ChatGPT

Health Insurance Fraud Detection Project Blueprint

Project Overview

Health insurance fraud costs billions and involves falsified claims by providers, making detection critical [1]. In this project, we'll build an end-to-end fraud detection system using a public **Kaggle Healthcare Provider Fraud Detection** dataset. The goal is to **predict potentially fraudulent providers based on claims data** and identify which factors signal fraud [2]. We will use modern tools and best practices (circa 2025) to make this project portfolio-quality:

- **Databricks** – A unified analytics platform providing managed Spark clusters and collaborative notebooks (we'll use the 14-day free trial). Databricks lets you prepare data at massive scale and train/deploy ML models in one place [3].
- **Snowflake** (optional) – A cloud data warehouse. We'll mention how Snowflake could be integrated for data storage or queries, but it's not required to complete the project.
- **MLflow** – An open-source experiment tracking and model registry tool (integrated with Databricks). MLflow will log our parameters, metrics, and models [4] to keep track of experiments.
- **SHAP** – SHapley Additive exPlanations, a framework to interpret model predictions by quantifying each feature's contribution [5]. We'll use SHAP for model explainability.
- **Streamlit** – An open-source Python framework for building interactive data apps with just a few lines of code [6]. We'll create a Streamlit dashboard to showcase the model's predictions and insights, and deploy it on Streamlit Cloud for free.
- **Embedding/Vector Search** (advanced, optional) – We'll discuss using embeddings (numeric vector representations of data) and vector similarity search to enhance fraud detection. This is a cutting-edge technique in 2025 to find anomalous patterns by measuring distances between data points in vector space [7].

Throughout the project, we will emphasize clean, reusable code and an organized repository structure. Each major step is broken down with clear sub-steps so you can follow along from start to finish, even with minimal prior setup knowledge.

Step 1: Environment Setup (Databricks & GitHub Integration)

1. **Create a Databricks Account:** Sign up for the Databricks free trial (premium 14-day) using the *Express setup* (requires only an email). This gives you a managed workspace with limited free compute time [8]. Once signed up, navigate to your Databricks workspace (a web-based environment).

2. **Launch a Compute Cluster:** In Databricks, create a new cluster. Choose a Databricks Runtime (select a version with **ML** – e.g., "Databricks Runtime 13.x ML" – which comes preloaded with ML libraries like scikit-learn, MLflow, etc.). Use a modest instance type (like `4 cores, 8 GB RAM`) to start; this

should handle the Kaggle dataset. **Start the cluster** and wait for it to be up (status will turn to "Running").

3. **Import Kaggle Dataset into Databricks:** Download the **"Healthcare Provider Fraud Detection Analysis"** dataset from Kaggle. You have two options:
   a. **Using Kaggle API:** On your local machine, obtain your Kaggle API credentials (a `kaggle.json` with your username and key). In a Databricks notebook, install the Kaggle CLI (`%pip install kaggle`) and use it to download the dataset: e.g., `!kaggle datasets download -d rohitrox/healthcare-provider-fraud-detection-analysis`. This will download a ZIP file. Use Databricks file system utilities (`dbutils.fs`) to move and unzip this file into DBFS (Databricks File System), such as `/FileStore/fraud_data/`.
   b. **Manual Upload:** Alternatively, download the dataset ZIP manually from Kaggle, then in Databricks use the **Upload Data** interface to upload the files to `/FileStore/` in your workspace. This is feasible if the dataset is not too large.

4. **Verify Data Files:** The dataset comprises multiple CSV files: inpatient claims, outpatient claims, beneficiary details, and provider labels [9] [10]. For example: `InpatientData.csv`, `OutpatientData.csv`, `BeneficiaryData.csv`, `Train.csv` (with Provider IDs and fraud labels), and `Test.csv` (Provider IDs to predict). List the files in the directory to confirm they're present. If needed, use `pandas.read_csv` or Spark's `spark.read.csv` to peek at the first few lines of each file.

5. **(Optional) Set Up Snowflake Connection:** If you have a Snowflake trial or account and want to use it: load the CSVs into Snowflake (e.g., using Snowflake's UI or python connector). Then, in Databricks, you can use the Spark-Snowflake connector to read the data. For example, add the Snowflake Spark connector library to your cluster, and use:

   ```
   snowflake_options = {
       "sfUrl": "<YOUR_SNOWFLAKE_URL>",
       "sfUser": "<USER>", "sfPassword": "<PASSWORD>",
       "sfDatabase": "<DB>", "sfSchema": "<SCHEMA>", "sfWarehouse":
   "<WAREHOUSE>"
   }
   df_inp = spark.read.format("snowflake").options(**snowflake_options)\
             .option("dbtable", "INPATIENT_CLAIMS").load()
   ```

   Databricks supports reading/writing to Snowflake via the `spark.read.format("snowflake")` DataSource [11]. Ensure you store credentials securely (Databricks Secrets if on a paid plan, or skip Snowflake for now if not needed).

6. **GitHub Integration (Databricks Repos):** For version control, use **Databricks Repos** to sync with GitHub. In the Databricks sidebar, find "Repos", click **Add Repo** and authenticate with your GitHub account. Create a new GitHub repository for this project (e.g., "healthcare-fraud-detection") and link it. Databricks will clone the repo into your workspace. Now you can develop notebooks and code inside this repo and commit/push directly from Databricks. This ensures your work (notebooks,

scripts) is tracked on GitHub. *Tip:* You can create folders like `/notebooks` and add notebooks there. The Repos feature lets you manage branches, push, pull, and even do merges all within Databricks [12].

7. **Project Structure on Databricks:** Within your Repo, plan a structure (you can refine later): perhaps a `notebooks/` directory for exploratory work and a `src/` directory for Python modules. In Databricks, notebooks can be treated as files in the repo. We will follow this organization as we proceed, ensuring that by the end, code is modular and ready to be used in a Streamlit app.

Step 2: Data Ingestion and Preparation

1. **Load Data into Notebooks:** Start a new Notebook (e.g., `notebooks/01_data_ingestion_eda`). Use either Spark DataFrames or Pandas to load the CSVs from DBFS. For example, using Pandas:

```
import pandas as pd
inpatient_df = pd.read_csv("/dbfs/FileStore/fraud_data/InpatientData.csv")
outpatient_df = pd.read_csv("/dbfs/FileStore/fraud_data/
OutpatientData.csv")
bene_df = pd.read_csv("/dbfs/FileStore/fraud_data/BeneficiaryData.csv")
train_df = pd.read_csv("/dbfs/FileStore/fraud_data/Train.csv")
```

If the data is large, prefer Spark:

```
inpatient_sdf = spark.read.csv("dbfs:/FileStore/fraud_data/
InpatientData.csv", header=True, inferSchema=True)
```

and similarly for other files. (Databricks auto-mounts `/FileStore` on DBFS.)

2. **Understand the Dataset Structure:** The Kaggle dataset is relational:

3. **InpatientData**: Claims for hospital-admitted patients (inpatient visits) – includes provider ID, claim amounts, diagnoses, hospital stay duration, etc [13].
4. **OutpatientData**: Claims for outpatient visits (no admission) – provider ID, claim info for surgeries, therapies, etc [14].
5. **BeneficiaryData**: Beneficiary (patient) demographics – age/DOB, gender, chronic conditions, etc [15].

6. **Train**: A list of **Provider IDs** labeled `0` (not fraudulent) or `1` (fraudulent) for training [10]. (The **Test** file contains Provider IDs without labels – since this is a project, we can use only the train set and ignore test or treat it as data to predict on later.)

7. **Merge Dataframes:** Our prediction target is at the **provider level** (each provider is either fraudulent or not). However, the claims data is at the **claim level** (multiple claims per provider). We need to aggregate or derive features per provider from the claims. Steps to merge:

8. **Combine Inpatient and Outpatient:** Append these two datasets or handle them separately but similarly. You may add a column in each to indicate inpatient vs outpatient, then union them into a single `claims_df` for easier processing. Both have a `Provider` identifier that links to the Provider.
9. **Join Beneficiary info:** Claims data usually has a `BeneID` or similar for patient. We can join `claims_df` with `bene_df` on the beneficiary ID to bring in patient attributes (like age or conditions count). This way, features like "average patient age for provider's claims" could be derived.
10. **Aggregate by Provider:** Now group by `Provider`. Create features that summarize each provider's claims. For example:
    - Total number of claims (inpatient_count, outpatient_count).
    - Total amount billed by provider, average claim amount.
    - Number of distinct patients treated by provider.
    - Any abnormal patterns: e.g., percentage of claims for expensive procedures, average length of hospital stay (for inpatients), etc.
    - Demographics: average patient age, proportion of patients with certain chronic conditions (you can derive from beneficiary data).
    - You might also include binary flags like whether the provider mostly does inpatient vs outpatient (ratio).

11. Use Spark SQL or Pandas groupby to compute these features. Ensure the resulting DataFrame has one row per `Provider` with all aggregated feature columns.

12. **Prepare the Labeled Dataset:** Merge the aggregated provider features with `train_df` (the labels). `train_df` has two columns: Provider and whether they are fraud (1) or not (0). After joining, you have a master DataFrame `df_model` with features and the target `FraudLabel`. Drop the Provider ID column after merging (we won't use ID as a feature).

13. **Handle Missing and Data Cleaning:** Check for missing values or obviously incorrect data in the features:

14. If some providers have missing beneficiary info (maybe a provider had claims but no beneficiary data merged – unlikely if data is complete), decide how to handle (could fill with mean or a special value).
15. If any numeric features have NaNs, fill with 0 or mean as appropriate.

16. Outliers: note any extreme values (like an extremely high number of claims). We might not remove outliers (they could be genuine fraud signals), but it's good to note them.

17. **Class Imbalance Check:** Calculate how many providers are labeled fraud vs not fraud. If fraud cases are much fewer, we have an imbalance. For instance, if only ~10% providers are fraud, the model may need techniques to handle imbalance (we will address this during modeling with metrics and possibly resampling). Note the fraud percentage for reference.

18. **Split Data into Train/Validation:** Since the original "Train.csv" was the full labeled set, we should create our own split for modeling. Use an 80/20 or 70/30 split of providers for training and hold-out testing. Ensure the split is stratified (keep fraud ratio similar in both sets). For example, in Pandas:

```
from sklearn.model_selection import train_test_split
train_data, val_data = train_test_split(df_model, test_size=0.2,
stratify=df_model.FraudLabel, random_state=42)
```

Alternatively, use Spark random split (though stratification is easier with Pandas/Scikit-learn).

19. **Save Processed Data (Optional):** It's a good idea to save the prepared dataset (features and label) for reuse. You can save it as a CSV or Parquet in DBFS (e.g., `df_model.to_parquet("/dbfs/FileStore/fraud_data/df_model.parquet")`). This way, later steps or the Streamlit app could load this directly if needed.

By the end of Step 2, you have a clean dataset of providers with engineered features and a fraud label, ready for analysis and modeling. Document any assumptions or interesting observations (e.g., you might notice some providers have extremely high claim counts – potentially fraud signals).

Step 3: Exploratory Data Analysis (EDA)

1. **Class Distribution:** Start EDA by checking the balance of the classes. Calculate the percentage of providers labeled fraudulent. For example, "X% of providers in the training data are flagged as fraud." If the dataset shows a significant imbalance (e.g., only 10% fraud), note that this will influence modeling (we might focus on recall or use techniques to handle imbalance).

2. **Statistical Summaries:** Compare summary statistics for fraudulent vs non-fraudulent providers. Using the aggregated features:

3. For numeric features (like total claims, avg claim cost, distinct patients, etc.), compute the mean and median for fraud providers vs non-fraud. Often, fraudsters might have unusually high values in certain metrics (e.g., much higher average claim amount or number of procedures).

4. You can use groupby in Pandas: `df_model.groupby('FraudLabel').mean()` to see differences. Or plot boxplots for features by class.

5. **Visualizations:** Create several plots to visualize the data:

6. **Histogram or Boxplot of Key Features:** e.g., plot the distribution of "number of claims per provider" for fraud vs non-fraud. Look for differences (fraudulent providers might show patterns like more claims on average). Plot claim amount distributions similarly.
7. **Correlation Heatmap:** If many features, plot a correlation matrix to see how features correlate with each other and with the target (though if target is binary, point-biserial correlation or just compare means as above).
8. **Bar charts for categorical breakdowns:** If you derived categorical-like features (e.g., provider specialty, or region from beneficiaries), see if certain categories have higher fraud rates.

9. Use Databricks notebooks' built-in display capabilities or libraries like matplotlib/seaborn for these plots.

10. **Example EDA Insights:** While exploring, you might find for instance:

11. Fraudulent providers have on average 2x the number of claims of non-fraud providers.
12. Certain chronic conditions or patient age groups are more frequent in fraud cases (perhaps indicating abuse of certain treatments).

13. Maybe most providers are clustered in a normal range for a feature, but fraud ones are outliers (e.g., extremely high billing amounts). These could be red flags.

14. **Document Findings:** Markdown cells in the notebook should summarize what you find. For example: "From the EDA, we observe that fraudulent providers tend to have a higher median number of outpatient claims than non-fraud providers (50 vs 20). They also billed higher amounts on average. This aligns with common fraud patterns (e.g., **upcoding** or billing for excessive services) [16] [17]." Such insights not only guide our modeling (which features might be important) but will be useful later when explaining the model with SHAP.

15. **Decide on Feature Selection:** If any feature has no meaningful signal (for example, if two features are 100% correlated or a feature has the same value for all providers), you might drop them before modeling. Conversely, ensure all potentially useful features discovered in EDA are included in the model dataset.

Outcome of Step 3: A good understanding of the data and preliminary hypotheses about which features are predictive of fraud. This will inform our modeling strategy and also serve as material to discuss in the project README or report.

Step 4: Feature Engineering & Preprocessing

1. **Data Preprocessing Pipeline:** It's good practice to set up a preprocessing pipeline (even if simple). Ensure all the feature transformations from raw data to model input are reproducible. Since we already aggregated features in Step 2, now focus on any additional preprocessing:
2. **Scaling:** If you have features on very different scales (dollar amounts vs counts), some models (like logistic regression or neural nets) benefit from feature scaling. Tree-based models (RF, XGBoost) typically do not need scaling. Decide based on model choice; you can skip scaling for tree models.
3. **Encoding Categoricals:** If any categorical features remain (e.g., provider type, region, etc.), encode them. One-hot encoding is safest. Since our data is now one row per provider, one-hot won't blow up dimensionality too much if categories are reasonable. Use pandas `get_dummies` or sklearn's `OneHotEncoder`. Alternatively, for tree models, you can label-encode and let the model handle it (though true one-hot may be more interpretable).

4. **Imputation:** Fill any nulls as identified earlier. If a feature like average patient age is null (perhaps provider had no patients? unlikely), fill with a neutral value (overall average or 0 if that makes sense).

5. **Feature Selection/Importance (Optional):** If the feature space is large, you might do a quick feature importance check using a simple model. For example, train a quick Random Forest and get feature importances, or use mutual information scores, to see which features seem most influential. This can guide you to drop unimportant features or confirm the ones you plan to use. (This step is optional, as SHAP and model training later will also highlight important features.)

6. **Prepare Final Training Data:** By now, you should have `train_data` and `val_data` (or `test_data` if you split into three sets) ready. Separate the target and features:

```
X_train = train_data.drop('FraudLabel', axis=1)
y_train = train_data['FraudLabel']
X_val = val_data.drop('FraudLabel', axis=1)
y_val = val_data['FraudLabel']
```

   Keep the Provider ID aside if you still have it (for reference or output, but it should not be in features).

7. **Balance Strategy (If Needed):** If fraud cases are very sparse, consider strategies:

8. **Class weight:** Many algorithms (sklearn's LogisticRegression, RandomForest, etc.) allow a `class_weight='balanced'` parameter to automatically weight the minority class higher. This is an easy way to handle imbalance.
9. **Resampling:** Alternatively, use SMOTE or random oversampling to augment the minority class in the training set [18] [19]. In Python, you could use `imblearn.over_sampling.SMOTE` to generate synthetic providers (though synthetic provider features might be a bit abstract). Oversampling may not be necessary if using tree models with class weights, but it's good to mention or consider if performance is an issue.

10. **Evaluation focus:** Regardless of method, plan to evaluate using metrics that account for imbalance (like ROC-AUC, precision/recall, F1, etc., not just accuracy).

11. **Spark ML (Optional):** If you prefer to use Spark's MLlib for modeling (to leverage the cluster for larger data), set up a Spark ML pipeline. For example, use `VectorAssembler` to combine feature columns into a feature vector, then a classifier like `pyspark.ml.classification.GBTClassifier`. Ensure to incorporate steps like one-hot encoding using `OneHotEncoderEstimator` and imputation using `Imputer` in the pipeline if using Spark. Given the moderate size of Kaggle data, using Pandas/Scikit-learn on the driver is usually fine; Spark is optional here unless you want to demonstrate it.

12. **Organize Code into Scripts:** At this stage, consider refactoring EDA and data prep code into reusable functions. For instance, if you wrote code in the notebook to aggregate features, move that logic into a function in a Python script (e.g., `src/data_processing.py`) like `def create_provider_features(inpatient_df, outpatient_df, bene_df): ...`. This makes it easier to reuse (and later, you could call this function if you had to process new incoming data or the test set). Keep the notebook for running the functions and doing quick analysis, but heavy-lifting code in scripts to keep things clean.

By the end of Step 4, you have a final cleaned and prepared dataset (`X_train, y_train, X_val, y_val`) and possibly some helper code structured in scripts. Now it's time to build and evaluate models.

Step 5: Model Training and Experiment Tracking

1. **Baseline Model Selection:** Start with a simple model to set a baseline. For fraud detection (tabular data), good choices are:
2. **Logistic Regression:** A basic linear model (with class weights for imbalance).
3. **Decision Tree / Random Forest:** Non-linear and handles features without scaling.
4. **XGBoost or LightGBM:** Gradient boosting trees often perform well in structured data competitions (XGBoost in particular was popular on this Kaggle task). These can handle our mix of features and capture complex interactions.

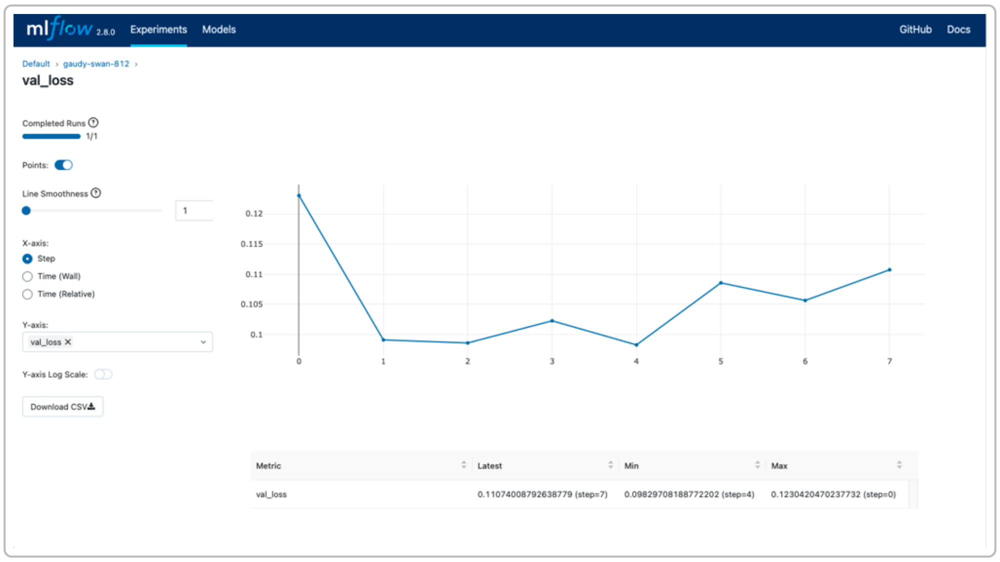
Begin with one model (say Random Forest) to get the process in place, then iterate.

1. **Set up MLflow Tracking:** In Databricks notebooks, MLflow is pre-installed and ready. You can use the MLflow API to log experiments. For example:

```
import mlflow
mlflow.set_experiment("/Users/<your_user>/health-fraud-exp")  # set an
experiment name/path

with mlflow.start_run():
    model = RandomForestClassifier(n_estimators=100,
class_weight='balanced', random_state=42)
    model.fit(X_train, y_train)
    preds = model.predict(X_val)
    # Log parameters and metrics
    mlflow.log_param("model", "RandomForest")
    mlflow.log_param("n_estimators", 100)
    mlflow.log_metric("Accuracy", accuracy_score(y_val, preds))
    mlflow.log_metric("F1", f1_score(y_val, preds))
    mlflow.log_metric("ROC_AUC", roc_auc_score(y_val,
model.predict_proba(X_val)[:,1]))
    # Log model
    mlflow.sklearn.log_model(model, artifact_path="model")
```

This will record the run in MLflow. Databricks provides a UI to see these runs under the experiment. MLflow tracking lets you log parameters, metrics, tags, and artifacts (like the model file) for each run [4]. It's a good practice to log any metric of interest (precision, recall, etc., in addition to accuracy) especially for imbalanced data. You can also log confusion matrix plots or feature importance plots as artifacts for each run.



Metric	Latest	Min	Max
val_loss	0.11074008792638779 (step=7)	0.09829708188772202 (step=4)	0.1230420470237732 (step=0)

Example: MLflow's UI can plot metrics across training runs. The image above shows a sample MLflow chart of validation loss vs. training step for a model run [20]. In our case, we might log the ROC curve or simply final metrics, and later compare runs in the MLflow experiment UI.

1. **Train and Tune Models:** Perform multiple training runs, trying different algorithms and hyperparameters:
2. Run 1: Logistic Regression (logistic regression might have lower raw accuracy but could yield insight – also test with/without regularization, etc).
3. Run 2: Random Forest (vary number of trees, depth).
4. Run 3: XGBoost or LightGBM (tweak tree depth, learning rate, etc).
5. Possibly run 4: A simple Neural Network (though tabular fraud data is usually best with tree models). Use **MLflow** to track each of these. Give each run a descriptive name or use `mlflow.set_tag("desc","XGBoost baseline")` so you can identify them later.

Also consider using Databricks' integrated tools like Hyperopt for hyperparameter tuning (Databricks has a `HyperOpt` integration that can log to MLflow). For example, you could use Hyperopt to find the best XGBoost parameters (max_depth, learning_rate, etc.) automatically. If you do, log the results as a separate MLflow run.

1. **Evaluate Model Performance:** For each model, evaluate on the validation set:
2. Print out confusion matrix, classification report (precision, recall, F1). Particularly check **Recall** on the fraud class (this is important – we often want to catch as many fraud cases as possible, even at the expense of some false alarms).
3. Calculate AUC-ROC and perhaps Precision-Recall curve. Log or plot these. A high AUC is good, but with imbalance, PR curve might be more telling.

4. Use MLflow to log these evaluation results (metrics or even figures). For example, you can log a matplotlib plot by saving it and using `mlflow.log_artifact("pr_curve.png")`.

5. **Select the Best Model:** After several runs, compare them in MLflow's experiment UI. You can sort by a metric (e.g., choose the model with highest F1 or highest recall at an acceptable precision). Suppose XGBoost with certain parameters gave the best F1 score. That will be our chosen model to

move forward with. Register this model in MLflow Model Registry (if using Databricks Managed MLflow) or simply note which run it was and use that model artifact.

6. **Register/Save the Model:** In Databricks, you can **register the model** to the MLflow Model Registry for versioning. This can be done via the UI or code:

```
result = mlflow.register_model("runs:/<RUN_ID>/model",
"HealthcareFraudModel")
mlflow.transition_model_version_stage(result.name, result.version,
stage="Production")
```

   This step is optional for our case (mostly used in collaborative settings). At minimum, **save the model artifact**. You can use `mlflow.sklearn.log_model` as above, which saves the model pickled. You can later download this from Databricks. Alternatively, use joblib to save model to the DBFS or local file: `joblib.dump(best_model, '/dbfs/FileStore/fraud_model.pkl')`.

7. **Test on Hold-out (if any):** If you set aside a portion as a final test set, evaluate the best model now on that test set to get an unbiased performance estimate. Log that result as well (but do not train further on test set).

8. **Interpret Results:** Summarize how well the model is doing. For example: *"Our best model (XGBoost) achieves 0.92 AUC on the validation set. It identifies ~85% of fraudulent providers (recall = 0.85) while keeping false positives relatively low (precision = 0.80). This is a significant improvement over baseline logistic regression."* These notes will be useful in your README and to guide the explainability step next.

Throughout Step 5, maintain clear documentation in the notebook: explain why you choose certain hyperparameters, and log observations about each experiment. Thanks to MLflow, you have a record of all runs, which makes your research process reproducible and transparent.

Step 6: Model Explainability with SHAP

1. **Introduction to SHAP:** Now that we have a trained model, we want to understand *why* it predicts a provider as fraud or not. We'll use **SHAP (SHapley Additive exPlanations)** for this. SHAP assigns each feature an importance value for a given prediction, indicating how much that feature contributed to pushing the prediction towards fraud or not [5]. In essence, it's based on game theory (Shapley values) to fairly attribute the model's output to the input features.

2. **Install and Setup:** If SHAP is not already installed on the cluster, install it (`%pip install shap`). Import the library in your notebook:

```
import shap
shap.initjs()  # for JS visualizations if in notebook
```

Load your best model (if not already in memory). If using XGBoost/LightGBM, SHAP has built-in optimizations; if using sklearn RandomForest or Logistic, SHAP will use the model as a black box (still fine).

3. **Global Feature Importance (SHAP Summary Plot):** Use a subset of the data (e.g., the validation set) to compute SHAP values:

```
explainer = shap.Explainer(best_model, X_train)  # or use
shap.TreeExplainer for tree models
shap_values = explainer(X_val)
shap.summary_plot(shap_values, X_val)
```

   The **summary plot** will show each feature's impact on the model output, with features ranked by importance. This is a key output: it tells us which features strongly influence the fraud predictions overall. For instance, you might see "TotalClaims" or "AvgClaimAmount" at the top, indicating they contribute the most (positive SHAP value pushing towards fraud for high values, perhaps).

4. **Interpret Global Importance:** From the SHAP summary:

5. Identify the top features. Suppose "InpatientClaimsCount" and "OutpatientClaimsCount" are top features and have mostly positive SHAP values for fraud (meaning more claims = more likely fraud, which makes intuitive sense).
6. Another feature might be "AveragePatientAge" with negative SHAP (meaning older patient age might reduce fraud likelihood perhaps, or vice versa).

7. Document 2-3 of the most significant features and what SHAP indicates about them. This directly addresses the project goal of discovering important variables for fraud detection [21].

8. **Local Explanation (Specific Provider):** Pick an example provider:

9. One that the model predicted as fraud (preferably a true positive from validation), and one non-fraud. Use SHAP to explain each:

```
provider_idx = 0  # index of a particular provider in X_val
shap.force_plot(explainer.expected_value, shap_values[provider_idx,:],
X_val.iloc[provider_idx])
```

   This force plot (or use `shap.waterfall_plot` for a clearer view in notebooks) will show how each feature for that provider pushes the prediction. For a fraudulent provider, you might see features like "TotalClaims = 120" pushing the model output strongly towards fraud, whereas "AverageAge = 82" might push it slightly toward non-fraud (if older patients are less associated with fraud in this dataset). For a non-fraud provider, see if the reverse is true (e.g., fewer claims, etc., leading to non-fraud prediction).

10. **Note:** In Databricks notebooks, SHAP plots may render as interactive graphs. Ensure you have run `shap.initjs()` and are using the latest plot functions. If they don't display, you can use

`shap.plots.bar` or `shap.plots.waterfall` which output as matplotlib figures (these can be saved).

11. **Summarize Explainability Insights:** Write down the key insights:

12. "SHAP analysis reveals that the **number of claims filed by a provider** is the strongest indicator of fraud – providers with unusually high claim counts are far more likely to be flagged (SHAP values > 0 for fraud). **Average claim amount** and **number of distinct patients** are also important features; high values push the prediction towards fraud. This aligns with domain intuition: fraudulent providers often bill for far more services than typical [22]. Features like patient demographics (average age, etc.) had lesser impact in the model."

13. By explaining individual predictions, we can also validate the model's reasoning. If the model were using any spurious correlations, SHAP might help spot that (e.g., if a seemingly irrelevant feature had high importance, we'd investigate). In our case, we expect features derived from claim counts and amounts to dominate, which is reasonable.

14. **Save SHAP Outputs (Optional):** Save important plots for inclusion in your report or dashboard. For instance, save the summary plot as an image (`plt.savefig('shap_summary.png')`) and log it via MLflow or just download from notebook. We might use it in the Streamlit app or documentation to illustrate feature importance.

By completing Step 6, we have an interpretable model. We can confidently articulate **why** the model flags certain providers – an important aspect for stakeholders like insurance auditors who need justification for investigating a provider. This also adds credibility to your project write-up.

Step 7: (Optional) Advanced Enhancement – Embeddings & Vector Similarity

This step is not required, but can add a novel dimension to your project, showcasing cutting-edge techniques using embeddings and vector databases.

1. **Concept of Embeddings:** In modern ML, an **embedding** is a numeric vector representation of data that captures semantic similarity. While our main model uses engineered features, we could also leverage embeddings for anomaly detection. For example, we could create an embedding for each provider based on their claims pattern (perhaps using an autoencoder or a language model on claim descriptions). The idea is to represent each provider as a point in high-dimensional space such that similar providers (in behavior) are nearby.

2. **Use Case – Similarity-Based Fraud Detection:** Fraudulent providers might be "far" from normal providers in this embedding space. Using a vector search approach, we can find the nearest neighbors of each provider and measure distances:

3. Compute an embedding for each provider. One approach: If there are text fields (like diagnosis codes or procedure descriptions), use a pre-trained language model (like a transformer) to embed those

texts. Another approach: simply use the features we created but normalize them and treat that vector as an embedding.

4. Use a library like **FAISS** (Facebook AI Similarity Search) or an open-source **vector database** to index all provider vectors.
5. For each provider, query the nearest neighbors. For a normal provider, neighbors will be other similar normal providers. A fraudulent provider might either cluster together with others (if there's a fraud pattern) or be an outlier.

6. You can then derive an "anomaly score" by how far a provider is from its k-nearest neighbors on average.

7. **Implement a Simple Vector Search:** As a demonstration, you can do:

```python
import numpy as np
from sklearn.neighbors import NearestNeighbors
X = df_model.drop('FraudLabel', axis=1).values  # feature matrix as vector
nbrs = NearestNeighbors(n_neighbors=5).fit(X)
distances, indices = nbrs.kneighbors(X)
# distances[i] gives distances to 5 nearest neighbors of provider i
anomaly_score = distances.mean(axis=1)
```

   Now, see if fraudulent providers have higher anomaly scores on average. If yes, this indicates our feature space embeds them as outliers.

8. **Leverage Advanced Embeddings (Optional):** If you want to go further:

9. Use an **autoencoder**: Train a neural network to reconstruct the provider feature vector. Fraudulent providers might have higher reconstruction error (because they have unusual patterns the autoencoder (trained mostly on normals) can't recreate well). Those errors can flag anomalies.

10. Use a **vector database**: Tools like Pinecone, Weaviate, or Milvus allow efficient similarity search at scale. This is likely overkill for our data size, but you could mention it as an enterprise solution: e.g., in production, one could embed each claim or provider and store in a vector DB for real-time anomaly queries.

11. **Benefit of Vector Approach:** Highlight the insight: By representing data in a vector space, even unsupervised anomalies can be detected – subtle deviations from normal patterns become measurable [23]. This could potentially catch new fraud types that the supervised model (trained on known fraud) might miss. For instance, if a provider suddenly starts behaving very differently from its peers, a vector similarity alert might flag it even if the pattern wasn't in training data.

12. **Document and (Optionally) Use in App:** If your findings are interesting (say you find a particular provider that was not labeled fraud but is a big outlier), you can mention this as a potential fraud candidate. This is beyond our supervised scope, but it's a great talking point. If implementing in Streamlit, you could allow a user to select a provider and see its nearest neighbors or anomaly score.

This advanced step demonstrates knowledge of unsupervised fraud detection techniques. It's optional and can be skipped if you prefer to focus on the main supervised pipeline. However, including even a brief analysis here can set your project apart by showing you are aware of **vector search** and **embedding** methods that are state-of-the-art in 2025 for anomaly detection and retrieval-augmented insights [7] .

Step 8: Project Organization and Best Practices

By now, you have many pieces: data processing code, EDA, modeling, etc. It's crucial to organize these for clarity and reusability, especially as a portfolio project.

1. **Directory Layout:** Organize your GitHub repository in a logical structure. For example:

```
healthcare-fraud-detection/
|── notebooks/
|      ├── 01_data_ingestion_eda.ipynb
|      ├── 02_modeling.ipynb
|      └── 03_explainability.ipynb
|── src/
|      ├── data_processing.py
|      ├── modeling.py
|      └── utils.py
|── dashboard/
|      ├── app.py
|      └── requirements.txt
|── models/
|      └── fraud_model.pkl  (saved model artifact, if small enough to
include)
|── README.md
└── (other files like .gitignore, etc.)
```

In this structure, **notebooks/** contains the research and step-by-step development (ordered by number). **src/** contains reusable code as Python modules (functions for loading data, engineering features, training model, etc.). The **dashboard/** will contain the Streamlit app code. The **models/** folder can store the final trained model or you could rely on MLflow for that. Adjust as needed, but keep it coherent.

2. **Clean, Documented Code:** Go through your notebooks and ensure they are well-commented and interspersed with markdown explaining each step. Anyone reading the notebook or script should understand what's happening. For scripts in `src/`, add docstrings to functions. For example, in `data_processing.py`:

```python
def create_provider_features(in_df, out_df, bene_df):
    """
    Merge inpatient, outpatient, and beneficiary data to create provider-
```

```
level features.
    Returns a DataFrame with one row per Provider and engineered features.
    """
    # code...
    return provider_df
```

This way, if someone wants to reproduce or reuse parts of your code, it's easy to follow.

3. **Use Git Effectively:** Commit your changes often with descriptive messages ("engineered features for provider aggregation", "added XGBoost model training with MLflow tracking", etc.). Push to GitHub. This not only backs up your work but demonstrates version control practice. If you are the sole contributor, a single branch (main) is fine, but you could use feature branches (e.g., a branch for "modeling") to showcase a collaborative workflow, then merge into main. In any case, ensure the GitHub repo always has the latest code corresponding to your final results.

4. **Avoid Committing Sensitive or Large Data:** Do **not** commit the raw Kaggle data to Git. It's large and possibly not permitted. Instead, in README, instruct the user to download the dataset from Kaggle. If you have small derived datasets or an example subset, that's fine to include, but generally keep data out of the repo. The code should assume data is present in a given path (or offer to download via Kaggle API if a user runs it).

5. **README.md:** This is the front page of your project on GitHub, so make it count:

6. Write a clear description of the project (objective, dataset, methods).
7. List the tools used (pandas, scikit-learn, PySpark, Databricks, MLflow, SHAP, Streamlit, etc.).
8. Provide instructions for someone to run the project. For example: *"To reproduce this project: 1) Download data from Kaggle and place in* `data/` *folder. 2) Run notebooks in order (or run the script* `src/modeling.py` *to train model). 3) Use* `dashboard/app.py` *with Streamlit to launch the web app."*
9. Include the architecture or steps at a glance. You can even embed small images of the pipeline or results (like the SHAP summary plot) for visual appeal.

10. Provide a link to the live Streamlit app (once deployed in Step 10) so readers can directly interact with the model.

11. **Testing (Optional):** For a professional touch, you might add a couple of unit tests in a `tests/` directory. For instance, test that `create_provider_features()` returns expected columns given sample input. This isn't strictly necessary for a portfolio project, but it shows engineering rigor.

12. **Notebook vs. Script Balance:** It's okay to have some analysis only in notebooks (EDA is often exploratory). But for the core pipeline (data prep -> training -> prediction), ensure there is a script or at least a well-structured notebook that runs end-to-end. An interested reader or recruiter should be able to execute your code easily to see results. That might mean providing a Databricks notebook export or a standard `.ipynb` that can run in Jupyter with minimal tweaks (perhaps using a smaller sample of data if needed).

By following these practices, you elevate the project from just code to a **professional portfolio piece**. The repository will not only solve the problem but also communicate your skills in organizing and documenting a data science project.

Step 9: Building the Streamlit Dashboard

The final step is to create an interactive dashboard to showcase your model. Streamlit allows us to turn our Python code into a shareable web app easily.

1. **Setup a Local Environment for Streamlit:** It's often easier to develop the Streamlit app on your local machine or a separate environment from Databricks. Ensure you have Python 3.x installed and create a virtual environment (using venv or conda). Install streamlit: `pip install streamlit`. Also install any other dependencies needed for the app (pandas, numpy, shap, scikit-learn, etc.). You can reuse the `requirements.txt` we'll prepare for deployment to install everything.

2. **Export the Trained Model:** Decide how to use the model in the app:

3. Easiest: **Save the model to a file** (e.g., `fraud_model.pkl`). If you used MLflow, you can download the logged model from Databricks. For example, in Databricks notebook:

   ```
   mlflow.artifacts.download_artifacts(run_id="<best_run_id>",
   artifact_path="model", dst_path=".")
   ```

   This will get a local copy of the model (you might get a folder with a pickle inside). Or if you saved with joblib to DBFS, use Databricks CLI or `dbutils.fs.cp` to copy it to local. Once you have the model file, include it in your GitHub repo (if it's not too large, typically a few MB should be fine).

4. Alternative: use MLflow's model registry in the app. Streamlit could fetch the model via MLflow if you host an MLflow server, but that adds complexity and requires exposing credentials. For simplicity, we'll proceed with a static model file.

5. **Design the App Interface:** Create a file `dashboard/app.py` (or just `app.py` if the repo is mainly the app). This script will define the web interface and logic:

6. Use `st.title()` and `st.markdown()` to add a title and description for the app (e.g., "Healthcare Fraud Detector – Predict if a provider is likely fraudulent").
7. **Input Section:** Decide what inputs the user can provide. Since our model works at provider level, one approach is to let the user input the key features for a hypothetical provider and get a prediction. For example, inputs could be:
   - Number of inpatient claims (slider or number input).
   - Number of outpatient claims.
   - Total billed amount.
   - Average patient age.
   - etc. (Use a selection of the most important features as per SHAP to make it simple for a user).
8. Alternatively, you could allow uploading a file or selecting a provider ID from the dataset to see the prediction. But a manual input of features is straightforward for demonstration.

9. Use Streamlit widgets: `st.number_input` for numerical features, `st.selectbox` for any categorical (if applicable), `st.button` for triggering prediction.

10. **Load Model and Make Prediction:** In the app code, load the saved model at start:

    ```
    import pandas as pd
    import joblib
    model = joblib.load("fraud_model.pkl")
    ```

    (Ensure the `fraud_model.pkl` is in the same directory or provide correct path). When the user fills the form and clicks a "Predict" button:

    ```
    if st.button("Predict Fraud Risk"):
        # gather input into a dataframe
        input_data = pd.DataFrame({
            'InpatientClaims': [inpatient_input],
            'OutpatientClaims': [outpatient_input],
            'TotalBilled': [billed_input],
            # ... other features
        })
        pred_prob = model.predict_proba(input_data)[0][1]  # probability of
    fraud class
        pred_class = model.predict(input_data)[0]
        if pred_class == 1:
            st.error(f"  The provider is likely Fraudulent! Risk score:
    {pred_prob:.2f}")
        else:
            st.success(f"  The provider is likely Legitimate. Fraud risk
    score: {pred_prob:.2f}")
    ```

    Style the output message with emojis or markdown as shown to make it clear.

11. **Display Model Insights in the App:** One powerful addition is to use SHAP (or simpler, feature contributions) to explain the prediction right in the app:

12. After predicting, compute SHAP values for the input. You can use the same `explainer` approach but note that loading a large explainer or background data in a Streamlit app might be slow. A trick: since this is a single prediction, use the model's built-in feature importance or a simplified SHAP (like TreeExplainer with `model` and just the single input).
13. For example, for tree models:

    ```
    import shap
    explainer = shap.TreeExplainer(model)
    shap_values = explainer(input_data)
    shap_html =
    ```

```
shap.plots._waterfall.waterfall_legacy(explainer.expected_value[1],
shap_values.values[0], feature_names=input_data.columns, show=False)
st.components.v1.html(shap_html, height=300)
```

This might embed a SHAP waterfall plot showing how each feature influenced the probability for that prediction. (There are also `st.pyplot` or `shap.force_plot` ways to show it – you may need to experiment, as SHAP plots on Streamlit sometimes require using Plotly or matplotlib backend).

14. If SHAP is too heavy, at least show the top features for this prediction. For instance, you could manually print something like: "Top factors for this prediction: High OutpatientClaims increased fraud risk by X. Low AverageAge decreased risk by Y."

15. **Add Additional Sections:** Enhance the dashboard with tabs or sections:

16. **Overall Model Performance:** You could show the metrics from validation (accuracy, F1) in a small table or metric widgets. For example, `st.metric("Validation AUC", 0.92)`. Hardcode these or load from a JSON if you saved metrics.
17. **Feature Importance Plot:** Use the SHAP summary plot (saved as image) or a bar chart of feature importances from the model. You can do `st.image("shap_summary.png")` if you have that file, or dynamically create a matplotlib bar chart of model.feature_importances_ for tree models and use `st.pyplot`.

18. **Instructions/Info:** Provide a brief explanation on how to use the app and what the outputs mean. Also maybe a disclaimer that this is a demo and not actual medical fraud adjudication tool, etc.

19. **Test the App Locally:** Run `streamlit run app.py` in your local environment. This should open a browser showing the app. Test different inputs (for sanity, you can plug in some values from your dataset – e.g., a known fraud provider's feature values – to see if it predicts fraud). Adjust the UI if needed (for example, set reasonable min/max on number inputs based on dataset ranges).

20. **Prepare `requirements.txt`:** List all packages your app needs, with specific versions if possible. Common ones:

```
streamlit==1.x.x
scikit-learn==1.2.2
pandas==1.5.3
numpy==1.24.3
shap==0.41.0
matplotlib==3.7.1
```

etc. Pin versions that you used to avoid surprises. This file will be used by Streamlit Cloud to install dependencies.

21. **Finalize App Code and Push to GitHub:** Ensure that `app.py` and `requirements.txt` (and model file, if included) are in the repo. They can be at repo root or in a `dashboard` folder –

Streamlit Cloud will allow you to specify the app location. Write a short README section about the app usage as well.

With Step 9, you have a fully functional Streamlit application that users (or recruiters) can interact with to see your fraud detection model in action. This greatly enhances the demonstration of your project's results.

Step 10: Deploying the Dashboard on Streamlit Cloud

Now that the app is ready, sharing it is as easy as deploying to **Streamlit Community Cloud** (formerly Streamlit Sharing). This will allow anyone to access your app via a URL, without needing to run code locally.

1. **Set Up Streamlit Cloud Account:** Go to streamlit.io and sign up (you can use GitHub to sign in). Streamlit Cloud is free for public GitHub repos. After logging in, you'll have a workspace where you can deploy apps.

2. **Deploy the App from GitHub:** In Streamlit Cloud, click **"New app"**. It will prompt you to select the GitHub repo, branch, and the app script path. Select your repository and the branch (e.g., `main`). For the file path, enter the path to `app.py` (for example `dashboard/app.py` or simply `app.py` if at root). Then click **Deploy**. Streamlit will build the app – it checks out your repo, creates an environment, installs the `requirements.txt` packages, and runs the app. This process usually takes a few minutes on first deploy [24].

3. **Configure Secrets (if any):** If your app needed any secrets (API keys, etc.), you'd add them in the Streamlit Cloud settings (under **Secrets**). In our case, we likely don't need secrets. The model is bundled and data is static. (If you were pulling data from Snowflake or an API, you'd put credentials here and access via `st.secrets` in the app.)

4. **Await Deployment:** Streamlit Cloud will show logs of the build. If there are errors (e.g., missing packages or wrong file path), fix them (update your repo) and the app will redeploy. Common issues might be large files – note that Streamlit Cloud limits the repo size and memory. Our dataset isn't in the repo, and the model file if included should be reasonably sized, so we should be fine.

5. **Test the Live App:** Once deployed, you'll get a URL like `https://<your-user>-<app-name>.streamlit.app`. Open that in your browser. Test the functionality just as you did locally. The app might be a bit slower on first use due to cold start (and SHAP calculations if any). Everything from input to prediction should work. Check that the outputs (messages, charts) display correctly.

6. **Share & Document:** Copy this app link and add it to your README (e.g., "**Live Demo**: Healthcare Fraud Detection App"). You can also share it with others to show your work. The app will remain live as long as the repo exists and you don't exceed usage limits (free tier gives sufficient resources for demo purposes).

7. **Streamlit Cloud Management:** In your Streamlit Cloud workspace, you can set up things like scheduled re-runs or manage versions. But for now, a manual trigger to re-run (Streamlit provides a button to rerun or you can just refresh the page) is enough.

Streamlit Cloud simplifies deployment – it's essentially one-click from a GitHub repo [25]. The result is a publicly accessible tool demonstrating the model's capabilities in an interactive way, which is highly valuable for a portfolio project.

Conclusion

By following this blueprint, you've implemented a comprehensive end-to-end data science project on health insurance fraud detection:

- We began with **data ingestion** on Databricks, combining multiple healthcare claims tables into a meaningful training set [9].
- Through **EDA**, we uncovered patterns and prepared features that differentiate fraudulent providers.
- We built and tuned a **machine learning model** (like XGBoost) to predict fraud, using **MLflow** to track experiments for reproducibility and comparison [26].
- Using **SHAP**, we added interpretability, identifying which features (e.g., excessive claims, high charges) drive the model's predictions [5]. This addresses the business need for explainable AI in fraud detection.
- We structured our code and notes in an organized repository, following best practices of notebook-script separation and version control.
- Finally, we created a **Streamlit dashboard** to present the model in action, allowing users to input data and see predictions with explanatory insight. The app was deployed on Streamlit Cloud, making our project accessible to others with ease.

Throughout the project, we also discussed advanced considerations like how Snowflake could be used for scalable data storage and how embedding-based anomaly detection could complement the solution, reflecting modern 2025 techniques [7].

You now have a portfolio-quality project that demonstrates skills in data engineering, machine learning, cloud platforms, and communication of results. A next step could be to integrate this with a live database or to automate the pipeline (for example, using Databricks Jobs to retrain the model periodically, or deploying the model as an API). Nonetheless, the delivered system provides a solid foundation and a clear narrative – from raw data to actionable insights – showcasing your expertise in end-to-end data science project development.

---

[1] [16] [22] Healthcare Provider Fraud Detection And Analysis — Machine learning | by Rohan kumar soni | Medium
https://rohansoni-jssaten2019.medium.com/healthcare-provider-fraud-detection-and-analysis-machine-learning-6af6366caff2

[2] [9] [10] [13] [14] [15] [17] [21] Detecting Medicare Provider Fraud with Machine Learning | HackerNoon
https://hackernoon.com/detecting-medicare-provider-fraud-with-machine-learning

[3] What is the Databricks Unified Data Analytics Platform?
https://www.databricks.com/glossary/unified-data-analytics-platform

[4] [26] Track model development using MLflow | Databricks Documentation
https://docs.databricks.com/aws/en/mlflow/tracking

[5] SHAP : A Comprehensive Guide to SHapley Additive exPlanations | GeeksforGeeks
https://www.geeksforgeeks.org/shap-a-comprehensive-guide-to-shapley-additive-explanations/

[6] Streamlit • A faster way to build and share data apps
https://streamlit.io/

[7] [23] Understanding Vector Search and Vector Databases: A Comprehensive Guide | by Manjit Singh | Medium
https://manjit28.medium.com/understanding-vector-search-and-vector-databases-a-comprehensive-guide-1b8bf2bdf0fe

[8] Start a Databricks free trial | Databricks Documentation
https://docs.databricks.com/aws/en/getting-started/free-trial

[11] Using Snowflake and Databricks Together / Blogs / Perficient
https://blogs.perficient.com/2024/03/15/using-snowflake-and-databricks-together/

[12] Databricks Repos: Git Integration | Databricks
https://www.databricks.com/product/repos

[18] [19] Credit Card Fraud Prediction model, SMOTE & model Explainability using Shap values. | Medium
https://omkargawade.medium.com/how-to-make-credit-card-defaulter-prediction-model-with-model-explainability-using-shap-values-a4ae4cbba077?source=rss-79b20752c44d------2

[20] MLflow Tracking | MLflow
https://mlflow.org/docs/latest/tracking/

[24] [25] Prep and deploy your app on Community Cloud - Streamlit Docs
https://docs.streamlit.io/deploy/streamlit-community-cloud/deploy-your-app