Dividend Capture Strategy Analysis

```
import pandas as pd
import numpy as np
import os
import datetime
os.getcwd()
```

```
'C:\\Users\\shaneshe\\OneDrive - USC Marshall School of Business\\FBE 551 Fall 2024'
```

Let's read in our dataset and look at its structure. We have company identifiers, dates, and then information about the company and the dividend event. Note that we don't have consecutive dates; we have only isolated dates that correspond to an "event" -- the payment of a dividend.

```
div = pd.read_csv('CRSP_Dividends.csv',parse_dates=[1])
div
```

	PERMNO	date	SHRCD	PERMCO	DISTCD	DIVAMT	PRC	VOL	RET	BID	ASK	SHROUT	OPENPRC	RETX	vwre
0	10001	1986-03-10	11	7953	1232	0.095	-6.2500	100.0	0.015200	NaN	NaN	985.0	NaN	0.000000	0.0043
1	10001	1986-06-09	11	7953	1232	0.105	-6.1875	1050.0	0.016970	NaN	NaN	985.0	NaN	0.000000	-0.0193
2	10001	1986-09-08	11	7953	1232	0.105	6.7500	1610.0	0.054615	6.375	6.75	985.0	NaN	0.038462	-0.0108
3	10001	1986-12-08	11	7953	1232	0.105	6.5000	400.0	0.016154	6.500	7.00	991.0	NaN	0.000000	-0.0009
4	10001	1987-03-09	11	7953	1232	0.105	6.1250	650.0	0.027629	5.875	6.25	991.0	NaN	0.010309	-0.0064
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
458215	93429	2022-05-27	11	53447	1232	0.480	111.8500	491916.0	0.026782	111.620	111.70	106172.0	109.29	0.022395	0.0244
458216	93429	2022-08-30	11	53447	1232	0.480	117.7800	825426.0	-0.017203	117.940	117.99	106189.0	119.83	-0.021192	-0.0114

Step 1: Clean and describe the data

Here I am going to remove all cases where there is zero volume traded. We are trying to see the daily price response to the payment of a dividend, and if there are no trades then we can't expect to get a reliable indicator of the true price, or what price we might have actually been able to trade at. In general it's best to remove circumstances like this, with no liquidity, from your analysis.

I'm also calculating the prior closing price using the RETX (price return without dividend). It's important to use that rather than the RET, which includes the dividend as part of the return and would therefore deliver a faulty prior closing price.

Otherwise, calculating market capitalization and some cleanup as described by the question.

```
div=div[div['VOL']>0]
div=div[div['DIVAMT']>0.01]
div['PRC']=abs(div['PRC'])
div['PriorClose'] = div['PRC']/(1+div['RETX'])
div['mkcap']=div['SHROUT']*div['PRC']
div=div[div['mkcap'] > 50000]
```

Step 2. Examine price change relative to dividend

We can now look at the ratio between the price change and the dividend amount. Our prediction, consistent with the literature, is that the price change should be less than the dividend amount, and this ratio should be less than 1.

Also calculating the stock yield (based upon this single dividend payment), the spread, and the year of the event.

```
div['ratio'] = (div['PriorClose']-div['PRC'])/div['DIVAMT']
div['yield']=div['DIVAMT']/div['PriorClose']
div['spread']=2*(div['ASK']-div['BID'])/(div['ASK']+div['BID'])
```

```
div['spread']=2*(div['ASK']-div['BID'])/(div['ASK']+div['BID'])
div['year']=pd.DatetimeIndex(div['date']).year


stats=div.describe()
stats
```

	PERMNO	date	SHRCD	PERMCO	DISTCD	DIVAMT	PRC	VOL	
count	320489.000000	320489	320489.000000	320489.000000	320489.000000	320489.000000	320489.000000	3.204890e+05	320489.000
mean	47158.684828	1994-12-21 10:32:54.170720128	10.871244	19179.751882	1232.683477	0.254771	37.423163	6.451611e+05	0.002
min	10001.000000	1962-01-02 00:00:00	10.000000	4.000000	1202.000000	0.010800	1.010000	1.000000e+00	-0.729
25%	22939.000000	1982-10-08 00:00:00	11.000000	9452.000000	1232.000000	0.100000	18.750000	5.700000e+03	-0.008
50%	44169.000000	1995-05-22 00:00:00	11.000000	20836.000000	1232.000000	0.190000	28.250000	3.465100e+04	0.002
75%	75257.000000	2008-04-16 00:00:00	11.000000	22426.000000	1232.000000	0.320000	43.000000	2.655000e+05	0.012
max	93429.000000	2023-06-30 00:00:00	11.000000	59543.000000	1294.000000	85.000000	4280.040040	2.561843e+08	0.696
std	26545.149942	NaN	0.334931	12193.527648	5.642725	0.503116	54.863435	3.138198e+06	0.022

8 rows × 21 columns

```
stats['ratio']
```

```
count    320489.000000
mean          0.648548
min        -485.002094
25%          -0.833333
50%           0.736928
75%           2.272679
max         922.868789
std           9.395435
Name: ratio, dtype: float64
```

So the average price drop is only 64% of the amount of the dividend payment -- matches up well to what has been documented in prior work. We see extreme values at the min and the max; may want to windsorize or otherwise handle extreme events like this.

This indiates that there could indeed be a profitable "dividend caputure" strategy. This is an event where the price response across a large number of observations does not match up to that predicted by the efficient markets hypothesis. We could potentially make money by buying at the close, receiving the dividend, and then selling our shares -- likely we are selling our shares at a loss, but the loss will be less than the amount of the dividend recieved.

Step 2 Part 2: Describe returns to buying close-to-close

First we need to add the risk-free rate; we can mostly copy the work from HW#2

```
irx = pd.read_csv('^IRX.csv',  parse_dates=[0])
irx.head()
```

	Date	Open	High	Low	Close	Adj Close	Volume
0	1960-01-04	4.52	4.52	4.52	4.52	4.52	0.0
1	1960-01-05	4.55	4.55	4.55	4.55	4.55	0.0
2	1960-01-06	4.68	4.68	4.68	4.68	4.68	0.0
3	1960-01-07	4.63	4.63	4.63	4.63	4.63	0.0
4	1960-01-08	4.59	4.59	4.59	4.59	4.59	0.0

```
irx = pd.read_csv('^IRX.csv',  parse_dates=[0])
irx['Tbill yield'] = irx['Close']
irx['Tbill yield filled'] = irx['Tbill yield'].fillna(method='ffill')
irx['dow'] = irx['Date'].dt.dayofweek
```

```
irx['lagdow'] = irx['dow'].shift()
irx['numdays'] = 1
irx.loc[irx['dow'] > (irx['lagdow']+1), 'numdays'] = irx['dow'] - irx['lagdow']
irx.loc[irx['dow'] < irx['lagdow'], 'numdays'] = 3 + irx['dow'] + (4 - irx['lagdow'])
irx.loc['2001-09-17','numdays'] = 7
irx['Tbill ret'] = irx['Tbill yield filled'].shift() * irx['numdays'] / 365 / 100
```

```
irx = irx[['Date','Tbill ret']]
irx.rename(columns={"Date": "date"},inplace=True)
```

```
div.sort_values(by='date',inplace=True)
```

Our events aren't sorted by date; nor are they consecutive days. So we can't just port over the column from one data set to the other as we did in homework 2. Here a merge is more appropriate.

```
div = div.merge(irx, how='left', on=['date'])
```

```
div['ex ret'] = div['RET'] - div['Tbill ret']
stats=div.describe()
```

```
print('Annualized return: ',stats.loc['mean','RET']*252)
print('Annualized standard deviation: ', stats.loc['std','RET']*252**0.5)
```

```
Annualized return:  0.6089028407714461
Annualized standard deviation:  0.35818097546428385
```

```
t=stats.loc['mean','RET']*stats.loc['count','RET']**0.5/stats.loc['std','RET']
sharpe = stats.loc['mean','ex ret']/stats.loc['std','RET']
```

```
t
```

```
60.62500381755986
```

```
sharpe*252**0.5
```

```
1.614393066979313
```

The results here are promising. If we were to just buy stocks at the close prior to their ex-div day, and sell at the subsequent close, we earn the RET (which is the price change plus the dividend). That annualized return is quite large at 60.9%. The standard deviation of those returns is also large, annualized at 35.8%. (Note that this is across events, not across days.)

The t-stat on whether this event return is different from zero is 60.6 (!!) and a "Sharpe" ratio across events comes in at 1.61 -- quite promising!

Step 3: Add betas and correct for CAPM

I'll read in the betas, construct a year variable for merging (making sure to match the beta from the prior year to all the events for the subsequent year), and sort by PERMNO and YEAR for the merge. I'll also sort my dividend data.

```
betas = pd.read_csv('Yearly Betas.csv',parse_dates=[1])
```

```
betas=betas[['PERMNO','DATE','b_mkt']]
betas['YEAR'] = pd.DatetimeIndex(betas['DATE']).year
betas['YEAR'] = betas['YEAR'] +1
betas.drop(['DATE'],axis=1,inplace=True)
betas.sort_values(by=['PERMNO','YEAR'],inplace=True)
```

```
div['YEAR']=pd.DatetimeIndex(div['date']).year
div.sort_values(by=['PERMNO','YEAR'],inplace=True)
```

```
betas.head()
```

	PERMNO	b_mkt	YEAR
0	10001	0.0730	1989
1	10001	0.0799	1990
2	10001	0.0986	1991
3	10001	-0.0132	1992
4	10001	-0.0178	1993

```
div = div.merge(betas, how='left', on=['PERMNO','YEAR'])
```

Here I'm going to create a new data set that removes those events where we couldn't match a beta. I don't necessarily want to drop those permanently though.

```
div_betas = div.dropna(subset=['b_mkt'])
```

```
div_betas['capm_ret'] = div_betas['b_mkt']*(div_betas['vwretd'] - div_betas['Tbill ret'])
div_betas['ab_ret'] = div_betas['ex ret'] - div_betas['capm_ret']
stats = div_betas.describe()
```

```
<ipython-input-23-0d1937b50284>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view
  div_betas['capm_ret'] = div_betas['b_mkt']*(div_betas['vwretd'] - div_betas['Tbill ret'])
<ipython-input-23-0d1937b50284>:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view
  div_betas['ab_ret'] = div_betas['ex ret'] - div_betas['capm_ret']
```

```
print('Annualized return: ',stats.loc['mean','ab_ret']*252)
print('Annualized standard deviation: ', stats.loc['std','ab_ret']*252**0.5)
print('Average beta: ', stats.loc['mean','b_mkt'])
```

```
Annualized return:  0.48748035662154665
Annualized standard deviation:  0.32880022740972936
Average beta:  0.9661402008805072
```

```
t=stats.loc['mean','ab_ret']*stats.loc['count','ab_ret']**0.5/stats.loc['std','ab_ret']
t
```

```
50.692454853893935
```

```
sharpe = stats.loc['mean','ab_ret']/stats.loc['std','RET']
sharpe*252**0.5
```

```
1.3701242741206974
```

Results are as we might expect. Returns are lower than before, because some of the return we were capturing wasn't due to the dividend capture event but rather the upward drift of the market over the course of the day, as noted in the literature. We have successfully adjusted for this by removing the CAPM expected return and examining the residual (abnormal return).

Those returns are still 48% per year, much higher than zero, and significantly so with a t-stat of 50. The "Sharpe" ratio is also lower than before, as we would expect.

Step 4: compute a strategy of buying at prior close and selling at the open

This is relatively simple; we buy at the prior close, and receive the open price plus the dividend payment. We will have to calculate this return ourselves however; niether the RET nor RETX is the right number here as those represent close-to-close returns.

Note that here I have gone back to using the full sample, without removing the events that didn't have a beta matched.

```
div['CO_ret'] = (div['OPENPRC'] + div['DIVAMT']) / div['PriorClose'] -1
stats = div.describe()


print('Annualized return: ',stats.loc['mean','CO_ret']*252)
print('Annualized standard deviation: ', stats.loc['std','CO_ret']*252**0.5)
```

```
Annualized return:  0.3948827134300498
Annualized standard deviation:  0.23581282782141816
```

```
t=stats.loc['mean','CO_ret']*stats.loc['count','CO_ret']**0.5/stats.loc['std','CO_ret']
sharpe = stats.loc['mean','CO_ret']/stats.loc['std','CO_ret']
```

```
t
```

```
44.72452608152515
```

```
sharpe*252**0.5
```

```
1.674559934157169
```

Results relatively consistent here; returns a bit lower, and volatility also lower than our CAPM-corrected residuals.

Step 5: account for transaction costs

The Kalay paper proposes that transaction costs are responsible for the lower price drop than predicted by efficient markets. That is, arbitrageurs are not capturing this spread because it is too costly to trade these stocks and the transaction costs will eat up all of the expected return. We can test that by simply subracting off the transaction costs from the full-day returns.

Note that here we are using the spread from the close at the end of the event day. To be precise, we should use the bid/ask spread from both the day prior and the event day (and average the two). This is okay as long as the spread as a percent is stable over time (and evidence supports that).

```
div['tcost']=div['RET']-div['spread']
```

```
div['tcost'].describe()
```

```
count    190722.000000
mean         -0.007891
std           0.028642
min          -1.315794
25%          -0.019263
50%          -0.005080
75%           0.006060
max           0.694063
Name: tcost, dtype: float64
```

This supports the Kalay observation. Once we account for the bid/ask spread, our returns are actually negative! And actually pretty close to zero -- it comes to about -2% annualized. So this makes sense in the end -- prices don't adjust fully, but what looks like a compelling opportunity on paper won't work in reality once we incorporate the necessary transaction costs.

Step 6: rank by a metric

Here we want to use a ranking procedure and see if we can sort the data in a way that might lead to better opportunities. I wrote a function that will take the metric of interest and then do the ranking and output some useful information. This will make it easy to look at a variety of metrics.

The function will output the results grouped by decile for the ranking metric itself; the returns for that decile; the average spread for that decile; and the after-transaction costs return for that decile. We are interested to see if we can find a spread for the last item here.

```
def deciles(metric):
    div['QUINTILE'] = pd.qcut(div[metric],q=10, labels=range(1,11))
    print('Metric: ', div.groupby(['QUINTILE'])[metric].mean() )
    print()
    print('Returns: ', div.groupby(['QUINTILE'])['RET'].mean() )
    print()
    print('Spreads: ', div.groupby(['QUINTILE'])['spread'].mean())
```

```
    print()
    print('After costs: ', div.groupby(['QUINTILE'])['tcost'].mean() )
```

One idea might be market cap -- small stocks might be a less efficient space and allow for greater returns to persist. Let's check!

```
deciles('mkcap')
```

```
Metric:  QUINTILE
1     6.564689e+04
2     1.062130e+05
3     1.683883e+05
4     2.645698e+05
5     4.205461e+05
6     6.836939e+05
7     1.153327e+06
8     2.174753e+06
9     5.052452e+06
10    4.234471e+07
Name: mkcap, dtype: float64

Returns:  QUINTILE
1     0.003733
2     0.003367
3     0.003422
4     0.002753
5     0.002436
6     0.002327
7     0.001814
8     0.001653
9     0.001530
10    0.001129
Name: RET, dtype: float64

Spreads:  QUINTILE
1     0.031958
2     0.025189
3     0.018629
4     0.014272
5     0.010890
6     0.007928
7     0.006186
8     0.004445
9     0.003270
10    0.001965
Name: spread, dtype: float64

After costs:  QUINTILE
1    -0.026479
2    -0.020768
3    -0.014359
4    -0.010995
5    -0.007790
6    -0.005106
7    -0.004306
8    -0.002877
9    -0.001776
10   -0.000822
Name: tcost, dtype: float64
```

Interesting -- small cap stocks (starting in decile 1) show returns to this event three times as large as the largest cap stocks! However, it is also more costly to trade small cap stocks. The average spread is about 15 times as large! So when we look at the after cost returns, the smallest decile actually shows the worst returns, and the largest shows the best. Returns are negative across all deciles, however.

```
deciles('b_mkt')
```

```
Metric:  QUINTILE
1     0.158589
2     0.439477
3     0.610723
4     0.751539
5     0.876915
6     0.998009
7     1.124582
8     1.272629
9     1.472352
10    1.957031
Name: b_mkt, dtype: float64

Returns:  QUINTILE
```

```
1     0.002890
2     0.002042
3     0.002143
4     0.002347
5     0.002248
6     0.002526
7     0.002424
8     0.002317
9     0.002474
10    0.002268
Name: RET, dtype: float64

Spreads:  QUINTILE
1     0.014030
2     0.012590
3     0.011202
4     0.010428
5     0.009441
6     0.009188
7     0.008458
8     0.007269
9     0.006758
10    0.005703
Name: spread, dtype: float64

After costs:  QUINTILE
1    -0.010706
2    -0.009542
3    -0.008500
4    -0.007742
5    -0.006979
6    -0.006615
7    -0.006282
8    -0.005276
9    -0.004772
10   -0.003893
Name: tcost, dtype: float64
```

Not too much going on with beta; the highest beta stocks are the cheapest to trade, and therefore have the best after-cost returns, but still negative.

```
deciles('yield')
```

```
Metric:  QUINTILE
1     0.001407
2     0.002715
3     0.003783
4     0.004835
5     0.005923
6     0.007093
7     0.008407
8     0.010098
9     0.012698
10    0.023369
Name: yield, dtype: float64

Returns:  QUINTILE
1     0.000844
2     0.001716
3     0.002073
4     0.002406
5     0.002831
6     0.003105
7     0.003281
8     0.003327
9     0.003275
10    0.001304
Name: RET, dtype: float64

Spreads:  QUINTILE
1     0.008163
2     0.009207
3     0.009742
4     0.010481
5     0.010898
6     0.011228
7     0.011602
8     0.012281
9     0.012829
10    0.011811
Name: spread, dtype: float64
```

```
After costs:  QUINTILE
1    -0.007424
2    -0.007569
3    -0.007721
4    -0.007973
5    -0.008021
6    -0.007850
7    -0.007974
8    -0.008244
9    -0.008733
10   -0.008073
Name: tcost, dtype: float64
```

The Kalay paper noted some differences in the price change/dividend ratio across yields, and we see that as well. However the higher yielding stocks also have higher spreads. So, everything still negative.

It seems that transaction costs are the real culprit; higher returns (on paper) seem to come along with higher transaction costs. What if we actually sorted on the transaction cost variable itself? Note that I am cheating a little bit here; the spread is forward-looking information and not known prior to the event day. I'm relying on the spread being stable through time (if I sorted on yesterday's spread, I would hope to get very similar results.) But this is an assumption and we would definitely want to check that before proceeding much further.

```
deciles('spread')
```

```
Metric:  QUINTILE
1     0.000099
2     0.000321
3     0.000584
4     0.001112
5     0.002487
6     0.005487
7     0.009448
8     0.014578
9     0.022979
10    0.048265
Name: spread, dtype: float64

Returns:  QUINTILE
1     0.001388
2     0.001212
3     0.001471
4     0.001574
5     0.002158
6     0.002578
7     0.003178
8     0.003676
9     0.004187
10    0.004985
Name: RET, dtype: float64

Spreads:  QUINTILE
1     0.000099
2     0.000321
3     0.000584
4     0.001112
5     0.002487
6     0.005487
7     0.009448
8     0.014578
9     0.022979
10    0.048265
Name: spread, dtype: float64

After costs:  QUINTILE
1     0.001289
2     0.000890
3     0.000888
4     0.000462
5    -0.000328
6    -0.002909
7    -0.006270
8    -0.010902
9    -0.018792
10   -0.043280
Name: tcost, dtype: float64
```

So this is interesting. Spreads rise across deciles (by construction; we've sorted on this). Returns also rise across deciles; The stocks that are most costly to trade also have the highest returns. (This is not quite the same thing as saying that the stocks that have the highest returns also have the highest cost to trade.) As it turns out, the spreads rise much faster than the returns. So perhaps we have a possibility: by focusing on the stocks that are the most liquid and have the lowest spreads, we may uncover a profiatable opportunity.

Let's pull out that first decile with the lowest spreads and examine it in more detail.

```
div_liquid = div[div['QUINTILE'] == 1]
div_liquid['ex ret tcost'] = div_liquid['tcost'] - div_liquid['Tbill ret']
stats = div_liquid.describe()
```

```
<ipython-input-53-cadb99a882d4>:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view
  div_liquid['ex ret tcost'] = div_liquid['tcost'] - div_liquid['Tbill ret']
```

```
print('Annualized return: ',stats.loc['mean','tcost']*252)
print('Annualized standard deviation: ', stats.loc['std','tcost']*252**0.5)
```

```
Annualized return:  0.32482246897327344
Annualized standard deviation:  0.28999301922038345
```

```
t=stats.loc['mean','tcost']*stats.loc['count','tcost']**0.5/stats.loc['std','tcost']
sharpe = stats.loc['mean','ex ret tcost']/stats.loc['std','ex ret tcost']*252**0.5
```

```
t
```

```
9.744683845211572
```

```
sharpe
```

```
1.0916538901340958
```

```
div_liquid['mkcap'].mean()
```

```
35000631.764542826
```

Results have definitely deteriorated, but are still positive (32% annualized) with a strong t-stat and compelling Sharpe ratio.

What would next steps be? First verify with prior-day spreads to make this an implementable strategy. Then construct a calendar time return series where you average your exposure across all the available dividend payers on a given day. (How do you handle days for which you don't have a dividend event?) Then control for risk, look for out-of-sample evidence, see if spreads/returns have changed over time.