# Regression-based Particle Position Detection in Resistive Silicon Detectors

Chiara Roberta Casale
*Politecnico di Torino*
Student ID: s322574
Email: s322574@studenti.polito.it

Yegane Bagheri
*Politecnico di Torino*
Student ID: s327779
Email: s327779@studenti.polito.it

***Abstract*—In this report, we propose a possible approach to the problem of predicting the position of particles through a sensor called RSD (Resistive Silicon Detector). Our solution consists of finding the relevant features of the signal and using them to train a regression model.**

## I. Problem overview

In this paper, we present a solution to a regression problem aimed at predicting the position of particles as they pass through a sensor. The training set consists of 385,500 events, each characterized by the following features:

- `pmax[0]`, `pmax[1]`, ..., `pmax[17]`: Magnitude of the positive peak of the signal in mV.
- `negpmax[0]`, `negpmax[1]`, ..., `negpmax[17]`: Magnitude of the negative peak of the signal in mV.
- `tmax[0]`, `tmax[1]`, ..., `tmax[17]`: Delay (in ns) from a reference time when the positive peak of the signal occurs.
- `area[0]`, `area[1]`, ..., `area[17]`: Area under the signal.
- `rms[0]`, `rms[1]`, ..., `rms[17]`: Root mean square (RMS) value of the signal.

The features `x` and `y` represent the position of the particle in a given event and serve as the target features.

The evaluation set consists of 128,500 events and lacks the target features `x` and `y`.

The data provided for this competition originates from an RSD (Resistive Silicon Detector) sensor, which detects particle passage across its 2-dimensional surface. Measurements are collected by 12 pads distributed across the sensor, with each pad recording `pmax, negpmax, tmax, area, rms` for every event. Notably, there are 18 readings of each feature, instead of 12, meaning that a subset of the features contains noise.

We should note that all variables, including the target variables, are numerical, and there are no missing data.

We can notice that there are 3,855 tuples of `x` and `y`, indicating 100 events for every combination of `x` and `y`. The values of `x` and `y` range from 200 to 600, and plotting their values reveals a pattern, as depicted in Figure 1. We know that the pads are asterisk-shaped, we can assume that the particles can not pass through them.

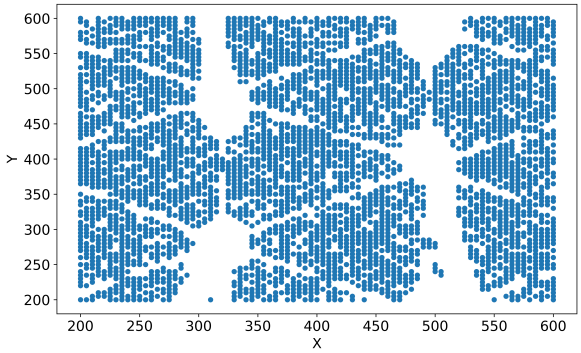


Fig. 1. Pattern of x and y in the sensor

Deciding the approach to identifying the noise from the 18 readings and then applying it has been the primary focus of the preprocessing phase.

## II. Proposed approach

### A. Preprocessing

We explored two distinct approaches for identifying noise in the dataset:

1) Computing the z-score by row and replacing outliers.
2) Conducting various analyses to compare columns and dropping those deemed noisy.

For the first approach, we assumed that noise could exist in different columns for each row, a hypothesis later discarded for the second approach. As each row represents an event (the pathway of a particle), we initially calculated the noise in each row by computing the z-score. For each set of 18 readings of the features, we:

- Computed the z-score by row.
- Replaced outliers with the next valid observation in the row.

This approach revealed that for some columns, almost 100% of values were replaced, indicating these columns were entirely composed of noise. After noticing this, we attempted a hybrid path by dropping some columns and replacing noise in others for each row. However, we discarded this approach due to its lack of control over the impact of replaced values on the

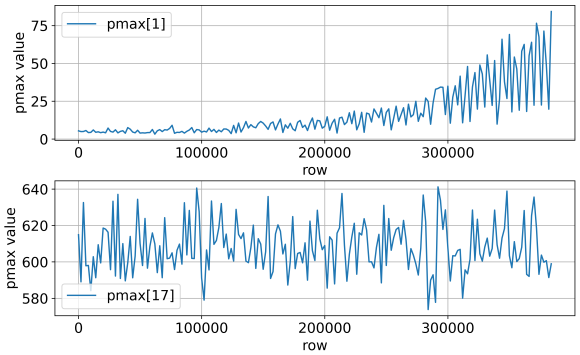


Fig. 2. Comparison between pmax[1] and pmax[17]

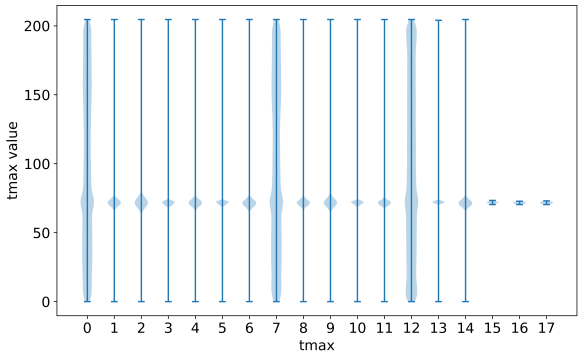


Fig. 3. Violin plots for tmax

model. It became clear deleting the noisy columns emerged as the more effective and straightforward solution.

For the second approach, we started with the opposite assumption: noise is consistently present in the same columns for every row. To identify noisy columns, we:

- Plotted graphs to compare columns with one another.
- Analyzed and compared the range of values of the columns by computing their min, max, mean, and percentiles (0.1, 0.25, 0.5, 0.75, and 0.9).

Firstly, we compared their minimum and maximum values, noticing if there were any differences. By using the percentiles, we confirmed whether differences existed or if variations in their minimum or maximum values were attributable to outliers. We also used the percentiles to examine if the values were concentrated within specific ranges and to identify any differences. The columns we dropped are listed below:

- For `pmax`, most columns show values between 0 and 150. Exceptions include `pmax[0]`, `pmax[7]`, `pmax[12]` (which have much lower values), and `pmax[15]`, `pmax[16]`, `pmax[17]` (much higher values). Figure 2 illustrates this visually by comparing `pmax[1]` with `pmax[17]`.
- `negpmax` exhibits more variations, making it difficult to identify the noise. `negpmax[16]` and `negpmax[17]` differ significantly to the rest. By the 75th percentile, most values are just below zero, while `negpmax[16]` and `negpmax[17]` are around -40.
- Every `area` is highly correlated with its `pmax`, except for the columns with the same pad of the deleted `pmax`. Similarly to `pmax`, columns `area[0]`, `area[7]`, `area[12]` have a smaller range, while `area[15]`, `area[16]`, `area[17]` have larger ranges.
- For `tmax`, columns have a range of 0 to 204.6, except for `tmax[15]`, `tmax[16]`, `tmax[17]`. While most of the values are concentrated near 70 for all the columns, `tmax[15]`, `tmax[16]`, `tmax[17]` have a different distribution. The Violin plot in Figure 3 visualizes these variations.

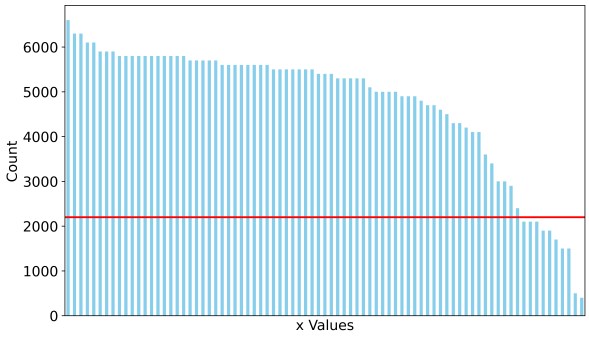


Fig. 4. Frequency of x's values

To recap, the columns we dropped, because we identified them as noise, are: `pmax[0]`, `area[0]`, `tmax[0]`, `pmax[7]`, `area[7]`, `tmax[7]`, `pmax[12]`, `area[12]`, `tmax[12]`, `pmax[15]`, `area[15]`, `tmax[15]`, `pmax[16]`, `negpmax[16]`, `area[16]`, `tmax[16]`, `pmax[17]`, `negpmax[17]`, `area[17]`, `tmax[17]`.

In addition to noise identification, we removed rows with infrequent occurrences. Some `x` values were less common than others, particularly those corresponding to the asterisk pads (Figure 3). We chose to delete `x` values with fewer than 2200 occurrences.

The final preprocessing step involved feature selection, where we removed all `rms` columns due to their lower importance when fitting the model. Considering the extensive number of columns in the dataset, we knew that reducing the number of columns was going to be beneficial.

### *B. Model selection*

The evaluation score used for the public score, which we also used to evaluate our models, is a Euclidean Distance. It is computed as follows: for all $n$ events $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with predictions

$(\hat{x}_1, \hat{y}_1), (\hat{x}_2, \hat{y}_2), \ldots, (\hat{x}_n, \hat{y}_n)$,

$$d = \frac{1}{n} \sum_{i=1}^{n} \sqrt{(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2}$$

The models we chose to work with are:

- **Random Forest Regression**: It is a tree-based ensemble model that leverages the decision tree model by using subsets of data to improve accuracy and control overfitting. We expected this model to work well with *n_estimators=100*, a default parameter, as it has already been used for position reconstruction in RSD sensors [1]. Due to the extended time required for running Random Forest with such a large dataset, we performed additional feature selection for this model. We dropped the remaining `area` and `tmax` columns, leaving only the `pmax` and `negpmax` columns not deleted during the noise identification process.
- **Ridge**: We used this model as a baseline.
- **Gradient Boosting Regression**: This approach aims to improve predictions resulting from a decision tree (similar to Random Forest). It can be applied to both regression and classification tasks. The trees are grown sequentially, with each tree fitting on a modified version of the original dataset, using information (specifically the residuals) from previously grown trees. The boosting approach learns slowly, enhancing performance compared to fitting a single large decision tree to the data [2].

Among the many libraries for Gradient Boosting, we initially tested the estimator provided by *scikit-learn*. Still, we discarded it after realizing it took a similar time to Random Forest but with a worse performance. Instead, we used **XGBoost**, which is notably faster than other similar methods (like Gradient Boosting and Random Forest) but requires more hyperparameter tuning for optimal performance [3].

As discussed in the next section, we then searched for the best hyperparameters to further improve the score.

### C. Hyperparameters tuning

For tuning Ridge and Random Forest, we conducted a grid search with different combinations of parameters. We used the K-fold cross-validation inherently present in GridSearchCV, setting $K = 3$. This optimization was performed on the full development set, thanks to the cross-validation.

XGBoost Regression requires tuning many parameters, making a simple grid search impractical. To address this, we opted for **Optuna**. Optuna is a hyperparameter optimization framework designed to minimize/maximize an objective function by taking a set of hyperparameters as input and returning its score. Each optimization process is a study, and each objective function evaluation is a trial. Optuna efficiently explores large search spaces and prunes unpromising trials using a stochastic approach [4].

Before initiating the tuning process, we performed a train-test split on our dataset, allocating 80% of the data to the training set and the remaining 20% to the test set. After identifying the best parameters through Optuna, we fitted the full development set.

TABLE I
HYPERPARAMETER TUNING

| **Model** | **Parameter configuration** |
|---|---|
| Random Forest Regression | max_depth = [7, None]<br>max_features = ['auto', 'sqrt'] |
| Ridge | alpha = 0.001 $\rightarrow$ 10000, log scale<br>solver = ['auto', 'svd', 'sparse_cg', 'lsqr'] |
| XGBoost Regressor | n_estimators = [100, 200, 300]<br>learning_rate = 0.001 $\rightarrow$ 0.1, log scale<br>max_depth = 1 $\rightarrow$ 10, step 1<br>subsample = 0.05 $\rightarrow$ 1.0<br>colsample_bytree = 0.05 $\rightarrow$ 1.0<br>min_child_weight = 1 $\rightarrow$ 20, step 1 |

TABLE II
HYPERPARAMETERS SELECTED FOR XGBOOST REGRESSOR

| Parameter | Value | Parameter | Value |
|---|---|---|---|
| n_estimators | 300 | subsample | 0.872 |
| learning rate | 0.026 | colsample_bytree | 0.746 |
| max_depth | 10 | min_child_weight | 20 |

The search space for all the models is summarized in Table I.

## III. RESULTS

We can now compare the results obtained with our three models, using the hyperparameters discussed previously.

- For Ridge Regression, the best combination of hyperparameters is {alpha = 0.1, solver = 'sparse_cg'}. With this configuration, we achieve a distance of 18.157 on the public score. We fitted this model as a baseline, but its result is still relatively high. We can assume that models based on linear regression may not be well-suited for this particular task.
- For Random Forest Regression, the best combination of hyperparameters is {max_depth = None, max_features = 'sqrt'}. With this configuration, we achieve a distance of 5.170 on the public score.
- For XGBoost Regression, the best combination of hyperparameters is shown in Table II. With this configuration, we achieve a distance of 4.959 on the public score.

At the moment of writing this, the best score on the platform is a distance of 3.754. We explain what we could have done differently to achieve a better score in the next section.

## IV. DISCUSSION

The results show how Random Forest Regression is an apt model for this task. Even with heavy feature selection, and only by tuning a few hyperparameters, Random Forest gives a good result. XGBoost Regression tuned with Optuna gives a better result in a faster time, as we expected.

As already mentioned, both results are noticeably worse than the best result on the platform. Here are a few considerations on what we could have done differently. First of all, we could have explored other approaches for Random Forest Regression. We considered `pmax` and `negpmax` as the

most important features, but a different configuration could have given better results. Even by drastically reducing the number of features, the hyperparameter tuning took a long time. Alternatives to grid search could be used to mitigate this issue.

While the hyperparameter tuning with Optuna is optimal for XGBoost, it has the disadvantage of training the model using a train-test split. The Optuna library offers an experimental hyperparameter tuning framework called OptunaSearchCV. It trains the model using Optuna and it can inherently use a K-fold cross-validation. While it is still an experimental method, it could offer a better result thanks to the cross-validation.

A different approach could have been doing more pre-processing. One of the patterns we noticed is that there are 100 rows for each position. We assumed the signals measured from the same position should be relatively similar. After examining and comparing the data, we mostly confirmed that assumption. There were exceptions: some pads had very different measures from the same measure within the same position. While we did not further explore this topic, it would be interesting to know how managing these anomalies would affect the performance.

## REFERENCES


[1] M. Tornago, F. Giobergia, L. Menzio, F. Siviero, R. Arcidiacono, N. Cartiglia, M. Costa, M. Ferrero, G. Gioachin, M. Mandurrino, and V. Sola, "Silicon sensors with resistive read-out: Machine learning techniques for ultimate spatial resolution," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 1047, p. 167816, 2023.

[2] G. James, D. Witten, T. Hastie, and R. Tibshirani., *An Introduction to Statistical Learning*. Springer New York, 2013.

[3] C. Bentéjac, A. Csörgő, and G. Martínez-Muñoz, "A comparative analysis of gradient boosting algorithms," *Artificial Intelligence Review*, vol. 54, pp. 1937–1967, Mar 2021.

[4] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, (New York, NY, USA), p. 2623–2631, Association for Computing Machinery, 2019.