ML@NOVA: OverfittingScientists

SEMI-SUPERVISED LEARNING FOR PREDICTING SURVIVAL TIME IN MULTIPLE MYELOMA PATIENTS

Team identification

Name 1: Filipe Colla David

Number 1: 70666

Name 2: Miguel Fontes

Number 2: 55119

Final score: 2.45798

Leaderboard private ranking: 11

Task 1 – Setting the Baseline

Index:

Task 1.0
Data Analysis

Dataset Description

- The dataset given to us in this project consists of simulated clinical data for multiple myeloma patients.
- It mirrors common real-world scenarios by including incomplete information, represented as *NaN* values.
- Just like we did in the previous project, we started by exploring the features and their meaning in the overall scheme of the theme.
- We began by analyzing the 9 features provided to us and categorizing them into 7 potential input features and 1 essential label.

Feature Selection and Analysis

Throughout this project, we will use the following features and label as a basis to create our models:

Features:

- Age (Integer)
  - Age of the patient;
- Gender (Binary)
  - Biological sex of the patient;
- Stage (Integer 1-4)
  - Extent of the cancer – raging from 1 (less severe) to 4 (very serious)
- Genetic Risk (Real 0-1)
  - Combined information on a patient's genetic cancer risk – raging between 0 and 1;
- Treatment Type (Binary)
  - Type of treatment administered;
- Comorbidity Index (Integer)
  - Quantifies the number and severity of additional comorbid conditions – 0 indicates no comorbid conditions;
- Treatment Response (Binary)
  - Effectiveness of the treatment administered – 0 indicates a poor or inadequate response;

Label and Censored Columns

After separating the input features, we identified 1 label and 1 feature that, although not used in the models, provide crucial insights into the overall dataset provided to us:

Label:

- Survival Time (Continuous)
  - Duration from the start of the study or treatment to the event of interest
    - Event of interest can be viewed as death or until the last follow-up with the patient.

Extra feature:

- Censored (Binary)
  - Serves as an indicator and informs us whether the survival time is censored
    - 1 indicates that the patient either gave up or survived beyond the study's end
    - 0 indicates that the patient was recorded to have died during the study

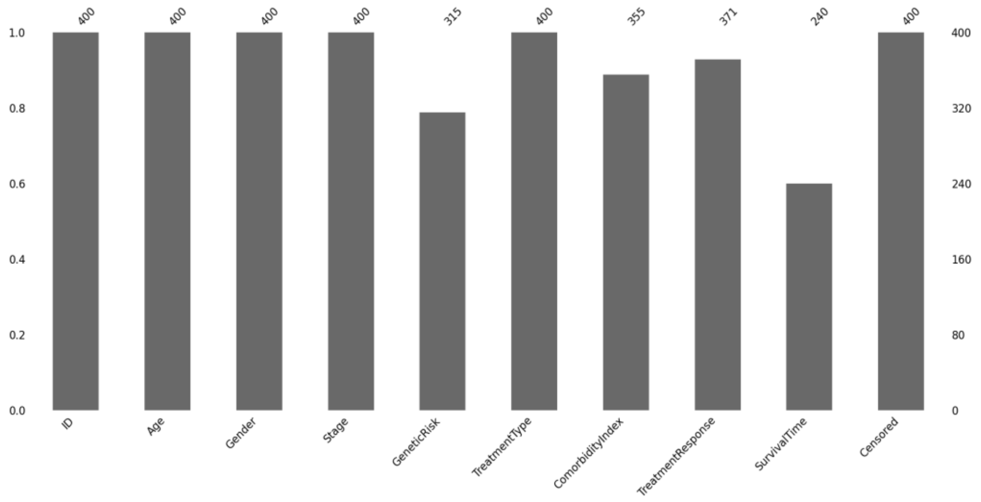
In this project, the data is right-censored, meaning that a data point—in this case, Survival Time—can exceed a certain value, but the exact amount is unknown and is therefore considered censored.

Missing Data

The Dataset for this project contains missing data points, represented as *NaN*.

We began by taking note that the dataset contains information on a total of 400 patients.

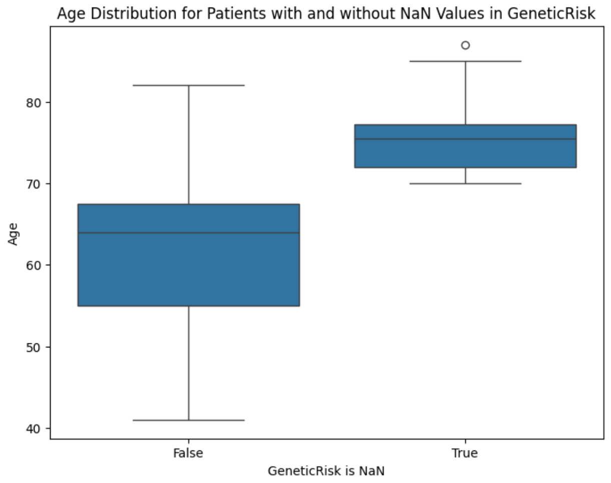
From this plot, we can observe that:
- "**Genetic Risk**" has **85** missing values
- "**Comorbity Index**" has **45** missing values
- "**Treatment Response**" has **29** missing values
- "**Survival Time**" has **160** missing values

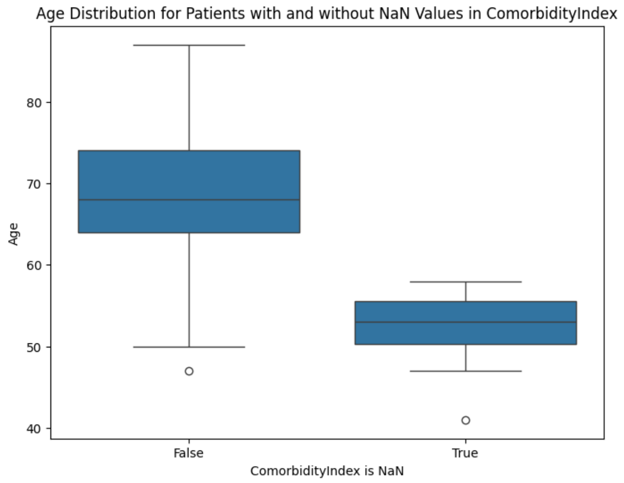


More on Missing Data

We can also observe some interesting correlation between missing data and the age of the patients:

- Mostly older patients don't have the Genetic Risk test defined.
- Mostly younger patients don't have the Comorbidity Index defined.

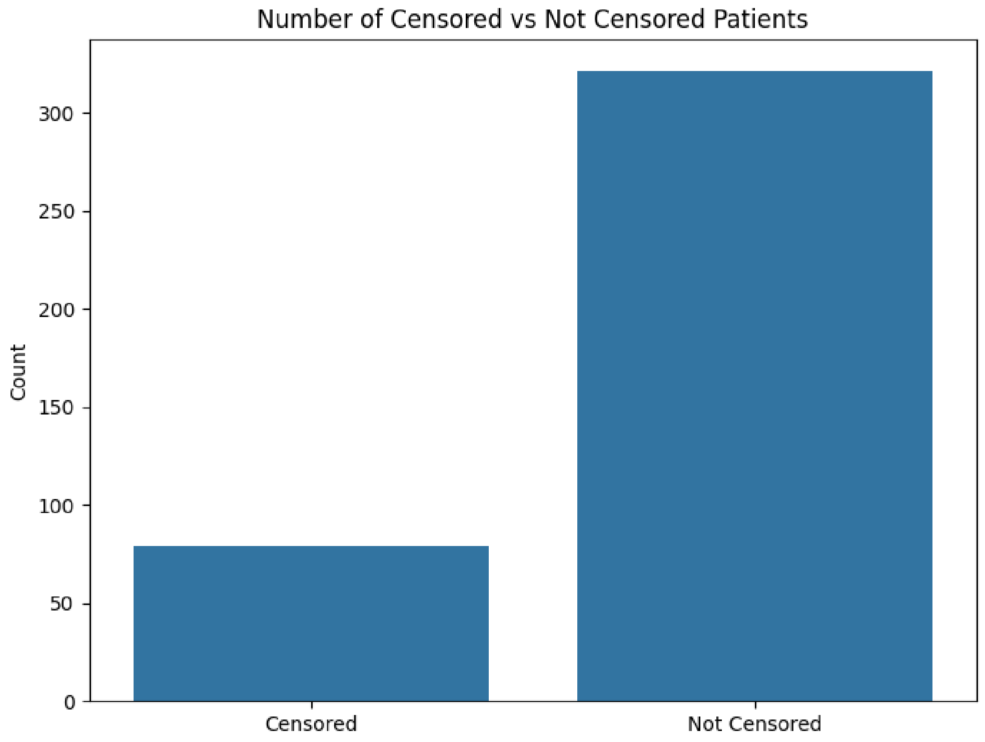




Censored Values

A censored value represents a data point, from which the outcome from the study is partially known.

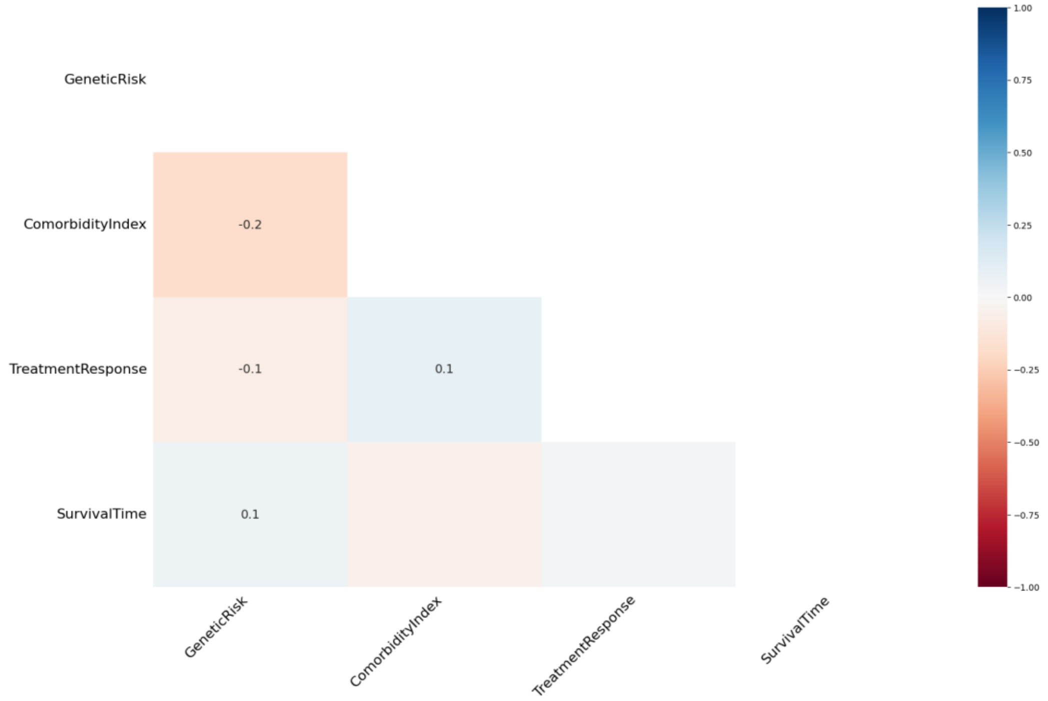
Analysis and training with censored data must be done with some bias mitigation, as some values might not represent the true outcome of the patient.



Missingness Features Correlation

In the following plot, we can see that Genetic Risk shares some correlation with Survival Time, Treatment Response and Comorbidity Index, even if almost minimal.

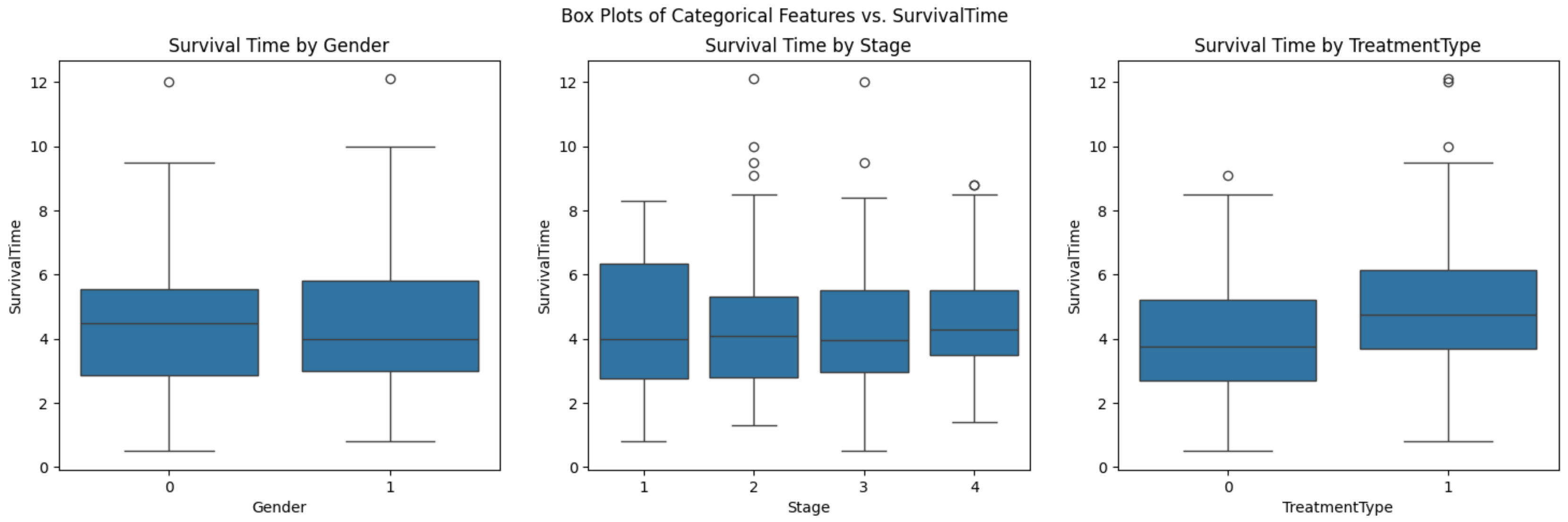
The low values of correlation indicates that the missingness[1] is most likely random and unrelated between the features.



1. Missingness refers to the absence of data points or values in a dataset

More on Features Correlation (Categorical)

It's also valuable to look at the correlation between categorical features, such as Gender, Stage, and Treatment Type with the Survival Time. As mentioned previously, when dealing with Survival Time some of the data points are censored, additional attention is required to not miss interpret the data.

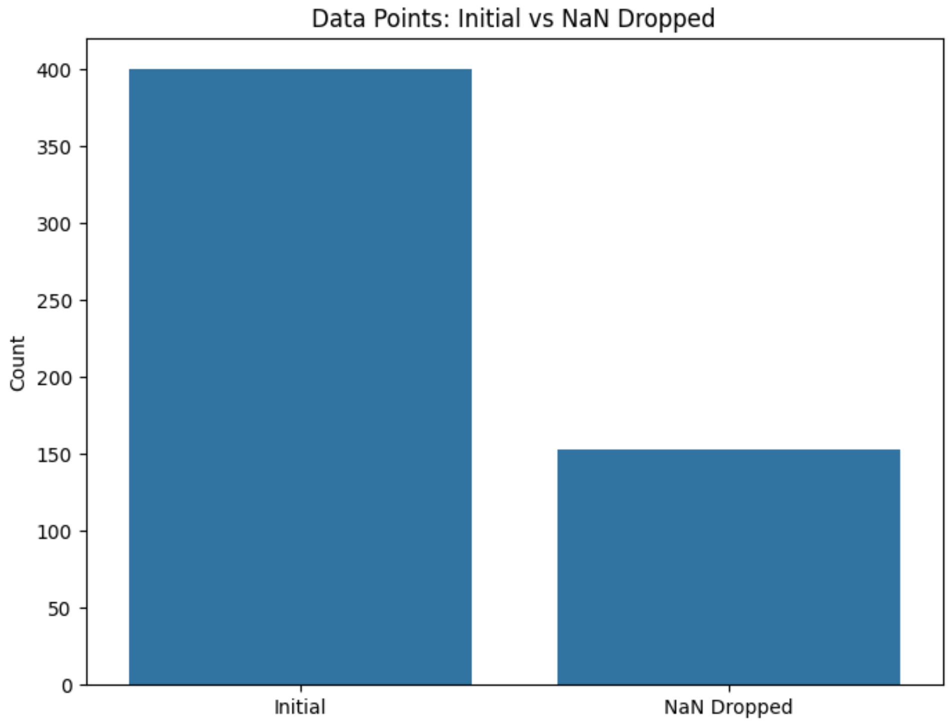


Task 1.1
Data Preparation and Validation Pipeline

Dropping Missing Points

One valid strategy, to handle missing data, would be to drop all missing data entries. (More on additional strategies in future slides)

After dropping the *NaN* values, the dataset has just **153** points, which Is considerably lower than the initial **400**.

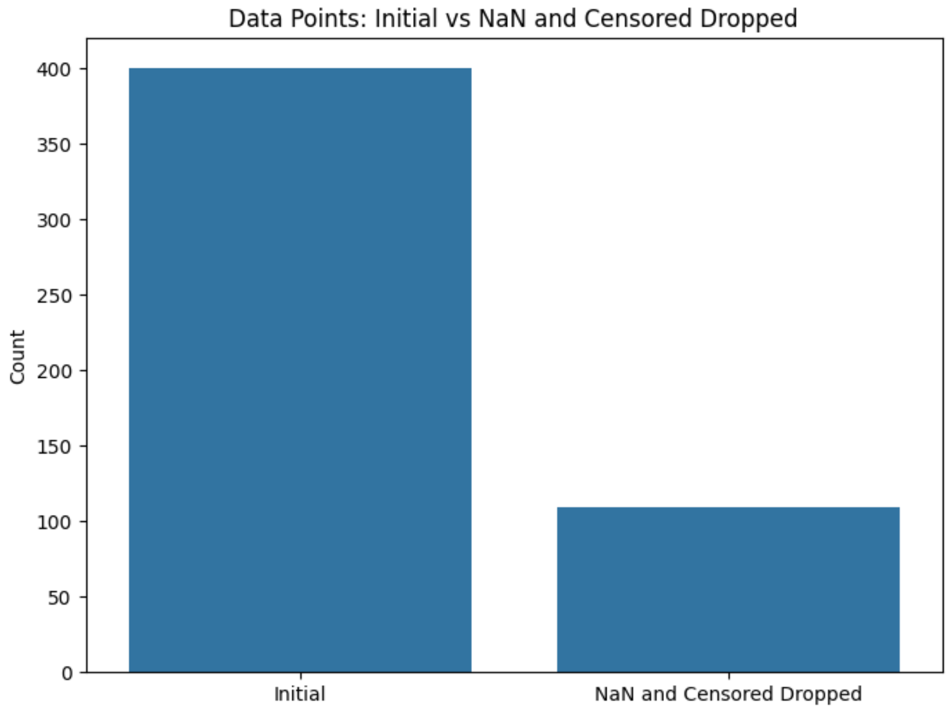


Dropping Censored Points

As mentioned previously, censored points are not a true representation of the experiment.

One valid strategy to handle this issue, would be to drop these points, alongside with the *NaN* entries.

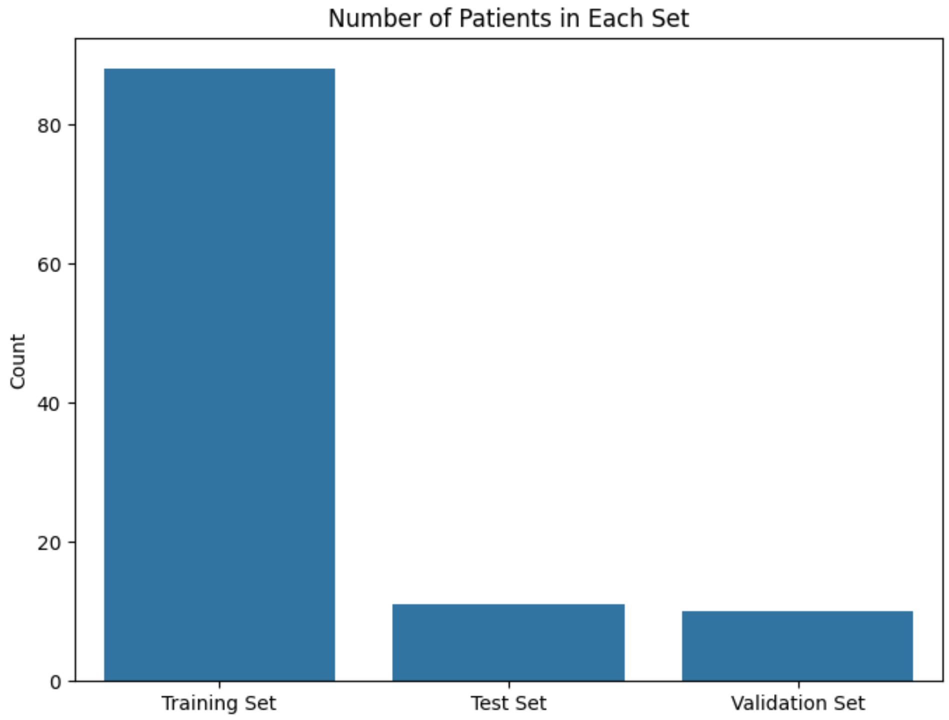
With this approach we are left with 109 points.



Fitting a baseline model

While it is technically possible to proceed with a 80/10/10 split-further dividing an already small dataset.

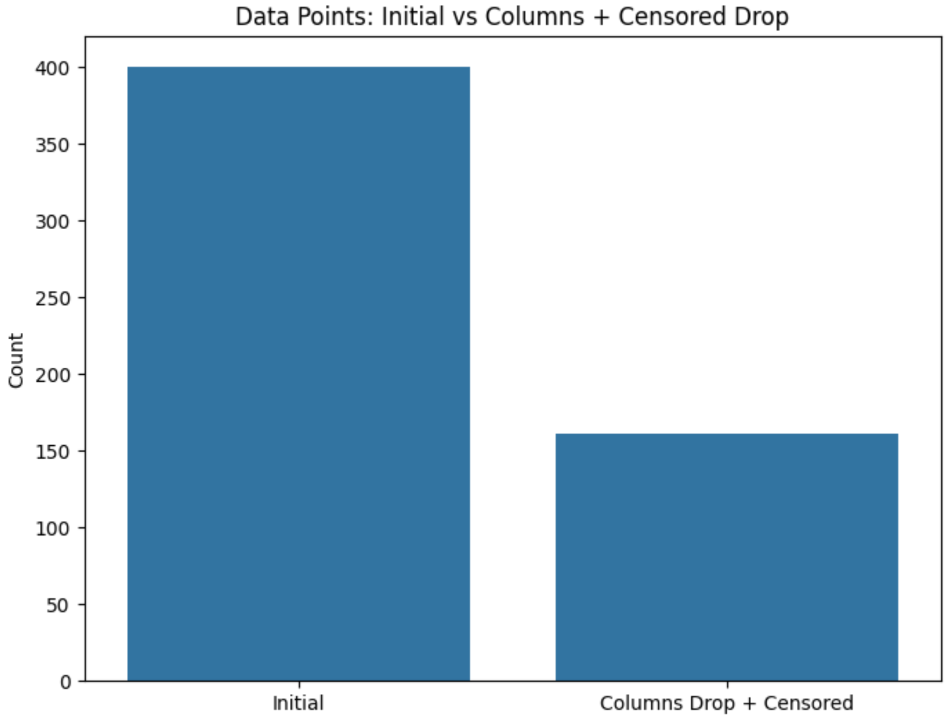
This approach will leave only 88 data points for training



Dropping Features with Missing Points

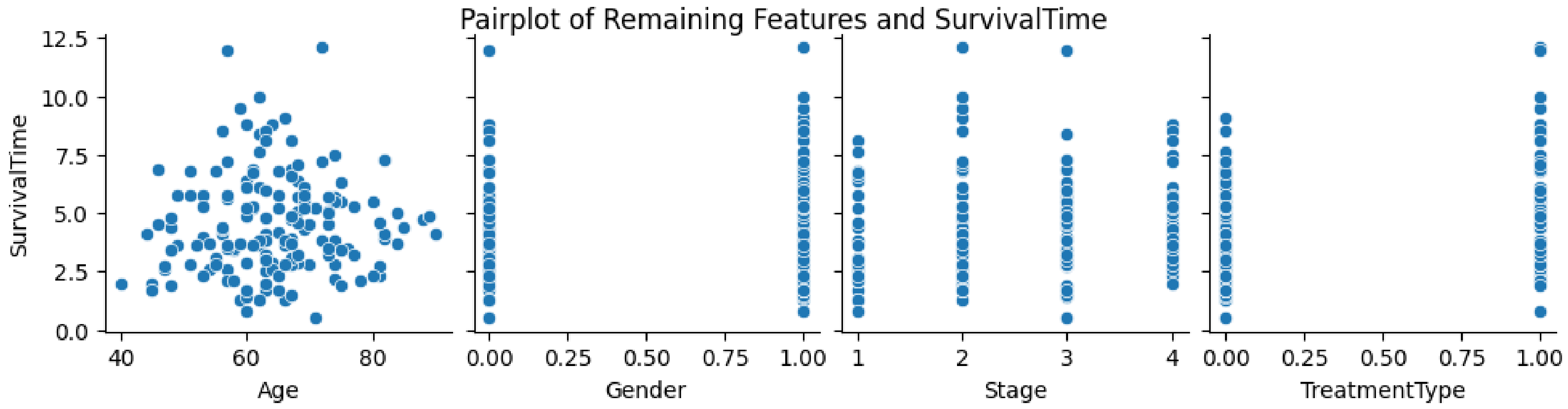
In contrast with the approach previously mentioned, another valid one would be to drop the columns that contain missing data and then remove the censored data points.

After removing the features columns with missing values, which in this case are GeneticRisk, ComorbidityIndex and TreatmentResponse, and then removing the Censored data points, the dataset was reduced to 161 data points, resulting in a loss of nearly almost 60% (59.75%) of the data.



Pair plot between remaining features and target variable

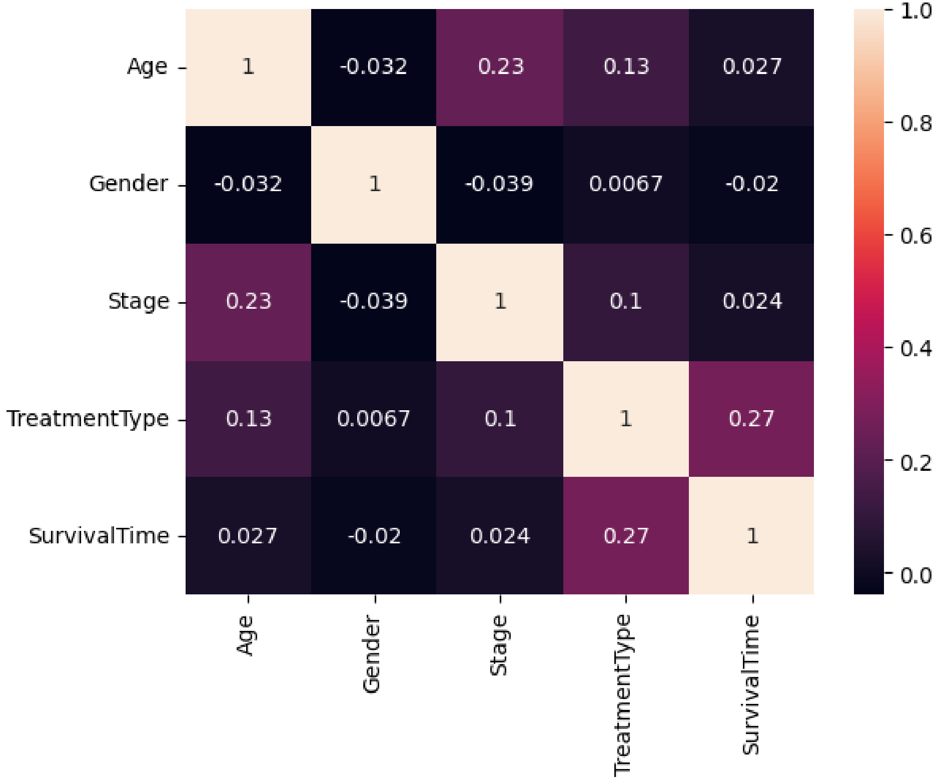
Some of these analysis can be found the Task 1.0 analysis.
In Summary, there's little correlation between variables.



Heatmap between remaining features and target variable

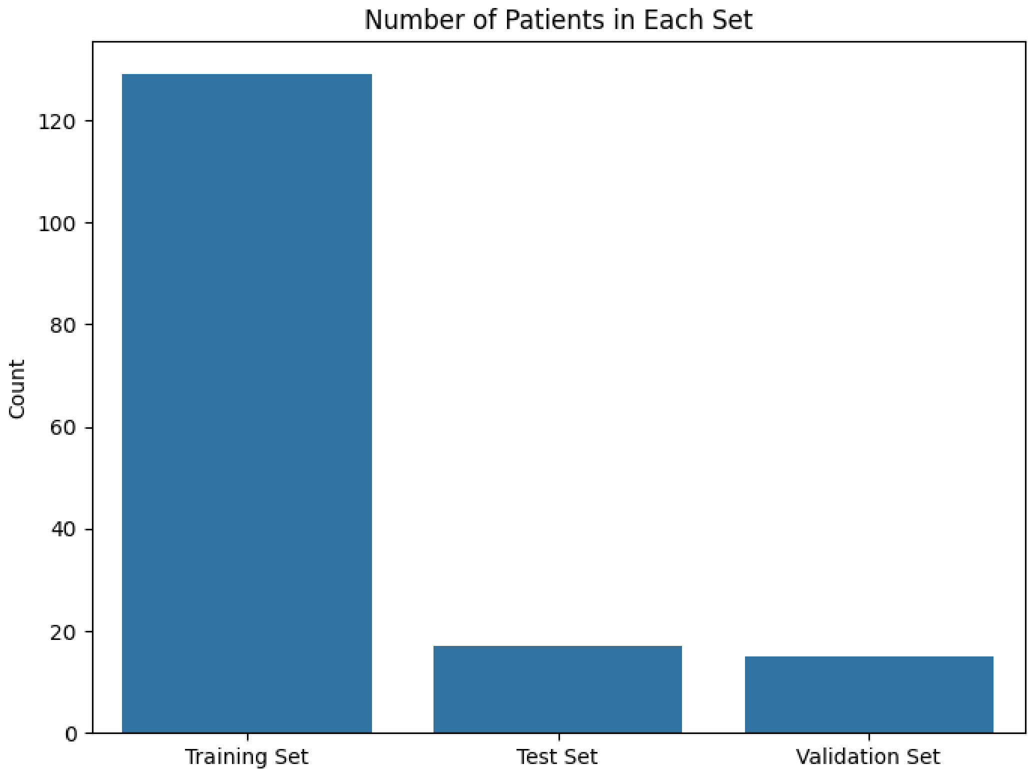
To further reinforce what was analyzed in the previous slide, we can see that there is correlation, even if small, between the feature TreatmentType and SurvivalTime. This means we can hypothesize that a patient that goes through with the treatment, is likely to survive longer, even if it's just a little bit more.

Not only do we have a small correlation between those 2 but also between 2 features: Age and Stage. This could indicate that, as a patient grows old, there's a possibility that the stage of the cancer to also grow in severity



Fitting a baseline model

As explained earlier (slide 16), while an 80/10/10 split allows for training on data from 129 patients, reserving 10% of the data for validation further reduces the amount available for training the model, potentially leading to an underperforming model.

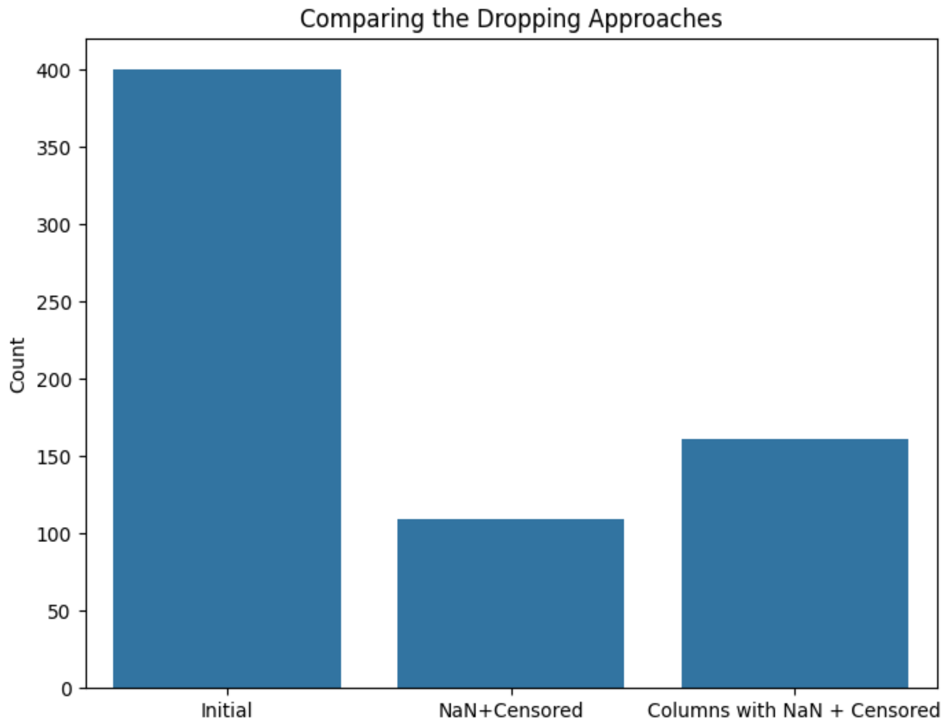


Dropping Approaches

Removing the features with NaN values, in addition to Censored data points, allows us to retain more data points and, therefore, more information overall.

A potential drawback with these approaches is that we are removing crucial information that is likely valuable for the models we aim to create.

In summary, these might not be the most effective and ideal approaches, because when creating a model, we should aim to maximize the number of available data points.

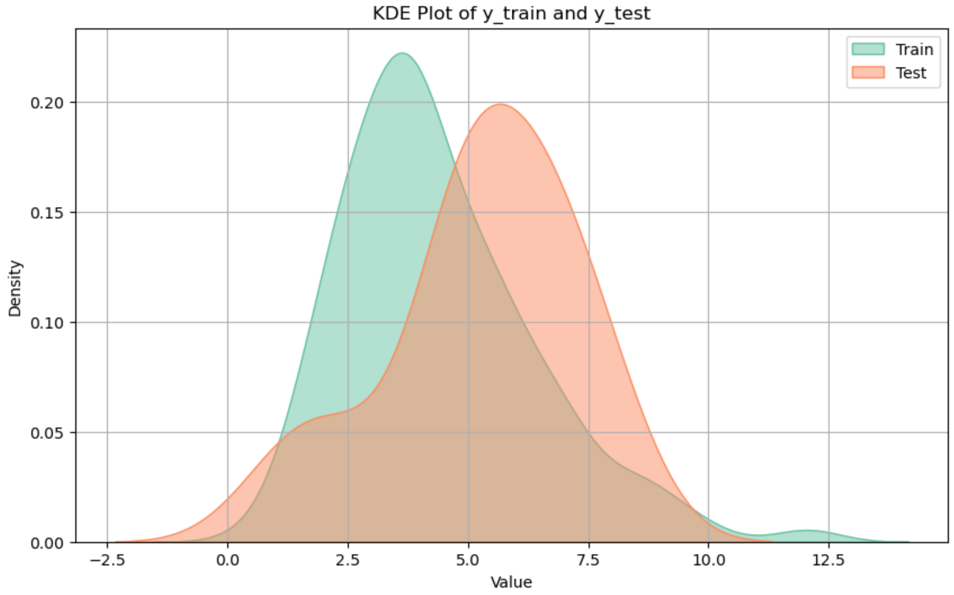


Data splitting

When doing a data split for train and testing, it's important that we have a similar distribution of the target entries across splits, as to avoid having a biased model trained on a subset of data that doesn't follow a similar distribution of the overall data.

When doing the first submission for our baseline model, we've noticed that the values were different from excepted, which led us to believe that our validating strategy was not correct.

In this particular dataset, we've noticed this discrepancy, when doing a normal test-train split, as it can be seen on the KDE plot (Kernal Density Estimate).

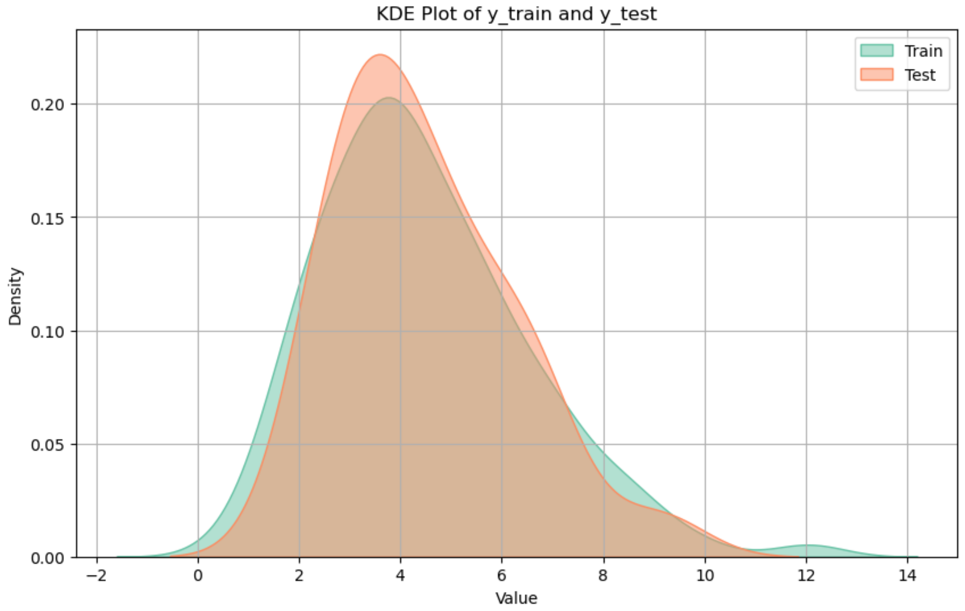


Stratified Split

Like we demonstrated in the previous slide, a normal train-test split approach would not work for this small dataset, since it's very hard for a random split to get a similar distribution on both sets.

A Stratified data split allows us to create both train/test sets while still maintaining the distribution of not only the target feature, but also the censored data and the missing values (NaNs).

In this KDE (Kernel Density Estimate) plot, we can observe that our stratified split maintains the same distribution on both tests, even using the same percentage as the previous split.

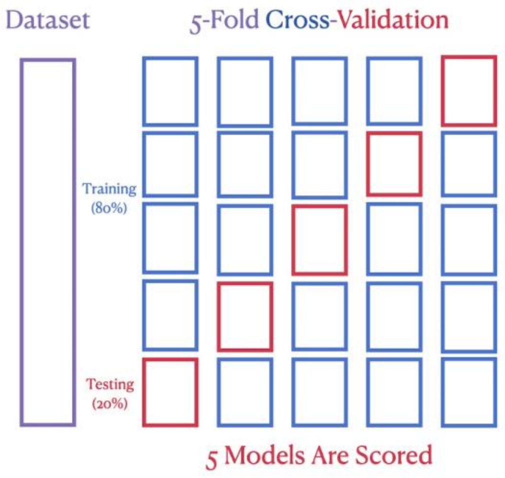


Hyperparameter Tunning

To validate the results of our hyperparameter tunning, and avoid data leakage, we used K-Fold Cross Validation.

K-Fold Cross Validation is a technique used for hyperparameter tuning, where the given dataset is divided into 'k' subsets (folds) to evaluate model performance more reliably. In each iteration, k-1 folds are used for training, and the remaining fold is used for validation.

This process is repeated k times, ensuring every fold is used for validation once.



Task 1.2
Learn the Baseline Model

Baseline Pipeline Preparation

- We made a method that created and fitted a baseline pipeline and returned it in the end.
- We initially create a baseline pipeline that incorporates a Standard Scaler to normalize the values in each feature, followed by a Linear Regression model as the chosen regressor, since this is the baseline pipeline.
- The pipeline was then trained using the training set defined earlier, ensuring consistent and comparable results when applying the same train and test sets to future models.
- Since the baseline pipeline uses Linear Regression, we didn't run any hyperparameter tunning.

Prediction using Baseline Pipeline

Using the prepared baseline pipeline, we made predictions on the test set and, since we only used uncensored data to train the model, calculated the error using MSE as the loss function. The results were as follows:

Max Error	Min Error	Mean Error	Std Dev of Error	cMSE
4.961	0.171	1.544	2.107	4.445

Given the small and limited number of data points available, the baseline model performed better than initially expected, This could be due to the limited number of data points used for training the model.

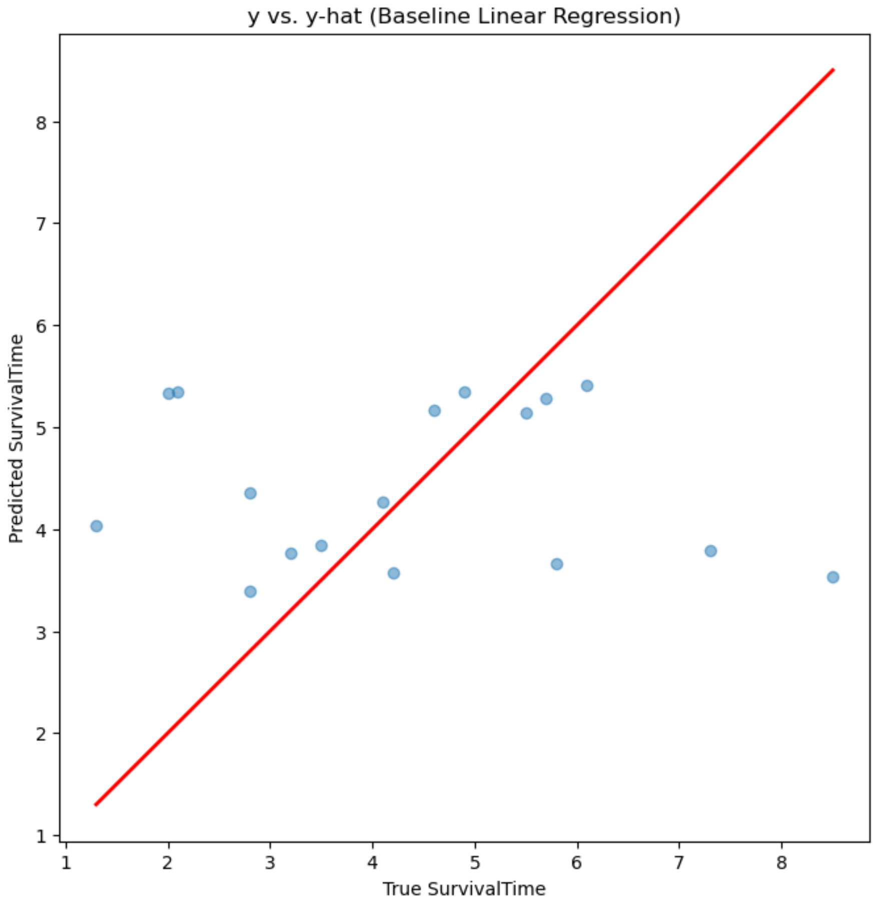
Using the robust data split explained in the previous slides, we achieved a relative error of approximately 10%. This indicates that the difference between the local cMSE and the Kaggle cMSE is around 10%, despite the limitations of the small dataset available to us.

y-yhat Plot Analysis

By analyzing this plot, we can confidently state that the baseline model performs reasonably well, especially considering that we used a Linear Regression to achieve these results. This reinforces the points made in the previous slide.

It is important to note that the presence of even a single outlier (data points significantly deviating from the "perfect" red line) can have a bigger negative impact on the model's performance compared to points closer to the line.

This effect is amplified by the small number of data points in our test set, as such outliers can significantly skew key metrics like the mean and standard deviation, as highlighted in the previous slide.



Task 1.3
Learn with the cMSE

cMSE (Censored Mean Squared Error)

A censored value represents a data point, from which the outcome from the study is partially known.

When training the models with censored values, it's important to try account for the uncertainty associated with these values.

To mitigate these uncertainties, we use the cMSE as the loss function to minimize when training the model.

The cMSE is defined by:

$$L(\alpha) = \frac{1}{N}\sum_{i=1}^{N}\Big((1-c)(y_i-\hat{y}_i)^2 + c\max(0, y_i-\hat{y}_i)^2\Big)$$

Where:

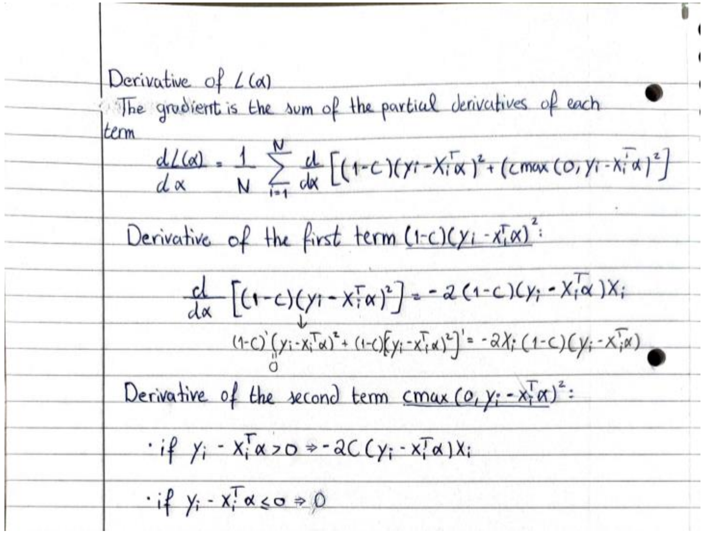
$\hat{y}_i = X_i^\top \alpha$ is the predicted values vector

$X \in \mathbb{R}^{d\times N}$ is the feature matrix.

$\alpha \in \mathbb{R}^d$ is the weight vector.

$y \in \mathbb{R}^N$ is the target vector.

Step by step formulation of the derivative



Derivative of $L(\alpha)$

The gradient is the sum of the partial derivatives of each term

$$\frac{dL(\alpha)}{d\alpha} = \frac{1}{N}\sum_{i=1}^{N}\frac{d}{d\alpha}\left[(1-c)(y_i - X_i^T\alpha)^2 + (c\max(0, y_i - x_i^T\alpha)^2\right]$$

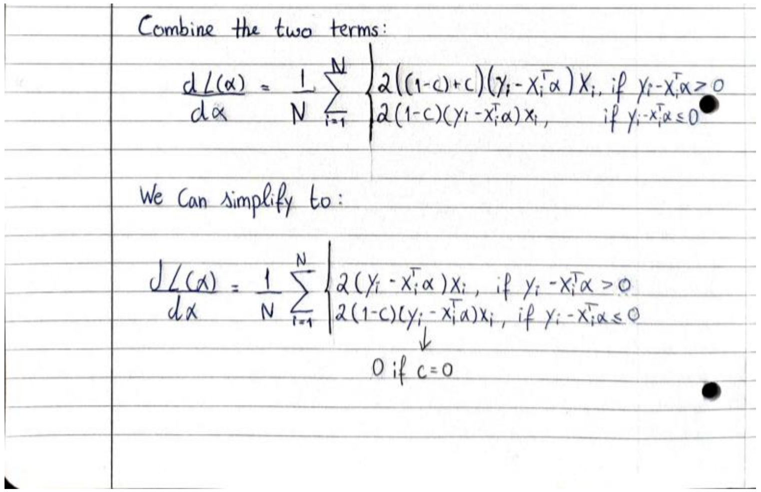
Derivative of the first term $\underline{(1-c)(y_i - x_i^T\alpha)}^2$:

$$\frac{d}{d\alpha}\left[(1-c)\underset{\downarrow}{(}y_i - x_i^T\alpha)^2\right] = -2(1-c)(y_i - X_i^T\alpha)X_i$$

$$\underset{\overset{\shortparallel}{0}}{(1-c)'}(y_i - x_i^T\alpha)^2 + (1-c)\left[(y_i - x_i^T\alpha)^2\right]' = -2X_i(1-c)(y_i - x_i^T\alpha)$$

Derivative of the second term $\underline{c\max(0, y_i - x_i^T\alpha)}^2$:

- if $y_i - x_i^T\alpha > 0 \Rightarrow -2C(y_i - x_i^T\alpha)X_i$
- if $y_i - x_i^T\alpha \leq 0 \Rightarrow 0$



Combine the two terms:

$$\frac{dL(\alpha)}{d\alpha} = \frac{1}{N}\sum_{i=1}^{N}\begin{cases} 2((1-c)+c)(y_i - X_i^T\alpha)X_i, & \text{if } y_i - X_i^T\alpha \geq 0 \\ 2(1-c)(y_i - x_i^T\alpha)x_i, & \text{if } y_i - x_i^T\alpha \leq 0 \end{cases}$$

We Can simplify to:

$$\frac{dL(\alpha)}{d\alpha} = \frac{1}{N}\sum_{i=1}^{N}\begin{cases} 2(y_i - X_i^T\alpha)X_i, & \text{if } y_i - X_i^T\alpha > 0 \\ \underset{\substack{\downarrow \\ 0 \text{ if } c=0}}{2(1-c)(y_i - x_i^T\alpha)x_i}, & \text{if } y_i - x_i^T\alpha \leq 0 \end{cases}$$

Derivative of the cMSE

$$\frac{\partial L}{\partial \alpha} = \frac{1}{N} \sum_{i=1}^{N} \begin{cases} 2(y_i - X_i^\top \alpha) X_i, & \text{if } y_i - X_i^\top \alpha > 0 \\ 2(1-c)(y_i - X_i^\top \alpha) X_i, & \text{if } y_i - X_i^\top \alpha \leq 0 \end{cases}$$

Where:

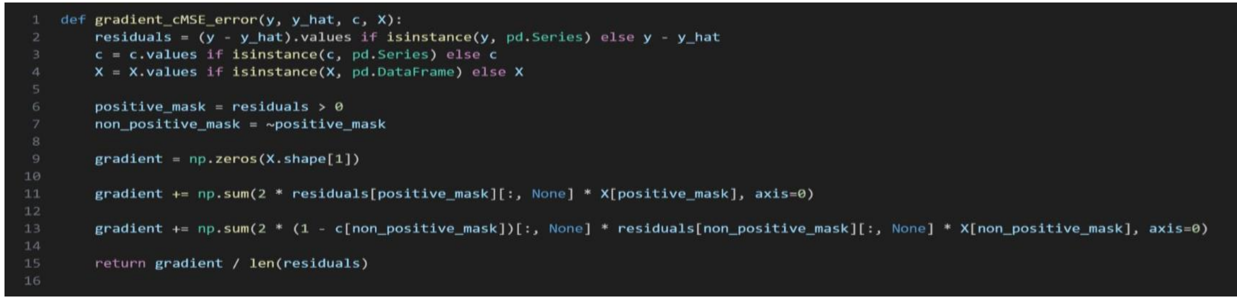
$\frac{\partial L}{\partial \alpha}$ is the partial derivative of the Loss Function for the weight vector α

$\hat{y}_i = X_i^\top \alpha$ is the predicted values vector

$X \in \mathbb{R}^{d \times N}$ is the feature matrix.

$y \in \mathbb{R}^N$ is the target vector.

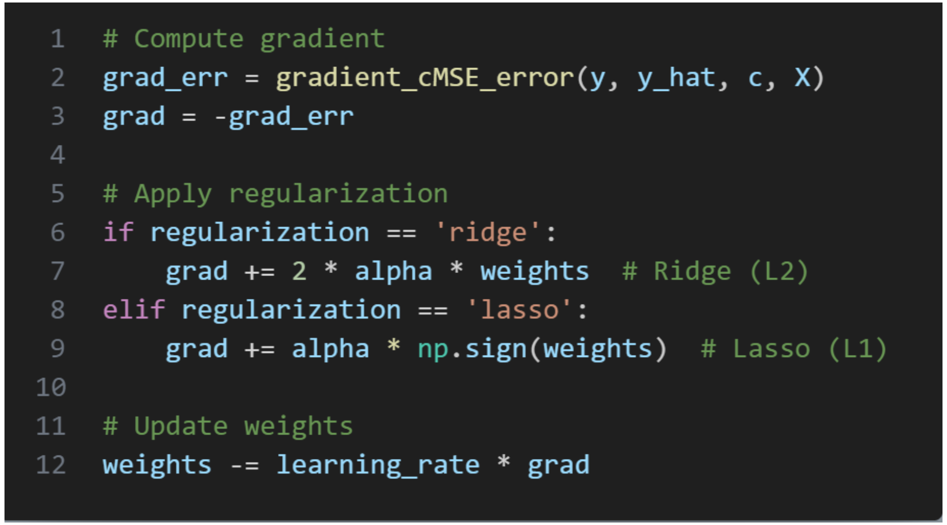
$\alpha \in \mathbb{R}^d$ is the weight vector.



```python
def gradient_cMSE_error(y, y_hat, c, X):
    residuals = (y - y_hat).values if isinstance(y, pd.Series) else y - y_hat
    c = c.values if isinstance(c, pd.Series) else c
    X = X.values if isinstance(X, pd.DataFrame) else X

    positive_mask = residuals > 0
    non_positive_mask = ~positive_mask

    gradient = np.zeros(X.shape[1])

    gradient += np.sum(2 * residuals[positive_mask][:, None] * X[positive_mask], axis=0)

    gradient += np.sum(2 * (1 - c[non_positive_mask])[:, None] * residuals[non_positive_mask][:, None] * X[non_positive_mask], axis=0)

    return gradient / len(residuals)

```

Code snippet

After calculating the cMSE loss, we compute the gradient, optionally apply regularization (Ridge or Lasso), and then update the weights to minimize the loss.



```python
# Compute gradient
grad_err = gradient_cMSE_error(y, y_hat, c, X)
grad = -grad_err

# Apply regularization
if regularization == 'ridge':
    grad += 2 * alpha * weights  # Ridge (L2)
elif regularization == 'lasso':
    grad += alpha * np.sign(weights)  # Lasso (L1)

# Update weights
weights -= learning_rate * grad
```

Evaluation

Model	Max Error	Min Error	Mean Error	Std Dev of Error	cMSE
Baseline	4.961	0.171	1.544	2.107	4.445
Gradient Descent	4.420	0.031	1.590	1.919	3.438
Gradient Descent (Ridge)	4.677	0.100	1.565	1.903	3.402
Gradient Descent (Lasso)	4.624	0.122	1.576	1.905	3.416

Using the following parameters for Gradient Descent: iterations=10000000, learning_rate=0.0000001, alpha=0.1 and the same threshold value, we can observe that the results align with expectations.

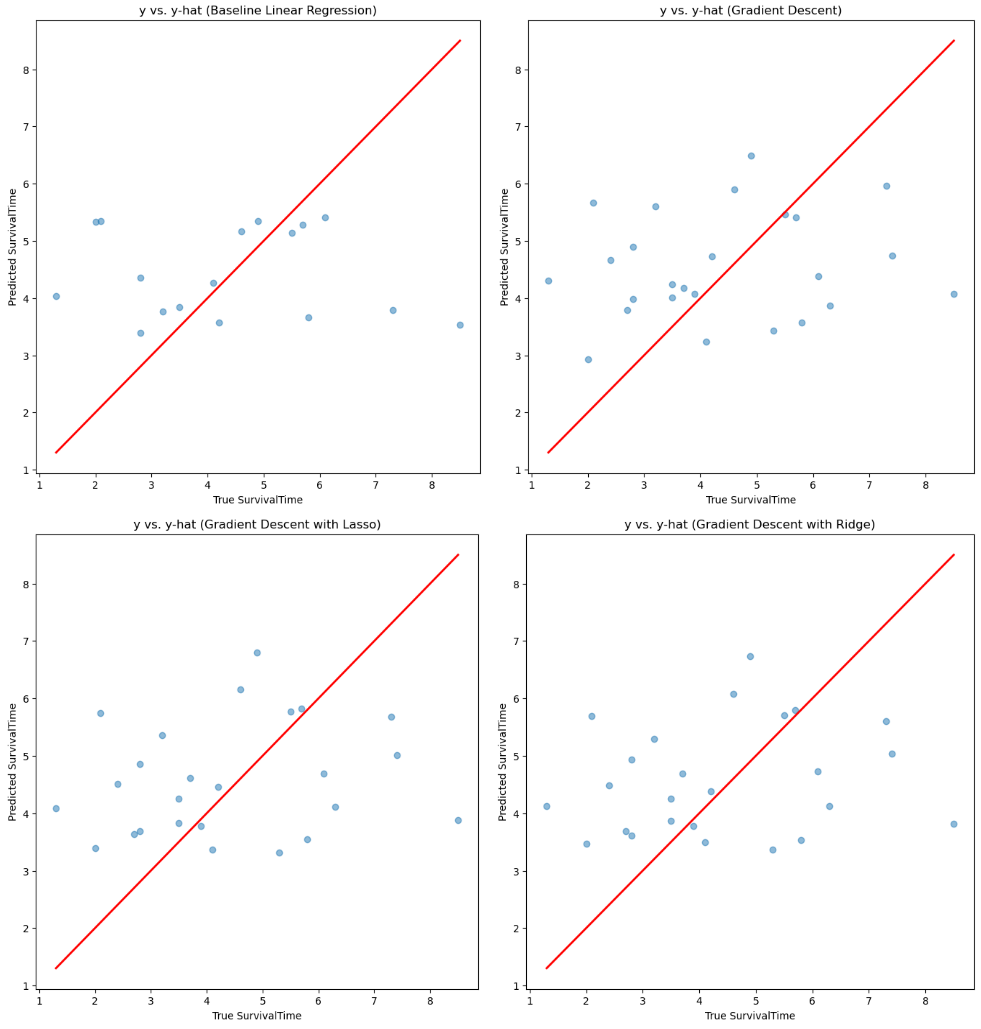
Ridge regularization outperforms Lasso, gradient descent without regularization and the baseline, while Lasso still shows improved performance over the non-regularized version.

All these models outperform the baseline model, as they not only make usage of more data points for training but also use a specialized loss function, and it's derivative, for better model creation and evaluation.

Evaluation (Cont.)

By comparing the y vs. y_hat plots, we can see that when using gradient descent to predict, not only are there more points being predicted, which is a consequence of adding the rows with censored entries, but also there's a better prediction made by these models.

It's important to note that, although we can see that the three Gradient Descent plots are very similar between them, Ridge's model performs slightly better than the rest.



Task 2 – Nonlinear Models

Index:

Task 2.1 Development

Pre-development analysis

- We began by evaluating the hyperparameters that needed tuning and identifying their type as well.
- For Polynomial Regression, there's only one hyperparameter to tune: the degree of the polynomial.
- For k-Nearest Neighbors (kNN) Regression, there's more hyperparameters that require tuning. We tested the following:
    - The number of neighbors (K);
    - The distance metrics for measuring neighbor proximity (Euclidean or Manhattan);
    - The weighting scheme for neighbors (uniform or distance-based).
- We started with the simpler hyperparameter tuning, that being the Polynomial Regression, using the same data split established in Task 1.1.

Polynomial hyperparameter tuning

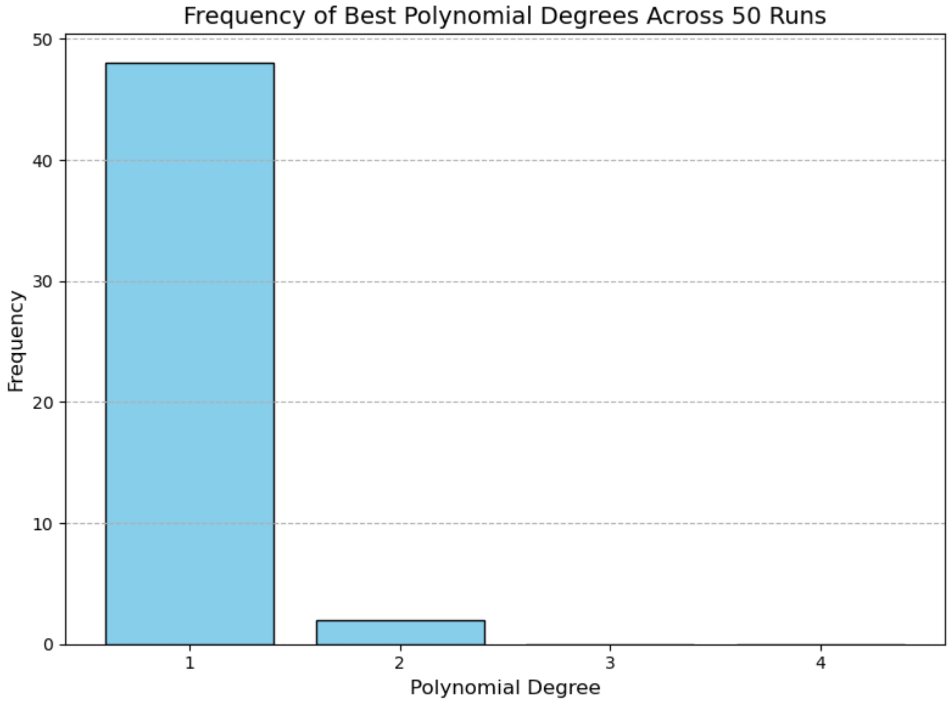
- We used the training set of the split to tune the hyperparameter for Polynomial Regression, reserving the test set for final model evaluation of both models we want to create.
- To achieve this, we developed a function that uses the KFold technique (explained on slide 24).
- For cross-validation, we set the number of folds to 10, allowing the training set to be further split into training (80%) and validation (10%) sets.
- We developed a function that not only returned the best degree but also the feature complexity of each degree, the loss values for both training and validation sets.

Hyperparameter tunning

We ran our function 5, 20, and 50 times, using a shuffled KFold split with no fixed random state, to observe whether the best polynomial degree for the Polynomial Regression varied across different splits.

We came to the conclusion that, even with varying splits, the degree was consistently 1, with only rare exceptions where it reached 2.

This result makes sense when considering the size of the dataset relative to the model's complexity, with 216 training data points and only 4 features used to build said model.

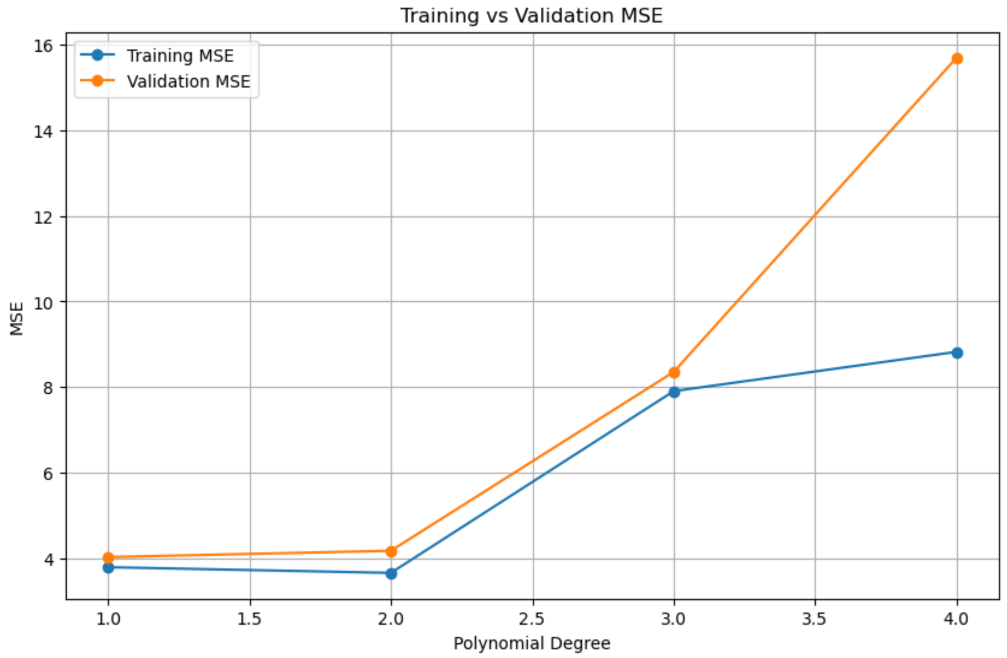


Overfitting prevention

We calculated the MSE for each fold and averaged the results for each polynomial degree.

From the plot, we can observe that when the polynomial degree increases by 1, the training error decreases. However, the validation error increases, suggesting that the model starts to overfit the training data, performing worse on unseen data.

This confirms what we said in the previous slide, that the best polynomial degree is 1.

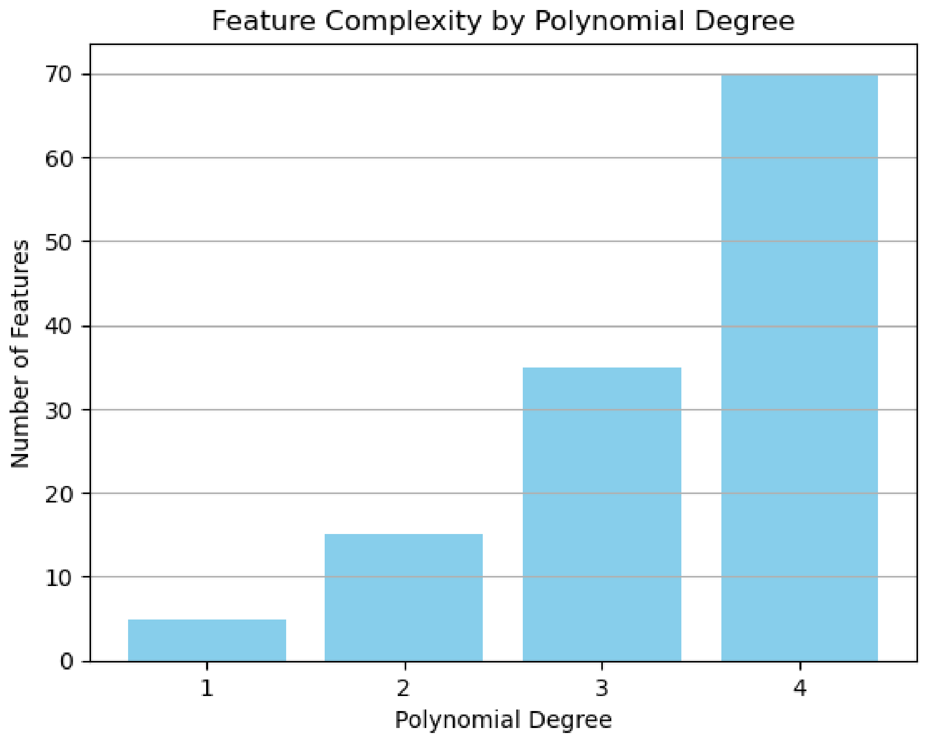


Feature Complexity

This figure perfectly demonstrates the exponential growth in feature complexity as the polynomial degree increases in our dataset, from degree 1 to 4.

While higher degrees allow models to capture hidden patterns in the data, they also risk overfitting, especially with small datasets.

This hyperparameter tunning highlights the trade-off between model performance and feature complexity in designing more robust models.



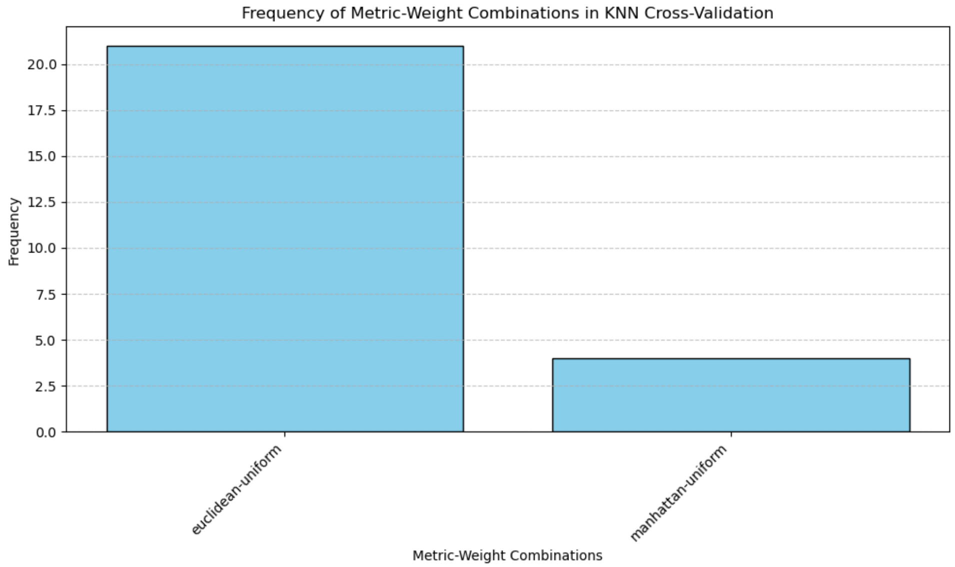
kNN hyperparameter tunning

- Like the approach used for tuning the polynomial model, we utilized the training set to tune the hyperparameters for the kNN model.
- We set the number of folds to 10, allowing the training set to be further split into training (80%) and validation (10%) sets.
- Our experimentation not only focused on the k-neighbors but also on two types of Weights and two Metrics:
  - Weights: Uniform vs. Weighted;
    - Uniform Weights: Treats all neighbors equally;
    - Distance-Based Weights: Assigns higher importance to closer neighbors;
  - Metrics: Euclidean vs. Manhattan;
    - Euclidean Distance: Assumes straight-line paths;
    - Manhattan Distance: Assumes step-by-step movement;
- For each combination of weights and metrics, we calculated the Mean Squared Error (MSE) for all folds and averaged the results across different values of k. This process was repeated for all four combinations of weights and metrics, basically running four separate MSE calculations for each k-value.
- We developed a function that not only returned the best Metric and Weight type but also the k value for that combination and the respective loss values for both training and validation sets.

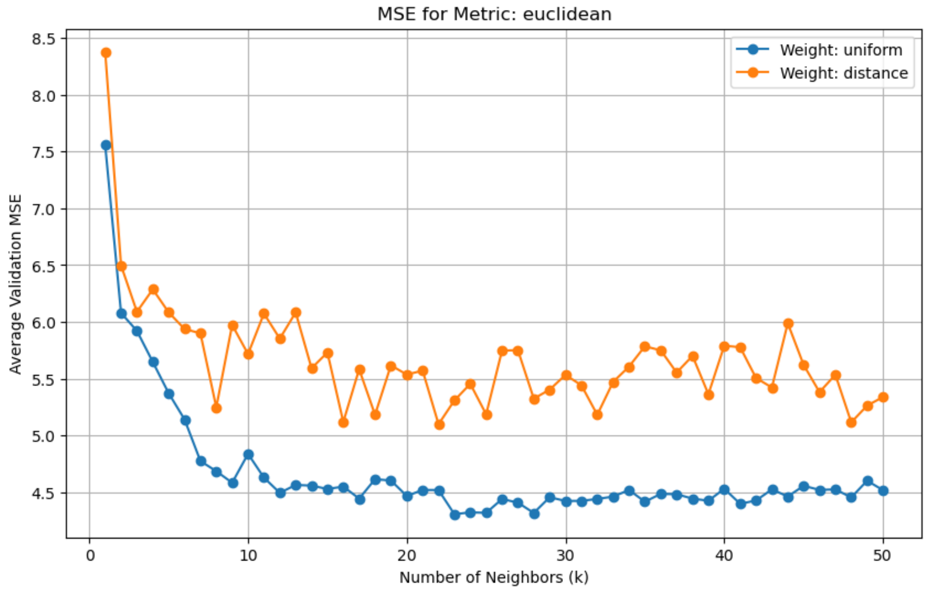
Hyperparameter combinations

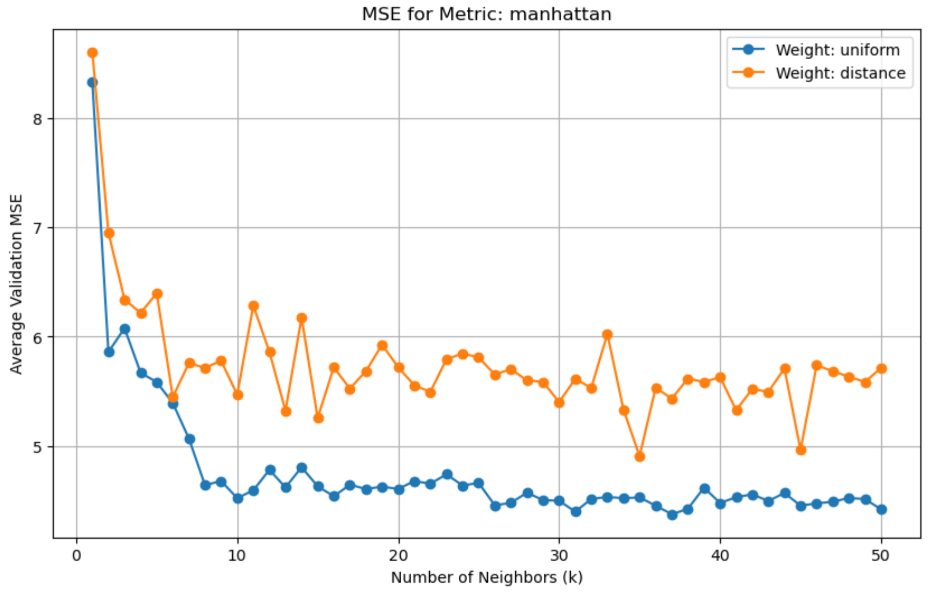
After running the hyperparameter tuning function 25 times, with k ranging from 1 to 50 and evaluating the Weights and Metrics previously mentioned, we concluded that the Uniform weight consistently outperformed Distance based weights.

Regarding the distance metrics, while there is no "best" metric, the Euclidean metric demonstrated better performance in most of the cases.



Comparing Hyperparameters





These plots reinforce what we analyzed in the previous slide.
It is impossible to determine a "final" best k-value across multiple splits. Since our method returns the best model for each specific combination of hyperparameters, this ensures that, after tuning the hyperparameters, we are always using the most optimal kNN model to make predictions on the test set.

Task 2.2 Evaluation

Evaluation

Model	Max Error	Min Error	Mean Error	Std Dev of Error	cMSE
Polynomal Regression	4.536	0.003	1.387	1.812	3.020
KNN	4.490	0.063	1.550	1.886	3.388
Baseline	4.961	0.171	1.544	2.107	4.445
Gradient Descent (Ridge)	4.677	0.100	1.565	1.903	3.402

By analyzing the table, we observe that polynomial regression outperforms the other trained models.

Even with a polynomial regression of degree 1, when trained using censored data and the specialized cMSE loss function, it surpasses the baseline model that although equivalent to polynomial regression of degree 1, was trained on different data, which highlights the importance of the more appropriate loss function and the usage of more data points.

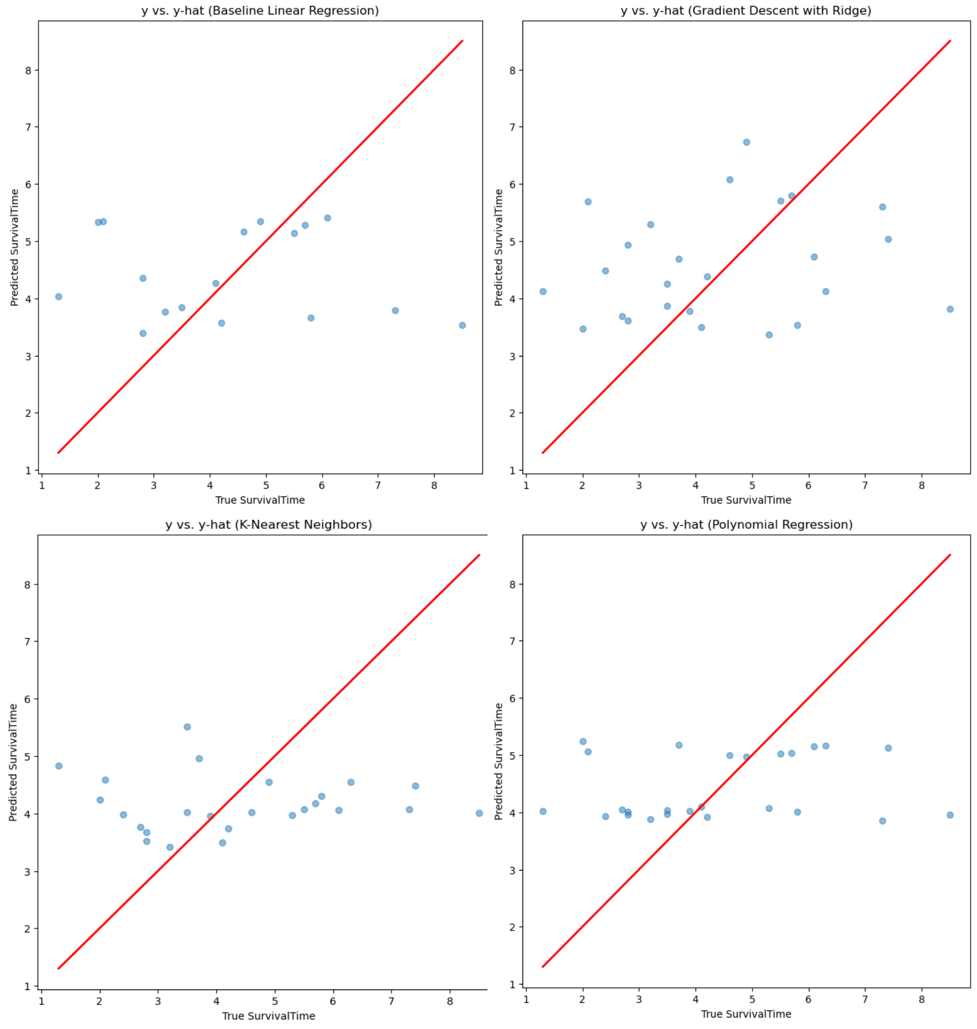
We can say that KNN doesn't perform well on this dataset because of the limited number of data points. As a result, there is no ideal value for k—a small k risks overfitting to the noise in the dataset, while a larger k averages over too few meaningful neighbors, leading to underfitting.

Evaluation (Cont.)

By comparing the y vs. y_hat plots, we can see that, although it's a linear prediction, the polynomial regression has fewer points spread out and more points closer to the "perfect" red line, reinforcing what we analyzed in the previous slide.

Comparing the KNN to the models made in task 1, we can say that it performed better, given the predicted points are closer to the red line.

It's worth mentioning that, although the difference between the Local cMSE score and the Kaggle cMSE score, also known as the Relative Error, was higher than expected it reflected the performance disparity between the better-performing polynomial regression model and the other models we created, such as kNN.



Task 3 – Handling Missing Data

Index:

Task 3.1 – Missing Data Imputation

Missing Data Imputation

- In the previous tasks, we've been working without the features where the datapoints have missing data.
- The goal of this task is to explore different techniques of imputing the missing data entries, this will allow us to fill the NaN entries with values so that we can take advantage of a bigger dataset.
- This is still a supervised learning task, so we won't be imputing the unlabeled data.

Simple Imputer

When using the Simple Imputer, there are several methods to impute missing values. We explored methods that replace missing values across each column using:

- Mean
- Median
- Frequent

Model	MSE
Mean model	3.038923
Median Model	3.134671
Most Frequent Model	3.263605

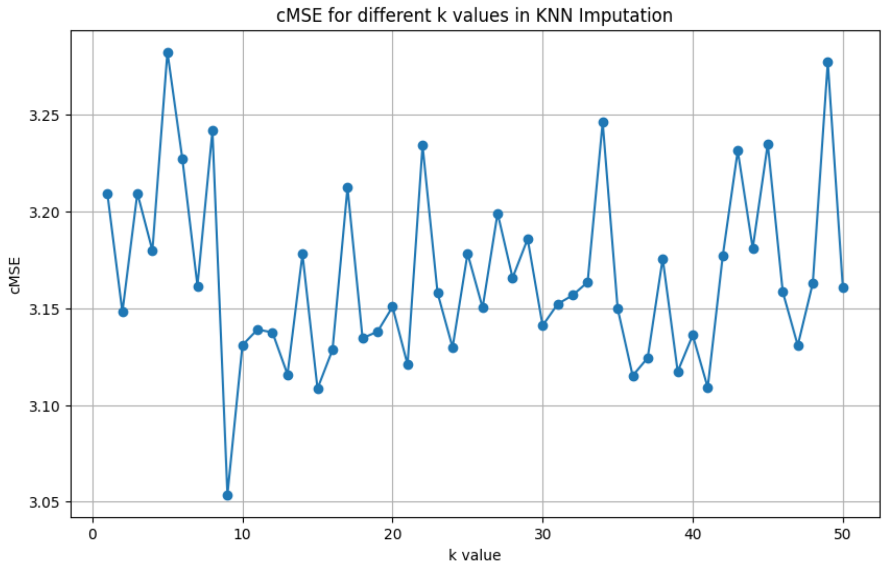
These different imputing techniques yielded very similar results, however, the model imputed using the mean had a slightly better result.

kNN Imputer

The kNN imputer provides imputation by filling the missing values using the k-Nearest Neighbors approach.

In the graph we can see the mean error evolution over the k neighbors chosen to impute the data.

The overall best result is using k = 8



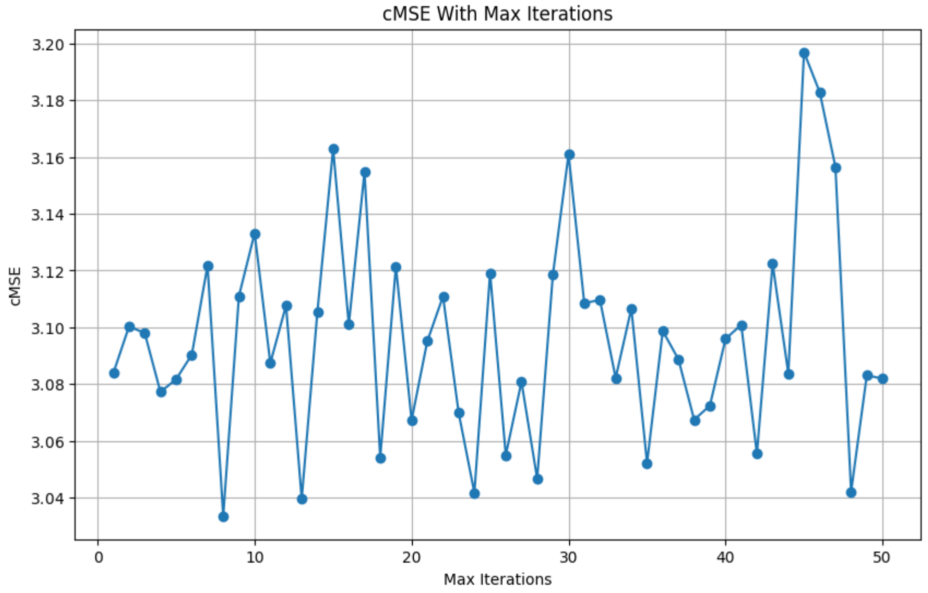
Model	Max Error	Min Error	Mean Error	Std Dev of Error	cMSE
kNN Imputer (k = 8)	3.730551	0.026738	1.315788	1.609138	2.324343

Interactive Imputer

The Interactive Imputer is a multivariate imputer that estimates each feature from all the others.

A strategy for imputing missing values by modeling each feature with missing values as a function of other features in a round-robin fashion.

It also introduces the notion of iteration, where a value is predicted based on a sequence of steps.



Model	Max Error	Min Error	Mean Error	Std Dev of Error	cMse
Iterative (max_iter=8)	3.716857	0.021804	1.351380	1.676996	2.524902

Overview

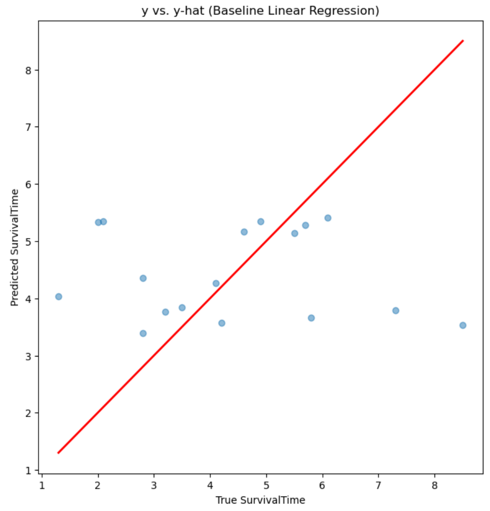
Model	Max Error	Min Error	Mean Error	Std Dev of Error	cMSE
Baseline	5.195	0.084	1.415	1.831	3.390
Mean model	3.722244	0.019853	1.322866	1.610295	2.325515
Median Model	3.780130	0.000037	1.222028	1.539981	2.354777
Most Frequent Model	3.780130	0.000037	1.222028	1.539981	2.404777
kNN Imputer (k = 8)	3.730551	0.026738	1.315788	1.609138	2.324343
Iterative (max_iter=8)	3.716857	0.021804	1.351380	1.676996	2.524902

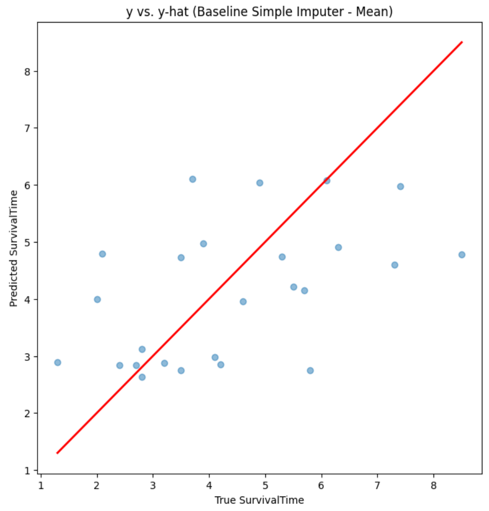
When comparing these results, we can see that the imputation models have a significant lower error than the baseline. When comparing the different imputations, we can also observe that the KNN Imputation has the lowest score, however, is not very significant.

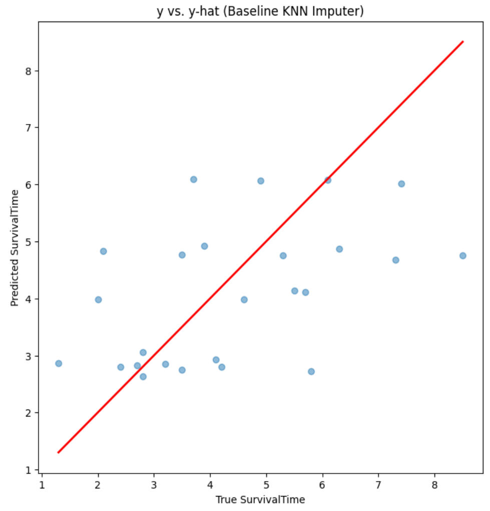
Overview (Cont.)

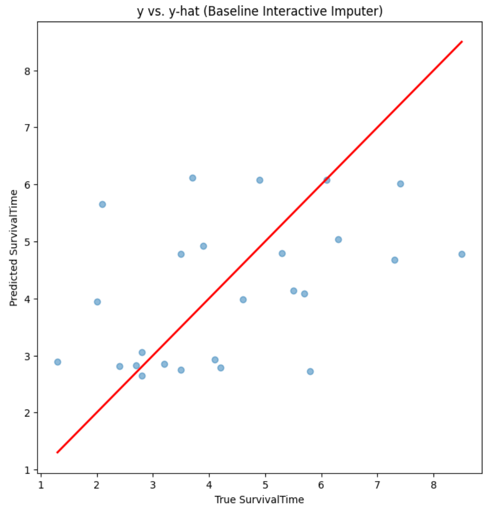
By comparing the y vs. y_hat plots between the baseline and the baseline with imputed values, we can see that imputed models have more points being predicted, which is a consequence of now having NaN entries in the dataset.

Additionally, we can see that imputation plots are very similar between them.









Task 3.2 - Models that don't require Imputation

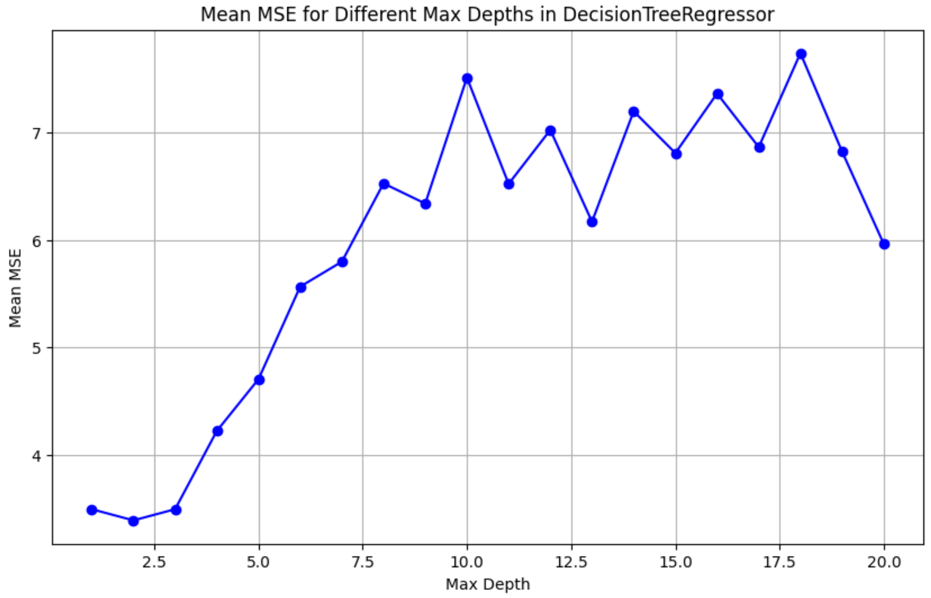
Models that don't require imputation

- In contrast with the last section, this one explores the path of using models that don't require imputation, meaning that these models will be developed without changing/removing the missing values.
- In the following slides we'll be exploring Decision Trees, RandomForests, HistGradientBoostingRegressor and Catboost.

Decision Tree

- By using a Decision Tree where we can predict the labels by learning decision rules from the data. It recursively splits the dataset into smaller subsets based on feature values, optimizing to minimize error in each subset.

- In the plot we can observe the evolution of the Mean Squared Error (MSE) for different values of the maximum depth of the decision tree. Initially, as the depth increases, the MSE decreases because the model becomes more flexible and better fits the data. However, beyond a certain depth, the MSE starts fluctuating and increasing due to overfitting.



Model	Max Error	Min Error	Mean Error	Std Dev of Error	cMse
Decision Tree	3.948148	0.051852	1.311834	1.689445	2.572212

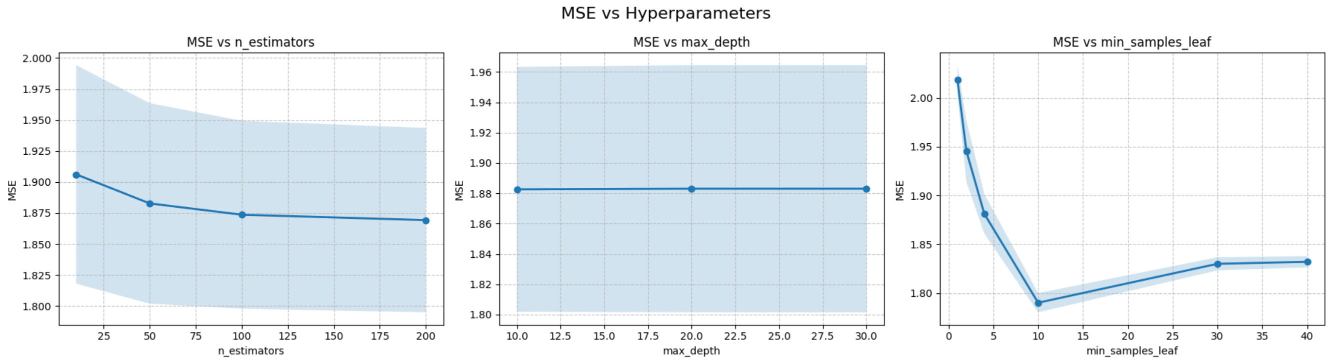
Random Forests Regressor

- A random forest is a meta estimator that fits a number of decision tree regressors sequentially on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. Trees in the forest use the best split strategy.

- There are multiple parameters to configure, but we found the following ones to the most relevant.
    - Number of estimators: The number of trees in the forest.
    - Max Depths: The maximum depth of the tree.
    - Min Samples Split: The minimum number of samples required to split an internal node.
    - Min Sample Leafs: The minimum number of samples required to be at a leaf node.
- Our evaluations contain a Min and a Max error, to illustrate that, we've included it with a light blue shadow to represent the interval.

Random Forests Regressor

- **Number of Estimators**: Increasing the number of trees reduces MSE initially as the ensemble becomes more robust, but improvements taper off as the ensemble saturates.
- **Maximum Depth**: MSE stabilizes with depth, indicating that deeper trees do not necessarily improve performance in this case.
- **Minimum Samples per Leaf:** Reducing the minimum samples per leaf improves MSE up to a point, as smaller leaves allow for finer splits, but further reductions may overfit the data.



Model	Max Error	Min Error	Mean Error	Std Dev of Error	cMse
Random Forest	4.351976	0.008076	1.371991	1.672649	2.629983

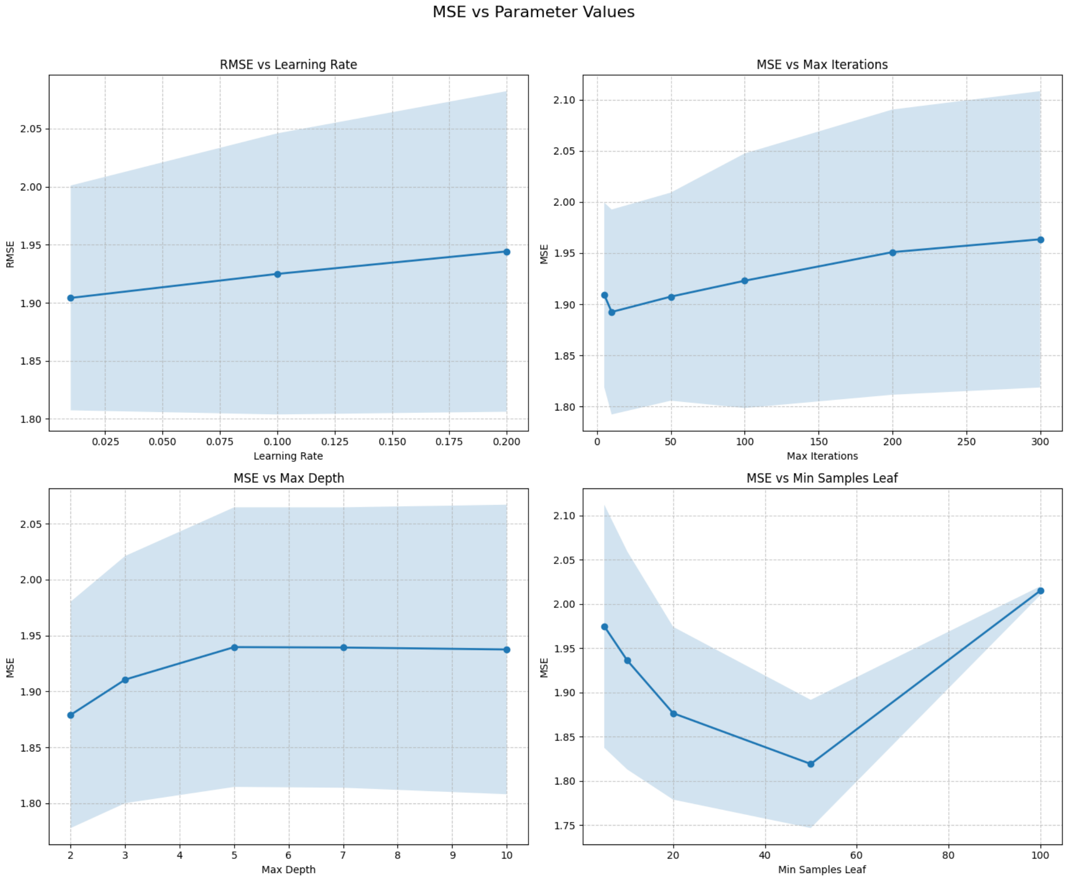
HistGradientBoostingRegressor

- In contrast with Random Forests Regressor, Histogram-Based Gradient Boosting builds decision trees sequentially, with each tree trying to correct the errors (residuals) made by the previous ones. The trees are added iteratively to minimize a loss function, typically by fitting to the pseudo-residuals of the previous model.

- There are multiple parameters to configure, but we found the following ones to the most relevant.
    - Number of estimators: The number of trees in the forest.
    - Max Depths: The maximum depth of the tree.
    - Min Samples Split: The minimum number of samples required to split an internal node.
    - Min Sample Leafs: The minimum number of samples required to be at a leaf node.
- Our evaluations contain a Min and a Max error, to illustrate that, we've included it with a light blue shadow to represent the interval.

HistGradientBoostingRegressor (Cont.)

- From these plots we can observe the evolution of the MSE vs the evolution of the different mentioned in the previous slide:
    - **Learning rate:** Lower values (~0.025) yield better results
    - **Max iterations:** Optimal performance at 25-50 iterations, diminishing returns after
    - **Tree depth:** Shallow trees (2-3 levels) perform best
    - **Min samples per leaf:** Sweet spot around 50-60 samples



Model	Max Error	Min Error	Mean Error	Std Dev of Error	cMse
HistGradientBoosting	4.036754	0.010545	1.355310	1.638894	2.427744

CatBoostRegressor

- CatBoostRegressor is similar to HistGradientBoostingRegressor because it is still a Boosting Regressor, however, they handle categorical features differently. It also has support for censored data, for training this model, we used this tutorial and implemented the **Accelerated Failure Time (AFT)** model.
- Contrary to the previous model, there is some preparation needed in order to fit the model.

Preparation:

- When using censored data in Catboost we should express target variable in interval form, since our data is right-censored, we will use the interval of the form [A, +∞].
- The lower bound will be all the SurvivalTime entries, the upper bound, will be all the censored SurvivalTime entries, otherwise it will be infinity.
- Identify the categorical features in this data set (Gender, Stage, TreatmentType, TreatmentResponse), and convert them to strings.
- Create a Pool to wrap the new train and test dataset created, and specify the categorical features.
- Fit the model

CatBoostRegressor – Fit, Evalutation Metric and Results

- To fit the model, we had the choice of using three different distributions:
  - Normal
  - Logistic
  - Extreme
- The evaluation of these models, is done using Mean Absolute Error (MAE).

Model	Train MAE	Test MAE
Normal	0.80	0.92
Logistic	0.83	0.97
Extreme	0.85	0.92

- The predictions of this model were higher than the others, however, they were very similar to our Kaggle Submission score.
  - **Kaggle**: 3.28320
  - **Local**: 3.41464
- For that reason, we choose this one over others that had a higher discrepancies of local errors vs Kaggle submission error

Model	Max Error	Min Error	Mean Error	Std Dev of Error	cMse
Catboost	4.164301	0.010179	1.866218	1.913986	3.414604

Task 3.3 - Evaluation

Comparing the Results of Task 3.1 and 3.2

Model	Max Error	Min Error	Mean Error	Std Dev of Error	cMSE
Baseline	5.195	0.084	1.415	1.831	3.390
KNN Imput Model	3.730551	0.026738	1.315788	1.609138	2.324343
Mean Imputation	3.722244	0.019853	1.322866	1.610295	2.325515
HistGradientBoosting	4.036754	0.010545	1.355310	1.638894	2.427744
Iterative	3.716857	0.021804	1.351380	1.676996	2.524902
DecisionTree	3.948148	0.051852	1.311834	1.689445	2.572212
Random Forest	4.351976	0.008076	1.371991	1.672649	2.629983
Catboost	4.164301	0.010179	1.866218	1.913986	3.414604

- The best model in this analysis was the KNN Imputation model, followed closely by the Mean Imputation. This indicates that for our dataset, using imputation rather than Boosting, Ensembles or Decision Trees works better.

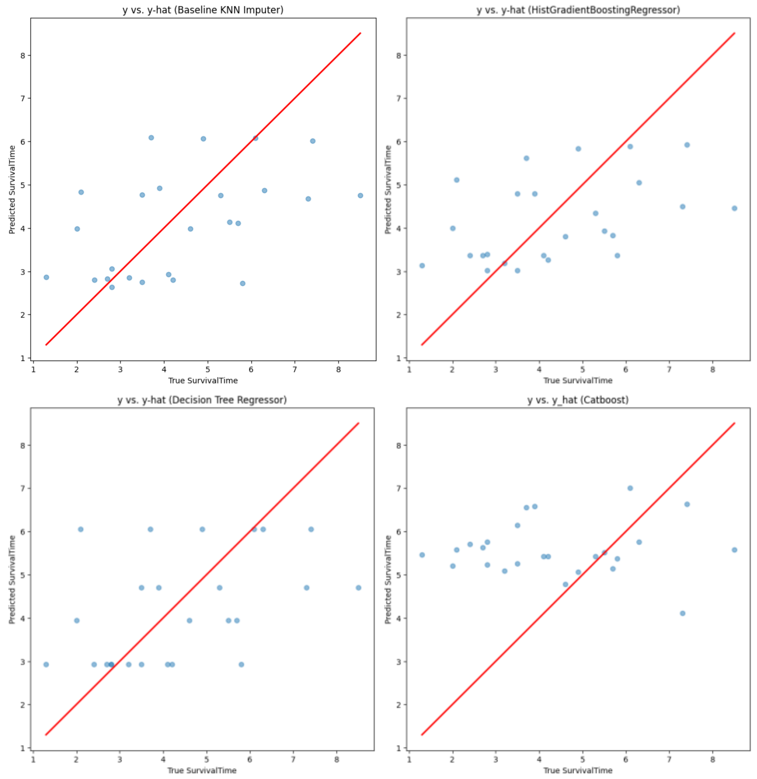
Comparing the Results of Task 3.1 and 3.2 (Cont.)

- In comparison with the evaluation made for the Imputation models, here we can see the y vs. y_hat plots, are very different from one another.

- HistGradientBoostingRegressor is the most similar to the kNN imputer.

- The Decision Tree Regressor, has a very distinct mapping of the predicted points, as if they were divided across quadrants, they don't seem to be placed randomly as the other plots. This is most likely due to the Decision Tree algorithm, that partitions the feature space into decision regions.

- The CatBoostRegressor, is the one with higher error, we can observe this from the plot, since most of the points are very distant from the true value.

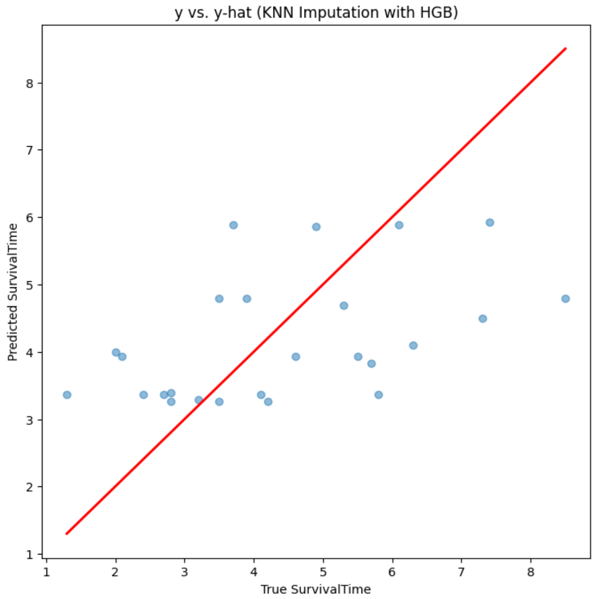


Trying the best imputation strategies of Task 3.1, with the best model of task 3.2

Model	Max Error	Min Error	Mean Error	Std Dev of Error	cMSE
Baseline	5.195	0.084	1.415	1.831	3.390
KNN Imputation with HGB	3.706374	0.090734	1.337794	1.592056	2.235545
KNN Imput Model	3.730551	0.026738	1.315788	1.609138	2.324343
HistGradientBoosting	4.036754	0.010545	1.355310	1.638894	2.427744

- By combining these two techniques we can see an improvement of the cMSE, additionally, when comparing the y vs. y-hat plot with the previous, we can see that overall, the points are closer to each other.

- This is the case because imputation creates a solid foundation (complete data) for Gradient Boosting, where for each tree that it builds it has more points to correct the errors, allowing it to perform more better predictions.



Task 4 – Semi-Supervised Learning

Index:

Task 4.1
Imputation with Labeled and Unlabeled Data

Imputation on both label and unlabeled data

In this task, we began by using one of the best imputers identified in Task 3.1, the KNN imputer, to handle the missing data provided at the start of the project.

Since it's important to preserve the original (and only) data split established in task 1.1, we merged both X_train and X_test, creating a combined dataset with all 400 initial points, including censored and missing data in both features and labels.

We then fit and transformed the combined feature dataset using the KNN imputer and split it back into X_train and X_test, resulting in 400 datapoints with no missing values in the feature set.

Since this remains a supervised learning task, albeit semi-supervised in this task, we did not impute missing labels. As such we removed datapoints with unlabeled target values, in both sets, before proceeding.

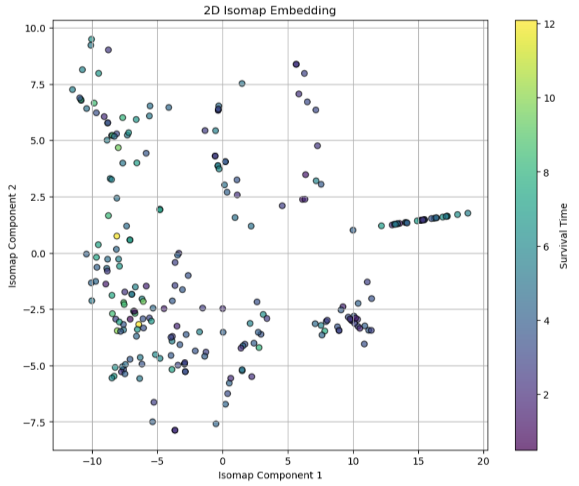
It is important to highlight that, although a similar imputation process was performed in Task 3.1, the results differ due to a key methodological change. In Task 3.1, we removed the unlabeled datapoints before imputing the features. In contrast, for this task, we retained the unlabeled data during imputation, providing the imputer with a larger dataset and potentially improving the quality of the imputation..

Model	Max Error	Min Error	Mean Error	Std Dev of Error	cMSE
KNN Imputer w/o Unlabeled	3.730	0.026	1.315	1.609	2.324
KNN Imputer with Unlabeled	3.700	0.032	1.344	1.623	2.331

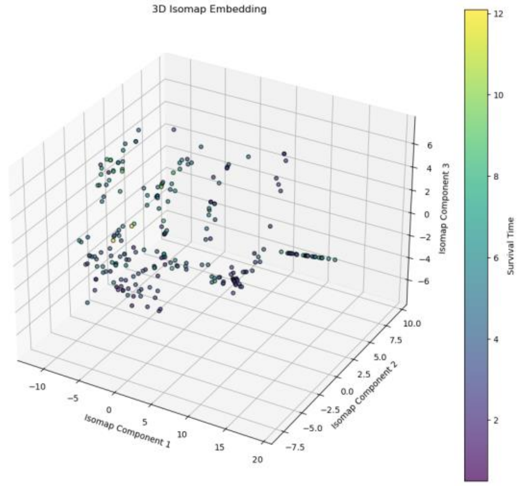
Isomap on imputed data

After imputing the dataset, we proceeded with the creation of an Isomap model to reduce the dataset's dimensionality. This allowed us to not only represent the feature set visually in 2D and 3D plots, providing a clearer way to see its structure (and potential clusters) but also train a linear regression model using said reduced dataset, enabling us to evaluate whether dimensionality reduction could improve (or not) the model's performance.

By transforming the original feature set into a lower-dimensional representation, we could create plots like these:



Isomap reduction to 2 features



Isomap reduction to 3 features

This approach also helped us determine whether the reduced feature space maintained meaningful relationships between data points, ultimately supporting better model evaluation and visualization.

Task 4.2 Evaluation

Evaluation

Model	Max Error	Min Error	Mean Error	Std Dev of Error	cMSE
Baseline	4.961	0.171	1.544	2.107	4.445
KNN Imputer with Unlabeled	3.700	0.032	1.344	1.623	2.331
Isomap Reduction	3.968	0.036	1.426	1.771	2.841

We performed 50 iterations of dimensionality reduction using Isomap and cross-validating with KFold on the train set, testing feature reductions from 2 to 6 dimensions.

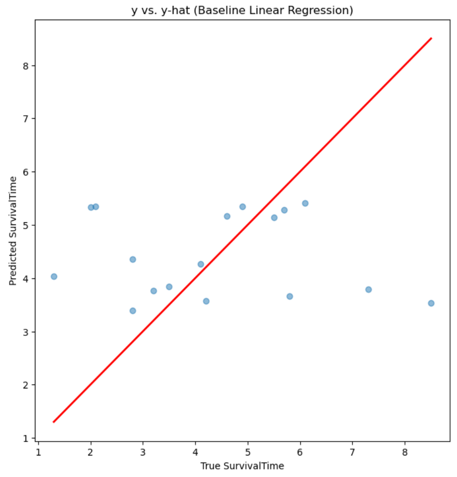
Our tests showed that the best performance of the linear regression model occurred consistently when the number of features was reduced to 5 or 6, or even maintain all 7, suggesting that reducing the features beyond 2 was rarely optimal.

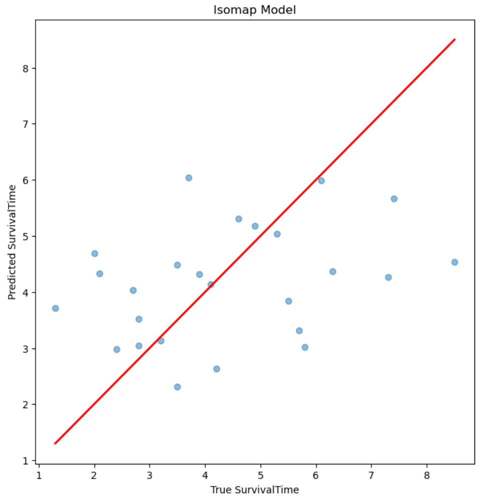
This behavior can be probably be explained by the limited number of data points in our dataset and the already small number of original features. Further reducing dimensionality likely caused the model to lose essential information, impacting its performance. Additionally, splitting the training set into two further subsets during K-Fold cross-validation may have introduced additional variability, leading to less stable model evaluations.

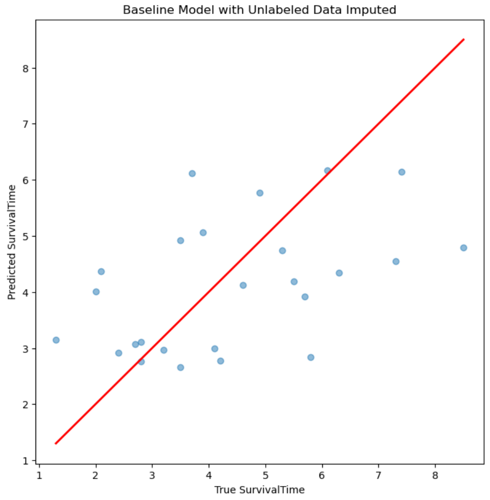
Evaluation (Cont.)

Comparing the y vs. y_hat plots, we can observe better performance in both the Isomap and Unlabeled Data Imputed models. Among these, the imputed model stands out due to its lower cMSE value and fewer data points deviating significantly from the "perfect" red line.

It's important to note that, since the imputed methods in this task and task 3.1 were the same, the result of both can (and in this case are) be similar and as such, display similar y_yhat plots.





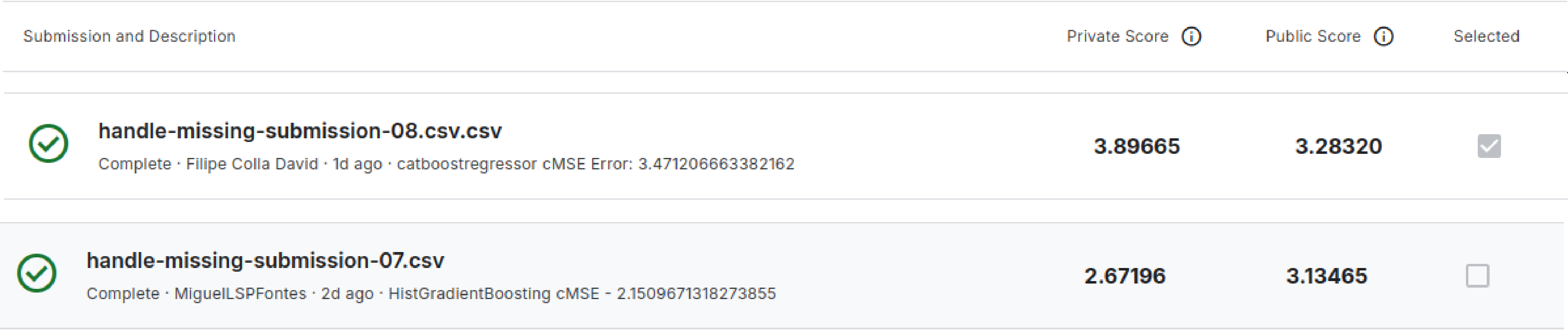


Post Kaggle closing thoughts

Analyzing the submission choice for Kaggle

- Often when submitting to Kaggle, we've noticed some discrepancies from our local and the Kaggle evaluation. We though that our validation was not correct, and we choose the values that were much closer in terms of error. In hindsight, after the private scores were released, we got to see that our validation was in fact correct.

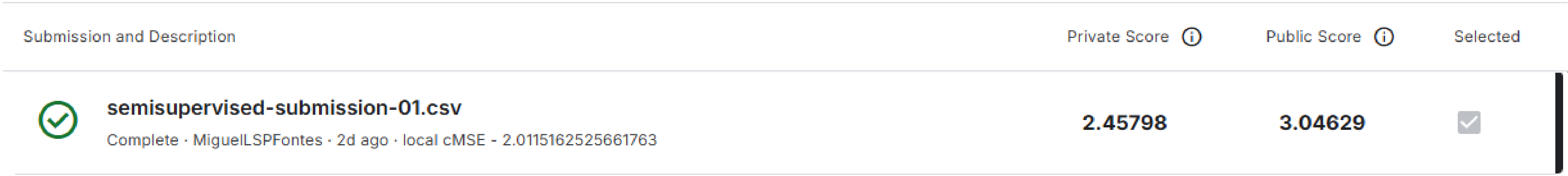
- From the following screenshots, we can see that there wasn't a big discrepancy from the local score vs the Kaggle score of the "handle-missing-submission-08.csv.csv" (3.47 vs 3,28), however, the "handle-missing-submission-07.csv" was not very close (2.15 vs 3.13).



Submission and Description	Private Score	Public Score	Selected
handle-missing-submission-08.csv.csv Complete · Filipe Colla David · 1d ago · catboostregressor cMSE Error: 3.471206663382162	3.89665	3.28320	✓
handle-missing-submission-07.csv Complete · MiguelLSPFontes · 2d ago · HistGradientBoosting cMSE - 2.1509671318273855	2.67196	3.13465	☐

Analyzing the submission choice for Kaggle

- Going back on the topic of discrepancies between our local cMSE and Kaggle cMSE discussed last slide, while we had our doubts that our models could yield such good results, shown by the difference between local cMSE and public Kaggle cMSE (2.0115 vs 3.0462), we still wanted to try and have faith that our validation process was a success and chose the best model, regardless of whether the relative error was bigger than others.
- In the end, this decision paid off. Our chosen model achieved the best private Kaggle score out of all the submissions we did. While it wasn't our final submission for the semi-supervised task, it still stands out as one of the top-performing models we developed.



Submission and Description	Private Score	Public Score	Selected
semisupervised-submission-01.csv Complete · MiguelLSPFontes · 2d ago · local cMSE - 2.0115162525661763	2.45798	3.04629	✓

Overall assessment

What went wrong

As mentioned in the previous slides, having a stratified split is crucial in a dataset of this size, otherwise, the results end up being inconsistent over different splits. This is what happened to us, and we invested sometime until we could figure out what was wrong.

By our understanding, some descriptions of the assignment could be more complete and somewhat less vague, making it hard to continue working in the project without clarifying our doubts with the teacher first. Additionally, having more resources such as the AFT from Task 3.2 one, would be a great way to point us towards valid approaches for this project.

Our miss understanding of the differences from the local error and the Kaggle submission, led us to make wrong decisions when choosing one of the two best submissions.

What went great

While having a small dataset has its own difficulties, it also has its benefits, we can take advantage of more complex models, since they don't take as long as they do with bigger datasets.

Once we understood that we needed a stratified split, with the help of the teacher, training the models and verifying their results was much more stable, even if we sometimes encountered discrepancies when comparing to the Kaggle Submissions.

We caught on that KNN was going to be a model that would underperform, regardless of the hyperparameters given.