INTRODUCTION AND METHODOLOGY

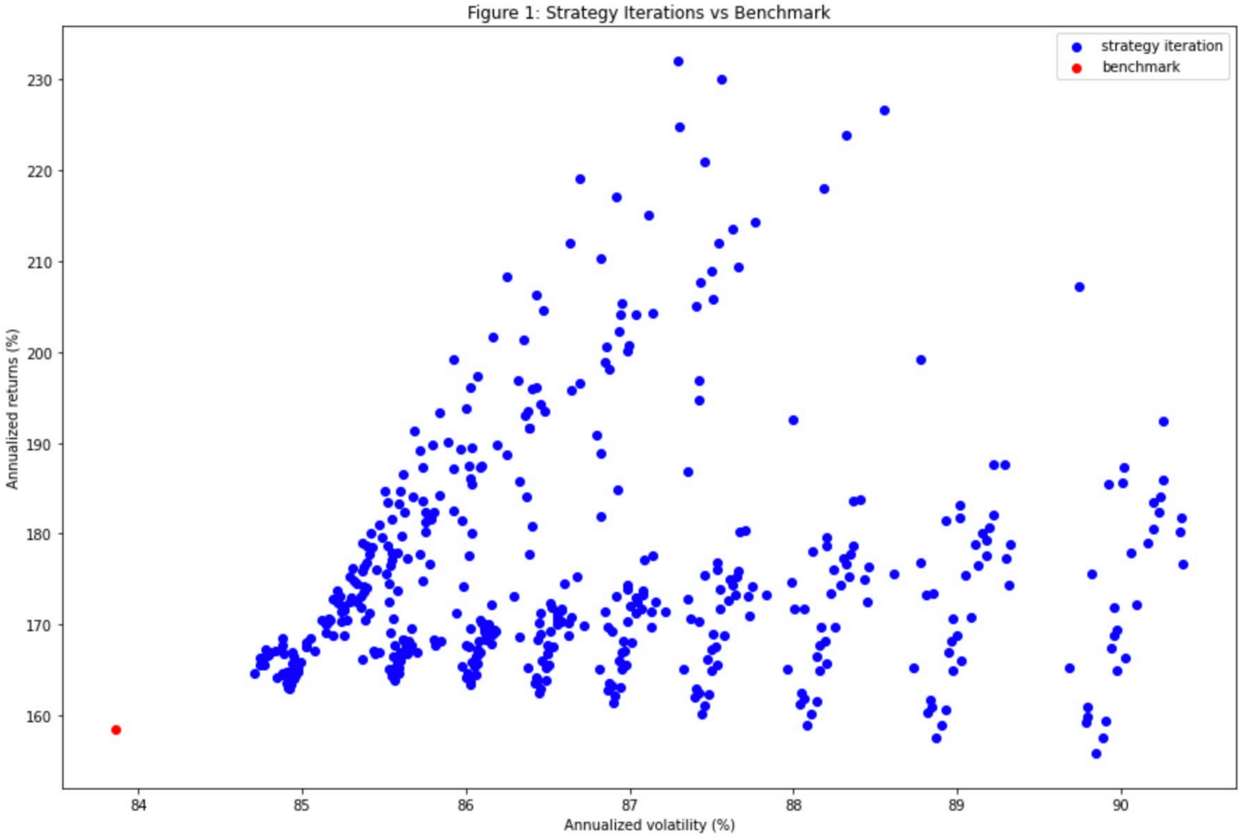
The objective of this paper is to find the optimal parameters for a simple moving average (SMA) and rebalancing strategy, to apply that strategy to a portfolio comprised of Bitcoin (BTC) and Ethereum (ETH), and to compare it against a benchmark strategy.

The trading strategy rebalances based the relationship between a short and long SMA computed from the ratio BTC:ETH. The short SMA- initially smaller- exceeding the long SMA ("crossing up"), suggests a future increase in the price of the underlying asset, while the inverse SMA relationship ("crossing down") points to a decrease. Given that the underlying asset is a ratio, a bullish signal implies a strengthening of BTC relative to ETH, while a bearish signal shows relative weakness. The strategy rebalances the portfolio every time the SMAs cross, selling a fixed percentage ("short percentage") of the weakening asset in order to buy the strengthening asset. The initial portfolio is comprised of 50% BTC and 50% ETH. The benchmark portfolio is the initial portfolio, ignoring all buy and sell signals.

Alpha Vantage (www.alphavantage.co) provides 1000 days of daily price data for BTC and ETH (data acquisition methodology is available in data.py). The first half of the data is used as a training set in order to find the optimal combination of short SMA length, long SMA length, and short percent size. The second half of the data is used to backtest the best strategies fitted to the training set. The strategy combines every instance of short SMA (5, 7, 9. 11, 13, 15 days), long SMA (20, 25, 30, 35, 40, 45, 50, 55 , 60 days), and short percentage (10, 20, 30, 40, 50, 60, 70, 80, 90%) to yield 486 unique parameter sets. Training details are available in training.py.

TRAINING RESULTS

Figure 1: Strategy Iterations vs Benchmark



The strategy uses data from 2019-06-23 to 2020-09-04 to train. The benchmark portfolio displays a mean annualized return of 158.6% and annualized standard deviation of 83.9% during the training period, resulting in a Sharpe ratio of 1.89. While every strategy iteration suffers from higher volatility than the benchmark (min: 84.7%, mean: 86.7%, max: 90.4%), an overwhelming majority (99.4%) boasts higher returns (min: 155.9%, mean: 175.4%, max: 232%). The average Sharpe ratio among the iterations is 202.2%, with 90.1% of iterations displaying a better ratio than the benchmark.

APPENDIX

All code is freely available at: https://github.com/PabloBaiocchi/simpleSmaStrategy

data.py

```
import pandas as pd

def getRawData(symbol,alphavantageApiKey):
    baseUrl='https://www.alphavantage.co/query'
    params={
        'function':'DIGITAL_CURRENCY_DAILY',
        'symbol':symbol,
        'market':'USD',
        'apikey':alphavantageApiKey
    }
    response=requests.get(baseUrl,params=params)
    return response.json()

def rawToDf(raw):
    timeSeries=raw['Time Series (Digital Currency Daily)']
    rows=[{'date':key,'price':timeSeries[key]['4a. close (USD)']} for key in timeSeries.keys()]
    df=pd.DataFrame(rows)
    df['date']=pd.to_datetime(df.date)
    df['price']=df.price.astype(float)
    return df

def getData(alphavantageApiKey):
    btcRaw=getRawData('BTC',alphavantageApiKey)
    ethRaw=getRawData('ETH',alphavantageApiKey)

    ethDf=rawToDf(ethRaw)
    btcDf=rawToDf(btcRaw)

    df=ethDf.merge(btcDf,on='date',suffixes=['_eth','_btc'])
    df['btc_eth']=df.price_btc/df.price_eth
    df.sort_values('date',inplace=True)
    return df

def storeData(filePath,alphavantageApiKey):
    data=getData(alphavantageApiKey)
    data.to_csv(filePath)
```

training.py

```
import itertools
import numpy as np
import pandas as pd

def rebalance(longAssetAmount,shortAssetAmount,longAssetPrice,shortAssetPrice,percentShort):
    cash=shortAssetAmount*shortAssetPrice*percentShort
    newShortAssetAmount=shortAssetAmount*(1-percentShort)
    newLongAssetAmount=longAssetAmount+cash/longAssetPrice
    return (newLongAssetAmount,newShortAssetAmount)

def getSmaTupples():
```

```
    short=np.arange(5,16,2)
    long=np.arange(20,61,5)
    return list(itertools.product(short,long))

def smaCross(short,long,short_before,long_before):
    if short_before<long_before and short>long:
        return 'cross-up'
    if short_before>long_before and short<long:
        return 'cross-down'
    return '-'

def getSignal(shortSma,longSma,priceSeries):
    df=priceSeries.copy().to_frame()
    rootCol='price'
    df.columns=[rootCol]
    shortSmaCol=f'sma_{shortSma}'
    longSmaCol=f'_sma_{longSma}'

    df[shortSmaCol]=df[rootCol].rolling(int(shortSma)).mean()
    df[longSmaCol]=df[rootCol].rolling(int(longSma)).mean()

    longSmaColBefore=f'{longSmaCol}_before'
    shortSmaColBefore=f'{shortSmaCol}_before'
    df[longSmaColBefore]=df[longSmaCol].shift(1)
    df[shortSmaColBefore]=df[shortSmaCol].shift(1)
    df.dropna(inplace=True)
    signal=df.apply(lambda row:
smaCross(row[shortSmaCol],row[longSmaCol],row[shortSmaColBefore],row[longSmaColBefore]),axis=1)
    return signal

def trainingCutoffIndex(df,trainingSize):
    return int(len(df)*trainingSize)

def getInitialPosition(df,initialInvestment):
    firstRow=df.iloc[0]
    eth=initialInvestment/2/firstRow.price_eth
    btc=initialInvestment/2/firstRow.price_btc
    return eth,btc

def runIteration(trainDf,signal,percentShort,initialInvestment):
    resultRows=[]
    frame=trainDf.iloc[len(trainDf)-len(signal):].copy()
    eth,btc=getInitialPosition(frame,initialInvestment)
    frame['signal']=signal
    for index,row in frame.iterrows():
        if row['signal']=='cross-down':
            eth,btc=rebalance(eth,btc,row.price_eth,row.price_btc,percentShort)
        if row['signal']=='cross-up':
            btc,eth=rebalance(btc,eth,row.price_btc,row.price_eth,percentShort)
        resultRows.append({'date':row.date,'eth':eth,'btc':btc})
    return pd.DataFrame(resultRows)

def getSignals(trainDf):
    resultList=[]
    for shortSma,longSma in getSmaTupples():
        signal=getSignal(shortSma,longSma,trainDf['btc_eth'])
```

```
        resultList.append({
            'shortSma':shortSma,
            'longSma':longSma,
            'signalSeries':signal
        })
    return resultList

def getTrainDf(df,percentTrain):
    df.sort_values('date',inplace=True)
    return df[:trainingCutoffIndex(df,percentTrain)].copy()

def normalizeSignalLengths(signalPojos):
    lengths=[len(pojo['signalSeries'])for pojo in signalPojos]
    shortest=min(lengths)
    for pojo in signalPojos:
        signal=pojo['signalSeries']
        signal=signal[len(signal)-shortest:]
        pojo['signalSeries']=signal

def train(df,percentTrain,initialInvestment):
    trainDf=getTrainDf(df,percentTrain)
    signalPojos=getSignals(trainDf)
    normalizeSignalLengths(signalPojos)
    percentShorts=np.arange(.1,1,.1)
    results=[]
    for pojo in signalPojos:
        for ps in percentShorts:
            resultFrame=runIteration(trainDf,pojo['signalSeries'],ps,initialInvestment)
            results.append({
                'shortSma':pojo['shortSma'],
                'longSma':pojo['longSma'],
                'percentShort':ps,
                'signalSeries':pojo['signalSeries'],
                'resultFrame':resultFrame
            })
    return results

def summarizeIteration(iteration,priceDf):
    combined=iteration['resultFrame'].merge(priceDf,on='date')
    combined['portfolio_value']=combined.eth*combined.price_eth+combined.btc*combined.price_btc
    combined['perc_return']=combined.portfolio_value.pct_change()
    combined.dropna(inplace=True)
    iteration['portfolioFrame']=combined[['date','portfolio_value','perc_return']].copy()
    iteration['annualizedReturn']=(1+combined.perc_return.mean())**365
    iteration['annualizedVolatility']=365**.5*combined.perc_return.std()
    return {
        'short_sma':iteration['shortSma'],
        'long_sma':iteration['longSma'],
        'percent_short':iteration['percentShort'],
        'annualized_return':iteration['annualizedReturn'],
        'annualized_volatility':iteration['annualizedVolatility']
    }

def summarizeTraining(results,priceDf):
    summaries=[summarizeIteration(iteration,priceDf) for iteration in results]
    summary=pd.DataFrame(summaries)
```

```
    summary['sharpe']=summary.annualized_return/summary.annualized_volatility
    return summary

def getBenchmark(dates,priceDf):
    benchmarkDf=dates.to_frame()
    eth,btc=getInitialPosition(priceDf[priceDf.date==dates.min()],100000)
    benchmarkDf['btc']=np.ones(len(benchmarkDf))*btc
    benchmarkDf['eth']=np.ones(len(benchmarkDf))*eth
    benchmark={
    'shortSma':0,
    'longSma':0,
    'percentShort':0,
    'resultFrame':benchmarkDf
    }
    benchmarkSummary=summarizeIteration(benchmark,priceDf)
```

In [1]:

```python
import data
import training
```

In [2]:

```python
csvPath='/Users/pablo/Desktop/simpleSmaStrategy/priceData.csv'
apiKey=''
# only needs to be called once
# data.storeData(csvPath,apiKey)
```

In [3]:

```python
import pandas as pd

df=pd.read_csv(csvPath,index_col=0)
```

In [4]:

```python
trainingResults=training.train(df,.5,100000)
```

In [5]:

```python
summary=training.summarizeTraining(trainingResults,df)
```

In [6]:

```python
summary.sort_values('sharpe',ascending=False)
```

Out[6]:

	short_sma	long_sma	percent_short	annualized_return	annualized_volatility	sharpe
17	5	25	0.9	2.320048	0.872952	2.657702
422	15	25	0.9	2.300259	0.875585	2.627110
8	5	20	0.9	2.247043	0.873057	2.573765
35	5	35	0.9	2.265319	0.885573	2.558026
26	5	30	0.9	2.237894	0.883249	2.533707
...	...	...	...	...	...	...
467	15	50	0.9	1.592814	0.897856	1.774019
485	15	60	0.9	1.594953	0.899089	1.773966
475	15	55	0.8	1.575550	0.888730	1.772810
395	13	55	0.9	1.575298	0.898857	1.752556
476	15	55	0.9	1.558731	0.898438	1.734934

486 rows × 6 columns

In [7]:

```python
benchmarkDates=trainingResults[0]['resultFrame'].date.copy()
benchmark=training.getBenchmark(benchmarkDates,df)
benchmark
```

Out[7]:

```
{'shortSma': 0,
 'longSma': 0,
 'percentShort': 0,
```

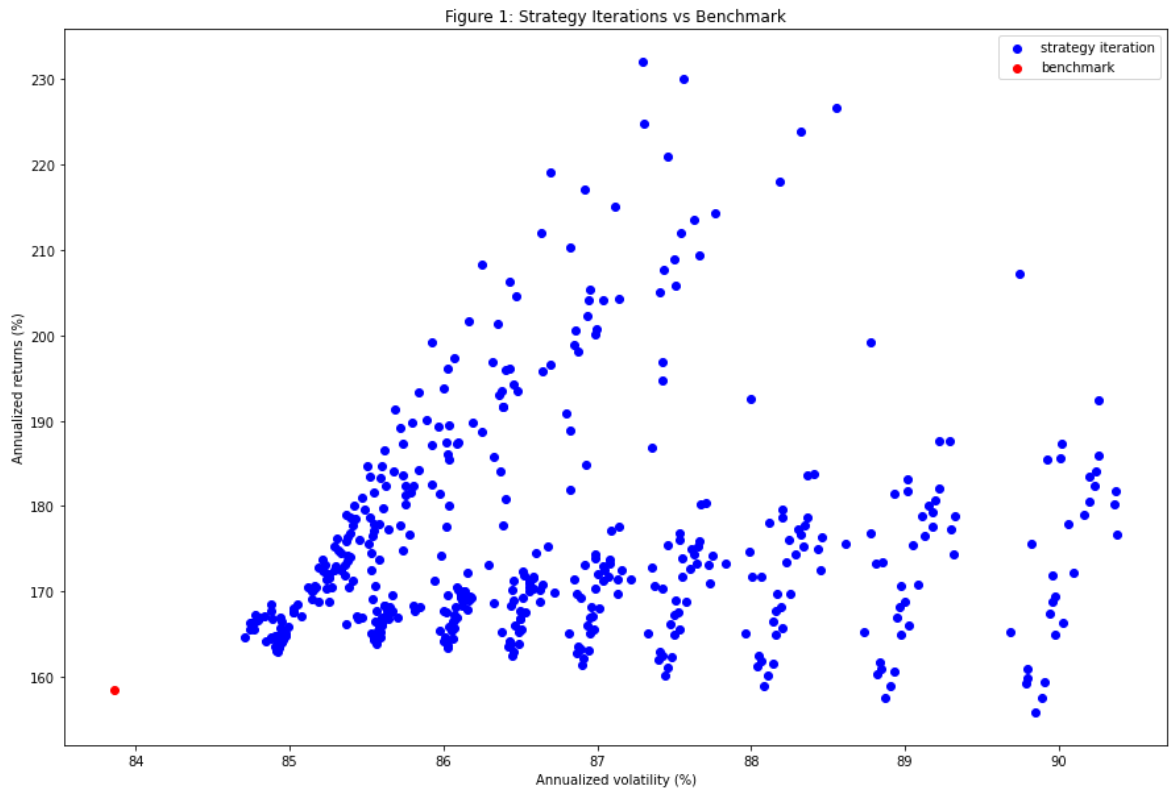
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
btc          eth
```

```
0    2019-06-23  4.584603  162.064048
1    2019-06-24  4.584603  162.064048
2    2019-06-25  4.584603  162.064048
3    2019-06-26  4.584603  162.064048
4    2019-06-27  4.584603  162.064048
..          ...       ...         ...
435  2020-08-31  4.584603  162.064048
436  2020-09-01  4.584603  162.064048
437  2020-09-02  4.584603  162.064048
438  2020-09-03  4.584603  162.064048
439  2020-09-04  4.584603  162.064048

[440 rows x 3 columns],
'portfolioFrame':            date  portfolio_value  perc_return
1    2019-06-24    101195.714255     0.011957
2    2019-06-25    105772.452699     0.045227
3    2019-06-26    115032.790425     0.087550
4    2019-06-27    100295.313490    -0.128115
5    2019-06-28    107261.986130     0.069462
..          ...              ...          ...
435  2020-08-31    123710.140427     0.004086
436  2020-09-01    131702.746354     0.064608
437  2020-09-02    123439.082324    -0.062745
438  2020-09-03    108302.997977    -0.122620
439  2020-09-04    110328.702922     0.018704

[439 rows x 3 columns],
'annualizedReturn': 1.5857667412343828,
'annualizedVolatility': 0.8386328830158948}
```

In [8]:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(15,10))

plt.scatter(summary.annualized_volatility*100,summary.annualized_return*100,co
plt.scatter([benchmark['annualizedVolatility']*100],[benchmark['annualizedRetu
plt.legend()
plt.xlabel('Annualized volatility (%)')
plt.ylabel('Annualized returns (%)')
_=plt.title('Figure 1: Strategy Iterations vs Benchmark')
```



In [9]:

```
benchmarkSharpe=benchmark['annualizedReturn']/benchmark['annualizedVolatility'
benchmarkSharpe
```

Out[9]: 1.8908950189642486

In [10]:

```
summary.sharpe.mean()
```

Out[10]: 2.0224005904959443

In [11]:

```
higherSharpeThanBenchmark=len(summary[summary.sharpe>benchmarkSharpe])/len(sum
higherSharpeThanBenchmark
```

Out[11]: 0.9012345679012346

In [12]:

```
summary.annualized_return.min()
```

Out[12]: 1.5587312040806791

In [13]:

```
summary.annualized_return.max()
```

Out[13]: 2.3200476306914655

In [14]:

```
summary.annualized_return.mean()
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

In [15]:

```
higherReturnThanBenchmark=len(summary[summary.annualized_return>benchmark['ann
higherReturnThanBenchmark
```

Out[15]: 0.9938271604938271

In [16]:

```
summary.annualized_volatility.min()
```

Out[16]: 0.8470918397493876

In [17]:

```
summary.annualized_volatility.max()
```

Out[17]: 0.9037938746286073

In [18]:

```
summary.annualized_volatility.mean()
```

Out[18]: 0.8671095617114396