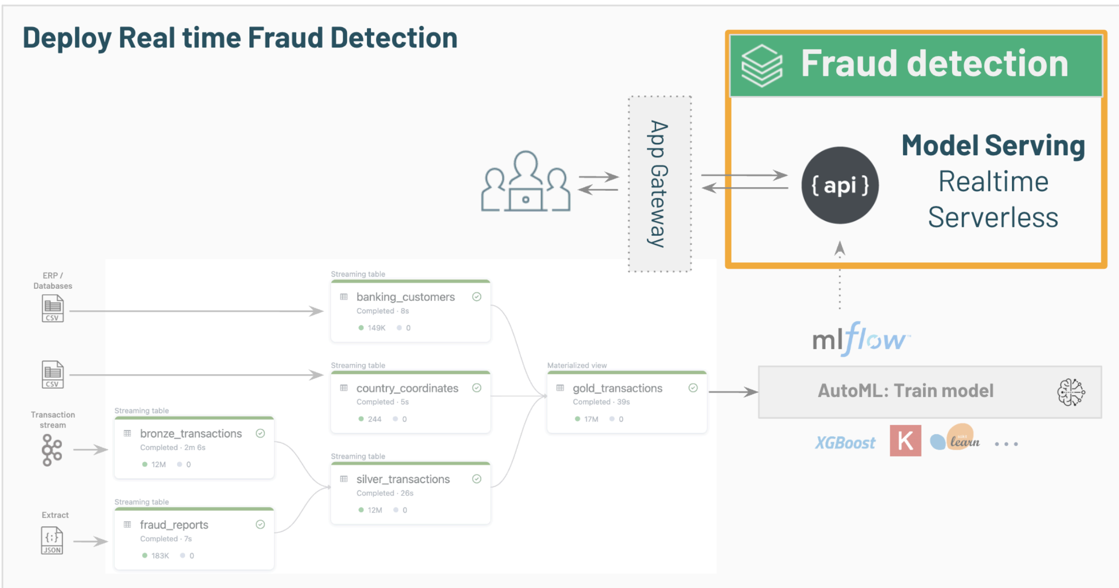
Banking Transaction Fraud Model

by Nathan Alanmanou

Table of Contents

Summary



Companies value the utility that data provides in helping inform business decisions and making processes more efficient. We can make the most of our data by laying out the full process of transformations that take place from the source to the end product that is directly applicable to business needs.

For this project, a banking company wants to visualize their transaction data and create a prediction model so that they can better understand the factors that lead to fraudulent transactions. By collecting and analyzing data in real time, this project helps financial institutions identify potentially fraudulent transactions as they occur, allowing for immediate action to prevent financial loss.

This project uses Python, SQL, Sci-kit learn, Pandas, and an assortment of technology native to the Databricks platform to handle data ingestion, storage, transformations, model building and deployment, and data analysis. The results are visualized in a ReactJS application at nathanalanmanou.com. This project also aims to follow industry best practices through workflows for orchestration, delta lake for data integrity through ACID properties, medallion architecture to curate data from raw to refined forms, and of course, documentation to help external users understand what's going on, which opens the door for further improvement.

System Design (why Databricks?)

Companies increasingly recognize data as a focal point for their operations. Data volume and variety continues to balloon, which highlights stress points in legacy procedures. Databricks addresses many of these issues and then some, providing more opportunity than ever to turn data into tangible value. This section covers some of the broader benefits, and the chapters following will incorporate explanations of further benefits through the project use case.

Unified Platform

Databricks offers a single platform for data engineering, data science, machine learning, and analytics. Previously projects required a hodge-podge of different services, which meant more time having to learn and configure each service, harder time collaborating, and overall opened up a bunch of inefficiencies.

Scalability

Databricks' custom technology and its Apache Spark foundation provide immense support for scalability without significant degradation in performance.

Declarative Frameworks

Declarative frameworks allow the user to state what they want to happen, and the engine will figure out the optimal way to achieve it, abstracting away the burden of figuring it out. Delta Live Tables allow for efficient and performant data pipelines, while there's still structured streaming available for a more manual option.

Governance

Unity Catalog makes everything incredibly organized, and makes data asset administration very easy. This project uses personally identifiable information, and so governance plays a crucial role in ensuring that the data and analytics processes are secure, compliant, and efficiently managed.

Data Extraction

Data Source

This project's source data is built with PaySim, an open source banking transactions simulator. PaySim simulates mobile money transactions based on a sample of real transactions extracted from one month of financial logs from a mobile money service implemented in an African country.

Bronze Transactions

The delta live table (DLT) bronze_transactions represents the historical banking transaction to be trained on fraud detection

Column	Type	Column	Type
amount	double	newBalanceDest	double
countryDest	string	newBalanceOrig	double
countryOrig	string	oldBalanceDest	double
customer_id	string	oldBalanceOrig	double
id	string	step	bigint
isUnauthorizedOv...	bigint	type	string
nameDest	string	_rescued_data	string
nameOrig	string		

Banking Customers

The DLT banking_customers represents customer data coming from csv files

Column	Type	Column	Type
firstname	string	last_country_logg...	string
lastname	string	creation_date	string
email	string	last_activity_date	string
address	string	age_group	double
country	string	id	string
last_country_logg...	string	_rescued_data	string

Country Coordinates

The DLT country_coordinates provides additional information that will be merged

Column	Type	Column	Type
country	string	lat_avg	string
alpha2_code	string	long_avg	string
alpha3_code	string	_rescued_data	string
numeric_code	string		

Fraud Reports

The DLT fraud_reports represents the reported fraudulent transactions

Column	Type
is_fraud	string
id	string
_rescue...	string

Data Transformation

Silver Table

The silver_transactions table will incrementally consume data from bronze_transactions, and will:

- Clean up the codes of the countries of origin and destination (removing the "--")
- Calculate the difference between the Originating and Destination Balances.
- Enforce id and customer_id columns

```
CREATE LIVE TABLE silver_transactions (

  CONSTRAINT correct_data EXPECT (id IS NOT NULL),

  CONSTRAINT correct_customer_id EXPECT (customer_id IS NOT NULL)

)

AS

  SELECT * EXCEPT(countryOrig, countryDest, t._rescued_data,
f._rescued_data),

          regexp_replace(countryOrig, "\-\-", "") as countryOrig,

          regexp_replace(countryDest, "\-\-", "") as countryDest,

          newBalanceOrig - oldBalanceOrig as diffOrig,

          newBalanceDest - oldBalanceDest as diffDest

FROM STREAM(live.bronze_transactions) t

  LEFT JOIN live.fraud_reports f using(id)
```

Gold Table

The table gold_transactions integrates transactional data with customer and geographical information, creating a comprehensive dataset that can be used for analysis and machine learning training:

```
CREATE LIVE TABLE gold_transactions (

  CONSTRAINT amount_decent EXPECT (amount > 10)

)
```

```
AS

  SELECT t.* EXCEPT(countryOrig, countryDest, is_fraud), c.* EXCEPT(id,
_rescued_data),

          boolean(coalesce(is_fraud, 0)) as is_fraud,

          o.alpha3_code as countryOrig, o.country as countryOrig_name,
o.long_avg as countryLongOrig_long, o.lat_avg as countryLatOrig_lat,

          d.alpha3_code as countryDest, d.country as countryDest_name,
d.long_avg as countryLongDest_long, d.lat_avg as countryLatDest_lat

FROM live.silver_transactions t

  INNER JOIN live.country_coordinates o ON t.countryOrig=o.alpha3_code

  INNER JOIN live.country_coordinates d ON t.countryDest=d.alpha3_code

  INNER JOIN live.banking_customers c ON c.id=t.customer_id
```

Model Development

Gold transactions were used for training using AutoML

Select Supported Columns

```
from databricks.automl_runtime.sklearn.column_selector import
ColumnSelector

supported_cols = ["step", "countryLatOrig_lat", "diffDest",
"oldBalanceOrig", "countryOrig_name", "age_group", "type",
"countryLongOrig_long", "last_country_logged", "amount", "diffOrig",
"countryOrig", "newBalanceDest", "oldBalanceDest", "newBalanceOrig",
"countryLatDest_lat", "isUnauthorizedOverdraft", "countryDest",
"countryDest_name", "country", "countryLongDest_long"]

col_selector = ColumnSelector(supported_cols)
```

Boolean Columns

For each column, impute missing values and then convert into ones and zeros

```
from sklearn.compose import ColumnTransformer

from sklearn.impute import SimpleImputer

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import FunctionTransformer

from sklearn.preprocessing import OneHotEncoder as SklearnOneHotEncoder




bool_imputers = []



bool_pipeline = Pipeline(steps=[

    ("cast_type", FunctionTransformer(lambda df: df.astype(object))),

    ("imputers", ColumnTransformer(bool_imputers,
remainder="passthrough")),
```

```
    ("onehot", SklearnOneHotEncoder(handle_unknown="ignore",
drop="first")),

])


bool_transformers = [("boolean", bool_pipeline,
["isUnauthorizedOverdraft"])]
```

Numerical Columns

Missing values for numerical columns are imputed with mean by default

```
from sklearn.compose import ColumnTransformer

from sklearn.impute import SimpleImputer

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import FunctionTransformer, StandardScaler


num_imputers = []

num_imputers.append(("impute_mean", SimpleImputer(), ["age_group",
"amount", "diffDest", "diffOrig", "isUnauthorizedOverdraft",
"newBalanceDest", "newBalanceOrig", "oldBalanceDest", "oldBalanceOrig",
"step"]))


numerical_pipeline = Pipeline(steps=[

    ("converter", FunctionTransformer(lambda df: df.apply(pd.to_numeric,
errors="coerce"))),

    ("imputers", ColumnTransformer(num_imputers)),

    ("standardizer", StandardScaler()),

])
```

```
numerical_transformers = [("numerical", numerical_pipeline, ["step",
"isUnauthorizedOverdraft", "diffOrig", "newBalanceDest", "diffDest",
"oldBalanceDest", "amount", "newBalanceOrig", "oldBalanceOrig",
"age_group"])]
```

Low-cardinality categoricals

Convert each low-cardinality categorical column into multiple binary columns through one-hot encoding. For each input categorical column (string or numeric), the number of output columns is equal to the number of unique values in the input column.

```
from databricks.automl_runtime.sklearn import OneHotEncoder

from sklearn.compose import ColumnTransformer

from sklearn.impute import SimpleImputer

from sklearn.pipeline import Pipeline



one_hot_imputers = []



one_hot_pipeline = Pipeline(steps=[

    ("imputers", ColumnTransformer(one_hot_imputers,
remainder="passthrough")),

    ("one_hot_encoder", OneHotEncoder(handle_unknown="indicator")),

])



categorical_one_hot_transformers = [("onehot", one_hot_pipeline,
["age_group", "country", "countryDest", "countryDest_name",
"countryLatDest_lat", "countryLatOrig_lat", "countryLongDest_long",
"countryLongOrig_long", "countryOrig", "countryOrig_name",
"last_country_logged", "type"])]

from sklearn.compose import ColumnTransformer
```

```
transformers = bool_transformers + numerical_transformers +
categorical_one_hot_transformers


preprocessor = ColumnTransformer(transformers, remainder="passthrough",
sparse_threshold=1)
```

Train - Validation - Test Split (60 - 20 - 20)

```
# AutoML completed train - validation - test split internally and used
_automl_split_col_624d to specify the set

split_col = [c for c in df_loaded.columns if
c.startswith('_automl_split_col')][0]


split_train_df = df_loaded.loc[df_loaded[split_col] == "train"]

split_val_df = df_loaded.loc[df_loaded[split_col] == "val"]

split_test_df = df_loaded.loc[df_loaded[split_col] == "test"]


# Separate target column from features and drop _automl_split_col_xxx

X_train = split_train_df.drop([target_col, split_col], axis=1)

y_train = split_train_df[target_col]


X_val = split_val_df.drop([target_col, split_col], axis=1)

y_val = split_val_df[target_col]


X_test = split_test_df.drop([target_col, split_col], axis=1)

y_test = split_test_df[target_col]
```

Define The Objective Function

```
import mlflow

from mlflow.models import Model, infer_signature, ModelSignature

from mlflow.pyfunc import PyFuncModel

from mlflow import pyfunc

import sklearn

from sklearn import set_config

from sklearn.pipeline import Pipeline



from hyperopt import hp, tpe, fmin, STATUS_OK, Trials



def objective(params):

  with mlflow.start_run(experiment_id=run['experiment_id']) as
mlflow_run:

    skrf_classifier = RandomForestClassifier(n_jobs=-1, **params)



    model = Pipeline([

        ("column_selector", col_selector),

        ("preprocessor", preprocessor),

        ("classifier", skrf_classifier),

    ])



    # Enable automatic logging of input samples, metrics, parameters, and
models

    mlflow.sklearn.autolog(
```

```
        log_input_examples=True,

        silent=True)


    model.fit(X_train, y_train)




    # Log metrics for the training set

    mlflow_model = Model()

    pyfunc.add_to_model(mlflow_model, loader_module="mlflow.sklearn")

    pyfunc_model = PyFuncModel(model_meta=mlflow_model, model_impl=model)

    X_train[target_col] = y_train

    training_eval_result = mlflow.evaluate(

        model=pyfunc_model,

        data=X_train,

        targets=target_col,

        model_type="classifier",

        evaluator_config = {"log_model_explainability": False,

                            "metric_prefix": "training_" , "pos_label": 1
}

    )

    skrf_training_metrics = training_eval_result.metrics

    # Log metrics for the validation set

    X_val[target_col] = y_val

    val_eval_result = mlflow.evaluate(
```

```
    model=pyfunc_model,
    data=X_val,
    targets=target_col,
    model_type="classifier",
    evaluator_config = {"log_model_explainability": False,
                        "metric_prefix": "val_" , "pos_label": 1 }
)
skrf_val_metrics = val_eval_result.metrics
# Log metrics for the test set
X_test[target_col] = y_test
test_eval_result = mlflow.evaluate(
    model=pyfunc_model,
    data=X_test,
    targets=target_col,
    model_type="classifier",
    evaluator_config = {"log_model_explainability": False,
                        "metric_prefix": "test_" , "pos_label": 1 }
)
skrf_test_metrics = test_eval_result.metrics

loss = skrf_val_metrics["val_f1_score"]

# Truncate metric key names so they can be displayed together
```

```
    skrf_val_metrics = {k.replace("val_", ""): v for k, v in
skrf_val_metrics.items()}

    skrf_test_metrics = {k.replace("test_", ""): v for k, v in
skrf_test_metrics.items()}


    return {

      "loss": loss,

      "status": STATUS_OK,

      "val_metrics": skrf_val_metrics,

      "test_metrics": skrf_test_metrics,

      "model": model,

      "run": mlflow_run,

    }
```

Configure Hyperparameter Search Space

```
space = {

  "bootstrap": False,

  "criterion": "entropy",

  "max_depth": 6,

  "max_features": 0.367463146587338,

  "min_samples_leaf": 0.4667519090518267,

  "min_samples_split": 0.10899077622524798,

  "n_estimators": 19,

  "random_state": 441215911,

}
```

Run Trials

```
trials = Trials()

fmin(fn=objective,

     space=space,

     algo=tpe.suggest,

     max_evals=1,  # Increase this when widening the hyperparameter
search space.

     trials=trials)



best_result = trials.best_trial["result"]

model = best_result["model"]

mlflow_run = best_result["run"]



display(

  pd.DataFrame(

    [best_result["val_metrics"], best_result["test_metrics"]],

    index=["validation", "test"]))



set_config(display="diagram")

model
```

Model Evaluation / Deployment

Metrics for the best trial:

	Train	Validation	Test
f1_score	0.949	0.950	0.949
recall_score	0.950	0.951	0.952
roc_auc	0.981	0.981	0.981
false_negatives	5351.000	1756.000	1729.000
false_positives	5557.000	1845.000	1898.000
example_count	322429.000	107123.000	107158.000
precision_score	0.948	0.949	0.947
true_positives	102238.000	34130.000	34089.000
precision_recall_auc	0.956	0.956	0.956
true_negatives	209283.000	69392.000	69442.000
log_loss	0.240	0.240	0.241
score	0.966	0.966	0.966
accuracy_score	0.966	0.966	0.966

Deployment currently in the works, and will be implemented soon under the ‘Model’ tab of nathanalanmanou.com

Conclusion

This transaction fraud model project represents a significant leap forward in the fight against financial fraud, leveraging the power of the Databricks Lakehouse architecture to unify data management and analytics. By integrating data ingestion, storage, and advanced analytics within a single platform, this project enables real-time fraud detection and prevention capabilities that are both scalable and efficient. The use of Delta Lake ensures data reliability and performance, while Delta Live Tables simplify the operational complexity of data pipelines, allowing for seamless data transformation and quality assurance.

The project's end-to-end approach, from data ingestion to machine learning model deployment, exemplifies how modern data platforms like Databricks can drive significant improvements in fraud detection processes, ultimately safeguarding financial transactions and enhancing customer trust.