Entropy Pooling

1.0 Introduction

Entropy pooling is a technique used in portfolio optimization to seamlessly blend prior probabilities with new information. The core idea is to use relative entropy, also known as Kullback-Leibler divergence, to measure the difference between two probability distributions. By doing so, we can adjust the prior distribution just enough to incorporate new views or information, while still staying as close as possible to the original distribution. This method ensures that the updated (posterior) distribution respects the new data but avoids making drastic changes based on the new information alone, thus maintaining a balance between historical data and current insights.

In practical terms, entropy pooling sets up an optimization problem where the goal is to minimize the divergence between the prior and posterior distributions, subject to the constraints imposed by the new views. These views are typically expressed as linear constraints on expected returns or other relevant metrics. The result is a posterior distribution that accurately reflects the new views without straying too far from the original beliefs. This approach is particularly valuable in financial contexts, where it allows for a measured and consistent incorporation of new information, enhancing the stability and performance of investment strategies while managing risks effectively.

2.0 The Problem Definition

1. Consider a probability space $(\boldsymbol{\Omega}, \boldsymbol{F}, \boldsymbol{P})$ where $\boldsymbol{\Omega}$ is the set of all possible outcomes, $\boldsymbol{F}$ is the summation of events, and $\boldsymbol{P}$ is the probability measure. Suppose we have **n** scenarios for asset returns with prior probabilities given by a vector $\boldsymbol{P}$, where:
$$\sum_{i=1}^{n} p_i = 1 \ and \ p_i \geq 0 \ \forall \ i \in N$$

2. We have a prior distribution of asset returns $\boldsymbol{P} = (\boldsymbol{p_1}.\boldsymbol{p_2}, \ldots., \boldsymbol{p_n})$ where $\boldsymbol{p_i}$ is the probability associated with the i^{th} scenario.

3. We incorporate new information or views into the model as linear constraints on the expected returns. For instance, if we have $\boldsymbol{m}$ views, they can be represented as:
$$\boldsymbol{q \ = \ argmin \, \{x^T \ (ln \, x - ln \, p)\}}$$
subject to constraints: $Ax \ = \ b$ **or** $Gx \ <= \ h$
depending on the whether we have our views on equality or inequality.
Where:
   - $A, G \in R^{m \times n}$, which represents the linear constraints
   - q represents the posterior probability distribution
   - $b, h \in R^{m \times 1}$, which is a matrix representing the values of the views

4. The entropy function calculates the Kullback-Leibler divergence between the prior and posterior probabilities as shown below. This function calculates the Kullback-Leibler

divergence between the prior and posterior probabilities. This measures the "distance" between the two probability distributions.

$$relative\ entropy\ (p,q) = q^T(\log q - \log p)$$

5. The optimization problem is to find the posterior distribution **q** that minimizes the relative entropy (Kullback-Leibler divergence) with respect to the prior distribution P:

$$q = argmin_x \{x^T(\log x - \log p)\} \qquad (\mathbf{1})$$

2.1 Lagrangian Formulation of the Optimization Problem

With lagrangian we can solve a problem numerically subject to some constraints, when we need to minimize or maximize something.

The Lagrangian is given as follows:

$$\boldsymbol{L}(\boldsymbol{x},\boldsymbol{\lambda},\boldsymbol{v}) = x^T\ (\ln q + \ln p) + \boldsymbol{\lambda}^{\boldsymbol{T}}(\boldsymbol{Gx}-\boldsymbol{h}) + \boldsymbol{v}^{\boldsymbol{T}}(\boldsymbol{Ax}-\boldsymbol{b}) \qquad (\mathbf{2})$$

Where $\boldsymbol{\lambda}$ is the vector of Lagrange multipliers for the views and v is the Lagrange multiplier for the normalization constraint.

2.2 First Order Conditions

To find the optimal $\boldsymbol{q}$, we find the first order differential of $L\ wrt\ x$:

$$\frac{dL}{dx} = \ln x\ - \ln p + 1 + \ G^T\lambda + \ A^T v$$

Equating $\frac{dL}{dx}$ = 0, we get an equation in terms of x

$$\ln x = \ln p - \mathbf{1} - \ G^T\lambda - \ A^T v$$

$$x(\lambda,v) = \ e^{\ln p - \mathbf{1} - G^T\lambda - A^T v} \qquad (\mathbf{3})$$

where **1** is the s dimensional vectors of one's

The solution illustrates that positivity constraints on scenario probabilities $x\ \geq\ 0$ are automatically satisfied and can therefore be omitted.

The Lagrange dual function is given as :

$$G\ (\lambda, v)\ =\ L(x\ (\lambda, v),\ \lambda,\ v)$$

As we can see here, it is only a function of the Lagrange multipliers $\lambda\ and\ v$ and therefore has dimension equal to the number of views in addition to a Lagrange multiplier for the requirement that posterior probabilities sum to 1.

Hence, this dual problem is solved as

$$x\ (\lambda\star, v\star)\ =\ argmax_{\lambda>0,v}\ G\ (\lambda, v)$$

and subsequently recover the solution to the original equation (3) by computing

$$x(\lambda,v) = \ e^{\ln p - \mathbf{1} - G^T\lambda - A^T v}$$

$$x\,(\lambda\star,\nu\star) = e^{\ln p - \mathbf{1} - G^T\lambda\star - A^T\nu\star}$$

$$q = x\,(\lambda\star,\nu\star)$$

3.0 Numerical Schemes

3.1 Sequential Quadratic Programming (SQP)

Sequential Quadratic Programming is an iterative method for nonlinear optimization. It solves a series of quadratic programming subproblems, each of which approximates the original nonlinear problem more closely. SQP can be used to minimize the Kullback-Leibler divergence subject to linear constraints. Each iteration involves solving a quadratic approximation to the original problem, updating the solution iteratively until convergence.

3.2 Interior-Point Methods

Interior-Point methods are used to solve large-scale linear and nonlinear convex optimization problems. They work by iteratively improving a solution within the feasible region defined by the constraints. Interior-Point methods can handle the constraints in the entropy pooling optimization problem effectively, ensuring that the solution stays within the feasible region while minimizing the relative entropy.

3.3 Gradient Descent Methods

Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. It involves taking steps proportional to the negative of the gradient of the function at the current point.

3.4 Newton's Method

Newton's method is an iterative method for finding successively better approximations to the roots of a real-valued function. For optimization, it can be used to find the stationary points of the objective function. Newton's method can be adapted to solve the entropy pooling problem by iteratively updating the solution based on the second-order Taylor expansion of the objective function.

3.5 Conjugate Gradient Method

The conjugate gradient method is an algorithm for the numerical solution of systems of linear equations whose matrix is symmetric and positive-definite. Conjugate gradient methods can be employed to solve the linear systems arising in the quadratic subproblems of SQP or interior-point methods.

4.0 Python Code and Explanation

Importing the required libraries and functions:

```python
import pandas as pd
import numpy as np
from entropy_pooling import ep
```

The array includes the indexes of the chosen columns of the assets. The imported dataframe object is stored in **df**.

```python
col_indices = [20, 21, 22, 23, 24]
df = pd.read_csv('RETURNS_new.csv', usecols=col_indices)
```

Column headers are added with the stock tickers. Rows with null values are dropped and the returns are stored in **returns_df**.

```python
df.columns = ['GOOGL', 'AAPL', 'ADBE', 'META', 'MSFT']
returns_df = df.dropna().iloc[1:]
```

The Dataframe is converted into a NumPy array.

```python
R = returns_df.values.astype(float)
S = R.shape[0]
```

A uniform prior probability distribution is created and the prior mean and volatility is calculated using this.

```python
p = np.ones((S, 1)) / S
means_prior = p.T @ R
vols_prior = np.sqrt(p.T @ (R - means_prior)**2)
```

The views are defined in a dictionary, with the values being in decimal.

The expected mean of apple stock is adjusted to 0.21%.

The volatility of microsoft is constrained to be less than equal to 1.5%

The expected mean of google stock is adjusted to 0.53%

The volatility of meta stock is adjusted to 2.51%

```python
views = {
    'AAPL_mean': 0.21 / 100,
    'MSFT_vol': 1.5 / 100,
    'GOOGL_mean': 0.53 / 100,
    'META_vol': 2.51 / 100
}
```

The sum of probabilities must be equal to 1 and the mean views of some assets is defined.

```
A_eq = np.vstack((
    np.ones((1, S)),   # Sum of probabilities must be 1
    R[:, 1],           # AAPL mean view
    R[:, 0]            # GOOGL mean view
))

b_eq = np.array([1, views['AAPL_mean'], views['GOOGL_mean']]).reshape(-1, 1)
```

Similarly, the inequality constraints are defined.

```
G_ineq = np.vstack((
    (R[:, 4] - means_prior[0, 4])**2, # MSFT volatility view
    (R[:, 3] - means_prior[0, 3])**2  # META volatility view
))

h_ineq = np.array([views['MSFT_vol']**2, views['META_vol']**2]).reshape(-1, 1)
```

```
# Calculating posterior probabilities using entropy pooling
posterior_prob = ep(p, A, b, G, h)

# Calculating posterior mean
posterior_mean = (posterior_prob.T)*100 @ R
```

The posterior probabilities and mean are calculated.

The posterior volatility is calculated for each asset as the square root of weighted average of squared deviations from posterior mean. The results are subsequently outputted.

```
posterior_vols = np.zeros(R.shape[1])
for i in range(R.shape[1]):
    squared_deviations = (R[:, i] - (posterior_mean[0, i] / 100))**2
    posterior_vols[i] = np.round(np.sqrt(posterior_prob.T @ squared_deviations).item() * 100, decimals=4)
```

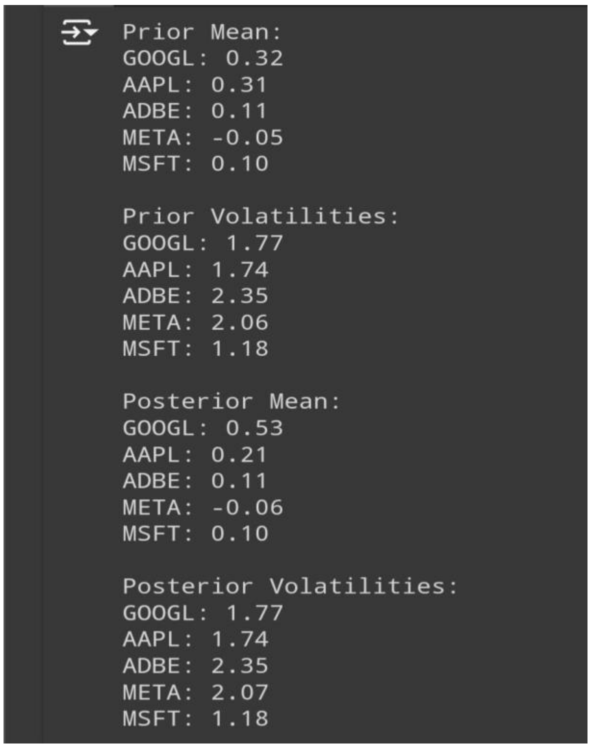
```
# Display results
print("Prior Mean:")
for i, ticker in enumerate(['GOOGL', 'AAPL', 'ADBE', 'META', 'MSFT']):
    print(f"{ticker}: {means_prior[0, i] * 100:.2f}")

print("\nPrior Volatilities:")
for i, ticker in enumerate(['GOOGL', 'AAPL', 'ADBE', 'META', 'MSFT']):
    print(f"{ticker}: {vols_prior[0, i] * 100:.2f}")

print("\nPosterior Mean:")
for i, ticker in enumerate(['GOOGL', 'AAPL', 'ADBE', 'META', 'MSFT']):
    print(f"{ticker}: {posterior_mean[0, i]:.2f}")

print("\nPosterior Volatilities:")
for i, ticker in enumerate(['GOOGL', 'AAPL', 'ADBE', 'META', 'MSFT']):
    print(f"{ticker}: {posterior_vols[i]:.2f}")
```

5.0 Result

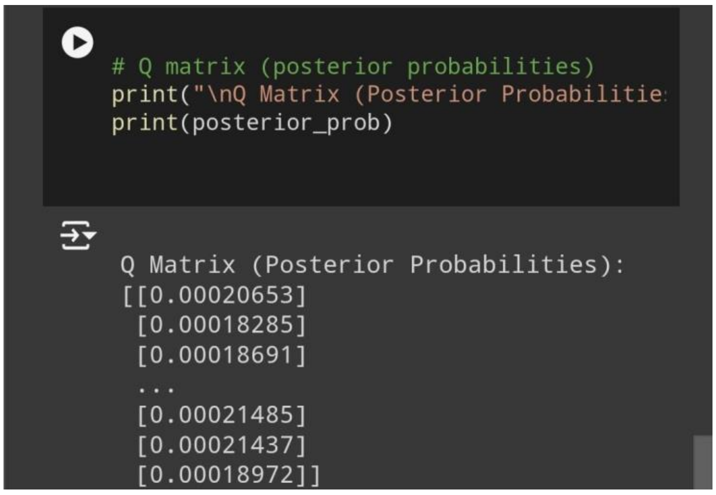


```
Prior Mean:
GOOGL: 0.32
AAPL: 0.31
ADBE: 0.11
META: -0.05
MSFT: 0.10

Prior Volatilities:
GOOGL: 1.77
AAPL: 1.74
ADBE: 2.35
META: 2.06
MSFT: 1.18

Posterior Mean:
GOOGL: 0.53
AAPL: 0.21
ADBE: 0.11
META: -0.06
MSFT: 0.10

Posterior Volatilities:
GOOGL: 1.77
AAPL: 1.74
ADBE: 2.35
META: 2.07
MSFT: 1.18
```



```
# Q matrix (posterior probabilities)
print("\nQ Matrix (Posterior Probabilitie
print(posterior_prob)
```

```
Q Matrix (Posterior Probabilities):
[[0.00020653]
 [0.00018285]
 [0.00018691]
 ...
 [0.00021485]
 [0.00021437]
 [0.00018972]]
```

As we can see here, the posterior matrix change according to our views from the prior one i.e. p, our views get incorporated in the prior distribution p, and we get a number of posterior matrices x, among them, we choose the one with minimum relative entropy. The final one is q.

6.0 Contributions:

Rama, Ayushi, Rishit, Shahbaaz

We are extremely happy for the completion of this project of Entropy Pooling under the Quantitative Finance Cohort’24. We would like to express our deepest gratitude to our project instructor, Kshitij Anand, for his invaluable guidance, support, and encouragement throughout the duration of this project. His expertise and insights were instrumental in shaping the direction and outcome of this work.