POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Impact of duplicated values and variable type issues in a dataset on the performance of ML classification algorithms

Project Report
Data and Information Quality

Project ID: 17
Data Quality issues: duplication & variable type
ML task: classification

Authors:

Claudio Galimberti -
Luca Gerin -

Academic Year: 2023-24

Contents

1 Introduction

1.1. Scenario

Data Quality (DQ) is becoming increasingly important for the success of machine learning (ML) tasks, including classification. The quality of the input data directly impacts the performance, accuracy, and reliability of machine learning models.
To maximize the benefits of machine learning models, it's crucial to invest resources, with the correct trade-off, in ensuring the quality of the data to be used for training and evaluation. The machine learning pipeline must include data exploration, cleaning and validation, to achieve reliable and accurate classification results.

However, requirements for having a good DQ are changing: we must no longer just ensure a level of data quality for the traditional aspects, such as Completeness, Accuracy, or Consistency. The success of a ML analysis can depend a multitude of new data issues, such as Dimensionality, Feature Dependency, or Distinctness.
Data quality specialists must have a good understanding of the impact of both the "traditional" and ''new" DQ issues on a ML analysis.

1.2. Objective

The objective of this project is to investigate the impact on a machine learning task, that is classification, of some data quality issues. The chosen data quality issues that will be examined are the presence of duplicated values and of different variable types in the dataset.

A more complete awareness of the consequences of these data quality problems on the performance of the classification task by ML algorithms is to be achieved, in order to have a quantitative understanding of their influence, and be able in the future to make more informed decisions on how much and where to invest in DQ when facing a new ML challenge.

1.3. Method

In order to evaluate the performance of ML classification algorithms (DecisionTree, LogisticRegression, KNN, RandomForest, AdaBoost, MLP) when they utilize data with different data quality level, we will set up a pipeline in python, divided in 6 main phases. This pipeline is composed as follows:

- Data Collection: a dataset suitable for the desired ML task is generated
- Data Pollution: errors related to the data quality issues under analysis are injected into the dataset
- Data Analysis and Evaluation: the performance of the ML task performed using the *polluted* data is evaluated with different metrics, and plots are produced to show the results
- Data Preparation: if possible, different DQ improvement techniques are applied to correct the injected DQ issue
- Data Analysis and Evaluation: the performance of the ML task performed using the *cleaned* data is evaluated with different metrics, and plots are produced to show the results

The pipeline is represented in figure 1.1.

The metrics of interest to evaluate the ML tasks are performance, over-fitting and speed.

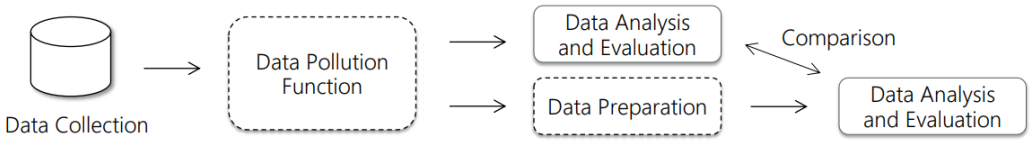


Figure 1.1: Pipeline for the project.

When evaluating the behavior of the task with variable type issues, the data preparation and second data analysis phases are skipped, because this problem is solved only by encoding the variables in a way that can be utilized by the ML algorithms. However, we expect the presence of such variables to influence the results.

For what concerns the duplicated values, 10 experiments will be performed. In each of them, different kinds of duplicates are injected, and different scenarios in which the percentage of duplicated values varies will be analyzed.
For what concerns the presence of features with different variable types, 10 experiments will be performed, each one with different combinations of types of data inside the dataset.

Intermediate and final results will be at the end compared.

2 | Setup choices

2.1. Data collection

The starting dataset and corresponding labels are generated using the `make_dataset_for_classification()` function, defined as follows:

```
def make_dataset_for_classification(n_samples, n_features, n_informative, n_redundant, n_repeated,
                                    n_classes, n_clusters_per_class, weights, flip_y, class_sep,
                                    hypercube, seed):
    X, y = make_classification(n_samples=n_samples, n_features=n_features,
                               n_informative=n_informative, n_redundant=n_redundant,
                               n_repeated=n_repeated, n_classes=n_classes,
                               n_clusters_per_class=n_clusters_per_class, weights=weights,
                               flip_y=flip_y, class_sep=class_sep, hypercube=hypercube,
                               random_state=seed)
    return X, y
```

The function makes use of the `sklearn.datasets.make_classification()` function, that is a utility in the scikit-learn library that generates synthetic datasets for binary or multi-class classification tasks, and is particularly useful for creating datasets for testing and prototyping machine learning models.

The function is called with the following standard parameters:

```
    X, y = make_dataset_for_classification(n_samples=1000, n_features=5, n_informative=5,
                   n_redundant=0, n_repeated=0, n_classes=2, n_clusters_per_class=2,
                   weights=None, flip_y=0.01, class_sep=1.0, hypercube=True, seed=2023)
```

We decided to start from a standard dataset, in order to have total control over the data quality issues to inject in it.

Part of the generated dataset, that has 1000 rows, can be seen in figure 2.1.

	0	1	2	3	4
0	-3.044852	0.401225	1.876653	1.566605	-2.132950
1	1.445783	1.330753	1.140285	0.706253	2.422165
2	2.404213	1.620161	-0.710273	-3.793487	1.183956
3	1.573431	-0.558834	0.359672	0.777892	2.637753
4	-0.849001	2.114740	3.329424	-1.255837	-1.173801
5	-1.752346	0.251198	-2.229832	-0.823034	-1.455396
6	0.414611	-1.407412	-2.849978	-3.032828	-1.189222
7	0.130171	-2.016133	-0.542940	-2.755919	0.404544

Figure 2.1: Part of the generated data to use for the experiments.

2.2. Experiments

2.2.1. Experiments for the duplicated values issue

To test the effect of duplicated values, we carried out 10 different experiments, each one injecting a different combination of kinds of duplicated values.

We set up functions to create each duplicate value by creating a new tuple starting from an existing one but modifying one or more (numeric) features in one of the following ways:

- *round*: the value is rounded to a certain decimal
- *noise*: some noise or offset is added to the value

The combinations of types of duplicates injected for the 10 experiments are shown in table 2.1

Table 2.1: Duplicated values experiments.

#	Experiment	Description
1	`Noise 1 column`	Duplicated values are generated starting from one random row and modifying one feature of that row using a noising function
2	`Round 1 column`	Duplicated values are generated starting from one random row and modifying one feature of that row using a rounding function
3	`Noise 2 columns of the same row`	Duplicated values are generated starting from one random row and modifying two features of that row using a noising function
4	`Round 2 columns of the same row`	Duplicated values are generated starting from one random row and modifying two features of that row using a rounding function
5	`Noise 1 column, round a different column of the same row`	Duplicated values are generated starting from one random row modifying one feature of that row using a noising function and modifying a different feature of the same row using a rounding function
6	`Noising and rounding applied together to 2 columns of the same row`	Duplicated values are generated starting from one random row using a noising function and a rounding function applied together to two features
7	`Noise 3 different columns of the same row`	Duplicated values are generated starting from one random row and modifying three features of that row using a noising function
8	`Round 3 different columns of the same row`	Duplicated values are generated starting from one random row and modifying three features of that row using a rounding function
9	`Noise 2 columns, round a different column of the same row`	Duplicated values are generated starting from one random row modifying two features of that row using a noising function and modifying a different feature of the same row using a rounding function
10	`Noise 1 column, round 2 different columns of the same row`	Duplicated values are generated starting from one random row modifying one feature of that row using a noising function and modifying two other features of the same row using a rounding function

For each experiment, 10 datasets are produced with a different percentage of generated duplicated values, from **5%** to **50%** with increasing steps of 5%.

2.2.2. Experiments for the variable type issue

For what concerns the variable type issue, we conducted 10 experiments, adding each time to the dataset different new columns containing categorical values.

The columns containing categorical values we decided to employ and combine in our experiments are generated as follows:

- A column containing boolean values `True` or `False`, respectively if the sum of all the numeric features of the row is positive or negative
- A column containing enumeration values `POSITIVE` or `NEGATIVE`, respectively if the sum of all the numeric features of the row is positive or negative
- A column containing six different enumeration values representing *colors*, assigned according to the equally wide intervals between the minimum and maximum in which falls the median absolute deviation of the numeric values of each row.
- A column containing enumeration of *number names*, always assigned according to equally wide intervals in which the median absolute deviation falls, but with a parametric and variable number of intervals

While values generated according to the sign of the sum of the values of a row are equally distributed, the values generated using the median absolute deviation have values that are more often present than others.

The 10 experiments are configured as described in table 2.2

Table 2.2: variable type experiments.

#	**Experiment**
1	Add a boolean column for the sign of the sum of the values of each row
2	Add a column with colors, based on the median absolute deviation
3	Add a column with number names up to 10, based on the median absolute deviation
4	Add a column with number names up to 30, based on the median absolute deviation
5	Add a boolean column and a column with either *POSITIVE* or *NEGATIVE*, both according to the sign of the sum of the values of each row
6	Add a boolean column for the sign of the sum of the values of each row, and a column with number names up to 10, based on the median absolute deviation
7	Add a column with colors, based on the median absolute deviation, and a column with number names up to 10, based on the median absolute deviation
8	Add a column with colors, based on the median absolute deviation, and a column with number names up to 30, based on the median absolute deviation
9	Add a boolean column for the sign of the sum of the values of each row, a column with colors, based on the median absolute deviation, and a column with number names up to 10, based on the median absolute deviation
10	Add a boolean column for the sign of the sum of the values of each row, a column with colors, based on the median absolute deviation, and two other columns with number names up to 10 and up to 30, based on the median absolute deviation

The ratio is to gradually increase the number and heterogeneity of categorical values present in the data.

3 Pipeline implementation

3.1. Pipeline steps

Our process starts with data exploration, to have a better understanding of the dataset we are going to use to measure the impact of data quality issues.
The generated dataset is polluted in different ways along 10 experiments for each of the data quality dimensions of interest of duplicated values and variable types, and the resulting data is fed to different classification algorithms to measure its performance.
Afterwards, each duplicated values issue is solved using data cleaning techniques, and the output data is tested again.
The objective is to compare the behavior of the polluted data with the one of the same data after it has been cleaned.

3.2. Data exploration

Basic data exploration is performed, examining:

- the number of columns and rows generated, respectively 1000 and 5
- the types of all features, that are `float64`
- the total number of cells that is 5000
- the number of null and not null values that are respectively 0 and 5000
- the completeness that is 100%
- the presence of exact duplicated values, that are found to be absent

The correlation heatmap is generated and shown in figure 3.1, with no results worthy of note.

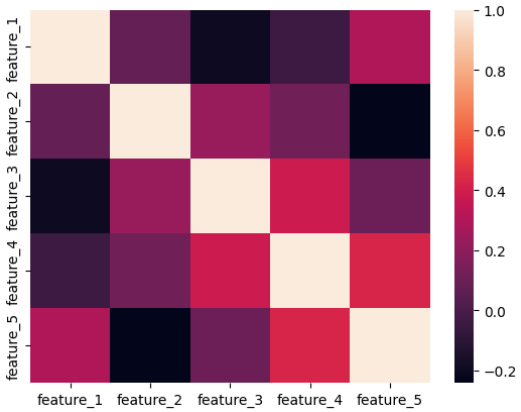


Figure 3.1: Correlation heatmap for the data features.

The distribution of the values for each feature is analyzed by looking at histograms, and no strange behavior is detected. The histograms are shown in figure 3.2.

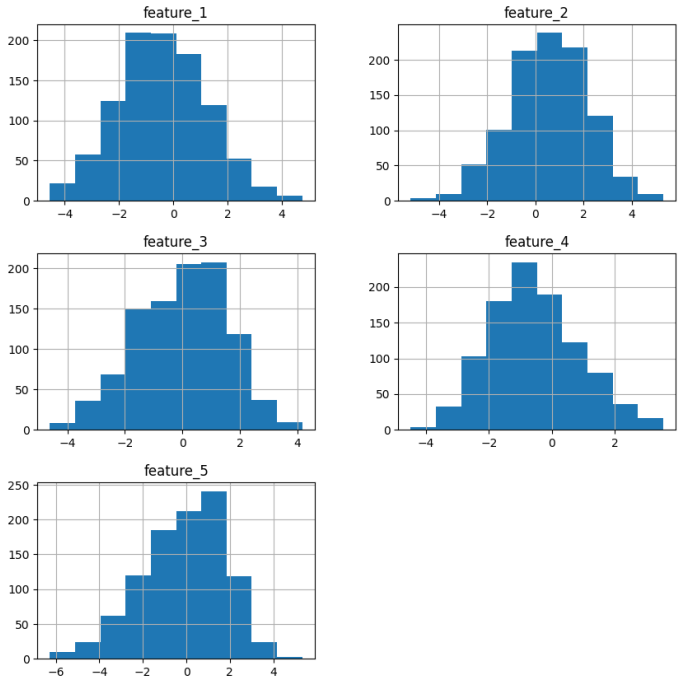


Figure 3.2: Histograms of the distribution of the data features.

3.3. Data pollution

The selected data quality issues of duplicated values and different variable types are injected into the data to generate the desired experiments. To do so, we wrote specific functions taking different parameters to quantify the degree of pollution to inject each time.

3.3.1. Duplicated values injection

To inject duplicates into the data, the following function is employed:

```
polluting_data(dataset, mask, duplicates, function, parameter, modified,
               combining, together, labels)
```

Where the parameters are the followings:

- *dataset*: the dataset to pollute
- *mask*: a mask defining the rows that will be polluted (in this case duplicated)
- *duplicates*: the number of duplicates to create for each value that will be duplicated
- *function*: the function to use to pollute the data when creating a duplicate, it can either be:
  - `round`: the duplicated feature's value is rounded
  - `noise`: the duplicated feature's value is polluted with some noise or offset
- *parameter*: a parameter specific for the function specified with the parameter *function*
- *modified*: a list to hold the modified columns, which is used to keep track of the columns modified for each row in case they have to be reused or discarded when combining more pollution techniques

- *combining*: a boolean determining if the techniques are combined on different columns of the same row
- *together*: a boolean to determine if to pollute the same columns by applying different techniques to the same feature
- *labels*: the list of labels to which the labels of the polluted data must be added

A mask is created using the following function:

```
mask_creator(percentage)
```

This function takes as input a number between 0 and 1, defining the percentage of values that must have a 1 in the mask, and that will be in the future utilized to generate duplicates.
As previously described, this percentage will go from the 5% of the tuples to the 50%, with increasing steps of 5% each time, to conduct 10 tests per experiment.

The *parameter* is chosen randomly in intervals depending on the chosen function to use. If performing a rounding function, the *parameter* is a random number between 2 and 4, representing the number of decimals to round to. If performing a noising function, the *parameter* is a value sampled from a Gaussian distribution with mean in 0 and standard deviation of 0.1, representing the noise to apply to the values.

For each of the 10 experiments, a list of 10 polluted datasets with increasing number of duplicated values is produced.

3.3.2. Variable types injection

In order to generate new features, we exploited functions of the available data, to produce categorical columns that depend on the numerical ones.

Three are the kinds of categorical features that we contemplated.

The first categorical feature we generated is based on the sign of the sum of the numerical values. The function to create this column is the following:

```
add_sum_POS_NEG(df)
```

This function takes as input a dataset and adds a column, named *sum_ sign*, containing either the value `POSITIVE` or `NEGATIVE` according to the sign of the sum of the numerical values.

A similar function `add_sum_POS_NEG_bool(df)` behaves similarly but instead of putting these two values in the new column, now named *sum_ sign_ positive*, puts a *Bool* that is `True` if the sign of the sum of the numerical values is positive, `False` otherwise.

```
add_sum_POS_NEG_bool(df)
```

The second and third categorical features added depend on the interval in which the value of the median absolute deviation (mad) of the numerical columns falls.

The function `add_mad_colors(df)` adds a column, named *mad_ colors*, containing a color out of 6 possible ones depending on the value of the mad.

```
mad_to_color(df)
```

The function `add_mad_numbers(df, num_intervals, col_name='mad_num')` adds a column, named *mad_ num* by default, containing an enum representing a Number in the *num_ intervals* range. The value of the mad of the numerical columns is divided in *num_ intervals* intervals, and a enum.Number is assigned according to which one of these intervals the mad falls in for the considered row.

```
add_mad_numbers(df, num_intervals, col_name='mad_num')
```

A table to which some columns are added using these functions is shown as example in figure 3.3.

	0	1	2	3	4	sum_sign	mad_colors	mad_num	mad_num_2
0	-3.044852	0.401225	1.876653	1.566605	-2.132950	Sign.POSITIVE	Color.YELLOW	Number.SIX	Number.EIGHTEEN
1	1.445783	1.330753	1.140285	0.706253	2.422165	Sign.POSITIVE	Color.GREEN	Number.ONE	Number.THREE
2	2.404213	1.620161	-0.710273	-3.793487	1.183956	Sign.NEGATIVE	Color.ORANGE	Number.SIX	Number.EIGHTEEN
3	1.573431	-0.558834	0.359672	0.777892	2.637753	Sign.POSITIVE	Color.BLUE	Number.THREE	Number.EIGHT
4	-0.849001	2.114740	3.329424	-1.255837	-1.173801	Sign.POSITIVE	Color.YELLOW	Number.SIX	Number.SIXTEEN
...	...	...	...	...	...	...	...	...	...
995	0.852847	-0.726076	-2.160768	-3.265680	-0.749785	Sign.NEGATIVE	Color.PURPLE	Number.FOUR	Number.ELEVEN
996	0.482150	0.830578	0.418977	1.189807	2.350933	Sign.POSITIVE	Color.GREEN	Number.TWO	Number.FOUR
997	0.173973	-1.095862	0.148571	-1.312195	1.049151	Sign.NEGATIVE	Color.BLUE	Number.THREE	Number.SEVEN
998	-1.475855	-0.718746	-2.108539	-1.984390	-2.674930	Sign.NEGATIVE	Color.GREEN	Number.TWO	Number.FOUR
999	-1.730336	0.861486	1.964623	-0.612533	-1.514616	Sign.POSITIVE	Color.PURPLE	Number.FOUR	Number.TWELVE

1000 rows × 9 columns

Figure 3.3: Example of pollution adding categorical values with the described functions.

3.4. Polluted data analysis

The polluted data generated in 10 experiments for the duplicated values issue and in 10 experiments for the variable types issue is utilized to train different classification ML algorithms. Results in terms of performance, distance between trainig and test sets, and speed, are produced and plotted to be analyzed.

```
results = classification(X, y, algorithm, SEED)
```

3.5. Data preparation

Data preparation is performed only for the issue of duplicated data. The variable type issue is left in the database and considerations will be made on the effect of the presence of the added features.

3.5.1. Duplicate detection and data fusion

The first step of duplicate detection is to drop the exact duplicates, if present, and this is accomplished using the `pandas.DataFrame.drop_duplicates()` function.

To find the couples of tuples that will be considered and treated as duplicates, the function `find_matches(df)` is defined. It works by exploiting the *recordlinkage* library and the sorted neighborhood method to reduce the search space.

```
indexer1 = recordlinkage.index.SortedNeighbourhood(
    left_on='feature_2', window=91
)
```

The sorting for the sorted neighborhood method is done using as key the value of *feature_2* column (but other columns would have worked too), and the window size is set to 91. The reason for a window size that is almost the 10% of the size of the dataset is that we will not employ a multi-pass strategy, as results are convincing and efficiency is considered high enough.

Candidate links are compared using the following rules:

```
compare_c = recordlinkage.Compare()
compare_c.numeric('feature_0','feature_0', method='gauss', offset=0.1, scale=0.1,
                  origin=0.0, label='feature_0')
compare_c.numeric('feature_1','feature_1', method='gauss', offset=0.1, scale=0.1,
                  origin=0.0, label='feature_1')
compare_c.numeric('feature_2','feature_2', method='gauss', offset=0.1, scale=0.1,
                  origin=0.0, label='feature_2')
```

```
compare_c.numeric('feature_3','feature_3', method='gauss', offset=0.1, scale=0.1,
                  origin=0.0, label='feature_3')
compare_c.numeric('feature_4','feature_4', method='gauss', offset=0.1, scale=0.1,
                  origin=0.0, label='feature_4')
```

Where `numeric()` is abbreviation for:

```
classrecordlinkage.compare.Numeric(left_on, right_on, method='linear', offset=0.0,
                                   scale=1.0, origin=0.0, missing_value=0.0, label=None)
```

The *method* parameter defines the metric used for numeric values comparison. The chosen method to clean our data is the `gauss` one, that seemed the most reasonable for our case, as the score given to a couple of similar attributes rapidly decreases after a certain threshold is exceeded. The *offset* and *scale* parameters, used by the similarity measure, are set after a trial-and-error process aimed at improving the results.

After the matches are found, a choice of what to do with the duplicated values had to be made. Initially, we employed a strategy consisting in replacing the two duplicated rows with a row containing as values the average of the values of the two rows, but noticed that this caused a drop in the ML algorithms performance, and so we decided to shift to a simple dropping strategy, in which the second tuple of each couple of duplicated values is dropped.

The function performing data fusion is the following:

```
solve_duplicates_with_dropping(df, labels)
```

3.6. Cleaned data analysis

The cleaned data of the 10 experiments for the duplicated values issue and the 10 experiments for the variable types issue is again utilized to train the same set of classification ML algorithms. Results in terms of performance, distance between trainig and test sets, and speed, are produced and plotted to be analyzed, in comparison with the results produced by polluted data.

4 Results

4.1. Duplicated data

The plots of the results for what concerns duplicated data are shown at the end of the document, from figure 4.1 to figure 4.20.

4.1.1. Results obtained

Looking at the various plots produced by the model trained after the pollution and after the successive cleaning, the linear growing trend of the model's performance when trained with the polluted datasets, compared to the training with the cleaned ones, is evident. This trend is the opposite of what we expected at the beginning of the project, because we thought that the performance of the model would decrease with the introduction of duplicated values, only to grow after the cleaning phase.
Instead, the more the dataset is polluted, the more it seems that the performance of the model is increasing, despite the combination of parameters used to increase the distance between a value and its generated duplicate.

We tried to increase the distance, in terms of difference, between the original dataset and the polluted ones by rising the parameters of pollution, incrementing the number of rounded decimals and incrementing the variance of the Gaussian function used for noising, but the results obtained were very similar to the experiments with the initial parameters.

4.1.2. Performance consideration

These results can be justified considering a well known topic of deep learning which is "Data augmentation". Data augmentation is a technique commonly used in machine learning and deep learning to artificially increase the diversity of a training dataset by applying various transformations to the existing data. The goal is to improve the model's generalization and robustness by exposing it to a wider range of variations in the input data.

This is clearly what we have done: by polluting our dataset with duplicated values we have increased the number of its samples by applying some linear transformations (rounding and noising) to the original values, producing some new samples that consist of new information from which the model can learn.

Instead, an analysis of the performance resulting from training with the cleaned datasets reveals a constant trend, which indicates that the performance remains unchanged through the experiment, that initially differed with relation to the percentage of polluted data, and now that they are cleaned have the same shape. This is coherent because of the fact that, upon cleaning the datasets, they all revert to the same number of rows as the original dataset, as if augmentation was never applied.

4.1.3. Distance consideration

Taking into consideration the distance parameter, we can see that the data pollution, which acts as data augmentation, allows the model to generalize better on unseen data.

Indeed the distance between the training and the test set performances is decreasing in all the experiments conduced with the polluted datasets, which is another signal of the improvement of the model, since it indicates that the model overfits less when trained with augmented samples, meaning that the test prediction performances

are more similar to the training ones.

Instead, considering the distance parameter produced by the training with the cleaned dataset, we can notice that it remains unchanged through the experiments, that had each a different pollution composition and now have the same shape.
This is again coherent, because when the datasets are cleaned they all return to have the same number of rows as the original dataset.

4.1.4. Speed consideration

The speed (time to train the model) parameter shows the diversity of time taken by the different algorithms considered for the training and shows obviously a slight increasing trend the more records are added to the dataset due to pollution.

4.2. Variable types

The plots of the results for what concerns variable type issues are shown at the end of the document, in figure 4.21.

4.2.1. Performance consideration

We can see how the performance decreases with the increase of the number of distinct values contained in the added categorical column from experiment 1 to experiment 4.
Experiment 5 adds 2 new categorical features instead of 1, but has a lower number of possible distinct values that can appear in the new columns with respect to the preceding experiment, and thus the performance goes back to being high. Experiments 6, 7 and 8 also consist in adding two new categorical features, but with an increasing number of possible values to be found in these columns, hence the down-going trend of the performance graph.
Experiment 9 adds 3 new features, for a total of possible values that is lower than the total of experiment 8, and we notice that consequently the performance has a small improvement.
Finally, Experiment 10 adds 4 new features and the performance drops again, as expected.

The performance of the model is lower for datasets that contain more distinct categorical values, because the classification algorithms need to consider more elements when deciding what class to assign to each data point. The more complexity a dataset embeds, the more data is needed to properly train a model to perform classification on it. Since the dataset number of rows is always the same in all the experiments, but the complexity of the data is heterogeneous, diversity is found in the performance of the ML algorithms when trained with the differently polluted dataset.

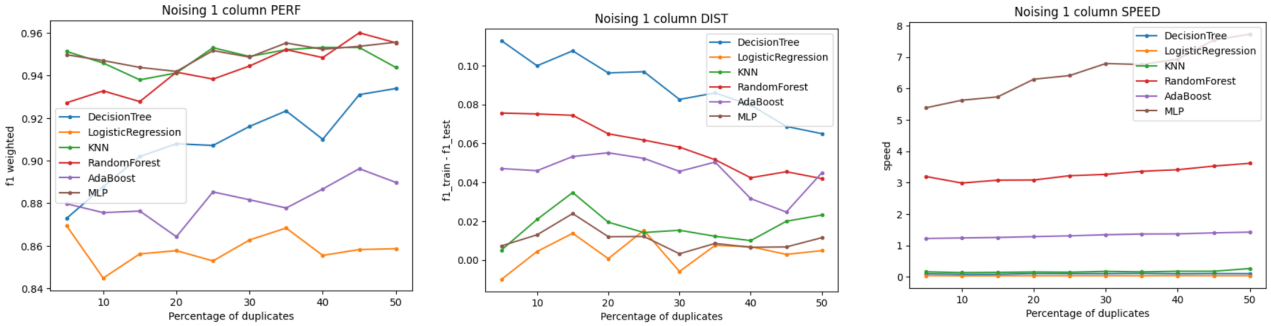
4.2.2. Distance consideration

The distance has a behavior that is coherently the opposite of the behavior of the performance, increasing when the second decreases and vice versa. The reason is more or less the same as that of the duplicates. When the distance increases, the model loses generalization and hence there is more distance between training and test predictions.

4.2.3. Speed consideration

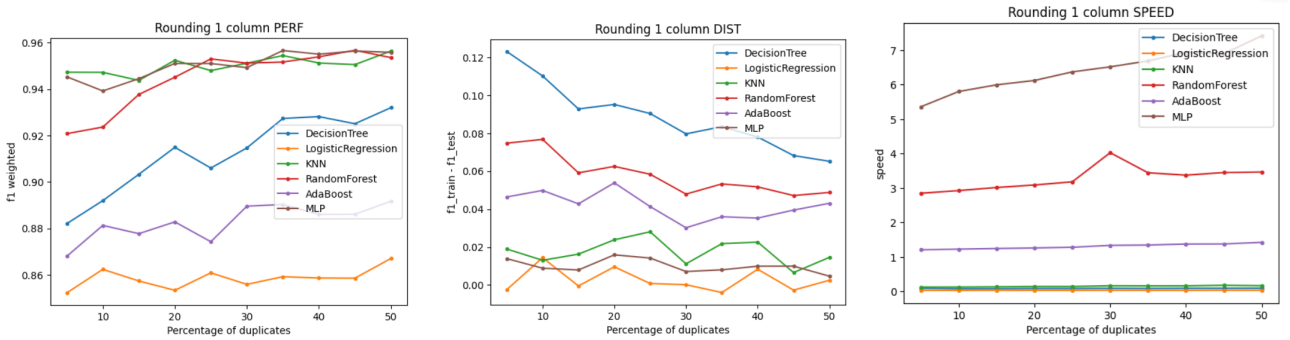
The speed (time to train the model) increases with the increase of the total number of values that can be found in the added features, and so with the complexity that the ML algorithms can find in creating classes for the classification.

4.3. Plots



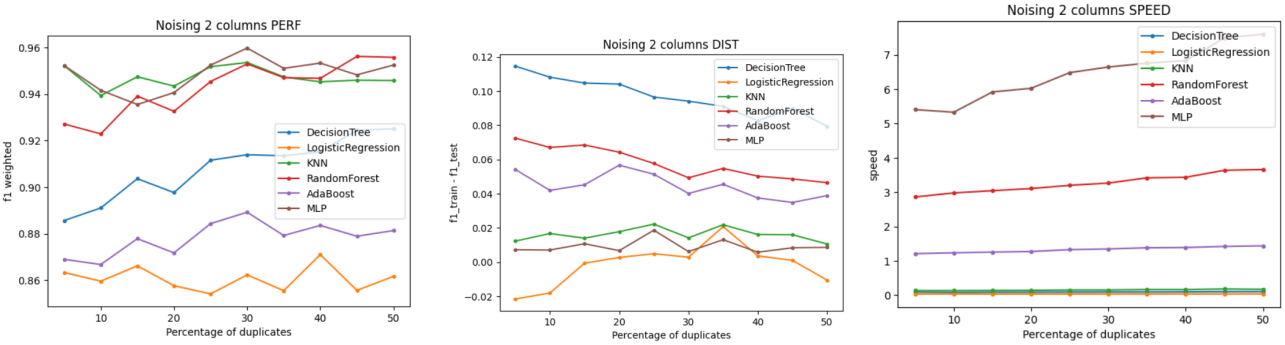
(a) Performance (b) Distance (c) Speed

Figure 4.1: Results of experiment 1 with data polluted with duplicated values.



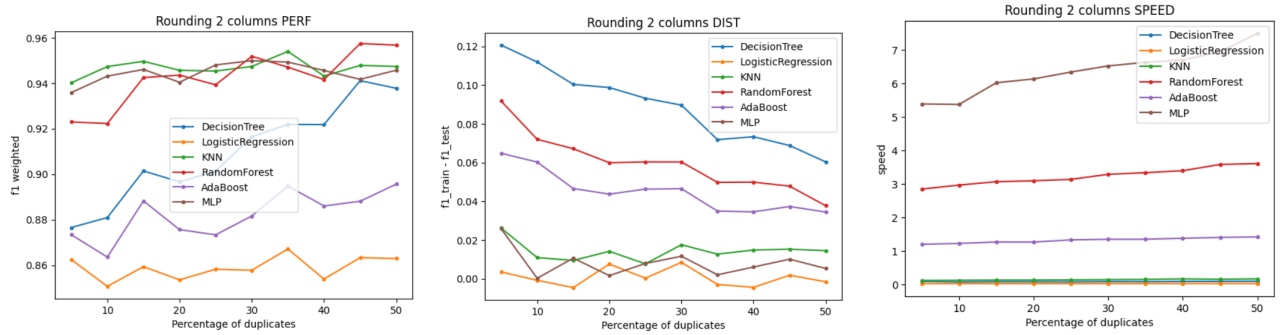
(a) Performance (b) Distance (c) Speed

Figure 4.2: Results of experiment 2 with data polluted with duplicated values.



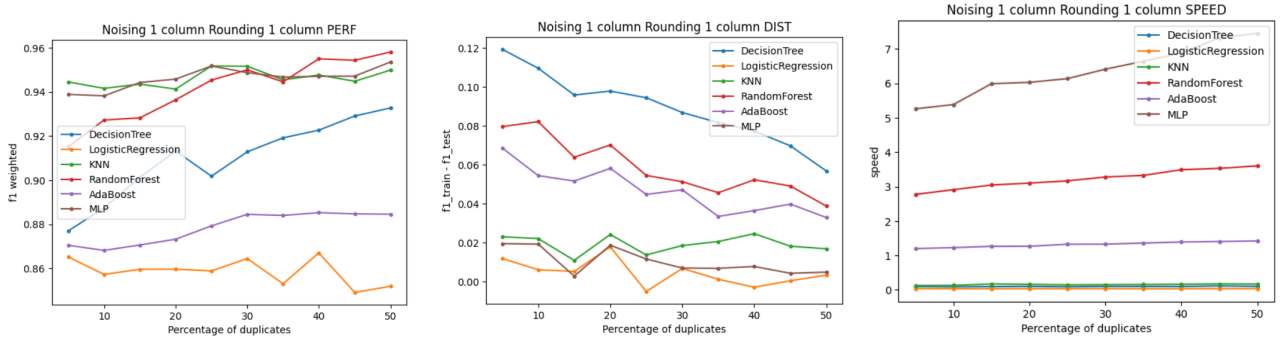
(a) Performance (b) Distance (c) Speed

Figure 4.3: Results of experiment 3 with data polluted with duplicated values.



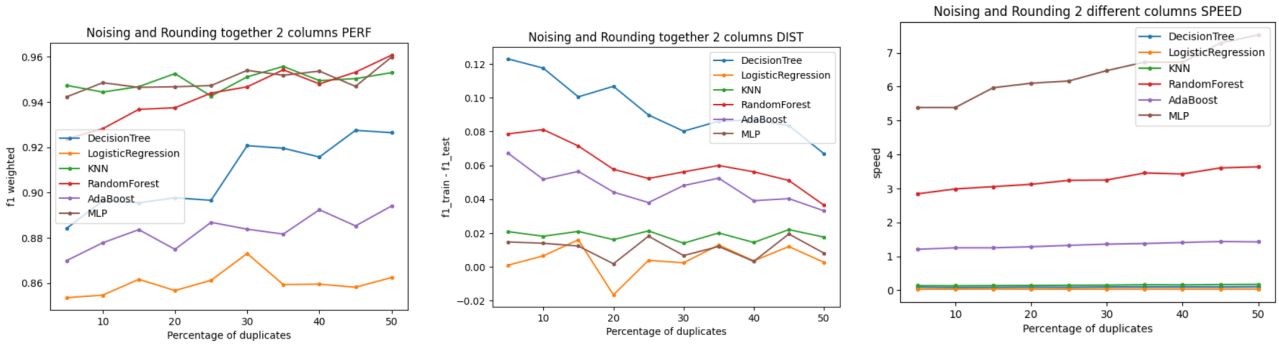
(a) Performance (b) Distance (c) Speed

Figure 4.4: Results of experiment 4 with data polluted with duplicated values.



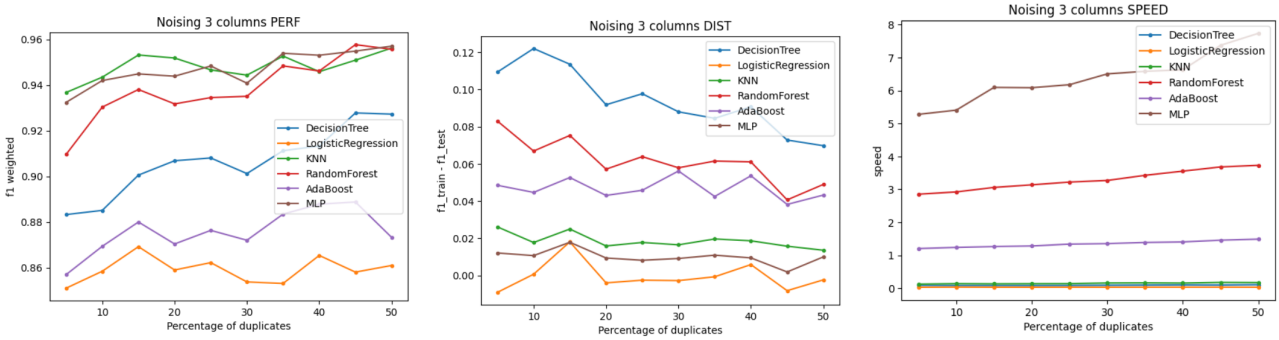
(a) Performance (b) Distance (c) Speed

Figure 4.5: Results of experiment 5 with data polluted with duplicated values.



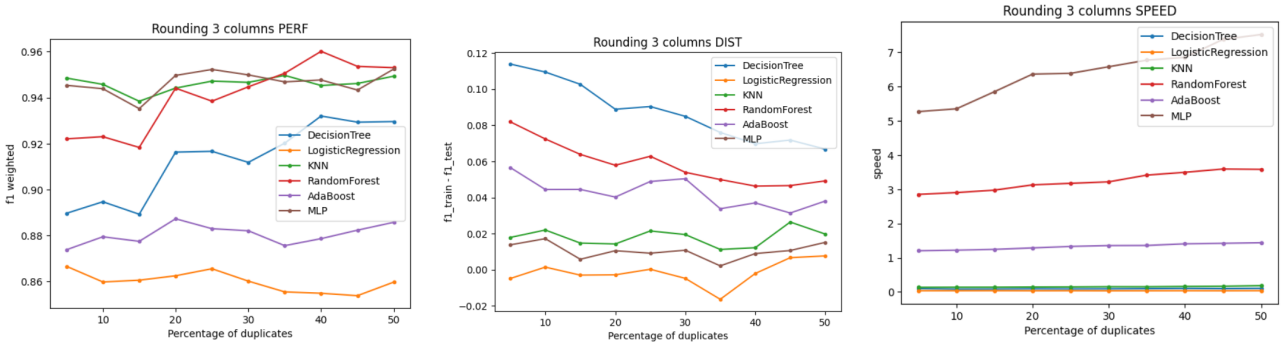
(a) Performance (b) Distance (c) Speed

Figure 4.6: Results of experiment 6 with data polluted with duplicated values.



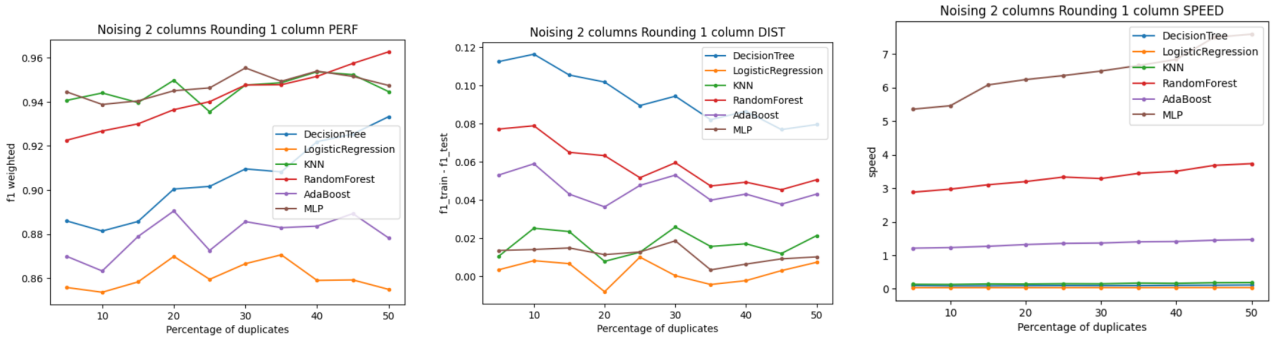
(a) Performance (b) Distance (c) Speed

Figure 4.7: Results of experiment 7 with data polluted with duplicated values.



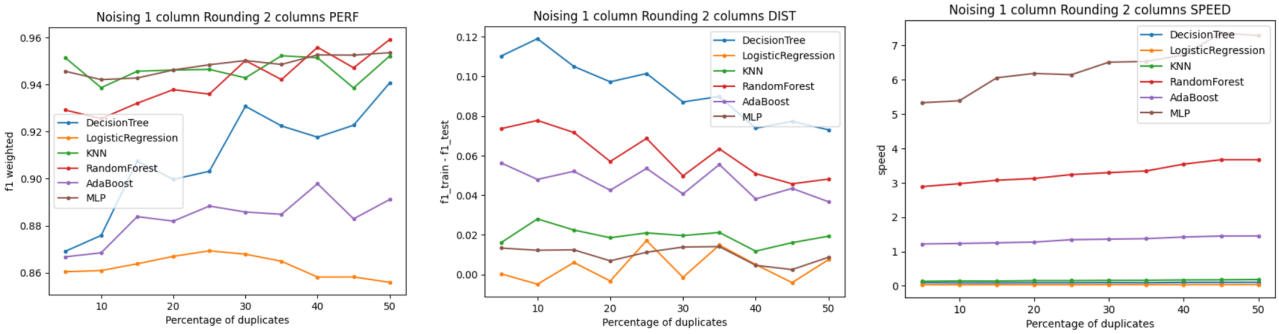
(a) Performance (b) Distance (c) Speed

Figure 4.8: Results of experiment 8 with data polluted with duplicated values.



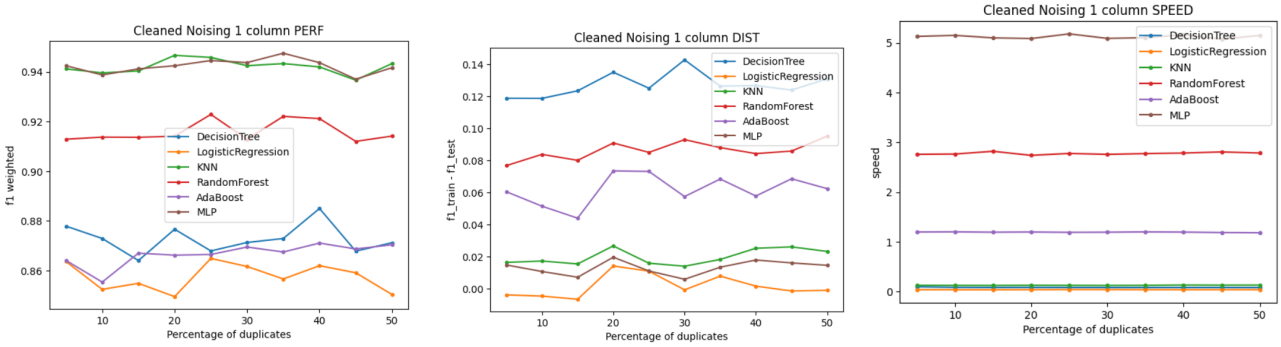
(a) Performance (b) Distance (c) Speed

Figure 4.9: Results of experiment 9 with data polluted with duplicated values.



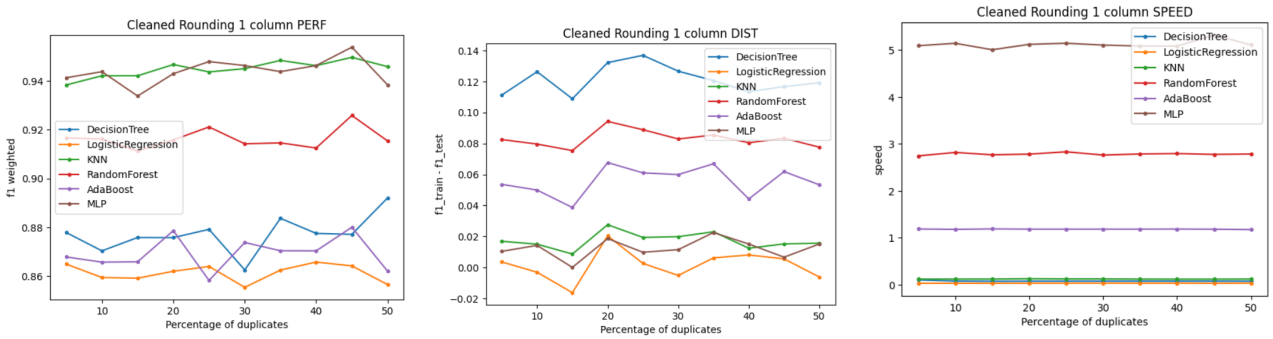
(a) Performance (b) Distance (c) Speed

Figure 4.10: Results of experiment 10 with data polluted with duplicated values.



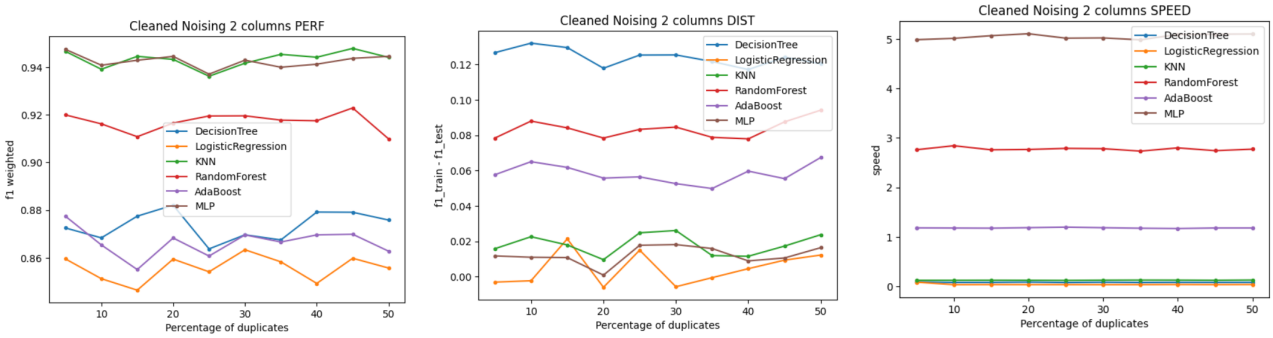
(a) Performance (b) Distance (c) Speed

Figure 4.11: Results of experiment 1 with cleaned data from duplicated values.



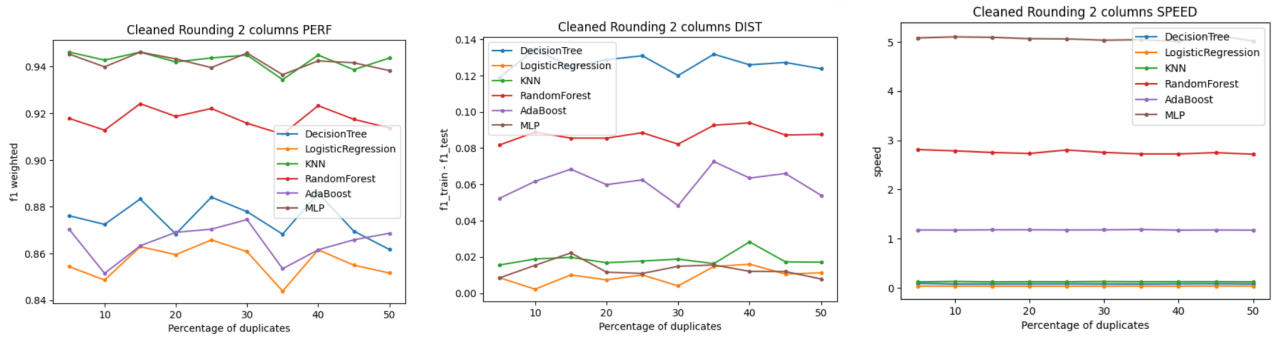
(a) Performance (b) Distance (c) Speed

Figure 4.12: Results of experiment 2 with cleaned data from duplicated values.



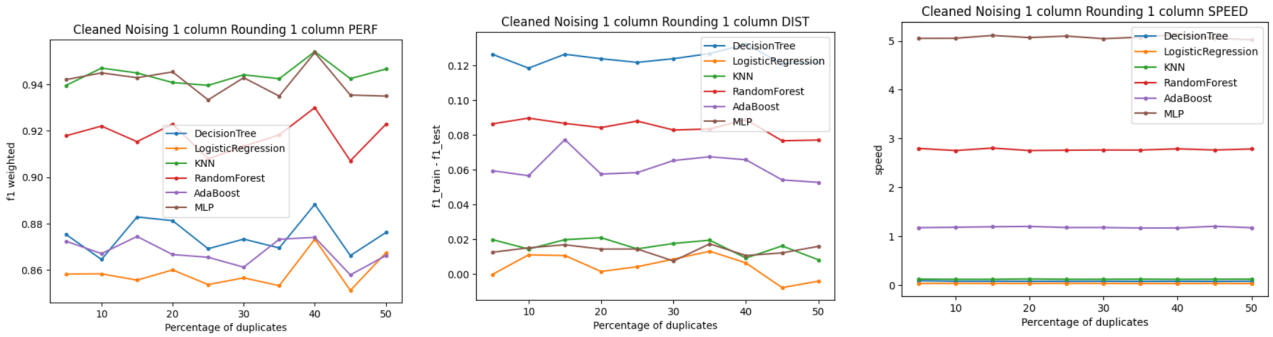
(a) Performance (b) Distance (c) Speed

Figure 4.13: Results of experiment 3 with cleaned data from duplicated values.



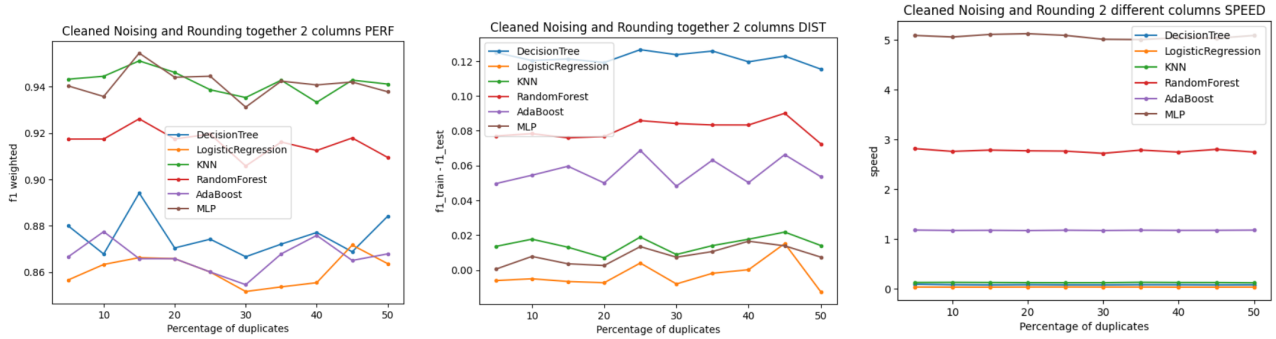
(a) Performance (b) Distance (c) Speed

Figure 4.14: Results of experiment 4 with cleaned data from duplicated values.



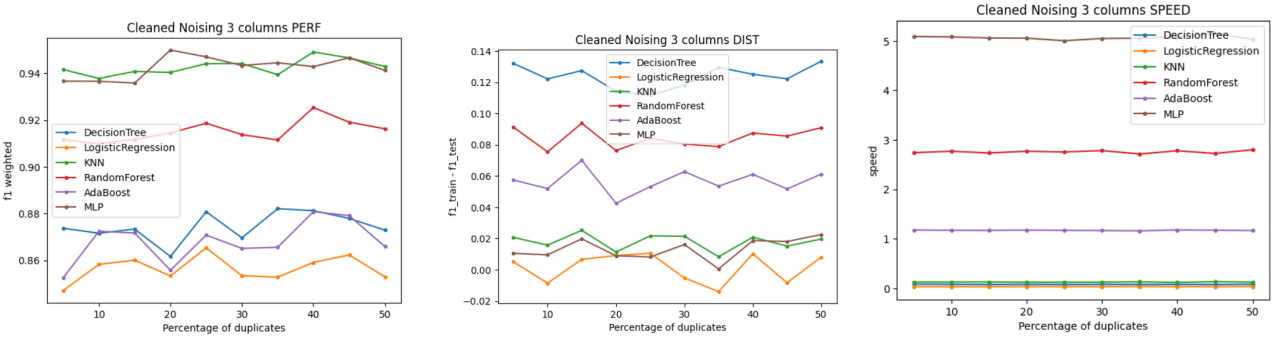
(a) Performance (b) Distance (c) Speed

Figure 4.15: Results of experiment 5 with cleaned data from duplicated values.



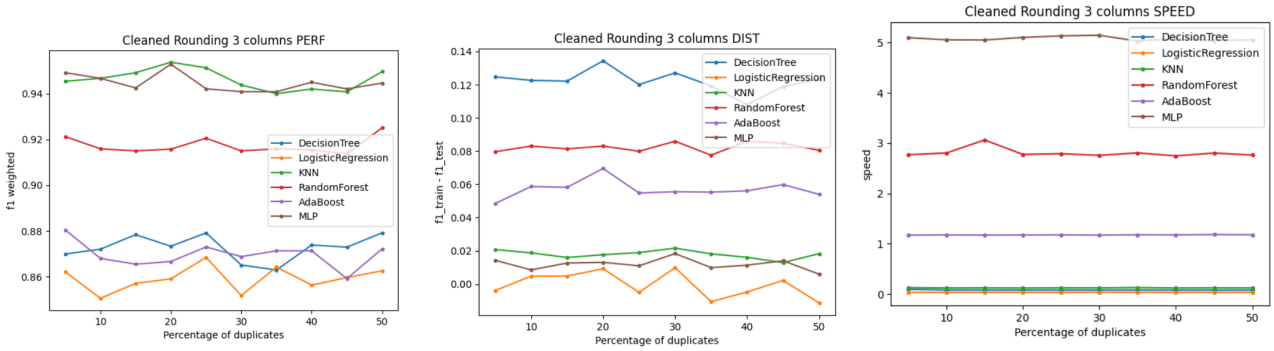
(a) Performance (b) Distance (c) Speed

Figure 4.16: Results of experiment 6 with cleaned data from duplicated values.



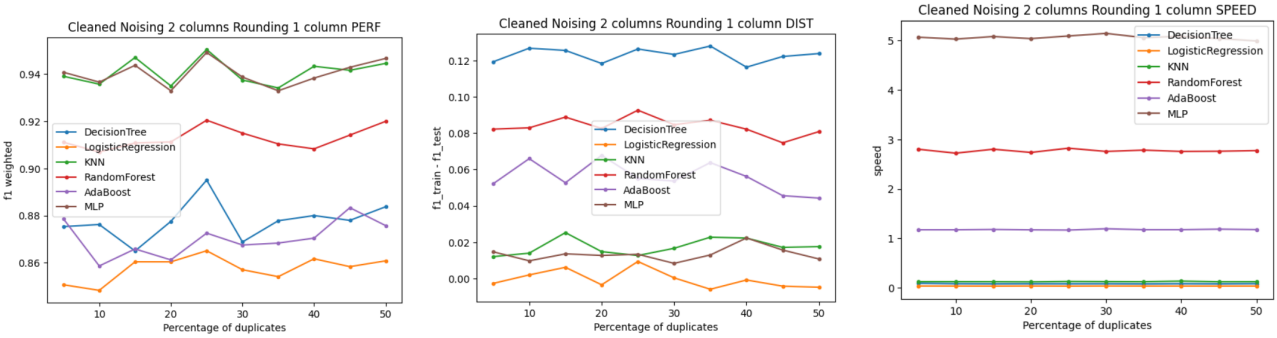
(a) Performance (b) Distance (c) Speed

Figure 4.17: Results of experiment 7 with cleaned data from duplicated values.



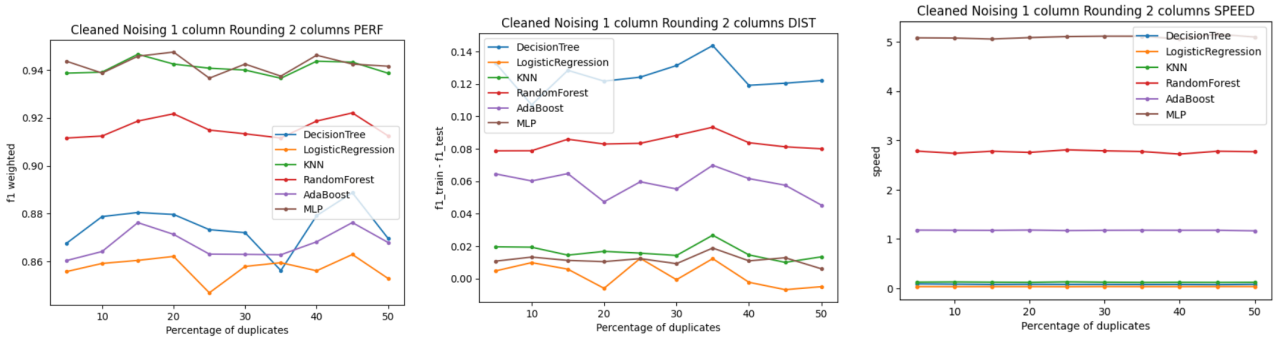
(a) Performance (b) Distance (c) Speed

Figure 4.18: Results of experiment 8 with cleaned data from duplicated values.



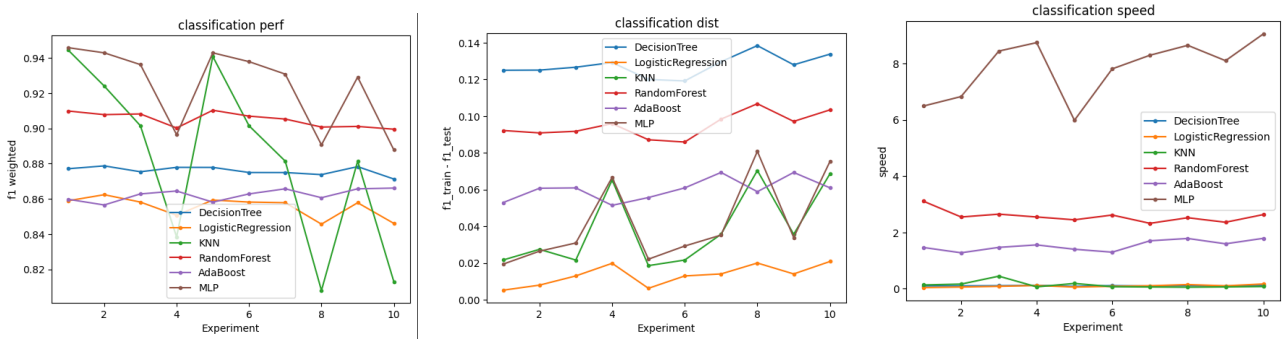
(a) Performance (b) Distance (c) Speed

Figure 4.19: Results of experiment 9 with cleaned data from duplicated values.



(a) Performance (b) Distance (c) Speed

Figure 4.20: Results of experiment 10 with cleaned data from duplicated values.



(a) Performance (b) Distance (c) Speed

Figure 4.21: Results of the 10 experiments with **variable types** issues.