Solution - Crypto Time Series

Solution: Crypto Time Series

Import Libraries and Dataset

In [1]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns
sns.set_style('whitegrid')
```

In [2]:
```
df = pd.read_csv('data/bitcoin_time_series.csv', index_col=0)
```

Exploratory Analysis and Data Cleaning

In [3]:
```
# Example observations
df.head()
```

Out[3]:

	avg_block_size	avg_trx_cost	confirmation_time	difficulty	hash_rate	market_cap	pric
2015-01-01	0.184772	22.262785	7.150000	4.064096e+10	335365.290092	4.317111e+09	315
2015-01-02	0.252396	17.550759	6.933333	4.064096e+10	323243.653101	4.324529e+09	316
2015-01-03	0.289052	15.004813	6.433333	4.064096e+10	331324.744428	4.136728e+09	302
2015-01-04	0.296036	13.202969	8.033333	4.064096e+10	335365.290092	3.708212e+09	270
2015-01-05	0.320334	12.447482	6.816667	4.064096e+10	339405.835756	3.789717e+09	276

The first thing to notice is that the values for each feature are on vastly different scales.

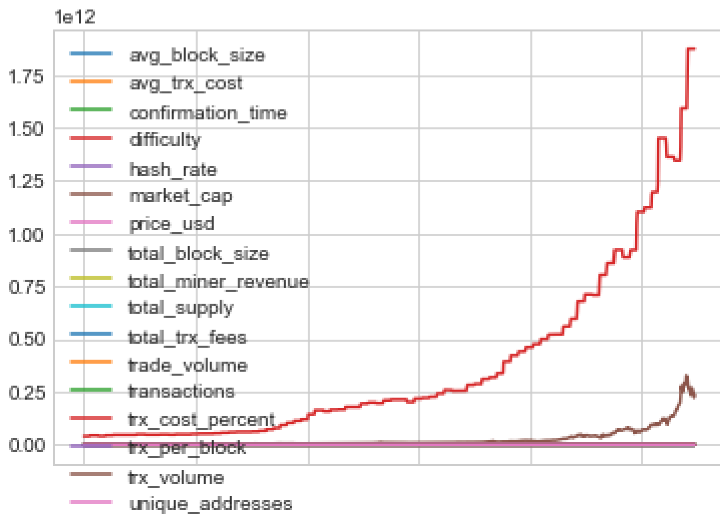
Since we're dealing with time series data, a traditional histogram would not be too helpful.

However, since the values are on different scales, a single time series plot is also not very enlightening:

In [4]:
```
# Not very useful
```

```
df.plot()
```

Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x101877828>

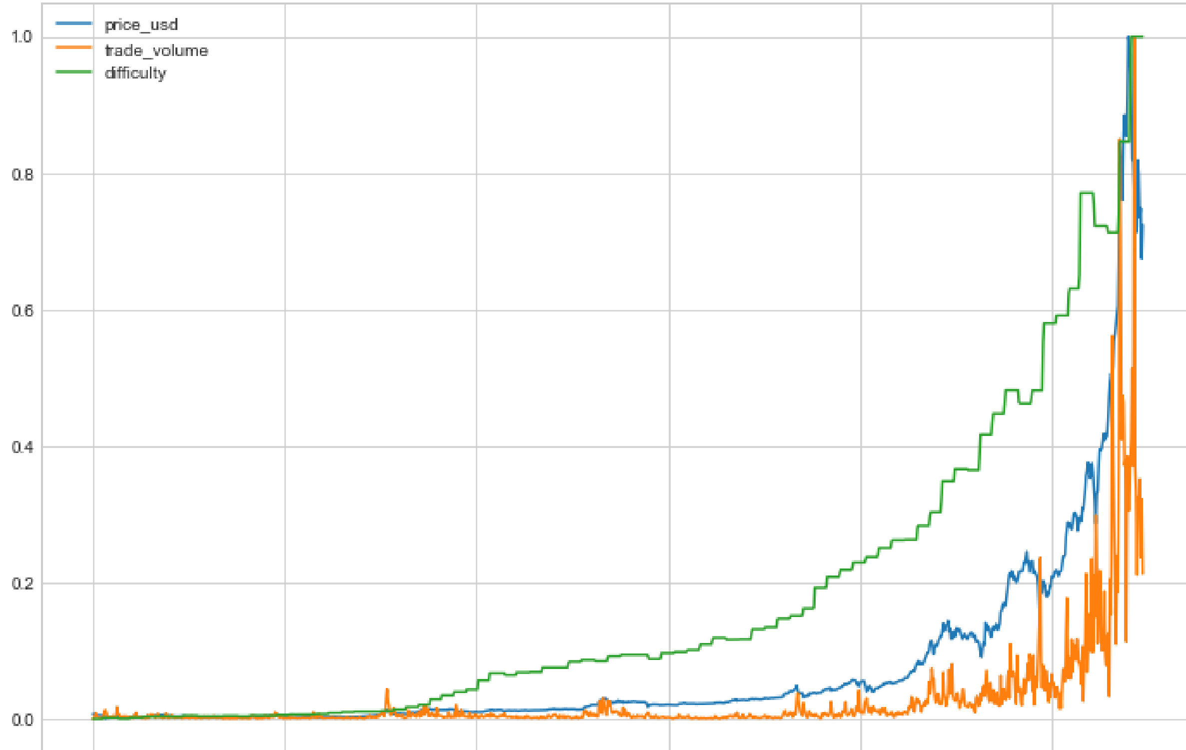


Instead, just for the purpose of creating digestable data visualizations, we can first normalize the data and then pick of a subset of features to plot.

For example:

In [5]:
```
# Normalize to range of 0 to 1
df_norm = (df - df.min())/(df.max() - df.min())

# Plot subset of features
df_norm[['price_usd', 'trade_volume', 'difficulty']].plot(figsize=(12,8))
```

Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0x10d7c7438>



We won't dwell on the visualizations for this challenge, since the actual objective is quite involved (**Tip:** Remember to streamline your analyses as much as possible!).

However, we should first still check for missing values.

In [6]:
```
# Check for missing values
df.isnull().sum()
```

Out[6]:
```
avg_block_size          0
avg_trx_cost            0
confirmation_time       0
difficulty              0
hash_rate               0
market_cap              0
price_usd               0
total_block_size        0
total_miner_revenue     0
total_supply            0
total_trx_fees          0
trade_volume            3
transactions            0
trx_cost_percent        0
trx_per_block           0
trx_volume              0
unique_addresses        0
dtype: int64
```

It looks like `trade_volume` has 3 missing values. Since there are so few, we can take a quick look at them:

In [7]:
```
df[df.trade_volume.isnull()]
```

Out[7]:

	avg_block_size	avg_trx_cost	confirmation_time	difficulty	hash_rate	market_cap	pric
2015-01-07	0.345613	11.061633	7.433333	4.064096e+10	299000.379118	3.791828e+09	276
2015-02-03	0.390975	8.570101	9.066667	4.127287e+10	276977.568109	3.262655e+09	236
2015-09-03	0.600776	6.305732	10.750000	5.425663e+10	407263.870142	3.313828e+09	227

Nothing else about those dates seems strange, so it may be just a matter of data collection errors.

For missing values in time series data, especially when so few are missing, we should perform a simple [linear interpolation](#).

In [8]:
```
# Linear interpolation
df.interpolate(inplace=True)
```

Building ABT

For financial time series, log returns/changes are generally preferred over raw prices/values. Log returns have a number of useful qualities, such as mitigating [autocorrelation](#) and normalizing the features (so that all features are comparable on the same scale).

Therefore, to build our analytical base table, we will calculate log changes for our data.

In [9]:
```
# Calculate log changes
abt = np.log( df / df.shift(1) )
```

Next, we will create the target variable: whether Bitcoin's price grew by over 2% over the next day.

In [10]:
```
# Create a target variable
abt['y'] = ((df.price_usd.shift(-1) / df.price_usd) > 1.02) * 1
```

In [11]:
```
abt.head()
```

Out[11]:

	avg_block_size	avg_trx_cost	confirmation_time	difficulty	hash_rate	market_cap	price_usd	tota
2015-01-01	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Nal

	avg_block_size	avg_trx_cost	confirmation_time	difficulty	hash_rate	market_cap	price_usd	tot
2015-01-02	0.311876	-0.237819	-0.030772	0.0	-0.036814	0.001717	0.001424	0.0
2015-01-03	0.135607	-0.156726	-0.074848	0.0	0.024693	-0.044398	-0.044698	0.0
2015-01-04	0.023876	-0.127929	0.222107	0.0	0.012121	-0.109355	-0.109659	0.0
2015-01-05	0.078883	-0.058923	-0.164229	0.0	0.011976	0.021742	0.021435	0.0

By calculating log changes, we've just introduced new `NaN` values at the edge of our dataset. Let's just drop that date.

In [12]:
```
abt.dropna(inplace=True)
```

Also, one useful thing to check is the actual percentage of the target variable that belongs to the positive class. If it's too low, then we may need to use [methods to handle imbalanced classes](#).

In [13]:
```
abt.y.mean()
```

Out[13]:
```
0.23013698630136986
```

In this situation, it looks like we're fine on that front.

Correlations

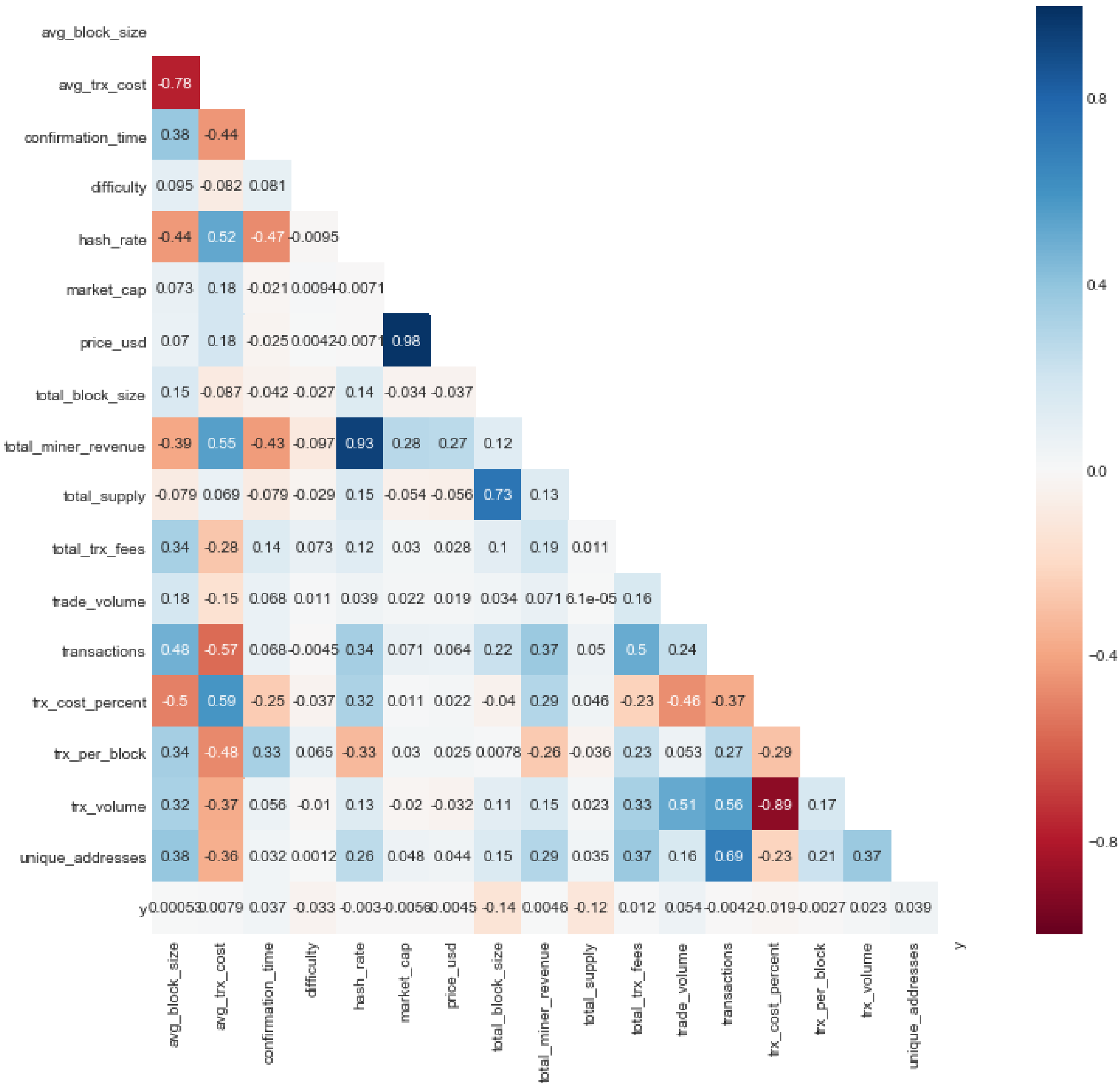
After building the ABT for a time series dataset, we like to take a step back and plot the correlations between the different features before proceeding. This provides an intuitive understanding of the relationships between the features.

In [14]:
```
def plot_corr_heatmap(corr):
    plt.figure(figsize=(12,11))

    # Make to cover top triangle
    mask = np.zeros_like(corr)
    mask[np.triu_indices_from(mask)] = True

    # Plot heatmap
    sns.heatmap(corr,
                vmin=-1.0,
                vmax=1.0,
```

```
                    mask=mask,
                    cmap='RdBu',
                    annot=True)
    plt.show()
```

In [15]:

```
plot_corr_heatmap( abt.corr() )
```



Walk-Forward Analysis

It's time to proceed with our walk-forward analysis. We'll need a few more libraries to help us.

In [16]:
```
# Machine learning
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_auc_score

# To calculate last day of a month
from calendar import monthrange

# To add/subtract days
from datetime import datetime
from dateutil.relativedelta import relativedelta
```

First, we should convert our indices into `datetime` objects. This is good practice when working with time series data.

In [17]:
```
# Convert indices to datetime objects
abt.index = pd.Series( pd.to_datetime( abt.index, format='%Y-%m-%d' ) )
```

We'll use a helper function to calculate AUROC score:

In [18]:
```
def calc_roc_score(clf, X_test, y_test):
    pred = clf.predict_proba(X_test)
    pred = [p[1] for p in pred]
    return roc_auc_score(y_test, pred)
```

Next, we will structure our walk-forward analysis.

Tip: There are many ways to structure the walk-forward analysis. However, it's imperative that you make sure at each window's start date, you are *only* using data available before that date. These structures can get tricky to write, especially if you're simultaneously testing multiple modeling methods like in this challenge. One tip is to write the code for just one method first so that you can make sure it works before duplicating it.

In [19]:
```
year = 2017
months = range(1, 13)

method1_results = []
method2_results = []
```

```
method3_results = []

for month in months:
    # Test window start date
    test_start = datetime(year, month, 1)

    # Test window end date
    last_day = monthrange(year, month)[1]
    test_end = datetime(year,month,last_day)

    # Train window start dates
    train1_start = test_start - relativedelta(months=1)
    train2_start = test_start - relativedelta(months=3)

    # Train window end date
    train_end = test_start - relativedelta(days=1)

    # Test set
    test = abt.loc[test_start:test_end,:]
    y_test = test.y
    X_test = test.drop('y', axis=1)

    # Train set 1
    train1 = abt.loc[train1_start:train_end,:]
    y_train1 = train1.y
    X_train1 = train1.drop('y', axis=1)

    # Train set 2
    train2 = abt.loc[train2_start:train_end,:]
    y_train2 = train2.y
    X_train2 = train2.drop('y', axis=1)

    # Train set 3
    train3 = abt.loc[:train_end,:]
    y_train3 = train3.y
    X_train3 = train3.drop('y', axis=1)

    # Fit models
    clf1 = RandomForestClassifier(random_state=123).fit(X_train1, y_trair
    clf2 = RandomForestClassifier(random_state=123).fit(X_train2, y_trair
    clf3 = RandomForestClassifier(random_state=123).fit(X_train3, y_trair

    # Store AUROC
    method1_results.append(calc_roc_score(clf1, X_test, y_test))
    method2_results.append(calc_roc_score(clf2, X_test, y_test))
    method3_results.append(calc_roc_score(clf3, X_test, y_test))
```

We save our results in a new table.

In [20]:
```
results_df = pd.DataFrame({
    'month' : months,
    'method1' : method1_results,
    'method2' : method2_results,
    'method3' : method3_results
})
```

In [21]:
```
results_df
```

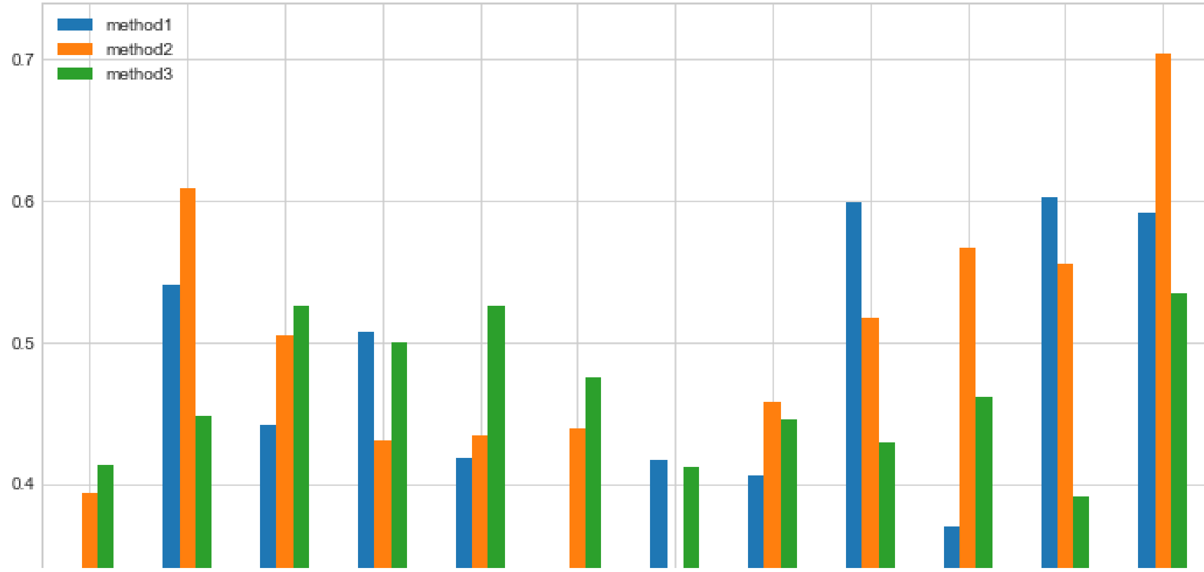
Out[21]:

	method1	method2	method3	month
0	0.320652	0.394022	0.413043	1
1	0.540936	0.608187	0.447368	2
2	0.441919	0.505051	0.525253	3
3	0.506944	0.430556	0.500000	4
4	0.418803	0.433761	0.525641	5
5	0.261574	0.439815	0.474537	6
6	0.416667	0.307143	0.411905	7
7	0.406250	0.458333	0.445833	8
8	0.599432	0.517045	0.428977	9
9	0.369658	0.566239	0.461538	10
10	0.602222	0.555556	0.391111	11
11	0.590909	0.704545	0.534091	12

We can visualize each model's performance for each month:

In [22]:
```
results_df.plot(kind='bar', x='month', figsize=(12,11))
```

Out[22]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a139ad828>
```



Tip reminder: If you're given this type challenge and discover your models are not predictive, don't panic! Employers are looking for the way you structure your analysis and whether you can avoid errors… they are not expecting you to build a profitable trading algorithm in under 4 hours!

All in all, these models do not appear to be predictive.

In [23]:
```
results_df.median()
```

Out[23]:
```
method1     0.430361
method2     0.481692
method3     0.454453
month       6.500000
dtype: float64
```

One thing to note is that our models sometimes produced AUROC scores of under 0.5. This is totally plausible. An AUROC score of 0.5 means the model predicts no better than random, so for some months, our models performed **worse than random.**

In some other contexts, we may be able to just reverse the labels of our positive class, but this is different. We performed a walk-forward analysis to evaluate how our model would perform with different training set window sizes. We cannot retroactively tamper with the labels, and we certainly can't swap back and forth.

For example, let's say you just trained the model on July 1st, 2017. In hindsight, we could say that if we just swapped the positive class labels, we would've seen an AUROC of 0.7. However, at the time, there's no way of knowing that. If we followed our model for the month of July, we would have actually *realized* the worse-than-random AUROC of 0.3.

Tip: Remember to state the limitations of an analysis and what you would do to improve it if given more time or data. For example, for this analysis, we can try to improve our models by engineering more features (e.g. using more than 1 lag day), incorporate other time series datasets (e.g. data from other cryptocurrencies), or even try recurrent neural networks (especially if we can obtain more granular data).