**1.3 Task 3: Fraud detection in a government agency (spotlight: MLOps)**

You are working for a government agency supporting people in need with financial and consultancy promotion. In the first years, the programs you implemented were quite successful. In the last couple of years, though, the numbers indicate that there seems to be an increasing number of fraud suspicions. This results in the fact that people who actually need the support of the agency's programs are no longer able to obtain it. There are thousands of applications each month, a system has been installed for applicants to fill in the program application forms online. It is not viable for your department to manually check each of these applications. As the data scientists of your department, you're assigned the task to design and implement a system that automatically detects fraud in incoming proposals with a certain probability and integrates seamlessly with the agency's running systems. It is a requirement for your system to be adaptable in a straightforward way and regularly. As the political and social situation changes constantly, the incoming application data is also affected by these changes. Accordingly, your system has to be designed so that the machine learning model behind it can be retrained and re-implemented easily.

1. Design the conceptual architecture of your system. By doing so, consider data ingestion, storage processing, and handling requests to the prediction model as a service. Draft a visual overview of your architectural design, showing which data and processes are handed over by which application to the next. This will also guide you through the next steps of your project.

2. Choose an open data source that can serve as sample data for your project. A good starting point for your research are open data science competition websites, such as Kaggle. Alternatively, you can also produce your own fictional sample data.

3. Build a simple fraud detection model using Python. Do not put too much effort into building this model, and just make sure that it performs the task at hand and check basic statistical measures.

4. Package your model in a way that it can take data over a standardized RESTful API and respond with a probability for fraud. You might want to use Python libraries for this, such as Flask or mlflow. A good starting point for this step is the documentation of the respective Python.

5. Make sure that the performance of your predictive model as a service can be easily monitored. A good starting point is mlflow. You should also guarantee that only those eyes, which are meant to see the data and use the system, have access to it. Use a MLOps platform, such as Azure DevOps, Jenkins, GitHub actions, etc., to automate your process. Use this MLOps platform to trigger the training and deployment of your model.

6. Implement your system in a way that the MLOps platform re-trains the model after a given time (one month) or after the incoming data has changed by a threshold value to account for data drift. Test your setup by simulating one year of new data and monthly re-training. To do that, modify your data for each simulated month and see if your MLOps platform can catch these changes and re-trains the model. Make sure that the reimplemented model as a service is still reachable over RESTful API.

7. It is unnecessary to implement your system in the cloud to obtain the highest grade for your project. Implementing to the cloud is a little more elaborate, though you might want to challenge yourself and gain some extra points for this effort.

**For your final oral project report, also consider these questions from the module description:**

• What were the challenges of integrating a predictive model into an application or service?

• What are the constraints of implementing a predictive model as a service?

• Which requirements for data acquisition, storage, and processing had to be met, and how did you achieve this?

• What are monitoring components required for reliable execution of the predictive model? What is the design of your system? Present a visual draft of your system regarding storing, accessing, and serving the predictive model.

• In your oral presentation, provide a link for your audience to follow and reproduce your code.

**Step 1: Design the conceptual architecture**

1. **Identify the main components**
   * **Data ingestion**: How your system obtains or receives the data (e.g., from an online form submission). Django service, Data from Kaggle
   * **Data storage**: Where your data is stored (e.g., a database, cloud storage, data lake, etc.). MySQL
   * **Data processing and feature engineering**: Steps to clean, preprocess, transform the data for modeling. **Python scripts** using pandas, NumPy, scikit-learn for transformations.
   * **Model training environment**: Script or pipeline that trains (and re-trains) your fraud detection model. Python env, **MLflow** for experiment tracking.
   * **Model registry and versioning**: Where you store different versions of your model (e.g., MLflow Model Registry).
   * **Serving layer (RESTful API)**: Exposes an endpoint for fraud prediction requests.
   * **Monitoring and logging**: Monitor model performance, track usage, and detect data drift.
   * **MLOps orchestration**: Coordinates training, deployment, and re-training triggers (Azure DevOps, GitHub Actions, Jenkins, etc.).
2. **Draw a simple architecture diagram** (a **visual** draft might look like the diagram described below). For example:

scss

Copy code

┌────────────────────────────────────────────────┐

│ Online Application Portal (UI) │

│ Users fill forms, submit application details │

└────────────────────────────────────────────────┘

│

│ (1) Ingest data

▼

┌────────────────────────────────────────────────┐

│ Data Ingestion Service │

│ - Validation, parsing, scheduling │

└────────────────────────────────────────────────┘

│

│ (2) Store raw data

▼

┌────────────────────────────────────────────────┐

│ Data Storage (Database/Data Lake) │

│ (structured, unstructured) │

└────────────────────────────────────────────────┘

│

│ (3) Trigger ETL / Data Processing

▼

┌────────────────────────────────────────────────┐

│ Data Processing & Feature Engineering │

│ - Clean, transform, create features │

└────────────────────────────────────────────────┘

│

│ (4) Store processed data

▼

┌────────────────────────────────────────────────┐

│ Processed Data Storage / Feature Store │

└────────────────────────────────────────────────┘

│

│ (5) Training pipeline triggered by MLOps platform (schedule, data drift, monthly, etc.)

▼

┌────────────────────────────────────────────────┐

│ Model Training Pipeline │

│ - Uses processed data, trains model │

│ - Logs metrics to MLflow / monitoring │

└────────────────────────────────────────────────┘

│

│ (6) Register model / Deploy

▼

┌────────────────────────────────────────────────┐

│ Model Registry / MLflow │

│ - Stores model artifacts, versions │

└────────────────────────────────────────────────┘

│

│ (7) Deploy new model to serving

▼

┌────────────────────────────────────────────────┐

│ Prediction Service (RESTful API) │

│ - Receives requests and returns fraud scores │

└────────────────────────────────────────────────┘

│

│ (8) Monitor usage and performance

▼

┌────────────────────────────────────────────────┐

│ Monitoring & Logging (MLflow, etc.) │

│ - Tracks data drift, model performance │

└────────────────────────────────────────────────┘

This architecture ensures all your moving parts (data ingestion, storage, model training, serving, etc.) are organized and can be updated or replaced as needed.

**Step 2: Choose an open data source**

* If you want real data, you can explore:
  + **Kaggle**: Search for “fraud detection” datasets (e.g., “Credit Card Fraud Detection”).
  + **UCI Machine Learning Repository**: May have datasets relevant to fraud or anomaly detection.
* Alternatively, **create synthetic data** matching the structure you need (columns that make sense for your domain—income, age, requested amount, etc.).

**Tips**

* Start small. You just need a CSV or similar format that you can upload to your environment.
* Make sure the dataset has a **target column** indicating “fraud” or “not fraud” to train a supervised model.

**Step 3: Build a simple fraud detection model in Python**

1. **Set up your environment**
   * Create a Python virtual environment:

bash

Copy code

python3 -m venv venv

source venv/bin/activate # On Windows: venv\Scripts\activate

* + Install necessary libraries:

bash

Copy code

pip install numpy pandas scikit-learn mlflow flask

1. **Load and explore your data**

python

Copy code

import pandas as pd

data = pd.read\_csv("fraud\_data.csv")

print(data.head())

print(data.info())

print(data.describe())

1. **Preprocessing and splitting**

python

Copy code

from sklearn.model\_selection import train\_test\_split

# Assume your target variable is 'is\_fraud' (0 or 1)

X = data.drop(columns=['is\_fraud'])

y = data['is\_fraud']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

1. **Train a simple model**

python

Copy code

from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(n\_estimators=100, random\_state=42)

model.fit(X\_train, y\_train)

1. **Evaluate your model**

python

Copy code

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

y\_pred = model.predict(X\_test)

print("Accuracy:", accuracy\_score(y\_test, y\_pred))

print("Precision:", precision\_score(y\_test, y\_pred))

print("Recall:", recall\_score(y\_test, y\_pred))

print("F1 Score:", f1\_score(y\_test, y\_pred))

You now have a basic fraud detection model. Don’t over-optimize—demonstrate that it works.

**Step 4: Package your model with a RESTful API**

Use **Flask** as a simple approach:

1. **Create app.py** (or any name you like):

python

Copy code

from flask import Flask, request, jsonify

import joblib

import numpy as np

app = Flask(\_\_name\_\_)

# Load the trained model (which you might save as model.pkl or similar)

model = joblib.load("model.pkl") # You can also save with pickle or joblib

@app.route("/predict", methods=["POST"])

def predict():

# Example: data is sent as JSON

# e.g., { "feature1": 10, "feature2": 3, ..., "featureN": 5 }

input\_data = request.json

# Convert input\_data to the correct shape

# For instance, create a list in the correct order of features

features = [input\_data["feature1"], input\_data["feature2"], ... ] # fill as needed

features = np.array(features).reshape(1, -1)

# Get prediction probability

prob = model.predict\_proba(features)[0][1]

# Return probability as JSON

return jsonify({"fraud\_probability": float(prob)})

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host="0.0.0.0", port=5000)

1. **Test your API locally**
   * In one terminal, run:

bash

Copy code

python app.py

* + In another terminal (or a tool like Postman), send a **POST** request:

bash

Copy code

curl -X POST -H "Content-Type: application/json" \

-d '{"feature1":10,"feature2":3,"feature3":5}' \

http://127.0.0.1:5000/predict

1. **Response** will look like:

json

Copy code

{ "fraud\_probability": 0.85 }

That’s your model as a service!

**Step 5: Set up performance monitoring**

**Using MLflow for tracking**

1. **Enable MLflow tracking** in your training script:

python

Copy code

import mlflow

import mlflow.sklearn

mlflow.start\_run()

model = RandomForestClassifier(n\_estimators=100, random\_state=42)

model.fit(X\_train, y\_train)

# Log metrics

y\_pred = model.predict(X\_test)

acc = accuracy\_score(y\_test, y\_pred)

mlflow.log\_metric("accuracy", acc)

# ... log more metrics as you like

# Log the model

mlflow.sklearn.log\_model(model, "model")

mlflow.end\_run()

1. **Monitor runs** by launching MLflow UI:

bash

Copy code

mlflow ui

Navigate to http://127.0.0.1:5000 (default MLflow UI port is 5000 or 5001).

**Access control**

* If you’re running on a server, implement **role-based access** or **basic authentication** to ensure only authorized personnel can see logs and predictions. This can be done via:
  + Flask-Login (for simple solutions), or
  + Reverse proxy configurations (Nginx with HTTP Basic Auth), or
  + Cloud solutions (Azure, AWS) with built-in identity management.

**Step 6: MLOps pipeline for automated training & deployment**

**Choose an MLOps platform**

* **Azure DevOps**: Pipelines for code and model deployment.
* **GitHub Actions**: YAML-based workflows that run when you push commits or on a schedule.
* **Jenkins**: A self-hosted solution for CI/CD.

**Example with GitHub Actions**

1. **Create a .github/workflows/train-and-deploy.yml** file in your GitHub repo:

yaml

Copy code

name: Train and Deploy Fraud Model

on:

push:

branches: [ "main" ]

schedule:

- cron: '0 0 1 \* \*' # Once a month on the 1st day at 00:00

jobs:

build-and-deploy:

runs-on: ubuntu-latest

steps:

- name: Check out repo

uses: actions/checkout@v2

- name: Set up Python

uses: actions/setup-python@v2

with:

python-version: '3.9'

- name: Install dependencies

run: |

pip install -r requirements.txt

# or run pip install scikit-learn pandas mlflow flask

- name: Run training script

run: |

python train.py # This script logs metrics to mlflow and saves your model

- name: Deploy

run: |

# Here you'd add commands to deploy the model to your production environment

# e.g., copy new model artifact to your server or container

echo "Deployment steps..."

1. **Data drift trigger**
   * During each run, your pipeline can check new data statistics. If a significant drift is detected, it triggers re-training automatically. You can implement this by:
     + Comparing statistical distributions from the newly arrived data vs. historical data.
     + If drift > threshold, proceed with re-training steps in the pipeline.
2. **Simulate a year of data**
   * Keep modifying your dataset each month (change distributions, add more or fewer fraud cases, etc.).
   * Commit/push the new data to your repo or have an automated process that triggers the pipeline.
   * Verify that your pipeline runs monthly (or whenever you push changes) and re-trains the model.
   * Ensure the newly trained model is deployed and that the **RESTful API** is updated to use the latest model.

**Step 7: (Optional) Deploy to the Cloud**

* If you want an extra challenge, deploy to:
  + **Azure App Service** or **AWS Elastic Beanstalk** for the Flask API.
  + Use a cloud-based **MLflow** or an **Azure ML** workspace for tracking and model registry.

This is not mandatory, but it provides real-world experience.

**Questions from the module description (for your presentation)**

1. **Challenges of integrating a predictive model into an application or service**
   * **Data format mismatch** between the model and incoming requests.
   * **Scalability**: Handling many requests (load balancing, concurrency).
   * **Versioning**: Ensuring new model versions don’t break existing applications.
2. **Constraints of implementing a predictive model as a service**
   * **Latency requirements**: Must respond within certain time limits.
   * **Data privacy** and security: Especially important in government agencies.
   * **Infrastructure and cost**: Hosting the service, compute resources.
3. **Requirements for data acquisition, storage, and processing**
   * **Reliable ingestion** of large volumes of application data.
   * **Secure storage** (data encryption, access controls).
   * **Preprocessing consistency**: The same transformations are applied during both training and inference.
4. **Monitoring components for reliable execution**
   * **MLflow** or equivalent to track model performance, drift, logs.
   * **Application logs** to track errors, uptime (e.g., with monitoring tools like Grafana, Prometheus).
   * **Alerting systems** (e.g., Slack, email) if performance metrics degrade or data drift is detected.
5. **Design of your system & visual draft**
   * Refer to the architecture diagram in **Step 1**.
6. **Provide a link for your audience to reproduce your code**
   * Host your code in a public GitHub repo or similar, and share that link in your presentation.

**Bringing it all together**

1. **Create a small prototype** locally:
   * Data → Train → Evaluate → REST API → Test predictions.
2. **Add monitoring** with MLflow to track runs and logs.
3. **Automate** with an MLOps pipeline (GitHub Actions or similar).
4. **Simulate** data drift or monthly updates → Ensure pipeline re-trains and updates the model service.
5. **Document** or present your work with a **live demonstration** showing:
   * The architecture,
   * The code repo link,
   * The pipeline logs/triggers,
   * The final REST endpoint usage.

By following these steps, you demonstrate not only the data science model-building portion but also how to operationalize (MLOps) that model—aligning with the requirements of your government agency project. Good luck with your implementation and presentation!