HM2

Introduction

Conformal inference (CI) is a statistical framework that provides a way to quantify uncertainty in predictions made by machine learning models. By leveraging past data, it constructs prediction intervals that are guaranteed to contain the true outcome with a specified probability

As the popularity of the machine learning and black box methods and the cost of making wrong decisions grow, it is crucial to develope tools to quantify the uncertainty of their prediction.
In the paper “Adaptive Conformal Inference”, Isaac Gibbs and Emmanuel J. Candes have developed methods for constructing these prediction sets. one thing that worths mentioning is that in previous methodologies, the algorithms share the same common guarantee that “if the training and test data are exchangable”. while exchangability is a common assumption, there are many real-world applications in which we do not expect the marginal distibution of (X_t, Y_t) to be stationary. for example in finance and economics, market behaviour can shift drastically in response to new legislation for major world events. but this paper develops “Adaptive Conformal Inference (ACI)” to form prediction sets that are robust to changes in the marginal distribution of the data.
now with these explainations, we delve deeper into the method and the different approaches for predictions between CI and ACI.

1.Conformal Inference

suppose we train a machine learning model $f(X)$ for predicting the value of Y on the data set $(X_1, Y_1), (X_2, Y_2) \ldots (X_n, Y_n)$. the model predicts $\hat{y_i} = f(X_i)$ for each training point. now we define conformity score $S(X, Y)$ that measures how well the value y alignes with the predictions of our fitted model.

$$S(X, Y) = |\hat{y} - y|$$

To determine if $\hat{y}$is a reasonable prediction, a calibration set (separate from the training data) is used to compute quantiles of conformity scores, $Q(p)$, ensuring that $S(X_t, y) \leq Q(1-\alpha)$ guarantees the desired coverage probability $1-\alpha$. Under the assumption of exchangeability, this method provides marginal coverage, with the set of reasonable predictions
$C_t = y : S(X_t, y) \leq Q(1-\alpha)$
the method oulined above is often referred to as “split conformal inference”.This refers to the fact that we have split the observed data between a training set used to fit the regression model and a withheld calibration set. The adaptive conformal inference method developed in this article can also be easily adjusted to work with full conformal inference in which data splitting is avoided at the cost of greater computational resources.

2.Adapting conformal inference to distribution shifts**

before delving deep into the adaptive conformal inference, we shall provide further definitions:

1.conformity scores $S(.)$:

The conformity score $S(X, Y)$ quantifies how well a candidate value Y aligns with the predictions of a fitted model given the input X.

Examples:

Absolute Residuals: For a regression model that outputs a point prediction $\hat{\mu}(X)$:

$$S(X, Y) : |Y - \hat{\mu}(X)|$$

Quantile-Based Score: For quantile regression models estimating the $\frac{\alpha}{2}$ and $1 - \frac{\alpha}{2}$ quantiles:

$$S(X, Y) = \max \left\{ q(X; \frac{\alpha}{2}) - Y , Y - q(X; 1 - \frac{\alpha}{2}) \right\}$$

2. quantile function $Q(.)$:

The quantile function $\hat{Q(p)}$ determines the threshold for conformity scores based on a specified probability p. It is derived from the empirical distribution of the conformity scores computed on a calibration dataset.

$$\hat{Q} = \inf \left\{ s : \left(\frac{1}{D_{cal}} \sum_{(X_r, Y_r) \in D_{cal}} I_{S(X_r, Y_r) \leq s} \right) \geq p \right\}$$

The quantile function determines the cutoff value for $S(X, Y$ such that a specified fraction (p) of the calibration scores fall below this value.

3. realized miscoverage rate:

The realized miscoverage rate quantifies how often a prediction set fails to contain the true value of the response variable Y_t at a specific time t. Mathematically, it is defined as:

$$M_t(\alpha) := P(S_t(X_t, Y_t)) > \hat{Q_t}(1 - \alpha))$$

In the general case where the distribution of the data is shifting over time both $S(.)$ and $\hat{Q(.)}$ functions should be regularly re-estimated to align with the most recent observations. Since the data distribution changes, $M_t(\alpha)$ may not match α as expected. However, there exists an alternative parameter $\alpha_t^* \in [0, 1]$ such that $M_t(\alpha_t^*) \approx \alpha$. To ensure that $\hat{Q(.)}$ is continuous, it can be adjusted to remove jump discontinuities, making it non-decreasing and bounded between $-\infty$ and ∞. With these adjustments, $M(\cdot)$ becomes non-decreasing, enabling the definition of α_t^* as:

$$\alpha_t^* = \sup \{\beta \in [0, 1] : M_t(\beta) \leq \alpha\}$$

more over if we additionally assume that

$$P(S_t(X_t, Y_t) = \hat{Q_t}(1 - \alpha_t^*)) = 0$$

then we will see that $M_t(\alpha_t^*) = \alpha$. So, in particular we find that by correctly calibrating the argument to $\hat{Q(.)}$ we can achieve either approximate or exact marginal coverage.

To dynamically update α_t^*, the algorithm evaluates whether recent prediction sets under- or

over-cover the true values Y_t.In particular, let α_1 denote our initial estimate (in our experiments we will choose $\alpha_1 = \alpha$). Recursively define the sequence of miscoverage events:

$$err_t := \begin{cases} 1 & \text{if } Y_t \notin \hat{C}(\alpha_t) \\ 0 & \text{otherwise} \end{cases}$$

where:

$$\hat{C_t}(\alpha_t) := \left\{ y : S_t(X_t, y) \leq \hat{Q_t}(1 - \alpha_t) \right\}$$

A simple online update rule adjusts α_t based on observed errors:

$$\alpha_{t+1} := \alpha_t + \gamma(\alpha - err_t)$$

Another method for updating the α that we use is as following:

$$\alpha_{t+1} = \alpha_t + \gamma(\alpha - \sum_{s=1}^{t} \omega_s err_s)$$

The choice of γ gives a tradeoff between adaptability and stability. While raising the value ofγ will make the method more adaptive to observed distribution shifts, it will also induce greater volatility in he value of α_t. In practice, large fluctuations in α_t may be undesirable as it allows the method to oscillate between outputting small conservative and large anti-conservative prediction sets. in environments with greater distributional shift the algorithm needs to be more adapatable and thus γ should be chosen to be larger. In our experiments we will take $\gamma = 0.005$. This value was chosen because it was found to give relatively stable trajectories for α_t while still being sufficiently large as to allow α_t to adapt to observed shifts. In agreement with the general principles outlined above we found that larger values of γ also successfully protect against distribution shifts, while taking α_t to be too small causes adaptive conformal inference to perform similar to non-adaptive methods that hold $\alpha_t = \alpha$ constant across time.

Implementation of the ACI model on Microsoft

In the following, we will implement the ACI method along with the Non Adaptive conformal prediction and compare the results.

```
library(quantmod)
library(quantreg)
library(rugarch)
library(tidyverse)
### Volatility prediction
garchConformalForcasting <- function(returns,alpha = 0.05,gamma = 0.001,lo
okback=1250,garchP=1,garchQ=1,startUp = 100,verbose=FALSE,updateMethod="Mo
mentum",momentumBW = 0.95){
  myT <- length(returns)
  T0 <- max(startUp,lookback)
  garchSpec <- ugarchspec(mean.model=list(armaOrder = c(0, 0),include.mean
=FALSE),variance.model=list(model="sGARCH",garchOrder=c(garchP,garchQ)),di
stribution.model="norm")
  simple_alphat <- alpha
  normalized_alphat <- alpha
  ### Initialize data storage variables
  nonAdaptive_simple_errSeqNC <- rep(0,myT-T0+1)
  nonAdaptive_normalized_errSeqNC <- rep(0,myT-T0+1)

  adaptive_simple_errSeqOC <- rep(0,myT-T0+1)
  adaptive_normalized_errSeqOC <- rep(0,myT-T0+1)



  simple_alphaSequence <- rep(alpha,myT-T0+1)
  normalized_alphaSequence <- rep(alpha,myT-T0+1)
  simple_scores <- rep(0,myT-T0+1)
  normalized_scores <- rep(0,myT-T0+1)



  for(t in T0:myT){
    if(verbose){
      print(t)
    }
    ### Fit garch model and compute new conformity score
    garchFit <- ugarchfit(garchSpec, returns[(t-lookback+1):(t-1) ],solver
="hybrid")
    sigmaNext <- sigma(ugarchforecast(garchFit,n.ahead=1))

    #calculate scores
    simple_scores[t-T0 + 1]<- abs(returns[t]^2- sigmaNext^2)
    normalized_scores[t-T0 + 1] <- abs(returns[t]^2- sigmaNext^2)/sigmaNex
t^2

    #calculate recent score
    simple_recentScores <- simple_scores[max(t-T0+1 - lookback + 1,1):(t-T
```

```
0)]
    normalized_recentScores <- normalized_scores[max(t-T0+1 - lookback +
1,1):(t-T0)]

    #simple and normalized non adaptive errors
    nonAdaptive_simple_errSeqNC[t-T0+1] <- as.numeric(simple_scores[t-T0 +
1] > quantile(simple_recentScores,1-alpha))
    nonAdaptive_normalized_errSeqNC[t-T0+1] <- as.numeric(normalized_score
s[t-T0 + 1] > quantile(normalized_recentScores,1-alpha))

    #simple and normalized adaptive errors
    adaptive_simple_errSeqOC[t-T0+1] <- as.numeric(simple_scores[t-T0 + 1]
> quantile(simple_recentScores,1-simple_alphat))
    adaptive_normalized_errSeqOC[t-T0+1] <- as.numeric(normalized_scores[t
-T0 + 1] > quantile(normalized_recentScores,1-normalized_alphat))

    ### update alphat
    simple_alphaSequence[t-T0+1] <- simple_alphat
    normalized_alphaSequence[t-T0+1] <- normalized_alphat
    if(updateMethod=="Simple"){
      simple_alphat <- simple_alphat + gamma*(alpha - adaptive_simple_errS
eqOC[t-T0+1])
      normalized_alphat <- normalized_alphat + gamma*(alpha - adaptive_nor
malized_errSeqOC[t-T0+1])
    }else if(updateMethod=="Momentum"){
      w <- rev(momentumBW^(1:(t-T0+1)))
      w <- w/sum(w)
      simple_alphat <- simple_alphat + gamma*(alpha - sum(adaptive_simple_
errSeqOC[1:(t-T0+1)]*w))
      normalized_alphat <- normalized_alphat + gamma*(alpha - sum(adaptive
_normalized_errSeqOC[1:(t-T0+1)]*w))
    }
    if(t %% 100 == 0){
      print(sprintf("Done %g steps",t))
    }
  }

  length_nonAdaptive_simple_errSeqNC <- length(nonAdaptive_simple_errSeqN
C)
  length_nonAdaptive_normalized_errSeqNC <- length(nonAdaptive_normalized_
errSeqNC)

  length_adaptive_simple_errSeqOC <- length(adaptive_simple_errSeqOC)
  length_adaptive_normalized_errSeqOC <- length(adaptive_normalized_errSeq
OC)

  #define non adaptive localCov variables
  simple_localCov_alpha <- rep(NA,length_nonAdaptive_simple_errSeqNC -500+
```

```
1)
  normalized_localCov_alpha <- rep(NA,length_nonAdaptive_simple_errSeqNC -
500+1)

  #define adaptive localCov variables
  simple_localCov_alphat <- rep(NA,length_adaptive_simple_errSeqOC -500+1)
  normalized_localCov_alphat <- rep(NA,length_adaptive_simple_errSeqOC -50
0+1)

  #calculate the non adaptive local Cov level
  for(t in 251:(length_nonAdaptive_simple_errSeqNC - 250 +1)){
    simple_localCov_alpha[t-250] <- 1 - mean(nonAdaptive_simple_errSeqNC
[(t-250):(t+250-1)])
  }
  for(t in 251:(length_nonAdaptive_normalized_errSeqNC - 250 +1)){
    normalized_localCov_alpha[t-250] <- 1 - mean(nonAdaptive_normalized_er
rSeqNC[(t-250):(t+250-1)])
  }

  #calculate the adaptive local Cov level
  for(t in 251:(length_adaptive_simple_errSeqOC - 250 +1)){
    simple_localCov_alphat[t-250] <- 1 - mean(adaptive_simple_errSeqOC[(t-
250):(t+250-1)])
  }

  for(t in 251:(length_adaptive_normalized_errSeqOC - 250 +1)){
    normalized_localCov_alphat[t-250] <- 1 - mean(adaptive_normalized_errS
eqOC[(t-250):(t+250-1)])
  }



  return(list(simple_localCov_alpha, normalized_localCov_alpha, simple_loc
alCov_alphat,normalized_localCov_alphat))
}
```

The code above includes a function which calculates the local coverage level using the simple and normalized scores for both CP and ACP. in the following we print the result graphs with different update methods and adaption rates:

first we import needed packages and download the data using the quantmod package and calculate the simple returns

```
getSymbols("MSFT", from = "2014-01-01", to = "2023-12-31")
```

```
## [1] "MSFT"
```

```
msft <- Cl(MSFT)
returns <- dailyReturn(msft)
returns <- na.omit(returns)  # Rimuove eventuali NA
```

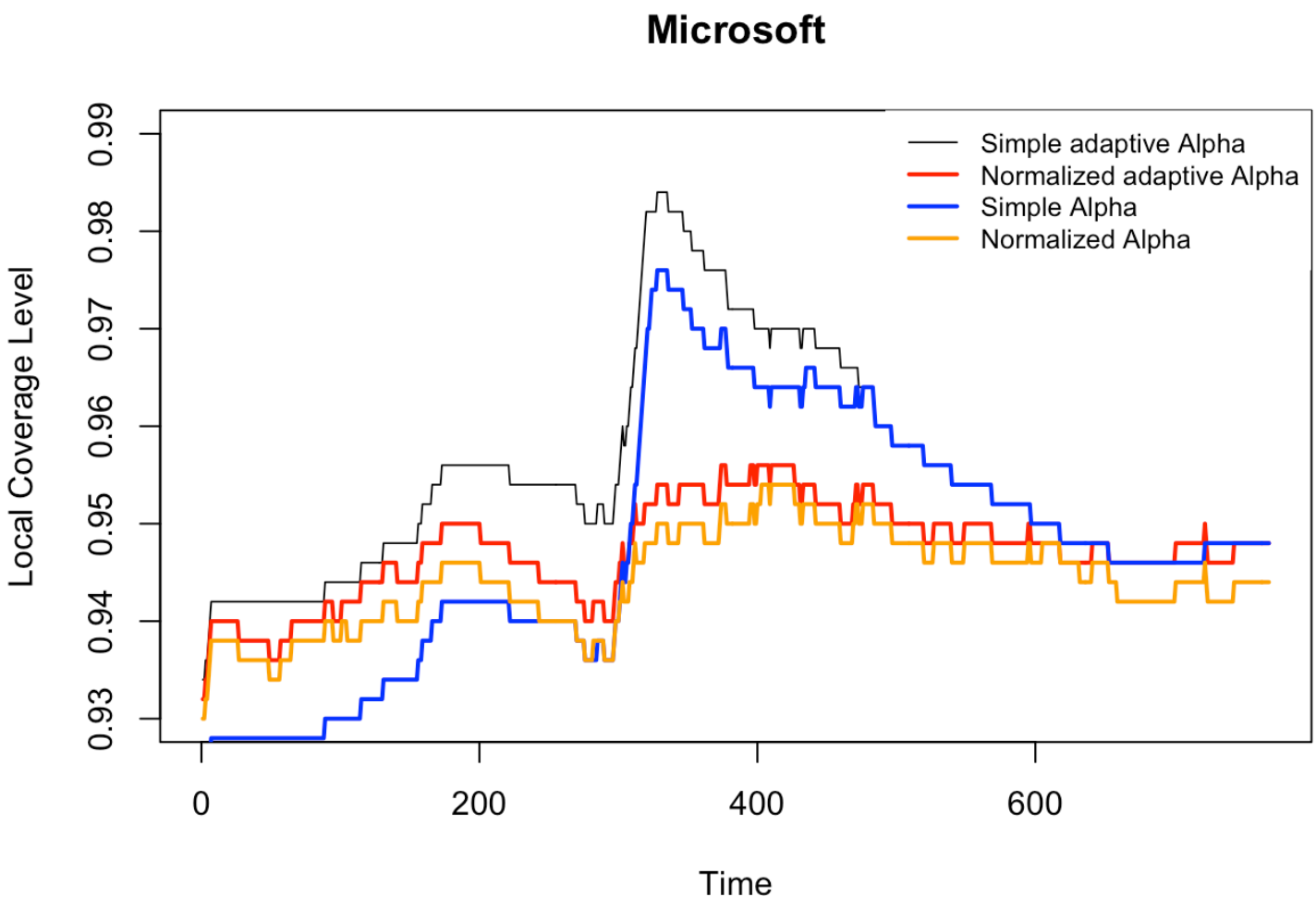
1) In the first series of analysis we evaluate the impact of $\gamma = 0.001$ on the coverage rate:

```
results1 <- garchConformalForcasting(returns = returns, alpha = 0.05, gamm
a = 0.001,lookback=1250,garchP=1,garchQ=1,startUp = 100,verbose=FALSE,upda
teMethod="Simple")
```

```
## [1] "Done 1300 steps"
## [1] "Done 1400 steps"
## [1] "Done 1500 steps"
## [1] "Done 1600 steps"
## [1] "Done 1700 steps"
## [1] "Done 1800 steps"
## [1] "Done 1900 steps"
## [1] "Done 2000 steps"
## [1] "Done 2100 steps"
## [1] "Done 2200 steps"
## [1] "Done 2300 steps"
## [1] "Done 2400 steps"
## [1] "Done 2500 steps"
```

```
#date <- index(msft)[1250:length(index(msft))]
simple_localCov_alpha1 <- results1[[1]]
normalized_localCov_alpha1 <- results1[[2]]
simple_localCov_alphat1 <- results1[[3]]
normalized_localCov_alphat1 <- results1[[4]]
```

```
# Plot the first series (localCov_alphat) in black
plot(simple_localCov_alphat1, type = "l", col = "black", main = "Microsof
t", xlab = "Time", ylab = "Local Coverage Level",
     ylim = c(0.93,0.99) , lwd = 0.9)
# Add the second series (localCov_alpha) in blue
lines(normalized_localCov_alphat1, col = "red", lwd = 2)
lines(simple_localCov_alpha1, col = "blue", lwd = 2)
lines(normalized_localCov_alpha1, col = "orange", lwd = 2)
legend("topright",
       legend = c("Simple adaptive Alpha ",
                  "Normalized adaptive Alpha",
                  "Simple Alpha",
                  "Normalized Alpha"),
       col = c("black", "red", "blue", "orange"),
       lty = 1,
       lwd = c(0.9, 2, 2, 2),
       cex = 0.8,
       box.lty = 0)
```

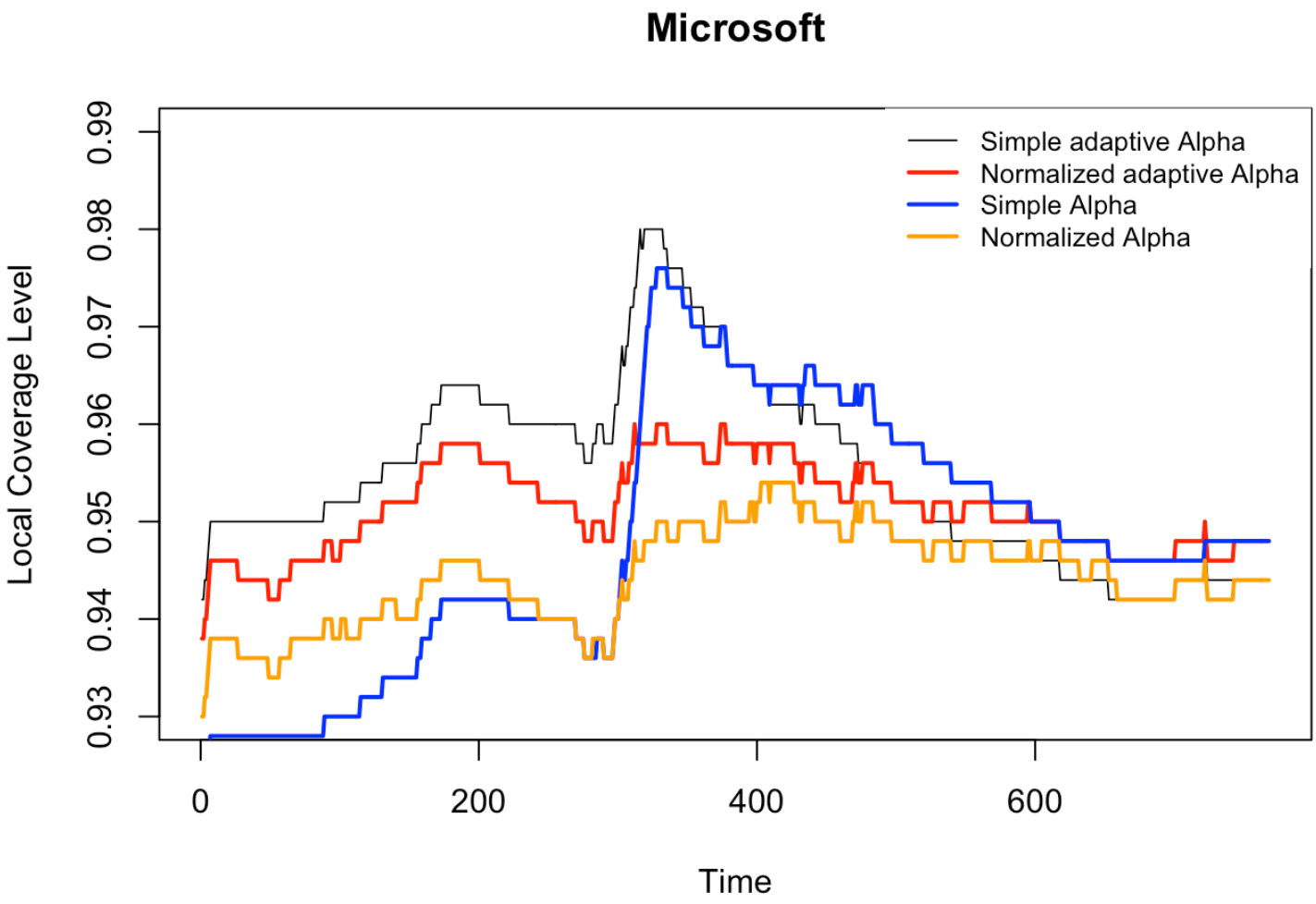


2)

another analysis is done using a $\gamma = 0.003$:

```
## [1] "Done 1300 steps"
## [1] "Done 1400 steps"
## [1] "Done 1500 steps"
## [1] "Done 1600 steps"
## [1] "Done 1700 steps"
## [1] "Done 1800 steps"
## [1] "Done 1900 steps"
## [1] "Done 2000 steps"
## [1] "Done 2100 steps"
## [1] "Done 2200 steps"
## [1] "Done 2300 steps"
## [1] "Done 2400 steps"
## [1] "Done 2500 steps"
```

```
# Plot the first series (localCov_alphat) in black
plot(simple_localCov_alphat2, type = "l", col = "black", main = "Microsof
t", xlab = "Time", ylab = "Local Coverage Level",
     ylim = c(0.93,0.99) , lwd = 0.9)
# Add the second series (localCov_alpha) in blue
lines(normalized_localCov_alphat2, col = "red", lwd = 2)
lines(simple_localCov_alpha2, col = "blue", lwd = 2)
lines(normalized_localCov_alpha2, col = "orange", lwd = 2)
# Add a legend
legend("topright",
       legend = c("Simple adaptive Alpha ",
                  "Normalized adaptive Alpha",
                  "Simple Alpha",
                  "Normalized Alpha"),
       col = c("black", "red", "blue", "orange"),
       lty = 1,
       lwd = c(0.9, 2, 2, 2),
       cex = 0.8,
       box.lty = 0)
```



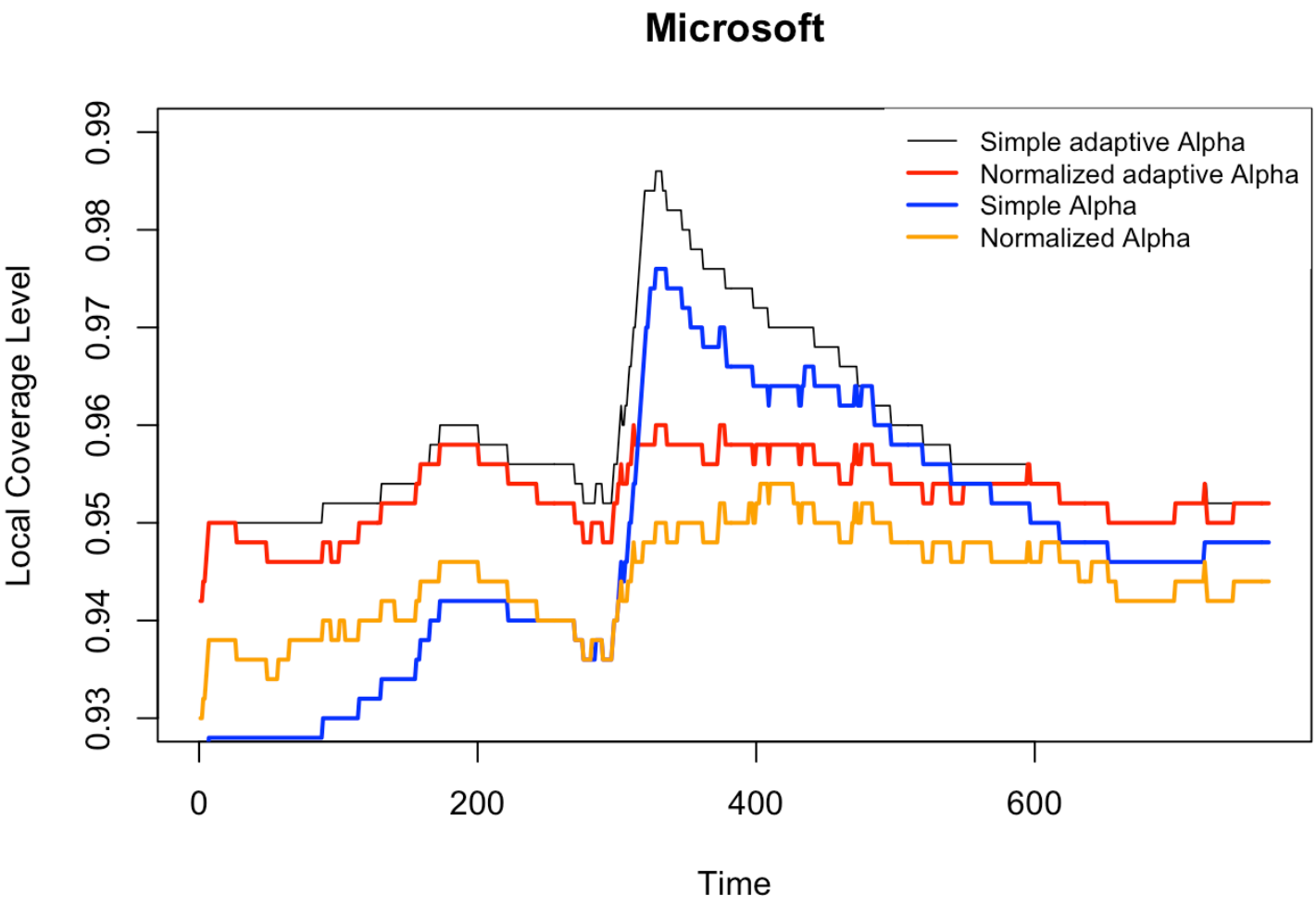
As we can see from both the plots above, the normalized adaptive alpha performes better for Microsoft. and also another aspect of these plots that worth mentioning is that as we change the value of $\gamma = 0.001$ to $\gamma = 0.003$, this increase results in a better performance.generally speaking, higher adaption rates makes the method more responsive to changes in the data distribution and we can say that in rapidly changing data distributions which the data distribution shift frequently (e.g., in our case the Microsoft data which is a highly volatile financial instrument), a higher adaptation rate allows the model to adjust quickly and maintain valid prediction intervals.

```
#analysing the coverage level with momentum update method
results4 <- garchConformalForcasting(returns = returns, alpha = 0.05, gamm
a = 0.003,lookback=1250,garchP=1,garchQ=1,startUp = 100,verbose=FALSE,upda
teMethod="Momentum")
```

```
## [1] "Done 1300 steps"
## [1] "Done 1400 steps"
## [1] "Done 1500 steps"
## [1] "Done 1600 steps"
## [1] "Done 1700 steps"
## [1] "Done 1800 steps"
## [1] "Done 1900 steps"
## [1] "Done 2000 steps"
## [1] "Done 2100 steps"
## [1] "Done 2200 steps"
## [1] "Done 2300 steps"
## [1] "Done 2400 steps"
## [1] "Done 2500 steps"
```

```
simple_localCov_alpha4 <- results4[[1]]
normalized_localCov_alpha4 <- results4[[2]]
simple_localCov_alphat4 <- results4[[3]]
normalized_localCov_alphat4 <- results4[[4]]
```

```
# Plot the first series (localCov_alphat) in black
plot(simple_localCov_alphat4, type = "l", col = "black", main = "Microsof
t", xlab = "Time", ylab = "Local Coverage Level",
     ylim = c(0.93,0.99) , lwd = 0.9)
# Add the second series (localCov_alpha) in blue
lines(normalized_localCov_alphat4, col = "red", lwd = 2)
lines(simple_localCov_alpha4, col = "blue", lwd = 2)
lines(normalized_localCov_alpha4, col = "orange", lwd = 2)
# Add a legend
legend("topright",
       legend = c("Simple adaptive Alpha ",
                  "Normalized adaptive Alpha",
                  "Simple Alpha",
                  "Normalized Alpha"),
       col = c("black", "red", "blue", "orange"),
       lty = 1,
       lwd = c(0.9, 2, 2, 2),
       cex = 0.8,
       box.lty = 0)
```



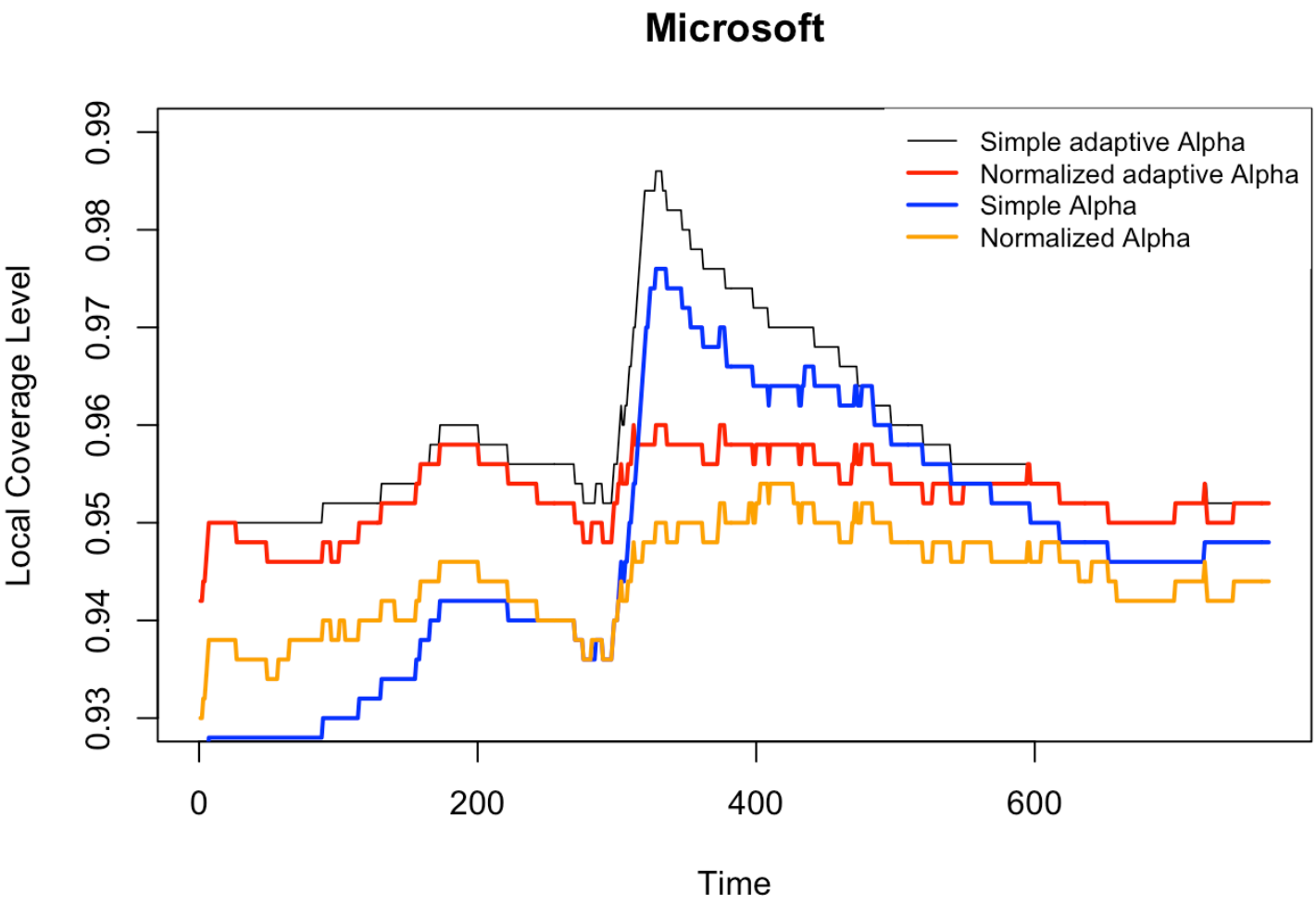
we see that also having $\gamma = 0.003$, the momentum method(eq.3) for normalized adaptive interval, has better and more stable local coverage

```
#analysing the coverage level with momentum update method
results5 <- garchConformalForcasting(returns = returns, alpha = 0.05, gamm
a = 0.001,lookback=1250,garchP=1,garchQ=1,startUp = 100,verbose=FALSE,upda
teMethod="Momentum")
```

```
## [1] "Done 1300 steps"
## [1] "Done 1400 steps"
## [1] "Done 1500 steps"
## [1] "Done 1600 steps"
## [1] "Done 1700 steps"
## [1] "Done 1800 steps"
## [1] "Done 1900 steps"
## [1] "Done 2000 steps"
## [1] "Done 2100 steps"
## [1] "Done 2200 steps"
## [1] "Done 2300 steps"
## [1] "Done 2400 steps"
## [1] "Done 2500 steps"
```

```
simple_localCov_alpha5 <- results4[[1]]
normalized_localCov_alpha5 <- results4[[2]]
simple_localCov_alphat5 <- results4[[3]]
normalized_localCov_alphat5 <- results4[[4]]
```

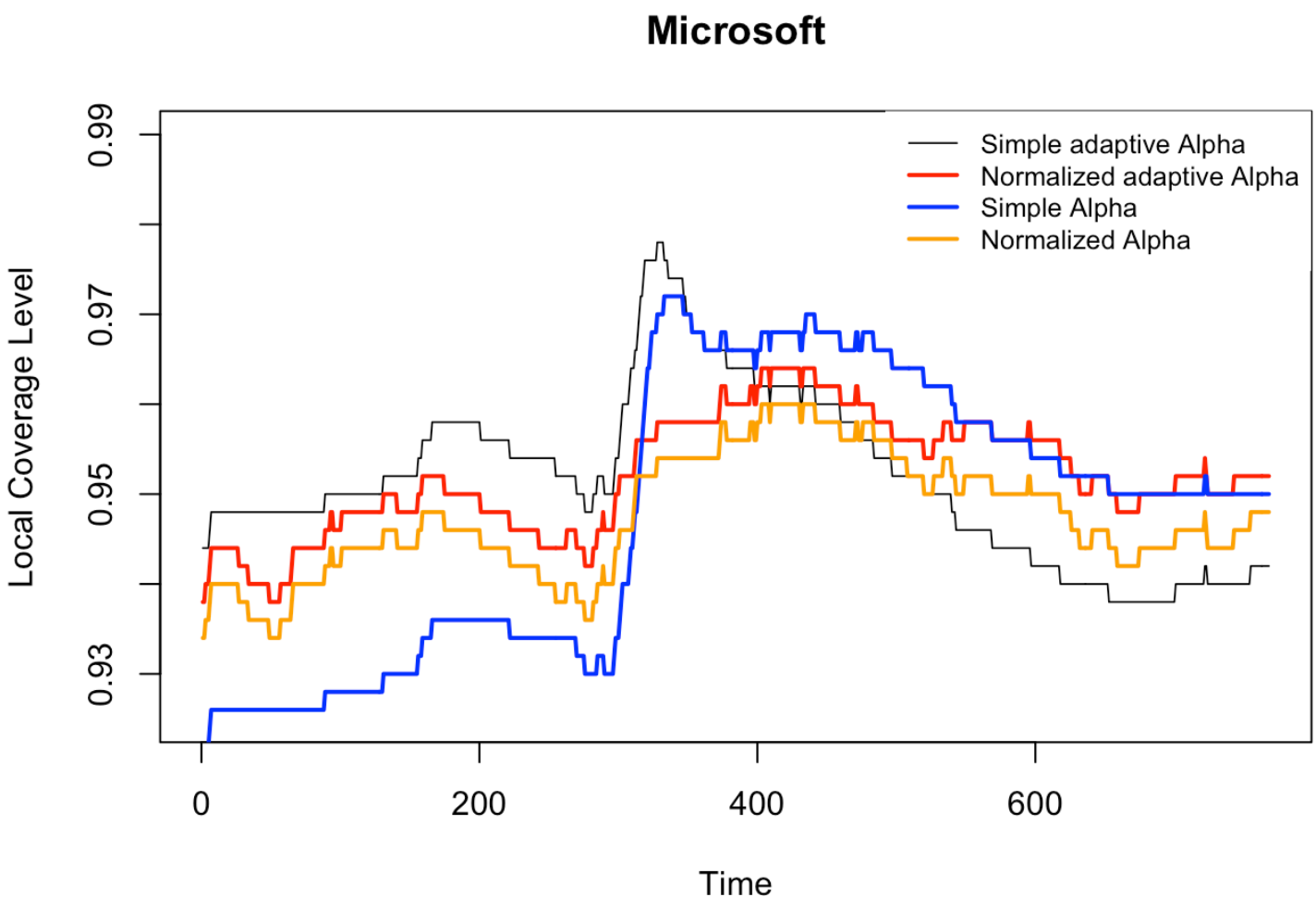
```
# Plot the first series (localCov_alphat) in black
plot(simple_localCov_alphat5, type = "l", col = "black", main = "Microsof
t", xlab = "Time", ylab = "Local Coverage Level",
     ylim = c(0.93,0.99) , lwd = 0.9)
# Add the second series (localCov_alpha) in blue
lines(normalized_localCov_alphat5, col = "red", lwd = 2)
lines(simple_localCov_alpha5, col = "blue", lwd = 2)
lines(normalized_localCov_alpha5, col = "orange", lwd = 2)
# Add a legend
legend("topright",
       legend = c("Simple adaptive Alpha ",
                  "Normalized adaptive Alpha",
                  "Simple Alpha",
                  "Normalized Alpha"),
       col = c("black", "red", "blue", "orange"),
       lty = 1,
       lwd = c(0.9, 2, 2, 2),
       cex = 0.8,
       box.lty = 0)
```



also the graph above, shows the estimation using a $\gamma = 0.001$ and still the performance of mormalized adaptive conformal inference is better compared to others.
now we develope the previous function to another level to use an eGARCH to handle the asymmetries. also in the previous model we assumed that the returns follow a normal distribution but here we change our assumption about to a student-t distribution.

```
## [1] "Done 1300 steps"
## [1] "Done 1400 steps"
## [1] "Done 1500 steps"
## [1] "Done 1600 steps"
## [1] "Done 1700 steps"
## [1] "Done 1800 steps"
## [1] "Done 1900 steps"
## [1] "Done 2000 steps"
## [1] "Done 2100 steps"
## [1] "Done 2200 steps"
## [1] "Done 2300 steps"
## [1] "Done 2400 steps"
## [1] "Done 2500 steps"
```

```
# Plot the first series (localCov_alphat) in black
plot(simple_localCov_alphat3, type = "l", col = "black", main = "Microsof
t", xlab = "Time", ylab = "Local Coverage Level",
     ylim = c(0.925,0.99) , lwd = 0.9)
# Add the second series (localCov_alpha) in blue
lines(normalized_localCov_alphat3, col = "red", lwd = 2)
lines(simple_localCov_alpha3, col = "blue", lwd = 2)
lines(normalized_localCov_alpha3, col = "orange", lwd = 2)
# Add a legend
legend("topright",
       legend = c("Simple adaptive Alpha ",
                  "Normalized adaptive Alpha",
                  "Simple Alpha",
                  "Normalized Alpha"),
       col = c("black", "red", "blue", "orange"),
       lty = 1,
       lwd = c(0.9, 2, 2, 2),
       cex = 0.8,
       box.lty = 0)
```



As it is clear from the plot above, the normalized adaptive CP also has a better performance with less fluctuations and more stable local coverage rate using an eGARCH model instead of sGARCH and also changing the assumption for the distribution from normal to student-t.

3.Aggregate Adaptive Conformal Inference

Well as mentioned in the task one, in conformal prediction, we have N training samples and realizations of random variable and we aim to predict a new observation y_{n+1} at x_{N+1} given a miscoverage rate $\alpha \in [0,1]$(typically 0.1 or 0.05). the aim is to build apredictive interval $C_\alpha such that:$
$P\left(Y_{n+1} \in C_\alpha(X_{n+1})\right) \geq 1-\alpha$ with C_α as small as possible.
Generally the Adaptive Conformal Inference (ACI) is designed to adapt CP to temporal distribution shifts. The idea of ACI is twofold. First, one considers an online procedure with a random split, i.e., train and calibrates are random subsets of the last T_0 points. Second, to improve adaptation when the data is highly shifted, an effective miscoverage level α_t, updated recursively, is used instead of the target level α:

$$C_{\alpha_t}(x_t) = \left[\hat{\mu}(x_t) \pm Q_{1-\alpha_t}(S_{Cal_t})\right]$$

$$\alpha_{t+1} = \alpha + \gamma(\alpha - \mathbb{I}\left\{y_t \notin C_{\alpha_t}(x_t)\right\}$$

for $\gamma \geq 0^3$. if ACI does not cover at time t, then $\alpha_{t+1} \leq \alpha_t$, and the size of the predictive interval increases.

Impact of γ on ACI efficiency:

The choice of the parameter γ strongly impacts the behaviour of ACI: while the method always satisfies the asymptotic validity property.

Theorem3: Assume that: (i) $\alpha \in Q$; (ii) the scores are exchangeable with quantile function Q; (iii) the quantile function is perfectly estimated at each time (as defined above); (iv) the quantile function Q is bounded and $C^4([0,1])$. Then, for all $\gamma > 0$, $(\alpha_t)_{t>0}$ forms a Markov Chain, that admits a stationary distribution π_γ , and as $\gamma \to 0$:

$$E_{\pi_\gamma}[L] = L_0 + Q''(1-\alpha)\frac{\gamma}{2}\alpha(1-\alpha) + O(\gamma^{3/2}).$$

based on this theorem, ACI on exchangeable scores degrades the efficiency linearly with γ compared to CP. This is an important takeaway from the analysis, that underlines that such adaptive algorithms may actually hinder the performance if the data does not have any temporal dependency, and a small γ is preferable. For example, if the residuals are standard gaussians, for $\alpha = 0.01$, setting $\gamma = 0.03$ (resp. $\gamma = 0.05$) will increase the length by 1.59 (resp. by 3.38) with respect to $\gamma = 0$.

To prevent the critical choice of γ an ideal solution is an adaptive strategy with a time dependent γ. We propose two strategies based on running ACI for $K \in N$ values $(\gamma_k)_k \leq K$ of γ, chosen by the user. instead of picking one γ in the grid, we introduce an adaptive aggregation of experts with expert k being ACI with parameter γ_k.

At each step t, it performs two independent aggregations of the K-ACI intervals one for each

bound. and outputs $C_t(\cdot) = \left[b_t^l(\cdot), b_t^u(\cdot)\right]$. Aggregation computes an optimal weighted mean of the experts where the weights $\omega(l)$, $\omega(u)$ assigned to expert k depend on all experts performances (suffered losses) at time steps $1, \cdots, t$. We use the pinball loss ρ_β, as it is frequent in quantile regression, where the pinball parameter β is chosen to $\alpha/2$ (resp. $1 - \alpha/2$) for the lower (resp. upper) bound. These losses are plugged in the aggregation rule ϕ. Finally, the aggregation rule can include the computation of the gradients of the loss. As aggregation rules require bounded experts, a thresholding step is added.

Implementation

in the AgACI paper the authors have used the quantile regression for building the intervals, but in our case we are going to use the garch prediction from the task one for testing the method. at the beginning we create a weighting matrix with an equal initial weights for the γ_k. then in the model we predict the value of the volatility in $t + 1$ and assign it to the expert prediction.then we calculate the scores using the pinball loss function.

in the next step we update the weighting matrix. the scores with lower values take higher weights in the matrix. then we calculate a new γ_{new} using the weighted average of γ_k and update each α_t

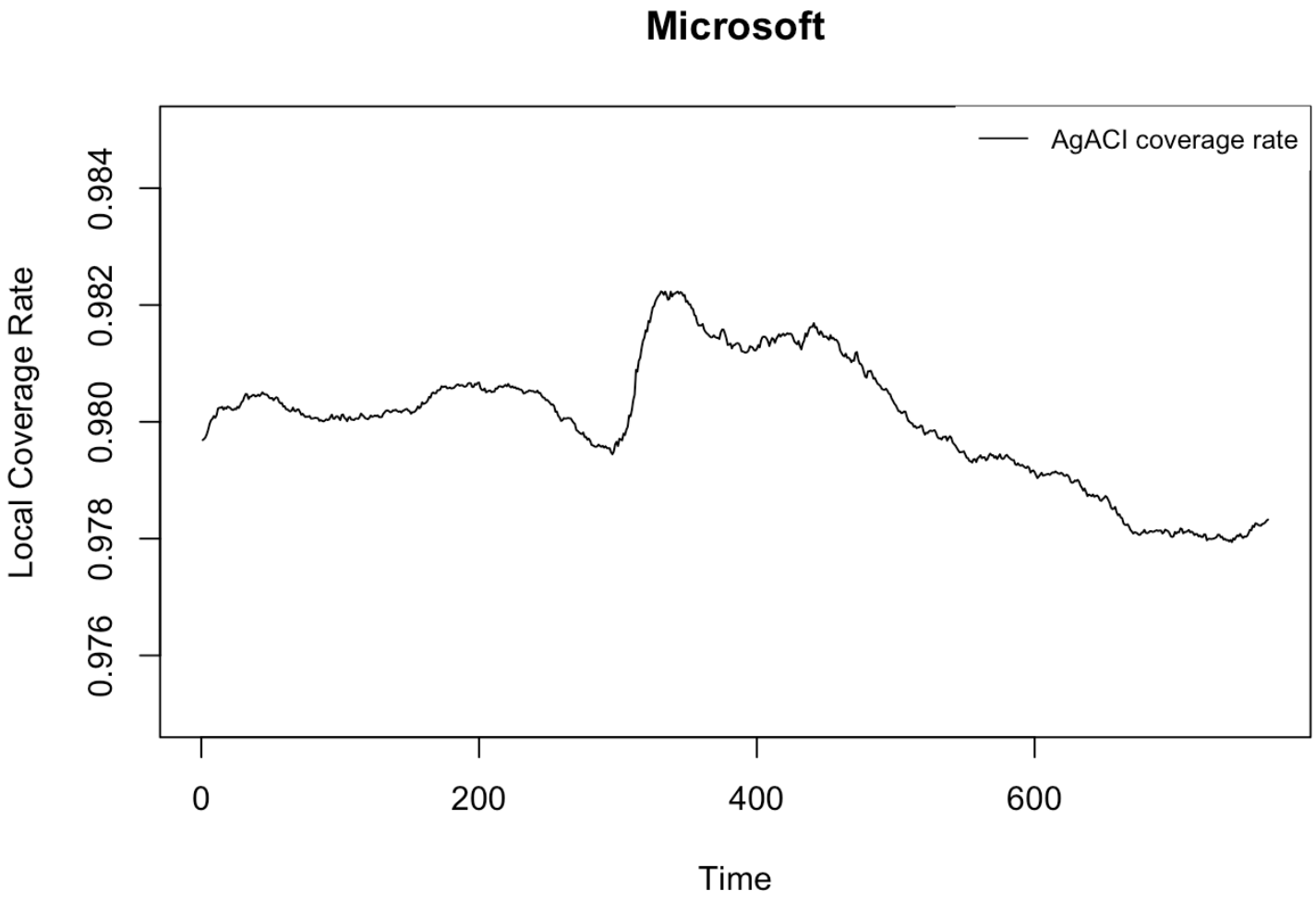
```r
# Pinball Loss Function
pinball_loss <- function(y, pred, beta) {
  loss <- ifelse(y > pred, beta * (y - pred), (1 - beta) * (pred - y))
  return(loss)
}

garchConformalWithExperts <- function(returns, alpha = 0.05, gamma_grid =
seq(0.01, 0.05, length.out = 5),
                                      lookback = 1250, garchP = 1, garchQ
= 1, startUp = 100, verbose = FALSE) {

  # Initialize arrays to store expert predictions, scores, and weights
  alphat <- rep(alpha, length(gamma_grid))
  myT <- length(returns)
  T0 <- max(startUp, lookback)
  beta_l <- rep(0, length(gamma_grid))
  beta_u <- rep(0, length(gamma_grid))
  initialWeight <- 1 / length(gamma_grid)
  gamma_new <- 0
  sigmaNext_perturbed <- 0
  sigmaNext <- 0
  alphaSequence <- rep(alpha, myT - T0 + 1)
  scores <- matrix(0, nrow = length(gamma_grid), ncol = (myT - T0 + 1))
  weightsmatrix <- matrix(0, nrow = length(gamma_grid), ncol = (myT - T0 +
1))  # Weighting for each expert
  expert_predictions <- matrix(0, nrow = length(gamma_grid), ncol = (myT -
T0 + 1))  # Store predictions
  garchSpec <- ugarchspec(mean.model = list(armaOrder = c(0, 0), include.m
ean = FALSE),
                          variance.model = list(model = "sGARCH", garchOrd
er = c(garchP, garchQ)),
                          distribution.model = "norm")

  # Initialize alpha and beta values for each gamma
  for (k in 1:length(gamma_grid)) {
    alphat[k] <- alphat[k] + initialWeight * gamma_grid[k] * alpha
    beta_l[k] <- alphat[k]
    beta_u[k] <- 1 - alphat[k]
  }

  for (t in T0:myT) {
    if (verbose) {
      print(t)
    }

    # For each expert (k), compute the prediction and the pinball loss
    for (k in 1:length(gamma_grid)) {
```

```
      garchFit <- ugarchfit(garchSpec, returns[(t - lookback + 1):(t -
1)], solver = "hybrid")
      garchForecast <- ugarchforecast(garchFit, n.ahead = 1)
      sigmaNext <- sigma(ugarchforecast(garchFit,n.ahead=1))
      sigmaNext_perturbed <- sigmaNext * (1 + gamma_grid[k])  # Add gamma-
based perturbation

      expert_predictions[k, t - T0 + 1] <- sigmaNext_perturbed
      lowQloss <- pinball_loss(returns[t], expert_predictions[k, t - T0 +
1], beta_l[k])
      highQloss <- pinball_loss(returns[t], expert_predictions[k, t - T0 +
1], beta_u[k])
      scores[k, t - T0 + 1] <- lowQloss + highQloss

    }

    # Calculate weights based on the scores
    for (k in 1:length(gamma_grid)) {
      weightsmatrix[k, t - T0 + 1] <- 1 / (scores[k, t - T0 + 1] + 1e-10)
# Avoid division by zero
    }
    weightsmatrix[, t - T0 + 1] <- weightsmatrix[, t - T0 + 1] / sum(weigh
tsmatrix[, t - T0 + 1])

    # Update gamma using the weighted average of gamma_grid
    gamma_new <- mean(weightsmatrix[, t - T0 + 1] * gamma_grid)

    # Step 5: Update alphat for each gamma
    for (k in 1:length(gamma_grid)) {
      alphat[k] <- alphat[k] + gamma_new * (alpha - scores[k, t - T0 + 1])
      beta_l[k] <- alphat[k]
      beta_u[k] <- 1 - alphat[k]
    }

    # Step 6: Combine into a single alphat if needed
    final_alphat <- mean(alphat)
    alphaSequence[t - T0 + 1] <- final_alphat

    if (t %% 100 == 0) {
      print(sprintf("Done %g steps", t))
    }
  }

  return(list(expert_predictions, scores, alphaSequence, weightsmatrix))
}
```

```
# Run the model with expert predictions
gamma_grid = seq(0.001, 0.05, length.out = 5)
testResults <- garchConformalWithExperts(
  returns, alpha = 0.05, gamma_grid = gamma_grid , lookback = 1250,
  garchP = 1, garchQ = 1, startUp = 100, verbose = FALSE
)
```

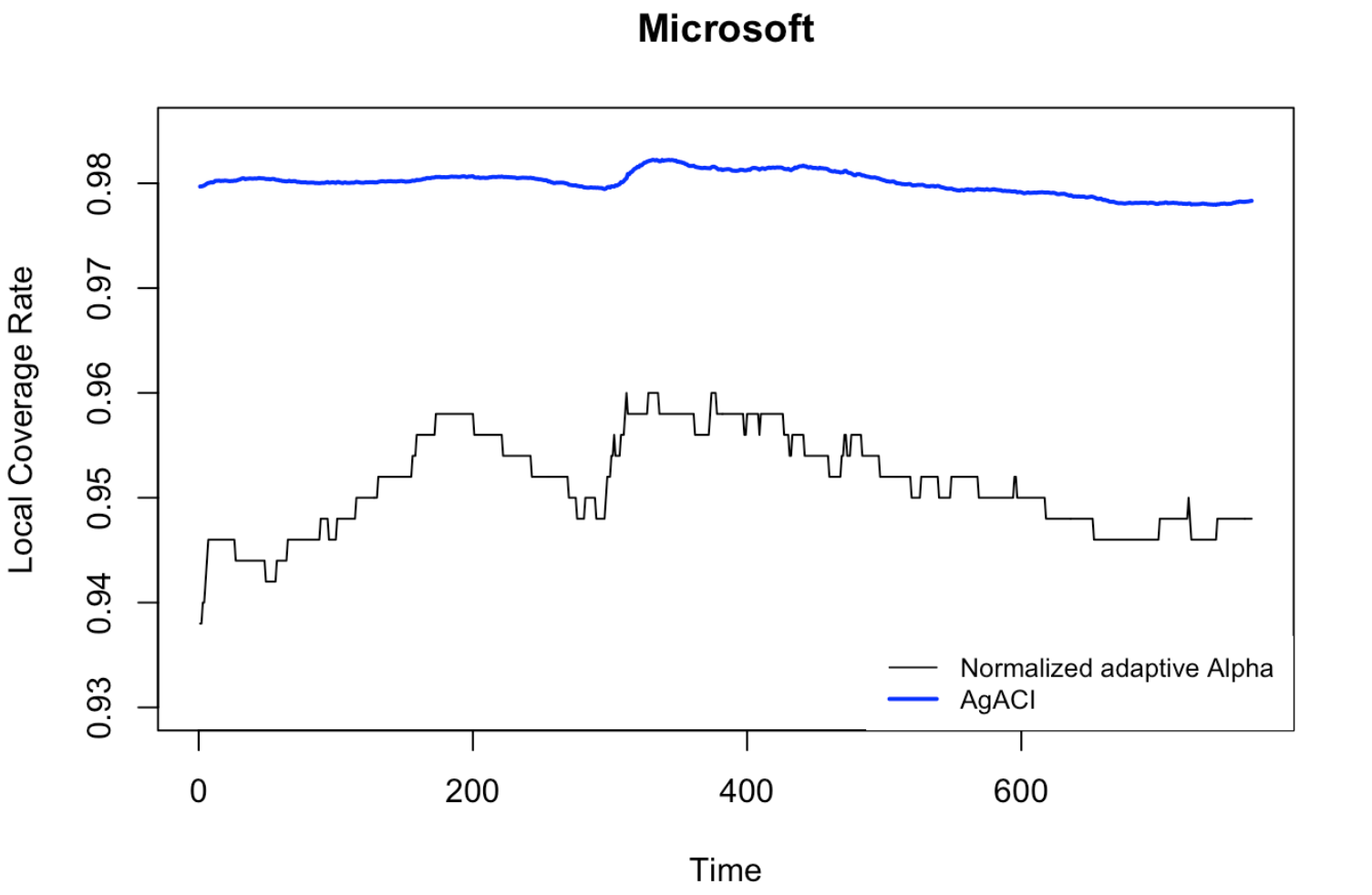
```
## [1] "Done 1300 steps"
## [1] "Done 1400 steps"
## [1] "Done 1500 steps"
## [1] "Done 1600 steps"
## [1] "Done 1700 steps"
## [1] "Done 1800 steps"
## [1] "Done 1900 steps"
## [1] "Done 2000 steps"
## [1] "Done 2100 steps"
## [1] "Done 2200 steps"
## [1] "Done 2300 steps"
## [1] "Done 2400 steps"
## [1] "Done 2500 steps"
```

```
scores <- testResults[[2]]
length_scores = length(scores[1,])
localCov <- matrix(0,nrow=length(gamma_grid),ncol = length_scores - 500+1)
for(k in 1:length(gamma_grid)){
  for(t in 251:(length_scores - 250 +1)){
    localCov[k,t-250] <- 1 - mean(scores[k,(t-250):(t+250-1)])
  }
}
plot(localCov[1,], type='l', ylim=c(0.975,0.985), main= "Microsoft",xlab
="Time", ylab= "Local Coverage Rate")
legend("topright",
       legend = c("AgACI coverage rate"),
       col = c("black"),
       lty = 1,
       lwd = c(0.9),
       cex = 0.8,
       box.lty = 0)
```



As we can see from the plot, the coverage rate for AgACI is higher than ACI and also it is more stable compared to that. in the following we plot both the AgACI and ACI in one graph to show the differences visually:

```
scores <- testResults[[2]]
length_scores = length(scores[1,])
localCov <- matrix(0,nrow=length(gamma_grid),ncol = length_scores - 500+1)
for(k in 1:length(gamma_grid)){
  for(t in 251:(length_scores - 250 +1)){
    localCov[k,t-250] <- 1 - mean(scores[k,(t-250):(t+250-1)])
  }
}
plot(normalized_localCov_alphat2, type='l', ylim=c(0.93,0.985), xlab="Tim
e", ylab= "Local Coverage Rate", main = "Microsoft")

lines(localCov[1,], col = "blue", lwd = 2)
legend("bottomright",
       legend = c("Normalized adaptive Alpha ",
                  "AgACI"),
       col = c("black", "blue"),
       lty = 1,
       lwd = c(0.9, 2),
       cex = 0.8,
       box.lty = 0)
```



So comparing the results of two models, we see that the AgACI shows less fluctuations which shows some kind of stability. but on the other hand having a 0.98 Coverage rate may be a result of a wider and less efficient interval.

4.Conclusion:

So as seen in the report, we implemented the CI, ACI and AgACI methods for constructing conformal intervals for predictions made by sGARCH and eGARCH models. At the begining, we tried to calculate the local coverage rate using a fixed and adaptive confidence interval and in the following we also used aggregations to also update the γ.

To conclude the report we reported the results for all the methods and we see that on our data the AgACI method has more stable but less efficient results. although it worths mentioning that for confidently state that AgACI generally performs in a higher level, we should test it on other datasets as well.