Enhancing Babies' Sleep Schedule Prediction through Machine Learning

UNIVERSITY OF TURKU
Department of Computing
Master of Science (Tech) Thesis
Health Technology
June 2024
Anna Fernandez-Rajal i Sabala

Supervisors:
Academic Ph.D. Tapio Pahikkala
M.S. Yuning Wang
Company Alexander Brokking

This Master Thesis has been done within a double degree at EIT Digital School, **KTH Royal Institute of Technology** and **University of Turku**. It has been conducted at the company **Napper** [1].

UNIVERSITY OF TURKU
Department of Computing

Anna Fernandez-Rajal i Sabala: Enhancing Babies' Sleep Schedule Prediction through Machine Learning

Master of Science (Tech) Thesis, 136p.
Health Technology
June 2024

In recent years, there has been a growing interest in improving sleep quality and understanding sleep patterns. This thesis will focus on enhancing sleeping schedules for babies through machine learning. Establishing a consistent bedtime routine at a young age is crucial, as it offers numerous health benefits and can prevent sleep-related issues later in life. Despite this, many parents still find it challenging to manage their babies' sleep schedules effectively.
This master's thesis explores the integration of machine learning algorithms into babies' sleep schedule predictions to provide more accurate and personalized recommendations. It focuses on the integration of advanced data analytics and processing techniques to improve sleep forecasts. Recognizing the importance of data cleanliness and processing efficiency, the research delves into different steps for preparing and analyzing the dataset on sleep and baby tracking information. With a strong focus also on feature analysis, it later dives into various machine learning models and assesses their effectiveness and performance. The regression task with machine learning models includes K-Nearest Neighbors (KNN), XGBoost, Random Forests (RF), Long Short-Term Memory networks (LSTM), and Recurrent Neural Networks (RNN).
The project offers a methodical approach that includes background information, relevant literature, dataset specifics, suggested techniques, findings, and conclusions. The results demonstrate a clear potential to improve the current sleep schedules with machine learning to achieve the desired goals, with performance metrics showing proximity to baseline expectations. This thesis contributes to the field by advancing the methodology of baby sleep tracking, ultimately aiming to enhance the well-being of infants and ease the challenges faced by parents in managing their babies' sleep routines.

Keywords: machine learning, data analytics, baby sleep schedules, feature analysis

Contents

List of Figures

List of Tables

List of acronyms

CNN	Convolutional Neural Network
DL	Deep Learning
KNN	K-Nearest Neighbor
LR	Logistic Regression
LSTM	Long Short-term Memory
MAE	Mean Absolute Error
MedAE	Median Absolute Error
MLP	Multilayer Perceptron
ML	Machine Learning
MSE	Mean Squared Error
NLP	Natural Language Processing
RF	Random Forest
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic curve
SVM	Support Vector Machines

1 Introduction

Optimizing and understanding sleep patterns, especially in infants, has been increasingly researched recently. Studies have revealed that a regular and strong nighttime routine is essential to a child's growth and general well-being [2] [3]. A regular sleep schedule in early childhood is associated with many health advantages and can avoid later sleep-related problems [4] [5]. However, it can be difficult to establish a consistent sleep schedule for infants, particularly those under the age of two years [6]. Hence, there is a pressing need for effective methods that can help both newborns and their caregivers get better sleep.

Napper, a popular app designed for parents [1], focuses on improving babies' sleep. It has gained the trust of many parents worldwide and stands out as a leading parenting app. This project, in collaboration with Napper, aims to enhance the prediction of baby sleep patterns by integrating machine learning algorithms into Napper's existing framework. By leveraging a comprehensive dataset on baby tracking and sleep patterns, the project aims to optimize sleep schedule predictions through data-driven methods. The thesis details the process of data processing and feature engineering, followed by an evaluation of various machine learning and neural network models to provide a robust comparison.

By systematically analyzing and integrating these advanced predictive models, the project aspires to significantly improve the accuracy and effectiveness of baby sleep schedule predictions. Ultimately, this would contribute to better sleep quality for infants and a more manageable routine for their parents or caregivers.

1.1 Problem statement

The primary objective of this thesis is to integrate machine learning algorithms into the prediction of baby sleep schedules. This integration aims to significantly enhance the accuracy and personalization of sleep pattern predictions, offering users precise insights specifically tailored to their children.

The intended outcomes include increased precision and accuracy in sleep schedule recommendations, leading to more effective and customized solutions. Ultimately, the thesis aims to transform the current sleep scheduling approach into a more advanced, data-driven model, thereby improving sleep outcomes for babies and easing the management of sleep routines for parents. The processing and feature engineering of the data will play a key role in achieving these results.

1.2 Research questions

- What data processing steps are required to ensure a realistic representation of babies' sleep patterns collected from an app, considering varying levels of adherence by users?
- Which features significantly enhance the predictive capabilities of sleep timings within the available data?
- How can machine learning algorithms be effectively integrated to improve the accuracy and efficacy of sleep suggestions for users?
- Given user-specific data, which machine learning models are better suited to provide precise sleep schedule recommendations that generalize across individual variances?

1.3 Contributions

This master's thesis has been conducted in collaboration with Napper, a leading company in the parenting and baby sleep sector. The project has significantly benefited from Napper's extensive resources, including its comprehensive data repository and internal software development experience. The integration of advanced machine learning algorithms into Napper's existing infrastructure has been made possible through the company's expertise in the digital healthcare industry.

Napper's current algorithmic framework served as a foundational platform for this project, enabling the seamless incorporation and enhancement of predictive models. Napper's established resources and collaborative efforts ensured that the advancements made were not only theoretically sound but also practically applicable within Napper's operational environment.

By combining academic research with practical industry applications, this thesis aims to contribute to both the scientific community and Napper's mission to improve baby sleep schedules through advanced technology.

1.4 Delimitations

One of the major limitations of this thesis lies in the inherent challenges associated with the processing and utilization of the collected data. The reliance on user adherence for data entry introduces potential inaccuracies and inconsistencies, thereby compromising the overall quality of the dataset.

Furthermore, the dropout rate among app users poses a significant constraint on the dataset's comprehensiveness and representation. Particularly as babies transition beyond the specified age range of interest for the app, the dropout rates become higher. As a result, the analyses and conclusions drawn from the available data may not fully capture the diversity and complexity of sleeping behaviors exhibited by babies during their first two years of life.

Another notable constraint arises from the inherent biases present within the user-recorded data, influenced by the previously recommended schedule provided by the app before users logged the data. These biases may lead to skewed or idealized representations of sleeping patterns, rather than an objective portrayal of reality. Consequently, the dataset may not accurately reflect the true distribution of sleeping behaviors among babies, thereby limiting the generalization and applicability of the research findings.

Addressing these limitations through rigorous quality control measures and processing the data beforehand could enhance the reliability and validity of future research in this domain. This would ultimately contribute to a more comprehensive understanding of babies' sleep habits and could provide important information about the impact on child development and well-being.

1.5 Structure

The remainder of this thesis report is organized as follows. Chapter 2 provides the theoretical background on babies' sleeping schedules and patterns. It also includes technical background and description of the machine learning models used and the evaluation metrics. Chapter 3 presents related scientific work which supports the chosen methodology. Chapter 4 provides details on the dataset used for the project, including the data collection and initial data cleaning. Chapter 5 delves into the experimental setup and the evaluative approach, detailing the feature extraction and selection process and comparing different proposed machine learning models for predicting babies' sleeping schedules.

Chapter 6 evaluates the performance of all implementations and compares the results. Finally, Chapter 7 presents a comparison of the different algorithms, addresses the limitations, and provides recommendations for future work.

2 Background

This chapter describes the background knowledge necessary to understand the methodology of this thesis. Section 2.1 and Section 2.2 provide brief overviews of babies' sleeping habits and the processing techniques employed, respectively. Section 2.3 describes the machine learning algorithms used in the methodology of this thesis. Moreover, Section 2.4 includes an overview of the evaluation metrics used to evaluate the results and to compare the performances of the models.

2.1 Babies' sleeping habits

Promoting a baby's health and well-being throughout infancy and beyond requires an understanding of their sleeping patterns. Sleep is essential for an infant's cognitive development, emotional control, and physical growth [7], [8]. Babies' sleep habits, however, vary greatly based on a number of factors, including age, personality traits, and their surroundings [9]–[11].

The domain of pediatric sleep medicine has led to significant findings regarding the factors that affect infants' sleep patterns and the potential impact of sleep problems on their growth and well-being [9]. Previous studies, described next, have examined infants' sleep duration and patterns at different developmental stages.

Research published [12], [13] has found that infants undergo significant changes in sleep times during the first year of life, with gradual consolidation of nighttime sleep and decreasing frequency of nighttime awakenings. Understanding these de-

Stage	**Age (months)**	**Naps (number)**	**Sleep - day (hours)**	**Sleep - night (hours)**	**Total sleep (hours)**
Newborn	0–4	3–5	7–9	8–9	16–18
Infant	4–12	2–3	4–5	9–10	12–16
Toddler	12–24	2	2–3	11	11–14

Table 2.1: Sleep Charting by Age [15]

velopmental changes is key for parents and caregivers in establishing healthy sleep habits in infants. Several resources recommend certain sleep times during the day [10], [14], [15]. Infants from 4 to 12 months should sleep 12 to 16 hours each day (including naps) to promote optimal health [15]. For babies up to 2 months old, the lack of a developed circadian rhythm makes it harder to follow these indicators [16]. Children from 1 to 2 years of age should sleep 11 to 14 hours on a regular basis. The number of recommended naps per day and the duration of sleep are shown in Table 2.1 [15].

As mentioned previously, several factors can affect how well a baby sleeps, including the outside world, the behavior of the parents, and the traits of the infant. Infants' sleep onset and quality are influenced by parental activities such as bedtime routines and sleep cues [11]. Additionally, an infant's ability to fall and stay asleep can be impacted by factors such as lighting, noise levels, and room temperature.

Children can develop sleep disorders or unhealthy sleep behavior later in life, which may be consequences of experiences as infants. Research has been done into the prevalence of sleep issues in newborns and its correlation with parental mental health [17]. Identifying and addressing sleep disorders in infancy is essential for promoting healthy sleep and preventing long-term consequences.

Research in this area, targeted at resolving infant sleep issues and promoting healthy sleeping habits is increasingly important. Behavioral interventions, such as parental education courses and sleep training methods, have been shown to be successful in helping babies develop healthy sleeping patterns [18]. When adopting such methods into practice, it is essential to take personal traits and cultural considerations into account.

In conclusion, supporting a baby's general health and well-being requires an understanding of their sleeping patterns. This field of study has yielded important insights into the factors affecting newborns' sleep patterns and duration, the frequency of sleep issues, and the most effective methods to treat sleep disorders. Healthcare providers can help infants get the sleep they require for optimal development by promoting evidence-based strategies and offering implementation assistance to parents and carers.

2.2 Preprocessing techniques

Data preprocessing is a critical step in machine learning (ML) pipelines, and the choice of scaling technique drastically influences model performance. Z-score standardization and min-max standardization, two prevalent scaling methods, offer distinct advantages tailored to the requirements of different learning algorithms.

Z-score standardization operates by transforming existing features to have a mean of zero and a standard deviation of one. This normalization ensures that each feature contributes proportionately to the learning process. It is particularly relevant in neural networks such as Long Short-Term Memory (LSTM) networks, as will be seen later. These networks rely on activation functions such as sigmoid or tanh, which are sensitive to the scale of input data. The transformation for z-score standardization can be expressed mathematically as:

$$z = \frac{x - \mu}{\sigma}$$

The standardized value is denoted by z, the original feature value is denoted by x, the feature mean is denoted by μ, and the standard deviation is denoted by σ.

Min-max normalization rescales features to a given range, usually between zero and one. By employing a simple transformation formula, min-max normalization ensures that each feature is bounded within the designated range. The transformation for min-max normalization can be expressed mathematically as:

$$x_{\text{scaled}} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

In this case, x_{scaled} represents the features' scaled value, x represents its original value, $\min(x)$ represents its minimum value, and $\max(x)$ represents its highest value.

In conclusion, selecting suitable preprocessing methods is essential for optimizing model performance. Leveraging z-score standardization for neural networks and min-max normalization for algorithms such as XGBoost enables different preprocessing methods, enhancing model effectiveness across varied datasets. These scaling techniques not only ensure model stability and convergence, but also ensure better generalization in ML applications.

2.3 Machine Learning Models

ML revolves around algorithms which enhance their performance through data analysis. ML has applications across several domains such as computer vision, speech recognition, and medicine [19], [20]. A ML model must learn from a selection of data. This learning phase, where the model studies examples, is known as training. Once the model has been trained, we use new and unseen data points for the evaluation. A successful ML model depends on its ability to generalize, in order to accurately predict the outcomes for unseen data.

ML primarily employs three methodologies: supervised learning, unsupervised learning, and reinforcement learning [21], [22]. This thesis concentrates on supervised learning techniques. In supervised learning, data comes already labeled with target values or outcomes. This means that each training example in the dataset has an associated label or response that the model aims to predict.

Within supervised learning, there are two main types of tasks: classification and regression. Classification involves predicting a categorical label or class. Regression, however, is aimed at predicting continuous numerical values [19], [23]. For instance, predicting the price of a house based on existing features such as size, location, age, and previous house prices. These labels allow the supervised learning algorithm to learn the relationship between the input features and the output labels, which can then be used to make predictions on new, unseen data. This predictive modeling technique is used in many fields, from finance to healthcare [24], [25]. This thesis focuses on supervised regression learning to predict sleep schedules.

2.3.1 Regression

Training a regression model involves using historical data where the target variable (e.g., house price) is known. This dataset, named the training set, serves as the foundation for the model to learn patterns and relationships between input features and the target variable. After the model has been trained, its performance is assessed using a different dataset, called the test set, which includes examples that have not yet been seen (e.g., future home prices) [26] [27].

The ability of the model to generalize, or generate correct predictions on new, unseen data, is a key component of regression. A well-generalized regression model can effectively estimate outcomes for scenarios that were not seen in the training phase. Here, the historical data includes input-output pairs, with the goal being to learn a mapping from inputs to continuous outputs.

This thesis employs and compares several ML models. Among the classical ML methods, we will use K-Nearest Neighbours (KNN), XGBoost, and Random Forest Regressor (RF). Neural network structures will also be tested, such as Recurrent

Neural Networks (RNN) and Long Short-Term Memory (LSTM) networks, which are known to be effective in processing sequential data [28].

Basic Machine Learning Models

- **KNN**

  KNN regression predicts the continuous value of the target variable by averaging or weighting the values of its k nearest neighbors [29]. With a distance metric (such as the Euclidean distance) to locate the k nearest neighbors of a given query point, KNN predicts the value of the query point by averaging (or weighted averaging) the values of their target variables.

  The selection of k is crucial in KNN regression as it determines the balance between variance and bias. A smoother model with lower variance and potentially higher bias is generated with a larger k, while a smaller k results in a more flexible model with higher variation and lower bias. Based on the dataset's characteristics, the chosen distance metrics and weighting options can be applied to adapt the model better to the dataset.

- **XGBoost**

  XGBoost is an ensemble learning technique designed to predict continuous target variables by iteratively generating a sequence of decision trees [30]. Each decision tree in XGBoost regression is trained to minimize a loss function, for example, the mean squared error (MSE), which calculates the variance between the values predicted and those observed. The XGBoost regression's objective function is provided by:

$$\text{obj} = \sum_{i=1}^{n} L(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

  where the loss function is $L(y_i, \hat{y}_i)$, the predicted value is $\hat{y}_i$, the regularisation term is $\Omega(f_k)$, which controls the complexity of each of the trees, and finally the number of trees is regulated by K.

In order to minimize the objective function and promote greedy tree growth, XGBoost uses gradient descent optimization. Pruning and shrinkage (learning rate) are two regularisation strategies that are used to improve generalization performance and avoid overfitting.

- **Random Forest**

  RF regression is an ensemble learning that creates several decision trees during training and produces the average prediction of each tree. By combining the strength of ensemble learning with the simplicity of decision trees, it can reduce overfitting and increase prediction accuracy [31].

  There are multiple steps in the RF algorithm. Initially, it uses bootstrap sampling to randomly select a number of samples from the dataset. It uses a random subset of features at each split to construct a decision tree for each sample. Next, a different bootstrap sample is used for training each decision tree. Finally, the algorithm averages the predictions made by each tree to obtain the final result for regression.

  Key parameters used to define the model are: the maximum depth of each tree, the minimum number of samples needed to split an internal node, the minimum number of samples needed to be at a leaf node, and the number of trees in the forest defined by $n\_estimators$. Additionally, the *bootstrap* parameter determines whether bootstrap samples are used when building trees. If set to True, each tree is trained on a random subset of the data with replacement.

  RF has several benefits. By averaging multiple decision trees, it reduces overfitting and becomes more resistant to outliers. It works well with big datasets and is an effective solution for regression tasks.

Advanced Neural Networks

Deep learning uses powerful neural networks to understand and interpret complex data, with a focus on making accurate predictions on new, unseen examples. Training a deep learning model involves using many examples to teach it to recognize patterns and features. After training, you can observe how well the model performs by testing it on unseen data. Not only does the model need to learn from the samples it was trained on, but it also needs to comprehend the underlying concepts sufficiently to be able to forecast new data accurately [32]. This ability to generalize is key in deep learning.

In this thesis, we used two advanced neural networks, RNN and LSTM, which were found to be the most common in the relevant research, as will be detailed in Chapter 3.

- **RNN**

  RNNs are artificial neural networks designed to model sequential data by maintaining a hidden state that captures information from past inputs [33]. RNNs are helpful for applications such as time series prediction, audio recognition, and natural language processing (NLP) as they can handle sequences of any length [34]. When using RNN, the hidden state h_t at time step t is calculated based on the current input x_t and the prior hidden state h_{t-1} using a set of parameters that can be trained which are W_{hh} and W_{xh}, as well as biases b_h. The forward pass equation for an RNN is typically defined as follows:

  $$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

  The output y_t at each time step t is then computed based on the hidden state h_t using another set of parameters W_{hy} and b_y, and is typically passed through a softmax function for classification tasks.

$$y_t = \text{softmax}(W_{hy}h_t + b_y)$$

Although RNNs are widely used models for sequential data, they are limited in their ability to capture long-term dependencies in sequences, an issue referred to as the vanishing gradient problem. To solve this problem, LSTM networks can be used in their place [33].

- **LSTM**

  LSTM networks are a type of RNN specifically designed to capture complex temporal connections in sequential data [35]. LSTMs are considered more robust than RNNs as they can remember and use information selectively over longer sequences than regular RNNs, since they have memory cells and gating mechanisms. They can effectively capture long-term dependencies in sequential data while reducing the problem of the vanishing gradient problem that is frequently found in conventional RNNs, as mentioned earlier. Because of this, LSTMs have become an essential component of sequential data modeling and are widely used in a range of fields where temporal correlations are important.

  The forget gate, input gate, and output gate are the three fundamental gates which makeup LSTMs. By choosing what to keep, what to reject, and what to output to next time steps, these gates control the flow of information. LSTMs constantly modify their internal state in accordance with the input observations.

  The forward pass equations for an LSTM are as follows:

  1. Forget Gate. Determines what information to discard from the cell state.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

2. Input Gate. Determines which values to update.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

3. Cell State Update. Updates the cell state C_t.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

4. Output Gate. Determines the output based on the cell state.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

2.4 Evaluation metrics

In evaluating the performance of ML models, choosing appropriate evaluation metrics is the key to understanding how well the models are performing and enabling effective comparison between different models. In our analysis, we employed several key evaluation metrics which enabled a better understanding of the performance of the models: mean absolute error (MAE), median absolute error (MedAE), R^2 (also known as the coefficient of determination), and accuracy.

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|$$

MAE is a fundamental statistic used to determine the average magnitude of errors between predicted $\hat{y}_i$ and actual values y_i. It first calculates the absolute difference between each predicted value and its matching true value and then computes the average of these absolute differences.

MedAE complements MAE by providing a robust measure of central tendency that is less sensitive to outliers. Instead of averaging all absolute errors as MAE does, MedAE computes the median of the absolute differences. Because it offers a more reliable estimate of error magnitude, this metric is especially helpful in situations when the dataset contains outliers or when the distribution of errors is skewed.

In a regression model, the coefficient of determination, or R^2, measures the percentage of the dependent variable's variation that can be predicted from, and therefore attributed to, the independent variables [36]. It provides insight into how well the model captures the variability in the data. The values of R^2 range from 0 to 1, where 0 indicates no linear relationship between the variables and 1 represents a perfect fit. We have seen in the literature that R^2 is a useful indicator for regression assignments and it is commonly applied to similar applications. As the formula below indicates, where $\bar{y}$ is the mean of the observed values y_i.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

Accuracy is a common evaluation metric for classification tasks, measuring the proportion of correctly classified instances out of the total number of instances. It provides a clear indication of the model's overall correctness in predicting class labels.

Selecting the right metrics is essential for assessing ML models' performance, for comparing models and for understanding how good the predictions are. The evaluation metrics we chose to use in our analysis, as described above, enabled us to acquire a deeper understanding of the model's performance. By employing a combination of these evaluation metrics, we gained a comprehensive understanding of the strengths and weaknesses of our ML models. Together, these metrics enable us to make informed decisions about model selection and optimization to achieve the desired performance outcomes.

3 Related Work

This chapter delves into the current research related to the methodology of this thesis. In Section 3.1, we explore various methodologies and technologies aimed at understanding and predicting sleep patterns. Previous research adopted diverse approaches, including ML-based sleep analysis, individualized sleep scheduling, and wearable sensor data analysis, to improve understanding of the complex dynamics of sleep behaviors.

In Section 3.2, we examine similar approaches applied in other domains, specifically focusing on adherence prediction in user-based datasets. By drawing parallels with fitness and menstrual cycle tracking applications, we explore the utilization of ML methodologies to forecast adherence to health and fitness routines. These studies highlight innovative approaches to address data limitations and improve prediction accuracy, offering valuable information that has been leveraged in the context of sleep pattern prediction.

Collectively, these sections present a thorough summary of the literature, laying the groundwork for the technique used in this thesis and providing suggestions for other study directions.

3.1 Sleep pattern understanding and prediction

Previous research has taken different approaches aimed at better understanding and predicting of sleep patterns, employing a diverse array of methodologies and technologies.

3.1.1 Machine learning-based sleep analysis

ML-based sleep analysis has, in recent years, become a highly studied field, with research showing the potential of extracting valuable insights from daily logs [37]. One study presented a computational framework for forecasting sleep efficiency in individuals with insomnia, utilizing data from smart bands capturing sleep records and daily activities [37]. To handle missing data, the framework used in that study employs Improved Generative Adversarial Imputation Networks (Imp-GAIN), including an interpretable LSTM-Attention (LA Block) neural network model for sleep efficiency prediction. The dataset is a time series on daily sleep and behavioral records. Complementing this, [38] introduced Bedtime Prediction (BTP), a groundbreaking bedtime predicting algorithm. BTP utilizes smartphone screen status data, providing a novel and data-driven approach to anticipate sleep onset.

Advanced ML techniques have further propelled sleep prediction capabilities. [39] developed a sophisticated sleep prediction algorithm capable of forecasting sleep-/wake states and estimating sleep parameters accurately. By integrating ML technology, their approach offers a comprehensive tool for understanding sleep behaviors and identifying potential disturbances. Furthermore, [40] leveraged convolutional neural networks to predict sleep quality from sensor data with remarkable precision. This advanced methodology not only enhances our understanding of sleep patterns but also opens new avenues for personalized sleep interventions. Additionally, [41] utilized ML techniques to predict sleep behavior in smart homes, shedding light on environmental factors that impact sleep quality. Through the integration of ambient sensors and IoT devices, their study offers insights into creating conducive sleep environments tailored to individual needs. The study's objective was to discern residents' potential sleep behavior in relation to other activities and their health implications [41]. The findings revealed that SVM outperformed other classifiers. Consequently, the authors proposed that their research facilitates early detection of diseases associated with sleep disorders and recommended further enhancements for future studies [41].

Moreover, personal projects such as [42] show the tangible impact of data-driven approaches in addressing real-world sleep challenges. Through the application of ML and data science, parents gain actionable strategies to manage sleep-related issues, underscoring the transformative potential of personalized interventions.

3.1.2 Wearable technology

Advances in wearable technology and sensor data analysis have significantly contributed to the understanding of sleep patterns. One study [43] used wearable sensor data to predict changes in sleep quality. Similarly, [44] explored the prediction of sleep efficiency from wearable device data. Their findings informed the development of sleep monitoring features but also deepened our understanding of the relationship between wearable device metrics and sleep quality [43], [44]. They employed a diverse array of ML models, including Multilayer Perceptron (MLP), Long Short-Term Memory (LSTM), Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), Random Forest (RF), Support Vector Machine (SVM), and Logistic Regression (LR), to assess their effectiveness in analyzing sleep data. Evaluation metrics such as accuracy, receiver operating characteristic (ROC) curve, precision, recall, and F1-score were utilized to evaluate the performance of these models [43], [44]. Upon evaluation, the LSTM model emerged as the top performer, demonstrating superior predictive capabilities. Following closely behind was the CNN. Additionally, Random Forest and SVM performed the best among non-neural network models, showcasing notable performance in sleep data analysis [43], [44].

Additionally, [45] demonstrated the efficacy of deep learning techniques in analyzing sensor data to predict sleep quality accurately. The researchers utilized data collected from wearables equipped with accelerometers, enabling continuous monitoring of physical activity and sleep patterns. The study's objective was to predict sleep efficiency using deep learning approaches based on physical activity data.

3.1.3 Individualized sleep scheduling

The work of [46] underscores the importance of individualized approaches to sleep scheduling, particularly in the context of college students. Their study considered various factors such as class schedules and personal preferences. They employed Electroencephalography (EEG) data to gain insights into sleeping patterns and subsequently utilized SVM to forecast the optimal sleeping times for young users [46]. While considering CNN for analysis, the study found that SVM resulted in a slightly superior performance, with K-Nearest Neighbors (KNN) also being explored as an alternative method which, in this particular case, had worse efficacy than SVM [46]. Other research [47] found that increased physical activity and appropriate exposure to natural light during the day were significantly associated with improved sleep quality, also showing that XGBoost models had better performance than deep learning models such as LSTM.

Real-world data analysis and personal experiences offer invaluable insights into sleep behaviors. [48] leveraged user data from a mobile application to gain insights into infant and toddler sleep patterns. By analyzing large-scale datasets, their study provided practical guidance for parents navigating the complexities of early childhood sleep. The study offered detailed visualizations illustrating the development of sleep patterns in infants over the first three years of life. Key observations included variations in daytime sleep session length (naps) with age, a consistent morning wake time among children aged 5–36 months, and evolving sleep patterns characterized by individual variability. Sleep patterns started to consolidate around 5–6 months, with longer nighttime sleep duration and increasing consistency in daytime sleep [48]. Notably, bedtime variability had a greater impact on nighttime sleep duration compared to wake times, highlighting the importance of establishing consistent bedtime routines for promoting healthy sleep habits in infants and young children.

3.2 Data adherence and generative models

Given the nature of our dataset, being collected from logs registered into the app by users, it is relevant to take into account the importance of user adherence when using applications. Other application fields were found which had researched this issue and addressed, for example, adherence to mobile applications for fitness or menstrual cycle tracking.

ML methodologies have revolutionized the prediction of menstrual cycle characteristics and physical exercise adherence, offering innovative solutions to address data limitations and enhance prediction accuracy across various studies.

Recent studies have highlighted various approaches to modeling physiological patterns. For instance, [49] explored the modeling of menstrual cycle length across the reproductive lifespan, emphasizing the significance of within-woman variance. Their analysis of menstrual diary data provided valuable insights into the cyclical variations experienced by women and underscored the importance of accounting for individual differences. The authors proposed a generative modeling framework based on Gaussian processes and Bayesian neural networks, showing how incorporating uncertainty estimation into predictive models can result in more accurate and reliable predictions. On a related note, [50] delved into understanding menstrual cycle dynamics among athletes, leveraging state-space models to capture menstrual cycle length variations in physically active individuals. By examining menstrual health data in the context of athletic performance, they shed light on the interplay between exercise intensity and cycle regularity, offering tailored insights for this specific population.

Collaborating, [51] and [52], addressed the challenge of self-tracking artifacts in menstrual health data through generative modeling approaches. They proposed a novel method for calculating calibrated predictions, effectively mitigating biases and inconsistencies inherent in self-reported data to improve prediction accuracy. Their approach provided a robust framework for handling uncertainties in menstrual cycle prediction [51], [52]. The papers presented a hierarchical model for predicting menstrual cycle lengths while addressing potential tracking artifacts. By leveraging individual-specific patterns and population-wide characteristics, the model offers interpretable insights into menstrual behavior. Its hierarchical framework accommodates individual-level variability and population behaviors, ensuring scalability and adaptability to new users without the extensive need to retrain the model [51], [52]. Similarly, the authors from [53] proposed a model addressing user adherence variations in menstrual tracking apps, ensuring accurate cycle length predictions. Their approach, complemented by RNN, demonstrates robustness and reproducibility. Handling missing data and integrating user-specific and population-wide trends are key aspects. Their generative model framework enhances interpretability and prediction accuracy.

Finally, [54] proposed a sequential prediction method for menstrual cycle lengths, leveraging historical data to forecast future cycle characteristics. The authors employed a Bayesian hierarchical dynamic model to understand individual variability among women. This framework enables the transfer of information across subjects, compensating for limited individual data. By employing both individual and population models within this hierarchical structure, the approach facilitates robust predictions and information sharing across users.

Similar techniques have been used in physical exercise adherence prediction, including [55] who leveraged deep learning methodologies to forecast exercise adherence in fitness apps. Their approach harnessed the power of deep neural networks to extract intricate patterns from limited datasets, enabling more accurate predictions of individuals' adherence to exercise routines.

In summary, these studies together highlight diverse applications of ML techniques in addressing data limitations and improving prediction accuracy in user-provided data. Through innovative methodologies and different algorithms, researchers have advanced predictive modeling techniques, contributing to the refinement of health-related applications and leading to more personalized interventions.

4 Dataset

In this chapter, we introduce the dataset used in this thesis to train and test the different models. We also present the exploration of our data collection and cleaning processes, both fundamental to laying the groundwork for this thesis. In Section 4.1 we detail the procedures employed to collect the dataset, and the structure and format of the data.

Following this, in Section 4.2 we describe the refinement of the existing dataset to ensure suitability for model training. Here, we outline the steps undertaken to normalize timestamps between time zones and remove anomalies, all aimed at strengthening the dataset's robustness and relevance to our research objectives. This lays a foundation for the subsequent chapters, where we dive deeper into its analysis and interpretation.

4.1 Data collection

The data used was collected from the Napper app, where users can track babies' activities such as sleeping, feeding, and nursing. This app, as mentioned previously in Section 1.3, started in 2020 and therefore data exists from January 2020 onwards. In this case, the last test was done on data up to and including May 2024.

Throughout the last years, the data were collected in different forms and different parameters were tracked by the user. It was therefore decided to focus on the data obtained from 1st April 2022 onwards. Moreover, given that this project focuses on

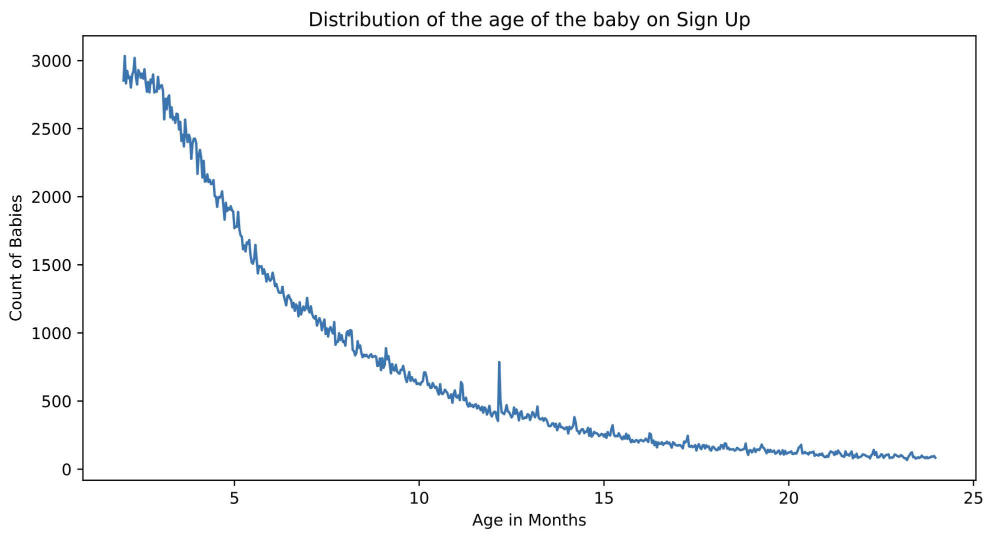


Figure 4.1: Distribution of babies' age on sign up

the sleep schedule, only those logs related to the wake-up time, the naps during the day, and the bedtime were used.

Data from around 10,000 babies were used for this project. This was the total number of babies who had sufficient data entered into the app for the prediction to be possible. This allowed us to have enough data to train our models and to get a realistic view of babies' sleeping patterns.

The Napper app focuses on babies from 2 to 24 months old, hence the data collected were mainly within this age range. Figure 4.1 includes some statistics on babies' ages when registered to the app. It was also found that the adherence to the app and tracking of the sleeping times lessened after 24 months.

It is considered that after two months of age the sleeping patterns of a baby should be more stabilized, being difficult in the first two months due to the lack of a developed circadian rhythm, as detailed in Section 2.1. As a result of this, there were not current predictions for younger babies. Hence, from our dataset, the data from babies younger than 2 months, and babies older than 24 months, were removed.

In order to reduce outliers and select the logs that would provide the most accurate representation of the sleep pattern, only babies within the top 6th percentile of logs were taken. Following these reductions, a total of 13.5 million logs made up the dataset that was used to obtain the results seen further on in this thesis.

In addition to sleeping and napping times, users of the app are also able to track when the baby is fed (and some additional details e.g., breastfed, pumped milk, formula milk, solids), if they have had a fever or taken any medicine, and if they woke up during the night. Even though not all of these features were used for the study, they were analyzed and research was done to understand the relationship they could have with the sleeping patterns of babies.

Prior to the final dataset, the quality of the data and consistency were thoroughly evaluated, particularly due to the nature of the user-provided data and the accompanying adherence issues. Aside from the sleep logs, a dataset with baby metadata was used, containing their time zones and dates of birth. As a first step, the personal information from each baby was merged into each of the logs corresponding to it.

4.2 Data cleaning

Given that the dataset was based on self-reported information from app users, careful attention was placed on removing outliers and values believed to have been wrongly registered in the app. This could have been values that were not representative of a baby's sleep, such as a reported duration of 24 hours of continuous sleep, or those that were not possible to achieve, such as naps registered before the morning waking up time.

As a first step, the timestamps were normalized, so that all the times in the dataset were adjusted to the users' local time. All users were synced to the same timezone, to reduce the difference as much as possible in order to train the model regardless of the timezone. The time location offset was stored to later fix back these times to the users' timezone.

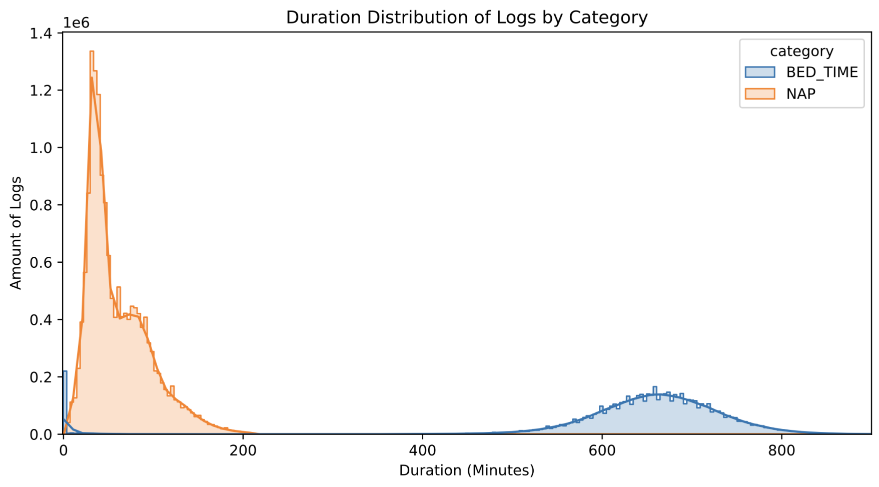


Figure 4.2: Distribution of duration of naps and sleep at night

It was also considered to remove the whole day for babies who had any of the following conditions present on that date: when the first registered waking up time was later than 14.00 or earlier than 04.00, or when the final night bedtime was logged before 17:00. To have a good baseline for a daily schedule, days when the times of waking up or the time of bedtime were not logged were also removed. As for the number of naps, any day that had naps longer than 3.5 hours / 210 minutes was deleted, which can be seen on Figure 4.2, where no naps with more than 210 minutes are present. This was decided after observing that the percentage of these cases was low and thus they were considered outliers.

As will be seen later in Chapter 5, one of the variables predicted was the number of naps for a baby on the next day. There is a tendency to reduce the number of naps as the baby gets older and therefore some limits were added to choose those babies that will be used to train the models. If the number of naps per day diverged by more than 2.5 with respect to the mean at that age, those days were also removed, as can be seen after processing on Figure 4.3.

The Napper app has a learning phase of 7 days for adapting predictions to each baby's sleeping pattern. It is after this time that users can expect fully individualized

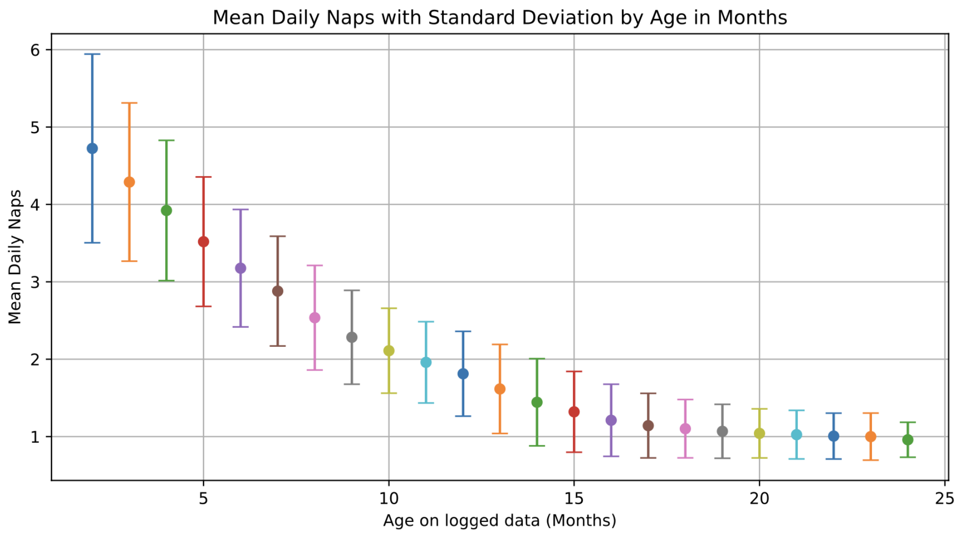


Figure 4.3: Distribution of number of naps per day by age

sleep schedules. In line with this, we limited the scope of the thesis to only making predictions for babies who had at least 7 consecutive days tracked. Data points from babies with fewer consecutive days tracked were thus filtered out.

5 Proposed Method

In this chapter, we introduce our model architecture. We start by detailing and justifying the decision to adopt two distinct models in Section 5.1, each tailored to predict different parameters essential for the babies' sleep patterns. Section 5.2 subsequently delves into the process of feature extraction and selection, where we explain the methodologies employed to refine our dataset and enhance its suitability for model training. Feature scaling steps are also included, giving light to the significance of prepossessing techniques in optimizing model performance and stability.

Following this, Section 5.3 goes into detail on the model training, delineating the steps involved in the train-test split and reasoning behind the selection of models utilized in our study. Through these sequential discussions, we aim to provide a comprehensive understanding of our proposed model architecture, setting the stage for the subsequent results and findings presented in this thesis.

5.1 Proposed model architecture

Figure 5.1 shows the steps and overall architecture of the project. After preprocessing the initial data, we dived into the feature engineering and splitting the dataset into training and testing. Finally, we trained our ML models and evaluated the results.

When it comes to the baby's sleep schedule, several variables define it that we wanted to predict. As detailed in Section 2.1, the babies in the dataset slept several

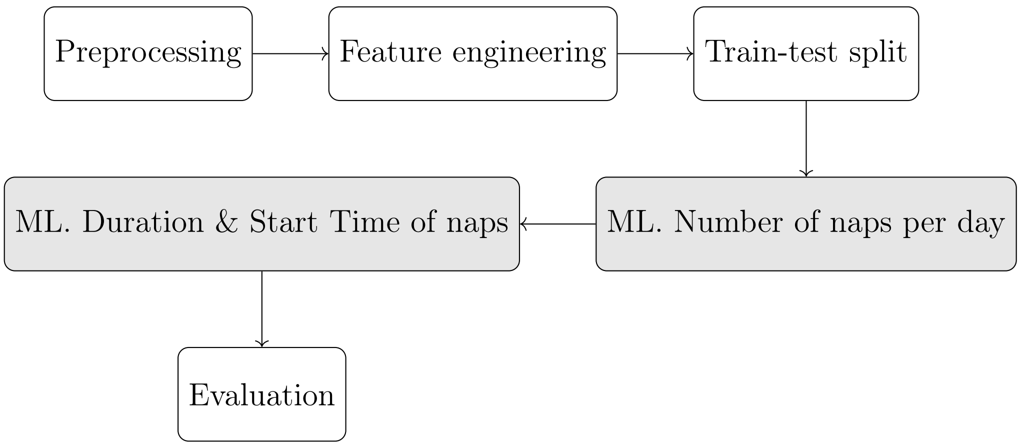


Figure 5.1: High-level architecture of the methodology

times a day, ranging from 0-5 times a day, which decreased as they got older. The schedule prediction was modeled as a two-step process. A key aspect of the task was to determine the structure of the input data and output predictions.

Our approach involved two distinct models tailored to different aspects of the sleep prediction task. The first model focused exclusively on predicting the number of naps that a baby will have in a day. This task was key as it formed the foundation for subsequent predictions and provided a baseline for daily sleep patterns. The second model was designed to predict the duration and starting times of these naps. By separating these tasks, we were able to specialize each model to handle the unique characteristics and challenges of predicting counts versus continuous time variables.

As detailed previously in Section 4.1, we were working with three categories of logs related to the sleep, those corresponding to the wake-up times, those referring to the naps and finally the ones determining the bedtime at the end of the day. The categories were encoded so that we could work with them. An example of how the final data were then organized can be seen in Table 5.1 where, for each log, there is information on the number of naps that day, the number of the specific nap, and the starting time and duration of each log. The first predicted variable was the number of naps per day, and that was fed into the next model to come up with the next two variables, the starting time and duration. The duration only had a meaningful value for the nap logs, since for the wake-up and bedtime it was set to 0, given that the nap start time was the one we wanted to predict.

5.2 Feature engineering

The feature extraction and selection were main parts of the project. By adding rolling statistics as features we were able to add previous data that reflected the time component.

In this section, the different features incorporated and statistics extracted are detailed in Section 5.2.1 and Section 5.2.2. Later on, in Section 5.2.3, the features finally used to train our model are further detailed. Finally, we describe the scaling applied to the features in Section 5.2.4.

5.2.1 Quality checked dataset

Once the data cleaning and first steps of processing were done, we had a quality-checked dataset to start working with. These processes and the reasoning behind the decisions taken are described in Section 4.2.

To add more information, some additional fields that were not in the original dataset were added. For each log, the age at the time of logging was added as a way to compute a timeline for the baby's age and how its sleeping schedule should be. This was computed in months with one decimal point. Therefore, all babies in the dataset have an age on log ranging from 2.0 to 24.0, as shown in Table 5.1.

For each nap entry, the duration was calculated between the starting time and ending time of the recorded log. For the wake-up and bedtime logs the duration was set to 0 given that, with the starting time, it was enough to come up with the full sleeping schedule.

babyid	age	date	naps_day	categ.	num_nap	start	duration
0	12.2	2023-07-11	2	2	0	07:42:33	0.0
0	12.2	2023-07-11	2	1	1	11:17:41	47.64
0	12.2	2023-07-11	2	1	2	15:50:59	46.03
0	12.2	2023-07-11	2	0	3	20:37:09	0.0
1	13.1	2023-07-11	1	2	0	08:35:40	0.0
1	13.1	2023-07-11	1	1	1	12:56:19	42.35
1	13.1	2023-07-11	1	0	2	20:04:00	0.0

Table 5.1: Final data format

For each day and baby, the number of naps on that day was counted and added as another field to the log, so that each row had this information. Also, the number of that nap was recorded, so that for each log, there was the number of naps on that day and the number of the nap for that log. This was useful later to compute the features and to group the naps within each other depending on what number they were on the day, in order to add some statistical features.

The time at which each log started or ended was modified to be in seconds epoch and on a 24-hour cycle, to have an easier interpretation when looking at the results. Both appeared to perform equally well when working with them so, for better understanding, the 24-hour cycle was used for the model.

Two additional fields were added to remove outliers. One of them was the previous wake-up time, referring to the time awake passed since the last sleep. The other was the accumulated wake time during the day, adding up all the time that day that the baby had been awake. Both of these were set to 0 when their value was more than 20 hours, considered then to have skipped days of tracking and assuming the baby had slept overnight.

5.2.2 Feature extraction

As will be explained in Section 5.3, the features were computed separately for each of the three parameters we wanted to predict: the number of naps the next day, and the duration and start time for each nap. Since two separate models were trained, one for the number of naps, and another for the duration and start time of each of the naps, two different sets of features were obtained.

Given that for each log registered we only used the data recorded that we wanted to predict, new statistical features were generated and added to the data. Therefore adding to each log some previous representation of the data. Rolling statistics were used in order to add statistical dependencies based on the previous data.

For these statistics, three were considered, the mean, the standard deviation (std), and the root mean square. For each of them, a window size of previous samples was generated to compute their statistics to add to each of the new data points, shifting one so that it would not include the current one, to avoid leaking testing data into the model.

In order to get the rolling statistics, we had to first group the data into different sets to only extract the data relevant to each case. For example, if we were trying to get the starting time of a third nap on a day when 4 were recorded, ideally we would only want the mean to be based on previous third naps on days with 4 naps recorded, rather than all the previous naps. This was very important given that the starting times varied quite drastically.

For the number of naps per day, the statistics were first calculated on all the available data of babies of the same age in months. Each new log therefore had the mean and std of the previous number of naps of babies that were the same age, and represented all babies of that age. Secondly, another set of features was added by grouping the logs based on the baby identifier so that, for each log, a new field with the rolling statistics of the same baby was added. For this second grouping, a smaller window size was added to represent shorter patterns.

For the duration and start time of each of the naps, two groupings were also done, using two separate window sizes to better represent them. Both groupings were done with the number of naps that day and the number of the current nap, so that the statistics were counted on the most similar naps, as detailed before. The first grouping was done with all the previous data available, resulting in quite broad statistics. The second grouping also added the baby id to those two parameters, so that there was a personalized statistic on the latest trends for the specific baby and, for this, a smaller window size was used.

5.2.3 Feature selection

The selection of features has been shown to be a key step to the later performance of ML models. Some features, from the ones previously detailed, were kept for the training processes of both models. This process was done in order to remove the features that could cause loss of information, and that have very little to no correlation with the values that were to be predicted.

One of the main advantages of reducing the number of features in the model is that this reduces the dimensionality of the data, which therefore eases the training speed and makes it easier to interpret. Of the previously detailed features, the ones used to train each of the two models were the following:

1. **Number of naps that day**

    - Identifier for each baby
    - Mean number of naps for babies at that age
    - Standard deviation of the number of naps for babies at that age
    - Rolling mean number of naps grouped on the identifier of each baby, with a window size corresponding to a week
    - Rolling standard deviation grouped of the number of naps on the identifier of each baby, with a window size corresponding to a week

2. **Duration and start time**

- Identifier for each baby
- Category of the log, can correspond to the wake-up times, the naps, or the bedtimes.
- Number of each nap

Duration, for each nap

- Mean duration on all available data within babies of the same age for that nap
- Standard deviation duration on all available data within babies of the same age for that nap
- Rolling mean duration grouped on the number of naps and the number of naps on that day and the baby identifier. Using a window size representing a week
- Rolling standard deviation duration grouped on the number of naps and the number of naps on that day and the baby identifier. Using a window size representing a week

Start time, for each nap

- Mean start time on all available data within babies of the same age for that nap
- Standard deviation start time on all available data within babies of the same age for that nap
- Rolling mean start time grouped on the number of naps and the number of naps on that day and the baby identifier. Using a window size representing a week

- Rolling standard deviation start time grouped on the number of naps and the number of naps on that day and the baby identifier. Using a window size representing a week

5.2.4 Feature scaling

Once the features were selected for each of the models, they were scaled before being fed into any of the ML models. Both z-score standardization and min-max normalization, detailed in Section 2.2, were used. For this we used the StandardScaler and MinMaxScaler from the scikit learn package [56].

XGBoost is generally quite robust to feature scaling due to its nature of handling tree-based algorithms, so it does not rely on the distance metric. Using StandardScaler, which scales the features to have zero mean and unit variance, can sometimes improve performance, especially when the features have very different scales [57]. We saw this improvement in our case.

For both RNN and LSTM, the features were scaled using MinMaxScaler, given that they often perform better when the input features are scaled to a specific range [58]. MinMaxScaler helps in preventing the gradients from vanishing during training by keeping the input features within a consistent range, aiding in the stability of the training process.

5.3 Model training

In this section, we detail the process of training our ML models. This involved splitting it into training and test sets and then applying various models to predict sleep patterns. The goal was to develop models that can accurately predict future sleep behavior based on historical data.

The sleep schedule, as mentioned earlier, was described with three variables which, when combined, returned a full-day schedule: the number of naps each day, the duration of the naps, and the start times of the naps. Given that the last two variables depended on the first variable, two models were used.

5.3.1 Train - test split

Given the sequential nature of the dataset, the train and test split was done so that the last full day of recorded data for each baby was saved for testing, while all previous data were used for training.

This method ensured that the models were evaluated on data they had not seen during training. This approach is particularly effective on time series data, where the temporal sequence of observations is necessary for prediction accuracy. By using the most recent data for testing, we mimicked a real-world scenario where the model was applied to predict future sleep patterns based on past observations.

The first step was to predict the number of naps and, for that, only the data logs corresponding to wake-up times were used, as a reference to each day. The reason behind not using the whole dataset for the first model was to avoid data leaking; it was harder to predict the number of naps per day seeing as the days with more naps had a bigger representation of data than those with less. It also resulted in a bigger representation of the statistical features. Before training the first model, the split among dates was done, finding the latest day stored for each baby and saving those as a series for the split. After training the first model, the outcome of it was fed to the second model to generate the exact schedule for the day and baby.

The whole dataset was then split according to the saved matches between each baby and their latest recorded date, and the two datasets became one for training and the other for testing.

5.3.2 Used models

The regression algorithms presented in Section 2.3.1 were trained using the set of parameters and set-up described below. The algorithms were implemented using both scikit-learn [56] and Tensorflow [59].

The following parameters are the ones that were tested and which showed the best results. For each of the models, some of the parameters were modified until they reached the best value.

KNN

The number of neighbors in our k-nearest neighbor regressor was set up to 3. The weight parameter value was set to uniform, meaning that each neighbor point affected the decision with equal weight. The distance metric was equal to the standard Euclidean metric.

XGBoost

The XGBoost model was configured with the following parameters. The number of boosting rounds, specified by num_estimators, was set to 100. The learning rate was 0.1, which controls the step size at each iteration while moving toward a minimum of the loss function. The objective parameter was the squared error, specifying the learning task and the corresponding objective function to be minimized. It was seen to perform better than when the objective was set to the absolute error. To ensure the reproducibility of results, the random_state was set to 42.

Random Forest

The maximum depth of each decision tree (max_depth) parameter was set to 10, which controls the maximum depth of each decision tree in the forest. This helps balance the trade-off between achieving a low error on the training data and minimizing the risk of overfitting. The number of trees (n_estimators) parameter was set to 100, indicating that the forest is composed of 100 decision trees, which enhances the model's stability and accuracy. The bootstrap parameter was set to True, meaning that bootstrap samples are used when building the trees. This introduces randomness into the model, helping to improve robustness and prevent overfitting. Again, to ensure reproducibility of results, the random seed was set to 42.

LSTM

The Long Short-Term Memory model was configured with the following parameters. The number of epochs was set to 3, representing the number of times the learning algorithm will work through the entire training dataset. The batch size was 32, indicating the number of samples that will be propagated through the network at once. The loss function was the MSE between the predicted value and the ground truth.

The model architecture was quite basic and included an LSTM layer with 100 units and an input shape which matched the reshaped training data. A Dense layer was used, with 1 unit in the first model to predict the number of naps and 2 units for the duration and start time. Also, a ReLU activation function was added. The model was compiled using the Adam optimizer and trained on the reshaped training data with a validation split of 20%, without shuffling, to maintain the sequential traits of the data. Training was conducted with a verbosity level of 1 to monitor the training process.

RNN

The RNN model was similar to the LSTM. The number of epochs was also set to 3. The batch size was 32. The loss function was the MSE, which measures the average squared difference between the estimated values and the actual value. The architecture included a SimpleRNN layer with 100 units and an input shape which matched the reshaped training data. A Dense layer was used, with 1 unit in the first model to predict the number of naps and 2 units for the duration and start time, and a ReLU activation function was added. Again, it was compiled using the Adam optimizer and using a validation split of 20%, without shuffling.

6 Results and Discussion

In this chapter, we present the results obtained by the trained ML models. All of these were obtained using the dataset detailed in Section 5.2.1, after all processing and feature engineering. We used the models which were trained and tested as described previously in Section 5.3.1.

For the first model, to predict the number of naps, the evaluation process involved comparing the predicted number of naps against the actual recorded number for each day in the test set. The performance was assessed using metrics including Mean Absolute Error (MAE), Median Absolute Error (MedAE), accuracy, and R-squared (R^2), as detailed in Section 2.4. The second model, for the duration and starting time, underwent a similar evaluation process but focused on continuous variables, hence accuracy was not used in its evaluation, but MAE, MedAE, and R2 were still used.

In Section 6.1, a series of visual analyses are presented to better understand the strengths and weaknesses of both models for each of the cases. Later on, in Section 6.2, a comparison between the different algorithms on each of the models is presented.

6.1 Performance evaluation

To thoroughly evaluate the performance of our predictions, a series of visual and quantitative analyses were performed. The primary metrics used for this evaluation included the plots comparing the actual vs. the predicted results. Secondary evaluation metrics included the residual plots. Both of these visual tools provided a clear representation of the model's accuracy and highlight areas where predictions may deviate from actual values.

The actual vs. predicted plots served a fundamental purpose in performance evaluation. By plotting the actual data points against the predictions made by our models, we could visually assess how closely the model's predictions matched the true values. Ideally, all data points should be exactly on a 45° line (on the red line in the actual vs. predicted graphs below). Deviations from this line indicate errors in the predictions, which can be systematically analyzed to understand the model's strengths and weaknesses.

Residuals can help identify patterns that may suggest model bias or variance issues. Defined as the difference between the actual and predicted values, they provide insights into the distribution and magnitude of prediction errors. The residual results should be scattered around the horizontal axis (the red line in the residual graphs below) for optimal predictions.

Other plots were developed to further understand the neural networks, such as the learning curves, which can be found in Appendix A.

6.1.1 Model 1. Number of naps per day

The first model predicted the number of naps per day. In Table 6.1 we analyze the performance of each of the algorithms upon the chosen evaluation metrics. LSTM and RNN both achieved the best performance, with an accuracy of 81.50% and a MAE of only 0.19 naps. The R^2 was 0.55 and was followed very closely by XGBoost and Random Forest. KNN performed the poorest, with the lowest accuracy of 77.79%, and the lowest R^2 of 0.43 compared to the other models. The R^2 metric, of around 50%, shows that the models only achieved a moderate fit, suggesting that they explained around half of the variability in the data.

	KNNReg	**XGBoost**	**RF**	**LSTM**	**RNN**
MAE	0.23	0.20	0.20	0.19	0.19
MedAE	0.00	0.00	0.00	0.00	0.00
Accuracy	77.79%	81.11%	81.09%	81.50%	81.50%
R^2	0.43	0.54	0.54	0.55	0.55

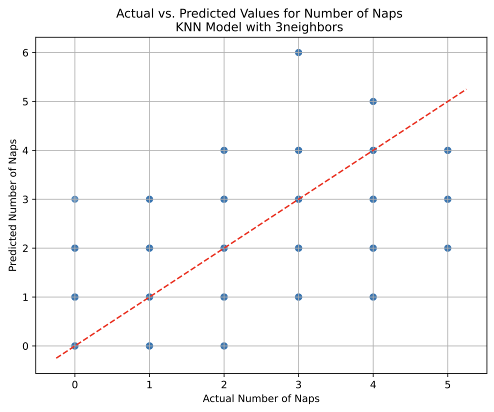
Table 6.1: Results on Number of naps per day

Some of the results for the algorithms used to predict the number of naps are far from the optimal red line in the figures below. KNN was the poorest model, shown by it being the algorithm which adapts least to the diagonal trend, as seen in Figure 6.1. This can indicate poorer predictions and also potential outliers or areas where the model does not perform as well.

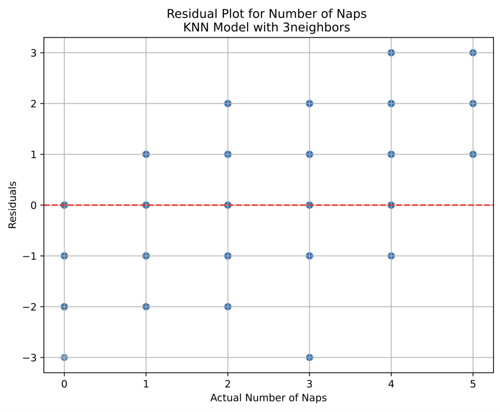
The actual vs. predicted plots below show how, in most of the algorithms, especially for KNN, XGBoost, and Random Forest, the models predicted a lower number of naps when the actual number was quite high. Similarly, it predicted higher numbers of naps when the actual values were quite low. For KNN, shown in Figure 6.1, the residual plot shows that the model failed at a higher scale at predicting the number of naps. When working with XGBoost, shown in Figure 6.2, and Random Forests, shown in Figure 6.3, the residuals were less and the models were more capable of identifying the patterns. Both neural networks, LSTM shown in Figure 6.4 and RNN shown in Figure 6.5, performed quite similarly.

All the models have in common that, when evaluating their performance, the number of the residual number of naps was quite high, suggesting that the variance of errors varied across predictions. Large residuals could be a sign of outliers, given the nature of the data. However, the 45° tendency in the actual vs. predicted plots shows that the model was able to capture the general trend.

KNN



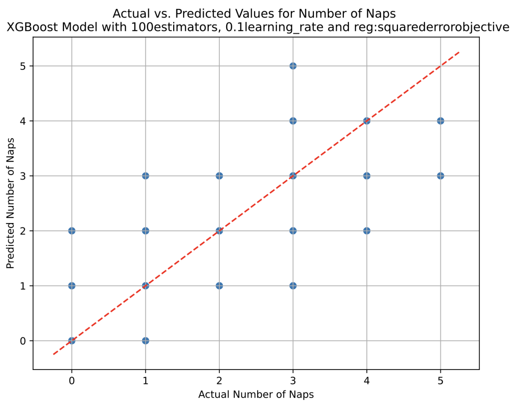
(a)Actual vs. Predicted



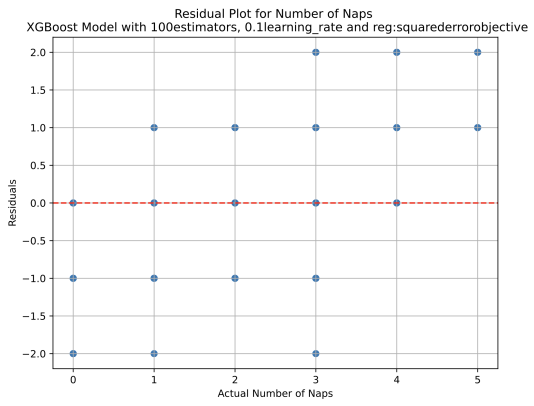
(b)Residual

Figure 6.1: KNN - number of naps

XGBoost



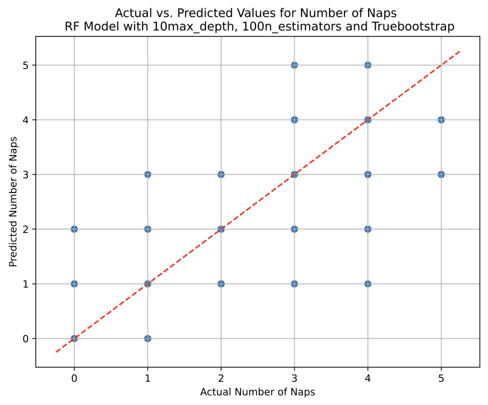
(a)Actual vs. Predicted



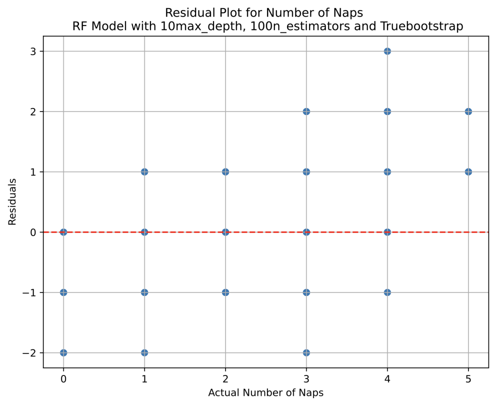
(b)Residual

Figure 6.2: XGBoost - number of naps

Random Forest Regressor



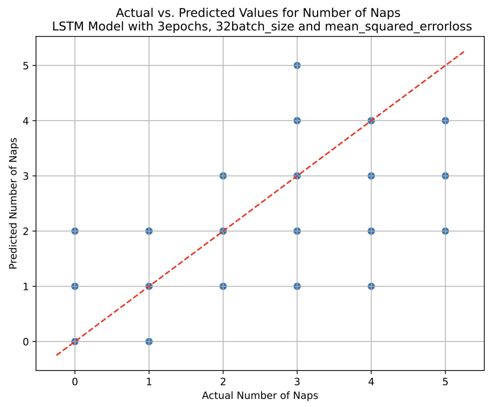
(a)Actual vs. Predicted



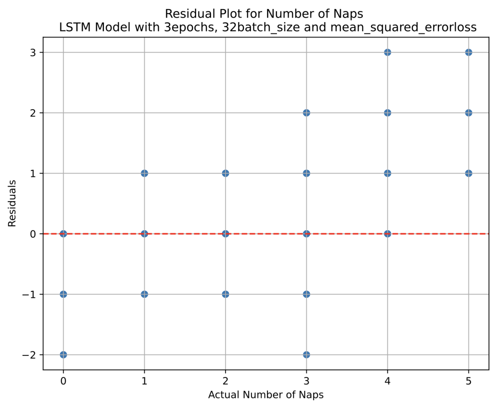
(b)Residual

Figure 6.3: Random Forest Regressor - number of naps

LSTM



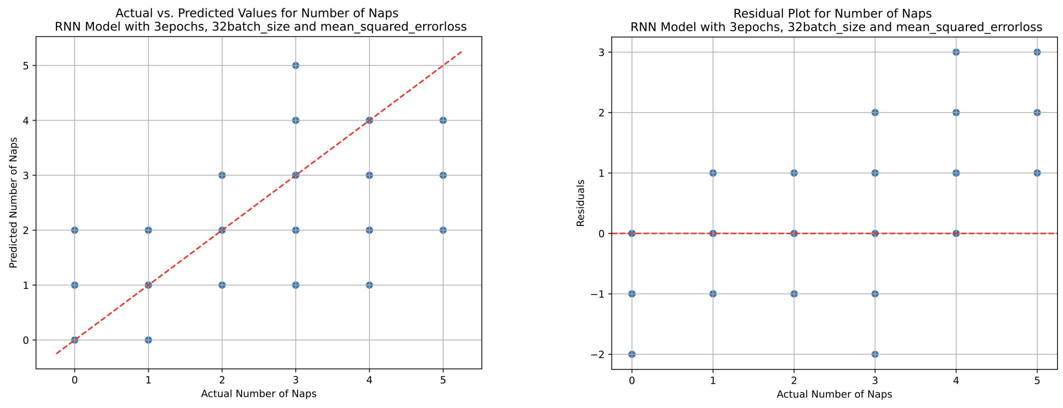
(a)Actual vs. Predicted



(b)Residual

Figure 6.4: LSTM - number of naps

RNN



(a)Actual vs. Predicted (b)Residual

Figure 6.5: RNN - number of naps

6.1.2 Model 2. Duration and Start time of naps

The second model predicted the duration and starting time for the different naps, waking-up times, and bedtimes. Table 6.2 shows the performance of each of the algorithms on the duration, and Table 6.3 on the starting time. Both of them were analyzed with the chosen evaluation metrics, as described at the beginning of Section 6.1.

Based on the results shown in Table 6.2, the best performing model was Random Forests with a 19.69 minutes MAE, a 15.29 minutes MedAE and a R^2 of 0.51. This was followed closely by XGBoost with a 19.71 MAE, a 15.31 MedAE, and an identical R^2 of 0.51. Similarly, as shown in Table 6.3, both of these models also performed the best when predicting the starting time. For this, XGBoost performed best with a 30.53 minutes MAE, 21.75 minutes MedAE, and a R^2 of 0.98. Random Forests was closely behind with a slightly higher MAE and MedAE of 33.85 minutes and 24.42 minutes, respectively, and the same R^2 score of 0.98. The models performed better at predicting the starting time than the duration across all algorithms. The results from the R^2 metric demonstrated this, with sleep starting time scores of around 90%, while only of 50% for the duration.

	KNNReg	**XGBoost**	**RF**	**LSTM**	**RNN**
MAE	23.01	19.71	19.69	19.95	20.35
MedAE	17.34	15.31	15.29	15.64	16.08
R^2	0.31	0.51	0.51	0.50	0.49

Table 6.2: Results on Duration (in minutes)

	KNNReg	**XGBoost**	**RF**	**LSTM**	**RNN**
MAE	39.96	30.53	33.85	44.02	61.63
MedAE	27.98	21.75	24.43	38.23	51.77
R^2	0.97	0.98	0.98	0.97	0.94

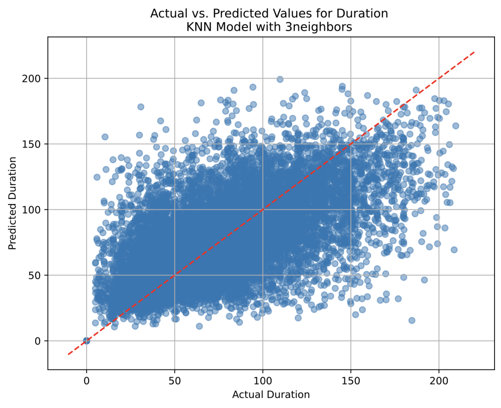
Table 6.3: Results on Starting time (in minutes)

The actual vs. predicted plots show that, for all models, the tendency to the 45° optimal line was achieved, especially when evaluating the starting times.

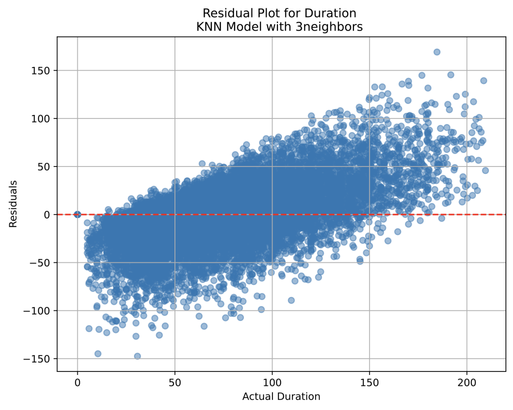
When looking into the residuals of the starting times, in most of the algorithms we saw a clear distinction between the wake-up and nap times with respect to the bedtimes. This can be seen clearly for RNN in Figure 6.10, and slightly less for LSTM in Figure 6.9, where the residuals were split in two when the time was later than 17.30hrs. A step further occurred with Random Forest, as shown in Figure 6.8, where the wake-up times and different naps could be vaguely distinguished.

As for the duration, for all algorithms the trend on the residuals was higher as the actual duration became larger. With KNN, as shown in Figure 6.6, this issue can be seen quite prominently, and the results on the actual vs. predicted values were also more sparse than with the other models.

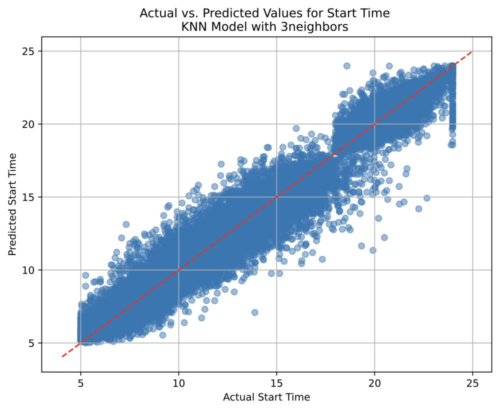
KNN



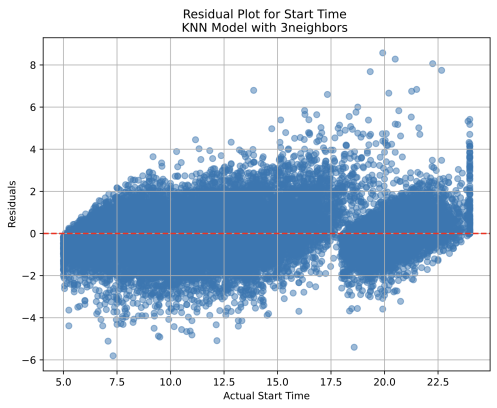
(a)Duration - Actual vs. Predicted



(b)Duration - Residual



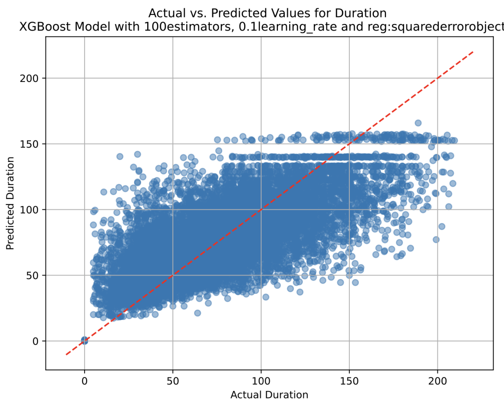
(c)Start time - Actual vs. Predicted



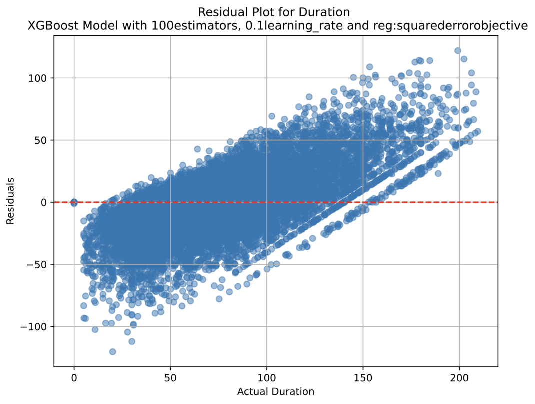
(d)Start time - Residual

Figure 6.6: KNN - duration & starting time

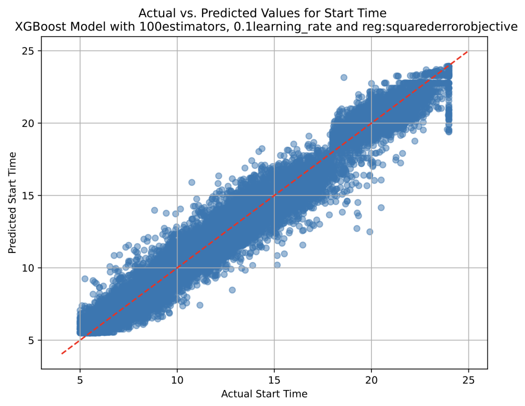
XGBoost



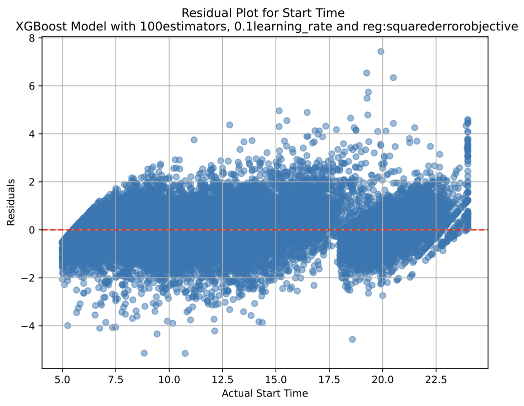
(a)Duration - Actual vs. Predicted



(b)Duration - Residual



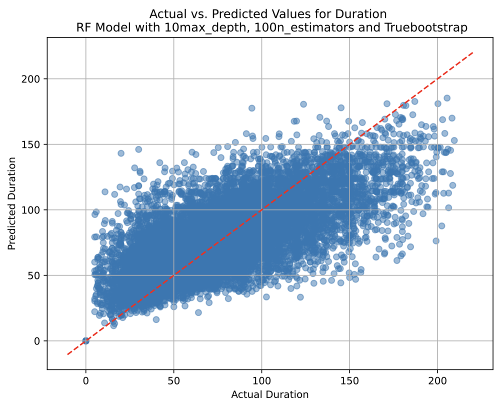
(c)Starting time - Actual vs. Predicted



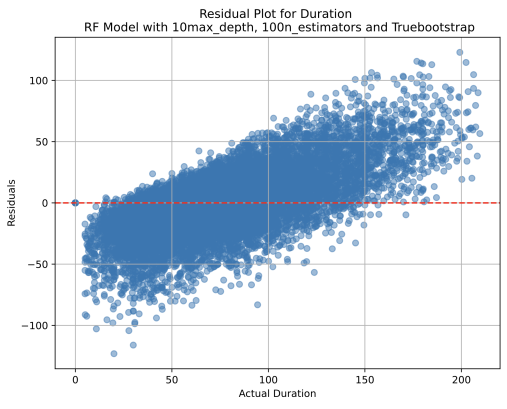
(d)Starting time - Residual

Figure 6.7: XGBoost - duration & starting time

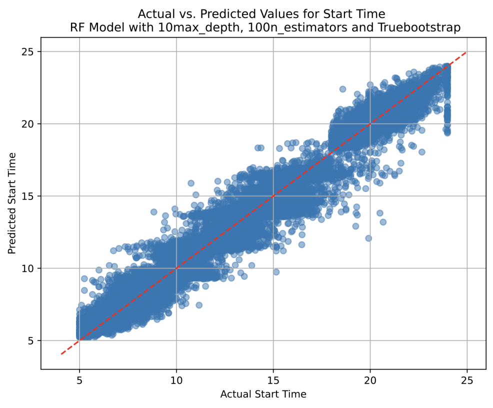
Random Forest Regressor



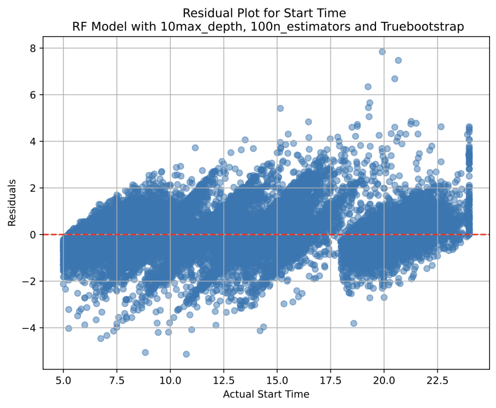
(a)Duration - Actual vs. Predicted



(b)Duration - Residual



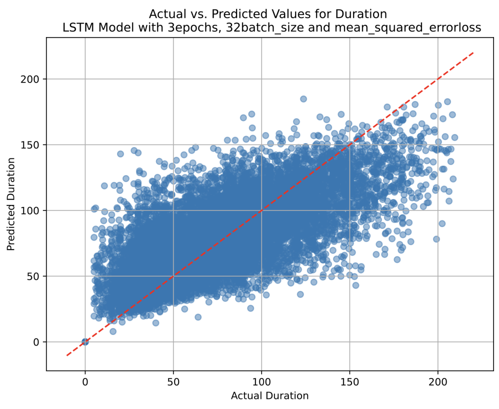
(c)Starting time - Actual vs. Predicted



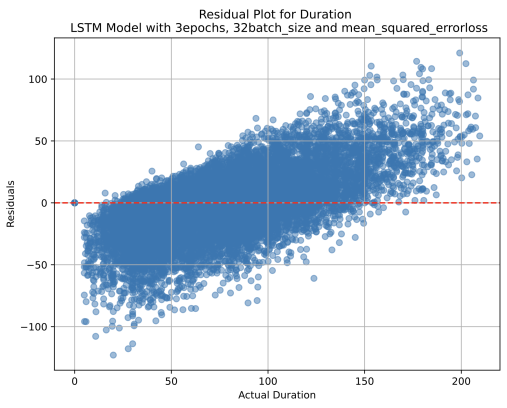
(d)Starting time - Residual

Figure 6.8: Random Forest Regressor - duration & starting time

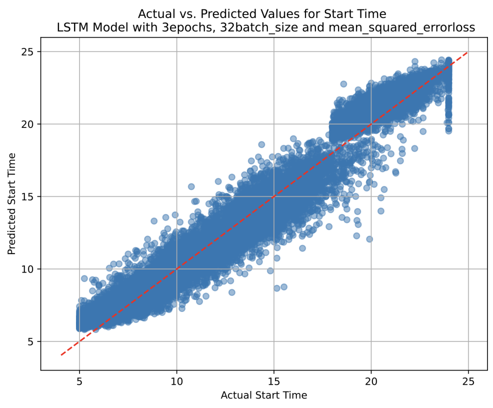
LSTM



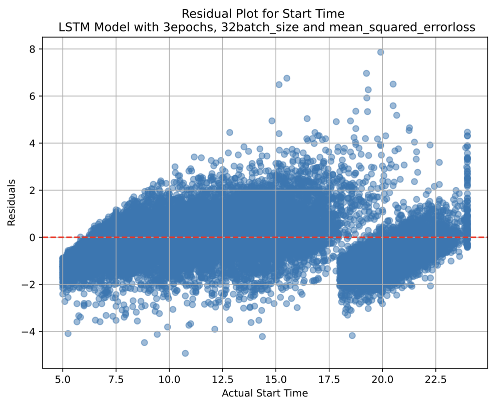
(a)Duration - Actual vs. Predicted



(b)Duration - Residual



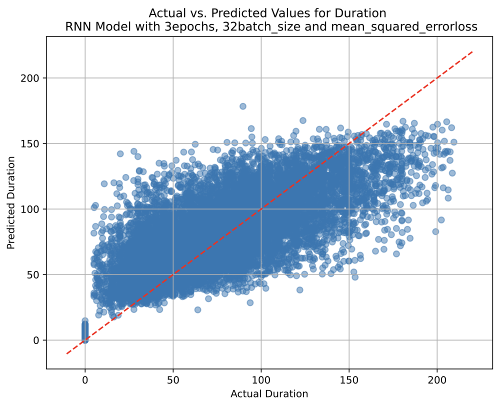
(c)Starting time - Actual vs. Predicted



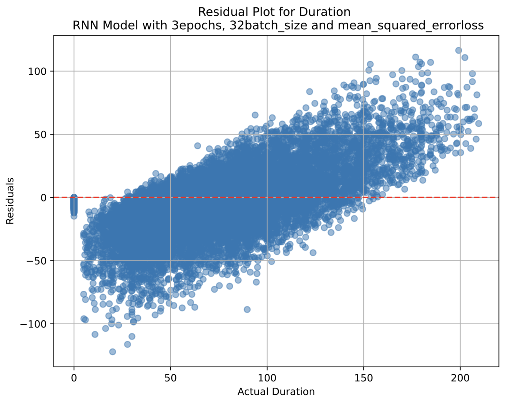
(d)Starting time - Residual

Figure 6.9: LSTM - duration & starting time

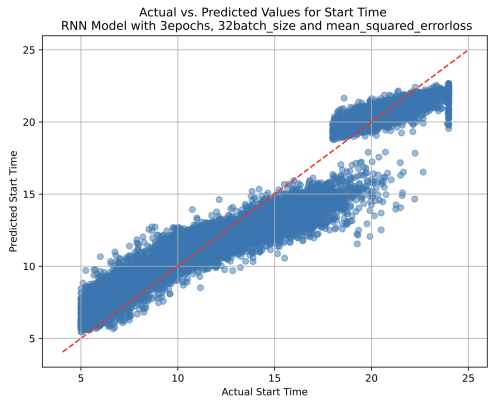
RNN



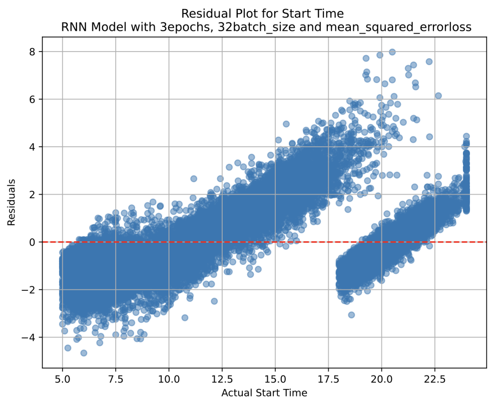
(a)Duration - Actual vs. Predicted



(b)Duration - Residual



(c)Starting time - Actual vs. Predicted



(d)Starting time - Residual

Figure 6.10: RNN - duration & starting time

6.2 Discussion

The results of our model evaluations, detailed in Table 6.4, revealed insightful performance metrics for each of the applied ML techniques. The evaluation focused on several aspects however, for comparison and discussion only, the following have been chosen: the Mean Absolute Error (MAE) of the second model for predicting nap start times and duration, and the accuracy of the first model for predicting the number of naps. These two metrics were considered representative of the error and accuracy of the models' desired predictions, prioritizing the minimization of their error. Both models were evaluated using the same metrics however, for predicting the number of naps we aimed to ensure that the accuracy was as high as possible, while for predicting the duration and starting time, the MAE was prioritized.

	KNNReg	**XGBoost**	**RF**	**LSTM**	**RNN**
Start-time MAE	39.96	30.53	33.85	44.02	61.63
Duration MAE	23.01	19.71	19.69	19.95	20.35
N. of naps Accuracy	77.79%	81.11%	81.09%	81.50%	81.50%

Table 6.4: Model comparisons

For predicting nap start times, XGBoost demonstrated the best performance with an MAE of 30.53 minutes, indicating a high level of precision in predicting when naps would begin, followed by KNN. For predicting the duration of naps, Random Forest and XGBoost again led the performance, with MAEs of 19.69 and 19.71, respectively. These results highlight that XGBoost is particularly adept at handling structured data related to sleep timings.

For predicting the number of naps, the accuracy metric showed that the LSTM and RNN models slightly outperformed the others, with an accuracy of 81.11% for both. KNN also performed well with an accuracy of 81.11%. These accuracy rates reflect the models' effectiveness in correctly predicting the number of naps a baby will take in a day.

Overall, the results indicate that while XGBoost excels in predicting the start times and duration of naps, LSTM and RNN models are better suited for accurately predicting the number of naps. This understanding of each model's strengths provides valuable guidance for selecting the appropriate techniques for different aspects of sleep prediction.

When comparing our results to the baseline from the app, which was obtained from the initial suggestions on the app compared to what users tracked, we saw that the difference was quite small, as shown in Table 6.5. In the case of the duration,

	Best performing	**Napper**
Start-time MAE	30.53	31.62
Duration MAE	19.71	25.81
Number of naps Accuracy	81.50%	93.25%

Table 6.5: Model comparisons with baseline from app
For the number of naps, LSTM was used, and for the duration and starting time XGBoost.

our best model exceeded the performance of the baseline and for the starting time, it was very close. We had to take into account that many of the results from users would have already been based on those suggestions, therefore the high accuracy on the number of naps predictions was likely subject to bias. The parents would have received a suggestion and they may have tried to follow it, for instance if the app suggested their child has 2 naps, they may have tried to adapt their schedule to that.

7 Conclusions

This thesis focused on predicting sleep schedules using a variety of machine learning models, with a particular focus on handling and processing the data. Through extensive analysis and experimentation, a range of models were tested, including K-Nearest Neighbors, XGBoost, Random Forests, Long Short-Term Memory Networks, and Recurrent Neural Networks, to predict babies' sleep patterns based on historical data.

The effectiveness of each model was assessed using a structured approach to data preparation and evaluation. The train-test split strategy, where the last day of data for each baby was reserved for testing, ensured a realistic evaluation of the models' predictive capabilities. This methodology provided a robust framework for understanding how well the models could generalize to unseen data.

In predicting the number of naps, LSTM and RNN emerged as the top performers, excelling in capturing the temporal dependencies inherent in sleep patterns. However, when it came to forecasting the duration and timing of the sleeping times, XGBoost and Random Forest notably outperformed the other models. This disparity in performance showcased the unique strengths of each model in addressing specific aspects of the sleep prediction task. While LSTM and RNN excelled in capturing sequential patterns and temporal dynamics, XGBoost's strength lied in handling structured data and delivering robust predictions. This highlights the importance of leveraging a diverse set of models to effectively address different aspects of complex prediction tasks, ultimately leading to more accurate and comprehensive results.

Despite the promising results, this study also highlighted several challenges, such as handling missing data, reducing the impact of outliers, and mitigating bias inherent in the data. The bias from the suggestions provided to the users before they registered data on the app posed a significant challenge to understanding the full precision of the results.

In summary, this thesis produced encouraging results in the domain of sleep prediction using machine learning models. By investigating a variety of models and addressing significant data challenges, the study achieved notable success.

7.1 Future work

Throughout this project, several ideas for future work were developed however, due to time constraints, they were not able to be explored.

One potential direction for future work would be to apply time series analysis techniques to tackle similar prediction problems. Considering the sequential nature of the data, this approach may offer improved predictive performance and better capture the dynamic behavior of sleep patterns over time. Another possibility for exploration is to reframe the task as a classification problem, where the goal is to classify whether a given period of time corresponds to a nap (sleep) or not. By treating sleep detection as a binary classification task, more classification algorithms and techniques could be applied to accurately differentiate between sleep and wake periods.

Improving the reduction of outliers and handling missing data is another avenue for further research. Leveraging machine learning techniques, such as the Improved Generative Adversarial Imputation Network proposed by [37], could help address challenges related to missing data and data consistency.

A further opportunity for future work lies in using the data fields described in Section 5.2.1: previous wake time, and accumulated wake time. These could offer valuable insights into sleep patterns and could be leveraged to adjust predictions in real time. While the current model predicted the next full day's sleep pattern, future research could explore strategies for continuously updating predictions based on real-time inference from these additional features.

References

[1] *Napper website*, https://napper.app/en/.

[2] P. S. McDowall, B. C. Galland, A. J. Campbell, and D. E. Elder, ”Parent knowledge of children’s sleep: A systematic review”, *Sleep medicine reviews*, vol. 31, pp. 39–47, 2017. DOI: 10.1016/j.smrv.2016.01.002.

[3] R. E. Hatton and M. Gardani, ”Maternal perceptions of advice on sleep in young children: How, what, and when?”, *British Journal of Health Psychology*, vol. 23, no. 2, pp. 476–495, 2018. DOI: 10.1111/bjhp.12300.

[4] J. A. Mindell and A. A. Williamson, ”Benefits of a bedtime routine in young children: Sleep, development, and beyond”, *Sleep medicine reviews*, vol. 40, pp. 93–108, 2018. DOI: 10.1016/j.smrv.2017.10.007.

[5] L. Matricciani, C. Paquet, B. Galland, M. Short, and T. Olds, ”Children’s sleep and health: A meta-review”, *Sleep medicine reviews*, vol. 46, pp. 136–150, 2019. DOI: 10.1016/j.smrv.2019.04.011.

[6] B. H. Fiese, T. Cai, C. Sutter, and K. K. Bost, ”Bedtimes, bedtime routines, and children’s sleep across the first 2 years of life”, *Sleep*, vol. 44, no. 8, 2021. DOI: 10.1093/sleep/zsab045.

[7] E. Tham, N. Schneider, and B. Broekman, ”Infant sleep and its relation with cognition and growth: A narrative review”, *Nature and Science of Sleep*, vol. Volume 9, pp. 135–149, May 2017. DOI: 10.2147/NSS.S125992.

[8] J. Mindell and A. Williamson, ”Benefits of a bedtime routine in young children: Sleep, development, and beyond”, *Sleep Medicine Reviews*, vol. 40, Nov. 2017. DOI: 10.1016/j.smrv.2017.10.007.

[9] S. L. Allen, M. D. Howlett, J. A. Coulombe, and P. V. Corkum, ”Abcs of sleeping: A review of the evidence behind pediatric sleep practice recommendations”, *Sleep medicine reviews*, vol. 29, pp. 1–14, 2016. DOI: 10.1016/j.smrv.2015.08.006.

[10] O. Bruni, E. Baumgartner, S. Sette, *et al.*, ”Longitudinal study of sleep behavior in normal infants during the first year of life”, *Journal of Clinical Sleep Medicine*, vol. 10, no. 10, pp. 1119–1127, 2014. DOI: 10.5664/jcsm.4114.

[11] B. C. Galland, B. J. Taylor, D. E. Elder, and P. Herbison, ”Normal sleep patterns in infants and children: A systematic review of observational studies”, *Sleep Medicine Reviews*, vol. 16, no. 3, pp. 213–222, 2012, ISSN: 1087-0792. DOI: 10.1016/j.smrv.2011.06.001.

[12] F. G. Vital-Lopez, T. J. Balkin, and J. Reifman, ”Models for predicting sleep latency and sleep duration”, *Sleep*, vol. 44, no. 5, zsaa263, Nov. 2020, ISSN: 0161-8105. DOI: 10.1093/sleep/zsaa263.

[13] O. Bruni, E. Baumgartner, S. Sette, *et al.*, ”Longitudinal study of sleep behavior in normal infants during the first year of life”, *Journal of clinical sleep medicine : JCSM : official publication of the American Academy of Sleep Medicine*, vol. 10, Sep. 2014. DOI: 10.5664/jcsm.4114.

[14] S. Paruthi, L. J. Brooks, C. D’Ambrosio, *et al.*, ”Recommended amount of sleep for pediatric populations: A consensus statement of the american academy of sleep medicine”, *Journal of Clinical Sleep Medicine*, vol. 12, no. 06, pp. 785–786, 2016. DOI: 10.5664/jcsm.5866.

[15] L. Gelman, *Baby and children sleep chart*, `https://www.parents.com/baby/sleep/basics/baby-and-children-sleep-chart/`.

[16] K. McGraw, R. Hoffmann, C. Harker, and J. H. Herman, ”The development of circadian rhythms in a human infant”, *Sleep*, vol. 22, no. 3, pp. 303–310, 1999. DOI: `10.1093/sleep/22.3.303`.

[17] A. T. Newton and G. J. Reid, ”Regular, intermittent, and spontaneous: Patterns of preschool children’s nap behavior and their correlates”, *Sleep Medicine*, vol. 102, pp. 105–116, 2023, ISSN: 1389-9457. DOI: `10.1016/j.sleep.2022.12.019`.

[18] UNICEF *et al.*, ”World bank group, unicef urge greater investment in early childhood development”, 2016. [Online]. Available: `https://www.worldbank.org/en/news/press-release/2016/04/14/world-bank-group-unicef-urge-greater-investment-in-early-childhood-development#`.

[19] I. H. Sarker, ”Machine learning: Algorithms, real-world applications and research directions”, *SN computer science*, vol. 2, no. 3, p. 160, 2021. DOI: `10.1007/s42979-021-00592-x`.

[20] A. B. Nassif, I. Shahin, I. Attili, M. Azzeh, and K. Shaalan, ”Speech recognition using deep neural networks: A systematic review”, *IEEE access*, vol. 7, pp. 19 143–19 165, 2019. DOI: `10.1109/ACCESS.2019.2896880`.

[21] J. Alzubi, A. Nayyar, and A. Kumar, ”Machine learning from theory to algorithms: An overview”, *Journal of Physics: Conference Series*, vol. 1142, no. 1, p. 012 012, Nov. 2018. DOI: `10.1088/1742-6596/1142/1/012012`.

[22] B. Mahesh, ”Machine learning algorithms -a review”, Jan. 2019. DOI: `10.21275/ART20203995`.

[23] S. Badillo, B. Banfai, F. Birzele, *et al.*, ”An introduction to machine learning”, *Clinical pharmacology & therapeutics*, vol. 107, no. 4, pp. 871–885, 2020. DOI: 10.1002/cpt.1796.

[24] J. Tanuwijaya and S. Hansun, ”Lq45 stock index prediction using k-nearest neighbors regression”, vol. 8, Sep. 2019. DOI: 10.35940/ijrte.C4663.098319.

[25] Z. Yao and W. Ruzzo, ”A regression-based k nearest neighbor algorithm for gene function prediction from heterogeneous data”, *BMC bioinformatics*, vol. 7 Suppl 1, S11, Feb. 2006. DOI: 10.1186/1471-2105-7-S1-S11.

[26] S. Imandoust and M. Bolandraftar, ”Application of k-nearest neighbor (knn) approach for predicting economic events: Theoretical background”, Jan. 2013, pp. 605–610. [Online]. Available: https://api.semanticscholar.org/CorpusID:15532755.

[27] F. Tan, ”Regression analysis and prediction using lstm model and machine learning methods”, *Journal of Physics: Conference Series*, vol. 1982, no. 1, p. 012 013, Jul. 2021. DOI: 10.1088/1742-6596/1982/1/012013.

[28] A. H. Mirza and S. Cosan, ”Computer network intrusion detection using sequential lstm neural networks autoencoders”, in *2018 26th Signal Processing and Communications Applications Conference (SIU)*, 2018, pp. 1–4. DOI: 10.1109/SIU.2018.8404689.

[29] K. Taunk, S. De, S. Verma, and A. Swetapadma, ”A brief review of nearest neighbor algorithm for learning and classification”, in *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*, 2019, pp. 1255–1260. DOI: 10.1109/ICCS45141.2019.9065747.

[30] T. Chen and C. Guestrin, ”Xgboost: A scalable tree boosting system”, *CoRR*, 2016. arXiv: 1603.02754.

[31] D. Borup, B. J. Christensen, N. S. Mühlbach, and M. S. Nielsen, ”Targeting predictors in random forest regression”, *International Journal of Forecasting*, vol. 39, no. 2, pp. 841–868, 2023, ISSN: 0169-2070. DOI: 10.1016/j.ijforecast.2022.02.010.

[32] W. Samek, G. Montavon, S. Lapuschkin, C. J. Anders, and K.-R. Müller, ”Explaining deep neural networks and beyond: A review of methods and applications”, *Proceedings of the IEEE*, vol. 109, no. 3, pp. 247–278, 2021. DOI: 10.1109/JPROC.2021.3060483.

[33] A. Sherstinsky, ”Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network”, *Physica D: Nonlinear Phenomena*, vol. 404, p. 132 306, 2020, ISSN: 0167-2789. DOI: 10.1016/j.physd.2019.132306.

[34] C. Thomas, ”Recurrent neural networks and natural language processing”, 2019. [Online]. Available: https://towardsdatascience.com/recurrent-neural-networks-and-natural-languageprocessing-73af640c2aa1.

[35] Y. Yu, X. Si, C. Hu, and J. Zhang, ”A review of recurrent neural networks: Lstm cells and network architectures”, *Neural computation*, vol. 31, no. 7, pp. 1235–1270, 2019. DOI: 10.1162/neco_a_01199.

[36] D. Chicco, M. J. Warrens, and G. Jurman, ”The coefficient of determination r-squared is more informative than smape, mae, mape, mse and rmse in regression analysis evaluation”, *Peerj computer science*, vol. 7, e623, 2021. DOI: 10.7717/peerj-cs.623.

[37] S. Park, C.-T. Li, S. Han, C. Hsu, S. W. Lee, and M. Cha, ”Learning sleep quality from daily logs”, in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD ’19, Anchorage, AK, USA: Association for Computing Machinery, 2019, pp. 2421–2429, ISBN: 9781450362016. DOI: 10.1145/3292500.3330792.

[38] K. Niu, S. Zhang, H. Jiao, C. Cheng, and C. Wang, ”Btp: A bedtime predicting algorithm via smartphone screen status”, *Wireless Communications and Mobile Computing*, vol. 2018, pp. 1–11, Oct. 2018. DOI: 10.1155/2018/7619102.

[39] k. Park, S. Lee, S. Wang, *et al.*, ”Sleep Prediction Algorithm Based On Deep Learning Technology”, *Sleep*, vol. 42, no. Supplement$_1$, A172–A172, Apr. 2019, ISSN: 0161-8105. DOI: 10.1093/sleep/zsz067.425.

[40] V. R. K. Sathish, W. L. Woo, and E. S. L. Ho, ”Predicting sleeping quality using convolutional neural networks”, *ArXiv*, 2022. DOI: 10.48550/arXiv.2204.13584.

[41] M. D H and J. Kasubi, ”Predicting of sleep behaviour in smart homes based on multi-residents using machine learning techniques”, *SN Computer Science*, vol. 2, Jul. 2021. DOI: 10.1007/s42979-021-00643-3.

[42] T. Lund, ”Hacking my infant twins’ sleep with machine learning and data science”, 2016. [Online]. Available: https://medium.com/dad-on-the-run/hacking-my-infant-twins-sleep-with-machine-learning-and-data-science-6c1e38a71677.

[43] W. Hidayat, T. Tambunan, and R. Budiawan, ”Empowering wearable sensor generated data to predict changes in individual’s sleep quality”, May 2018. DOI: 10.1109/ICoICT.2018.8528750.

[44] A. Sathyanarayana, J. Srivastava, and L. Fernandez-Luque, ”The science of sweet dreams: Predicting sleep efficiency from wearable device data”, *Computer*, vol. 50, pp. 30–38, Apr. 2017. DOI: 10.1109/MC.2017.91.

[45] A. Sathyanarayana, S. Joty, L. Fernandez-Luque, *et al.*, ”Sleep quality prediction from wearable data using deep learning”, *JMIR Mhealth Uhealth*, vol. 4, no. 4, e125, Nov. 2016, ISSN: 2291-5222. DOI: 10.2196/mhealth.6562.

[46] O. Azuara and Z. Gillette, ”Using machine learning to determine optimal sleeping schedules of individual college students”, in Oct. 2022, pp. 13–25, ISBN: 978-3-031-17901-3. DOI: 10.1007/978-3-031-17902-0_2.

[47] K. Park, S. Lee, C. Lee, *et al.*, ”Prediction of good sleep with physical activity and light exposure: A preliminary study”, *Journal of clinical sleep medicine: JCSM: official publication of the American Academy of Sleep Medicine*, vol. 18, Jan. 2022. DOI: 10.5664/jcsm.9872.

[48] J. Mindell, E. Leichman, J. Composto, C. Lee, B. Bhullar, and R. Walters, ”Development of infant and toddler sleep patterns: Real-world data from a mobile application”, *Journal of sleep research*, vol. 25, Jun. 2016. DOI: 10.1111/jsr.12414.

[49] S. D. Harlow, X. Lin, and M. Ho, ”Analysis of menstrual diary data across the reproductive life span applicability of the bipartite model approach and the importance of within-woman variance”, *Journal of Clinical Epidemiology*, vol. 53, no. 7, pp. 722–733, 2000, ISSN: 0895-4356. DOI: 10.1016/S0895-4356(99)00202-4.

[50] T. Oliveira, G. Bruinvels, C. Pedlar, and J. Newell, ”Modelling menstrual cycle length in athletes using state-space models”, Dec. 2020. DOI: 10.21203/rs.3.rs-122553/v1.

[51] I. Urteaga, K. Li, A. Shea, V. J. Vitzthum, C. H. Wiggins, and N. Elhadad, ”A generative modeling approach to calibrated predictions: A use case on menstrual cycle length prediction”, in *Proceedings of the 6th Machine Learning for Healthcare Conference*, K. Jung, S. Yeung, M. Sendak, M. Sjoding, and R. Ranganath, Eds., ser. Proceedings of Machine Learning Research, vol. 149, PMLR, Aug. 2021, pp. 535–566. [Online]. Available: https://proceedings.mlr.press/v149/urteaga21a.html.

[52] K. Li, I. Urteaga, A. Shea, V. Vitzthum, C. Wiggins, and N. Elhadad, "A generative, predictive model for menstrual cycle lengths that accounts for potential self-tracking artifacts in mobile health data", Feb. 2021. DOI: 10.48550/arXiv.2102.12439.

[53] K. Li, I. Urteaga, A. Shea, V. J. Vitzthum, C. H. Wiggins, and N. Elhadad, "A predictive model for next cycle start date that accounts for adherence in menstrual self-tracking", *Journal of the American Medical Informatics Association*, vol. 29, no. 1, pp. 3–11, Sep. 2021, ISSN: 1527-974X. DOI: 10.1093/jamia/ocab182.

[54] P. Bortot, G. Masarotto, and B. Scarpa, "Sequential predictions of menstrual cycle lengths", *Biostatistics*, vol. 11, no. 4, pp. 741–755, Apr. 2010, ISSN: 1465-4644. DOI: 10.1093/biostatistics/kxq020.

[55] O. Bastidas, S. Zahia, A. Fuente-Vidal, *et al.*, "Predicting physical exercise adherence in fitness apps using a deep learning approach", *International Journal of Environmental Research and Public Health*, vol. 18, p. 10 769, Oct. 2021. DOI: 10.3390/ijerph182010769.

[56] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python", *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. [Online]. Available: www.scikit-learn.org/.

[57] M. M. Ahsan, M. A. P. Mahmud, P. K. Saha, K. D. Gupta, and Z. Siddique, "Effect of data scaling methods on machine learning algorithms and model performance", *Technologies*, vol. 9, no. 3, 2021, ISSN: 2227-7080. DOI: 10.3390/technologies9030052.

[58] T. D.K., P. B.G, and F. Xiong, "Auto-detection of epileptic seizure events using deep neural network with different feature scaling techniques", *Pattern*

Recognition Letters, vol. 128, pp. 544–550, 2019, ISSN: 0167-8655. DOI: `10.1016/j.patrec.2019.10.029`.

[59] Martín Abadi, Ashish Agarwal, Paul Barham, *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: `www.tensorflow.org/`.

Appendix A Loss Curves

A.1 Learning curve on Neural Networks

A learning curve is a graphical representation that shows the model's performance on both the training and validation datasets as the training progresses. This curve helps in understanding how well the model is learning and can indicate issues such as over-fitting or under-fitting.

LSTM

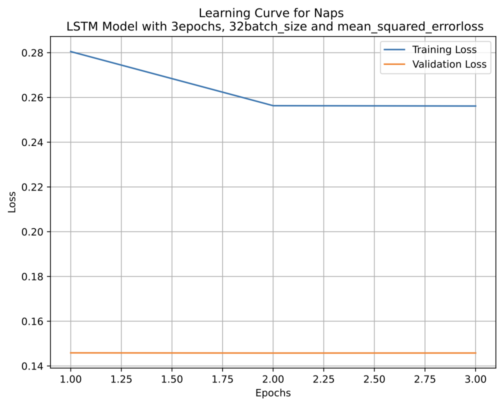


Figure A.1: LSTM Learning curve - number of naps

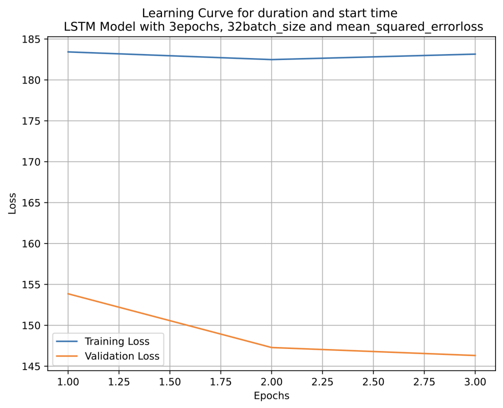


Figure A.2: LSTM Learning curve - duration & starting time

RNN

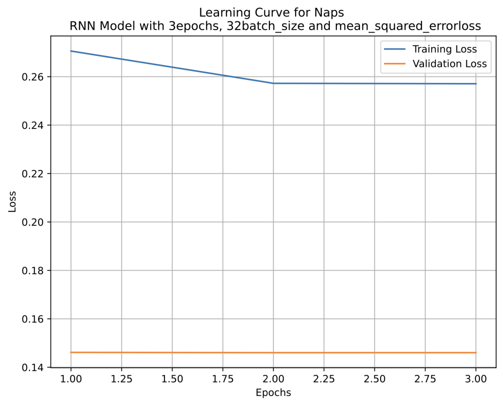


Figure A.3: RNN Learning curve - number of naps

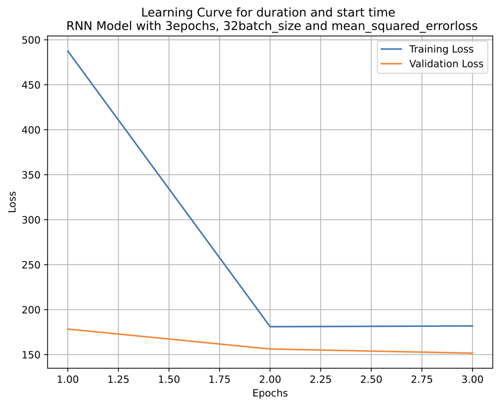


Figure A.4: RNN Learning curve - duration & starting time

Appendix B Code

The main code for this thesis was written using Python, which was adapted from previous Jupyter Notebooks. Found here are the most relevant files used to execute the code pipeline. In order to do so, the main.py file, in Section B.1, has to be called. This one calls preprocess.py, in Section B.2, and features.py, in Section B.3. Once we have the dataset ready to use for the machine learning models, the main code calls on ml.py, Section B.4, which feeds from models.py, Section B.5 and plots.py Section B.6.

B.1 MAIN

The script imports necessary modules for preprocessing, feature extraction, and machine learning tasks, along with pandas for data manipulation and sklearn for label encoding.

It defines a main function responsible for reading log and baby data, preprocessing, and performing evaluations using machine learning models. The main function takes several parameters such as paths for data and output files, along with flags for preprocessing and feature extraction. It initializes parameters for window size, model information, and number of days to predict, which are then passed to the rest of functions. Based on the flags, it either preprocesses the data or reads preprocessed data from files. Next, it iterates over the specified number of days to predict, splitting the data into training and testing sets, and performs machine learning evaluations. Finally, it evaluates predictions, saves results, and returns accumulated predicted data and evaluation metrics. The script also includes conditional statements to run the main function based on command-line arguments.

```python
from preprocess import *
from features import *
from ml import *
import sys
import pandas as pd
from sklearn.preprocessing import LabelEncoder


#----------------- MAIN -----------------#
def main(preprocess, features, path_log, path_baby,
    output_path_processed, output_path_stats, models_path,
    results_path, features_path):
    """
    Main function that reads the log and baby data, preprocesses
        the data, and performs the evaluation.
    """
    window_size = {
        'naps_babyid': '7',
        'duration_babyid': '7',
        'start_babyid': '7'
    }

    model_info = {
        'retrain': False,
        'model_type': 'XGBoost', # 'LSTM', 'RNN', 'KNN', 'RF'

        # params LSTM or RNN
        'num_epoch': '3',
        'batch_size': '32',
        'loss': 'mean_squared_error',

        # params XGBoost
        'num_estimators': '100',
        'learning_rate': '0.1',
        'objective': 'reg:squarederror',

        # params KNN
        'num_neighbors': '3',

        # params RF
        'max_depth': 10,
        'n_estimators': 100,
        'bootstrap': True
    }

    num_days_to_predict = 1
    evaluate_on_only_naps = True
    orig_log_size = 30044030

    column_names = ['id', 'babyId', 'category', 'createdAt', 'end',
        'start', 'timezone_offset_minutes', 'skipped', 'birthday']

    if preprocess == True:
        print(' - About to read log')
        df_log = pd.read_csv(path_log, names=column_names)
        df_log.drop(df_log.index[0], inplace=True)
        orig_log_size = df_log.shape[0]
        print(f' - Reading data done, original data size: {
            orig_log_size}')
        print(' - Preprocess starting')
        df_proc = preprocessing(df_log, output_path_processed,
            output_path_stats)
    elif preprocess == False:
        df_proc = pd.read_csv(output_path_processed)

    df = df_proc.copy()
    label_encoder = LabelEncoder()
    df['babyId'] = label_encoder.fit_transform(df['babyId'])
    joblib.dump(label_encoder, os.path.join(models_path, '
        baby_encoder.joblib'))
    category_encoder = LabelEncoder()
    df['category'] = category_encoder.fit_transform(df['category'])
    joblib.dump(category_encoder, os.path.join(models_path, '
        category_encoder.joblib'))

    df = df.sort_values(by=['start'])
    last_day_per_baby = df.groupby('babyId')['date'].max()

    if features == True:
        print(' - Feature extraction starting')
        print(df.columns)
        df_feat_naps = add_features_num_naps(df, window_size)
        df_feat = add_features(df, window_size, output_path_stats)

        df_feat_naps = df_feat_naps[['babyId', 'date', 'category',
            'num_nap', 'age_on_log_dec', 'naps_that_day','
            mean_naps_at_age', 'std_naps_at_age', '
            rolling_mean_naps_that_day_babyid', '
            rolling_std_naps_that_day_babyid']]
        df_feat = df_feat[['babyId', 'date', 'category', 'num_nap',
             'age_on_log_dec', 'duration', 'start_24hr', 'start', '
            timezone_offset_minutes', 'end', 'naps_that_day', '
            mean_duration', 'mean_start_24hr', 'std_duration', '
            std_start_24hr', 'rolling_mean_duration_babyid', '
            rolling_std_duration_babyid', 'rolling_mean_start_24hr_babyid', 'rolling_std_start_24hr_babyid']]
        df_feat_naps.to_parquet(os.path.join(features_path, 'sample_filtered_70_features_naps.parquet'))
        df_feat.to_parquet(os.path.join(features_path, 'sample_filtered_70_features.parquet'))

    elif features == False:
        df_feat_naps = pd.read_parquet(os.path.join(features_path, 'sample_filtered_70_features_naps.parquet'))
        df_feat = pd.read_parquet(os.path.join(features_path, 'sample_filtered_70_features.parquet'))
    predicted_data_accumulated = pd.DataFrame()
    evaluation_metrics = pd.DataFrame(columns=['mae_naps', 'accuracy_naps', 'medae_naps', 'r2_naps', 'mae_duration', 'medae_duration', 'r2_duration', 'mae_start', 'medae_start', 'r2_start'])

    training_sizes = []
    testing_sizes = []

    for i in range(0, num_days_to_predict):
        # print(f'Iteration {i+1} out of {num_days_to_predict}')
        print(' - Train/test split starting')
        training_data_naps, testing_data_naps, training_data, testing_data = train_test_split(df_feat_naps, df_feat, last_day_per_baby, i)
        training_sizes.append(training_data.shape[0])
        testing_sizes.append(testing_data.shape[0])
        predicted_data, y_pred_naps, y_test_naps = perform_evaluation(training_data_naps, testing_data_naps, training_data, testing_data, label_encoder,
            category_encoder, models_path, results_path, model_info)
        predicted_data_accumulated = pd.concat([
            predicted_data_accumulated, predicted_data])
    evaluation_metrics = evaluate_predictions(
        predicted_data_accumulated,  y_pred_naps, y_test_naps,
        evaluation_metrics, df_proc.shape[0], training_sizes,
        testing_sizes, orig_log_size, model_info,
        evaluate_on_only_naps)
    save_results(predicted_data_accumulated, evaluation_metrics,
        model_info, results_path)

    return predicted_data_accumulated, evaluation_metrics

if __name__ == "__main__":
    preprocess = sys.argv[1]
    preprocess = True if preprocess == 'True' else False
    features = sys.argv[2]
    features = True if features == 'True' else False
    path_log = '/home/ubuntu/ANNA-THESIS/DATA/log/
        baby_logs_sample_filtered_70.csv'
    path_baby = '/home/ubuntu/ANNA-THESIS/DATA/baby/part-00000-
        f022d9fe-b173-434c-8200-4af77a4e96bd-c000.snappy.parquet'
    output_path_processed = '/home/ubuntu/ANNA-THESIS/DATA/
        processed/processed_baby_logs_sample_filtered_70.csv'
    output_path_stats = '/home/ubuntu/ANNA-THESIS/DATA/processed/
        stats_baby_logs_sample_filtered_70.csv'
    features_path = '/home/ubuntu/ANNA-THESIS/DATA/features/'
    models_path = '/home/ubuntu/ANNA-THESIS/DATA/models/'
    results_path = '/home/ubuntu/ANNA-THESIS/DATA/results/'
    main(preprocess, features, path_log, path_baby,
        output_path_processed, output_path_stats, models_path,
        results_path, features_path)
```

B.2 PREPROCESS

This script processes a log dataframe for baby sleep data. The preprocessing functions include merging and renaming columns, normalizing timezones, calculating the baby's age at the time of the log, and fixing data by filtering out unwanted records. Additionally, it calculates the duration of each sleep event, the number of naps per day, and various statistics such as the previous wake time and the mean number of naps at different ages. The processed data and statistics are then saved to specified file paths.

```python
import pandas as pd

# --------------------- PRE PROCESSING ---------------------
def merge_log_baby (df_log):
    """
     Merge the log and baby dataframes.
    """
    df_log['birthday'] = pd.to_datetime(df_log['birthday'])
    df_log['birthday'] = df_log['birthday'].dt.tz_localize(None)
    df_log = df_log.rename(columns={'babyid': 'babyId'})
    earliest_birthday = df_log['birthday'].min()
    latest_birthday = df_log['birthday'].max()
    print(f'Earliest birthday: {earliest_birthday}')
    print(f'Latest birthday: {latest_birthday}')
    df = df_log.copy()
    df['createdAt'] = pd.to_datetime(df['createdAt'])
    df['createdAt'] = df['createdAt'].dt.tz_localize(None)
    df['start'] = pd.to_datetime(df['start'])
    df['start'] = df['start'].dt.tz_localize(None)
    df['end'] = pd.to_datetime(df['end'])
    df['end'] = df['end'].dt.tz_localize(None)
    df['createdAt'] = pd.to_datetime(df['createdAt'])
    df['birthday'] = pd.to_datetime(df['birthday'])

    return df

def normalize_timezone(df):
    """
    Normalize the timezone of the log dataframe.
    """
    df['start'] = pd.to_datetime(df['start'])
    df['end'] = pd.to_datetime(df['end'])
    df.dropna(subset=['timezone_offset_minutes'], inplace=True)
    df['start'] = pd.to_datetime(df['start']) + pd.to_timedelta(df[
        'timezone_offset_minutes'].astype(float).astype(int), unit='
        m')
    df['end'] =  pd.to_datetime(df['end']) + pd.to_timedelta(df['
        timezone_offset_minutes'].astype(float).astype(int), unit='m
        ')
    df['createdAt'] =  pd.to_datetime(df['createdAt']) + pd.
        to_timedelta(df['timezone_offset_minutes'].astype(float).
        astype(int), unit='m')

    return df

def age_on_log(df):
    """
    Compute the age of the baby at the time of the log.
    """
    df['age_on_log'] = df['createdAt'] - df['birthday']
    df['age_on_log'] = df['age_on_log'].dt.days
    df['age_on_log'] = df['age_on_log'] / 30
    df['start'] = pd.to_datetime(df['start'])
    df['date'] = df['start'].dt.date
    df['date'] = pd.to_datetime(df['date'])

    return df

def fix_data(df):
    """
    Fix the data from the log dataframe.
    Remove logs from dates earlier than August 2020.
    Take only logs under 30 months.
    """
    df = df[df['date'] >= '2020-08-01']
    df = df[df['age_on_log'] >= 2]
    df = df[df['age_on_log'] <= 24]

    return df

def duration(df_2):
    """
    Compute the duration of each entry.
    """
    df_2['duration'] = df_2['end'] - df_2['start']
    df_2['duration'] = df_2['duration'].dt.total_seconds() / 60
    df_2.loc[df_2['category'] == 'BED_TIME', 'duration'] = 0

    return df_2

def naps_per_day(df):
    """
    Compute the number of naps per day.
    """
    df_nap = df[df['category'] == 'NAP']
    df_nap['naps_that_day'] = 0
    df_nap = df_nap[((df_nap['end']-df_nap['start']).dt.
        total_seconds()/60) >= 5]
    df_nap = df_nap[((df_nap['end']-df_nap['start']).dt.
        total_seconds()/60) <= 300]
    df_nap = df_nap.groupby(['babyId', 'date']).agg({'naps_that_day
        ': 'count'}).reset_index()
    df_nap = df_nap[~df_nap.duplicated(keep='first')]
    df_merged = pd.merge(df, df_nap[['babyId', 'date', '
        naps_that_day']], left_on=['babyId', 'date'], right_on=['
        babyId', 'date'], how='left')
    df_merged['naps_that_day'] = df_merged['naps_that_day'].fillna
        (0)
    df_merged = df_merged[((df_merged['category'] != 'NAP') | (((
        df_merged['end']-df_merged['start']).dt.total_seconds()/60)
        >= 5))]
    df_merged = df_merged[((df_merged['category'] != 'NAP') | (((
        df_merged['end']-df_merged['start']).dt.total_seconds()/60)
        <= 600))]
    df_2 = df_merged.copy()
    df_2 = df_merged[['babyId','category', 'date', 'start', 'end',
        'age_on_log', 'duration', 'naps_that_day', '
        timezone_offset_minutes']]
    df_2['start'] = pd.to_datetime(df_2['start'])
    df_2['end'] = pd.to_datetime(df_2['end'])

    return df_2

def num_nap(df):
    """
    Compute the number of nap that day.
    """
    df['num_nap'] = 0
    df.loc[df['category'] == 'BED_TIME', 'num_nap'] = df.loc[df['
        category'] == 'BED_TIME', 'naps_that_day'] + 1
```

```python
    nap_entries = df['category'] == 'NAP'
    df = df.sort_values(['babyId', 'start'])
    df.loc[nap_entries, 'num_nap'] = df[nap_entries].groupby(['babyId', 'date']).cumcount() + 1

    return df

def prev_wake_time(df):
    """
    Compute the previous wake time.
    The previous wake time is the time between the end of the previous sleep and the start of the current sleep.
    If the previous wake time is more than 20 hours, it is set to 0.
    """
    filtered_df = df[df['category'].isin(['NAP', 'BED_TIME', 'WOKE_UP'])]
    filtered_df = filtered_df.sort_values(['babyId', 'start'])
    filtered_df['time_diff'] = filtered_df.groupby('babyId')['start'].diff().dt.total_seconds() / 60
    filtered_df['previous_wake_time'] = (filtered_df['start'] - filtered_df['end'].shift()).fillna(pd.Timedelta(seconds=0)).dt.total_seconds() / 60
    mask = (filtered_df['start'] - filtered_df['end'].shift()) >= pd.Timedelta(days=1)
    filtered_df.loc[mask, 'previous_wake_time'] = 0
    mask = (filtered_df['start'] < filtered_df['end'].shift())
    filtered_df.loc[mask, 'previous_wake_time'] = 0
    filtered_df.loc[filtered_df['time_diff'] >= 1320, 'time_diff'] = 0
filtered_df.loc[filtered_df['time_diff'].isna(), 'previous_wake_time'] = 0
    df['prev_wake_time'] = 0
    df.loc[df['category'].isin(['NAP', 'BED_TIME', 'WOKE_UP']), '
        prev_wake_time'] = filtered_df['previous_wake_time']
    df.loc[df['prev_wake_time'] > 1200, 'prev_wake_time'] = 0  #
        more than 20 hours
    df.loc[df['prev_wake_time'] < -1200, 'prev_wake_time'] = 0  #
        more than 20 hours
    # Accumulated wake time
    df['accumulated_wake_time'] = df.groupby(['babyId', 'date'])['
        prev_wake_time'].cumsum()

    return df

def mean_naps_age_on_log(df):
    """
    Data analysis to compute the mean duration of naps, bed time
        and the number of naps per day.
    """
    df['age_on_log_dec'] = df['age_on_log'].astype(float).round(1)
    df['age_on_log'] = df['age_on_log'].astype(int)
    df['age_on_log'] = df['age_on_log'].astype(str)
    df['age_on_log'] = df['age_on_log'].str.replace('.0', '', regex
        =False)
    df['age_on_log'] = df['age_on_log'].astype(int)
    means_naps_age = df[df['category'] == 'NAP'].groupby(['
        age_on_log_dec']).agg({'naps_that_day': 'mean'}).reset_index
        ()
    means_naps_age.rename(columns={'naps_that_day': '
        mean_naps_at_age'}, inplace=True)

    return  df, means_naps_age

def save_stats (df_stats, path_stats, df, path_new):
    """
    Save the statistics and the processed log dataframe to parquet
       files with snappy compression method.
    """
    df['timezone_offset_minutes'].fillna(0, inplace=True)
    df['timezone_offset_minutes'] = pd.to_numeric(df['
       timezone_offset_minutes'], errors='coerce')
    df_stats.to_csv(path_stats)
    df.to_csv(path_new)

def take_sleep_logs(df):
    """
    Take the naps that have an age of at least 2 months.
    """
    nap_data = df[df['category'].isin(['NAP', 'WOKE_UP', 'BED_TIME'
       ])]

    return nap_data

def time_epochs(df):
    """
    Compute the start time in epoch format.
    """
    df_start = df['start'].apply(lambda x: x.strftime('%H:%M:%S'))
    df_start = pd.to_datetime(df_start, format='%H:%M:%S')
    df['start_epoch'] = df_start.apply(lambda x: x.timestamp()/60)
       # in minutes

    return df

def time_to_24hr_cycle(df, time_column):
    """
    Convert the time to a 24-hour cycle.
    """
```

```python
    df_copy = df.copy()
    df_copy['start_24hr'] = df_copy[time_column].apply(lambda x: x.
        hour + x.minute / 60 if not pd.isnull(x) else None)
    return df_copy

def reduce_outliers(df, means_naps_age):
    """
    Remove the outliers from the log dataframe.
    Remove the BED_TIME category that have duration 0 and delete
        the whole day if baby WOKE_UP after 14:00.
    """
    df['start'] = pd.to_datetime(df['start'])
    means_naps_age = means_naps_age.rename(columns={'
        mean_naps_at_age': 'mean_naps_at_age'})
    df = pd.merge(df, means_naps_age, on='age_on_log_dec', how='
        left')
    # Delete the whole day if
    mask = (
        (df['category'] == 'WOKE_UP') & (df['start'].dt.hour >= 14)
             |
        (df['category'] == 'WOKE_UP') & (df['start'].dt.hour <= 4)
            |
        (df['category'] == 'BED_TIME') & (df['start'].dt.hour <=
            17) |
        (df['category'] == 'NAP') & (df['duration'] >= 210) |
        (df['category'] == 'NAP') & (df['start'].dt.hour <= 5)
    )
    indices_to_delete = df[mask].index
    babyId_date_combinations = df.loc[indices_to_delete, ['babyId',
         'start']].apply(lambda x: (x['babyId'], x['start'].date()),
         axis=1)
    df = df[~((df.set_index(['babyId', df['start'].dt.date]).index.
        isin(babyId_date_combinations)))]
    no_wokeup_or_bedtime_indices = df.groupby(['babyId', df['start'
        ].dt.date]).filter(lambda x: not (('WOKE_UP' in x['category'
        ].values) and ('BED_TIME' in x['category'].values))).index
    df = df.drop(index=no_wokeup_or_bedtime_indices)
    print('      - number of logs to remove because they dont have
        wokeup / bedtime:', len(no_wokeup_or_bedtime_indices))
    df['log_count_per_baby'] = df.groupby('babyId')['babyId'].
        transform('size')
    df = df[df['log_count_per_baby'] > 25]
    df = df.drop(columns=['log_count_per_baby'])

    return df

def means_data(df):
    df['start'] = pd.to_datetime(df['start'])
    std_naps_age = df[df['category'] == 'NAP'].groupby(['
        age_on_log_dec']).agg({'naps_that_day': ['std']}).
        reset_index()
    std_naps_age.columns = ['age_on_log_dec', 'std_naps_at_age']
    df = df.merge(std_naps_age, on='age_on_log_dec', how='left')
    df_stats = df[df['category'].isin(['NAP', 'BED_TIME', 'WOKE_UP'
        ])].groupby(['age_on_log_dec', 'category', 'naps_that_day',
        'num_nap']).agg({'duration': ['mean', 'std'], 'start_24hr':
        ['mean', 'std']}).reset_index()
    df_stats.columns = ['age_on_log_dec', 'category', '
        naps_that_day', 'num_nap', 'duration_mean', 'duration_std',
        'start_24hr_mean', 'start_24hr_std']
    df_stats_melted_duration_mean = pd.melt(df_stats, id_vars=['
        age_on_log_dec', 'category', 'naps_that_day', 'num_nap'],
        value_vars=['duration_mean'],var_name='duration_mean',
        value_name='duration_mean_value')
    df_stats_melted_duration_std = pd.melt(df_stats, id_vars=['
        age_on_log_dec', 'category', 'naps_that_day', 'num_nap'],
```

```python
value_vars=['duration_std'],var_name='duration_std', value_name='duration_std_value')
    df_stats_melted_start_mean = pd.melt(df_stats, id_vars=['age_on_log_dec', 'category', 'naps_that_day', 'num_nap'], value_vars=['start_24hr_mean'], var_name='start_24hr_mean', value_name='start_24hr_mean_value')
    df_stats_melted_start_std = pd.melt(df_stats, id_vars=['age_on_log_dec', 'category', 'naps_that_day', 'num_nap'], value_vars=['start_24hr_std'], var_name='start_24hr_std', value_name='start_24hr_std_value')
    df_stats_combined = pd.concat([df_stats_melted_duration_mean, df_stats_melted_duration_std['duration_std_value'],
    df_stats_melted_start_mean['start_24hr_mean_value'], df_stats_melted_start_std['start_24hr_std_value']], axis=1)
    df_stats_combined.drop(columns=['duration_mean'], inplace=True)
    return df, df_stats_combined

#--------------------- MAIN  ---------------------
def preprocessing(df_log, output_path_processed, output_path_stats):
    """
    Preprocess the data and save the statistics and the processed log dataframe to parquet files.
    """
    df = merge_log_baby(df_log)
    print('    - After merge log:', df.shape[0])
    df = normalize_timezone(df)
    print('    - After normalize timezone:', df.shape[0])
    df = age_on_log(df)
    print('    - After age on log:', df.shape[0])
    df = fix_data(df)
    print('    - After fix data:', df.shape[0])
    df = duration(df)
    print('     - After duration:', df.shape[0])
    df = naps_per_day(df)
    print('     - After naps per day:', df.shape[0])
    df = num_nap(df)
    print('     - After num nap:', df.shape[0])
    df = prev_wake_time(df)
    print('     - After prev wake time:', df.shape[0])
    df, means_naps_age = mean_naps_age_on_log(df)
    print('     - After means data:', df.shape[0])
    df_naps = take_sleep_logs(df)
    print('     - After take sleep logs:', df_naps.shape[0])
    df_time = time_epochs(df_naps)
    print('     - After time epochs:', df_time.shape[0])
    df_time_24 = time_to_24hr_cycle(df_time, 'start')
    print('     - After time to 24hr cycle:', df_time_24.shape[0])
    df = reduce_outliers(df_time_24, means_naps_age)
    print('     - After reduce outliers:', df.shape[0])
    df, df_stats  = means_data(df)
    save_stats(df_stats, output_path_stats, df,
        output_path_processed)

    return df
```

B.3 FEATURES

The code defines functions to calculate rolling mean and standard deviation for specified columns in a dataframe, optionally grouped by other columns. These functions (rolling_mean and rolling_std) shift data by a given number of periods and use a defined window size.

The add_general_features function merges and renames statistical features in the dataframe. The main functions, add_features_num_naps and add_features, add these rolling statistics to baby sleep data, focusing on naps per day, duration, and start time. They ensure the data is sorted and properly typed. The script's main section processes the data by calling these functions.

```python
import pandas as pd
import numpy as np

# --------------------- STATISTICAL FEATURES ---------------------
def rolling_mean(df, target_col, n=1, groupby_cols=None,
    window_size=2, name_col=None):
    """
    Calculates the rolling mean of a target column, with the option
        to group by other columns.
    """

    rolling_mean_col = f'rolling_mean_{target_col}_{name_col}'
    if groupby_cols:
        grouped = df.groupby(groupby_cols)[target_col].transform(
            lambda x: x.shift(periods=-n).rolling(window=window_size
            , min_periods=1).mean())
    else:
        grouped = df[target_col].shift(periods=-n).rolling(window=
            window_size, min_periods=1).mean()
    df[rolling_mean_col] = grouped

    return df

def rolling_std(df, target_col, n=1, groupby_cols=None, window_size
    =2, name_col=None):
    """
    Calculates the rolling standard deviation of a target column,
        with the option to group by other columns.
    """
    rolling_std_col = f'rolling_std_{target_col}_{name_col}'

    if groupby_cols:
        grouped = df.groupby(groupby_cols)[target_col].transform(
            lambda x: x.shift(periods=-n).rolling(window=window_size, min_periods=1).std())
    else:
        grouped = df[target_col].shift(periods=-n).rolling(window=window_size, min_periods=1).std()
    df[rolling_std_col] = grouped

    return df

def add_general_features(df, stats):
    df = df.merge(stats, how='left', on=['age_on_log_dec', 'naps_that_day', 'num_nap'])
    df = df.rename(columns={'category_x': 'category', 'duration_mean_value': 'mean_duration', 'duration_std_value': 'std_duration', 'start_24hr_mean_value': 'mean_start_24hr', 'start_24hr_std_value': 'std_start_24hr'})
    df.drop(columns=['category_y'], inplace=True)

    return df

def add_features_num_naps(preprocessed_data, window_size):
    """
    Adds rolling mean, rolling standard deviation, and rolling root mean square features to the preprocessed data.
    """
    window_size_naps_babyid = int(window_size['naps_babyid'])
    df = preprocessed_data
    df = df.sort_values(['start'])
    # NAPS
    n = 1
    woke_up_logs = df[df['category'] == 2]
    woke_up_logs = woke_up_logs.sort_values(['start'])
    woke_up_logs = rolling_mean(woke_up_logs, 'naps_that_day', n, ['babyId'], window_size_naps_babyid, 'babyid')
    woke_up_logs = rolling_std(woke_up_logs, 'naps_that_day', n, ['babyId'], window_size_naps_babyid, 'babyid')
    print('     - number of naps features added')
    columns_to_round = ['mean_naps_at_age', 'std_naps_at_age', 'rolling_mean_naps_that_day_babyid', 'rolling_std_naps_that_day_babyid']
    columns_to_int = ['timezone_offset_minutes', 'naps_that_day', 'num_nap', 'category']
    woke_up_logs[columns_to_round] = woke_up_logs[columns_to_round].astype(float).round(3)
    woke_up_logs[columns_to_int] = woke_up_logs[columns_to_int].astype('int8')

    return woke_up_logs

def add_features(preprocessed_data, window_size, stats_features):
    """
    Adds rolling mean, rolling standard deviation, and rolling root mean square features to the preprocessed data.
    """
    stats = pd.read_csv(stats_features)
    df = add_general_features(preprocessed_data, stats)
    window_size_duration_babyid = int(window_size['duration_babyid'])
    window_size_start_babyid = int(window_size['start_babyid'])
    df = df.sort_values(['start'])
    # DURATION
    n = 1
    df = rolling_mean(df, 'duration', n, ['num_nap', 'naps_that_day', 'babyId'], window_size_duration_babyid, 'babyid')
    df = rolling_std(df, 'duration', n, ['num_nap', 'naps_that_day', 'babyId'], window_size_duration_babyid, 'babyid')
    # START TIME
    n = 1
    df = rolling_mean(df, 'start_24hr',n, ['num_nap', 'naps_that_day', 'babyId'], window_size_start_babyid, 'babyid')
    df = rolling_std(df, 'start_24hr', n, ['num_nap', 'naps_that_day', 'babyId'], window_size_start_babyid, 'babyid')
    print('    - duration and start time features added')
    columns_to_round = ['start_24hr', 'mean_duration', 'mean_start_24hr', 'std_duration', 'std_start_24hr', 'rolling_mean_duration_babyid', 'rolling_std_duration_babyid', 'rolling_mean_start_24hr_babyid', 'rolling_std_start_24hr_babyid']
    columns_to_int = ['timezone_offset_minutes', 'naps_that_day', 'num_nap', 'category']
    df[columns_to_round] = df[columns_to_round].astype(float).round(3)
    df[columns_to_int] = df[columns_to_int].astype('int8')

    return df
```

B.4 ML

This script processes baby sleep data by importing necessary libraries and defining functions for data handling and model evaluation. It includes cycle_to_time for converting cycle times to a 24-hour format, train_test_split for splitting data into training and testing sets, and plot_features_importance for visualizing feature correlations. The predicted_outcomes function generates a dataframe of predicted outcomes for nap duration and start times. evaluate_predictions assesses prediction performance with various metrics. The main evaluation is handled by perform_evaluation, which manages the prediction process and generates plots, while save_results saves the evaluation metrics and predicted data to CSV files, detailing model parameters. This provides a streamlined workflow for analyzing baby sleep patterns with machine learning models.

```python
import pandas as pd
from plots import *
from models import *
import numpy as np
from sklearn.metrics import mean_absolute_error, r2_score,
    median_absolute_error, accuracy_score
from sklearn.metrics import mean_absolute_error
import seaborn as sns
import matplotlib.pyplot as plt

import os


"""-------------------------------"""
def cycle_to_time(df, cycle_column):
    """
    Convert cycle time to 24-hour time format.

    Args:
    - df (pd.DataFrame): Dataframe with the cycle time column.
    - cycle_column (str): Name of the cycle time column.

    Returns:
    - df_copy (pd.DataFrame): Copy of the input dataframe with the
        cycle time converted to 24-hour time format.
    """

    df_copy = df.copy()
    df_copy['start_time_predicted_24hr'] = pd.to_datetime(df_copy[
        cycle_column] * 60, unit='m').dt.strftime('%H:%M:%S')
    return df_copy[['start_time_predicted_24hr']]


def train_test_split(df_feat_naps, df_feat, last_day_per_baby, i=0)
    :
    """
    Splits the data into training and testing sets.
    For the training set for naps per day only the last 3 weeks of
        data are used.
    And the testing set is the last day of data for each baby.

    Args:
    - df (pd.DataFrame): The data to split into training and
        testing sets.

    Returns:
    - training_data (pd.DataFrame): The training data.
    - training_data_naps (pd.DataFrame): The training data for naps
        per day.
    - testing_data (pd.DataFrame): The testing data.
    """

    df_feat.dropna(inplace=True)
    df_feat_naps.dropna(inplace=True)
```

```python
    last_day_per_baby = df_feat_naps.groupby('babyId')['date'].max()
    print('    - number i passed:', i)
    last_day_per_baby = pd.to_datetime(last_day_per_baby)
    df_feat_naps['date'] = pd.to_datetime(df_feat_naps['date'])
    df_feat['date'] = pd.to_datetime(df_feat['date'])
    last_day_per_baby = last_day_per_baby - pd.DateOffset(i)
    training_data_naps = df_feat_naps[df_feat_naps.apply(lambda x: x['date'] < last_day_per_baby[x['babyId']], axis=1)]
    testing_data_naps = df_feat_naps[df_feat_naps.apply(lambda x: x['date'] == last_day_per_baby[x['babyId']], axis=1)]
    # last_day_per_baby = df_feat.groupby('babyId')['date'].max()
    training_data = df_feat[df_feat.apply(lambda x: x['date'] < last_day_per_baby[x['babyId']], axis=1)]
    testing_data = df_feat[df_feat.apply(lambda x: x['date'] == last_day_per_baby[x['babyId']], axis=1)]

    return training_data_naps, testing_data_naps, training_data, testing_data


def plot_features_importance(results_path, df, title):
    plt.figure(figsize=(10, 7))
    single_unique_columns = df.columns[df.nunique() == 1]
    df = df.drop(columns=single_unique_columns)
    df = df.drop(columns=['babyId', 'date'])
    if 'start' in df.columns:
        df = df.drop(columns=['start', 'end'])
    sns.set(rc = {'figure.figsize' : (10, 7)})
    corr = df.corr()
    corr_map = sns.heatmap(corr,  fmt = ".1g", cmap = "coolwarm")
    plt.title(f'Correlation Matrix of Features for {title}')
    plt.savefig(os.path.join(results_path, f'
        correlation_matrix_features_{title}.png'))


def predicted_outcomes (X_test, y_pred, y_test, X_test_naps,
    y_pred_naps, y_test_naps, label_encoder, category_encoder):
    """
    Create a dataframe with the predicted outcomes.
    """
    predicted_data = y_pred
    y_pred_naps_df = pd.DataFrame(y_pred_naps, columns=['
        predicted_naps_per_day'])
    y_test_naps_df = pd.DataFrame(y_test_naps.values, columns=['
        actual_naps_per_day'])
    merged_a = pd.concat([y_pred_naps_df, y_test_naps_df], axis=1)
    merged_a.index = y_test_naps.index
    merged_b = pd.concat([X_test_naps, merged_a], axis=1)
    merged_df = pd.merge(X_test, merged_b[['babyId', 'date', '
        predicted_naps_per_day', 'actual_naps_per_day']], on=['
        babyId', 'date'], how='left')
    y_pred_naps_extended = merged_df['predicted_naps_per_day']
    y_test_naps_extended = merged_df['actual_naps_per_day']
    actual_duration = y_test['duration'].values
    actual_start_24hr = y_test['start_24hr'].values
    predicted_data = pd.DataFrame({
            'babyId': X_test['babyId'],
            'babyId_orig': label_encoder.inverse_transform(X_test['
                babyId']),
            'category': X_test['category'],
            'category_orig': category_encoder.inverse_transform(
                X_test['category']),
            'age_on_log_dec': X_test['age_on_log_dec'],
            'date': X_test['date'],
            'num_nap': X_test['num_nap'],
            'predicted_naps_per_day': y_pred_naps_extended.values,
            'actual_naps_per_day':y_test_naps_extended.values,
            'predicted_duration': y_pred[:, 0],
            'actual_duration': actual_duration,
            'predicted_start_24hr': y_pred[:, 1],
            'actual_start_24hr': actual_start_24hr
        })
    predicted_data['predicted_start'] = cycle_to_time(
        predicted_data, 'predicted_start_24hr')
    predicted_data['actual_start'] = cycle_to_time(predicted_data,
        'actual_start_24hr')

    return predicted_data, y_pred_naps, y_test_naps

def evaluate_predictions(predicted_data, y_pred_naps, y_test_naps,
    evaluation_metrics, size_processed, training_sizes,
    testing_sizes, orig_log_size, model_info, only_naps=False):
    """
    Evaluate the predicted outcomes.
    """
    model_type = model_info['model_type']
    if model_type == 'XGBoost':
        params = {
            'N. estimators': int(model_info['num_estimators']),
            'Learning rate': float(model_info['learning_rate']),
            'Objective': model_info['objective'],
        }
    elif model_type == 'KNN':
        params = {
            'N. Neighbors': model_info['num_neighbors'],
        }
    elif model_type == 'LSTM' or model_type == 'RNN':
        params = {
            'N. epochs': int(model_info['num_epoch']),
            'Batch size': int(model_info['batch_size']),
            'Loss': model_info['loss'],
        }
    elif model_type == 'RF':
        params = {
            'N. estimators': int(model_info['n_estimators']),
            'Max depth': float(model_info['max_depth']),
            'Bootstrap': model_info['bootstrap'],
        }

    if only_naps == True:
        # from predicted data drop the rows with category_orig set to BED_TIME and WOKE_UP for the duration field
        predicted_data_nap = predicted_data[predicted_data['category'] != 0]
        predicted_data_nap = predicted_data_nap[predicted_data_nap['category'] != 2]
    else:
        predicted_data_nap = predicted_data

    print('')
    print('----------------------------------------------------------------------
    ')
    print(f'             PREDICTION RESULTS  -  {model_type}')
    print('')
    print('Size of original dataset:', orig_log_size)
    print('Size of processed dataset:', size_processed)
    for i, (train_size, test_size) in enumerate(zip(training_sizes, testing_sizes), start=1):
        print(f"Iteration {i}: Training size - {train_size}, Testing size - {test_size}")
    print('Number of babies:', predicted_data['babyId'].nunique())
    print('Parameters:', params)
    print('')
    print('---NAPS_PER_DAY---')
    mae_naps = mean_absolute_error(y_test_naps, y_pred_naps)
    print(f"Mean Absolute Error (MAE) for Naps per Day: {mae_naps
        :.2f} naps")
    accuracy_naps = accuracy_score(y_test_naps, y_pred_naps) * 100
    print(f"Accuracy for Naps per Day: {accuracy_naps:.2f}")
    medae_naps = median_absolute_error(y_test_naps, y_pred_naps)
    print(f"Median Absolute Error (MedAE) for Naps per Day: {
        medae_naps:.2f} naps")
    r2_naps = r2_score(y_test_naps, y_pred_naps)
    print(f"R^2 for Naps per Day: {r2_naps:.2f}")

    print('')
    print('---DURATION---')
    mae_duration = mean_absolute_error(predicted_data_nap['
        actual_duration'], predicted_data_nap['predicted_duration'])
    print(f"Mean Absolute Error (MAE) for Duration: {mae_duration
        :.2f} minutes")
    medae_duration = median_absolute_error(predicted_data_nap['
        actual_duration'], predicted_data_nap['predicted_duration'])
    print(f"Median Absolute Error (MedAE) for Duration: {
        medae_duration:.2f} minutes")
    r2_duration = r2_score(predicted_data_nap['actual_duration'],
        predicted_data_nap['predicted_duration'])
    print(f"R^2 for Duration: {r2_duration:.2f}")

    print('')
    print('---START_TIME---')
    mae_start = (mean_absolute_error(predicted_data['
        actual_start_24hr'], predicted_data['predicted_start_24hr'])
```

```python
)*60
    print(f"Mean Absolute Error (MAE) for Start Time: {mae_start:.2f} minutes")
    medae_start = (median_absolute_error(predicted_data['actual_start_24hr'], predicted_data['predicted_start_24hr']))*60
    print(f"Median Absolute Error (MedAE) for Start Time: {medae_start:.2f} minutes")
    r2_start = r2_score(predicted_data['actual_start_24hr'], predicted_data['predicted_start_24hr'])
    print(f"R^2 for Start Time: {r2_start:.2f}")
    new_row = pd.DataFrame({
        'mae_naps': [mae_naps],
        'accuracy_naps': [accuracy_naps],
        'medae_naps': [medae_naps],
        'r2_naps': [r2_naps],
        'mae_duration': [mae_duration],
        'medae_duration': [medae_duration],
        'r2_duration': [r2_duration],
        'mae_start': [mae_start],
        'medae_start': [medae_start],
        'r2_start': [r2_start]
    })
    evaluation_metrics = pd.concat([evaluation_metrics, new_row], ignore_index=True)

    return evaluation_metrics

def perform_evaluation(training_data_naps, testing_data_naps, training_data, testing_data, label_encoder, category_encoder, models_path, results_path, model_info):
    """
    Perform evaluation of machine learning models.
    """
    print('    - Naps starting')
    X_test_naps, y_pred_naps, y_test_naps, history_naps =
       predict_naps(training_data_naps, testing_data_naps,
       model_info, models_path)
    print('    - Duration and start time starting')
    X_test, y_pred, y_test, history = predict_log(training_data,
       testing_data, model_info, models_path)
    # Make predictions
    predicted_data, y_pred_naps, y_test_naps = predicted_outcomes(
       X_test, y_pred, y_test, X_test_naps, y_pred_naps,
       y_test_naps, label_encoder, category_encoder)
    # PLOTS
    save_plots(predicted_data, results_path, history_naps, history,
        model_info)

    return predicted_data, y_pred_naps, y_test_naps

def save_results(predicted_data, evaluation_metrics, model_info,
    results_path):
    """
    Save the results of the evaluation.
    """
    model_type = model_info['model_type']
    if model_type == 'XGBoost':
        param_1 = int(model_info['num_estimators'])
        param_1_txt = 'estimators'
        param_2 = float(model_info['learning_rate'])
        param_2_txt = 'learning_rate'
        param_3 = model_info['objective']
        param_3_txt = 'objective'
    elif model_type == 'LSTM' or model_type == 'RNN':
        param_1 = int(model_info['num_epoch'])
        param_1_txt = 'epochs'
        param_2 = int(model_info['batch_size'])
        param_2_txt = 'batch_size'
        param_3 = model_info['loss']
        param_3_txt = 'loss'
    elif model_type == 'KNN':
        param_1 = int(model_info['num_neighbors'])
        param_1_txt = 'neighbors'
        param_2 = ''
        param_2_txt = ''
        param_3 = ''
        param_3_txt = ''
    elif model_type == 'RF':
        param_1 = model_info['max_depth']
        param_1_txt = 'max_depth'
        param_2 = int(model_info['n_estimators'])
        param_2_txt = 'n_estimators'
        param_3 = int(model_info['bootstrap'])
        param_3_txt = 'bootstrap'
    results_csv_path = os.path.join(results_path, f'{model_type}_{param_1}{param_1_txt}_{param_2}{param_2_txt}_{param_3}{param_3_txt}_results_all_logs_2024-04-15.csv')
    predicted_data.to_csv(results_csv_path)
    metric_csv_path = os.path.join(results_path, f'{model_type}_{param_1}{param_1_txt}_{param_2}{param_2_txt}_{param_3}{param_3_txt}_metrics_all_logs_2024-04-15.csv')
    evaluation_metrics.to_csv(metric_csv_path)
```

B.5 MODELS

The provided code includes functions for predicting nap patterns and sleep logs using various machine learning models like XGBoost, LSTM, RNN, KNN, and Random

Forest. It incorporates the use of libraries such as joblib for model serialization, numpy for numerical operations, scikit-learn for traditional ML models, and keras for deep learning models. Each prediction function supports model retraining and loading pre-trained models, scaling input data, and saving the trained models and scalers. The code handles the training and prediction processes for both nap counts and sleep logs, ensuring the data is appropriately processed and models are properly managed.

```
import joblib
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsRegressor
from keras.layers import LSTM, Dense, SimpleRNN
from keras.layers import Dropout, Bidirectional
from tensorflow.keras.models import load_model
from sklearn.neighbors import KNeighborsRegressor
import xgboost as xgb
import os

def predict_naps(training_data_naps, testing_data_nap, model_info,
    models_path):
    model_type = model_info['model_type']
    if model_type == 'XGBoost':
        X_test, y_pred, y_test, history = predict_naps_xgboost(
            training_data_naps, testing_data_nap, model_info,
            models_path)
    elif model_type == 'LSTM':
        X_test, y_pred, y_test, history = predict_naps_lstm(
            training_data_naps, testing_data_nap, model_info,
            models_path)
    elif model_type == 'RNN':
        X_test, y_pred, y_test, history = predict_naps_rnn(
            training_data_naps, testing_data_nap, model_info,
            models_path)
    elif model_type == 'KNN':
        X_test, y_pred, y_test, history = predict_naps_knn(
            training_data_naps, testing_data_nap, model_info,
            models_path)
    elif model_type == 'RF':
        X_test, y_pred, y_test, history = predict_naps_rf(
            training_data_naps, testing_data_nap, model_info,
            models_path)

    return X_test, y_pred, y_test, history

def predict_log(training_data, testing_data, model_info,
   models_path):
    model_type = model_info['model_type']
    if model_type == 'XGBoost':
        X_test, y_pred, y_test, history = predict_log_xgboost(
            training_data, testing_data, model_info, models_path)
    elif model_type == 'LSTM':
        X_test, y_pred, y_test, history = predict_log_lstm(
            training_data, testing_data, model_info, models_path)
    elif model_type == 'RNN':
        X_test, y_pred, y_test, history = predict_log_rnn(
            training_data, testing_data, model_info, models_path)
    elif model_type == 'KNN':
        X_test, y_pred, y_test, history = predict_log_knn(
            training_data, testing_data, model_info, models_path)
```

```python
    elif model_type == 'RF':
        X_test, y_pred, y_test, history = predict_log_rf(training_data, testing_data, model_info, models_path)

    return X_test, y_pred, y_test, history

def predict_naps_xgboost(training_data_naps, testing_data_naps, model_info, models_path):
    retrain = model_info['retrain']
    model_type = model_info['model_type']
    n_estimators = int(model_info['num_estimators'])
    learning_rate = float(model_info['learning_rate'])
    objective = model_info['objective']
    model_path = os.path.join(models_path, f'{model_type}_model_naps_{n_estimators}_{learning_rate}_{objective}_sample_filtered.pkl')
    scaler_path = os.path.join(models_path, f'{model_type}_scaler_naps_{n_estimators}_{learning_rate}_{objective}_sample_filtered.joblib')
    X_train = training_data_naps.drop(['naps_that_day', 'num_nap', 'category', 'age_on_log_dec'], axis=1)
    X_test = testing_data_naps.drop(['naps_that_day', 'num_nap', 'category', 'age_on_log_dec'], axis=1)
    y_train = training_data_naps['naps_that_day']
    y_test = testing_data_naps['naps_that_day']
    X_train_processed = X_train.drop('date', axis=1)
    X_test_processed = X_test.drop('date', axis=1)
    print('    - Columns passed for training number of naps:', X_train_processed.columns.tolist())
    if retrain == True:
        if not os.path.exists(model_path):
            print('Model NOT FOUND, training from scratch...')
            retrain = False
        else:
            xgb_naps = joblib.load(model_path)
            scaler = joblib.load(scaler_path)
            # Scale features
        scaler.partial_fit(X_train_processed)
            X_train_scaled = scaler.transform(X_train_processed)
            X_test_scaled = scaler.transform(X_test_processed)
            # Training the model for 'naps_that_day'
            history = xgb_naps.fit(X_train_scaled, y_train)
            # Predicting for 'naps_that_day'
            y_pred = xgb_naps.predict(X_test_scaled)
    if retrain == False:
        # Scale features
        scaler = StandardScaler()
        X_train_scaled = scaler.fit_transform(X_train_processed)
        X_test_scaled = scaler.transform(X_test_processed)
        # Building and training the model for 'naps_that_day'
        xgb_naps = xgb.XGBRegressor(objective=objective,
           n_estimators=n_estimators, learning_rate=learning_rate,
           random_state=42)
        history = xgb_naps.fit(X_train_scaled, y_train)
        # Predicting for 'naps_that_day'
        y_pred = xgb_naps.predict(X_test_scaled)
    # Rounding 'naps_that_day' predictions
    y_pred = np.round(y_pred, 0)
    joblib.dump(xgb_naps, model_path)
    joblib.dump(scaler, scaler_path)

    return X_test, y_pred, y_test, history

def predict_log_xgboost(training_data, testing_data, model_info,
    models_path):
    retrain = model_info['retrain']
```

```python
    model_type = model_info['model_type']
    n_estimators = int(model_info['num_estimators'])
    learning_rate = float(model_info['learning_rate'])
    objective = model_info['objective']
    model_path = os.path.join(models_path, f'{model_type}_model_double_{n_estimators}_{learning_rate}_{objective}_sample_filtered.pkl')
    scaler_path = os.path.join(models_path, f'{model_type}_scaler_double_{n_estimators}_{learning_rate}_{objective}_sample_filtered.joblib')
    X_train = training_data.drop(['duration', 'start_24hr', 'naps_that_day'], axis=1)
    X_test = testing_data.drop(['duration', 'start_24hr', 'naps_that_day'], axis=1)
    # y_train = training_data[['duration', 'start_24hr']]
    y_train_dur = training_data['duration']
    y_train_start = training_data['start_24hr']
    y_test = testing_data[['duration', 'start_24hr']]
    X_train_processed = X_train.drop(['date', 'start', 'age_on_log_dec', 'timezone_offset_minutes', 'end'], axis=1)
    X_test_processed = X_test.drop(['date', 'start', 'age_on_log_dec', 'timezone_offset_minutes', 'end'], axis=1)
    print('    - Columns passed for training duration and start time:',  X_train_processed.columns.tolist())
    if retrain == True:
        if not os.path.exists(model_path):
            print('Model NOT FOUND, training from scratch...')
            retrain = False
        else:
            xgb_double = joblib.load(model_path)
            scaler = joblib.load(scaler_path)
            # Scale features
        scaler.partial_fit(X_train_processed)
```

```python
            X_train_scaled = scaler.transform(X_train_processed)
            X_test_scaled = scaler.transform(X_test_processed)
            # Training the model for 'duration' and 'start_24hr'
            history = xgb_double.fit(X_train_scaled, np.column_stack((y_train_dur, y_train_start)))
            # Predicting for 'duration' and 'start_24hr'
            y_duration_pred, y_start_pred = xgb_double.predict(X_test_scaled).T
            y_pred = np.column_stack((y_duration_pred, y_start_pred))
    if retrain == False:
        # Scale features
        scaler = StandardScaler()
        X_train_scaled = scaler.fit_transform(X_train_processed)
        X_test_scaled = scaler.transform(X_test_processed)
        # Building and training the model for 'duration' and 'start_24hr'
        xgb_double = xgb.XGBRegressor(objective=objective, n_estimators=n_estimators, learning_rate=learning_rate, random_state=42)
        history = xgb_double.fit(X_train_scaled, np.column_stack((y_train_dur, y_train_start)))
        # Predicting for 'duration' and 'start_24hr'
        y_duration_pred, y_start_pred = xgb_double.predict(X_test_scaled).T
        y_pred = np.column_stack((y_duration_pred, y_start_pred))
    joblib.dump(xgb_double, model_path)
    joblib.dump(scaler, scaler_path)

    return X_test, y_pred, y_test, history

def predict_naps_knn(training_data_naps, testing_data_naps, model_info, models_path):
    n_neighbors = int(model_info['num_neighbors'])
    model_path = os.path.join(models_path, f'KNN_model_naps_{n_neighbors}_sample_filtered.pkl')
    X_train = training_data_naps.drop(['naps_that_day', 'num_nap', 'category', 'age_on_log_dec'], axis=1)
    X_test = testing_data_naps.drop(['naps_that_day', 'num_nap', 'category', 'age_on_log_dec'], axis=1)
    y_train = training_data_naps['naps_that_day']
    y_test = testing_data_naps['naps_that_day']
    X_train_processed = X_train.drop('date', axis=1)
    X_test_processed = X_test.drop('date', axis=1)
    if not os.path.exists(model_path):
        print('Model NOT FOUND, training from scratch...')
        knn_naps = KNeighborsRegressor(n_neighbors=n_neighbors)
        knn_naps.fit(X_train_processed, y_train)
    else:
        knn_naps = joblib.load(model_path)
    y_pred = knn_naps.predict(X_test_processed)
    y_pred = np.round(y_pred, 0)
    joblib.dump(knn_naps, model_path)
    return X_test, y_pred, y_test, None  # No history for KNN

def predict_log_knn(training_data, testing_data, model_info, models_path):
    n_neighbors = int(model_info['num_neighbors'])
    model_path = os.path.join(models_path, f'KNN_model_log_{n_neighbors}_sample_filtered.pkl')
    X_train = training_data.drop(['duration', 'start_24hr', 'naps_that_day'], axis=1)
    X_test = testing_data.drop(['duration', 'start_24hr', 'naps_that_day'], axis=1)
    # y_train = training_data[['duration', 'start_24hr']]
    y_train_dur = training_data['duration']
```

```python
    y_train_start = training_data['start_24hr']
    y_test = testing_data[['duration', 'start_24hr']]
    X_train_processed = X_train.drop(['date', 'start', 'age_on_log_dec', 'timezone_offset_minutes', 'end'], axis=1)
    X_test_processed = X_test.drop(['date', 'start', 'age_on_log_dec', 'timezone_offset_minutes', 'end'], axis=1)
    if not os.path.exists(model_path):
        print('Model NOT FOUND, training from scratch...')
        knn_log = KNeighborsRegressor(n_neighbors=n_neighbors)
        knn_log.fit(X_train_processed, np.column_stack((y_train_dur, y_train_start)))
    else:
        knn_log = joblib.load(model_path)
    y_duration_pred, y_start_pred = knn_log.predict(X_test_processed).T
    y_pred = np.column_stack((y_duration_pred, y_start_pred))
    joblib.dump(knn_log, model_path)

    return X_test, y_pred, y_test, None  # No history for KNN

def predict_naps_rf(training_data_naps, testing_data_naps, model_info, models_path):
    retrain = model_info['retrain']
    max_depth = model_info['max_depth']
    bootstrap = model_info['bootstrap']
    model_type = model_info['model_type']
    n_estimators = int(model_info['num_estimators'])
    model_path = os.path.join(models_path, f'{model_type}_model_naps_{n_estimators}_sample_filtered.pkl')
    scaler_path = os.path.join(models_path, f'{model_type}_scaler_naps_{n_estimators}_sample_filtered.joblib')
    X_train = training_data_naps.drop(['naps_that_day', 'num_nap', 'category', 'age_on_log_dec'], axis=1)
    X_test = testing_data_naps.drop(['naps_that_day', 'num_nap', '
        category', 'age_on_log_dec'], axis=1)
    y_train = training_data_naps['naps_that_day']
    y_test = testing_data_naps['naps_that_day']
    X_train_processed = X_train.drop('date', axis=1)
    X_test_processed = X_test.drop('date', axis=1)
    print('     - Columns passed for training number of naps:',
        X_train_processed.columns.tolist())
    if retrain:
        if not os.path.exists(model_path):
            print('Model NOT FOUND, training from scratch...')
            retrain = False
        else:
            rf_naps = joblib.load(model_path)
            scaler = joblib.load(scaler_path)
            # Scale features
        scaler.partial_fit(X_train_processed)
            X_train_scaled = scaler.transform(X_train_processed)
            X_test_scaled = scaler.transform(X_test_processed)
            # Training the model for 'naps_that_day'
            history = rf_naps.fit(X_train_scaled, y_train)
            # Predicting for 'naps_that_day'
            y_pred = rf_naps.predict(X_test_scaled)
    if not retrain:
        # Scale features
        scaler = StandardScaler()
        X_train_scaled = scaler.fit_transform(X_train_processed)
        X_test_scaled = scaler.transform(X_test_processed)
        # Building and training the model for 'naps_that_day'
        rf_naps = RandomForestRegressor(n_estimators=n_estimators,
            bootstrap=bootstrap, max_depth=max_depth, random_state
            =42)
        history = rf_naps.fit(X_train_scaled, y_train)
```

```python
        # Predicting for 'naps_that_day'
        y_pred = rf_naps.predict(X_test_scaled)
    # Rounding 'naps_that_day' predictions
    y_pred = np.round(y_pred, 0)
    joblib.dump(rf_naps, model_path)
    joblib.dump(scaler, scaler_path)

    return X_test, y_pred, y_test, history

def predict_log_rf(training_data, testing_data, model_info, models_path):
    retrain = model_info['retrain']
    max_depth = model_info['max_depth']
    bootstrap = model_info['bootstrap']
    model_type = model_info['model_type']
    n_estimators = int(model_info['num_estimators'])
    model_path = os.path.join(models_path, f'{model_type}_model_double_{n_estimators}_sample_filtered.pkl')
    scaler_path = os.path.join(models_path, f'{model_type}_scaler_double_{n_estimators}_sample_filtered.joblib')
    X_train = training_data.drop(['duration', 'start_24hr', 'naps_that_day'], axis=1)
    X_test = testing_data.drop(['duration', 'start_24hr', 'naps_that_day'], axis=1)
    y_train_dur = training_data['duration']
    y_train_start = training_data['start_24hr']
    y_test = testing_data[['duration', 'start_24hr']]
    X_train_processed = X_train.drop(['date', 'start', 'age_on_log_dec', 'timezone_offset_minutes', 'end'], axis=1)
    X_test_processed = X_test.drop(['date', 'start', 'age_on_log_dec', 'timezone_offset_minutes', 'end'], axis=1)
    print('    - Columns passed for training duration and start time:',  X_train_processed.columns.tolist())
    if retrain:
        if not os.path.exists(model_path):
            print('Model NOT FOUND, training from scratch...')
            retrain = False
        else:
            rf_double = joblib.load(model_path)
            scaler = joblib.load(scaler_path)
            # Scale features
        scaler.partial_fit(X_train_processed)
            X_train_scaled = scaler.transform(X_train_processed)
            X_test_scaled = scaler.transform(X_test_processed)
            # Training the model for 'duration' and 'start_24hr'
            history = rf_double.fit(X_train_scaled, np.column_stack
                ((y_train_dur, y_train_start)))
            # Predicting for 'duration' and 'start_24hr'
            y_duration_pred, y_start_pred = rf_double.predict(
                X_test_scaled).T
            y_pred = np.column_stack((y_duration_pred, y_start_pred
                ))
    if not retrain:
        # Scale features
        scaler = StandardScaler()
        X_train_scaled = scaler.fit_transform(X_train_processed)
        X_test_scaled = scaler.transform(X_test_processed)
        # Building and training the model for 'duration' and '
            start_24hr'
        rf_double = RandomForestRegressor(n_estimators=n_estimators
            , bootstrap=bootstrap, max_depth=max_depth, random_state
            =42)
        history = rf_double.fit(X_train_scaled, np.column_stack((
            y_train_dur, y_train_start)))
        # Predicting for 'duration' and 'start_24hr'
        y_duration_pred, y_start_pred = rf_double.predict(
            X_test_scaled).T
        y_pred = np.column_stack((y_duration_pred, y_start_pred))
    joblib.dump(rf_double, model_path)
    joblib.dump(scaler, scaler_path)

    return X_test, y_pred, y_test, history

def predict_naps_lstm(training_data_naps, testing_data_naps,
   model_info, models_path):
    retrain = model_info['retrain']
    model_type = model_info['model_type']
    n_epoch = int(model_info['num_epoch'])
    n_batch = int(model_info['batch_size'])
    loss = model_info['loss']
    model_path = os.path.join(models_path, f'{model_type}
       _model_naps_{n_epoch}_{n_batch}_{loss}_sample_filtered.keras
       ')
    scaler_path = os.path.join(models_path, f'{model_type}
       _scaler_naps_{n_epoch}_{n_batch}_{loss}_sample_filtered.
       joblib')
    X_train = training_data_naps.drop(['naps_that_day', 'num_nap',
       'category', 'age_on_log_dec'], axis=1)
    X_test = testing_data_naps.drop(['naps_that_day', 'num_nap', '
       category', 'age_on_log_dec'], axis=1)
    y_train = training_data_naps['naps_that_day']
    y_test = testing_data_naps['naps_that_day']
    X_train_processed = X_train.drop('date', axis=1)
    X_test_processed = X_test.drop('date', axis=1)
    print('    - Columns passed for training number of naps:',
       X_train_processed.columns.tolist())
    if retrain == True:
        if not os.path.exists(model_path):
            print('Model NOT FOUND, training from scratch...')
```

```python
            retrain = False
        else:
            model_naps = load_model(model_path)
            scaler = joblib.load(scaler_path)
            # Scale features
        scaler.partial_fit(X_train_processed)
            X_train_scaled = scaler.transform(X_train_processed)
            X_test_scaled = scaler.transform(X_test_processed)
            # Reshape for LSTM
            X_train_reshaped = np.reshape(X_train_scaled, (X_train_scaled.shape[0], 1, X_train_scaled.shape[1]))
            X_test_reshaped = np.reshape(X_test_scaled, (X_test_scaled.shape[0], 1, X_test_scaled.shape[1]))
            # Training the model for 'naps_that_day'
            history = model_naps.fit(X_train_reshaped, y_train)
            # Predicting for 'naps_that_day'
            y_pred = model_naps.predict(X_test_reshaped)
    if retrain == False:
        # Scale features
        scaler = MinMaxScaler()
        X_train_scaled = scaler.fit_transform(X_train_processed)
        X_test_scaled = scaler.transform(X_test_processed)
        # Reshape for LSTM
        X_train_reshaped = np.reshape(X_train_scaled, (X_train_scaled.shape[0], 1, X_train_scaled.shape[1]))
        X_test_reshaped = np.reshape(X_test_scaled, (X_test_scaled.shape[0], 1, X_test_scaled.shape[1]))
        # Building and training the model for 'naps_that_day'
        model_naps = Sequential()
        model_naps.add(LSTM(units=100, input_shape=(X_train_reshaped.shape[1], X_train_reshaped.shape[2])))
        model_naps.add(Dense(units=1, activation='relu'))
        model_naps.compile(optimizer='adam', loss=loss)
        history = model_naps.fit(X_train_reshaped, y_train, epochs=n_epoch, batch_size=n_batch, validation_split=0.2, shuffle=False, verbose=1)
        # Predicting for 'naps_that_day'
        y_pred = model_naps.predict(X_test_reshaped)
    # Rounding 'naps_that_day' predictions
    y_pred = np.round(y_pred)
    model_naps.save(model_path, include_optimizer=False)
    joblib.dump(scaler, scaler_path)

    return X_test, y_pred, y_test, history

def predict_log_lstm(training_data, testing_data, model_info, models_path):
    retrain = model_info['retrain']
    model_type = model_info['model_type']
    n_epoch = int(model_info['num_epoch'])
    n_batch = int(model_info['batch_size'])
    loss = model_info['loss']
    model_path = os.path.join(models_path, f'{model_type}_model_double_{n_epoch}_{n_batch}_{loss}_sample_filtered.keras')
    scaler_path = os.path.join(models_path, f'{model_type}_scaler_double_{n_epoch}_{n_batch}_{loss}_sample_filtered.joblib')
    X_train = training_data.drop(['duration', 'start_24hr', 'naps_that_day'], axis=1)
    X_test = testing_data.drop(['duration', 'start_24hr', 'naps_that_day'], axis=1)
    y_train = training_data[['duration', 'start_24hr']]
    y_test = testing_data[['duration', 'start_24hr']]
    X_train_processed = X_train.drop(['date', 'start', '
```

```python
age_on_log_dec', 'timezone_offset_minutes', 'end'], axis=1)
    X_test_processed = X_test.drop(['date', 'start', 'age_on_log_dec', 'timezone_offset_minutes', 'end'], axis=1)
    print('    - Columns passed for training duration and start time:', X_train_processed.columns.tolist())
    if retrain == True:
        if not os.path.exists(model_path):
            print('Model NOT FOUND, training from scratch...')
            retrain = False
        else:
            model = load_model(model_path)
            scaler = joblib.load(scaler_path)
            # Scale features
        scaler.partial_fit(X_train_processed)
            X_train_scaled = scaler.transform(X_train_processed)
            X_test_scaled = scaler.transform(X_test_processed)
            # Reshape for LSTM
            X_train_reshaped = np.reshape(X_train_scaled, (X_train_scaled.shape[0], 1, X_train_scaled.shape[1]))
            X_test_reshaped = np.reshape(X_test_scaled, (X_test_scaled.shape[0], 1, X_test_scaled.shape[1]))
            # Training the model for 'duration' and 'start_24hr'
            history = model.fit(X_train_reshaped, y_train)
            # Predicting for 'duration' and 'start_24hr'
            y_pred = model.predict(X_test_reshaped)
    if retrain == False:
        # Scale features
        scaler = MinMaxScaler()
        X_train_scaled = scaler.fit_transform(X_train_processed)
        X_test_scaled = scaler.transform(X_test_processed)
        # Reshape for LSTM
        X_train_reshaped = np.reshape(X_train_scaled, (
            X_train_scaled.shape[0], 1, X_train_scaled.shape[1]))
        X_test_reshaped = np.reshape(X_test_scaled, (X_test_scaled.shape[0], 1, X_test_scaled.shape[1]))
        # Building and training the model for 'duration' and 'start_24hr'
        model = Sequential()
        model.add(LSTM(units=100, input_shape=(X_train_reshaped.shape[1], X_train_reshaped.shape[2])))
        model.add(Dense(units=2, activation='relu'))
        model.compile(optimizer='adam', loss=loss)
        history = model.fit(X_train_reshaped, y_train, epochs=n_epoch, batch_size=n_batch, validation_split=0.2, shuffle=False, verbose=1)
        # Predicting for 'duration' and 'start_24hr'
        y_pred = model.predict(X_test_reshaped)
    model.save(model_path, include_optimizer=False)
    joblib.dump(scaler, scaler_path)

    return X_test, y_pred, y_test, history


def predict_naps_rnn(training_data_naps, testing_data_naps, model_info, models_path):
    retrain = model_info['retrain']
    model_type = model_info['model_type']
    n_epoch = int(model_info['num_epoch'])
    n_batch = int(model_info['batch_size'])
    loss = model_info['loss']
    model_path = os.path.join(models_path, f'{model_type}_model_naps_{n_epoch}_{n_batch}_{loss}_sample_filtered.keras')
    scaler_path = os.path.join(models_path, f'{model_type}_scaler_naps_{n_epoch}_{n_batch}_{loss}_sample_filtered.
        joblib')
    X_train = training_data_naps.drop(['naps_that_day', 'num_nap',
        'category', 'age_on_log_dec'], axis=1)
    X_test = testing_data_naps.drop(['naps_that_day', 'num_nap', '
        category', 'age_on_log_dec'], axis=1)
    y_train = training_data_naps['naps_that_day']
    y_test = testing_data_naps['naps_that_day']
    X_train_processed = X_train.drop('date', axis=1)
    X_test_processed = X_test.drop('date', axis=1)
    print('    - Columns passed for training number of naps:',
        X_train_processed.columns.tolist())
    if retrain == True:
        if not os.path.exists(model_path):
            print('Model NOT FOUND, training from scratch...')
            retrain = False
        else:
            model_naps = load_model(model_path)
            scaler = joblib.load(scaler_path)
            # Scale features
        scaler.partial_fit(X_train_processed)
            X_train_scaled = scaler.transform(X_train_processed)
            X_test_scaled = scaler.transform(X_test_processed)
            # Reshape for RNN
            X_train_reshaped = np.reshape(X_train_scaled, (
                X_train_scaled.shape[0], 1, X_train_scaled.shape[1])
                )
            X_test_reshaped = np.reshape(X_test_scaled, (
                X_test_scaled.shape[0], 1, X_test_scaled.shape[1]))
            # Training the model for 'naps_that_day'
            history = model_naps.fit(X_train_reshaped, y_train)
            # Predicting for 'naps_that_day'
            y_pred = model_naps.predict(X_test_reshaped)
    if retrain == False:
        # Scale features
        scaler = MinMaxScaler()
        X_train_scaled = scaler.fit_transform(X_train_processed)
        X_test_scaled = scaler.transform(X_test_processed)
        # Reshape for RNN
        X_train_reshaped = np.reshape(X_train_scaled, (
            X_train_scaled.shape[0], 1, X_train_scaled.shape[1]))
        X_test_reshaped = np.reshape(X_test_scaled, (X_test_scaled.
            shape[0], 1, X_test_scaled.shape[1]))
        # Building and training the model for 'naps_that_day'
        model_naps = Sequential()
        model_naps.add(SimpleRNN(units=100, input_shape=(
            X_train_reshaped.shape[1], X_train_reshaped.shape[2])))
        model_naps.add(Dense(units=1, activation='relu'))
        model_naps.compile(optimizer='adam', loss=loss)
        history = model_naps.fit(X_train_reshaped, y_train, epochs=
            n_epoch, batch_size=n_batch, validation_split=0.2,
            shuffle=False, verbose=1)

        # Predicting for 'naps_that_day'
        y_pred = model_naps.predict(X_test_reshaped)
    # Rounding 'naps_that_day' predictions
    y_pred = np.round(y_pred)
    model_naps.save(model_path, include_optimizer=False)
    joblib.dump(scaler, scaler_path)
    return X_test, y_pred, y_test, history

def predict_log_rnn(training_data, testing_data, model_info,
    models_path):
    retrain = model_info['retrain']
    model_type = model_info['model_type']
    n_epoch = int(model_info['num_epoch'])
    n_batch = int(model_info['batch_size'])
    loss = model_info['loss']
    model_path = os.path.join(models_path, f'{model_type}
       _model_double_{n_epoch}_{n_batch}_{loss}_sample_filtered.
       keras')
    scaler_path = os.path.join(models_path, f'{model_type}
       _scaler_double_{n_epoch}_{n_batch}_{loss}_sample_filtered.
       joblib')
    X_train = training_data.drop(['duration', 'start_24hr', '
       naps_that_day'], axis=1)
    X_test = testing_data.drop(['duration', 'start_24hr', '
       naps_that_day'], axis=1)
    y_train = training_data[['duration', 'start_24hr']]
    y_test = testing_data[['duration', 'start_24hr']]
    X_train_processed = X_train.drop(['date', 'start', '
       age_on_log_dec', 'timezone_offset_minutes', 'end'], axis=1)
    X_test_processed = X_test.drop(['date', 'start', '
       age_on_log_dec', 'timezone_offset_minutes', 'end'], axis=1)
    print('    - Columns passed for training duration and start
       time:',  X_train_processed.columns.tolist())
    if retrain == True:
        if not os.path.exists(model_path):
            print('Model NOT FOUND, training from scratch...')
            retrain = False
        else:
            model = load_model(model_path)
            scaler = joblib.load(scaler_path)
            # Scale features
         scaler.partial_fit(X_train_processed)
            X_train_scaled = scaler.transform(X_train_processed)
            X_test_scaled = scaler.transform(X_test_processed)
            # Reshape for RNN
            X_train_reshaped = np.reshape(X_train_scaled, (
               X_train_scaled.shape[0], 1, X_train_scaled.shape[1])
```

```
                )
            X_test_reshaped = np.reshape(X_test_scaled, (X_test_scaled.shape[0], 1, X_test_scaled.shape[1]))
            # Training the model for 'duration' and 'start_24hr'
            history = model.fit(X_train_reshaped, y_train)
            # Predicting for 'duration' and 'start_24hr'
            y_pred = model.predict(X_test_reshaped)
    if retrain == False:
        # Scale features
        scaler = MinMaxScaler()
        X_train_scaled = scaler.fit_transform(X_train_processed)
        X_test_scaled = scaler.transform(X_test_processed)
        # Reshape for RNN
        X_train_reshaped = np.reshape(X_train_scaled, (X_train_scaled.shape[0], 1, X_train_scaled.shape[1]))
        X_test_reshaped = np.reshape(X_test_scaled, (X_test_scaled.shape[0], 1, X_test_scaled.shape[1]))
        # Building and training the model for 'duration' and 'start_24hr'
        model = Sequential()
        model.add(SimpleRNN(units=100, input_shape=(X_train_reshaped.shape[1], X_train_reshaped.shape[2])))
        model.add(Dense(units=2, activation='relu'))
        model.compile(optimizer='adam', loss=loss)
        history = model.fit(X_train_reshaped, y_train, epochs=n_epoch, batch_size=n_batch, validation_split=0.2, shuffle=False, verbose=1)
        # Predicting for 'duration' and 'start_24hr'
        y_pred = model.predict(X_test_reshaped)
    model.save(model_path, include_optimizer=False)
    joblib.dump(scaler, scaler_path)

    return X_test, y_pred, y_test, history
```

B.6 PLOTS

This script generates and saves various plots to visualize the results of different machine learning models used for predicting nap-related data. It includes a main function, save_plots, that selects specific plots to create based on the model type (XGBoost, KNN, LSTM, RNN, RF). The script contains helper functions to generate scatter plots for actual vs. predicted values, residual plots to show prediction errors, and learning curves to illustrate the model's training and validation performance over epochs. These plots are saved as PDF files in the specified directory, helping in the assessment and comparison of model performance.

```python
import os
import matplotlib.pyplot as plt
import numpy as np

def save_plots(predicted_data, results_path, history_naps, history,
    model_info):
    model_type = model_info['model_type']
    if model_type == 'XGBoost':
        params = {
            'model_type': 'XGBoost',
            'param_1': int(model_info['num_estimators']),
            'param_1_txt': 'estimators',
            'param_2': float(model_info['learning_rate']),
            'param_2_txt': 'learning_rate',
            'param_3': model_info['objective'],
            'param_3_txt': 'objective'
        }
        save_plot_actual_vs_predicted(predicted_data, results_path, params)
        save_plot_residuals(predicted_data, results_path, params)
    elif model_type == 'KNN':
        params = {
            'model_type': 'KNN',
            'param_1': int(model_info['num_neighbors']),
            'param_1_txt': 'neighbors',
            'param_2': '',
            'param_2_txt': '',
            'param_3': '',
            'param_3_txt': ''
        }
        save_plot_actual_vs_predicted(predicted_data, results_path, params)
        save_plot_residuals(predicted_data, results_path, params)
    elif model_type == 'LSTM':
        params = {
            'model_type': 'LSTM',
            'param_1': int(model_info['num_epoch']),
            'param_1_txt': 'epochs',
            'param_2': int(model_info['batch_size']),
            'param_2_txt': 'batch_size',
            'param_3': model_info['loss'],
            'param_3_txt': 'loss'
        }
        save_plot_actual_vs_predicted(predicted_data, results_path, params)
        save_plot_residuals(predicted_data, results_path, params)
        save_plot_learning_curve(history_naps, history, results_path, params)
    elif model_type == 'RNN':
        params = {
```

```python
            'model_type': 'RNN',
            'param_1': int(model_info['num_epoch']),
            'param_1_txt': 'epochs',
            'param_2': int(model_info['batch_size']),
            'param_2_txt': 'batch_size',
            'param_3': model_info['loss'],
            'param_3_txt': 'loss'
        }
        save_plot_actual_vs_predicted(predicted_data, results_path, params)
        save_plot_residuals(predicted_data, results_path, params)
        save_plot_learning_curve(history_naps, history, results_path, params)
    elif model_type == 'RF':
        params = {
            'model_type': 'RF',
            'param_1' : int(model_info['max_depth']),
            'param_1_txt': 'max_depth',
            'param_2': int(model_info['n_estimators']),
            'param_2_txt': 'n_estimators',
            'param_3': model_info['bootstrap'],
            'param_3_txt': 'bootstrap'
        }
        save_plot_actual_vs_predicted(predicted_data, results_path, params)
        save_plot_residuals(predicted_data, results_path, params)

def save_plot_actual_vs_predicted(predicted_data, save_path, params):
    model_type = params['model_type']
    param_1 = params['param_1']
    param_1_txt = params['param_1_txt']
    param_2 = params['param_2']
    param_2_txt = params['param_2_txt']
    param_3 = params['param_3']
    param_3_txt = params['param_3_txt']
    plt.figure(figsize=(8, 6))
    plt.scatter(predicted_data['actual_duration'], predicted_data['predicted_duration'], alpha=0.5)
    ax = plt.gca()
    ax.plot(plt.xlim(), plt.xlim(), color='red', linestyle='--')
    plt.xlabel('Actual Duration')
    plt.ylabel('Predicted Duration')
    plt.title(f'Actual vs. Predicted Values for Duration \n {model_type} Model with {param_1}{param_1_txt}, {param_2}{param_2_txt} and {param_3}{param_3_txt}')
    plt.grid(True)
    plt.savefig(os.path.join(save_path, f'{model_type}_{param_1}{param_1_txt}_{param_2}{param_2_txt}_{param_3}{param_3_txt}_actual_vs_predicted_variable_duration.pdf'))
    plt.close()
    plt.figure(figsize=(8, 6))
    plt.scatter(predicted_data['actual_start_24hr'], predicted_data['predicted_start_24hr'], alpha=0.5)
    ax = plt.gca()
    ax.plot(plt.xlim(), plt.xlim(), color='red', linestyle='--')
    plt.xlabel('Actual Start Time')
    plt.ylabel('Predicted Start Time')
    plt.title(f'Actual vs. Predicted Values for Start Time \n {model_type} Model with {param_1}{param_1_txt}, {param_2}{param_2_txt} and {param_3}{param_3_txt}')
    plt.grid(True)
    plt.savefig(os.path.join(save_path, f'{model_type}_{param_1}{param_1_txt}_{param_2}{param_2_txt}_{param_3}{param_3_txt}_actual_vs_predicted_variable_start_time.pdf'))
    plt.close()
    plt.figure(figsize=(8, 6))
    plt.scatter(predicted_data['actual_naps_per_day'], predicted_data['predicted_naps_per_day'], alpha=0.5)
    ax = plt.gca()
    ax.plot(plt.xlim(), plt.xlim(), color='red', linestyle='--')
    plt.xlabel('Actual Number of Naps')
    plt.ylabel('Predicted Number of Naps')
    plt.title(f'Actual vs. Predicted Values for Number of Naps \n {model_type} Model with {param_1}{param_1_txt}, {param_2}{param_2_txt} and {param_3}{param_3_txt}')
    plt.grid(True)
    plt.savefig(os.path.join(save_path, f'{model_type}_{param_1}{param_1_txt}_{param_2}{param_2_txt}_{param_3}{param_3_txt}_actual_vs_predicted_number_of_naps.pdf'))
    plt.close()

def save_plot_residuals(predicted_data, save_path, params):
    model_type = params['model_type']
    param_1 = params['param_1']
    param_1_txt = params['param_1_txt']
    param_2 = params['param_2']
    param_2_txt = params['param_2_txt']
    param_3 = params['param_3']
    param_3_txt = params['param_3_txt']
    plt.figure(figsize=(8, 6))
    residuals_duration = predicted_data['actual_duration'] - predicted_data['predicted_duration']
    plt.scatter(predicted_data['actual_duration'], residuals_duration, alpha=0.5)
    plt.axhline(y=0, color='red', linestyle='--')
    plt.xlabel('Actual Duration')
    plt.ylabel('Residuals')
    plt.title(f'Residual Plot for Duration \n {model_type} Model
        with {param_1}{param_1_txt}, {param_2}{param_2_txt} and {
        param_3}{param_3_txt}')
    plt.grid(True)
    plt.savefig(os.path.join(save_path, f'{model_type}_{param_1}{
        param_1_txt}_{param_2}{param_2_txt}_{param_3}{param_3_txt}
        _residual_plot_variable_duration.pdf'))
    plt.close()
    plt.figure(figsize=(8, 6))
    residuals_start_time = predicted_data['actual_start_24hr'] -
        predicted_data['predicted_start_24hr']
    plt.scatter(predicted_data['actual_start_24hr'],
        residuals_start_time, alpha=0.5)
    plt.axhline(y=0, color='red', linestyle='--')
    plt.xlabel('Actual Start Time')
    plt.ylabel('Residuals')
    plt.title(f'Residual Plot for Start Time \n {model_type} Model
        with {param_1}{param_1_txt}, {param_2}{param_2_txt} and {
        param_3}{param_3_txt}')
    plt.grid(True)
    plt.savefig(os.path.join(save_path, f'{model_type}_{param_1}{
        param_1_txt}_{param_2}{param_2_txt}_{param_3}{param_3_txt}
        _residual_plot_variable_start_time.pdf'))
    plt.close()
    plt.figure(figsize=(8, 6))
    residuals_naps = predicted_data['actual_naps_per_day'] -
        predicted_data['predicted_naps_per_day']
    plt.scatter(predicted_data['actual_naps_per_day'],
        residuals_naps, alpha=0.5)
    plt.axhline(y=0, color='red', linestyle='--')
    plt.xlabel('Actual Number of Naps')
    plt.ylabel('Residuals')
    plt.title(f'Residual Plot for Number of Naps \n {model_type}
        Model with {param_1}{param_1_txt}, {param_2}{param_2_txt}
        and {param_3}{param_3_txt}')
    plt.grid(True)
    plt.savefig(os.path.join(save_path, f'{model_type}_{param_1}{
        param_1_txt}_{param_2}{param_2_txt}_{param_3}{param_3_txt}
        _residual_plot_variable_number_of_naps.pdf'))
    plt.close()

def save_plot_learning_curve(history_naps, history, save_path,
    params, metric='Loss', metric_label='Loss'):
    model_type = params['model_type']
    param_1 = params['param_1']
    param_1_txt = params['param_1_txt']
    param_2 = params['param_2']
    param_2_txt = params['param_2_txt']
    param_3 = params['param_3']
    param_3_txt = params['param_3_txt']
    epochs = np.arange(1, len(history.history['loss']) + 1)
    plt.figure(figsize=(8, 6))
    plt.plot(epochs, history.history['loss'], label='Training ' +
        metric_label)
    if 'val_loss' in history.history:
        plt.plot(epochs, history.history['val_loss'], label='
            Validation ' + metric_label)
    plt.xlabel('Epochs')
    plt.ylabel(metric)
    plt.title(f'Learning Curve for duration and start time \n {
        model_type} Model with {param_1}{param_1_txt}, {param_2}{
        param_2_txt} and {param_3}{param_3_txt}')
    plt.legend()
    plt.grid(True)
    plt.savefig(os.path.join(save_path, f'{model_type}_{param_1}{
        param_1_txt}_{param_2}{param_2_txt}_{param_3}{param_3_txt}
        _learning_curve_duration_start.pdf'))
    plt.close()
    epochs = np.arange(1, len(history_naps.history['loss']) + 1)
    plt.figure(figsize=(8, 6))
    plt.plot(epochs, history_naps.history['loss'], label='Training
        ' + metric_label)
    if 'val_loss' in history_naps.history:
        plt.plot(epochs, history_naps.history['val_loss'], label='
            Validation ' + metric_label)
    plt.xlabel('Epochs')
    plt.ylabel(metric)
    plt.title(f'Learning Curve for Naps \n {model_type} Model with
        {param_1}{param_1_txt}, {param_2}{param_2_txt} and {param_3
        }{param_3_txt}')
    plt.legend()
    plt.grid(True)
    plt.savefig(os.path.join(save_path, f'{model_type}_{param_1}{
        param_1_txt}_{param_2}{param_2_txt}_{param_3}{param_3_txt}
        _learning_curve_naps.pdf'))
```