ARCH/GARCH Volatility Forecasting

What is volatility

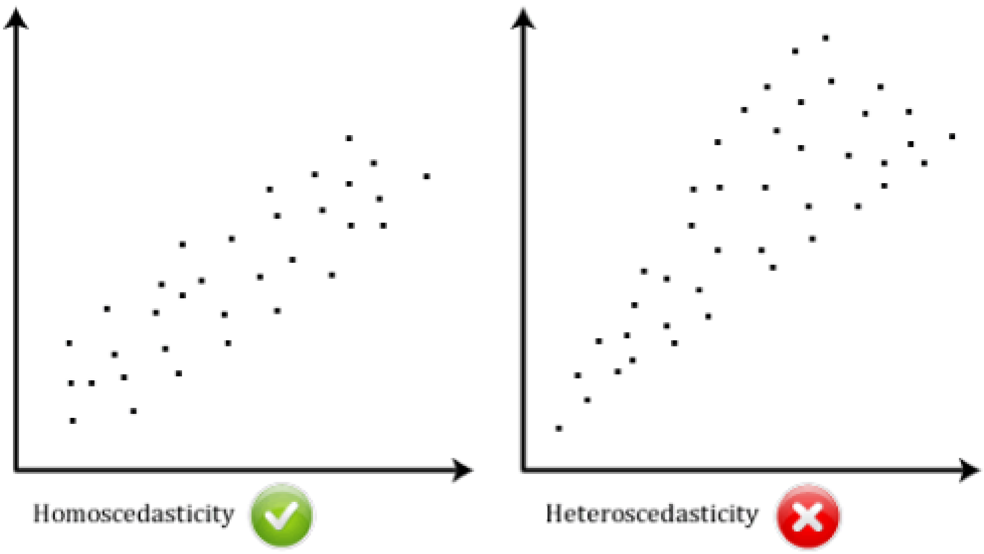
- Describes the dispersion of financial asset returns over time.
- Often computed as the standard deviation or variance of price returns.
- The higher the volatility, the riskier a financial asset.

The challenge of volatility modeling

`Heteroskedasticity:`

- In ancient Greek: "different" (hetero) + "dispersion" (skedasis)
- A time series demonstrates varying volatility systematically over time.

Homoskedasticity vs Heteroskedasticity



Imports & Settings

In [1]:

```
import datetime as dt
import sys
import numpy as np
from numpy import cumsum, log, polyfit, sqrt, std, subtract
from numpy.random import randn
import pandas as pd
from pandas_datareader import data as web
import seaborn as sns
from pylab import rcParams
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from arch import arch_model
from numpy.linalg import LinAlgError
from scipy import stats
import statsmodels.api as sm
import statsmodels.tsa.api as tsa
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import acf, q_stat, adfuller
from sklearn.metrics import mean_squared_error
from scipy.stats import probplot, moment
from arch import arch_model
from arch.univariate import ConstantMean, GARCH, Normal
from sklearn.model_selection import TimeSeriesSplit
import warnings
```

In [2]:

```
%matplotlib inline
pd.set_option('display.max_columns', None)
warnings.filterwarnings('ignore')
sns.set(style="darkgrid", color_codes=True)
rcParams['figure.figsize'] = 8,4
```

Hurst Exponent function

The `Hurst Exponent` is a statistical measure used to classify time series and infer the level of difficulty in predicting and choosing an appropriate model for the series at hand. The Hurst exponent is used as a measure of long-term memory of time series. It relates to the autocorrelations of the time series, and the rate at which these decrease as the lag between pairs of values increases.

- Value near `0.5` indicates a random series.
- Value near `0` indicates a mean reverting series.
- Value near `1` indicates a trending series.

In [3]:

```
def hurst(ts):
    """Returns the Hurst Exponent of the time series vector ts"""
    # Create the range of lag values
    lags = range(2, 100)

    # Calculate the array of the variances of the lagged differences
    tau = [sqrt(std(subtract(ts[lag:], ts[:-lag]))) for lag in lags]

    # Use a linear fit to estimate the Hurst Exponent
    poly = polyfit(log(lags), log(tau), 1)

    # Return the Hurst exponent from the polyfit output
    return poly[0]*2.0
```

Correlogram Plot

In [4]:

```
def plot_correlogram(x, lags=None, title=None):
    lags = min(10, int(len(x)/5)) if lags is None else lags
    fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(12, 8))
    x.plot(ax=axes[0][0])
    q_p = np.max(q_stat(acf(x, nlags=lags), len(x))[1])
    stats = f'Q-Stat: {np.max(q_p):>8.2f}\nADF: {adfuller(x)[1]:>11.2f} \nHurst: {round
(hurst(x.values),2)}'
    axes[0][0].text(x=.02, y=.85, s=stats, transform=axes[0][0].transAxes)
    probplot(x, plot=axes[0][1])
    mean, var, skew, kurtosis = moment(x, moment=[1, 2, 3, 4])
    s = f'Mean: {mean:>12.2f}\nSD: {np.sqrt(var):>16.2f}\nSkew: {skew:12.2f}\nKurtosis:
{kurtosis:9.2f}'
    axes[0][1].text(x=.02, y=.75, s=s, transform=axes[0][1].transAxes)
    plot_acf(x=x, lags=lags, zero=False, ax=axes[1][0])
    plot_pacf(x, lags=lags, zero=False, ax=axes[1][1])
    axes[1][0].set_xlabel('Lag')
    axes[1][1].set_xlabel('Lag')
    fig.suptitle(title, fontsize=20)
    fig.tight_layout()
    fig.subplots_adjust(top=.9)
```

Download S&P 500 Index Data

We will use daily `S&P 500` returns from `2005-2020` to demonstrate the usage of a `GARCH` model

In [5]:

```
start = pd.Timestamp('2005-01-01')
end = pd.Timestamp('2020-04-09')

sp_data = web.DataReader('SPY', 'yahoo', start, end)\
      [['High','Low','Open','Close','Volume','Adj Close']]

sp_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 3844 entries, 2005-01-03 to 2020-04-09
Data columns (total 6 columns):
High         3844 non-null float64
Low          3844 non-null float64
Open         3844 non-null float64
Close        3844 non-null float64
Volume       3844 non-null float64
Adj Close    3844 non-null float64
dtypes: float64(6)
memory usage: 210.2 KB
```

In [6]:

```
sp_data.head()
```

Out[6]:

	High	Low	Open	Close	Volume	Adj Close
Date						
2005-01-03	121.760002	119.900002	121.559998	120.300003	55748000.0	88.533607
2005-01-04	120.540001	118.440002	120.459999	118.830002	69167600.0	87.451752
2005-01-05	119.250000	118.000000	118.739998	118.010002	65667300.0	86.848251
2005-01-06	119.150002	118.260002	118.440002	118.610001	47814700.0	87.289810
2005-01-07	119.230003	118.129997	118.970001	118.440002	55847700.0	87.164726

Observe volatility clustering

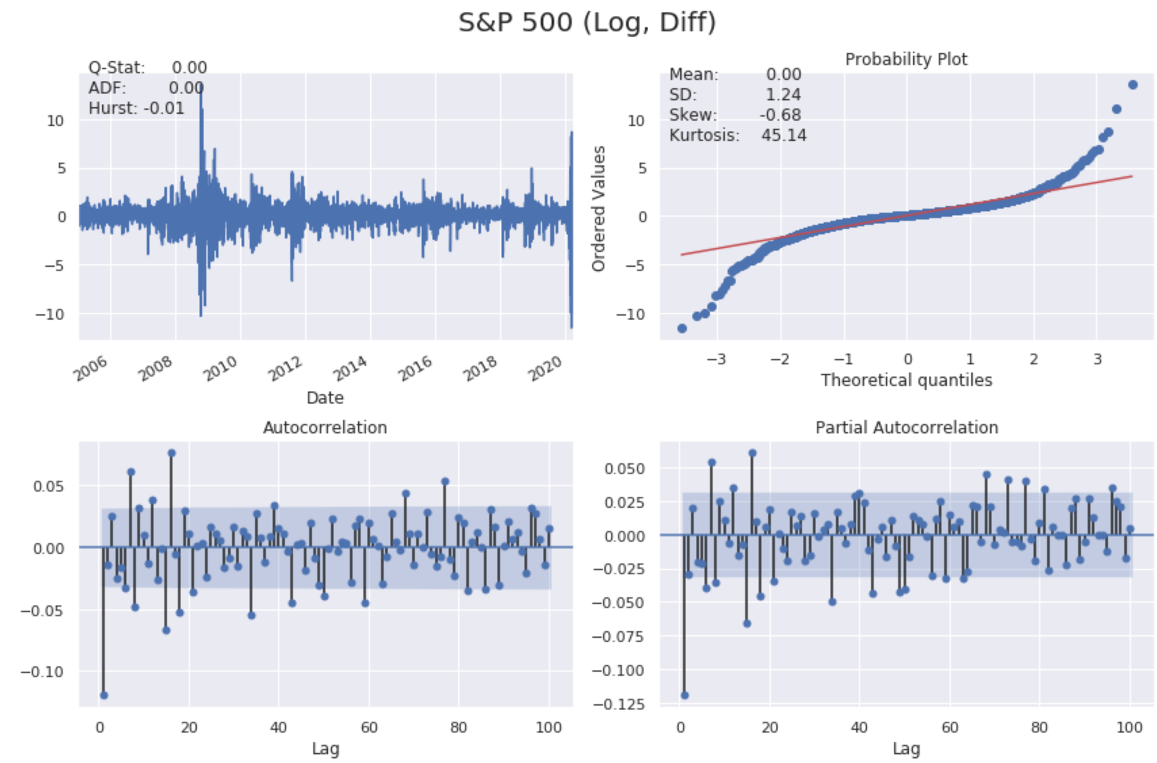
`Volatility clustering` refers to the observation that "large changes tend to be followed by large changes, of either sign, and small changes tend to be followed by small changes.

- Volatility clustering is frequently observed in financial market data, and it poses a challenge for time series modeling.

with the `S&P 500` daily price dataset we calculate daily returns as the percentage price changes, plot the results and observe its behavior over time.

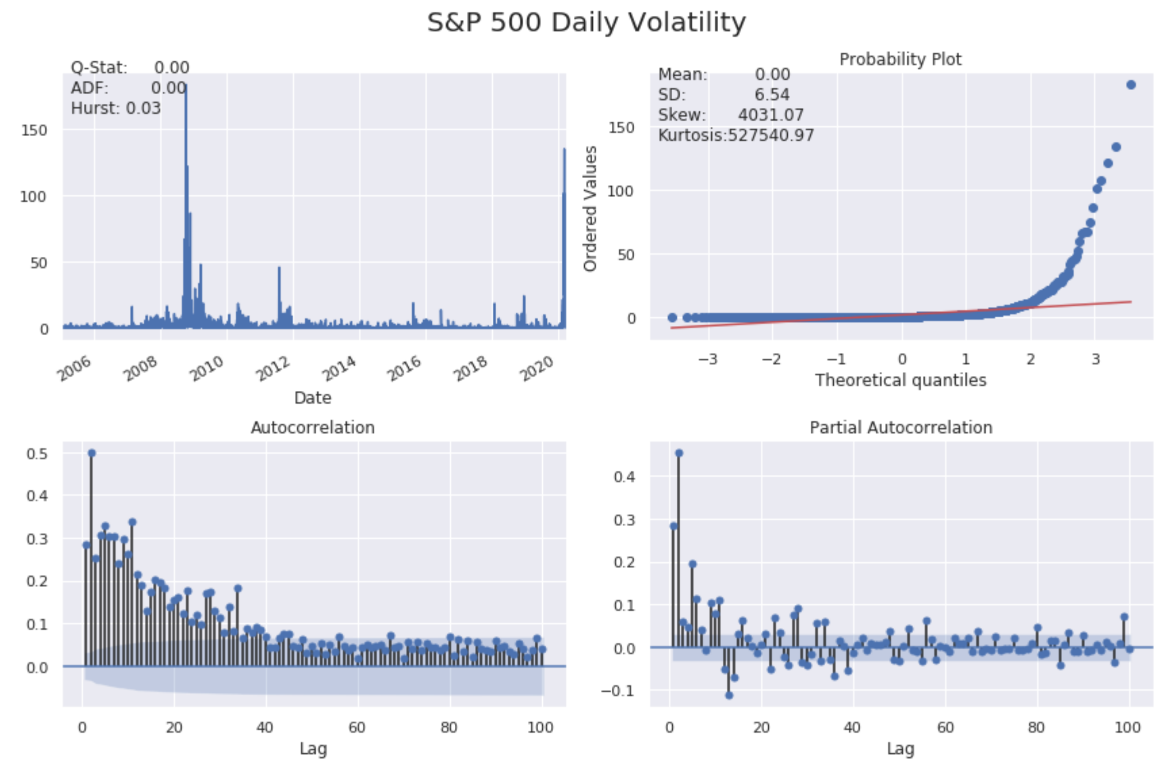
In [7]:

```
# Calculate daily returns as percentage price changes
sp_data['Return'] = 100 * (sp_data['Close'].pct_change())
sp_data['Log_Return'] = np.log(sp_data['Close']).diff().mul(100) # rescale to faciliate optimization
sp_data = sp_data.dropna()

# Plot ACF, PACF and Q-Q plot and get ADF p-value of series
plot_correlogram(sp_data['Log_Return'], lags=100, title='S&P 500 (Log, Diff)')
```



In [8]:

```
plot_correlogram(sp_data['Log_Return'].sub(sp_data['Log_Return'].mean()).pow(2), lags=100, title='S&P 500 Daily Volatility')
```



Calculate volatility

We compute and convert volatility of price returns in Python.

Firstly, we compute the daily volatility as the standard deviation of price returns. Then convert the daily volatility to monthly and annual volatility.

In [9]:

```
# Calculate daily std of returns
std_daily = sp_data['Return'].std()
print(f'Daily volatility: {round(std_daily,2)}%')

# Convert daily volatility to monthly volatility
std_monthly = np.sqrt(21) * std_daily
print(f'\nMonthly volatility: {round(std_monthly,2)}%')

# Convert daily volatility to annaul volatility
std_annual = np.sqrt(252) * std_daily
print(f'\nAnnual volatility: {round(std_annual,2)}%')
```

```
Daily volatility: 1.24%

Monthly volatility: 5.67%

Annual volatility: 19.63%
```

ARCH and GARCH

First came the `ARCH`

- Auto Regressive Conditional Heteroskedasticity
- Developed by Robert F. Engle (Nobel prize laureate `2003`)

Then came the `GARCH`

- "Generalized" `ARCH`
- Developed by Tim Bollerslev (Robert F. Engle's student)

Model notations

- Expected return:
$$\mu = Expected|r_t|I(t-1)$$
- Expected volatility:
$$\sigma^2 = Expected[(r_t - \mu_t)^2|I(t-1)]$$
- Residual (prediction error):
$$r_t = \mu + \epsilon_t$$
- Volatility is related to the residuals:
$$\epsilon_t = \sigma_t * \zeta(WhiteNoise)$$
- `White noise (z):` Uncorrelated random variables with a zero mean and a finite variance

Model intuition

- `Autoregressive` : predict future behavior based on past behavior.
- Volatility as a weighted average of past information.

GARCH(1,1) parameter constraints

- All parameters are non-negative, so the variance cannot be negative.
$$\omega, \alpha, \beta >= 0$$
- Model estimations are "mean-reverting" to the long-run variance.
$$\alpha + \beta < 1$$
- long-run variance:
$$\omega \;/\; (1 - \alpha - \beta)$$

GARCH(1,1) parameter dynamics

- The larger the `α` , the bigger the immediate impact of the shock
- The larger the `β` , the longer the duration of the impact

Given the `GARCH(1,1)` model equation as:
$$GARCH(1,1): \sigma_t^2 \;= \omega + \alpha\epsilon_{t-1}^2 + \beta\sigma_{t-1}^2$$

Intuitively, `GARCH` variance forecast can be interpreted as a weighted average of three different variance forecasts.

- One is a constant variance that corresponds to the long run average.
- The second is the new information that was not available when the previous forecast was made.
- The third is the forecast that was made in the previous period.

The weights on these three forecasts determine how fast the variance changes with new information and how fast it reverts to its long run mean.

Simulate ARCH and GARCH series

We will simulate an `ARCH(1)` and `GARCH(1,1)` time series respectively using a function `simulate_GARCH(n, omega, alpha, beta = 0)` .

Recall the difference between an `ARCH(1)` and a `GARCH(1,1`) model is: besides an autoregressive component of `α` multiplying `lag-1` residual squared, a GARCH model includes a moving average component of `β` multiplying `lag-1` variance.

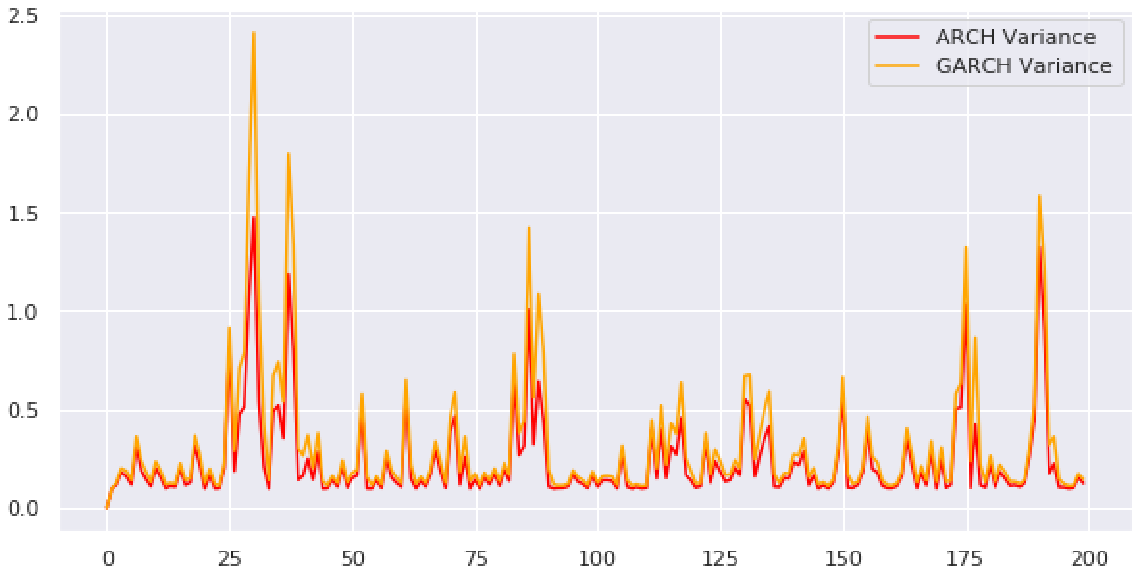
The function will simulate an `ARCH/GARCH` series based on `n` (number of simulations), `omega` , `alpha` , and beta (`0` by default) you specify. It will return simulated residuals and variances.

In [10]:

```
def simulate_GARCH(n, omega, alpha, beta = 0):
    np.random.seed(4)
    # Initialize the parameters
    white_noise = np.random.normal(size = n)
    resid = np.zeros_like(white_noise)
    variance = np.zeros_like(white_noise)

    for t in range(1, n):
        # Simulate the variance (sigma squared)
        variance[t] = omega + alpha * resid[t-1]**2 + beta * variance[t-1]
        # Simulate the residuals
        resid[t] = np.sqrt(variance[t]) * white_noise[t]

    return resid, variance
```

In [11]:

```
# Simulate a ARCH(1) series
arch_resid, arch_variance = simulate_GARCH(n= 200,
                                           omega = 0.1, alpha = 0.7)
# Simulate a GARCH(1,1) series
garch_resid, garch_variance = simulate_GARCH(n= 200,
                                             omega = 0.1, alpha = 0.7,
                                             beta = 0.1)

# Plot the ARCH variance
plt.figure(figsize=(10,5))
plt.plot(arch_variance, color = 'red', label = 'ARCH Variance')

# Plot the GARCH variance
plt.plot(garch_variance, color = 'orange', label = 'GARCH Variance')
plt.legend()
plt.show()
```



Observe the impact of model parameters

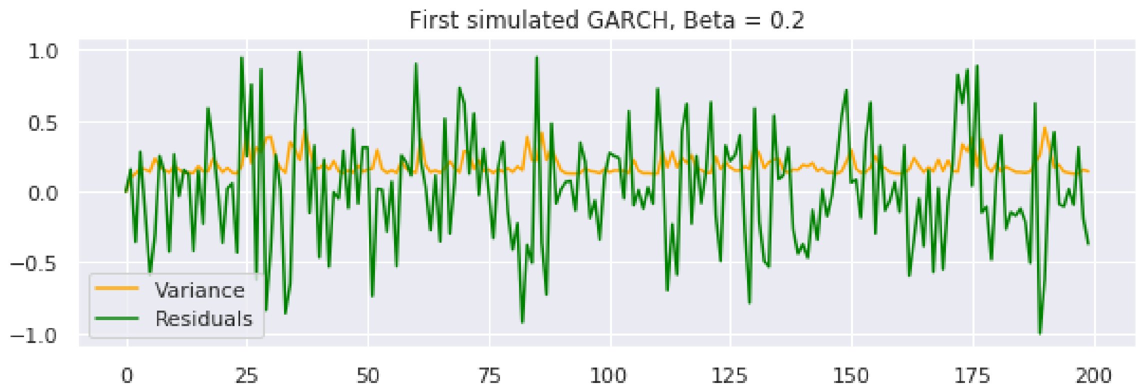
We will call the function `simulate_GARCH()` again, and study the impact of GARCH model parameters on simulated results.

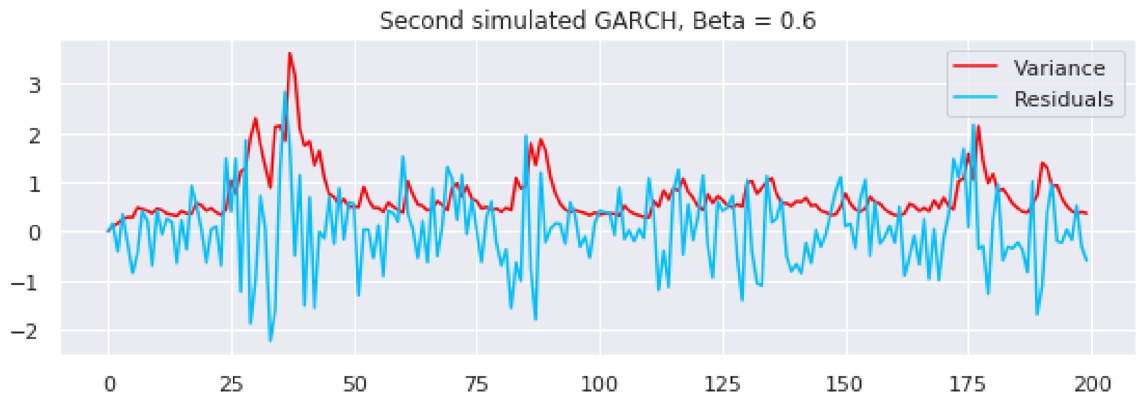
Specifically, we will simulate two `GARCH(1,1)` time series, they have the same `omega` and `alpha` , but different `beta` as input.

Recall in `GARCH(1,1)` , since β is the coefficient of `lag-1` variance, if the α is fixed, the larger the β , the longer the duration of the impact. In other words, high or low volatility periods tend to persist. Pay attention to the plotted results and see whether we can verify the β impact.

In [12]:

```
# First simulated GARCH
plt.figure(figsize=(10,3))
sim_resid, sim_variance = simulate_GARCH(n = 200,  omega = 0.1, alpha = 0.3, beta = 0.2
)
plt.plot(sim_variance, color = 'orange', label = 'Variance')
plt.plot(sim_resid, color = 'green', label = 'Residuals')
plt.title('First simulated GARCH, Beta = 0.2')
plt.legend(loc='best')
plt.show()

# Second simulated GARCH
plt.figure(figsize=(10,3))
sim_resid, sim_variance = simulate_GARCH(n = 200,  omega = 0.1, alpha = 0.3, beta = 0.6
)
plt.plot(sim_variance, color = 'red', label = 'Variance')
plt.plot(sim_resid, color = 'deepskyblue', label = 'Residuals')
plt.title('Second simulated GARCH, Beta = 0.6')
plt.legend(loc='best')
plt.show()
```





Implement a basic GARCH model

We will get familiar with the Python `arch` package, and use its functions such as `arch_model()` to implement a `GARCH(1,1)` model.

First define a basic `GARCH(1,1)` model, then fit the model, review the model fitting summary, and plot the results.

In [13]:

```
# Specify GARCH model assumptions
basic_gm = arch_model(sp_data['Return'], p = 1, q = 1,
                      mean = 'constant', vol = 'GARCH', dist = 'normal')
# Fit the model
gm_result = basic_gm.fit(update_freq = 4)
```

```
Iteration:      4,   Func. Count:     36,   Neg. LLF: 4997.204823442513
Iteration:      8,   Func. Count:     64,   Neg. LLF: 4993.358477465824
Iteration:     12,   Func. Count:     88,   Neg. LLF: 4992.874678320535
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 4992.874653095375
            Iterations: 13
            Function evaluations: 94
            Gradient evaluations: 13
```

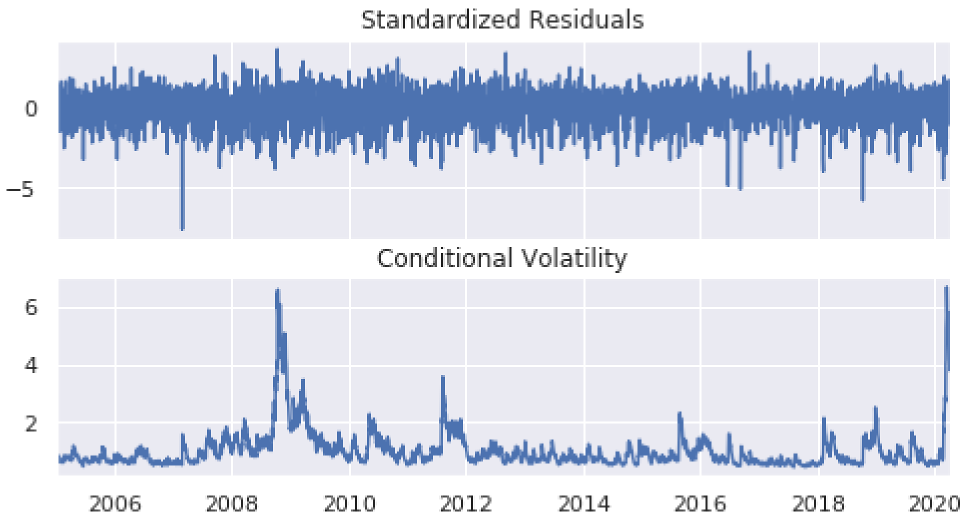
In [14]:

```
# Display model fitting summary
print(gm_result.summary())
```

```
                     Constant Mean - GARCH Model Results
==========================================================================
====
Dep. Variable:                      Return   R-squared:                  -
0.001
Mean Model:                  Constant Mean   Adj. R-squared:             -
0.001
Vol Model:                           GARCH   Log-Likelihood:          -499
2.87
Distribution:                       Normal   AIC:                      999
3.75
Method:                 Maximum Likelihood   BIC:                      100
18.8
                                             No. Observations:
3843
Date:                     Mon, Apr 13 2020   Df Residuals:
3839
Time:                             20:55:47   Df Model:
4
                                 Mean Model
==========================================================================
==
                 coef    std err          t      P>|t|      95.0% Conf. In
t.
--------------------------------------------------------------------------
--
mu             0.0720  1.188e-02      6.062  1.343e-09 [4.873e-02,9.529e-0
2]
                              Volatility Model
==========================================================================
==
                 coef    std err          t      P>|t|      95.0% Conf. In
t.
--------------------------------------------------------------------------
--
omega          0.0268  5.823e-03      4.603  4.172e-06 [1.539e-02,3.821e-0
2]
alpha[1]       0.1414  1.592e-02      8.885  6.370e-19     [  0.110,  0.17
3]
beta[1]        0.8372  1.540e-02     54.376      0.000     [  0.807,  0.86
7]
==========================================================================
==

Covariance estimator: robust
```

In [15]:

```
# Plot fitted results
gm_result.plot()
plt.show()
```



Make forecast with GARCH models

We will practice making a basic volatility forecast.

We will call `.forecast()` to make a prediction. By default it produces a `1-step` ahead estimate. You can use `horizon = n` to specify longer forward periods.

In [16]:

```
# Make 5-period ahead forecast
gm_forecast = gm_result.forecast(horizon = 5)

# Print the forecast variance
print(gm_forecast.variance[-1:])
```

```
                  h.1        h.2        h.3        h.4        h.5
Date
2020-04-09  12.396423  12.157934  11.924549  11.696159  11.472656
```

`h.1` in row `"2020-04-09"` : is a 1-step ahead forecast made using data up to and including that date

Distribution assumptions

Why make assumptions?

- Volatility is not directly observable
- GARCH model use residuals as volatility shocks

$$r_t = \mu + t + \epsilon_t$$

- Volatility is related to the residuals:

$$\epsilon_t = \sigma_t * \zeta(WhiteNoise)$$

Standardized residuals

- Residual = predicted return - mean return

$$residuals = \epsilon_t = r_t - \mu_t$$

- Standardized residual = residual / return volatility

$$stdResid = \frac{\epsilon_t}{\sigma_t}$$

GARCH models make distribution assumptions about the residuals and the mean return. Financial time series data often does not follow a normal distribution. In financial time series it is much more likely to observe extreme positive and negative values that are far away from the mean. to improve a GARCH models distribution assumptions to be more representative of real financial data we can specify the models distribution assumption to be a Student's t-distribution. A Student's t-distribution is symmetric and bell shaped similar to a normal distribution but has fatter tails making it more prone to producing values that fall far away from its mean. The nu (ν) parameter indicates its shape the larger the ν the more peaked the curve becomes.

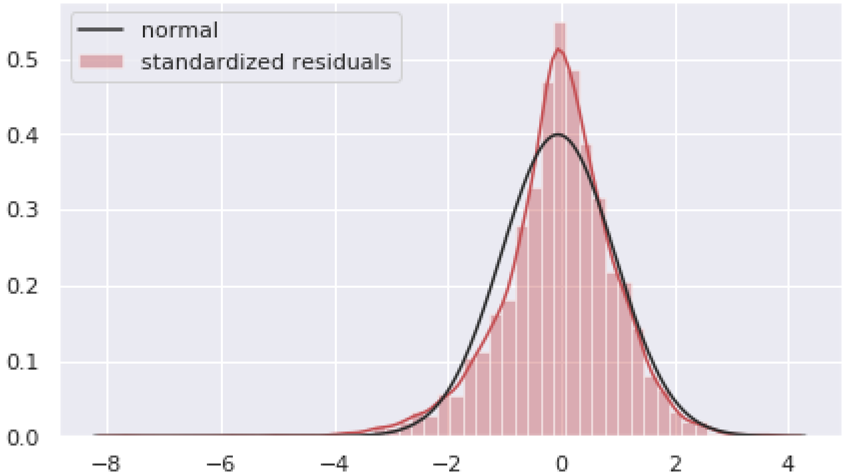
GARCH models enable one to specify the distribution assumptions of the standardized residuals. By default, a normal distribution is assumed, which has a symmetric, bell-shaped probability density curve. Other options include Student's t-distribution and skewed Student's t-distribution.

Plot distribution of standardized residuals

We will practice computing the standardized residuals from a fitted GARCH model, and then plot its histogram together with a normal fit distribution `normal_resid` .

In [17]:

```
# Obtain model estimated residuals and volatility
gm_resid = gm_result.resid
gm_std = gm_result.conditional_volatility

# Calculate the standardized residuals
gm_std_resid = gm_resid /gm_std

# Plot the histogram of the standardized residuals
plt.figure(figsize=(7,4))
sns.distplot(gm_std_resid, norm_hist=True, fit=stats.norm, bins=50, color='r')
plt.legend(('normal', 'standardized residuals'))
plt.show()
```



Fit a GARCH with skewed t-distribution

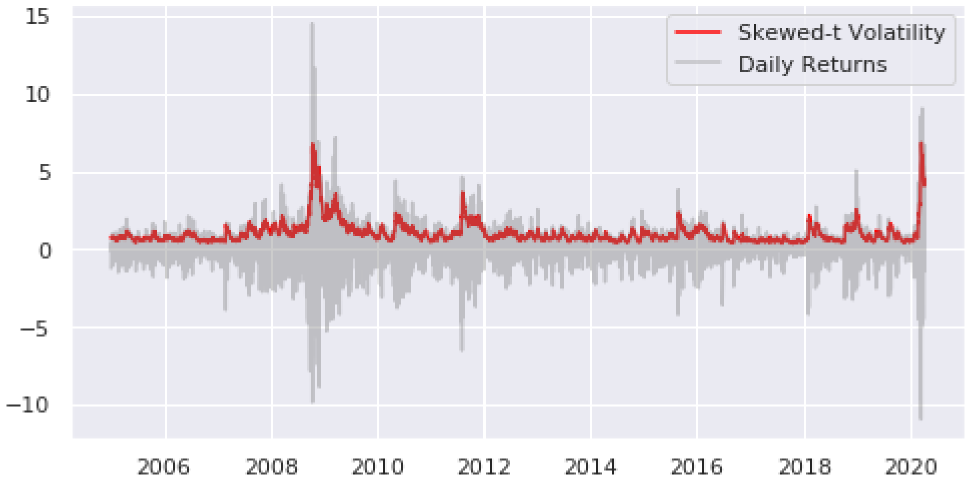
We will improve the GARCH model by using a skewed Student's t-distribution assumption. In addition.

In [18]:

```
# Specify GARCH model assumptions
skewt_gm = arch_model(sp_data['Return'], p = 1, q = 1, mean = 'constant', vol = 'GARCH'
, dist = 'skewt')

# Fit the model
skewt_result = skewt_gm.fit(disp = 'off')

# Get model estimated volatility
skewt_vol = skewt_result.conditional_volatility
```

In [19]:

```
# Plot model fitting results
plt.plot(skewt_vol, color = 'red', label = 'Skewed-t Volatility')
plt.plot(sp_data['Return'], color = 'grey',
         label = 'Daily Returns', alpha = 0.4)
plt.legend(loc = 'upper right')
plt.show()
```



Mean model specifications

- **Constant mean:** generally works well with most financial return data.
- **Autoregressive mean:** model the mean as an autoregressive (AR) process.
- **Zero mean:** Use when the mean has been modeled separately to fit the residuals of the separate model to estimate volatility (preferred method).

Here we model the log-returns of the S&P 500 data with an `ARMA` model and then fit the models residuals to estimate the volatility of the returns series with a `GARCH` model.

Searching over model orders to find the optimal number of lags

In [20]:

```
import pmdarima as pm

model = pm.auto_arima(sp_data['Log_Return'],

d=0, # non-seasonal difference order
start_p=1, # initial guess for p
start_q=1, # initial guess for q
max_p=4, # max value of p to test
max_q=4, # max value of q to test

seasonal=False, # is the time series seasonal

information_criterion='bic', # used to select best model
trace=True, # print results whilst training
error_action='ignore', # ignore orders that don't work
stepwise=True, # apply intelligent order search

)
```

```
Performing stepwise search to minimize bic
Fit ARIMA: (1, 0, 1)x(0, 0, 0, 0) (constant=True); AIC=12497.795, BIC=1252
2.811, Time=0.288 seconds
Fit ARIMA: (0, 0, 0)x(0, 0, 0, 0) (constant=True); AIC=12551.752, BIC=1256
4.260, Time=0.062 seconds
Fit ARIMA: (1, 0, 0)x(0, 0, 0, 0) (constant=True); AIC=12498.379, BIC=1251
7.141, Time=0.110 seconds
Fit ARIMA: (0, 0, 1)x(0, 0, 0, 0) (constant=True); AIC=12496.119, BIC=1251
4.881, Time=0.185 seconds
Fit ARIMA: (0, 0, 0)x(0, 0, 0, 0) (constant=False); AIC=12550.945, BIC=125
57.199, Time=0.037 seconds
Fit ARIMA: (0, 0, 2)x(0, 0, 0, 0) (constant=True); AIC=12497.705, BIC=1252
2.721, Time=0.453 seconds
Fit ARIMA: (1, 0, 2)x(0, 0, 0, 0) (constant=True); AIC=12493.967, BIC=1252
5.237, Time=1.152 seconds
Total fit time: 2.290 seconds
```

In [21]:

```
print(model.summary())
```

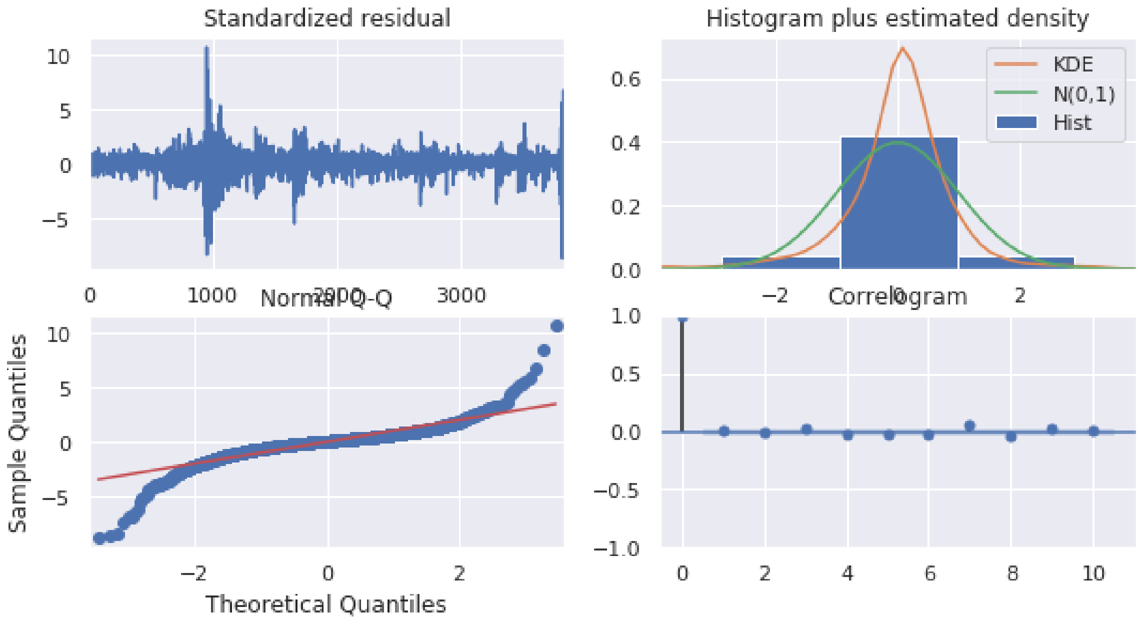
```
                               SARIMAX Results
==========================================================================
====
Dep. Variable:                      y   No. Observations:
3843
Model:               SARIMAX(0, 0, 1)   Log Likelihood               -624
5.060
Date:                Mon, 13 Apr 2020   AIC                          1249
6.119
Time:                        20:55:59   BIC                          1251
4.881
Sample:                             0   HQIC                         1250
2.783
                               - 3843
Covariance Type:                  opg
==========================================================================
====
                 coef    std err          z      P>|z|      [0.025      0.
975]
--------------------------------------------------------------------------
----
intercept      0.0218      0.018      1.180      0.238      -0.014
0.058
ma.L1         -0.1242      0.008    -16.509      0.000      -0.139      -
0.109
sigma2         1.5101      0.012    123.107      0.000       1.486
1.534
==========================================================================
=========
Ljung-Box (Q):                      115.36   Jarque-Bera (JB):
38226.99
Prob(Q):                              0.00   Prob(JB):
0.00
Heteroskedasticity (H):               0.58   Skew:
-0.58
Prob(H) (two-sided):                  0.00   Kurtosis:
18.41
==========================================================================
=========

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (com
plex-step).
```

In [22]:

```
# Fit best model
_arma_model = sm.tsa.SARIMAX(endog=sp_data['Log_Return'],order=(0, 0, 1))
_model_result = _arma_model.fit()
```

In [23]:

```
# Plot model residuals
_model_result.plot_diagnostics(figsize=(10, 5))
plt.show()
```



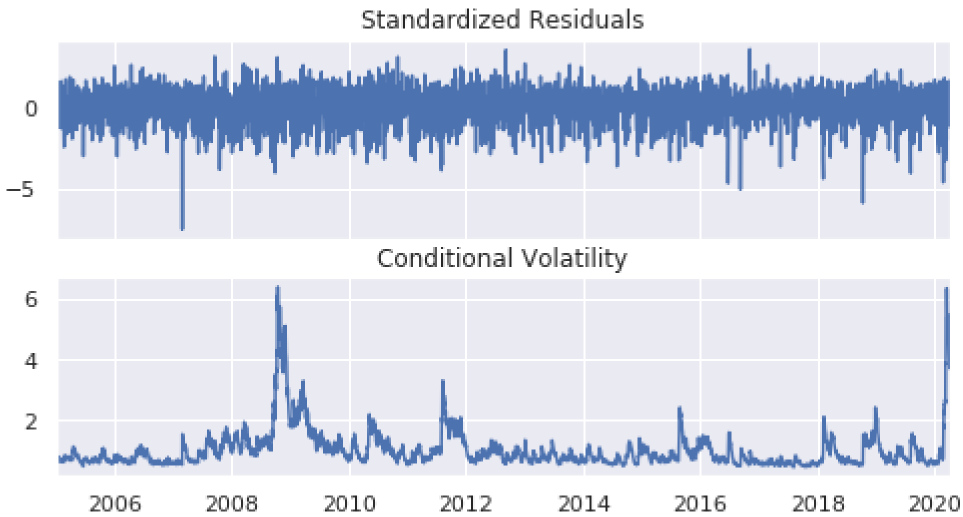
In [24]:

```
# Fit GARCH model with ARMA model residuals
_garch_model = arch_model(_model_result.resid, mean='Zero', p=1, q=1)
_garch_result = _garch_model.fit(disp = 'off')
print(_garch_result.summary())
```

```
                       Zero Mean - GARCH Model Results
==========================================================================
====
Dep. Variable:                      None   R-squared:
0.000
Mean Model:                    Zero Mean   Adj. R-squared:
0.000
Vol Model:                         GARCH   Log-Likelihood:             -502
0.26
Distribution:                     Normal   AIC:                         100
46.5
Method:               Maximum Likelihood   BIC:                         100
65.3
                                           No. Observations:
3843
Date:                   Mon, Apr 13 2020   Df Residuals:
3840
Time:                           20:56:02   Df Model:
3
                              Volatility Model
==========================================================================
==
                 coef    std err          t      P>|t|      95.0% Conf. In
t.
--------------------------------------------------------------------------
--
omega          0.0258  5.748e-03      4.481  7.443e-06 [1.449e-02,3.702e-0
2]
alpha[1]       0.1319  1.524e-02      8.650  5.169e-18    [  0.102,  0.16
2]
beta[1]        0.8467  1.526e-02     55.488      0.000    [  0.817,  0.87
7]
==========================================================================
==

Covariance estimator: robust
```

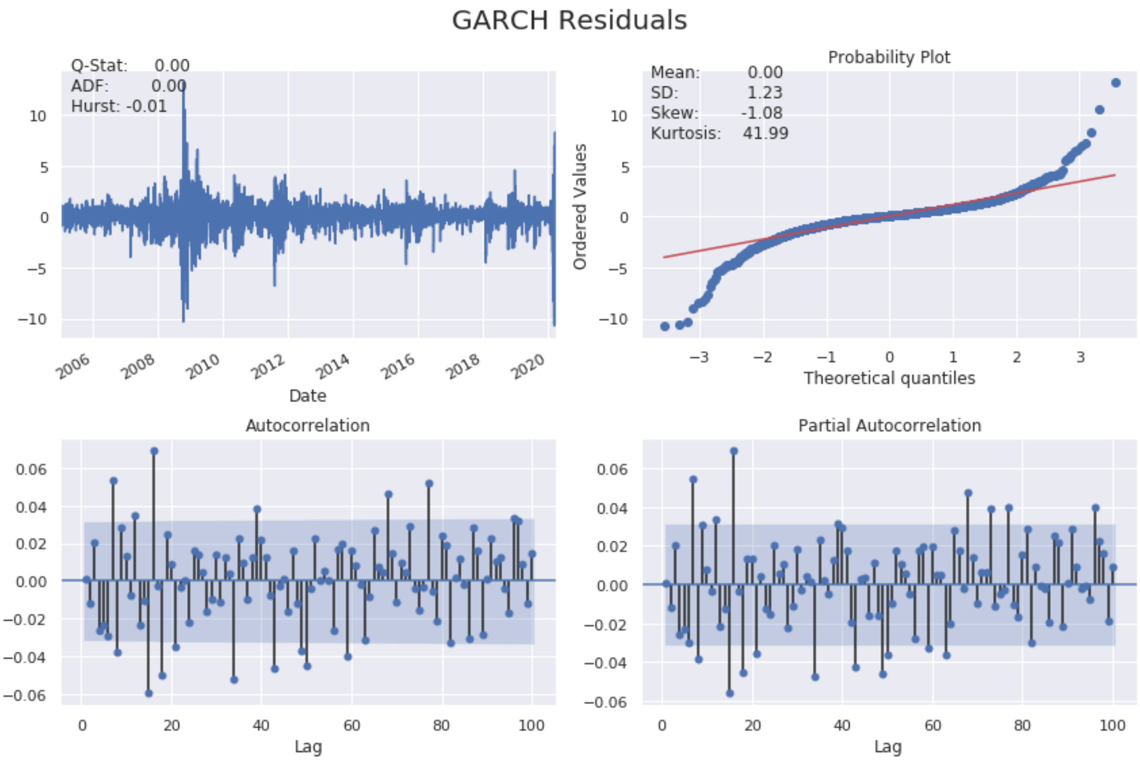
In [25]:

```
# Plot GARCH model fitted results
_garch_result.plot()
plt.show()
```



In [26]:

```
plot_correlogram(_garch_result.resid.dropna(), lags=100, title='GARCH Residuals')
```



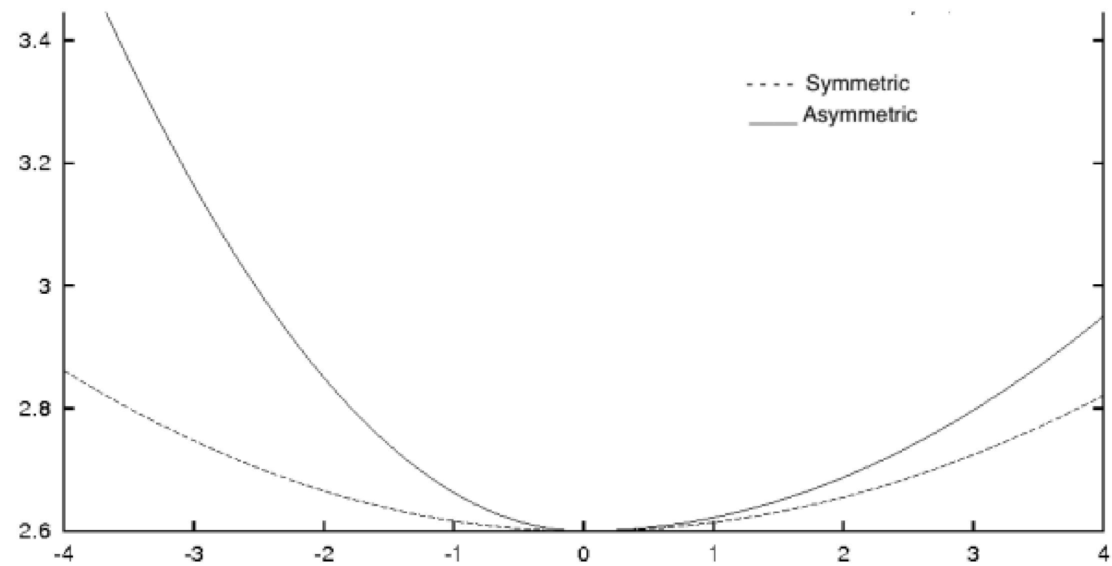
Modeling of asymmetric responses of volatility

Leverage effect

- Debt-equity Ratio = Debt / Equity
- Stock price goes down, debt-equity ratio goes up
- Riskier!

GARCH models assume positive and negative news has a symmetric impact on volatility. However, in reality the market tends to take the stairs up and the elevator down. In other words, the impact is usually asymmetric, and negative news tends to affect the volatility more than positive news.

News impact curve:



GARCH models that account for asymmetric shocks:

- `GJR-GARCH`
  - A popular option to model asymmetric shocks
  - GJR-GARCH in Python: `arch_model(my_data, p = 1, q = 1, o = 1, mean = 'constant', vol = 'GARCH')`

GJR-GARCH:

$$\sigma_t^2 = \omega + (\alpha + \gamma I_{t-1} + \beta\sigma_{t-1}^2$$

$$I_{t-1} := \begin{cases} 0 \text{ if } r_{t-1} \geq \mu \\ 1 \text{ if } r_{t-1} < \mu \end{cases}$$

- `EGARCH`
  - Exponential GARCH
  - Add a conditional component to model the asymmetry in shocks similar to the GJR-GARCH
  - No non-negative constraints on alpha, beta so it runs faster
  - GJR-GARCH in Python: `arch_model(my_data, p = 1, q = 1, o = 1, mean = 'constant', vol = EGARCH)`

EGARCH:

$$log\ \sigma_t^2 = \omega + \sum_{k=1}^{q} \beta_k g(Z_t - k) + \sum_{k=1}^{p} \alpha_k\ log\ \sigma_{t-k}^2$$

Fit GARCH models to cryptocurrency

Financial markets tend to react to positive and negative news shocks very differently, and one example is the dramatic swings observed in the `cryptocurrency` market in recent years.

We will implement a `GJR-GARCH` and an `EGARCH` model respectively in Python, which are popular choices to model the asymmetric responses of volatility.

Load the daily **`Bitcoin price`** from the Blockchain.com API:

In [27]:

```
bitcoin_data = pd.read_csv('https://api.blockchain.info/charts/market-price?start=2010-
10-09&timespan=12years&format=csv',
                          names=['Timestamp','Close'], index_col='Timestamp')
bitcoin_data.index = pd.to_datetime(bitcoin_data.index, format='%Y-%m-%d')
bitcoin_data = bitcoin_data.loc[(bitcoin_data != 0.0).any(axis=1)]
bitcoin_data['Return'] = np.log(bitcoin_data['Close']).diff().mul(100) # rescale to fac
iliate optimization
bitcoin_data = bitcoin_data.dropna()
bitcoin_data.info()
```

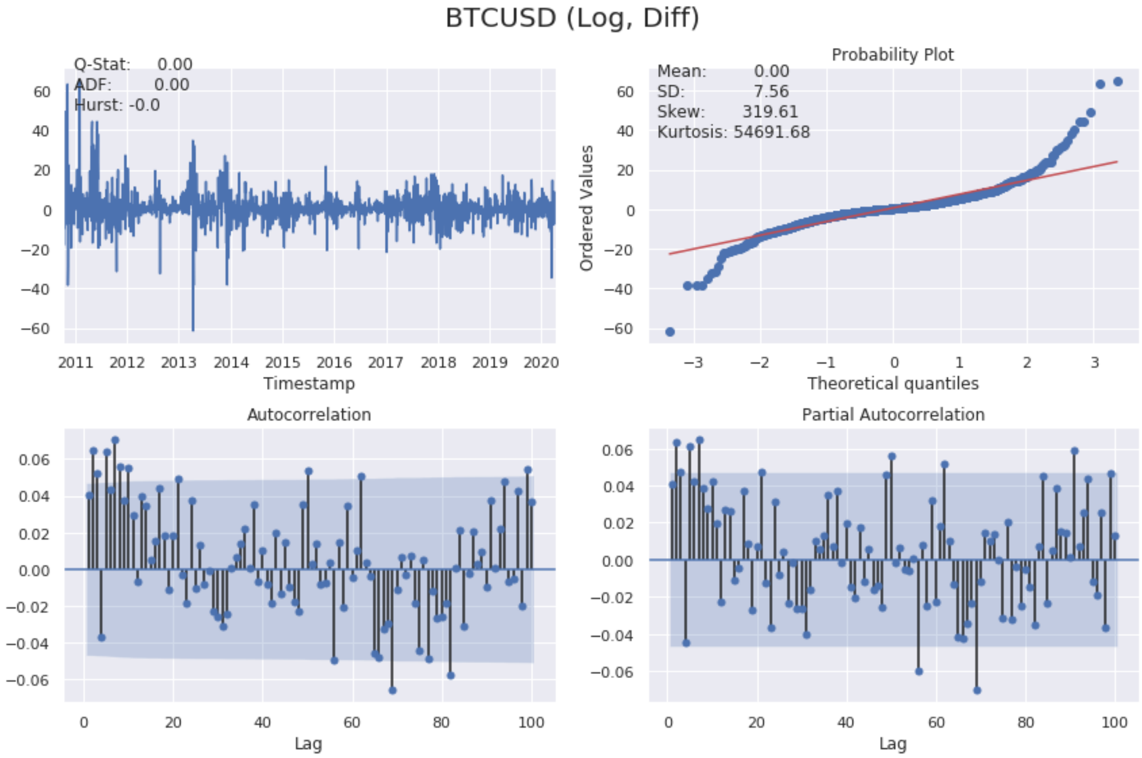
```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1737 entries, 2010-10-11 to 2020-04-13
Data columns (total 2 columns):
Close     1737 non-null float64
Return    1737 non-null float64
dtypes: float64(2)
memory usage: 40.7 KB
```

In [28]:

```
# Plot bitcoin price data
fig, ax1 = plt.subplots(figsize=(13, 5))
ax1.set_yscale('log')
ax1.plot(bitcoin_data.index, bitcoin_data.Close, color='b', label='BTCUSD')
ax1.set_xlabel('Date')
ax1.set_ylabel('BTCUSD Close')
ax1.legend()
plt.show()
```

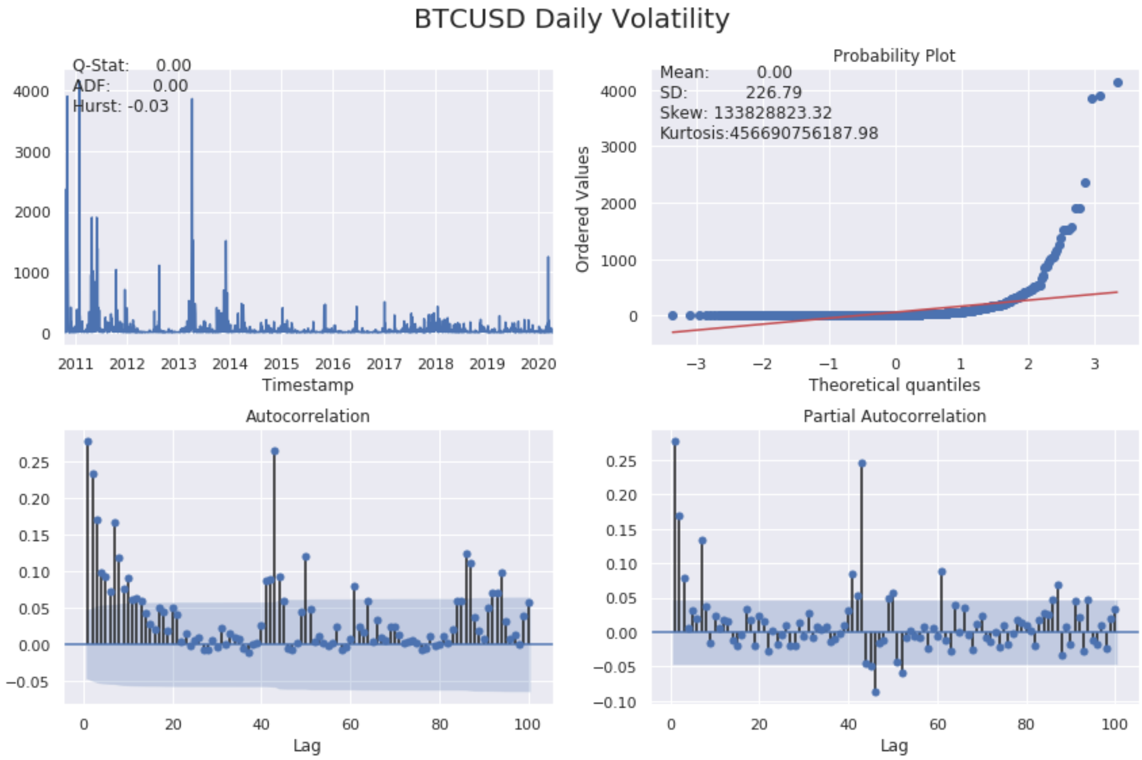
In [29]:

```
plot_correlogram(bitcoin_data['Return'], lags=100, title='BTCUSD (Log, Diff)')
```



In [30]:

```
plot_correlogram(bitcoin_data['Return'].sub(bitcoin_data['Return'].mean()).pow(2), lags
=100, title='BTCUSD Daily Volatility')
```



In [31]:

```
# Specify GJR-GARCH model assumptions
gjr_gm = arch_model(bitcoin_data['Return'], p = 1, q = 1, o = 1, vol = 'GARCH', dist =
't')

# Fit the model
gjrgm_result = gjr_gm.fit(disp = 'off')

# Print model fitting summary
print(gjrgm_result.summary())
```

```
                   Constant Mean - GJR-GARCH Model Results
==========================================================================
==========
Dep. Variable:                      Return   R-squared:
-0.002
Mean Model:                  Constant Mean   Adj. R-squared:
-0.002
Vol Model:                       GJR-GARCH   Log-Likelihood:
-5367.33
Distribution:      Standardized Student's t   AIC:
10746.7
Method:                 Maximum Likelihood   BIC:
10779.4
                                             No. Observations:
1737
Date:                     Mon, Apr 13 2020   Df Residuals:
1731
Time:                             20:56:12   Df Model:
6
                                Mean Model
========================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
------------------------------------------------------------------------
mu             0.3132  7.888e-02      3.970  7.188e-05 [  0.159,  0.468]
                              Volatility Model
==========================================================================
=
                 coef    std err          t      P>|t|     95.0% Conf. In
t.
--------------------------------------------------------------------------
-
omega          0.9396      0.527      1.784  7.447e-02 [-9.284e-02,  1.97
2]
alpha[1]       0.2593  3.937e-02      6.586  4.524e-11    [  0.182,  0.33
6]
gamma[1]      -0.1036  3.438e-02     -3.012  2.598e-03 [ -0.171,-3.616e-0
2]
beta[1]        0.7925  4.067e-02     19.487  1.416e-84    [  0.713,  0.87
2]
                              Distribution
========================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
------------------------------------------------------------------------
nu             3.6242      0.231     15.717  1.150e-55 [  3.172,  4.076]
========================================================================

Covariance estimator: robust
```

In [32]:

```
# Specify EGARCH model assumptions
egarch_gm = arch_model(bitcoin_data['Return'], p = 1, q = 1, o = 1, vol = 'EGARCH', dis
t = 't')

# Fit the model
egarch_result = egarch_gm.fit(disp = 'off')

# Print model fitting summary
print(egarch_result.summary())
```

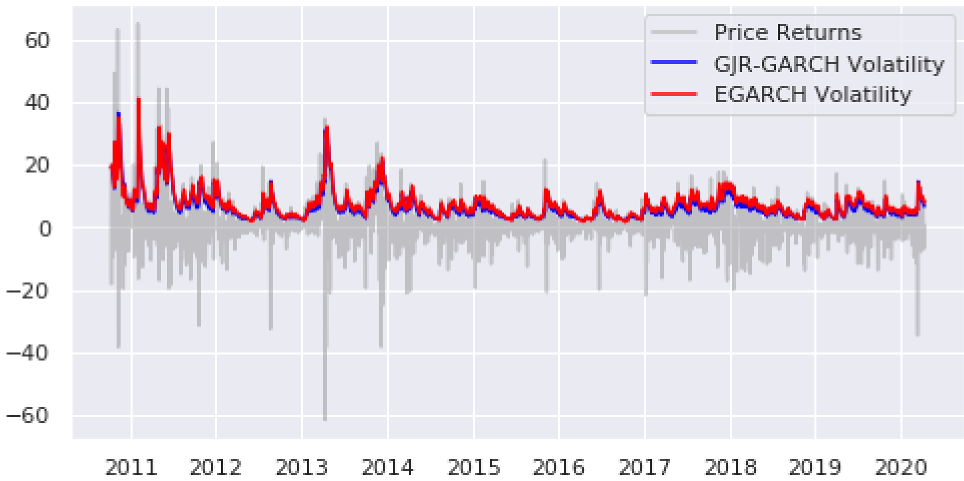
```
                        Constant Mean - EGARCH Model Results
==========================================================================
==========
Dep. Variable:                     Return   R-squared:
-0.002
Mean Model:                 Constant Mean   Adj. R-squared:
-0.002
Vol Model:                         EGARCH   Log-Likelihood:
-5354.83
Distribution:      Standardized Student's t   AIC:
10721.7
Method:                Maximum Likelihood   BIC:
10754.4
                                            No. Observations:
1737
Date:                    Mon, Apr 13 2020   Df Residuals:
1731
Time:                            20:56:13   Df Model:
6
                               Mean Model
========================================================================
                 coef    std err          t       P>|t|  95.0% Conf. Int.
------------------------------------------------------------------------
mu             0.3391  7.816e-02      4.338  1.435e-05 [  0.186,  0.492]
                            Volatility Model
==========================================================================
                 coef    std err          t       P>|t|    95.0% Conf. Int.
--------------------------------------------------------------------------
omega          0.1923  5.059e-02      3.800  1.447e-04 [9.309e-02,  0.291]
alpha[1]       0.4074  5.340e-02      7.629  2.359e-14   [  0.303,  0.512]
gamma[1]       0.0689  2.090e-02      3.298  9.732e-04 [2.796e-02,  0.110]
beta[1]        0.9656  1.087e-02     88.862      0.000   [  0.944,  0.987]
                              Distribution
========================================================================
                 coef    std err          t       P>|t|  95.0% Conf. Int.
------------------------------------------------------------------------
nu             3.0283      0.234     12.959  2.093e-38 [  2.570,  3.486]
========================================================================

Covariance estimator: robust
```

Compare GJR-GARCH with EGARCH

Previously we fitted a GJR-GARCH and EGARCH model with Bitcoin return time series. Now we will compare the estimated conditional volatility from the two models by plotting their results.

In [33]:

```
gjrgm_vol = gjrgm_result.conditional_volatility
egarch_vol = egarch_result.conditional_volatility

# Plot the actual Bitcoin returns
plt.plot(bitcoin_data['Return'], color = 'grey', alpha = 0.4, label = 'Price Returns')

# Plot GJR-GARCH estimated volatility
plt.plot(gjrgm_vol, color = 'blue', label = 'GJR-GARCH Volatility')

# Plot EGARCH  estimated volatility
plt.plot(egarch_vol, color = 'red', label = 'EGARCH Volatility')

plt.legend(loc = 'upper right')
plt.show()
```



In [34]:

```
# Print each models BIC
print(f'GJR-GARCH BIC: {gjrgm_result.bic}')
print(f'\nEGARCH BIC: {egarch_result.bic}')
```

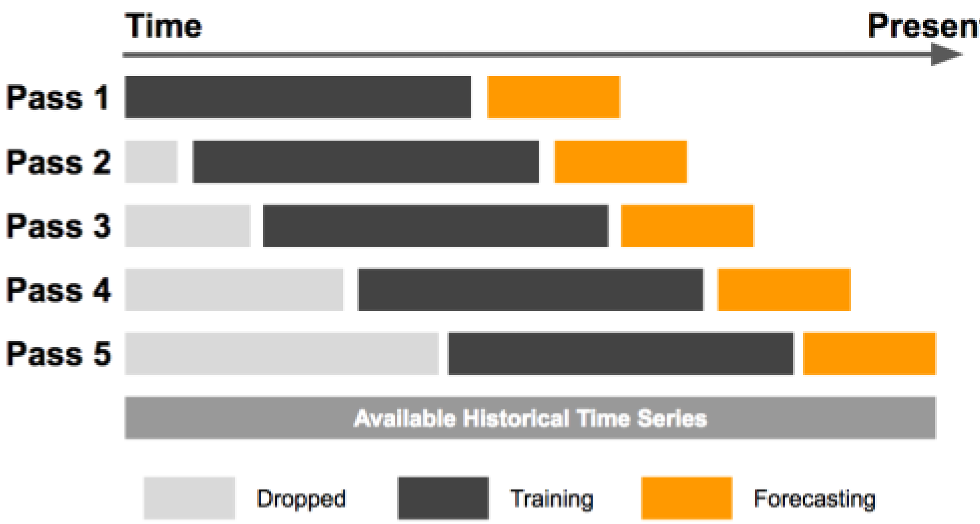
```
GJR-GARCH BIC: 10779.424172630153

EGARCH BIC: 10754.409909903035
```

Overall both GJR-GARCH and EGARCH models did a good job of fitting the actual data. Comparatively, GJR-GARCH is more conservative in volatility estimation when applying it to the Bitcoin dataset, but EGARCH yields the better model.

GARCH rolling window forecast

- **Expanding window forecast:** Continuously add new data points to the sample.

- **Fixed rolling window forecast:** New data points are added while old ones are dropped from the sample.



- Rolling window forecast
- Avoids lookback bias
- Less subject to overfitting
- Adapt forecast to new observations

How to determine window size ?

- Usually determined on a case-by-case basis.
- Too wide window size: include obsolete data that may lead to high bias.
- Too narrow window size: exclude relevant data that may lead to higher variance.
- The optimal window size: trade-off to balance bias and variance.

Fixed rolling window forecast

Rolling-window forecasts are very popular for financial time series modeling. We will practice how to implement GARCH model forecasts with a fixed rolling window.

First define the window size inside `.fit()` , and perform the forecast with a for-loop. Note since the window size remains fixed, both the start and end points increment after each iteration.

In [35]:

```
index = sp_data.index
start_loc = 0
end_loc = np.where(index >= '2020-1-1')[0].min()
forecasts = {}
for i in range(70):
    sys.stdout.write('-')
    sys.stdout.flush()
    res = _garch_model.fit(first_obs=start_loc + i, last_obs=i + end_loc, disp='off')
    temp = res.forecast(horizon=1).variance
    fcast = temp.iloc[i + end_loc - 1]
    forecasts[fcast.name] = fcast
print(' Done!')
variance_fixedwin = pd.DataFrame(forecasts).T
```

```
---------------------------------------------------------------------- Don
e!
```

Implement expanding window forecast

In [36]:

```
index = sp_data.index
start_loc = 0
end_loc = np.where(index >= '2020-1-1')[0].min()
forecasts = {}
for i in range(70):
    sys.stdout.write('-')
    sys.stdout.flush()
    res = _garch_model.fit(first_obs = start_loc, last_obs = i + end_loc, disp = 'off')
    temp = res.forecast(horizon=1).variance
    fcast = temp.iloc[i + end_loc - 1]
    forecasts[fcast.name] = fcast
print(' Done!')
variance_expandwin = pd.DataFrame(forecasts).T
```

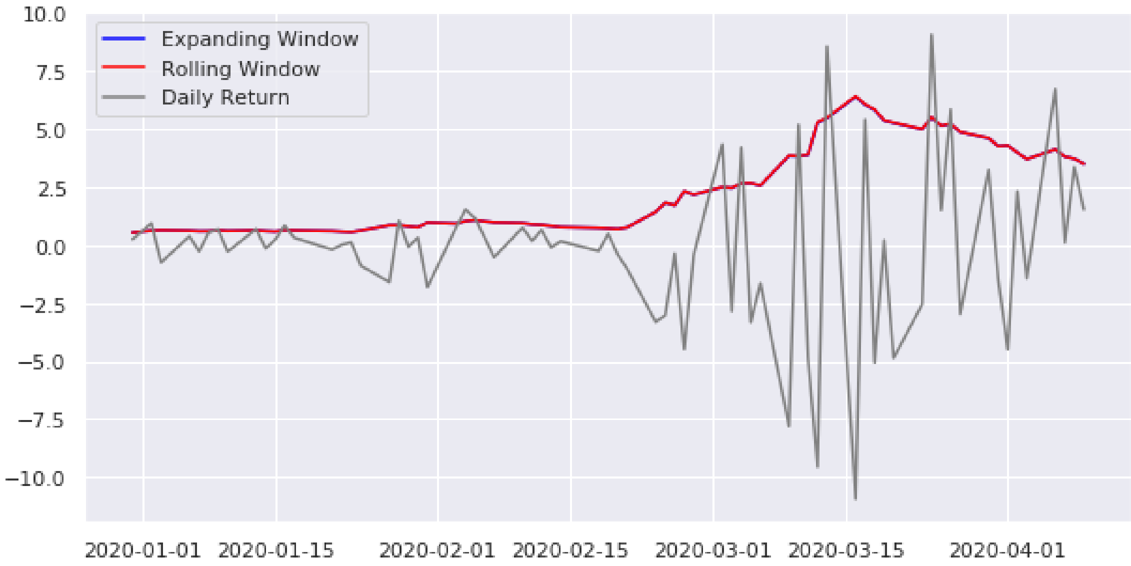
```
---------------------------------------------------------------------- Don
e!
```

Compare forecast results

Different rolling window approaches can generate different forecast results. Here we will take a closer look by comparing these forecast results.

In [37]:

```
# Calculate volatility from variance forecast with an expanding window
vol_expandwin = np.sqrt(variance_expandwin)

# Calculate volatility from variance forecast with a fixed rolling window
vol_fixedwin = np.sqrt(variance_fixedwin)

# Plot results
plt.figure(figsize=(10,5))

# Plot volatility forecast with an expanding window
plt.plot(vol_expandwin, color = 'blue', label='Expanding Window')

# Plot volatility forecast with a fixed rolling window
plt.plot(vol_fixedwin, color = 'red', label='Rolling Window')

plt.plot(sp_data.Return.loc[variance_expandwin.index], color = 'grey', label='Daily Ret
urn')

plt.legend()
plt.show()
```



Simplify the model with p-values

Leonardo da Vinci once said: "Simplicity is the ultimate sophistication." It also applies to data science modeling. We will practice using the p-values to decide the necessity of model parameters, and define a parsimonious model without insignificant parameters.

The null hypothesis is the parameter value is zero. If the p-value is larger than a given confidence level, the null hypothesis cannot be rejected, meaning the parameter is not statistically significant, hence not necessary.

In [38]:

```
# Get parameter stats from model summary
para_summary = pd.DataFrame({'parameter':gm_result.params,
                             'p-value': gm_result.pvalues})

# Print out parameter stats
print(para_summary)
```

```
          parameter       p-value
mu         0.072010  1.342505e-09
omega      0.026800  4.172162e-06
alpha[1]   0.141425  6.370351e-19
beta[1]    0.837174  0.000000e+00
```

Simplify the model with t-statistics

Besides p-values, t-statistics can also help decide the necessity of model parameters. We will practice using t-statistics to assess the significance of model parameters.

Whats a T-statistic? The `t-statistic` is computed as the estimated parameter value subtracted by its expected mean (zero in this case), and divided by its standard error. The absolute value of the t-statistic is a distance measure, that tells you how many standard errors the estimated parameter is away from `0`. As a rule of thumb, if the t-statistic is larger than `2`, you can reject the null hypothesis.

In [39]:

```
# Get parameter stats from model summary
para_summary = pd.DataFrame({'parameter':gm_result.params,
                             'std-err': gm_result.std_err,
                             't-value': gm_result.tvalues})

# Verify t-statistic by manual calculation
calculated_t = para_summary['parameter']/para_summary['std-err']

# Print parameter stats
print(para_summary)
```

```
          parameter   std-err    t-value
mu         0.072010  0.011878   6.062226
omega      0.026800  0.005823   4.602616
alpha[1]   0.141425  0.015917   8.885379
beta[1]    0.837174  0.015396  54.376076
```

Ljung-Box test

The Ljung-Box tests whether any of a group of autocorrelations of a time series are different from zero.

- H0: the data is independently distributed
- P-value < `5%` : the model is not sound

We will practice detecting autocorrelation in the standardized residuals by performing a Ljung-Box test.

The null hypothesis of Ljung-Box test is: the data is independently distributed. If the p-value is larger than the specified significance level, the null hypothesis cannot be rejected. In other words, there is no clear sign of autocorrelations and the model is valid.

In [40]:

```
# Import the Python module
from statsmodels.stats.diagnostic import acorr_ljungbox

# Perform the Ljung-Box test
lb_test = acorr_ljungbox(gm_std_resid , lags = 10)

# Store p-values in DataFrame
df = pd.DataFrame({'P-values': lb_test[1]}).T

# Create column names for each lag
col_num = df.shape[1]
col_names = ['lag_'+str(num) for num in list(range(1,col_num+1,1))]

# Display the p-values
df.columns = col_names
df
```

Out[40]:

	lag_1	lag_2	lag_3	lag_4	lag_5	lag_6	lag_7	lag_8	l
P-values	0.023542	0.062199	0.129365	0.225382	0.070697	0.044125	0.058047	0.064484	0.06

In [41]:

```
# Display the significant lags
mask = df < 0.05
df[mask].dropna(axis=1)
```

Out[41]:

	lag_1	lag_6
P-values	0.023542	0.044125

Goodness of fit

Can model do a good job explaining the data?

1. `Maximum likelihood`
2. `Information criteria (AIC, BIC)`

`Maximum likelihood:` Maximize the probability of getting the data observed under the assumed model. Preferred models have larger likelihood values.

`Maximum likelihood estimation:` In statistics, maximum likelihood estimation (MLE) is a method of estimating the parameters of a probability distribution by maximizing a likelihood function, so that under the assumed statistical model the observed data is most probable. The point in the parameter space that maximizes the likelihood function is called the maximum likelihood estimate.

`Likelihood function:` In statistics, the likelihood function (often simply called the likelihood) measures the goodness of fit of a statistical model to a sample of data for given values of the unknown parameters. It is formed from the joint probability distribution of the sample, but viewed and used as a function of the parameters only, thus treating the random variables as fixed at the observed values. The likelihood function describes a hypersurface whose peak, if it exists, represents the combination of model parameter values that maximize the probability of drawing the sample obtained.

Pick a winner based on log-likelihood

We will practice using log-likelihood to choose a model with the best fit.

In [42]:

```
# Print the log-likelihodd of normal GARCH
print('Log-likelihood of normal GARCH :', gm_result.loglikelihood)
# Print the log-likelihodd of skewt GARCH
print('Log-likelihood of skewt GARCH :', skewt_result.loglikelihood)
```

```
Log-likelihood of normal GARCH : -4992.874653095375
Log-likelihood of skewt GARCH : -4863.396191942436
```

Backtesting with MAE, MSE

We will practice how to evaluate model performance by conducting backtesting. The out-of-sample forecast accuracy is assessed by calculating MSE and MAE.

In [43]:

```
from sklearn.metrics import mean_absolute_error, mean_squared_error
```

In [44]:

```
def evaluate(observation, forecast):
    # Call sklearn function to calculate MAE
    mae = mean_absolute_error(observation, forecast)
    print(f'Mean Absolute Error (MAE): {round(mae,3)}')
    # Call sklearn function to calculate MSE
    mse = mean_squared_error(observation, forecast)
    print(f'Mean Squared Error (MSE): {round(mse,3)}')
    return mae, mse

# Backtest model with MAE, MSE
evaluate(bitcoin_data['Return'].sub(bitcoin_data['Return'].mean()).pow(2), egarch_vol**
2)
```

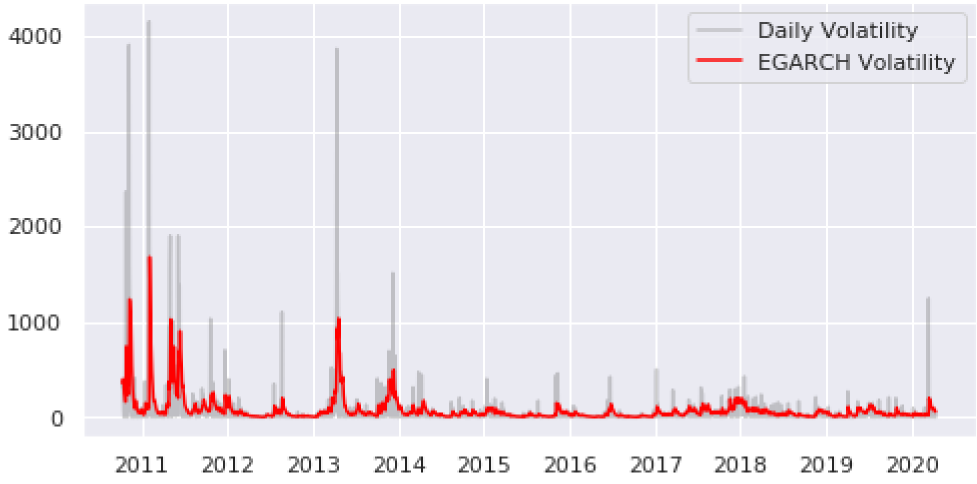
```
Mean Absolute Error (MAE): 76.556
Mean Squared Error (MSE): 49849.168
```

Out[44]:

```
(76.55622066807487, 49849.167908735326)
```

In [45]:

```
# Plot the actual Bitcoin volatility
plt.plot(bitcoin_data['Return'].sub(bitcoin_data['Return'].mean()).pow(2),
         color = 'grey', alpha = 0.4, label = 'Daily Volatility')

# Plot EGARCH  estimated volatility
plt.plot(egarch_vol**2, color = 'red', label = 'EGARCH Volatility')

plt.legend(loc = 'upper right')
plt.show()
```



Simulating Forecasts

When using simulation- or bootstrap-based forecasts, an additional attribute of an `ARCHModelForecast` object is meaningful – `simulation`.

Bootstrap Forecasts

Bootstrap-based forecasts are nearly identical to simulation-based forecasts except that the values used to simulate the process are computed from historical data rather than using the assumed distribution of the residuals. Forecasts produced using this method also return an ARCHModelForecastSimulation containing information about the simulated paths.

In [48]:

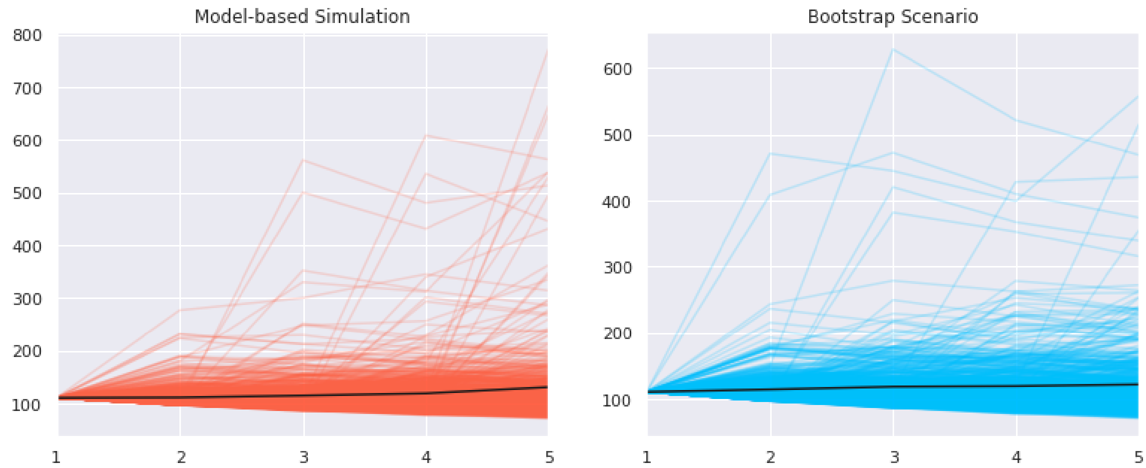
```
# The paths for the final observation
sim_forecasts = egarch_result.forecast(horizon=5, method='simulation')
sim_paths = sim_forecasts.simulations.residual_variances[-1].T
sim = sim_forecasts.simulations

bs_forecasts = egarch_result.forecast(horizon=5, method='bootstrap')
bs_paths = bs_forecasts.simulations.residual_variances[-1].T
bs = bs_forecasts.simulations
```

Comparing the paths

The paths are available on the attribute `simulations`. Plotting the paths shows important differences between the two scenarios beyond the average differences. Both start at the same point.

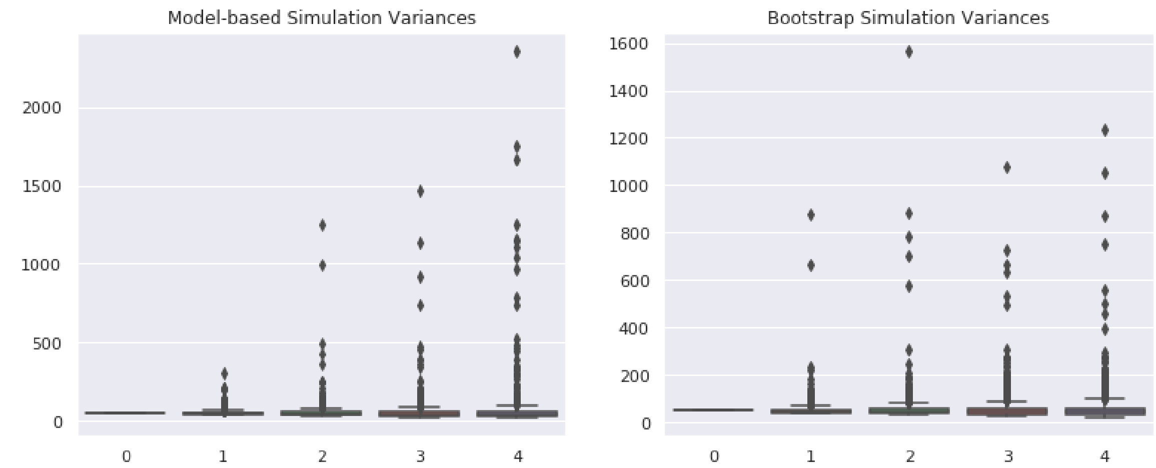
In [49]:

```
fig, axes = plt.subplots(1, 2, figsize=(13,5))

x = np.arange(1, 6)

# Plot the paths and the mean, set the axis to have the same limit
axes[0].plot(x, np.sqrt(252 * sim_paths), color='tomato', alpha=0.2)
axes[0].plot(x, np.sqrt(252 * sim_forecasts.residual_variance.iloc[-1]),
    color='k', alpha=1)

axes[0].set_title('Model-based Simulation')
axes[0].set_xticks(np.arange(1, 6))
axes[0].set_xlim(1, 5)

axes[1].plot(x, np.sqrt(252 * bs_paths), color='deepskyblue', alpha=0.2)
axes[1].plot(x,np.sqrt(252 * bs_forecasts.residual_variance.iloc[-1]),
    color='k', alpha=1)

axes[1].set_xticks(np.arange(1, 6))
axes[1].set_xlim(1, 5)

axes[1].set_title('Bootstrap Scenario')
plt.show()
```



In [50]:

```
# Plot Simulation Variances
fig, axes = plt.subplots(1, 2, figsize=(13,5))

sns.boxplot(data=sim.variances[-1], ax=axes[0])
sns.boxplot(data=bs.variances[-1], ax=axes[1])

axes[0].set_title('Model-based Simulation Variances')
axes[1].set_title('Bootstrap Simulation Variances')

plt.show()
```



VaR in financial risk management

What is VaR? VaR stands for Value at Risk

Three ingredients:

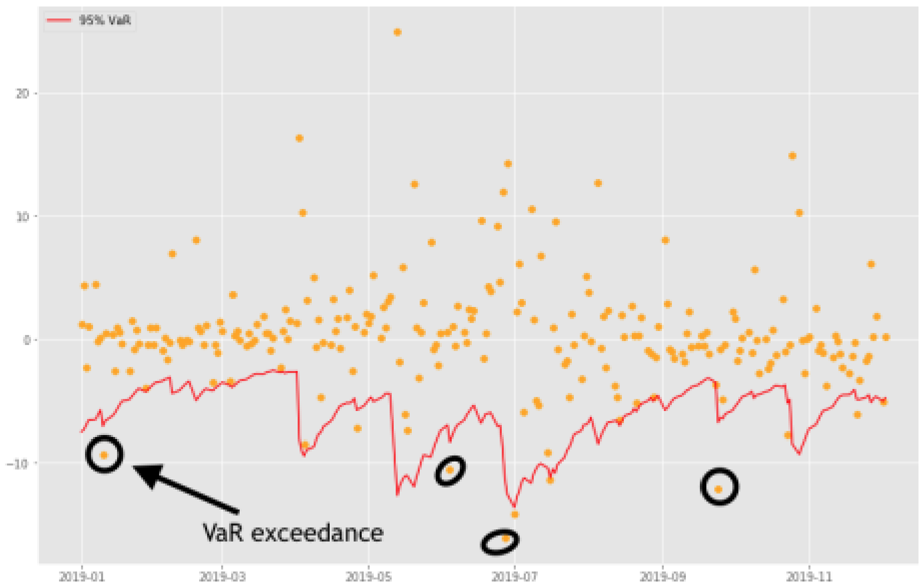
1. portfolio
2. time horizon
3. probability

VaR examples

- 1-day 5% VaR of $1 million
  - 5% probability the portfolio will fall in value by 1 million dollars or more over a 1-day period
- 10-day 1% VaR of $9 million
  - 1% probability the portfolio will fall in value by 9 million dollars or more over a 10-day period

VaR in risk management

- Set risk limits
- VaR exceedance: portfolio loss exceeds the VaR



Suppose a `5%` daily VaR and `252` trading days in a year. A valued VaR model should have less than `13` VaR exceedance in a year, that is `5% * 252` if there are more exceedances the model is under estimating the risk.

Dynamic VaR with GARCH

- More realistic VaR estimation with GARCH
- VaR = `mean + (GARCH vol) * quantile`

Parametric VaR

- Estimate quantiles based on GARCH assumed distribution of the standardized residuals.

Empirical VaR

- Estimate quantiles based on the observed distribution of the GARCH standardized residuals.

Compute parametric VaR

We will practice estimating dynamic `5%` and `1%` daily VaRs with a parametric approach.

Recall there are three steps to perform a forward VaR estimation. Step 1 is to use a GARCH model to make variance forecasts. Step 2 is to obtain the GARCH forward-looking mean and volatility. And Step 3 is to compute the quantile according to a given confidence level. The parametric approach estimates quantiles from an assumed distribution assumption.

In [51]:

```
am = arch_model(bitcoin_data['Return'], p = 1, q = 1, o = 1, vol = 'EGARCH', dist = 't'
)
res = am.fit(disp='off', last_obs='2018-01-01')
```

In [52]:

```
forecasts = res.forecast(start='2019-01-01')
cond_mean = forecasts.mean['2019':]
cond_var = forecasts.variance['2019':]
q = am.distribution.ppf([0.01, 0.05], res.params[5])
print(q)
```

```
[-2.63562572 -1.38237684]
```

In [53]:

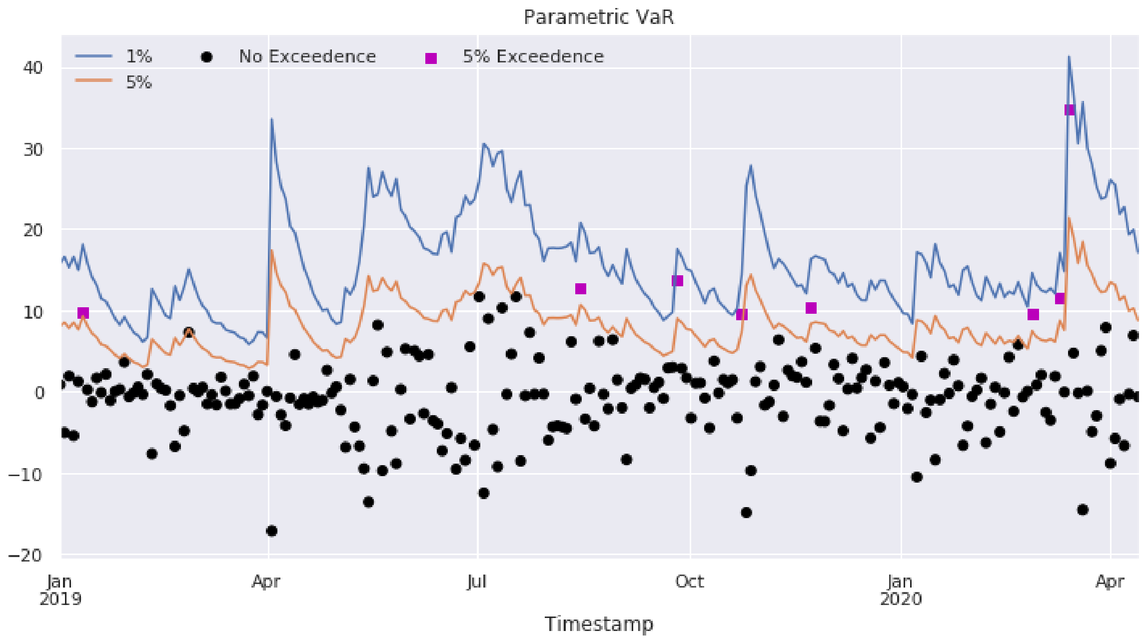
```
value_at_risk = -cond_mean.values - np.sqrt(cond_var).values * q[None, :]
value_at_risk = pd.DataFrame(value_at_risk, columns=['1%', '5%'], index=cond_var.index)
value_at_risk.describe()
```

Out[53]:

	1%	5%
count	235.000000	235.000000
mean	15.967809	8.181169
std	6.370272	3.341186
min	5.832790	2.865385
25%	11.615279	5.898281
50%	14.780231	7.558288
75%	19.352119	9.956228
max	41.222396	21.427115

In [54]:

```
ax = value_at_risk.plot(legend=False, figsize=(12,6))
xl = ax.set_xlim(value_at_risk.index[0], value_at_risk.index[-1])

rets_2019 = bitcoin_data.Return['2019':]
rets_2019.name = 'BTCUSD Return'

c = []
for idx in value_at_risk.index:
    if rets_2019[idx] > -value_at_risk.loc[idx, '5%']:
        c.append('#000000')
    elif rets_2019[idx] < -value_at_risk.loc[idx, '1%']:
        c.append('#BB0000')
    else:
        c.append('#BB00BB')

c = np.array(c, dtype='object')

labels = {

    '#BB0000': '1% Exceedence',
    '#BB00BB': '5% Exceedence',
    '#000000': 'No Exceedence'
}

markers = {'#BB0000': 'x', '#BB00BB': 's', '#000000': 'o'}

for color in np.unique(c):
    sel = c == color
    ax.scatter(
        rets_2019.index[sel],
        -rets_2019.loc[sel],
        marker=markers[color],
        c=c[sel],
        label=labels[color])

ax.set_title('Parametric VaR')
ax.legend(frameon=False, ncol=3)

plt.show()
```



Compute empirical VaR

We will practice estimating dynamic `5%` and `1%` daily VaRs with an empirical approach.

The difference between parametric VaR and empirical VaR is how the quantiles are estimated. The parametric approach estimates quantiles from an assumed distribution assumption, while the empirical approach estimates quantiles from an observed distribution of the standardized residuals.

In [55]:

```
# Obtain model estimated residuals and volatility
gm_resid = res.resid
gm_std = res.conditional_volatility

# Calculate the standardized residuals
gm_std_resid = gm_resid /gm_std

# Obtain the empirical quantiles
q = gm_std_resid.quantile([.01, .05])
print(q)
```

```
0.01   -2.565749
0.05   -1.306565
dtype: float64
```

In [56]:

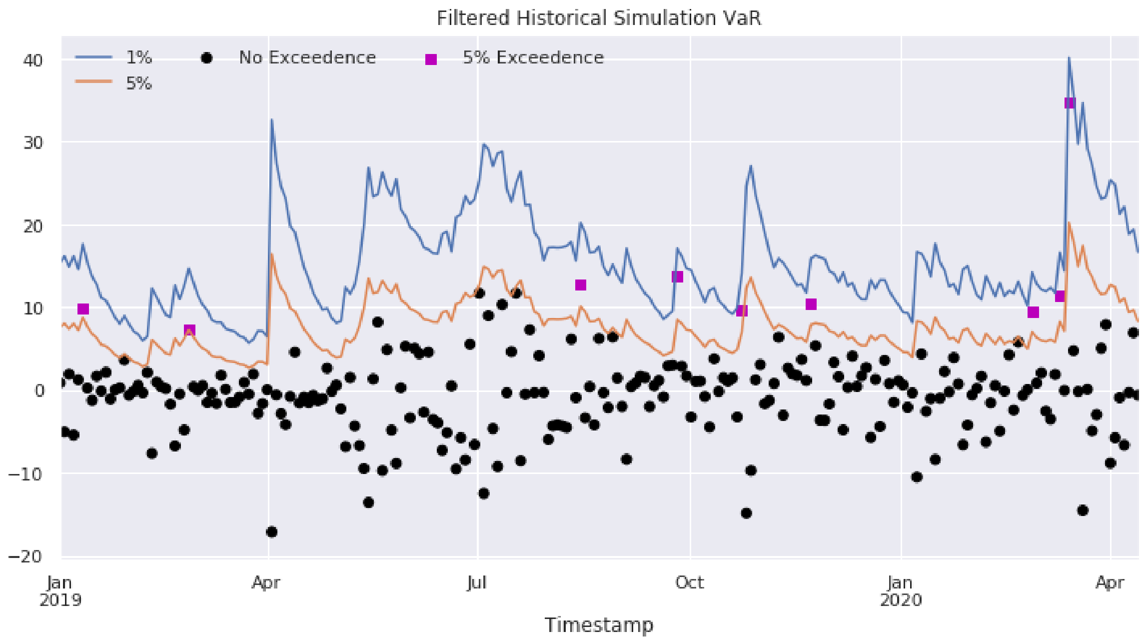
```
value_at_risk = -cond_mean.values - np.sqrt(cond_var).values * q.values[None, :]
value_at_risk = pd.DataFrame(value_at_risk, columns=['1%', '5%'], index=cond_var.index)
value_at_risk.describe()
```

Out[56]:

	1%	5%
count	235.000000	235.000000
mean	15.533653	7.710136
std	6.201381	3.157949
min	5.667338	2.685880
25%	11.296519	5.552446
50%	14.377560	7.121415
75%	18.828237	9.387847
max	40.118680	20.229649

In [57]:

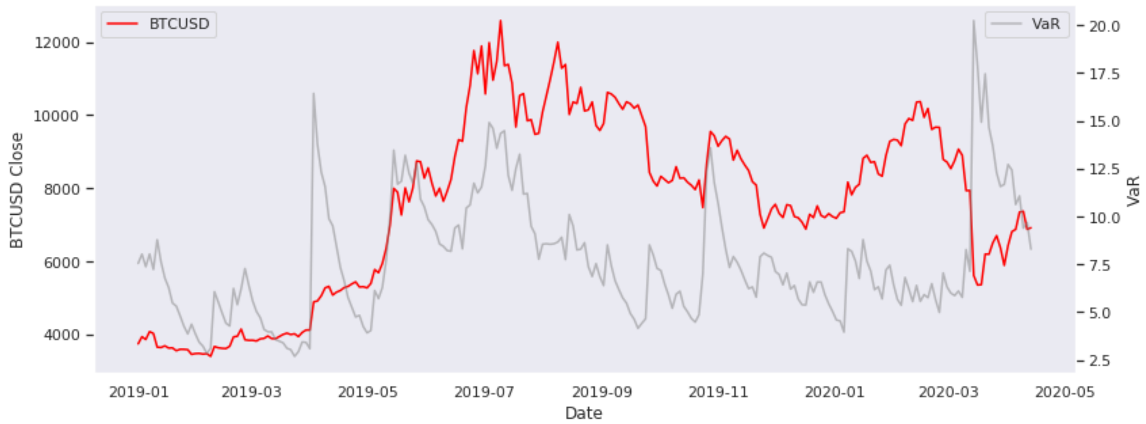
```
ax = value_at_risk.plot(legend=False, figsize=(12,6))
xl = ax.set_xlim(value_at_risk.index[0], value_at_risk.index[-1])

rets_2019 = bitcoin_data.Return['2019':]
rets_2019.name = 'BTCUSD Return'

c = []
for idx in value_at_risk.index:
    if rets_2019[idx] > -value_at_risk.loc[idx, '5%']:
        c.append('#000000')
    elif rets_2019[idx] < -value_at_risk.loc[idx, '1%']:
        c.append('#BB0000')
    else:
        c.append('#BB00BB')

c = np.array(c, dtype='object')
for color in np.unique(c):
    sel = c == color
    ax.scatter(
        rets_2019.index[sel],
        -rets_2019.loc[sel],
        marker=markers[color],
        c=c[sel],
        label=labels[color])

ax.set_title('Filtered Historical Simulation VaR')
ax.legend(frameon=False, ncol=3)

plt.show()
```



Plot the 5% daily VaR with the price

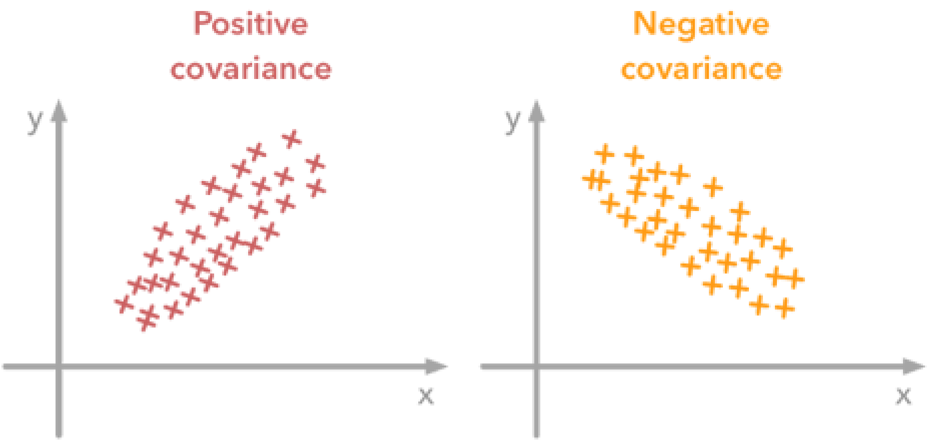
In [58]:

```
fig, ax1 = plt.subplots(figsize=(13, 5))
ax2 = ax1.twinx()

ax1.grid(False)
ax2.grid(False)

ax1.plot(bitcoin_data.loc[value_at_risk.index[0]:].index, bitcoin_data.loc[value_at_ris
k.index[0]:].Close,
         color='red', label='BTCUSD')
ax2.plot(value_at_risk['5%'].index, value_at_risk['5%'], color='grey', label='VaR', alp
ha=0.5)

ax1.set_xlabel('Date')
ax1.set_ylabel('BTCUSD Close')
ax2.set_ylabel('VaR')

ax1.legend(loc='upper left')
ax2.legend()

plt.show()
```



What is covariance?

- Describe the relationship between movement of two variables.
- Positive covariance: move together.
- Negative covariance; move in the opposite directions.



Dynamic covariance with GARCH

If two asset returns have correlation ρ and time-varying volatility of σ_1 and σ_2 :
$Covariance = p * \sigma_1 * \sigma_2$

Compute GARCH covariance

We will practice computing dynamic covariance with GARCH models.

Download Price Data

In [59]:

```
start = pd.Timestamp('2012-01-01')
end = pd.Timestamp('2020-01-06')

sp_data = web.DataReader('SPY', 'yahoo', start, end)\
      [['High','Low','Open','Close','Volume','Adj Close']]

sp_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2015 entries, 2012-01-03 to 2020-01-06
Data columns (total 6 columns):
High         2015 non-null float64
Low          2015 non-null float64
Open         2015 non-null float64
Close        2015 non-null float64
Volume       2015 non-null float64
Adj Close    2015 non-null float64
dtypes: float64(6)
memory usage: 110.2 KB
```

In [60]:

```
tmf_data = web.DataReader('TMF', 'yahoo', start, end)\
      [['High','Low','Open','Close','Volume','Adj Close']]

tmf_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2015 entries, 2012-01-03 to 2020-01-06
Data columns (total 6 columns):
High         2015 non-null float64
Low          2015 non-null float64
Open         2015 non-null float64
Close        2015 non-null float64
Volume       2015 non-null float64
Adj Close    2015 non-null float64
dtypes: float64(6)
memory usage: 110.2 KB
```

In [61]:

```
upro_data = web.DataReader('UPRO', 'yahoo', start, end)\
      [['High','Low','Open','Close','Volume','Adj Close']]

upro_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2015 entries, 2012-01-03 to 2020-01-06
Data columns (total 6 columns):
High         2015 non-null float64
Low          2015 non-null float64
Open         2015 non-null float64
Close        2015 non-null float64
Volume       2015 non-null float64
Adj Close    2015 non-null float64
dtypes: float64(6)
memory usage: 110.2 KB
```

Plot Price Data

In [62]:

```
prices = pd.DataFrame({'UPRO': upro_data['Close'],
                       'SPY': sp_data['Close'],
                       'TMF': tmf_data['Close']})

prices.div(prices.iloc[0,:]).plot(figsize=(12, 6))# Normalize Prices
```

Out[62]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f1bd7deada0>
```

Calculate Returns

In [63]:

```
tmf_data['Return'] = np.log(tmf_data['Close']).diff().mul(100) # rescale to faciliate o
ptimization
tmf_data = tmf_data.dropna()

upro_data['Return'] = np.log(upro_data['Close']).diff().mul(100) # rescale to faciliate
optimization
upro_data = upro_data.dropna()

sp_data['Return'] = np.log(sp_data['Close']).diff().mul(100) # rescale to faciliate opt
imization
sp_data = sp_data.dropna()
```

Find best Model

In [64]:

```
sp_model = pm.auto_arima(sp_data['Return'],

d=0, # non-seasonal difference order
start_p=1, # initial guess for p
start_q=1, # initial guess for q
max_p=4, # max value of p to test
max_q=4, # max value of q to test

seasonal=False, # is the time series seasonal

information_criterion='bic', # used to select best model
trace=True, # print results whilst training
error_action='ignore', # ignore orders that don't work
stepwise=True, # apply intelligent order search

)
```

```
Performing stepwise search to minimize bic
Fit ARIMA: (1, 0, 1)x(0, 0, 0, 0) (constant=True); AIC=4868.619, BIC=4891.
051, Time=1.001 seconds
Fit ARIMA: (0, 0, 0)x(0, 0, 0, 0) (constant=True); AIC=4876.688, BIC=4887.
904, Time=0.128 seconds
Fit ARIMA: (1, 0, 0)x(0, 0, 0, 0) (constant=True); AIC=4878.162, BIC=4894.
986, Time=0.089 seconds
Fit ARIMA: (0, 0, 1)x(0, 0, 0, 0) (constant=True); AIC=4878.123, BIC=4894.
947, Time=0.119 seconds
Fit ARIMA: (0, 0, 0)x(0, 0, 0, 0) (constant=False); AIC=4881.225, BIC=488
6.833, Time=0.047 seconds
Total fit time: 1.385 seconds
```

In [65]:

```
print(sp_model.summary())
```

```
                               SARIMAX Results
==========================================================================
====
Dep. Variable:                      y   No. Observations:
2014
Model:                        SARIMAX   Log Likelihood               -243
9.613
Date:                Mon, 13 Apr 2020   AIC                           488
1.225
Time:                        20:57:21   BIC                           488
6.833
Sample:                             0   HQIC                          488
3.283
                               - 2014
Covariance Type:                  opg
==========================================================================
====
                 coef    std err          z      P>|z|      [0.025      0.
975]
--------------------------------------------------------------------------
----
sigma2         0.6602      0.013     51.238      0.000       0.635
0.685
==========================================================================
=========
Ljung-Box (Q):                       52.15   Jarque-Bera (JB):
1010.05
Prob(Q):                              0.09   Prob(JB):
0.00
Heteroskedasticity (H):               1.36   Skew:
-0.47
Prob(H) (two-sided):                  0.00   Kurtosis:
6.34
==========================================================================
=========

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (com
plex-step).
```

In [66]:

```
tmf_model = pm.auto_arima(tmf_data['Return'],

d=0, # non-seasonal difference order
start_p=1, # initial guess for p
start_q=1, # initial guess for q
max_p=4, # max value of p to test
max_q=4, # max value of q to test

seasonal=False, # is the time series seasonal

information_criterion='bic', # used to select best model
trace=True, # print results whilst training
error_action='ignore', # ignore orders that don't work
stepwise=True, # apply intelligent order search

)
```

```
Performing stepwise search to minimize bic
Fit ARIMA: (1, 0, 1)x(0, 0, 0, 0) (constant=True); AIC=9083.397, BIC=9105.
829, Time=0.604 seconds
Fit ARIMA: (0, 0, 0)x(0, 0, 0, 0) (constant=True); AIC=9087.566, BIC=9098.
782, Time=0.034 seconds
Fit ARIMA: (1, 0, 0)x(0, 0, 0, 0) (constant=True); AIC=9087.706, BIC=9104.
529, Time=0.111 seconds
Fit ARIMA: (0, 0, 1)x(0, 0, 0, 0) (constant=True); AIC=9087.779, BIC=9104.
602, Time=0.107 seconds
Fit ARIMA: (0, 0, 0)x(0, 0, 0, 0) (constant=False); AIC=9085.776, BIC=909
1.384, Time=0.017 seconds
Total fit time: 0.875 seconds
```

In [67]:

```
print(tmf_model.summary())
```

```
                               SARIMAX Results                                
==========================================================================
====
Dep. Variable:                      y   No. Observations:                 
2014
Model:                        SARIMAX   Log Likelihood               -454
1.888
Date:                Mon, 13 Apr 2020   AIC                           908
5.776
Time:                        20:57:25   BIC                           909
1.384
Sample:                             0   HQIC                          908
7.834
                               - 2014                                         
Covariance Type:                  opg                                         
==========================================================================
====
                 coef    std err          z      P>|z|      [0.025      0.
975]
--------------------------------------------------------------------------
----
sigma2         5.3251      0.137     38.969      0.000       5.057       
5.593
==========================================================================
=========
Ljung-Box (Q):                       67.03   Jarque-Bera (JB):            
133.29
Prob(Q):                              0.00   Prob(JB):                    
0.00
Heteroskedasticity (H):               0.65   Skew:                        
-0.36
Prob(H) (two-sided):                  0.00   Kurtosis:                    
4.03
==========================================================================
=========

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (com
plex-step).
```

In [68]:

```
upro_model = pm.auto_arima(upro_data['Return'],

d=0, # non-seasonal difference order
start_p=1, # initial guess for p
start_q=1, # initial guess for q
max_p=4, # max value of p to test
max_q=4, # max value of q to test

seasonal=False, # is the time series seasonal

information_criterion='bic', # used to select best model
trace=True, # print results whilst training
error_action='ignore', # ignore orders that don't work
stepwise=True, # apply intelligent order search

)
```

```
Performing stepwise search to minimize bic
Fit ARIMA: (1, 0, 1)x(0, 0, 0, 0) (constant=True); AIC=9276.130, BIC=9298.
561, Time=0.802 seconds
Fit ARIMA: (0, 0, 0)x(0, 0, 0, 0) (constant=True); AIC=9281.192, BIC=9292.
408, Time=0.027 seconds
Fit ARIMA: (1, 0, 0)x(0, 0, 0, 0) (constant=True); AIC=9282.926, BIC=9299.
750, Time=0.101 seconds
Fit ARIMA: (0, 0, 1)x(0, 0, 0, 0) (constant=True); AIC=9282.909, BIC=9299.
733, Time=0.128 seconds
Fit ARIMA: (0, 0, 0)x(0, 0, 0, 0) (constant=False); AIC=9284.930, BIC=929
0.538, Time=0.018 seconds
Total fit time: 1.078 seconds
```

In [69]:

```
print(upro_model.summary())
```

```
                               SARIMAX Results                                
==========================================================================
====
Dep. Variable:                      y   No. Observations:                 
2014
Model:                        SARIMAX   Log Likelihood               -464
1.465
Date:                Mon, 13 Apr 2020   AIC                           928
4.930
Time:                        20:57:28   BIC                           929
0.538
Sample:                             0   HQIC                          928
6.988
                               - 2014                                         
Covariance Type:                  opg                                         
==========================================================================
====
                 coef    std err          z      P>|z|      [0.025      0.
975]
--------------------------------------------------------------------------
----
sigma2         5.8786      0.115     51.315      0.000       5.654       
6.103
==========================================================================
=========
Ljung-Box (Q):                       55.94   Jarque-Bera (JB):            
1069.17
Prob(Q):                              0.05   Prob(JB):                    
0.00
Heteroskedasticity (H):               1.35   Skew:                        
-0.58
Prob(H) (two-sided):                  0.00   Kurtosis:                    
6.37
==========================================================================
=========

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (com
plex-step).
```

Fit best Model

In [70]:

```
_arma_sp = sm.tsa.SARIMAX(endog=sp_data['Return'],order=(0, 0, 0))
_sp_model_result = _arma_sp.fit()

egarch_sp = arch_model(_sp_model_result.resid, p = 1, q = 1, o = 1, vol = 'EGARCH', dis
t = 't', mean = 'zero')
sp_gm_result = egarch_sp.fit(disp = 'off')
print(sp_gm_result.summary())
```

```
                        Zero Mean - EGARCH Model Results
==========================================================================
==========
Dep. Variable:                      None   R-squared:
0.000
Mean Model:                    Zero Mean   Adj. R-squared:
0.000
Vol Model:                        EGARCH   Log-Likelihood:
-2103.72
Distribution:      Standardized Student's t   AIC:
4217.44
Method:                  Maximum Likelihood   BIC:
4245.48
                                            No. Observations:
2014
Date:                   Mon, Apr 13 2020   Df Residuals:
2009
Time:                           20:57:30   Df Model:
5
                               Volatility Model
==========================================================================
====
                 coef    std err          t      P>|t|        95.0% Conf.
Int.
--------------------------------------------------------------------------
----
omega         -0.0316  8.850e-03     -3.566  3.625e-04 [-4.891e-02,-1.421e
-02]
alpha[1]       0.1628  2.676e-02      6.084  1.170e-09       [  0.110,  0.
215]
gamma[1]      -0.2603  2.503e-02    -10.398  2.519e-25       [ -0.309, -0.
211]
beta[1]        0.9257  1.093e-02     84.729      0.000       [  0.904,  0.
947]
                              Distribution
========================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
------------------------------------------------------------------------
nu             6.9236      1.063      6.515  7.293e-11 [  4.841,  9.007]
========================================================================

Covariance estimator: robust
```

In [71]:

```
_arma_tmf = sm.tsa.SARIMAX(endog=tmf_data['Return'],order=(0, 0, 0))
_tmf_model_result = _arma_tmf.fit()

egarch_tmf = arch_model(_tmf_model_result.resid, p = 1, q = 1, o = 1, vol = 'EGARCH', d
ist = 't', mean = 'zero')
tmf_gm_result = egarch_tmf.fit(disp = 'off')
print(tmf_gm_result.summary())
```

```
                       Zero Mean - EGARCH Model Results
==========================================================================
==========
Dep. Variable:                      None   R-squared:
0.000
Mean Model:                    Zero Mean   Adj. R-squared:
0.000
Vol Model:                        EGARCH   Log-Likelihood:
-4477.57
Distribution:      Standardized Student's t   AIC:
8965.14
Method:                 Maximum Likelihood   BIC:
8993.18
                                           No. Observations:
2014
Date:                   Mon, Apr 13 2020   Df Residuals:
2009
Time:                           20:57:31   Df Model:
5
                              Volatility Model
==========================================================================
===
                 coef    std err          t      P>|t|      95.0% Conf. I
nt.
--------------------------------------------------------------------------
---
omega      5.5220e-03  4.664e-03      1.184      0.236 [-3.619e-03,1.466e-
02]
alpha[1]       0.0492  1.476e-02      3.335  8.534e-04  [2.030e-02,7.816e-
02]
gamma[1]       0.0181  6.884e-03      2.625  8.671e-03  [4.576e-03,3.156e-
02]
beta[1]        0.9967  2.766e-03    360.341      0.000      [  0.991,   1.0
02]
                              Distribution
========================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
------------------------------------------------------------------------
nu            15.2415      5.301      2.875  4.038e-03 [  4.852, 25.631]
========================================================================

Covariance estimator: robust
```

In [72]:

```
_arma_upro = sm.tsa.SARIMAX(endog=upro_data['Return'],order=(0, 0, 0))
_upro_model_result = _arma_upro.fit()

egarch_upro = arch_model(_upro_model_result.resid, p = 1, q = 1, o = 1, vol = 'EGARCH',
dist = 't', mean = 'zero')
upro_gm_result = egarch_upro.fit(disp = 'off')
print(upro_gm_result.summary())
```

```
                       Zero Mean - EGARCH Model Results
==========================================================================
==========
Dep. Variable:                        None   R-squared:
0.000
Mean Model:                      Zero Mean   Adj. R-squared:
0.000
Vol Model:                          EGARCH   Log-Likelihood:
-4285.39
Distribution:      Standardized Student's t   AIC:
8580.78
Method:                 Maximum Likelihood   BIC:
8608.82
                                             No. Observations:
2014
Date:                     Mon, Apr 13 2020   Df Residuals:
2009
Time:                             20:57:33   Df Model:
5
                             Volatility Model
==========================================================================
                 coef    std err          t      P>|t|    95.0% Conf. Int.
--------------------------------------------------------------------------
omega          0.1352  1.834e-02      7.372  1.680e-13 [9.926e-02,  0.171]
alpha[1]       0.1640  2.538e-02      6.463  1.029e-10  [  0.114,  0.214]
gamma[1]      -0.2714  2.590e-02    -10.477  1.097e-25  [ -0.322, -0.221]
beta[1]        0.9226  1.088e-02     84.812      0.000  [  0.901,  0.944]
                              Distribution
========================================================================
                 coef    std err          t      P>|t|  95.0% Conf. Int.
------------------------------------------------------------------------
nu             6.4228      0.947      6.782  1.181e-11 [  4.567,  8.279]
========================================================================

Covariance estimator: robust
```

In [73]:

```
# Step 1: Fit GARCH models and obtain volatility for each return series
vol_tmf = tmf_gm_result.conditional_volatility
vol_upro = upro_gm_result.conditional_volatility
```

In [74]:

```
# Step 2: Compute standardized residuals from the tted GARCH models
resid_tmf = tmf_gm_result.resid/vol_tmf
resid_upro = upro_gm_result.resid/vol_upro
```

In [75]:

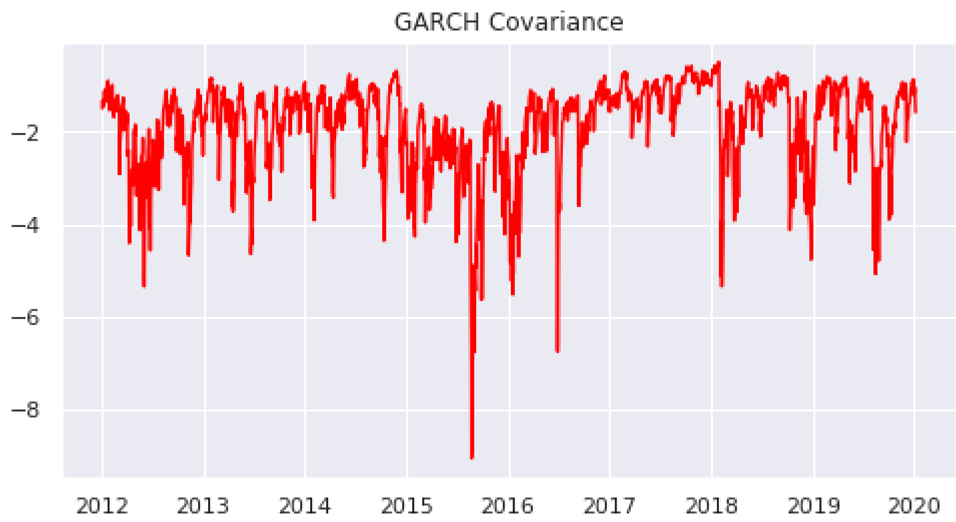
```
# Step 3: Compute ρ as simple correlation of standardized residuals
corr = np.corrcoef(resid_tmf, resid_upro)[0,1]
```

In [76]:

```
# Step 4: Compute GARCH covariance by multiplying the correlation and volatility.
covariance = corr * vol_tmf * vol_upro
```

In [77]:

```
# Plot the data
plt.plot(covariance, color = 'red')
plt.title('GARCH Covariance')
plt.show()
```



Compute dynamic portfolio variance

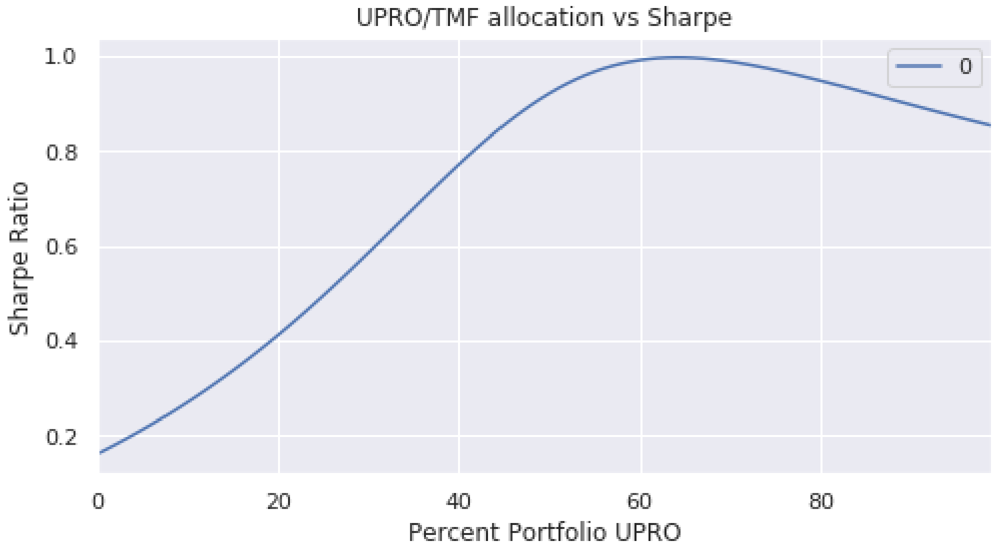
We will practice computing the variance of a simple two-asset portfolio with GARCH dynamic covariance.

The Modern Portfolio Theory states that there is an optimal way to construct a portfolio to take advantage of the diversification effect, so one can obtain a desired level of expected return with the minimum risk. This effect is especially evident when the covariance between asset returns is negative.

Find best max sharpe ratio weights

In [78]:

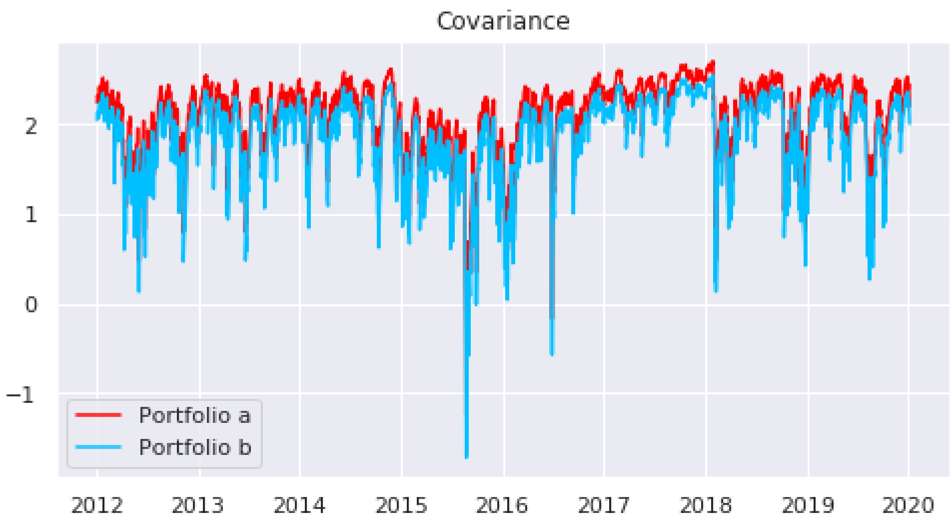
```
data = {}
for perc in range(100):
    daily = (1 - perc / 100) * tmf_data["Return"] + (perc / 100) * upro_data["Return"]
    data[perc] = daily.mean() / daily.std() * (252 ** 0.5)

sx = pd.Series(data)
s = pd.DataFrame(sx, index=sx.index)
ax = s.plot(title="UPRO/TMF allocation vs Sharpe")
ax.set_ylabel("Sharpe Ratio")
ax.set_xlabel("Percent Portfolio UPRO")
plt.show()

tmf_w = 100 - s.idxmax()[0]
print('Optimal UPRO Weight =', s.idxmax()[0])
print('Optimal TNF Weight =', tmf_w)
print('Optimal Sharpe Ratio =', s.max()[0])
```




```
Optimal UPRO Weight = 64
Optimal TNF Weight = 36
Optimal Sharpe Ratio = 0.9957468623983499
```

In [79]:

```
# Define weights
Wa1 = 0.64
Wa2 = 0.36

Wb1 = 0.49
Wb2 = 1 - Wb1

# Calculate individual returns variance
variance_upro = np.var(upro_data['Return'])
variance_tmf = np.var(tmf_data['Return'])

# Calculate portfolio variance
portvar_a = Wa1**2 * variance_tmf + Wa2**2 * variance_upro + 2*Wa1*Wa2*covariance
portvar_b = Wb1**2 * variance_tmf + Wb2**2 * variance_upro + 2*Wb1*Wb2*covariance

# Plot the data
plt.plot(portvar_a, color = 'red', label = 'Portfolio a')
plt.plot(portvar_b, color = 'deepskyblue', label = 'Portfolio b')
plt.title('Covariance')
plt.legend(loc = 'best')
plt.show()
```



What is Beta?

Stock Beta: A measure of stock volatility in relation to the general market.

Systematic risk: The portion of the risk that cannot be diversified away.

Beta in portfolio management

- Gauge investment risk
- `Market Beta = 1:` used as benchmark
- `Beta > 1:` the stock bears more risks than the general market
- `Beta < 1:` the stock bears less risks than the general market

Dynamic Beta with GARCH

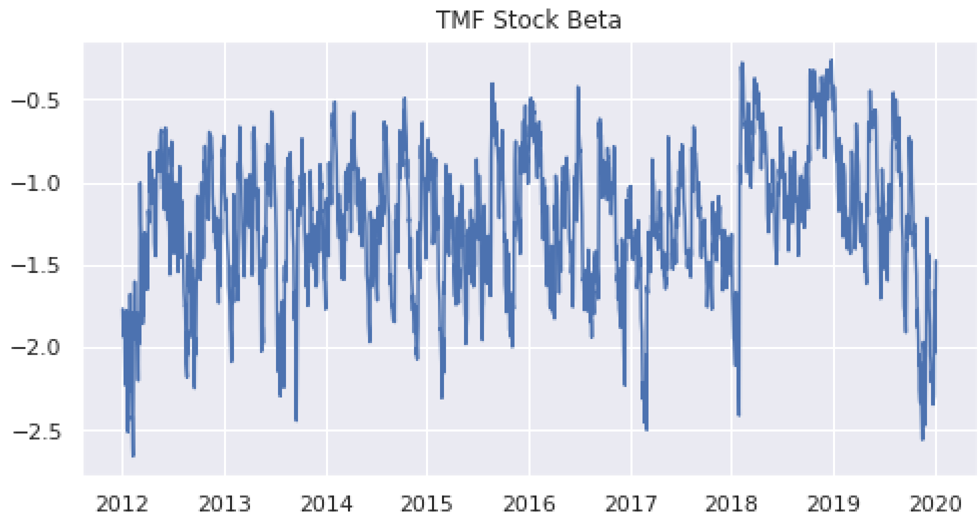
$$Beta = p * \frac{\sigma_{stock}}{\sigma_{market}}$$

In [80]:

```
# 1). Compute correlation between S&P500 and stock
resid_stock = tmf_gm_result.resid / tmf_gm_result.conditional_volatility
resid_sp500 = sp_gm_result.resid / sp_gm_result.conditional_volatility

correlation = np.corrcoef(resid_stock, resid_sp500)[0, 1]

# 2). Compute dynamic Beta for the stock
stock_beta = correlation * (tmf_gm_result.conditional_volatility / sp_gm_result.conditi
onal_volatility)
```

In [81]:

```
# Plot the Beta
plt.title('TMF Stock Beta')
plt.plot(stock_beta)
plt.show()
```



In [ ]:

```

```