Group 6: Bitcoin Transaction Graph

# FINAL PROJECT

By Yuchao Wu

## Table of Contents

## 1.0 Introduction

In this project, an anonymized dataset will be analyzed and explored, which contains transaction graph collected from the Bitcoin blockchain. These transactions include licit categories (exchanges, wallet providers, miners, licit services, etc.) and illicit ones (scams, malware, terrorist organizations, ransomware, Ponzi schemes, etc.) [1]. A graph is generally comprised of node and edge. For this problem, a single node in the graph is a transaction and an edge represent a flow of Bitcoins between one transaction and the other.

For this project, the primary goal is to adopt a graph-based machine learning (ML) model to classify the licit and illicit Bitcoin transactions. To achieve this, both supervised and semi-supervised machine learning algorithms are utilized. Once the model with the best performance (e.g., high F1 score or accuracy) is established, it will be used to classify the unlabelled data points or records (e.g., Bitcoin transactions). All the models are built with Pyspark in Data Bricks Community edition.

To be more specific, two approaches are performed as shown in figure 1.1. For the first approach, the original bitcoin dataset will directly be fit into the machine learning models (e.g., random forest and logistic regression). Alternatively, the graph analysis is also directly applied on the orginal dataset with label propagation techniques for prediction. For the second approach, the graph generated features (e.g., PageRank and degrees) will be added to the original dataset and then trained with the traditional machine learning models. All of the above models will be evaluated and compared at end of the report. The model with the best performance will be used to predict unlabelled records.

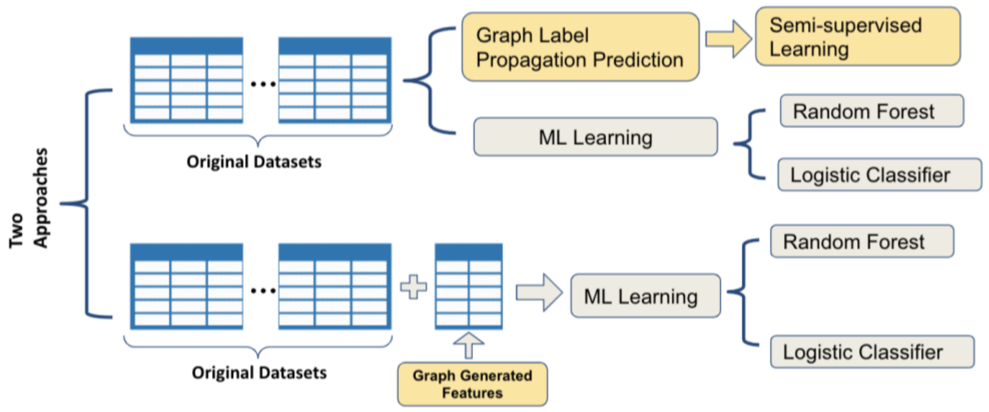


Figure 1.1: Schematic diagram for two approaches

## 2.0 Dataset

There are three datasets for this project including a feature dataset, edge dataset and a label dataset. First, the three datasets are combined and joined in Spark SQL to generate one dataset, which contains all the features and labels. Due to the fact the dataset is pre-engineered by data scientists, it does not require any data cleaning process. Also, the dataset is of great quality without any NULL values so that we could focus more on developing and tuning the models. The dataset has about 203,769 nodes and 234,355 edges. Initially, two percent of the nodes are labeled as class (illicit) and twenty-one percent are labelled as class 2 (licit). The remaining transactions (e.g., 77%) are not labelled and named as unknown. The graph figure 2.1 illustrates the class compositions for entire dataset.

Each node has 166 features, which are not provided with any detailed information and cannot be interpreted easily by human because of intellectual property issues. The first 94 features are local information about the transaction, which includes the time step (this will be removed from the training dataset as it is not relevant to the target), number of inputs and outputs, transaction fee, output volume and etc. [1]. The remaining 72 features are aggregated features obtained from the transaction information, such as maximum, minimum, standard deviation and correlation coefficients of the neighbour transactions for the same information data (number of inputs/outputs, transaction fee, etc.) [1]. The figure 2.1.2 illustrates the portion of the features for the dataset.

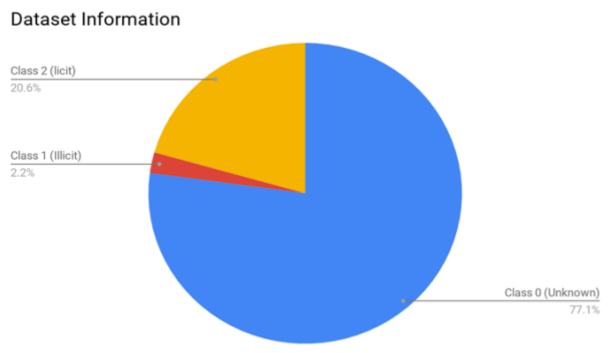


Figure 2.1.1: Label Information

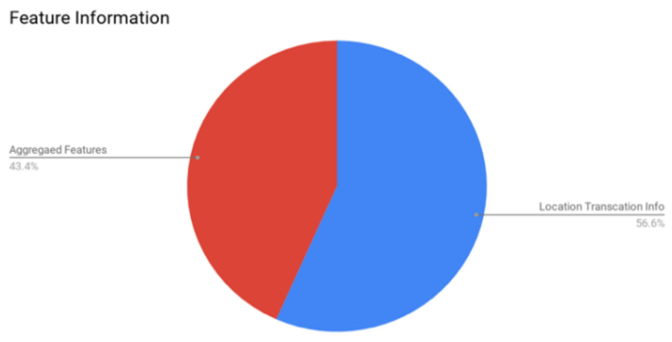


Figure 2.1.2: Feature Information

## 3.0 Graph Analysis

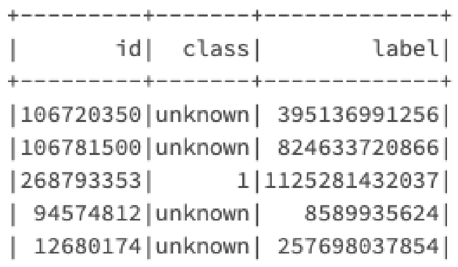
### 3.1 Graph Frame

According to GraphFrames guideline published on Databircks, GraphFrames is a package of Apache Spark and used to generate DataFrame-based graphs [2]. This package extends the functionality and capability of GraphX by incorporating the features of Spark Data Frames. The common algorithms involved in GraphFrame includes Label propagation, PageRank and Degrees. In this report, I will explore how these concepts and topics will be utilized in the Bitcoin transactions analysis.

To begin with, the GraphFrame will be created with vertex and edge Data Frames. As mentioned in previous section, our datasets are comprised of transaction features and edge lists, which will be used as vertices and edges for GraphFrame generations. In this case, each node or vertices represents a Bitcoin transaction and an edge can be interpreted as a flow of Bitcoins between one transaction and the other.

## 3.2 Label Propagation – Semi-supervised Learning

Once the Graph Frame is built, next we will start developing the first graph-based model using Label Propagation Algorithm (LPA). This is a semi-supervised learning method and tends to be very suitable for our problem since the majority (e.g., 70%) of the datasets are not labelled properly. Implementation of LPA would allow us to assign labels to unlabelled records by propagating labels through the dataset. In Pyspark, label propagation process can be achieved by applying the function, labelPropagation(), to a specific graphframe dataset. As for our Bitcoin dataset, a new community label will be given to each data point (e.g., transaction ID) and basically predicts which community it belongs to, generating an additional column shown as in figure 3.2.1.



```
+---------+-------+-------------+
|       id|  class|        label|
+---------+-------+-------------+
|106720350|unknown| 395136991256|
|106781500|unknown| 824633720866|
|268793353|      1|1125281432037|
| 94574812|unknown|   8589935624|
| 12680174|unknown| 257698037854|
```

Figure 3.2.1: Label Propagation

Once the community labels are established, we could use this additional feature to train and predict the unknown data point. The basic idea behind this algorithm is illustrated in figure 3.2.2. To be more specific, the unlabelled Bitcoin transactions (e.g., grey dots) could be predicted based on the community (the label we just created from previous step) they belong to. For example, if a community is consisted of mostly illicit transactions, the likelihood of the remaining unlabelled transactions belong to this community are also likely to be illicit. Following this logic, we are able to predict and classify most of the unlabelled data point. The final outputs are summarized in table 3.2.1. It is noted that the column, “predicted_class”, contains the unknowns because the majority of this community are not labelled or there could be no single transaction in this community being labelled.

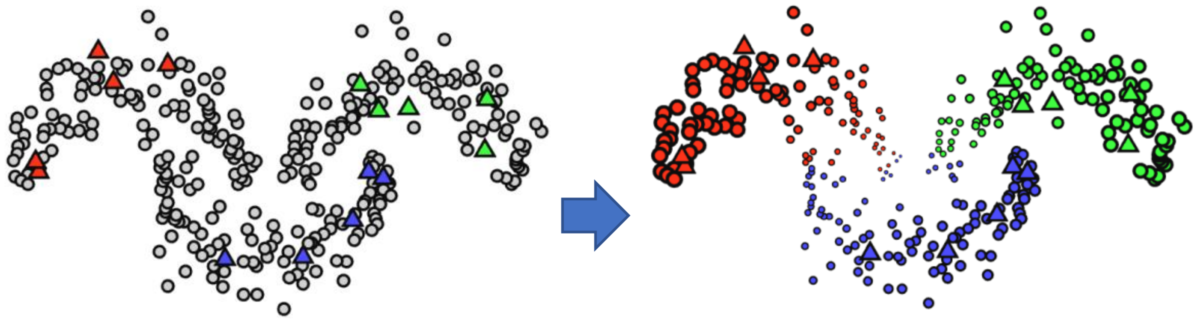
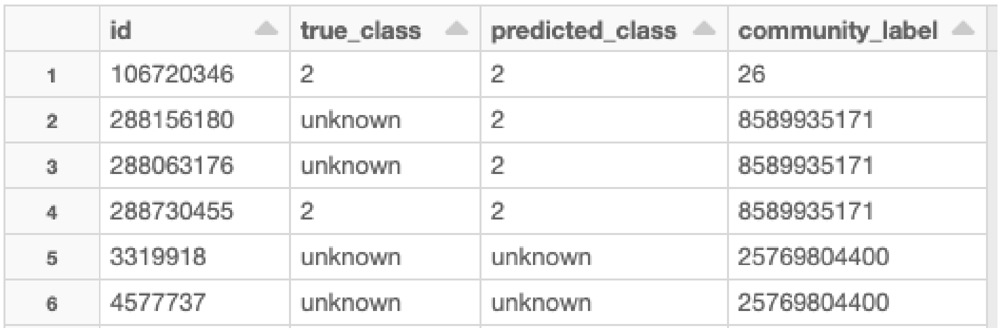


Figure 3.2.2: Illustration of Prediction with Label Propagation

Table 3.2.1: Predicted results with label propagation



| | id | true_class | predicted_class | community_label |
|---|---|---|---|---|
| 1 | 106720346 | 2 | 2 | 26 |
| 2 | 288156180 | unknown | 2 | 8589935171 |
| 3 | 288063176 | unknown | 2 | 8589935171 |
| 4 | 288730455 | 2 | 2 | 8589935171 |
| 5 | 3319918 | unknown | unknown | 25769804400 |
| 6 | 4577737 | unknown | unknown | 25769804400 |

As a result, the model has successfully predicted 119,802 nodes out of total 203,769 data points. From the section 2.0, it is given that there are total 203,769 nodes in our dataset with 46,564 being labelled (either as illicit or licit class) and 157,205 data points not being labelled. This means that the model is able to predict about 53% of all the data points (53% = 119,802/203,769) and about 45% of unlabelled data points (45% = (119,802 – 46,564) / 157,205). However, 83,967 records cannot be predicted by the model, which is about 41% of the total records and 55% of unlabelled data. This could be seen as a limitation for the model and we need to incorporate the machine learning model for the classification of the rest of data.

Table 3.2.2: Portion of Data predicted Correctly

| | % of Total Records | % of Unlabelled Records |
|---|---|---|
| **Correct Prediction** | 59% | 45% |
| **Incorrect Prediction** | 41% | 55% |
| **sum** | 100% | 100% |

Although the model is not capable of predicting all the records, it has very good performance with an accuracy of about 97.5%. As shown in table 3.2.3, the model has successfully predicted 3832 records as class 1 (TF) and 41582 records as class 2 (TP) while 713 for class 1 (FP) and 437 (FN) for class 2 are incorrect. Overall, almost all the metrics for his model have a value of around 98% which is a great starting point for this Bitcoin classification task.

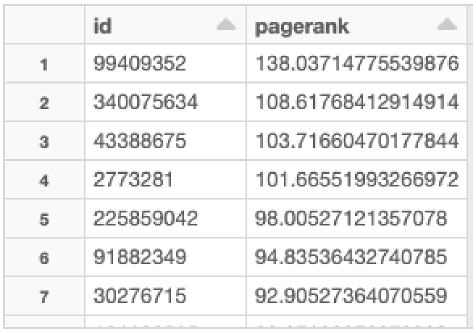
Table 3.2.3 Confusion matrix and evaluation metrics for label propagation results

| | | Actual | |
|---|---|---|---|
| | | Positive | Negative |
| Predicted | Positive | 41582 | 713 |
| | Negative | 437 | 3832 |

| Metrics | Values |
|---|---|
| Precision | 0.98314222 |
| Recall | 0.98959994 |
| Accuracy | 0.97530281 |
| F1 | 0.98636051 |

## 3.3 Page Rank

PageRank is used by Google Search engine for ranking of retrieved results, which measures the importance of the web pages [3]. As for our project, the PageRank is used to identify important vertices in the graph generated previously. This algorithm assigns a weighting score to every data or node of the graph dataset (network). Such weight will rank the relative importance of a node within the graph based on their traffic to other nodes. For example, the node with higher rank has higher possibility to be the illicit transaction (e.g., Bitcoin gambling website). By applying the function, pageRank(), to the graph dataset, we are able to compute the PageRank value for each record or transaction id as shown in figure 3.3.1. The detailed code is demonstrated in the appendix C.

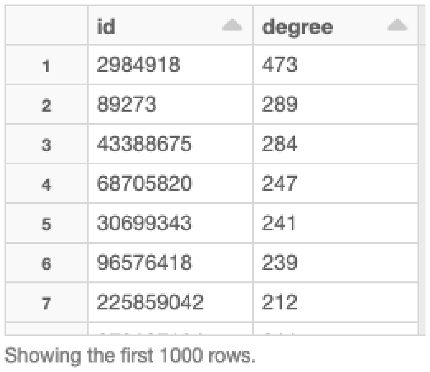


| | id | pagerank |
|---|---|---|
| 1 | 99409352 | 138.03714775539876 |
| 2 | 340075634 | 108.61768412914914 |
| 3 | 43388675 | 103.71660470177844 |
| 4 | 2773281 | 101.66551993266972 |
| 5 | 225859042 | 98.00527121357078 |
| 6 | 91882349 | 94.83536432740785 |
| 7 | 30276715 | 92.90527364070559 |

Figure 3.3.1: PageRank results for each transaction ID Descending order)

## 3.4 Degrees

In graph theory, the degree of a vertex or a node is the number of edges that are incident to the vertex. This concept is very important as we are able to see the centrality of the vertices by plotting the degree graph. As shown in the figure 3.4.1, we could see the id 2984918 has the greatest number of degrees of 473, meaning this node has the most connections and hence could be the illicit transaction. This technique is very useful for the fraud detection in a graph analysis. Both degrees and PageRank can be appended to the original dataset and fed into the machine learning models. The following section will explain in more detail of how these additional two features could further boost the model accuracy.



| | id | degree |
|---|---|---|
| 1 | 2984918 | 473 |
| 2 | 89273 | 289 |
| 3 | 43388675 | 284 |
| 4 | 68705820 | 247 |
| 5 | 30699343 | 241 |
| 6 | 96576418 | 239 |
| 7 | 225859042 | 212 |

Showing the first 1000 rows.

Figure 3.4.1: Degree results for each transaction ID (Descending order)

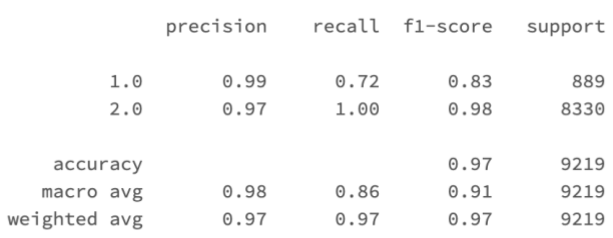
## 4.0 ML Models

In this section, we will combine both graph analysis and traditional machine learning models to generate a classification model. According to the research report (Anti-Money Laundering in Bitcoin) conducted by IBM and MIT, Random Forest (RF) performs the best among the Logistic Regression (LR), Multilayer Perceptron (MLP) and Graph Convolutional Networks(GC) [4]. As for our study, we will adopt the graph-based machine RF and LR models to classify the licit and illicit transactions.

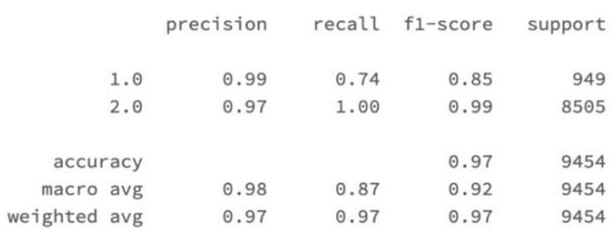
### Random Forest

Random forest is a supervised learning algorithm and also an ensemble of decision trees [5]. Basically, it creates multiple decision trees and combine them together to produce a more accurate prediction [5].

Initially, RF model was trained with only graph features (e.g., two features PageRank and degrees only), but the accuracy and F1 score are between 84% to 89%. Then, we trained the RF model with the original dataset without adding additional graph features, which boosts the accuracy and F1 score to around 97%. Next, the additional two features are appended to the dataset and fit into RF model, which increases the F1 score and accuracy slightly, but the values are still around 97%. The best RF model has 10 number of trees and a maximum depth of 10 (other parameters, such as, maxBins and checkpoint interval are all default values) after grid search and cross validation. Due to limitation of the Data Brick Community edition, we are unable to fit many hypermeter options to the model as the Data Bricks tend to crash after a long period of running. Thus, we decided to only test several pairs of hypermeters (e.g., MaxDepth and numTrees) but this could be further improved with an upgraded Data Brick notebook. The printed Pyspark classification report results are shown as the following:



```
              precision    recall  f1-score   support

         1.0       0.99      0.72      0.83       889
         2.0       0.97      1.00      0.98      8330

    accuracy                           0.97      9219
   macro avg       0.98      0.86      0.91      9219
weighted avg       0.97      0.97      0.97      9219
```



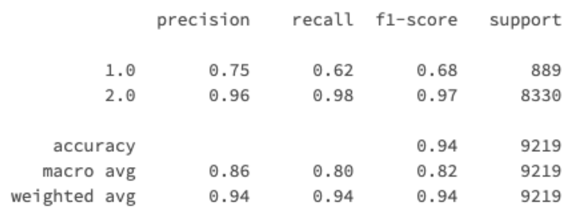
```
              precision    recall  f1-score   support

         1.0       0.99      0.74      0.85       949
         2.0       0.97      1.00      0.99      8505

    accuracy                           0.97      9454
   macro avg       0.98      0.87      0.92      9454
weighted avg       0.97      0.97      0.97      9454
```

RF - Results on Original Dataset RF - Results on Graph Feature Dataset

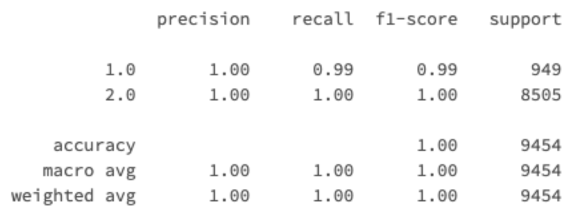
Overall, it shows that the model has a very good performance, and this could be due to several reasons. First, the Bitcoin dataset has a very good quality with handcrafted features, meaning the features are engineered by data scientist. Thus, the dataset is cleaned, and features are highly relevant to the targets. Second, we added two additional graph features, PageRank and Degrees, which represent the importance and centrality for each truncation and highly related to the licit and illicit transactions. Third, RF model is generally for fraud detection with superior performance even without the tunning. The detailed code for this section can be found in Appendix E- ML (Random Forest).

## Logistics Regression

Next, the Logistic Regression (LR) classifier is trained to compare with the RD model. LR is a great model for binary classification problem where the output is either 1 or 0 class. Similarly, the LR model was initially trained with graph feature only, which give us an accuracy and F1 score around 85% to 89%. Next, we developed the model with original dataset without adding additional graph features. The result is a bit worse than RL model with both accuracy and F1 score of around 94%. We have noticed there are some room to improve the model and then add the graph features to original dataset. Finally, the LR regression has a F1 and Accuracy between 99% to 100% after conducting grid search and cross validation. This significant increase makes the logistic regression over performed than the RF model for this application. This could be due to the fact Bitcoin transaction is a binary classification problem where LR is slightly better than RF. The PySpark classification report results can be shown as following:



```
              precision    recall  f1-score   support

         1.0       0.75      0.62      0.68       889
         2.0       0.96      0.98      0.97      8330

    accuracy                           0.94      9219
   macro avg       0.86      0.80      0.82      9219
weighted avg       0.94      0.94      0.94      9219
```



```
              precision    recall  f1-score   support

         1.0       1.00      0.99      0.99       949
         2.0       1.00      1.00      1.00      8505

    accuracy                           1.00      9454
   macro avg       1.00      1.00      1.00      9454
weighted avg       1.00      1.00      1.00      9454
```

LR - Results on Original Dataset LR - Results on Graph Feature Dataset

For the hyperparameter tuning, we tested different λ from 0, 0.01 to 0.5 (e.g., regParam), which corresponds to a regularization rate. As for penalty term, we only consider adding L2 term to the exiting model. This is because L1 tends to shrink coefficients to zero and useful for feature selection while L2 tends to shrink coefficients evenly [6]. Thus, we may consider adding a little L2 penalty to the model during the hyperparameter tuning process.

Due to the limitation of the Data Brick community edition, we only tested several regParam (lambda) with the rest of parameters remained at default values. Also, to have only L2 penalty in our model, we did not assign any value to elasticNetParam but keep it as default value of zero. This means when $\lambda > 0$ (i.e. regParam >0) and $\alpha = 0$ (i.e. elasticNetParam =0), then the penalty is an L2 penalty [7]. When $\lambda = 0$, the penalty term has no effect [7]. As a result, it shows that the LR model perform the best when $\lambda$ is close to 0 (e.g. no penalty or very small penalty). The detailed Pyspark code for this section is in Appendix F.

## 5.0 Model Comparisons

Finally, all the models are compared and summarized in the following table. It is shown that all the models tend to have very good performance (above 90% of accuracy). Label propagation has an accuracy and F1 score of around 97%, but it can only predict 45% of the unlabelled data points. Both LR and RF classifiers can predict all the unlabelled data with LR performs relatively better (99%) after adding graph features. RF tends to be better than LR before appending the graph features.

| Models | F1 Score on labeled dataset (without Graph features) | Accuracy on labeled dataset (without Graph features) | Updated Accuracy after adding Graph features | Updated F1 after adding Graph features | Portion of unlabelled data predicted? |
|---|---|---|---|---|---|
| Label Propagation | 98.6% | 97.53% | Not Applicable | Not Applicable | 45% |
| LR classifier | 94% | 94% | 99% | 99% | 100% |
| RF classifier | 97% | 97% | 97% | 97% | 100% |

## 6.0 Conclusion

In conclusion, we have implemented several methods to build the classification model for Bitcoin transaction. First, we utilized the graph label propagation techniques to predict labels, which gives a very good performance (F1: 98%), but unable to predict all the unlabelled data. Second, we then trained the models (both LR and RF) with graph features only and they both give us an F1 score of around 85%. Next, we trained the models with original dataset which boosts the F1 score to above 90% (LR: 94% and RF:97%). Finally, we append the graph features to the dataset and fit into the models which give us even better result (LR:99% and RF:97%). Overall, Logistic Regression classifier perform the best in terms of classifying the licit and illicit Bitcoin transactions among the other candidates. For the future improvement, we could even add one more graph feature, outdegree of the node, to the original dataset to train the models. This may help to further improve the model performance as the outdegree shows the importance and centrality of certain node, which is useful for training the model. In addition, a more powerful Data Brick notebook would also be helpful for the model development since it would allow us to facilitate a better hyperparameter tuning process by conducting a more sophisticated grid search on a broader range of parameters.

## Bibliography

[1] "Elliptic Data Set," Kaggle Competition, 2019. [Online]. Available: https://www.kaggle.com/ellipticco/elliptic-data-set.

[2] Databicks, "GraphFrames User Guide-Python," 21 July 2020. [Online]. Available: https://docs.databricks.com/spark/latest/graph-analysis/graphframes/user-guide-python.html.

[3] Geeksforgeeks, "Page Rank Algorithm and Implementation," 29 10 2018. [Online]. Available: https://www.geeksforgeeks.org/page-rank-algorithm-implementation/.

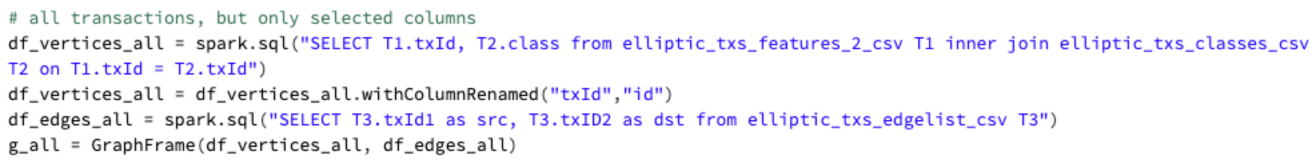
[4] M. Weber, "Anti-Money Laundering in Bitcoin: Experimenting with Graph Convolutional Networks for Financial Forensics," MIT-IBM, 31 July 2019. [Online]. Available: https://scinapse.io/papers/2965374788.

[5] N. Donges, 16 June 2019. [Online]. Available: https://builtin.com/data-science/random-forest-algorithm.

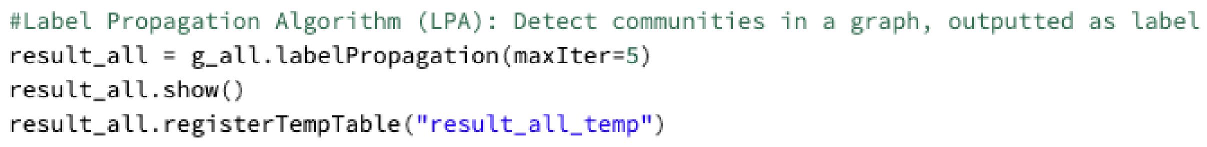
[6] T. parr, "The difference between L1 and L2 regularization," [Online]. Available: https://explained.ai/regularization/L1vsL2.html.

[7] W. Feng, "Regularization," Learning Apache Spark with Python, 05 Feb 2020. [Online]. Available: https://runawayhorse001.github.io/LearningApacheSpark/reg.html.

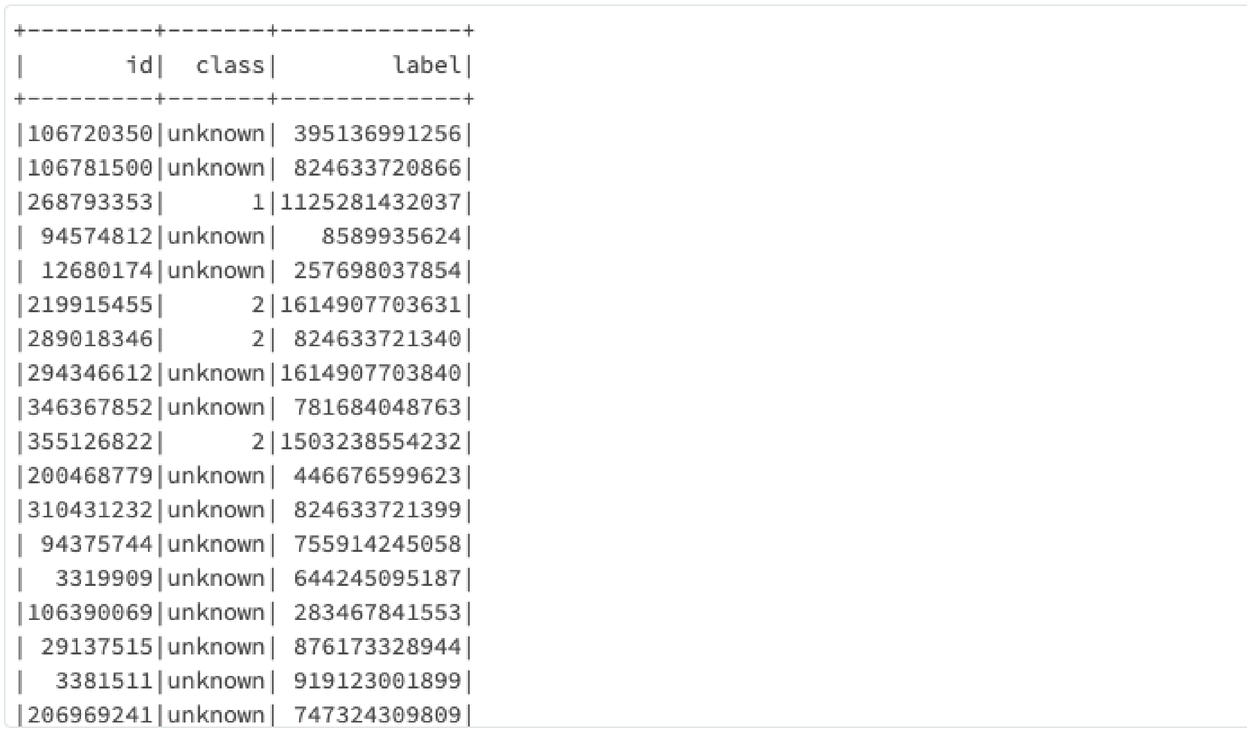
## Appendix A – Creating GraphFrame and Label Propagation



```
# all transactions, but only selected columns
df_vertices_all = spark.sql("SELECT T1.txId, T2.class from elliptic_txs_features_2_csv T1 inner join elliptic_txs_classes_csv
T2 on T1.txId = T2.txId")
df_vertices_all = df_vertices_all.withColumnRenamed("txId","id")
df_edges_all = spark.sql("SELECT T3.txId1 as src, T3.txID2 as dst from elliptic_txs_edgelist_csv T3")
g_all = GraphFrame(df_vertices_all, df_edges_all)
```



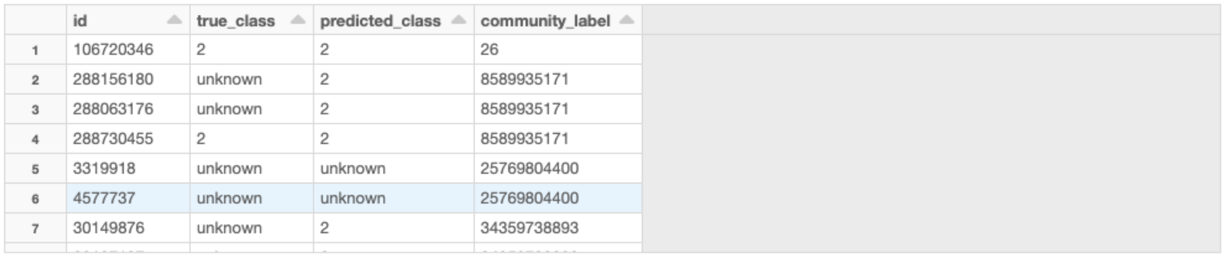
```
#Label Propagation Algorithm (LPA): Detect communities in a graph, outputted as label
result_all = g_all.labelPropagation(maxIter=5)
result_all.show()
result_all.registerTempTable("result_all_temp")
```



```
+---------+-------+-------------+
|       id|  class|        label|
+---------+-------+-------------+
|106720350|unknown| 395136991256|
|106781500|unknown| 824633720866|
|268793353|      1|1125281432037|
| 94574812|unknown|   8589935624|
| 12680174|unknown| 257698037854|
|219915455|      2|1614907703631|
|289018346|      2| 824633721340|
|294346612|unknown|1614907703840|
|346367852|unknown| 781684048763|
|355126822|      2|1503238554232|
|200468779|unknown| 446676599623|
|310431232|unknown| 824633721399|
| 94375744|unknown| 755914245058|
|  3319909|unknown| 644245095187|
|106390069|unknown| 283467841553|
| 29137515|unknown| 876173328944|
|  3381511|unknown| 919123001899|
|206969241|unknown| 747324309809|
```

## Appendix B – Prediction with Label Propagation

```
# prediction algorithm
# select label and the highest counted id classes for that label
# if unknown counts > other counts (class 1 and class 2), we prioritize and rank other counts first (class 1 and class 2)

result_classify = spark.sql("select distinct label, class, count(class) as count ,ROW_NUMBER() OVER (PARTITION BY label order
by label) as Row from result_all_temp group by label, class order by label, count desc")
result_classify.registerTempTable("result_classify_temp")
step1 = result_classify.count()
result_classify_1 = spark.sql("select distinct label, class, count, ROW_NUMBER() OVER (PARTITION BY label order by label) as
Row from result_classify_temp where class <> 'unknown' order by label, count desc")
result_classify_1.registerTempTable("result_classify_1_temp")
step1_1 = result_classify_1.count()
result_classify_1 = spark.sql("select distinct label, class as predicted_class from result_classify_1_temp where Row = '1'")
result_classify_1.registerTempTable("result_classify_1_temp")
step1_2 = result_classify_1.count()
result_classify = spark.sql("select distinct label, class as predicted_class from result_classify_temp where Row = '1'")
result_classify.registerTempTable("result_classify_temp")
step2 = result_classify.count()

# join back the ids based on community_label and write the predicted_class too
# some predict_class is unknown because there is no single id in this community that is being labelled.

predicted_results = spark.sql("select t1.id, t1.class as true_class, case when t3.predicted_class is not null then
t3.predicted_class else t2.predicted_class end as predicted_class, t1.label as community_label from result_all_temp t1 left
join result_classify_temp t2 on t1.label = t2.label left join result_classify_1_temp t3 on t1.label = t3.label")
predicted_results.registerTempTable("predicted_results_temp")
display(predicted_results)
```

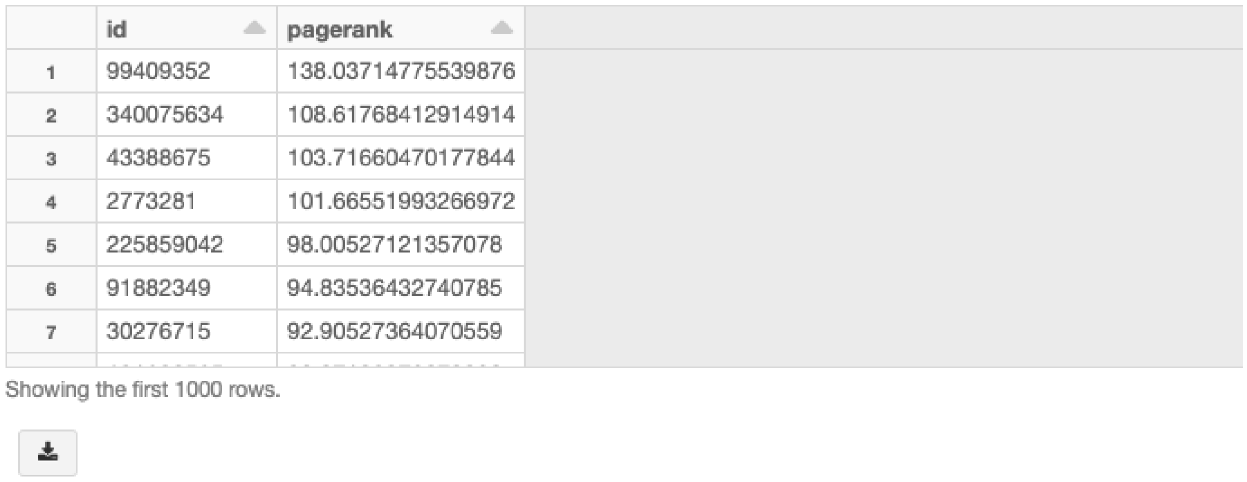


|  | id | true_class | predicted_class | community_label |
|---|---|---|---|---|
| 1 | 106720346 | 2 | 2 | 26 |
| 2 | 288156180 | unknown | 2 | 8589935171 |
| 3 | 288063176 | unknown | 2 | 8589935171 |
| 4 | 288730455 | 2 | 2 | 8589935171 |
| 5 | 3319918 | unknown | unknown | 25769804400 |
| 6 | 4577737 | unknown | unknown | 25769804400 |
| 7 | 30149876 | unknown | 2 | 34359738893 |

Showing the first 1000 rows.

## Appendix C - PageRank

```
#PageRank: Identify important vertices in a graph
ranks_all = g_all.pageRank(maxIter = 5)
display(ranks_all.vertices.select("id","pagerank").orderBy(desc("pagerank")))
```

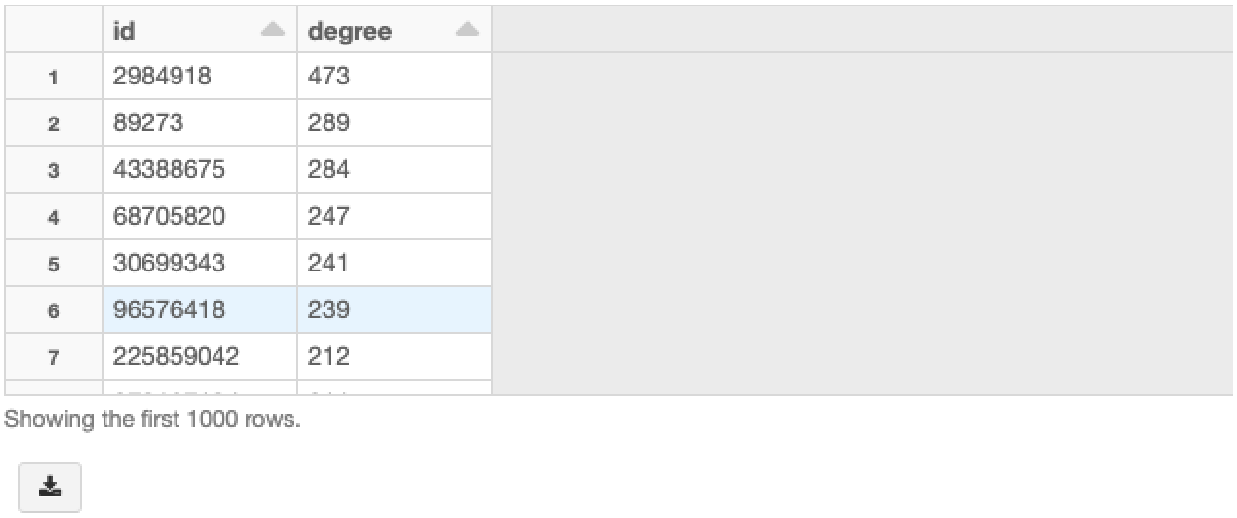


|  | id | pagerank |
|---|---|---|
| 1 | 99409352 | 138.03714775539876 |
| 2 | 340075634 | 108.61768412914914 |
| 3 | 43388675 | 103.71660470177844 |
| 4 | 2773281 | 101.66551993266972 |
| 5 | 225859042 | 98.00527121357078 |
| 6 | 91882349 | 94.83536432740785 |
| 7 | 30276715 | 92.90527364070559 |

Showing the first 1000 rows.

## Appendix D - Degrees

```
degrees_all = g_all.degrees.sort(desc("degree"))
display( degrees_all )
```



|  | id | degree |
|---|---|---|
| 1 | 2984918 | 473 |
| 2 | 89273 | 289 |
| 3 | 43388675 | 284 |
| 4 | 68705820 | 247 |
| 5 | 30699343 | 241 |
| 6 | 96576418 | 239 |
| 7 | 225859042 | 212 |

Showing the first 1000 rows.

## Appendix E – ML (Random Forest)

### Random Forest on Original Dataset

```
# Reference: https://towardsdatascience.com/first-time-machine-learning-model-with-pyspark-3684cf406f54
from pyspark.ml.classification import RandomForestClassifier
rf = RandomForestClassifier(labelCol='class', featuresCol='features')

# Fit a pipline
from pyspark.ml import Pipeline
pipeline = Pipeline(stages=[assembler, rf])

# Hyperparamet Grid (Reference: https://www.silect.is/blog/2019/4/2/random-forest-in-spark-ml)
from pyspark.ml.tuning import ParamGridBuilder
import numpy as np

paramGrid = ParamGridBuilder() \
    .addGrid(rf.numTrees,[5, 10] ) \
    .addGrid(rf.maxDepth, [5, 10]) \
    .build()

# Cross Validation
from pyspark.ml.tuning import CrossValidator
from pyspark.ml.evaluation import BinaryClassificationEvaluator

crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=BinaryClassificationEvaluator(labelCol="class"),parallelism = 3,
                          numFolds=3)

# Fit the model on Training dataset
import mlflow
cvModel = crossval.fit(training_data)

# Evualtaion matrix report
import sklearn
y_true = rf_predictions.select(['class']).collect()
y_pred = rf_predictions.select(['prediction']).collect()

from sklearn.metrics import classification_report, confusion_matrix
print(classification_report(y_true, y_pred))
```

```
              precision    recall  f1-score   support

         1.0       0.99      0.72      0.83       889
         2.0       0.97      1.00      0.98      8330

    accuracy                           0.97      9219
   macro avg       0.98      0.86      0.91      9219
weighted avg       0.97      0.97      0.97      9219
```

### Random Forest on Graph Feature Dataset

```
# Reference: https://towardsdatascience.com/first-time-machine-learning-model-with-pyspark-3684cf406f54
from pyspark.ml.classification import RandomForestClassifier

# train and test data split
(training_data, test_data) = transformed_data.randomSplit([0.8,0.2], seed =2020)

# Creat a RF model
rf = RandomForestClassifier(labelCol='class', featuresCol='features')
```

```python
# Hyperparamet Grid (Reference: https://www.silect.is/blog/2019/4/2/random-forest-in-spark-ml)
from pyspark.ml.tuning import ParamGridBuilder
import numpy as np

paramGrid = ParamGridBuilder() \
    .addGrid(rf.numTrees,[5, 10] ) \
    .addGrid(rf.maxDepth, [5, 10]) \
    .build()
```

```python
# Cross Validation
from pyspark.ml.tuning import CrossValidator
#from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.evaluation import BinaryClassificationEvaluator
crossval = CrossValidator(estimator=rf, estimatorParamMaps=paramGrid, evaluator=BinaryClassificationEvaluator(labelCol="class"),parallelism = 3,
numFolds=2)
```

```python
# Fit the model on Training dataset
import mlflow
cvModel = crossval.fit(training_data)
```

```python
# Make precision on testing dataset dfdf
rf_predictions = cvModel.transform(test_data)
```

```python
# Evualtaion matrix report
import sklearn
y_true = rf_predictions.select(['class']).collect()
y_pred = rf_predictions.select(['prediction']).collect()

from sklearn.metrics import classification_report, confusion_matrix
print(classification_report(y_true, y_pred))
```

```
              precision    recall  f1-score   support

         1.0       0.99      0.74      0.85       949
         2.0       0.97      1.00      0.99      8505

    accuracy                           0.97      9454
   macro avg       0.98      0.87      0.92      9454
weighted avg       0.97      0.97      0.97      9454
```

```python
# Extract the hyperparameters
bestPipeline_RF = cvModel.bestModel
bestParams_RF = cvModel.extractParamMap()
bestPipeline_RF
```

```
Out[34]: RandomForestClassificationModel: uid=RandomForestClassifier_2734116d485d, numTrees=10, numClasses=3, numFeatures=168
```

# Appendix F – ML (Logistic Regression)

## Logistic Regression on Original Dataset

```
# Reference: https://towardsdatascience.com/first-time-machine-learning-model-with-pyspark-3684cf406f54
from pyspark.ml.classification import LogisticRegression

# Createa logistic Model
lr = LogisticRegression(featuresCol = 'features', labelCol = 'class', maxIter=10)

# Hyperparamet Grid (Reference: https://www.silect.is/blog/2019/4/2/random-forest-in-spark-ml)
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# train and test data split
(training_data, test_data) = data_float.randomSplit([0.8,0.2], seed =2020)

# Fit a pipline
pipeline_lr = Pipeline(stages=[assembler, lr])

# Define a Hyperparamter Grid
paramGrid_lr = ParamGridBuilder().addGrid(lr.regParam, [0,0.01, 0.5]).build()

#Cross Validation
cv_lr = CrossValidator(estimator=pipeline_lr,estimatorParamMaps=paramGrid_lr,evaluator=BinaryClassificationEvaluator(labelCol="class"), numFolds=3)

# Fit the logistic mdoel on training dataset
cvModel_lr = cv_lr.fit(training_data)

# Make precision on testing dataset
lr_predictions = cvModel_lr.transform(test_data)

import sklearn
y_true = lr_predictions.select(['class']).collect()
y_pred = lr_predictions.select(['prediction']).collect()

from sklearn.metrics import classification_report, confusion_matrix
print(classification_report(y_true, y_pred))
```

```
              precision    recall  f1-score   support

         1.0       0.75      0.62      0.68       889
         2.0       0.96      0.98      0.97      8330

    accuracy                           0.94      9219
   macro avg       0.86      0.80      0.82      9219
weighted avg       0.94      0.94      0.94      9219
```

## Logistic Regression on Graph Feature Dataset

```
# Reference: https://towardsdatascience.com/first-time-machine-learning-model-with-pyspark-3684cf406f54
from pyspark.ml.classification import LogisticRegression

# Hyperparamet Grid (Reference: https://www.silect.is/blog/2019/4/2/random-forest-in-spark-ml)
# train and test data split
(training_data, test_data) = transformed_data.randomSplit([0.8,0.2], seed =2020)

# Createa logistic Model
lr = LogisticRegression(featuresCol = 'features', labelCol = 'class', maxIter=10)

# Define a Hyperparamter Grid
paramGrid_lr = ParamGridBuilder().addGrid(lr.regParam, [0,0.01, 0.5]).build() # regParam corresponds to lamb
penalty term has no eﬀect, and the estimates produced by ridge regression will be equal to least squares

#Cross Validation
from pyspark.ml.evaluation import BinaryClassificationEvaluator
cv_lr = CrossValidator(estimator=lr, estimatorParamMaps=paramGrid_lr,evaluator=BinaryClassificationEvaluator(labelCol="class"),numFolds=3)

# Fit the logistic mdoel on training dataset
cvModel_lr = cv_lr.fit(training_data)

# Make precision on testing dataset
lr_predictions = cvModel_lr.transform(test_data)
```

```
import sklearn
y_true = lr_predictions.select(['class']).collect()
y_pred = lr_predictions.select(['prediction']).collect()

from sklearn.metrics import classification_report, confusion_matrix
print(classification_report(y_true, y_pred))
```

```
              precision    recall  f1-score   support

         1.0       1.00      0.99      0.99       949
         2.0       1.00      1.00      1.00      8505

    accuracy                           1.00      9454
   macro avg       1.00      1.00      1.00      9454
weighted avg       1.00      1.00      1.00      9454
```

```
# Extract hte hyperparameters for the best Logistric model
bestPipeline_lr = cvModel_lr.bestModel
bestParams_lr = cvModel_lr.extractParamMap()

bestPipeline_lr
```

```
Out[46]: LogisticRegressionModel: uid=LogisticRegression_3f15c7ead6ff, numClasses=3, numFeatures=168
```