

folder/uio-fp-navn-eng.pdf

teleskopgutt2.jpeg

Master's thesis

A novel application of machine learning to develop pointing models for current and future radio/sub-millimeter telescopes

Bendik Nyheim

Computational Science: Physics
60 ECTS study points

Department of Physics
Faculty of Mathematics and Natural Sciences

Spring 2023

folder/uio-fp

Bendik Nyheim

A novel application of machine
learning to develop pointing
models for current and future
radio/sub-millimeter telescopes

Supervisors:

Signe Riemer Sørensen (Sintef)

Rodrigo Parrar (ESO)

Claudia Cicone (UiO)

Contents

1	Introduction	3
2	Related Works	4
3	Atronomical Background	5
4	Machine Learning Background	6
4.1	Supervised learning	6
4.2	Loss/Cost	6
4.2.1	Loss Functions	6
4.3	Train/test set	7
4.4	Scaling	7
4.5	Decision Trees	8
	Bagging	8
	Random Forest	9
	Boosting	9
	Gradient Boosting	9
4.6	Neural Networks	10
4.6.1	Backpropagation	11
4.6.2	Gradient Descent	12
4.6.3	Stochastic Gradient Descent	12
4.6.4	Momentum GPT	13
4.6.5	Adam GPT	13
4.6.6	Activation functions	14
	Tanh	14
	ReLU	14
	GeLU	14
4.7	Model Explainability	15
4.7.1	SHAP	15
4.7.2	SAGE	16
4.8	Mutual Information	16
5	Method	17
5.1	Cleaning pointing scan data	17
5.1.1	Cleaning criteria	18
5.1.2	Pointing scan classifier	18
	Method	18
	Results	19
5.2	Scan duration analysis	19
5.2.1	Analysis	19

5.2.2	Algorithm	20
5.2.3	Results	20
5.3	Feature Engineering	22
	Median values	22
	Sum of all change	22
	Change since the last correction	23
	Max change in time interval	23
	Position of the sun	23
5.3.1	List of features	23
5.4	Machine Learning Experiments	24
5.4.1	Experiment 1: Pointing Model using Neural Networks	24
	Feature Selection	24
	Model Architecture	24
	Loss Function and Model Evaluation	27
5.4.2	Experiment 2: Pointing Correction Model	27
	Feature Selection	28
	Model Architecture	28
	Model Evaluation	29
6	Results	30
6.1	Experiment 1: Pointing Model using Neural Networks	30
6.2	Experiment 2: Pointing Correction Model	30
7	Discussion	34
8	Conclusion	35
.1	Transformation of pointing offsets and corrections	36
	Bibliography	40

Chapter 1

Introduction

Chapter 2

Related Works

Chapter 3

Astronomical Background

Chapter 4

Machine Learning Background

4.1 Supervised learning

Supervised learning is a subfield of machine learning that refers to training a model to predict a specific target value based on input data. In this context, we refer to the input data as "features." The training is supervised when paired with the corresponding target value for prediction. There are two types of supervised learning, regression and classification. In regression, we predict a continuous variable, while in classification predict a binary value, true or false. The model architecture of the model can be the same regardless of predicting a true/false or continuous value. The difference is in the loss function, which is used to evaluate the model's performance during training. The last layer activation function for neural networks is different for regression and classification. In-depth explanations of this will come in the following sections.

4.2 Loss/Cost

In machine learning, the loss of a model refers to the discrepancy between the predicted and true values. It is calculated using a specific function designed to penalize incorrect predictions and measure the model's performance. The ultimate goal of any machine learning model is to minimize the loss and thereby reduce the difference between predicted and desired outputs. To achieve this, the model is trained by calculating the gradient of the loss function with respect to different components in the model. These gradients determine how the model is adjusted to minimize the loss through an iterative process. As a result, the model is optimized to make better predictions and achieve higher accuracy.

The most common loss function for regression is the mean squared error

$$\mathcal{L}(y, \tilde{y}) = \frac{1}{N} \sum_{i=1}^N (y - \tilde{y})^2, \quad (4.1)$$

where \tilde{y} is the prediction, y the true value, and N the number of predictions.

4.2.1 Loss Functions

Loss functions are used to evaluate the performance of the machine learning model during training. We consider two different loss functions when predicting azimuth and elevation simultaneously with the same model, such as a neural network. One loss

function considers the offset in azimuth and elevation separately, and one considers the total distance. Let \tilde{y}_{Az} and \tilde{y}_{El} denote the prediction for the offset in azimuth and elevation, respectively. y_{Az} and y_{El} are the true values. The first loss function is the mean squared error

$$\mathcal{L}_{\text{MSE}} = \frac{1}{2N} \sum_i^N \left((y_{Az,i} - \tilde{y}_{Az,i})^2 + (y_{El,i} - \tilde{y}_{El,i})^2 \right), \quad (4.2)$$

where N is the number of predictions.

For the second loss function, we use the mean squared distance

$$\mathcal{L}_{\text{MSD}} = \frac{1}{N} \sum_i^N \left[(y_{Az,i} - \tilde{y}_{Az,i})^2 + (y_{El,i} - \tilde{y}_{El,i})^2 \right], \quad (4.3)$$

It is difficult to predict the effects of these loss functions if any at all, but one difference could be that \mathcal{L}_{MSE} is more sensitive to outliers, and \mathcal{L}_{MSD} reduces the offsets more evenly.

For models with a single output azimuth or elevation, we use the regular mean squared error (4.1)

4.3 Train/test set

Machine learning models can be highly complex and fit all the data points in a dataset. While this can result in perfect predictions on the training data, it often leads to poor performance on new data, a phenomenon known as overfitting. To counteract this, the data is typically split into two parts - a training set and a validation set. The model is trained on the training set, and the error on the validation set is used to evaluate the model's performance. By using a separate set of data for validation, we can better estimate the model's performance on new data and avoid overfitting.

When the error on the training data is low, the model has low bias. However, if the model is too complex, it may also have high variance, meaning that it is overly sensitive to the training data and unable to generalize well to new data. A model with high variance may perform well on the training data, but its performance on new data may be poor. The key to building a good model is to balance bias and variance and to find the right level of complexity that will allow the model to generalize well. Proper selection of the train/test split ratio and other techniques, such as regularization, can help achieve this balance and improve the model's performance.

One usually picks the machine learning model with the best performance on the validation set, but this performance is not a reasonable estimate of the expected performance on future predictions. That is because many models are usually trained, and the model with the best performance on the validation set could have gotten lucky. Therefore, a third test set is used to get an unbiased estimate of the model's performance. The data in the test set is not used when training or validating and is only used to estimate the final model's performance.

4.4 Scaling

In machine learning, some models, such as neural networks, are highly sensitive to the scale of input data. The inputs to a model often contain different types of data

with varying scales. Neural networks use weights to transform the input data, and each neuron in a fully connected network receives data from every input feature. If the input features have different scales, training the weights can be slow and unstable. Scaling the input data to have the same scale improves the speed and performance of the model. In contrast, tree-based models are not affected by the range scale of the data since they consist of tests and not mathematical operations.

The most common scaling method is to standardize the data to have zero mean and a standard deviation of one. This is achieved by subtracting the mean and dividing by the standard deviation. Mathematically, the standardization of a feature x is represented as:

$$x_{scaled} = \frac{x - \mu}{\sigma} \quad (4.4)$$

where μ is the mean of the feature values, and σ is the standard deviation of the feature values. The mean and standard deviation are computed using the following equations:

$$\mu = \frac{1}{n} \sum_{i=1}^n x_i \quad (4.5)$$

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2}, \quad (4.6)$$

where n is the number of observations of the feature. In addition to standardization, other scaling methods, such as min-max and robust scaling, are also used in specific cases. Overall, scaling is a crucial step in preprocessing data for machine learning, as it can significantly impact the performance of a model. However, for tree-based methods, scaling has no effect, as predictions are made based on conditions in the data, not mathematical operations.

4.5 Decision Trees

Decision trees are tree-like models that make decisions based on conditions. As shown in Figure 4.1, each circle represents a node with various types, including decision nodes that split into two other nodes and leaf/terminal nodes that do not. The root node is the topmost decision node. Given an observation, a single path to a leaf node represents the prediction made by the decision tree.

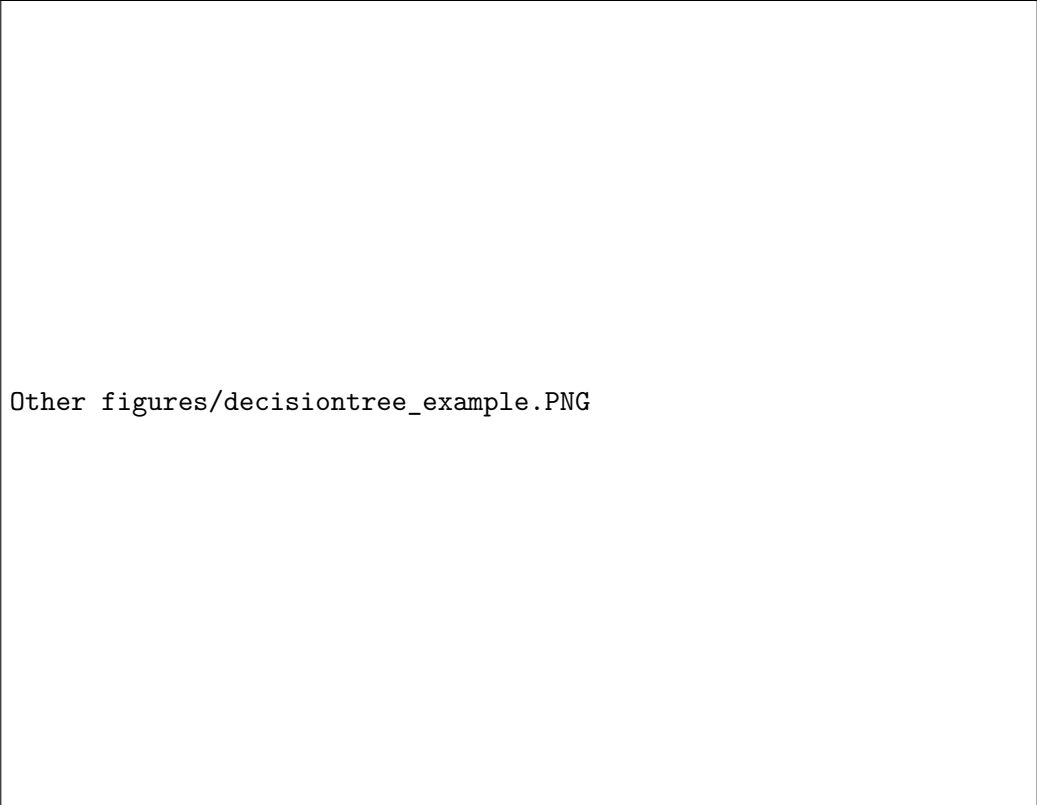
Trees are constructed greedily from the top, meaning that each split is made to minimize the loss function at the current step without considering future splits. More than a single decision tree is required for complex problems. Various methods exist to improve decision tree models, as Figure 4.2 demonstrates. The final step in the figure is XGBoost (Extreme Gradient Boost), a highly efficient and high-performing machine learning algorithm. This section will briefly cover the methods used to optimize decision trees for prediction. [6]

Bagging Bagging, also known as Bootstrap Aggregation, is a method for training an ensemble of models that contribute to the final prediction. Each model is trained using bootstrapped data (resampled from the original dataset with replacement), resulting in diverse decision trees. The final prediction is the average of all ensemble models.

Random Forest Random forest is based on bagging, where each tree in the ensemble is made using only a randomly chosen subset of features. This often leads to better generalization and reduced overfitting.

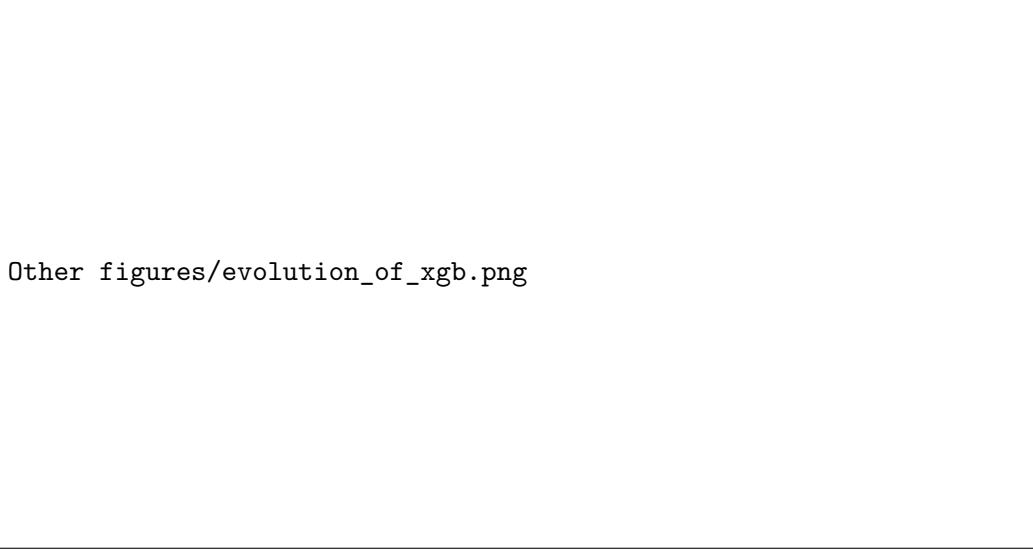
Boosting In boosting, an ensemble is created, but the trees are not made independently. They are trained one by one, considering the previous trees. A sample weight is assigned to each sample used to train a tree based on the current ensemble's accuracy. Samples with significant prediction errors are assigned larger weights, and those with accurate predictions are assigned lower weights. The final prediction is a weighted sum of all ensemble predictions, with weights based on each tree's accuracy.

Gradient Boosting Like in regular boosting, an ensemble of trees is created iteratively by considering the errors made by previous trees. The process starts with a constant model that predicts the mean of all samples. The gradient of the loss function with respect to each sample is calculated, and a tree is made to predict these gradients. The new prediction is the constant plus a small step in the direction of the predicted gradients. Repeated iteration with small steps in the gradient direction helps reduce both bias and variance.



Other figures/decisiontree_example.PNG

Figure 4.1: Decision tree with 3 decision nodes and 5 leaf nodes.



Other figures/evolution_of_xgb.png

Figure 4.2: Evolution of XGBoost. [7]

4.6 Neural Networks

A Neural Network (NN) is an Artificial Intelligence (AI) model composed of inter-connected neurons inspired by biological neural networks in animal brains. These networks are arranged in layers, as shown in Figure 4.3, and consist of an input layer, one or more hidden layers, and an output layer. The size of the hidden layer(s) varies depending on the nature of the problem. A neural network processes input to produce an output, ideally close to the true value.

Each connection in an NN has a trainable weight w_{jk}^l , representing the weight from the k^{th} neuron in layer $(l - 1)$ to the j^{th} neuron in layer l . Each neuron also has its own bias b_j , added to its output to prevent the input to its activation function σ from being zero. The activation function σ applied to the neuron's output is the final transformation before passing data to the next layer. This nonlinear function is crucial in allowing NNs to learn nonlinear relationships in data [1].

The following is the mathematical explanation of how a neuron processes the outputs from the previous layer.

$$a_j^l = \sigma \left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l \right) = \sigma(z_j^l) \quad (4.7)$$

The quantity

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \quad (4.8)$$

will be helpful when explaining how to optimize a neural network and can be considered the weighted input for neuron j in layer l .

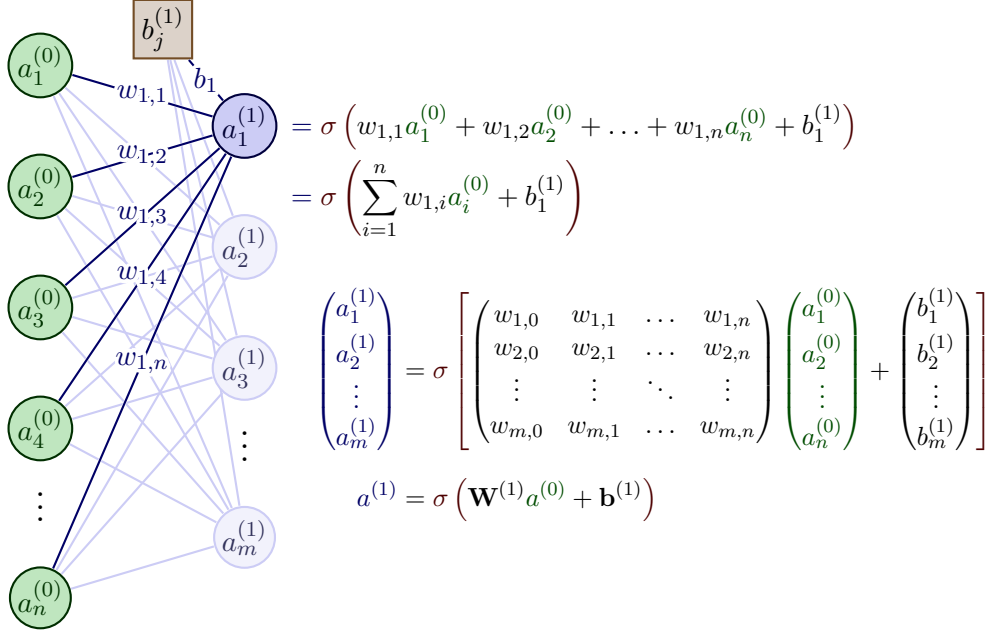


Figure 4.3: This is an illustration of how information is passed through and processed in a neural network. Generated using TikZ [11]

4.6.1 Backpropagation

Backpropagation[9] is a fundamental algorithm in training artificial neural networks. It calculates the gradient of the loss function with respect to all the weights and biases in the network, allowing for updating these parameters to reduce the loss. The algorithm is based on four key equations, which we describe in this section.

We define the error in the j^{th} neuron in the l^{th} layer by

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l} \sigma'(z_j^l) \quad (4.9)$$

This can also be considered the partial derivative of the cost function with respect to the bias in neuron j in layer l , as

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial b_j^l} \frac{\partial b_j^l}{\partial z_j^l} = \frac{\partial C}{\partial b_j^l}, \quad (4.10)$$

where we have used the relation $\partial b_j^l / \partial z_j^l = 1$ from rearranging equation (4.8). The next equation relates the error in a neuron with the errors in the neurons in the subsequent layer.

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \left(\sum_k \delta_k^{l+1} w_{kj}^{l+1} \right) \sigma'(z_j^l) \quad (4.11)$$

Note that the indices on the weight w are now swapped. We may think of this equation as an error propagating backward by multiplying the error in layer $l+1$ with the transpose of the weight connecting layer l with $l+1$. We derive the final equation from the partial derivative of the cost function with respect to the weight w_{jk}^l

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1} \quad (4.12)$$

Equation (4.9) lets us calculate the error in the last layer, and using equation (4.11), we can propagate this error backward through the network, calculating the error for all the neurons. We then use equations (4.10) and (4.12) to calculate the gradient of the cost function with respect to the weights and biases.

4.6.2 Gradient Descent

Gradient Descent (GD) is an iterative optimization algorithm used in machine learning for minimizing a differentiable function. The goal of GD is to update the model's trainable parameters in such a way that the loss function is minimized. In mathematical terms, we aim to find the values of the parameters $\boldsymbol{\theta}$ that minimize the objective function $\mathcal{L}(\mathbf{x}, \boldsymbol{\theta})$, where \mathbf{x} represents the input data. The loss function is typically defined as the mean squared error (4.1) for regression problems, so

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N (y_i - f(\mathbf{x}_i, \boldsymbol{\theta}))^2, \quad (4.13)$$

where $f(\mathbf{x}_i, \boldsymbol{\theta})$ is the output of the model for input data \mathbf{x}_i , and y_i is the target value.

To achieve the goal, GD involves calculating the gradient of the loss function with respect to the model's trainable parameters and updating them iteratively by taking a small step in the negative direction of the gradient. The iterative update rule can be expressed as follows:

$$\mathbf{v}_t = \eta_t \nabla_{\boldsymbol{\theta}} \mathcal{L}(\mathbf{x}, \boldsymbol{\theta}), \quad (4.14)$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t, \quad (4.15)$$

where η denotes the learning rate, and $\nabla_{\boldsymbol{\theta}}$ denotes the gradient with respect to $\boldsymbol{\theta}$. The learning rate determines the step size of the update, and it is important to choose a suitable value to ensure convergence of the optimization.

One major limitation of GD is that it can get stuck in local minima, yielding suboptimal results. The choice of initial parameter values $\boldsymbol{\theta}$ can also impact the final optimized model. Moreover, computing the gradient using the entire dataset can be computationally expensive for large datasets. To address these limitations, various modifications of GD have been proposed, such as stochastic gradient descent (SGD) and mini-batch gradient descent (MBGD), which compute the gradient using only a subset of the data at each iteration. These modifications can help to accelerate the convergence and improve the scalability of GD.

4.6.3 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a widely used optimization algorithm in machine learning that addresses some of the limitations of Gradient Descent (GD). Unlike GD, which computes the gradient using the entire dataset at each iteration, SGD computes the gradient using only a randomly sampled subset, called a mini-batch. This makes SGD more efficient and less computationally expensive than GD, particularly for large datasets. Furthermore, by randomly sampling mini-batches, SGD is more likely to escape local minima and converge to the global minimum. The update rule for SGD can be derived similarly to that for GD, with the only difference being the replacement of the full dataset with a mini-batch. By iteratively updating the model's parameters using mini-batches, SGD can converge faster and more robustly than GD.

However, SGD also has limitations to consider. If the learning rate is too large, the optimization may overshoot the minimum and fail to converge. On the other hand, if the learning rate is too small, the optimization may converge very slowly. In addition, if there are areas in the function space with small gradients, the optimization may stagnate and fail to converge. To address these limitations, various modifications of SGD have been proposed, such as adaptive learning rate methods like Adagrad and RMSprop, which adjusts the learning rate dynamically based on the history of the gradients. These modifications can improve the stability and convergence speed of SGD.

4.6.4 Momentum GPT

In practice, SGD is mostly used with momentum. Momentum serves as a memory of previous momenta and can improve the convergence speed of SGD, particularly in areas of the function space with low gradients, such as local minima.

The update rule for momentum can be expressed as follows:

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta_t \nabla_{\theta} \mathcal{L}(\mathbf{x}, \theta) \quad (4.16)$$

$$\theta_{t+1} = \theta_t - \mathbf{v}_t, \quad (4.17)$$

where γ is the momentum parameter with $0 \leq \gamma \leq 1$. The momentum term considers the update of the previous step, in addition to the gradients at the current step. By incorporating previous momenta, momentum can smooth out variations in the optimization trajectory and accelerate convergence towards the minimum.

Momentum is particularly useful when the gradient direction is consistent across many iterations, as it allows the optimization to maintain a higher velocity in the same direction. In contrast, in areas of high variance or noisy gradients, momentum may cause overshooting and slow down convergence. To address this, adaptive momentum methods like Adam have been proposed, which adjust the momentum parameter dynamically based on the history of the gradients. These methods can improve the convergence speed and stability of momentum-based optimization algorithms.

4.6.5 Adam GPT

Adam is an optimization algorithm that combines the benefits of both SGD with momentum and adaptive learning rate methods. It uses a running average of the first and second moments of the gradient to compute per-parameter adaptive learning rates. Adam updates the parameters iteratively as follows:

$$\mathbf{g}_t = \nabla_{\theta} \mathcal{L}(\mathbf{x}, \theta) \quad (4.18)$$

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \mathbf{g}_t \quad (4.19)$$

$$\mathbf{s}_t = \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2 \quad (4.20)$$

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1 - (\beta_1)^t} \quad (4.21)$$

$$\hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - (\beta_2)^t} \quad (4.22)$$

$$\theta_{t+1} = \theta_t - \eta_t \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}, \quad (4.23)$$

where \mathbf{g}_t denotes the gradient at time step t , \mathbf{m}_t and \mathbf{s}_t are the first and second moment estimates, respectively. β_1 and β_2 control the decay rate of the first and

second moments, respectively. η_t is the learning rate, and ϵ is a regularization constant to prevent division by zero.

Adam has several advantages over other optimization algorithms, including its ability to adaptively compute per-parameter learning rates and the robustness of its estimates to noise in the gradient. The adaptive learning rates can help speed up convergence and lead to better performance. Furthermore, the memory of previous first and second-order gradient estimates enables the algorithm to be more robust to noise and outliers in the data. As a result, Adam is widely used and has become the de facto standard optimization algorithm in deep learning.

4.6.6 Activation functions

Activation functions play a crucial role in training a neural network by allowing it to learn non-linear relationships between inputs and outputs. Different activation functions have varying properties; we will discuss some of the most common ones in this section. Properties like non-linearity, differentiability, monotonicity, smoothness, and zero-centering are important for activation functions. Non-linearity enables the model to capture complex relationships, differentiability is necessary for calculating the derivative of the loss function with respect to the trainable weights, monotonicity helps ensure stability in activation outputs, smoothness stabilizes gradients during training, and zero-centering balances the activation distribution within the model.

- Non-linearity enables the model to capture complex relationships
- Differentiability is necessary for calculating the derivative of the loss function with respect to the trainable weights
- Monotonicity helps ensure stability in activation outputs, smoothness stabilizes gradients during training
- Smoothness: A smooth activation function helps stabilize the gradients and training.
- Zero-centering balances the activation distribution within the model.

Tanh Tanh, the hyperbolic tangent function is given by

$$\text{Tanh}(x) = \frac{e^x + e^{-x}}{e^x - e^{-x}} \quad (4.24)$$

ReLU The Rectified Linear Unit (ReLU) activation function pushes all negative values to zero while leaving positive values unchanged, which introduces non-linearity while solving the vanishing gradients problem by having a gradient of either 0 or 1 for negative and positive values, respectively.

$$\text{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \quad (4.25)$$

GeLU The Gaussian Error Linear Unit (GeLU) is a smooth approximation of the ReLU function, given by

$$\text{GeLU}(x) = x\Phi(x), \quad (4.26)$$

where $\Phi(x)$ is the standard Gaussian cumulative distribution function.

GeLU can be approximated with

$$\text{GeLU}(x) \approx 0.5x \left(1 + \tanh \left[\sqrt{2/\pi} (x + 0.044715x^3) \right] \right), \quad (4.27)$$

which is faster to compute than the original definition but can result in worse performance. For computational efficiency, we used this approximation.

4.7 Model Explainability

In the context of machine learning, SHAP [5] and SAGE [3] apply the same idea to determine the contribution of each feature to a prediction. SHAP provides a local explanation by computing the contribution of each feature to the prediction of a single data point. On the other hand, SAGE provides a global explanation by computing each feature’s contribution to the model’s overall prediction performance. These methods allow us to understand the relationship between the features and the prediction, particularly useful when the model is too complex to interpret. Additionally, they provide a way to validate the model’s fairness and bias. By understanding which features contribute the most to a prediction, one can determine if the model is fair or biased and if the prediction is trustworthy.

Both SHAP and SAGE methods are based on Shapley values [10], a concept in game theory introduced by Lloyd Shapley in 1951. Shapley values determine each player’s contribution to a group’s surplus or overall value. The explanation below of Shapley values, SHAP, and SAGE is inspired by a blog post by Ian Covert [2].

The Shapley value for a player i in a cooperative game with d players is

$$\phi_i(w) = \frac{1}{d} \sum_{S \subseteq D \setminus \{i\}} \binom{d-1}{|S|}^{-1} [w(S \cup \{i\}) - w(S)] \quad (4.28)$$

where D is the set of all players, S is a coalition of players, $w(S)$ is the value of the coalition S , and $|S|$ is the number of players in the coalition. This formula satisfies four important conditions:

- **Efficiency:** The sum of all Shapley values is equal to the group’s total value.
- **Symmetry:** If two players i and j have the same impact on all coalitions with $w(S \cup \{i\}) = w(S \cup \{j\})$ for all S , they should have the same Shapley value $\phi_i(w) = \phi_j(w)$.
- **Dummy:** A player i that makes no contribution to the group with $w(S \cup \{i\}) = w(S)$, should receive a value of zero, or $\phi_i(w) = 0$.
- **Linearity:** A player’s value is proportional to their contribution to the group. If player i contributes twice as much as player j to the group’s overall worth, then player i should have twice the Shapley value.

4.7.1 SHAP

Shapley values explain how each feature (x^1, \dots, x^d) in a model f contributes to the deviation from the mean prediction $\mathbb{E}[f(x)]$ of the dataset for a single prediction. It assigns a value ϕ_1, \dots, ϕ_d to each feature that quantifies the feature’s influence on

the prediction $f(x)$. SHAP (Shapley Additive Explanations) computes approximate Shapley values for machine learning models.

We define a cooperative game $v_{f,x}$ to represent a prediction given the features x^S , as

$$v_{f,x}(S) = \mathbb{E} \left[f(X) | X^S = x^S \right], \quad (4.29)$$

where x^S are known, and the remaining features are treated as random variable $X^{\bar{S}}$ (where $\bar{S} = D \setminus S$). This is the mean prediction $f(X)$ when the unknown values follow the conditional distribution $X^{\bar{S}} | X^S = x^S$.

Using a subset of features from the prediction while sampling the rest from the dataset reduces the chance of improbable samples. Given this convention for making predictions, we can apply the Shapley value to define each feature's contribution to the prediction $f(X)$ using Shapley values $\phi_i(v_{f,x})$. A Shapley value of $\phi_i(v_{f,x}) > 0$ indicates that feature i contributes to an increase in prediction $f(X)$. A negative Shapley value $\phi_i(v_{f,x}) < 0$ indicates the opposite, that the feature contributes to a decrease in $f(X)$. Uninformative features will have small values $\phi_i(v_{f,x}) \approx 0$.

4.7.2 SAGE

SAGE (Shapley Additive Global Importance) explains how every feature contributes to the model's overall performance, and it relates to SHAP in a simple way. For a given feature, the global feature importance is the average SHAP value (for that feature) across all samples in the dataset. This is, however, different from how it is calculated in practice. A paper by Ian Covert et al. [4] on global feature importance proposes an algorithm that aims directly at a global feature explanation, unlike the SHAP values, which makes it faster. This is the algorithm used for approximating the SAGE values for the features in the thesis.

4.8 Mutual Information

Mutual information is a fundamental measure of the statistical dependence between two random variables, providing a way to quantify the amount of information one variable conveys about the other. For a pair of discrete random variables X and Y , we have

$$I(X; Y) = \sum_y \sum_x p(x, y) \log \left(\frac{p(x, y)}{p(x)p(y)} \right), \quad (4.30)$$

where $p(x, y)$, $p(x)$, and $p(y)$ are the joint and marginal probabilities, respectively. The mutual information captures linear and nonlinear relationships between variables, unlike Pearson's correlation coefficient, which can only detect linear relationships. However, mutual information has limitations in that it relies on binning the data, which can introduce bias and limit the resolution of the information.

Furthermore, estimating mutual information for high-dimensional data sets can be computationally expensive. Despite these limitations, mutual information remains a popular tool in feature selection, data visualization, and machine learning.

Chapter 5

Method

This section of the thesis outlines the methodology used for data analysis, cleaning, transformation, feature engineering, and machine learning experiments. Data-driven methods have the potential to learn all relations in the data, but the size of the dataset limits this. Therefore, cleaning the data and selecting features containing information relevant to the desired output is essential. With the system’s complexity, identifying relevant features can be challenging. To address this, we employ data-driven modeling to help identify important features while using feature engineering to incorporate our understanding of the system and create informative features.

Cleaning data involves removing irrelevant data that could confuse the model, thereby ensuring that the model learns from the most relevant information. Feature engineering involves incorporating domain knowledge into the model to create features that provide additional information.

We perform data analysis to decide which features to train our models. This analysis involves understanding the relationships between the different variables in the dataset and identifying which variables could be helpful in predicting the target variable. The outcome of the data analysis informs our decisions on which features to use in the models.

5.1 Cleaning pointing scan data

When utilizing data-driven modeling for predictive purposes, ensuring that the dataset is clean and informative is crucial. In this project, various factors may impact the quality of the data, and therefore, we implemented measures to clean the data based on our knowledge of the telescope’s operation. We employed a criteria-based approach and a machine learning classifier to remove pointing scans from the dataset. During the removal of pointing scans, it is important to strike a balance between removing noise and retaining relevant information. Outliers in the training data can introduce bias into machine learning models, as these data points may not accurately represent real-world conditions. Consequently, having outliers in the training data can be more damaging than removing good pointing scans. Therefore, we have a strict approach when cleaning the data to ensure high-quality datasets for model training.

5.1.1 Cleaning criteria

To eliminate unreliable or unusable scans, we applied criteria informed by the insights of astronomers at APEX. The following list outlines the criteria used to filter out such scans:

- Scans using the HOLO transmitter: These scans are aimed at a radio tower and are not realistic data for training an ML model.
- Scans using ZEUS2: These are highly experimental pointing scans and unreliable.
- Scans using CHAMP690: There are very few scans with this instrument.
- Scans in January and February of 2022: The weather is unreliable and there are few scans in this period.
- Scans that are tracking tests
- Scans after 17.09.2022 since we only have sensory data until this point

After this filtering, there are 5901 out of 8862 scans left.

5.1.2 Pointing scan classifier

Method

In addition to cleaning the data based on the criteria above, we had to remove the outright bad pointing scans (like ?? ??, ??). The scan quality is often obvious when inspecting the data visually, but it is hard to develop suitable measures to identify which scans are good or bad. Instead, we trained a classifier to predict whether a scan is of good or bad quality. We used an XGBoost classifier with 13 features as inputs, all of which are present in the pointing scan figures (?? and ??). The first 12 features are the amplitudes, FWHMs, pointing offsets, and these values' uncertainties. The last feature is the beamsize of the telescope for the given observing frequency.

We had to label a training set by manually looking at pointing scans. The size of the training set was 369 samples with 270 good and 99 bad scans. Table 5.1 shows the hyperparameters and search ranges we used when optimizing this model, along with the resulting best parameter values. We also used *scale_pos_weight* to consider the unbalanced classes, for which the value is the ratio of negative to positive classes (number of bad scans divided by number of good scans). We split the data into 80% for training and the rest for testing, corresponding to 295 and 74 samples for training and testing, respectively.

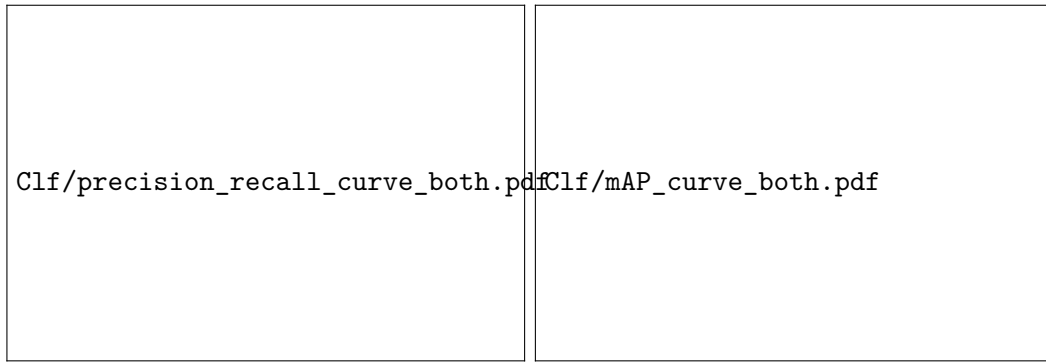
Table 5.1: This table presents a list of parameters we sampled during hyperparameter tuning for the pointing scan classifier. The table includes names, sampled distributions and corresponding ranges, and parameter values for the best model.

Parameter	Sample Distribution	Range	Best Parameter Value
max depth	Uniform	[1, 5]	2
n estimators	Uniform	[1, 80]	53

Results

The XGBoost classifier performed well with a 97% overall accuracy on the test set. Figure 5.1 shows the precision-recall curve on the left and the average precision curve on the right. From the precision-recall curve, it is clear that we can achieve close to 100% precision while still having a high recall. We select a large threshold such that the classifier removes most bad scans from the training data, because a bad pointing scan is potentially more harmful for the model than discarding a few good scans. The average precision curve shows an optimal threshold for maximizing the precision, which is about 80%.

Using the classifier to further clean the dataset, using prediction threshold 0.8, we remove another 575 scans, leaving us with 5326 scans for the rest of the analysis.



(a) Precision-recall curve on the test set. (b) Average precision for different classification threshold.

Figure 5.1: Precision-recall and average precision curve for the XGBoost classifier when classifying good and bad pointing scans in the test set.

5.2 Scan duration analysis

As mentioned in the database section ??, the scans' timestamps are not the accurate start time of a scan. The tiltmeter dump files with the flag indicating whether the telescope is idle, preparing to observe, or observing, is the only accurate data we have when the telescope performs a pointing scan. Therefore, we need to combine the timestamp of the pointing scan with the flag in the dump files to analyze the duration of scans.

5.2.1 Analysis

First, we convert the different scan flags to numbers. *IDLE* and *PREPARING* is the set 0, and *OBSERVING* is set to 1. Then we can subtract the previous rows from all rows, resulting in the value 1 when the scan starts, and -1 when it ends. Table 5.2 shows an example of the resulting table.

Time	Flag	Flag Integer	Δ
11:21:21	IDLE	0	0
11:21:22	PREPARING	0	0
11:21:23	OBSERVING	1	1
11:21:24	OBSERVING	1	0
11:21:25	OBSERVING	1	0
11:21:26	IDLE	0	-1

Table 5.2: This table shows the tiltmeter dump file containing the telescope state flag, and how we find the start ($\Delta = 1$) and end ($\Delta = -1$) of a scan.

5.2.2 Algorithm

With the scan timestamp and the observing flag from tiltmeter dumps, we used the following algorithm to obtain the start and end of pointing scans.

Algorithm 1 Find start and end of pointing scan

Input:

- Pointing scan timestamps $D = \{D_1, \dots, D_n\}$
- Timestamps $T = \{T_1, \dots, T_m\}$ and scan flag $F = \{F_1, \dots, F_m\}$

Output: Start and end of pointing scans $S = \{S_i, \dots, S_n\}$ and $E = \{E_i, \dots, E_n\}$

```

for  $i = 1, \dots, m$  do
  if  $F_i = \text{OBSERVING}$  then
     $F_i = 1$ 
  else
     $F_i = 0$ 
  end if
end for

```

```

for  $i = 1, \dots, n$  do
   $\hat{T} = \{T_j, \text{ if } T_j > D_i\}_j^m$ 
   $\hat{F} = \{F_j, \text{ if } T_j > D_i\}_j^m$ 
  for all  $t_i, f_i$  in  $\hat{T}, \hat{F}$  do
     $\Delta = f_i - f_{i-1}$ 
    if  $\Delta = 1$  then
       $S_i = t_i$ 
    end if
    if  $\Delta = -1$  then
       $E_i = t_i$ 
      Continue
    end if
  end for
end for

```

5.2.3 Results

By analyzing start and end timestamps for all the scans we had tiltmeter dumps for, we see that the first *OBSERVING* flag present after a scan is on average 53.9 seconds after the scan timestamp on average, with a standard deviation of 20.5 seconds.

Figure 5.2 shows boxplots of this time difference for each of the instruments, which strongly indicates that assuming the starting point of a scan is 53.9 seconds after the timestamp is reasonable. In the same plot, we also see that the starting time is fairly constant for the different instruments. The right plot of Figure 5.3 shows the time difference in seconds between the first observing flag after a scan timestamp throughout the year. From the plot, this stays constant over time.

Now that we have found the starting points of the pointing scans, we can look at their duration. The left plots in Figure 5.3 and 5.2 show the duration of the pointing scans for different instruments. From these figures, it is clear that the duration of a pointing scan varies a lot. A varying scan duration is problematic because we only have these tiltmeter dump files for $2875/8381 \approx 34\%$ of the pointing scans. To address this issue, we collected data for feature engineering over a shorter period of time. It is important to note that using data from after a pointing scan has ended can be inaccurate, as the telescope may start observing a different source. When examining the scatter plot of scan durations, we observed clusters of scans around 60-70 seconds, 120-130 seconds, and so on. To ensure accuracy, we used the mean scan duration grouped by instrument and shorter than 100 seconds as the cutoff for the duration of time from which we collected data. For the scans with an exact start and end time, we used this time period instead. [Add list of values](#)

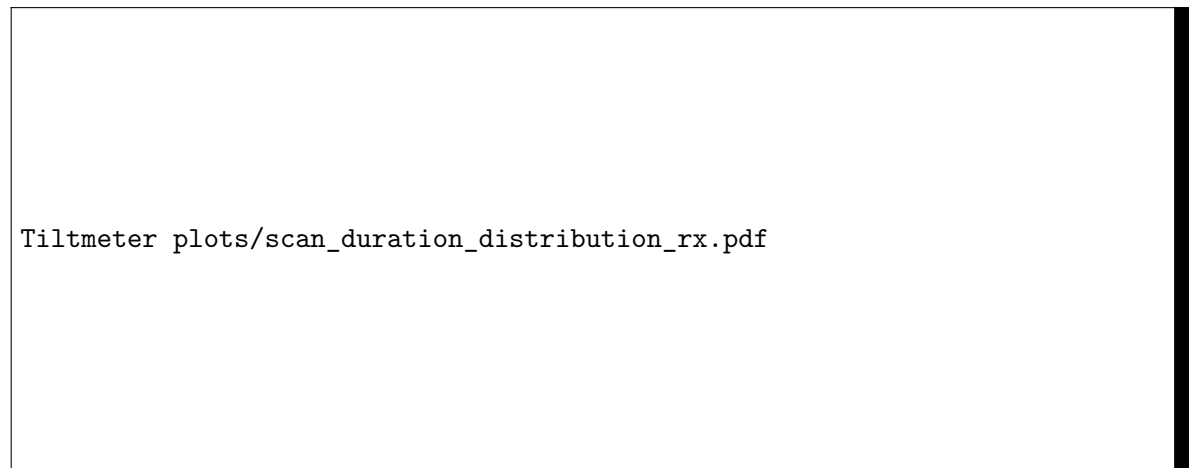


Figure 5.2: Box plot of the duration of scans, and the time difference between the timestamp of a scan and the actual start of it.

Tiltmeter plots/scan_duration_distribution_date.pdf

Figure 5.3: Scatter plot of the duration of scans, and the time difference between the timestamp of a scan and the actual start of it.

5.3 Feature Engineering

There are two main features engineered for this project; features that represent the system during a pointing scan and features that represent changes since the last correction. The idea behind this is simple. The correction used during a pointing scan represents the ideal correction for the system during the previous pointing scan. As there are a lot of factors and complex relationships, and we do not have large amounts of training data, it might be easier for the model to learn how these changes affect the pointing rather than learning all the relationships.

Table [ref table with a list of variables with median value](#) show all features.

Median values The median value of variables during a pointing scan is the most used feature.

Sum of all change To capture systematic error in pointing due to the telescope moving back and forth in azimuth and elevation, we sum over the positive and negative changes in these variables.

Given the time of the last pointing correction t_1 and the start of a pointing scan t_2 , the sum over the positive changes in a variable x_i is given by

$$X = \sum_{i=t_1+1}^{t_2} \max(0, x_i - x_{i-1}) \quad (5.1)$$

Similarly, the sum of negative changes in a variable is

$$X = \sum_{i=t_1+1}^{t_2} \min(0, x_i - x_{i-1}) \quad (5.2)$$

We make these features with azimuth and elevation.

Change since the last correction This feature is self-explanatory and is just the change in a variable since the pointing was corrected.

$$\Delta x = x_{t_2} - x_{t_1} \quad (5.3)$$

In order to make this feature more robust against noisy data, we instead consider the change in the median for a time interval around the last correction t_1 and the start of a pointing scan t_2

$$\Delta x = \text{median}(x_{t_2}, x_{t_2-1}, \dots, x_{t_2-p}) - \text{median}(x_{t_1}, x_{t_1+1}, \dots, x_{t_1+p}), \quad (5.4)$$

where p is the number of data points needed to cover a period of P minutes, given by $p = P \cdot \text{frequency}$. The unit of frequency is data points per minute, found in Table ??.

Max change in time interval In case the speed of the temperature change affects the deformation of the telescope's structure, we find the maximum temperature change in a given time interval since the last pointing correction.

$$X = \max(x_{t_1+p} - x_{t_1}, x_{t_1+p} - x_{t_1}, \dots, x_{t_2} - x_{t_2-p}), \quad (5.5)$$

Position of the sun Observers at the telescope report that the sun is affecting the pointing. It is most drastically affected when the sun sets or rises, likely due to rapid temperature change leading to deformation in the telescope structure. We also think the sun's position affects the pointing. For instance, if the sun is shining on the left side of the telescope, it will affect the pointing differently than if it is on the right side. Obtaining the sun's position for the telescope's location is done using the python module PyEphem [8].

Using the azimuth angle of the sun and the telescope, we can calculate the position of the sun with respect to the pointing with

$$\Delta Az_{\odot} = Az_t - Az_{\odot} \quad (5.6)$$

This will result in values outside the $[-180^\circ, 180^\circ]$. An example is if $Az_{\odot} = 179^\circ$ and $Az_t = -179^\circ$. The calculation in equation (5.6) yield $-179^\circ - 179^\circ = -358^\circ$, which corresponds to the sun being 358° to the right of the telescope, while it ideally should be 2° to the left. Therefore, we adjust the values accordingly

$$\Delta Az_{\odot} = Az_{\odot} + 360^\circ, \text{ for } \Delta Az_{\odot} < -180^\circ \quad (5.7)$$

$$\Delta Az_{\odot} = Az_{\odot} - 360^\circ, \text{ for } \Delta Az_{\odot} > 180^\circ \quad (5.8)$$

Here, the interval of the difference in azimuth is fixed to the interval $(-180^\circ, 180^\circ)$, where 0° means the telescope is pointing towards the sun in the azimuth direction. $\Delta Az_{\odot} = 90^\circ$ corresponds to the sun being direct to the left of the pointing direction.

Another measure tested is the total angle between the pointing and the sun's position. We calculate this using the following formula

$$\theta = \cos Az_t \cdot \cos El_t \cdot \cos Az_{\odot} \cdot \cos El_{\odot} + \sin Az_t \cdot \cos El_t \cdot \sin Az_{\odot} \cdot \cos El_{\odot} + \sin El_t \cdot \sin El_{\odot} \quad (5.9)$$

5.3.1 List of features

[add a list of all features for different calculations here](#)

5.4 Machine Learning Experiments

In this section, we will provide an overview of two machine learning experiments pertinent to the two research questions. [refer to section with research qs](#) The first experiment aims to investigate the effectiveness of neural networks in developing a pointing model that could replace the current linear model, which is created through linear regression. It explores the feasibility of a more sophisticated model in terms of pointing accuracy. The second experiment aims to examine the effectiveness of an XGBoost model in predicting pointing scan offsets to enhance the pointing accuracy. The primary objective of this experiment is to assess whether the proposed model can outperform the current model in terms of pointing accuracy.

5.4.1 Experiment 1: Pointing Model using Neural Networks

This experiment uses the raw dataset containing input coordinates, Az_{input} and El_{input} respectively, and corresponding true observed values Az_{observed} and El_{observed} .

The goal is to find a model f such that

$$f(X) \approx (\delta_{Az}, \delta_{El}) = (Az_{\text{observed}} - Az_{\text{input}}, El_{\text{observed}} - El_{\text{input}}) \quad (5.10)$$

We split the data into a train, validation, and test set. The last 15% of the data, which we sorted by date, is used for testing. We use the remaining 85% of the data for training and validation and split this set into 20% for training and 80% for validation. This results in $\approx 76\%$ and $\approx 24\%$ of the total dataset used for training and validation.

Feature Selection

Selecting the right features is essential in improving the pointing model's accuracy. This model uses two types of features: geometrical and harmonic terms already part of the current linear base model and new features extracted from the telescope's database. We identified relevant features by calculating Pearson's and Spearman's rank correlation to the offsets for all features. We analyzed the correlation of the geometrical terms, and harmonic terms using sine and cosine functions of azimuth and elevation up to the fifth order. Then, we chose the terms with the strongest correlation to the model and used them in all models. We made a list of the features we extracted from the database that showed a correlation equal to or greater than 0.1. During model training, we randomly selected a subset of 2 to 19 features from this list and used them to train the model.

Model Architecture

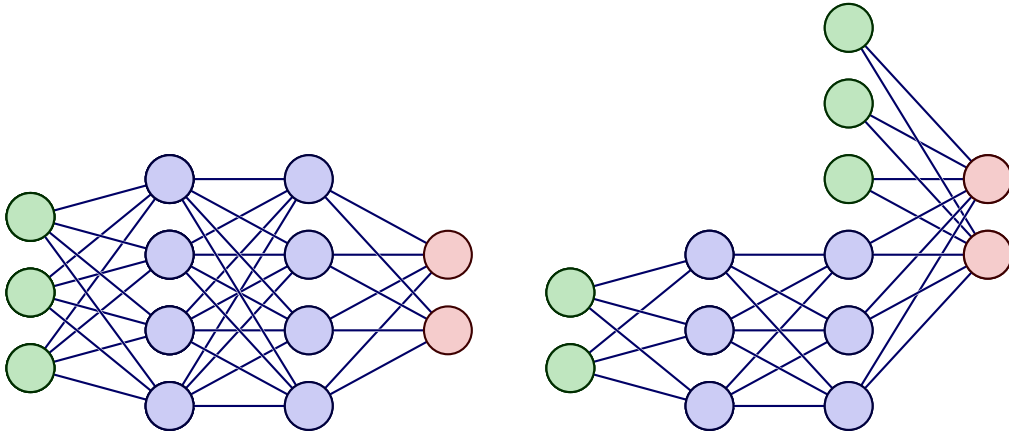
This experiment utilized four different model architectures. The first architecture involved feeding all input data into one, two, or three hidden layers. The other three architectures incorporated machine learning techniques by separating the geometrical and harmonic terms of the input data from the other features and processing them using distinct architectures. With these approaches, we intend to keep the current model's simplicity and performance while incorporating new features.

The following are the four different architectures:

1. **Regular Neural Network:** All features are passed through the same layers, all with a nonlinear activation function. See Figure 5.4a

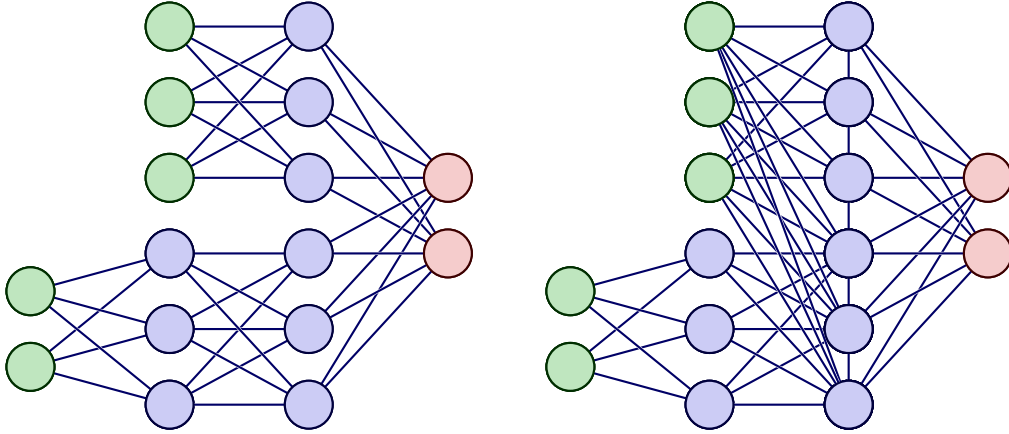
2. **Neural Network with Separated Features 1:** This architecture separates the input features into two groups: geometric and harmonic features, and the rest of the features. The geometric and harmonic features are connected directly to the linear output layer, while we pass the remaining features through layers with nonlinear activation functions. See Figure 5.4b
3. **Neural Network with Separated Features 2:** This architecture is similar to the previous architecture, but we feed the geometric and harmonic features through an additional layer of nonlinear activation function before connecting them to the output layer. See Figure 5.4c
4. **Neural Network with Separated Features 3:** This architecture combines the previous two architectures by passing the regular features through a few hidden layers with nonlinear activation functions before concatenating them with the geometric and harmonic features. We then pass the combined features through a final layer before connecting them to the output layer. See Figure 5.4d

These are visualized in Figure 5.4.



(a) **Regular neural network:** This is the standard neural network architecture without any feature separation. All features are connected to the same layers.

(b) **Neural network with separated features 1:** In this architecture, the geometric and harmonic features are separated from the other features and directly connected to the output layer without any nonlinear activation function.



(c) **Neural network with separated features 2:** Similar to the previous architecture, the geometric and harmonic features are separated from the other features. However, they are also processed by a nonlinear activation function before being connected to the output layer.

(d) **Neural network with separated features 3:** In this architecture, we concatenate the processed regular features to the geometric and harmonic features before being connected to the output layer.

Figure 5.4: These architectures were used to train a base pointing model on raw data.

The hyperparameters for the neural networks were randomly sampled from different distributions, as presented in Table 5.3. Some parameters were consistent across all models, such as the Adam optimization algorithm and the mean squared error loss function. In total, **input the number of fitted NNs here** networks of each architecture are trained for 300 epochs. We pick the model from the epoch with the best performance on the validation set.

Table 5.3: This table presents a list of parameters we sampled during hyperparameter tuning for the base pointing model. The table includes names, the distribution we sampled from, and corresponding ranges.

Name	Distribution Type	Range
hidden layers	uniform integer	[1,3]
hidden layer size	uniform integer	[20, 120]
learning rate	uniform	[0.001, 0.02]
batch size	uniform integer	[32, 512]
activation	categorical	[gelu, tanh]

Loss Function and Model Evaluation

To evaluate the performance of the models, we used the root mean squared (RMS), measured in arcseconds, on the test set. We calculate the RMS as follows:

$$\text{RMS} = \sqrt{\frac{1}{N} \sum_{i=1}^N \left((\tilde{\delta}_{Az,i} - \delta_{Az,i})^2 + (\tilde{\delta}_{El,i} - \delta_{El,i})^2 \right)}, \quad (5.11)$$

where $\tilde{\delta}_{Az}$ and $\tilde{\delta}_{El}$ are the predicted offsets, while δ_{Az} and δ_{El} are the true values. N is the number of observations in the test set.

This RMS is used to compare the performance of the models. It will also be compared with a benchmark linear regression model to see if a machine learning approach offers any improvements.

5.4.2 Experiment 2: Pointing Correction Model

This experiment aims to improve the accuracy of the existing pointing model by training XGBoost models to predict offsets obtained from pointing scans. To accomplish this, we utilized two different datasets, which we processed using the cleaning outlined in section 5.1. The difference between these datasets is that one contains the scans from all instruments, while the other only contains the scans from NFLASH230. By training our models on these datasets, we aim to reduce the pointing offset and improve the accuracy of the pointing. In addition, we varied the way we split the datasets for training and testing. We considered two cases:

- **Case 1:** The dataset is sorted by date and split into six equal-sized folds. We consider each of the folds one by one. For each of these folds, we use the last 1/6th of the data as a test set and the remaining 5/6th as training and validation.
- **Case 2:** The dataset is sorted by date and split into six equal-sized folds. We used 5/6 of the data for training and validation and the remaining for testing. We repeated this process six times, using each fold for testing once.

Figure 5.5 illustrates the two cases. In both cases, we trained and validated the model on 5/6 of the data and tested on the last 1/6. The difference is the amount of data used for training, which can indicate whether models trained on shorter or longer periods perform better. Using longer period, and thus more data, can help the model find complex relations. However, a smaller period may be better for learning relations that change over time, as we would expect less variation in a shorter period.

We also split the training and validation data such that scans from a given day only can be either in the training or validation set, not both. When splitting the data, we used 35% of the days for validation and 65% for training. This does not amount to precisely the same percentage of scans for the given split, but something close to it nonetheless.

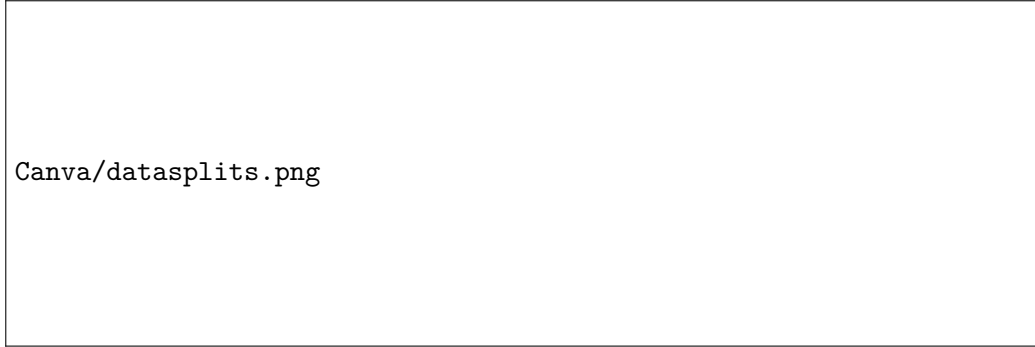


Figure 5.5: This figure shows two cross-validation cases: the orange region represents the train and validation set, the red region represents the test set, and the blue region is unused for evaluation. In **Case 1**, the dataset is split into six equal-sized folds sorted by date. For the selected fold, we use the last part (colored red) for testing and the remaining part (colored orange) for training and validation. This process is repeated six times, once for each fold. In **Case 2**, the dataset is again split into six equal-sized folds sorted by date. However, we use one whole fold for testing this time and the remaining five for training and validation. This process is repeated six times, with each fold used exactly once for testing.

Feature Selection

We trained models using a range of features, specifically $k = [2, 5, 10, 20, 30, 40, 50]$ features. We selected the k features that had the greatest mutual information for each model with the target value. This approach helps us identify the most important features to improve the model's performance. Selecting a subset of features can reduce the noise in the data. By selecting different numbers of features, we can explore the trade-off between model complexity and performance.[write and ref to mutual info in theory section.](#)

Model Architecture

We performed a Bayesian hyperparameter search for each model using the parameter space in Table 5.4. The search space includes eight hyperparameters that affect the model's complexity, such as the maximum depth of the trees, the regularization strength, and the learning rate. We used a uniform or log-uniform distribution to sample each hyperparameter within a specific range. We evaluated 200 different combinations of hyperparameters (for each dataset, cross-validation case, target variable, and the number of features selected) to find the optimal values for each model. The models were validated using the MSE, and we picked the model with the best performance on the validation set.

Table 5.4: This table presents a list of parameters we sampled during hyperparameter tuning for the pointing correction model. The table includes names, sampled distributions, and corresponding ranges.

Parameter	Sample Distribution	Range
max depth	Uniform	[1, 5]
reg lambda	Uniform	[0, 1]
colsample bytree	Uniform	[0.5, 1]
n estimators	Uniform	[20, 500]
learning rate	Log-Uniform	$[10^{-5}, 1]$
subsample	Uniform	[0.5, 1]
gamma	Log-Uniform	$[10^{-5}, 1]$
min child weight	Uniform	[1, 10]

Model Evaluation

To evaluate the performance of the models, we calculated the RMS on each test fold and compared it to the current RMS of the telescope on the same data. The RMS is calculated for azimuth and elevation separately since an XGBoost model only can predict one target. For a fold j and target either azimuth or elevation, we calculate the RMS by

$$RMS_{\text{target},j} = \sqrt{\frac{1}{N_j} \sum_{i=1}^{N_j} (\tilde{\delta}_{\text{target},ji} - \delta_{\text{target},ji})^2}, \quad (5.12)$$

where $\tilde{\delta}_{\text{target},ji}$ is the predicted pointing offset and $\delta_{\text{target},ji}$ is the true pointing offset for the i th pointing scan in fold j . N_j is the number of pointing scans in fold j . We then computed the ratio $r_{RMS,j}$ of the model's RMS to the current RMS for each fold. If the ratio is less than 1, it indicates that the XGBoost model provides an improvement over the current performance of the telescope for a given fold.

To obtain an overall measure of the model's performance compared to the current performance of the telescope, we averaged the ratios $r_{RMS,j}$ over all six test folds

$$\bar{r}_{RMS} = \sum_{i=1}^6 \frac{RMS_{\text{model},j}}{RMS_{\text{current},j}}. \quad (5.13)$$

This gives us an average ratio \bar{r}_{RMS} , which measures the improvement in performance provided by the XGBoost model. If $\bar{r}_{RMS} < 1$, it indicates that the XGBoost model outperforms the current pointing correction method on average across all test folds. By comparing the average ratio \bar{r}_{RMS} for the two different cross-validation cases in Figure 5.5 and the selected number of features, we can identify which models provide the best performance.

Chapter 6

Results

6.1 Experiment 1: Pointing Model using Neural Networks

Table ?? show the RMS in arcseconds on all the folds for the different model architectures. The results are from the models with the smallest mean RMS for each of the architectures. From the mean RMS over all folds, we see that the different architectures offer similar performance. We also see that the RMS of fold 1 is by far worse than the other folds. The lowest mean RMS is from the architecture where the non-linear features are connected to the geometrical and harmonic features.

Table ?? show the hyperparameter used for the best models. All architectures perform better with a single hidden layer. The regular neural network uses ReLU activation and MSE loss, while the other architectures use Tanh activation and MSD loss. The regular neural network also have more neurons in the hidden layer and a higher learning rate. The batch size is also varying. There seem to be some similarities between the hyperparameters chosen for the three architectures with separate features. However, given the large standard deviation of the mean RMS, there is probably bigger issues than hyperparameter tuning.

Table 6.3 lists the features used in each of the best model for all the architectures tested.

Table 6.1: RMS on all folds for the best model for all architectures

Network	RMS on test fold						Mean	STD
	1	2	3	4	5	6		
Regular	28.06	19.34	12.28	13.25	17.19	16.33	17.74	5.19
Sep 1	30.69	16.93	13.76	10.04	15.77	13.61	16.80	6.57
Sep 2	27.34	20.75	12.65	24.17	13.64	16.69	19.21	5.38
Sep 3	30.27	20.59	14.01	10.76	13.67	10.34	16.61	6.97

6.2 Experiment 2: Pointing Correction Model

Table ?? and ?? show the main results from experiment 2. They contain the average compared RMS (5.13) for azimuth and elevation models in case 1 and 2 for different

Table 6.2: Hyperparameters for the best model of each architecture

Architecture	Activation	Hidden Layers	Learning Rate	Batch Size	Loss
Regular	ReLU	[82]	0.0199	334	MSE
Sep1	Tanh	[40]	0.0098	101	MSD
Sep2	Tanh	[40]	0.0098	101	MSD
Sep3	Tanh	[26]	0.0039	358	MSD

Table 6.3: Features selected by hyperparameter search for each model

Feature	Sep 1	Sep 2	Sep 3	Regular
COMMANDAZ	x	x	x	x
COMMANDEL	x	x	x	x
DISP ABS3 MEDIAN 1	x	x	x	x
CA	x	x	x	
NPAE	x	x	x	
Constant	x	x	x	
$\cos(El)$	x	x	x	
$\cos(2 \cdot El)$	x	x	x	
$\cos(3 \cdot El)$	x	x	x	
$\cos(4 \cdot El)$	x	x	x	
$\cos(5 \cdot El)$	x	x	x	
$\sin(El)$	x	x	x	
$\sin(2 \cdot El)$	x	x	x	
$\sin(3 \cdot El)$	x	x	x	
$\sin(4 \cdot El)$	x	x	x	
$\sin(5 \cdot El)$	x	x	x	
WINDSPEED VAR 5	x	x	x	
DEL TILTTEMP MEDIAN 1	x	x	x	
DAZ DISP MEDIAN 1			x	
POSITIONY MEDIAN 1			x	
TILT1Y MEDIAN 1			x	
TEMPERATURE MEDIAN 1			x	
TILT1T MEDIAN 1			x	

numbers of features k used to train the models. We cleaned the datasets we used to train these models using the cleaning criteria and XGBoost classifier. Table ?? shows results for models trained on scans from all instruments, while Table ?? shows results for models trained on scans from NFLASH230 only. For both datasets, case 1 does not offer any improvement at all. On the other hand, models from case 2 improve the pointing accuracy for most numbers of selected features k .

First, the results from models predicting offsets for all instruments. Adding complexity seems to worsen the performance of azimuth models while improving the performance of elevation models. The best performance for azimuth models is for $k = 2$ with $\bar{r}_{RMS} = 0.980$ and $\sigma_{\bar{r}} = 0.059$ which is a slight improvement. For elevation models, the best performance is for $k = 50$ with $\bar{r}_{RMS} = 0.955$ and $\sigma_{\bar{r}} = 0.029$.

Second, the results from models predicting offsets for NFLASH230 only. The best models for azimuth are $k = 2$ and $k = 50$ with similar performance $\bar{r}_{RMS} = 0.948$ and $\bar{r}_{RMS} = 0.945$ respectively. For elevation models, $k = 50$ offers the best performance, with $\bar{r}_{RMS} = 0.940$. The standard deviations for these performance estimates are small.

We also conducted the same experiment for two other datasets, with the offsets and pointing corrections transformed to simulate a pointing correction after every scan. We tested splitting the training and validation data completely randomly for all four datasets, unlike the presented results, where we dedicate a day to either training or validation. For these results, see Appendix A (8).

Table 6.4: *tmp2022_clean_clf_results_table* Resulting RMS from Case 1 and 2 for XGBoost model predicting pointing offset. The dataset used to get these results contain all scans and is cleaned using the regular criteria and the XGBoost classifier. The training and validation data is split on days, meaning that all the scans for a given day are in the training or validation set and not both. The test set is unaffected by this.

k	Case 1				Case 2			
	Azimuth		Elevation		Azimuth		Elevation	
	Mean	STD	Mean	STD	Mean	STD	Mean	STD
2	1.007	0.003	1.232	0.055	0.980	0.059	0.964	0.016
5	1.003	0.003	1.170	0.028	0.990	0.067	0.964	0.016
10	1.288	0.101	1.116	0.015	1.001	0.102	0.979	0.059
20	1.580	0.082	1.121	0.023	1.018	0.130	0.971	0.036
30	1.606	0.110	1.107	0.018	1.026	0.151	0.957	0.018
40	1.528	0.111	1.068	0.010	1.026	0.137	0.973	0.044
50	1.758	0.121	1.061	0.027	1.018	0.114	0.955	0.029

Table ?? show the performane of the pointing correction model if we choose always choose the model with the lowest validation loss on and use that model for the test set. It should the performance estimate and standard deviation for the two different cases, and for different datasets. The data training and validation data is split on whole days, and the validation size is 0.43% of the training/validation set. This does not offer any improvement over the result presented above, but for an NFLASH230 model, this still proves to be a viable strategy. The performance using

Table 6.5: *tmp2022_clean_clf_nflash230_results_table* Resulting RMS from Case 1 and 2 for XGBoost model predicting pointing offset. The dataset used to get these results contain only NFLASH230 and is cleaned using the regular criteria and the XGBoost classifier. The training and validation data is split on days, meaning that all the scans for a given day are in the training or validation set and not both. The test set is unaffected by this.

k	Case 1				Case 2			
	Azimuth		Elevation		Azimuth		Elevation	
	Mean	STD	Mean	STD	Mean	STD	Mean	STD
2	0.982	0.014	1.020	0.024	0.948	0.056	0.972	0.081
5	1.366	0.077	1.198	0.034	0.983	0.142	0.953	0.097
10	1.383	0.087	1.155	0.047	0.957	0.080	0.967	0.087
20	1.252	0.119	1.126	0.071	0.972	0.131	0.949	0.069
30	1.335	0.226	1.094	0.041	0.963	0.093	0.959	0.077
40	1.146	0.036	1.058	0.020	0.961	0.089	0.948	0.077
50	1.202	0.131	1.062	0.022	0.945	0.073	0.940	0.075

this strategy for NFLASH230 is $\bar{r}_{RMS} = 0.958$ for azimuth and $\bar{r}_{RMS} = 0.941$ for elevation. Standard deviations are also small.

For a list of the 50 features with the most mutual information to the target variable, see ?? in the Appendix 8.

Table 6.6: Performance when choosing min validation for each fold. Train/val split on days. Test size 0.43.

Dataset	Case 1				Case 2			
	Azimuth		Elevation		Azimuth		Elevation	
	Mean	STD	Mean	STD	Mean	STD	Mean	STD
Transformed	2.140	0.498	1.073	0.040	1.008	0.073	0.949	0.022
Transformed NF230	1.242	0.047	1.006	0.009	0.999	0.096	1.074	0.201
Regular	1.730	0.126	1.170	0.028	1.016	0.114	0.994	0.065
Regular N230	1.251	0.131	1.198	0.033	0.958	0.055	0.941	0.079

Chapter 7

Discussion

Chapter 8

Conclusion

Appendices

Appendix A

Table 1: Performance when choosing min validation for each fold. Train/val split on days. Test size 0.5.

Dataset	Split on days		Random splits	
	Azimuth	Elevation	Azimuth	Elevation
Transformed	2.016	1.067	1.050	0.926
Transformed NF230	1.243	1.009	0.954	1.008
Regular	1.779	1.186	1.027	0.951
Regular N230	1.204	1.262	0.930	0.906

Table 2: All

Target	Fold	Colsample by tree	γ	η	Max Depth	Min child weight	Number of estimators	λ	Subsample
Az	1	0.536	0.013	0.018	5	8	269	1	0.699
	2	0.526	0.013	0.010	1	10	393	0.175	0.756
	3	0.970	0.063	0.023	2	2	460	0.071	0.800
	4	0.501	0.008	0.019	3	2	400	0.578	0.979
	5	0.530	0.031	0.187	2	3	69	0.277	0.931
	6	0.668	0.056	0.139	2	2	77	0.426	0.783
El	1	0.577	0.247	0.017	5	1	111	0.966	0.517
	2	0.645	0.078	0.063	5	2	33	0.010	0.787
	3	0.600	0.017	0.009	5	5	86	0.660	0.598
	4	0.608	0.019	0.037	5	10	368	0.329	0.600
	5	0.999	0.031	0.022	4	1	427	0.178	0.716
	6	0.788	0.369	0.097	2	1	354	0.552	0.796

Appendix B

.1 Transformation of pointing offsets and corrections

Table 5 show examples of pointing offsets and corrections applied during the pointing scans. The column "Original" show raw pointing scan data. The pointing corrections

Table 3: NFLASH230

Target	Fold	Colsample by tree	γ	η	Max Depth	Min child weight	Number of estimators	λ	Subsample
Az	1	0.720	0.125	0.126	1	4	201	0.247	0.864
	2	0.777	0.988	0.017	1	6	468	0.763	0.962
	3	0.517	0.298	0.014	2	7	470	0.509	0.762
	4	0.972	0.773	0.116	5	1	30	0.006	0.802
	5	0.897	0.112	0.008	5	10	435	0.658	0.819
	6	0.935	0.011	0.064	1	6	224	0.228	0.748
El	1	0.600	0.156	0.007	5	2	140	0.047	0.792
	2	0.999	0.148	0.029	5	1	38	0.998	0.580
	3	0.955	0.237	0.014	3	6	210	0.613	0.712
	4	0.537	0.014	0.034	2	10	352	0.537	0.594
	5	0.686	0.324	0.081	1	1	371	0.368	0.966
	6	0.862	0.306	0.184	1	10	157	0.029	0.701

Table 4: Validation and test performance for Case 2, all instruments left and nflash230 right. Performance when choosing the number of features showing best performance.

Target	Fold	Val RMS 1	Test RMS 1	Val RMS 2	Test RMS 2
Az	1	0.915	1.084	0.846	1.043
	2	0.971	0.928	0.886	0.953
	3	0.927	1.011	0.928	0.872
	4	0.945	0.952	0.882	0.962
	5	0.933	0.972	0.898	0.938
	6	0.937	0.935	0.930	0.923
El	1	0.964	0.975	0.916	1.014
	2	0.930	1.000	0.889	0.973
	3	0.966	0.957	0.886	1.025
	4	0.822	0.922	0.839	0.839
	5	0.830	0.934	0.802	0.888
	6	0.836	0.941	0.827	0.901

ca and ie are normally updated according to equations (??) and (??), as shown in the first two rows of the table. However, observers may choose not to update the pointing, particularly when the pointing offset is small, as illustrated in the consecutive row of the table. In other cases, the corrections may be updated but not according to equations (??) and (??). This can occur when a new science project is loaded and the pointing correction from the previous time that project was used is applied. These factors introduce several challenges for the training of a model:

- ca and ie should represent the optimal correction using all the information we have about the current state of a system. If we do not update the corrections, there is some information about the system (the previously observed pointing offset) that the model is not receiving.
- Some features are constructed as the change in variables since the last correction. If the corrections are not updated, this interval is longer than if they were, and the resulting features could be more prone to uncertainties and noise. A problem with the integration also occurs if the corrections are not updated according to the equations (??) and (??). Then, we do not know when those corrections represent the system, resulting in inaccurate features.

A possible solution to this problem is a two step procedure where we first transform the offsets and corrections to represent the system at the most recent pointing scan, and second use these transformed offsets as training labels and the transformed corrections as training inputs.

In practice, we do this by assuming the corrections ca and ie are updated after every pointing scan. This changes the correction applied during the next pointing scan, which further affects the observed pointing offset. This effect propagates throughout the whole dataset. The following formulas

$$\tilde{ca}_i = \tilde{ca}_{i-1} + \tilde{\delta}_{az,i-1} \quad (1)$$

$$\tilde{ie}_i = \tilde{ie}_{i-1} - \tilde{\delta}_{el,i-1} \quad (2)$$

$$\tilde{\delta}_{az,i} = \delta_{az,i} + ca_i - \tilde{ca}_i \quad (3)$$

$$\tilde{\delta}_{el,i} = \delta_{el,i} - ie_i + \tilde{ie}_i \quad (4)$$

Where the " \sim " denotes a transformed variable. Using these transformations, the corrections used for training and the resulting offset are similar to the ones that would be observed if the corrections were made according to the equations (??) and (??) after every pointing scan. The column "Transformed" in the table shows the transformed variables.

Table 5: **Original:** Example from the dataset of the observed pointing offsets and the corrections applied during the pointing scan. **Transformed:** Pointing offsets and corrections according to equations (1), (2), (3), and (4).

i	Original				Transformed			
	δ_{az}	δ_{el}	ca	ie	$\tilde{\delta}_{az}$	$\tilde{\delta}_{el}$	\tilde{ca}	\tilde{ie}
1	1.2	0.1	2.1	1.7	1.2	0.1	2.1	1.7
2	0.0	0.5	3.3	1.6	0.0	0.5	3.3	1.6
3	-1.1	0.0	3.3	1.6	-1.1	-0.5	3.3	1.1
4	0.6	0.7	2.2	1.6	0.7	0.7	2.2	1.6
5	0.9	1.4	2.2	1.6	0.2	0.7	2.8	0.9
6	1.0	1.1	2.2	1.6	0.1	-0.3	3.1	0.2
7	-0.9	1.2	3.1	0.5	-1.0	1.3	3.2	0.5
8	0.5	1.5	2.2	-0.7	0.5	1.4	2.2	-0.7
9	-0.3	0.4	2.2	-0.7	-0.8	-1.1	2.7	-2.2

Bibliography

- [1] Tianping Chen and Hong Chen. Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. *IEEE Transactions on Neural Networks*, 6(4):911–917, 1995.
- [2] Ian Covert. Understanding shap and sage. <https://iancovert.com/blog/understanding-shap-sage/>, 2020.
- [3] Ian Covert, Scott Lundberg, and Su-In Lee. Understanding global feature contributions with additive importance measures, 2020.
- [4] Ian Covert, Scott Lundberg, and Su-In Lee. Understanding global feature contributions with additive importance measures, 2020.
- [5] Scott Lundberg and Su-In Lee. A unified approach to interpreting model predictions, 2017.
- [6] Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre G.R. Day, Clint Richardson, Charles K. Fisher, and David J. Schwab. A high-bias, low-variance introduction to machine learning for physicists. *Physics Reports*, 810:1–124, may 2019.
- [7] Vishal Morde. Xgboost algorithm: Long she may rein. Downloaded January 2023.
- [8] Brandon Rhodes. Ephem, 2021. <https://pypi.org/project/ephem/>.
- [9] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.
- [10] Lloyd Shapley. A value for n-person games. *Contributions to the Theory of Games*, 2:307–317, 1953.
- [11] Till Tantau. The PGF/TikZ manual, 2021.