Report on Hackathon code Implementation

Antala Aviraj (*CS24MTECH14011*)

November 11, 2024

This report provides a clear, step-by-step guide for building a machine learning model to classify land use patterns. The process involves several key steps: preparing the data, handling any missing values, encoding labels, balancing the dataset, and training different machine learning models. Each step is designed to improve the model's F1 score and reliability.

The goal is to create a classification model that can accurately identify and categorize different land use patterns. This approach ensures that the data is well-prepared and that the model performs as accurately as possible, resulting in a practical and effective classification tool.

1] Data Loading

The first step is to load the dataset from a CSV file. Then, we split the dataset into two parts: the features and the target variable. The target variable is what we want the model to predict, while the features are the input data used to make predictions. Separating these ensures that we can work with the input data and the target outcomes independently during training.

2] Train-Test Split

When we split a dataset into training and testing parts, we usually use 80% of the data for training and 20% for testing. This means the model can learn from most of the data, while the remaining data (the test(validaton set) is used to evaluate how well the model performs on new, unseen examples. Testing on this separate set helps us measure how well the model generalizes, or applies what it learned, to data it hasn't seen before.

3] Handling Missing Values

Handling missing values is an important part of preparing the data. First, we calculate and visualize the percentage of missing data for each feature. Any features with more than 50% missing values are removed to avoid adding noise to the model. For the remaining columns, we fill in missing values using different methods based on the type of data:

- **Mean Imputation**: For numerical data, missing values are replaced with the average of the available values.
- **Median Imputation**: For skewed numerical data, missing values are replaced with the median value.
- **Mode Imputation**: For categorical data, missing values are filled in with the most common value in the column.But in our case only target is categorical data.

These methods help make sure missing values don't harm the model's accuracy and keep the data as reliable as possible.

4] Label Encoding

In many machine learning algorithms, the input data needs to be numerical. This means that if the target variable is categorical, it must be converted into numbers. Label encoding is the process used to do this, where each unique category is given a corresponding integer. This allows the model to treat the categories as numbers, which is essential for algorithms that can't handle categorical data directly.

5] UID Removal

This datasets include columns with unique identifiers (UIDs) like user IDs. These columns don't provide any useful information for predicting the target variable, so they are often removed from the feature set. By removing these columns, we make the model simpler and reduce the risk of overfitting, as the model might memorize the data based on these unique identifiers.

6] Model Training and Evaluation

Once the dataset is preprocessed and balanced, multiple machine learning classifiers are trained and evaluated. The classifiers used in this pipeline include a variety of models, such as:

- **Random Forest**
- **Support Vector Machine (SVM)**
- **Logistic Regression**
- **K-Nearest Neighbors (KNN)**
- **Naive Bayes**
- **Decision Tree**

- **Gradient Boosting**
- **HistGradientBoosting**
- **XGBoost**
- **LightGBM**

These classifiers are evaluated using k-fold cross-validation, which divides the data into k subsets (or folds). The model is trained on $k-1$ folds and tested on the remaining fold. This process is repeated for each fold, ensuring that each data point is used for both training and testing. The model's performance is assessed using the F1 Macro score, which is particularly useful for imbalanced datasets, as it considers both precision and recall.

Data Preprocessing and Contribution

In the initial phase of our competiton, we focused on understanding the data and determining the most effective preprocessing techniques. On the first day, we removed columns with more than 50% missing values to reduce data sparsity. However, after observing limited improvement in the F1 score, we adjusted the threshold to 70% the following day. This change allowed us to retain more features, enhancing the dataset's robustness. Additionally, with the 70% threshold, we identified several columns that required imputation, using mean and median values to fill missing data, thereby preserving the integrity of the dataset.

Experiments and observations with Different Models

To evaluate model performance, we initially experimented with various algorithms, including Gradient Boosting, Random Forest, and K-Nearest Neighbors (KNN).

On the first day, we began by testing the Random Forest and KNN models. Although KNN was not expected to perform well on this dataset due to its complexity, it surprisingly outperformed Random Forest. Subsequently, we tested the Gradient Boosting model; however, it yielded suboptimal results, indicating that it might not be suitable for our data characteristics.

The following day, we divided tasks to test models independently for a more detailed assessment. I focused on an XGBoost model, which initially achieved a high F1 score of 0.52 on the training data. However, the model's F1 score on the private test dataset dropped to 0.38, suggesting overfitting. In response, I increased the missing value threshold to 90%, which provided a modest improvement in performance, though it did not fully resolve the overfitting issue.

7] XGBoost Model Experiment and Observations

This report presents the steps taken to optimize an XGBoost classifier on an imbalanced dataset. The objective was to enhance the model's performance by tuning hyperparameters and handling class imbalance, focusing on achieving the best possible F1 score.

Class Imbalance Handling

To address the dataset's imbalance, the class ratio was calculated. For binary classification, the `scale_pos_weight` parameter was set based on this ratio to prioritize the minority class, reducing bias towards the majority class.

Parameter Search with RandomizedSearchCV

The model's hyperparameters were fine-tuned using `RandomizedSearchCV`. The parameters optimized include:

- `n_estimators`: Number of trees in the ensemble
- `learning_rate`: Step size at each iteration
- `max_depth`: Maximum depth of the trees
- `min_child_weight`: Minimum sum of instance weights required in a child node
- `subsample` and `colsample_bytree`: Parameters controlling randomness
- `gamma`: Minimum loss reduction required for further partitioning

The `RandomizedSearchCV` performed a random search over 50 parameter combinations with 3-fold cross-validation, allowing efficient discovery of the best configuration.

Sample Weights for Imbalance

For binary classification, sample weights were assigned to focus on the minority class. Instances of the minority class were given higher weights, encouraging the model to classify them correctly, which is crucial for balanced performance metrics.

Model Training and Cross-Validation

The model was trained within the `RandomizedSearchCV` framework, with an evaluation set used to monitor performance. Cross-validation provided a robust performance estimate, and verbosity was minimized to focus on key information.

Performance Evaluation

The optimal parameters for the model were as follows:

```
{'subsample': 0.8, 'n_estimators': 500, 'min_child_weight': 5,
 'max_depth': 7, 'learning_rate': 0.15, 'gamma': 0.2,
 'colsample_bytree': 0.7}
```

The final optimized F1 score on the validation set was:

Optimized Test F1 Score: 0.5211

Best Model Selection and Evaluation

After conducting various experiments with different models, our team explored the performance of the TabNet model, which initially achieved an F1 score of 0.422. We then experimented with balancing the dataset to assess its impact on TabNet's performance. However, this adjustment did not lead to any notable improvement in the F1 score.

Additionally, we tested a balanced random forest model, which performed well but did not surpass TabNet's performance. Consequently, we decided to proceed with the TabNet model as our primary choice. After completing all experiments, the final F1 score on the public dataset was recorded at 0.43.