Signal Prediction on Catalonia Cell Coverage

Yang Xing, Xinyue Zeng(✉), and Farshid Alizadeh-Shabdiz

Boston University Metropolitan College, Boston, MA 02215, USA
zengxiny@bu.edu

Abstract. The GenCat Mobile Coverage app is designed to collect information on the mobile telephone network coverage in Catalonia. Through an Android app, citizens can contribute to this initiative by recording data. The dataset compiled by the platform offers a comprehensive view of the data collected between 2015 and 2017, encompassing 20 features with 11,744,914 observations. This research aims to predict signal strengths of the mobile network in Catalonia using various factors, including location, network supplier, signal strength and other relevant features.

Keywords: Spark · Classification · Regression · Stacking · Model Evaluation

1 Introduction

The dataset was acquired from Google Cloud Big Query [1], we utilized SQL queries to partition it into three segments on the platform. Subsequently, we combined these segments locally using Python. Following data preprocessing, the number of observations decreased from 11,744,914 to 8,281,531, while the number of features increased from 20 to 36. We employed classification and regression algorithms to predict the target variable "signal" and compared the Root Mean Square Error (RMSE) of each model. Hyperparameter tuning was performed for each algorithm to identify the optimal model. Lastly, we applied stacking techniques with regression models to enhance performance.

2 State of Art

With the rising popularity of Python, numerous free libraries and packages have been developed to facilitate the implementation of distributed computation frameworks for machine learning algorithms [2]. Among the most widely used tools are Spark and Hadoop, both of which offer flexible, fault-tolerant, and scalable environments. Spark utilizes RDD (Resilient Distributed Dataset) as its primary data structure, whereas Hadoop relies on HDFS (Hadoop Distributed File System). Notably, Spark leverages RAM memory to store temporary results, resulting in faster processing times compared to Hadoop in most scenarios. As stated by Samadi [3], Spark outperforms Hadoop in terms of execution time for typical algorithms such as PageRank and Word-count.

T. Zlateva and G. Tuparov (Eds.): CSECS 2023, LNICST 514, pp. 58–72, 2023.
https://doi.org/10.1007/978-3-031-44668-9_5

Regarding Hadoop, Python offers several libraries that supports its functionalities. One such library is Hadoop Streaming, which was utilized by Dede [4] to process the extensive dataset, Cassandra. Additionally, another option available is Mrjob [5], an open-source wrapper designed for Hadoop Streaming.

When it comes to Spark, the primary package supporting it is PySpark. Within PySpark, MLlib serves as Apache Spark's scalable machine learning library [6]. Spark demonstrates exceptional performance in iterative computation, which makes it well-suited for MLlib's fast execution. Moreover, MLlib incorporates high-quality algorithms that leverage iteration, often yielding superior results compared to one-pass approximations commonly used in MapReduce.

Machine learning is a well-established research field within computer science, and it plays a crucial role in the development of classification and predictive analysis systems. MLlib in PySpark supports two main categories of algorithms in supervised learning: classification and regression [7]. For classification, MLlib provides support for logistic regression, decision trees, random forests, gradient-boosted trees, Support Vector Machine (SVM), and Naïve Bayes algorithms. However, the number of algorithms for regression is relatively limited. For instance, SVM is solely implemented for classification purposes.

In terms of ensemble methods, generalized forms like stacking, boosting, and voting algorithms are not supported. However, Kaur [8] implements an ensemble method using a voting algorithm. It is important to note that algorithms are not applicable for regression problems. Therefore, our objective is to construct a pipeline that implements a stacking ensemble method using PySpark's MLlib packages within the Spark environment.

3 Data Preprocessing

3.1 Coding Environment and Package Versions

PySpark was utilized to execute Python code within the Spark framework for our experiment. Throughout the entire study, we employed Python 3.9 and Spark 3.3.

3.2 Dataset Overview

The raw dataset utilized in this study was obtained from Google Big Query Dataset Platform. It consists of a total of 11,744,914 data samples and encompasses 20 features, including 'date', 'hour', 'lat', 'long', 'signal', 'network', 'operator', 'status', 'description', 'net', 'speed', 'satellites', 'precision', 'provider', 'activity', 'downloadSpeed', 'uploadSpeed', 'postal code', 'town name', and 'position geom'. These features can be categorized into three groups: time features ('date' and 'hour') related to the record's timestamp, geographical features ('lat', 'long', 'postal code', 'town name', and 'position geom') dependent on end-users' location, and business features associated with cell phone service providers.

To enhance the clarity and meaningfulness of the dataset, we performed several feature transformations. Firstly, we converted the 'hour' feature from a 12-h to a 24-h format. Additionally, we categorized the operators, listing only the top seven by name,

while grouping the remaining operators into a single category called 'others'. This approach was employed to address the presence of numerous small companies in the real market, which would lead to a large sparse one-hot encoded matrix if included individuality. Thus, the 'operator' feature was reclassified into nine distinct groups. Furthermore, we applied other preprocessing methods to the dataset.

3.3 Feature Selection

Examine Irrelevant Features. Five features, namely 'date', 'signal', 'hour', 'activity', and 'speed' were considered.

Seasonal Effect. We extracted month information from both 'date' and 'month' features. Subsequently, we examined the correlation between 'month' and the class feature 'signal'. The results indicated no substantial evidence suggesting a seasonal effect on the signal. As a result, the 'date' feature was deemed irrelevant and excluded from further analysis.

Signal and Hour. We explored the correlation between 'hour' feature and the class feature 'signal' to determine if the signal exhibited variations throughout the day. After conducting the analysis, we concluded that 'hour' feature had no relevance to the signal and thus was eliminated from the feature set.

Speed and Activity. We investigated the relationship between the 'speed' and 'activity' features. The interpretation of the 'speed' feature could either represent the network communication speed or the moving speed of the signal source. To discern its nature, we examined the distribution of speeds across different activity groups. The results revealed a significant difference in the distribution, indicating that the 'speed' feature primarily represents the moving speed of the signal source.

Deleted Features. Six features were removed from the analysis for various reasons. The first feature to be eliminated was 'description' since it held the same meaning as 'status' feature. The second and third features, 'download speed' and 'upload speed', were excluded due to the excessive number of missing values present in both. Additionally, the fourth, fifth, and sixth features, namely 'postal code', 'town name' and 'position_geom', duplicating the information already captured by the 'lat', and 'long' features.

3.4 Preparation and Transformation

To prepare the data for analysis, several preprocessing steps were performed. Firstly, we addressed outliers in the dataset by applying the IQR (Interquartile Range) method to all remaining features, effectively removing any extreme values. Subsequently, for categorical features, we employed the StringIndexer() and OneHotEncoder() functions from the pyspark.ml.feature module, enabling us to convert the categorical variables into numerical representations through one-hot encoding. This process ensured compatibility with the machine learning algorithms employed in the study. Finally, to facilitate fair comparisons and optimize the performance of numerical features, we employed the StandardScaler() function from the pyspark.ml.feature module to standardize the data. This step improved the interpretability and convergence of the models utilized in subsequent analysis.

4 Classification Algorithms

4.1 Approach Overview

In this section, we present an overview of our approach, which involves the utilization of quantile-based grouping for conducting a multidimensional analysis of mobile network coverage data obtained through the GenCat Mobile Coverage app. The class feature, denoted as 'signal', consists of integer values ranging from 0 to 33, making it amenable to classification-based prediction. Our approach involves dividing the class feature into three distinct groups based on quantiles: below Q1, between Q1 and Q3, and above Q3. This grouping enables us to perform classification analysis and facilitates the comparison of results from regression algorithms, thereby allowing us to evaluate different approaches.

To conduct the analysis, we employ two models: Support Vector Machine (SVM) and logistic regression. The data is partitioned into three groups on the quantiles, with the first model classifying data below Q1 and the second model classifying data above Q3. The remaining data falling between Q1 and Q3 are labeled as within the IQR. Labels are assigned to each group based on the quality of network coverage, with a value of 2 denoting poor coverage, 1 representing good coverage, and 0 indicating moderate coverage.

In evaluating the performance of the classification models, it is essential to consider various metrics that provide insight into their effectiveness. One such metric is the confusion matrix, which presents a comprehensive view of the classification results by indicating the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) for a given set of predictions. TP and TN represent correct predictions, while FP and FN represent incorrect predictions.

Furthermore, to enable a comparative evaluation between classification and regression models, we estimate the numeric value of the predicted class by taking the mean value within each range. This approach provides a more nuanced understanding of network coverage quality across different locations in Catalonia and facilities comparisons with regression analysis by calculating the Root Mean Square Error (RMSE). By combining both classification and regression analyses and assigning labels based on the quality of network coverage, out approach yields valuable insights that can directly inform policy decisions and drive improvements in network infrastructure.

4.2 Support Vector Machine

Algorithm Description. Support Vector Machines (SVM) is a powerful supervised learning algorithm used for both classification and regression tasks. It is particularly effective in scenarios where the data has complex decision boundaries. SVM seeks to find an optimal hyperplane that separates the data points of different classes with the maximum margin.

Implementation. In our implementation, we utilized the Support vector Machine algorithm for classification tasks in PySpark. Specifically, we employ the LinearSVC() function from the pyspark.ml.classificaiton.

Hyperparameters. When applying LinearSVC(), we focused on tuning three key hyperparameters. The first hyperparameter, maxIter, represents the maximum number of iterations performed during the training process. Careful selection this parameter can ensure convergence to an optimal solution.

The second hyperparameter, regParam, plays a crucial role in balancing the tradeoff between bias and variance in the SVM model. It determines the regularization strength, controlling the complexity of the model. By finding the optimal value for regParam, we can strike a balance between underfitting and overfitting.

The third hyperparameter, fitIntercept, is related to the inclusion of an intercept term in the linear equation of the SVM model. This intercept term allows the decision boundary to be shifted vertically, potentially improving the accuracy of the SVM model. When fitIntercept is set to True, an intercept term is incorporated during training process.

4.3 Logistic Regression

Algorithm Description. Logistic Regression is a widely used statistical model for binary classification. It extends the concept of linear regression by applying the logistic function to estimate the probability of the binary response variable. Logistic Regression is known for its simplicity and interpretability, making it a popular choice for various classification tasks.

Implementation. In our implementation, we utilize the PySpark Logistic Regression algorithm for classification problems. We employ the LogisticRegression() function from the pyspark.ml.classification module.

Hyperparameters. LogisticRegression() offers several hyperparameters for customization. One significant hyperparameter is regParam, which controls the strength of L1 or L2 regularization applied to the logistic regression model. By adjusting the regParam value, we can control the extent of regularization applied to the model, which helps prevent overfitting and improves the generalization performance.

The second hyperparameter, elasticNetParam, determines the balance between L1 and L2 regularization in the logistic regression model. When regParam is set to a positive value, the model applies L1 or L2 regularization based on the regularization type specified by the elasticNetParam parameter. A value of 0 for regParam indicates no regularization, resulting in a standard logistic regression model.

5 Regression Algorithms

5.1 Approach Overview

In addition to considering signal with categorical labels and applying classification algorithms, we can also treat signals as numerical features and leverage regression algorithms for analysis. When running regression algorithms, the output of predictions will be in the form of float numbers. To align with the integer labels of ‘signal’, we will employ the round() function to map the predicted numbers to integers.

Another distinction between classification and regression lies in evaluating the performance of a specific algorithm. For this purpose, we will utilize packages in pyspark.mllib.evaluation and employ the RegressionEvaluator to access performance. Specifically, we will employ the Root Mean Square Error (RMSE) as the evaluation metric. RMSE is a widely used measure for evaluating the quality of predictions, representing the Euclidean distance between predictions and the true measured values.

To compute RMSE, calculate the residual (difference between prediction and truth) for each data point, compute the norm of the residuals, calculate the mean of the residuals, and finally take the square root of this mean. RMSE is commonly employed in supervised learning applications as it requires true measurements at each predicted data point. Equation (1) expresses the computation of RMSE:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} ||y(i) - \hat{y}||^2}{N}} \tag{1}$$

In our investigation of regression models supported by Spark, we found that it offers a limited number of regression algorithms, including Linear Regression, Random Forest, Gradient Boosted Trees, Survival Regression, Isotonic regression, Factorization machines regression and Decision Trees Regression. Given the nature of our problem, we have opted to utilize linear regression, random forest regression and gradient-boosted tree regression. Since random forest and gradient-boosted tree algorithms are both derived from decision trees, we have chosen not to consider decision tree regression.

We also explored whether Spark supports sklearn (scikits.learn). It appears that in Spark 2.0, there was a package called spark-sklearn, but it has been deprecated subsequently. Currently, the functionality of spark-sklearn has been incorporated into Joblib-spark, which provides an Apache Spark backend for Joblib to distribute tasks on a Spark cluster. However, it does have certain limitations. For instance, it does not generally support running model inference and feature engineering in parallel.

5.2 Linear Regression

Algorithm Description. Linear Regression is a fundamental and widely used regression algorithm that models the relationship between a dependent variable and one or more independent variables. It measures a linear relationship between the input features and the target variable, making it a simple yet effective approach for regression analysis.

In Linear Regression, the algorithm estimates the coefficients of a linear equation that best fits the given data points. It aims to minimize the difference between the predicted values and the actual values by adjusting the coefficients using optimization techniques such as gradient descent.

Implementation. Linear regression is implemented as a supervised algorithm that utilizes the ordinary least squares method to estimate the coefficients of the linear equation. In out implementation, we employ the LinearRegression() function from the PySpark MLlib to perform the training process.

Hyperparameters. In the context of LinearRegression(), our focus was on three key hyperparameters. The first one is maxIter, which specifies the maximum number of iterations. Since linear regression employs Newton's method for gradient descent, it typically converges swiftly. By appropriately setting maxIter, we can optimize computational efficiency by avoiding unnecessary iterations.

The second hyperparameter is regParam, which determines the fraction of a regularization applied to the overall loss function. Generally, increasing regParam imposes a greaeter penalty from regularization. Additionally, in conjunction with regParam, we utilized the elastic net param to regulate the mixture of L1 and L2 penalties.

5.3 Random Forest

Algorithm Description. Random Forest is a popular machine learning algorithm that leverages ensemble learning and decision trees to construct a robust and accurate predictive model. It excels in both classification and regression tasks due to its versatility and effectiveness.

The name "Random Forest" stems from the algorithm's approach of building an ensemble or a "forest" of decision trees. Each tree is trained on a random subset of the training data and features, ensuring diversity, and reducing overfitting [9].

Implementation. Random Forest is implemented as a meta estimator that fits multiple decision trees on different subsets of the dataset and employs averaging to enhance predictive accuracy while mitigating overfitting. In our implementation, we utilize the RandomForestRegressor() function from the pyspark.ml.regression library.

Hyperparameters. Two key parameters are selected for tuning: the number of trees and the depth of trees. It is important to note that increasing either the depth or the number of trees substantially impacts the runtime of the algorithm.

5.4 Gradient Boosted Trees

Algorithm Description. Gradient Boosted Tree is a powerful machine learning algorithm that combines the principles of decision trees and gradient boosting to create a highly accurate predictive model. It builds the model in a stage-wise manner by sequentially adding multiple decision trees. The algorithm works by minimizing a loss function using gradient descent, thereby improving the model's predictive performance iteratively.[10].

Implementation. Gradient boosting regression trees are implemented by utilizing the GBTRegressor() function from the pyspark.ml.regression library. This algorithm builds an ensemble of decision trees, where each tree is trained on a subset of the training data and learns to make predictions based on the residuals from the previous trees.

Hyperparameters. To tune the Gradient Boosted Trees algorithm, we focus on three key hyperparameters: stepSize, maxDepth, and maxIter.

The stepSize parameter controls the learning rate, determining the contribution of each tree to the final prediction. A smaller stepSize value leads to a more cautions learning approach.

The maxDepth parameter determines the maximum depth of each individual decision tree in the ensemble. A deeper tree allows the model to capture more complex patterns but can also increase the risk of overfitting.

Finally, the maxIter parameter specifies the maximum number of iterations or boosting stages for the algorithm.

6 Hyperparameter Tuning

6.1 GridSearch and Parameter Setting

To optimize the performance of our models, we employed the GridSearch method in PySpark for hyperparameter values and identify the optimal configuration for each model. In our implementations, we utilized the ParamGridBuilder() function to set the parameters for each estimator in the pipeline.

6.2 Model Tuning Methods

Pyspark.ml.tuning provides two main functions for model tuning: CrossValidator() and TrainValidationSplit(). These functions assist in selecting the best hyperparameter values by performing systematic evaluations of the models. While CrossValidator generally yields better results, it requires a longer training time, making it more suitable for cases where data size is limited. In our study, however, we are dealing with a large dataset, and training time is a critical consideration. Therefore, we chose to utilize TrainValidationSplit() for the entire hyperparameter tuning process.

6.3 Pipeline Model and Training

To streamline the entire workflow, we constructed a Pipeline model that integrates all the components, including data preprocessing and the models themselves. The Pipeline model allows us to apply a consistent set of transformations to the data and train the models efficiently. We used the fit() function to train the pipeline model on the training data, ensuring that all necessary preprocessing steps and model training were execute in a unified manner.

By employing the GridSearch method, setting appropriate parameters, and utilizing the TrainValidationSplit() function within the Pipeline model, we were able to systematically explore different hyperparameter combinations and select the optimal settings for our models, this approach ensures that our models are fine-tuned and capable of providing accurate predictions on the mobile network coverage data.

6.4 Best Params

(See Tables 1 and 2).

Table 1. Best params for classification.

Support Vector Machine		Logistic Regression	
Model 1	Model 2	Model 1	Model 2
maxIter = 2, regParam = 0.005, fitIntercept = True	maxIter = 10, regParam = 0.01, fitIntercept = True	elasticNetParam = 0.0, regParam = 0.005	elasticNetParam = 0.0, regParam = 0.001

Table 2. Best params for regression.

Linear Regression	Random Forest	Gradient Boosted Tree
maxIter = 100, regParam = 0.02, elasticNetParam = 0.0	numTrees = 22, maxDepth = 15	stepSize = 0.5, maxDepth = 10, maxIter = 20

7 Outputs from the Google Cloud Platform

7.1 Outputs from Classification

Since we are doing multi-class classification, it is important to evaluate the performance of our algorithms using appropriate metrics. In addition, we select logistic regression as a baseline algorithm to compare the performance of the other algorithms (Tables 3 and 4).

Table 3. Confusion matrix from logistic regression.

Actual |Prediction	TP	FP	TN	FN
Label = 0	892704	747579	10116	5917
Label = 1	9208	6777	1267180	373151
Label = 2	33	15	1280965	375303

Table 4. Confusion matrix from SVM.

Actual |Prediction	TP	FP	TN	FN
Label = 0	896267	757201	1601	1688
Label = 1	198	946	1273085	382528
Label = 2	470	1675	1279006	375606

7.2 Outputs from Regression

Linear Regression. When evaluating the performance of regression algorithms, we rely on RMSE as the key measurement. To assess the performance of different algorithms and parameter combinations, we compute the RMSE for both the training and testing datasets (with a ratio of 0.8:0.2). By comparing these values, we can evaluate how well the algorithms generalize to unseen data and identify potential overfitting or underfitting issues.

Linear regression is chosen as the baseline algorithm for comparison with other approaches, including random forests and gradient-boosted trees. By comparing the performance of these algorithms to the baseline linear regression, we can gain insights into their relative effectiveness and suitability for our regression task.

Through our experiments and analysis using From the Google Cloud Platform (GCP) statistics, we have observed that different parameter combinations have minimal impact on the performance of linear regression. This observation supports the notion that linear regression can serve as a robust and reliable baseline for our regression task (Tables 5).

Table 5. RMSE from linear regression.

maxIter	regParam	elasticnet	trainRMSE	testRMSE
20	0.1	0	7.02	7.03
20	0.1	0.25	7.02	7.03
20	0.1	0.5	7.02	7.03
20	0.1	0.75	7.03	7.03
20	0.1	1	7.04	7.04
20	0.05	0	7.02	7.03
20	0.05	0.25	7.02	7.03
20	0.05	0.5	7.03	7.03
20	0.05	0.75	7.03	7.03
20	0.05	1	7.03	7.03
20	0.01	0	7.02	7.02
20	0.01	0.25	7.02	7.03
20	0.01	0.5	7.02	7.03
20	0.01	0.75	7.02	7.03
20	0.01	1	7.02	7.03

Random Forest. When applying the random forest algorithm, we carefully consider two crucial parameters: numTrees and maxDepth. These parameters play a significant role in controlling the performance and behavior of the random forest model.

During our analysis, we examine the effect of varying these parameters on the performances of the random forest algorithm. By observing the RMSE values obtained from the testing set and comparing them with the RMSE values from the training set, we can gain insights into the model's generalization capabilities and potential overfitting tendencies. If there is a significant increase in RMSE on the testing set or a large gap between the training and testing set RMSE, it indicates that the model may be overfitting to the training data.

By carefully analyzing the relationship between these parameters and the performance metrics, we can determine the optimal values for numTrees and maxDepth that strike a balance between capturing complex patterns in the data and avoiding overfitting. This analysis allows us to select the best performing random forest model (Tables 6).

Table 6. RMSE from random forest.

numTrees	maxDepth	RMSE train	RMSE test
21	11	6.73	6.74
21	13	6.65	6.67
21	15	6.54	6.59
22	11	6.73	6.74
22	13	6.65	6.67
22	15	6.54	6.59

Gradient Boosted Trees. When utilizing the gradient boosted trees algorithm, we pay close attention to two critical parameters: stepSize and maxDepth. These parameters play a significant role in controlling the behavior of the gradient boosted trees and addressing the issue of overfitting.

By observing the RMSE values on the test set and comparing them with the RMSE values on the training set, we can conclude that if there is a significant increase in RMSE on the testing set or a substantial gap between the training and testing set RMSE, it indicates that the model may be overfitting to the training data (Tables 7 and 8).

Table 7. RMSE from gradient boosted trees.

stepSize	maxDepth	maxIter	RMSE train	RMSE test
0.5	5	10	6.8	6.8
0.5	5	15	6.78	6.79
0.5	5	20	6.77	6.77
0.5	10	10	6.58	6.62
0.5	10	15	6.53	6.59
0.5	10	20	6.5	6.57
0.5	15	10	6.13	6.67
0.5	15	15	5.96	6.75
0.5	15	20	5.83	6.83

Using the combinations of parameters that yield the best RMSE score on the test set for each algorithm, we compare the performances between the different algorithms. The results as follows:

Table 8. Best RMSE from all models.

Model	RMSE test
SVM	11.78
Logistic Regression	8.63
Linear Regression	7.02
Random Forest	6.59
Gradient Boosted Tree	6.57

8 Stacking Models

8.1 Algorithm

Stacking, also referred to as Stacked Generalization, is an ensemble machine learning technique that leverages a meta-learning algorithm to learn the optimal combination of predictions from multiple base machine learning algorithms. The fundamental advantage of stacking lies in its capability to exploit the strengths of diverse models, leading to predictions that surpass those of individual models in the ensemble.

In our study, we will utilize the three - Linear Regression, Random Forest Regression, and Gradient Boosted Trees Regression - as the base algorithms for stacking. Linear regression will be selected as the ensemble algorithm to fuse the predictions generated

by the base models. To facilitate the stacking process, we have implemented two distinct training pipelines, each with its own merits and considerations.

The first training pipeline involves a straightforward split of dataset into training and testing sets. Within this pipeline, both the base algorithms and the ensemble algorithm are trained on the training data. This approach offers the advantage of consolidating all model training procedures into a single pipeline, streamlining the training process. However, it is crucial to note that this method may expose the ensemble algorithm to the risk of severe overfitting, as it heavily relies on the predictions generated by the base models.

The second training option involves a three-part split of the data, consisting of train data, validation data, and test data. The train data is exclusively utilized for training the base models, while the validation data is employed to train the ensemble algorithm using the predictions generated by the base models as input features. Finally, the performance of the stacked model is evaluated using the test data. This approach allows for improved control over overfitting concerns by utilizing the validation data to fine-tune the hyperparameters of the ensemble algorithm.

8.2 Implementation

In our implementation, we employ linear regression, random forest regression, and gradient boosted trees regression as the base models. Each base model is carefully selected based on the best set of parameters determined in the previous sections of our study. The stack algorithm is then utilized to combine the results from each individual model. The detailed process involves training each base algorithm separately and utilizing the predicted values – outputted by the base models – as new features for predicting values based on linear regression.

To facilitate the implementation, we utilize the pipelines functionality from the PySpark library. However, it is important to note that the usage of pipelines introduces certain considerations. Notably, pipelines operate on the same RDD (Resilient Distributed Dataset), resulting in two different methods being implemented based on whether the training set it divided for the base models and the stack model.

In the first method, a single pipeline is established to encompass the entire workflow, including data preprocessing, training of the individual models, prediction of values, transformation of predicted values into new features, and training of stack model. This single pipeline computation takes place sequentially on the training set. The model saved from this pipeline is subsequently applied to the test set to calculate the RMSE. Utilizing a single pipeline enhances computational efficiency by involving only one RDD in the training process. However, it may raise concerns regarding overfitting, as each data sample is utilized twice – once in the base models and once in the stack model.

The second method involves the establishment of two pipelines: one for the base models and another for the stack model. In this approach, the training set is split into two parts. The first part is employed to train the base models, while the second part is used to train the stack model. This two-stage training process begins with training the base models on training set 1, followed by passing all model information to stage 2. In stage 2, the stack model is trained on training set 2 using predicted values generated by

the models saved in stage 1. Notably, in this method, each data sample will only be used once. But it will cause longer time to proceed.

8.3 Outputs from Google Cloud Platform

The stack model was executed on the Google Cloud Platform, and the training process took approximately 49 min to complete. The performance of the ensemble model surpassed that of the individual base models, highlighting the efficacy of the stacking approach. We observed a notable discrepancy between the train RMSE and test RMSE suggests the presence of overfitting when utilizing the first option, proving the importance of employing appropriate validation techniques to mitigate overfitting risks.

RMSE for each base model. LR: 7.02, RF: 6.59, GBT: 6.57.
RMSE for single pipeline. Train: 5.59, Test: 6.92.
RMSE for two pipelines. Train: 6.512, Test: 6.517.

9 Conclusion

9.1 Findings on Running GCP

(1) Classification and regression models can be considered when dealing with integer type labels. Both types of models have shown potential effectiveness in handling such labels, and the choice depends on the specific problem and desired outcome.
(2) When tuning hyperparameters, it is advisable to start with smaller values to reduce the overall running time on the GCP. This approach allows for faster experimentation and exploration of different parameter settings.
(3) Splitting the training data appropriately for base models and ensemble models is crucial when creating the stack ensemble model. This separation ensures that the base models capture relevant patterns and provide reliable predictions as inputs to the ensemble model.

9.2 Findings on Signal Coverage Prediction

(1) The dataset is sufficiently large, indicating that removing null values and outliers will not significantly reduce the dataset's size. However, due to the dataset's large volume, careful feature selection is essential. Including irrelevant features can increase computational complexity and potentially lead to decreased model performance.
(2) When dealing with multiple classes in the class feature, grouping them into two or three classes for classification models and assigning the mean value from each group's range may not yield optimal. Instead, directly fitting regression models to the data can potentially provide better predictions.
(3) The obtained RMSE values for both the train and test data are not satisfactory, suggesting the presence of other influential features not included int the dataset. Exploring additional relevant features or collecting more comprehensive data may lead to improved prediction accuracy.

References

1. Dataset: Catalonia cell coverage. https://console.cloud.google.com/marketplace/product/gencat/cell_coverage?hl=zh-cn&project=firm-retina-379321&pli=1
2. Stančin, I, Jović, A.: An overview and comparison of free Python libraries for data mining and big data analysis. In: 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp. 977–982, Opatija, Croatia (2019). https://doi.org/10.23919/MIPRO.2019.8757088
3. Samadi, Y., Zbakh, M., Tadonki, C.: Performance comparison between Hadoop and Spark frameworks using HiBench benchmarks. Concurrency Comput. Pract. Exp. **30**(12), e4367 (2018)
4. Dede, E., Sendir, B., Kuzlu, P., Weachock, J., Govindaraju, M., Ramakrishnan, L.: Processing Cassandra datasets with hadoop-streaming based approaches. IEEE Trans. Serv. Comput. **9**(1), 46–58 (2016). https://doi.org/10.1109/TSC.2015.2444838
5. Mrjob, version 0.7.4, David, Marin. The Python MapReduce library (2020). https://pypi.org/project/mrjob/
6. Tellez, A., Pumperla, M., Malohlava, M.: Mastering Machine Learning with Spark 2.x (2017)
7. Yadav, R.: Spark Cookbook: over 60 recipes on spark, covering spark core, spark SQL, spark streaming, MLlib, and GraphX libraries (2015)
8. Kaur, G.: A comparison of two hybrid ensemble techniques for network anomaly detection in spark distributed environment. J. Inf. Secur. Appl. **55**, 102601 (2020). https://doi.org/10.1016/j.jisa.2020.102601
9. Ho, T.K.: Random decision forests. In: Proceedings of 3rd International Conference on Document Analysis and Recognition, vol.1, pp. 278–282, Montreal, QC, Canada (1995). https://doi.org/10.1109/ICDAR.1995.598994
10. Duffy, N., Helmbold, D.: Boosting methods for regression. Mach. Learn. **47**, 153–200 (2002). https://doi.org/10.1023/A:1013685603443