LUISS GUIDO CARLI

Department of Business and Management

Bachelor's Degree in Management and Computer Science
Course of Data Analysis for Business

Bachelor's Degree Thesis

Movie box office analysis and inference with time series models and tree-based methods

Supervisor

Prof. Francesco IAFRATE

Candidate

Simone DI GREGORIO

Student ID

241161

Academic Year 2021/2022

Summary

This thesis explores the entertainment industry in a purely quantitative way, combining time series data concerning box office revenue with movie-specific regressors in one hybrid model. Since time series analysis is a subset of data science that requires specific tools, a big chunk of the text is dedicated to the introduction of the basic concepts of the subject, which paves the ground for more complex topics, such as time series modelling with ARIMA and Exponential Smoothing. The last part of the thesis is concerned with a quick review of two of the main tree-based methods, before using them to model the variability not induced by time. Each of the steps in each of the chapters is assisted by abundant visualization, in order to aid interpretability. The analysis was completely done exploiting the possibilities offered by the R programming language, while the scraping was done with Python; details regarding the scripts can be found in the appendix.

Table of Contents

List of Tables

List of Figures

Chapter 1

Entertainment and Analytics, this thesis and how it came to be

1.1 Analytics at the core of the entertainment industry?

Generally speaking, data is a powerful source of value, a key to understanding present, past and future, and most importantly a compass for the direction of most if not all economic investments, especially for customer facing sectors. It is no surprise that the entertainment industry, one of the most marketing driven and customer facing environments, with huge amount of capital invested in high risk projects, has moved towards this direction, mainly due to the huge push brought by the overall *servitization* trend.

Services by definition build their supremacy by exploiting data: they forge their user experience through data (recommendation systems), they select, upgrade and produce their content based on data, they try to infer the reaction of consumers to each and every single step. More briefly, they create value through data. Great giants in the entertainment industry have been walking this path for decades now, and a giant such as Netflix has seen its success due to this kind of behaviour.

Although the current situation is quite eloquent, we do not have to specifically talk about streaming and web-driven services to see the importance of analytics for entertainment, and specifically for the movie industry. Test screenings, web advertising, the focus on the comfort zone of franchises and the extreme caution when giving green light to a new project are all signs of an active or passive attention to data, which has always been latent in every step of production and

distribution, but more so with an exponential growth of data sources.

In particular, there is a publicly available source of data which has always established the ultimate fate of a production, the king of traditional entertainment data and the most widely discussed piece of data on social networks and press coverage: gross boxoffice revenue, of which BoxOffice Mojo and The Numbers are the number one holders in aggregate form. A proof of the value given by such a basic metric, especially when combined in a complete dataset with other essential pieces of data, can be given by the price at which Amazon lists the access to the dataset combining IMDb (market leader for movie data) and BoxOffice Mojo (both owned by Amazon): 400'000$ for just twelve months.

Even though we cannot hold such an important set of data, there is still a lot of information to be extracted from a subset of it (although noisy), a subset that has to be built through extensive work. It is thus necessary to retrieve data from the Internet (through APIs, for example, as we did) and to manipulate it and clean it in order to give it value by making it appropriate for models, descriptive statistics approaches and visualization.

In the next two sections we will first deep dive into how this thesis came to be, before starting to go technical by introducing the overall structure of the pipeline and by explaining how data was retrieved.

1.2 How was this thesis born?

This thesis stems from a project carried over for the Business and Marketing Analytics Course with Professor Villarroel, supported by KNIME, a company developing an integrated development environment designed to deal with every task in a data science pipeline with almost no code. The project was a collaboration among A. Brizzante, G.Iadisernia, A.Ballerino, F.Buzzi, F.Cardarelli and me.

The workflow we presented has a lot in common with what I am presenting here, but the content of this thesis steps up some of the overall complexity and presents a different approach in the methodology. More specifically, the KNIME workflow ignored the time scope (which is instead the most important focus of this thesis) and employed just a linear regression to test linear relationships between the covariates, together with their interactions, and the response, with the help of some advanced feature engineering, assisted by deep learning and cloud services.

The notable work we did can be retrieved (and freely explored and reutilized) by visiting the KNIME Hub, a repository of documentation, components, extensions and workflows for the KNIME Analytics Platform.

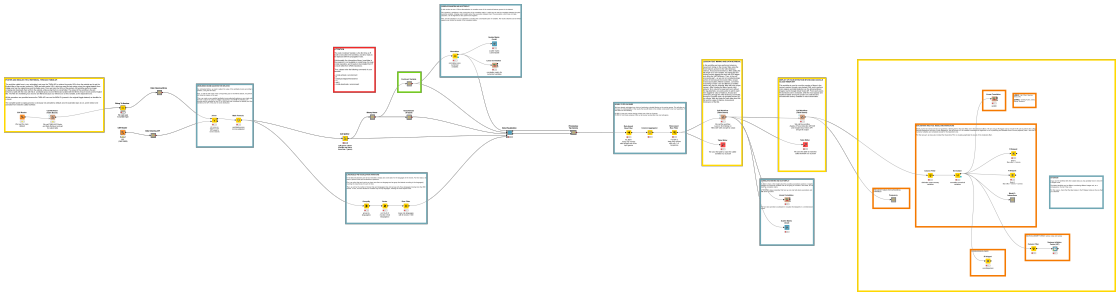

Figure 1.1: Above the main KNIME workflow of the project we have built. The workflow calls various sub-modules and represents the complete pipeline.

1.3 Introduction to the overall structure of the pipeline of this thesis

This thesis is basically structured in a two-fold way, with two different approaches which unfold one after the other, resulting completely complementary.

On one side, the first step in the pipeline regards time series modelling, trying to capture the information coming just from the temporal progression over months and years of the average of the box office revenue (in the United States market). To achieve this, a monthly time series of averages of inflation-adjusted box office is computed from the overall dataset, thoroughly analyzed (and visualized) and modelled with **ETS** (Exponential Smoothing) and **ARIMA** (AutoRegressive Integrated Moving Average). This will pave the way to the introduction of fundamental concepts of time series and time series analysis, such as autocorrelation and stationarity.

On the other hand, we use tree-based methods to capture movie-specific information stemming from residuals. In this particular case, the residuals are in this way defined:

$$r_i = y_i - f(i) \tag{1.1}$$

In the above equation r_i is the residual, y_i is the the gross US box office revenue of the movie and $f(i)$ is the function that maps the movie index i to the predicted average (the forecast) of the US box office for the month of release. In training, the residual is thus calculated and used for fitting the tree-based methods; in *inference mode* the equation, explicit with respect to y_i, is used for making predictions, once we have the predictions from the time series models and the ones from the tree-based methods. It goes without saying that in training $f(i)$ represents fitted values for ARIMA and ETS[1], while in inference $f(i)$ represents actual forecasts.

[1]Fitted values for time series models are simply one-step-ahead forecasts.

To be clear, the overall idea is thus the one of isolating the *time effect* affecting box office outcomes and then modelling the residual part, thus performing two different analysis at two different levels. After all, at the least, there is a strong seasonality in theatrical attendance, and we will see that in the data; it would thus be really naive not to consider the information stemming from time in the analysis.

Similar hybrid models to the one just presented and illustrated can be found across the literature, with two particularly related examples in Guo et al.(2021) [1] and in Kaytez (2020)[2].

Of course, all of this was made possible by scraping the necessary data and by building the dataset (before manipulating it). Before concluding this chapter we indeed briefly deal with scraping and how it was tackled for this thesis.

1.4 Scraping and data retrieval

The initial data for this work comes from a Kaggle[2] dataset built by scraping IMDb[3] (dataset which is now offline following a takedown request). The table has a lot of interesting features, such as *original title*, *duration*, *director*, *genre*, *synopsis*, and so on. However, the most important feature was the least semantically significant, the *IMDb ID*, meaning the identification code of the movie for the IMDb URLs.

IMDb ID	Original Title	Genre	Duration	Country	Director
tt1345836	The Dark Knight Rises	Action, Adventure	164	UK, USA	Christopher Nolan
tt1074638	Skyfall	Action, Adventure, Thriller	143	UK, USA, Turkey	Sam Mendes
tt0145487	Spider-Man	Action, Adventure, Sci-Fi	121	USA	Sam Raimi
tt1663202	The Revenant	Action, Adventure, Drama	156	USA, Hong Kong, Taiwan	Alejandro G. Iñárritu
tt0253474	The Pianist	Biography, Drama, Music	150	UK, France, Poland, Germany, USA	Roman Polanski
tt0317705	The Incredibles	Animation, Action, Adventure	115	USA	Brad Bird
tt0382932	Ratatouille	Animation, Adventure, Comedy	111	USA	Brad Bird, Jan Pinkava
tt0383574	Pirates of the Caribbean: Dead Man's Chest	Action, Adventure, Fantasy	151	USA	Gore Verbinski
tt3498820	Captain America: Civil War	Action, Adventure, Sci-Fi	147	USA	Anthony Russo, Joe Russo
tt4154796	Avengers: Endgame	Action, Adventure, Drama	181	USA	Anthony Russo, Joe Russo

Table 1.1: A very small sample of rows and columns from the original Kaggle Dataset. Notice the ID from IMDb.

This ID allowed me to reliably make REST GET calls for additional information to another - far more relaxed in its ToS - service, TMDB (The Movie Database/themoviedb.org), which offers a quite powerful and useful API.

TMDB is a totally crowd-sourced service, so I do not expect its data to be entirely accurate, but it was nonetheless an extremely precious database for this thesis and for the project we talked about in section 1.2. Most importantly, as

[2]Kaggle (kaggle.com) is a well known website for data scientists where people post datasets, share scripts and partake in competitions for specific tasks.

[3]IMDb (imdb.com) is the most important website/database for movie data, for consumers and professionals alike. It is owned by Amazon, like Box Office Mojo.

I have already explained, its API supports the *IMDb ID* as an identifier, and it thus allowed me to complete the information from the original dataset with the information contained into the TMDB structure.

The scraping was done through Python scripts and mainly consisted of extracting the English title (useful for text mining), which was missing, and the actual US original theatrical release date, since the ones present in the original dataset were not accurate at best and we needed them to be fairly reliable due to the monthly granularity of the analysis.

Additionally, the API was used to get the TMDB URL for the poster of each movie, and a complete set of images was thus downloaded. Images are not part of the analysis of this thesis, but they could enter the pipeline, for example through a convolutional neural network.

Most of the retrieval was not particularly difficult thanks to a good API documentation, with some careful handling needed for some specific steps. The code used for scraping can be found in Appendix A.1, where it is also briefly explained and commented (in addition to the comments already present in the code).

Chapter 2

Building the Time Series and Introduction to Time Series Analysis

2.1 Building the time series

In order to build the time series we need to use the original US theatrical release dates that were scraped from TMDB according to the documentation of the API and the related structure of the JSON response to the call. Even by being careful there are still multiple release dates associated to the same movie. I attempted to solve this by looking at the *note* field left by users that uploaded the release date for the movie on TMDB. This field specifies some additional pieces of information regarding the release date, and can be used for disambiguation. In my case, I simply used a set of manually written Regular Expressions to capture the notes that evidently highlighted mistakes, thus eliminating the related dates. The set of *multiple release dates* that escaped this filtering were simply eliminated altogether, since it was impossible to understand which were the correct ones.

Additionally, since we are dealing with time series involving economic data, it makes sense to make it more stationary (a concept we will focus on later on) by eliminating the trend induced by monetary inflation; this also allows to make the scale of the single gross box office revenues uniform.[3] To do this, we use a very important index for Macroeconomics, the Consumer Price Index (CPI), which carefully tracks prices over a basket of consumer goods.

The CPI time series we used comes from the US. Bureau of Labor Statistics[1],

[1]The CPI time series we have used has ID CUUR0000SA0 with base period 1982-84.

and refers to *CPI-U*, the index for all urban customers; it can be easily retrieved from the website of the Bureau.

Once retrieved the time series, to adjust for inflation at movie level a simple operation suffices, with y_i as usual representing the US gross box office revenue for the movie i and CPI_t as the index for the month t the movie was released in:

$$y_{i_{adj}} = \frac{y_i}{CPI_t} * 100 \tag{2.1}$$

Since we do not have enough data from movies from previous years (in particular, we have sparse box office data), this analysis starts from 1983. By choosing 1983 we also avoid a difficult to understand spike in 1982, probably due to the inflation adjustment and to the fact that with the end of the Great Inflation there might have been lags in the reaction of the overall economy towards monetary stability. Additionally, since the time series from 2017 onward seems particularly erratic and absolutely difficult to capture when compared to most of the previous years, we consider it up until December 2016.

Lastly, to avoid biased averages computed on underpopulated subsets of data, the mean for each month is not computed when having less than 10 movies to compute it with. This resulted in 6 gaps over a total of 400[2], a negligible set which was imputed with linear interpolation.

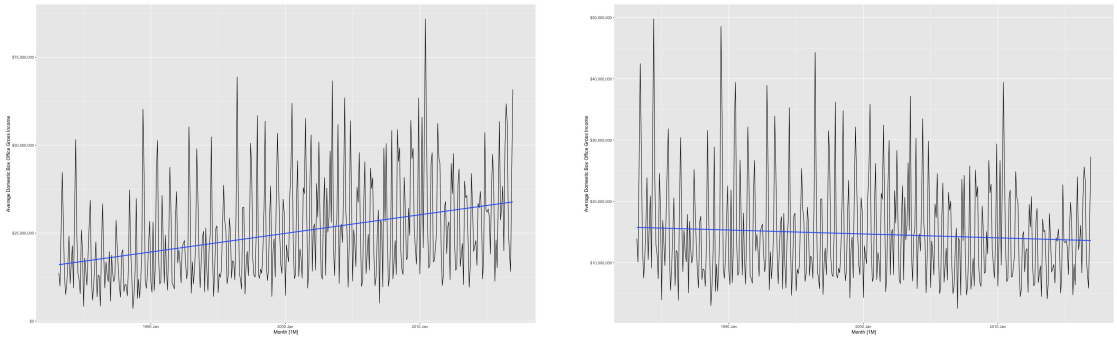

Figure 2.1: Side by side the same time series, but one adjusted with CPI for inflation (on the left), the other one not adjusted (on the right). Thanks to the regression line coloured in blue, it can be clearly seen that inflation induces an exogenous upward trend in the time series.

[2]The gaps make up just 1.5% of the total.

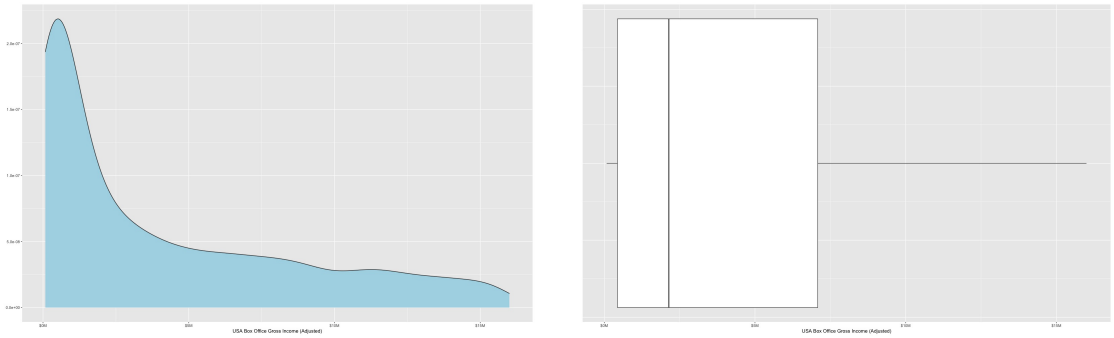

Figure 2.2: A closer look to the distribution of inflation-adjusted US gross box office revenue in the dataset. The plots show an extremely skewed to the left and leptokurtic distribution, with a skewness of 4.3 and an excess kurtosis of 29.1, indicating a distribution dominated by many outliers which here were in part left out in order to help the visualization.[3]

2.2 Fundamental concepts of time series analysis

Before continuing, in this paragraph we explain some important concepts of time series analysis, in order to introduce terminology and concepts discussed in the next chapter. This section is structured in subsections, one for each macro-topic tackled.

Firstly, the concept of components and decomposition is visualized and presented with the STL algorithm; secondly, autocorrelation and white noise are explored; thirdly, the concept of stationarity is briefly explained in order to make the elements behind ARIMA understandable.

2.2.1 Components and Decomposition

To introduce formally the concept of time series, we first have to talk about what a time series is made up of. First and foremost, a time series embodies three/four different components.[4]

The trend component It exists when there is a long-term increase or decrease in the data. The trend can *change direction* over time when shifting from increasing to decreasing, or viceversa. It is the general direction that the time

[3]Skewness and kurtosis are the third and fourth standardized central moment of a distribution. Excess kurtosis is the deviation from the kurtosis of a normal distributon, which has kurtosis equal to 3. *Mesokurtic*, *platykurtic* and *leptokurtic* relate to null, negative and positive excess kurtosis, respectively.

series takes macroscopically and does not need to be linear. The inflation trend eliminated with Consumer Price Index is a good example of a trend behaviour.

The seasonal component It is the periodic component, induced by seasonal patterns over the so-called fixed *seasonal period*[4], which can be thought in exactly the same way as the length of a period for a periodic function such as the well known trigonometric ones. The pattern related to the seasonality may (relatively slowly) change over time, for example in size. An example of seasonal behaviour of a time series may be the peak around Christmas of our box office time series, which is particularly seasonal.

The cycle component This component is related to seasonality, but it has one important difference, it is not anchored to a fixed period and the related fluctuation thus do not have a fixed frequency. Examples may be macroeconomic or business cycles, which can be delimited in time only ex-post. Due to the closeness between the concepts of cycle and trend, cycle is usually combined with trend when analyzing the time series.[3]

The remainder component It is just what cannot be explained by the decomposition in season and trend-cycle. This component is similar to model residuals, something typically associated to random noise.

These components can be added to assemble the series in an additive or in a multiplicative way, depending on whether the size of the seasonal fluctuation increases with the level of the series.[3]

$$Y_t = S_t + T_t + R_t \qquad (2.2) \qquad\qquad Y_t = S_t \cdot T_t \cdot R_t \qquad (2.3)$$

In order to extract and visualize the components given a time series, various types of algorithms exist. Since pure decomposition is not a focus of this thesis, we will focus on just one, **STL**, which stands for *Seasonal-Trend decomposition procedure based on LOESS*[5].

LOESS is local linear regression, or scatterplot smoothing, if you prefer, which is basically a local least square fit with weights assigned by a kernel (which normally uses some function of the Euclidean distance) to observations in the neighborhood of a point[5].[6] STL uses a mix of moving averages and LOESS to

[4]In some cases, there can be more than one seasonal pattern, one for each seasonal period. For example, when having daily data, we have three possible seasonal period, at week, month and year level.

[5]LOESS is a memory-based model (it needs the training observations in memory, like K-NN) and is a quite computationally expensive one, since each prediction requires one different local fit.

perform the decomposition, which can be handled only in an additive sense. A multiplicative decomposition can be forced[6] by log-transforming the response and back-transforming the components by exponentiation.[3]

Month	Average seasonal component (AD)	Average seasonal component (ML)
January	-1.12	-1.08
February	-0.34	-0.36
March	-0.38	-0.34
April	-0.93	-0.94
May	0.75	0.65
June	1.56	1.62
July	1.39	1.45
August	-0.36	-0.39
September	-1.16	-1.12
October	-0.96	-0.97
November	0.5	0.47
December	1.04	0.99

Table 2.1: Average seasonal component for each month calculated with multiplicative (ML) and additive (AD) decomposition, with standardization to make the entries comparable.

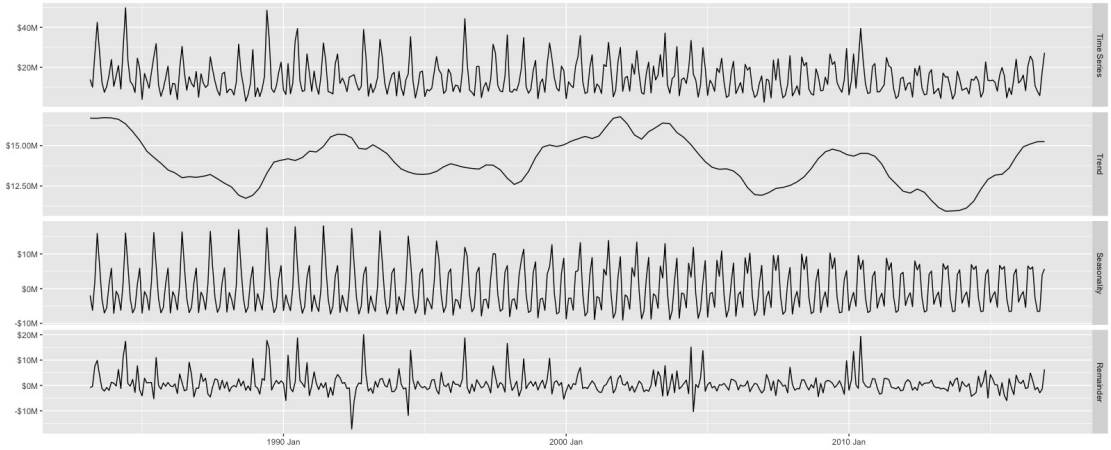

Figure 2.3: This additive STL decomposition of the US box office time series is plotted with ggplot and feasts on R. The default STL options are used, with *seasonal_window* set to 13 and *trend_window* set to 21.[3]

[6] $Y_t = S_t \cdot T_t \cdot R_t \iff \log Y_t = \log(S_t \cdot T_t \cdot R_t) = \log S_t + \log T_t + \log R_t$
This means that the additive decomposition of a log-transformed series can be considered equivalent to a multiplicative decomposition.

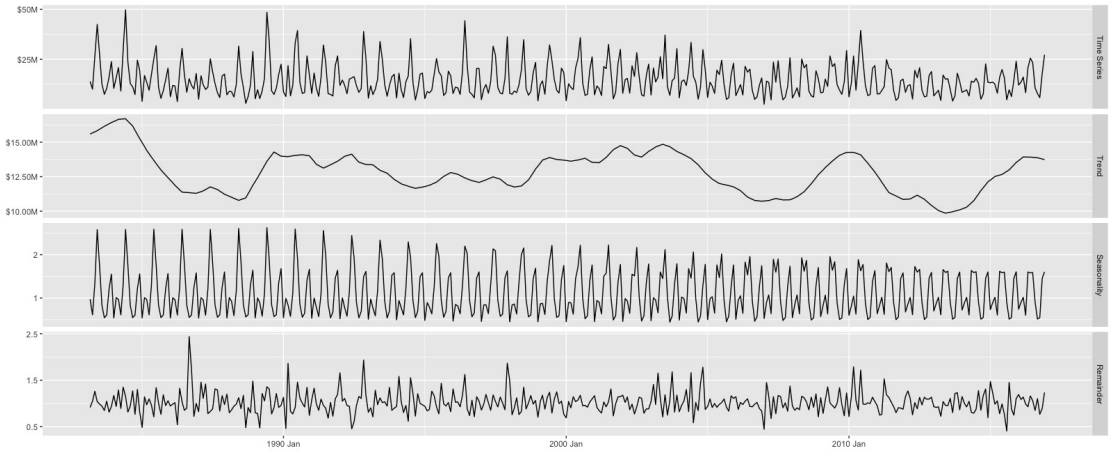

Figure 2.4: Notice that in this multiplicative decomposition seasonality and remainder do not have an unit of measure, and this makes sense by a dimensional analysis perspective, since they are just factors that multiply the trend component.

By looking at table 2.1 and at images 2.3 and 2.4, we can make this decomposition useful and link it to real-world scenarios and circumstances. We can see that, contrary to what happens in Italy (with distributors usually postponing releases to the autumn), summer is particularly strong for theatrical attendance, with additive and multiplicative seasonal components showing their highest values in June and July, at a level far higher than Christmas season itself! Notable is also the fact that following both peaks, the one around Christmas and the one in the summer, two deep troughs immediately follow, in January and in September[7], respectively.

2.2.2 The autocorrelation function

Autocorrelation is a powerful analytic tool for time series analysis which stems in a quite straightforward way from the concept of simple correlation. Where correlation acts as a proxy for covariance and measures the extent of the linear relationship between two distinct random variables, autocorrelation measures the extent of the linear relationship between a signal/a time series and a *lagged* version of itself. When we say *lagged* time series here we mean a negatively translated version of the original time series.

To introduce this concept and autocorrelation it may be best to present the backshift operator in advance, which will be particularly useful later on: [7]

$$By_t = y_{t-1} \qquad (2.4) \qquad\qquad B^k y_t = y_{t-k} \qquad (2.5)$$

[7]August is weak, but as far as we can see it is definitely not a trough

The backshift operator - applied over the whole series - thus returns the lagged time series. With backshift notation, sample autocorrelation is in this way defined, which is simply the sample Pearson's correlation coefficient using the concept of lag:

$$r_k = \frac{\sum_{t=k+1}^{T}(y_t - \bar{y})(B^k y_t - \bar{y})}{\sum_{t=1}^{T}(y_t - \bar{y})^2} \tag{2.6}$$

As you can see by the index k, the autocorrelation coefficient varies as the *amount of lag* changes, defining[8] the autocorrelation function (ACF), which is indeed a function of k:

$$f(k) = \sum_{l=1}^{\infty} r_l \, [\![l = k]\!] \tag{2.7}$$

Autocorrelation can be considered a proxy for the amount of information in the time series (trend and seasonality can be clearly seen with the ACF, for example), and we thus expect the residuals of a time series forecasting model to be white noise (gaussian, for ARIMA and ETS[9]), namely a time series having its real autocorrelations equal to 0.

The autocorrelation function plotted in 2.5 shows a strong seasonality highlighted by the peaks at lags multiple of 12, which is the length of the seasonal period. The negative peaks are due to the fact that the are two *maximums* in the seasonal pattern of the series: around December, and at a beginning of the summer. Lags that are around odd multiples of 3 are related to (weaker) negative autocorrelations due to the fact that there are two peaks in each seasonal period, so a shift brought by a lag of about 3 or 9 moves the index from a peak towards a trough.

Furthermore, the amplitude of the ACF remains approximately the same over the lag axis, suggesting the absence of a strong trend, when adjusting for inflation, and this can also be seen in Figure 2.3 and in Figure 2.4, by looking at the modest scale of the trend oscillation.[4]

When checking if a time series is truly white noise, the autocorrelations are obviously affected by fluctuations due to the sample estimation, so we cannot expect to find autocorrelations equal or very close to zero for every and each lag. An hypothesis test approach to test the null hypothesis of a white noise autocorrelation is given by a boundary of $\pm\frac{2}{\sqrt{T}}$ for a 5% significance level, with T being the length of the time series.[4] The 5% significance level implies a 5% probability of a Type I error (false positive), and thus we can expect for a white noise series about 5%

[8]In the equation defining the ACF, the square brackets are Iverson brackets. If the condition contained between them is satisfied, their value is 1, otherwise it is 0.

[9]For what concerns ETS, the gaussianity is related to the error considered, which can be additive (absolute) or multiplicative (relative). With multiplicative errors, gaussianity does not hold for the absolute residuals, of course.

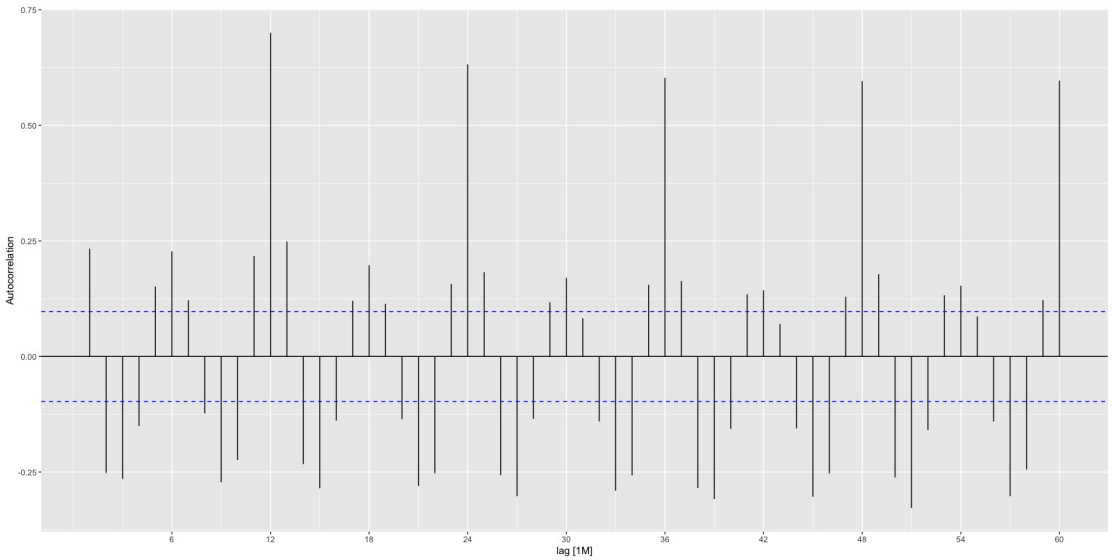

Figure 2.5: The plot of the autocorrelation function is called (auto)correlogram, this is the one for the US box office time series. The dashed blue line is basically a boundary for the hypothesis test with the time series being white noise as null.

(asymptotically, as per the weak law of large numbers) of calculated autocorrelations above the threshold.

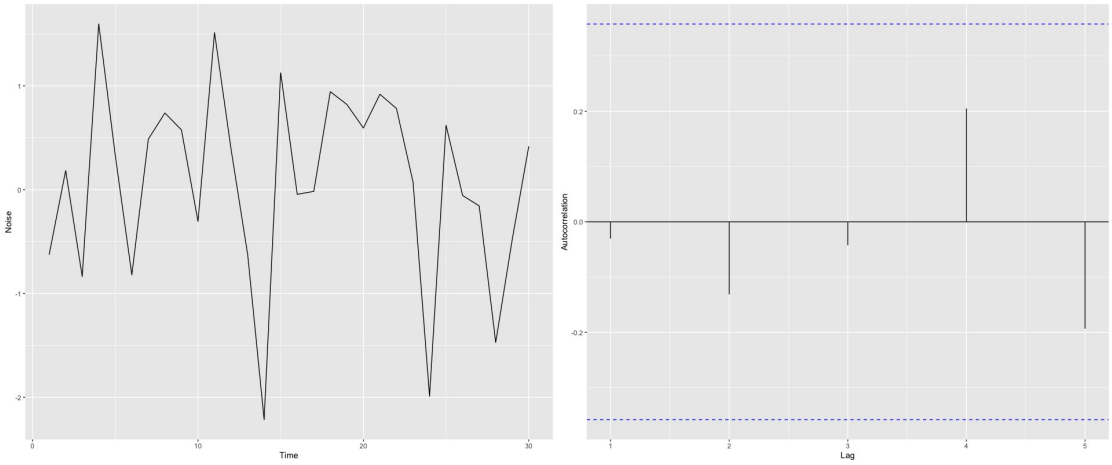

Figure 2.6: Time plot and autocorrelogram of a gaussian white noise time series, with each point in the series coming from a standard normal distribution. Notice how some of the autocorrelations are positive, but all quite low if not close to being null.

In Figure 2.5 the threshold is represented by the two dashed blue lines, and

we can therefore confidently say that the time series is not white noise. On the contrary, the autocorrelogram in Figure 2.6 shows autocorrelations which are all far lower than the threshold, highlighting the fact that they come from a white noise series.

A more formal way to test for white noise over many time lags is given by the Box-Pierce test and the Ljung-Box Test where we test the white noise null hypothesis by comparing the test statistics with a χ^2 distribution with degrees of freedom given by the number of lags considered[10]. The two test statistics, for Box-Pierce (left) and Ljung-Box (right) are the ones shown below, with r_k being the autocorellation for lag k[8]:

$$Q = T\sum_{k=1}^{l} r_k^2 \qquad (2.8) \qquad Q = T(T+2)\sum_{k=1}^{l}(T-k)^{-1}r_k^2 \qquad (2.9)$$

A careful eye can probably notice that the threshold $\pm\frac{2}{\sqrt{T}}$ we use when plotting the autocorrelation function in Figures 2.5 and 2.6 comes directly from the Box-Pierce test. If we test a single autocorrelation, $Q \sim \chi_1^2$ under the null hypothesis, and for a one sided test - such as this one - with a 5% significance level the threshold for the rejection of the null hypothesis is 3.84. Now, looking at the equation of the test statistic: $Q = Tr_k^2 \iff r_k = \pm\frac{\sqrt{Q}}{\sqrt{T}}$. Since $\sqrt{3.84} = 1.96 \approx 2$, the boundary for rejection becomes $\pm\frac{2}{\sqrt{T}}$.

It should be stressed that both the Box-Pierce and the Ljung-Box are so-called *portmanteau* tests, meaning that they have a narrowly defined null hypothesis and a very broadly defined alternative hypothesis. In this case, specifically, the null hypothesis is that true value of each and every autocorrelation is zero, while the alternative hypothesis, being the negation, is that at least one of the autocorrelations has a non-null value. Statistically speaking, the strong rejection of the null hypothesis for our US box office time series, as can be seen in Table 2.2, only allows us to state that at least one of the autocorrelations is truly non-null.

Box-Pierce test	Box-Pierce p-value	Ljung-Box test	Ljung-Box p-value
749.1	≈ 0	780.7	≈ 0

Table 2.2: The test statistics and the p-values for both the Box-Pierce and the Ljung-Box tests for the US box office time series. Notice that in both cases the null hypothesis of every autocorrelation being null is strongly rejected.

[10]More precisely, the degrees of freedom of the χ^2 decrease by the number of parameters estimated in the model whose residuals we are testing, in the case we are performing residual diagnostics. This is not the case here since we are dealing with raw data, with the numbers of parameters being thus 0.

2.2.3 Stationarity

This last part of this chapter deals with stationarity, which is an important property that needs to be enforced on the data in order to fit an ARIMA model.

A definition of stationarity is not simple and somewhat technical but can be given by looking at a sequence of an arbitrary number of observations in a time series as a realization of a multivariate distribution. Considering $\mathbf{X_t} = [X_t, ..., X_{t+s}]^\mathsf{T}$ as the random vector of such a multivariate distribution, if the series is stationary, for any s the distribution does not depend on t.[7] This has a lot of implications, such as mean and variance stability and the fact that the autocovariance/autocorrelation function is stable, meaning that the covariance between two points in a stationary series depends only on the lag separating them.

To give some examples, a gaussian white noise series is (strictly) stationary since each point in time is simply a random realization of a constant normal distribution, with any sequence of observations[11] being thus modelled by the same multivariate normal. Series with trend and seasonality are not stationary since the joint distribution of a sequence of points depends on time[12]. On the contrary, cyclic time series can be stationary, since cycles do not have fixed periods and thus the distribution of the points in the series is independent from time. As explained before, variance also needs to be specifically taken into account, due to the fact that heteroscedasticity can be easily present in a time series, for example when variance linearly increases with the level of the series.[7]

A natural way to solve heteroscedasticity and stabilize the variance is to log-transform the series, while the impact of trend and seasonality is normally greatly reduced by stabilizing the mean through *differencing*, which - as easily understood by its name - is an operation related to derivatives. Differencing consists in building a new time series made up of differences between points of the original one and their lagged equivalent. When the lag is equal to 1 we say that we are performing *first differencing* and we normally use that to tackle trend; when the lag is equal to the seasonal period we are performing *seasonal differencing* and we normally use that to tackle seasonality.[7]

Formally speaking and using the backshift operator, first differencing (left) and seasonal differencing (right) are this way defined, with a new observation y_t for each time index t:[13]

$$y'_t = (1 - B)y_t \qquad (2.10) \qquad\qquad y'_t = (1 - B^k)y_t \qquad (2.11)$$

Of course, if a time series presents both trend and seasonality, we need to apply

[11] Given same length.

[12] The mean vector of the multivariate distribution is not constant.

[13] k is the length of the seasonal period.

both types of differencing, while in some cases it may be useful to apply the same kind of differencing more than once in order to get stationarity, thus performing an higher degree operation. In fact, if the trend is quadratic two first differencing operations are required, if the trend is cubic three, if linear one, and so on. This stems from a quite trivial fact: for a trend induced by a polynomial of degree one the first order derivative is constant, for a trend induced by a polynomial of degree two the second order derivative is constant, for a trend induced by a polynomial of degree n the nth order derivative is constant. Since first differences compute the mean derivative in intervals between points in the discrete sequence over time, a constant derivative implies a constant value (when factoring out noise) for the differenced series and thus a stable mean. Therefore, differencing operations can be easily stacked when needed, with their order not changing the final result.

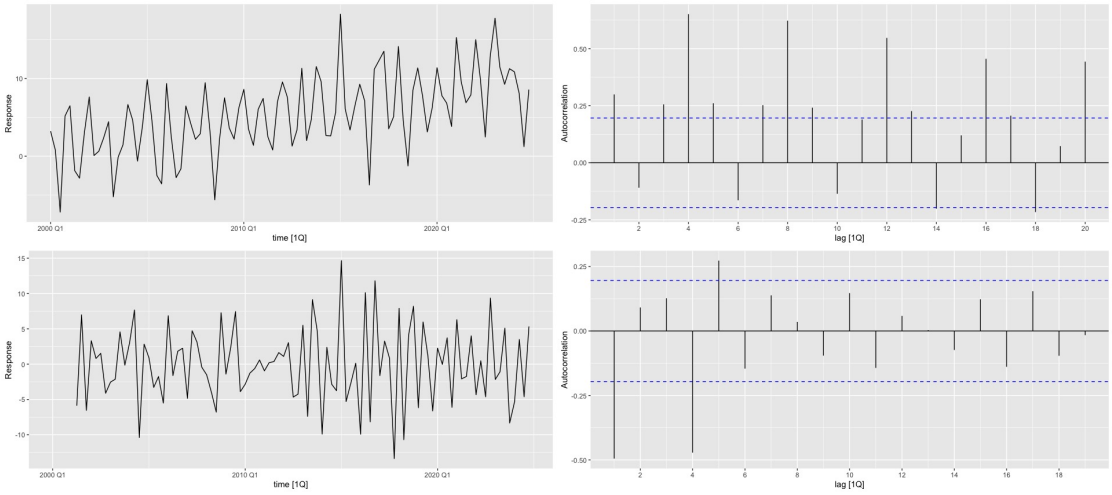

Figure 2.7: In the top two quadrants we can see an artificially generated time series (with linear trend, guassian noise and sinusoidal seasonality) and its autocorrelation function; in the other two the result of seasonal and first differencing is visualized.

Chapter 3

ARIMA and Exponential Smoothing models

3.1 Exponential Smoothing

One of the two time series models we are going to explore in detail in this chapter, Exponential Smoothing (or ETS) is a latent variable model/method which tries to forecast and explain time data according to latent states, trend, level and seasonality (not necessarily all together). Latent variables are variables that are not directly observable, but that can be inferred through observations, and that in turn explain those observations.

Each ETS[1] model, is defined by measurement equations that describe the actual observed data and by state equations which describe how the latent states evolve over time[9]. For example, a ETS(A,N,N), a model with additive error (A) which does not consider trend or seasonality (N, N), is simply described by the following two equations, where l_t is the level state at time t and ε_t is a gaussian error:

$$y_t = l_{t-1} + \varepsilon_t \qquad \text{Measurement equation} \tag{3.1a}$$

$$l_t = l_{t-1} + \alpha\varepsilon_t \qquad 0 < \alpha < 1 \qquad \text{State equation (level)} \tag{3.1b}$$

α is a smoothing parameter - bounded between 0 and 1 - tuned during training with maximum likelihood estimation/squared residuals minimization, and which determines how strongly the level state reacts to the current observation. Notice that due to the way the state equation is defined, the older the observations get, the less they are considered to explain and predict data, and the higher the smoothing parameter, the more this behaviour is dominant. Due to the restriction on the

[1]ETS can stand both for Exponential Smoothing and for Trend, Error and Seasonality.

values of α, the weights of past observations always decrease exponentially with time, and this is the reason for the *Exponential Smoothing* name. There is also another parameter to be estimated during training, l_0, which is the starting value of the level component.

Now it is time to step up the complexity, also more appropriate for the needs of a time series like the one we are dealing with, describing monthly US box office data. In order to do this we switch to a ETS(A,A,A) model, which includes errors, trend and seasonality in an additive way. The number of equations necessary to describe the model obviously increases, as the number of estimated parameters (and thus the degrees of freedom)[9]:

$$y_t = l_{t-1} + b_{t-1} + s_{t-m} + \varepsilon_t \qquad \text{Measurement equation} \quad (3.2a)$$

$$l_t = l_{t-1} + b_{t-1} + \alpha\varepsilon_t \qquad 0 < \alpha < 1 \qquad \text{State equation (level)} \quad (3.2b)$$

$$b_t = b_{t-1} + \beta\varepsilon_t \qquad 0 < \beta < \alpha \qquad \text{State equation (trend)} \quad (3.2c)$$

$$s_t = s_{t-m} + \gamma\varepsilon_t \qquad 0 < \gamma < 1 - \alpha \qquad \text{State equation (seasonality)} \quad (3.2d)$$

m is the seasonal period

As it was for the level, the state equations for trend and seasonality need the estimation of the initial states b_0 and s_0 in addition to the state-specific smoothing parameters β and ε.

For ETS models with multiplicative errors we just exploit residuals relative to the forecast[2] and update the equations accordingly. To give a clearer idea, below are the equations for a ETS(M,A,M) model, which considers both seasonality and errors in a multiplicative way[9]:

$$y_t = (l_{t-1} + b_{t-1})s_{t-m}(1 + \varepsilon_t) \qquad \text{Measurement equation} \quad (3.3a)$$

$$l_t = (l_{t-1} + b_{t-1})(1 + \alpha\varepsilon_t) \qquad 0 < \alpha < 1 \qquad \text{State equation (level)} \quad (3.3b)$$

$$b_t = b_{t-1} + \beta(l_{t-1} + b_{t-1})\varepsilon_t \qquad 0 < \beta < \alpha \qquad \text{State equation (trend)} \quad (3.3c)$$

$$s_t = s_{t-m}(1 + \gamma\varepsilon_t) \qquad 0 < \gamma < 1 - \alpha \qquad \text{State equation (seasonality)} \quad (3.3d)$$

m is the seasonal period

Multiplicative seasonality may cause problems of numerical instability[3] when combined with additive errors, leading to a general avoidance of ETS(A,.,M) models.[4] Additionally, multiplicative errors lead to instability when the forecasts

[2] $\varepsilon_t = \frac{y_t - \hat{y}_{t|t-1}}{\hat{y}_{t|t-1}}$

[3] The state equations are vulnerable to divisions by values close to zero.

[4] Numerical instability also affects some specific models with multiplicative trend, specifically, ETS(AMA) and ETS(MMA).

are prone to values close to zero, while in general multiplicative solutions do not really make sense when the response uses an interval scale and can thus reach negative values by exploiting the absence of an absolute zero.

Also notice that both multiplicative error and multiplicative seasonality are strictly related to what we have said in Subsection 2.2.1 when explaining multiplicative decomposition, since they allow the absolute effect of residual and seasonal components to vary according to the level of the series.

For what concerns point forecasts, the seasonal state remains fixed at the last update from observed data, with the trend component inducing the only long-term change in the forecast data. However, having a linear trend over the forecast horizon is not usually something desirable, since for $t \to \infty$ the limit of the forecast is not defined, approaching positive or negative infinity. A way to solve this is to use a damped trend where an exponentially decreasing factor multiplies the (absolute or relative) step-wise increase over the forecast horizon due to trend; this makes the forecast converge for $t \to \infty$ without exploding to (plus or minus) infinity. This dampening is regulated by parameter ϕ.

Now, among the many possibilities, how to choose the best ETS combination? We can do that by exploiting the maximized likelihoods of the models through information criteria like Akaike Information Criterion (AIC) or Bayesian Information Criterion (BIC). AIC or BIC are quite useful criteria for model evaluation because they take into consideration the log-likelihood by adjusting its value for the number of parameters estimated; this is done with a linear penalty, which is much higher for BIC. To perform the calculation, the number of parameters considered is just the number of smoothing parameters and the number of initial states (plus the sample variance of the residuals). We pick the model which returns the lowest AIC or BIC.

3.2 AutoRegressive Integrated Moving Average, ARIMA for friends

An AutoRegressive Integrated Moving Average[10] model, as the name states, is the union between two different kinds of models, Autoregressive models and Moving Average models, with the word *integrated* meaning the opposite operation to the *differencing* we defined in Subsection 2.2.3. In order to make things clearer, it makes sense to analyze each of this building blocks in a separate way.

3.2.1 Autoregression

An autoregressive model owes its name to the fact that it is a multivariate linear regression onto a random variable using as predictors the past values of the random

variable itself.[7] How much of the past is considered for each observation is defined by the order of the autoregression, defined by the parameter p, following the convention for ARIMA models. Formally speaking and using the backshift operator (see Equation 2.4), an autoregression of order p, $AR(p)$, is this way defined, with c as the intercept:

$$y_t = c + y_t(\phi_1 B + \phi_2 B^2 + \phi_3 B^3 + ... + \phi_p B^p) + \varepsilon_t \tag{3.4}$$

The ϕ parameters normally have constraints needed to enforce stationarity on the model: for a AR(2) model, the constraints are the following:

$$-1 < \phi_1 < 1, \;\; \phi_1 + \phi_2 < 1, \;\; \phi_2 - \phi_1 = 1 \tag{3.5}$$

3.2.2 Moving Average models

The rationale behind moving average model is very similar to the one behind autoregressive ones, with the only (big) difference that now the model is a regression onto its same *past* errors.[7] Again, the number of past errors considered is determined by the order of the autoregression, encoded by q, with $MA(q)$ being a moving average of order q. With the backshift operator:

$$y_t = \varepsilon_t(1 + \theta_1 B + \theta_2 B^2 + \theta_3 B^3 + ... + \theta_q B^q) \tag{3.6}$$

Interestingly enough, a $AR(p)$ model can be seen as a $MA(\infty)$, whatever the order of the autoregression and given stationarity constraints we talked about above in Subsection 3.2.1. The inverse holds only if a similar set of parameter restraints holds for the $MA(q)$ model.

3.2.3 ARIMA models and integration

The union between the models we have described in Subsections 3.2.1 and 3.2.2 gives birth to an ARMA model, which is a model suited only to stationary data, as we have made clear in Subsection 2.2.3 and in Subsection 3.2.1 with the enforcement of stationarity conditions on the autoregression.

Now, how can we factor in non-stationary behaviour into an ARMA model? We do that by simply making the series stationary with the *differencing* operation we talked about in Subsection 2.2.3 and by reversing this transformation when needed for predictions with the inverse operation, called *integration*, whose I completes the ARIMA name. The model is thus obtained with maximum likelihood estimation given an already *differenced* and stationary time series, but the actual predictions are given with respect to the non-differenced data.

A formulation of a non-seasonal ARIMA model can be given this way, using the backshift operator, given (p, d, q) hyperparameters[5][7]:

$$(1-\phi_1 B-\phi_2 B^2-...-\phi_p B^p)(1-B)^d y_t = c+\varepsilon_t(1+\theta_1 B+\theta_2 B^2+...+\theta_q B^q) \quad (3.7)$$

Equation 3.7 also allows us to give a more defined meaning to the stationarity and invertibility conditions we talked about in Subsections 3.2.1 and 3.2.2. First, we notice that we have two polynomials (called characteristic polynomials) in B of order p and q defined between round brackets in Equation 3.4. The stationarity constraints impose that the p inverse complex roots[6] of the autoregressive polynomial of order p all stay within the unit circle[7], with the invertibility constraints imposing the same for the moving average polynomial of order q with q complex roots.[7] Notice that, for the fundamental theorem of algebra, the polynomials are bounded to have respectively p and q complex roots, counting multiplicity.

Seasonal ARIMA models follow a similar formulation, this time with a whole other set of parameters and hyperparameters dedicated to the seasonal part of the model. The conventional codification of these parameters follows the one from the non-seasonal parameters, just in upper case. A seasonal ARIMA model is thus specified by two sets of hyperparameters, (p, d, q) and $(P, D, Q)_m$, where the second set of parameters is the seasonal one and m is the seasonal period:

$$\begin{aligned}&(1-\Phi_1 B^m-\Phi_2 B^{2m}-...-\Phi_P B^{Pm})(1-\phi_1 B-\phi_2 B^2-...-\phi_p B^p)(1-B^m)^D(1-B)^d y_t = \\ &c+\varepsilon_t(1+\theta_1 B+\theta_2 B^2+...+\theta_q B^q)(1+\Theta_1 B^m+\Theta_2 B^{2m}+...+\Theta_Q B^{Qm})\end{aligned} \quad (3.8)$$

Equation 3.8 shows that many interaction effects arise between the non-seasonal and seasonal terms, adding a new layer of complexity to the model, which at this point considers both the sets of terms in an organic way. Again, as for ETS in Section 3.1, we use the maximized likelihood in combination with information criteria in order to choose the best model that fits our data. As briefly explained at the beginning of this subsection, point forecasts are easily obtained by reversing the equations and by replacing future observations with their point forecasts. Notice that since errors/residuals are modelled as normal random variables with mean 0, their mean/point forecast is always zero.

---

[5] p is the order of the autoregression, d is the degree of differencing and q is the order of the moving average. The *(p,d,q)* notation is a standard way to codify ARIMA models.

[6] Given a complex number $z = a + bi$, its inverse is $z^{-1} = \frac{1}{a+bi} = \frac{a}{a^2+b^2} - \frac{b}{a^2+b^2}i$

[7] A complex number can be thought as a point with two coordinates, given by its real and imaginary part.

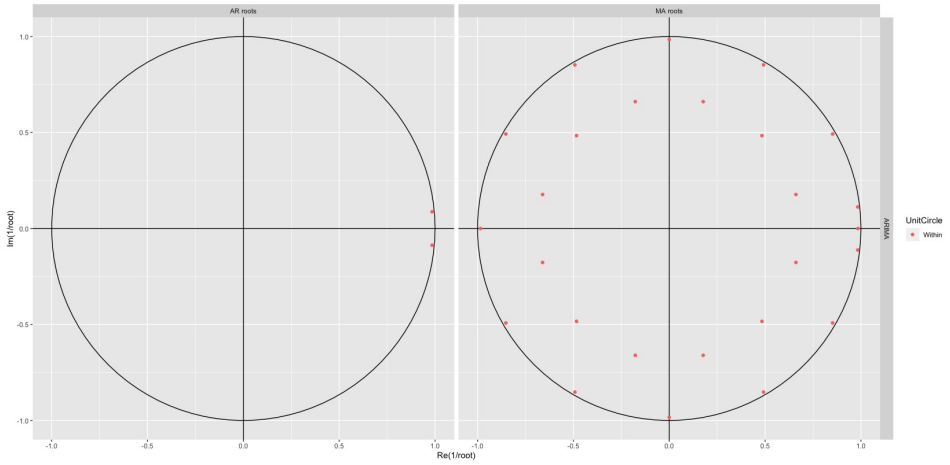

Figure 3.1: For explanatory purposes, the inverse characteristic roots of the seasonal ARIMA model from a later point in this chapter, Section 3.3. Every root is inside the unit circle, highlighting that the constraints on the ARIMA parameters are respected. Notice the high number of moving average roots: this is due to the hyperparameters of the model, which includes a seasonal moving average of second order. The seasonal moving average polynomial in the backshift operator has a high degree, and for the fundamental theorem of algebra the maximum number of distinct complex roots is equal to the degree itself.

3.3 ARIMA and ETS fitted to the US box office time series

Now, it is time to try these two models on our data, and more precisely on our time series, built in the way described in Section 2.1. To do this, we are using the *fable* package for R, a set of tools which allows to fit and compare time series models in an efficient and fast way. *fable* belongs to a set of packages called *tidyverts*, which also includes *feasts* and *tsibble*, two other packages that were extensively used for this thesis, respectively for descriptive statistics and data structures/data manipulation specific to the time series domain. Rob J Hyndman, one of the main contributor to the *tidyverts*, is one of the two authors of *Forecasts, Principles and Practice*, the book from which most of the theory regarding time series analysis in this thesis comes from. [3][4][8][9][7]

That said, we fit both an ETS and an ARIMA model[8] using as training data the values until December 2015, in order to leave one year of data to test the

[8]It should be noted that some ETS models are special cases of ARIMA models, but that is not the case for the ones selected for our use case. [7]

models. To perform model selection, *fable* excludes by default numerically unstable combinations for ETS (see Section 3.2.2), while to choose the best ARIMA a step-wise algorithm is used to simplify the exploration of the hyperparameter space; we disable it since we want the best model and we are not interested in the relatively little speed-up gained[9].

For ARIMA it is also important to have stationarity, and unit root KPSS tests and seasonal strength measures [10]are used to choose the right degree for normal and seasonal differences, respectively. Recalling the definition of stationarity in Subsection 2.2.3, differencing is sometimes not enough to obtain it, since we need to act at distribution level and differencing just stabilizes the conditional means. Variance also needs to be stabilized if needed, but in our case the series does not show evident signs of heteroscedasticity.

As per the model selection, the best ETS model is a ETS(M,N,M), a model similar to what we have formalized in Equation 3.3, just without the trend term and thus without the state equation that describes its behaviour. This makes sense, since when we factor out the inflation the time series does not seem trended at all.

For what concerns ARIMA, the best model is a ARIMA(2,0,2)$(0,1,2)_{12}$ without the constant, again confirming the fact that in this time series seasonality is strongly present but trend is totally lacking. In fact, non-seasonal/first differencing is not performed here. It is a good idea at this point to compare the performance of ARIMA and ETS with a much simpler method, the Seasonal Naïve one, which acts as a baseline, basically. Predictions for the SNAIVE model are trivially defined as the latest useful observation from the same season[11].

To quantify the performance of the models on test data, we are using root mean squared error (RMSE) and mean absolute percentage error (MAPE), which we can use since in this case our response variable uses a ratio scale[12]).

We can see that ARIMA performs definitely better for the 2016 horizon, with ETS far nearer to SNAIVE than to ARIMA, in terms of amount of variance not explained and absolute percentage error. We can also notice that from 3.2, where ARIMA follows the peaks of the series better than ETS.

The one above is a particularly partial view of the actual performance, however. A far more robust technique is time series cross-validation, which as its more famous non-sequential equivalent tries to test a model performance by continuously

[9]Another option meant to speed up the search is also turned off.

[10]Seasonal strength is just the ratio between the variance of the remainder component and the variance of the trended series, as obtained by a STL decomposition (for STL, see Subsection 2.2.1

[11]For monthly data, this means that we look at the last available observation from the same month.

[12]Revenue cannot reach a negative value and has a non-arbitrary zero.

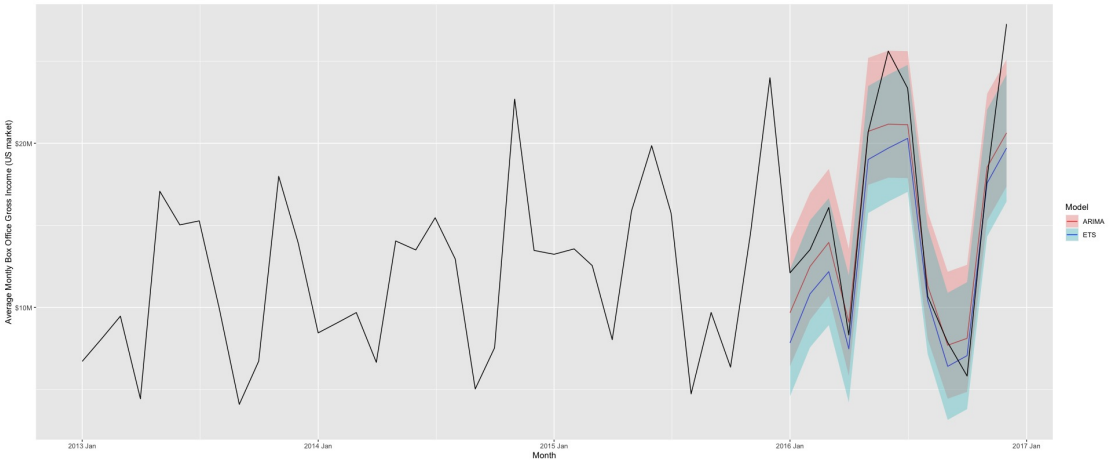

Figure 3.2: A plot of the real box office time series (black line) and of the forecasts from ARIMA (red line) and ETS (blue line). Given gaussianity of the errors, 95% prediction intervals are plotted as shadows (*ggplot* ribbon geom) around the point forecasts. Real box office data starts from 2013 in order to put a greater focus on the forecasts.

Model	RMSE	MAPE
ARIMA	2693297.9	12.7%
ETS	3529239.8	17.1%
SNAIVE	3966293.4	19.3%

Table 3.1: Test data accuracy metrics.

updating the test data. The models are fitted on a progressively higher amount of the time series, while keeping constant the forecast horizon.[8]

As you can see, with time series cross-validation over a 1 year forecast horizon the difference in accuracy between ETS and ARIMA becomes far less clear and it basically disappears, with both the models still performing significantly better than a naïve method. The relatively good performance of a seasonal naïve model is an indicator of the strong seasonality of the data, which overall seems to be incredibly difficult to fit when looking at any information other than periodicity[13].

Considering these metrics (equal performance on cross-validation, with test data better fitted by ARIMA), it makes sense to choose ARIMA. At this point,

[13]The additional explained variance we obtain by using relatively complex models like ARIMA and ETS is low.

Model	RMSE	MAPE
ARIMA	4617673.9	24.9%
ETS	4688216.0	24.8%
SNAIVE	5829149.8	28.7%

Table 3.2: Time series cross validation accuracy metrics.

some residual diagnostic is required in order to check that the test residuals are really gaussian white noise (i.e. the population autocovariances/autocorrelations are zero). In order words, is the ARIMA model able to extract all or most of the relevant information from the signal?

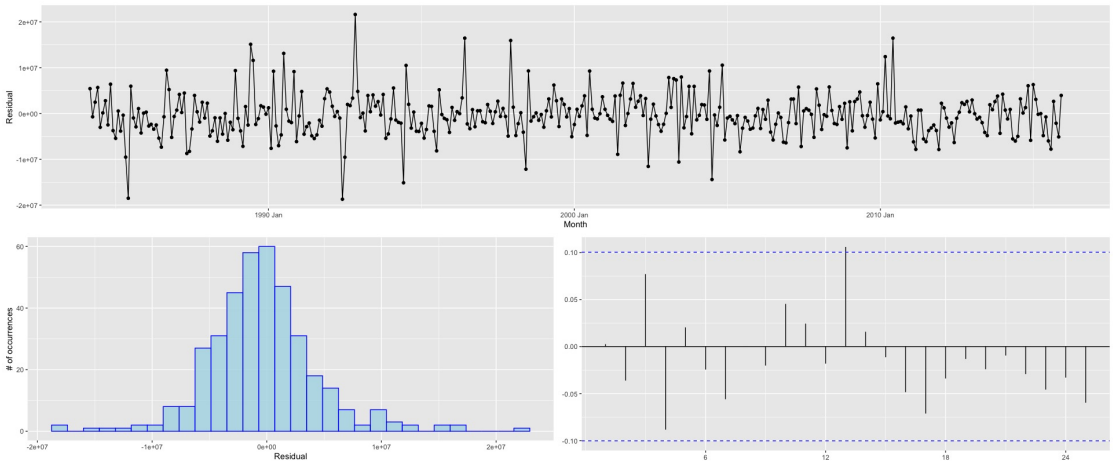

Figure 3.3: A complete visual overview of the residuals from ARIMA on test data. A time plot, a histogram and an autocorrelogram are presented.

Box-Pierce test	Box-Pierce p-value	Ljung-Box test	Ljung-Box p-value
19.1	0.388	19.7	0.349

Table 3.3: Test statistics and p-values for Box-Pierce and Ljung-Box tests on the residuals from the ARIMA model. Notice that to calculate the p-values we need to consider the degrees of freedom of the model, 6 in this case.

It is evident from Figure 3.3 that the ARIMA residuals are approximately normally distributed (as we would expect) with just one significant autocorrelation when considering them sequentially. With Table 3.3 we can also see the results from

applying Box-Pierce and Ljung-Box tests on the residuals, with very high p-values that show a complete lack of evidence towards non-null population autocorrelations. Therefore, the residuals have all the properties of a gaussian white noise series, so the model should have extracted most of the information from the data.

Chapter 4

Tree-based methods

The last part of this thesis deals with tree based methods, which are used to model exogenous regressors[1] by using as response the difference between the gross box office revenue of the movie and the fitted value from ARIMA for the month of the release date. The set of regressors here comprises important information such as the genre, the original language, the production company, the production country, the duration and the director of the movie, with frequency based text mining to account for title and synopsis. For directors and production company, the choice of what to consider in the model is done on a most-frequent basis, while for what concerns original languages the selection is made according to a most-spoken-in-the-world. Production countries are chosen by looking at the list of countries sorted by the number of productions hosted, with data about that coming from *thenumbers.com*, an important database for movie data we already talked about in Section 1.1. For more information regarding the quite complex pre-processing for this stage please look at the R code in Appendix A.2.5.

Even though I will not go in detail with tree based methods since it is not the main focus of this thesis, I will explain here the two algorithms which are used in this case, bagging (briefly, since it is really a standard and simple algorithm[2])and gradient boosting (more in depth).

4.1 Bagging

As anticipated, I will be quite brief since bagging is quite intuitive and easy to explain. The algorithm was formally introduced by Leo Breiman in 1996 with an

[1]Exogenous with respect to ARIMA.

[2]That said, it is still one of the most effective tree based algorithms out there and it has a quite strong statistical reasoning, along with its more robust variation, random forest.

incredibly famous paper, *Bagging Predictors* [11], which is a pillar of the modern machine learning field, along with the following one about the random forest algorithm, always by Breiman[12].

Bagging mainly addresses the fact that decision trees, although easy to explain and quite useful for interpretability, do not generalize well and are quite prone to overfitting, leading to high variance for the distribution of the prediction considered as a random variable. Bagging uses the fundamental properties of variance to solve or at least greatly reduce this problem, and it does that by building an ensemble of (CART) trees, each fitted to a set of observations sampled with replacement of the same size of the original dataset.

To be more formal, in a regression context, under the assumption of uncorrelated predictions[3] and equally distributed predictions, and with Y_i as the random variable representing one prediction: $Var\left(\frac{\sum_i^N Y_i}{N}\right) = \frac{Var(Y_i)}{n}$, where N is the number of trees in the ensemble model.

Given that the assumption of independent predictions is far from realistic, random forest takes the algorithm a step forward by selecting only a subset of the features for each tree in the ensemble, thus *breaking links* in the forest, with the result of reducing the covariance terms impacting the variance of the ensemble prediction.

Unfortunately, since the dataset that we are using here is incredibly sparse, it does not make sense to use random forest.

4.2 Gradient Boosting

Gradient boosting is a far more elegant and complex algorithm/concept with respect to Bagging or Random Forest. The algorithm we are using to fit it through R stems from two important papers written by Jerome H. Friedman in 1999.[13][14]

Very broadly speaking, gradient boosting can be seen as an extension of the concept of gradient descent to a tree-based method. Unfortunately, trees do not have parameters in the strict sense, and we cannot simply calculate a gradient of a differentiable cost function and move in the parameter space accordingly. It works quite differently and in a clever way, but the underlying idea of exploiting information from gradients of a loss is mostly the same.

The following description of the algorithm, with some tweaks, is taken from *The Elements of Statistical Learning*, a reference book for the subject[15]:

[3]They are not, since the underlying data is the same, even though the noise stemming from sampling (only 63% of the data is considered for each tree, asymptotically) and replacement do help. Random forest allows to reduce the covariance between the predictions.

Algorithm 1 Gradient Boosting algorithm

function GradientBoosting($D, \nu,\ M,\ L(y, f(x))$)

▷ D is the dataset of observations

▷ ν is the learning rate, a regularization constant for the ensemble procedure

▷ M is the number of weak learners in the ensemble

▷ $L(y, f(x))$ is the differentiable loss function which needs to be minimized for each observation. It is a function of y, the real observation value, and $f_m(x)$, the output of the ensemble model at stage m, after m weak learners are fitted

▷ Initialization

$f_0(x) = \arg\min_{\gamma} \sum_{i=1}^{N} f(x_i, \gamma)$ ▷ Initial output is set for every observation

$m \leftarrow 0$ ▷ Weak learner index is set to 0

▷ Execution

while $m \leq M$ **do**

$m \leftarrow m + 1$ ▷ Update weak learner index

$r_{i,m} = -\left[\frac{\partial f(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}$ ▷ Compute pseudo-residuals

▷ Fit a regression tree over the pseudo-residuals $r_{i,m}$ with resulting terminal regions $R_{j,m}$ with $j = 1, 2, ..., J_m$

$\gamma_{j,m} = \arg\min_{\gamma} \sum_{x_i \in R_{j,m}} L(y_i, f_{m-1}(x_i) + \gamma)$ ▷ Compute updates

$f_m(x) = f_{m-1} + \sum_{j=1}^{J_m} \nu\, \gamma_{j,m} [\![x \in R_{j,m}]\!]$ ▷ Update output of the ensemble[4]

end while

return $\hat{f}(x) = f_M(x)$ ▷ Prediction from the ensemble is returned

end function

[4]The square brackets are Iverson brackets. If the condition contained between them is satisfied, their value is 1, otherwise it is 0.

Algorithm 1 is a complete generalization of gradient boosting[5]: depending on the loss function we are feeding it with, the optimization problem changes and thus also the overall task of the method. For example, if the differentiable loss function is the binary cross entropy[6], the task of the gradient boosting algorithm is binary classification. In our case, the task is a regressive one.

The most common loss function for a regressive problem is just the squared residual expression[7], since it comes from the maximization of the log-likelihood given gaussianity. With this loss function, boosting is usually indeed simplified to the fitting of a weak learner on the residuals of the preceding sequence of weak learners, returning as output for each weak learner the average of the residuals in the leaf. This is also our case, but how does it come to that? For that, we need to follow the step of Algorithm 1.

Using the squared residual as the loss function for just a single observation, the pseudo-residuals[8] are trivially found:

$$-\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} = -\frac{\partial \frac{1}{2}(y_i - f(x_i))^2}{\partial f(x_i)} = -(y_i - f(x_i)) = f(x_i) - y_i \tag{4.1}$$

The γ output of the leaf that minimizes the loss is easy to compute. We find the only stationary point of the sum of the losses, which is the global minimum since the sum of the squared residuals is a convex function of γ. As we have said, the stationary point is just the mean of the residuals in the leaf, since:

$$\frac{\partial \sum_i \frac{1}{2}(y_i - f(x_i) - \gamma)^2}{\partial \gamma} = -\sum_i y_i - f(x_i) - \gamma = 0 \iff \gamma = mean(f(x_i) - y_i) \tag{4.2}$$

The analogy to gradient descent is more appropriate and more straightforward for this specific situation. Since each pseudo-residual is just a residual and each leaf output is just the mean of the residuals in that leaf, we are each time updating the outputs of the ensemble according to the gradient of the loss with respect to the outputs themselves. Notice that the loss we are talking about here is the sum of the squared residuals for each observation in the dataset.

[5]For multinomial classification, things are slightly more complicated since multiple weak learners are needed at each stage and a softmax is required.

[6]It needs to be manipulated a little bit in order for it to be a function of the log-odds and not of the probabilities.

[7]$L(x, \hat{x}) = \frac{1}{2}(x - \hat{x})^2$

[8]In this case the pseudo-residuals are the actual residuals, but this is just a specific situation. With an absolute error loss the pseudo-residual would just be the sign of the actual residual, for example.

Each entry of the gradient is the partial derivative of the loss with respect to a function[9], where each function is the mapping from an observation to its prediction. In simpler wording, the predictions themselves take the place of the parameters of a normal gradient descent.

$$\mathbf{f}_t(\mathbf{x}) \leftarrow \mathbf{f}_{t-1}(\mathbf{x}) - \nu \nabla_{\mathbf{f}} L(\mathbf{y}, \mathbf{f}_{t-1}(\mathbf{x})) \tag{4.3}$$

The update expression[10] written above is exactly the same of the gradient descent one. It is not precise since the gradient update is approximated with the means of the partial derivatives in the leaves, but the idea holds anyway.

To actually train a gradient boosting model we also have to choose its hyperparameters. There are mainly three: the learning rate, the number of weak learners and the maximum depth of each weak learner. Without some hyperparameter tuning gradient boosting easily overfits the data, not generalizing well.

A way to solve for this is the brute force approach, running a massive, parallelized cross validation on the method while searching the hyperparameter grid/space for the solution which minimizes the chosen loss metric. This is also our approach, as explained in the next section.

4.3 Tree-based methods using exogenous regressors and last stage of the hybrid model

Now, as explained at the beginning of this chapter and in Section 1.3, we fit the methods to the difference between ARIMA fitted values and the actual gross box office revenue. This allows us to link the time series modelling to the gross box office revenue of the single movie, thus completing our hybrid model.

Details of the implementation in R are given in Appendix A.2.6, but some information on the hyperparameter tuning and the fitting must be given here.

For what concerns Gradient Boosting, the hyperparameter tuning with 10-fold cross validation concerned the learning rate, the number of weak learners fitted and the interaction depth of each weak learner. To be clear, the interaction depth is just the maximum interaction level between features in a tree, considering the split at the root as a main effect, and each subsequent split as an interaction effect. In other

[9]The partial derivatives are the (pseudo-)residuals.

[10]In the update expression f is a vector valued function taking a vector valued input of dimensionality n, where n is the cardinality of the dataset. This can be noticed through the bold font. It is the function representing the whole set of predictions. $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$. As already explained, the loss here is real-valued and takes as input a vector of squared residuals of dimensionality n (considering the responses as a constant vector), since it is the sum of the squared residuals. $L : \mathbb{R}^n \to \mathbb{R}_+$.

words, the interaction depth is the maximum depth each weak learner can reach. In addition, sampling without replacement (70%) was performed to train each weak learner, as suggested in the Stochastic Gradient Boosting implementation by Friedman (2001)[14].

The results from the cross-validation[11] indicate that the most appropriate combination of hyperparameters is 800 for the number of trees, 8 for the interaction depth and 0.2 for the shrinkage rate. The importance of each predictor in the boosting process is also calculated with respect to the final model[12], as the reduction in performance due to random permutation on the predictor.

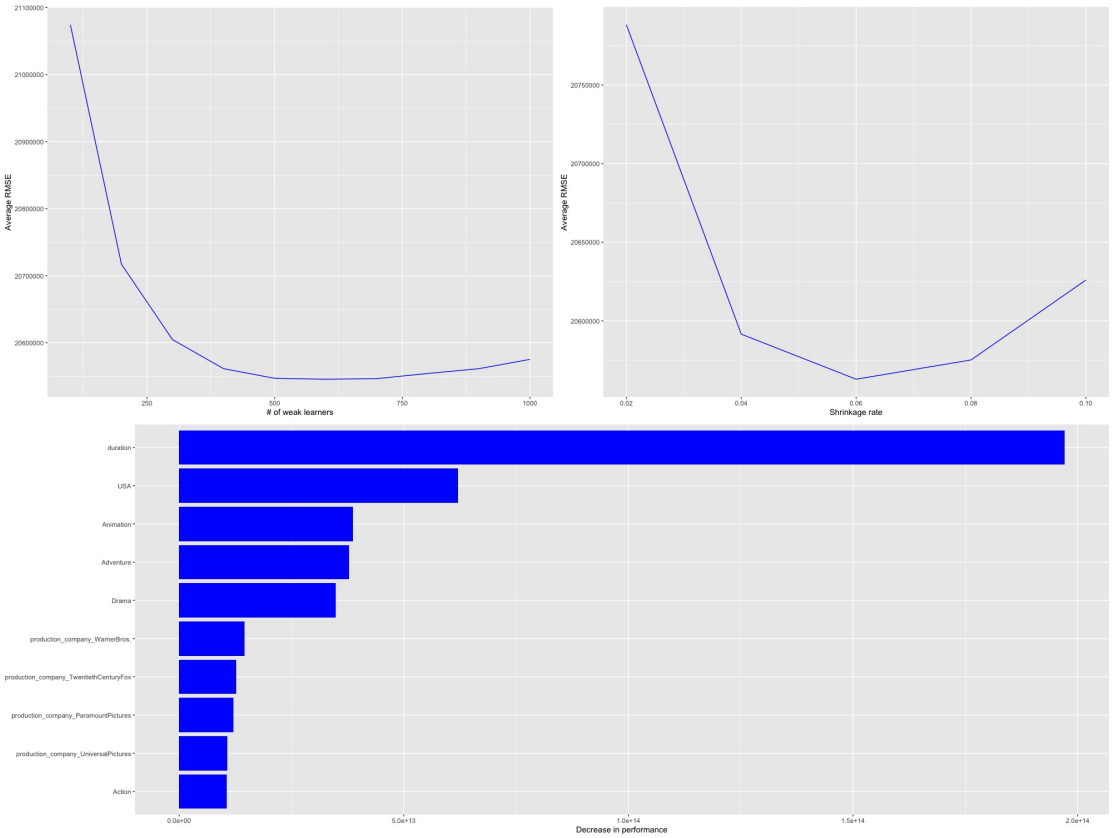

Figure 4.1: In the two upper tiles, the aggregated cross-validated RMSE as the number of weak learners (left) and the shrinkage parameter (right) vary. Below, the variable importance plot for gradient boosting with respect to the final model. Notice the (obvious) importance of the duration of each movie as a predictor.

[11]The grid for the hyperparameters considered 100 to 1000 by 100 for the number of trees, 1 to 8 for the interaction depth and 0.01 to 0.1 by 0.01 for the shrinkage rate.

[12]The one with the cross-validated hyperparameters and fit to the whole dataset.

Bagging is also (repeatedly) cross-validated, but this time just to have robust measures of its performance, and not to tune any hyperparameters, since bagging is not prone to overfitting at all (differently to boosting, which is quite delicate). 10-fold cross-validation is repeated 3 times and the number of trees in the ensemble is set to a pretty high number (1000). Below a variable importance plot is shown, with the importance computed as the total decrease of residual sum of squares induced by splits on the variable, averaged over all the trees.

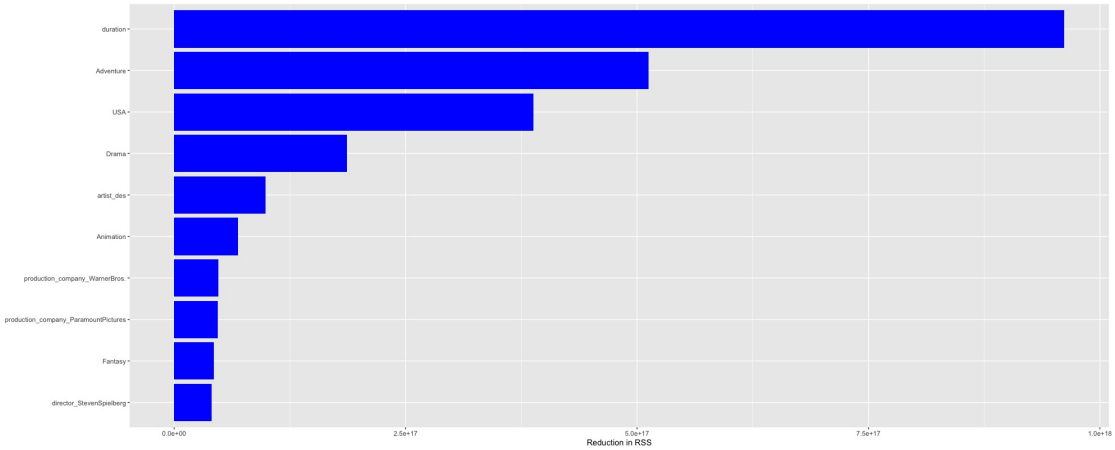

Figure 4.2: Variable importance plot for bagging: total decrease of RSS induced by splits on the variable, averaged over all the trees. Again, the duration is by far the most important variable.

Both the models explain about 40% of the variance of the response variable, with gradient boosting performing better, closer to 40%, and bagging hovering around 35%.

More precisely, table 4.1 shows the cross-validated R^2 of both the methods, which is exactly a measure of explained variance.

Gradient Boosting (R^2 CV)	Bagging (R^2 CV)
0.392	0.352

Table 4.1: Cross-validated R^2 for both tuned gradient boosting and bagging.

Given the cross-validated R^2, it makes sense to use boosting for the final stage of the overall hybrid model. At this point, we can sum the predictions from gradient boosting with the forecasts from ARIMA, thus obtaining an overall predictions which targets the actual US boxoffice revenue of the single movie.

The R^2 for the predictions of the hybrid model is easy to compute, being simply the ratio between the variance of the errors (mean squared residual) and

the variance of the response, thus dividing the unexplained variance by the total variance. Unfortunately, due to the quantity and the quality of the data (we cannot buy the access to refined databases) and probably also due to the simplicity of parts of our analysis, the explained variance on test data (2016 movies) is just 10% of the total. This low R^2 can be probably imputed to time series forecasts which are accurate only up to a certain level and to the wild fluctuations present in the time series itself, which are in turn probably caused by noise induced by the low amount of data.

Conclusions

This thesis presented a complete proof-of-concept for a hybrid model combining observation specific data with a time aggregated perspective, introducing in the meanwhile most of the main concepts of time series analysis, which were not taken for granted. In fact, a thorough review of the statistics and the theory behind time series was given, avoiding a sterile adoption of models in a black box fashion, which is sadly so common in the data science field. Tree-based methods and in particular gradient boosting were also rigorously treated, going more in depth than what is usual when exploiting these techniques, whose mathematical and statistical soundness is often overlooked.

In conclusion, although somewhat dehumanizing with respect to the artistic background of the entertainment field, it is clear and obvious that there is information there to be modelled, and that there is value to be extracted from it by gaining insights into the behaviour and the tastes of viewers. The hope is that this thesis managed to show that, highlighting patterns and thus the presence of a strong signal in this kind of data.

With more advanced models, more time and resources, and especially with a complete access to refined and comprehensive datasets that are kept behind very expensive paywalls, the results would probably be far more interesting, and so we are leaving to future and funded research the task to further explore the idea proposed in these pages.

Appendix A

Python and R Code

A.1 Python code for scraping

As explained in the thesis, to scrape the initial data to work with, extensive pieces of code dedicated to scraping were necessary. Most of the work was made to communicate with the TMDB API(see Section 1.4), make the appropriate GET requests and parse the results.

The following code is necessary to map each IMDb ID from the original dataset to a TMDB ID from the TMDB dataset. We are able to send a specific GET request containing the IMDb ID in order to get a JSON containing the data we need about the movie from the TMDB dataset. A lot of the code is also designed to allow the scraping to stop and start again when needed, which proves useful especially with limited time and resources.

Notice that to interact with the TMDB API a key is needed, and this can be obtained by registering on the TMDB website. Since it is a private key, it is not present in the code and it is called from an *.env* file.

The obtained TMDB IDs, together with the original IMDb IDs, are saved to a *.csv* file during execution.

```
import requests
import csv
import time
import os
import dotenv


os.chdir(os.path.split(os.path.abspath(__file__))[0])  # This allows
    to set the current directory to script folder
dotenv.load_dotenv(os.path.join(os.path.split(os.path.abspath(os.path
    .curdir))[0], '.env'))
```

```
api_key = os.getenv('TMDB_KEY')  # This is to avoid putting the key as plain text

EditMode = 'a'
resume_control = False
complete = True
resume = False

with open('IMDbmovies.csv', 'r') as first_line_check:
    next(first_line_check)  # Skip header
    id_reference = next(first_line_check).split(',')[0]  # This is useful in case we have not an existing CSV file

# These following lines make the program flexible, it can work with a new file or by appending the IDs to existing one
if os.path.exists('TMDB_ID.csv'):
    file_reader = open('TMDB_ID.csv', 'r')
    next(file_reader)   # Skip header
    for element in file_reader:
        id_reference = element.split(',')[0]
    file_reader.close()
    print('Existing CSV file found, the calls to the TMDB API will start again \n '
          'from the IMDbID of the last row in the file')
else:
    print('No existing CSV file found, a new one will be built')
    EditMode = 'w'


with open('IMDbmovies.csv', 'r') as original:
    next(original)  # Skip header of the CSV
    with open('TMDB_ID.csv', EditMode) as id_pairs:
        writer = csv.writer(id_pairs)
        if EditMode == 'w':
            writer.writerow(('IMDb_ID', 'TMDB_ID', 'English Title'))
    # Header is written if new csv file
        start_time = time.time()
        for film in original:
            elapsed = time.time()-start_time
            if film.split(',')[0] == id_reference:  # This condition allows scraping to start again where it was paused
                test = film.split(',')[0]
                resume = True
            if resume_control or EditMode == 'w':
                if elapsed <= 1800:  # Elapsed time reaches given time and then stops the execution
                    imdb_id = film.split(',')[0]
                    response = requests.get(f'https://api.themoviedb.org/3/find/{imdb_id}?api_key={api_key}'
```

```
                                                f'&external_source=
    imdb_id')
                    if response.status_code == 200:
                        dictionary_result = response.json()['
    movie_results']
                        if dictionary_result:
                            row = (imdb_id, dictionary_result[0]['id'
    ], dictionary_result[0]['title'])
                            writer.writerow(row)
                        print(f'Current IMDb ID used for the call to
    the API ---> {imdb_id}')
                else:
                    print('Maximum elapsed time reached, exiting
    program execution...')
                    complete = False  # This boolean allows the
    program to understand if scraping is complete
                    break
            if resume:
                resume_control = True
if complete:
    print('The program has parsed every line in the CSV file with the
    IMDB IDs, exiting program execution...')
```

The next code uses the IDs gathered in the previous *.csv* file to send other GET requests to the API, this time asking for the data on the release dates. These specific GET requests return a particularly nested JSON array containing the whole set of available release dates for the specific movie queried. The API documentation needs to be followed to understand how to navigate the JSON in order to only get theatrical release dates for the North American market. Different dates for the same movie are allowed, with the *note* field being later used (in R execution) to understand which one is correct, when possible. The release dates are stored in a *.csv* file, each of them paired with a TMDB ID to identify the related movie.

```
import requests
import csv
import time
import os
import dotenv

os.chdir(os.path.split(os.path.abspath(__file__))[0]) # Set cd to
    folder of the script
dotenv.load_dotenv(os.path.join(os.path.split(os.path.abspath(os.path
    .curdir))[0], '.env'))
api_key = os.getenv('TMDB_KEY')  # This is to avoid putting the key
    as plain text
```

```
10
11 EditMode = 'a'
12 complete = True
13 resume_control = False
14 resume = False
15
16 with open('TMDB_ID.csv', 'r') as first_line_check:
17     next(first_line_check)  # Skip header
18     id_reference = next(first_line_check).split(',')[1][:-1]  # This
    is useful in case we have not an existing CSV file
19
20 # The following lines make the program flexible, it can work with a
    new file or by appending the IDs to existing one
21 if os.path.exists('TMDB_Release.csv'):
22     file_reader = open('TMDB_Release.csv', 'r', newline='\n')
23     next(file_reader)  # Skip header
24     for element in file_reader:
25         id_reference = element.split(',')[0]
26     file_reader.close()
27     print('Existing CSV file found, the calls to the TMDB API will
    start again \n '
28           'from the last observation in the CSV file')
29 else:
30     print('No existing CSV file found, a new one will be built')
31     EditMode = 'w'
32
33 time_limit = input('How long do you want this program to run at most?
     Please insert time in minutes')
34
35 with open('TMDB_ID.csv', 'r') as original:
36     next(original)  # Skip header of the CSV
37     with open('TMDB_Release.csv', EditMode) as release_csv:
38         writer = csv.writer(release_csv)
39         if EditMode == 'w':
40             # Header is written if new csv file
41             writer.writerow(('TMDB_ID', 'Release_Date_US_Theatrical',
     'Release date note'))
42         start_time = time.time()
43         for film in original:
44             elapsed = time.time()-start_time
45             # This condition (below) allows scraping to start again
    where it was paused
46             if film.split(',')[1][:-1] == id_reference:
47                 resume = True
48             if resume_control or EditMode == 'w':
49                 if elapsed <= int(time_limit)*60:  # Elapsed time
    reaches x minutes and then stops
50                     tmdb_id = film.split(',')[1][:-1]
```

```
51                      response = requests.get(f'https://api.themoviedb.
    org/3/movie/{tmdb_id}/release_dates?'
52                                              f'api_key={api_key}')
53                      if response.status_code == 200:
54                          response = response.json()['results']  # From
     here, research in the nested dictionary
55                          for array_country in response:
56                              if array_country['iso_3166_1'] == 'US':
    # According to the schema of the JSON response
57                                  for array_release in array_country['
    release_dates']:
58                                      if array_release['type'] == 3:
59                                          release_date_us =
    array_release['release_date'][:10]
60                                          try:
61                                              note = array_release['
    note']
62                                          except KeyError:
63                                              note = ''
64                                          row = (tmdb_id,
    release_date_us, note)
65                                          writer.writerow(row)  # I
    removed a break here in order to catch
66                                          # the eventuality of multiple
     theatrical dates (movies that were released again)
67                                          # This possible inconsistency
     will be manually cleaned in later steps (with R)
68                                          # by looking at the notes of
    the release dates
69                                  break
70                          print(f'Current TMDB ID used for the call to
    the API ---> {tmdb_id}')
71                      else:
72                          print('Maximum elapsed time reached, exiting
    program execution...')
73                      complete = False  # This boolean variable allows
    the program to understand if scraping is complete
74                      break
75              if resume:
76                  resume_control = True
77  if complete:
78      print('The program has reached the end of the CSV file with the
    TMDB IDs, terminating program execution...')
```

A.2 R code for time series manipulation, analysis and modelling

A.2.1 Initial pre-processing

The following R code deals with most of the initial pre-processing of the dataset, a part of the pipeline that comes even before the actual manipulation of time series data. Here scraped data which was saved in *.csv* files is loaded into memory, release dates are cleaned through their *note* field[1] with regular expressions, boxoffice data is adjusted for inflation, inner joins are performed and data is filtered with respect to missing values in relevant fields.

```
library(tidyverse)
library(fpp3)
library(readxl)
library(zoo)
library(future)
library(parallel)
library(tictoc)
library(purrr)
library(tidytext)
library(randomForest)
library(doParallel)
library(caret)
library(ggthemes)
library(tm)
library(moments)
library(facetscales)

set.seed(1)


path_separator = .Platform$file.sep # We need the OS separator to
    generalize the code execution

tmdb_id = readr:: read_csv(paste('.','TMDB_ID.csv', sep =
    path_separator))

# We read the CSV dataset I have obtained by performing some scraping
     through a Python script
# This is a CSV with all the release dates for the original dataset
    that were present in the TMDB
# database. All the data was retrieved through the TMDB API
```

[1]Additional information left by users. TMDB is crowdsourced.

```
28
29 release_dates = readr:: read_csv(paste('.','TMDB_Release.csv', sep =
       path_separator))
30
31 # Set of regex to search for in order to remove some release dates
       according to their 'note' field
32 regex_set = paste(
33   '(?i)fest|(?i)re-|(?i)version|(?i)limited|(?i)cut|',
34   '(?i)edition|(?i)played|(?i)reissue|(?i)remastered|',
35   '(?i)restoration|(?i)guess|(?i)blu-ray|(?i)dvd|(?i)premiere|',
36   '(?i)rerelease|(?i)internet|(?i)event|(?i)edition|(?i)known',
37   sep = ''
38 )
39
40 # First, I select the set of release dates which refer to multiple
       TMDB IDs
41 # I try to clean them through regular expressions, the rest is
       discarded
42 release_dates = bind_rows(
43   release_dates %>% group_by(TMDB_ID) %>% mutate(count = n()) %>%
44     filter(count > 1) %>% filter(TMDB_ID != 86978) %>% # I am
       excluding an observation manually (it does not make sense)
45     filter(
46       is.na(`Release date note`) |
47         str_detect(
48           `Release date note`,
49           negate = T,
50           pattern = regex(regex_set)
51         )
52     ) %>% mutate(count = n()) %>% filter(count == 1),
53   release_dates %>% group_by(TMDB_ID) %>% mutate(count = n()) %>%
       filter(count == 1)
54 ) %>%
55   filter(is.na(`Release date note`) |
56            str_detect(
57              `Release date note`,
58              negate = T,
59              pattern = regex(regex_set)
60            )) %>%
61   select(-`Release date note`, -`count`)
62
63 # I am reading the original dataset ('IMDbMovies.csv')
64 imdb_dataset = readr:: read_csv(paste('.','IMDbMovies.csv', sep =
       path_separator)) %>%
65   rename(IMDb_ID = 'imdb_title_id')
66 # Inflation dataset is read
67 # Source for inflation/CPI dataset ---> U.S. Bureau of Labor
       Statistics
```

```
# (https://data.bls.gov/timeseries/CUUR0000SA0?years_option=all_years)
# base period 1982-1984
cpi = read_excel(paste('.','CPI.xlsx', sep = path_separator))
cpi = cpi %>% select(c(-HALF1, -HALF2)) %>% pivot_longer(2:13, names_to = 'Month', values_to = 'CPI') %>%
  mutate(Month = yearmonth(paste(Year, Month, sep = ','))) %>% select(-Year)


# Now, inner join between release dates dataset and original dataset
dataset = inner_join(tmdb_id, release_dates, by = 'TMDB_ID') %>% inner_join(imdb_dataset, by = 'IMDb_ID') %>%
  select(-date_published, -year) %>% filter(usa_gross_income != '' & str_detect(usa_gross_income, '$')) %>%
  mutate(Release_Date_US_Theatrical = yearmonth(Release_Date_US_Theatrical)) %>%
  mutate(usa_gross_income = parse_number(usa_gross_income)) %>% rename(Month = 'Release_Date_US_Theatrical')

# We adjust for united states inflation the USA gross revenue of the movies, removing trend
# We keep the non adjusted values for visualization
dataset = inner_join(dataset, cpi, by = "Month") %>%
  mutate(notadj_boxoffice = usa_gross_income, usa_gross_income = (usa_gross_income / CPI) * 100) %>% select(-CPI)

# Let's check for missing values in those fields (+ language and + country + description + title)
# These are relevant features features and need to be present
dataset %>% filter(is.na(country) | is.na(language) | is.na(genre) | is.na(description) | is.na(title)) %>% summarise(n())
dataset = dataset %>% filter(!(is.na(country) | is.na(language) | is.na(genre) | is.na(description) | is.na(title)))
```

A.2.2 Manipulation, exploration and visualization of the US box office time series

The following code takes care of building and manipulating the time series, using *tidyverse* and *tidyverts* at full capability. Data visualization is another of the main concerns of this piece of code and in general of a proper data exploration procedure, especially here, where many important concepts of time series analysis are introduced (see Chapter 2). The extremely powerful *ggplot2* library is the workhorse which allowed the rendering of literally every plot in this thesis.

More in detail, the time series of the monthly averages of US box office gross

revenue is built, missing values[2] are tracked and imputed, STL decompostion is performed and the Box-Pierce and Ljung-Box are introduced and applied on the series. As part of the focus on data visualization, Figures 2.1, 2.5, 2.3 and 2.4 are all defined below. The insightful Table 2.1 is also defined here.

```
### Building, exploring and visualizing the monthly boxoffice time series ------

# We start from the 1980s (avoiding a difficult to understand spike in 1982) and we exclude very recent years,
# we have about 10'000 movies remaining through which we estimate the monthly means,
# after all this filtering.

dataset = dataset %>% filter(format(Month, '%Y')>1982 & format(Month, '%Y')<2017)

# Compare non adjusted with adjusted data, two plots

dataset %>% group_by(Month) %>% filter(n() >= 10) %>%
  summarise(AverageMonthBoxOffice = mean(usa_gross_income)) %>%
  as_tsibble(index = Month) %>%
  autoplot() + labs(y = 'Average Domestic Box Office Gross Income') +
  scale_y_continuous(labels = scales::dollar_format()) +
  geom_smooth(method = 'lm', se = F)

dataset %>% group_by(Month) %>% filter(n() >= 10) %>%
  summarise(AverageMonthBoxOffice = mean(notadj_boxoffice)) %>%
  as_tsibble(index = Month) %>%
  autoplot() + labs(y = 'Average Domestic Box Office Gross Income') +
  scale_y_continuous(labels = scales::dollar_format()) +
  geom_smooth(method = 'lm', se = F)
dataset = dataset %>% select(-notadj_boxoffice)

# Now we have to fill the remaining gaps in the resulting time series,
# and we overwrite the monthly average box office feature with one with NAs in the series linearly interpolated from the existing points/averages
# There are only about 6 months/monthly averages missing in the series over a total of 400. The data imputation can definitely be acceptable, artificially data is just a tiny fraction (1.5%).
dataset %>% group_by(Month) %>% filter(n() >= 10) %>%
```

[2]If there are less than 10 observations in a month, the mean is not computed and month is a missing value in the series.

```
30    summarise(AverageMonthBoxOffice = mean(usa_gross_income)) %>%
31    as_tsibble(index = Month) %>% fill_gaps(AverageMonthBoxOffice = NA)
        %>% as_tibble() %>%
32    filter(!complete.cases(.)) %>% summarise(n())
33
34  # With the code below, the time series is finally complete
35
36  compl_series = dataset %>% group_by(Month) %>% filter(n() >= 10) %>%
37    summarise(AverageMonthBoxOffice = mean(usa_gross_income)) %>%
38    as_tsibble(index = Month) %>% fill_gaps(AverageMonthBoxOffice = NA)
        %>%
39    mutate(AverageMonthBoxOffice = na.approx(AverageMonthBoxOffice))
       %>%
40    mutate(AverageMonthBoxOffice = round(AverageMonthBoxOffice))
41
42
43  # Autocorrelation function plot
44  compl_series %>% ACF(lag_max = 60) %>% autoplot() + labs(y = '
      Autocorrelation',
45                                                    title = 'Correlogram of
      the adjusted U.S. boxoffice time series')
46
47  # Let's perform the Ljung-Box/Box-Pierce test in order to check if
      the first n autocorrelations in the ACF are significantly
      different from what one would expect from white noise
48  # The p-value is so low that the software rounds it to zero, so we
      can definitely reject the null hypothesis and say that there is
      some significant autocorrelation in this time series
49
50  df_portmant = compl_series %>% fabletools::features(
      AverageMonthBoxOffice, c(box_pierce, ljung_box), lag = 24, dof=0)
51  colnames(df_portmant) = c('Box Pierce test', 'Box Pierce p-value', '
      Ljung-Box test', 'Ljung-Box p-value')
52  write_csv(df_portmant, './portmanteau.csv')
53
54
55  # STL Decomposition
56
57  scales_multiplicative = list(Trend= scale_y_continuous(labels =
      scales::label_dollar(scale =1/10^6, suffix = 'M'),
58                                                    breaks = scales::
      extended_breaks(n = 3)),
59                'Time Series' = scale_y_continuous(labels = scales::
      label_dollar(scale =1/10^6, suffix = 'M'),
60                                                     breaks = scales::
      extended_breaks(n = 3)),
61                Seasonality = scale_y_continuous(breaks = scales::
      extended_breaks(n = 3)),
```

```
62                 Remainder = scale_y_continuous(breaks = scales::
    extended_breaks(n = 3)))
63
64 # Additive plot
65
66 compl_series %>% model(STL(AverageMonthBoxOffice~trend()+season(),
    robust = T)) %>% components() %>% autoplot() + labs(subtitle =
    NULL, title = NULL) +
67   scale_y_continuous(labels = scales::label_dollar(scale =1/10^6,
    suffix = 'M'), breaks = scales::extended_breaks(n = 3))
68 compl_series %>% model(STL(AverageMonthBoxOffice~trend()+season(),
    robust = T)) %>% components() %>%
69   as.tibble() %>% select(-.model, -season_adjust) %>%
70   rename('Trend' = trend, 'Seasonality' = season_year, 'Remainder' =
    remainder, 'Time Series' = AverageMonthBoxOffice) %>%
71   pivot_longer(cols = c(-Month), names_to = 'Component') %>%
72   mutate(Component = factor(.$Component, levels = c('Time Series', '
    Trend', 'Seasonality', 'Remainder'))) %>%
73   ggplot(aes(x = Month, y = value)) +
74   geom_line() +
75   theme(axis.title = element_blank()) +
76   scale_y_continuous(labels = scales::label_dollar(scale =1/10^6,
    suffix = 'M'), breaks = scales::extended_breaks(n = 4)) +
77   facet_grid(rows = 'Component', scales = 'free') +
78   labs(subtitle = NULL, title = NULL)
79
80 # Multiplicative plot
81 compl_series %>% model(STL(log(AverageMonthBoxOffice)~trend()+season
    (), robust = T)) %>% components() %>%
82   mutate(trend = exp(trend), season_year = exp(season_year),
    remainder = exp(remainder),
83          `log(AverageMonthBoxOffice)` = exp(`log(
    AverageMonthBoxOffice)`)) %>%
84   as.tibble() %>% select(-.model, -season_adjust) %>%
85   rename('Trend' = trend, 'Seasonality' = season_year, 'Remainder' =
    remainder, 'Time Series' = `log(AverageMonthBoxOffice)`) %>%
86   pivot_longer(cols = c(-Month), names_to = 'Component') %>%
87   mutate(Component = factor(.$Component, levels = c('Time Series', '
    Trend', 'Seasonality', 'Remainder'))) %>%
88   ggplot(aes(x = Month, y = value)) +
89   geom_line() +
90   theme(axis.title = element_blank()) +
91   facet_grid_sc(rows = 'Component', scales = list(y =
    scales_multiplicative)) +
92   labs(subtitle = NULL, title = NULL)
93
94 # Additive STL seasonality
95 season_STLadd = compl_series %>% model(STL(AverageMonthBoxOffice~
    trend()+season(), robust = T)) %>% components() %>%
```

```
96    as.tibble() %>% select(Month, season_year) %>%
97    rename(Seasonality = 'season_year') %>% mutate(Month = as.integer(
        month(Month))) %>%
98    group_by(Month) %>% summarise(`Average seasonal component (AD)` =
        mean(Seasonality)) %>%
99    mutate(`Average seasonal component (AD)` = round(scale(`Average
        seasonal component (AD)`), digits = 2), Month = month.name[Month])
100
101 # Multiplicative STL seasonality
102 season_STLmult = compl_series %>% model(STL(log(AverageMonthBoxOffice
        )~trend()+season(), robust = T)) %>% components() %>%
103   mutate(trend = exp(trend), season_year = exp(season_year),
        remainder = exp(remainder),
104           `log(AverageMonthBoxOffice)` = exp(`log(
        AverageMonthBoxOffice)`)) %>%
105   as.tibble() %>% select(Month, season_year) %>%
106   rename('Seasonality' = season_year) %>% mutate(Month = as.integer(
        month(Month))) %>%
107   group_by(Month) %>% summarise(`Average seasonal component (ML)` =
        mean(Seasonality)) %>%
108   mutate(`Average seasonal component (ML)` = round(scale(`Average
        seasonal component (ML)`), digits = 2),
109          Month = month.name[Month])
110
111 season_STLadd$`Average seasonal component (AD)` = season_STLadd$`
        Average seasonal component (AD)`[, 1]
112 season_STLmult$`Average seasonal component (ML)` = season_STLmult$`
        Average seasonal component (ML)`[, 1]
113
114 seasonal_STLtable = inner_join(season_STLadd, season_STLmult, by = '
        Month')
115 write_csv(seasonal_STLtable, './seasonalSTL.csv')
```

A.2.3 General visualization

The following code deals with more general visualization, illustrating the sample distribution of the US box office revenue feature together with another one of interest, the duration of each movie (see Figure 2.2).

Below you can also find the code used to plot Figures 2.6 and 2.7, used to explain white noise and differencing, respectively.

```
1 # Exploratory plots on revenue ------
2
3 dataset %>% ggplot(aes(x = usa_gross_income)) +
4   geom_boxplot() +
```

```
  theme(axis.ticks.y = element_blank(), axis.text.y = element_blank()) +
  xlab('USA Box Office Gross Income (Adjusted)') +
  scale_x_continuous(limits = c(80000, 16000000), labels = scales::label_dollar(scale = 1/10^6, suffix = 'M'))

dataset %>% ggplot(aes(x = usa_gross_income)) +
  geom_density(fill = 'lightblue') +
  ylab(element_blank()) +
  xlab('USA Box Office Gross Income (Adjusted)') +
  scale_y_continuous(labels = scales::format_format(scientific = TRUE)) +
  scale_x_continuous(limits = c(80000, 16000000), labels = scales::label_dollar(scale =1/10^6, suffix = 'M'))

skewness(dataset$usa_gross_income)
# Excess kurtosis
kurtosis(dataset$usa_gross_income)-3


# Artificial data plots ----
# Plotting of a simple white gaussian noise series
set.seed(1)
gaussian_whitenoise = tsibble(time = seq(1, 30, 1), noise = rnorm(30), index = time)
white_1 = autoplot(gaussian_whitenoise) + xlab('Time') + ylab('Noise')
white_2 = autoplot(ACF(gaussian_whitenoise, lag_max = 5)) + ylab('Autocorrelation') + xlab('Lag')
ggpubr::ggarrange(white_1, white_2)

# Plotting of a series with seasonality and trend + guassian noise
set.seed(1)
season_trend_series = tsibble(time = yearquarter(seq(from = as.Date('2000-01-01'), by = 'quarter', length.out = 100)),
                              response = seq(1, 100, 1)*0.1 +5*sin(pi/2*seq(1, 100, 1)) + 3*rnorm(100), index = time)
original_plot = autoplot(season_trend_series)+ylab('Response')
acf_original = autoplot(ACF(season_trend_series))+ylab('Autocorrelation')
detrended = season_trend_series %>% transmute(`Detrended Series` = difference(response, 1, 1))
stationary = detrended %>% transmute(`Stationary Series` = difference(`Detrended Series`, 4, 1))
stationary %>% fabletools::features(`Stationary Series`, c(~unitroot_kpss(.), ~feat_stl(., 4)))
stationary_plot = autoplot(stationary) + ylab('Response')
acf_stationary = autoplot(ACF(stationary)) + ylab('Autocorrelation')
```

```
ggpubr::ggarrange(original_plot, acf_original, stationary_plot,
    acf_stationary)
```

A.2.4 ARIMA and ETS modelling

Here you can find the code related to model fit, selection, and (cross) validation for ARIMA and ETS. Residuals from ARIMA are checked with a Ljung-Box test and visualized. Figures 3.2. 3.3 are defined here. The last lines of this code extract the fitted values from ARIMA that are used to build the response variable for the tree-based methods. Forecasts from ARIMA are also extracted in order to test the overall hybrid model afterwards.

```
# ARIMA and ETS modelling ------

# ETS/ARIMA 2016 Fitting, plotting and accuracy with SNAIVE
mable_models=compl_series %>% filter_index(~'Dec 2015') %>%
  model('ETS' = ETS(AverageMonthBoxOffice, ic = 'aic'),
        'ARIMA' = ARIMA(AverageMonthBoxOffice, ic = 'aic', stepwise=F
    , trace=T, approximation=FALSE),
        "SNAIVE" = SNAIVE(AverageMonthBoxOffice))

mable_models %>%
  select(-SNAIVE) %>%
  forecast(h = '1 year') %>% r44
  geom_ribbon(aes(group=`Model`, ymin=`.mean`-unlist(distributional::
    hilo(AverageMonthBoxOffice))[1],
                    ymax=`.mean`+unlist(hilo(AverageMonthBoxOffice))
    [2], fill=`Model`), alpha = 0.3)+
  scale_color_manual(values = c('red', 'blue')) +
  scale_y_continuous(labels = scales::label_dollar(scale =1/10^6,
    suffix = 'M'))+
  ylab("Average Montly Box Office Gross Income (US market)")+
  geom_line(data=compl_series %>% filter_index("2013"~.), mapping=aes
    (x=Month, y=AverageMonthBoxOffice))

test = mable_models %>%
  forecast(h="1 year")
fit_compare = mable_models %>%
  forecast(h="1 year") %>%
  accuracy(data=compl_series, measures=point_accuracy_measures)%>%
  select(-c(`.type`, `ME`, `MAE`,`MPE`,`MASE`,`RMSSE`,`ACF1`))

fit_compare%>%write_csv("./comparison.csv")


# Time series cross validation
```

```
cv_series = compl_series %>% stretch_tsibble(.step=12, .init=120)
cv_models=cv_series%>%model("ARIMA"=ARIMA(formula=
    AverageMonthBoxOffice~0+pdq(2,0,2)+PDQ(0,1,2)),
                    "ETS" = ETS(AverageMonthBoxOffice~error("M")+trend
    ("N")+season("M")),
                    "SNAIVE" = SNAIVE(AverageMonthBoxOffice))

accuracy_ts_cv = cv_models%>%
  forecast(h=12) %>%
  filter(!is.na(`.mean`)) %>%
  accuracy(compl_series, by=c('.model')) %>%
  select(-c(`.type`, `ME`, `MAE`,`MPE`,`MASE`,`RMSSE`,`ACF1`))

accuracy_ts_cv%>%write_csv("./accuracy_ts_cv.csv")


# Residual diagnosis/plot for ETS and ARIMA
first_tile = mable_models%>%select(ARIMA)%>%residuals()%>%select(-.
    model)%>%
  rename(Residual=`.resid`) %>%
  filter_index("March 1984"~.) %>%
  ggplot(aes(x=Month, y = Residual)) +
  geom_line()+
  geom_point()

second_tile = mable_models%>%select(ARIMA)%>%residuals()%>%select(-.
    model)%>%
  rename(Residual=`.resid`) %>%
  filter_index("March 1984"~.) %>%
  ggplot() +
  geom_histogram(aes(x=Residual), color="blue", fill = "lightblue",
    alpha=0.8)+
  ylab("# of occurrences")

third_tile = mable_models %>% select(ARIMA)%>%residuals()%>%select(-.
    model)%>%
  rename(Residual=`.resid`)%>%
  filter_index("March 1984"~.) %>% ACF() %>% autoplot() +
  ylab(NULL) + xlab(NULL)

ggpubr::ggarrange(first_tile,
                    ggpubr::ggarrange(second_tile, third_tile), nrow=2)

# Ljung-Box and Box test on the residuals to check that they are
    white noise
mable_models%>%select(ARIMA)%>%residuals()%>%select(-.model)%>%
  rename(Residual=`.resid`) %>% fabletools::features(Residual, c(
    box_pierce, ljung_box), lag=24, dof=6)

```

```
fit_arima = mable_models %>% select(ARIMA)

fitted_values = fit_arima %>% fitted()
```

A.2.5 Building the document-term matrix and the dummy variables: preparing the dataset of exogenous regressors for modelling

As explained in Chapter 4, the procedure to pre-process the exogenous regressors is not easy and not simple. In this part of the Appendix the code of that pre-processing pipeline is reported and explained.

As can be read in the comments inside the R script, the piece of code below extracts categorical variables (labels) from text fields that contain them. The original language, the genres and the production countries of each movie are reported in a list-type format, with every label separated by a comma. A dummy variable has to be built for every label, or at least for the most significant ones of each set.

```
# This part of the pre-processing is dedicated to extracting
# categorical variables from text fields where they are found
    together.
# Dummies are built for each of them
# Trimming of whitespace needs to be done before extracting the
    labels from the text field
# A list of the unique values for each field we are
# extracting labels/categorical variables from
# has to be saved for the following operations,
# since it is needed for the "across" tidyverse operator
# This needs to be done more than once, so I will define a function

splitter_categ = function (vec) {
  empty_vec = c()
  for (element in vec) {
    row_el = unlist(str_split(element, ','))
    for (item in row_el) {
      if (!is.element(item, empty_vec)) {
        empty_vec = append(empty_vec, item)
      }
    }
  }
  empty_vec = paste(paste('^', empty_vec, sep = ''), '$', sep = '')
  # regex expression to enforce strict match
  return(empty_vec)
}
```

```
25
26 # First in line is genre
27 dataset = dataset %>% mutate(genre = gsub(" ", "", genre, fixed =
       TRUE))
28 genre_vec = splitter_categ(dataset$genre)
29
30 dataset = dataset %>%
31   separate_rows(genre, sep = ',') %>% pivot_wider(values_from = genre
       , names_from = genre) %>%
32   mutate(across(matches(genre_vec), ~is.na(.))) %>%
33   mutate(across(matches(genre_vec), ~ifelse(.==TRUE, 0, 1)))
34
35 # Now we should do the same with languages and country, but selecting
       only the
36 # most prevalent languages and countries, manually. Italy and Italian
       are a bonus, since I am Italian
37 # Selection according to most spoken languages in the world
38 dataset = dataset %>% mutate(language = gsub(" ", "", language, fixed
       = TRUE))
39 language_vec = splitter_categ(dataset$language)
40
41 dataset_languages = dataset %>%
42   separate_rows(language, sep = ',') %>% pivot_wider(values_from =
       language, names_from = language) %>%
43   mutate(across(c(matches(language_vec)), ~is.na(.))) %>%
44   mutate(across(c(matches(language_vec)), ~ifelse(.==TRUE, 0, 1)))
       %>%
45   select(English, Chinese, Hindi,
46          Spanish, French, Arabic, Bengali, Russian, Portuguese,
       Indonesian,
47          Urdu, German, Italian)
48
49 dataset = cbind(dataset %>% select(-language), dataset_languages)
50
51 # The same also for country, selection according to volume of
       production data
52 # https://www.the-numbers.com/movies/production-countries/#tab=
       territory
53
54 dataset = dataset %>% mutate(country = gsub(" ", "", country, fixed =
       TRUE))
55 country_vec = splitter_categ(dataset$country)
56 dataset_country = dataset %>%
57   separate_rows(country, sep = ',') %>% pivot_wider(values_from =
       country, names_from = country) %>%
58   mutate(across(matches(country_vec), ~is.na(.))) %>%
59   mutate(across(matches(country_vec), ~ifelse(.==TRUE, 0, 1))) %>%
60   select(USA, UK, China, France, Japan, Germany, SouthKorea, Canada,
       Australia, India,
```

```
61            NewZealand, HongKong, Italy, Spain, Russia)
62
63 dataset = cbind(dataset %>% select(-country), dataset_country)
```

Now we need to extract information in a structured way from text, from the title and the synopsis of each movie. A very simple approach - when compared to the most basic models of the modern NLP world, like Word2Vec - is just to build a Document-Term matrix, using as weight the TF-IDF of each term and putting in place document frequency boundaries in order to reduce the dimensionality of the matrix. The *tm* R library takes care of the necessary pre-processing steps[3] with its *TermDocumentMatrix* function.

```
1 # let's put the English title in the dataset, it is more meaningful
2 # for text mining in our context (USA boxoffice)
3
4 # This is a dataframe which comes from the only piece of Python code
    which is not in the thesis,
5 # It is a script to scrape posters, which were not included in the
    final pipeline in the end,
6 # since deep learning and CNN were eventually excluded
7
8 # The only important thing is that it contains English titles (
    scraped with the TMDB API)
9 images_downloaded = read_csv(paste('.','images_downloaded.csv', sep =
     path_separator))
10 images_downloaded = images_downloaded %>% rename(IMDb_ID = 'IMDbID')
11 dataset = inner_join(dataset, images_downloaded, by = 'IMDb_ID', keep
     = F) %>%
12   select(-`Poster URL`) %>% relocate(`English title`,.before =
    original_title)
13
14
15 # Preparing a BOW for successive text mining steps, concerning title
    and synopsis
16 # The term document matrix function of the tm library takes care of
    everything,
17 # without us needing to preprocess words manually with a pipe.
18 # Important pre-processing steps that are performed are lower case
    conversion, stemming,
19 # stopwords removal and punctuation removal
20 # Document frequency bounds are put in place to
21 # reduce at least in part the dimensionality of the term document
    matrix
22 # TF-IDF is chosen as weight for the resulting term document matrix
```

[3]Stemming, punctuation removal, lower case conversion, stopwords removal.

```
23
24 control_list1 = list(tokenize = words, language = 'en',
25                       bounds = list(global = c(20, 300)), weighting =
       weightTfIdf, tolower = TRUE,
26                       removePunctuation = TRUE, stopwords = TRUE,
       stemming = TRUE)
27 bowTitle = tm::TermDocumentMatrix(tm::VCorpus(tm::VectorSource(
       dataset$`English title`)),
28                           control_list1)
29 bowTitle = t(as.matrix(bowTitle))
30 bowTitle = as.data.frame(bowTitle)
31 colnames(bowTitle) = paste(colnames(bowTitle), '_title')
32 bowTitle = cbind(dataset$TMDB_ID, bowTitle) %>% rename('TMDB_ID' = `
       dataset$TMDB_ID`)
33
34 control_list2 = list(tokenize = words, language = 'en',
35                       bounds = list(global = c(60, 300)), weighting =
       weightTfIdf, tolower = TRUE,
36                       removePunctuation = TRUE, stopwords = TRUE,
       stemming = TRUE, removeNumbers = TRUE)
37 bowDescription = tm::TermDocumentMatrix(tm::VCorpus(tm::VectorSource(
       dataset$description)),
38                                    control_list2)
39 bowDescription = t(as.matrix(bowDescription))
40 bowDescription = as.data.frame(bowDescription)
41 colnames(bowDescription) = paste(colnames(bowDescription), '_des')
42 bowDescription = cbind(dataset$TMDB_ID, bowDescription) %>% rename('
       TMDB_ID' = `dataset$TMDB_ID`)
```

The dummies for the directors and for the production companies are also added, considering only the highest ranking based on the frequency in the dataset. Results from various stages are merged and the dataset is now ready to be used to fit the tree-based methods.

```
1 # We also add some dummies to encode the director and production
      companies
2 # First, we check the most popular entries of each and save them in
      two vectors
3 # These popular entries are the only ones that are dummied (otherwise
       we would have way too much dummy
4 # variables and the matrix is already too sparse as it is)
5
6 pop_directors = dataset %>% group_by(director) %>% summarise(n()) %>%
7   arrange(desc(`n()`)) %>% slice_head(n = 20) %>% select(-`n()`) %>%
      unlist()
8 pop_production = dataset %>% group_by(production_company) %>%
      summarise(n()) %>%
```

```
  arrange(desc(`n()`)) %>% slice_head(n = 21) %>% filter(complete.
    cases(.)) %>%
  select(-`n()`) %>% unlist()
pop_directors = paste('director_', pop_directors, sep = '')
pop_production = paste('production_company_', pop_production, sep='')

directors_dummy = dataset %>% fastDummies::dummy_columns('director')
  %>%
  select(all_of(pop_directors), 'TMDB_ID') %>% mutate(across(
    pop_directors, ~replace_na(., 0)))
production_dummy = dataset %>% fastDummies::dummy_columns('
    production_company') %>%
  select(all_of(pop_production), 'TMDB_ID') %>% mutate(across(
    pop_production, ~replace_na(., 0)))


# Merging the datasets
# The last three lines are needed to avoid 'bad names' as column
    names

dataset_treated = dataset %>% left_join(bowTitle, by = 'TMDB_ID') %>%
  mutate(across(colnames(bowTitle), ~replace_na(., 0))) %>% left_join
    (bowDescription, by = 'TMDB_ID') %>%
  mutate(across(colnames(bowDescription), ~replace_na(., 0))) %>%
  inner_join(directors_dummy, by = 'TMDB_ID') %>% inner_join(
    production_dummy, by = 'TMDB_ID') %>%
  rename_with(~str_remove_all(., "[\n\\s]|\\(|\\)"), everything())
    %>%
  rename_with(~str_replace_all(., "-", '_'), everything())
```

The last step missing is adding the response for the fit. We take the difference between the gross box office revenue of each movie and the fitted value from ARIMA for the month of the release date. Train and test (observations from 2016) datasets are then obtained, and the test dataset is joined with the forecasts from ARIMA for that time span.

```
# We join the fitted values from ARIMA with the dataset and
# then we take the difference between the USA boxoffice and
# the fitted values. This will be the response of the tree-based
    models
fitted_values = fitted_values %>% select(-.model)
fitted_values = fitted_values %>% rename(Fitted = '.fitted')
dataset_treated = right_join(as_tibble(fitted_values),
    dataset_treated, by = 'Month') %>%
  relocate(Fitted, .after = worlwide_gross_income) %>%
  mutate(Response = ifelse(is.na(Fitted), NA, usa_gross_income-Fitted
    )) %>%
```

```
  relocate(Response, .after = Fitted)

# Let's stop the dataset for the model to 2015, in order to have a train test split
# Observations of 2016 are used for the testing of the overall model,
# like we have done with ARIMA
dataset_test = dataset_treated %>% filter(format(Month, '%Y')==2016)
dataset_train = dataset_treated %>% filter(format(Month, '%Y')<2016)

test_arima_forecast %>% as_tibble()
dataset_test = dataset_test %>% inner_join(test_arima_forecast, by = 'Month') %>% mutate(Fitted = `.mean`) %>% select(-.mean)

dataset_train = dataset_train %>% select(-c(Month,title, original_title, avg_vote, votes,
                        budget, metascore, reviews_from_users, reviews_from_critics, director, writer, production_company, actors,
                        description, worlwide_gross_income, Englishtitle))

dataset_test = dataset_test %>% select(-c(Month,title, original_title, avg_vote, votes,
                                          budget, metascore, reviews_from_users, reviews_from_critics, director, writer, production_company, actors,
                                          description, worlwide_gross_income, Englishtitle))
```

A.2.6 Fitting the tree based-methods and predictions from the overall hybrid model

The following code illustrates the fitting procedure for what concerns gradient boosting and bagging. The implementation used the R *caret* package, which offers a nice interface for an extensive set of underlying models which are supported from the respective libraries. Since brute-forcing the hyperparameter tuning (for gradient boosting) with cross-validation is one of the most expensive tasks in machine learning, the execution was parallelized in order to drastically reduce computation time.

Gradient boosting uses the stochastic implementation from Friedman (2001)[14], sampling without replacement 70% of the dataset for the fit of each weak learner.

Naturally, Figures 4.2 and 4.1 come from this piece of code.

```
# BAGGING
```

```
2
3  bag.control = trainControl(
4    method = 'repeatedcv',
5    number = 10,
6    repeats = 3,
7    verboseIter = TRUE,
8    summaryFunction = defaultSummary,
9    allowParallel = TRUE
10 )
11
12 n_cores = detectCores() - 4
13 cl = makeCluster(n_cores, type = 'FORK', outfile = "")
14 registerDoParallel(cl)
15 bag.fit = train(
16   Response ~.-IMDb_ID-TMDB_ID-usa_gross_income-Fitted,
17   method = 'rf',
18   ntree = 1000,
19   data = dataset_train,
20   trControl = bag.control,
21   metric = 'RMSE',
22   tuneGrid = data.frame(mtry = ncol(dataset_train)-4)
23 )
24 stopCluster(cl)
25 bag.fit$results
26
27 importance_bag = importance(bag.fit$finalModel, type=2)
28 importance_bag = tibble(name = rownames(importance_bag), `Reduction
       in RSS` = importance_bag)
29 attr(importance_bag$`Reduction in RSS`, "dimnames") = NULL
30 importance_bag$`Reduction in RSS`=importance_bag$`Reduction in RSS`[,
       1]
31
32 importance_bag %>% arrange(desc(`Reduction in RSS`)) %>% head(10) %>%
33   ggplot(aes(x=reorder(name, `Reduction in RSS`), y = `Reduction in
       RSS`)) +
34   geom_bar(stat="identity", fill = "blue") +xlab(NULL) + coord_flip()
35
36 # BOOSTING
37
38 set.seed(1)
39 boosting_control = trainControl(
40   method = 'cv',
41   number = 10,
42   summaryFunction = defaultSummary,
43   allowParallel = TRUE,
44   verboseIter = TRUE
45 )
46
```

```
47 tune_grid_boost = expand.grid(n.trees = seq(100, 1000, by = 100),
       interaction.depth = seq(1, 8, by = 1),
48                                shrinkage = seq(0.02, 0.1, by = 0.02),
        n.minobsinnode = 10)
49
50  n_cores = detectCores()-4
51  cl = makeCluster(n_cores, type = 'FORK', outfile= "")
52  registerDoParallel(cl)
53  boost.fit = train(
54    Response ~.-IMDb_ID-TMDB_ID-usa_gross_income-Fitted,
55    method = 'gbm',
56    bag.fraction = 0.7,
57    distribution = 'gaussian',
58    data = dataset_train,
59    trControl = boosting_control,
60    metric = 'RMSE',
61    verbose=TRUE,
62    tuneGrid = tune_grid_boost
63  )
64
65  stopCluster(cl)
66  boost.fit$bestTune
67
68  # Variable importance plot boosting
69  variable_importance = tibble(names = boost.fit$finalModel$var.names,
       importance = gbm::permutation.test.gbm(boost.fit$finalModel,
70
                                              n.trees = 800))
71  variable_importance = variable_importance %>% arrange(desc(
      importance))
72  grid_3 = variable_importance %>% head(10) %>% ggplot(aes(x=reorder(
      names, importance), y =importance)) +
73    geom_bar(stat = "identity", fill="blue") + coord_flip() + xlab(
      NULL) + ylab("Decrease in performance")
74
75  # CV plots boostings
76  cv_result = boost.fit$results
77  grid_cv_1 = cv_result %>% group_by(n.trees) %>% summarise(
      AverageRMSE = mean(RMSE)) %>%
78    ggplot(aes(x=n.trees, y=AverageRMSE)) +
79    geom_line(color="blue") + ylab("Average RMSE") + xlab("# of weak
      learners")
80
81  grid_cv_2 = cv_result %>% group_by(shrinkage) %>% summarise(
      AverageRMSE=mean(RMSE)) %>%
82    ggplot(aes(x=shrinkage, y=AverageRMSE)) +
83    geom_line(color="blue") + ylab("Average RMSE") + xlab("Shrinkage
      rate")
84
```

```
85 ggpubr::ggarrange(ggpubr::ggarrange(grid_cv_1, grid_cv_2), grid_3,
     nrow = 2)
```

At this point, computing the predictions from the overall hybrid model on test data is easy, since we just have to add the ARIMA forecasts to the predictions from gradient boosting. Computing the resulting R^2 on test data is also simple: it is just the ratio between the variance of the residuals and the variance of the response.

```
1 boost_test_pred = predict(boost.fit$finalModel, dataset_test, 800)
2 dataset_test = dataset_test %>% add_column("prediction_boosting"=
    boost_test_pred)
3 dataset_test = dataset_test %>% mutate("hybrid model preditions"=
    Fitted + prediction_boosting)
4 dataset_test = dataset_test %>% mutate("hybrid_model_errors" = `
    hybrid model preditions`- usa_gross_income)
5 var(dataset_test$hybrid_model_errors)/var(
    dataset_test$usa_gross_income)
```

Bibliography

[1] Ni Guo, Wei Chen, Manli Wang, Zijian Tian, and Haoyue Jin. «Appling an Improved Method Based on ARIMA Model to Predict the Short-Term Electricity Consumption Transmitted by the Internet of Things (IoT)». In: *Wireless Communications and Mobile Computing* 2021 (Apr. 2021), p. 6610273. ISSN: 1530-8669. DOI: 10.1155/2021/6610273. URL: https://doi.org/10.1155/2021/6610273 (cit. on p. 4).

[2] Fazil Kaytez. «A hybrid approach based on autoregressive integrated moving average and least-square support vector machine for long-term forecasting of net electricity consumption». In: *Energy* 197 (2020), p. 117200. ISSN: 0360-5442. DOI: https://doi.org/10.1016/j.energy.2020.117200. URL: https://www.sciencedirect.com/science/article/pii/S0360544220303078 (cit. on p. 4).

[3] R.J. Hyndman and G. Athanasopoulos. «Forecasting: Principles and Practice». In: Third Edition. otexts.com/fpp3/decomposition.html | Constantly updated, freely accessible online, visited 16-07-2022. Melbourne, AU: OTexts, 2021. Chap. 3 (cit. on pp. 6, 9, 10, 22).

[4] R.J. Hyndman and G. Athanasopoulos. «Forecasting: Principles and Practice». In: Third Edition. otexts.com/fpp3/graphics.html | Constantly updated, freely accessible online, visited 16-07-2022. Melbourne, AU: OTexts, 2021. Chap. 2 (cit. on pp. 8, 12, 22).

[5] R. B. Cleveland, W. S. Cleveland, J. E. McRae, and I. J. Terpenning. «STL: A Seasonal Trend Decomposition Procedure Based on LOESS». In: *Journal of Official Statistics* 6 (Jan. 1990), pp. 3–73 (cit. on p. 9).

[6] G. James, D. Witten, T. Hastie, and Tibshirani R. «Introduction to Statistical Learning». In: Second Edition. Freely available at statlearning.com. Springer Science+Business Media, 2021. Chap. 7.6, pp. 304–306 (cit. on p. 9).

[7] R.J. Hyndman and G. Athanasopoulos. «Forecasting: Principles and Practice». In: Third Edition. otexts.com/fpp3/https://otexts.com/fpp3/arima.html | Constantly updated, freely accessible online, visited 16-07-2022. Melbourne, AU: OTexts, 2021. Chap. 9 (cit. on pp. 11, 15, 20–22).

[8] R.J. Hyndman and G. Athanasopoulos. «Forecasting: Principles and Practice». In: Third Edition. https://otexts.com/fpp3/toolbox.html | Constantly updated, freely accessible online, visited 16-07-2022. Melbourne, AU: OTexts, 2021. Chap. 5 (cit. on pp. 14, 22, 24).

[9] R.J. Hyndman and G. Athanasopoulos. «Forecasting: Principles and Practice». In: Third Edition. https://otexts.com/fpp3/expsmooth.html | Constantly updated, freely accessible online, visited 16-07-2022. Melbourne, AU: OTexts, 2021. Chap. 8 (cit. on pp. 17, 18, 22).

[10] George E.P. Box and Gwilym M. Jenkins. *Time Series Analysis: Forecasting and Control*. Holden-Day, 1970 (cit. on p. 19).

[11] Leo Breiman. «Bagging predictors». In: *Machine Learning* 24.2 (Aug. 1996), pp. 123–140. ISSN: 1573-0565. DOI: 10.1007/BF00058655. URL: https://doi.org/10.1007/BF00058655 (cit. on p. 28).

[12] Leo Breiman. «Random Forests». In: *Machine Learning* 45.1 (Oct. 2001), pp. 5–32. ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324 (cit. on p. 28).

[13] Jerome H. Friedman. «Greedy function approximation: A gradient boosting machine.» In: *The Annals of Statistics* 29.5 (2001), pp. 1189–1232. DOI: 10.1214/aos/1013203451. URL: https://doi.org/10.1214/aos/1013203451 (cit. on p. 28).

[14] Jerome H. Friedman. «Stochastic gradient boosting». In: *Computational Statistics Data Analysis* 38.4 (2002). Nonlinear Methods and Data Mining, pp. 367–378. ISSN: 0167-9473. DOI: https://doi.org/10.1016/S0167-9473(01)00065-2. URL: https://www.sciencedirect.com/science/article/pii/S0167947301000652 (cit. on pp. 28, 32, 57).

[15] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. «The Elements of Statistical Learning». In: Second Edition. Springer New York, 2009. Chap. 10 (cit. on p. 28).