CS5011: P2 - Machine Learning

190018035

March 13th 2024

Contents

1 Introduction

For this practical, I was tasked with implementing and evaluating different machine learning models on the Pump It Up: Data Mining the Water Table [1] dataset. The checklist below details my achievements in the practical for each of the specified parts.

1.1 Project Achievements

- Part 1: Attempted and Fully Working
- Part 2: Attempted and Fully Working

1.2 Usage Instructions

To run the script, navigate to the root directory and run the following command:

```
python3 part1.py <train-input-file> <train-labels-file> <test-input-file> <
    numerical-preprocessing> <categorical-preprocessing> <model-type> <test-
    prediction-output-file>
```

The values for each of the arguments are equivalent to that described in the assignment specification. However, one difference is that the `<numerical-preprocessing>` and `<categorical-preprocessing>` arguments now take in a `Manual` option, which allows features to be preprocessed based on the encoding that are most suitable for the feature.

2 Part 1

In this section, I will detail the various preprocessing steps I took to prepare the data for the machine learning models.

2.1 Preprocessing Steps

Prior to implementing the machine learning models, I performed a number of preprocessing steps to clean and prepare the data for the models. These steps were guided by a deep exploration of the dataset, where I analyzed the distribution of the features, the relationships between the features, and other important characteristics of the dataset. In conjuction with the steps outlined below, I will also discuss my findings from the exploration that lead to the decisions I made.

2.1.1 Cleaning the Data

The following steps were taken to clean the data:

Removing Irrelevant Features The first step I took was to remove any features that were irrelevant to the prediction task. These features either did not directly contribute to the status of the water pumps or contained too many missing values. The features that were removed include:

- `id` The identifier for the water pump is not relevant to the prediction task
- `wpt_name` The name of the water pump is unlikely to contribute to the prediction task
- `scheme_name` The name of the water scheme is unlikely to contribute to the prediction task
- `num_private` This feature contains mostly missing values, with over 99.24% of the values missing (defined as 0). Additionally, the feature does not have a clear definition, so it is unclear how it would contribute to the prediction task.
- `amount_tsh` Similarly, this feature contains mostly missing values, with over 70.1% of the values missing (also defined as 0).

Removing Single-Value Features I also removed any features that only contained a single value, as they do not contribute to the prediction task. There was only one feature that met this criteria, which was `recorded_by`.

Removing Redundant Features There were a number of features that were redundant, meaning they contained the same information as another feature. I removed these features to reduce the dimensionality of the dataset. There were a few considerations I made when deciding which of the features that are redundant to remove. The first consideration was the cardinality of the feature - having too many unique values would make the feature difficult to encode, whilst having too few unique values would make the feature less informative. The second consideration was the number of missing values in the feature - if a feature had a large number of missing values, it would be less informative. The features that were removed include:

- `payment` This feature contains the same information as `payment_type` with the same value counts, so I removed it and retained `payment_type`.
- `quantity` Similarly, this feature contains the same information as `quantity_group` with the same value counts, so I removed it and retained `quantity_group`.
- `extraction_type` This feaure was removed in favor of `extraction_type_group` and `extraction_type_class`. The `extraction_type_class` feature provides a more general classification of the extraction type. It contains only 7 unique values, which makes it easier to encode, and the smallest value count for a given category for this feature is 117. The `extraction_type_group` feature, on the other hand, contains 13 unique values, where the smallest value count for a given category is 98. Finally, the `extraction_type` feature contains 18 unique values, where the smallest value count for a given category is 2. From this information, we can see that the `extraction_type_class` feature is the most general, and the `extraction_type` feature is the most specific, thereby providing a balance between the general and specific information provided to the model.

- scheme_management This feature was removed in favor of management. These two features contain most of the same categories; however, the scheme_management feature contains 3878 missing/unknown values whereas management only contains 561 unknown values. This makes the management feature more informative and easier to encode.
- water_quality This feature was removed in favor of quality_group. These two features contain mostly the same information, where water_quality splits "salty" and "flouride" into two subcategories (introducing a "salty abandoned" and "flouride abandoned" subcategory). These subcategories contain comparatively few values, so I decided to remove the water_quality feature and retain the quality_group feature to reduce the dimensionality of the dataset.
- source This feature was removed in favor of source_type and source_class. The source_class feature is the most general, containing only 3 unique values, and the source_type feature sits in the middle, containing 7 unique values and containing very similar columns to source. The source feature contains 10 unique values. Since source_class provides the most general information, and since source contains the highest cardinality, I decided to remove the source feature and retain the other two so there is a balance between the general and specific information provided to the model.
- waterpoint_type_group This feature was removed in favor of waterpoint_type. The waterpoint_type splits up the "communal standpipe" category into "communal standpipe" and "communal standpipe multiple". However, since "communal standpipe multiple" only contains contains relatively few values (6103), I decided to remove the waterpoint_type_group feature and retain the waterpoint_type feature to reduce the dimensionality of the dataset.

Replacing Construction Year with Decades The construction_year feature contained a large number of missing values, with 34.8% of the values missing. To make the feature more informative, I decided to categorize the construction years into decades. This would allow the model to learn the relationship between the construction year and the status of the water pumps, without imputing values that might distort the underlying distribution of the data. The decades were defined from 1960s to the 2010s, with the missing values being replaced with the "Unknown" category.

Imputing Missing Values There were a number of numerical features that contained missing values, specifically longitude, gps_height, population and district_code. I wanted to avoid blindly imputing the missing values with the mean or median, as this could distort the underlying distribution of the data. Instead, I decided to impute the missing values using the mean of the feature grouped by the region feature. To ensure that this imputation method was valid, I ran a chi-squared test to check if the region feature was independent of the longitude, gps_height, population and district_code features. The results of the chi-squared test are shown in Table 1.

	Chi2	p-value	dof
longitude vs region	1187933.26	$3.26e^{-133}$	1150320
district_code vs region	119074.99	0	360
gps_height vs region	531651.17	0	48760
population vs region	457007.23	0	21200

Table 1: Results of the chi-squared test between numerical features with missing values and the region feature

The p-values for the chi-squared test were all less than 0.05, which indicates that the region feature is not independent of the above features. Therefore, I decided to impute the missing values using the mean of the feature grouped by the region feature, aside from district_code, where I used the median (to avoid floating point district codes). For categorical features, I replaced the missing values with the "Unknown" category, aside from the permit and public_meeting features, which I imputed using the mode of the feature due to the small number of missing values.

Fixing Formatting Issues When exploring the funder and installer features, I found that there were a large number of unique values, with some of the values being the same but formatted differently and consisting of spelling mistakes. For example, "tanzania government" and "tanzania*n* government" were considered as two different unique values, and so was "regwa company of egypt" and "regwa company of *egpty*". To address this, I first converted all the values to lowercase and then removed any leading/trailing whitespace. I then employed a fuzzy matching algorithm to group potentially duplicates based on a similarity threshold, which I set to 95%. This threshold was determined through manual experimentation and set to a fairly high value to ensure that the algorithm didn't group values that were not duplicates. This approach helped consolidate the variations and reduce the cardinality of the features. Table 2 shows the reduction in the number of unique values for each of the features after fixing the formatting issues.

Feature	Original Unique Values	Reduced Unique Values
`funder`	1896	1883
`installer`	2145	1905

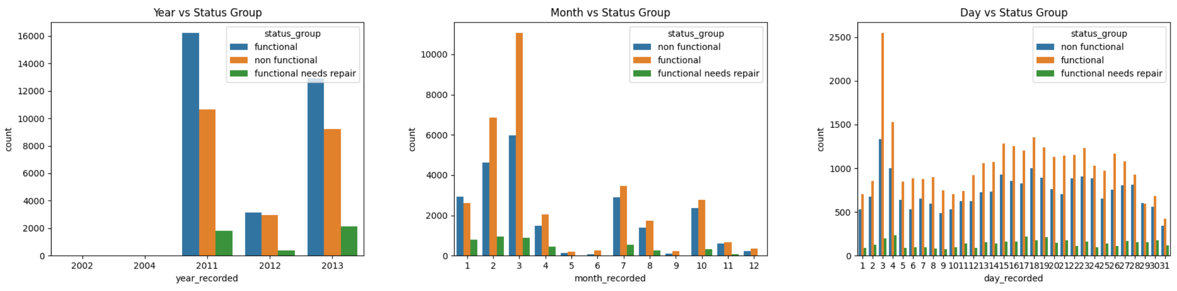
Table 2: Reduction in unique values for high cardinality features when fixing formatting issues

Limiting High Cardinality Features There were a few features in the dataset, specifically `funder`, `installer`, subvillage and `ward`, that contained a large number of unique values. Such high cardinality could pose challenges, as encoding these features could significantly increase the dimensionality of the dataset, potentially leading to issues like overfitting and increased computational demand. To address this, I set a threshold (95%) that identifies the most frequent categories that cumulatively account for 95% of the values in the feature. This threshold was determined through manual experimentation, but can be set through hyperparameter optimization. Through this strategy, I was able to reduce the number of dummy variables needed for encoding and makng the dataset size more manageable. The remaining categories were replaced with the "Other" category. Additionally, I also replaced the missing values with the "Unknown" category. Table 3 details the following reductions in the number of unique values for each of the features (after fixing formatting issues):

Feature	Original Unique Values	Reduced Unique Values
`funder`	1883	405
`installer`	1905	397
`subvillage`	19287	16336
`ward`	2092	1578

Table 3: Reduction in unique values for high cardinality features when applying the threshold of 95%

Converting Datetime Features The `date_recorded` feature was converted to a datetime object and then split into three separate features: `year`, `month`, and `day`. By doing so, I was able to extract the temporal information from the feature, which could be useful for the model to learn the relationship between the date the water pump was recorded and the functionality of the water pump. In Figure 1, we can see the distribution for each temporal feature. We can see that the year and month recorded display some seasonal trends, where certain months (e.g January, February, March) and certain years (e.g 2011, 2013) have a higher number of recorded water pumps. The day recorded, however, does not display any clear trends. When dropping the day recorded feature, however, the performance of the models did slightly decrease, so I decided to retain the feature.



(a) Distribution of the year recorded (b) Distribution of the month recorded (c) Distribution of the day recorded

Figure 1: Distribution of the date recorded with respect to the status of the water pump

2.1.2 Encoding the Data

In addition to the encoders specified in the assignment specification, I also implemented a `Manual` encoder, which allows features to be preprocessed based on the encoding that are most suitable for the feature. High-cardinality features were encoded using the Target Encoding method, and low-cardinality features were encoded using the One-Hot Encoding method. The only ordinal feature in the dataset, `decade`, was encoded using the Ordinal Encoding method. The numerical features that were encoded using the Standard Scaling method only included the `longitude`, `latitude`, `gps_height` and `population`.

2.2 Design Decisions

Before evaluating the performance of the models, I made a few design decisions to not only investigate its impact on the performance of the model, but also as a way to satisfy some of the data preprocessing requirements that each model has. These decisions are detailed below.

2.2.1 Encoder Parameters

There were two main parameters that I found to have an impact on the model's ability to run. For the One-Hot Encoding method, I set the `handle_unknown` parameter to "ignore" to avoid errors when the test set contains categories that were not present in the training set. I used a similar approach for the Target Encoding method, where I set the `handle_unknown` parameter to -1 for the same reason.

2.2.2 Model Parameters

I wanted to investigate a few key hyperparameters for the Random Forest and HistGradientBoostingClassifier models. The default settings provide a strong baseline for the models, but I wanted to see the impact of changing some of the default parameter settings that significantly influence their performance. The parameters I chose for each model, along with the reasons for choosing them, are detailed below.

Random Forest

- `n_estimators` This is the number of trees in the forest. Investigating this parameter is crucial as more trees can improve the model's accuracy and robustness, at the cost of eventually plateauing and increasing the computational demands. For this reason, I set the `n_estimator`'s parameter to 200 (100 higher than the default) to see if the model's performance would improve.
- `max_depth` This is the maximum depth that the tree can grow. Investigating this parameter is crucial as a deeper tree can capture more complex patterns within the data, but can also lead to overfitting if it is too deep. The default value for this parameter is "None", which means that the tree will grow until all leaves are pure. I set the `max_depth` parameter to 20 to see if limiting the depth can reduce the risk of overfitting while still allowing the tree to capture complex patterns.
- `max_features` This parameter defines the number of features to consider when deciding upon a split. A value with a low parameter might not capture enough information for a strong prediction, while too high a value might lead to overfitting. I set the `max_features` parameter to "log2" (different from the default "sqrt") to see if taking the log of the number of features provides a better balance between capturing enough information and avoiding overfitting than the square root of the number of features.

HistGradientBoostingClassifier

- `learning_rate` This parameter shrinks the contribution of each tree to control the rate of learning. A low learning rate can lead to a more robust model, but at the cost of increased computational demands. I set the `learning_rate` parameter to 0.05 (different from the default 0.1) to see if a lower learning rate can improve the model's robustness.
- `min_samples_leaf` This parameter is the minimum number of samples required to be at a leaf node. A higher value forces the model to make a split only if there are enough samples to populate the leaf, which deters the model from making splits that capture noise. I increased

the `min_samples_leaf` parameter to 100 (different from the default 20) to see if a higher values avoids the model from capturing noise.

2.3 Evaluation

To evaluate the performance of each model, I ran a 5-fold cross-validation on the training set. I experimented with different preprocessing encodings, including Standard Scaling (SS) and Passthrough (PT) for the numerical features, and One-Hot Encoding (OHE), Ordinal Encoding (Ord), and Target Encoding (Target) for the categorical features. The models where ran with the default hyper-parameters. The results of the experiments are shown in the Table 4 below.

	SS + OHE	SS + Ord	SS + Target	PT + OHE	PT + Ord	PT + Target
`RandomForestClassifier`	80.74%	80.79%	80.4%	80.2%	79.09%	79.85%
`LogisticRegression`	78.49%	62.24%	75.52%	78.37%	61.44%	75.53%
`GradientBoostingClassifier`	75.96%	75.16%	77.53%	75.71%	74.56%	74.56%
`HistGradientBoostingClassifier`	79.39%	79.12%	79.62%	78.67%	78.37%	78.89%
`MLPClassifier`	77.59%	53.36%	78.16%	77.54%	54.33%	77.7%

Table 4: Results of each model with different preprocessing encodings

2.3.1 General Observations

This section will cover some key findings from the experiments with each model on default hyper-parameters and different preprocessing encodings.

Random Forest Random Forest performed the best out of all the models, with the best accuracy of 80.79% using Standard Scaling and Ordinal Encoding. This could be attributed to the model's inherent robustness to feature scaling and the ability to handle categorical features. It is possible that ordinal encoding provides a more meaningful way to represent categorical features than other types of preprocessing without increasing the dimensionality too much, aligning with Random Forest's capacity to manage features with high cardinality. However, we do see that scaling the numerical features does have a significant impact, since ordinal encoding performed the worst out of all the combinations when using the passthrough encoding.

Logistic Regression Logistic Regression performed the best when using One-Hot Encoding. This could be because one hot encoding ensures linear independence for each feature, whereas ordinal encoding might introduce some form of ordinality that might not be present in the data and target encoding might introduce some form of leakage. For this same reason, this may be why the model performed the worst when using ordinal encoding, as there is a clear violation of the assumption of linear independence.

Gradient Boosting Classifier Gradient Boosting Classifier performed best when target encoding is applied. This could be due to the model's gradient boosting mechanism. Target encoding could help reduce overfitting by smoothing out the effect of the rare categories, which is crucial for gradient boosting models. Additionally, since the model is tree-based, it can inherently handle categorical features, which aligns with the target encoding method.

HistGradientBoostingClassifier Across all models, HistGradientBoostingClassifier performed second best. There is a significant improvement in the model's performance when using standard scaling, as the accuracies are consistently higher than when using passthrough. This could be due to the model's ability to handle different scales in numerical features. By utilizing Standard Scaling, this might help in the model's ability to learn the relationships between the features and the target variable.

MLP Classifier One notable observation is that MLPClassifier performs the worst when using ordinal encoding. This could be due to the model's reliance on gradient-based optimization which can be sensitive to feature scaling. Since ordinal encoding exacts an ordinality that might not be present in the data, this could potentially confuse the model and lead to poor performance.

2.3.2 Random Forest with Parameter Setting

Below are the results of the Random Forest model with different hyper-parameters using Standard Scaling and Ordinal Encoding. Each parameter was tested on its own to see the impact on the model's default performance. The results are shown in Table 5.

Default	n_esimators	max_depth	max_features
80.79%	80.81%	80.82%	80.80%

Table 5: Results of Random Forest with different hyper-parameters using Standard Scaling and OrdinalEncoder

From these results, we can make a few observations. Firstly, by increasing the number of trees, the model's performance slightly improves. This is expected, as more trees can improve the model's accuracy and robustness. Limiting the depth of the tree also slightly improves the model's performance, signalling that the default tree depth might be too deep and is leading to overfitting. Finally, taking the log of the number of features as opposed to the default method of square rooting the number of features also slightly improves the model's performance. This could be due to the log leading to a more balanced consideration of feature importance across the trees, improving the generalization of the model.

2.3.3 HistGradientBoostingClassifier with Parameter Setting

Below are the results of the HistGradientBoostingClassifier model with different hyper-parameters using Standard Scaling and TargetEncoder, which is shown in Table 6.

Default	learning_rate	min_samples_leaf
79.62%	79.24%	79.64%

Table 6: Results of HistGradientBoostingClassifier with different hyper-parameters using Standard Scaling and TargetEncoder

One notable observation from these results is lowering the learning rate to 0.05 from 0.1 actually decreases the model's performance. While this is unexpected, one possible explanation is that the learning rate is too low, causing the model to learn more slowly and not converge to the best solution. A future experiment could be to increase the number of iterations to see if the model's performance improves.

3 Part 2

For this part of the practical, I utilized Optuna [2], an automated hyper-parameter optimization (HPO) tool to find the best hyper-parameters for each of the models. In this section, I will detail my HPO process and analyze the results of the experiments.

3.1 Hyper-Parameter Optimization

I decided to use Optuna to perform the hyper-parameter optimization for the Random Forest and Logistic Regression models. I chose Random Forest as from the previous part, it was the best performing model, and Logistic Regression as it is a simple model that can be used as a benchmark. I also wanted to see how HPO differs for different families of models. The hyper-parameters that I optimized for each of the models are detailed below.

3.1.1 Experimental Setup

To conduct HPO, I first defined the search space for each of the models. For each model, I did some research to get a preliminary understanding of what the most important hyper-parameters are, and then defined the search space based on this information. For Random Forest, the following hyper-parameters were identified as the most influential [3]:

- `n_estimators` The number of trees in the forest

- `max_depth` The maximum depth of the tree
- `min_samples_split` The minimum number of samples required to split an internal node
- `min_samples_leaf` The minimum number of samples required to be at a leaf node
- `max_features` The number of features to consider when deciding upon a split

Similarly, for Logistic Regression, the following hyper-parameters were identified as the most influential [4]:

- `C` Inverse of regularization strength
- `penalty` The norm used in the penalization
- `solver` The algorithm to use in the optimization problem
- `class_weight` Weights associated with classes
- `tol` Maximum number of iterations taken for the solvers to converge
- `max_iter` Maximum number of iterations taken for the solvers to converge

Along with the model parameter search space, I also decided to include the preprocessing encodings as part of the search space. While the results in the previous part showed the optimal encodings for both these models, I wanted to ensure that the HPO process could find the best encodings for the models. Additionally, I wanted to see if having the `Manual` encoding for both numerical and categorical features as part of the search space would lead to better results. Stratifed 5-fold cross-validation was used to evaluate the performance of each model, and the best hyper-parameters were selected based on the highest accuracy. Values in the search space were modified appropriately with experimentation to ensure that the search space was not too large, which could lead to long runtimes, or too small, which could lead to suboptimal results. Optuna then used the TPE (Tree-structured Parzen Estimator) algorithm to sample the search space and find the best hyper-parameters for each model [2]. Once the best hyper-parameters were found, the models were trained on the test set using the best hyper-parameters to evaluate their performance.

3.2 Evaluation

The results of the hyper-parameter optimization for each of the models are shown in Table 7. The best hyper-parameters for each model are shown in the table, along with the accuracy of the model using the best hyper-parameters.

	Default Accuracy	Tuned Accuracy
`RandomForestClassifier`	80.79%	79.64%
`LogisticRegression`	78.13%	78.44%

3.2.1 Random Forest

When running the trials, I found that the most important hyper-parameters for the Random Forest model were `cat_encoder`, `max_depth`, and `max_features`, as shown in Figure 2. These parameters were the most influential in determining the performance of the model.

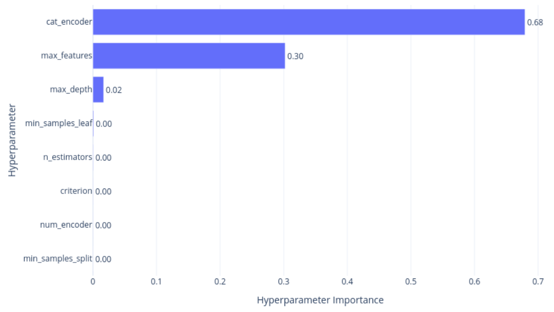
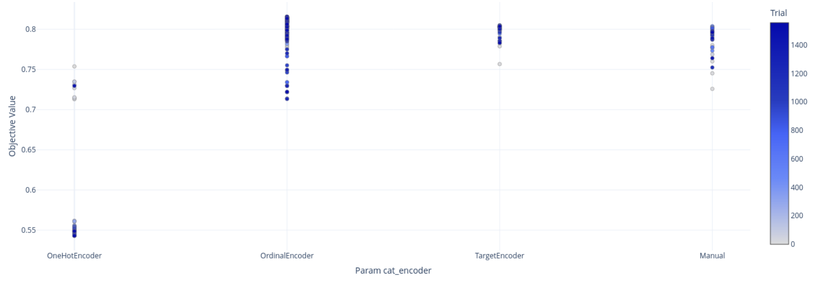


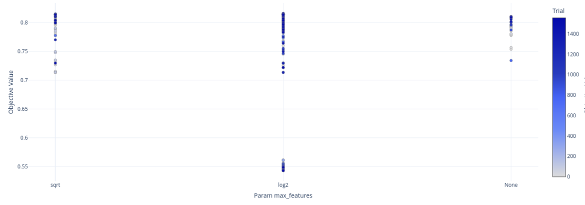
Figure 2: Feature Importance for Random Forest Model

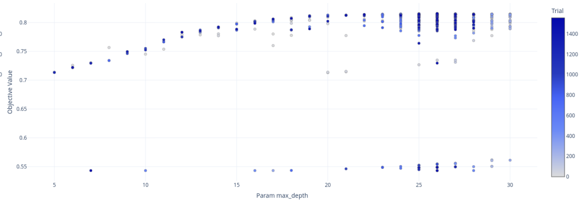
Interestingly, the categorical encoder plays a significant role in the model's performance, making up up to 68% of the total influence on predicting the target feature. As mentioned previously, this could be due to the model's inherent ability to handle categorical features, and the choice of encoding could significantly impact the model's performance. The maximum features parameter also plays a significant role in the model's performance, as this determines the number of features to consider when deciding upon a split. Finally, the maximum depth parameter plays a moderate role in the model's performance, as this determines the maximum depth of the tree. This aligns with the results from the previous part, where experimenting with the maximum features and maximum depth parameters led to a slight improvement in the model's performance.

We can investigate these important hyper-parameters by observing how each category of the parameters affects the accuracy of the model.



(a) Slice plot for the categorical encoder





(b) Slice plot for the maximum features

(c) Slice plot for the maximum depth

Figure 3: Important features vs trial performance for Random Forest Model

One notable observation from the categorical encoder slice plot is that the model performs very poorly when using the One-Hot Encoder. This could be due to the model's inability to handle the large number of dummy variables that are created when using One-Hot Encoding. It is also interesting that despite the Manual setting encoding each feature based on the encoding that is most suitable for the feature, the model's performance is still not as good as when using the Ordinal Encoder. This speaks to the model's ability to handle ordinal features. The slice plot of the maximum features parameter shows that a large amount of trials preferred a value of ”log2” for the maximum features parameter. This could be due to the log of the number of features providing a better balance between capturing enough information and avoiding overfitting than the square root of the number of features. Finally, the slice plot of the maximum depth parameter shows that the number of trials, along with the accuracy, increases as maximum depth increases up until around a value of 28, suggesting that a value higher than 28 may lead to overfitting.

3.2.2 Logistic Regression

When running the trials, I found that the most important hyper-parameters for the Logistic Regression model were `cat_encoder`, `solver`, and `class_weight` and `tol`, as shown in Figure 4.

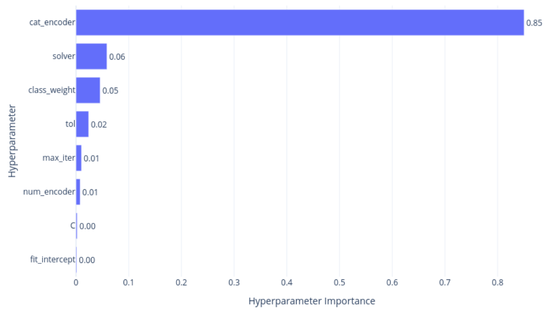


Figure 4: Feature Importance for Logistic Regression Model

As with the Random Forest model, the categorical encoder plays a significant role in the model's performance, this time making up up to 85% of the total influence on predicting the target feature. The other important features, solver, class weight and tol, all play a moderate role in the model's performance. This aligns with the results from the previous part, where the choice of encoding (specifically using Ordinal Encoder) played a significant role in the model's performance.

We can investigate these important hyper-parameters by observing how each category of the parameters affects the accuracy of the model.

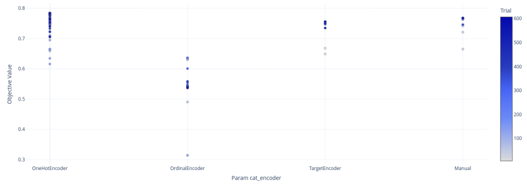


Figure 5: Slice plot for the categorical encoder

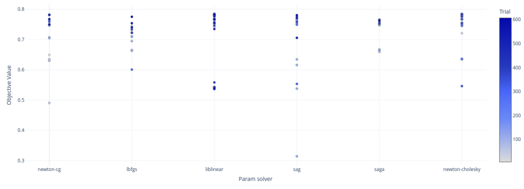


Figure 6: Slice plot for the solver

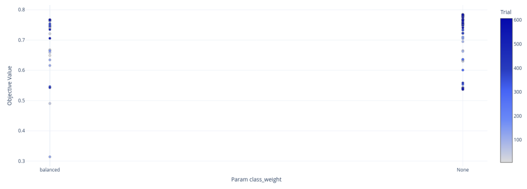


Figure 7: Slice plot for the class weight

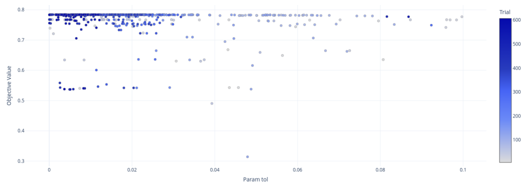


Figure 8: Slice plot for the tolerance

Figure 9: Overall caption for all four figures.

The slice plot of the categorical encoder confirms our results from the previous part: logistic regression performs best when using One-Hot Encoding and performs worst when using Ordinal Encoder, reiterating the model's reliance on linear independence for each feature. The slice plot of the solver parameter shows that the model performs best when using the "liblinear" solver, though this is not as clear since the solver does not play as significant a role in the model's performance as the categorical encoder. One last notable observation is that the slice plot of the tolerance seems to show that the model prefers a smaller tolerance, indicative of the model benefiting from a more thorough optimization process.

4 Conclusion

In this practical, I implemented and evaluated different machine learning models on the Pump It Up: Data Mining the Water Table dataset. I performed a number of preprocessing steps to clean and prepare the data for the models, and made a few design decisions to investigate its impact on the performance of the model. I also utilized Optuna to perform the hyper-parameter optimization for the Random Forest and Logistic Regression models. When submitting the predictions to the competition, I achieved an accuracy of 82.36% on the test set using the tuned RandomForestClassifier, which placed me at #823 on the leaderboard.

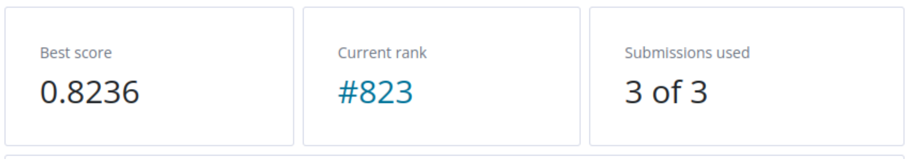

Best score
0.8236
Current rank
#823
Submissions used
3 of 3

Figure 10: Test set accuracy and leaderboard position

References

[1] URL: https://www.drivendata.org/competitions/7/pump-it-up-data-mining-the-water-table/page/23/.

[2] Takuya Akiba et al. "Optuna: A Next-generation Hyperparameter Optimization Framework". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.

[3] James Thorn. *Random Forest: Hyperparameters and how to fine-tune them.* Sept. 2021. URL: https://towardsdatascience.com/random-forest-hyperparameters-and-how-to-fine-tune-them-17aee785ee0d.

[4] Jason Brownlee. *Tune hyperparameters for Classification Machine Learning Algorithms.* Aug. 2020. URL: https://machinelearningmastery.com/hyperparameters-for-classification-machine-learning-algorithms/.