Bankroll Management with the Kelly Criterion

pushpendre

April 2021

1 Introduction

In this note I give a self-contained explanation of the Kelly method for bankroll management in gambling. My goal is to use this method for allocating assets to my option purchases and stock purchases and I'll give worked out examples for doing so. I'll also account for transaction costs in my examples, since options are not free, but interestingly stock purchases are free now.

NOTE 1: In the first half of this note I'll focus on only bets where we can't lose more than we put in! For example, when we buy an option. In contrast, when we write an option then we can lose more than we put in. I'll derive the formulaes separately for those situation in a later half. The reason for tackling these things separately is that dealing with both of these situations in a single shot is very confusing, and writing options is more complicated for a retail investor like me.

NOTE 2: A second thing to remember is that we can't use the kelly criterion for portfolio allocation amongst stocks. Stocks dont behave like bets. When I take ownership of stock the time horizon of ownership is not fixed.

NOTE 3: I'll start with the belief that maximizing the Expected value for the log of the wealth is desirable. I wont justfiy this. Intuitively – for me – the log of the wealth saturates quickly, so this method curbs our greed, and it stops our mathemtical methods from becoming too unrealistic. Other mathematical justfication is also available in the literature on Kelly betting. See `https://en.wikipedia.org/wiki/Kelly_criterion`

2 The (Basic) Method

Consider the following situation:

a	Total dollar value of assets.
c	Total dollar value of available cash that can be invested.
β	The relative size of the bet. We put βc on the bet.
t	The transaction cost for entering the bet. We call something a transaction cost when the returns are not related to this number at all. Total cost of entering bet $= t + \beta c$. In all the following analysis I assume that $t << \beta c$. This way total cost of entering bet $= \beta c$.
p	The probability of the bet being favorable.
r_0	The first order coefficient of return when we lose bet. Return $= (\beta c)(r_0 - 1)$. $r_0 > 0$ under our assumption in Note 1 above.
r_1	The first order coefficient of return when we win bet. Return $= (\beta c)(r_1 - 1)$ $r_1 > 1$ is sensibile, but it is possible that $r_1 >> 1$. In double or nothing $r_1 = 2, r_0 = 1$

For example, if we have total assets of $123K$ USD but only $23K$ can be invested, and we have identified a bet where we plan on putting $10K$ in, and if we win then we get back our initial investment and an extra $14K$ for a total of $24K$ but if we lose then we lose all of our investment, and the chance of success if 80%, and we have to pay the broker a 100 USD for entering the bet then the variables are:

$a = 123K, c = 23K, \beta = 10/23, t = 0.1K, p = 0.8, r_1 = (10 + 24)/10 = 3.4, r_0 = 0.0$

We have three possibilities.

1. We dont enter the bet. Our wealth after the bet is c.

2. We enter the bet and we win. Our wealth after bet is $c(1 + \beta(r_1))$.

3. We enter the bet and we lose. Our wealth after bet is $c(1 + \beta(r_0))$

The expected value of log wealth after entering the bet is

$$\log(c) + p \log(1 + \beta(r_1 - 1)) + (1 - p) \log(1 + \beta(r_0 - 1)) \tag{1}$$

.

The maxima occurs in the interval $\beta \in [0, 1]$ at

$$\beta = \frac{\alpha - 1}{r'_1 - \alpha r'_0} \text{where } \alpha = \frac{-(r'_1)p}{(r'_0)(1 - p)} \text{where } r'_1 = r_1 - 1 \text{ and } r'_0 = r_0 - 1 \tag{2}$$

For instance, in the example above the value of $\alpha = 9.6$ and optimal value of beta, i.e. β^* equals 0.716. In this case the expected value of log-wealth is $0.234 + \log(c)$, since this is greater than $\log(c)$ we should enter the bet.

So the kelly criterion is asking us to put in 71% of our cash in this bet.

3 The (generalized) method

Let's continue to consider the problem of BUYING call/put options. But generalize the above problem in two ways:

1. Multiple bet sizing

2. Considering a range of outcomes for a single bet.

3.1 Multiple bet sizing

Consider the situation where we want to size multiple bets. We can simply add more variables to our model and use constrained numerical multi-variate optimization methods instead of deriving closed form formulae.

Say we have B bets. The expected value of log wealth is

$$\log(c) + \sum_{i=1}^{B} p_i \log(1 + \beta_i(r_{1i} - 1) + (1 - p_i) \log(1 + \beta_i(r_{0i} - 1)) \tag{3}$$

We want to maximize this function on the set $\beta_i \in [0, 1], \sum_i \beta_i < 1$. This is a concave function maximized on a convex set so it has a unique optima, and it should be easy to solve. So let's consider three bets.

1. $\beta_1 =?, p_1 = 0.80, r_{11} = (10 + 24)/10 = 3.4, r_{01} = 0.0$

2. $\beta_2 =?, p_2 = 0.25, r_{12} = (10 + 100)/10 = 11, r_{02} = 0.0$

3. $\beta_3 =?, p_2 = 0.50, r_{12} = (10 + 35)/10 = 4.5, r_{02} = 0.0$

The main difference is in the risk-reward profile between the three bets. Using the code below we can see that Kelly wants us to split our bankroll in the ratio $[61.8\%, 12.5\%, 25.6\%]$ between these three bets. Interestingly if we optimize the bankrolls individually we get $[0.7], [0.17], [0.35]$ and normalizing these individual bets will give us $[0.57, 0.14, 0.29]$, so "Multi-kelli" is giving higher weightage to bet number 1 which has a higher chance of success!

```
from scipy.optimize import (
    minimize, Bounds, LinearConstraint)
import jax.numpy as jnp
from jax import grad, jit, vmap

def f(x):
```

```python
    x = jnp.array(x)
    rt = [1.0, 1.00, 1.0]
    r1 = [3.4, 11.0, 4.5]
    r0 = [0.0, 0.00, 0.0]
    p =  [0.8, 0.25, 0.5]
    v = 0
    for i in range(3):
        v += (p[i] * jnp.log(1 + x[i] * (r1[i] - rt[i]))
              + (1 - p[i]) * jnp.log(1 + x[i] * (r0[i] - rt[i])))
    # Numerical optimization libraries are written to minimize
    # a function, so return (-v), so that minimizing this
    # is equal to maximizing (v)
    return -v

g = grad(f, argnums=0)
minimize(
    f,
    x0=jnp.array([0.1, 0.1, 0.1]),
    jac=g,
    constraints=[LinearConstraint(jnp.array([[1, 1, 1]]),
                                  jnp.array([0.0]),
                                  jnp.array([1.0]))],
    bounds=Bounds(0, 1)
)
"""
fun: -0.8100225925445557
     jac: array([-0.24881762, -0.2509017 , -0.2507056 ])
 message: 'Optimization terminated successfully.'
    nfev: 8
     nit: 6
    njev: 6
  status: 0
 success: True
       x: array([0.61838606, 0.12551913, 0.25609481])
"""
```

3.2 Single bet with multiple potential outcomes

Unfortunately real-life bets – specially options – are not binary. There is a continuum of outcomes; for example, we may have 50% probability of losing everything and a 45% chance of doubling our money, and a 5% chance of tripling it. How should we handle this situation? Let S be the possible scenarios then we can simply plug the scenarios into our objective function as follows

$$\log(c) + \sum_{i=1}^{B} \sum_{j=1}^{S} p_{ij} \log(1 + \beta_i (r_{ji} - 1)) \tag{4}$$

For example, consider that we have a more sophisticated model of our bets, shown in the table below, where each tuple shows a probability and its associated return:

Bet 1	(0.4, 0.0)	(0.5, 1)	(0.1, 2)
Bet 2	(0.3, 0.0)	(0.4, 2)	(0.3, 3)
Bet 3	(0.6, 0.0)	(0.1, 2)	(0.3, 3)

For instance, the right-most cell of the last row says that the probability of tripling my money in bet 3 is 0.4. And Bet 1 has a 40% chance of losing everything, and a 50% chance of breaking even, and a 10% chance of doubling our money. Kelly says to break our portfolio investments as follows

```python
from scipy.optimize import (
    minimize, Bounds, LinearConstraint)
import jax.numpy as jnp
from jax import grad, jit, vmap

def f(x):
    x = jnp.array(x)
    M = [[(0.4, 0.0) , (0.5, 1) , (0.1, 2)],
         [(0.3, 0.0) , (0.4, 2) , (0.3, 3)],
         [(0.6, 0.0) , (0.1, 2) , (0.3, 3)]]
    B = S = 3
    v = 0
    for i in [0, 1, 2]: #, 1, 2]:
        for j in [0, 1, 2]:
            v += M[i][j][0] * jnp.log1p(x[i] * (M[i][j][1] - 1))
    # Numerical optimization libraries are written to minimize
    # a function, so return (-v), so that minimizing this
    # is equal to maximizing (v)
    return -v

g = grad(f, argnums=0)
minimize(
    f,
    x0=jnp.array([0.1, 0.1, 0.1]),
    jac=g,
    constraints=[LinearConstraint(jnp.array([[1, 1, 1]]),
                                  jnp.array([0.0]),
                                  jnp.array([1.0]))],
    bounds=Bounds(0, 1)
)
"""
fun: -0.16524207592010498
     jac: array([ 0.30000001,  0.00032932, -0.00139785])
 message: 'Optimization terminated successfully.'
    nfev: 6
     nit: 5
    njev: 5
  status: 0
 success: True
       x: array([0.        , 0.47983115, 0.05568557])
"""
```