Active trading strategy based on price & volume data

Getting the data

In [1]:
```
# Importing the required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
plt.style.use("seaborn")
```

In [2]:
```
# Creating a dataframe and setting real time as index
data = pd.read_csv("bitcoin.csv", parse_dates = ["Date"], index_col = "Date")
data
```

Out[2]:

Date	Open	High	Low	Close	Volume
2017-08-17 04:00:00	4261.48	4313.62	4261.32	4308.83	47.181009
2017-08-17 05:00:00	4308.83	4328.69	4291.37	4315.32	23.234916
2017-08-17 06:00:00	4330.29	4345.45	4309.37	4324.35	7.229691
2017-08-17 07:00:00	4316.62	4349.99	4287.41	4349.99	4.443249
2017-08-17 08:00:00	4333.32	4377.85	4333.32	4360.69	0.972807
...	...	...	...	...	...
2021-10-07 05:00:00	55073.20	55073.21	54545.07	54735.76	2251.122020
2021-10-07 06:00:00	54735.77	54968.06	54375.83	54534.16	1783.004260
2021-10-07 07:00:00	54534.16	54793.26	54235.33	54755.92	4163.431360
2021-10-07 08:00:00	54755.91	54778.91	54400.00	54538.30	2049.382180
2021-10-07 09:00:00	54538.31	54547.30	53786.13	53995.50	2739.153610

36168 rows × 5 columns

In [3]:
```
# Getting insights into the data
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 36168 entries, 2017-08-17 04:00:00 to 2021-10-07 09:00:00
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Open    36168 non-null  float64
 1   High    36168 non-null  float64
 2   Low     36168 non-null  float64
 3   Close   36168 non-null  float64
 4   Volume  36168 non-null  float64
dtypes: float64(5)
memory usage: 1.7 MB
```

Data Quality Report

- Our strategy is based on price and volume change. Hence we only need those 2 columns for our machine learning model. Therefore the first step would be to remove all the other unnecessary columns.
- After performing the first step we would be left with just 2 independent features. We cannot directly use those two columns because we want our model to work on changes in those values. Hence as derived features we will be getting two new columns which would have instances having values calculated based on the changes in the independent features every hour.
- More insight: Absolute price and volume changes make no sense! So the two new derived features would be the percentage changes in those two independent features.
- There will be an issue of outliers as some values in the derived features would take values either equal to -infinity or +infinity. We need to tackle those outliers and we will replace those values with missing values and going ahead further would drop the rows contatining those missing values. This is because the k-means clustering which we will be using in this algorithm doesn't allow missing values.

In [4]:
```
# Keeping the required columns
data = data[["Close", "Volume"]].copy()
```

In [5]:
```
data
```

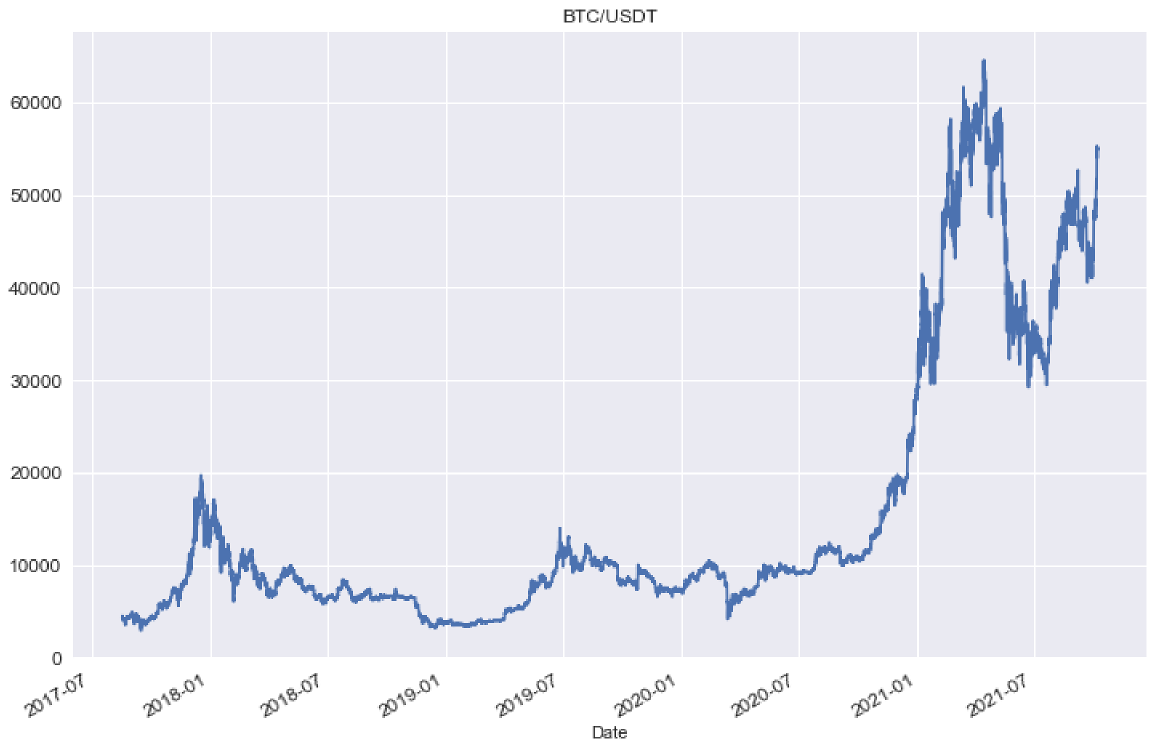
Out[5]:

Date	Close	Volume
2017-08-17 04:00:00	4308.83	47.181009
2017-08-17 05:00:00	4315.32	23.234916
2017-08-17 06:00:00	4324.35	7.229691
2017-08-17 07:00:00	4349.99	4.443249
2017-08-17 08:00:00	4360.69	0.972807
...	...	...
2021-10-07 05:00:00	54735.76	2251.122020
2021-10-07 06:00:00	54534.16	1783.004260
2021-10-07 07:00:00	54755.92	4163.431360
2021-10-07 08:00:00	54538.30	2049.382180
2021-10-07 09:00:00	53995.50	2739.153610

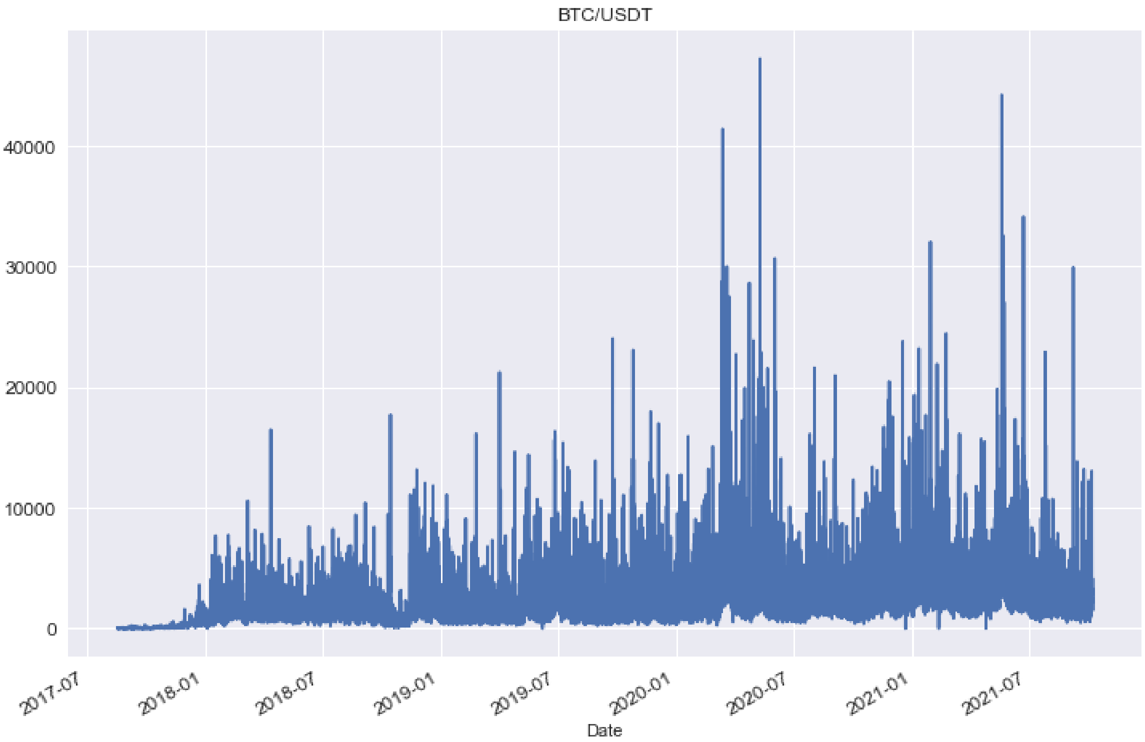
36168 rows × 2 columns

Data Analysis / Visual Inspection

In [6]:
```
# Graph of closing price over the whole timeframe
data.Close.plot(figsize = (12, 8), title = "BTC/USDT", fontsize = 12)
plt.show()
```



In [7]:
```
# Graph of volume over the whole timeframe
data.Volume.plot(figsize = (12, 8), title = "BTC/USDT", fontsize = 12)
plt.show()
```



In [8]:
```
# Calculating percentage change in price and adding the derived feature to the dataframe
data["returns"] = np.log(data.Close.div(data.Close.shift(1)))
data
```

Out[8]:

Date	Close	Volume	returns
2017-08-17 04:00:00	4308.83	47.181009	NaN
2017-08-17 05:00:00	4315.32	23.234916	0.001505
2017-08-17 06:00:00	4324.35	7.229691	0.002090
2017-08-17 07:00:00	4349.99	4.443249	0.005912
2017-08-17 08:00:00	4360.69	0.972807	0.002457
...	...	...	...
2021-10-07 05:00:00	54735.76	2251.122020	-0.006146
2021-10-07 06:00:00	54534.16	1783.004260	-0.003690
2021-10-07 07:00:00	54755.92	4163.431360	0.004058
2021-10-07 08:00:00	54538.30	2049.382180	-0.003982
2021-10-07 09:00:00	53995.50	2739.153610	-0.010002

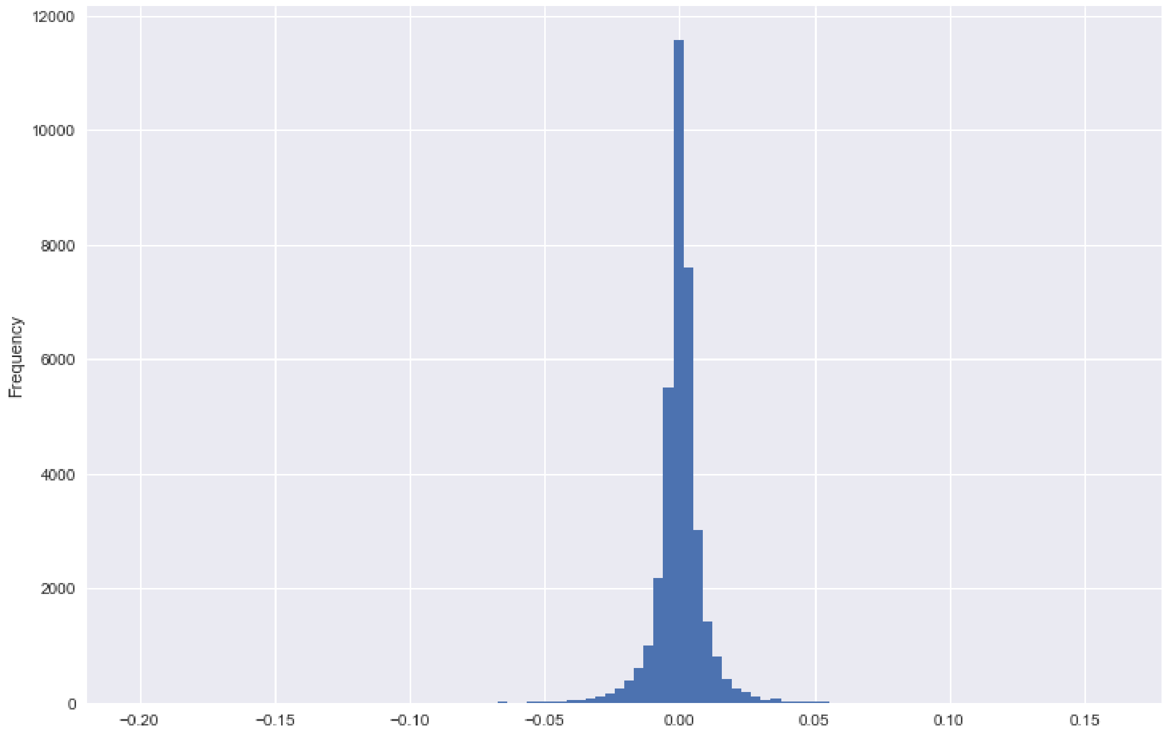
36168 rows × 3 columns

In [9]:
```
# Continuous features report
data.describe()
```

Out[9]:

	Close	Volume	returns
count	36168.000000	36168.000000	36167.000000
mean	15211.287479	2121.344201	0.000070
std	14918.059912	2211.660869	0.009669
min	2919.000000	0.000000	-0.201033
25%	6619.987500	910.157520	-0.002955
50%	9110.620000	1551.676864	0.000139
75%	13411.242500	2603.584828	0.003258
max	64577.260000	47255.762685	0.160280

In [10]:
```
# Visualising the distribution of column 'returns'
data.returns.plot(kind = "hist", bins = 100, figsize = (12,8))
plt.show()
```



In [11]:
```
# Getting information about maximum positive returns. (+16% in 1 hour shows high volatility -> High Re
ward)
data.returns.nlargest(10)
```

Out[11]:
```
Date
2020-03-13 02:00:00    0.160280
2017-09-15 12:00:00    0.131731
2020-03-15 21:00:00    0.129546
2017-09-15 14:00:00    0.117777
2021-01-29 08:00:00    0.116145
2017-09-05 02:00:00    0.113257
2018-01-17 16:00:00    0.108790
2018-04-12 11:00:00    0.103325
2018-10-15 06:00:00    0.100727
2019-07-18 14:00:00    0.089576
Name: returns, dtype: float64
```

In [12]:
```
# Getting information about maximum negative returns. (-20% in 1 hour shows high volatility -> High Ri
sk)
data.returns.nsmallest(10)
```

Out[12]:
```
Date
2020-03-12 10:00:00   -0.201033
2020-03-12 23:00:00   -0.189707
2020-03-13 01:00:00   -0.119449
2017-12-28 02:00:00   -0.108097
2017-12-22 13:00:00   -0.107858
2017-09-05 01:00:00   -0.099818
2017-08-22 04:00:00   -0.098295
2020-03-15 22:00:00   -0.095180
2021-05-19 12:00:00   -0.093810
2019-09-24 18:00:00   -0.093730
Name: returns, dtype: float64
```

In [ ]:

Baseline Model: A simple Buy and Hold "Strategy"

Assumption: Invest 1 USD(T) in BTC on 2017-08-17 and hold until 2021-10-07 (no further trades).

In [13]:
```
data
```

Out[13]:

	Close	Volume	returns
Date			
2017-08-17 04:00:00	4308.83	47.181009	NaN
2017-08-17 05:00:00	4315.32	23.234916	0.001505
2017-08-17 06:00:00	4324.35	7.229691	0.002090
2017-08-17 07:00:00	4349.99	4.443249	0.005912
2017-08-17 08:00:00	4360.69	0.972807	0.002457
...	...	...	...
2021-10-07 05:00:00	54735.76	2251.122020	-0.006146
2021-10-07 06:00:00	54534.16	1783.004260	-0.003690
2021-10-07 07:00:00	54755.92	4163.431360	0.004058
2021-10-07 08:00:00	54538.30	2049.382180	-0.003982
2021-10-07 09:00:00	53995.50	2739.153610	-0.010002

36168 rows × 3 columns

In [14]:
```
# Calculating investment multiple for every hour (Basically the value shows how much worth is your ass
et at any given time [absolute values normalized to 1])
data.Close / data.Close[0]
```

Out[14]:
```
Date
2017-08-17 04:00:00     1.000000
2017-08-17 05:00:00     1.001506
2017-08-17 06:00:00     1.003602
2017-08-17 07:00:00     1.009552
2017-08-17 08:00:00     1.012036
                          ...
2021-10-07 05:00:00    12.703161
2021-10-07 06:00:00    12.656373
2021-10-07 07:00:00    12.707839
2021-10-07 08:00:00    12.657334
2021-10-07 09:00:00    12.531360
Name: Close, Length: 36168, dtype: float64
```

In [15]:
```
# Calculating investment multiple mathematically
data.returns.sum()
multiple = np.exp(data.returns.sum())
multiple
```

Out[15]:
```
12.531360021165671
```

In [16]:
```
# Normalized Prices with Base Value 1 (Adding the feature for better understanding)
data["creturns"] = data.returns.cumsum().apply(np.exp)
```

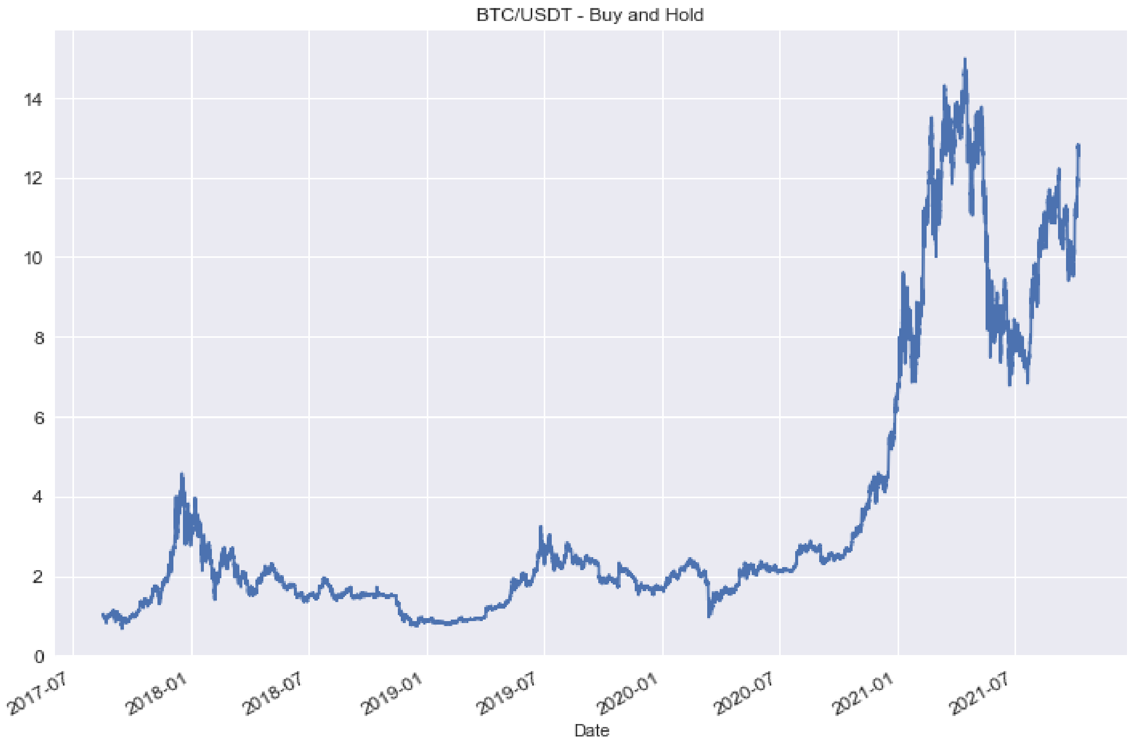
In [17]:
```
data
```

Out[17]:

	Close	Volume	returns	creturns
Date				
2017-08-17 04:00:00	4308.83	47.181009	NaN	NaN
2017-08-17 05:00:00	4315.32	23.234916	0.001505	1.001506
2017-08-17 06:00:00	4324.35	7.229691	0.002090	1.003602
2017-08-17 07:00:00	4349.99	4.443249	0.005912	1.009552
2017-08-17 08:00:00	4360.69	0.972807	0.002457	1.012036
...	...	...	...	...
2021-10-07 05:00:00	54735.76	2251.122020	-0.006146	12.703161
2021-10-07 06:00:00	54534.16	1783.004260	-0.003690	12.656373
2021-10-07 07:00:00	54755.92	4163.431360	0.004058	12.707839
2021-10-07 08:00:00	54538.30	2049.382180	-0.003982	12.657334
2021-10-07 09:00:00	53995.50	2739.153610	-0.010002	12.531360

36168 rows × 4 columns

In [18]:
```
# Graphical representation of the column 'creturns'
data.creturns.plot(figsize = (12, 8), title = "BTC/USDT - Buy and Hold", fontsize = 12)
plt.show()
```



Performance Measurement / Evaluation

Mean Return & Risk

In [19]:
```
# Mean return over the whole timeframe
mu = data.returns.mean()
mu
```

Out[19]: 6.990445168833971e-05

In [20]:
```
# Standard Deviation is an excellent parameter to measure the risk of the asset.
std = data.returns.std()
std
```

Out[20]: 0.009669001511177732

- We have a standard deviation of nearly 1%. It shows risk associated with the BTC. So standard deviation of 1% every hour tells us the asset is highly volatile and that investing money in BTC is very risky.

Annualized Mean Return and Risk

In [21]:
```
number_of_periods = 24 * 365.25
number_of_periods
```

Out[21]: 8766.0

In [22]:
```
# Annual Reward associated with BTC
ann_mean = mu * number_of_periods
ann_mean
```

Out[22]: 0.6127824234999859

In [23]:
```
# Annualised Risk associated with BTC
ann_std = std * np.sqrt(number_of_periods)
ann_std
```

Out[23]: 0.9052788232893756

- We have a standard deviation of nearly 90%. It shows risk associated with the BTC. So standard deviation of 90% every year tells us the asset is highly volatile and that investing money in BTC is very risky.

CAGR (Compound Annual Growth Rate)

In [24]:
```
cagr = np.exp(ann_mean) - 1
cagr
```
Out[24]:
```
0.8455593891678417
```

Risk-adjusted Return ("Sharpe Ratio")

In [25]:
```
# Sharpe Ratio based on annualised mean
ann_mean / ann_std
```
Out[25]:
```
0.6768991030557973
```

- Technical indicator of the performance of any stock or crypto currency. Higher value means better performance

In [26]:
```
# Sharpe ratio based on compound annualised mean
cagr / ann_std
```
Out[26]:
```
0.9340319992192677
```

In [ ]:
```

```

Preparing the Data for the Trading Strategy

In [27]:
```
data
```
Out[27]:

	Close	Volume	returns	creturns
Date				
2017-08-17 04:00:00	4308.83	47.181009	NaN	NaN
2017-08-17 05:00:00	4315.32	23.234916	0.001505	1.001506
2017-08-17 06:00:00	4324.35	7.229691	0.002090	1.003602
2017-08-17 07:00:00	4349.99	4.443249	0.005912	1.009552
2017-08-17 08:00:00	4360.69	0.972807	0.002457	1.012036
...	...	...	...	...
2021-10-07 05:00:00	54735.76	2251.122020	-0.006146	12.703161
2021-10-07 06:00:00	54534.16	1783.004260	-0.003690	12.656373
2021-10-07 07:00:00	54755.92	4163.431360	0.004058	12.707839
2021-10-07 08:00:00	54538.30	2049.382180	-0.003982	12.657334
2021-10-07 09:00:00	53995.50	2739.153610	-0.010002	12.531360

36168 rows × 4 columns

Adding the Feature "Change in Trading Volume (log)"

In [28]:
```
# Calculating percentage change in volume and adding the derived feature to the dataframe
data["vol_ch"] = np.log(data.Volume.div(data.Volume.shift(1)))
data
```

```
/Users/aamir/opt/anaconda3/lib/python3.8/site-packages/pandas/core/arraylike.py:358: RuntimeWarning:
divide by zero encountered in log
  result = getattr(ufunc, method)(*inputs, **kwargs)
```

Out[28]:

	Close	Volume	returns	creturns	vol_ch
Date					
2017-08-17 04:00:00	4308.83	47.181009	NaN	NaN	NaN
2017-08-17 05:00:00	4315.32	23.234916	0.001505	1.001506	-0.708335
2017-08-17 06:00:00	4324.35	7.229691	0.002090	1.003602	-1.167460
2017-08-17 07:00:00	4349.99	4.443249	0.005912	1.009552	-0.486810
2017-08-17 08:00:00	4360.69	0.972807	0.002457	1.012036	-1.518955
...	...	...	...	...	...
2021-10-07 05:00:00	54735.76	2251.122020	-0.006146	12.703161	0.439863
2021-10-07 06:00:00	54534.16	1783.004260	-0.003690	12.656373	-0.233129
2021-10-07 07:00:00	54755.92	4163.431360	0.004058	12.707839	0.848040
2021-10-07 08:00:00	54538.30	2049.382180	-0.003982	12.657334	-0.708801
2021-10-07 09:00:00	53995.50	2739.153610	-0.010002	12.531360	0.290111

36168 rows × 5 columns

Data Cleaning (Removing Outliers)

- As discussed in the data quality report

In [29]:
```
data.vol_ch.nsmallest(20)
```

Out[29]:
```
Date
2019-06-07 21:00:00        -inf
2020-12-21 14:00:00        -inf
2021-02-11 03:00:00        -inf
2021-04-25 04:00:00   -5.644090
2018-01-04 03:00:00   -5.428025
2019-06-07 20:00:00   -4.780619
2017-08-19 23:00:00   -3.801014
2017-08-20 09:00:00   -3.782857
2017-08-26 04:00:00   -3.470297
2017-12-04 06:00:00   -3.178488
2017-09-24 21:00:00   -2.749294
2017-08-21 09:00:00   -2.643555
2017-08-19 19:00:00   -2.357538
2020-06-09 01:00:00   -2.157645
2017-08-20 03:00:00   -2.106886
2018-10-06 03:00:00   -2.072737
2017-09-12 20:00:00   -1.948525
2019-10-13 21:00:00   -1.905406
2017-10-07 04:00:00   -1.901772
2020-03-04 09:00:00   -1.841755
Name: vol_ch, dtype: float64
```

In [30]:
```
data.vol_ch.nlargest(20)
```

Out[30]:
```
Date
2019-06-07 22:00:00         inf
2020-12-21 18:00:00         inf
2021-02-11 05:00:00         inf
2018-01-04 05:00:00    5.256246
2021-04-25 08:00:00    5.051831
2017-08-20 00:00:00    3.794985
2017-08-26 05:00:00    3.428566
2017-08-20 11:00:00    2.904046
2017-10-12 00:00:00    2.884007
2017-12-04 07:00:00    2.851238
2019-07-27 10:00:00    2.808519
2017-08-20 15:00:00    2.779948
2019-12-16 18:00:00    2.658757
2017-08-20 05:00:00    2.585916
2019-09-06 17:00:00    2.579215
2018-04-12 11:00:00    2.562466
2020-06-10 18:00:00    2.537417
2018-10-15 05:00:00    2.525149
2019-01-19 10:00:00    2.512394
2019-07-28 22:00:00    2.490934
Name: vol_ch, dtype: float64
```

- We get such infinity values because there is a chance that there was absolutely no volume change in a given hour.

In [31]:
```
# Tackling outliers
data.loc[data.vol_ch > 3, "vol_ch"] = np.nan
data.loc[data.vol_ch < -3, "vol_ch"] = np.nan
```

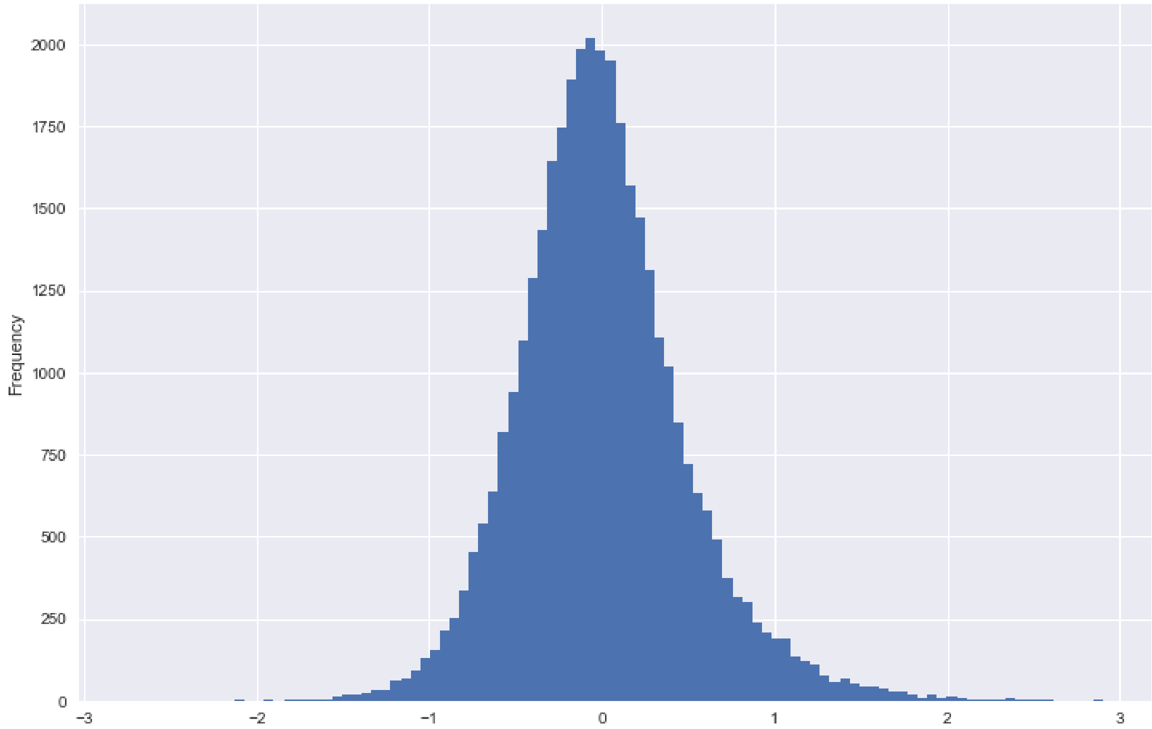
In [32]:
```
data.dropna(inplace= True)
```

In [33]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 36150 entries, 2017-08-17 05:00:00 to 2021-10-07 09:00:00
Data columns (total 5 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   Close     36150 non-null  float64
 1   Volume    36150 non-null  float64
 2   returns   36150 non-null  float64
 3   creturns  36150 non-null  float64
 4   vol_ch    36150 non-null  float64
dtypes: float64(5)
memory usage: 1.7 MB
```

- We can see we have some missing values in the last column

In [34]:
```
# Visualising the distribution of column 'vol_ch'
data.vol_ch.plot(kind = "hist", bins = 100, figsize = (12,8))
plt.show()
```



Applying k-means clustering to develop a trading strategy (Unsupervised Learning)

Question 1: Is there a relationship between price changes and volume changes?
(e.g. rapid Increase in Trading Volume triggers extreme Price changes)

Question 2: Can we use return/vol_ch clusters to (partly) forecast future returns?
Can our machine learning model pick that relation and use it

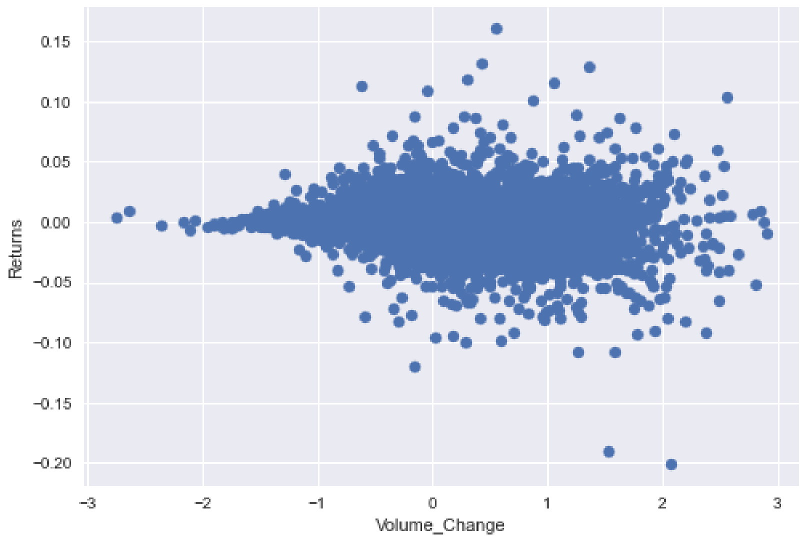
In [35]:
```
data
```

Out[35]:

	Close	Volume	returns	creturns	vol_ch
Date					
2017-08-17 05:00:00	4315.32	23.234916	0.001505	1.001506	-0.708335
2017-08-17 06:00:00	4324.35	7.229691	0.002090	1.003602	-1.167460
2017-08-17 07:00:00	4349.99	4.443249	0.005912	1.009552	-0.486810
2017-08-17 08:00:00	4360.69	0.972807	0.002457	1.012036	-1.518955
2017-08-17 09:00:00	4444.00	10.763623	0.018925	1.031370	2.403742
...	...	...	...	...	...
2021-10-07 05:00:00	54735.76	2251.122020	-0.006146	12.703161	0.439863
2021-10-07 06:00:00	54534.16	1783.004260	-0.003690	12.656373	-0.233129
2021-10-07 07:00:00	54755.92	4163.431360	0.004058	12.707839	0.848040
2021-10-07 08:00:00	54538.30	2049.382180	-0.003982	12.657334	-0.708801
2021-10-07 09:00:00	53995.50	2739.153610	-0.010002	12.531360	0.290111

36150 rows × 5 columns

In [36]:
```python
# Two dimensional graph showing the percentage price and volume changes values
plt.scatter(x = data.vol_ch, y = data.returns)
plt.xlabel("Volume_Change")
plt.ylabel("Returns")
plt.show()
```



In [37]:
```python
# Getting the data without timeindex to perform clustering
data_clustering = pd.read_csv('bitcoin.csv')
data_clustering = data_clustering[['Close', 'Volume']]
data_clustering['vol_ch'] = np.log(data_clustering.Volume.div(data_clustering.Volume.shift(1)))
data_clustering['returns'] = np.log(data_clustering.Close.div(data_clustering.Close.shift(1)))
data_clustering.drop(columns= ['Volume', 'Close'], inplace= True)
data_clustering.dropna(inplace= True)
data_clustering.loc[data_clustering.vol_ch > 3, "vol_ch"] = np.nan
data_clustering.loc[data_clustering.vol_ch < -3, "vol_ch"] = np.nan
data_clustering.dropna(inplace= True)
data_clustering
```

```
/Users/aamir/opt/anaconda3/lib/python3.8/site-packages/pandas/core/arraylike.py:358: RuntimeWarning:
divide by zero encountered in log
  result = getattr(ufunc, method)(*inputs, **kwargs)
```

Out[37]:

	vol_ch	returns
1	-0.708335	0.001505
2	-1.167460	0.002090
3	-0.486810	0.005912
4	-1.518955	0.002457
5	2.403742	0.018925
...	...	...
36163	0.439863	-0.006146
36164	-0.233129	-0.003690
36165	0.848040	0.004058
36166	-0.708801	-0.003982
36167	0.290111	-0.010002

36150 rows × 2 columns

In [38]:
```python
X = data_clustering.iloc[:, :].values
X
```

Out[38]:
```
array([[-0.70833531,  0.00150508],
       [-1.16745985,  0.00209036],
       [-0.48681043,  0.00591171],
       ...,
       [ 0.84803985,  0.0040582 ],
       [-0.70880121, -0.00398228],
       [ 0.2901106 , -0.0100025 ]])
```

In [39]:
```
# Applying k-means clustering
kmeans = KMeans(n_clusters = 10, init = 'k-means++', random_state= 42)
y_kmeans = kmeans.fit_predict(X)
y_kmeans
```

Out[39]:
```
array([3, 7, 8, ..., 2, 3, 0], dtype=int32)
```

In [40]:
```
# Visualising the clusters
plt.scatter(X[y_kmeans == 0, 0], X[y_kmeans == 0, 1], s = 100, c = 'red', label = 'Cluster 1')
plt.scatter(X[y_kmeans == 1, 0], X[y_kmeans == 1, 1], s = 100, c = 'blue', label = 'Cluster 2')
plt.scatter(X[y_kmeans == 2, 0], X[y_kmeans == 2, 1], s = 100, c = 'green', label = 'Cluster 3')
plt.scatter(X[y_kmeans == 3, 0], X[y_kmeans == 3, 1], s = 100, c = 'cyan', label = 'Cluster 4')
plt.scatter(X[y_kmeans == 4, 0], X[y_kmeans == 4, 1], s = 100, c = 'magenta', label = 'Cluster 5')
plt.scatter(X[y_kmeans == 5, 0], X[y_kmeans == 5, 1], s = 100, c = 'purple', label = 'Cluster 6')
plt.scatter(X[y_kmeans == 6, 0], X[y_kmeans == 6, 1], s = 100, c = 'black', label = 'Cluster 7')
plt.scatter(X[y_kmeans == 7, 0], X[y_kmeans == 7, 1], s = 100, c = 'brown', label = 'Cluster 8')
plt.scatter(X[y_kmeans == 8, 0], X[y_kmeans == 8, 1], s = 100, c = 'yellow', label = 'Cluster 9')
plt.scatter(X[y_kmeans == 9, 0], X[y_kmeans == 9, 1], s = 100, c = 'pink', label = 'Cluster 10')
plt.scatter(X[y_kmeans == 10, 0], X[y_kmeans == 10, 1], s = 100, c = 'violet', label = 'Cluster 11')
plt.xlabel('Volume Change')
plt.ylabel('Returns')
plt.legend()
plt.show()
```

In [41]:
```
# Adding the cluster column to the dataframe
data_clustering['cluster'] = y_kmeans
data_clustering
```

Out[41]:

	vol_ch	returns	cluster
1	-0.708335	0.001505	3
2	-1.167460	0.002090	7
3	-0.486810	0.005912	8
4	-1.518955	0.002457	7
5	2.403742	0.018925	9
...	...	...	...
36163	0.439863	-0.006146	4
36164	-0.233129	-0.003690	1
36165	0.848040	0.004058	2
36166	-0.708801	-0.003982	3
36167	0.290111	-0.010002	0

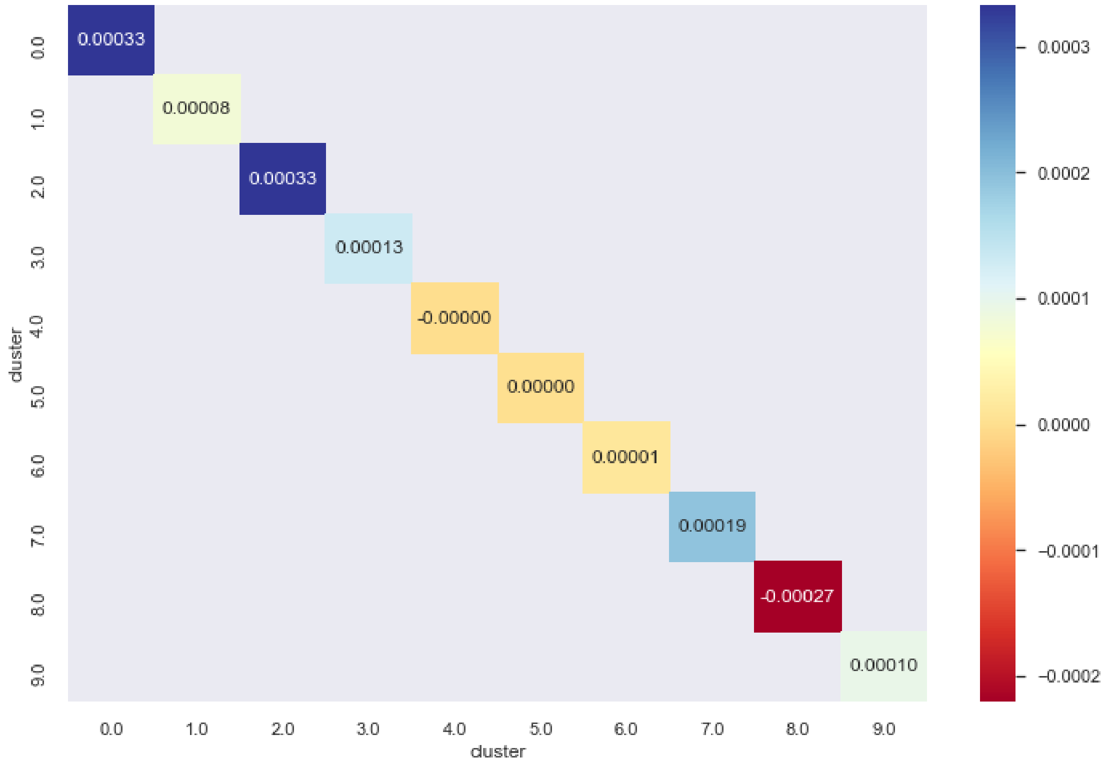
36150 rows × 3 columns

In [42]:
```
# Calculating mean returns for the next hour and plotting it in a matrix for each cluster by calculati
ng the average mean return over the whole timeframe
matrix_clustering = pd.crosstab(data_clustering.cluster.shift(), data_clustering.cluster.shift(),
                     values = data_clustering.returns, aggfunc = np.mean)
matrix_clustering
```

Out[42]:

cluster	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0
cluster										
0.0	0.000333	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1.0	NaN	0.000079	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2.0	NaN	NaN	0.000332	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3.0	NaN	NaN	NaN	0.000131	NaN	NaN	NaN	NaN	NaN	NaN
4.0	NaN	NaN	NaN	NaN	-0.000002	NaN	NaN	NaN	NaN	NaN
5.0	NaN	NaN	NaN	NaN	NaN	0.000001	NaN	NaN	NaN	NaN
6.0	NaN	NaN	NaN	NaN	NaN	NaN	0.000015	NaN	NaN	NaN
7.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.000194	NaN	NaN
8.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	-0.000268	NaN
9.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.000096

In [43]:
```
# Plotting the above matrix in a graph form for better visualisation and understanding
plt.figure(figsize=(12, 8))
sns.set(font_scale=1)
sns.heatmap(matrix_clustering, cmap = "RdYlBu", annot = True, robust = True, fmt = ".5f")
plt.show()
```



In [44]:
```
# Trading position -> long(1) for all bars: Buy-and-Hold
data_clustering["position"] = 1
data_clustering
```

Out[44]:

	vol_ch	returns	cluster	position
1	-0.708335	0.001505	3	1
2	-1.167460	0.002090	7	1
3	-0.486810	0.005912	8	1
4	-1.518955	0.002457	7	1
5	2.403742	0.018925	9	1
...	...	...	...	...
36163	0.439863	-0.006146	4	1
36164	-0.233129	-0.003690	1	1
36165	0.848040	0.004058	2	1
36166	-0.708801	-0.003982	3	1
36167	0.290111	-0.010002	0	1

36150 rows × 4 columns

- We set value=1 (hold onto the asset) for every instance initially because we only want to sell our asset when we get a selling signal (value=0) from our model

In [45]:
```
# Generating sell signals for the clusters having negative return for the next hour
condition1 = data_clustering.cluster == 4
condition2 = data_clustering.cluster == 8
data_clustering.loc[condition1|condition2, "position"] = 0
data_clustering
```

Out[45]:

	vol_ch	returns	cluster	position
1	-0.708335	0.001505	3	1
2	-1.167460	0.002090	7	1
3	-0.486810	0.005912	8	0
4	-1.518955	0.002457	7	1
5	2.403742	0.018925	9	1
...	...	...	...	...
36163	0.439863	-0.006146	4	0
36164	-0.233129	-0.003690	1	1
36165	0.848040	0.004058	2	1
36166	-0.708801	-0.003982	3	1
36167	0.290111	-0.010002	0	1

36150 rows × 4 columns

In [46]:
```
data_clustering.position.value_counts()
```

Out[46]:
```
1    27259
0     8891
Name: position, dtype: int64
```

In [47]:
```
# Calculating returns based on the trading strategy
data_clustering["strategy_kmeans"] = data_clustering.position.shift(1) * data_clustering["returns"]
data_clustering
```

Out[47]:

	vol_ch	returns	cluster	position	strategy_kmeans
1	-0.708335	0.001505	3	1	NaN
2	-1.167460	0.002090	7	1	0.002090
3	-0.486810	0.005912	8	0	0.005912
4	-1.518955	0.002457	7	1	0.000000
5	2.403742	0.018925	9	1	0.018925
...	...	...	...	...	...
36163	0.439863	-0.006146	4	0	-0.006146
36164	-0.233129	-0.003690	1	1	-0.000000
36165	0.848040	0.004058	2	1	0.004058
36166	-0.708801	-0.003982	3	1	-0.003982
36167	0.290111	-0.010002	0	1	-0.010002

36150 rows × 5 columns

In [48]:
```
# Comparing the rewards with the baseline model by calculating investment multiple for both
data_clustering[["returns", "strategy_kmeans"]].sum().apply(np.exp)
```

Out[48]:
```
returns            12.682299
strategy_kmeans    47.176133
dtype: float64
```

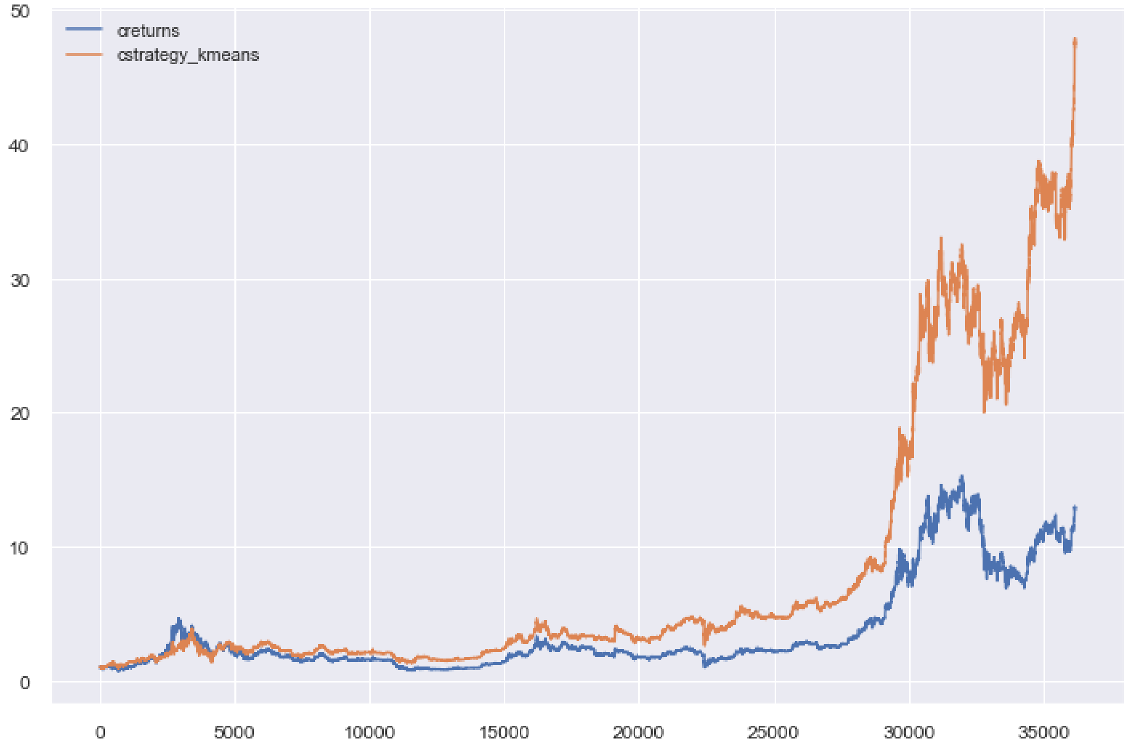
In [49]:
```
# Normalized price with base = 1 for strategy
data_clustering["cstrategy_kmeans"] = data_clustering["strategy_kmeans"].cumsum().apply(np.exp)
data_clustering
```

Out[49]:

	vol_ch	returns	cluster	position	strategy_kmeans	cstrategy_kmeans
1	-0.708335	0.001505	3	1	NaN	NaN
2	-1.167460	0.002090	7	1	0.002090	1.002093
3	-0.486810	0.005912	8	0	0.005912	1.008034
4	-1.518955	0.002457	7	1	0.000000	1.008034
5	2.403742	0.018925	9	1	0.018925	1.027292
...	...	...	...	...	...	...
36163	0.439863	-0.006146	4	0	-0.006146	47.646763
36164	-0.233129	-0.003690	1	1	-0.000000	47.646763
36165	0.848040	0.004058	2	1	0.004058	47.840516
36166	-0.708801	-0.003982	3	1	-0.003982	47.650380
36167	0.290111	-0.010002	0	1	-0.010002	47.176133

36150 rows × 6 columns

In [50]:
```
# Visualising the performance of the model compared to baseline model
data_clustering["creturns"] = data_clustering.returns.cumsum().apply(np.exp)
data_clustering[["creturns", "cstrategy_kmeans"]].plot(figsize = (12 , 8), fontsize = 12)
plt.show()
```



Evaluation

In [51]:
```
# Annualized returns comparison (Reward associated with both models)
tp_year = 24 * 365.25
# Calculating 1 hour trading periods per year
ann_mean = data_clustering[["returns", "strategy_kmeans"]].mean() * tp_year
ann_mean
```

Out[51]:
```
returns            0.615974
strategy_kmeans    0.934554
dtype: float64
```

- Here we can clearly see that annualised return obtained by k-means clustering trading strategy is much higher than our baseline model

In [52]:
```
# Annualized standard deviation comparison (Risk associated with both models)
ann_std = data_clustering[["returns", "strategy_kmeans"]].std() * np.sqrt(tp_year)
ann_std
```

Out[52]:
```
returns            0.905397
strategy_kmeans    0.788502
dtype: float64
```

- Here we can see the risk associated with our asset has reduced. Standard Deviation of 79% is still a high risk investment but we already knew about that from the domain knowledge that investing in crypto is highly risky

In [53]:
```
# Comparing sharpe ratio for both models
sharpe = (np.exp(ann_mean) - 1) / ann_std
sharpe
```

Out[53]:
```
returns            0.940426
strategy_kmeans    1.960778
dtype: float64
```

- The sharpe ratio of our machine learning model is more than double the sharpe ratio of baseline model. This indicates a much better performance from the unsupervised model which we implemented

In [ ]:

Applying Logistic Regression to develop a trading strategy (Supervised Learning)

In [54]:
```
data_clustering
```

Out[54]:

	vol_ch	returns	cluster	position	strategy_kmeans	cstrategy_kmeans	creturns
1	-0.708335	0.001505	3	1	NaN	NaN	1.001506
2	-1.167460	0.002090	7	1	0.002090	1.002093	1.003602
3	-0.486810	0.005912	8	0	0.005912	1.008034	1.009552
4	-1.518955	0.002457	7	1	0.000000	1.008034	1.012036
5	2.403742	0.018925	9	1	0.018925	1.027292	1.031370
...	...	...	...	...	...	...	...
36163	0.439863	-0.006146	4	0	-0.006146	47.646763	12.856169
36164	-0.233129	-0.003690	1	1	-0.000000	47.646763	12.808817
36165	0.848040	0.004058	2	1	0.004058	47.840516	12.860904
36166	-0.708801	-0.003982	3	1	-0.003982	47.650380	12.809790
36167	0.290111	-0.010002	0	1	-0.010002	47.176133	12.682299

36150 rows × 7 columns

In [55]:
```
# Generating hold and sell values based on change in absolute values of volume
output = []
for i in range(0, len(data)-1):
    x = data.Volume[i] - data.Volume[i+1]
    if(x<0):
        y = 0
        output.append(y)
    else:
            y = 1
            output.append(y)
```

In [56]:
```
# Populating the dataframe with obtained values
output.append(0)
data_clustering['output'] = output
data_clustering
```

Out[56]:

	vol_ch	returns	cluster	position	strategy_kmeans	cstrategy_kmeans	creturns	output
1	-0.708335	0.001505	3	1	NaN	NaN	1.001506	1
2	-1.167460	0.002090	7	1	0.002090	1.002093	1.003602	1
3	-0.486810	0.005912	8	0	0.005912	1.008034	1.009552	1
4	-1.518955	0.002457	7	1	0.000000	1.008034	1.012036	0
5	2.403742	0.018925	9	1	0.018925	1.027292	1.031370	0
...	...	...	...	...	...	...	...	...
36163	0.439863	-0.006146	4	0	-0.006146	47.646763	12.856169	1
36164	-0.233129	-0.003690	1	1	-0.000000	47.646763	12.808817	0
36165	0.848040	0.004058	2	1	0.004058	47.840516	12.860904	1
36166	-0.708801	-0.003982	3	1	-0.003982	47.650380	12.809790	0
36167	0.290111	-0.010002	0	1	-0.010002	47.176133	12.682299	0

36150 rows × 8 columns

In [57]:
```
# Data cleaning
data_clustering.dropna(inplace= True)
data_clustering
```

Out[57]:

	vol_ch	returns	cluster	position	strategy_kmeans	cstrategy_kmeans	creturns	output
2	-1.167460	0.002090	7	1	0.002090	1.002093	1.003602	1
3	-0.486810	0.005912	8	0	0.005912	1.008034	1.009552	1
4	-1.518955	0.002457	7	1	0.000000	1.008034	1.012036	0
5	2.403742	0.018925	9	1	0.018925	1.027292	1.031370	0
6	0.837305	0.003594	2	1	0.003594	1.030991	1.035084	0
...	...	...	...	...	...	...	...	...
36163	0.439863	-0.006146	4	0	-0.006146	47.646763	12.856169	1
36164	-0.233129	-0.003690	1	1	-0.000000	47.646763	12.808817	0
36165	0.848040	0.004058	2	1	0.004058	47.840516	12.860904	1
36166	-0.708801	-0.003982	3	1	-0.003982	47.650380	12.809790	0
36167	0.290111	-0.010002	0	1	-0.010002	47.176133	12.682299	0

36149 rows × 8 columns

In [58]:
```
# Creating variables for logistic regression model
X = data_clustering.iloc[:, [1,0]].values
y = data_clustering.iloc[:, -1].values
```

In [59]:
```
# Training the logistic regression model
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state = 0)
classifier.fit(X, y)
```

Out[59]:
```
▾         LogisticRegression
LogisticRegression(random_state=0)
```

In [60]:
```
# Predicting the output positions
y_pred = classifier.predict(X)
y_pred
```

Out[60]:
```
array([0, 0, 0, ..., 1, 0, 1])
```

In [61]:
```
# Calculating returns based on the trading strategy
x = pd.DataFrame(X)
data_clustering['y_pred'] = y_pred
data_clustering['strategy_lr'] = data_clustering['y_pred'].shift(1) * data_clustering['returns']
data_clustering
```

Out[61]:

	vol_ch	returns	cluster	position	strategy_kmeans	cstrategy_kmeans	creturns	output	y_pred	strategy_lr
2	-1.167460	0.002090	7	1	0.002090	1.002093	1.003602	1	0	NaN
3	-0.486810	0.005912	8	0	0.005912	1.008034	1.009552	1	0	0.000000
4	-1.518955	0.002457	7	1	0.000000	1.008034	1.012036	0	0	0.000000
5	2.403742	0.018925	9	1	0.018925	1.027292	1.031370	0	1	0.000000
6	0.837305	0.003594	2	1	0.003594	1.030991	1.035084	0	1	0.003594
...	...	...	...	...	...	...	...	...	...	...
36163	0.439863	-0.006146	4	0	-0.006146	47.646763	12.856169	1	1	-0.006146
36164	-0.233129	-0.003690	1	1	-0.000000	47.646763	12.808817	0	0	-0.003690
36165	0.848040	0.004058	2	1	0.004058	47.840516	12.860904	1	1	0.000000
36166	-0.708801	-0.003982	3	1	-0.003982	47.650380	12.809790	0	0	-0.003982
36167	0.290111	-0.010002	0	1	-0.010002	47.176133	12.682299	0	1	-0.000000

36149 rows × 10 columns

In [62]:
```
# Comparing the rewards with other models by calculating investment multiple for all
data_clustering[["returns","strategy_kmeans", "strategy_lr"]].sum().apply(np.exp)
```

Out[62]:
```
returns            12.663225
strategy_kmeans    47.176133
strategy_lr        15.983888
dtype: float64
```

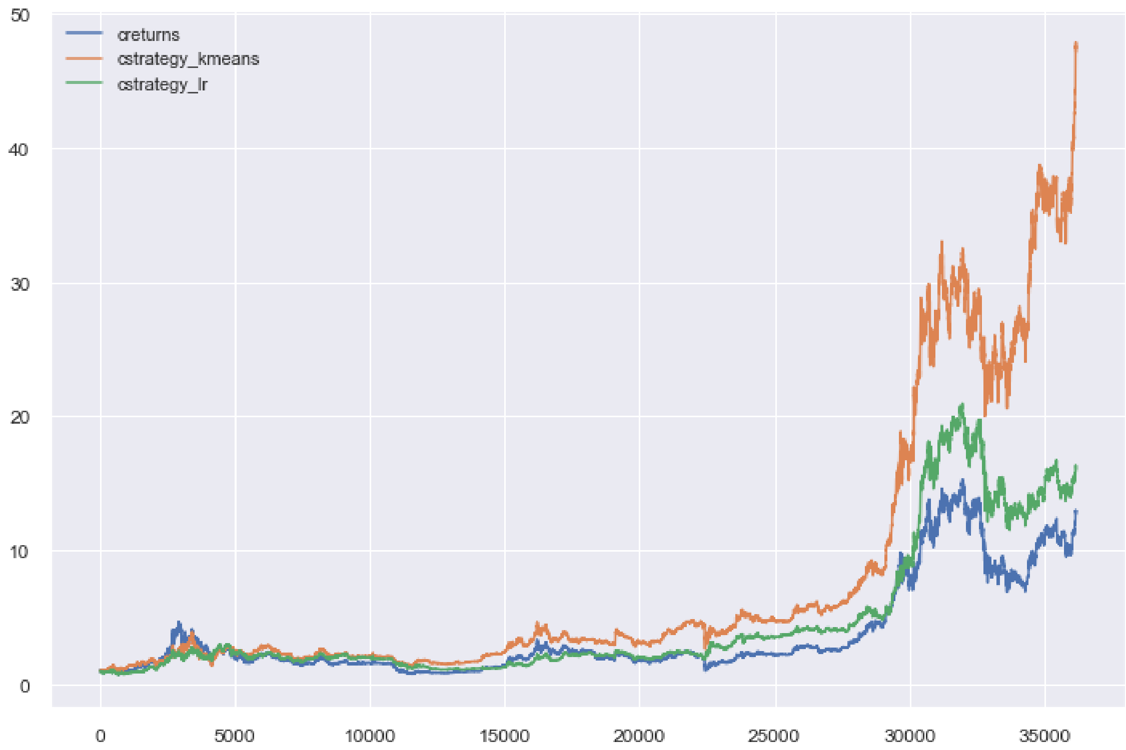
In [63]:
```
# Normalized price with base = 1 for strategy
data_clustering["cstrategy_lr"] = data_clustering["strategy_lr"].cumsum().apply(np.exp)
data_clustering
```

Out[63]:

	vol_ch	returns	cluster	position	strategy_kmeans	cstrategy_kmeans	creturns	output	y_pred	strategy_lr	cstrategy_lr
2	-1.167460	0.002090	7	1	0.002090	1.002093	1.003602	1	0	NaN	NaN
3	-0.486810	0.005912	8	0	0.005912	1.008034	1.009552	1	0	0.000000	1.000000
4	-1.518955	0.002457	7	1	0.000000	1.008034	1.012036	0	0	0.000000	1.000000
5	2.403742	0.018925	9	1	0.018925	1.027292	1.031370	0	1	0.000000	1.000000
6	0.837305	0.003594	2	1	0.003594	1.030991	1.035084	0	1	0.003594	1.003600
...	...	...	...	...	...	...	...	...	...	...	...
36163	0.439863	-0.006146	4	0	-0.006146	47.646763	12.856169	1	1	-0.006146	16.106992
36164	-0.233129	-0.003690	1	1	-0.000000	47.646763	12.808817	0	0	-0.003690	16.047668
36165	0.848040	0.004058	2	1	0.004058	47.840516	12.860904	1	1	0.000000	16.047668
36166	-0.708801	-0.003982	3	1	-0.003982	47.650380	12.809790	0	0	-0.003982	15.983888
36167	0.290111	-0.010002	0	1	-0.010002	47.176133	12.682299	0	1	-0.000000	15.983888

36149 rows × 11 columns

In [64]:
```
# Visualising the performance of the model compared to other models
data_clustering[["creturns", "cstrategy_kmeans", "cstrategy_lr"]].plot(figsize = (12 , 8), fontsize =
12)
plt.show()
```



Evaluation

In [65]:
```python
# Annualized returns comparison (Reward associated with all models)
tp_year = 24 * 365.25
# Calculating 1 hour trading periods per year
ann_mean = data_clustering[["returns", "strategy_kmeans", "strategy_lr"]].mean() * tp_year
ann_mean
```

Out[65]:
```
returns            0.615626
strategy_kmeans    0.934554
strategy_lr        0.672117
dtype: float64
```

- Here we can clearly see that annualised return obtained by **k-means** clustering trading strategy is much higher than all other models

In [66]:
```python
# Annualized standard deviation comparison (Risk associated with all models)
ann_std = data_clustering[["returns", "strategy_kmeans", "strategy_lr"]].std() * np.sqrt(tp_year)
ann_std
```

Out[66]:
```
returns            0.905409
strategy_kmeans    0.788502
strategy_lr        0.707954
dtype: float64
```

- Here we can see the risk associated with our asset has reduced the most in **logistic regression** model. Standard Deviation of 70% is still a high risk investment but we already knew about that from the domain knowledge that investing in crypto is highly risky

In [67]:
```python
# Comparing sharpe ratio for all models
sharpe = (np.exp(ann_mean) - 1) / ann_std
sharpe
```

Out[67]:
```
returns            0.939702
strategy_kmeans    1.960778
strategy_lr        1.353731
dtype: float64
```

- The sharpe ratio of **k-means** clustering model is more than double the sharpe ratio of baseline model. It is also much higher than the logistic regression model. This indicates a much better performance from the unsupervised model which we implemented than both the other models