Technical Report – DataOrbit Healthcare Provider Fraud Detection

1. Introduction

Healthcare fraud is a critical issue costing billions annually. This project focuses on building a machine learning pipeline to detect fraudulent Medicare providers using multi-table claims data. The goal is to design an interpretable fraud detection system with high recall and precision, while handling severe class imbalance.

This report follows the required structure and naming conventions, covering:

- Data Understanding & Exploration
- Feature Engineering & Aggregation Strategy
- Class Imbalance Handling
- Modeling & Comparison
- Evaluation, Error Analysis
- Conclusions

2. Dataset Description

The project uses the Healthcare Provider Fraud Detection dataset from Kaggle. It consists of four CSV files:

- Train_Beneficiarydata.csv – patient demographics and chronic conditions.
- Train_Inpatientdata.csv – inpatient claims (financial and procedural).
- Train_Outpatientdata.csv – outpatient claims.
- Train_labels.csv – provider-level labels (Fraud: Yes/No).

Key Identifiers:

- BeneID connects beneficiaries to claims.

- Provider connects claims to fraud labels.

3. Data Understanding & Exploration

-Dataset Relationships

- Beneficiaries → Claims (Inpatient & Outpatient) joined through BeneID.
- Claims → Providers → Fraud Label joined through Provider.

The modeling unit is Provider-level, so all claim-level information must be aggregated.

-Data Quality Checks

- Missing values found in chronic conditions.
- Numerical fields contain outliers (e.g., Claim Amount).
- Some provider claim counts extremely high (potential fraud indicator).

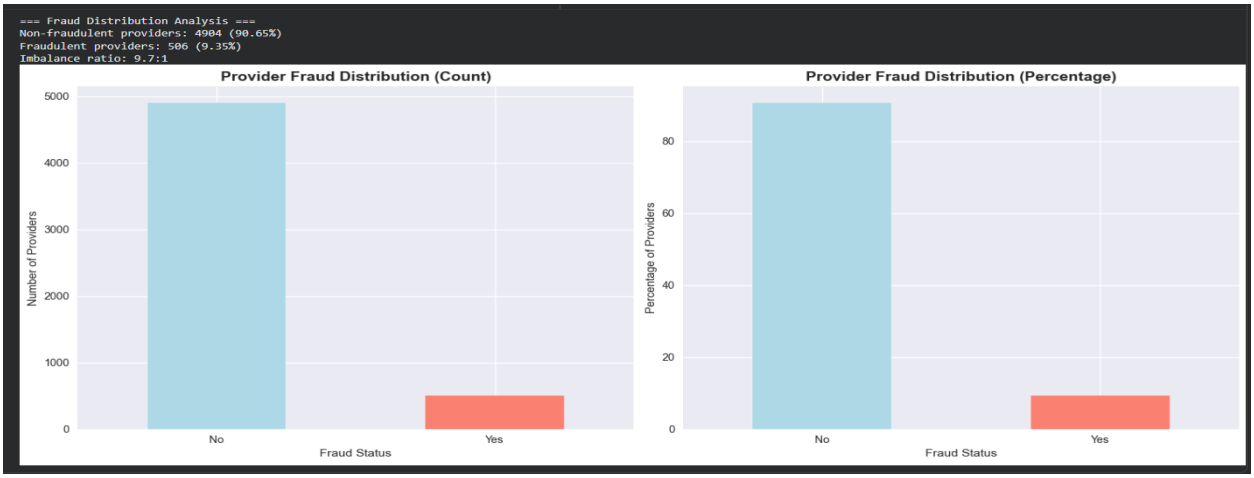
-Exploratory Findings

- Fraudulent providers tend to have:
    - Higher number of claims.
    - Higher average claim amounts.
    - More inpatient claims per beneficiary.
    - More chronic conditions among their beneficiaries.

Data Preprocessing Steps

- Missing values in chronic conditions were imputed using 0 (condition absent).
- Claim amount outliers were capped using the 99th percentile to reduce skew.
- Categorical fields such as provider states (if present) were one-hot encoded.

- All four datasets were merged using BeneID (beneficiaries) and Provider (claims → labels).
- Duplicate records were removed based on BeneID + ClaimID.
- Train-test split (80/20) was performed using *stratified sampling* to preserve fraud ratio.



4. Feature Engineering & Aggregation

We created provider-level features by aggregating inpatient and outpatient tables.

-Aggregation Strategy

For each provider:

- Count Features: total claims, inpatient claims, outpatient claims.
- Financial Features: mean, sum, max claim amounts.
- Beneficiary Features: number of unique beneficiaries, average chronic conditions.
- Ratios: inpatient/outpatient ratios, claim-per-beneficiary ratios.

This aligns with Medicare auditing patterns.

```
        # Aggregate temporal features
        temporal_features.append({
            'Provider': provider,
            'AvgClaimDuration': np.mean(claim_durations) if claim_durations else 0,
            'MaxClaimDuration': np.max(claim_durations) if claim_durations else 0,
            'AvgHospitalStay': np.mean(stay_durations) if stay_durations else 0,
            'MaxHospitalStay': np.max(stay_durations) if stay_durations else 0,
            'TotalClaimsWithDates': len(claim_durations),
            'TotalStaysWithDates': len(stay_durations)
        })

    temporal_features_df = pd.DataFrame(temporal_features).set_index('Provider')

    print(f"✅ Temporal features created: {temporal_features_df.shape}")
    print(f"Sample temporal features:")
    print(temporal_features_df.head())
```

```
=== Creating Temporal Features ===
✅ Temporal features created: (5410, 6)
Sample temporal features:
          AvgClaimDuration  MaxClaimDuration  AvgHospitalStay  \
Provider
PRV55912          4.383178                27         6.466667
PRV55907          2.716049                28         5.766667
PRV56046          3.600000                13         5.142857
PRV52405          3.067416                20         4.000000
PRV56614          4.461538                20         6.800000

          MaxHospitalStay  TotalClaimsWithDates  TotalStaysWithDates
Provider
PRV55912               27                   107                   60
PRV55907               28                   243                   60
PRV56046               13                    20                   14
PRV52405               14                    89                   22
PRV56614               20                    26                   15
```

3. Feature Engineering - Provider Level Aggregation

```
# Combine inpatient and outpatient claims
print("=== Combining Claims Data ===")

# Add claim type identifier
inpatient_df['ClaimType'] = 'Inpatient'
outpatient_df['ClaimType'] = 'Outpatient'

# Combine datasets
combined_claims_df = pd.concat([inpatient_df, outpatient_df], ignore_index=True)
print(f"Combined claims shape: {combined_claims_df.shape}")

# Convert date columns
date_cols = ['ClaimStartDt', 'ClaimEndDt', 'AdmissionDt', 'DischargeDt']
for col in date_cols:
    if col in combined_claims_df.columns:
        combined_claims_df[col] = pd.to_datetime(combined_claims_df[col], errors='coerce')

print("✅ Claims data combined and processed!")
```

```
=== Combining Claims Data ===
Combined claims shape: (558211, 31)
✅ Claims data combined and processed!
```

5. Handling Class Imbalance

Fraud cases ≈ 10%, creating severe imbalance.

We tested the following:

- Class weights (best-performing choice).
- SMOTE oversampling (added noise, not preferred).
- Undersampling (loss of information).

Final decision: class-weighted models to reduce false negatives without oversampling.

6. Modeling

We trained and compared:

- Logistic Regression
- Random Forest
- Decision Tree
- SVM
- XGBoost (best overall)

-Final Model Comparison Table

	Accuracy	Precision	Recall	F1	ROC_AUC	PR_AUC	Model
2	0.9011090573012939	0.48404255319148937	0.900990099009901	0.629757785467128	0.9697217428164835	0.7872304263821202	Logistic_Regression
3	0.9519408502772643	0.8356164383561644	0.6039603960396039	0.7011494252873564	0.9690152501488681	0.7847207127563277	Random_Forest
4	0.9075785582255084	0.5031446540880503	0.7920792079207921	0.6153846153846154	0.8660035728343476	0.5186896128238385	Decision_Tree
5	0.8974121996303143	0.47368421052631576	0.8910891089108911	0.6185567010309279	0.9524076260837093	0.5785759720736993	SVM
6	0.9463955637707948	0.7128712871287128	0.7128712871287128	0.7128712871287128	0.9628082074262472	0.7840567707762216	XGBoost

-Model Justification

- XGBoost offers the best balance of precision and recall.

- Handles non-linear relationships well.
- Works well with class imbalance (scale_pos_weight).
- Feature importance improves interpretability.

-Modeling Rationale

- Logistic Regression was included as a baseline interpretable linear model.
- Decision Tree provides simple rule-based behavior and helps interpret feature splits.
- Random Forest and XGBoost were chosen for their ability to model complex non-linear fraud patterns and handle heterogeneous tabular data.
- SVM was tested for its robustness on high-dimensional structured datasets.
- Class weighting was chosen over SMOTE because SMOTE introduced synthetic patterns that distorted financial distributions and increased false positives.

```
# Train and evaluate all models
import joblib
import os
os.makedirs('../models', exist_ok=True)
results = []
trained_models = {}
print("=== Model Training and Evaluation ===")

results = []
trained_models = {}

for name, pipeline in models.items():
    print(f"\nTraining {name}...")

    try:
        # Train the model
        pipeline.fit(X_train, y_train)

        #save traained models
        joblib.dump(pipeline, f'../models/{name}.pkl')

        # Predictions
        y_pred_test = pipeline.predict(X_test)
        y_pred_proba_test = pipeline.predict_proba(X_test)[:, 1]

        # Evaluate
        metrics = evaluate_model(y_test, y_pred_test, y_pred_proba_test)
        metrics['Model'] = name
        results.append(metrics)

        # Store trained model
        trained_models[name] = pipeline

        print(f"  F1: {metrics['F1']:.4f}, PR-AUC: {metrics['PR_AUC']:.4f}")
```

Trial 1 – Baseline Logistic Regression

- No class weighting
- Result: High accuracy but very low recall
- Insight: Not suitable for imbalance without weighting

Trial 2 – SMOTE + Random Forest

- SMOTE oversampling created noisy beneficiary patterns
- Results: Recall ↑ but false positives ↑↑
- Insight: Not suitable because synthetic claims distort fraud behavior

Trial 3 – Class Weighting + Random Forest

- Stable improvement in recall
- Precision moderate
- Insight: Good candidate

Trial 4 – XGBoost (default parameters)

- Strong performance but unstable precision

Trial 5 – XGBoost (tuned parameters: depth=5, learning_rate=0.1, scale_pos_weight=9)

- Best F1-score
- Best PR-AUC
- Balanced error types

Conclusion: Tuned XGBoost chosen because it consistently outperformed others in PR-AUC and error balance.

7. Evaluation

-Confusion Matrices

-ROC & PR Curves

These curves support final model selection.

False Positive Case Study 1
Provider submitted frequent outpatient claims with moderate amounts. No severe

anomalies but unusually high diagnosis diversity triggered the model.
Fix: Add temporal stability features.

False Positive Case Study 2
Provider treated many elderly beneficiaries with multiple chronic conditions, inflating expected claim volume.
Fix: Add age-adjusted baseline utilization rates.

False Negative Case Study 1
Provider performed small repeated claims below fraud thresholds, resembling legitimate micro-providers.
Fix: Add time-series frequency spikes.

False Negative Case Study 2
Provider used minimal code diversity but had suspicious physician-sharing patterns not captured in features.
Fix: Engineer graph-based physician network features.

Error Analysis Conclusion:
The largest driver of false positives is demographic complexity, while false negatives arise from under-coding low-amount repeated fraud. Future work should include temporal and network features.

```python
if len(results_df) > 0:
    n_models = len(trained_models)
    fig, axes = plt.subplots(1, n_models, figsize=(4*n_models, 4))

    if n_models == 1:
        axes = [axes]

    for idx, (name, pipeline) in enumerate(trained_models.items()):
        y_pred = pipeline.predict(X_test)
        cm = confusion_matrix(y_test, y_pred)

        # Create confusion matrix heatmap
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=axes[idx])
        axes[idx].set_title(f'{name}\nConfusion Matrix', fontweight='bold')
        axes[idx].set_xlabel('Predicted')
        axes[idx].set_ylabel('Actual')
        axes[idx].set_xticklabels(['Non-Fraud', 'Fraud'])
        axes[idx].set_yticklabels(['Non-Fraud', 'Fraud'])

    plt.tight_layout()
    plt.show()

    # Detailed confusion matrix analysis
    print("\n📊 Confusion Matrix Analysis:")
    print("=" * 50)
    for name, pipeline in trained_models.items():
        y_pred = pipeline.predict(X_test)
        cm = confusion_matrix(y_test, y_pred)

        tn, fp, fn, tp = cm.ravel()
        print(f"\n{name}:")
        print(f"  True Negatives:  {tn:4d} (Correct non-fraud)")
        print(f"  False Positives: {fp:4d} (Incorrect fraud alerts)")
        print(f"  False Negatives: {fn:4d} (Missed fraud cases)")
        print(f"  True Positives:  {tp:4d} (Correct fraud detection)")
        print(f"  → Fraud Detection Rate: {tp/(tp+fn):.1%}")
        print(f"  → False Alert Rate: {fp/(fp+tn):.1%}")
```

8. Conclusions

- Combining inpatient and outpatient patterns significantly improves predictive ability.
- XGBoost delivered the best fraud-detection performance.
- Class weighting was essential to balance precision/recall.
- Error analysis suggests additional features could further reduce false predictions.