# 05_kelly_rule

September 29, 2021

## 1 How to size your bets - The Kelly Rule

he Kelly rule has a long history in gambling because it provides guidance on how much to stake on each of an (infinite) sequence of bets with varying (but favorable) odds to maximize terminal wealth. It was published as A New Interpretation of the Information Rate in 1956 by John Kelly who was a colleague of Claude Shannon's at Bell Labs. He was intrigued by bets placed on candidates at the new quiz show The $64,000 Question, where a viewer on the west coast used the three-hour delay to obtain insider information about the winners.

Kelly drew a connection to Shannon's information theory to solve for the bet that is optimal for long-term capital growth when the odds are favorable, but uncertainty remains. His rule maximizes logarithmic wealth as a function of the odds of success of each game, and includes implicit bankruptcy protection since log(0) is negative infinity so that a Kelly gambler would naturally avoid losing everything.

### 1.1 Imports

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
%matplotlib inline
from pathlib import Path

import numpy as np
from numpy.linalg import inv
from numpy.random import dirichlet
import pandas as pd

from sympy import symbols, solve, log, diff
from scipy.optimize import minimize_scalar, newton, minimize
from scipy.integrate import quad
from scipy.stats import norm

import matplotlib.pyplot as plt
import seaborn as sns
```

[3]:
```
sns.set_style('whitegrid')
np.random.seed(42)
```

[4]:
```
DATA_STORE = Path('..', 'data', 'assets.h5')
```

### 1.2 The optimal size of a bet

Kelly began by analyzing games with a binary win-lose outcome. The key variables are: - b: The odds define the amount won for a \$1 bet. Odds = 5/1 implies a \$5 gain if the bet wins, plus recovery of the \$1 capital. - p: The probability defines the likelihood of a favorable outcome. - f: The share of the current capital to bet. - V: The value of the capital as a result of betting.

The Kelly rule aims to maximize the value's growth rate, G, of infinitely-repeated bets (see Chapter 5 for background).

$$G = \lim_{N \to \infty} = \frac{1}{N} \log \frac{V_N}{V_0}$$

We can maximize the rate of growth G by maximizing G with respect to f, as illustrated using sympy as follows:

[5]:
```
share, odds, probability = symbols('share odds probability')
Value = probability * log(1 + odds * share) + (1 - probability) * log(1 - share)
solve(diff(Value, share), share)
```

[5]:
```
[(odds*probability + probability - 1)/odds]
```

[6]:
```
f, p = symbols('f p')
y = p * log(1 + f) + (1 - p) * log(1 - f)
solve(diff(y, f), f)
```

[6]:
```
[2*p - 1]
```

### 1.3 Get S&P 500 Data

[7]:
```
with pd.HDFStore(DATA_STORE) as store:
    sp500 = store['sp500/stooq'].close
```

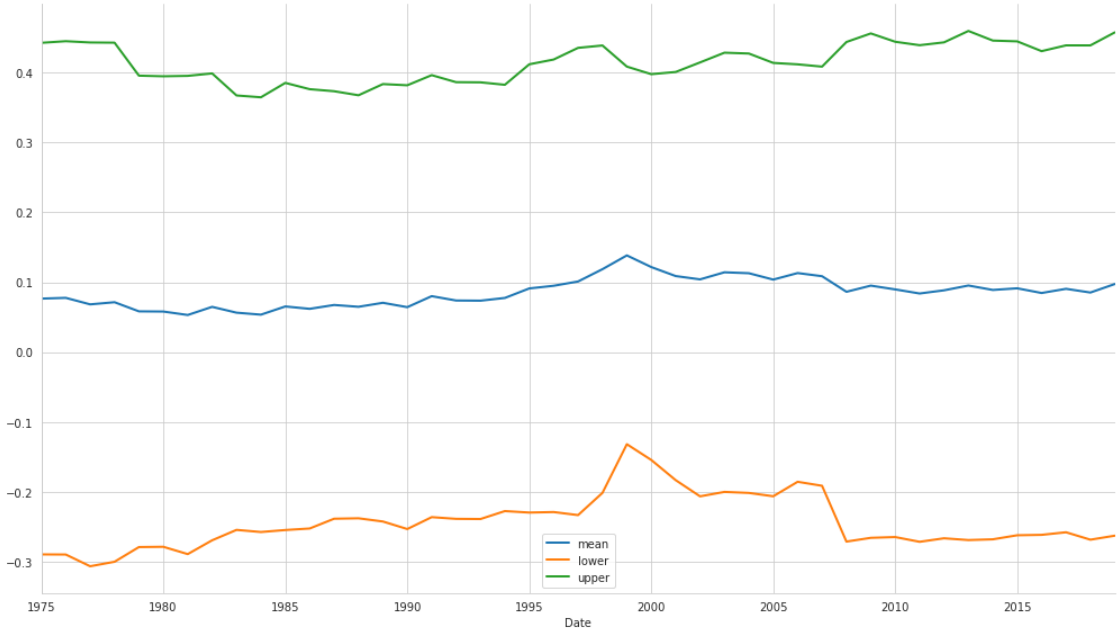
#### 1.3.1 Compute Returns & Standard Deviation

[8]:
```
annual_returns = sp500.resample('A').last().pct_change().dropna().
↪to_frame('sp500')
```

[9]:
```
return_params = annual_returns.sp500.rolling(25).agg(['mean', 'std']).dropna()
```

[10]:
```
return_ci = (return_params[['mean']]
                .assign(lower=return_params['mean'].sub(return_params['std'].
↪mul(2)))
                .assign(upper=return_params['mean'].add(return_params['std'].
↪mul(2))))
```

[11]:
```
return_ci.plot(lw=2, figsize=(14, 8))
plt.tight_layout()
sns.despine();
```




### 1.3.2 Kelly Rule for a Single Asset - Index Returns

In a financial market context, both outcomes and alternatives are more complex, but the Kelly rule logic does still apply. It was made popular by Ed Thorp, who first applied it profitably to gambling (described in Beat the Dealer) and later started the successful hedge fund Princeton/Newport Partners.

With continuous outcomes, the growth rate of capital is defined by an integrate over the probability distribution of the different returns that can be optimized numerically. We can solve this expression (see book) for the optimal f* using the `scipy.optimize` module:

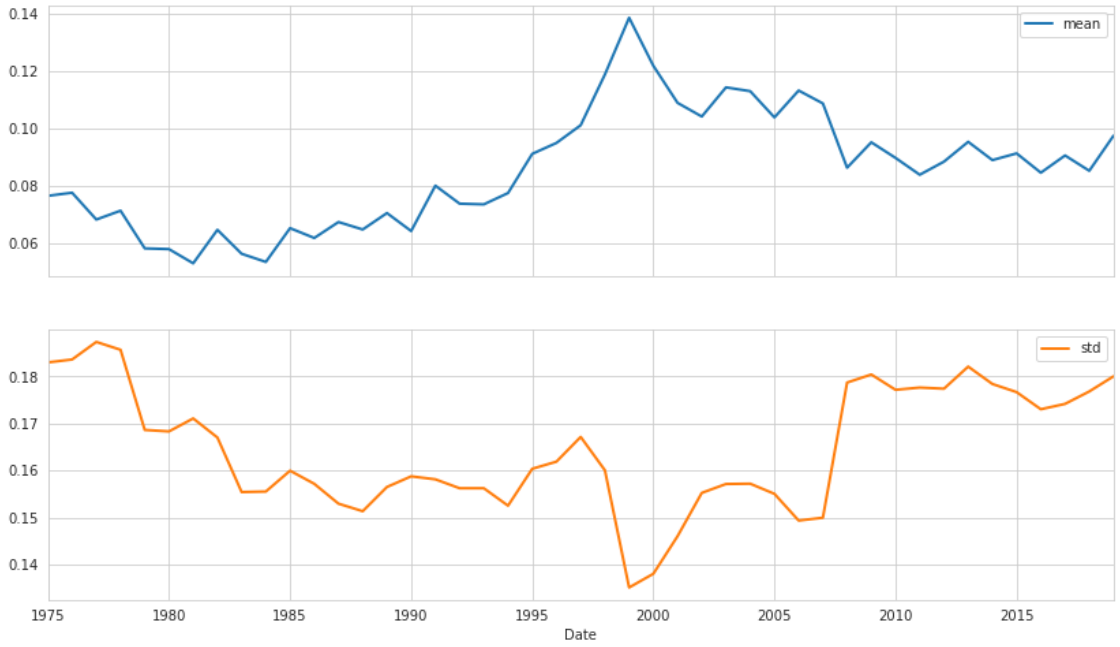
[12]:
```
def norm_integral(f, mean, std):
    val, er = quad(lambda s: np.log(1 + f * s) * norm.pdf(s, mean, std),
                             mean - 3 * std,
                             mean + 3 * std)
    return -val
```

[13]:
```
def norm_dev_integral(f, mean, std):
    val, er = quad(lambda s: (s / (1 + f * s)) * norm.pdf(s, mean, std),␣
 ↪m-3*std, mean+3*std)
    return val
```

[14]:
```
def get_kelly_share(data):
    solution = minimize_scalar(norm_integral,
                               args=(data['mean'], data['std']),
                               bounds=[0, 2],
                               method='bounded')
    return solution.x
```

[15]:
```
annual_returns['f'] = return_params.apply(get_kelly_share, axis=1)
```

[16]:
```
return_params.plot(subplots=True, lw=2, figsize=(14, 8));
```



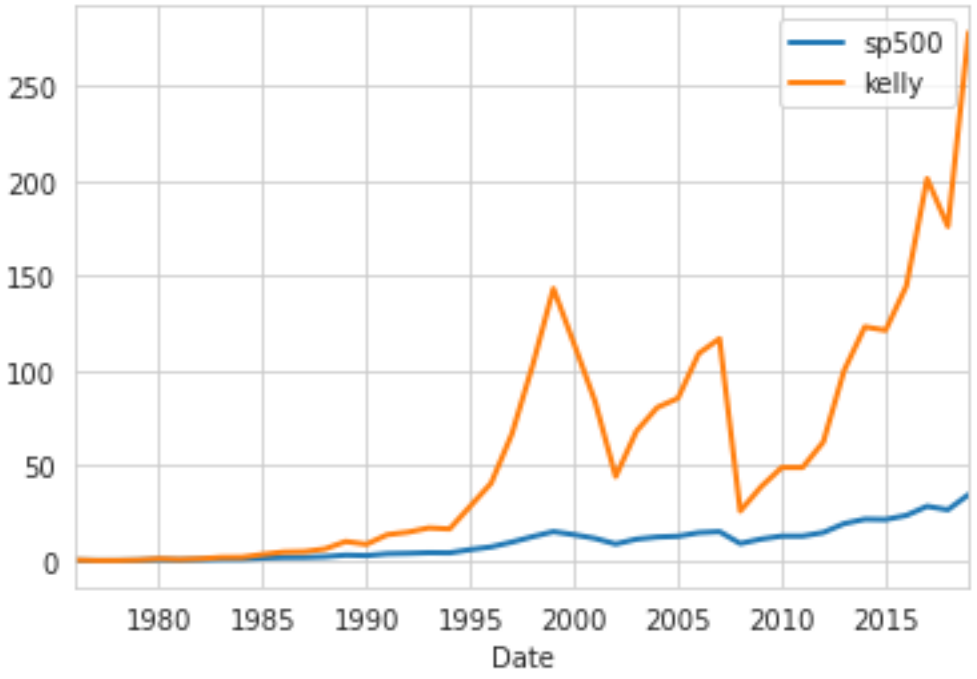

[17]:
```
annual_returns.tail()
```

[17]:

| Date | sp500 | f |
|---|---|---|
| 2015-12-31 | -0.007266 | 1.999996 |
| 2016-12-31 | 0.095350 | 1.999996 |
| 2017-12-31 | 0.194200 | 1.999996 |
| 2018-12-31 | -0.062373 | 1.999996 |
| 2019-12-31 | 0.288781 | 1.999996 |

### 1.3.3 Performance Evaluation

[18]:
```
(annual_returns[['sp500']]
 .assign(kelly=annual_returns.sp500.mul(annual_returns.f.shift()))
 .dropna()
 .loc['1900':]
 .add(1)
 .cumprod()
 .sub(1)
 .plot(lw=2));
```




[19]:
```
annual_returns.f.describe()
```

[19]:
```
count    45.000000
mean      1.979025
std       0.062282
min       1.708206
25%       1.999996
50%       1.999996
75%       1.999996
max       1.999996
Name: f, dtype: float64
```

[20]:
```
return_ci.head()
```

[20]:

| Date | mean | lower | upper |
|---|---|---|---|
| 1975-12-31 | 0.076574 | -0.289442 | 0.442591 |
| 1976-12-31 | 0.077649 | -0.289600 | 0.444897 |
| 1977-12-31 | 0.068336 | -0.306402 | 0.443074 |
| 1978-12-31 | 0.071410 | -0.299973 | 0.442794 |
| 1979-12-31 | 0.058325 | -0.278930 | 0.395581 |

### 1.3.4 Compute Kelly Fraction

[21]:
```
m = .058
s = .216
```

[22]:
```
# Option 1: minimize the expectation integral
sol = minimize_scalar(norm_integral, args=(m, s), bounds=[0., 2.],␣
 ↪method='bounded')
print('Optimal Kelly fraction: {:.4f}'.format(sol.x))
```

```
Optimal Kelly fraction: 1.1974
```

[23]:
```
# Option 2: take the derivative of the expectation and make it null
x0 = newton(norm_dev_integral, .1, args=(m, s))
print('Optimal Kelly fraction: {:.4f}'.format(x0))
```

```
Optimal Kelly fraction: 1.1974
```

## 1.4 Kelly Rule for Multiple Assets

We will use an example with various equities. E. Chan (2008) illustrates how to arrive at a multi-asset application of the Kelly Rule, and that the result is equivalent to the (potentially levered) maximum Sharpe ratio portfolio from the mean-variance optimization.

The computation involves the dot product of the precision matrix, which is the inverse of the covariance matrix, and the return matrix:

[24]:
```
with pd.HDFStore(DATA_STORE) as store:
    sp500_stocks = store['sp500/stocks'].index
    prices = store['quandl/wiki/prices'].adj_close.unstack('ticker').
 ↪filter(sp500_stocks)
```

[25]:
```
prices.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 14277 entries, 1962-01-02 to 2018-03-27
Columns: 482 entries, MMM to ZTS
dtypes: float64(482)
memory usage: 52.6 MB
```

[26]:
```
monthly_returns = prices.loc['1988':'2017'].resample('M').last().pct_change().
 ↪dropna(how='all').dropna(axis=1)
stocks = monthly_returns.columns
monthly_returns.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 359 entries, 1988-02-29 to 2017-12-31
Freq: M
Columns: 207 entries, MMM to XRX
dtypes: float64(207)
memory usage: 583.4 KB
```

### 1.4.1 Compute Precision Matrix

[27]:
```
cov = monthly_returns.cov()
precision_matrix = pd.DataFrame(inv(cov), index=stocks, columns=stocks)
```

[28]:
```
kelly_allocation = monthly_returns.mean().dot(precision_matrix)
```

[29]:
```
kelly_allocation.describe()
```

[29]:
```
count    207.000000
mean       0.237375
std        3.470213
min      -11.586454
25%       -1.750238
50%        0.093455
75%        2.505289
max        9.255888
dtype: float64
```

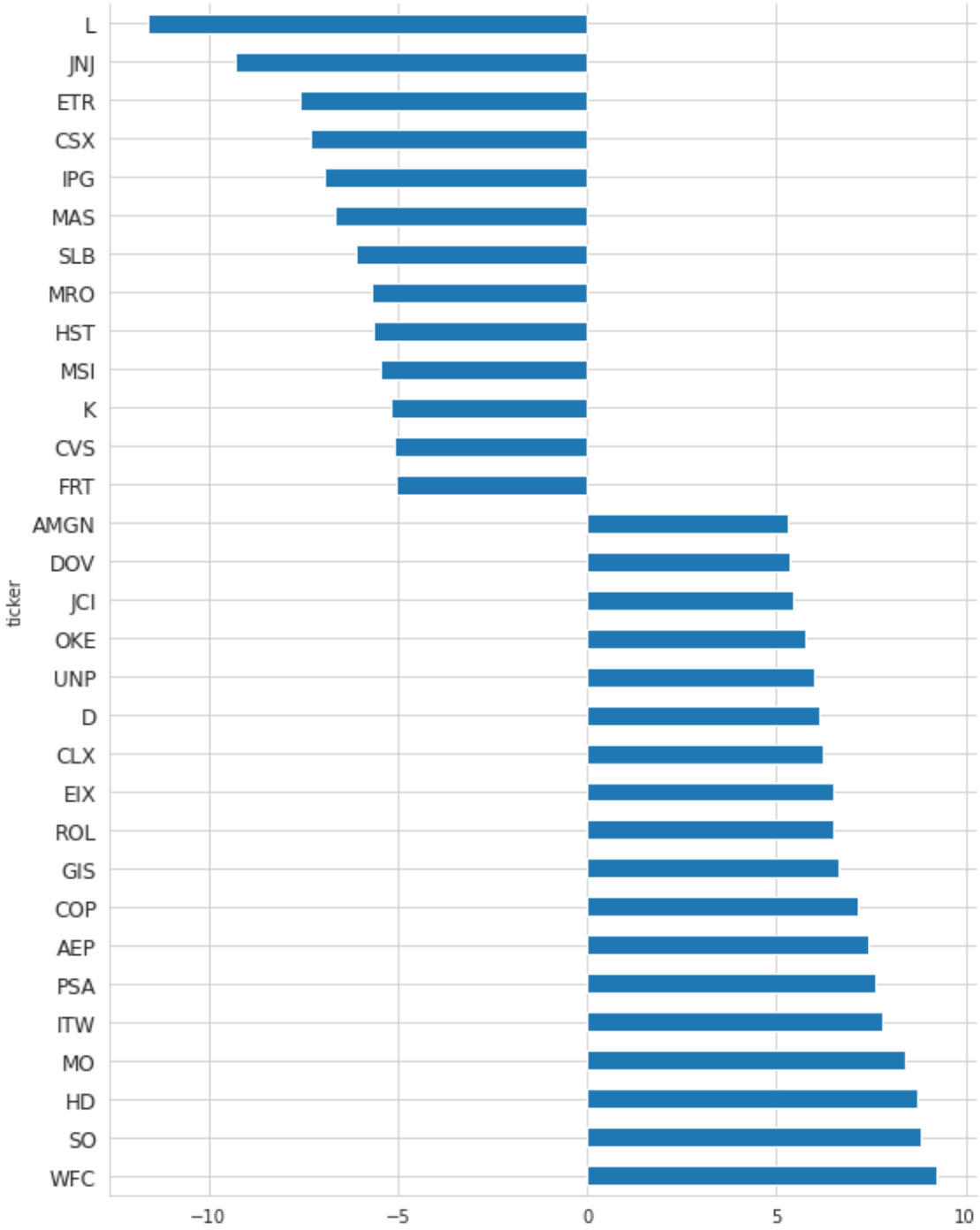
[30]:
```
kelly_allocation.sum()
```

[30]:
```
49.13669441959205
```

### 1.4.2 Largest Portfolio Allocation

The plot shows the tickers that receive an allocation weight $>$ 5x their value:

[31]:
```
kelly_allocation[kelly_allocation.abs()>5].sort_values(ascending=False).plot.
 ↪barh(figsize=(8, 10))
plt.yticks(fontsize=12)
sns.despine()
plt.tight_layout();
```




### 1.4.3 Performance vs SP500

The Kelly rule does really well. But it has also been computed from historical data..

[32]:
```
ax = monthly_returns.loc['2010':].mul(kelly_allocation.div(kelly_allocation.
 ↪sum())).sum(1).to_frame('Kelly').add(1).cumprod().sub(1).
 ↪plot(figsize=(14,4));
sp500.filter(monthly_returns.loc['2010':].index).pct_change().add(1).cumprod().
 ↪sub(1).to_frame('SP500').plot(ax=ax, legend=True)
plt.tight_layout()
sns.despine();
```