathrm{CointResidual\_GDX}_{t-1} + \varepsilon_t$$

According to the following regression model, the actual mode specifics is:

$$\Delta \mathrm{GDX}_t = 2.1281 \Delta \mathrm{GLD}_t - 0.0408\mathrm{CointResidual\_GDX}_{t-1} + \varepsilon_t$$

The 2 coefficients are statistical significant.

For simplicity, only 1 model is conducted.

In [12]:
```
symbol_0 = raw_data[['lagged_residual', (symbols[0],'open_normalized_change')]]
symbol_1 = raw_data[(symbols[1],'open_normalized_change')]

results, residual = linear_regression(symbol_0, symbol_1)

print('\tcoefficient 2: ', results['coefficients'][0], ' std error: ', results['coefficients_stderr'][0], " t score:
print('\tcoefficient 1: ', results['coefficients'][1], ' std error: ', results['coefficients_stderr'][0], " t score:
```

```
	coefficient 2:  -0.040777061274042285  std error:  0.0085474432014809  t score:  -4.770673558494942
	coefficient 1:  2.12809017129277  std error:  0.0085474432014809  t score:  44.724520388420906
```

2.4 Quality of Mean-Reversion

Neither the final project nor the lecture handouts give a clear definition of how to calculate half-life, so I paraphrase it with some simplification here as an reference (based on book: analysis of financial time series by RUEY TSAY, Wiley)

Assume we have a weak stationary auto regression model AR(1):

$X_t = \theta X_{t-1} + C + \epsilon_t,\ where\ |\theta| < 1$, (1),

and by default,

$E[X_t] = \mu$, taking expectation on both sides of (1) and do a bit transformation we have

$\mu = \frac{C}{1-\theta}$ (2)

define $Y_t = X_t - E[X_t]$ (3), which is the distance to the constant mean

Combining (2) and (3), we have

$Y_t = \theta Y_{t-1} + \epsilon_t$ (4)

by definition, half-life of mean-reverting would be the time within which the mean-reverting process is expected to revert in half to its stationary mean, or mathematically:

$E[Y_{t+h}] = \frac{1}{2} * Y_t$

according to formula (4), we have

$E[Y_{t+h}] = \theta Y_{t+h-1} = \theta^2 Y_{t+h-2} = \ldots = \theta^h Y_t$

$h = -\frac{ln(2)}{\ln(|\theta|)} = \frac{ln(0.5)}{\ln(|\theta|)}$ (5)

based on (1), we have $E[X_t] = \theta E[X_{t-1}] + C$ Thus

$\mu = E[X_t] = \frac{C}{1-\theta}$ (6)

$\sigma_{eq} = \sqrt{\frac{SSE*\tau}{1-e^{-2\theta}}}$ (7)

Recall from (1), θ would be the coefficient of the 1-period lag cointegrated residual from the AR(1) model. So we can get it either by running an AR(1) model or simply regress the cointegrated residual by 1-period lag itself, which is $16.495\ periods$ in this excercise

below is the code to get AR(1) model's coefficients and Sum of squared error.

In [13]:
```
from statsmodels.tsa.ar_model import AutoReg


result, residual = linear_regression(add_constant(stationary_residual.shift(1))[1:],stationary_residual[1:])

half_life = -np.log(2) / np.log(result['coefficients'][1])
print ('Theta (Day):       %s' % result['coefficients'][1])
print ('C:       %s' % result['coefficients'][0])
print ('Half Life (Day):       %s' % half_life)
print ('SSE:       %s' % result['sse'])
print ('TAU:       %s' % str(1/252))
```

```
Theta (Day):       0.9586103130590673
C:       -0.0002259309422314719
Half Life (Day):       16.397842126863104
SSE:       0.32009564881652075
TAU:       0.003968253968253968
```

Based on (5) (6) (7) and the result from linear regression above:

$\mu = E[X_t] = \frac{C}{1-\theta} = -0.00023/(1-0.9586) = -0.0056$

$\sigma_{eq} = \sqrt{\frac{SSE*\tau}{1-e^{-2\theta}}} = \sqrt{\frac{0.3201*0.003968}{1-e^{-2*0.9588}}} = 0.0396$

$h = -\frac{ln(2)}{\ln(|\theta|)} = \frac{ln(0.5)}{\ln(|\theta|)} = 16.5\ \ days$

3.0 Trading strategy

Based on 2.4, we can conclude that trading bounds are:

$\mu \pm Z\sigma_{eq} = -0.0056 \pm Z * 0.0396$, where Z is a boundary factor, for this project I'd choose 1.2 for the lower bound, and 1.5 for the upper bound (just some arbitrary numbers not scientific), where

$e^t = Price^t_{GDX} - 1.4869 * Price^t_{GLD} + 0.5622$

So the trading strategy would be:

$when\ e_{eq} >> \mu + Z_{upper} * \sigma = 0.0617\ enter\ with\ [-P_{GDX}, +1.5077 * P_{GLD}]$ and hold until exit.

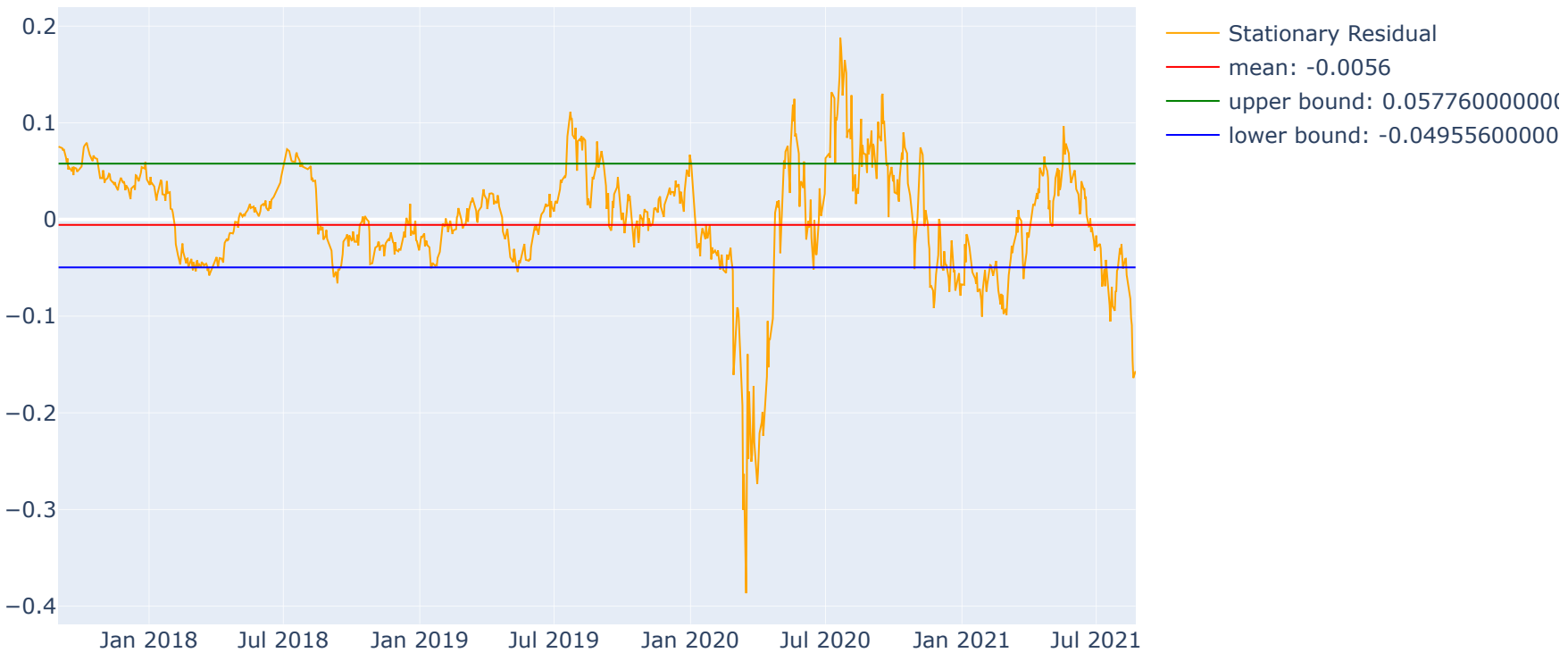
$when\ e_{eq} << \mu + Z_{lower} * \sigma = -0.0531\ enter\ with\ [P_{GDX}, -1.5077 * P_{GLD}]$ and hold until exit.

$when\ e_{eq} == \mu\ exit$

assuming we either buy/sell/hold only 1 pair of GLD, GDX (namely, can't do more trade if holding positions)

Plot below gives a basic ideas:

In [14]:
```
import plotly.graph_objects as go
from plotly.subplots import make_subplots

Z_upper = 1.6
Z_lower = 1.11
mu = -0.0056
sigma = 0.0396

upper_bound = mu + Z_upper*sigma
lower_bound = mu - Z_lower*sigma

go.Figure(data=[
    go.Scatter(x=stationary_residual.index, y=stationary_residual, line=dict(color='orange', width=1), name="Statior
    go.Scatter(x=stationary_residual.index, y=[mu]*stationary_residual.shape[0], line=dict(color='red', width=1), na
    go.Scatter(x=stationary_residual.index, y=[upper_bound]*stationary_residual.shape[0], line=dict(color='green', w
    go.Scatter(x=stationary_residual.index, y=[lower_bound]*stationary_residual.shape[0], line=dict(color='blue', wi
]).show()
```



4.0 In-sample backtest

To reiterrate, the strategy would be: either buy/sell/hold only 1 pair of GLD, GDX (namely, can only trade more after exit from current position)

below was the code for in-sample backtest:

we create a few columns to track performance:

1. cash: indicates how many cash we have on each trading day, initially it's 1

2. position: instruments we hold each day (how many GLD, and GDX), for example: {'GDX':-1,'GLD':1.5077} means short 1 share GDX, and long 1.5077 shared GLD.
3. action: long or short, for example long {'GDX':-1,'GLD':1.5077} means short 1 shared GDX and long 1.5077 GLD.

Notice: the following code doesn't account for trading cost nor short selling cost, and assume less than 1 share is possible

In [33]:

```
## Preparing data for backtest
test_data = raw_data[[('GLD','open_normalized'), ('GDX','open_normalized')]]
## merge residual with test_data
test_data = test_data.merge(pd.DataFrame(stationary_residual[1:],columns=['residual']),left_index=True, right_index=
test_data['total_value'] = [1] + [None]*(test_data.shape[0]-1)
test_data['cash'] = [1] + [None]*(test_data.shape[0]-1)
test_data['position'] = [[None,None]]*test_data.shape[0]
test_data['action'] = [None]*test_data.shape[0]

initial_cash =1
Z_upper = 1.6
Z_lower = 1.11
mu = -0.0056
sigma = 0.0396
upper_bound = mu + Z_upper*sigma
lower_bound = mu - Z_lower*sigma
has_position = False
previous_residual = 1
previous_day = None
GLD = ('GLD', 'open_normalized')
GDX = ('GDX', 'open_normalized')

for day in test_data.index:

    # long [-GDX,+1.5077*GLD]
    if test_data.ix[day].residual > upper_bound and not has_position:
        has_position = True
        test_data.loc[day,'action'] = 'long'
        test_data.loc[day,'position'] = "{'GDX':-1,'GLD':1.5077}"

        if day != test_data.index[0]:
            test_data.loc[day,'cash'] = test_data.ix[previous_day]['cash'] - 1.5077*test_data.ix[day][GLD] + test_da
        else:
            test_data.loc[day,'cash'] = initial_cash - 1.5077*test_data.ix[day][GLD] + test_data.ix[day][GDX]
        test_data.loc[day,'total_value'] = test_data.loc[day,'cash'] + 1.5077*test_data.ix[day][GLD] - test_data.ix[


    # long [GDX,-1.5077*GLD]
    elif test_data.ix[day].residual < lower_bound and not has_position:
        has_position = True
        test_data.loc[day,'action'] = 'long'
        test_data.loc[day,'position'] = "{'GDX':1,'GLD':-1.5077}"

        if day != test_data.index[0]:
            test_data.loc[day,'cash'] = test_data.ix[previous_day]['cash'] + 1.5077*test_data.ix[day][GLD] - test_da
        else:
            test_data.loc[day,'cash'] = initial_cash + 1.5077*test_data.ix[day][GLD] - test_data.ix[day][GDX]

        test_data.loc[day,'total_value'] = test_data.loc[day,'cash'] -1.5077*test_data.ix[day][GLD] + test_data.ix[


    #  reisdual fall from above upper bound to below mean
    # short [-GDX,+1.5077*GLD]
    elif test_data.ix[day].residual < mu and previous_residual >= mu and has_position:
        has_position = False
        test_data.loc[day,'action'] = 'short'
        test_data.loc[day,'position']="{'GDX':0,'GLD':0}"
        test_data.loc[day,'cash'] = test_data.ix[previous_day]['cash'] + 1.5077*test_data.ix[day][GLD] - test_data.i
        test_data.loc[day,'total_value'] = test_data.loc[day,'cash']


    #  reisdual rise from below upper bound to above mean
    # short [GDX,-1.5077*GLD]
    elif test_data.ix[day].residual > mu and previous_residual <= mu and has_position:
        has_position = False
        test_data.loc[day,'action'] = 'short'
        test_data.loc[day,'position']="{'GDX':0,'GLD':0}"
        test_data.loc[day,'cash'] = test_data.ix[previous_day]['cash'] - 1.5077*test_data.ix[day][GLD] + test_data.i
        test_data.loc[day,'total_value'] = test_data.loc[day,'cash']

    else:
        test_data.loc[day,'cash'] = test_data.ix[previous_day]['cash']
        position = eval(test_data.ix[previous_day]['position'])
        test_data.loc[day,'position'] = test_data.ix[previous_day]['position']

        if position.values() != [None, None]:
            test_data.loc[day,'total_value'] = test_data.ix[previous_day]['cash'] + position['GDX']*test_data.ix[day
        test_data.loc[day,'total_value'] = test_data.ix[previous_day]['cash'] +  position['GDX']*test_data.ix[day][G

    previous_residual = test_data.ix[day].residual
    previous_day = day
    previous_total_value = test_data['total_value']

```

what's printed out at the next cell shows when we actually traded based on the algo:

In [34]:
```
1 test_data[~test_data.action.isnull()]
```

Out[34]:

time	(GLD, open_normalized)	(GDX, open_normalized)	residual	total_value	cash	position	action
2017-09-05 00:00:00-04:00	1.005079	1.006441	0.074245	1.000000	0.491084	{'GDX':-1,'GLD':1.5077}	long
2018-02-05 00:00:00-05:00	1.006428	0.925926	-0.008277	1.082549	1.082549	{'GDX':0,'GLD':0}	short
2018-02-23 00:00:00-05:00	1.000397	0.875604	-0.049631	1.082549	1.715244	{'GDX':1,'GLD':-1.5077}	long
2018-04-27 00:00:00-04:00	0.993572	0.912641	-0.002446	1.129876	1.129876	{'GDX':0,'GLD':0}	short
2018-07-02 00:00:00-04:00	0.937862	0.890499	0.058246	1.129876	0.606361	{'GDX':-1,'GLD':1.5077}	long
2018-08-17 00:00:00-04:00	0.886358	0.740338	-0.015336	1.202385	1.202385	{'GDX':0,'GLD':0}	short
2018-09-06 00:00:00-04:00	0.903817	0.724638	-0.056995	1.202385	1.840432	{'GDX':1,'GLD':-1.5077}	long
2018-10-15 00:00:00-04:00	0.921514	0.805958	-0.001988	1.257024	1.257024	{'GDX':0,'GLD':0}	short
2019-01-16 00:00:00-05:00	0.968495	0.828100	-0.049700	1.257024	1.889123	{'GDX':1,'GLD':-1.5077}	long
2019-02-01 00:00:00-05:00	0.991112	0.907005	-0.004424	1.301829	1.301829	{'GDX':0,'GLD':0}	short
2019-05-13 00:00:00-04:00	0.973177	0.830515	-0.054247	1.301829	1.938572	{'GDX':1,'GLD':-1.5077}	long
2019-06-12 00:00:00-04:00	0.996746	0.916667	-0.003140	1.352444	1.352444	{'GDX':0,'GLD':0}	short
2019-07-18 00:00:00-04:00	1.062059	1.083333	0.066415	1.352444	0.834512	{'GDX':-1,'GLD':1.5077}	long
2019-09-13 00:00:00-04:00	1.124752	1.103462	-0.006673	1.426838	1.426838	{'GDX':0,'GLD':0}	short
2019-12-31 00:00:00-05:00	1.137291	1.195652	0.066873	1.426838	0.907797	{'GDX':-1,'GLD':1.5077}	long
2020-01-08 00:00:00-05:00	1.178399	1.182367	-0.007534	1.502101	1.502101	{'GDX':0,'GLD':0}	short
2020-02-10 00:00:00-05:00	1.176176	1.134863	-0.051734	1.502101	2.140560	{'GDX':1,'GLD':-1.5077}	long
2020-04-24 00:00:00-04:00	1.294580	1.369565	0.006917	1.558287	1.558287	{'GDX':0,'GLD':0}	short
2020-05-06 00:00:00-04:00	1.261725	1.374799	0.061001	1.558287	1.030782	{'GDX':-1,'GLD':1.5077}	long
2020-06-05 00:00:00-04:00	1.259583	1.289855	-0.020757	1.640000	1.640000	{'GDX':0,'GLD':0}	short
2020-06-15 00:00:00-04:00	1.273867	1.280193	-0.051658	1.640000	2.280416	{'GDX':1,'GLD':-1.5077}	long
2020-06-16 00:00:00-04:00	1.284422	1.347021	-0.000524	1.690914	1.690914	{'GDX':0,'GLD':0}	short
2020-07-01 00:00:00-04:00	1.325688	1.472222	0.063319	1.690914	1.164396	{'GDX':-1,'GLD':1.5077}	long
2020-10-27 00:00:00-04:00	1.418221	1.540258	-0.006229	1.762390	1.762390	{'GDX':0,'GLD':0}	short
2020-10-29 00:00:00-04:00	1.388937	1.451691	-0.051255	1.762390	2.404800	{'GDX':1,'GLD':-1.5077}	long
2020-11-02 00:00:00-05:00	1.408221	1.533414	0.001796	1.815039	1.815039	{'GDX':0,'GLD':0}	short
2020-11-06 00:00:00-05:00	1.455440	1.676329	0.074501	1.815039	1.297001	{'GDX':-1,'GLD':1.5077}	long
2020-11-11 00:00:00-05:00	1.386358	1.492351	-0.006760	1.894862	1.894862	{'GDX':0,'GLD':0}	short
2020-11-19 00:00:00-05:00	1.383224	1.423913	-0.070537	1.894862	2.556435	{'GDX':1,'GLD':-1.5077}	long
2020-12-01 00:00:00-05:00	1.347195	1.441224	0.000344	1.966494	1.966494	{'GDX':0,'GLD':0}	short
2020-12-07 00:00:00-05:00	1.369415	1.421095	-0.052824	1.966494	2.610066	{'GDX':1,'GLD':-1.5077}	long
2021-03-16 00:00:00-04:00	1.288469	1.355878	0.002315	2.023319	2.023319	{'GDX':0,'GLD':0}	short
2021-03-25 00:00:00-04:00	1.294023	1.300326	-0.061495	2.023319	2.673992	{'GDX':1,'GLD':-1.5077}	long
2021-04-05 00:00:00-04:00	1.282081	1.352254	0.008191	2.093253	2.093253	{'GDX':0,'GLD':0}	short
2021-04-22 00:00:00-04:00	1.326165	1.474638	0.065027	2.093253	1.568432	{'GDX':-1,'GLD':1.5077}	long
2021-05-03 00:00:00-04:00	1.330133	1.408213	-0.007298	2.165660	2.165660	{'GDX':0,'GLD':0}	short
2021-05-17 00:00:00-04:00	1.375288	1.540660	0.058009	2.165660	1.632799	{'GDX':-1,'GLD':1.5077}	long
2021-06-21 00:00:00-04:00	1.321482	1.395382	-0.007267	2.229816	2.229816	{'GDX':0,'GLD':0}	short
2021-07-09 00:00:00-04:00	1.339894	1.360709	-0.069316	2.229816	2.889265	{'GDX':1,'GLD':-1.5077}	long

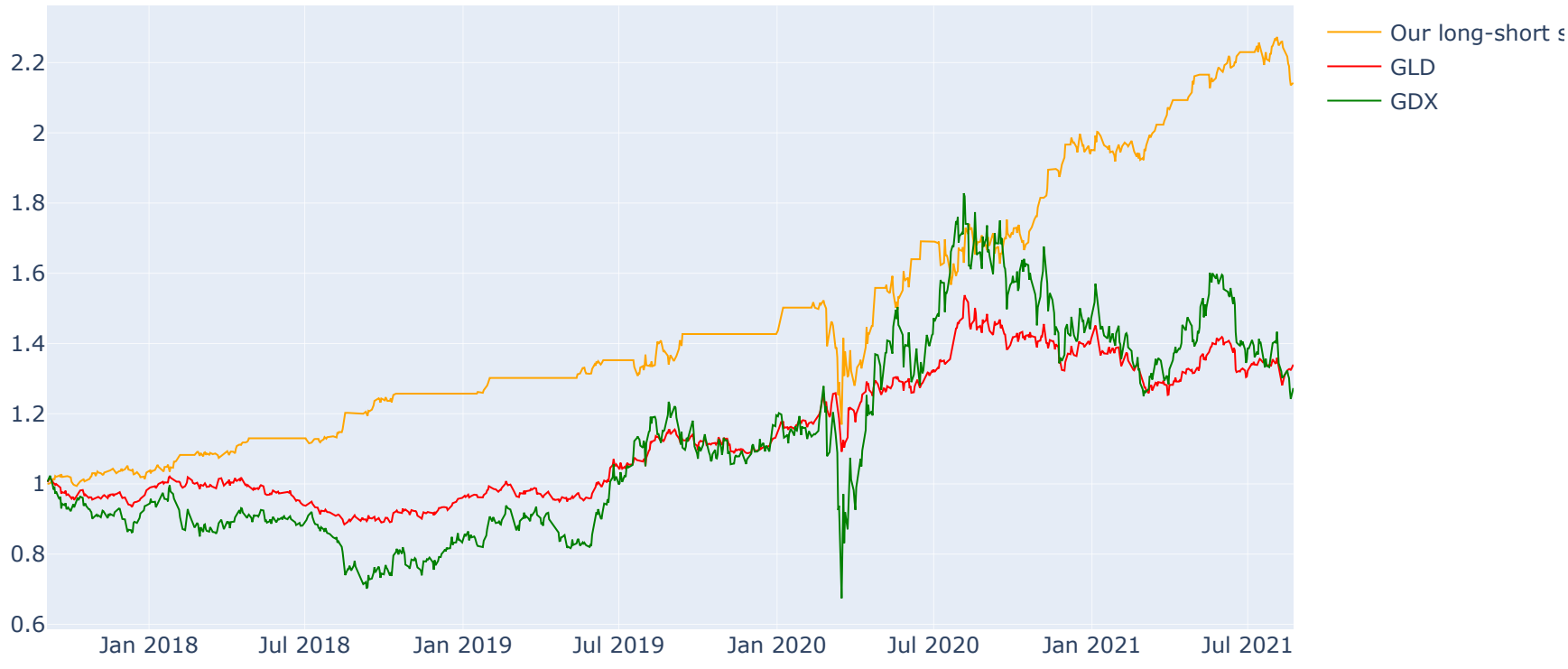
4.1 Performance comparison (vs holding GDX or GLD only):

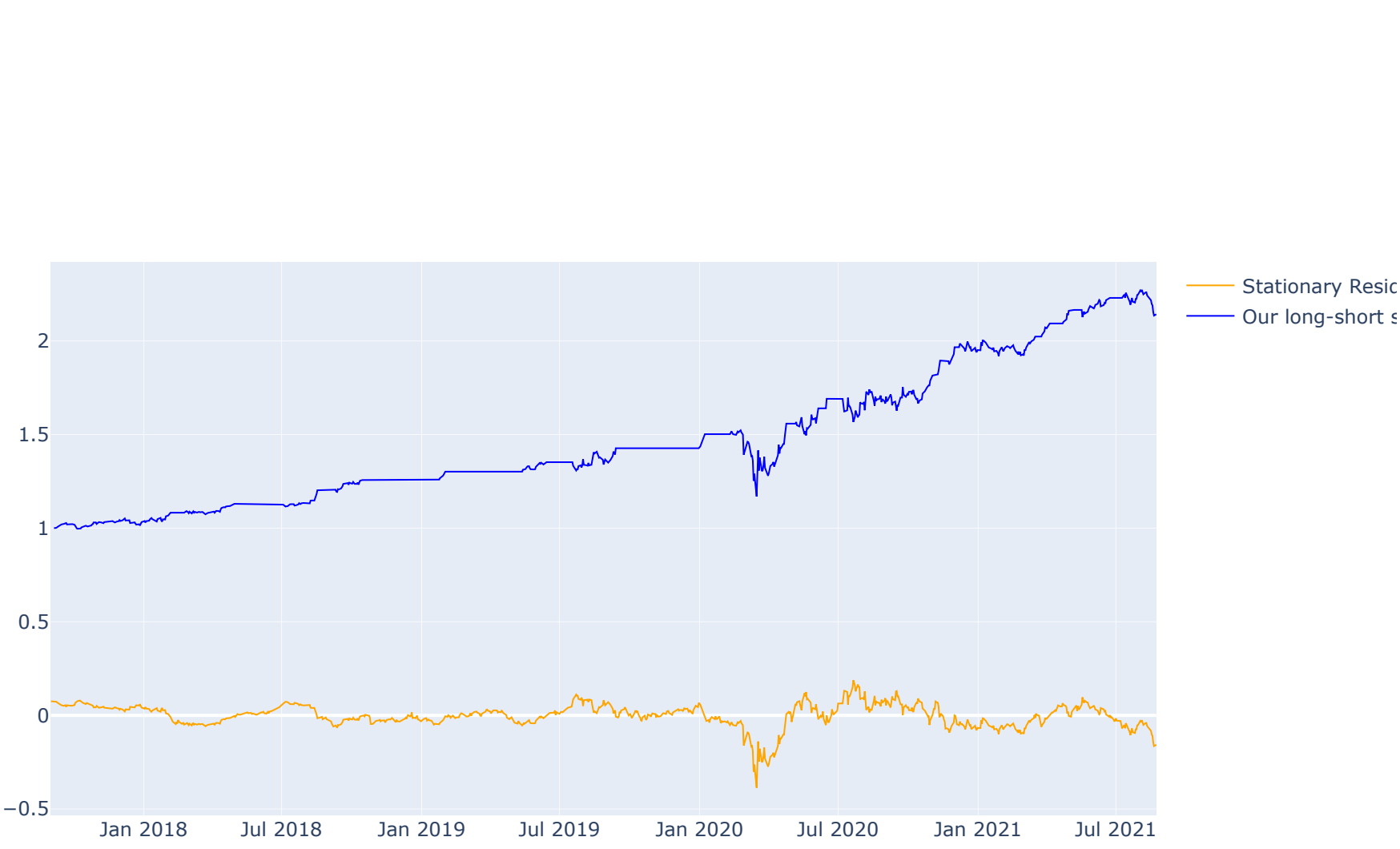
NOTICE: a copy of the data generated by the code

below is a plot of the two instruments together with the algo, assuming they all start from 1 dollar. As we can see our strategy is better in terms of stability as well as absolute return for the last 1000 days(if what i did was correct). Again, this is purely assuming no trading cost, no short-selling cost!

In [41]:

```
import plotly.graph_objects as go
from plotly.subplots import make_subplots

GLD = ('GLD', 'open_normalized')
GDX = ('GDX', 'open_normalized')

go.Figure(data=[
    go.Scatter(x=test_data.index, y=test_data.total_value, line=dict(color='orange', width=1), name="Our long-short
    go.Scatter(x=test_data.index, y=test_data[GLD], line=dict(color='red', width=1), name=f"GLD"),
    go.Scatter(x=test_data.index, y=test_data[GDX], line=dict(color='green', width=1), name=f"GDX"),
]).show()

Z_upper = 1.6
Z_lower = 1.11
mu = -0.0056
sigma = 0.0396

upper_bound = mu + Z_upper*sigma
lower_bound = mu - Z_lower*sigma

go.Figure(data=[
    go.Scatter(x=stationary_residual.index, y=stationary_residual, line=dict(color='orange', width=1), name="Statior
    go.Scatter(x=test_data.index, y=test_data.total_value, line=dict(color='blue', width=1), name="Our long-short st
]).show()
```





4.2 More analysis on the strategy (from pyfolio) and some observations

1. A long-short strategy on GDX and GLD is a better compared with just longing GLD, GDX on paper (in-sample test).
2. The largest drawback happened from 2020 Feb to May when the global financial markets crashed by global pandemic, where the mean-reverting strategy doesn't seem to be effective, not a total stranger from history!
3. our strategy actually looks better when it turns to return, stability as indicated by the following table,

	A	B	C	D	E	F	G	H
1		**Annual return**	**Annual volatility**	**Cumulative returns**	**Sharpe ratio**	**Max drawdown**	**Sortino ratio**	**Daily value at risk**
2	GLD	7.5%	14.2%	33.3%	0.58	-18.5%	0.82	-1.8%
3	GDX	6.1%	40.8%	26.5%	0.35	-47.3%	0.51	-5.1%
4	our strategy	21.2%	17.8%	114.2%	1.17	-23.2%	1.78	-2.2%

4. Again, this assumes no trading cost, no short-selling cost, and fractional share is available.

4.2.1 performance analysis on the pair trading strategy

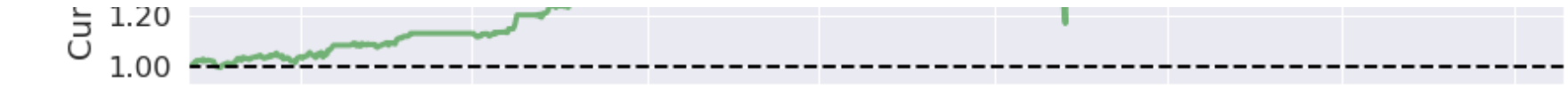
In [38]:
```
1 pf.create_full_tear_sheet(test_data.total_value.pct_change())
```

Start date	2017-09-05
End date	2021-08-23
Total months	47

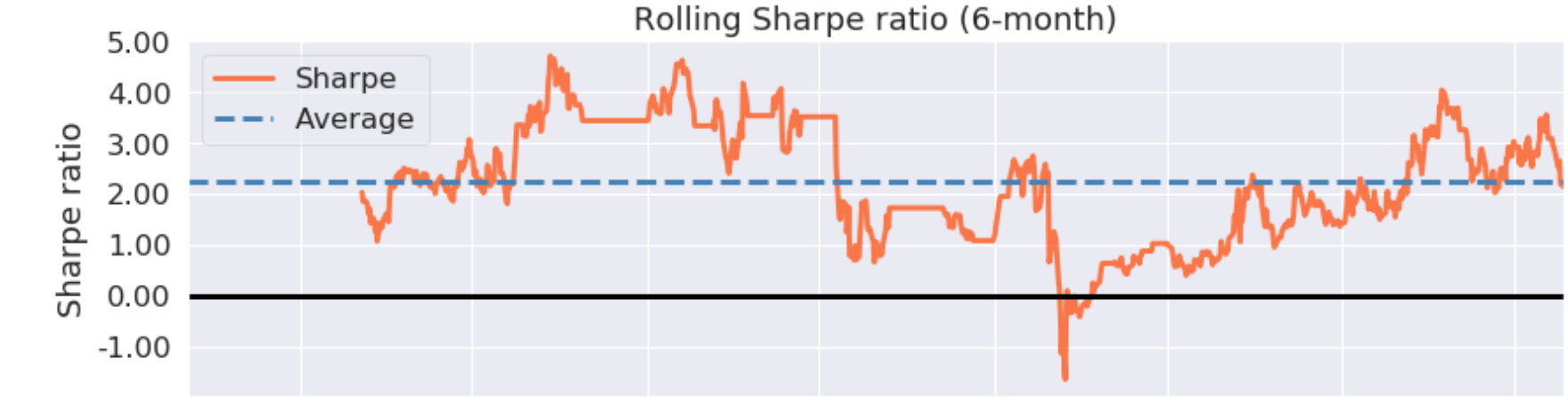
	Backtest
Annual return	21.2%
Cumulative returns	114.2%
Annual volatility	17.8%
Sharpe ratio	1.17
Calmar ratio	0.91
Stability	0.95
Max drawdown	-23.2%
Omega ratio	1.40
Sortino ratio	1.78
Skew	NaN
Kurtosis	NaN
Tail ratio	1.37
Daily value at risk	-2.2%

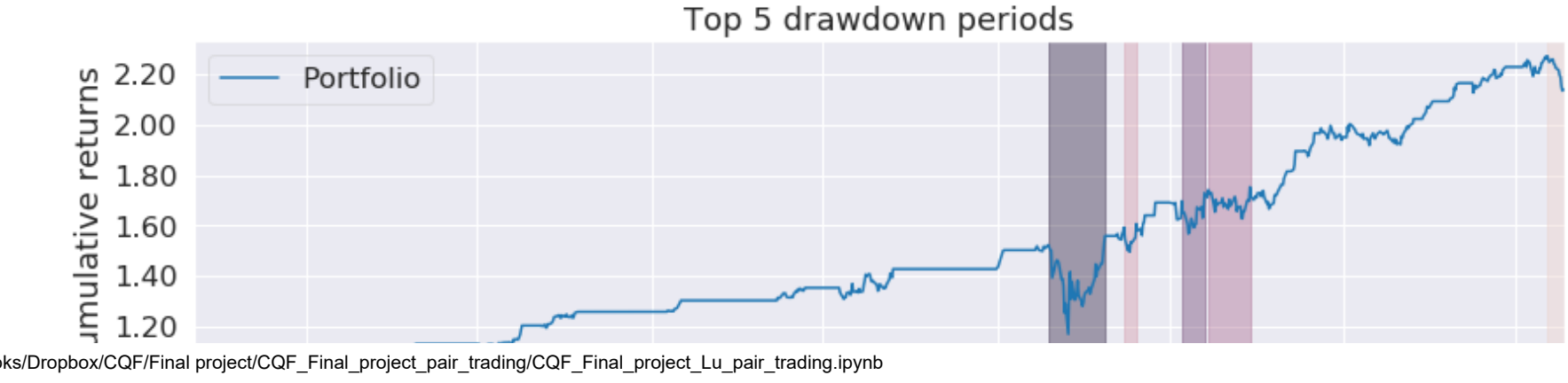
Worst drawdown periods	Net drawdown in %	Peak date	Valley date	Recovery date	Duration
0	23.23	2020-02-24	2020-03-16	2020-04-23	44
1	7.62	2020-07-14	2020-07-21	2020-08-07	19
2	6.53	2020-08-11	2020-09-16	2020-09-24	33
3	6.08	2020-05-14	2020-05-20	2020-05-27	10
4	6.05	2021-08-04	2021-08-20	NaT	NaN

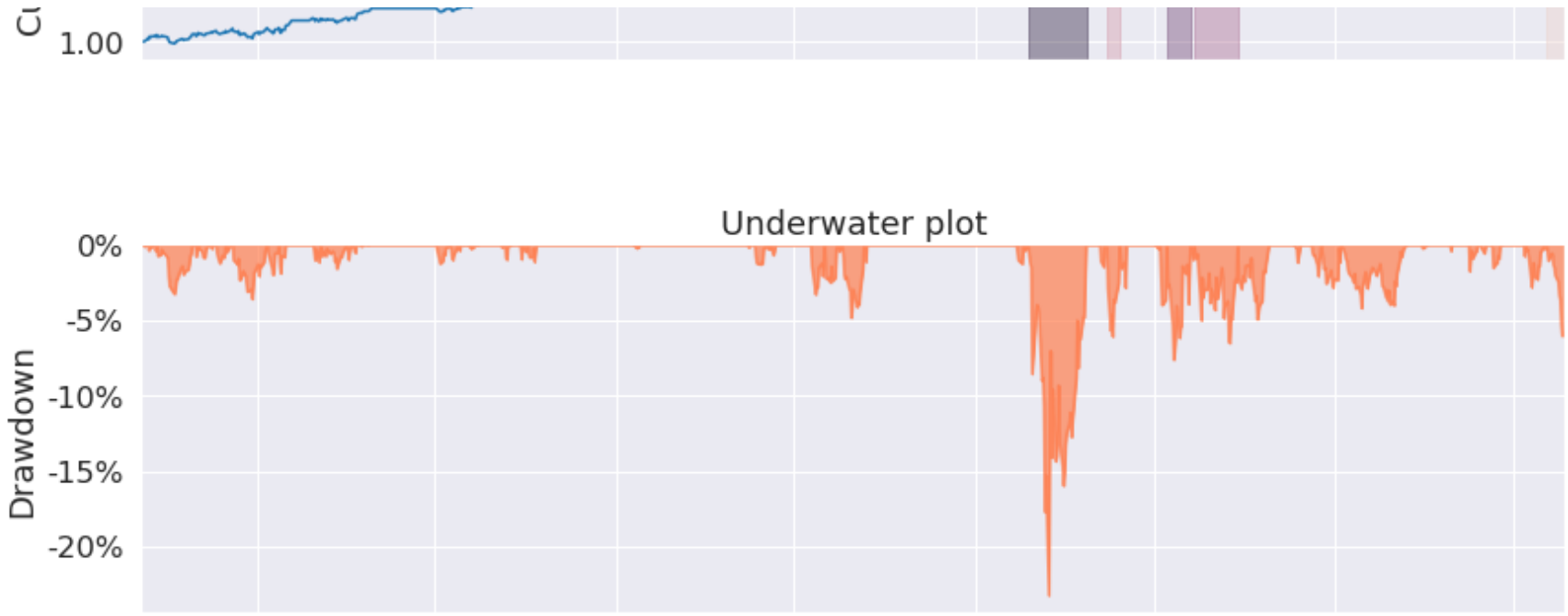
Stress Events	mean	min	max
New Normal	0.08%	-9.41%	15.78%

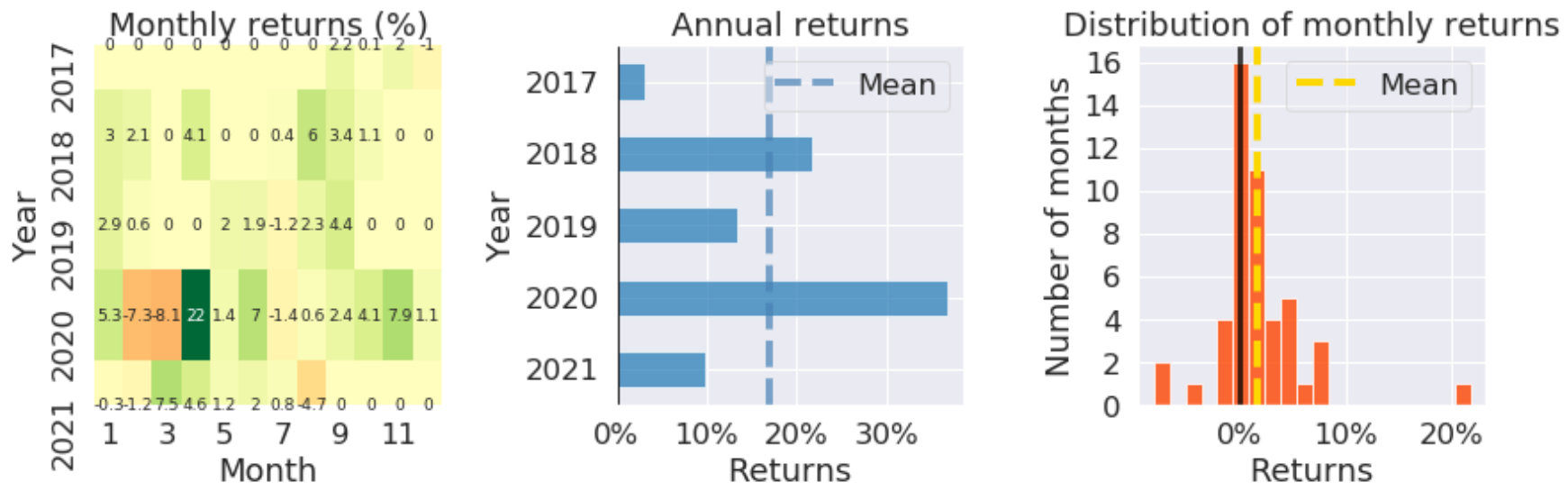

Cur
1.20
1.00

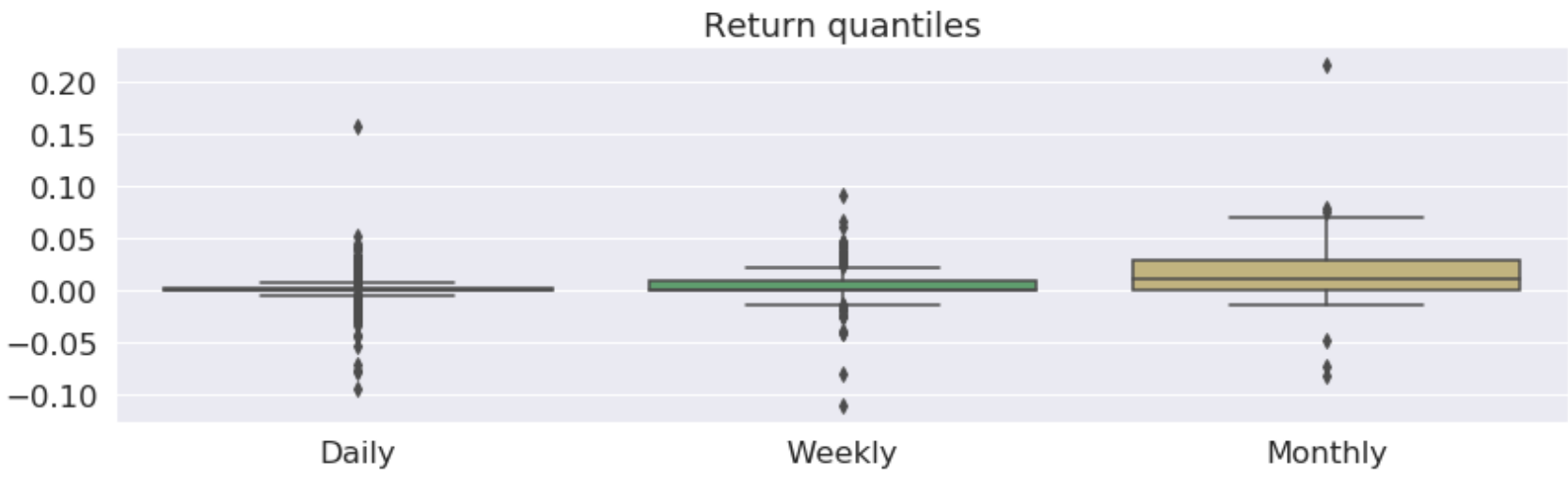

Cumulative returns on logarithmic scale
Backtest
Cumulative returns
2.2 × 10^0
2 × 10^0
1.8 × 10^0
1.6 × 10^0
1.4 × 10^0
1.2 × 10^0
1.00
Returns
0.15
0.10
0.05
0.00
−0.05
−0.10
Rolling volatility (6-month)
Volatility
Average volatility
Volatility
0.40
0.30
0.20
0.10
0.00

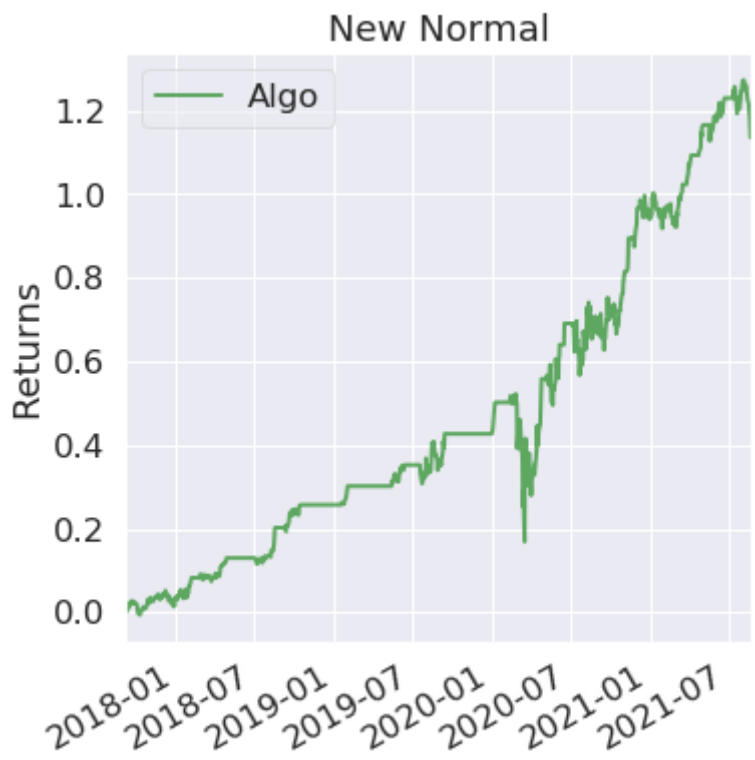

Rolling Sharpe ratio (6-month)
Sharpe
Average
Sharpe ratio
5.00
4.00
3.00
2.00
1.00
0.00
-1.00


Top 5 drawdown periods
Portfolio
umulative returns
2.20
2.00
1.80
1.60
1.40
1.20









4.2.2 impact of short selling and trading cost:

short-selling cost:

Sorry haven't got time to finish this part with solid code/math.. but just wanted to put quick discussion on it in a not so scientific way, recall from section 4.0, we have a trading activity records available, based on that we can assume that borrowing cost (https://www.newyorkfed.org/markets/reference-rates/obfr (https://www.newyorkfed.org/markets/reference-rates/obfr), which the industry uses as a short selling cost and varies around 8% nowadays but it does jumps up to 20% in financial crisis), we can reasonably make some inference that the actual annual returns would be less than 21.2% based on the short-selling cost at specific time as well as how long the short-selling would be

trading cost:

nowadaysmany brokers offer 0 commission for stock trading even for individual traders (robinhood traders eg). if there is trading cost, different clients would face different amount of commission. Guess we could account for the trading cost as a fixed cost or percentage charge, and it's easier to get this analysis based on the trading activity cost.

4.2.3 performance analysis on GLD, GDX

In [68]:
```
import pyfolio as pf
print("performance analysis of GLD: ")
pf.show_perf_stats(test_data[GLD].pct_change())
print("performance analysis of GDX: ")
pf.show_perf_stats(test_data[GDX].pct_change())
```

```
performance analysis of GLD:
```

Start date	2017-09-05
End date	2021-08-23
Total months	47

	Backtest
Annual return	7.5%
Cumulative returns	33.3%
Annual volatility	14.2%
Sharpe ratio	0.58
Calmar ratio	0.41
Stability	0.79
Max drawdown	-18.5%
Omega ratio	1.11
Sortino ratio	0.82
Skew	NaN
Kurtosis	NaN
Tail ratio	1.04
Daily value at risk	-1.8%

```
performance analysis of GDX:
```

Start date	2017-09-05
End date	2021-08-23
Total months	47

	Backtest
Annual return	6.1%
Cumulative returns	26.5%
Annual volatility	40.8%
Sharpe ratio	0.35
Calmar ratio	0.13
Stability	0.70
Max drawdown	-47.3%
Omega ratio	1.07
Sortino ratio	0.51
Skew	NaN
Kurtosis	NaN
Tail ratio	1.14
Daily value at risk	-5.1%

In [ ]:
```
1
```