Kernel distribution, option covering and systematic strategy

Guillaume Attila, Alexandre Brouardelle, Florent Wei Han,
Adel Khennouf, Angelo Pavia–Desmars

March 16, 2024

Supervised by Matthieu Garcin

Abstract

Our project explore the application of statistic methods for financial assets management, presenting a complete analysis of density estimation, option returns forecasting, systematic strategy and risk management.

The studies begin with discussing and implementing non-parametric density estimation methods, particularly kernel density estimation, which are used to modelize assets classes returns distributions. These techniques rely on the selection of a free parameter, crucial for the smoothness of the estimation: the bandwidth. They give informations on the underlying probability distribution of assets returns, and are essential for good risk management and forecasting.

The project explores methodologies for forecasting the distribution of option returns, essential for accurate pricing and effective risk management. We then propose systematic trading strategies, including moving average crossover strategies based on kernel density estimations of asset returns. These strategies aim to dynamically generate buy and sell signals by leveraging short and long-term moving averages of return distributions to optimize portfolio performance and mitigate risks.

Our research emphasizes the importance of risk management in asset management and trading, discussing techniques such as delta-vega replication strategies to manage exposure to asset price changes and volatility. Performance evaluation of trading strategies is conducted using measures such as maximum drawdown, volatility, and total return, with backtests used to assess effectiveness on historical data.

Contents

1 Introduction

The probability distribution of asset returns is essential in finance, particularly in asset management, whether to quantify risks or expected gains from holding a security. Among the numerous existing probability laws, the most accurate ones are non-parametric laws. Kernel-based laws, in particular, have several attractive features that have led to their widespread use. However, these kernel laws rely on selecting a free parameter that characterizes the smoothness of the density function. A substantial statistical literature has addressed this problem. The most common choice for the free parameter is to minimize the asymptotic integrated mean squared error of the density function compared to the true density. Unfortunately, this true density is practically never known, making the evaluation of the error uncertain. Various techniques are then employed to mitigate this difficulty, such as using a statistical plug-in, cross-validation, etc. Among other possible approaches, three will be specifically considered:

- Selection of the free parameter by simulation in the non-parametric law with Markov Chain Monte Carlo (MCMC).
- Selection of the free parameter maximizing the predictive power of the estimated density, using Probability Integral Transform (PIT).
- Selection of the free parameter maximizing the "complexity" of the estimated density, where complexity, coming from econophysics, characterizes, in statistical terms, the "distance" between two situations: overfitting and underfitting.

The objective of this project is to build a tool for estimating these densities with automatic selection of the free parameter, to do so for both unconditional and conditional densities, and to apply it to financially relevant situations:

- Forecasting option returns and hedging: Option valuation relies on a risk-neutral probability density, which is parametric, straightforward, and not directly related to the historical density discussed earlier. However, knowing the historical distribution of underlying asset returns, one can calculate the expected value or any quantile value of the option at a future date. This allows quantifying the expected return for an option as well as its Value at Risk (VaR), information useful for options traders.

- Construction of a systematic strategy: Knowledge of the historical probability density provides not only an expectation of future returns but also a set of statistics that can be used as inputs for a more advanced forecasting model. The goal here is to build a trading algorithm whose performance and risk need to be tested.

2 Non-parametric Density Estimation

We assume we are given n independent and identically distributed (i.i.d.) observations $X_1, \ldots, X_n$, of probability density funstion f. The kernel density is defined, for $x \in \mathbb{R}$, by

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - X_i}{h}\right) \tag{1}$$

where $h > 0$ is the bandwidth and K is the kernel, a function following the same rules as a probability density function, positive and integrable. The corresponding cumulative distribution function (cdf) is

$$\hat{F}_h(x) = \frac{1}{n} \sum_{i=1}^{n} \kappa\left(\frac{x - X_i}{h}\right) \tag{2}$$

where $\kappa(x) = \int_{-\infty}^{x} K(y)\, dy$.

And here is some commonly used kernel functions. We will only use Gaussian and Epanechnikov kernel in this

1. Gaussian (Normal):

$$K(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}u^2} \tag{3}$$

2. Tophat (Uniform):

$$K(u) = \begin{cases} \frac{1}{2} & \text{if } |u| \leq 1 \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

3. Epanechnikov:

$$K(u) = \begin{cases} \frac{3}{4}(1 - u^2) & \text{if } |u| \leq 1 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

4. Exponential:

$$K(u) = \frac{1}{2} e^{-|u|} \tag{6}$$

5. Linear (Triangular):

$$K(u) = \begin{cases} 1 - |u| & \text{if } |u| \leq 1 \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

6. Cosine:

$$K(u) = \begin{cases} \frac{\pi}{4} \cos\left(\frac{\pi u}{2}\right) & \text{if } |u| \leq 1 \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

For the following section, all the data used for the graphs are from the one and only Nasdaq index from 2021 to 2023.

2.1 Classical Approach with AMISE and LOOCV

2.1.1 Rule of Thumb

The first two methods for bandwidth selection are the most famous and straightforward:

- Scott's rule of thumb (Scott, 1992):

$$h \approx \frac{3}{5} \cdot \hat{\sigma} n^{-1/3} \tag{9}$$

- Silverman's rule of thumb (Silverman, 1986):

$$h = 0.9 \cdot \min(\hat{\sigma}, \frac{IQR}{1.34}) n^{-1/5} \tag{10}$$

Although they are simple and easy to compute, they have limitations. The first one requires the data from the normal distribution. If you work with another kind of distribution, you will not get meaningful results using Scott's rule of thumb. The second rule is more robust but it doesn't work well in complicated cases.

Here is our implementation of the two function :

Julia code

```
    function silverman_bandwidth(returns)
    n = length(returns)
    std_dev = std(returns)
    iqr_val = iqr(returns)
    return 0.9 * min(std_dev, iqr_val / 1.34) * n ^ (-1/5)
end
```

Julia code

```
    function scott_bandwidth(returns)
    n = length(returns)
    std_dev = std(returns)
    return 3.5 * std_dev * n ^ (-1/3)
end
```

By plotting it, we can see better how the kernel density estimation approach our returns with selected bandwidth using the two empirical rules. we have obtain a value of 0.32 for the silverman bandwidth and 0.60 for the scott bandwidth. By looking at the results, we will have a preference over the silverman one as it fits the data in a better way.

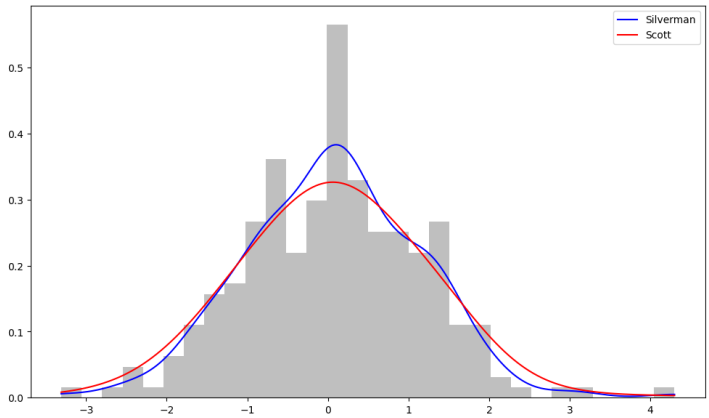

Figure 1: First approach of kernel density estimation with bandwidth selection using empirical rules

2.1.2 AMISE

The AMISE (Asymptotic Mean Integrated Squared Error) approach is a methods used for selecting the bandwidth when estimating kernel density. The choice of bandwidth is crucial as it determines how smooth or rough the density estimation will be. AMISE focuses on minimizing the average integrated squared error between the kernel density estimator and the true unknown density. It balances the estimator's variance (imprecision due to random sampling) and bias (systematic error). AMISE is particularly useful when data follow a known distribution or when assumptions can be made about the underlying density shape. However, computing the exact optimal AMISE bandwidth usually requires another preliminary estimation.

The useful feature of AMISE(h) is that its minimizer is simply calculated (Jones, Marron, Sheather, 1996):

$$h_{\text{AMISE}} = \left[\frac{R(K)}{nR(f'') \left(\int x^2 K \right)^2} \right]^{1/5} \tag{11}$$

In implementing the AMISE method, we obtain a bandwidth of 0.29. So let's look up on the kernel density estimation of the returns on our data, with each gaussian and epanechnikov kernel :

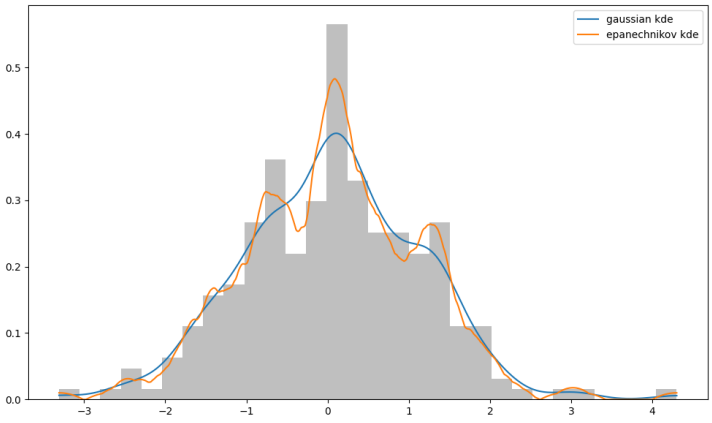

Figure 2: Kernel density estimation with optimized bandwidth by AMISE approach over the histogram of the returns

A bandwidth of 0.29 seem to be a good estimation for the AMISE approach, especially for the Epanechnikov kernel, wich fit the returns of our data in a good way.

2.1.3 LOOCV

The LOOCV (Leave-One-Out Cross-Validation) approach is an other method used for selecting the bandwidth in KDE. LOOC aims to choose the bandwidth that maximizes the probability of the observed dataset, making the observed data as "likely" as possible under the density estimator. For each observation, density is estimated by leaving

that observation aside (hence the name "Leave-One-Out"). The optimal bandwidth maximizes the sum of the logarithms of these densities. LOOC is non-parametric, making no assumptions about the underlying density shape, thus applicable to a wide range of situations but can be computationally expensive for large samples.

The LOOCV error in order to find the best bandwidth is given by the formula :

$$h_{\text{LOOCV}} = \arg\max_{h} \sum_{i=1}^{n} \log(\hat{f}_{-i}(x_i; h)) \tag{12}$$

So, we implemented this formula on our data and in a range of bandwidth going from 0 to 1. The best bandwidth is where the curve of each kernel function on the graph is at the lowest point of the LOOCV error axis :

- Optimal bandwidth for Gaussian kernel : **0.52** with a LOOCV error of 1.55
- Optimal bandwidth for Epanechnikov kernel : **1** with a LOOCV error of 1.6
- Optimal bandwidth for Tophat kernel : **0.6** with a LOOCV error of 1.65

The best approach seems to be the Gaussian kernel one, with the lowest LOOCV error. By plotting the density estimation with this kernel, we can therefore see that the bandwidth may be too large because the curve of the kde is very smoothed.

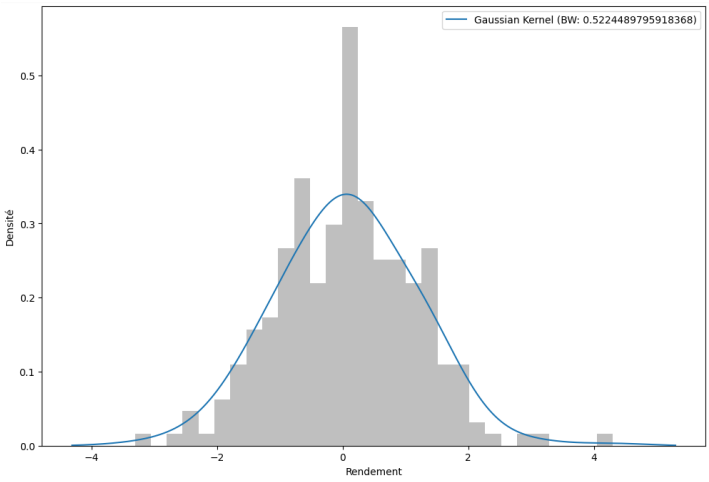

Figure 3: Estimated density of returns using Gaussian kernel with bandwidth h=0.52

2.2 Free Parameter Selection using Markov Chain Monte Carlo (MCMC)

The MCMC method is a tool for estimating complex probability distributions by generating a sequence of samples from a target distribution.

The Metropolis-Hastings is a Markov Chain Monte Carlo algorithm for selecting the free parameter in the kernel function is an iterative process used to estimate density optimally. It proposes a new bandwidth value at each iteration, accepting or rejecting it based on the acceptance probability determined by the density difference between the current proposal and the new proposal. This allows finding an appropriate bandwidth

that optimizes the accuracy of density estimation, crucial for precise data analysis.

In our algorithm, we have the bandwidth value h_{MCMC} derived from the MCMC method and refined by the average of accepted samples, providing an optimal compromise between smoothing and precision in density estimation. This averaging approach ensures a reliable representation of the data distribution, reducing the impact of random variations.
Here is the implementation of the Acceptance-Rejection function proper to the Metropolis-Hastings algorithm, and the function to compute the optimal bandwidth for this method :

Python code

```
def accept_reject(bandwidth, new_bandwidth, data):
    log_likelihood_old = np.sum(np.log(kernel_density(data, data,
        bandwidth)))
    log_likelihood_new = np.sum(np.log(kernel_density(data, data, new_
        bandwidth)))
    acceptance_ratio = np.exp(log_likelihood_new - log_likelihood_old)
    return acceptance_ratio > np.random.uniform()

def mcmc_bandwidth(data,kernel_density ,initial_bandwidth, n, proposal_
    width):
    current_bandwidth = initial_bandwidth
    accepted_samples = []
    for _ in range(n):
        new_bandwidth = current_bandwidth + np.random.normal(scale=
            proposal_width)
        if new_bandwidth > 0 and accept_reject(current_bandwidth, new_
            bandwidth, data):
            current_bandwidth = new_bandwidth
        accepted_samples.append(current_bandwidth)
    return accepted_samples

initial_bandwidth = 0.2
n = 1000
proposal_width = 0.1
h = mcmc_bandwidth(data, kernel_density, initial_bandwidth, n, proposal_
    width)
h_mcmc = np.mean(h)
print(h_mcmc)
```

The function **accept_reject** implements the acceptance-rejection step of the Metropolis-Hastings algorithm. It computes the log-likelihood of the current bandwidth (**bandwidth**) and the proposed new bandwidth (**new_bandwidth**) based on the data. Then, it computes the acceptance ratio and decides whether to accept the new bandwidth based on this ratio.

The function **mcmc_bandwidth** performs the Metropolis-Hastings MCMC algorithm to estimate h. It starts with an initial bandwidth (**initial_bandwidth**) and iteratively proposes new bandwidth values by adding a random value drawn from a normal distribution with standard deviation **proposal_width**. It then accepts or rejects these proposed values using the **accept_reject** function based on the likelihood ratio. The function

returns a list of accepted bandwidth samples after n iterations.

With initiating parameters, we have $h_{MCMC} = 0.32$. So with this estimated bandwidth, we can plot the samples using Metropolois-Hastings and then plot the estimated density :

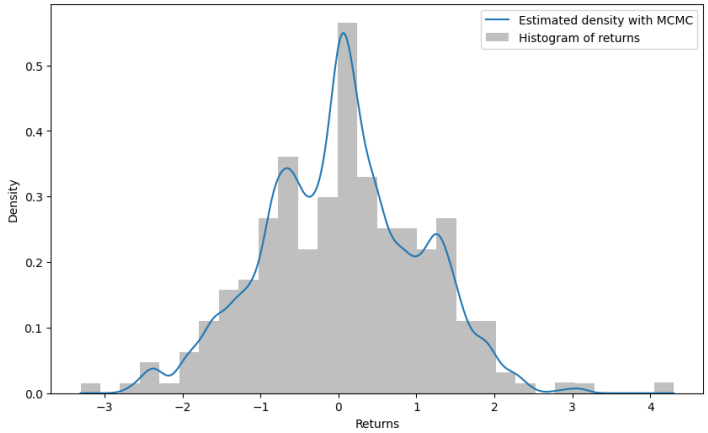

Figure 4: Estimated density of MCMC sampling compared to the histogram of real returns with bandwidth h=0.32

This estimated density graph illustrates the effectiveness of the Metropolis-Hastings algorithm in capturing our data distribution. As the density curve is very close to the histogram of real data, it indicates a good calibration of the MCMC model. We will keep this method for further utilisations.

2.3 Free Parameter Selection using Probability Integral Transform (PIT)

The Probability Integral Transform (PIT) is a technique used to assess the adequacy of a probabilistic model. It transforms estimated probabilities into a uniform distribution. In the context of bandwidth selection for time-varying kernel density estimation, PIT is used to evaluate the quality of the estimated density in predicting new observations. By applying PIT to the residuals obtained from density forecasting, one can assess the predictive performance of the density estimator for different choices of bandwidth.

Python code

```
def dynamique_kernel_cdf(data, x, bandwidth, w, t0):
t = len(data)
mu = data.mean()
sigma = data.std()

kernel_valeurs_cdf = 0
for i in range(t0):
    cdf_value = norm.cdf((x - data[i]) / bandwidth, mu, sigma**0.5)
    kernel_valeurs_cdf += w**(t0-i) * cdf_value * ((1-w)/(1-w**t0))
for i in range(t0+1, t):
    cdf_value = norm.cdf((x - data[i]) / bandwidth, mu, sigma**0.5)
```

```
        kernel_valeurs_cdf = w * kernel_valeurs_cdf + (1-w) * cdf_value

    return kernel_valeurs_cdf

def calcul_dynamique_cdf(t,h,w,t0):
  return dynamique_kernel_cdf(returns, returns[t],h,w,t0)

t0=200
nu=22

min_statistic= float('inf')
h_PIT=None
w_PIT=None

h_valeurs=np.linspace(0.3,0.8,6)
w_valeurs=np.linspace(1-(1/nu),0.99,15)

for h in h_valeurs:
  for w in w_valeurs:
    dynamique_cdf_valeur=[calcul_dynamique_cdf(t,h,w,t0) for t in range(
        t0+1, len(returns))]
    statistic = d_nu(dynamique_cdf_valeur,t0,nu)
    if statistic< min_statistic:
      min_statistic=statistic
      h_PIT=h
      w_PIT=w
print('Meilleure h:', h_PIT)
print('Meilleur w:', w_PIT)
```

In the optimization process, h_PIT and w_PIT store the best parameters found. The method involves generating grids of potential h and w values and iterating through all combinations. For each pair (h, w), the optimization statistic is computed by evaluating the dynamic cumulative distribution function and measuring the fit against a uniform distribution using d_nu. The optimal parameters are then selected based on the computed statistics.
We have for output a bandwidth h=0.3 which is quite good at first look and w have a value of 0.95.
We can plot the estimated density of returns, and see that the density curve is not so well representatitve of the returns distribution, with a peek not very important on 0.

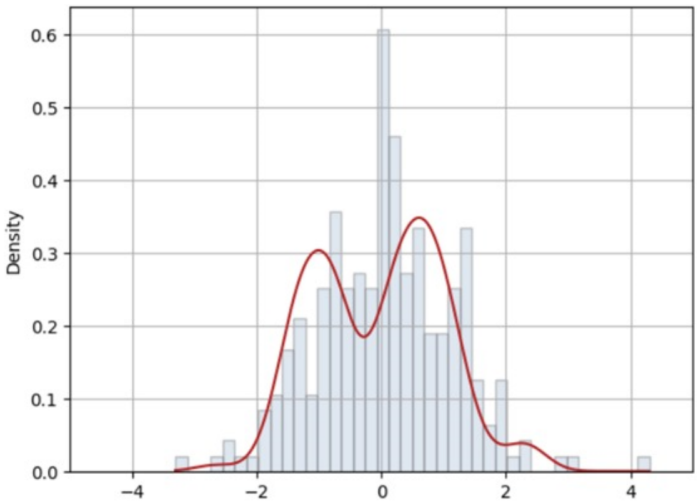

Figure 5: Estimated density using PIT, h=0.3

2.4 Free Parameter Selection by Complexity

The main objective is to propose an alternative method for bandwidth selection based on principles from information theory and the physics of complex systems. This approach is considered a way to optimize a new measure of complexity, while avoiding the overfitting and underfitting problems encountered in traditional methods.
Current methods for bandwidth selection, which typically attempt to minimize accuracy criteria such as the integrated mean squared error (MISE), do not take into account the underlying complexity of financial data.
In the literature (e.g.[7]), the author proposes a new specific complexity measure for kernel density estimation in financial data, which is the minimum function between divergences. One is the divergence between the estimated cumulative distribution function and the empirical distribution, and the other is the divergence between the estimated probability density function and the parametric distribution (which is Gaussian here). This complexity measure is designed to achieve low values in cases of overfitting and underfitting. Therefore, an appropriate fit of the true density should be a compromise between these two extreme cases and should correspond to a higher value of complexity. The selected bandwidth will thus be the one maximizing complexity.

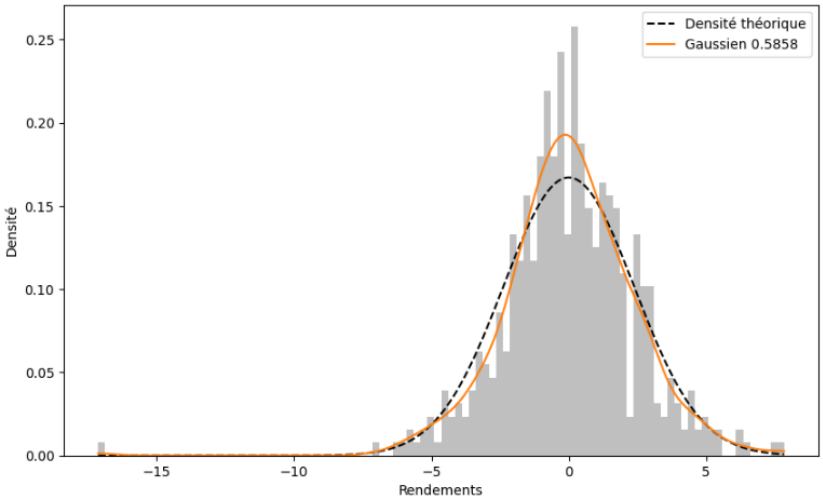

Figure 6: Kernel Density Estimation with Selected Bandwidth by Complexity h=0.5858

3 Conditional Version

The conditional distribution in finance refers to the probability distribution of financial asset returns given certain information or conditions. In other words, it describes how the returns of an asset evolve based on explanatory variables or information available at a given moment. Additionally, they quantify the probability of future returns given current information.

Reminder for Kernel estimator of Conditional Densities :
The natural kernel estimator of $f(y \mid x)$ is (e.g., Scott 1992, p. 220)

$$\hat{f}(y \mid x) = \frac{\hat{g}(x, y)}{\hat{h}(x)}, \tag{13}$$

where

$$\hat{g}(x, y) = \frac{1}{nab} \sum_{j=1}^{n} K\left(\frac{\|x - X_j\|}{a}\right) K\left(\frac{\|y - Y_j\|}{b}\right) \tag{14}$$

is the kernel estimator of $g(x, y)$, and

$$\hat{h}(x) = \frac{1}{na} \sum_{j=1}^{n} K\left(\frac{\|x - X_j\|}{a}\right) \tag{15}$$

is the kernel estimator of $h(x)$. Here, a control the smoothness between conditional densities in the x direction and b control the smoothness between conditional densities in the y direction.

In this section, we will use the Volume of the Nasdaq datas as the explanatory variable.

3.1 Bootstrap Bandwidth Selection Approach

To estimate a conditional density using bootstrap, we repeatedly draw bootstrap samples from our observed data, conditional on the values of the covariates. Then, for each bootstrap sample, you estimate the conditional density using a suitable density estimation method. By averaging these estimates across all bootstrap samples, you can obtain an approximation of the conditional density.
The Bootstrap method is powerful for selecting an optimal bandwidth in density estimation, especially when dealing with uncertainty in the data. It involves repeatedly resampling from the original dataset, estimating the density for each resampled dataset, and then assessing the performance of different bandwidth.

On our implementation with volume (that we divided in four quartile) as the conditional variable, we computed mean of error on all bootstrap resampled. The error is calculated as the squared integrated distance between the densities estimated for the bootstrap sample and the original dataset. Finally, the bandwidth (chosen on an interval of bandwidth values) that minimizes this average error is chosen as optimal for the current quartile.

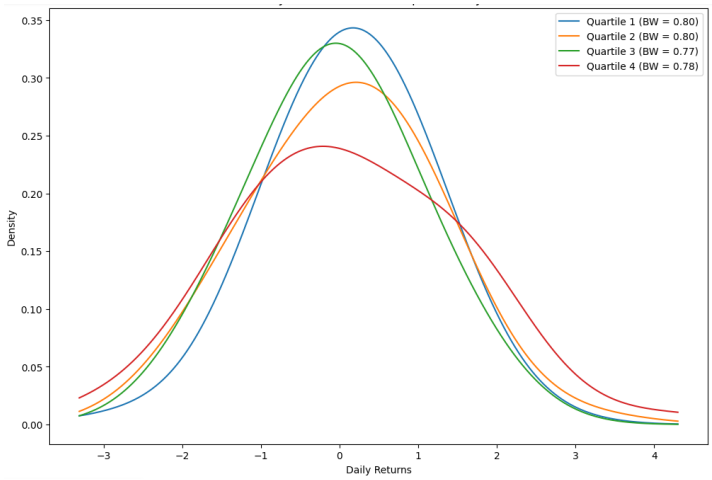

Figure 7: Estimated Density of Returns Conditional with different quartile of the Volume

This show that our method bootstrap error minimization to choose the bandwidth may be ameliorated, as the large bandwidths for each quartile leads to a very smooth conditional density estimation of the returns. We should have maybe change our error metric, with for example Kullback-Leibler metric that may be more informative or sensitive to change in the density estimation. Also, the method used to calculate the error between the estimated densities on the bootstrap resamples and the original dataset may not be sensitive to the difference in bandwidth within this range.

3.2 Method with a Threshold on Volume

To go on further analysis on the influence of volume on returns, we fixed a threshold wich is the median of the volume of our data. Simply with this fixed variable, we plot the density of the returns when they are over the threshold, and when they are under. Then we can compare the influence of bigger and smaller volume on the historical returns:

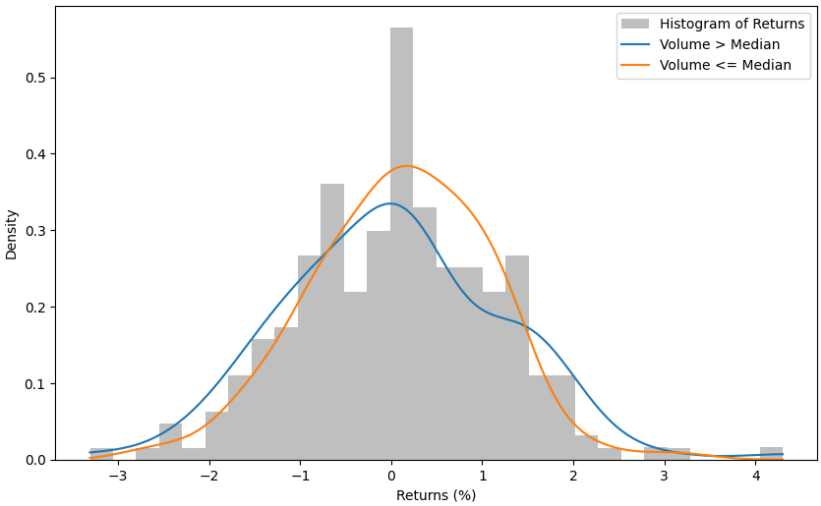

Figure 8: Estimated Density of Returns Conditional to Volume with fixed threshold, Gaussian Kernel

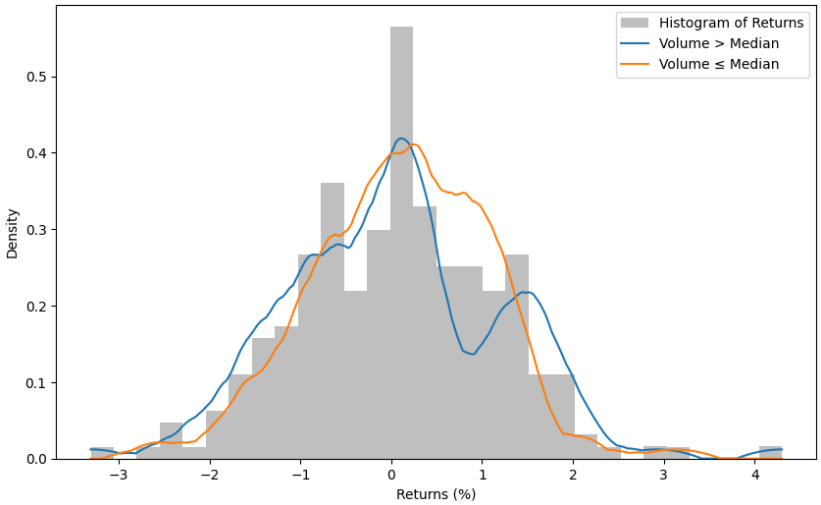

Figure 9: Estimated Density of Returns Conditional to Volume with fixed threshold, Epanechnikov Kernel

We can observe that the comportment of the density estimation seems to be approximately the same when we fix a threshold of volume. The estimation using Epanechnikov kernel seem to be a better approach as it fits good over the returns, whereas the Gaussian kernel may oversmooth the data.

4 Application to a Broad Class of Assets

What are the main goals of applying our kernel density estimations methods to differents class of assets?

Estimating historical probability densities can be used to build systematic trading models. These models use past return statistics to make automated trading decisions. Application to a large asset class allows testing these strategies on a diversified set of financial instruments.

Different asset classes have distinct characteristics in terms of volatility, return distribution, and behavior. Applying modeling methods to various asset classes allows adaptation to the specifics of each category, which is essential for effective asset management.

Diversification is a key strategy to mitigate risks. When applying return modeling methods to a large asset class, it allows investors to better diversify their portfolio by including a variety of financial assets such as stocks, indexes, cryptocurrencies or commodities. This can help reduce the overall volatility of the portfolio.

4.1 Application to Stocks

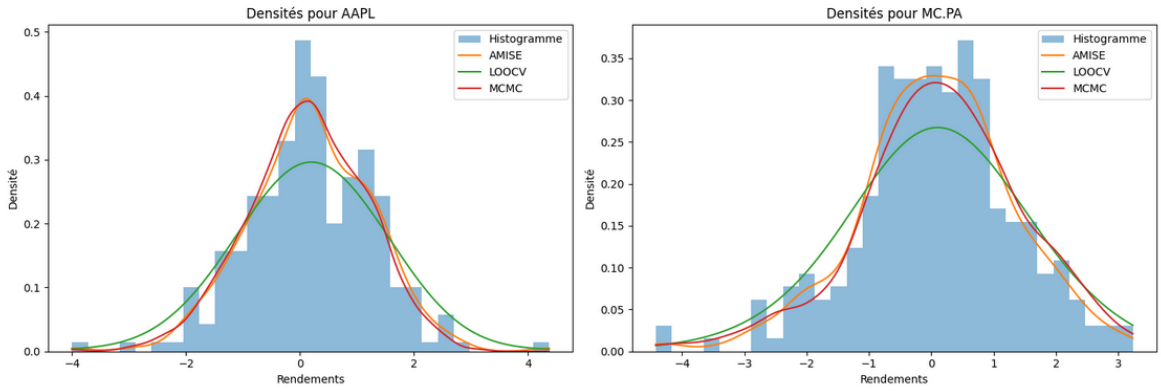

Figure 10: Stock AAPL : Apple, Estimated Density of Return with Different Methods

Figure 11: Stock MC.PA : LVMH - Louis Vuitton Moët Hennessy, Estimated Density of Return with Different Methods

4.2 Application to Commodities

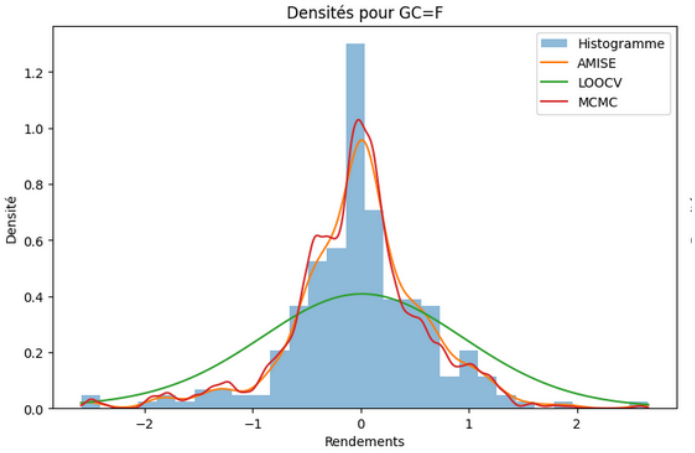

Figure 12: Commodity GC=F : Gold, Estimated Density of Return with Different Methods

4.3 Application to Indexes

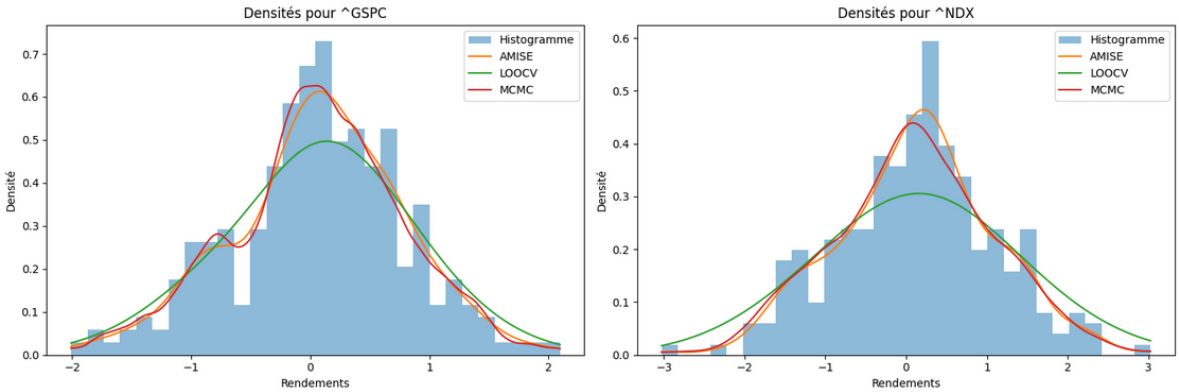

Figure 13: Index GSPC : SP500, Estimated Density of Returns with Different Methods

Figure 14: Index NDX : NASQAD 100, Estimated Density of Returns with Different Methods

4.4 Application to Cryptocurrencies

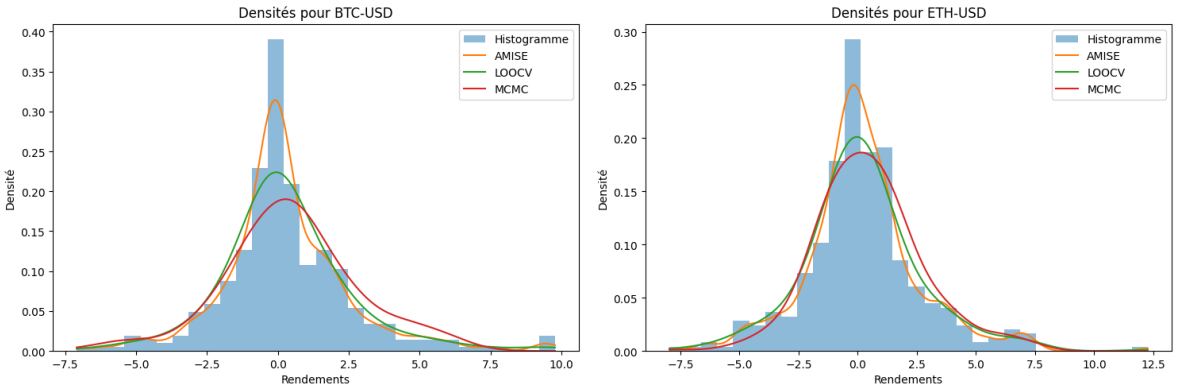

Figure 15: Crypto BTC-USD : Bitcoin US dollar, Estimated Density of Returns with Different Methods

Figure 16: Crypto ETH-USD : Ethereum US dollar, Estimated Density of Returns with Different Methods

The application to a large amount of assets allows us to see the most suitable models for each asset in an empirical way. We note that some models perform better on extreme events and on the sensitivities to variations in yield densities. This empowers financial analysts to tailor their approach according to the unique characteristics of each asset, thereby optimizing the risk-return profile of investment portfolios. Additionally, this extensive application aids in detecting systemic risks and asset-specific vulnerabilities, which is critical for portfolio diversification and risk mitigation strategies.

By evaluating model performance across different assets, we can also discern patterns that may be indicative of broader market behaviors. This can lead to the development of more sophisticated financial instruments and investment strategies that are responsive to the underlying risk factors associated with each asset class.

5 Forecasting Option Returns and Hedging

5.1 Distribution of the Future Value of the Option

After studying the different ways of estimating densities using nonparametric kernel density functions, we have carried out a process to evaluate the future price distribution of a financial option using a Monte Carlo simulation approach based on nonparametric historical daily returns, and the application of the Black-Scholes model for option pricing.
To achieve this, we have implemented several functions:

- The first function, **simulate_prices()**, calculates a price trajectory for an underlying asset over a given period, based on an initial price **S0** and a series of daily returns. This function creates an array to store the simulated prices, initializes the first price with S0, then calculates each subsequent price by adjusting the previous price by the corresponding daily return.
- The **sample_returns()** function randomly selects, with replacement, a series of daily returns from a data set of historical returns, enabling the generation of various future price scenarios for the underlying asset.
- **get_k_price()** uses **sample_returns()** to generate a sample of returns, then simulates a price trajectory from these returns using **simulate_prices**, starting from the asset's last known price. This function returns the final simulated asset price after k days.
- **n_simu()** repeats the price simulation process n times to generate a distribution of possible future values for the underlying asset after k days, using sampled returns from the supplied data set.
- The **black_scholes_call()** function calculates the price of a call option using the parameters of the Black-Scholes model, while **vega()** calculates the option's vega, a measure of the sensitivity of the option price to the volatility of the underlying asset.
- The **find_volatility_newton_raphson()** function uses the Newton-Raphson method to find the implied volatility of the underlying asset that corresponds to the market price of an option, iteratively adjusting an initial estimate until the difference between the price calculated via Black-Scholes and the market price is negligible.

Using data from MC.PA (LVMH), we obtain these plots :

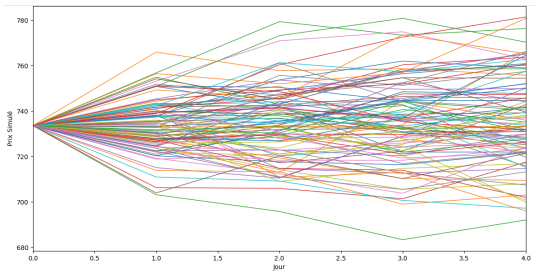

Figure 17: 100 Simulated Price Trajectories over 5 Days

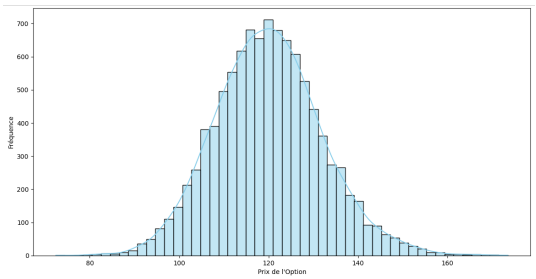

Figure 18: Distribution of Simulated Option Prices

We have a **Sigma of 36.44**. This represents the implied volatility of the underlying asset expressed as a percentage. A crucial parameter in the Black-Scholes model, it reflects market expectations regarding the amplitude of future movements in the price of the underlying asset.

5.2 Risk Measures and Expected Returns

In this section, we have implemented functions that combine risk assessment and return anticipation for a financial asset, using several analytical methods based on historical return data. These methods include Value at Risk (VaR) calculation, Conditional Value at Risk (CVaR), and volatility forecasting using the GARCH model. Each of these approaches plays a crucial role in risk management and the development of investment strategies.

The first code calculates the VaR of a portfolio using kernel density estimation (KDE). VaR measures the worst-case scenario of expected loss on a portfolio, at a given confidence level, over a specific time horizon. The **var_from_kde()** function estimates the probability density of historical returns using KDE, then calculates the cumulative distribution function (CDF) to determine the maximum possible loss that will not be exceeded with a certain level of confidence. VaR is obtained by identifying the return value corresponding to the specified confidence level, thus reflecting the risk of extreme loss over the horizon under consideration.

Julia code

```
function var_from_kde(returns, confidence_level, bandwidth)
    x = range(minimum(returns), maximum(returns), length=1000)
    kde = kernel_density_estimate(x, returns, bandwidth)
    cdf = cumsum(kde, dims=1) ./ sum(kde)
    var_index = findfirst(cdf .> confidence_level)
    var = x[var_index]
    return -var
end

confidence_level = 0.05
bandwidth = silverman_bandwidth(returns)
var = var_from_kde(returns, confidence_level, bandwidth)
```

Then the **cvar_from_kde calculates()** the CVaR, also known as Value at Conditional Risk. CVaR provides a more complete measure of risk by averaging losses that exceed VaR, giving an estimate of the expected loss in worst-case scenarios beyond VaR. This is done by filtering the yields that are below the calculated negative VaR and taking their average. CVaR is often considered a more robust and informative risk measure than VaR alone because it takes into account the tail shape of the loss distribution.

Julia code

```
function cvar_from_kde(returns, var)
    filtered_returns = filter(r -> r <= -var, returns)
    cvar = mean(filtered_returns)
    return cvar
end

var = var_from_kde(returns, confidence_level, bandwidth)
confidence_level = 0.05
bandwidth = silverman_bandwidth(returns)
cvar = cvar_from_kde(returns, var)
println("CVaR: ", cvar)
```

The last function implements a volatility forecast using the Generalized Autoregressive Conditional Heteroskedasticity (GARCH) model. The GARCH model is commonly used to model the volatility of financial returns, as it captures the volatility clusters and the thickness of the yield distribution tails. The **garch_forecast()** function calculates the expected volatility for each period based on a constant term coefficient (**a0**), the impact of past returns on future volatility (**a1**), and the influence of past volatility on future volatility (**b1**). This approach generates a time series of expected volatility, providing valuable insights for risk management and investment decision-making.

Julia code

```
function garch_forecast(returns, a0, a1, b1)
    n = length(returns)
    var_forecast = zeros(n)
    var_forecast[1] = var(returns)

    for t in 2:n
        var_forecast[t] = a0 + a1 * (returns[t-1] ^ 2) + b1 * var_
            forecast[t-1]
    end

    return var_forecast
end

a0 = 0.00001
a1 = 0.1
b1 = 0.8

garch_forecast(returns, a0, a1, b1)
```

The objective of this part was to develop risk measurement tools and yield expectations that will serve us in the next part.

In our application on the **MC.PA** action data, we obtain a **VAR of 2.37** and a **CVaR of -2.60**.We can interpret the VaR (95%) as the worst case scenario of a loss that will not exceed 2.37% of the portfolio value. The CVaR goes beyond the VaR by calculating the average losses in the worst case scenario, that is, losses that exceed the

VaR. Thus, on average, the expected losses in the worst 5% of cases will be at least 2.60%.

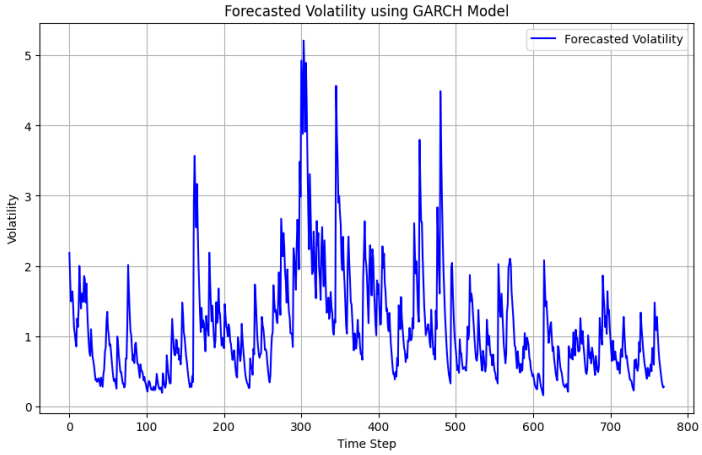

Figure 19: Forecasted Volatility using GARCH Model on MC.PA (LVMH) data

The GARCH graph allows us to visualize the var variation over the study period. The peaks on the chart represent periods where volatility is expected to be higher. During these periods, the risk associated with investing in this asset is greater, as prices are likely to vary further. On the other hand, valleys or periods where the volatility line is lower suggest moments of relative stability where the asset is less volatile. The risk is perceived as lower, and prices are less likely to undergo major changes.

5.3 Delta/vega replication strategy

The key measure of risk we consider is the **Delta** of the option. This is the first partial derivative of the option price with respect to the price of the underlying asset and is, therefore, a measure of exposure to small changes in the price of the underlying asset. The position taken in the underlying asset can reduce or eliminate this exposure.

Vega is the partial derivative of the value of the portfolio with respect to the volatility of the asset price and is a measure of exposure to volatility changes. A trader is typically subject to limits on the permissible size of a portfolio's vega exposure but has discretion on how vega is managed within those limits. Whereas, delta can be changed by taking a position in the underlying asset, vega can be changed only by taking a position in an option or other derivative.

Our objective to hedge options is to achieve a delta and vega neutral position. Delta neutral portfolio result in selling or buying a quantity of the underlying asset equal to the option delta, in order that that the position (profit or loss) realised with options will be compensate by the inverse results on the underlying position. A vega neutral position involves constructing an options portfolio in such a way that it is immune to changes in the implied volatility of the underlying asset. When a trader decides to sell a call option on a stock, for example, they expose themselves to an increase in the

volatility of the underlying asset. To hedge against this, they can buy or sell options with opposing vegas.

Here is the implementation of the delta and vega functions to compute options greeks:

Julia code

```
# Delta function
function delta(S, K, T, r, sigma, option_type::String="call")
    d1 = (log(S / K) + (r + 0.5 * sigma^2) * T) / (sigma * sqrt(T))
    if option_type == "call"
        return cdf(Normal(0, 1), d1)
    elseif option_type == "put"
        return cdf(Normal(0, 1), d1) - 1
    else
        error("Invalid option type. Please choose 'call' or 'put'.")
    end
end

# Vega function
function vega(S, K, T, r, sigma)
    d1 = (log(S / K) + (r + 0.5 * sigma^2) * T) / (sigma * sqrt(T))
    return S * pdf(Normal(0, 1), d1) * sqrt(T)
end
```

When applying these functions on our option on Nasdaq, we found a delta very close to 1, suggesting that the option is very deep in-the-money, and a vega close to 0, which means that our option is almost not affected by change in the implied volatility of the underlying.
To hedge our portfolio, we define a function with a delta neutral target. We also add a vega neutral target for general case, but in our data it will be the same as computed because it is close enough to 0.

As we understand these fonctionnality, we can implement our portfolio optimisation:

Julia code

```
    function cost_function(x)
    n_calls, n_puts, n_shares = x
    portfolio_delta = n_calls * delta_call + n_puts * delta_put + n_
        shares
    portfolio_vega = n_calls * vega_call + n_puts * vega_put
    delta_diff = (portfolio_delta - delta_target)^2
    vega_diff = (portfolio_vega - vega_target)^2
    return delta_diff + vega_diff
end  # This `end` statement closes the `cost_function` function
    definition

x0 = [100.0, 100.0, 100.0]
```

```
result = optimize(cost_function, x0, LBFGS(), Optim.Options(g_tol = 1e-6)
    )

n_calls_opt, n_puts_opt, n_shares_opt = Optim.minimizer(result)

println("Optimal number of call options: ", n_calls_opt)
println("Optimal number of put options: ", n_puts_opt)
println("Optimal number of underlying shares: ", n_shares_opt)
```

cost_function(x) is defined to calculate the cost to minimize. This function takes as input a vector x containing the number of call options, the number of put options, and the number of shares in the portfolio. It then calculates the delta and vega of the portfolio, and returns the sum of the squares of the differences between these values and the delta and vega targets. The cost function is used to assess how closely a given portfolio configuration adheres to the specified targets in terms of delta and vega.

The initial conditions for optimization are defined in x0, representing the initial number of call options (100), put options (100), and shares in the portfolio (100).

The Optim_minimizer function is called to minimize the cost function cost_function using the LBFGS (Limited-memory Broyden-Fletcher-Goldfarb-Shanno) optimisation method. By minimizing this cost function, we aim to find the portfolio configuration that minimizes the deviation from the delta and vega targets. In other words, we seek to find the optimal number of call options, put options, and shares to hold in the portfolio to best achieve these targets.

We have as results, to optimize our portfolio, an order where we should sell approximately all our underlying position, sell all the call options and keep the put options.

6 Systematic strategy

6.1 Backtest code

To apply a backtest on our strategies, we made a **Backtest** class which is structured to test the performance of a given trading strategy by comparing it against a benchmark. Let's break down the components of this class and its functionalities:

- The constructor (**__init__**) initializes an instance of the Backtest class with several parameters including asset prices (prices), a trading strategy function (**strategy_function()**), initial cash available for trading (**initial_cash()**), parameters for the trading strategy (**strategy_params()**), and a transaction cost expressed as a percentage of the transaction amount (**transaction_cost()**). It stores the input parameters as instance attributes. Moreover, the strategy function is called with the provided prices and strategy parameters, returning a DataFrame of positions (buy/sell signals) and the portfolio value over the testing period is calculated by calling **calculate_portfolio_value()** method during initialization.
- The portfolio value calculation function (**calculate_portfolio_value()**) calculates the time series of the portfolio value taking into account positions, asset prices, and transaction costs. Iterates over each position to update the quantity of assets held and available cash after accounting for transaction costs for buying or selling. The value of the portfolio at each time point is calculated as the sum of cash and the market value of held assets. And then, it returns a Pandas Series of portfolio values indexed by date.
- The maximum drawdown calculation function (**calculate_max_drawdown()**) calculates the maximum drawdown of the portfolio over the testing period, which is a measure of the largest peak-to-trough decline.It returns the minimum drawdown, representing the maximum loss from a peak.
- The volatility calculation function (**calculate_volatility()**) calculates the annualized volatility of portfolio returns, a measure of risk. Daily portfolio returns are computed, and their standard deviation (volatility) is calculated and annualized by multiplying with the square root of 252 (the number of trading days in a year).
- The total return calculation function (**calculate_total_return()**) calculates the total return of the portfolio over the testing period. The total return is computed as the percentage change between the final and initial portfolio values.
- The performance plotting function (**plot_performance()**) compares the performance of the trading strategy against a benchmark. It prints performance metrics for both the strategy and the benchmark, including maximum drawdown, volatility, and total return.
- The density plotting function (**plot_densities()**) plots the density of monthly returns for both the strategy and the benchmark. It calculates monthly returns, their mean, variance, and median for both the strategy and the benchmark, prints expected returns and variance for both.Moreover, it uses a kernel density estimate to plot the distribution of returns along with the median for both the strategy and benchmark.

6.2 Expected return and volatility strategy

Python code

```
def calcule_esperance_volatilite(prices, window):
    esperance_list = []
    volatilite_list = []

    for i in range(window, len(prices)):
        window_prices = prices[i-window:i]
        rendements = window_prices.pct_change().dropna()

        if len(rendements) > 1:
            bandwidth = scott_bandwidth(rendements)
            x = np.linspace(rendements.min(), rendements.max(), 100)
            kde = kernel_density_estimate(x, rendements, bandwidth)
            esperance = np.sum(x * kde) / np.sum(kde)
            volatilite = np.sqrt(np.sum((x**2) * kde) / np.sum(kde) -
                esperance**2)
        else:
            esperance = 0
            volatilite = 0
        esperance_list.append(esperance)
        volatilite_list.append(volatilite)

    esperance_df = pd.DataFrame(esperance_list, index=prices.index[window
        :], columns=['Esperance'])
    volatilite_df = pd.DataFrame(volatilite_list, index=prices.index[
        window:], columns=['Volatilite'])

    return esperance_df, volatilite_df
```

This approach leverages the Kernel Density Estimation (KDE) to calculate the expectation and volatility, which are then used to inform buy or sell decisions. This function iterates over the price data, using each window of prices to calculate the returns. For each window, it computes the returns' percentage change and drops any missing values. If sufficient data is available, the function calculates the kernel density estimate for the returns over a range of values, determined by the returns' minimum and maximum values. The bandwidth for KDE is determined using a method implied to be scott_bandwidth, which adapts the bandwidth to the data's variability. The expectation is calculated as a weighted average of the values, and the volatility is computed as the square root of the weighted variance of these values. These calculated values are stored in lists and eventually returned as Pandas DataFrames, providing a time series of expected returns and volatility.

By calculating expected returns and volatility over a window of n dates, we can assume that, statistically, the next values of the asset will follow what is expected. So, if the expected return is positive, we can expect the price to rise, and vice versa. Moreover, using a threshold for this expectation enables us to filter out weak signals or market noise, and focus on opportunities with a higher probability of profit. in addition to this first criterion, we can consider the volatility, which allows us to detect sharp market movements (representing uncertainty). This can signal both significant

opportunities and risks.

We can thus imagine a dynamic allocation strategy for a portfolio based on these signals. This strategy seeks to maximize gains in favorable market conditions (by buying stocks when expectations are positive and volatility high), while limiting losses in unfavorable conditions (by liquidating positions when expectations are negative and volatility high). By choosing not to act when market conditions are not clearly defined, it also makes it possible to maintain a safe position, avoiding hasty reactions to false alarms or minor market movements. It's a balanced approach that takes into account both earnings potential and risk management.

To set up this strategy, we have made the **strategie_esperance_volatilite()** function, which utilizes the expectation and volatility data to generate trading signals. It first calculates these metrics over the specified window by calling the **calcule_esperance_volatilite()** function. It then initializes a DataFrame to store the trading signals, defaulting to 0, indicating no action. The function iterates over the price data beyond the initial window period, checking the expectation and volatility at each point. A buy signal (1) is generated if the expectation of returns is positive and the volatility exceeds a specified threshold, indicating confidence in upward movement with significant price variation. Conversely, a sell signal (-1) is generated if the expectation of returns is slightly negative (beyond a minimal threshold) and the volatility is above the threshold, suggesting a cautious approach to downward-trending markets with high variability. The trading signals are compiled into a DataFrame.

This strategy is particularly notable for its dynamic approach to trading, as it adapts to changing market conditions by continuously recalculating expectation and volatility. The use of KDE allows for a nuanced understanding of the returns distribution, potentially offering more informed trading decisions than simpler average-based approaches. However, the effectiveness of this strategy would depend on the appropriateness of the chosen window size, volatility threshold, and the accuracy of the KDE implementation in capturing the underlying distribution of returns.

The next step is to explore different combinations of time periods (window) and volatility thresholds (vol_threshold), with the aim of identifying the best configuration that maximizes the portfolio's final value. Then, we use these parameters in the **strategie_esperance_volatilite()** function to have the signal of buying, selling or no action to the next day. By using the **Backtest** class, we obtain this backtest for the period of October 2022 to October 2023 on the Nas100 :

Python code

```
Strategy - Max Drawdown: -8.70%, Volatility: 15.32%, Total Return: 27.95%
Benchmark - Max Drawdown: -11.05%, Volatility: 21.04%, Total Return:
    18.86%
```

Despite that we have a better max drawdown, volatility and total return, the strategy seems to follow the underlying even when there are huge drawdown. It's like we can't isolate trends. This led us to add indicators to the strategy in order to

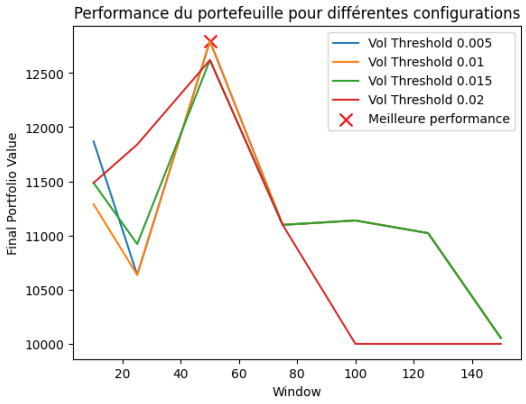

Figure 20: Best configuration: Window = 50.0, VolThreshold = 0.005

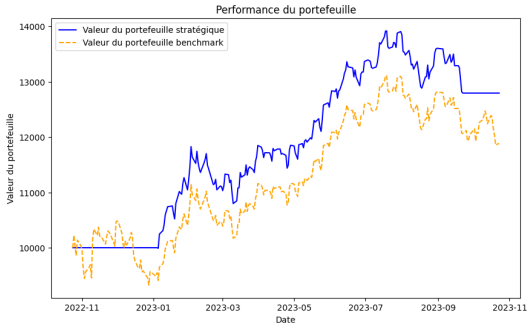

Figure 21: Strategy based on Expected vs Benchmark

capture asset trends.

6.3 Predictive model with several statistics derived from the estimated density

6.3.1 Crossing moving averages of kernel densities

To improve on the previous strategy, we'll be using the moving average crossover method. This strategy is popular in technical analysis as it attempts to capture the moment when a short-term trend deviates significantly from a long-term trend, thus offering buying or selling opportunities based on forecasts of changes in market direction. So we're going to try to protect ourselves from downturns by detecting downtrends.

Python code

```
def strategie_croisement_moyennes_mobiles(prices, window, vol_threshold,
    short_window, long_window):
    esperance, _ = calcule_esperance_volatilite(prices, window)

    esperance_short_ma = esperance['Esperance'].rolling(window=short_
        window, min_periods=1).mean()
    esperance_long_ma = esperance['Esperance'].rolling(window=long_window
        , min_periods=1).mean()

    signal = pd.DataFrame(index=prices.index, data=0, columns=['Signal'])

    valid_indices = esperance_short_ma.dropna().index.intersection(
        esperance_long_ma.dropna().index)

    for i in valid_indices:
        current_idx = prices.index.get_loc(i)
        if current_idx > 0:
            prev_idx = prices.index[current_idx - 1]
            if esperance_short_ma.loc[i] > esperance_long_ma.loc[i] and
                esperance_short_ma.loc[prev_idx] <= esperance_long_ma.loc[
                prev_idx]:
                signal.at[i, 'Signal'] = 1
```

```
            elif esperance_short_ma.loc[i] < esperance_long_ma.loc[i] and
                 esperance_short_ma.loc[prev_idx] >= esperance_long_ma.loc
                [prev_idx]:
                 signal.at[i, 'Signal'] = -1

    return signal
```

This function begins by calculating the expectation of returns over a specified time window for the prices provided. This operation uses an external function **calcule_esperance_volatilite()**, which, in addition to the expectation, also calculates the volatility, but only the expectation is retained for this strategy.

Two moving averages of expected returns are then calculated: one short (**esperance_short_ma**) and one long (**esperance_long_ma**). The short moving average uses a smaller number of periods (short_window), making it more responsive to recent price changes. The long moving average, calculated over a larger number of periods (**long_window**), is slower and reflects trends over a longer period.

A DataFrame named signal is initialized to record trading signals. By default, all values are initialized to 0, indicating the absence of a signal. Then we fill the DataFrame with :

- **1** : Buy signal, when the short moving average crosses above the long moving average, indicating a potential upward trend reversal. To avoid false alarms, the function ensures that the short moving average was below or equal to the long moving average at the previous index.

- **-1** : Conversely, sell signal, when the short moving average crosses below the long moving average, suggesting a possible downward trend reversal. The condition that the short moving average was greater than or equal to the long moving average on the previous index is also verified to confirm the signal.

We applied a backtest of this strategy on few assets and we obtained these results :

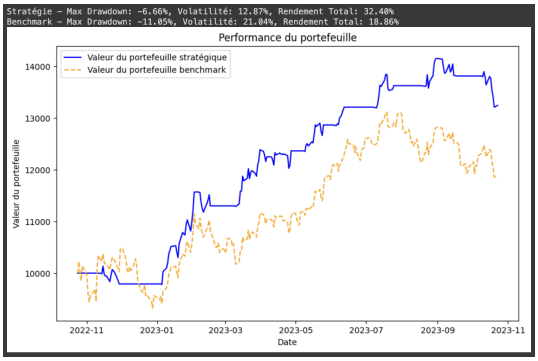

Figure 22: MA Strategy vs Benchmark with NAS100

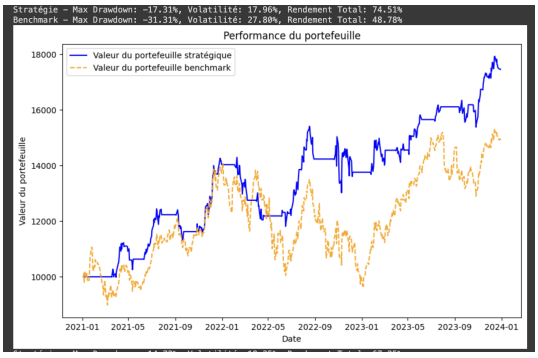

Figure 23: MA Strategy vs Benchmark with AAPL (Apple)

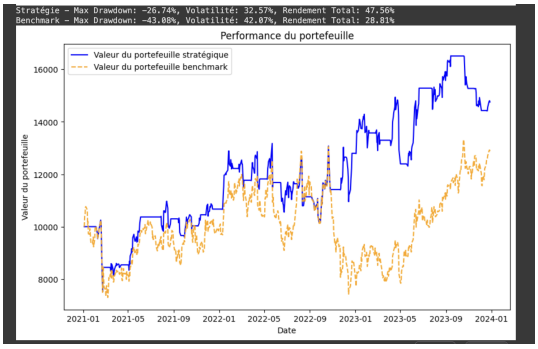

Figure 24: MA Strategy vs Benchmark with 0700.HK (Tencent)

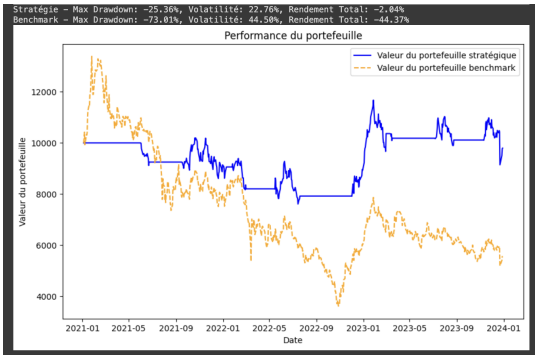

Figure 25: MA Strategy vs Benchmark with MC.PA (LVMH)

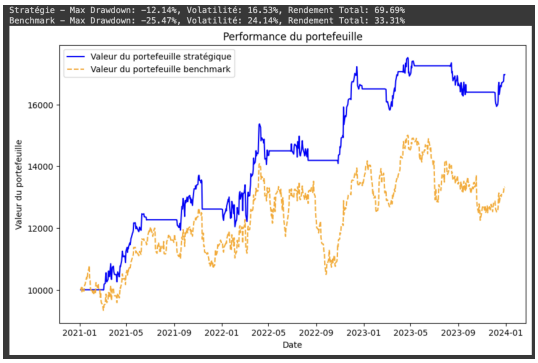

Figure 26: MA Strategy vs Benchmark with PETR4.S (Petroleo)

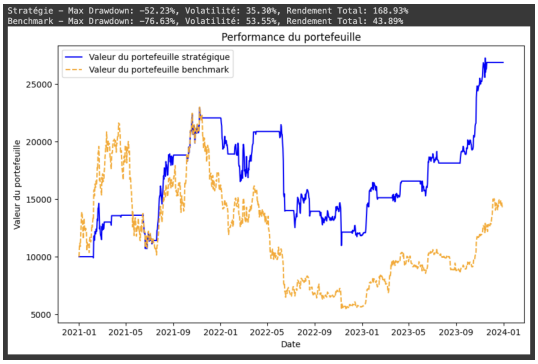

Figure 27: MA Strategy vs Benchmark with BTC

We have a strategy that seems suited to all kinds of assets and, in all the above cases, enables us to reduce max drawdown and volatility while maximizing total return. We can clearly see that this technique of crossing moving averages allows us to dissociate the different trends and thus avoid falls in asset prices, while exposing ourselves to rises. To go a step further in assessing the performance of this strategy, we can plot the density of returns, allowing us to analyze the shape of the distribution, including skewness and kurtosis. This helps identify whether returns tend to be positive or negative, and whether the strategy tends to produce extreme results.

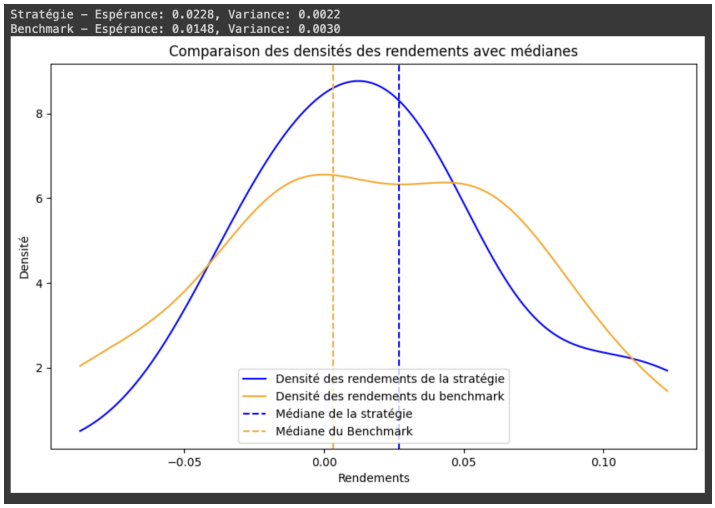

Figure 28: Estimated density of monthly return of the strategy vs benchmark one's

The shape of the yield density reveals the inherent risk of the strategy. Here, the distribution of the strategy has shorter tails indicating a lower risk of having extreme returns (positive or negative), and highlights a potential resistance to market shocks. Compared to the benchmark, this reveals that the strategy mitigates these risks. In addition, the distribution of the strategy's returns at a higher median (we will tend to outperform the benchmark more often) and is slightly narrower (less volatility) means that we have more stable returns than the benchmark

In order to see more generally the result of the strategy, this graph allows us to see that over the long term, we will tend to outperform the market :

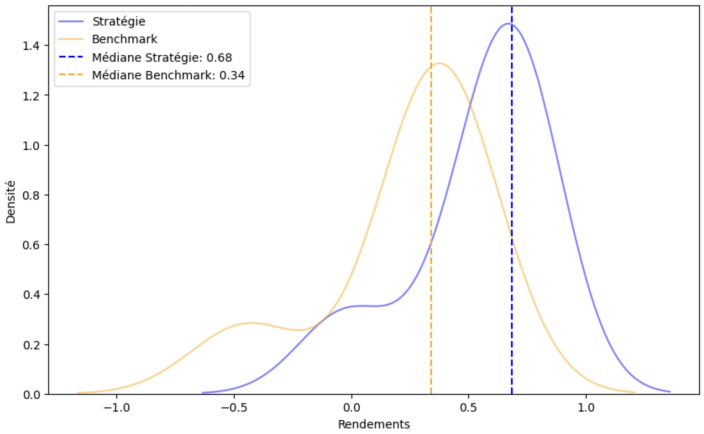

Figure 29: The yield density of the strategy on multiple assets vs benchmark one's

To conclude on this strategy, it can be stated that we can maximise the return while reducing the risk exposure by detecting bearish periods (the consequences are the decreasing of the max drowdown and the volatily).

However, we might wonder what it would be like if, instead of remaining passive during market downturns, we couldn't take advantage of them.

6.3.2 Crossing moving averages of kernel densities with shorts positions

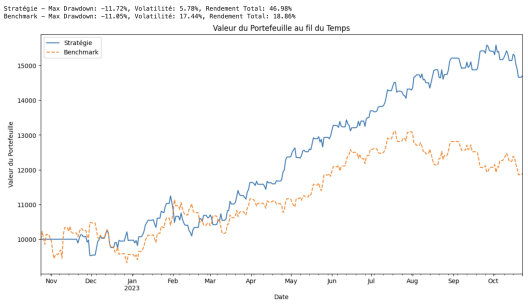

Figure 30: MA Strategy with shorts vs Benchmark with NAS100

Figure 31: MA Strategy with shorts vs Benchmark with 0700.HK (Tencent)

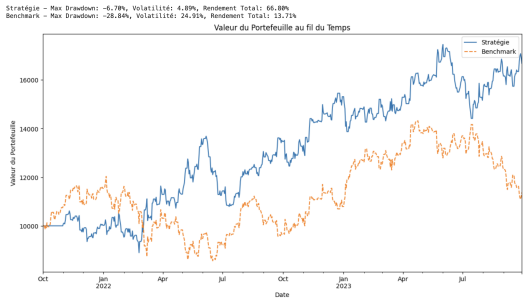

Figure 32: MA Strategy with shorts vs Benchmark with MC.PA (LVMH)

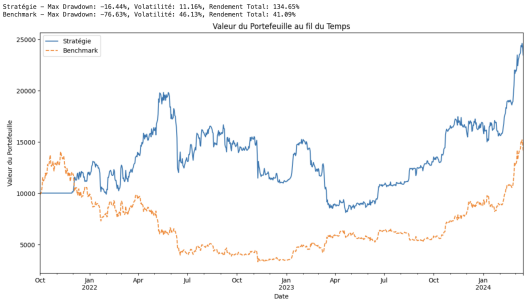

Figure 33: MA Strategy with shorts vs Benchmark with BTC

Here, the strategy consisted in buying assets when the short moving average of the expectation crossed above the long moving average, but when the average crossed in the other direction, not only did we liquidate our position if we had one, but we also exposed ourselves to the downside with a short position.

Once again, the strategy tends to outperform the benchmark index, while reducing volatility and large downward movements.

We wanted to test the process on relatively explosive assets. It seems to react well in some cases, such as the FTX token (FTT).

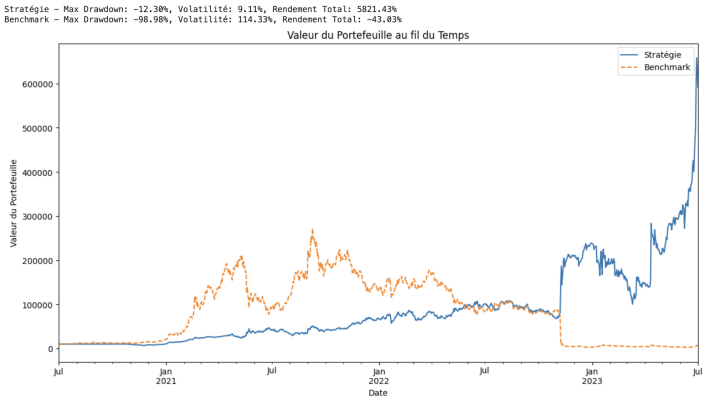

Figure 34: MA Strategy with shorts vs Benchmark with FTT (FTX Token)

However, in the case of the GameStop share, we noticed that it failed to take advantage of violent market variations.

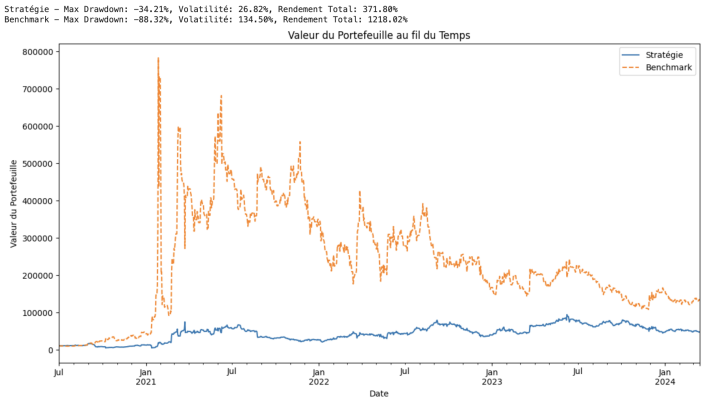

Figure 35: MA Strategy with shorts vs Benchmark with GME (GameStop)

We can therefore retain these two strategies as solutions for dynamically allocating our portfolios (Moving Average Crossover Strategy) or for trading assets (Moving Average Crossover Strategy with short positions).

7 Conclusion

The project was structured around several main axes, ranging from non-parametric density estimation to concrete application in option trading and hedging strategies. By adopting various approaches to free parameter selection, such as the use of MCMCs, maximizing predictive power via PITs, and maximizing complexity, this research has not only enriched the statistical toolbox available to financial analysts, but has also proposed new avenues for risk management and return optimization.

The application of these methods to the forecasting of option returns and the construction of systematic strategies has demonstrated their potential, offering valuable insights for the valuation of options and the development of trading algorithms based on robust historical data. The ability to quantify expected return and VaR for options, and to integrate this information into concrete trading strategies, illustrates the practical relevance of the tool developed.

In conclusion, this project has taken an important step forward in the quest for a more nuanced understanding and more effective application of probability laws in finance. It has not only contributed to academic progress in the field of non-parametric density estimation, but has also opened up new perspectives for finance professionals seeking more accurate and adaptive tools for risk management and return optimization.

As we continue to explore the implications of these findings, future research may focus on integrating these advanced techniques with real-time data analytics and machine learning algorithms. Such integration has the potential to further revolutionize our approach to financial modeling and risk assessment, ensuring that our strategies remain robust in the face of ever-evolving market dynamics. This pursuit opens a gateway to an era of precision finance, where data-driven insights could lead to more resilient investment frameworks and a deeper grasp of the complex forces that shape our financial systems.

We would like to thanks Matthieu Garcin for supervising us along this passionating subject, and to Alexis Bogroff for giving us feedback and guided us all year into the final render.

8 Bibliographie

References

[1] Silverman, B. W. (1986). *Density estimation for statistics and data analysis*. CRC press.

[2] Jones, M. C., Marron, J. S., & Sheather, S. J. (1996). A brief survey of bandwidth selection for density estimation. *Journal of the American statistical association*, 91(433), 401-407.

[3] Tsybakov, A. B. (2003). *Introduction à l'estimation non paramétrique* (Vol. 41). Springer Science & Business Media.

[4] Zhang, X., Hyndman, R. J., & King, M. L. (2004). Bandwidth selection for multivariate kernel density estimation using MCMC (No. 9/04). Monash University, Department of Econometrics and Business Statistics.

[5] Harvey, A., & Oryshchenko, V. (2012). Kernel density estimation for time series data. *International journal of forecasting*, 28(1), 3-14.

[6] Garcin, M., Klein, J., & Laaribi, S. (2020). Estimation of time-varying kernel densities and chronology of the impact of COVID-19 on financial markets. *arXiv preprint*.

[7] Garcin, M. (2023). Complexity measure, kernel density estimation, bandwidth selection, and the efficient market hypothesis. *arXiv preprint*.

[8] Hyndman, R. J., Bashtannyk, D. M., & Grunwald, G. K. (1996). Estimating and visualizing conditional densities. *Journal of Computational and Graphical Statistics*, 5(4), 315-336.

[9] Bashtannyk, D. M., & Hyndman, R. J. (2001). Bandwidth selection for kernel conditional density estimation. *Computational Statistics & Data Analysis*, 36(3), 279-298.

[10] Cao, J., Chen, J., Farghadani, S., Hull, J., Poulos, Z., Wang, Z., & Yuan, J. (2023). Gamma and vega hedging using deep distributional reinforcement learning. *Frontiers in Artificial Intelligence*, 6, 1129370.

[11] Israelov, Roni and Kelly, Bryan T., Forecasting the Distribution of Option Returns (September 7, 2017).