Comparaison des modèles SARIMA et des modèles d'apprentissage automatique pour la prévision des données temporelles

BENABOU Mohamed El Ghali

March 2023

Introduction

Les séries temporelles sont des données qui varient avec le temps et sont souvent utilisées pour comprendre les tendances et les modèles dans de nombreux domaines tels que l'économie, la finance, la météorologie et bien d'autres. La modélisation des séries temporelles est importante pour la prévision et l'analyse de ces données.

Deux des approches les plus populaires pour modéliser les séries temporelles sont les méthodes de Box-Jenkins et les modèles de forêt aléatoire. La méthode de Box-Jenkins est basée sur l'identification, l'estimation et la vérification des modèles ARIMA (AutoRegressive Integrated Moving Average), tandis que les modèles de forêt aléatoire sont des méthodes d'apprentissage automatique qui construisent des ensembles d'arbres de décision pour modéliser les relations entre les variables.

Dans ce projet, nous allons comparer l'efficacité de ces deux approches en utilisant un modèle SARIMA (Seasonal ARIMA) et un modèle de forêt aléatoire pour modéliser une série temporelle donnée. Nous allons évaluer les performances de chaque modèle en utilisant des métriques telles que l'erreur moyenne absolue (MAE), l'erreur quadratique moyenne (RMSE) et le coefficient de détermination (R^2).

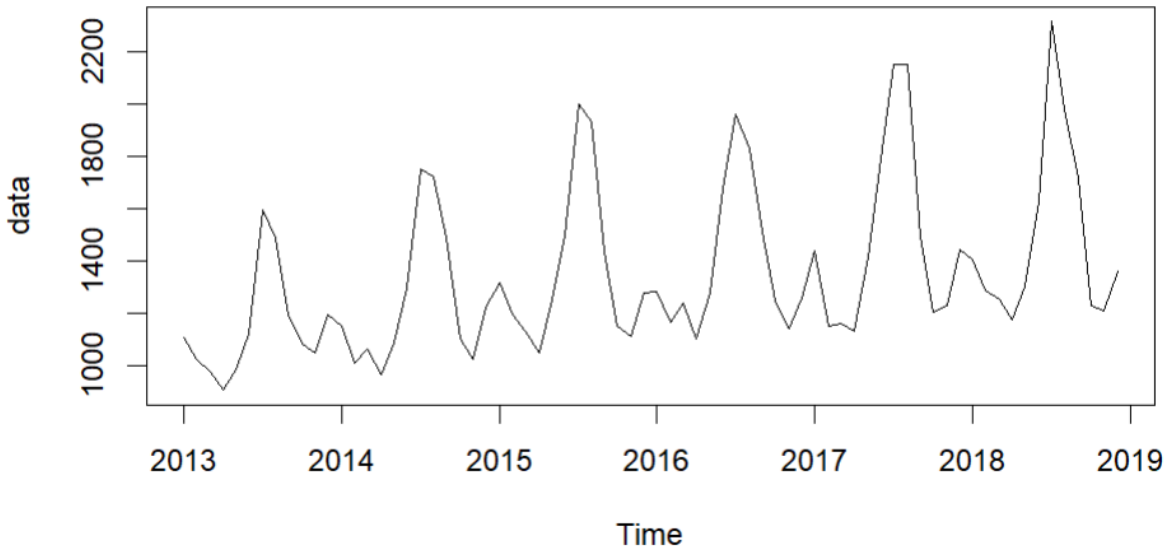
L'objectif de ce projet est de déterminer quelle méthode est la plus appropriée pour modéliser une série temporelle donnée en fonction de sa complexité et de la précision des prévisions.

Présentation des données

Le tableau suivant représente les achats d'électricité de la RDC de 2013 à 2018 :

Année	Jan	Fév	Mar	Avr	Mai	Juin	Juil	Août	Sep	Oct	Nov	Déc
2013	1105.1	1019.8	973.5	904.4	981.1	1119.9	1596.4	1484.5	1191.7	1079.9	1044.3	1194.2
2014	1151.6	1005.2	1058.9	964.0	1085.7	1297.2	1752.5	1721.1	1487.3	1102.3	1020.0	1222.9
2015	1313.2	1195.4	1128.4	1046.2	1254.2	1498.8	2000.5	1931.0	1432.6	1150.1	1108.1	1278.0
2016	1282.8	1163.2	1235.4	1102.7	1278.6	1679.5	1959.3	1827.3	1512.4	1249.3	1140.9	1254.5
2017	1437.3	1149.3	1160.8	1127.4	1414.2	1790.0	2154.2	2149.8	1490.4	1204.6	1225.1	1441.0
2018	1401.4	1288.5	1254.0	1137.9	1302.6	1622.3	2317.6	1981.1	1716.3	1225.5	1210.2	1361.5

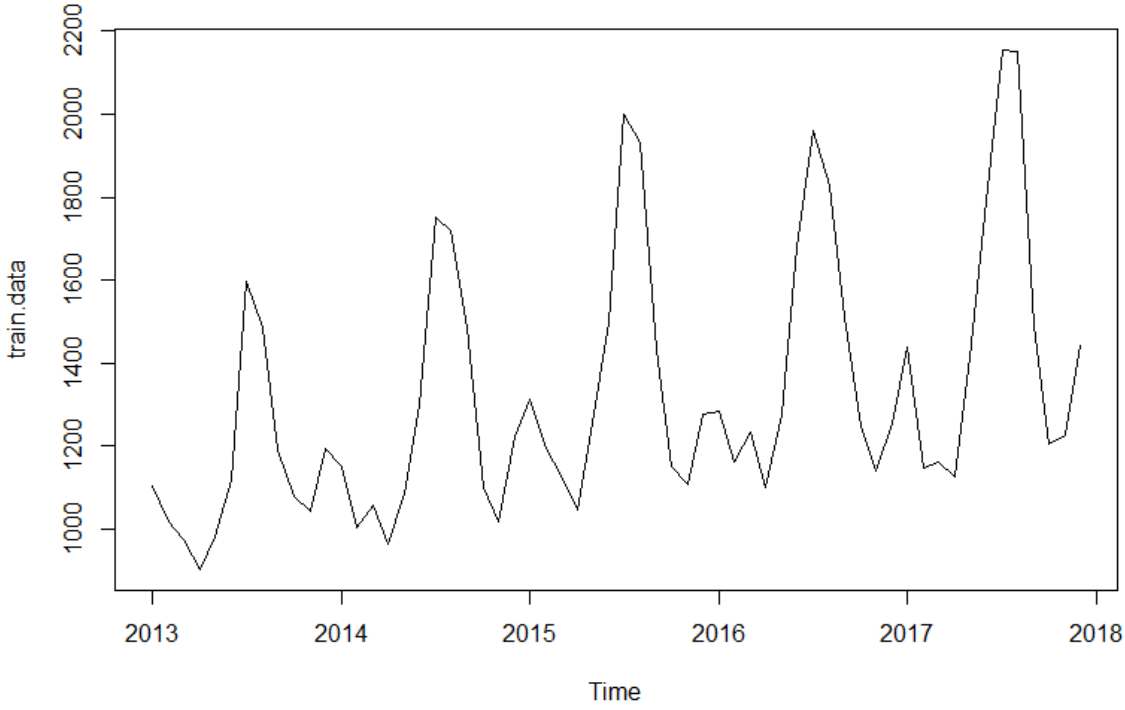
Graphe initial des données



On commencera par partager les données en deux partie. Une partie (de l'année 2013 à l'année 2017) sera utilisée pour entraîner nos modèles et l'autre partie (l'année 2018) nous servira pour les tester et les comparer par la suite.

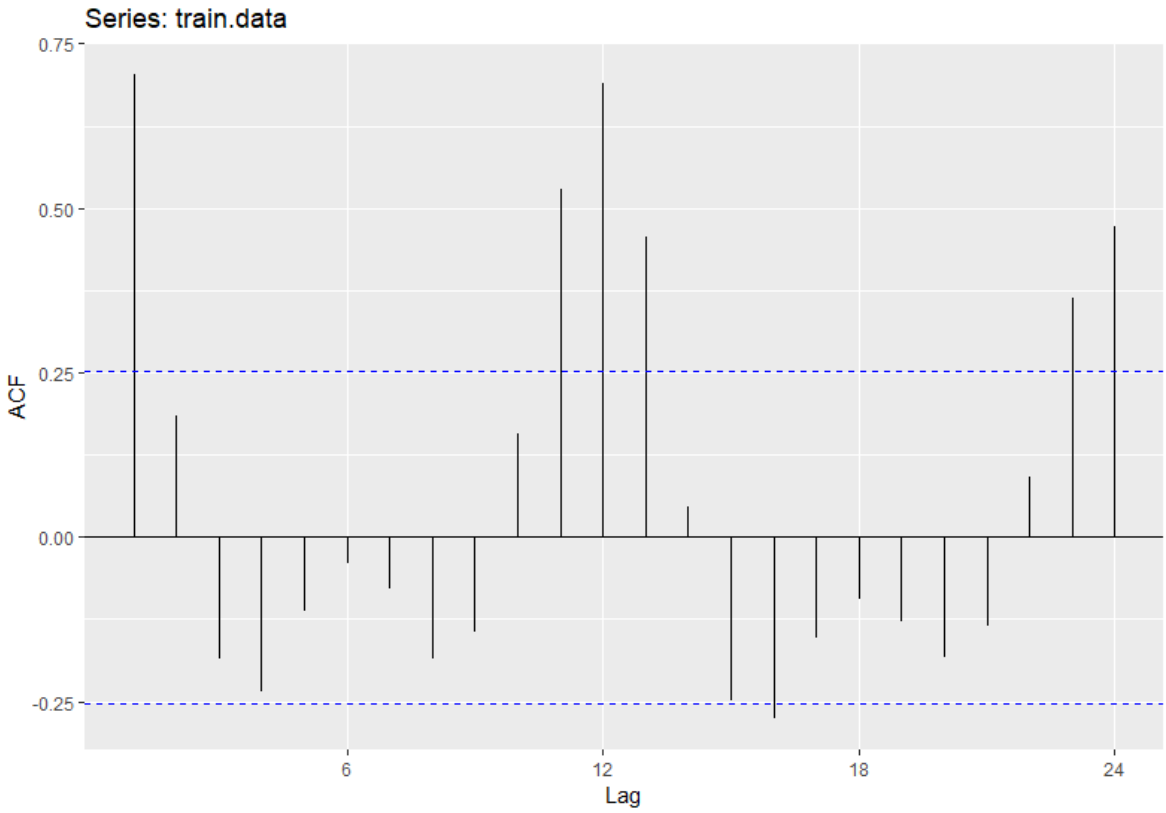
Partage des données

```
> train.data = ts(data = d[1:60], start = 2013, frequency = 12)
> train.data
        Jan    Feb    Mar    Apr    May    Jun    Jul    Aug    Sep    Oct    Nov    Dec
2013 1105.1 1019.8  973.5  904.4  981.1 1119.9 1596.4 1484.5 1191.7 1079.9 1044.3 1194.2
2014 1151.6 1005.2 1058.9  964.0 1085.7 1297.2 1752.5 1721.1 1487.3 1102.3 1020.0 1222.9
2015 1313.2 1195.4 1128.4 1046.2 1254.2 1498.8 2000.5 1931.0 1432.6 1150.1 1108.1 1278.0
2016 1282.8 1163.2 1235.4 1102.7 1278.6 1679.5 1959.3 1827.3 1512.4 1249.3 1140.9 1254.5
2017 1437.3 1149.3 1160.8 1127.4 1414.2 1790.0 2154.2 2149.8 1490.4 1204.6 1225.1 1441.0
>
> test.data = ts(data = d[61:72], start = 2018, frequency = 12)
> test.data
        Jan    Feb    Mar    Apr    May    Jun    Jul    Aug    Sep    Oct    Nov    Dec
2018 1401.4 1288.5 1254.0 1173.9 1302.6 1622.3 2317.6 1981.1 1716.3 1225.5 1210.2 1361.5
```

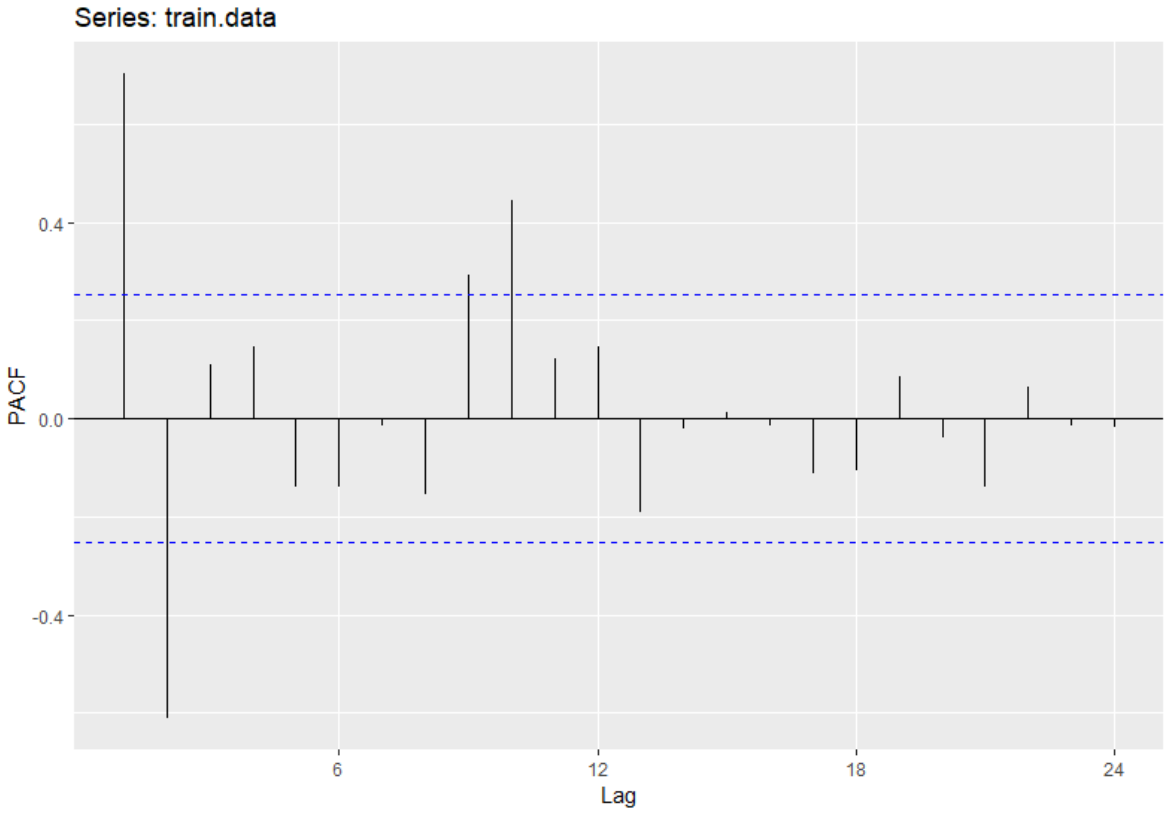


Cette figure montre le graphique de la série chronologique de l'ensemble de données de 2013 à 2018. La représentation graphique de la série peut nous donner une idée de la tendance, de la saisonnalité et de la variance de la série. Cette information est importante pour déterminer si la série doit être pré-traitée avant d'être utilisée pour l'entraînement de modèles.

1 Modèle ARIMA



L'ACF mesure la corrélation entre les observations d'une série à différents décalages temporels. Dans cette représentation graphique, chaque barre représente la corrélation entre la série à un décalage temporel donné. Nous remarquons que les corrélations sont assez fortes sur plusieurs décalages, sans présenter de décroissance rapide ou linéaire. Cela peut indiquer que la série chronologique n'est pas stationnaire, ce qui peut avoir des conséquences importantes sur la performance des modèles que nous allons construire. Nous pourrions donc devoir appliquer des transformations de la série, telles que la différenciation, pour obtenir une série stationnaire avant de poursuivre notre analyse. Nous remarquons aussi que les corrélations présentent une répétition environ tous les 12 décalages, ce qui peut indiquer la présence d'une saisonnalité dans les données. La détection de la saisonnalité est importante car elle peut avoir un impact significatif sur le choix du modèle et des hyperparamètres appropriés.



La PACF mesure la corrélation entre les observations d'une série à différents décalages temporels, en éliminant les effets des corrélations à des décalages intermédiaires. Dans cette représentation graphique, chaque barre représente la corrélation partielle entre la série à un décalage temporel donné. Nous remarquons que les corrélations sont significatives à plusieurs décalages. Cela peut indiquer que la série a une structure AR(10) ou bien qu'elle est non stationnaire. On devra donc rendre la série stationnaire avant de pouvoir conclure.

```
> adf.test(train.data)

        Augmented Dickey-Fuller Test

data:  train.data
Dickey-Fuller = -3.9424, Lag order = 3, p-value = 0.01822
alternative hypothesis: stationary

> kpss.test(train.data)

        KPSS Test for Level Stationarity

data:  train.data
KPSS Level = 0.4866, Truncation lag parameter = 3, p-value = 0.04469
```

L'image montre les résultats des tests ADF et KPSS appliqués à notre série chronologique. Le test ADF a une hypothèse nulle (H_0) selon laquelle la série n'est pas stationnaire. La p-value associée au test ADF est de 0.01822, ce qui est inférieur au seuil de 0,05. Par conséquent, nous rejetons H_0 et concluons que la série est stationnaire au niveau de signification de 5%. Le test KPSS a une hypothèse nulle (H_0) selon laquelle la série est stationnaire. La p-value associée au test KPSS est de 0.04469, ce

qui est également inférieur au seuil de 0,05. Cependant, contrairement au test ADF, le rejet de H_0 dans le test KPSS signifie que la série est non-stationnaire. Il est important de noter que les tests ADF et KPSS ont des hypothèses inverses, mais des résultats cohérents peuvent renforcer la conclusion que nous avons obtenue. Dans notre cas, les résultats des deux tests ne sont pas congruents ce qui pourrait encore une fois indiquer que les données ne sont pas stationnaires.

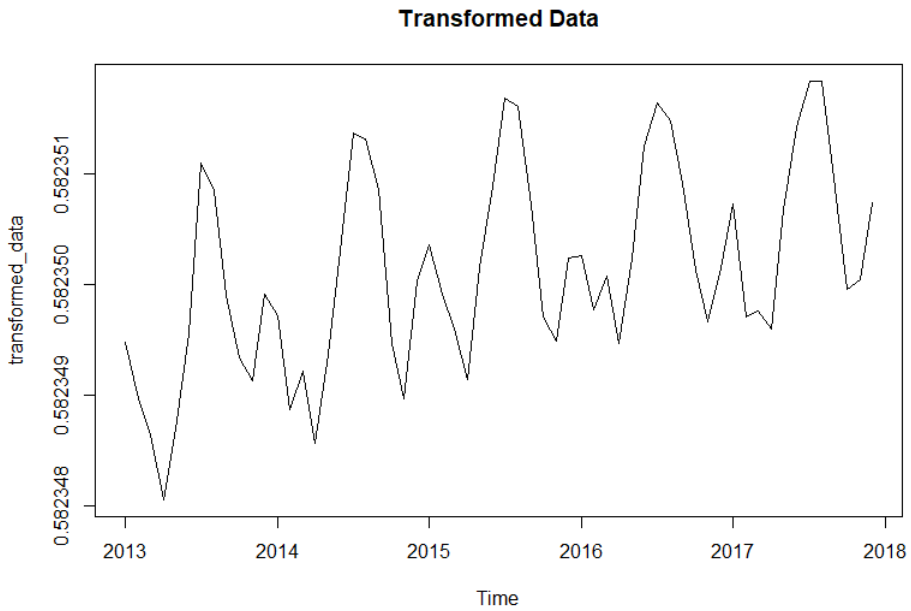
On commencera par une transformation de Box-Cox sur nos données :

```
> bc = boxcox(train.data~1, lambda = seq(-5,5,0.1))
> lambda  = bc$x[which.max(bc$y)]
> lambda
[1] -1.7
```

La figure montre le code utilisé pour effectuer la transformation Box-Cox sur les données d'entraînement. Tout d'abord, la fonction 'boxcox' est appelée avec les données d'entraînement et une plage de valeurs lambda allant de -5 à 5 avec un pas de 0,1. Ensuite, la fonction 'which.max' est utilisée pour trouver l'index de la valeur lambda qui maximise la fonction de vraisemblance. La valeur optimale de lambda est ensuite extraite de la plage de valeurs lambda en utilisant cet index. La valeur optimale trouvée est de -1.7, indiquant que la transformation Box-Cox optimale pour ces données est une transformation inversée. Cette étape est importante pour améliorer les performances du modèle, en particulier si les résidus ne sont pas normalement distribués ou si leur variance varie considérablement.

```
transformed_data <- if (lambda == 0) log(train.data) else (train.data^lambda - 1) / lambda
```

On effectue par la suite la transformation Box-Cox sur les données d'entraînement avec la valeur optimale de lambda trouvée plus tôt.

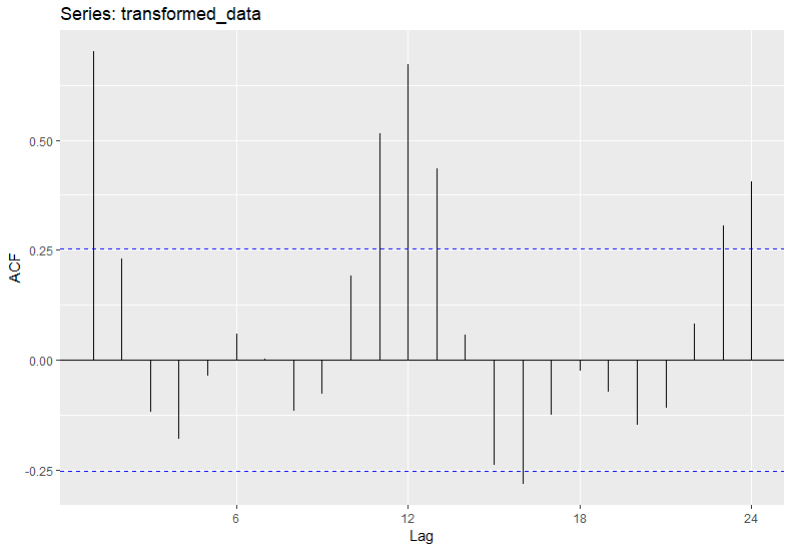


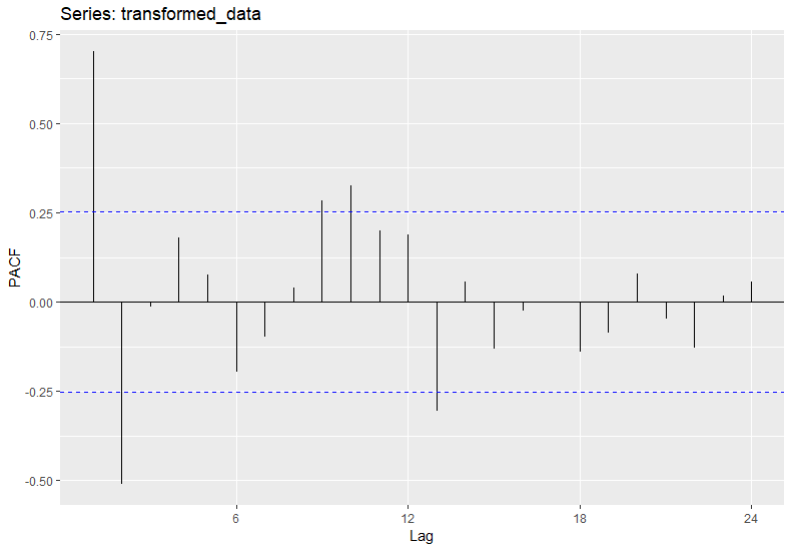
Le graphe des données transformées par la méthode de Box-Cox permet de visualiser l'effet de la transformation sur les données. Cette transformation vise à rendre la distribution des données plus proche d'une distribution normale, ce qui est souvent un prérequis pour l'utilisation de certains modèles

statistiques.

A première vue, on peut remarquer que la transformation de Box-Cox n'a pas eu un grand effet sur la distribution des données dans ce cas particulier. Cependant, il est important de noter que l'effet de cette transformation dépend de la distribution initiale des données. Dans certains cas, la transformation peut avoir un effet plus prononcé et permettre de mieux modéliser les données.

Il est également important de considérer que même si la transformation n'a pas eu un effet visible sur le graphe, elle peut encore améliorer la performance des modèles prédictifs en permettant de mieux respecter les hypothèses statistiques du modèle. Ainsi, il est important de ne pas négliger l'utilisation de cette transformation, même si elle ne semble pas avoir un effet visible sur le graphe des données transformées.





On remarque que la transformation de Box-Cox n'a pas (ou a peu) affecté les graphes de l'ACF et le PACF.

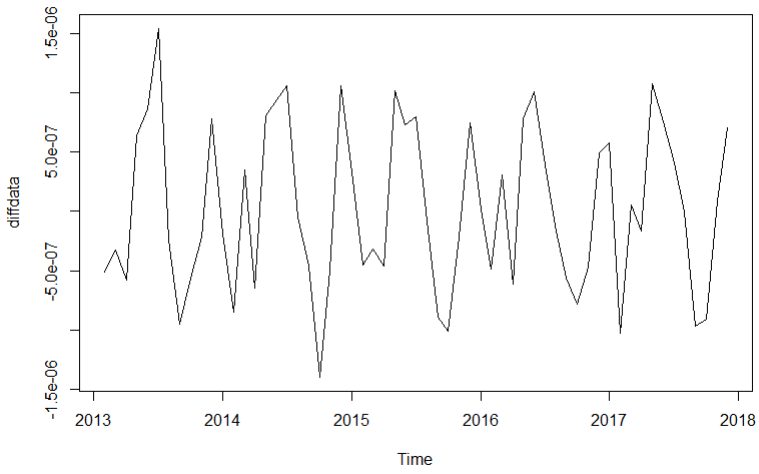
```
> adf.test(transformed_data)

        Augmented Dickey-Fuller Test

data:  transformed_data
Dickey-Fuller = -4.4122, Lag order = 3, p-value = 0.01
alternative hypothesis: stationary

Warning message:
In adf.test(transformed_data) : p-value smaller than printed p-value
> kpss.test(transformed_data)

        KPSS Test for Level Stationarity

data:  transformed_data
KPSS Level = 0.63531, Truncation lag parameter = 3, p-value = 0.01943
```

Les résultats des tests ADF et KPSS appliqués à notre série chronologique transformée nous donne une p-value associée au test ADF inférieure à 0.01, ce qui est inférieur au seuil de 0,05. Par conséquent, nous rejetons H_0 et concluons que la série est stationnaire au niveau de signification de 5% et une p-value associée au test KPSS est de 0.01943, ce qui est également inférieur au seuil de 0,05 qui signifie que la série est non-stationnaire. Les résultats des deux tests ne sont toujours pas congruents ce qui pourrait encore une fois indiquer que les données ne sont toujours pas stationnaires. Ce qui est normal car n'a pas encore stabiliser la moyenne.

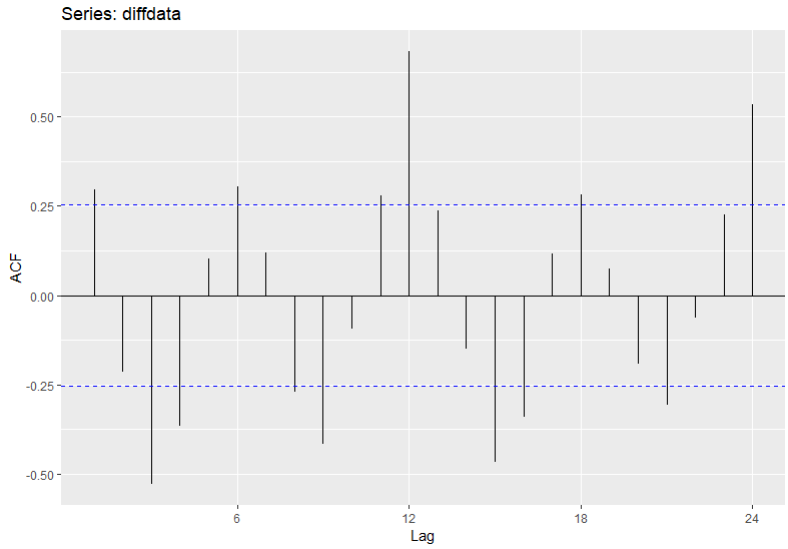
```
> nsdiffs(trans.data, alpha = 0.05)
[1] 1
```

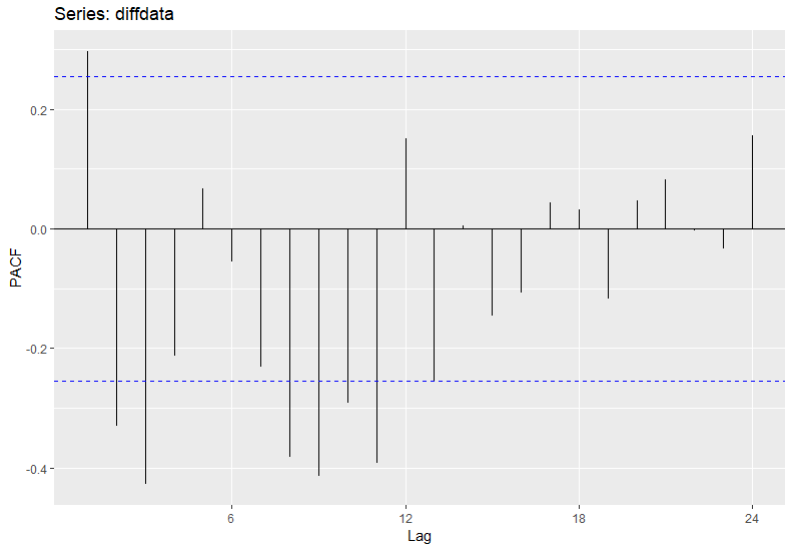
On utilise par la suite la fonction nsdiff pour voir combien de différentiation est nécessaire pour stabiliser la série transformée.

diffdata = diff(transformed_data)



Après avoir différencié les données, le nouveau graphe montre une série temporelle qui ne semble plus avoir de tendances et de structures à long terme. Cela suggère que la différenciation a peut-être permis d'éliminer une tendance linéaire qui était présente dans les données initiales.





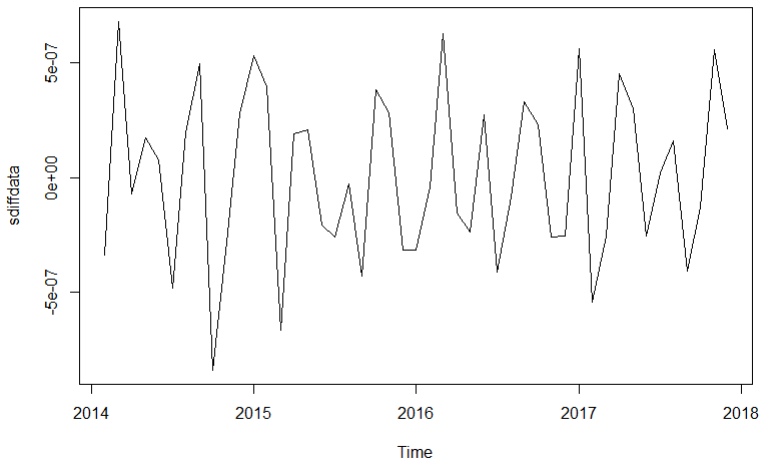
On retrace l'ACF et le PACF et on remarque qu'il y a une répétition très distinguée après un délai de 12 dans l'ACF ce qui suggère l'existence d'une saisonnalité de fréquence 12 qu'on bien remarqué plus tôt dans les tracés des séries avant et après transformation Box-Cox.

```
> adf.test(diffdata)  #normal differentiation

        Augmented Dickey-Fuller Test

data:  diffdata
Dickey-Fuller = -6.4436, Lag order = 3, p-value = 0.01
alternative hypothesis: stationary

Warning message:
In adf.test(diffdata) : p-value smaller than printed p-value
> kpss.test(diffdata)

        KPSS Test for Level Stationarity

data:  diffdata
KPSS Level = 0.023311, Truncation lag parameter = 3, p-value = 0.1

Warning message:
In kpss.test(diffdata) : p-value greater than printed p-value
```

Les nouveaux résultats des tests ADF et KPSS appliqués à notre série chronologique transformée puis différenciée nous donne une p-value associée au test ADF inférieure à 0.01, ce qui est inférieur au seuil de 0,05. Par conséquent, nous rejetons H_0 et concluons que la série est stationnaire au niveau de signification de 5% et une p-value associée au test KPSS supérieur à 0.1, ce qui est supérieur au seuil de 0,05 qui signifie que la série est stationnaire pour ce test aussi. Les résultats des deux tests supportent que les données sont stationnaires.

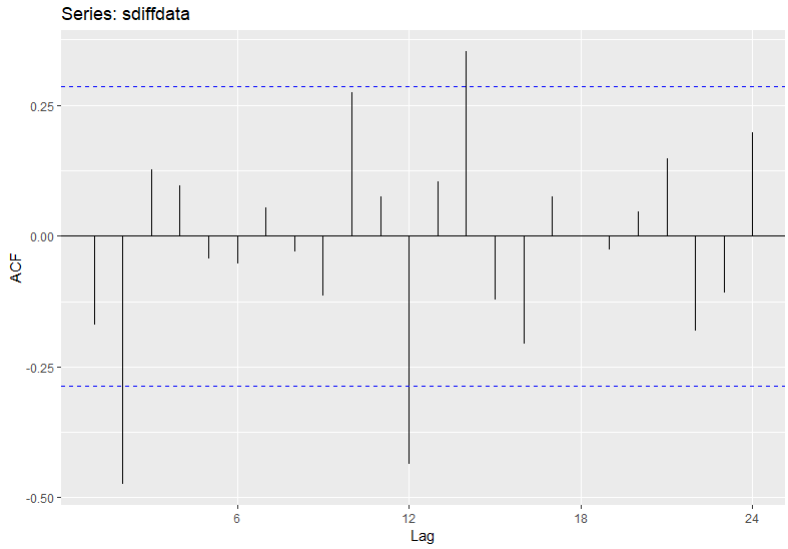
On préférera quand même faire une différenciation de délai 12 pour pouvoir enlever la saisonnalité :

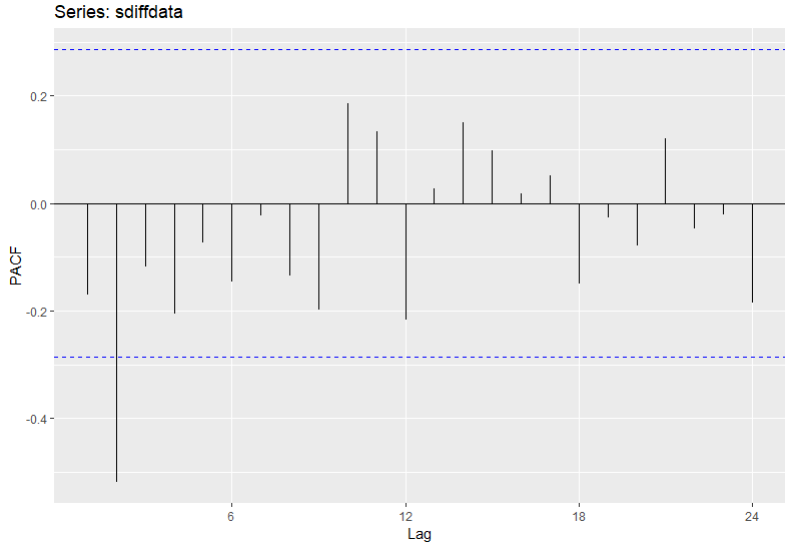
sdiffdata = diff(diffdata, 12)



On retrace le graphe des données après avoir différencié les données avec un délai de 12. Ce nouveau graphe montre une série temporelle qui ne semble plus avoir de tendances et de structures à long terme ni de saisonnalité. Cela suggère que la différenciation a peut-être permis d'éliminer la saisonnalité qui était présente dans les données avant différentiation.

On trace l'ACF et le PACF des nouvelles données :





L'ACF est toujours bruyant mais nous remarquons que dans le PACF, les corrélations ne sont significatives qu'au décalage 2. Cela peut indiquer que la série a une structure ARMA(2,q).

On vérifie à nouveau la stationnarité dans notre cas :

```
> adf.test(sdiffdata)

        Augmented Dickey-Fuller Test

data:  sdiffdata
Dickey-Fuller = -5.6899, Lag order = 3, p-value = 0.01
alternative hypothesis: stationary

Warning message:
In adf.test(sdiffdata) : p-value smaller than printed p-value
> kpss.test(sdiffdata)

        KPSS Test for Level Stationarity

data:  sdiffdata
KPSS Level = 0.072379, Truncation lag parameter = 3, p-value = 0.1

Warning message:
In kpss.test(sdiffdata) : p-value greater than printed p-value
```

On trouve que les résultats des tests de stationnarité n'ont pas changé depuis la dernière fois donc la série est toujours stationnaire.

On commence alors à tester les différents modèles avec AR(2) :

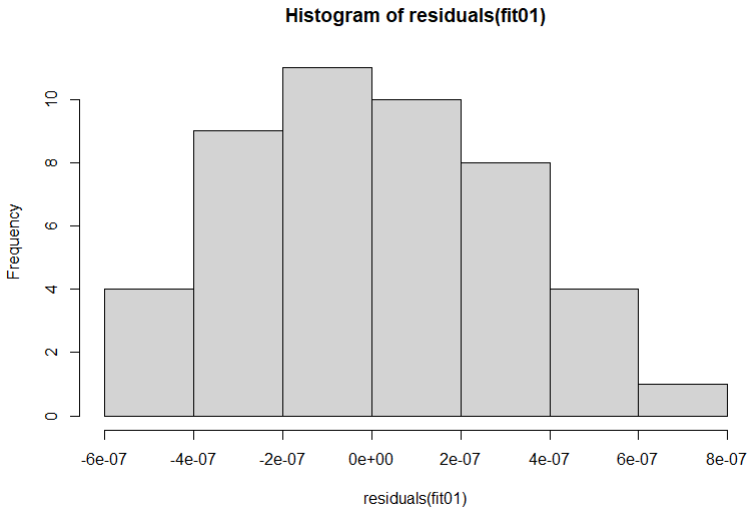
```
fit01 <- arima(sdiffdata, c(2, 0, 0))
fit01
tsdiag(fit01)

fit02 <- arima(sdiffdata, c(2, 0, 1))
fit02
tsdiag(fit02)

fit03 <- arima(sdiffdata, c(2, 0, 2))
fit03
tsdiag(fit03)
```

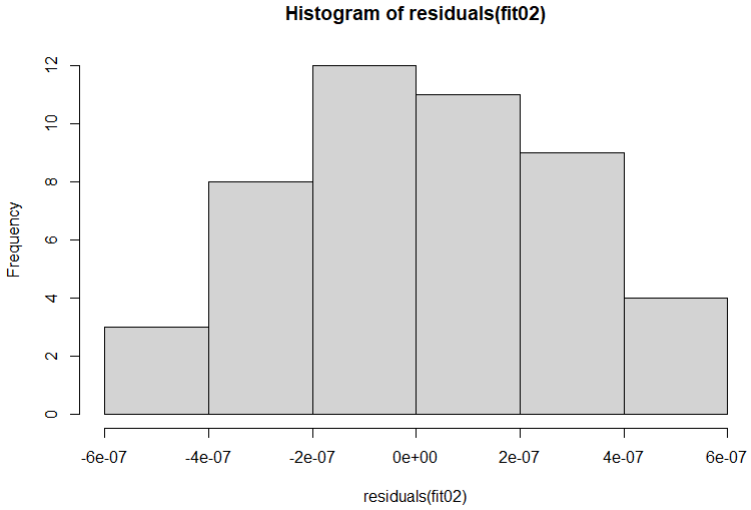
Le premier modèle sera donc un AR(2), le deuxième un ARMA(2,1) et le troisième un ARMA (2,2).

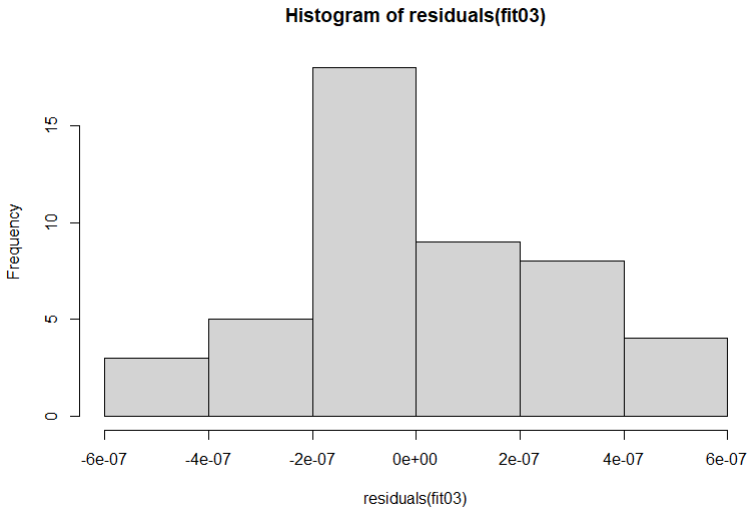
On considère un modèle comme étant bon, si les résidus standardisés suivent un bruit blanc. Ceci revient aux conditions suivantes :

- Le distribution des résidus doit suivre une loi normale centrée réduite.
- L'ACF doit être 1 au délai 0 et non significatif dans le reste des délais.
- Les p-values pour le test statistique de Ljung-Box doivent toutes être au-dessus de 0.05 pour avoir l'indépendance.

On regarde l'histogramme des résidus des différents modèles et on utilise le test de Shapiro-Wilk pour voir si les résidus suivent une loi normale.


Histogram of residuals(fit01)
Frequency
0
2
4
6
8
10
-6e-07
-4e-07
-2e-07
0e+00
2e-07
4e-07
6e-07
8e-07
residuals(fit01)


Histogram of residuals(fit02)
Frequency
0
2
4
6
8
10
12
-6e-07
-4e-07
-2e-07
0e+00
2e-07
4e-07
6e-07
residuals(fit02)

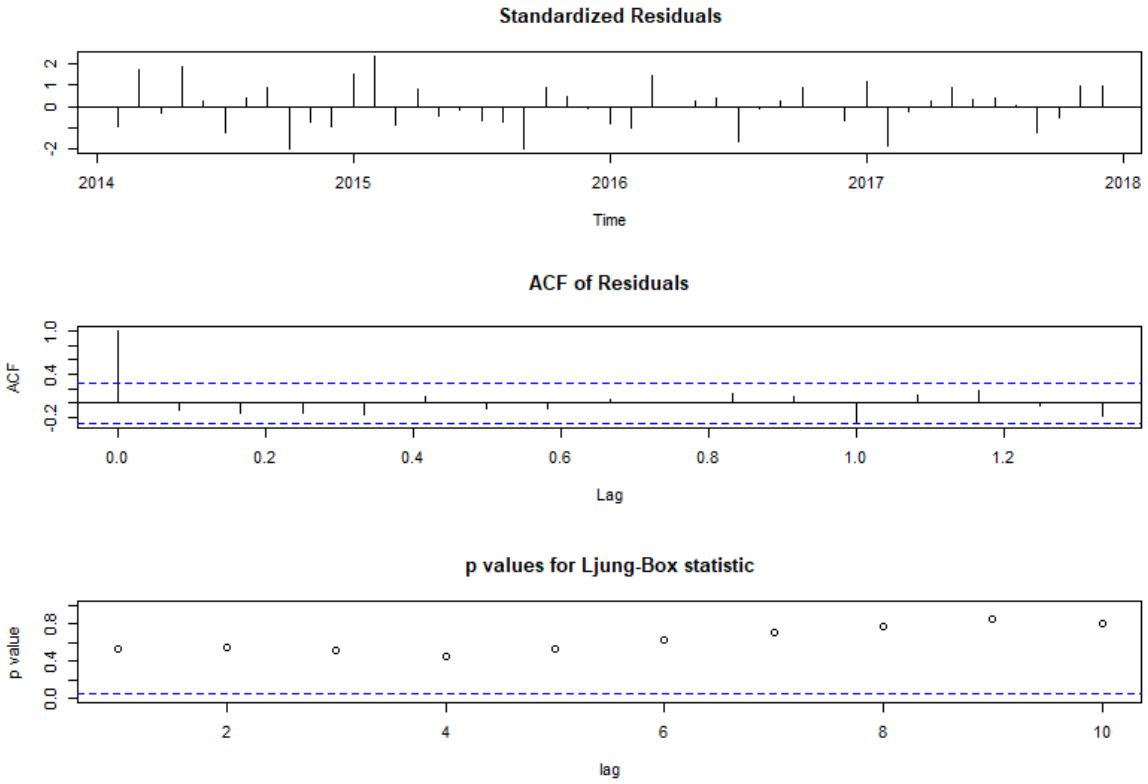


Les 3 histogrammes suggèrent que les résidus suivent des lois normales.

```
> shapiro.test(residuals(fit01))

        Shapiro-Wilk normality test

data:  residuals(fit01)
W = 0.98828, p-value = 0.9144

> shapiro.test(residuals(fit02))

        Shapiro-Wilk normality test

data:  residuals(fit02)
W = 0.97418, p-value = 0.3785

> shapiro.test(residuals(fit03))

        Shapiro-Wilk normality test

data:  residuals(fit03)
W = 0.97432, p-value = 0.383
```

Les tests de Shapiro-Wilk ont pour p-values 0.9144, 0.3785 et 0.383 pour les trois modèles AR(2), ARMA(2,1) et ARMA(2,2) respectivement qui sont toutes des valeurs supérieures à 0.05 et donc les résidus des trois modèles suivent une loi normale.

Modèle Fit01 - AR(2)

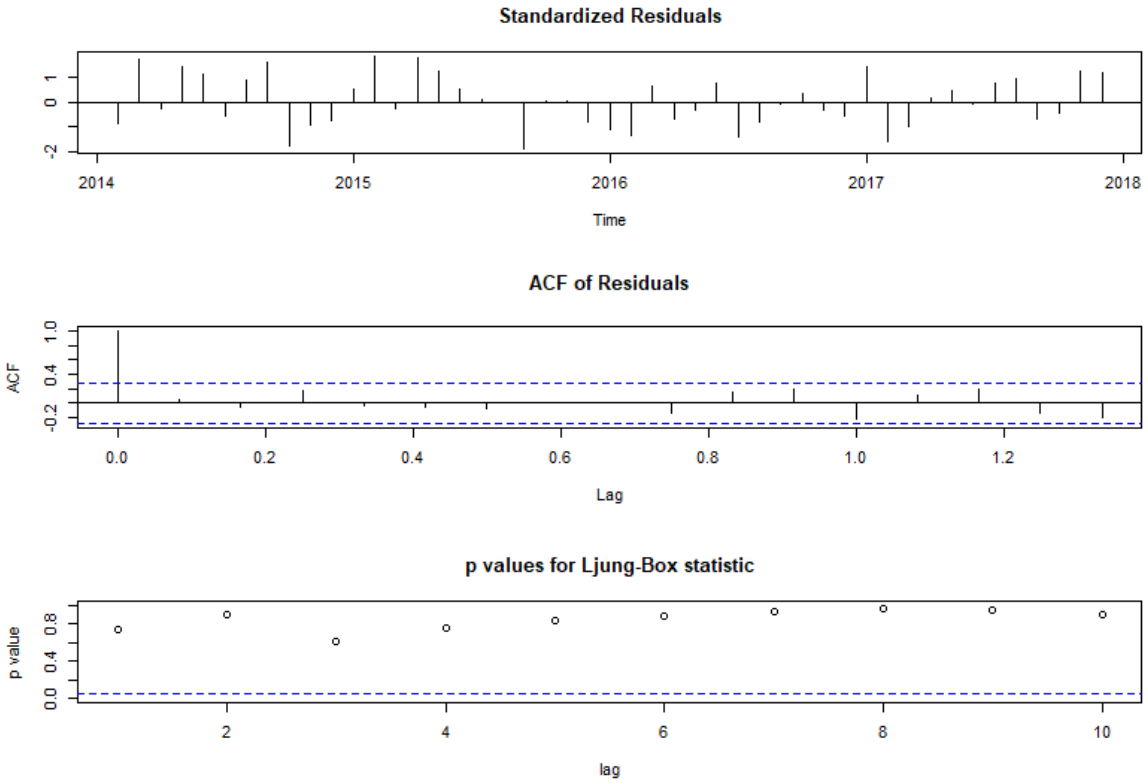


Pour ce premier modèle AR(2), on a les trois conditions sont respectées. A l'exception peut-être dans le graphe de l'ACF au lag 12 des résidus.

```
> fit01

Call:
arima(x = sdiffdata, order = c(2, 0, 0))

Coefficients:
          ar1      ar2  intercept
      -0.2626  -0.5759          0
s.e.   0.1127   0.0657        NaN

sigma^2 estimated as 8.917e-14:  log likelihood = 639.03,  aic = -1270.05
```

On dira que ce modèle est un bon modèle et qu'il modélise bien nos données transformées et on notera son AIC = -1270.05 pour pouvoir le comparer avec le reste des modèles retenus.

Modèle Fit02 - ARMA(2,1)

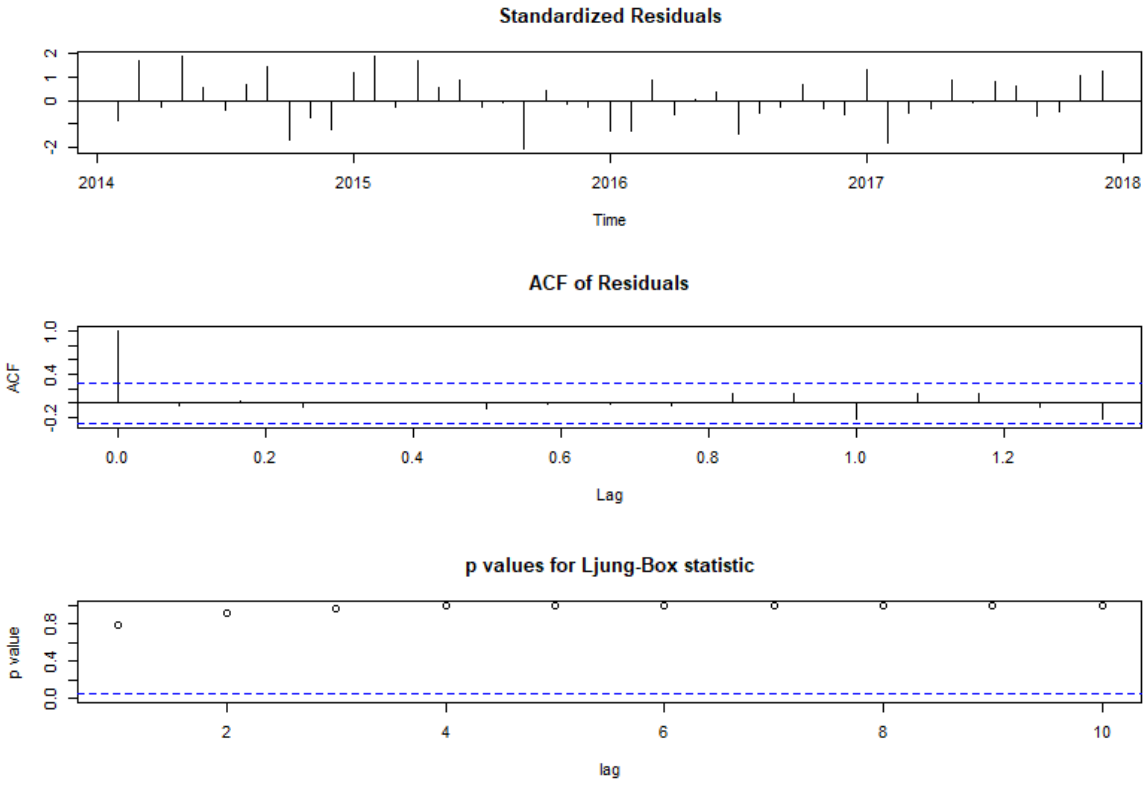


Ce deuxième modèle ARMA(2,1), constitue aussi un bon modèle car il vérifie les trois conditions.

```
> fit02

Call:
arima(x = sdiffdata, order = c(2, 0, 1))

Coefficients:
         ar1      ar2      ma1  intercept
      0.2902  -0.4192  -1.0000          0
s.e.  0.1356   0.0373   0.0795        NaN

sigma^2 estimated as 6.977e-14:  log likelihood = 642.95,  aic = -1275.9
```

On notera alors son AIC = -1275.9 pour pouvoir le comparer avec le reste des modèles retenus.

Modèle Fit03 - ARMA(2,2)



Ce troisième modèle ARMA(2,2), constitue aussi un bon modèle car il vérifie les trois conditions.

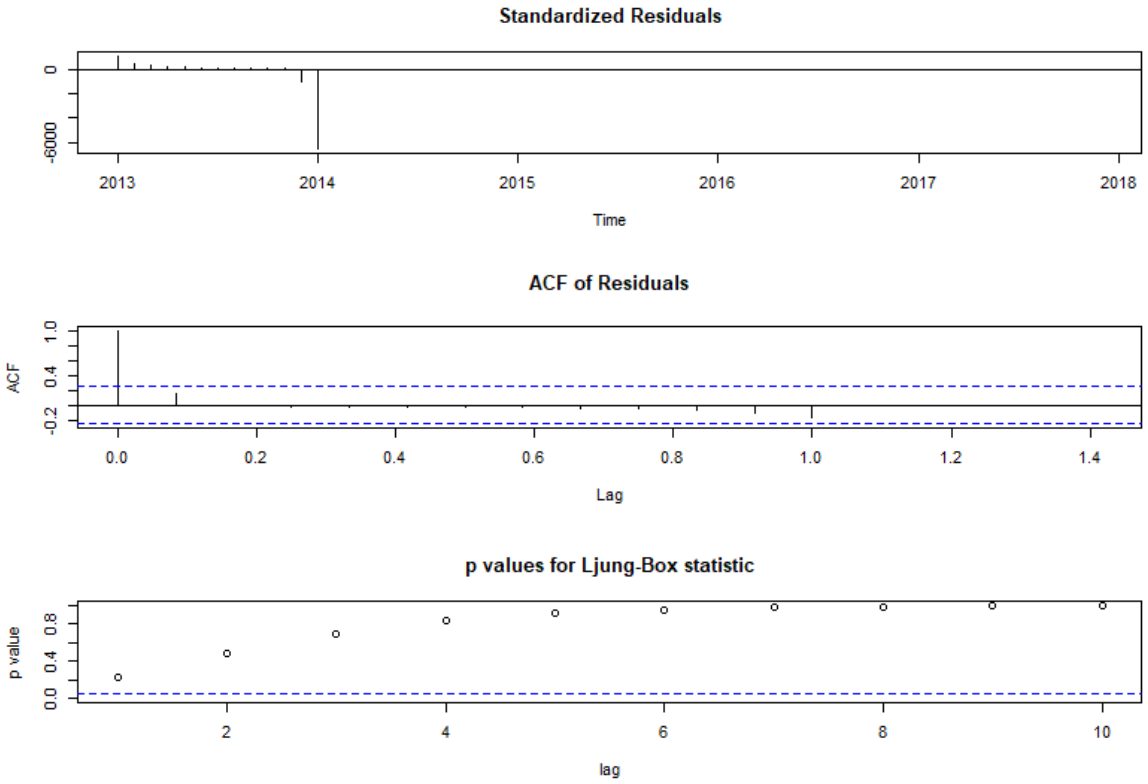
```
> fit03

Call:
arima(x = sdiffdata, order = c(2, 0, 2))

Coefficients:
          ar1      ar2      ma1      ma2  intercept
      -0.2143  -0.2964  -0.3603  -0.6396          0
s.e.   0.2426   0.0339   0.2464   0.2406        NaN

sigma^2 estimated as 6.559e-14:  log likelihood = 644.46,  aic = -1276.92
```

On notera aussi son AIC = -1276.92 pour pouvoir le comparer avec le reste des modèles retenus.

En comparant l'AIC des trois modèles retenus, on a le meilleur modèle est le troisième modèle : Fit03 - ARMA(2,2). Ceci est équivalent à la modèlisation de nos données transformées par Box-Cox (transformed-data) par le modèle SARIMA(2,1,2)x$(0,1,0)_{[12]}$.

Modèle SARIMA(2,1,2)x(0,1,0)$_{[12]}$

```
model <- arima(transformed_data, c(2, 1, 2),
               seasonal = list(order = c(0,1, 0), period = 12))
model
tsdiag(model)
```



On remarque que la normalité pose ici un problème lors du diagnostique mais ce n'est pas rééllement le cas. Le fait d'avoir utilisé un modèle SARIMA(2,1,2)x(0,1,0)$_{[12]}$ fait de sorte qu'on n'a pas les données pour prédire les 13 premières observations par le modèle ce qui nous donne un très grand écart entre les données réélles et les données prédites et donc de très grandes valeurs pour les 13 premiers résidus.

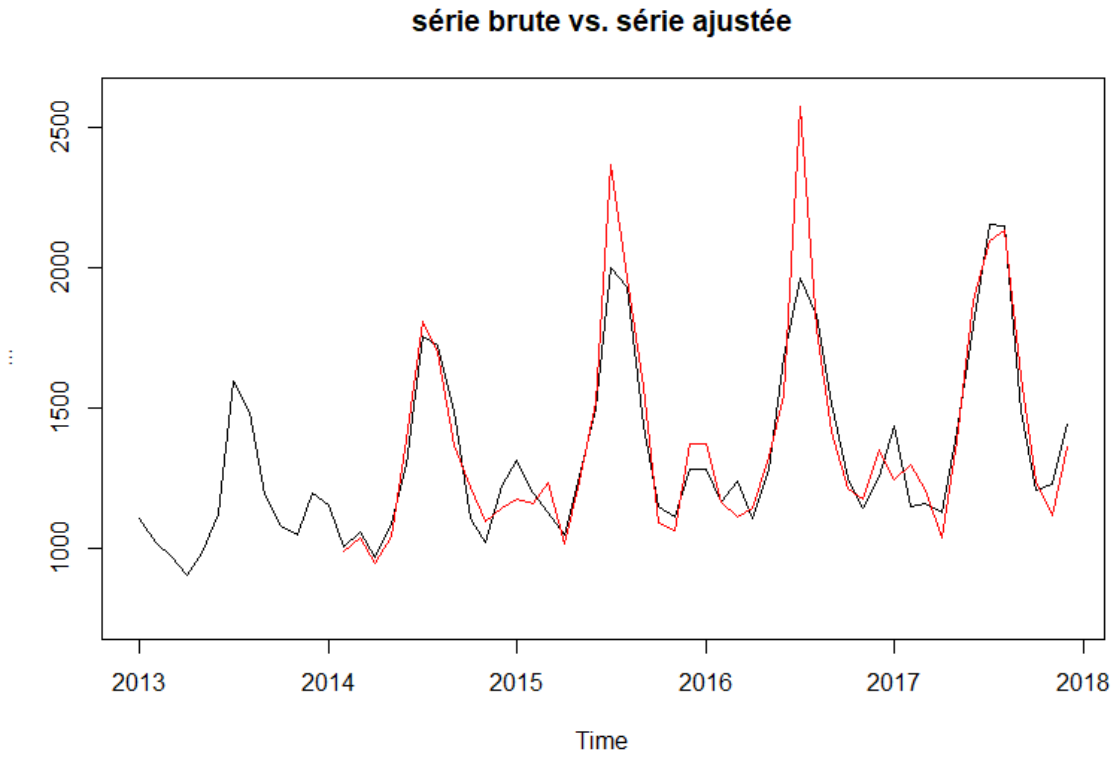
Pour pouvoir vérifier dans ce cas la normalité, on prendra nos résidus à partir de la 14 observation, on va les centrer et réduire et utiliser le test de Shapiro-Wilk pour pouvoir conclure :

```
> x = as.numeric(residuals(model))[14:60]
> shapiro.test(scale(x,center=TRUE,scale=TRUE))

        Shapiro-Wilk normality test

data:  scale(x, center = TRUE, scale = TRUE)
W = 0.97829, p-value = 0.5237
```

On a un p-value de 0.5237 pour le test de Shapiro-Wilk et donc on a bel et bien la normalité dans ce cas.

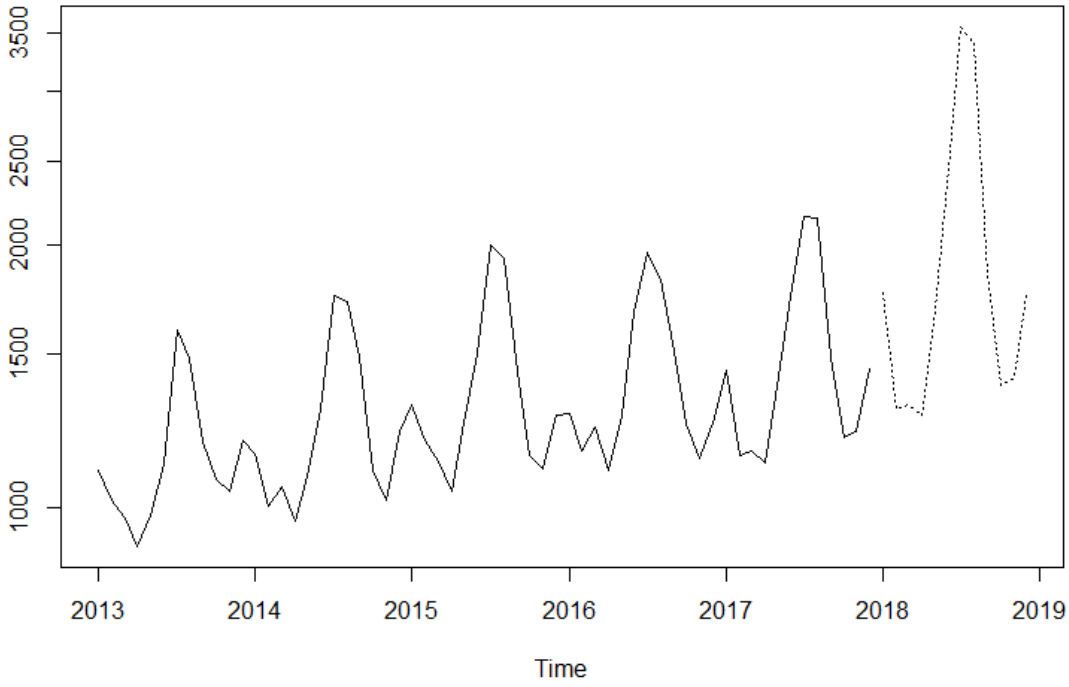
On représente alors les données réélles et prédite par notre modèle dans un graphe :

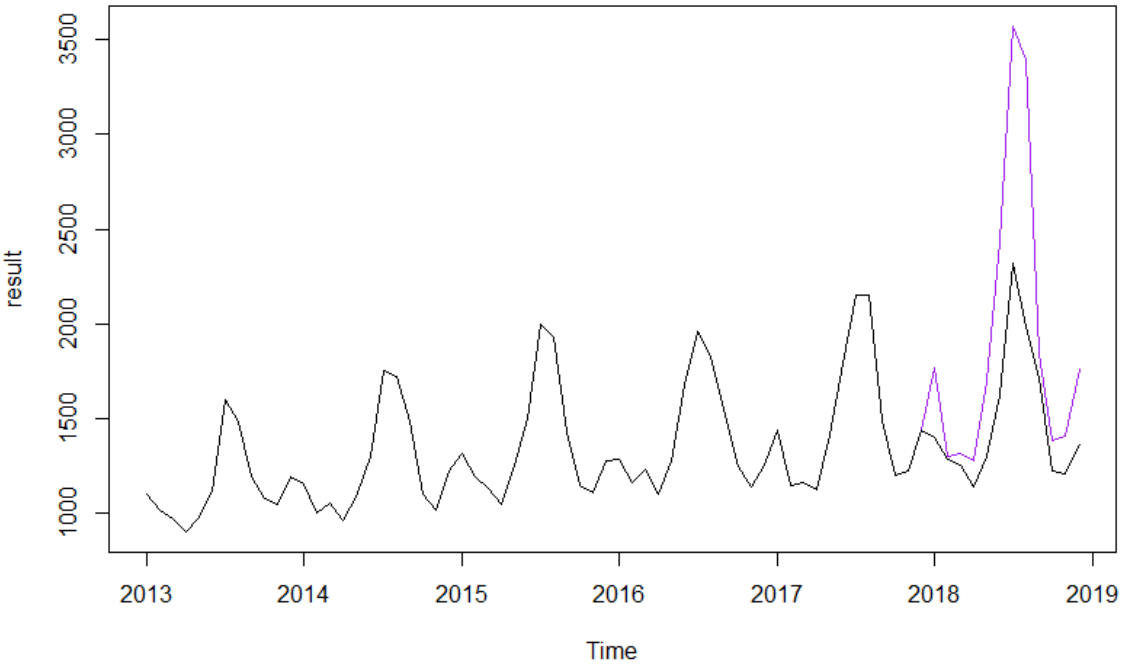
```
plot(train.data, main = "série brute vs. série ajustée", ylab = "...")
lines(fitted_data, col = "red")
```



Et enfin, on fait la prédiction des données de l'année 2018 en utilisant l'inverse de la transformation de Box-Cox sur les données prédites par notre modèle SARIMA(2,1,2)x$(0,1,0)_{[12]}$:

```
pred = predict(model, n.ahead = 12)
prediction = (pred$pred*lambda+1)^(1/lambda)

ts.plot(train.data,prediction, log = "y", lty = c(1,3))
```


3500
2500
2000
1500
1000
2013
2014
2015
2016
2017
2018
2019
Time


result
3500
3000
2500
2000
1500
1000
2013
2014
2015
2016
2017
2018
2019
Time

On utilisera par la suite ces valeurs prédites pour évaluer les performances de ce modèle et pouvoir le comparer au modèle de forêt aléatoire. On calcule pour cela le MSE :

```
> mean((datum-d)^2)
[1] 66621.89
```

On trouve que le MSE pour le modèle SARIMA est de : 66 621,89.

2 Modèle de Forêt Aléatoire

Random Forest ou Forêt Aléatoire est un algorithme d'apprentissage automatique supervisé qui fait partie de la famille des arbres de décision. Il s'agit d'une technique d'ensemble learning, qui combine plusieurs arbres de décision pour produire des prédictions plus précises.

Dans un Random Forest, plusieurs arbres de décision sont créés en utilisant des échantillons aléatoires de données d'entraînement et de variables. Chaque arbre de décision est construit en sélectionnant une variable aléatoire à chaque nœud de décision.

Une fois que tous les arbres de décision ont été créés, le modèle combine leurs prédictions pour produire une prédiction finale. Le résultat final est obtenu en prenant la moyenne des prédictions des différents arbres de décision pour la régression, tandis que pour la classification, la prédiction finale est déterminée par la majorité des prédictions des différents arbres de décision.

L'utilisation d'un Random Forest présente plusieurs avantages. Tout d'abord, il est résistant aux données manquantes et aux valeurs aberrantes. En outre, il est capable de gérer des ensembles de données avec un grand nombre de variables sans sur-ajustement.

Le Random Forest peut être utilisé pour la classification et la régression, ce qui en fait une technique d'apprentissage automatique très polyvalente mais non adaptée à notre étude directement.

Pour utiliser le Random Forest pour la prédiction de séries temporelles, il est généralement nécessaire de transformer le problème en un problème de régression. La transformation consiste à utiliser les observations passées pour prédire la prochaine observation.

Méthodologie

On va suivre les étapes suivantes pour modéliser nos données en utilisant l'algorithme de Forêt aléatoire :

- Les données de la série temporelle sont divisées en ensembles d'entraînement et de test pour permettre l'entraînement du modèle sur les données d'entraînement et l'évaluation de sa performance sur les données de test. Cela va nous permettre par la suite de comparer le modèle final retenu avec le modèle ARIMA en utilisant le MAPE.
- Les données de la série temporelle sont transformées en données adaptées à l'apprentissage supervisé en utilisant la technique de "Time Delay Embedding". Cette transformation permet de capturer des motifs temporels complexes qui ne peuvent pas être détectés à partir de la série temporelle originale.

- Les données d'entraînement sont divisées en ensembles d'entraînement et de validation afin d'évaluer différents hyperparamètres du modèle et de sélectionner ceux qui donnent les meilleures performances sur les données de validation. Cette étape est importante pour éviter le surajustement du modèle aux données d'entraînement et pour garantir une bonne généralisation du modèle aux nouvelles données.
- Différentes approches sont essayées pour améliorer les prédictions. Notamment, la méthode de prévision directe est utilisée au lieu de la méthode récursive pour limiter l'accumulation des erreurs de prévision. En outre, la tendance et la saisonnalité sont supprimées en utilisant les différenciations ∇ et ∇_{12} sur les données, ainsi qu'en utilisant la transformation de Box-Cox pour stabiliser la variance de la série temporelle.

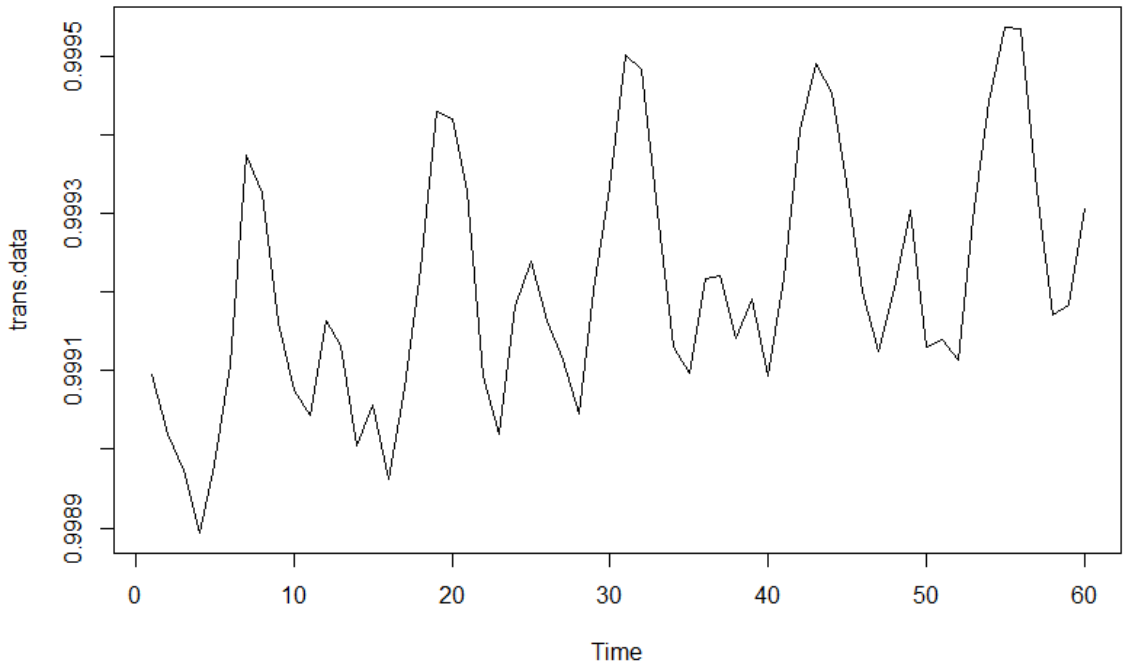
2.1 Méthode récursive

Nous allons commencer par effectuer les transformations de Box-Cox sur les données, tout comme nous l'avions fait lors de la dernière fois. Cette fois-ci, nous allons utiliser directement la fonction BoxCox.lambda pour déterminer la valeur de lambda à utiliser pour la transformation.

```
> lambda  = BoxCox.lambda(train.data)
> lambda
[1] -0.9999242
```

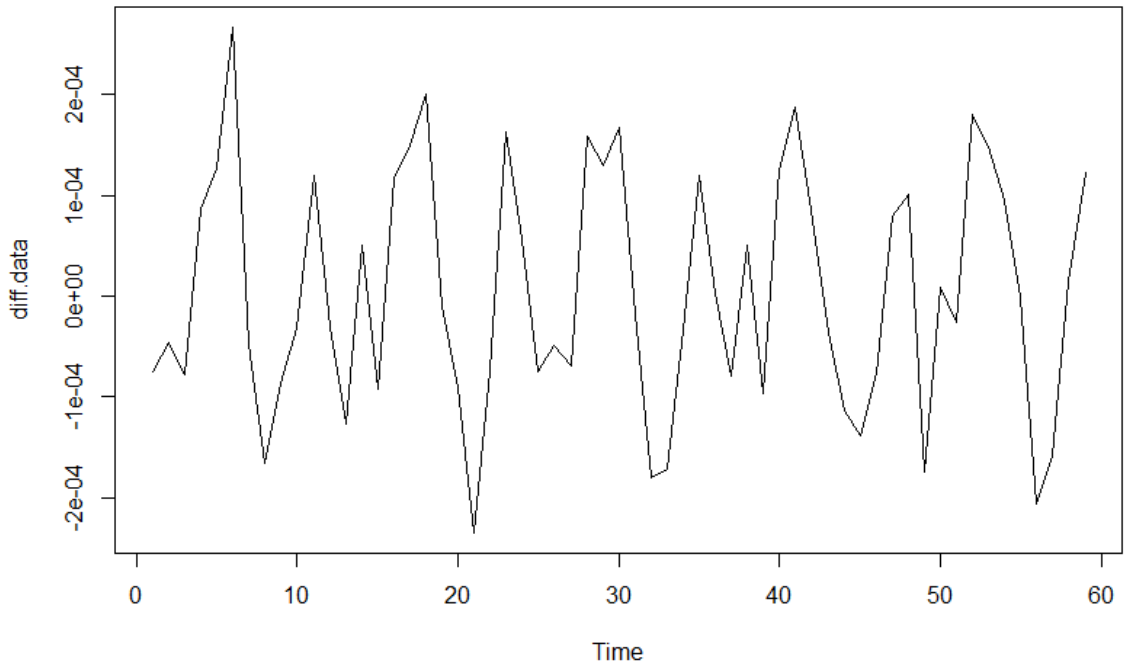
Nous avons obtenu un coefficient de -0.9999242 en utilisant la fonction BoxCox.lambda. Nous allons donc effectuer la transformation de Box-Cox avec ce coefficient en utilisant la formule : $y = 1 - \frac{1}{x}$. Voici les données transformées ainsi que le graphe correspondant.

```
> trans.data = 1-1/train.data
> trans.data
 [1] 0.9990951 0.9990194 0.9989728 0.9988943 0.9989807 0.9991071 0.9993736 0.9993264 0.9991609 0.9990740
[11] 0.9990424 0.9991626 0.9991316 0.9990052 0.9990556 0.9989627 0.9990789 0.9992291 0.9994294 0.9994190
[21] 0.9993276 0.9990928 0.9990196 0.9991823 0.9992385 0.9991635 0.9991138 0.9990442 0.9992027 0.9993328
[31] 0.9995001 0.9994821 0.9993020 0.9991305 0.9990976 0.9992175 0.9992205 0.9991403 0.9991905 0.9990931
[41] 0.9992179 0.9994046 0.9994896 0.9994527 0.9993388 0.9991996 0.9991235 0.9992029 0.9993043 0.9991299
[51] 0.9991385 0.9991130 0.9992929 0.9994413 0.9995358 0.9995348 0.9993290 0.9991698 0.9991837 0.9993060
```



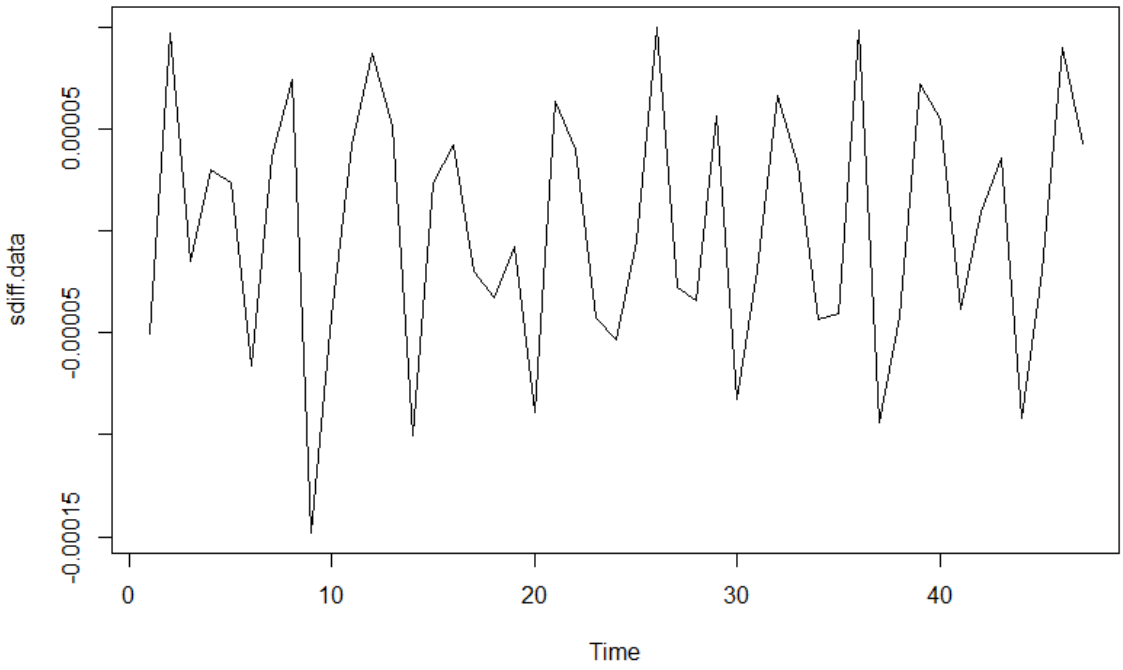
Comme nous l'avions mentionné précédemment, nous allons différencier les données avec un délai de 1. Voici les données différenciées ainsi que le graphe correspondant.

```
> diff.data = diff(trans.data)
> diff.data
 [1] -7.568894e-05 -4.663694e-05 -7.848407e-05  8.644135e-05  1.263272e-04  2.665274e-04 -4.721806e-05
 [8] -1.655099e-04 -8.687430e-05 -3.156757e-05  1.201986e-04 -3.097640e-05 -1.264698e-04  5.045066e-05
[15] -9.296816e-05  1.162796e-04  1.501736e-04  2.002777e-04 -1.041035e-05 -9.133554e-05 -2.348347e-04
[22] -7.319811e-05  1.626638e-04  5.622972e-05 -7.504144e-05 -4.967049e-05 -6.962962e-05  1.585192e-04
[29]  1.301206e-04  1.673254e-04 -1.799136e-05 -1.801652e-04 -1.714581e-04 -3.295602e-05  1.199730e-04
[36]  2.927868e-06 -8.015264e-05  5.024296e-05 -9.741054e-05  1.247595e-04  1.866901e-04  8.502894e-05
[43] -3.686915e-05 -1.139452e-04 -1.392475e-04 -7.605276e-05  7.937068e-05  1.013814e-04 -1.743459e-04
[50]  8.619995e-06 -2.552178e-05  1.798831e-04  1.484543e-04  9.444977e-05 -9.500984e-07 -2.058013e-04
[57] -1.591903e-04  1.389119e-05  1.222974e-04
```



Ensuite, nous allons différencier les données avec un délai de 12 pour enlever la saisonnalité. Bien que ce ne soit pas nécessaire pour utiliser l'algorithme de Forêt aléatoire, il est préférable d'avoir une série temporelle stable pour obtenir de meilleurs résultats. Voici les nouvelles données ainsi que le graphe correspondant.

```
sdiff.data = diff(diff.data, 12)
plot.ts(sdiff.data)
```



Cependant, nous sommes confrontés à un premier obstacle : nos données ne sont pas sous la forme d'un problème de régression ou de classification. Nous allons donc devoir utiliser une méthode appelée "time delay embedding", qui consiste à expliquer les données futures en utilisant les données passées. Cette transformation permettra de reformuler nos données sous la forme d'un problème de régression.

Le choix de la dimension de l'embedding est crucial. Si nous prenons une dimension trop grande, nous risquons de perdre de l'information car l'ajout de chaque dimension réduit le nombre d'observations. En revanche, si nous prenons une dimension trop petite, nous risquons de ne pas avoir suffisamment de variables explicatives pour expliquer les données futures.

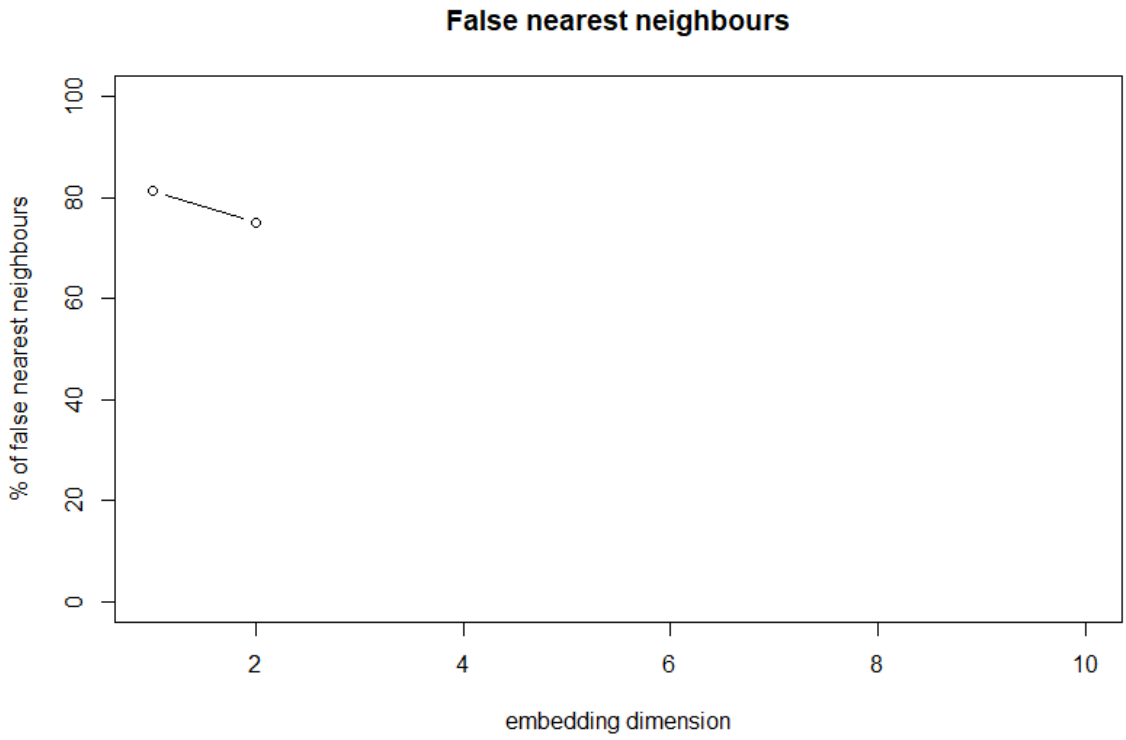
Pour trouver la meilleure dimension d'embedding, nous allons utiliser la méthode des "false nearest neighbors" (FNN). Cette méthode consiste à comparer les distances entre les points d'une série temporelle pour différentes dimensions d'embedding, afin de déterminer si les points proches dans l'espace d'origine restent proches dans l'espace d'embedding. Si les points proches se retrouvent éloignés dans l'espace d'embedding, alors cela indique qu'une dimension d'embedding plus grande est nécessaire. En revanche, si les points proches restent proches dans l'espace d'embedding, alors cela suggère que la dimension d'embedding actuelle est adéquate. La méthode FNN est une technique utile pour déterminer la dimension optimale de l'embedding, car elle permet de sélectionner une dimension qui préserve au mieux la structure géométrique des données.

La fonction en question a cinq paramètres qui doivent être spécifiés pour son utilisation. Le premier est "series", qui représente les données que l'on souhaite analyser. Ensuite, on a "m", qui est la dimension maximale de l'embedding que l'on souhaite tester. Le paramètre "d" représente le retard (ou décalage) que l'on souhaite utiliser pour créer l'embedding. Les deux derniers paramètres, "t" et "rt", correspondent respectivement au nombre minimum et maximum d'itérations pour lesquelles le critère de convergence doit être satisfait. Enfin, le paramètre "eps" correspond à une valeur seuil

utilisée pour déterminer si deux points sont considérés comme étant "proches" ou "loin" l'un de l'autre.

Dans notre cas, nous avons appliqué cette méthode sur les données de notre série temporelle en prenant les paramètres suivants: m = 10, d = 1, t = 10, rt = 10 et eps = écart-type de la série divisé par 10. La sortie de cette méthode nous donne un tableau de fraction et de total pour chaque dimension jusqu'à la dimension maximale que nous avons spécifiée. Les fractions et les totaux représentent le pourcentage de points qui ne sont pas des voisins faux dans l'espace m-dimensionnel. En utilisant cette méthode, nous avons trouvé que la dimension optimale pour notre série temporelle est de 3, ce qui signifie que nous pouvons expliquer les données futures en utilisant les données passées dans un espace tridimensionnel.

```
> fn = false.nearest(series = d, m=10, d=1, t=10, rt=10, eps = sd(d)/10)
> fn
              m1       m2       m3       m4       m5       m6       m7       m8       m9       m10
fraction   0.8125   0.7500
total    192.0000  16.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000
```



Après avoir appliqué la méthode de Time Delay Embedding sur les données, on les transforme en un data frame et on renomme les variables pour une meilleure lisibilité et interprétation des résultats.

```
> em.data = embed(sdiff.data, 3)
> head(em.data)
              [,1]          [,2]          [,3]
[1,] -1.448408e-05  9.708760e-05 -5.078089e-05
[2,]  2.983830e-05 -1.448408e-05  9.708760e-05
[3,]  2.384638e-05  2.983830e-05 -1.448408e-05
[4,] -6.624971e-05  2.384638e-05  2.983830e-05
[5,]  3.680771e-05 -6.624971e-05  2.384638e-05
[6,]  7.417434e-05  3.680771e-05 -6.624971e-05
```

```
> em.data = data.frame(em.data)
> head(em.data)
              X1            X2            X3
1 -1.448408e-05  9.708760e-05 -5.078089e-05
2  2.983830e-05 -1.448408e-05  9.708760e-05
3  2.384638e-05  2.983830e-05 -1.448408e-05
4 -6.624971e-05  2.384638e-05  2.983830e-05
5  3.680771e-05 -6.624971e-05  2.384638e-05
6  7.417434e-05  3.680771e-05 -6.624971e-05

> colnames(em.data)=c("yt","yt-1", "yt-2")
> em.data = data.frame(em.data)
> head(em.data)
              yt          yt.1          yt.2
1 -1.448408e-05  9.708760e-05 -5.078089e-05
2  2.983830e-05 -1.448408e-05  9.708760e-05
3  2.384638e-05  2.983830e-05 -1.448408e-05
4 -6.624971e-05  2.384638e-05  2.983830e-05
5  3.680771e-05 -6.624971e-05  2.384638e-05
6  7.417434e-05  3.680771e-05 -6.624971e-05
```

On utilisera par la suite y au lieu de em.data pour ne pas avoir à refaire tous les calculs quand on fait une transformation sur les données embedded.

```
y = em.data
```

La méthode récursive (ou régression récursive) est une méthode statistique utilisée pour prédire les valeurs futures d'une série chronologique. Elle consiste à ajuster un modèle de régression linéaire sur les observations passées et à utiliser ce modèle pour prédire la prochaine observation. Ensuite, on utilise cette prédiction comme une nouvelle observation pour ajuster à nouveau le modèle de régression, et ainsi de suite, jusqu'à ce que l'on atteigne la fin de la série chronologique. Cette méthode est appelée "récursive" car elle utilise une régression linéaire "récursive" pour prédire les valeurs futures en se basant sur les valeurs passées. La méthode récursive est souvent utilisée pour prédire des séries chronologiques saisonnières, où il y a une périodicité dans les données.

Dans le contexte de notre analyse, nous allons utiliser deux observations à la fois pour prédire la suivante. À chaque nouvelle prédiction, nous éliminerons la plus ancienne des deux observations précédentes et ajouterons la plus récente, puis nous continuerons le processus en utilisant les deux observations mises à jour pour faire la prédiction suivante. Ce processus sera répété jusqu'à ce que toutes les observations soient prédites dans notre cas 12.

Pour effectuer les prédictions, nous utiliserons ces données initiales.

```
> t = tail(em.data[,-3],1)
> t
              yt         yt.1
45 4.292669e-05 8.994395e-05
```

Les hyperparamètres sont des paramètres qui ne sont pas directement appris par l'algorithme, mais qui doivent être choisis avant l'apprentissage pour optimiser les performances du modèle. Les hyperparamètres peuvent affecter considérablement les performances du modèle et leur choix doit donc être soigneusement considéré.

La méthode du Hyperparameter Grid consiste à créer une grille (ou une liste) de combinaisons d'hyperparamètres à tester, et à entraîner le modèle pour chacune de ces combinaisons afin de trouver

la meilleure configuration. Par exemple, pour un algorithme de forêt aléatoire, les hyperparamètres peuvent inclure le nombre d'arbres dans la forêt, la profondeur maximale de chaque arbre, le nombre minimum d'échantillons requis pour diviser un nœud, etc.

Pour utiliser cette méthode, il faut d'abord définir une plage de valeurs pour chaque hyperparamètre à tester, puis créer une grille qui contient toutes les combinaisons possibles de ces valeurs. Ensuite, on entraîne le modèle pour chaque combinaison de la grille en utilisant une technique de validation croisée pour évaluer les performances du modèle.

Le choix de la meilleure configuration d'hyperparamètres est généralement basé sur la performance du modèle sur un ensemble de données de validation distinct de l'ensemble de données d'apprentissage. Une fois que les meilleurs hyperparamètres ont été trouvés, on peut entraîner le modèle sur l'ensemble de données d'apprentissage complet en utilisant cette configuration optimale.

Dans notre cas, les deux principaux hyperparamètres sont le nombre d'arbres de décision (ntree) et le nombre de variables choisies aléatoirement à chaque nœud de l'arbre (mtry). Notre grille contient deux valeurs possibles pour mtry (1 ou 2) et quarante valeurs pour ntree allant de 100 à 4000 par incréments de 100. Cela signifie que chaque combinaison possible de mtry et ntree sera évaluée pour déterminer les paramètres qui fournissent les meilleures performances pour l'algorithme de forêt aléatoire.

Une fois que chaque combinaison de paramètres est évaluée, on peut utiliser les résultats pour choisir les meilleurs paramètres pour l'algorithme. Les performances peuvent être mesurées à l'aide du MSE. Les meilleurs hyperparamètres seront ceux qui donnent le MSE le plus faible sur notre ensemble de validation.

```
> hyper_grid = expand.grid(mtry = c(1,2),
+                          ntree = seq(100,4000,100))
> head(hyper_grid,10)
   mtry ntree
1     1   100
2     2   100
3     1   200
4     2   200
5     1   300
6     2   300
7     1   400
8     2   400
9     1   500
10    2   500
```

Après avoir créé la grille, on procède à la création de nos données de validation. Ensuite, comme mentionné précédemment, nous évaluons chaque combinaison d'hyperparamètres dans la grille pour déterminer celle qui minimise le MSE.

```
> z1 = head(y,35) #apprentissage
> z2 = tail(y,10)[-1] #Prédicteurs pour la validation
> z2 = data.frame(z2)
> z3 = as.numeric(unlist(tail(y,10)[1])) # Valeurs réélles pour la validation
```

```
for (i in 1:80) {
  # création d'un modèle pour chaque hyperparamètre et calcul du MSE
  model = randomForest(yt ~ yt.1+yt.2,
                       data = z1,
                       ntree = hyper_grid$ntree[i],
                       mtry = hyper_grid$mtry[i],
                       nodesize = 1)

  y_actual = z3
  y_actual
  y_predicted <- as.vector(predict(model, newdata = z2))
  class(y_predicted)
  m = mean((y_actual - y_predicted)^2)
  m
  mse = c(mse,m)
}

mse
```

```
> mse
 [1] 2.142201e-09 2.148721e-09 2.168662e-09 2.064349e-09 1.996232e-09 2.018515e-09 1.935839e-09
 [8] 2.051269e-09 1.996691e-09 1.969043e-09 1.988810e-09 1.985688e-09 2.168834e-09 1.971147e-09
[15] 1.959317e-09 2.035233e-09 2.081618e-09 1.970600e-09 2.024095e-09 1.981644e-09 2.061166e-09
[22] 1.958975e-09 2.023185e-09 1.995818e-09 2.037635e-09 1.995321e-09 2.080356e-09 2.023891e-09
[29] 1.983658e-09 1.949178e-09 2.079895e-09 2.021575e-09 2.041198e-09 1.960450e-09 1.963818e-09
[36] 2.001968e-09 2.021887e-09 1.964517e-09 1.986487e-09 1.926395e-09 2.041301e-09 1.969185e-09
[43] 2.010853e-09 1.955795e-09 2.009542e-09 2.038722e-09 2.083937e-09 1.963668e-09 1.967562e-09
[50] 2.014271e-09 2.022201e-09 1.980230e-09 1.979023e-09 2.051593e-09 1.993395e-09 1.988969e-09
[57] 2.040826e-09 1.968633e-09 2.047048e-09 2.011793e-09 2.021561e-09 1.986906e-09 2.012541e-09
[64] 1.950623e-09 2.069915e-09 1.994769e-09 2.051873e-09 1.973711e-09 1.990709e-09 1.997728e-09
[71] 2.011493e-09 1.951734e-09 2.070600e-09 1.964132e-09 1.969533e-09 1.976088e-09 1.989623e-09
[78] 2.014094e-09 2.062990e-09 2.009018e-09
```

```
> j = which.min(mse)
> j
[1] 40
```

On crée, enfin, un modèle avec les hyperparamètres qu'on a trouvé.

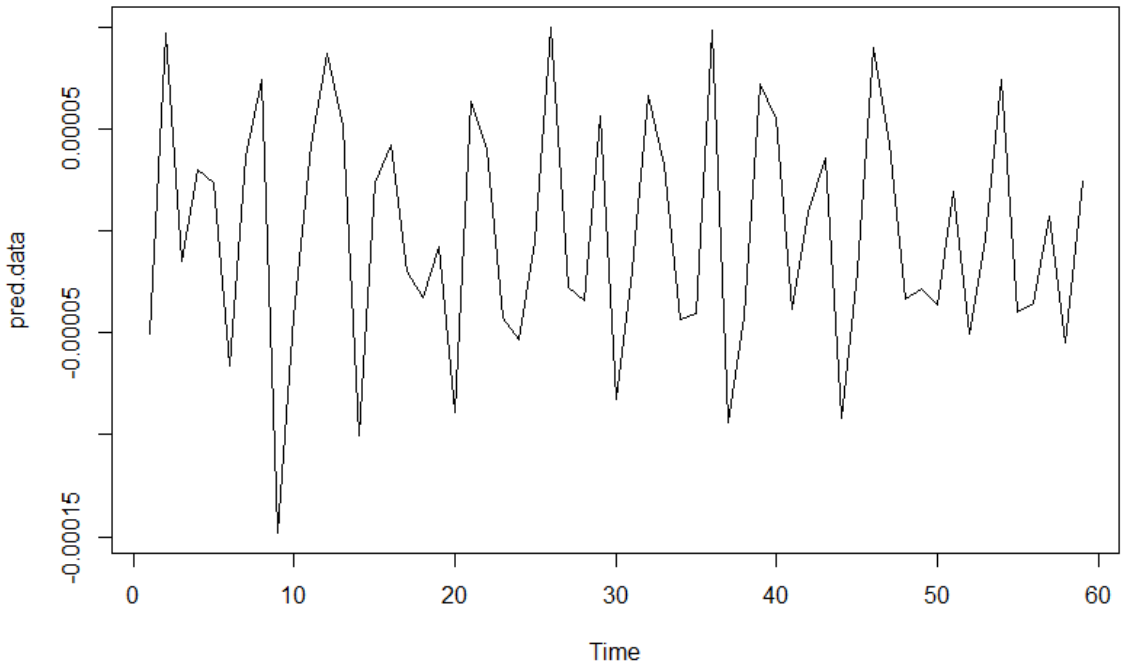
```
rd = randomForest(yt ~ yt.1+yt.2,
                  data = z1,
                  ntree = hyper_grid$ntree[j],
                  mtry = hyper_grid$mtry[j],
                  nodesize = 1)
```

Et on trouve les prédictions d'une façon récursive.

```
predictions = c()

for (i in 1:12){
  colnames(t)=c("yt.1","yt.2")
  predictions = c(predictions,predict(rd, newdata = t))
  t = cbind(tail(predictions,1),t[1])
  t
  predictions
}
```

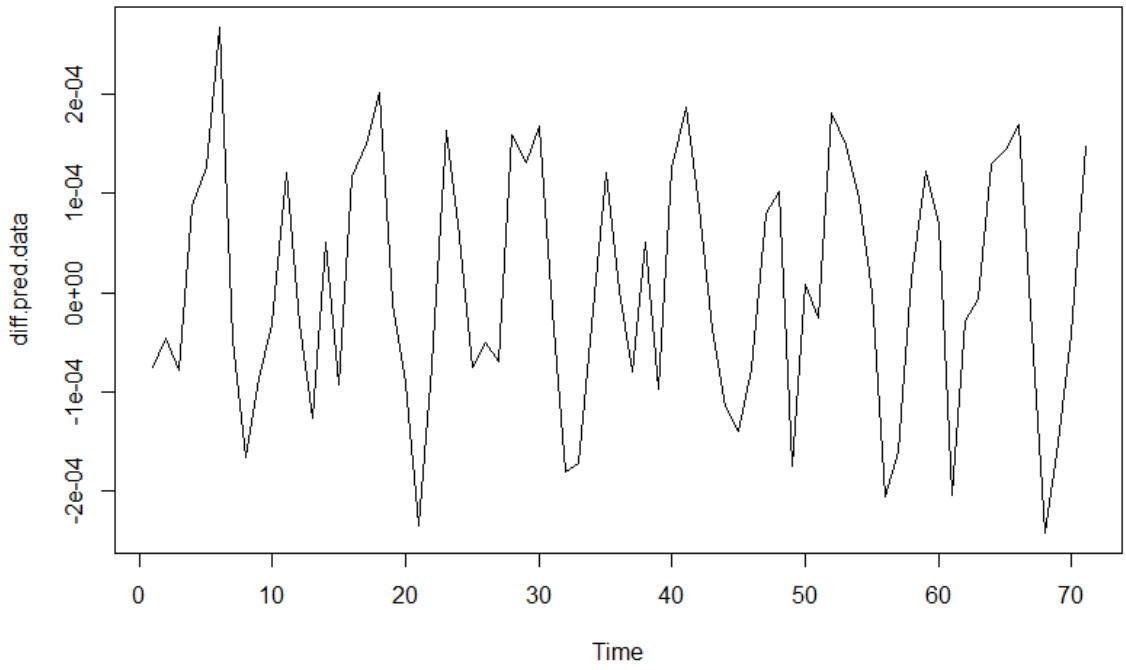
```
> predictions
           45            45            45            45            45            45            45
-3.301707e-05 -2.861651e-05 -3.630178e-05  1.937682e-05 -5.062025e-05 -3.772818e-06  7.433683e-05
           45            45            45            45            45
-3.985338e-05 -3.540318e-05  7.515759e-06 -5.471025e-05  2.412726e-05
```

```
pred.data = c(sdiff.data, as.vector(predictions))
plot.ts(pred.data)
```



Ensuite, nous appliquons l'inverse de la différenciation avec un délai de 12 sur nos données prédites. Après cela, nous traçons notre graphique pour visualiser nos prévisions.

```
diff.pred = tail(diff.data, 12)+predictions
diff.pred.data = c(diff.data, diff.pred)

plot.ts(diff.pred.data)
```



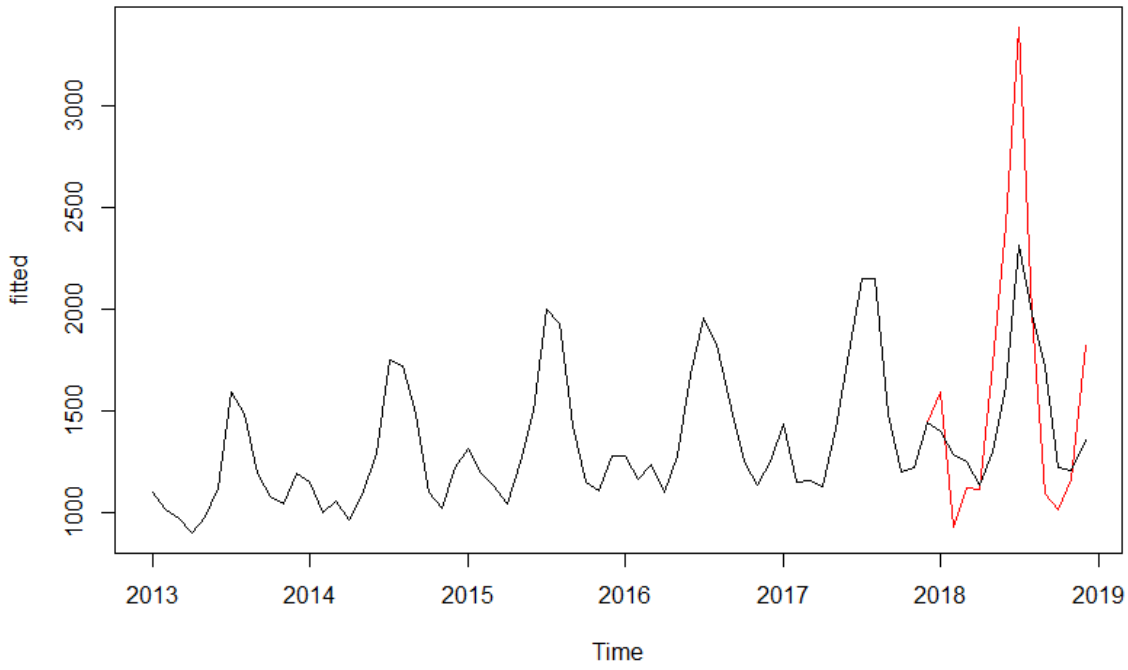
Ensuite, nous procédons à la reconstruction des données en inversant la différenciation avec un délai de 1.

```
pred = tail(trans.data, 12)+diff.pred
pred
datum = c(trans.data, pred)
```

Par la suite, il est nécessaire d'inverser la transformation de Box-Cox pour obtenir les prédictions en échelle d'origine. Ensuite, on compare les données réelles avec les données prédites par notre modèle récursif en traçant un graphe.

```
fitted = ts(1/(1-datum), start = 2013, frequency = 12)
fitted
initial = ts(data = d, start = 2013, frequency = 12)
```

```
plot(fitted, col = "red")
lines(initial)
```




```
> mean((datum-d)^2)
[1] 386754.7
```

On trouve que le MSE pour la méthode directe avec l'agorithme de Forêt aléatoire est de :386 754,7.

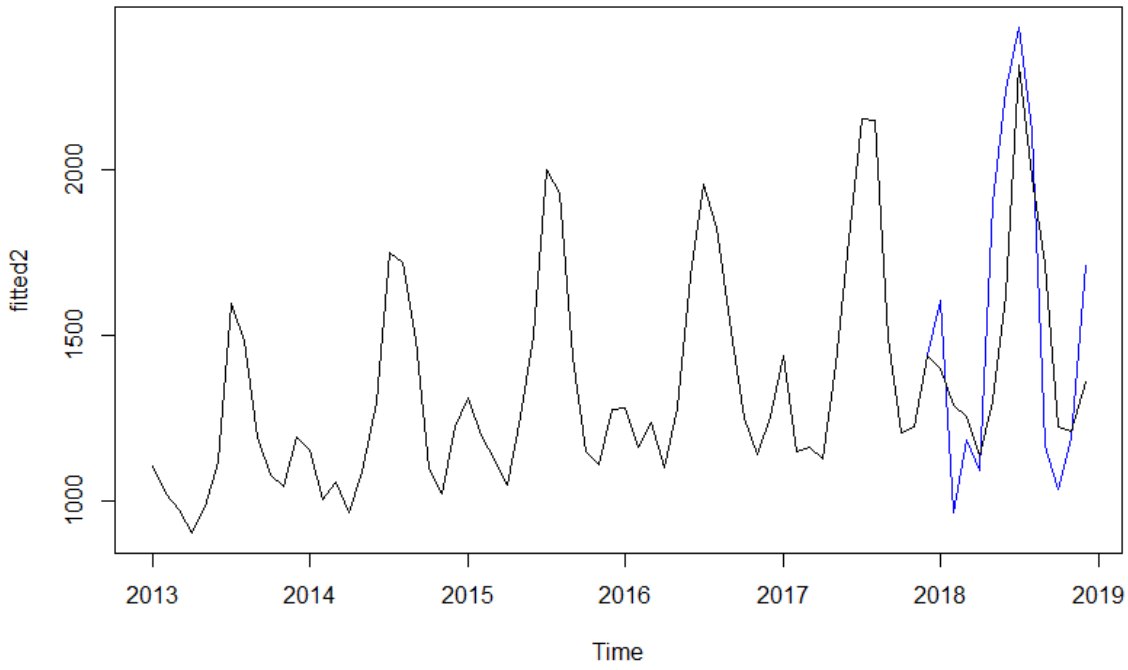
2.2 Méthode directe

La méthode directe consiste à créer un modèle pour chaque prédiction souhaitée, soit dans notre cas, 12 modèles, un modèle pour chacune des 12 prédictions. Contrairement à la méthode récursive qui utilise un seul modèle pour faire toutes les prédictions, la méthode directe aura un modèle pour chaque prédiction. Dans notre cas, nous avons choisi de chercher les meilleurs hyperparamètres pour chaque modèle, mais nous nous sommes limités à un maximum de 1000 arbres pour ne pas prendre trop de temps de calcul.

La méthode directe commencera par créer un premier modèle qui expliquera xt+1 par xt et xt-1, comme pour la méthode récursive. Mais contrairement à la méthode récursive, elle aura ensuite un deuxième modèle qui expliquera xt+2 par xt et xt-1, un troisième modèle pour xt+3 par xt et xt-1, et ainsi de suite jusqu'à la 12ème prédiction. Cette approche permet de ne pas avoir des résidus qui s'accumulent, mais prendra plus de temps car il faudra trouver les hyperparamètres pour chacun des douze modèles et ensuite utiliser chacun des modèles pour faire la prédiction correspondante. Dans tous les cas, on fera la prédiction en utilisant les deux dernières valeurs dont on dispose.

```
predictions = c()
y = em.data
for(i in 1:12){
  #on va prendre les meilleurs hyperparametres pour chaque modèle

  z1 = head(y,nrow(y)-10) #apprentissage
  z1
  z2 = tail(y,10)[-1] #validation prédicteurs
  z2 = data.frame(z2)
  z2
  z3 = as.numeric(unlist(tail(y,10)[1])) # validation valeurs réélles
  z3

  mse = c()

  for (i in 1:20) {
    # Fit model using hyperparameters
    model = randomForest(yt ~ yt.1+yt.2,
                         data = z1,
                         ntree = hyper_grid$ntree[i],
                         mtry = hyper_grid$mtry[i],
                         nodesize = 1)

    y_actual = z3
    y_actual
    y_predicted <- as.vector(predict(model, newdata = z2))
    class(y_predicted)
    m = mean((y_actual - y_predicted)^2)
    m
    mse = c(mse,m)
  }
  j = which.min(mse)
  j
  rd = randomForest(yt ~ yt.1+yt.2,
                            data = z1,
                            ntree = hyper_grid$ntree[j],
                            mtry = hyper_grid$mtry[j],
                            nodesize = 1)
  colnames(t) = c("yt-1","yt-2")
  t = data.frame(t)
  predictions = c(predictions,predict(rd, newdata = t))
  y = cbind(y[,1][-1],
            y[,2][-length(y[,2])],
            y[,3][-length(y[,3])])
  colnames(y) = c("yt","yt-1", "yt-2")
  y = data.frame(y)
}
```

Ensuite, on procède de la même manière pour inverser les prédictions obtenues avec la méthode directe et on trace le graphique montrant les données réelles comparées aux données prédites.




```
> mean((datum-d)^2)
[1] 19202.77
```

On trouve que le MSE pour la méthode directe avec l'agorithme de Forêt aléatoire est de : 19 202,77.

Conclusion

En conclusion, nous avons comparé les performances de SARIMA et de la forêt aléatoire sur des données temporelles. Nous avons utilisé les méthodes directe et récursive pour la forêt aléatoire, et avons constaté que la méthode directe a produit des résultats plus précis avec un MSE plus faible que la méthode récursive. Nous avons également constaté que la forêt aléatoire avec la méthode directe a produit des résultats meilleurs que SARIMA sur les mêmes données. Il est important de noter que ces résultats dépendent de la nature des données et que d'autres ensembles de données pourraient produire des résultats différents. Cependant, pour cet exemple particulier, la forêt aléatoire avec la méthode directe est une option solide pour la modélisation et la prédiction de séries chronologiques.