FIT5202 (TP3 2020)

A Comparison of Binary Classifiers Using Apache Spark for Scala

Big Data Group Project – Phase 3: Machine Learning Phase

GROUP 11

Alec David Vukovich
Darragh Brendan Ruddy
Jemal Mohammed-Nour
Matija Zivkovic
Myrnelle Jover

THE 5 PROJECT SUPERVISORS CO., PERTH, WESTERN AUSTRALIA

Table of Contents

Big Data Group Project – Phase 3

Team Members and Contribution

The team members, their roles and responsibilities for this phase of the project are summarised below.

Team Member	Contribution
Alec David Vukovich	Testing of different models and parameter tuning. Contributing to report writing. Modifications to phase 2 pipeline.
Darragh Brendan Ruddy	Pipeline prototyping, efficiency improvement, continuous integration.
Jemal Mohammed-Nour	Tested cross validation, independently tested logistic regression and drafted report.
Matija Zivkovic	Initial implementation of logistic regression and gradient boost. Research into parameter tuning methods and cross-folds validation. Performance analysis.
Myrnelle Jover	Independent testing of logistic regression. Research into binary classification models and feature encoding. Report editing.

Phase 3 Overview

Phase 3 of our Big Data Group Project is the development and deployment of a machine learning pipeline in Scala and Spark. Running the phase 3 notebook cleanses and reshapes the data found in phase 1, and some of the data wrangling from phase 2 was revised.

The dataset selected for this project (ks-projects-201801.csv) consists of Kickstarter campaigns from the popular crowdfunding platform, and is publicly available on Kaggle (Mouillé, 2017). The aim of a crowdfunding campaign is for a large number of people to pledge small amounts of money to achieve a specified funding goal. A successful campaign is one which meets its funding goals.

The process undertaken for this phase of the project was as follows:

1) Data transformation. Phase 2 cleansing and transformation pipeline was further refined prior to using data for machine learning.
2) ML Tuning involves tuning the ML algorithms and pipelines to produce the best model for the given task.
3) Model evaluation. Models were evaluated based on metrics and run time.

The outputs for this project are:

a) This report
b) The final machine learning pipeline – 'ML_pipeline.ipynb'. This pipeline incorporates phase 2 cleansing and reshaping as well as phase 3 machine learning. Individual team members worked on each segment of the pipeline, as outlined in the 'Team Members and Contribution' section above.
c) An 'Appendix' folder. This folder contains the machine learning models and their performance, which was collaboratively worked on by group members. In particular, 'pipeline_crossValidation_paramGrid.ipynb' contains two models (Gradient-Boosted Trees and Support Vector Machines) used for comparison with our final model (Logistic Regression). The 'classifier_runtimes.ipynb' notebook shows a comparison of run-time performance.

Model Selection and Justification

When determining which machine learning method to use on our data, we were required to ask ourselves the question, what do we want to predict? We wanted to know whether a Kickstarter project would be successful. This question led us to research out-of-the-box binary classification libraries for Spark. The Apache Spark documentation (Apache Spark, 2020) on classification and regression showed the following appropriate libraries:

- Logistic regression;
- Gradient-boosted decision trees; and
- Support vector machines.

We initially worked on the logistic regression (LR) model because it was the simplest to understand. The model takes in a response variable and a set of explanatory variables; this set can have one or more elements. The model predicts a success probability to our response variable (*stateBinary*) - the probability in itself is continuous but will output a binary prediction. If, for any data point, the probability of success is greater than 50%, the output is 1 - and 0 otherwise.

We wanted to compare these results to those of a model which could handle the imbalanced learning problem. Support vector machines (SVM) are regularly used in fraud detection by rephrasing the binary classification problem into an outlier detection problem, where a value of 1 is deemed as an outlier. Using this as our alternate model saved us from weighting and sampling techniques in the logistic regression model to rebalance the class proportions in the training data (Brownlee, 2020). A support vector machine uses projections into higher dimensions to determine its decision boundary, but this makes it very slow to run.

Gradient-boosted decision trees (GBT) are used in place of logistic regression models when the relationship between the label and feature variables are more complex. This is because GBT iteratively adds variables which will improve the model. Decision trees determine a binary cut-off for each decision branch, so they also tend to handle both qualitative and quantitative data effectively. However, this modelling process can lead to potential performance issues and can make the model sensitive to input data.

A common thread between the models we chose are that they are all supervised learning methods. From the research above, we expect the regression model to be the fastest.

Step 1) Data Transformation

A. Changes to Phase 2 Pipeline

One of the first changes to our original notebook was to copy the data file into HDFS using shell commands. We originally used SqlContext to read the contents into a DataFrame but as this is now deprecated, we opted to use SparkSession instead.

After prototyping a logistic regression implementation using our phase 2 output, we realised that using StringIndexer to handle categorical features and then encoding all features with OneHotEncoder, would be more readable, efficient and consistent than our original implementation. StringIndexer encodes a string column as label indices and OneHotEncoderEstimator maps an array of label index columns to binary vectors. The output of the OneHotEncoderEstimator was then passed into a VectorAssembler to assemble all the features as a single vector column, the format as required by Spark ML classification estimators.

We also noticed a class imbalance in *stateBinary* due to other states (e.g. 'cancelled') contributing to the number of campaign failures, so we decided to only consider 'successful' projects as 1 and 'failed' projects as 0, and the formatting of the "stateBinary" function was rewritten accordingly. We also changed the order of the pipeline steps so that the continuous column, *Duration(Days)*, was discretised by the "discretizeContinuousVar" function. Additionally, the "discretizeNameLength" function has been modified to fill Nulls as 0 rather than -1 as the OneHotEncoderEstimator had problems dealing with negative numbers.

B. Feature Selection

The following table shows the column-wise changes which led to our final feature selections. All final features were OneHotEncoded and assembled into a single feature vector using VectorAssembler.

Action	Reason for action	Columns affected
Dropped columns	To avoid data leakage, we dropped columns which should not be known at the time of a project launch.	• Backers • Pledged amount
	Columns do not provide useful data to inform the model.	• ID • Project name
	Replaced original columns with transformed versions to get rid of redundant information. The transformed versions are either discretized (e.g. from usd_goal_real to usd_goal_real_discrete) or aggregated (e.g. Duration (Days) = Deadline Date - Launched Date).	• Goal value • Deadline date • Launched date • State
	Variables are too highly correlated with state.	• USD pledged real
Renamed columns	For Spark ML to easily find the label column, we renamed stateBinary to label.	• Binary state values
Selected columns	StringIndexer and OneHotEncoderEstimator were used to format the columns appropriately for Spark ML, and VectorAssembler transformed the selected columns as one "feature" column.	• Category • Main category • Currency • Country • Launched month • Deadline month • Launched year • Deadline year • Duration (Days) discrete • USD goal real discrete • Name length binned

C. Attempts at Other Feature Selection Techniques

We also attempted a Principal Components Analysis (PCA) (Apache Spark, 2020) to select the features which have the most variation however, due to time constraints the technique was not able to be applied. It was difficult to use this technique as output visualisations (Principal Component and Scree plots) were not readily available through Spark with Scala.

Step 2) ML Tuning (Training and Validation)

A. Cross Validation

To find optimal model parameters, we chose to use Spark ML's built in CrossValidator class (Apache Spark, 2020) for *k*-fold cross validation to tune the model. A model's performance should not be assessed on how closely it fits the training data but how well it generalises to unseen data and cross validation is a technique that does exactly that.

K-fold cross validation involves splitting the training dataset up into *k* segments, and training and testing the model *k* times, using each segment as the test data set at least once. The advantages of this method is that it makes the most of available data by using all data for both training and testing, and the model is fit and assessed multiple times which ensures that the model validation metrics are representative of what would be expected when applying the model to a new unseen dataset.

The challenge of working with *k*-fold cross validation is that it is very computationally expensive, especially when combined with a parameter grid (discussed below). Increasing the number of folds enhances the benefits of *k*-fold cross validation but each fold means that an additional model needs to be fit. To balance computational time vs validation performance, we settled on 5-fold cross validation. This trains the model on 80% of the training set each iteration which is an acceptable amount given the overall dataset contains 331,465 records.

We set the seed set to 42 for train/test split in order to consistently compare results across models for a given metric, which also ensured result reproducibility.

B. Parameter Grid

We used arrays in a parameter grid to determine the best combination of parameters for tuning the models. The parameter grid tested 3 × 3 = 9 parameter combinations for LR, 4 × 3 = 12 parameter combinations for GBT and 3 × 4 = 12 parameter combinations for SVM. Arrays are shown in brackets.

Parameter	Description	LR Parameters	GBT Parameters	SVM Parameters
Threshold	The probability threshold for being classed as "1".	**Selection:** (0.4,0.5,0.6) **Best parameter:** 0.6	N/A	**Selection:** (0.4,0.5,0.6) **Best parameter:** 0.4
RegParam	The regularisation parameter. Higher RegParams reduce overfitting but can add bias.	**Selection:** (0.001,0.01,0.1,1) **Best parameter:** 0.01	N/A	**Selection:** (0.001,0.01,0.1,1) **Best parameter:** 0.001
MaxDepth	Sets the maximum number of nodes from root to furthest leaf node. Higher values create deeper trees.	N/A	**Selection:** (5,7,9,11) **Best parameter:** 9	N/A
StepSize	The learning rate. Smaller StepSizes are better for generalising but increase computational cost.	N/A	**Selection:** (0.05,0.1,0.2) **Best parameter:** 0.2	N/A

C. Evaluation Metrics to Select the Best Model

To evaluate the performance of the classification models, we made a comparison between them using the following metrics: accuracy, precision, recall, FPR and F1-Score (Apache Spark, 2020). The diagram below shows how to calculate these.

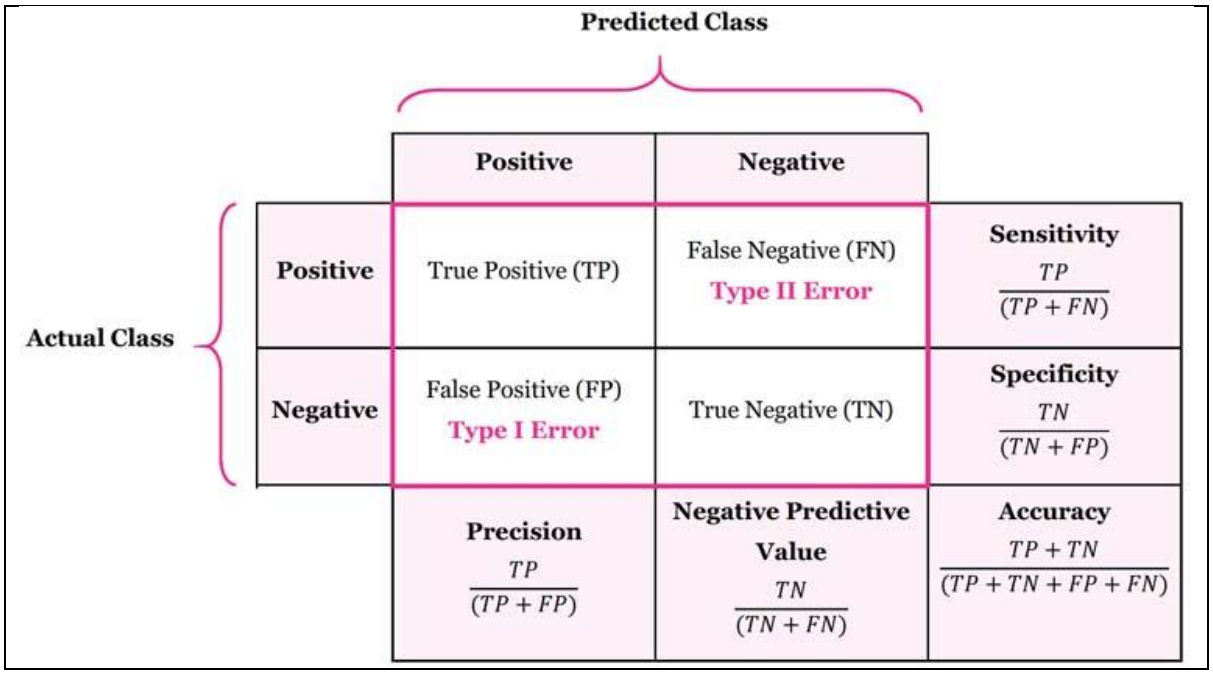

Figure 1 Evaluation metrics - definitions and calculations (Sirsat, 2019).

Accuracy is the most regularly used evaluation metric for classification tasks because it is not as cost-sensitive nor as biased as precision and recall, but it can be misleading when we have imbalanced class examples like we do. Accuracy is usually compared to the **F1-score**, which is calculated:

$$F1 = \frac{2TP}{2TP + FN + FP}$$

The F1-score is the weighted harmonic mean of the precision and recall, and is usually a better metric than accuracy for imbalanced class data.

Step 3) Model Evaluation

A. Comparison of Evaluation Metrics

The models were evaluated on the following metrics for the reasons outlined in *Step 2C: Evaluation Metrics to Select the Best Model,* and their results are outlined below.

	Logistic Regression (LR)	Gradient Boost (GBT)	Support Vector Machine (SVM)
Confusion matrix	Confusion matrix: 35675.0 17863.0 3829.0 8845.0	Confusion matrix: 33160.0 15486.0 6344.0 11222.0	Confusion matrix: 32131.0 13867.0 7373.0 12841.0
Accuracy	0.67238567027125	0.6703014559294388	0.6792122273908053
Precision	0.7936028142104069	0.7281859148168569	0.7118301696358423
Precision (Successes)	0.3311741800209675	0.42017373071738806	0.48079227197843344
Precision (Failures)	0.9030731065208586	0.8394086674767112	0.8133606723369785
Recall	0.67238567027125	0.6703014559294388	0.6792122273908052
Recall (Successes)	0.6978854347483037	0.6388477741090743	0.6352527950925101
Recall (Failures)	0.6663491351936942	0.681659334786005	0.6985303708856907
FPR	0.3081511003292521	0.3497943470343594	0.34542906141260443
F1-Score	0.7060514152388241	0.687244217572258	0.6892285782583696
Runtime (Seconds)	79	219	1236

Additionally, the models were timed to assess how long each model took to fit. Running times on our respective group members' machines differed in processing power and memory, but overall the Logistic Regression was the fastest model to fit. Our best result was 79 seconds for Logistic Regression, 219 seconds for the Gradient Boost classifier and 1236 seconds for the Support Vector Classifier. Given that model speed is important at scale and the similarity of performance metrics above, this was further reason for choosing the Logistic Regression in our final pipeline.

B. Further Improving Performance

Inspecting the Spark jobs on localhost:4040 allowed for identification of the largest performance bottlenecks. In the image below, the StringIndexer stage is repeated many times and takes approximately five seconds each time; similarly, the metrics output stage required several collect and count operations, which are slow to execute.

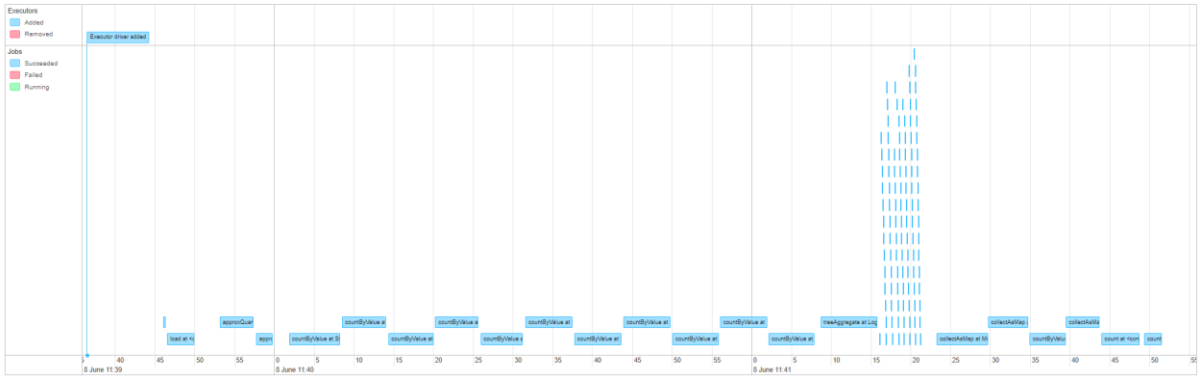

Figure 2: Spark jobs shows that the StringIndexer and metrics output stages are the largest performance bottlenecks.

By caching input DataFrames and RDDs before each of these stages, the overall runtime efficiency was increased by approximately 40%. The results can be seen in the image below, which shows the increased efficiency of StringIndexer and metric outputs after the initial operation.

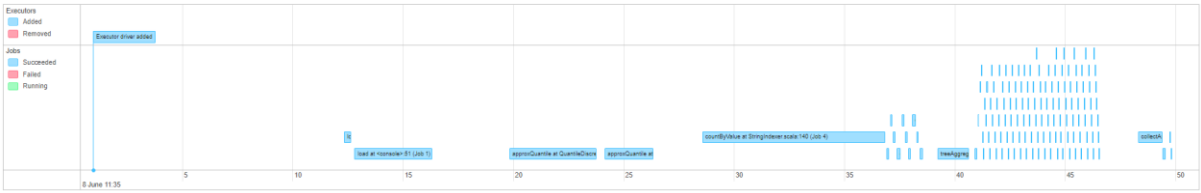

Figure 3: Improved runtime efficiency after caching before relevant pipeline stages.

Conclusion

All three models were comparable in accuracy and recall, with approx. ~0.67 for both. In all the other metrics, Logistic Regression outperformed the others; it has the highest weighted precision (0.79), lowest false positive rate (0.30) and highest F1-score (0.70). It also runs 2.6 and ~15 times faster than the Gradient-Boosted Tree and Support Vector Classifier, respectively. Its precision and recall metrics for success versus failure show that the Logistic Model makes more predictions of 'success' outcomes. It therefore gets less of its 'success' predictions correct (lower precision for success) than the other two models, but it also predicts a higher percentage of successes (higher recall for success) than the other two models. The interpretation of the recall and precision metrics depend on the business case - for instance, in the case of heart disease, making too many positive predictions (lower precision) may be fine, as a higher rate of heart disease capture is a preferred outcome. In our scenario, we are interested in capturing a higher percentage of successful projects, and therefore we accept the Logistic Model's tendency to make more predictions of 'success'.

The metrics for our Logistic Regression model show its F1-score (0.70) is higher than its accuracy (0.67), which means that one of the classes has a high sensitivity. Since this is true for all three models, then some of the inaccuracy which is present in the model *must be* due to the imbalance between campaign successes and failures. Without weighting and sampling techniques, any model fitted to the data will over-predict for failures - but this is a very common issue in machine learning.

References

Apache Spark. (2020). *Classification and Regression.* Retrieved from http://spark.apache.org/docs/latest/ml-classification-regression

Apache Spark. (2020). *Evaluation Metrics - RDD-based API.* Retrieved from MLlib Guide: https://spark.apache.org/docs/2.2.0/mllib-evaluation-metrics.html

Apache Spark. (2020). *MLlib Guides.* Retrieved May 2020, from Model Selection and Tuning: https://spark.apache.org/docs/latest/ml-tuning.html#cross-validation

Apache Spark. (2020). *MLlib Guides.* Retrieved May 2020, from Dimensionality Reduction: https://spark.apache.org/docs/latest/mllib-dimensionality-reduction#principal-component-analysis-pca

Brownlee, J. (2020, January 14). 8 Tactics to Combat Imbalanced Classes in Your Machine Learning Dataset. *Machine Learning Mastery*. Retrieved May 2020, from https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/

Mouillé, M. (2017, November 07). *Kickstarter Projects.* Retrieved April 2020, from Kaggle: https://www.kaggle.com/kemical/kickstarter-projects

Sirsat, M. (2019, April 29). *What is Confusion Matrix and Advanced Classification Metrics?* Retrieved May 2020, from Data Science and Machine Learning: https://manisha-sirsat.blogspot.com/2019/04/confusion-matrix.html?m=1