Escuela Técnica Superior de
Ingeniería Informática

MASTER'S THESIS

Reconocimiento de relaciones del genoma con la respuesta ante el cáncer

Written by
Sergio Santiago Sánchez

To obtain the title of
Master's Degree in Logic, Computation and Artificial Intelligence

Advisors
Miguel Ángel Gutiérrez Naranjo
Miguel Cárdenas Montes

July Examination Session, Curse 2024/25

Aquí la dedicatoria del trabajo

Acknowledgments

Quiero agradecer a X por...

También quiero agradecer a Y por...

Resumen

Incluya aquí un resumen de los aspectos generales de su trabajo, en español.

Palabras clave: Palabra clave 1, palabra clave 2, ..., palabra clave N

Abstract

This section should contain an English version of the Spanish abstract.

Keywords: Keyword 1, keyword 2, ..., keyword N

Contents

List of Figures

List of Tables

Code excerpt index

1. Introduction and motivations

There is currently an extensive body of publicly available research foccused on finding a cure for cancer. There is a huge amount of open access data [1, 2] that allows researchers worldwide to exchange knowledge with the community. This is how the scientific community tries to continue the chain until it achieves findings that can save lifes.

One line of research is the effect of a drug based on the patient's genome. There are records of patients and studies with the same medication and type of cancer but with different responses to treatment [3, 4]. Especially regarding the personalization of therapy since the variability raises critical questions. Decisions about which medication to administer and in what dosage directly affect people's quality of life. If a patient receives an excessive amount of medication, the consequences can be even worse than those of the cancer itself. Similarly, if the amount administered is less than required, the effects will not be as desired.

One of the main challenges of applying machine learning techniques, to this type of research, lies in the very nature of the data, which poses significant limitations. Generally, it is highly variable, with cancer records showing a large number of treatments for the same type and often containing a considerable amount of noise or missing values. At the same time, genetic information is very extensive, further complicating the search for a model that meets all requirements.

To address these challenges, the proposed models must be highly robust in the face of different datasets. They must be able to function correctly despite noise and missing values. In addition, these models must be updated regularly so that new knowledge about the disease leads to better and higher quality predictions. Without these periodic predictions, the risk of obsolescence increases dangerously.

This is undoubtedly one of the most demanding and sensitive areas of research, not only due to its technical complexity, but also because of the profound human responsibility it entails. Working with data derived from serious illnesses, such as cancer, requires a constant awareness that behind each data point is a person, with their own story, struggle and hope. The outcomes of such research have the potential to significantly improve patient care and quality of life. However, this potential is accompanied by a high degree of responsibility: a robust and well-validated finding can guide more effective treatments, while an error or oversight could lead to inappropriate clinical decisions, potentially harming lives. In this respect, it really is a matter of life or death. It is a field that demands not only technical excellence but also ethical rigor, caution and empathy. The risks are substantial but so are the opportunities to make meaningful, life-changing contributions to human health.

2. Core Knowledge

2.1. Introduction

There are currently numerous techniques used in artificial intelligence. These not only come from physics or mathematics, but are also closely related to economics and computer science. Such is the catalog of possibilities available to us that it is often difficult to keep up to date, as new ideas emerge every day. Therefore, in order to obtain a better general understanding of this work, we will explain the principles of some of the methods or techniques used in the project, since understanding them will make it easier to follow the workflow. s

2.2. Neural Networks

If we look at the history of science and technology, we can observe that many advances are inspired by real biological systems. For example, a camera shares many similarities with the human eye. Another example is the processor, which consists of a large number of transistors connected to perform calculations. Neural networks are also part of this trend, as their structure is approximately based on biological neural networks.

Neural networks are a type of machine learning model inspired by the human brain. They are designed to recognise patterns or relationships between data. This feature makes them the ideal type of model for images, language processing, or predictive modelling.

To fully comprehend this research, it is necessary to understand the following concepts.

2.2.1 Relevant layers for this research

There are numerous options available when selecting the appropriate layers for a neural network, each serving a distinct and well-defined purpose. Given the wide variability of existing architectures, this study will briefly describe the most influential layer types employed during the research, focusing on those that had the most significant impact on model performance and learning behavior.

- Dense: Also known as fully connected, it is one of the simplest and best-known layers. Each neuron receives one input from each neuron in the previous layer and produces one output per neuron.
- convolutional 1D layers: This is a layer used in neural networks to process sequential information, such as text, audio or genes. It allows one-dimensional convolution filters to be applied, thereby extracting relationships and patterns from the data. The output of a Conv1D layer is a

feature map that captures the patterns extracted from the input data. These maps highlight local relationships in the sequence, such as recurring motifs or trends. However, one potential drawback of stacking many convolutional layers is that the original low-level information can become increasingly abstract or diluted. While deeper layers tend to capture more complex features, they may also lose fine-grained details present in the raw data.

2.2.2 Regularization methods

Regularisation is applied to machine learning models to prevent overfitting, which occurs when they learn the training data too well, including noise and outliers. This results in poorer generalisation when faced with new data.

Some examples include:

- **L1 and L2 regularization**: Also known as Lasso (L1) and Ridge (L2) regularization, these techniques help prevent the model's weights from becoming too large. L1 Regularization encourages sparsity by driving less relevant weights to zero, effectively performing feature selection. L2 Regularization penalizes large weights more smoothly, distributing the error and promoting generalization.
- **Dropout**: This layer allows some neurons to be randomly reset to zero. This prevents the network from becoming overly dependent on a single neuron, thus avoiding overfitting.
- **Spatial dropout 1D**: This is an adaptation of dropout to networks in which convolutional layers are applied, so that the dimension of spatial dropout depends on the dimension of the convolutional layer, in this case 1.
- **Batch normalization**: Used to standardise the input to the layers, it reduces the impact of weight initialisation. It is very important to apply this layer before the activation function and not afterwards.
- **Max pooling**: It allows you to reduce the dimensionality of feature maps by selecting the maximum available using a sliding window.

2.3. Predictions trees

Decision trees are one of the most widely used types of machine learning algorithms, due not only to their great predictive power but also to their ease of interpretation, both in classification and regression tasks. The idea behind this algorithm is to divide the dataset into increasingly smaller subsets.

One of the main advantages is the interpretability they offer, without the need for any type of complex preprocessing or scaling. This makes them widely used in domains where understanding the reason behind each decision is important, such as biology or chemistry.

However, they do have disadvantages. For example, they are prone to overfitting, although there are techniques such as a posteriori pruning, maximum

depth or minimum number of examples per leaf, to avoid this. But undoubtedly, the main drawback is that they are unable to generalise correctly when the test data falls outside the training domain.

The main types of tree algorithms include:

- **CART**: It is a foundational decision tree algorithm, used for regression or classification tasks. It uses the Gini index or MSE to divide the data into different subsets, depending on whether it is a classification or regression problem.
- **RandomForest**: It is an ensemble learning method [1] used for both classification and regression tasks. It builds upon decision trees by combining multiple trees, which are independent, and aggregating their results to improve accuracy and reduce overfitting.
- **Gradient Boosted Trees**: This technique combines different decision trees, but unlike RandomForest, they are not independent. Each tree is trained with the aim of improving on those examples where the previous one failed. An example of this type of algorithm is XGBoost.

2.4. Shap values

Based on Shapley values from cooperative game theory, SHAP is capable of measuring the contribution of a feature to a result, i.e., it quantifies the contribution of variables to the model output, both locally, a single example, and globally, the entire dataset. This allows us to determine which variables are most relevant in a prediction.

The main advantage of SHAP over other explainability algorithms is that it provides a global view of the importance of the features in the entire dataset. In addition, thanks to its additive nature, it can be parallelised, as long as it is guaranteed that an example belongs to only one set, reducing computation times.

In addition, the implementation in Python [5] provides numerous graphs, thus facilitating the understanding of the data. For example:

- Summary plot: This graph combines feature importance with their directional effect on the model's output, providing an intuitive representation of which variables are most influential, whether they contribute to an increase or decrease in predictions, and the range of input values that drive such behavior. The representation encompasses the entire dataset.

[1] Algorithms that combine several models, usually of the same type, produce an output based on all of them, applying techniques such as average voting.

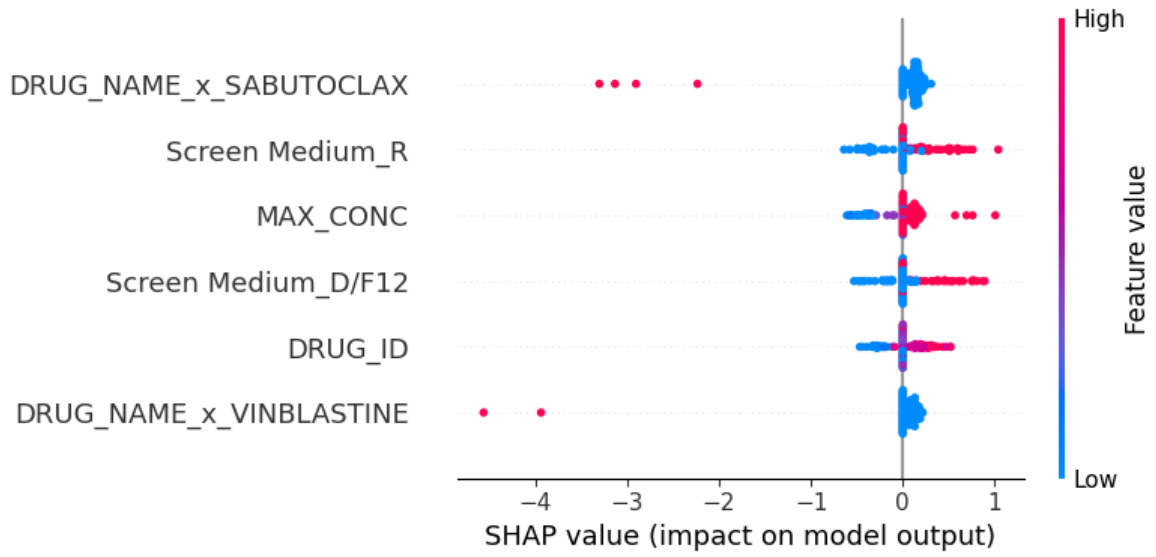

Figure 2.1: Example of Shap values representation: Summary plot.

  - Y-axis: The features sorted by relevance.
  - X-axis: SHAP value. Impact on model output, a positive value show a higher predicton and vice versa.
  - Dots: Each point represent a record of the dataset.
  - Color: Indicate the feature value for that example.

- Force plot: Used for local explainability. It provides the reasons why the prediction is higher or lower for an unique instance. In this type of graph, contrasting colors are used to show how each variable influences the prediction. Features that push the prediction higher are typically shown in red, while those that push it lower are shown in blue. This visualization helps to clearly distinguish between the variables that support and those that oppose the model's output for a specific instance.

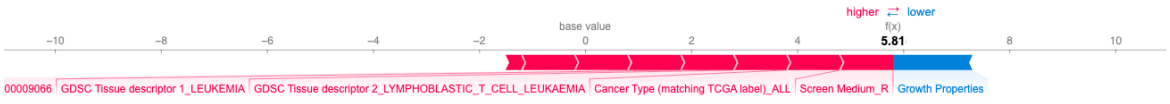

Figure 2.2: Example of Shap values representation: Force plot.

Figure 2.2 shows the variables that support increasing the value of the model output, while indicating that the Growth Properties variable is the only one that opposes this increase.

- Dependence plot: Used for global explainability. Show how a feature's value affect its SHAP value across the entire dataset helping to identify the relationship.

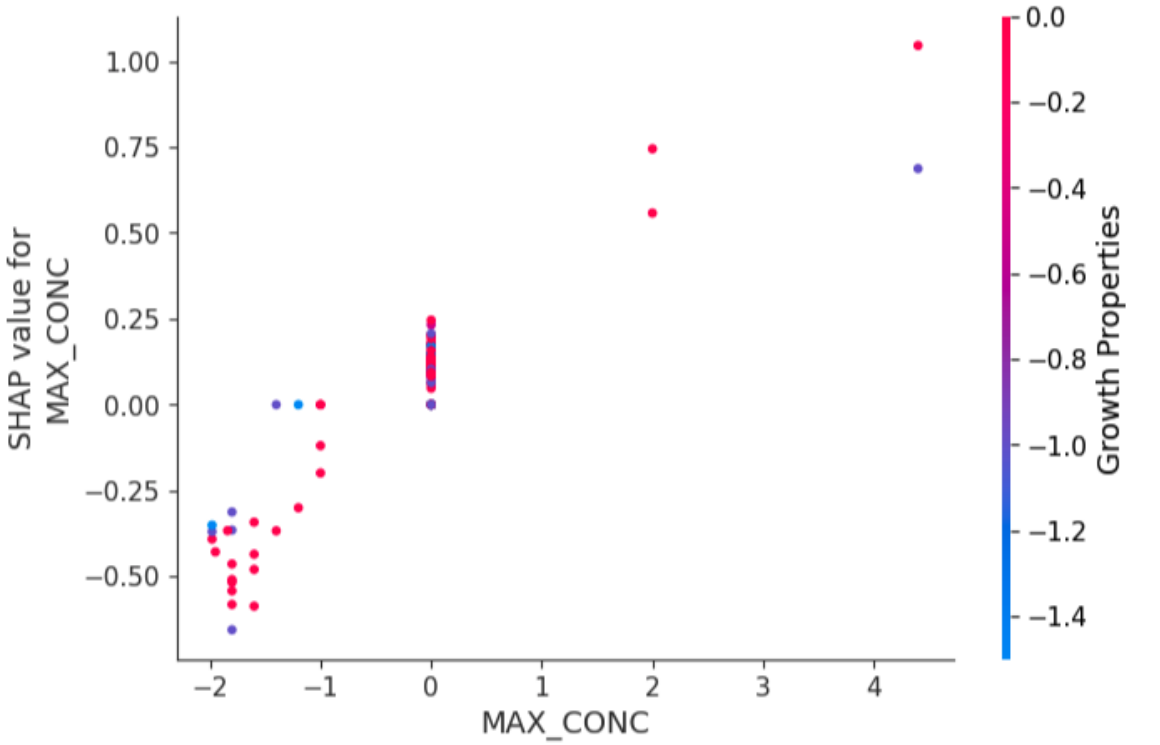

Figure 2.3: Example of Shap values representation: Dependence plot.

- Y-axis: The SHAP value for that variable, how much this feature contiributes to the model's output.
- X-axis: The value of the feature.
- Color: Indicate the second feature value.

In the example case shown in Figure 2.3, it can be seen that as the MAX_CONC variable decreases, it contributes less to a positive output, prompting a negative one.

3. Understanding the dataset

3.1. Introduction

To carry out this study, it is essential to identify a dataset that not only includes detailed information about patients' cancer diagnoses but also links each case to the administered drugs, their genetic profiles, and other relevant features. Furthermore, the dataset must meet a minimum standard of data quality. Given these stringent requirements, locating a suitable dataset is a non-trivial task.

After extensive research and evaluation of publicly available sources, the selected dataset is the Genomics of Drug Sensitivity in Cancer (GDSC) dataset [6], which is available through Kaggle. The primary aim of this resource is to make data public available from the original GDSC repository, including additional annotations to facilitate a downstream analysis.

The dataset includes detailed information about various cancer cell lines and their associated types. It provides genomic and molecular features such as mutation status, tissue origin, growth conditions and methylation patterns. On the pharmacological side it offers drug-specific information, including treatment response, administered concentrations and metadata related to drug selection. In addition, it includes a classification label that reflects the effectiveness of the drug response.

This research can be considered pioneering, as it aims to integrate genomic data, medication information, and patient responses into a unified predictive framework. Although some analyses are available in the dataset repository, a critical flaw has been observed in several of them: models often achieve surprisingly high accuracy by mistakenly including target-related variables among the input features, like RMSE, drug response or AUC. This results in information leakage, where the model has access to data it should not know at prediction time, thereby inflating performance metrics artificially.

Among the most critical variable is the logarithm of the half-maximal inhibitory concentration, denoted as LN_IC_{50}. This metric quantifies the drug dosage required to inhibit cell viability by 50%, making it a biologically meaningful and clinically relevant target for prediction. A model capable of accurately predicting LN_IC_{50} could serve as a decision-support tool in assigning optimal drug dosages to individual patients. Moreover, gaining insight into the factors that influence this value could offer valuable directions for further research in personalized medicine and oncology.

3.2. Description of variables

GDSC comprises a total of four files, each providing complementary information on the status of cancer cell lines, patient characteristics and prescribed medications. One of the most important features of this dataset is, that the information it contains comes from the COSMIC database [7], one of the most comprehensive collections on cancer cases and their treatment. In the following sections, each file will be briefly described along with the variables it contains, in order to provide an initial understanding of their relevance and potential utility for this research.

1. **GDSC2-dataset.csv**: Contains information about the medicines used and their effectiveness on the different patients. Among its variables are:
    - **DATASET**: Since the objective of the author in the mentioned dataset is to keep the information updated. He uses this variable to indicate which version it comes from, since the database *COSMIC* is updated regularly.
    - **NLME_RESULT_ID**: Unique identifier of the NLME model to which it is related.
    - **NLME_CURVE_ID**: Identifier of the dose-response curve fitted by the previous model.
    - **COSMIC_ID**: Identifier by which this record is known in the database *COSMIC*.
    - **CELL_LINE_NAME**: Name of the cancer cell line of the experiment.
    - **SANGER_MODEL_ID**: Identifier of the cell used by the Sanger Institute [8].
    - **TCGA_DESC**: Description of the type of cancer according to *The Cancer Genome Atlas*[1].
    - **DRUG_ID**: Identifier of the drug prescribed to the patient.
    - **DRUG_NAME**: Name of the drug used in the treatment.
    - **PUTATIVE_TARGET**: Refers to the original cellular target of the drug, i.e. which cell line the drug is intended to treat.
    - **PATHWAY_NAME**: The biological pathway affected by the drug, i.e. which series of cellular interactions the drug intake modifies.
    - **COMPANY_ID**: Identifier of the company that provides the drug.
    - **WEBRELEASE**: Date this information was published on the website.
    - **MIN_CONC**: Minimum used concentration of the drug during follow-up.
    - **MAX_CONC**: Maximum used concentration of the drug during follow-up.

[1]Project with the aim of cataloging genomic alterations due to the presence of cancer cells.

- **LN_IC50**: This variable represents the natural logarithm of the IC_{50} value (mean inhibitory concentration). A large value indicates that a greater dose of the drug is required to achieve the desired effect, suggesting lower sensitivity or resistance. Conversely, a lower value implies that only a small dose was sufficient to produce a significant biological response, indicating higher sensitivity to the drug.
- **AUC**: Area Under the Curve, a statistical measure that, in this case, reflects the effectiveness of the drug.
- **RMSE**: Error measure known as Root Mean Square Error which indicates the quality of the dose-response prediction performed by the NLME model.
- **Z_SCORE**: Standardized performance measure which intended to allow comparisons between different drugs and cell lines.

2. **Cell_Lines_Details.xlsx**: Contains information about the different cell lines, as well as data related to the patient's cancer. The variables it includes are:
   - **Sample Name**: Unique identifier for the cell line sample.
   - **COSMIC identifier**: Unique ID from the COSMIC database for the cell line.
   - **Whole Exome Sequencing (WES)**: Genetic mutation data obtained through whole exome sequencing.
   - **Copy Number Alterations (CNA)**: Data on gene copy number changes in the cell line.
   - **Gene Expression**: Information on gene expression levels in the cell line.
   - **Methylation**: Data on DNA methylation patterns in the cell line.
   - **Drug Response**: Information on how the cell line responds to various drugs.
   - **GDSC Tissue descriptor 1**: Primary tissue type classification which indicates the type of cancer.
   - **GDSC Tissue descriptor 2**: Secondary tissue type classification which can be interpreted as indicating the region affected by cancer.
   - **Cancer Type (matching TCGA label)**: Cancer type according to the TCGA classification.
   - **Microsatellite instability Status (MSI)**: Indicates the microsatellite instability status of the cell line.
   - **Screen Medium**: Growth medium used to culture the cell line.
   - **Growth Properties**: Characteristics of how the cell line grows in culture.
3. **Compounds-annotation.csv**: Information refering to the drug:
   - **DRUG_ID**: Unique identifier for the drug.

- **SCREENING_SITE**: Location where the drug screening was performed.
- **DRUG_NAME**: Name of the drug compound.
- **SYNONYMS**: Alternative names for the drug.
- **TARGET**: The molecular target(s) of the drug.
- **TARGET_PATHWAY**: The biological pathway(s) targeted by the drug.

4. **GDSC_DATASET.csv**: This dataset is the result of preprocessing performed by the original Kaggle author, based on three raw data files included in the same repository. The author applied extensive data cleaning and transformation techniques to reduce noise, enhance signal quality and incorporate relevant domain knowledge. The resulting file provides a curated version of the data that is ready for analysis and modeling. Among the variables included in this dataset are:

COSMIC_ID	**CELL_LINE_NAME**
TCGA_DESC	**DRUG_ID**
DRUG_NAME	**LN_IC50**
AUC	**Z_SCORE**
GDSC Tissue descriptor 1	**GDSC Tissue descriptor 2**
Cancer Type (matching TCGA label)	**Microsatellite instability Status (MSI)**
Screen Medium	**Growth Properties**
CNA	**Gene Expression**
Methylation	**TARGET**
TARGET_PATHWAY	

Table 3.1: List of variables present in the GDSC dataset.

Although the preprocessing of the GDSC_DATASET is very interesting, this data has not been used for this research. It eliminates some variables that were considered useful for improving the prediction results. For example, including "DRUG_RESPONSE" among the variables to be predicted could support the model understand some internal relationships in the data. However, given that the procedures applied could be helpful in resolving this research, some of the analysis provided has been taken into account.

3.3. Analyzing the information

The foccus is on LN_IC_{50}, which is intended to show how those drugs are adminstred to the patients. Therefore, the first step is to understand its distribution. This makes it possible to anticipate potential problems and to understand how it evolves.

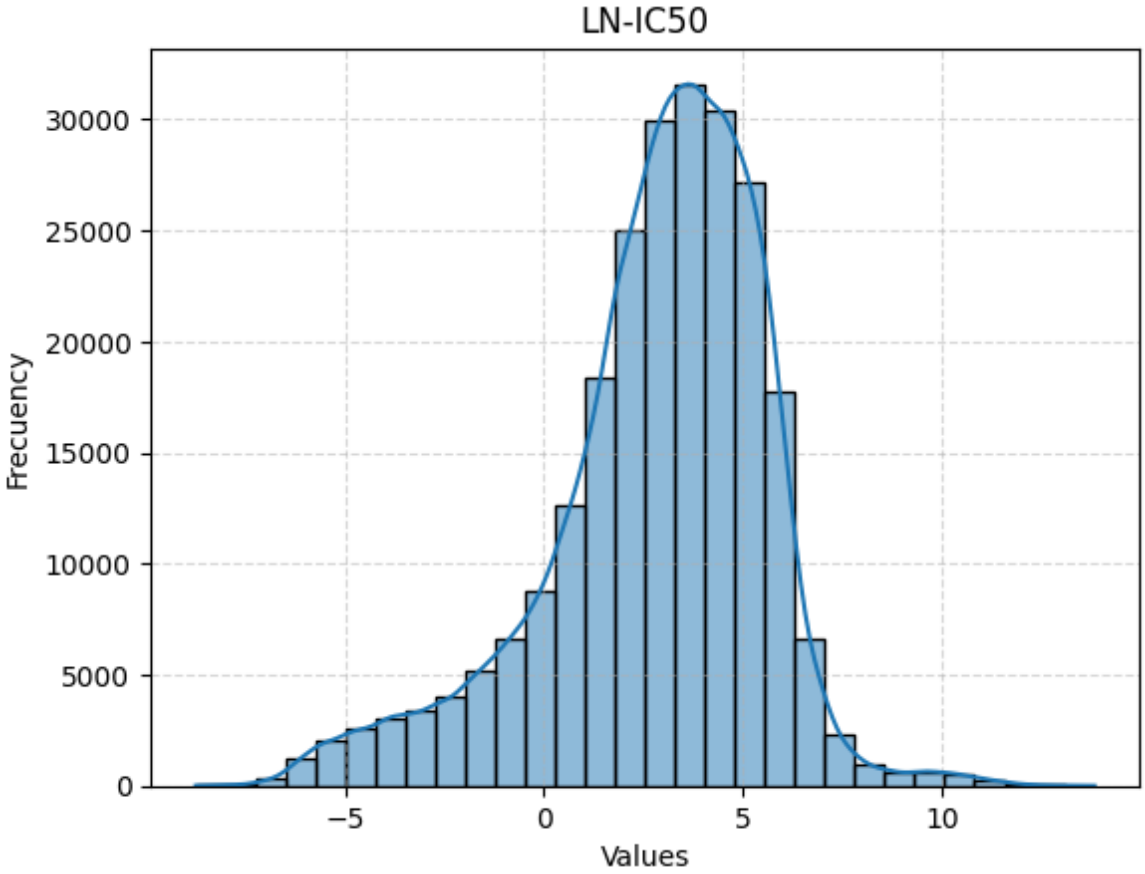

Figure 3.1: Distribution of the LN_IC_{50} variable.

Figure 3.1 shows how the LN_IC_{50} variable distribution has a Gaussian bell shape, in which some aspect can be observed:

- There are fewer examples on the right side of the distribution, which implies that the data has been slightly skewed when acquiring positive values.
- In the same area on the right, a small increase can be seen. This may indicate a population for which more records are available on an ad hoc basis, which could pose a small challenge during model training, as this population disrupts the natural progression of the normal distribution.
- In general it shows good symmetry, thus favouring the training of a model, ignoring the area mentioned above.
- The question arises as to whether the LN_IC_{50} variable follows a normal distribution or not. To find out, the Anderson and kstest tests were applied, visible in Listing 3, with the aim of obtaining an unambiguous answer. The result of these tests was that this variable does not follow a normal distribution.

Joining the data

Once the main target variable has been analysed, it is necessary to know how to join the files appropriately so that the records match their corresponding ones. Thus, referring to database terminology, it is needed to carry out various join operations between the three files, relating each of the foreign keys[2] to their corresponding primary keys.

There are unknown records in each of the three files, which show the following possibilities:

- Delete all null values and perform the joins afterwards.
- Join the three files and apply value imputation to try to infer the value of the unknown fields.
- Join the datasets and then delete the resulting null values.

To obtain the complete dataset, it is necessary to merge the GDSC2 and Cell_Lines_Details files using "COSMIC_ID" and "COSMIC identifier" keys respectively. Due to the fact both records come from different sections of the COSMIC database. The resulting dataset must be joined with Compounds-annotation using the "DRUG_ID" key, thus obtaining a record with all the data.

When making a decision, it is important to consider that value imputation could improve performance. However, if some of the records are deleted and assigned a status to those that are unknown, the future model could acquire a certain degree of resilience, strengthening the model. Based on this two decisions were made. Investigate two of the possibilities: in the first case, the datasets will be joined, applying a process of value imputation a posteriori. In the second case, null records will be deleted and then the files will be merged. Both resulting datasets will be used in a first regressor to check which data provides the most value. Although this may seem complex, the implementation is straightforward using notebook files[3], as the two workflows are nearly identical, except for the step involving imputation. This approach will be detailed in a subsequent section.

Filtering the information

Applying variable selection methods reduces the dimensionality of the dataset. This includes two key benefits: first, it reduces noise in the dataset and lowers the computational cost of training the model. Secondly, it reduces the potential of human error during the initial data collection process. In practical terms, if only 10 characteristics needed to be record instead of 20, the likelihood of introducing mistakes during data entry or measurement is significantly lowered.

The current dataset contains many identifiers related to external studies or various databases. These identifiers are disposable for this research.

[2]In the context of relational databases, it refers to an identifier which mutually referes to each other.

[3]Development environment that allows to run code organized in cells

- DATASET
- NLME_CURVE_ID
- COMPANY_ID
- COSMIC identifier
- Sample Name
- COSMIC_ID
- SYNONYMS

The next step is to check if any of the variables are obsolete. This condition occurs when all available records for that feature are the same. In order to get it, the code described in Listing 1 has been used. This makes it possible to identify obsolete variables and determine the size of each variable's domain.

Semantically, a one-to-one correspondence is expected between the variables DRUG_ID and DRUG_NAME, meaning that each drug ID should consistently map to a unique drug name. To evaluate this assumption, Pearson's correlation coefficient was calculated. The result, however, deviates from 1, indicating that the relationship is not perfectly consistent. To identify the specific instances of inconsistency, the function shown in Listing 2 was implemented. This function systematically traverses the dataset to ensure that each drug ID corresponds to the same drug name throughout. When mismatches are found, the function also attempts to determine which variable, either the ID or the name, is responsible for the discrepancy, thereby offering insights into potential data quality issues

This reveals that there is inconsistency in the names, i.e. some names are assigned to more than one identifier. To understand the nature of this inconsistency, the identifiers of the suppliers of these drugs are examined, as they may have a different identifier depending on the supplier, as can be seen in Listing 4.

Drug Name	**Detected IDs**	**Supplier IDs**
Docetaxel	[1007, 1819]	[1046, 1043]
Selumetinib	[1062, 1736]	[1046, 1001]
Oxaliplatin	[1089, 1806]	[1046, 1043]
Fulvestrant	[1200, 1816]	[1046, 1043]
Uprosertib	[1553, 2106]	[1046]
GSK343	[1627, 2037]	[1046, 1033]
Acetalax	[1803, 1804]	[1043]
Dactinomycin	[1811, 1911]	[1043, 1046]
Ulixertinib	[1908, 2047]	[1046]

Table 3.2: Detected inconsistencies between DRUG_NAME, DRUG_ID and their suppliers

As shown in Table 3.2, with the exception of three cases, there is an inconsistency

between the drug name and its identifier, moreover the medicine is supplied by a different provider. This information can be very useful, as variations between suppliers may involve subtle changes to some component of the drug formula, its preservation or concentration. Therefore, the drug name and drug ID variables will be retained in the dataset, while the company identifier will be omitted.

Modifying the representation of data

Many operations, such as computing correlations, visualizing data, or training machine learning models, require the dataset to be in a numerical format. Therefore, the next step involves transforming categorical variables into numerical representations. A widely used method is the one-hot encoding; however, this method significantly increases the dimensionality of the dataset, which may negatively affect performance and training time. As an alternative, ordinal encoding will be applied where appropriate, allowing the numerical representation to preserve the inherent order or semantic relationships of the original categories. For example, in a variable representing size with values such as *small*, *medium*, and *large*, them the values 0, 1, and 2 are assigned, which respectively maintains their natural ordering.

The following variables only express information of Yes, No or unknown value, making them clear candidates for replacement.

- Copy Number Alterations (CNA)
- Gene Expression
- Methylation
- Drug Response

The Growth Properties variable contains information regarding adhesion, differentiated into three levels: Adherent, Semi-Adherent, and Suspension. Therefore, applying a numerical substitution indicating the degree of adhesion represents a plausible option.

Finally, the Screen Medium variable only takes two values, ”R” and ”D/F12”, which can be replaced by 0 or 1.

In order to facilitate the application of these changes to future data, this entire process is carried out using Sklearn’s ColumnTransformer. This class allows you to perform a sequence of transformations on the data easily and effectively. The required steps are illustrated in Listing 5.

3.3.1 Checking the feasibility of assigning values

The imputation of unknown values could improve results, which is why research has been conducted in this context. The objective is to conduct a small test to verify whether imputing values improves accuracy or whether, on the contrary, allowing a certain degree of ambiguity in the data what improves the robustness of the model. To achieve this performance, following steps are needed:

To perform the imputation process, all variables must be represented as numbers. However, after the preprocessing described above, the dataset retains some categorical features. Since these variables cannot be expressed in numerical format while maintaining the relationship between them, the ideal solution would be to apply a one-hot encoder which in turn would significantly increase the computational cost of applying value imputation. Therefore, if Sklearn's ordinal encoder [9] is applied, categorical variables will be replaced by numerical ones, while it maintains the stored correspondence, if needed. This process can be observed in Listing 6.

Once the dataset does not contain categorical information, it is possible to apply imputation methods. During this research, the method selected was KNNImputer [10], which creates clusters between the data and imputes values based on the nearest centroids using the Euclidean distance. In other words, it assigns values based on their similarity to known data. The application of this technique can be seen in Listing 7.

Comparing results

Comparing the results between applying value imputation and assigning values to nulls after the unification process will allow us to determine which method is best for the research.

To perform the comparison, a neural network architecture consisting of three fully connected (dense) layers was implemented, as detailed in Listing 8. This structure was chosen to ensure a fair and consistent baseline for evaluating the impact of preprocessing. The network was trained both with and without imputed values. The results from each training scenario were then collected and analyzed to indicate the influence of the values imputation.

Metric	Value
MSE	0.843
RMSE	0.918
MAE	0.674

Table 3.3: Model performance in the test set after applying value imputation.

Metric	Value
MSE	0.320
RMSE	0.566
MAE	0.300

Table 3.4: Model performance in the test set without applying value imputation.

As can be seen in Tables 3.3 and 3.4, the model's performance is severely compromised after applying value imputation. Everything seems to indicate that adding some ambiguity to the data does indeed make the model more robust and versatile.

4. In search of solutions

4.1. Solving our regression problem

The estimation of the variable LN_IC_50 is considered highly beneficial in the fight against cancer. If this value can be accurately predicted, cases in which excessive medication is prescribed, potentially harming the patient's system, could be avoided. Patients requiring higher doses could be identified, thereby improving their outcomes during the course of the disease. To achieve this goal, it is essential that the target variables provided in the dataset are considered. Even if they are not all the primary objective, they can support the learning process of the algorithm, enabling more comprehensive and accurate predictions.

In line with this approach, the selected target variables, besides the LN_IC_50 indicator, include the AUC curve, RMSE value, Z-Score, and the Boolean drug response. These variables contribute to understanding the behavior and characteristics of LN_IC_50 across various blood samples, drugs, cancer types and other factors. Despite the preprocessing already performed, many variables still remain unordered or categorical in nature, such as blood samples, drug names or tissue descriptors. Therefore, one-hot encoding[1] will be applied to these features. This will be done using the get_dummies function from the pandas library, which is already being utilized for various other operations. However, this transformation may be viewed as controversial, since it significantly increases the dataset's dimensionality, from 21 variables to 1339.

The next step to prepare the data for model training, is to divide it into three distinct subsets: a training set, a validation set and a test set. This division helps ensure that the model learns from one portion of the data the training set, while its performance is periodically evaluated on a separate subset the validation set, reducing the risk of overfitting. During training, the model attempts to minimize error on the training data, but its hyperparameters and general behavior are refined based on its performance on the validation set. Finally, once the model is fully trained, its ability to generalize is assessed using the test set, since it is a collection of data it has never seen before. This evaluation provides an unbiased estimate of the model's performance on new unseen data.

Although this step may appear straightforward, it is indispensable. It is essential to ensure that these subsets are representative of the overall data distribution. If the data is split in an unsupervised or careless manner, it may result in certain populations being excluded from the training set. This, in turn, would hinder the model's ability to learn from those data groups and ultimately reducing the generalization capacity of the model.

[1] This is a method for converting categorical data into numerical format. Instead of assigning arbitrary numeric values to categories, one-hot encoding creates a binary vector for each category [11].

Therefore, after separating the data using sklearn's train_test_split method [12], the various distributions have been represented using bar charts. This allows to check the differences between the distributions in order to detect anomalies between them.

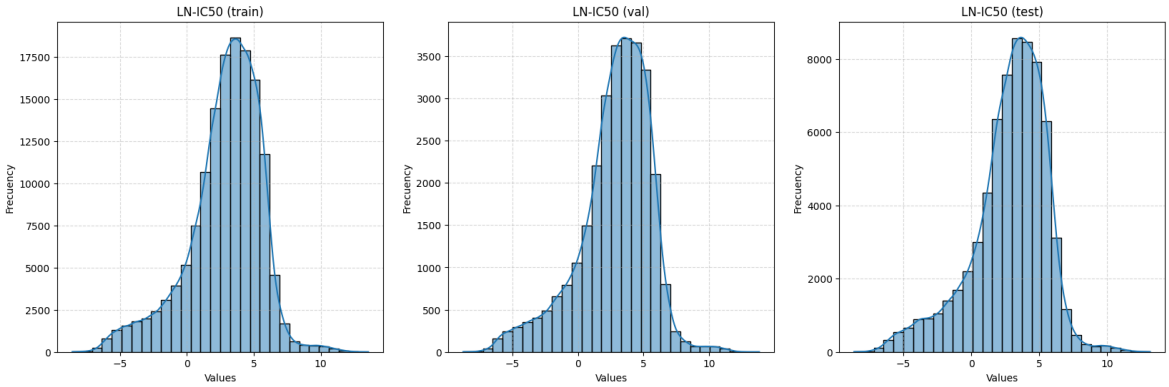

Figure 4.1: Distributions of the variable LN_IC_{50} in the different subsets.

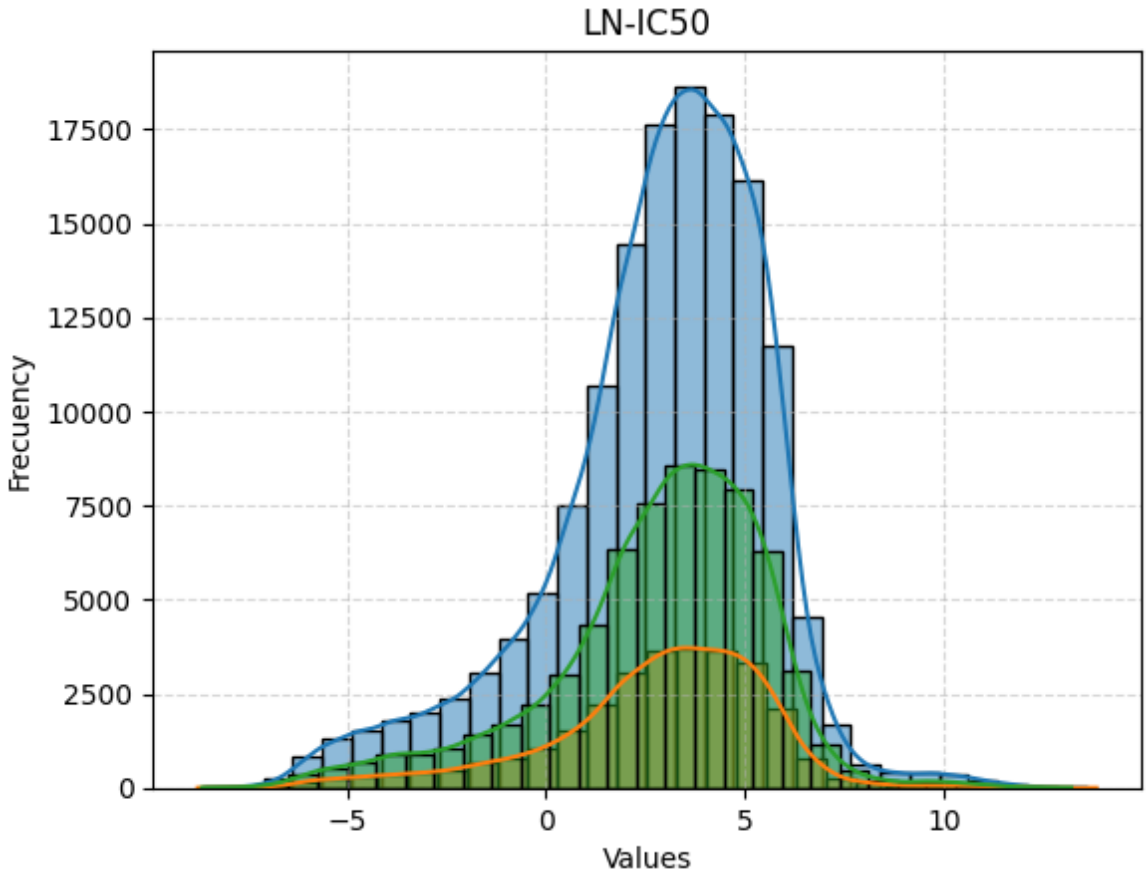

Figure 4.2: Distributions of the variable LN_IC_{50} in the different subsets represented in the same graph to facilitate comparison.

In order to achieve better results, the data can be standarize. This process involves representing all the information on the same scale, i.e. which is specially

important for algorithms like neural networks. Scikit-learn provides severals functions for this purpose, such as the StandardScaler [13], which use the mean and the standar desviation to standarize the data.

The transformation applied by `StandardScaler` follows the formula:

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

Where x is the original value, μ is the mean of the feature, and σ is the standard deviation.

Although the `StandardScaler` is widely used in many research studies, in this work it was decided to choose to use the `RobustScaler` [14] instead. This method has two main advantages: it standardizes the data while also reducing the influence of noise in the dataset. In particular, it is less sensitive to outliers, which is a highly valuable property when it is necessary to avoid misleading trends caused by extreme values. `RobustScaler` applies the formula:

$$x' = \frac{x - \text{median}(x)}{\text{IQR}(x)}$$

$$\text{IQR}(x) = Q_3(x) - Q_1(x)$$

Where x is the original value, x' is the scaled value, $median(x)$ is the median of that feature and IQR is the interquartile range[2].

Given their proven effectiveness in similar contexts, both neural networks and XGBoost are selected as candidate algorithms for model training.

- **Neural networks**: Neural networks are well-suited for handling high-dimensional data and typically outperform other algorithms in such contexts. In addition, applying convolutional layers could improve this predictive performance, by capturing local patterns or dependencies between related samples. This is particulary relevant, as the main aim to uncover potential relationships between blood samples, the drugs and the cancer react.
- **XGBoost**: Tree based model used to get great performance with tabular data, even they are able to overcome or match deep learning algorithms, like neural networks [15]. Moreover, training a tree-based model is typically faster and easier, compared to training a neural network, which takes more time. Another key advantage is interpretability, as tree models provide clearer insights into decision-making processes. However, they have one notable disadvantage: they are not able to extrapolate beyond the range of the train data due to how the algorithm learn [16].

[2]The difference between the first and the last quartile.

4.1.1 Predicting LN_IC_{50} using Neural Networks

Tensorflow is the framework selected for this reasearch due to it provides a smooth learning curve and includes built-in tools which helps to visualize the architecture and training progress.

During this research the following callbacks[3] will be used:

- PlotLossesKerasTF [17]: This package allows real-time visualization of training loss and other metrics during model training.
- ModelCheckpoint [18]: This callback save the best-performing model based on a chosen evaluation metric.
- EarlyStopping [19]: Sometimes the model is not able to learn more. In this situations is common that the validation loss does not improve. When the moment in captured, the training is stopped using this callback. This enables to reduce the training time and to avoid overfitting. There are some occasions, where the model is not able to improve the validation results after a few epochs. In order to fix this, the callback needs patience of 10 epoch.
- ReduceLROnPlateau [20]: Similar to the last one, this callback point out, in case the model stop of learning. In that moment it reduces the leaning rate in order to get an improvement in validation results. The patience considered was 7, as when this callback is applied the improvement is slower.

Comparison of loss functions

One of the main hyperparameters[4] that must be configured during model development is the loss function. The choice of an appropriate loss function can significantly influence the quality of the results. Therefore, a function that aligns well with the nature of the problem should be selected. In this context, three potential candidates have been considered:

- **MSE [21] (Mean Squared Error)**: Measures the averages squared difference between predicted values $\hat{y}$ and the real ones y. It is sensitive to large errors, but also to outliers. Sice the units are sqaured is less interpretable than other metrics like RMSE [22].

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

- **logCosh [23]**: It combines the best of MSE and MAE [24], it is les sensitive to outliers and smoother than MAE. This function follows the formula:

$$\text{LogCosh}(y, \hat{y}) = \sum_{i=1}^{n} \log\left(\cosh(\hat{y}_i - y_i)\right)$$

[3] Callbacks are functions or routines that are automatically called at specific points during training, such as at the end of an epoch or after a batch. They are commonly used for tasks like saving models, early stopping, adjusting the learning rate or logging training metrics.

[4] A configuration variable set before training, which often determines whether the model learns effectively.

This means in small errors it works like MSE and in the large ones, it grows more slowly like MAE. In addtion, it is derivable in all his domain, which ideal for neural network training.

- **Huber [25]**: This loss function try to replicate the principes of LogCosh, it works as squared error when this one is low, and it is linear for large errors. It requires tunning one hyperparameter, δ.

$$L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq \delta \\ \delta \cdot (|y - \hat{y}| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

Development of our regressor model

In order to determine the most appropriate approach for the research, some of methods were defined to increase the dynamism of the research. These can be found in the appendix in Listing 8, 9 and 10.

By default, the RMSE will be displayed during our training, just to interpret the learning progress.

Getting the tools to compare error functions

The initial training was conducted using Mean Squared Error (MSE) as the loss function. While simple, MSE is one of the most effective metrics for gaining insight into how well the model is learning during the early stages of training.

The training progress is illustrated in Figure 4.3.

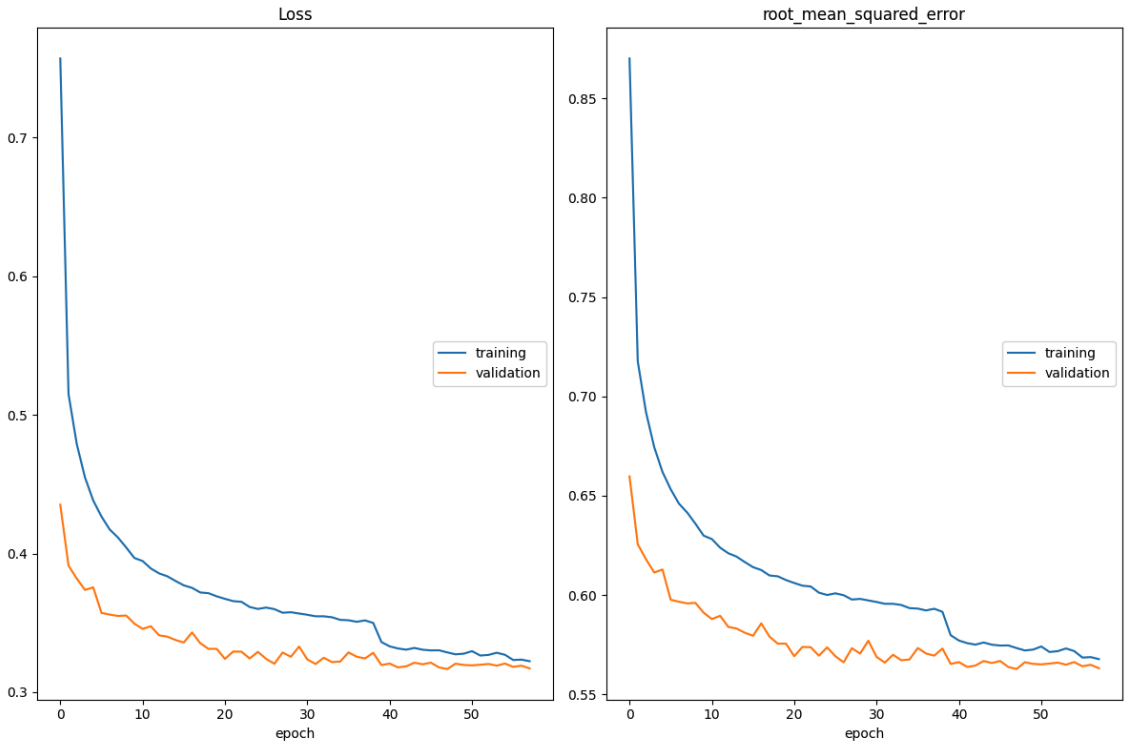

Figure 4.3: Study of the feasibility of employing MSE as a loss function using a neural network.

A summary of the training and validation performance, in terms of loss and RMSE, is provided in Table 4.1, which shows the final values of the loss function and RMSE at the end of training.

Metric	Best value
Loss (validation)	0.317
RMSE (validation)	0.563

Table 4.1: Training and validation performance summary (loss and RMSE).

Finally, the overall performance of the model on the evaluation dataset is summarized in Table 4.2:

Metric	Value
MSE	0.320
RMSE	0.566
MAE	0.300
R^2	0.575

Table 4.2: Performance metrics on the test set using MSE as loss function.

Figure 4.4 illustrates the relationship between the predicted values generated by our model and the corresponding actual values. In an ideal scenario, all points would lie precisely on the red diagonal line, which represents perfect predictions.

Although not all predictions fall exactly on this line, the majority are closely clustered around it, indicating a generally good predictive performance. However, it is worth noting that in cases where the actual values are particularly high, the model tends to underpredict, suggesting a limitation in capturing extreme values accurately.

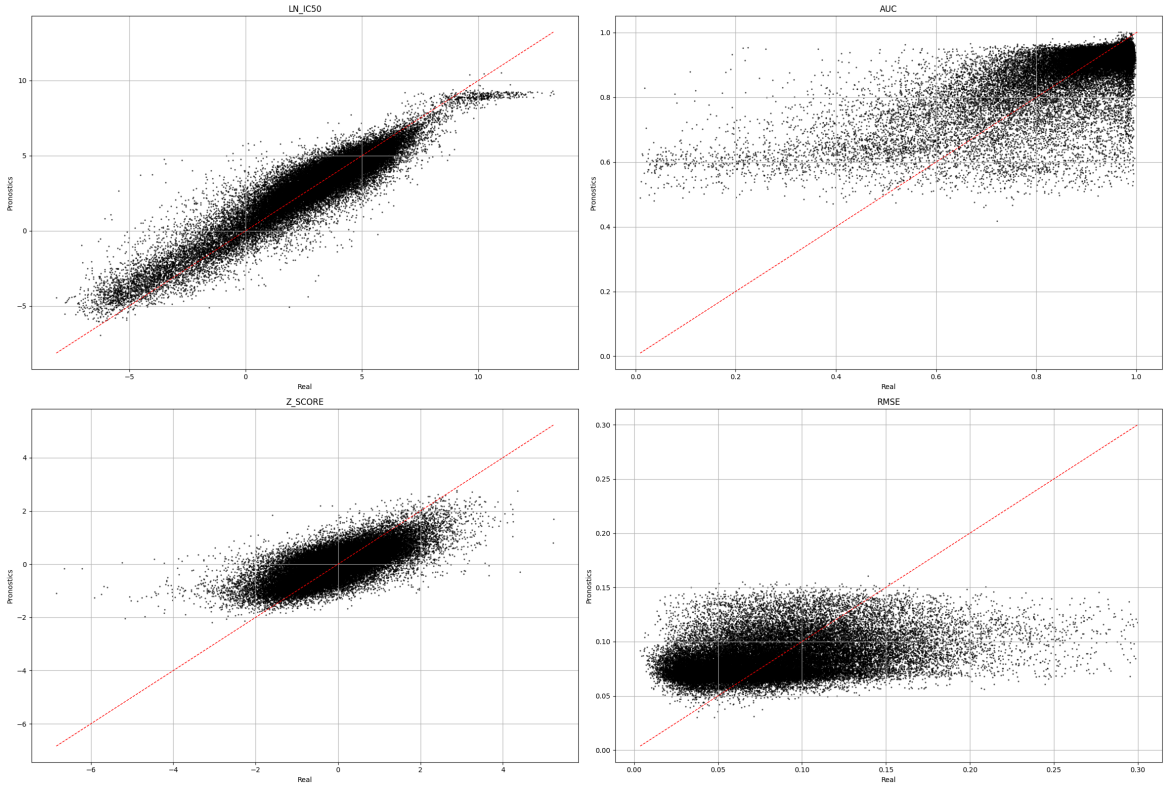

Figure 4.4: Result of employing MSE as a loss function using a neural network.

To further the research, the same process will be repeated with LogCosh as loss function. This could be a nice alternative because it works better with outliers than MSE and also it has a good performance with small error. For this reasons this one could be a great alternative.

In Figure 4.5 si displayed the difference between training and validation, something similar to what it was seen in Figure 4.3. But there is a clear difference, in the previous case it seems that the model is not able to learn more, while in this one more fluctuations can be observed during learning, which allow us to see how each time the model makes a mistake and increases the error, it rectifies in the next step. This change is due to the new loss function. While ReLU is unable to represent negative values, LogCosh can.

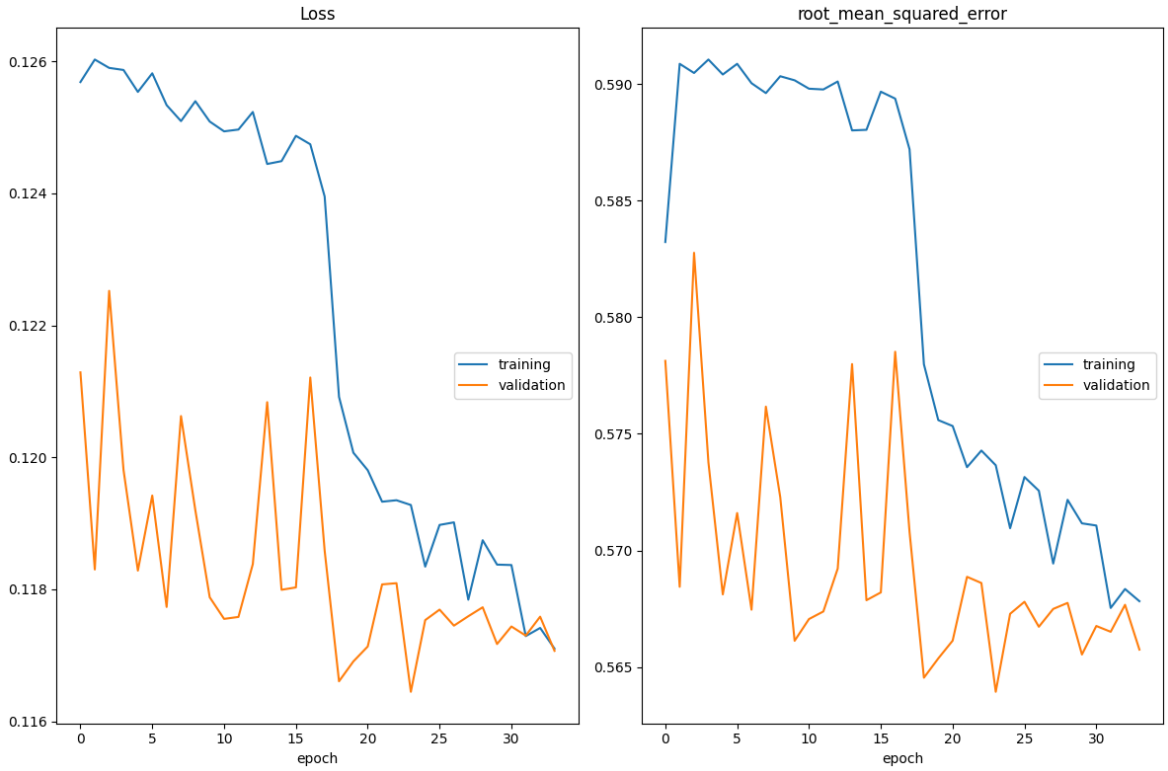

Figure 4.5: Study of the feasibility of employing logarithm of the hyperbolic cosine as a loss function using a neural network.

In this case the predictions over the test set were a little bit worse that the previous one.

Metric	Best value
Loss (validation)	0.116
RMSE (validation)	0.564

Table 4.3: Validation performance metrics at the end of training using LogCosh.

Metric	Value
MSE	0.321
RMSE	0.567
MAE	0.299
R^2	0.577

Table 4.4: Performance metrics on the test set using LogCosh as loss function.

Although at first glance the results shown in Figures 4.4 and 4.6 may appear nearly identical, a more in depth analysis reveals key differences. In Figure 4.4, the model trained with MSE appears to exhibit slightly less dispersion in its predictions, as the plot presents a more defined shape compared to that of Figure 4.6.

However, the evaluation of training progress tells a different story. In Figure 4.3, it can be observed that after only a few training iterations, the model

reaches a plateau, showing little to no further improvement—an indication of early stagnation. In contrast, Figure 4.5 displays a continued downward trend in the loss function, even in the final training stages. This suggests that the model trained with Log-Cosh loss retains the potential for further optimization, making it a more promising candidate than the one trained with MSE.

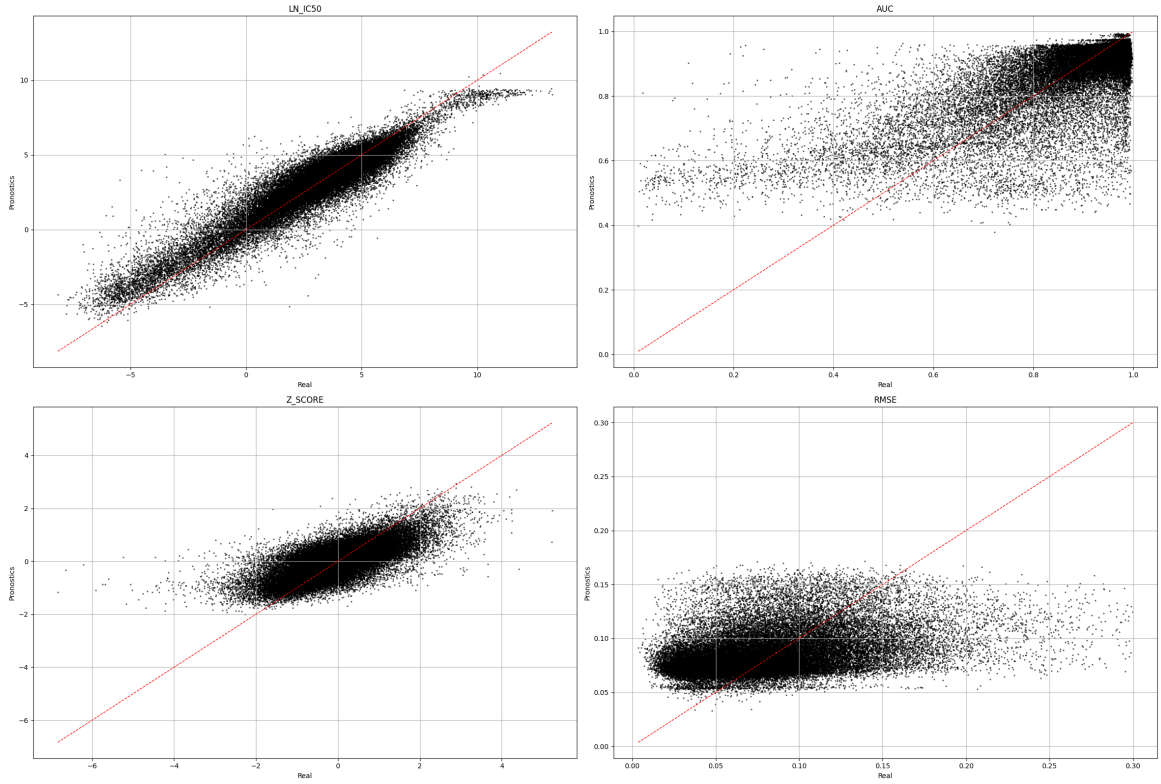

Figure 4.6: Result of employing logarithm of the hyperbolic cosine as a loss function using a neural network.

Finally, a new model is trained using the Huber loss function. Just as a reminder, this function and LogCosh have similar target but different points of views. The both are robust to reduce the impact of outliers, but this one require one hyperparameter, δ, which one is used as thresshold.

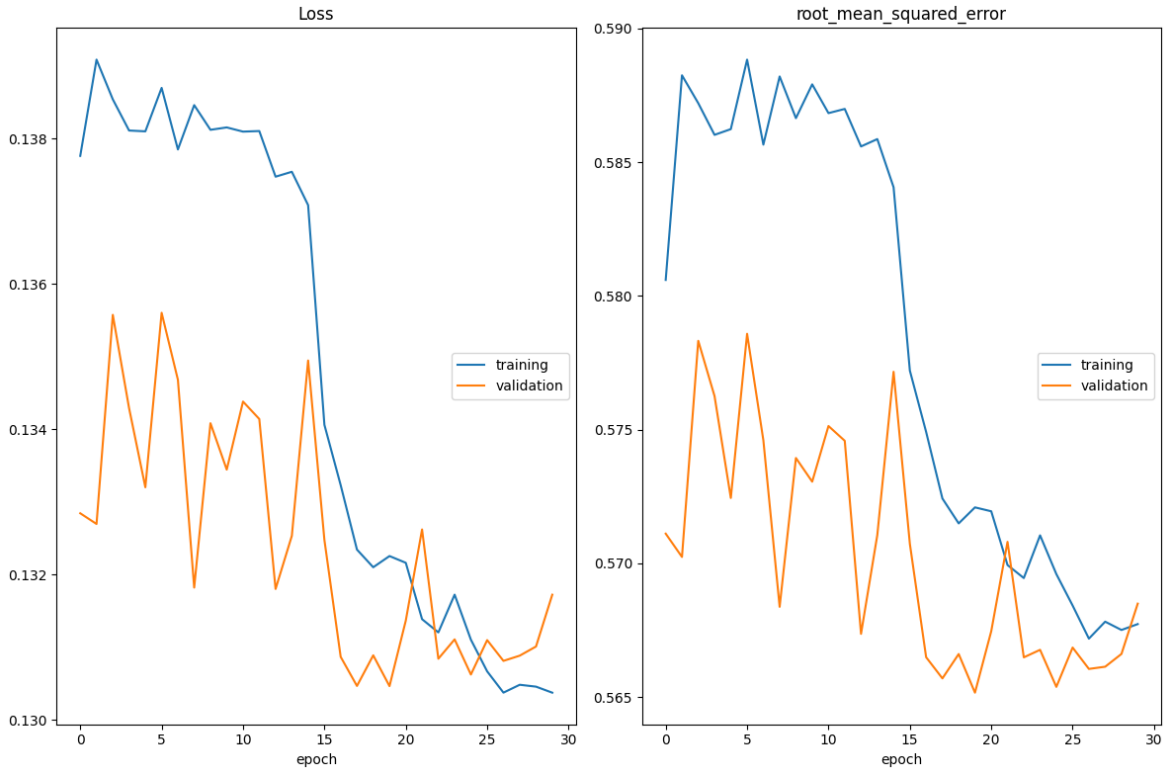

Figure 4.7: Study of the feasibility of employing Huber as a loss function using a neural network.

Metric	Best value
Loss (validation)	0.130
RMSE (validation)	0.565

Table 4.5: Validation performance metrics at the end of training using Huber loss.

Metric	Value
MSE	0.321
RMSE	0.567
MAE	0.297
R^2	0.586

Table 4.6: Performance metrics on the test set using Huber as loss function.

At the beginning of the training, the model trained with the Huber loss function seems really interesting, since, as illustrated in Figure 4.7, the results on the validation set are better than on the training set, a sign that the network is able to generalise certain behaviours. However, as the epochs go by, this difference not only reduces but the validation error exceeds the train error, gradually acquiring a clear upward trend, as can be seen in the same figure. This represents a major drawback, since if the model does not acquire or loses the ability to generalise, the results provided by the model are worthless, as it is not able to understand the nature of the progression.

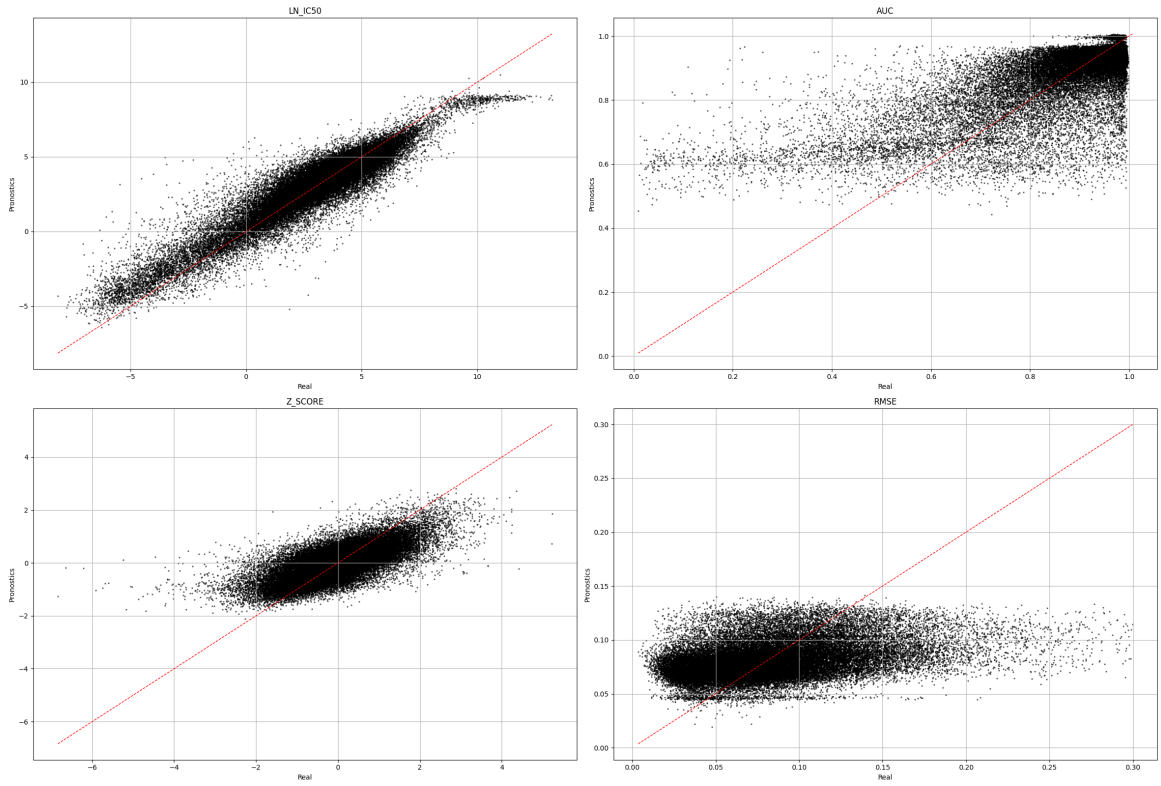

Figure 4.8: Result of employing Huber as a loss function using a neural network.

The three loss functions seen are interesting, as the implemented architectures that use them are able to estimate the correct form of LN_IC_{50} in a similar way. However, this is only the beginning of defining the final model, so the best loss function will be the one that gives us a hint that it may be the start of something more promising.

Based on the results, everything seems to indicate that using the MSE error as a loss function will probably not allow us to make much progress. As previous discussed, the learning of the model that implements it does not present large variations, since the epoch 25. Only minimal improvement in validation is observed after appliying the learning rate reduction callback.

The decision then lies between LogCosh and Huber. According to the graphs of the respective trainings, both experience oscillations in the error obtained, rectifying each time the error as the training progresses. However, as discussed in the previous analysis, Huber seems to contain an upward current from the last stage of training, while LogCosh tends to reduce the error. This is a key feature, as regularisation techniques could be used to enhance these signs of reduction and achieve a better result.

Applying convolutional layers to our problem

Once the error function has been selected, attention must be directed toward defining the architecture of the network responsible for performing the regression.

When thinking about how to organize the neurons there are many possible

options, and there is no formula that indicates how to do it. So it is a trial-and-error process, with a lot of research and reading involved. Because keeping abreast of current events can save us from unnecessary testing. Something really interesting, since this type of training is not too fast.

Several studies [26, 27, 28] explore the use of one dimensional convolutional (1D-CNN) layers to adress problems involving tabular data. However, the effectiveness of 1D-CNN depends on the existence of a natural relationship between the variables that comprise the dataset, such as genetic sequences or cellular information. Otherwise, this type of network is not appropriate, as it may fail to capture meaningful patterns or dependencies in data lacking inherent sequential structure. In cases where the main objective is to capture local interactions between variables, 1D CNN layers represent one of the most suitable architectures that can provide meaningful insights into such relationships. This is specially significance because one of the primary goals is explain the different aspect involved in the increases or decreases of the LN_IC_{50} variable, in order to apply that knowledge to future investigations.

After several researches [29] is of special significance. This study iontroduce an architecture that combines a soft-ordering mechanism with 1D CNN layers to solve a regression problem, as ilustrated in Figure 4.9. To enhance performance a dense layer is first applied to the input data to capture the spatial data or sequential relationships, without maintaining a strict order. The output is then reshape to the required dimension to allow processing through several convolutional layers.

During the process, a branch is extranted from the initial convolutional layers. Once the main flow line passes through the last convolutional, the second flow are merged into the first one by summation, to avoid gradient vanishing. The output of this summation is subjected to average pooling and flattening before reaching the last layer, dense one as mentioned before. Finally, a dense layer performs the predictions.

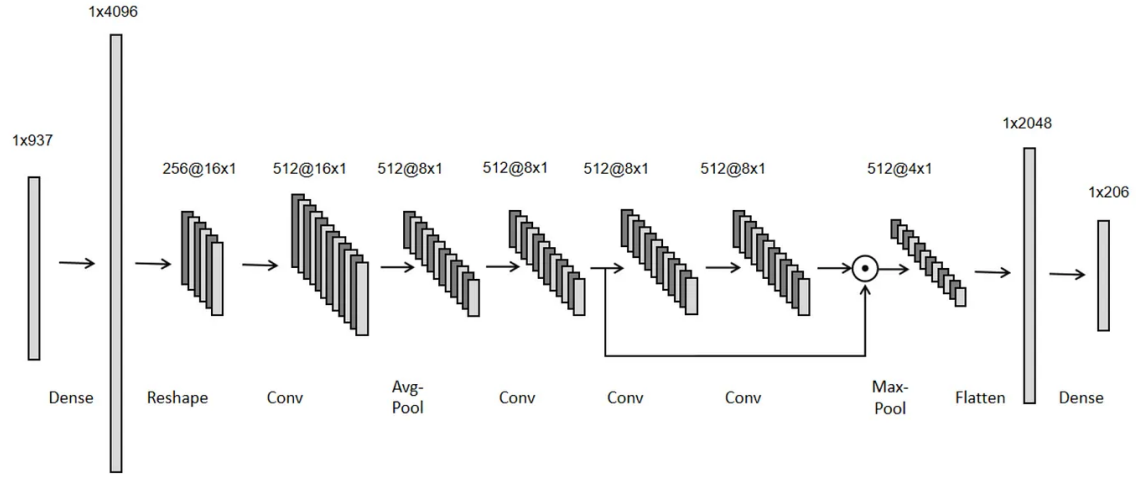

Figure 4.9: Soft-ordering with 1D convolutional architecture.

The process of reducing the error consisted of two stages.

1. Adapting the proposed architecture to the Kaggle data using tensorflow.
2. Modifying the initial version to incorporate techniques that address issues present in the proposed dataset.

Adapting the proposed architecture to the Kaggle data using tensorflow

The implementation was carried out using Tensorflow, based on the proposed scheme. As shown in Listing 11, various components from the library were utilized, such as the different types of layers, activation functions required and different regularisation techniques. This elements contributed an improvement in prediction quality, as illustrated in Figure 4.11. In this figure, the discrepance between the predicted and actual values of the LN_IC_{50} variable is visible reduced, indicating lower dispersion and a more accurate model, as reflected in the metrics in Table 4.7. In addition, it is very important to note that Figure 4.11 no longer shows the large error that when LN_IC_{50} variable took high values.

Metric	Value
Loss	0.098
RMSE	0.504

Table 4.7: Model performance metrics on the test set during the architecture adapting phase.

Despite improved results, it is still possible to achieve a better quality model.

To identify potential improvements to the model, it is important to analyze Figure 4.10, which illustrates the evolution of the loss function during training. As shown in the figure, the model initially exhibits high error, which progressively decreases as it learns to adjust its parameters. At a certain point, the loss drops significantly, suggesting that the model begins to generalize key patterns relevant to the regression task. However, this improvement plateaus after a few iterations, with no further reduction in error.

Moreover, a considerable gap is observed between the training and validation errors, with the training loss being significantly lower. This discrepancy suggests that, although there is no clear indication of overfitting, the model performs considerably better on the training set than on the validation set. This implies that it may not be capturing generalizable patterns well enough to accurately predict unseen data. Reducing this gap could therefore help to achieve better results.

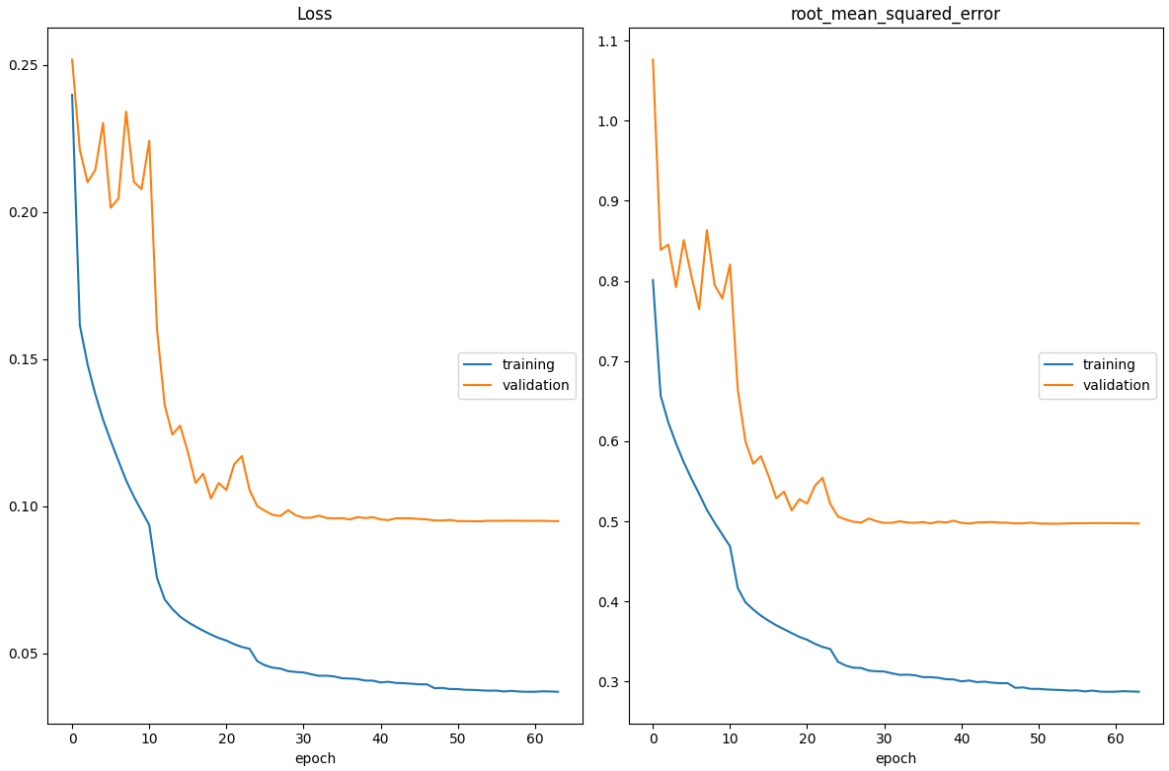

Figure 4.10: Training a neural network using convolutional layers and addition as a bridge.

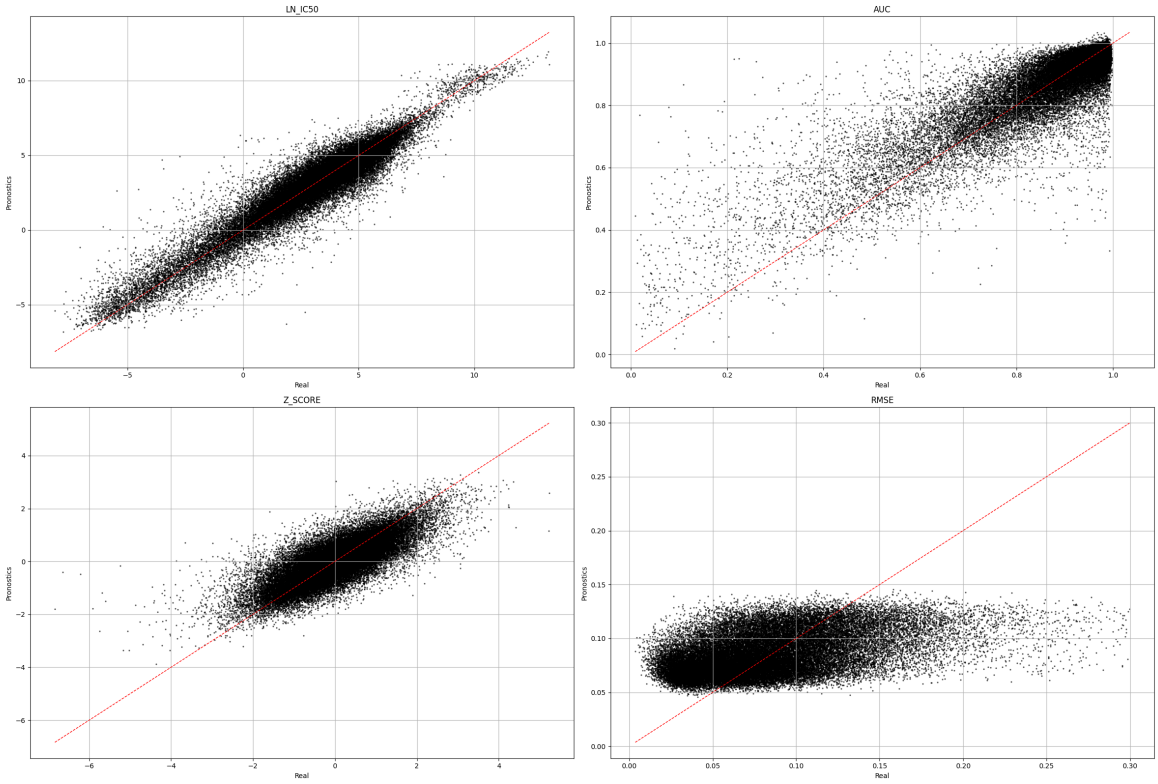

Figure 4.11: Result of employing a neural network using convolutional layers and addition as a bridge.

Modifying the initial version to incorporate techniques that address issues present in the proposed dataset

To mitigate overfitting, numerous experiments were conducted by increasing and varying different types of regularization techniques. However, the results remained largely unchanged, with only marginal improvements in error reduction. In contrast, experimenting with different activation functions led to more promising results. Initially, the functions suggested in the original paper were tested, but subsequently, other activation functions available in TensorFlow were explored. Among these, the SiLU function [30], also known as Swish, proved particularly effective. It significantly reduced the gap between training and validation errors.

This improvement can be attributed to the characteristics of SiLU compared to ReLU and the specific nature of our dataset. SiLU is a smoother function with outputs close to zero, as illustrated in Figure 4.12, making it more suitable in scenarios where small negative values may carry meaningful information. Unlike ReLU, which zeroes out all negative values, SiLU retains them with a small but non-zero output, allowing a more nuanced interpretation.

One of the main drawbacks of SiLU is its higher computational cost, as it requires computing a sigmoid function in addition to a multiplication operation. In contrast, ReLU simply computes the maximum between zero and the input value. The formula for SiLU is:

$$\text{SiLU}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}} \quad \text{where } \sigma(x) \text{ is the sigmoid function}$$

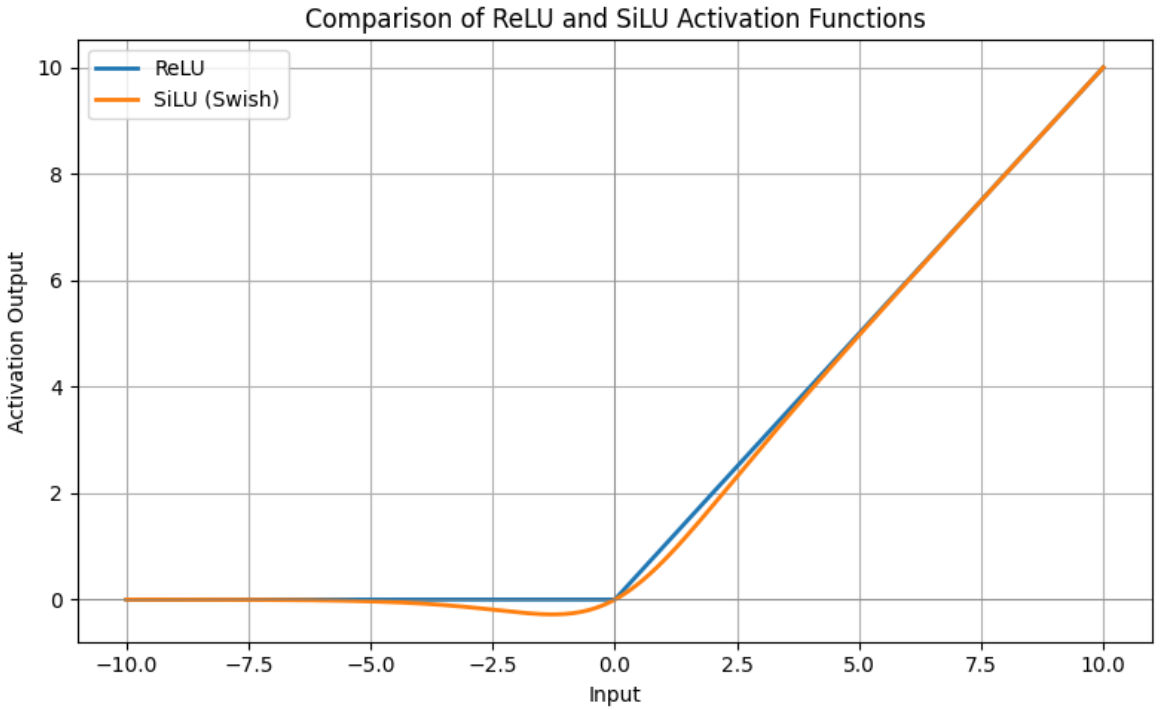

Figure 4.12: Comparision between SiLU and ReLU. In this picture it is observed the differeces between them around zero.

On this basis, the implementation, shown in Listing 12, was carried out again, adjusting some of the hyperparameters related to the regularization and the number of layers. As a result of these modifications, the difference between the train and valiadation was reduced, showing a narrower gap, as is ilustrate by Figure 4.13.

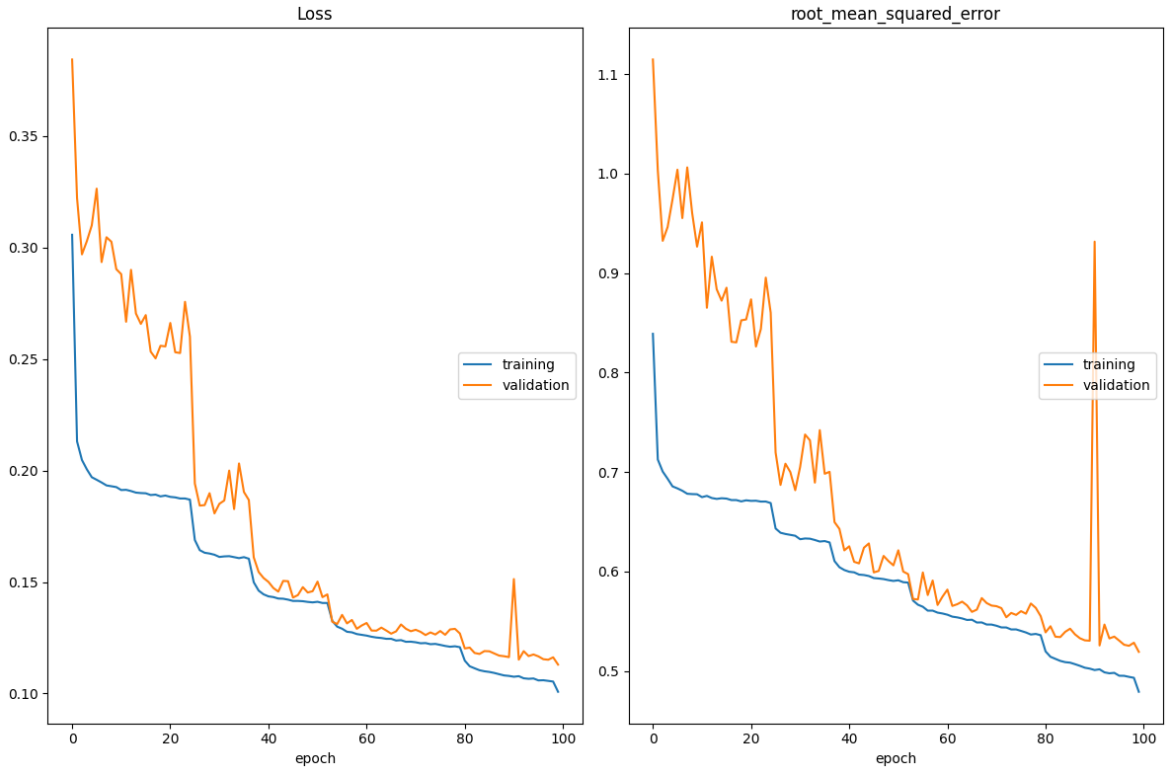

Figure 4.13: Training a neural network using convolutional layers and addition as a bridge, increasing the regularization and using SiLU as activation function.

Metric	Value
MSE	0.270
RMSE	0.520
MAE	0.257
R^2	0.685

Table 4.8: Performance metrics on the test set after applying the SiLU activation function.

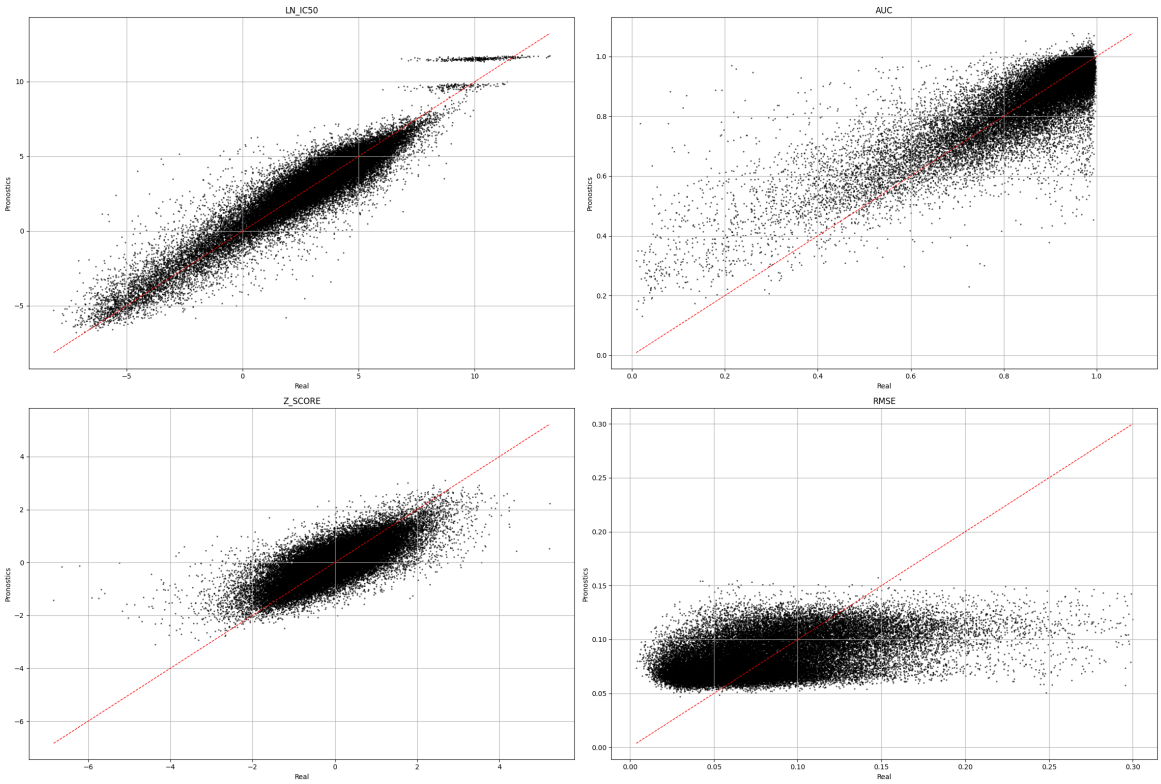

Figure 4.14: Result of employing a neural network using convolutional layers and addition as a bridge, increasing the regularization and using SiLU as activation function.

After this improvement in training stage, another challenge was found. The error was increased, as can be seen in Figure 4.14. Not only did the spread for the predictions for the LN_IC_{50} variable become wider, but the model also performed worse when predicting high values in comparation with the previous version.

This prompted a new line of research focused on improving the prediction results, trying to avoid increase the gap between training and validation again. In this situation, the best alternative was to investigate how to improve an architecture based on convolutions for other types of problems, in particular those related to images, since this is where most research has been done.

Custom architecture

After extensive research, it became clear that one way to cancel out neurons or drastically reduce their impact was through multiplication [31, 32]. Thus, thinking about how to adapt this mechanism to the current architecture came to the fore, as it could be the answer to the problem. If possible, the model could learn to discriminate between points that are of no interest, providing better results.

Motivated by this idea, various ways of modifying the architecture to introduce multiplication were considered, two of which stood out:

- Perform the addition as planned in the original model and then apply the multiplication. This adaptation caused problems in the performance of the

model.

- Replace the addition with multiplication, thereby reducing the freedom of the model compared to the previous approach. This design acts as an attention-like filter, helping the model distinguish important from irrelevant patterns.

After observing the results of both proof-of-concept tests, the decision was clear. The implementation focused on replacing one operation with the other. As a result of this modification, the model provides better results in the test set, as evidenced by the metrics presented in Table 4.9. Additionally, Figure 4.16 shows predictions with a lower degree of dispersion than any previous representation. This marks a key a milestone in the research, since the model achieved at this point is capable of estimating the value of the LN_IC_{50} variable with great precision. Notably, it maintains its performance even for data points with exceptionally high LN_IC_{50} values, not as it happened in Figure 4.14.

Given the absence of significant discrepancies between validation and test errors during training, this model is a strong candidate for applying explainability techniques. The insights derived could offer meaningful contributions to oncological research aimed at advancing cancer treatment

Metric	**Value**
MSE	0.240
RMSE	0.490
MAE	0.242
R^2	0.694

Table 4.9: Performance metrics on the test set after incorporating the multiplicative gating mechanism.

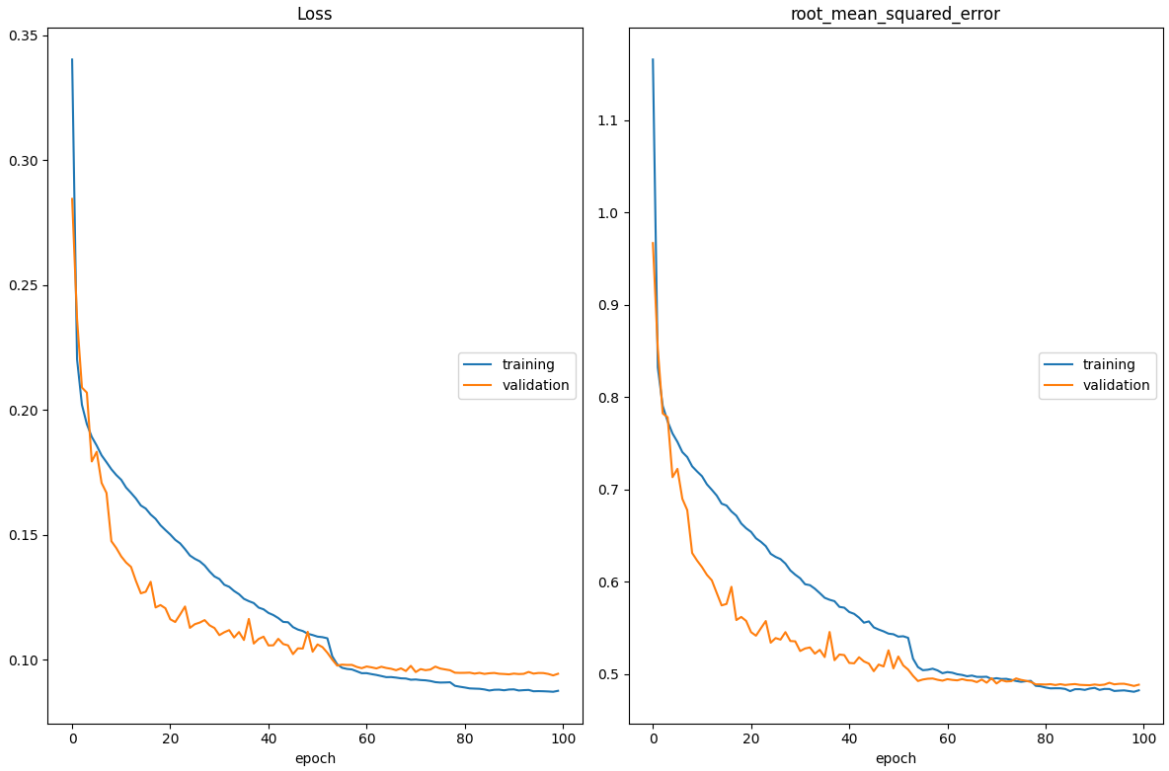

Figure 4.15: Training a neural network using convolutional layers and multiplication as a bridge.

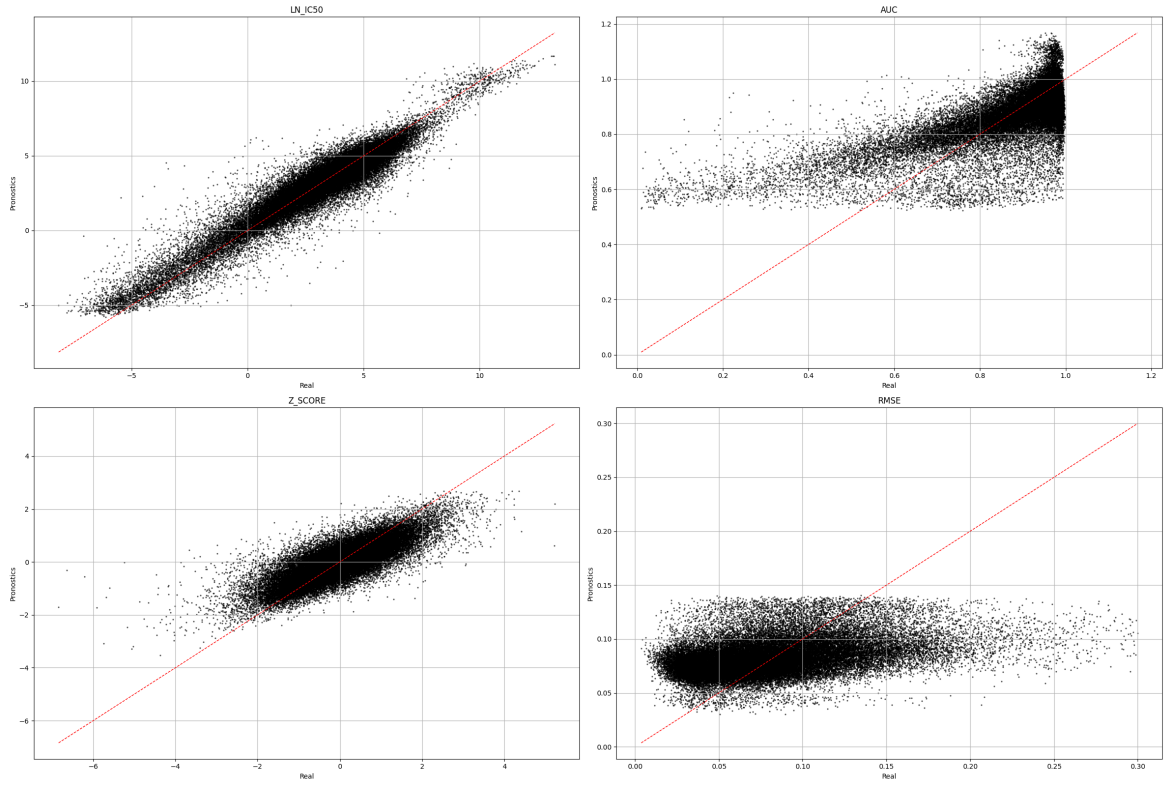

Figure 4.16: Result of employing a neural network using convolutional layers and multiplication as a bridge.

4.1.2 Application of XGBoost Regression Trees to our problem

Currently, much of the research being conducted uses neural networks, but one should never assume that one type of architecture is always optimal for all cases. For this reason, this study includes some experiments with tree-based algorithms as well, in order to increase the knowledge of the problem. As stated at the beginning of this chapter, the chosen algorithm is XGBoost because it has shown very good results in several articles [33, 34].

Tree-based models have a major disadvantage compared to networks, they are typically poor at extrapolation, meaning they struggle to predict values outside the range observed during training. But even so, they are characterised by their great robustness and, due to their nature, they allow the elimination of variables without adding cost, since it is the algorithm itself that determines the features that are relevant to achieve the objective.

To enable a fair comparison between models, an XGBoost model was trained using the same dataset that was previously used for the neural networks. This approach ensures that both models are evaluated under identical conditions, allowing us to assess their respective performance accurately.

Several training sessions were performed using the XGBoost library available for Python, through which the following results were obtained:

Metric	**Value**
MSE	1.253
RMSE	1.119
R^2	0.842

Table 4.10: Performance metrics of the evaluated model on test set using XGBoost.

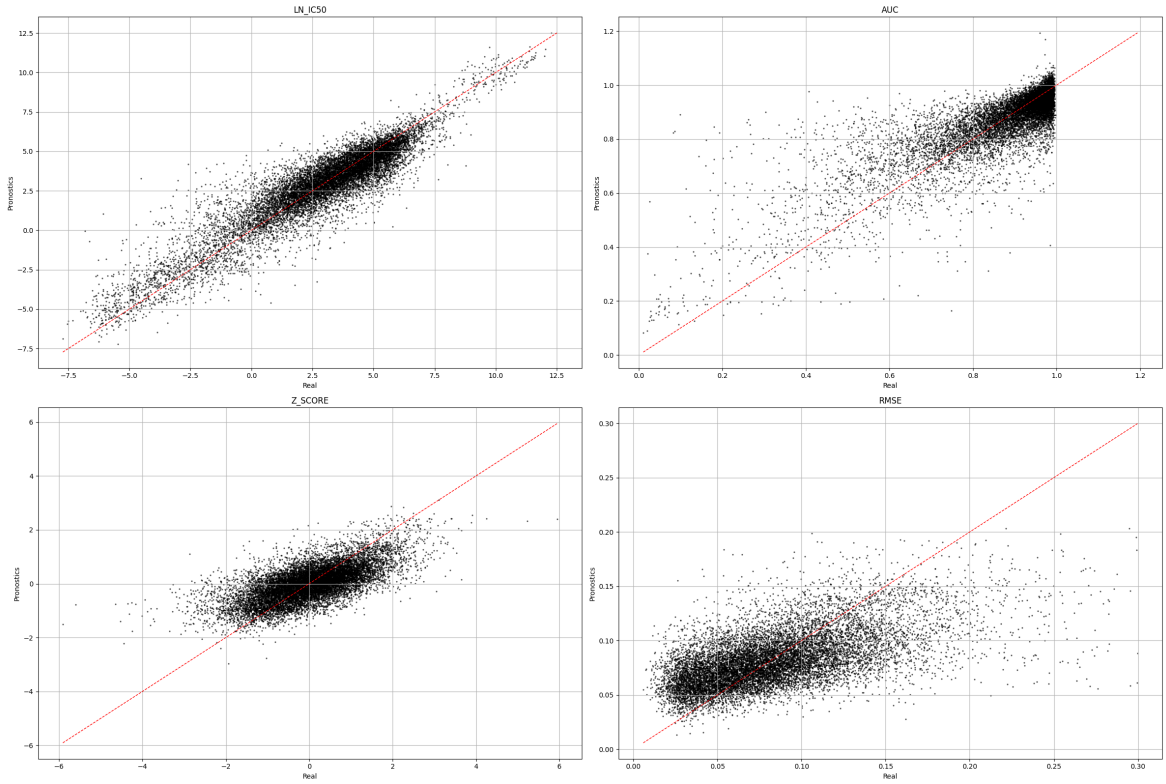

Figure 4.17: Results of XGBoost with target variables encoded using one-hot.

Examining the performance of the model trained with XGBoost, as shown in Figure 4.17 and Table 4.10, a decline in accuracy is observed compared to the neural network-based models. Specifically, the results suggest that XGBoost yields poorer predictions for the LN_IC_{50} variable, while slightly improving the error for the remaining target variables. In other words, it increases the error in LN_IC_{50} while reducing it for the others. However, this behavior is not desirable, as LN_IC_{50} is the primary variable of interest. The other target variables serve merely as auxiliary signals to support the model during training and help improve its performance on the main target.

To improve the results, several hyperparameter optimization algorithms were applied, including GridSearch [35] and RandomSearch [36]. However, due to the presence of numerous features encoded using one-hot encoding, executing these processes required substantial computational resources, particularly in terms of RAM. This limitation posed a challenge, as such resources were not readily available. Various alternatives were evaluated to address this issue. Ultimately, ordinal encoding [37] was applied to the one-hot encoded variables. This approach significantly reduced the number of columns, enabling the execution of different methods aimed at identifying dispensable variables. Among the algorithms used for this purpose were RFE [38] and SelectFromModel [39].

In order to narrow down the variables to be eliminated, both feature selection methods, SelectFromModel and RFE, were run and those that were common to the two subsets generated were chosen. In this way, it was possible to reduce the number of variables to only 8, from 20 that were initially generated. Subsequently,

RandomSearch was applied to try to find a good combination of hyperparameters, which one can be seen in Table 4.12.

After that, the model was trained again. Two training were executed, the fir, using the variables with the ordinal encoder, and the second training with one-hot coding. Unfortunately, both results were of lower quality than those offered by the neural networks, as can be seen in Table 4.11, which shows an increase in error. The best-performing model among the tested configurations was the one using one-hot encoding. It achieved the following evaluation metrics:

Metric	Value
MSE	1.318
RMSE	1.148
MAE	0.855
R^2	0.827

Table 4.11: Performance metrics of the XGBoost model using one-hot encoding after hyperparameter optimization and variable selection.

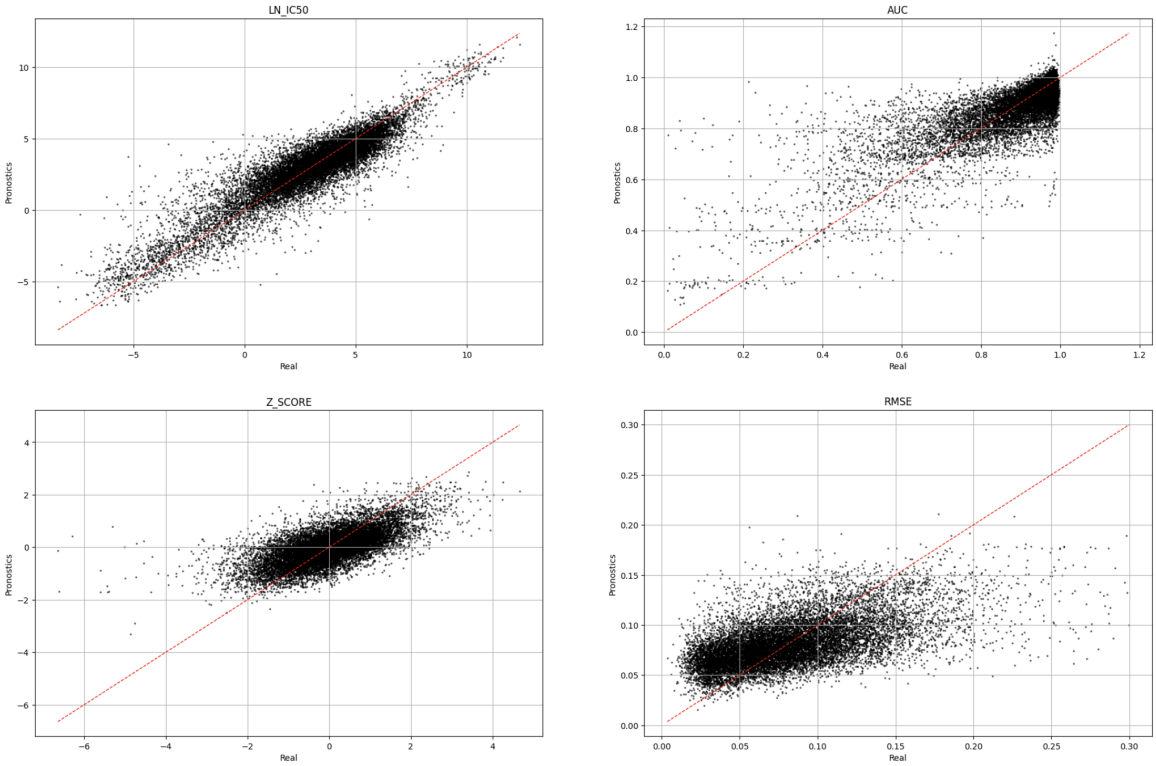

Figure 4.18: Representation of predictions using XGBoost, together with feature reduction and hyperparameter optimisation.

Parameter	Value
n_estimators	869
seed	123
learning_rate	0.028
max_depth	7
reg_alpha	0.040
reg_lambda	0.492
max_leaves	328

Table 4.12: Hyperparameters of the gradient boosting model after applying hyperparameter optimization.

4.2. A classification problem

Until now, research has been conducted using target variables in continuous format, with the exception of Drug Response, obtaining, by means of neural networks, a regressor capable of predicting the LN_IC_{50} variable and obtaining an RMSE error of less than 0.5. Having acquired this base, it is the ideal moment to bring our predictor closer to a case closer to real life. To achieve the variable LN_IC_{50} is discretised in such a way that it could be divided it into 30 ranges of equal distance between them, this allows to obtain an approximation to what would be a dose. This number of ranges was chosen because it provided the best balance between a sufficiently high number of categories and the overall accuracy of the predictions.

In other words, the range in which the value of LN_IC_{50} is found must be understood as the number of pills the patient should take. This idea makes a lot of sense, as getting closer to the concept of prescribing a certain number of pills allows to be somewhat more lax with the error made. For example, if two ibuprofens are taken instead of one, the negative effect is not appreciable. However, this is not the case when six are taken instead of one. Therefore, it can be inferred that making a slight error is not as serious as making a large one, an aspect that should certainly be considered when calculating the error.

Before any model is trained, the data will be visualized graphically to assess their separability, specifically, to determine whether the class ranges overlap in feature space or not. Since the dataset, after applying the ordinal encoder, contains more than 20 variables, direct representation is not feasible. Therefore, a dimensionality reduction process will be performed to project the data into 2D and 3D spaces, allowing for easier interpretation.

First, t-SNE [40] will be used to perform the compression process. This algorithm is highly effective at capturing local relationships; that is, within a group, the relationships between data points can be accurately preserved. This will allow an assessment of whether points belonging to the same group are located close together in space. If they are not, it indicates higher data variance, which would make it more challenging for a model to correctly predict group membership.

For this purpose, the t-SNE method from scikit-learn [41] will be used, as it enables visualization with only a few lines of code. However, the algorithm is

computationally expensive. To accelerate execution, the data obtained from the initial run will be saved using NumPy, so that subsequent executions can bypass the computational step. This process is illustrated in Listing 13.

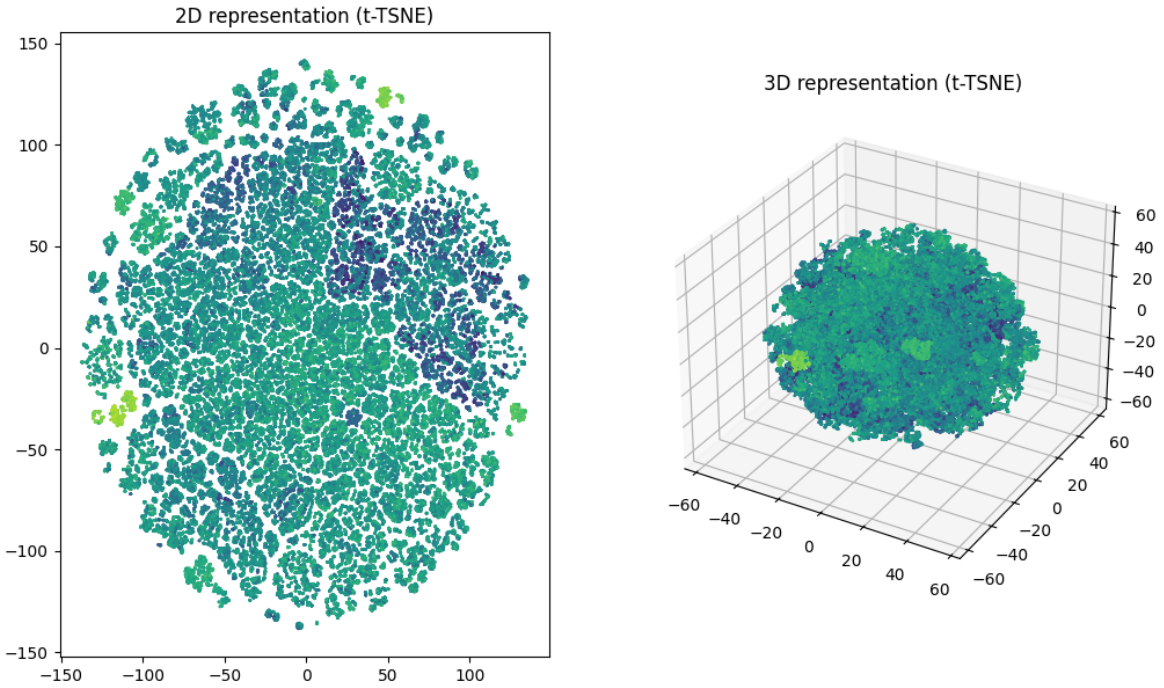

Figure 4.19: Representation of the data by t-SNE, where it can be seen the separation between points of the same cluster.

As a result, the representation shown in Figure 4.19 is obtained, in which the groups appear to be clearly distinguishable. However, a new question arises: although the clusters seem to be located close to one another in the plot, it is not certain whether this proximity reflects the true structure of the data, since t-SNE does not accurately capture global relationships, that is, the relationships between different groups.

To address this, UMAP [42] will be used, as it is capable of preserving both local and global non-linear relationships. The implementation will be carried out using the umap-learn library [43], as demonstrated in Listing 14. Similar to t-SNE, UMAP is computationally expensive; therefore, the same data-saving strategy will be applied to avoid repeated computations in future executions.

As a result, it can be observed in Figure 4.20 that the clusters are indeed well separated. Therefore, it can be assumed a priori that applying predictive models may yield good results.

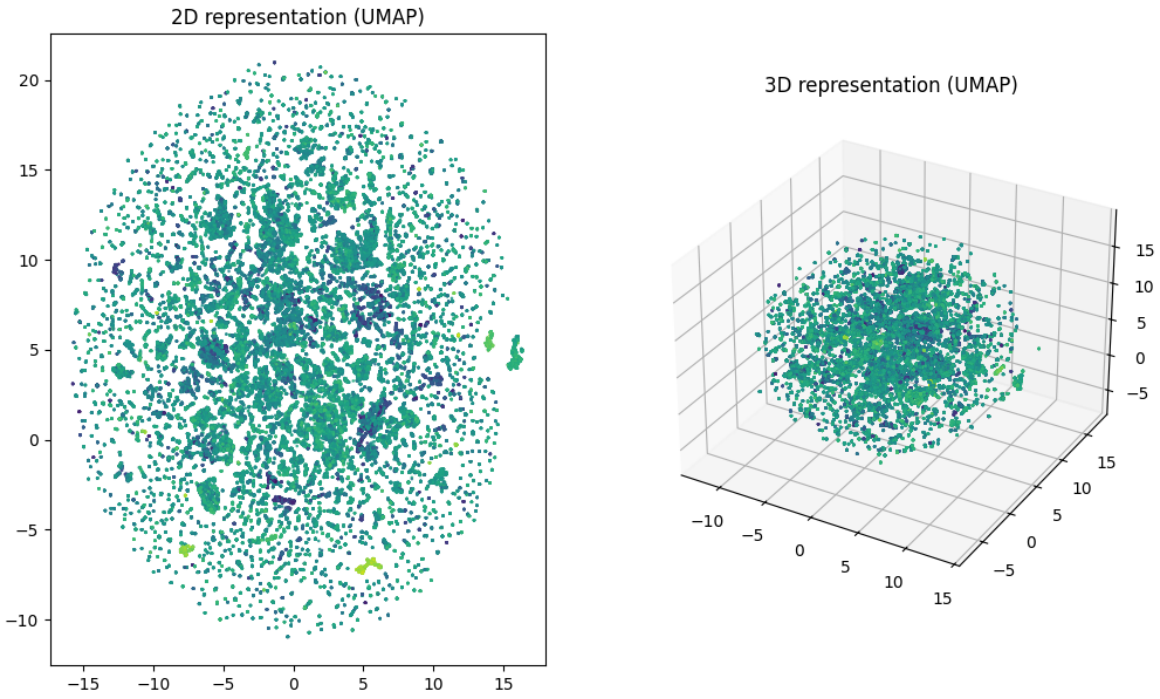

Figure 4.20: Representation of the data by UMAP, where it can be checked the distance between the different groups and their internal elements.

4.2.1 Discovering the correct group using Neural Networks

The objective of this section is to estimate the number of doses required by a patient using a neural network. To accomplish this, the previously developed architecture must be adapted to effectively model the problem, beginning with the dataset that includes a discretized version of the target variable.

The process has been divided into three main stages:

1. Adaptation of the existing architecture.
2. Expansion paths
3. Refinement of the loss function.

Adaptation of the existing architecture

Building on the previously trained neural network model, the objective of this section is to obtain a new network capable of estimating the number of doses of a drug needed to treat a person.

To gain an insight into the initial state of the problem, initial training should be carried out with only the necessary changes. In this way, a minimum performance level is established. To achieve this, it is necessary to change both the loss function and the model output.

Thus, the error function was replaced by categorical cross-entropy, with the aim of testing the model's accuracy. This loss metric is one of the simplest. With regard

to the model output, Softmax is the most recommended for this type of classification problem, as it provides a probability for each possible class.

As there is more than one target variable, the network must have more than one output layer simultaneously, as can be seen in Listing 15.

Once model training began, it successfully completed 100 epochs. The learning curves, shown in Figure 4.21, reveal that the model achieves better performance on the validation set than on the training set,an uncommon but noteworthy behavior. However, despite this apparent advantage, the overall quality of the predictions remains limited, with accuracy values not exceeding 0.4, as can be seen in Table 4.13. This indicates that, although generalization may be occurring, the model still struggles to produce reliable predictions.

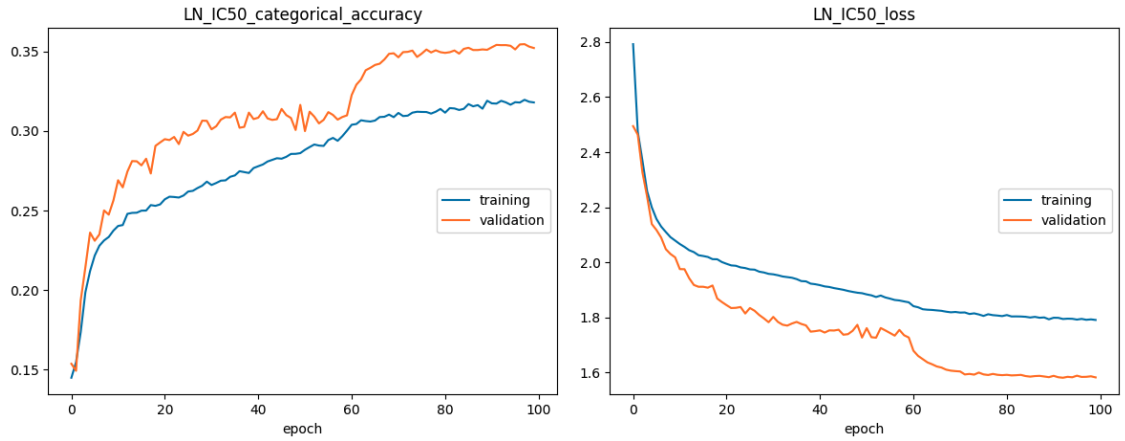

Figure 4.21: Model training progress during the first adaptation. This reflects a validation advantage over training.

Validation Metric	Best Value
LN_IC50_Categorical_Accuracy	0.355
LN_IC50_Loss	1.580

Table 4.13: Best validation results for LN_IC50 metrics.

Expansion paths

Refining the architecture at this point in the research is a very important task, so it is necessary to be very careful with the changes made. For this reason, two different alternatives will be evaluated. On the one hand, the loss function will be replaced by another that better fits unbalanced data. Since the histogram in Figure 3.1 shows there are classes with few examples within the dataset. On the other hand, there is the possibility of assigning priorities to the output layers, so that the network tends to favour improving the predictions of one variable over another. The current model has 5 outputs, so this idea could represent a significant improvement.

Expansion paths: Loss function

Among the loss functions suitable for classification tasks involving imbalanced classes, categorical focal cross-entropy is particularly worth considering. As an extension of the standard categorical cross-entropy, it introduces mechanisms specifically designed to mitigate the effects of class imbalance by emphasizing harder-to-classify examples. Applying this loss function can offer a useful indication of whether the class imbalance is significantly influencing model performance.

The formula of this function is:

$$\mathrm{FL}(y, \hat{y}) = -\sum_{i=1}^{C} \alpha_i \, y_i \, (1 - \hat{y}_i)^{\gamma} \log(\hat{y}_i)$$

- C: Total number of classes.
- y_i: True label for class i (one-hot encoded).
- $\hat{y}_i$: Predicted probability for class i.
- α_i: Class weight (optional, used for class balancing).
- γ: Focusing parameter (typically set to 2).
- $(1 - \hat{y}_i)^{\gamma}$: Modulating factor that down-weights easy (high-confidence) examples.

Since the implementation is being carried out using the Tensorflow API, making this change is very simple. It only requires changing one line of text, as can be seen in Listing 16.

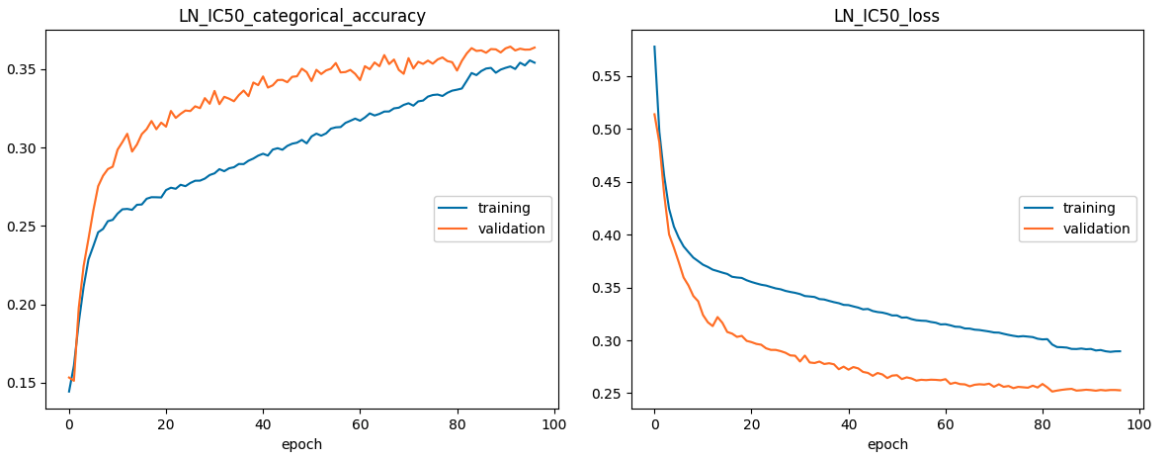

Figure 4.22: Model training progress applying categorical focal loss.

Validation Metric	Best Value
LN_IC50_Categorical_Accuracy	0.364
LN_IC50_Loss	0.252

Table 4.14: Best validation results for LN_IC50 metrics using focal loss.

The results obtained after this new modification can be extremely interesting. Both models, the previous one and this one, perform better on the validation set than on the training set. However, in the previous case, it can be seen that it was necessary to reduce the learning rate halfway through training in order to continue improving. In this case, the learning rate did not need to be reduced to obtain a substantial improvement in performance, as illustrated in Figure 4.22. After reducing it in the second training, no significant improvement is observed as can be seen in Table 4.14.

This can lead to two main ideas:

- The dataset has some unbalanced classes. Therefore, to learn to classify them correctly, the first model must reduce the learning rate in order to increase its accuracy. The second model, thanks to the loss function it incorporates, does not require this reduction.
- In turn, the first model was able to significantly improve the quality of its predictions after decreasing its learning rate, while the second did not. This could mean that by using cross-entropy the model can improve more, while with the focal version, it seems that training has reached a plateau.

Expansion paths: Priorities

Due to the high number of outputs in the current architecture, the model may attempt to learn all targets with equal emphasis, or worse, prioritize outputs that are not related to LN_IC_{50}. To address this, a weighting strategy was proposed in which the importance of each output is adjusted. This aims to improve the prediction performance for LN_IC_{50} by giving it higher priority during training.

This modification was implemented using the Keras Functional API, as shown in Listing 17. The loss contributions from all output layers were maintained, except for the one corresponding to LN_IC_{50}, which was assigned 50% more weight. The objective is to ensure that the model continues to consider the auxiliary outputs while explicitly encouraging better performance on the primary target, LN_IC_{50}.

Thanks to this approach, a considerable improvement was achieved during training, as shown in Figure 4.23 and Table 4.15. Here, both accuracy and error have improved. Once again, it was necessary to reduce the learning rate during training.

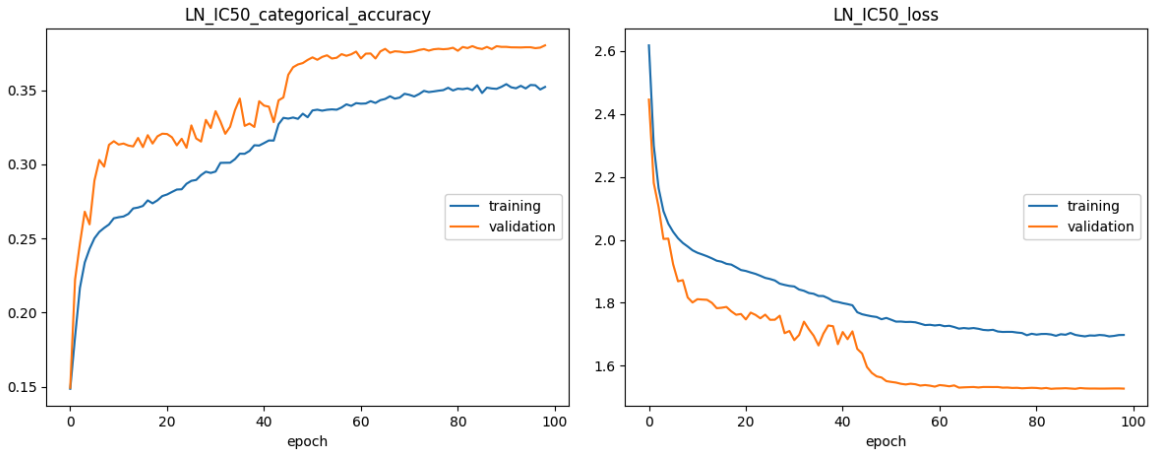

Figure 4.23: Model training progress by changing the importance of outputs.

Validation Metric	Best Value
LN_IC50_Categorical_Accuracy	0.380
LN_IC50_Loss	1.527

Table 4.15: Best validation results for LN_IC_{50} metrics after applying output weighting.

Everything seems to indicate that adjusting the weights of the outputs provides an increase in the model's performance. Likewise, during this training, it can be observed how, once again, the categorical cross-entropy error function has provided the necessary dynamism to improve the quality of the predictions by reducing the learning rate.

Refinement of the loss function

The experimental results suggest that performance can be improved by carefully designing the training process, specifically through a combination of output weighting and an appropriately chosen loss function.

While selecting the optimal combination of weights often involves empirical tuning, the choice of loss function is more critical, as it must be tailored to both the characteristics of the data and the nature of the problem. It is important to recall that the objective is not merely to predict the exact number of doses, but to minimize the severity of misclassifications. For example, mispredicting by one dose is less significant than mispredicting by five. Consequently, the loss function must reflect this asymmetry in error severity.

Additionally, the loss must support effective learning; otherwise, there is a risk of stagnation during training. To address these requirements, a custom loss function was developed for the final version of the classification model, as shown in Listing 18. This function penalizes predictions progressively based on their distance from the correct class, encouraging the model to make closer, even if not

exact, predictions. As a result, the number of highly incorrect classifications is significantly reduced.

As a result of all these modifications, the model significantly increases performance and reduces the difference between training and validation, all of which can be seen in Figure 4.24. This translates into more accurate and reliable predictions outside the training set.

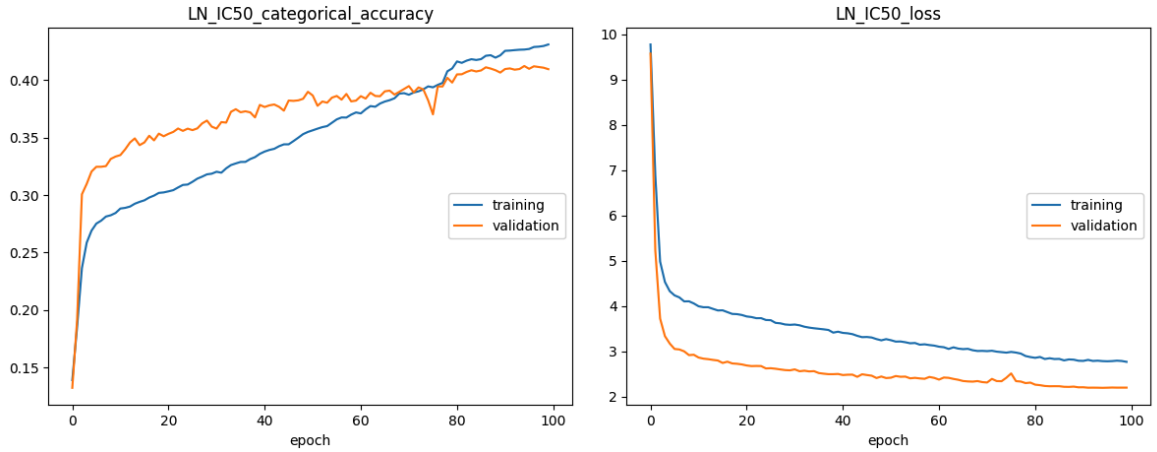

Figure 4.24: Progress of model training applying a customised error function, which takes into account both accuracy and the difference between the actual and predicted values.

Validation Metric	**Best Value**
LN_IC50_Categorical_Accuracy	0.412
LN_IC50_Loss	2.195

Table 4.16: Best validation results for LN_IC50 metrics using the custom loss function.

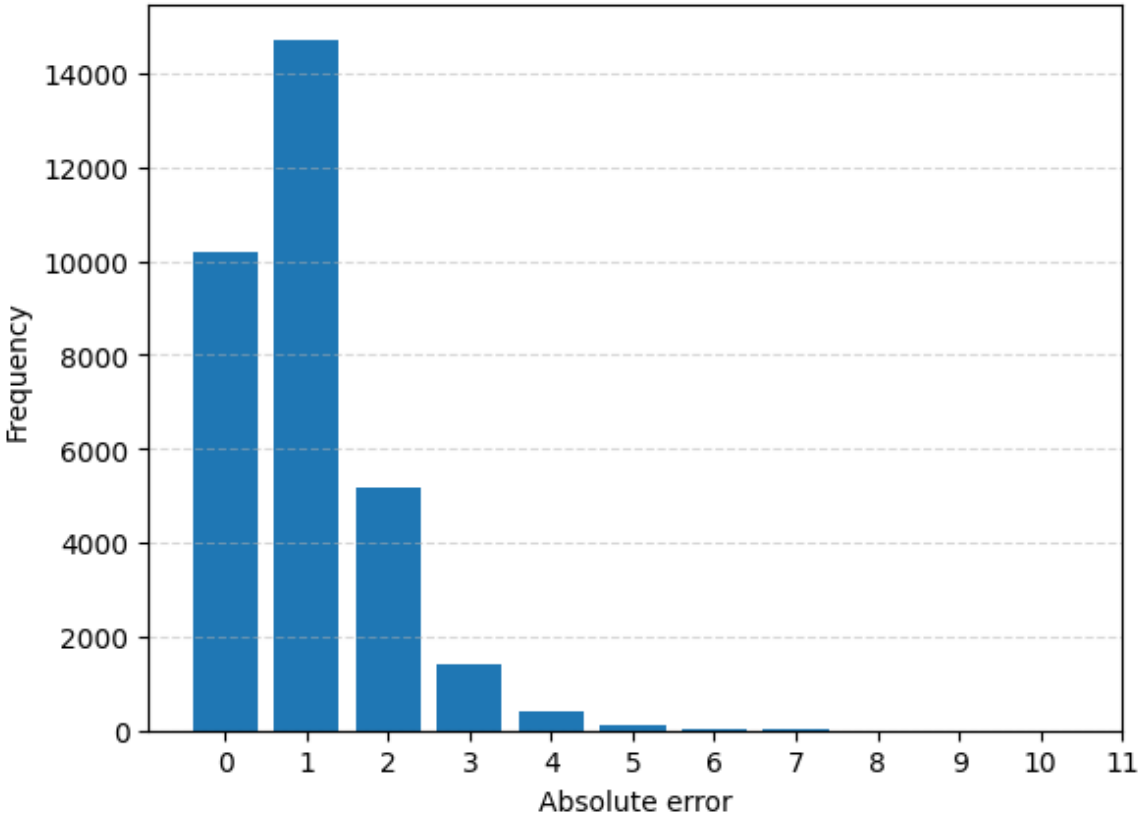

Figure 4.25: Bar chart representation of the absolute error committed in the test set. It allows us to verify that the model predicts with a low degree of error.

Test Metric	Value
MSE	1.879
RMSE	1.370
R^2	0.8693

Table 4.17: Regression performance metrics on the test set for LN_IC_50.

However, there is one aspect that cannot be assessed from the graph in Figure 4.24. In this iteration, an error function has been defined that should prevent large discrepancies between reality and prediction. This feature is corroborated in Figure 4.25, which illustrates a histogram in which the highest percentage of errors are between 0 and 2 doses. Therefore, the model effectively meets expectations.

Reaching this point in the research represents a milestone in the study, as the model is capable of deciding how many doses to administer, with a single-dose error in most cases. This allows for better management of prescriptions, thus improving patients' quality of life.

4.2.2 Testing XGBoost as a classifier

In this section, the method applied is XGBoost in a multi-class classification setting, using teh discretised variable LN_IC_{50} as in previous experiments. The aim is to

evaluate its predictive power and compare it with the results obtained using deep learning approaches.

Beyond accuracy, it is particularly interesting how XGBoost handles the ordinal nature of the target variable, where misclassifying a class by 1 is much less serious than misclassifying it by 5 or more. To achieve this, evaluations will be carried out using standard metrics, such as accuracy, but also some that take into account the distance between the prediction and the actual classes, including MAE and RMSE.

As the dynamic applied throughout the research, the model will be refined as the reading progresses. First, training will be carried out where only accuracy will be taken into account, and then training will also evaluate the difference between prediction and expected class. This initial process will allow us to compare and verify that this approach is the most appropriate.

Focusing on classification

To conduct the research using xgboost and focusing solely on the accuracy of the result, the most appropriate objective function is softmax. This setting enable multi-class classification, by directly modeling the probability distribution over all classes. Otherwise, the architecture would have to be modelled as one vs rest system, which would require a binary classifier for each class. This would increase the time and effort required.

For evaluating model performance, the chosen loss function is categorical cross-entropy, referred to in the XGBoost Python library as Multi-Class Log Loss (mlogloss). This metric is particularly appropriate because it not only assesses classification correctness but also takes into account the confidence of the predicted probabilities, in this way, a more nuanced assessment is provided than simple accuracy.

The flaw in this approach is that it does not take into account the severity of major errors. In fact, Figure 4.26 shows how the results obtained using this model falls significantly short of those produced by the neural network. This is further supported by the metrics shown in Table 4.18, which reveal a substantial increase in error.

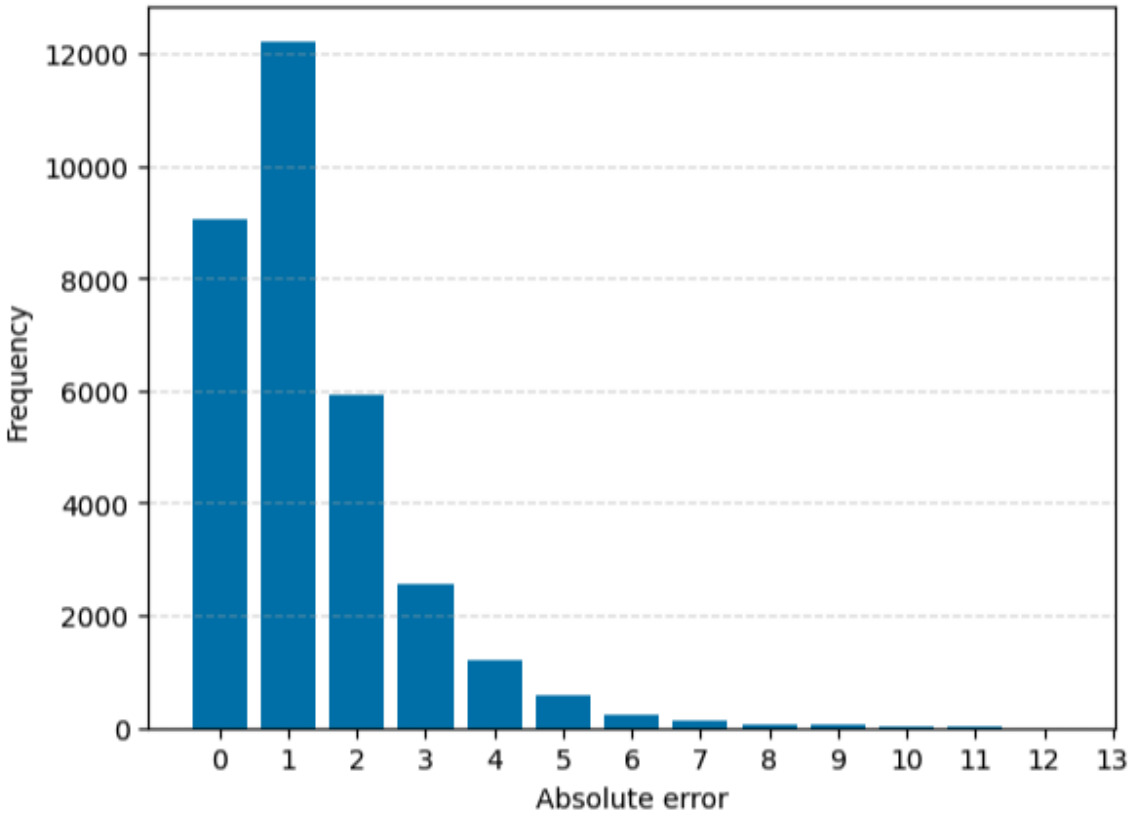

Figure 4.26: Representation of the absolute error committed with the test set using XGBoost and softmax. A substantial increase in error is observed, with the largest error committed being 15.

Test Metric	Value
MSE	4.092
RMSE	2.022
R^2	-18.148

Table 4.18: Performance metrics on the test set for LN_IC_{50} using XGBoost classifier predictions interpreted as numerical outputs.

Prioritizing distance to the right class

Applying a function that shows the importance of being close to the correct class should improve the quality of the model. In the case of XGBoost, the function used is squarederror, which implements the mean square error (MSE). In addition, it is necessary to modify the evaluation metric, in this case the root mean square error (RMSE), to facilitate the interpretation of progress.

After a series of training sessions, a result similar to that of neural networks was obtained, without reaching the same quality, but improving on the previous version. This can be seen in Figure 4.27, which shows how the proportion changes, increasing the number of cases in which only one error is made, or even none at

all. It also reduces all the others, as reflected in the metrics in Table 4.19, where the maximun error is 12.

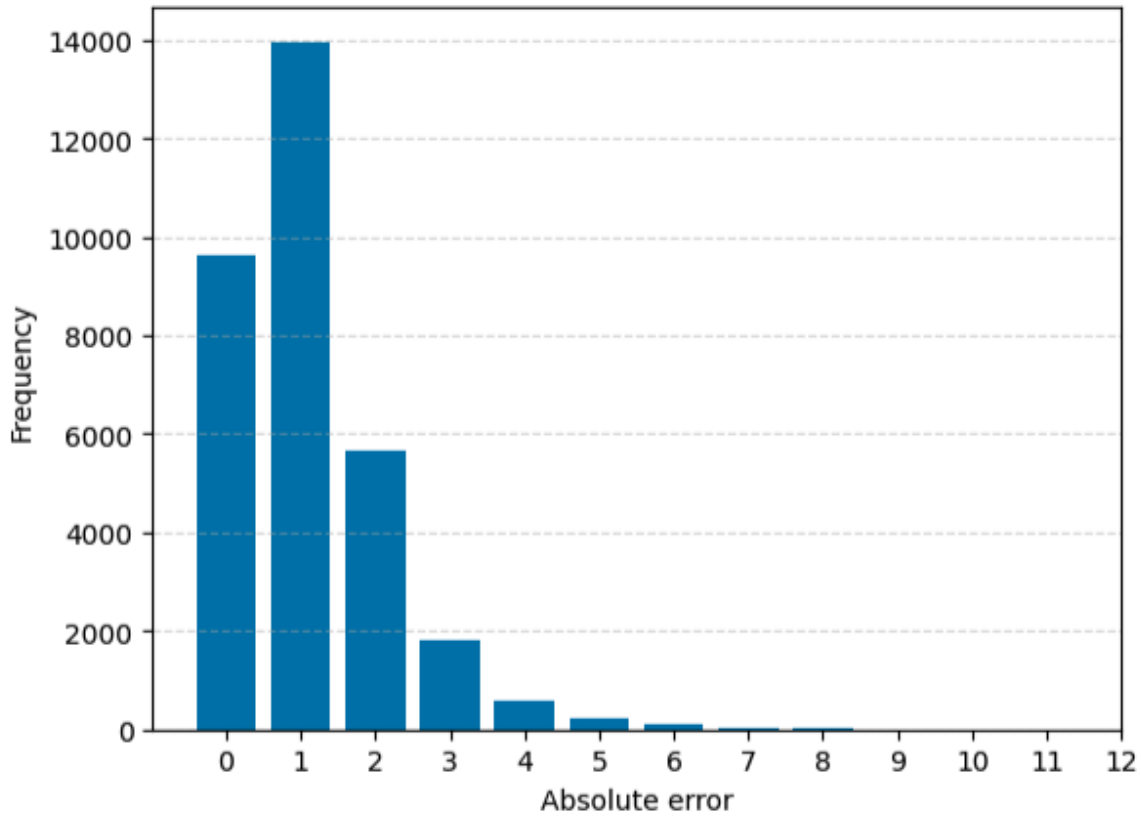

Figure 4.27: Representation of the absolute error committed with the test set using XGboost and MSE. A substantial increase in error is observed, with the largest error committed being 15.

Test Metric	Value
MSE	2.381
RMSE	1.543
R^2	0.834

Table 4.19: Performance metrics on the test set using XGBoost with `reg:squarederror`.

By applying the same distance-aware approach used in neural networks to the XGBoost model, the result obtained is very similar. Two key aspects of this result must be taken into account:

- Accuracy of predictions: While the overall performance is similar in terms of trend, the quality of the model is slightly lower. the model exhibits a higher number of instances where the error is greater than 3, a threshold that should not be exceeded too often, as such deviations could negatively impact clinical outcomes.

- Training time and cost: One of the major advantages of using XGBoost lies in its training efficiency. The model can be trained significantly faster and with much lower computational overhead compared to neural networks, which often require extensive resources and time to achieve optimal performance.

4.3. Shap: The reasons must be known.

This chapter focuses on applying all the knowledge acquired during the research to understand what the best trained models have learned. The objective is to extract the linear or non-linear relationships that the model has learned from the data.

This approach allows, on the one hand, to validate the correct learning of the model. For example, it could be the case that to infer the response, the model looks at patterns that have nothing to do with the efficiency of a drug, as would be the case with a model that groups several cases because it identifies that they have the same doctor. This knowledge, although entirely valid, implies that the learning has not acquired the quality it should, as it should predict based on a patient's data, not on who is treating them. On the other hand, knowing the features on which a model is based and to what extent can represent a major advance in cancer research, open up new avenues of research or demonstrate previously unknown theories.

Explainability aims to extract as much knowledge as possible from model learning, giving regressors preference over classifiers. This is because, in order to perform any type of classification in the domain of a regression problem, it is necessary to discretise the target variables, thereby losing information that could be useful. For this reason, even if regression and classification models have been trained, the choice of which model to use is limited to regressors.

Throughout this research, regressors based on two different modeling approaches were trained. First, models based on neural networks were trained, which are particularly capable of finding non-linear and complex relationships. Then, research was conducted on the feasibility of using tree-based models, hough generally less performant than neural networks, offer faster training times and lower computational cost. Thus, there are two candidates, a network-based model and a tree-based model. Given that the aim of this section is to learn more about the problem, he one that provides the best results must be selected t, which is the network-based model. This can be seen in Tables **??**, which show the performance of the best models of both architectures in the test set.

To apply explainability, the technique used is Shap values. It is very important to remember that these values are calculated based on a group of predictions, so choosing an appropriate set is crucial to obtaining meaningful results. For example, if the selected set only contains negative predictions, the analysis will not apply to all the data, which could lead to future misconceptions. Fortunately, the library used has a set of utilities, including shap.kmeans, which allows you to locate the clusters in the data set and select the most representative values based on their centroids.

Once the data on which to calculate the Shap values has been selected, it is necessary to choose the appropriate kernel based on the type of model. In the case

of neural networks, there is more than one, due to a compatibility issue between versions, "KernelExplainer" has been selected. The next step is to calculate the values. Given that Shap values are cumulative, it is useful to save them, as knowing the points selected to calculate them and the results they provide allows for further calculations at a later time or with another hardware.

Although there are many types of graphs for representing Shap values, the one used in this study is a "summary plot". This representation provides a very intuitive visualisation of what is happening within the model, slightly dispelling that black box perception.

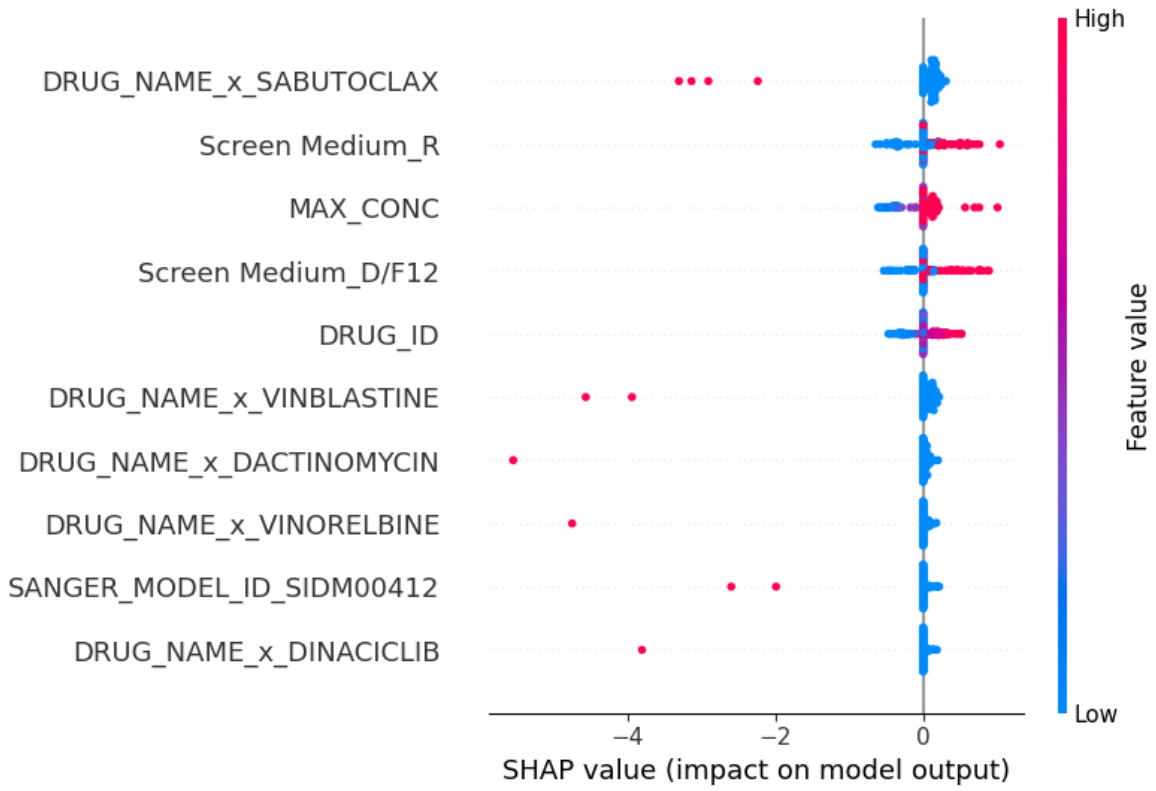

Figure 4.28: Shap "summary plot" graph. This one shows how the model behaves based on the values of its most representative variables.

The representation of Shap values, illustrated in Figure 4.28, provides information about the model's trend in output values, based on its most representative variables. By way of explanation:

- Each row represents the most influential variables for a given prediction, ordered from highest to lowest based on their contribution to the model's output.
- The X-axis represents the prediction value, which is higher the further to the right the point is located.
- Each point represents an instance of the dataset, with the position on the horizontal axis representing the impact on the variable. Cases closer to red

indicate that the variable has a high value within the dataset. If it is blue, it means that the value is low.

To interpret the image, it is important to note that all variables beginning with "DRUG_NAME", "Screen Medium", or "SANER" have been one-hot encoded. This encoding explains the pattern of numerous blue dots and only a few red ones, indicating that the model primarily values the absence of these features. In contrast, the variable "DRUG_ID" retains its original numerical values. Observing the color distribution and SHAP values for this feature suggests that the identifiers may have been stored in a specific order in the dataset, as higher values of "DRUG_ID" are associated with higher model predictions, and lower values with lower predictions

Regarding the listed drugs, their presence appears to be associated with lower predictions for the LN_IC_{50} variable, which is considered a favorable outcome for patients. Notably, one of the ten most influential features is linked to the blood sample type. Its presence consistently contributes to lower LN_IC_{50} values, reinforcing the hypothesis that genetic or biological factors, such as the type of sample, may significantly influence a patient's response to cancer treatments. This finding supports the broader theory that the human genome plays a crucial role in drug sensitivity

5. Conclusions

Appendix

```
obsolete = []
for col in dataset.columns:
    if col.upper() != ("cosmic_id".upper()):
        content = dataset[col].unique()
        if len(content) < 2:
            obsolete.append(col)
        suffix = "..." if len(content) > 10 else ""
        print(f"\033[1m* {col}\033[0m -> {content[:10]} {suffix}")
print(f"The obsolete columns are: {obsolete}")

dataset = dataset.drop(columns=obsolete)
```

Code extract 1: Find obsolete variables and determine the size of each variable's domain.

```
def check_id_name_consistency(df, id_col="DRUG_ID", name_col="DRUG_NAME_x
    "):
    names = df[name_col].unique()
    ids_values = df[id_col].unique()
    names_inconsistences = []
    ids_inconsistences = []
    for i in names:
        ids = df[df[name_col] == i][id_col].unique()
        if len(ids) > 1:
            print(f"names' inconsistencies: {i}")
            names_inconsistences.append(i)
    for i in ids_values:
        names = df[df[id_col] == i][name_col].unique()
        if len(names) > 1:
            print(f"ids' inconsistencies: {i}")
            ids_inconsistences.append(i)
    if len(names_inconsistences) > 0:
        print(f"There are inconsistencies in names: {names_inconsistences
    }")
    if len(ids_inconsistences) > 0:
        print(f"There are inconsistencies in ids: {ids_inconsistences}")
    return {"names": names_inconsistences, "ids": ids_inconsistences}

inconsistence_analysis = check_id_name_consistency(dataset)
```

Code extract 2: Check for matching values between the drug ID and drug name variables.

```python
from scipy.stats import kstest, anderson
stat, p = kstest(y["LN_IC50"], 'norm', args=(y["LN_IC50"].mean(), y["LN_IC50"].std()))
print(stat, p)

# >> Output: 0.07784598710281665 0.0

result = anderson(y["LN_IC50"], dist='norm')
print(result)

# >> Output
# >> AndersonResult(statistic=3269.8765859772393, critical_values=array([0.576, 0.656, 0.787, 0.918, 1.092]), significance_level=array([15. , 10. , 5. , 2.5, 1. ]), fit_result= params: FitParams(loc=2.7822776442090498, scale=2.8346892731645923)
# >>  success: True
# >>  message: ''anderson' successfully fit the distribution to the data.')
# >>
```

Code extract 3: Checking whether LN_IC_{50} follows a normal distribution.

```python
for name in inconsistence_analysis['names']:
    drug_ids_found = dataset[dataset["DRUG_NAME_x"].str.upper() == name.upper()]["DRUG_ID"].unique()
    companies_ids_found = dataset[dataset["DRUG_NAME_x"].str.upper() == name.upper()]["COMPANY_ID"].unique()
    print(f"-- For the name: {name:<12} ---> we detect this ids: {drug_ids_found} and this drug has as its supplier: {companies_ids_found} ")
print(f"Into the company column there are {dataset['COMPANY_ID'].isna().sum()} null values")
```

Code extract 4: Code intended to check the different suppliers assigned to each drug name.

```python
yes_no_map = {"N": -1, np.nan: 0, "Y": 1}
adhenrence_map = {"Adherent": 2, "Semi-Adherent": 1, "Suspension": 0}
screen_medium = {'R': 0, 'D/F12': 1}

class MapTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, column, mapping):
        self.column = column
        self.mapping = mapping

```

```
    def fit(self, X, y=None):
        return self

    def transform(self, X):
        X_ = X.copy()
        X_[self.column] = X_[self.column].map(self.mapping).astype(np.int64)
        return X_

column_pipeline = ColumnTransformer(
    transformers=[
        ('map_methylation', MapTransformer(column="Methylation", mapping=yes_no_map), ["Methylation"]),
        ('map_response', MapTransformer(column='Drug Response', mapping=yes_no_map), ['Drug Response']),
        ('map_cna', MapTransformer(column="Copy Number Alterations (CNA)", mapping=yes_no_map), ['Copy Number Alterations (CNA)']),
        ('map_expression', MapTransformer(column="Gene Expression", mapping=yes_no_map), ['Gene Expression']),
        ("map_adherence", MapTransformer(column="Growth Properties", mapping=adhenrence_map), ["Growth Properties"]),
        ("map_screen_medium", MapTransformer(column="Screen Medium", mapping=screen_medium), ["Screen Medium"])
    ],
    remainder="passthrough"
)

transformed_cols = ["Methylation", "Drug Response", "Copy Number Alterations (CNA)", "Gene Expression", "Growth Properties", "Screen Medium"]
remainder_cols = [col for col in dataset.columns if col not in transformed_cols]
column_names = transformed_cols + remainder_cols

dataset_copy = dataset.copy()
dataset_data_preprocessed = column_pipeline.fit_transform(dataset)
dataset_preprocessed = pd.DataFrame(dataset_data_preprocessed, columns=column_names)

for col in dataset_preprocessed.columns:
    if col in transformed_cols:
        dataset_preprocessed[col] = dataset_preprocessed[col].astype(int)
    else:
        dataset_preprocessed[col] = dataset[col].copy()
```

Code extract 5: Application of ColumnTransformer to map the data.

```
columns_to_impute = ["TCGA_DESC", "PUTATIVE_TARGET", "Cancer Type\n(matching TCGA label)", "Microsatellite \ninstability Status (MSI)", "TARGET"]
encoder_nan = OrdinalEncoder()
encoder_all = OrdinalEncoder()
dataset_encoded[columns_to_impute] = encoder_nan.fit_transform(dataset[columns_to_impute])
columns_not_encode = [i for i in categorical_variables if i not in columns_to_impute]
dataset_encoded[columns_not_encode] = encoder_all.fit_transform(dataset[columns_not_encode].astype(str))
```

Code extract 6: Encoding data in numerical format using Ordinal Encoder.

```
imputer = KNNImputer(n_neighbors=8) # This number was determined by checking the results obtained from various combinations.
dataset_imputed = imputer.fit_transform(dataset_encoded)
```

Code extract 7: Imputation of values using KnnImputer.

```
def generate_model(n_columns_x, n_columns_y):
    model = models.Sequential()
    model.add(layers.Input(shape=(n_columns_x,)))
    model.add(layers.Dense(32, activation='relu'))
    model.add(layers.Dropout(0.2))
    model.add(layers.Dense(32, activation='relu'))
    model.add(layers.Dropout(0.2))
    model.add(layers.Dense(n_columns_y))
    return model
```

Code extract 8: Function to generate a neural network. This code generates a very simple dense network, but we only want it to know which loss function to choose.

```
def train_sequential_model(X_train, y_train, X_val, y_val, X_test, y_test, model: models.Sequential, optimizer="adam", error="mse", metric=keras.metrics.RootMeanSquaredError()):
    model.compile(optimizer=optimizer, loss=error, metrics=[metric])
    model.summary()
    early_stopping = EarlyStopping(monitor='val_loss', patience=10, mode='min',restore_best_weights=True)
    model_checkpoint = ModelCheckpoint("best_model_v1.keras", monitor="val_loss", mode='min')
    reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=7, min_lr=0.000001)
    history=model.fit(X_train, y_train,
```

```
                    epochs=100,
                    batch_size=32,
                    validation_data=(X_val, y_val),
                    callbacks=[early_stopping, model_checkpoint,
    PlotLossesKerasTF(), reduce_lr],
                    verbose=2)
    show_evaluation(model, X_test, y_test)
    return model
```

Code extract 9: Function to compile, show the losses during learning, do a evaluation with test set and return the model trained.

```
def represent_scatter_matrix(ax, y_test_df, predition_values,
    target_variable, color_scatter="black"):
    if type(color_scatter) is str:
        ax.scatter(np.array(y_test_df[target_variable]), predition_values
    , s=2, alpha=0.5, color=color_scatter)
    else:
        ax.scatter(np.array(y_test_df[target_variable]), predition_values
    , s=2, alpha=0.5, c=color_scatter, cmap='tab10')
        legend_labels = np.unique(color_scatter)
        cmap = plt.get_cmap('tab10')
        legend_labels = np.unique(color_scatter)
        for label in legend_labels:
            color_index = label if label >= 0 else 9  # Handle -1 as "
    noise" with last color
            color = cmap.colors[color_index % len(cmap.colors)]
            ax.scatter([], [], color=color, label=f"Cluster {label}")
        ax.legend()
    ax.grid(True)

    min_val = min(np.min(y_test_df[target_variable]), np.min(
    predition_values))
    max_val = max(np.max(y_test_df[target_variable]), np.max(
    predition_values))
    ax.plot([min_val, max_val], [min_val, max_val], color='red',
    linestyle='--', linewidth=1)
    ax.set_ylabel('Pronostics')
    ax.set_xlabel('Real')
    ax.set_title(target_variable)

def show_evaluation(model, X_test, y_test):
    pred_test = model.predict(X_test)
    print(f"Evaluation value with test values: {model.evaluate(X_test,
    y_test)}")
    print(f"MSE: {mean_squared_error(y_test, pred_test)}")
    print(f"RMSE: {np.sqrt(mean_squared_error(y_test, pred_test))}")
```

```
    print(f"MAE: {mean_absolute_error(y_test, pred_test)}")
    print(f"R2: {r2_score(y_test, pred_test)}")
    rows = int(np.ceil(len(y_test.columns)/2))
    fig, axes = plt.subplots(rows, 2, figsize=(8 * rows, 8 * rows))
    axes = axes.flatten()
    for i, target in enumerate(y_test.columns):
        represent_scatter_matrix(axes[i], y_test_df=y_test,
    predition_values=pred_test[:, i], target_variable=target)
    fig.tight_layout()
    plt.show()
```

Code extract 10: Functions to show some metrics about the model. Also, we display a plot where you can see how much near is the predicted values from the real ones.

```
inputs = keras.Input((X_train.shape[1],))

x = layers.Dense(2048, activation="relu")(inputs)
x = layers.Dropout(0.3)(x)
x = layers.Reshape((16, 128))(x)
x = layers.Conv1D(256, kernel_size=(3), padding="same",
    kernel_regularizer=regularizers.l2(1e-4))(x)
x = layers.BatchNormalization()(x)
x = layers.ReLU()(x)
x = layers.SpatialDropout1D(0.3)(x)
x = layers.AveragePooling1D(pool_size=2)(x)
x = layers.BatchNormalization()(x)
x = layers.Conv1D(256, kernel_size=(3), padding="same",
    kernel_regularizer=regularizers.l2(1e-4))(x)
x = layers.BatchNormalization()(x)
x = layers.ReLU()(x)
x = layers.SpatialDropout1D(0.3)(x)
x_main = layers.Conv1D(256, kernel_size=(3), padding="same",
    kernel_regularizer=regularizers.l2(1e-4))(x)
x_main = layers.ReLU()(x_main)
x_main = layers.SpatialDropout1D(0.3)(x_main)
x_main = layers.Conv1D(256, kernel_size=(3), padding="same",
    kernel_regularizer=regularizers.l2(1e-4))(x_main)
x_main = layers.ReLU()(x_main)
x_main = layers.SpatialDropout1D(0.3)(x_main)
conc = layers.Add()([x, x_main])
x_final = layers.MaxPooling1D()(conc)
x_final = layers.GlobalMaxPooling1D()(x_final)
outputs = layers.Dense(y_test.shape[1])(x_final)

model_conv1d_v8 = keras.Model(inputs=inputs, outputs=outputs, name="
    convmodelv8")
model_conv1d_v8.compile(optimizer="adam", loss=keras.losses.LogCosh(),
```

```
    metrics=[keras.metrics.RootMeanSquaredError()])
model_conv1d_v8.summary()
early_stopping = EarlyStopping(monitor='val_loss', patience=10, mode='min
    ',restore_best_weights=True)
model_checkpoint = ModelCheckpoint("model_conv_8.keras", monitor="
    val_loss", mode='min')
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5,
     min_lr=0.000001)
model_conv1d_v8.fit(X_train, y_train,
                    epochs=100,
                    batch_size=32,
                    validation_data=(X_val, y_val),
                    callbacks=[early_stopping, model_checkpoint,
    PlotLossesKerasTF(), reduce_lr],
                    verbose=2)
model_conv1d_v8.evaluate(X_test, y_test)
```

Code extract 11: Definition of Soft-ordering with 1D convolutional architecture using tensorflow.

```
inputs = keras.Input((X_train.shape[1],))

x = layers.Dense(1024, activation="silu", kernel_regularizer=regularizers
    .l2(1e-4))(inputs)
x = layers.Dropout(0.3)(x)
x = layers.Reshape((16, 64))(x)
x = layers.Conv1D(256, kernel_size=(3), padding="same",
    kernel_regularizer=regularizers.l2(1e-4))(x)
x = layers.BatchNormalization()(x)
x = keras.activations.silu(x)
x = layers.SpatialDropout1D(0.3)(x)
x = layers.AveragePooling1D(pool_size=2)(x)
x = layers.BatchNormalization()(x)
x = layers.Conv1D(256, kernel_size=(3), padding="same",
    kernel_regularizer=regularizers.l2(1e-4))(x)
x = layers.BatchNormalization()(x)
x = keras.activations.silu(x)
x = layers.SpatialDropout1D(0.1)(x)
x_main = layers.Conv1D(256, kernel_size=(3), padding="same",
    kernel_regularizer=regularizers.l2(1e-4))(x)
x_main = keras.activations.silu(x_main)
x_main = layers.SpatialDropout1D(0.3)(x_main)
x_main = layers.Conv1D(256, kernel_size=(3), padding="same",
    kernel_regularizer=regularizers.l2(1e-4))(x_main)
x_main = keras.activations.silu(x_main)
x_main = layers.SpatialDropout1D(0.4)(x_main)
conc = layers.Add()([x, x_main])
```

```
x_final = layers.MaxPooling1D()(conc)
x_final = layers.Flatten()(x_final)
x_final = layers.Dropout(0.3)(x_final)
outputs = layers.Dense(y_test.shape[1])(x_final)

model_conv1d_v21 = keras.Model(inputs=inputs, outputs=outputs, name="
    convmodelv21")

model_conv1d_v21.compile(optimizer="adam", loss=keras.losses.LogCosh(),
    metrics=[keras.metrics.RootMeanSquaredError()])
model_conv1d_v21.summary()

early_stopping = EarlyStopping(monitor='val_loss', patience=10, mode='min
    ',restore_best_weights=True)
model_checkpoint = ModelCheckpoint("model_conv_21.keras", monitor="
    val_loss", mode='min')
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=7,
     min_lr=0.000001)

model_conv1d_v21.fit(X_train, y_train,
                    epochs=100,
                    batch_size=32,
                    validation_data=(X_val, y_val),
                    callbacks=[early_stopping, model_checkpoint,
    PlotLossesKerasTF(), reduce_lr],
                    verbose=2)
show_evaluation(model_conv1d_v21, X_test, y_test )
```

Code extract 12: Definition of Soft-ordering with 1D convolutional architecture using SiLU as activation function.

```
if os.path.exists("tsne_3d_result.npy"):
    X_3d_tsne = np.load("tsne_3d_result.npy")
    X_2d_tsne = np.load("tsne_2d_result.npy")
else:
    tsne_3d = TSNE(n_components=3, random_state=10, perplexity=40.0, init
    ='random', learning_rate='auto')
    X_3d_tsne = tsne_3d.fit_transform(X_data)
    np.save("tsne_3d_result.npy", X_3d_tsne)

    tsne_2d = TSNE(n_components=2, random_state=10, perplexity=40.0, init
    ='random', learning_rate='auto')
    X_2d_tsne = tsne_2d.fit_transform(X_data)
    np.save("tsne_2d_result.npy", X_2d_tsne)

fig = plt.figure(figsize=(12,7))
ax_1 = fig.add_subplot(121)
```

```
ax_1.set_title("2D representation (t-TSNE)")
sc = ax_1.scatter(X_2d_tsne[:, 0], X_2d_tsne[:, 1], c=y_data["LN_IC50"],
    cmap='viridis', alpha=0.8, s=s)

ax_2 = fig.add_subplot(122, projection="3d")
ax_2.set_title("3D representation (t-TSNE)")
sc = ax_2.scatter(X_3d_tsne[:, 0], X_3d_tsne[:, 1], X_3d_tsne[:, 2], c=
    y_data["LN_IC50"], cmap='viridis', alpha=0.8, s=s)
```

Code extract 13: Calculation of dimensionality reduction using t-SNE for data representation while respecting local distances.

```
if os.path.exists("tsne_3d_result.npy"):
    X_3d_tsne = np.load("tsne_3d_result.npy")
    X_2d_tsne = np.load("tsne_2d_result.npy")
else:
    tsne_3d = TSNE(n_components=3, random_state=10, perplexity=40.0, init
    ='random', learning_rate='auto')
    X_3d_tsne = tsne_3d.fit_transform(X_data)
    np.save("tsne_3d_result.npy", X_3d_tsne)

    tsne_2d = TSNE(n_components=2, random_state=10, perplexity=40.0, init
    ='random', learning_rate='auto')
    X_2d_tsne = tsne_2d.fit_transform(X_data)
    np.save("tsne_2d_result.npy", X_2d_tsne)

s=0.8
fig = plt.figure(figsize=(12,7))
ax_1 = fig.add_subplot(121)
ax_1.set_title("2D representation (t-TSNE)")
sc = ax_1.scatter(X_2d_tsne[:, 0], X_2d_tsne[:, 1], c=y_data["LN_IC50"],
    cmap='viridis', alpha=0.8, s=s)

ax_2 = fig.add_subplot(122, projection="3d")
ax_2.set_title("3D representation (t-TSNE)")
sc = ax_2.scatter(X_3d_tsne[:, 0], X_3d_tsne[:, 1], X_3d_tsne[:, 2], c=
    y_data["LN_IC50"], cmap='viridis', alpha=0.8, s=s)
```

Code extract 14: Calculation of dimensionality reduction using UMAP for data representation while respecting global and local distances.

```
input_shape = (X_train.shape[1],)
inputs = Input(shape=input_shape)

x = layers.Dense(2048, activation="silu")(inputs)
x = layers.Dropout(0.5)(x)
```

```
x = layers.Reshape((16, 128))(x)
x = layers.Conv1D(16, kernel_size=(5), padding="same", kernel_regularizer
    =regularizers.l2(1e-4))(x)
x = layers.BatchNormalization()(x)
x = keras.activations.silu(x)
x = layers.SpatialDropout1D(0.5)(x)
x = layers.AveragePooling1D(pool_size=2)(x)
x = layers.BatchNormalization()(x)
x = layers.Conv1D(16, kernel_size=(3), padding="same", kernel_regularizer
    =regularizers.l2(1e-4))(x)
x = layers.BatchNormalization()(x)
x = keras.activations.silu(x)
x = layers.SpatialDropout1D(0.6)(x)
x_main = layers.Conv1D(16, kernel_size=(3), padding="same",
    kernel_regularizer=regularizers.l2(1e-4))(x)
x_main = keras.activations.silu(x_main)
x_main = layers.SpatialDropout1D(0.5)(x_main)
x_main = layers.Conv1D(16, kernel_size=(3), padding="same",
    kernel_regularizer=regularizers.l2(1e-4))(x_main)
x_main = keras.activations.silu(x_main)
x_main = layers.SpatialDropout1D(0.6)(x_main)
conc = layers.Multiply()([x, x_main])
x_final = layers.MaxPooling1D(pool_size = 4, strides = 2)(conc)
x_final = layers.Flatten()(x_final)
x_final = layers.Dropout(0.4)(x_final)

out_ln_ic50 = Dense(kbins, activation='softmax', name='LN_IC50')(x_final)
out_auc = Dense(kbins, activation='softmax', name='AUC')(x_final)
out_dr = Dense(kbins, activation='softmax', name='Drug_Response')(x_final
    )
out_rmse = Dense(kbins, activation='softmax', name='RMSE')(x_final)
out_zscore = Dense(kbins, activation='softmax', name='Z_SCORE')(x_final)

model = Model(inputs=inputs, outputs=[out_ln_ic50, out_auc, out_dr,
    out_rmse, out_zscore])

model.compile(
    optimizer="adam",
    loss="categorical_crossentropy",
    metrics=["categorical_accuracy","categorical_accuracy","
    categorical_accuracy","categorical_accuracy","categorical_accuracy"]
)

early_stopping = tf.keras.callbacks.EarlyStopping(monitor='
    val_LN_IC50_loss', mode="min", patience=10,restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_LN_IC50_loss', factor=0.2,
    patience=7, min_lr=0.000001)
```

```
model_checkpoint = ModelCheckpoint("model_conv_softmax_1.keras", monitor=
    "val_LN_IC50_loss", mode='min')

history = model.fit(
    X_train,
    y_train_dict_onehot,
    validation_data=(X_val, y_val_dict_onehot),
    epochs=100,
    batch_size=64,
    callbacks=[early_stopping, PlotLossesKerasTF(), reduce_lr,
    model_checkpoint]
)
```

Code extract 15: First adaptation of the regressor network in the process of obtaining a classifier.

```
input_shape = (X_train.shape[1],)
inputs = Input(shape=input_shape)

x = layers.Dense(2048, activation="silu")(inputs)
x = layers.Dropout(0.5)(x)
x = layers.Reshape((16, 128))(x)
x = layers.Conv1D(16, kernel_size=(5), padding="same", kernel_regularizer
    =regularizers.l2(1e-4))(x)
x = layers.BatchNormalization()(x)
x = keras.activations.silu(x)
x = layers.SpatialDropout1D(0.5)(x)
x = layers.AveragePooling1D(pool_size=2)(x)
x = layers.BatchNormalization()(x)
x = layers.Conv1D(16, kernel_size=(3), padding="same", kernel_regularizer
    =regularizers.l2(1e-4))(x)
x = layers.BatchNormalization()(x)
x = keras.activations.silu(x)
x = layers.SpatialDropout1D(0.6)(x)
x_main = layers.Conv1D(16, kernel_size=(3), padding="same",
    kernel_regularizer=regularizers.l2(1e-4))(x)
x_main = keras.activations.silu(x_main)
x_main = layers.SpatialDropout1D(0.5)(x_main)
x_main = layers.Conv1D(16, kernel_size=(3), padding="same",
    kernel_regularizer=regularizers.l2(1e-4))(x_main)
x_main = keras.activations.silu(x_main)
x_main = layers.SpatialDropout1D(0.6)(x_main)
conc = layers.Multiply()([x, x_main])
x_final = layers.MaxPooling1D(pool_size = 4, strides = 2)(conc)
x_final = layers.Flatten()(x_final)
x_final = layers.Dropout(0.4)(x_final)

```

```
out_ln_ic50 = Dense(kbins, activation='softmax', name='LN_IC50')(x_final)
out_auc = Dense(kbins, activation='softmax', name='AUC')(x_final)
out_dr = Dense(kbins, activation='softmax', name='Drug_Response')(x_final
    )
out_rmse = Dense(kbins, activation='softmax', name='RMSE')(x_final)
out_zscore = Dense(kbins, activation='softmax', name='Z_SCORE')(x_final)

model = Model(inputs=inputs, outputs=[out_ln_ic50, out_auc, out_dr,
    out_rmse, out_zscore])

model.compile(
    optimizer="adam",
    loss="categorical_focal_crossentropy",
    metrics=["categorical_accuracy","categorical_accuracy","
    categorical_accuracy","categorical_accuracy","categorical_accuracy"]
)

early_stopping = tf.keras.callbacks.EarlyStopping(monitor='
    val_LN_IC50_loss', mode="min", patience=14,restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_LN_IC50_loss', factor=0.2,
    patience=7, min_lr=0.000001)
model_checkpoint = ModelCheckpoint("model_conv_softmax_2.keras", monitor=
    "val_LN_IC50_loss", mode='min')

history = model.fit(
    X_train,
    y_train_dict_onehot,
    validation_data=(X_val, y_val_dict_onehot),
    epochs=100,
    batch_size=64,
    callbacks=[early_stopping, PlotLossesKerasTF(), reduce_lr,
    model_checkpoint]
)
```

Code extract 16: Testing a different function to check for changes in model performance.

```
input_shape = (X_train.shape[1],)
inputs = Input(shape=input_shape)

x = layers.Dense(2048, activation="silu")(inputs)
x = layers.Dropout(0.5)(x)
x = layers.Reshape((16, 128))(x)
x = layers.Conv1D(16, kernel_size=(5), padding="same", kernel_regularizer
    =regularizers.l2(1e-4))(x)
x = layers.BatchNormalization()(x)
x = keras.activations.silu(x)
```

```
x = layers.SpatialDropout1D(0.5)(x)
x = layers.AveragePooling1D(pool_size=2)(x)
x = layers.BatchNormalization()(x)
x = layers.Conv1D(16, kernel_size=(3), padding="same", kernel_regularizer
    =regularizers.l2(1e-4))(x)
x = layers.BatchNormalization()(x)
x = keras.activations.silu(x)
x = layers.SpatialDropout1D(0.6)(x)
x_main = layers.Conv1D(16, kernel_size=(3), padding="same",
    kernel_regularizer=regularizers.l2(1e-4))(x)
x_main = keras.activations.silu(x_main)
x_main = layers.SpatialDropout1D(0.5)(x_main)
x_main = layers.Conv1D(16, kernel_size=(3), padding="same",
    kernel_regularizer=regularizers.l2(1e-4))(x_main)
x_main = keras.activations.silu(x_main)
x_main = layers.SpatialDropout1D(0.6)(x_main)
conc = layers.Multiply()([x, x_main])
x_final = layers.MaxPooling1D(pool_size = 4, strides = 2)(conc)
x_final = layers.Flatten()(x_final)
x_final = layers.Dropout(0.4)(x_final)

out_ln_ic50 = Dense(kbins, activation='softmax', name='LN_IC50')(x_final)
out_auc = Dense(kbins, activation='softmax', name='AUC')(x_final)
out_dr = Dense(kbins, activation='softmax', name='Drug_Response')(x_final
    )
out_rmse = Dense(kbins, activation='softmax', name='RMSE')(x_final)
out_zscore = Dense(kbins, activation='softmax', name='Z_SCORE')(x_final)

model = Model(inputs=inputs, outputs=[out_ln_ic50, out_auc, out_dr,
    out_rmse, out_zscore])


model.compile(
    optimizer="adam",
    loss={
        "LN_IC50": "categorical_crossentropy",
        "AUC": "categorical_crossentropy",
        "Drug_Response": "categorical_crossentropy",
        "RMSE": "categorical_crossentropy",
        "Z_SCORE": "categorical_crossentropy"
    },
    loss_weights={
        "LN_IC50": 1.5,
        "AUC": 1.0,
        "Drug_Response": 1.0,
        "RMSE": 1.0,
        "Z_SCORE": 1.0
```

```
    },
    metrics={
        "LN_IC50": "categorical_accuracy",
        "AUC": "categorical_accuracy",
        "Drug_Response": "categorical_accuracy",
        "RMSE": "categorical_accuracy",
        "Z_SCORE": "categorical_accuracy"
    }
)

early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_LN_IC50_loss', mode="min", patience=10,restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_LN_IC50_loss', factor=0.2, patience=7, min_lr=0.000001)
model_checkpoint = ModelCheckpoint("model_conv_softmax_3.keras", monitor="val_LN_IC50_loss", mode='min')

history = model.fit(
    X_train,
    y_train_dict_onehot,
    validation_data=(X_val, y_val_dict_onehot),
    epochs=100,
    batch_size=32,
    callbacks=[early_stopping, PlotLossesKerasTF(), reduce_lr, model_checkpoint]
)
```

Code extract 17: Assigning importance to target variables for the purpose of obtaining better results in LN_IC_{50}.

```
@keras.saving.register_keras_serializable()
class CombinedCrossEntropyMSE(tf.keras.losses.Loss):
    def __init__(self, reduction="auto", alpha=1.0, beta=1.0, name="combined_crossentropy_mse"):
        super().__init__(name=name)
        self.alpha = alpha
        self.beta = beta
        self.ce = tf.keras.losses.CategoricalCrossentropy(reduction='none')

    @classmethod
    def from_config(cls, config):
        return cls(**config)

    def call(self, y_true, y_pred):
        valid_mask = tf.reduce_any(y_true != -1, axis=-1)
        y_true = tf.boolean_mask(y_true, valid_mask)
```

```
        y_pred = tf.boolean_mask(y_pred, valid_mask)

        # Cross entropy loss (per sample)
        ce_loss = self.ce(y_true, y_pred)

        # Compute "soft" class expectation (differentiable)
        class_range = tf.cast(tf.range(tf.shape(y_pred)[-1]), tf.float32)
        true_class = tf.reduce_sum(y_true * class_range, axis=-1)
        pred_class = tf.reduce_sum(y_pred * class_range, axis=-1)

        mse_loss = tf.square(true_class - pred_class)

        # Final loss per sample
        total_loss = self.alpha * ce_loss + self.beta * mse_loss
        return tf.reduce_mean(total_loss)


input_shape = (X_train.shape[1],)
inputs = Input(shape=input_shape)

x = layers.Dense(2048, activation="silu")(inputs)
x = layers.Dropout(0.5)(x)
x = layers.Reshape((16, 128))(x)
x = layers.Conv1D(16, kernel_size=(5), padding="same", kernel_regularizer
    =regularizers.l2(1e-4))(x)
x = layers.BatchNormalization()(x)
x = keras.activations.silu(x)
x = layers.SpatialDropout1D(0.5)(x)
x = layers.AveragePooling1D(pool_size=2)(x)
x = layers.BatchNormalization()(x)
x = layers.Conv1D(16, kernel_size=(3), padding="same", kernel_regularizer
    =regularizers.l2(1e-4))(x)
x = layers.BatchNormalization()(x)
x = keras.activations.silu(x)
x = layers.SpatialDropout1D(0.6)(x)
x_main = layers.Conv1D(16, kernel_size=(3), padding="same",
    kernel_regularizer=regularizers.l2(1e-4))(x)
x_main = keras.activations.silu(x_main)
x_main = layers.SpatialDropout1D(0.5)(x_main)
x_main = layers.Conv1D(16, kernel_size=(3), padding="same",
    kernel_regularizer=regularizers.l2(1e-4))(x_main)
x_main = keras.activations.silu(x_main)
x_main = layers.SpatialDropout1D(0.6)(x_main)
conc = layers.Multiply()([x, x_main])
x_final = layers.MaxPooling1D(pool_size = 4, strides = 2)(conc)
x_final = layers.Flatten()(x_final)
x_final = layers.Dropout(0.4)(x_final)
```

```
59
60 out_ln_ic50 = Dense(kbins, activation='softmax', name='LN_IC50')(x_final)
61 out_auc = Dense(kbins, activation='softmax', name='AUC')(x_final)
62 out_dr = Dense(kbins, activation='softmax', name='Drug_Response')(x_final
       )
63 out_rmse = Dense(kbins, activation='softmax', name='RMSE')(x_final)
64 out_zscore = Dense(kbins, activation='softmax', name='Z_SCORE')(x_final)
65
66 model = Model(inputs=inputs, outputs=[out_ln_ic50, out_auc, out_dr,
       out_rmse, out_zscore])
67
68
69 model.compile(
70     optimizer="adam",
71     loss={
72         "LN_IC50": CombinedCrossEntropyMSE(alpha=1.0, beta=0.7),
73         "AUC": "categorical_crossentropy",
74         "Drug_Response": "categorical_crossentropy",
75         "RMSE": "categorical_crossentropy",
76         "Z_SCORE": "categorical_crossentropy"
77     },
78     loss_weights={
79         "LN_IC50": 1.6,
80         "AUC": 1.0,
81         "Drug_Response": 1.0,
82         "RMSE": 1.0,
83         "Z_SCORE": 1.0
84     },
85     metrics={
86         "LN_IC50": "categorical_accuracy",
87         "AUC": "categorical_accuracy",
88         "Drug_Response": "categorical_accuracy",
89         "RMSE": "categorical_accuracy",
90         "Z_SCORE": "categorical_accuracy"
91     }
92 )
93
94 early_stopping = tf.keras.callbacks.EarlyStopping(monitor='
       val_LN_IC50_loss', mode="min", patience=10,restore_best_weights=True)
95 reduce_lr = ReduceLROnPlateau(monitor='val_LN_IC50_loss', factor=0.2,
       patience=7, min_lr=0.000001)
96 model_checkpoint = ModelCheckpoint("model_conv_softmax_4.keras", monitor=
       "val_LN_IC50_loss", mode='min')
97
98 history = model.fit(
99     X_train,
100    y_train_dict_onehot,
```

```
    validation_data=(X_val, y_val_dict_onehot),
    epochs=100,
    batch_size=64,
    callbacks=[early_stopping, PlotLossesKerasTF(), reduce_lr,
    model_checkpoint]
)
```

Code extract 18: Definition of a custom error function, including categorical cross entropy and MSE.

Bibliography

[1] National Cancer Institute. The cancer genome atlas (tcga), 2024. URL https://www.cancer.gov/ccg/research/genome-sequencing/tcga. Accessed: 2025-06-26.

[2] Genomics of Drug Sensitivity in Cancer Project. Genomics of drug sensitivity in cancer (gdsc), 2024. URL https://www.cancerrxgene.org/. Accessed: 2025-06-26.

[3] Yehudit Hasin, Marcus Seldin, and Aldons Lusis. Multi-omics approaches to disease. *Genome Biology*, 18(1):83, 2017.

[4] Susana Garcia-Recio, Paola Zagami, Brooke M. Felsheim, Amy Wheless, Kerry Thomas, Renato Trimarchi, Lisa A. Carey, and Charles M. Perou. Understanding metastasis mixed-treatment responses through genomic analyses. *npj Breast Cancer*, 11(1):9, 2025.

[5] Scott M. Lundberg and Su-In Lee. Shap (shapley additive explanations), 2024. URL https://shap.readthedocs.io/en/latest/. Accessed: 2025-06-29.

[6] Samira Alipour. Genomics of drug sensitivity in cancer (gdsc). https://www.kaggle.com/datasets/samiraalipour/genomics-of-drug-sensitivity-in-cancer-gdsc, 2023. Accessed: 2025-06-23.

[7] Sanger Institute. Cosmic: Catalogue of somatic mutations in cancer. https://cancer.sanger.ac.uk/cosmic/login, 2025. Accessed: 2025-06-23.

[8] Sanger institute. https://www.sanger.ac.uk/, 2024. Accessed: 2025-06-23.

[9] Scikit-learn developers. *sklearn.preprocessing.OrdinalEncoder*. Scikit-learn, 2025. URL https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OrdinalEncoder.html. Version 1.7.0.

[10] scikit-learn developers. *sklearn.impute.KNNImputer*. scikit-learn, 2024. Accessed: 2025-06-23.

[11] Hey Amit. One-hot encoding explained, November 2024. URL https://medium.com/@heyamit10/one-hot-encoding-explained-0b0130ccd78e. Accessed: 2025-06-23.

[12] scikit-learn developers. *sklearn.impute.train_test_split*. scikit-learn, 2024. Accessed: 2025-06-23.

[13] scikit-learn developers. scikit-learn: Machine learning in python – `StandardScaler`. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html, 2011. Accessed: 2025-06-23.

[14] scikit-learn developers. scikit-learn: Machine learning in python – RobustScaler. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html, 2011. Accessed: 2025-06-23.

[15] Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. Why do tree-based models still outperform deep learning on tabular data? In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[16] Simon Müller. Overcoming the limitations of tree-based models in time series forecasting. Medium, February 2024. URL https://medium.com/@simon.peter.mueller/overcoming-the-limitations-of-tree-based-models-in-time-series-forecasting-c2c5bd Accessed on June 16, 2025.

[17] Piotr Migdał et al. livelossplot: Live training loss plot in jupyter notebook. https://pypi.org/project/livelossplot/, 2025. Version 0.5.6; Accessed: 2025-06-16.

[18] Google and the TensorFlow Team. ModelCheckpoint. TensorFlow API Documentation, . URL https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint. Version 2.16.1; accessed: 2025-06-16.

[19] Google and the TensorFlow Team. EarlyStopping. TensorFlow API Documentation, . URL https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping. Version 2.16.1; accessed: 2025-06-16.

[20] Google and the TensorFlow Team. ReduceLROnPlateau. TensorFlow API Documentation, . URL https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ReduceLROnPlateau. Version 2.16.1; accessed: 2025-06-16.

[21] Google and the TensorFlow Team. MSE. TensorFlow API Documentation, . URL https://www.tensorflow.org/api_docs/python/tf/keras/losses/MSE. Version 2.16.1; accessed: 2025-06-16.

[22] Google and the TensorFlow Team. tf.keras.metrics.RootMeanSquaredError. TensorFlow API Documentation, . URL https://www.tensorflow.org/api_docs/python/tf/keras/metrics/RootMeanSquaredError. Version 2.16.1; accessed: 2025-06-16.

[23] Google and the TensorFlow Team. LogCosh. TensorFlow API Documentation, . URL https://www.tensorflow.org/api_docs/python/tf/keras/losses/LogCosh. Version 2.16.1; accessed: 2025-06-16.

[24] Google and the TensorFlow Team. tf.keras.losses.MAE. TensorFlow API Documentation, . URL https://www.tensorflow.org/api_docs/python/tf/keras/losses/MAE. Version 2.16.1; accessed: 2025-06-16.

[25] Google and the TensorFlow Team. Huber. TensorFlow API Documentation, . URL https://www.tensorflow.org/api_docs/python/tf/keras/losses/Huber. Version 2.16.1; accessed: 2025-06-16.

[26] Sercan O. Arik and Tomas Pfister. Tabnet: Attentive interpretable tabular learning, 2020. URL https://arxiv.org/abs/1908.07442.

[27] Li-Wei Chang, Cheng-Te Li, Chun-Pai Yang, and Shou-de Lin. Learning on missing tabular data: Attention with self-supervision, not imputation, is all you need. 16(3), 2025. ISSN 2157-6904. doi: 10.1145/3729241. URL https://doi.org/10.1145/3729241.

[28] Alok Sharma, Edwin Vans, Daichi Shigemizu, Keith A. Boroevich, and Tatsuhiko Tsunoda. Deepinsight: A methodology to transform a non-image data to an image for convolution neural network architecture. *Scientific Reports*, 9(1):11399, 2019.

[29] M. S. Mathew, S. T. Kandukuri, and C. W. Omlin. Soft ordering 1-d cnn to estimate the capacity factor of windfarms for identifying the age-related performance degradation. In *Proceedings of the PHM Society European Conference*, volume 8, page 9, June 2024. doi: 10.36001/phme.2024.v8i1.4028.

[30] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning, 2017. URL https://arxiv.org/abs/1702.03118.

[31] Xudong Lin, Lin Ma, Wei Liu, and Shih-Fu Chang. Context-gated convolution, 2020. URL https://arxiv.org/abs/1910.05577.

[32] Towaki Takikawa, David Acuna, Varun Jampani, and Sanja Fidler. Gated-scnn: Gated shape cnns for semantic segmentation, 2019. URL https://arxiv.org/abs/1907.05740.

[33] Qi Zheng, Chang Yu, Jin Cao, Yongshun Xu, Qianwen Xing, and Yinxin Jin. Advanced payment security system:xgboost, lightgbm and smote integrated, 2024. URL https://arxiv.org/abs/2406.04658.

[34] Mizuho Nishio, Mitsuo Nishizawa, Osamu Sugiyama, Ryosuke Kojima, Masahiro Yakami, Tomohiro Kuroda, and Kaori Togashi. Computer-aided diagnosis of lung nodule using gradient tree boosting and bayesian optimization. *PLOS ONE*, 13(4):e0195875, April 2018. ISSN 1932-6203. doi: 10.1371/journal.pone.0195875. URL http://dx.doi.org/10.1371/journal.pone.0195875.

[35] Scikit-learn developers. *sklearn.model_selection.GridSearchCV*. Scikit-learn, 2025. URL https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html. Version 1.7.0.

[36] Scikit-learn developers. *sklearn.model_selection.RandomizedSearchCV*. Scikit-learn, 2025. URL https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html. Version 1.7.0.

[37] Scikit-learn developers. *sklearn.preprocessing.OneHotEncoder*. Scikit-learn, 2025. URL https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html. Version 1.7.0.

[38] Scikit-learn developers. *sklearn.feature_selection.RFE*. Scikit-learn, 2025. URL https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html. Version 1.7.0.

[39] Scikit-learn developers. *sklearn.feature_selection.SelectFromModel*. Scikit-learn, 2025. URL https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectFromModel.html. Version 1.7.0.

[40] T. Tony Cai and Rong Ma. Theoretical foundations of t-sne for visualizing high-dimensional clustered data, 2022. URL https://arxiv.org/abs/2105.07536.

[41] Scikit-learn developers. t-sne, 2025. Accessed: June 20, 2025.

[42] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction, 2020. URL https://arxiv.org/abs/1802.03426.

[43] Leland McInnes, John Healy, Nathaniel Saul, and Lukas Großberger. Umap: Uniform manifold approximation and projection for dimension reduction. https://umap-learn.readthedocs.io/en/latest/, 2025. Accessed: June 20, 2025.