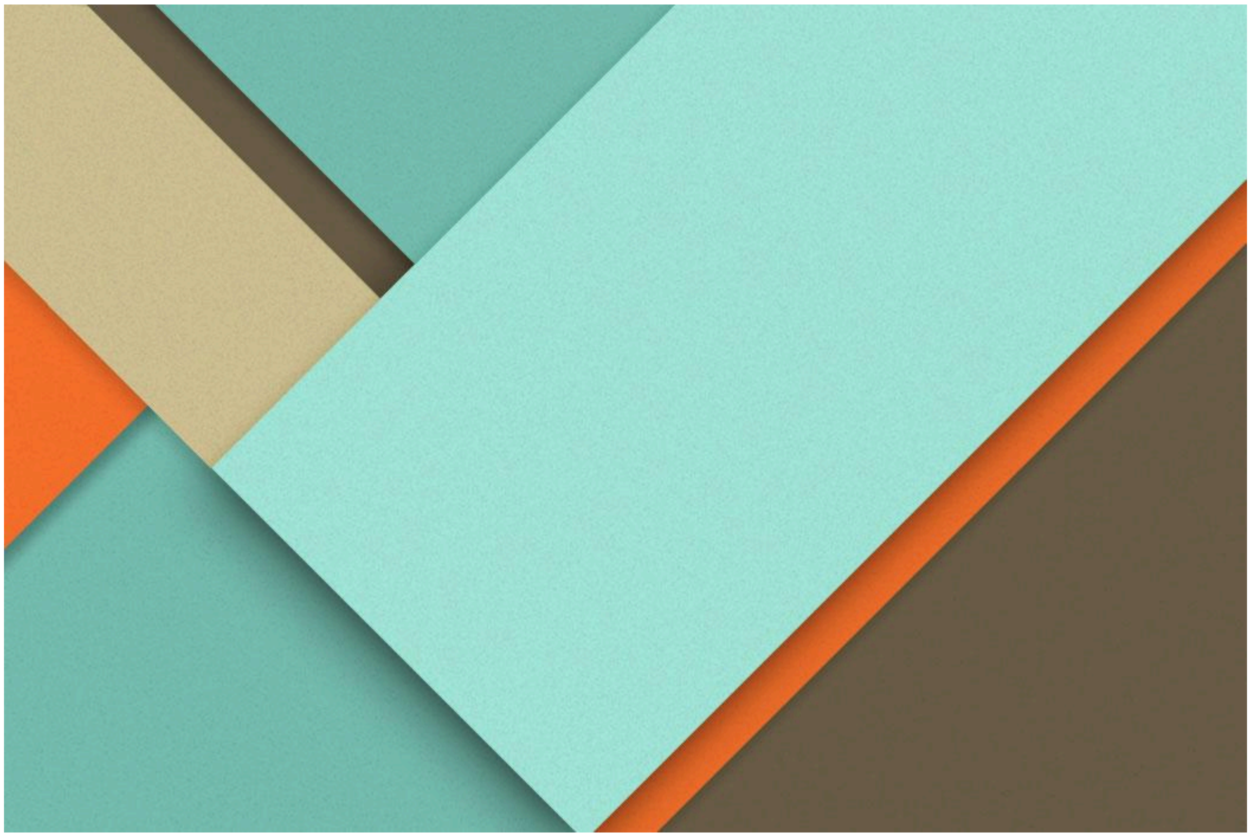


DeBlock Case: Suspension Probability Problem

02.10.2024

Eser İnan Arslan
Data Scientist
eserinanarslan@gmail.com

CONTENTS

Overview

The objective of this task is to demonstrate customer risk segmentation and off-boarding using machine learning techniques. Leveraging labeled customer and transaction data, a model will be developed to predict customer risk levels and identify individuals for off-boarding. The model is then applied to an unlabeled dataset to perform inference and generate predictions. This project evaluates skills in handling both labeled and unlabeled data to build a robust classification model, as well as the ability to develop a scalable solution for real-world applications.

Additionally, the report outlines steps for deploying the model on Google Cloud Platform (GCP), including performance monitoring and scalability strategies. Key performance indicators (KPIs) such as accuracy, precision, recall, and specific risk metrics will be tracked via dashboards and alerts. Considerations for scaling the model, addressing traffic surges, and ensuring system resilience are also discussed, ensuring the model can handle significant increases in data volume and maintain optimal performance.

A. Data Exploration and Preprocessing

1. Data Loading and Initial Exploration

The dataset consists of two key components: labeled user information and labeled transaction data. After successfully reading both datasets, the user data contained 500 rows and 16 columns, while the transaction data had 5945 rows and 11 columns. Initial data exploration revealed the need to address duplicates and missing values:

- After removing duplicate records, the labeled user information dataset was reduced to 455 rows.
- Further cleaning was done by removing rows with NaN values, which resulted in 432 rows of user data.

The two datasets were merged on the `user_id` column, yielding a combined dataset with 6622 rows and 26 columns. Data types were explored, and it was confirmed that there were no NaN values in critical columns like `user_id`, `session_id`, and `id`. However, columns such as `completed_at` contained missing values, which were handled later in the preprocessing steps.

2. Handling Missing Values and Outliers

For handling missing values, a function was created to fill object-type columns with `'unknown'` and numeric columns with `0`. This ensured that no missing data remained in the dataset, preserving the integrity of the features for the machine learning model.

Outliers were addressed during the feature engineering process, especially in transaction amounts. Columns such as `amount`, `total_amount`, and `average_amount` were computed to account for the sum and mean transaction values per user. This step helped identify extreme transaction values for further analysis.

3. Feature Engineering

Several key features were engineered to enhance the model's predictive power:

- **Transaction Frequency and Total Amount**: New features like transaction_count and total_amount were created by grouping transactions by user_id to capture user activity levels.
- **Process Time**: A process_time feature was calculated by computing the difference between created_at and completed_at, which captured the time taken for a transaction to complete.
- **Country and Continent Mapping**: The phone_number_country_code, country_of_residence, document_issuing_country, and locale columns were compared and recoded to indicate whether they matched. Additionally, countries were mapped to continents, and the continent names were converted into numeric codes for use in the model.

4. Normalization and Binning

Normalization was applied to key numerical columns (amount, process_time, transaction_count) using Min-Max scaling to ensure all features were on a comparable scale. Binning was performed for the total_amount column to group users based on transaction ranges, creating a feature that could segment users according to their financial activity.

5. Categorical Encoding

Categorical features such as country_of_residence, reason_to_use_app, and occupation were one-hot encoded to create dummy variables, converting them into a numerical format suitable for machine learning algorithms. This step ensured that all categorical data was properly prepared for model training.

6. Correlated Feature Selection

A correlation matrix was calculated to identify the most relevant features for predicting customer risk (i.e., the user_current_state). Features with a correlation above 0.05 were selected for inclusion in the model. This reduced the dimensionality of the dataset while retaining the most predictive variables.

The final dataset was then split into training and test sets, preparing it for model training and evaluation.

B. Model Training & Evaluation

1. Overview of Models

For this task, two classification algorithms were employed: **Random Forest Classifier** and **Gaussian Naive Bayes** (with both uncalibrated and isotonic calibration). These models were chosen based on their suitability for classifying binary outcomes, such as predicting whether a customer should be off-boarded due to risk (SUSPENDED) or allowed to remain (ACTIVE).

2. Random Forest Classifier

Hyperparameter Tuning: The Random Forest model was tuned using **GridSearchCV**, optimizing the following parameters:

- `n_estimators`: Number of trees in the forest (150, 250, 350),
- `max_depth`: Maximum depth of each tree (40, 35, 30, 25, 20),
- `min_samples_split`: Minimum number of samples required to split a node (5, 3, 2),
- `min_samples_leaf`: Minimum number of samples required to form a leaf node (5, 3, 2).

After running the grid search with 5-fold cross-validation, the optimal hyperparameters were determined as follows:

- `n_estimators`: 150,
- `max_depth`: 30,
- `min_samples_split`: 2,
- `min_samples_leaf`: 2.

Advantages of Random Forest:

- **Robustness to Overfitting**: Random Forest reduces the risk of overfitting by averaging multiple decision trees.
- **Handling Imbalanced Data**: It performs well even in the presence of class imbalances, which is important for detecting relatively rare cases like customer suspension.
- **Feature Importance**: Random Forest provides insight into feature importance, helping interpret which features are most relevant in predicting customer risk.

Results:

- **Precision**: 0.97 for the ACTIVE class and 0.92 for the SUSPENDED class.
- **Recall**: 0.97 for the ACTIVE class and 0.93 for the SUSPENDED class.
- **F1-Score**: 0.97 for the ACTIVE class and 0.93 for the SUSPENDED class.
- **Accuracy**: 96% overall.

Random Forest showed **high precision and recall** for both classes, indicating it is effective in minimizing both false positives and false negatives, making it highly reliable for tasks that require a balance between precision and recall.

3. Gaussian Naive Bayes

The **Gaussian Naive Bayes (GNB)** algorithm was applied in both uncalibrated and calibrated versions (using isotonic calibration). GNB assumes independence between features and is computationally efficient for large datasets.

Advantages of Gaussian Naive Bayes:

- **Simplicity**: It is computationally fast and works well with smaller datasets or as a baseline model.
- **Handling of Continuous Variables**: GNB is appropriate for continuous data like transaction amounts due to the Gaussian distribution assumption.

Uncalibrated Gaussian Naive Bayes Results:

- **Precision**: 0.98 for the ACTIVE class and 0.42 for the SUSPENDED class.
- **Recall**: 0.45 for the ACTIVE class and 0.98 for the SUSPENDED class.
- **F1-Score**: 0.61 for the ACTIVE class and 0.58 for the SUSPENDED class.
- **Accuracy**: 60% overall.

While the uncalibrated GNB achieved high recall for the SUSPENDED class, it struggled with precision, resulting in a large number of false positives. This makes it less suitable for cases where precision is prioritized.

Isotonic Calibrated Gaussian Naive Bayes Results:

- **Precision**: 0.83 for the ACTIVE class and 0.75 for the SUSPENDED class.
- **Recall**: 0.93 for the ACTIVE class and 0.51 for the SUSPENDED class.
- **F1-Score**: 0.87 for the ACTIVE class and 0.61 for the SUSPENDED class.
- **Accuracy**: 81% overall.

Calibration improved the precision of the model significantly, particularly for the SUSPENDED class, but it lowered the recall. This calibrated model provides a better balance between precision and recall than the uncalibrated version.

4. Model Comparison

- **Random Forest**: This model excelled in both **precision** and **recall**, making it well-suited for this problem. The high precision ensures fewer false positives, meaning fewer good customers are mistakenly off-boarded. High recall reduces the likelihood of false negatives, ensuring that most at-risk customers are identified.

- **Gaussian Naive Bayes (Calibrated)**: After calibration, this model provided a decent balance between precision and recall but was outperformed by the Random Forest in both areas.

5. Model Conclusion

Based on these results, **Random Forest** is the preferred model for customer risk segmentation and off-boarding due to its superior performance across all metrics. It is especially effective in scenarios where both precision and recall are critical, helping ensure that the model accurately identifies customers at risk without generating too many false positives.

C. Applying the Model to Unlabeled Data & Identifying Offboarding Customers

1. Risk Prediction

The same methodologies and preprocessing steps used during model training on the labeled dataset were consistently applied to the unlabeled customer and transaction datasets. This included handling missing values, normalizing numerical features, one-hot encoding categorical variables, and performing feature engineering on relevant features such as transaction frequency and total amount. These steps ensured that the input data for the model was processed in a manner identical to the training phase, guaranteeing consistency in predictions between the labeled and unlabeled datasets.

2. Feature Mapping and Filtering

During the preprocessing stage, the same **binning**, **mapping**, and **feature selection** techniques used on the training dataset were applied to the unlabeled dataset. This included leveraging the best-correlated features identified during training, without re-evaluating the test dataset's distribution. The thresholds and filtering criteria, such as the boundaries for transaction amount bins or feature correlations, were derived solely from the training data. This method helped avoid any leakage of information from the test dataset and ensured the integrity of the model's predictions.

For example, categorical variables like country_of_residence and numerical variables such as total_amount were preprocessed using the same binning and scaling techniques as the training data. The mappings for country codes and continent codes were also reused to ensure the dataset followed the same structure as the labeled data.

3. Feature Deduplication and Enrichment

In the unlabeled dataset, certain features like category_id and date_of_birth were either missing or contained too many NaN values to be useful. Efforts were made to enrich the test dataset using information from the training dataset; however, the results were not always adequate:

- **Category Data**: The category_id feature was enriched based on the training data, but it was found that over 70% of the entries in the unlabeled dataset had missing values. As a result, this feature was not used in the final model due to its limited value.
- **Date of Birth**: The date_of_birth feature was excluded from both the training and test datasets. In the training data, users in the labeled dataset differed significantly from those in the unlabeled dataset, and attempting to transfer this information led to a large number of missing values. Given its inconsistency and irrelevance, it was dropped entirely.

By handling these missing or irrelevant columns effectively, we ensured that the model focused only on the most important and reliable features.

4. Model Application

The models trained during the initial phase were directly applied to the unlabeled dataset without retraining. The Random Forest and Gaussian Naive Bayes classifiers were used to predict whether a customer falls into the SUSPENDED category, which would indicate they are at risk and potentially need off-boarding. The models were not retrained on the test data, ensuring the predictions were entirely based on the knowledge learned from the training dataset.

The predictions from the model were then used to identify at-risk customers for off-boarding, based on both the risk score predicted by the model and transaction patterns that align with the business objectives of minimizing customer risk while maintaining operational efficiency.

D. Scoring Methodology

To further refine the model predictions and produce a more nuanced risk score for each customer, a scoring function was developed. This approach allows for the integration of the strengths of multiple models by calculating a **suspension score** based on a weighted combination of predictions from different classifiers: Random Forest, Naive Bayes, and Calibrated Naive Bayes.

1. Scoring Function Overview

The function `calc_user_suspended_status_score` computes a weighted sum of the predicted probabilities from each model:

- **Random Forest Probability (`random_for_probability`)**: Weight = 0.96,
- **Naive Bayes Probability (`naive_bias_probability`)**: Weight = 0.60,
- **Calib Naive Bayes Probability (`cal_naive_bias_probability`)**: Weight = 0.81.

The final **Suspension Score** is a combination of these model outputs, with each model contributing according to its performance in previous evaluations. This ensures that more reliable models have a greater influence on the final score, balancing the strengths of each algorithm.

2. Normalization and Interpretation

Once the combined score is calculated, the data is **normalized** between 0 and 1 to ensure that all predictions are on a comparable scale. This normalized score provides a final risk measure, where:

- **Scores close to 1** indicate a high likelihood of customer suspension (high risk),
- **Scores close to 0** indicate a low likelihood of suspension (low risk).

The flexibility of this approach allows for adjusting the final classification based on business needs by applying different thresholds to the normalized score. For example:

- A threshold of 0.75 could be used for highly risk-averse scenarios, ensuring only the riskiest customers are flagged for off-boarding.
- A lower threshold, such as 0.50, could be used in scenarios where more aggressive risk management is desired, potentially flagging more customers.

3. Weighting Based on Model Performance

By weighting each model according to its accuracy (or other performance metrics), the final score is more reliable than relying on a single model. The Random Forest model, which had the highest accuracy and precision, is given the highest weight, while the Naive Bayes models are included to provide additional insights based on their own strengths, particularly in recall and calibrated performance.

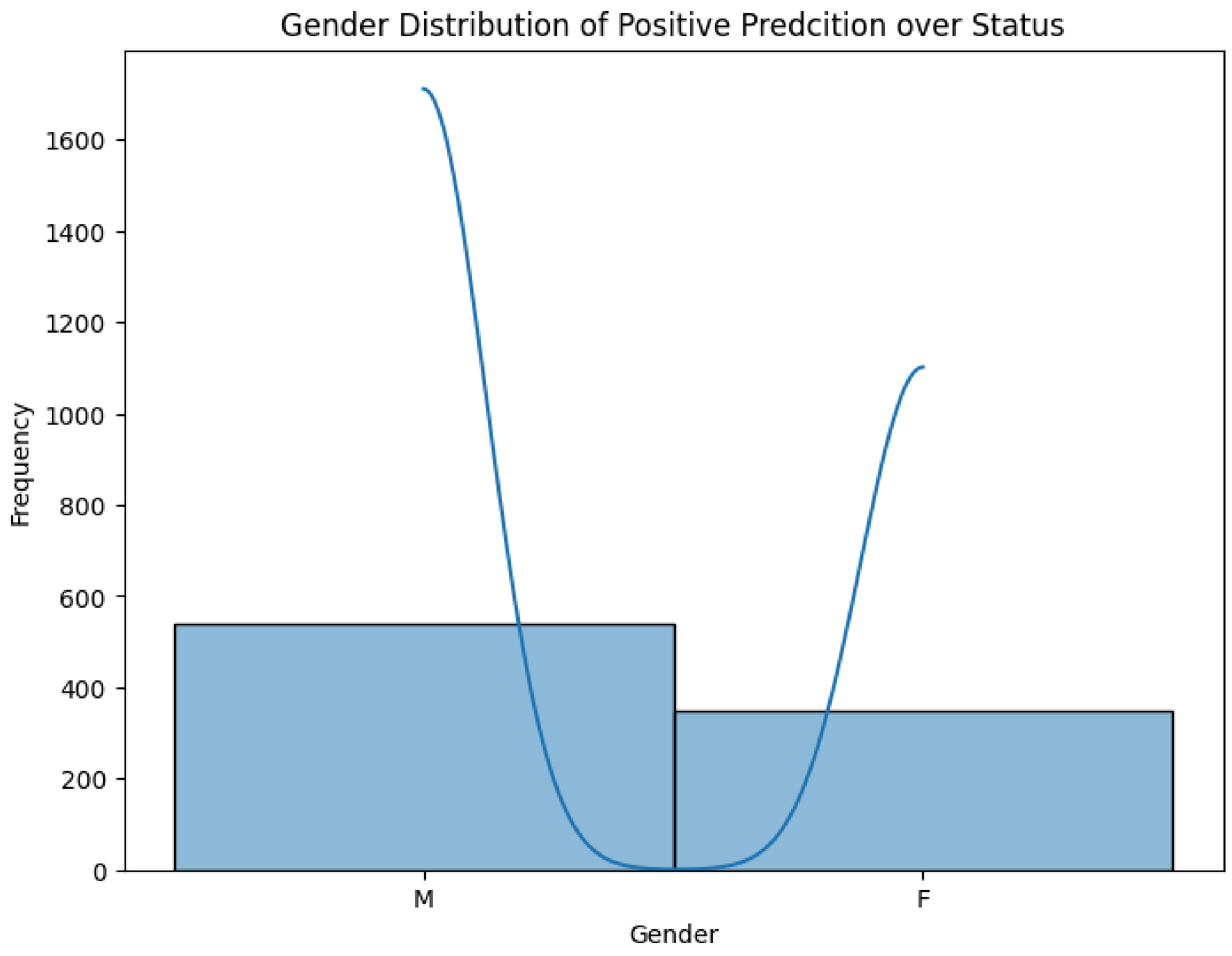
This approach effectively leverages multiple models and their diverse characteristics to produce a robust risk prediction system, adaptable to varying levels of risk tolerance.

E. Visualization Based on Random Forest Predictions

To further analyze the **SUSPENDED** customers identified by the Random Forest model, the following visualizations will provide insights into key characteristics of these customers:

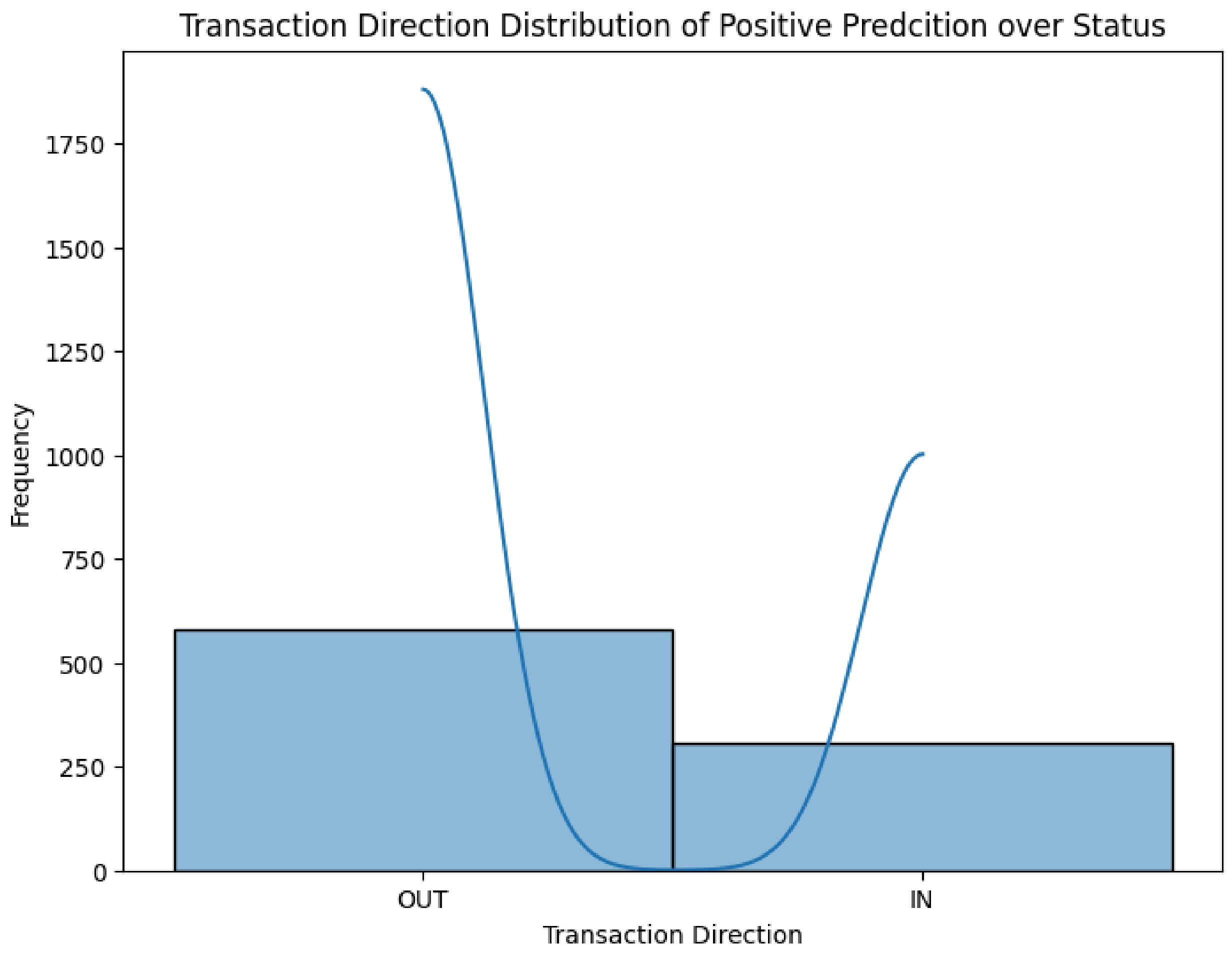
1. Gender Distribution of SUSPENDED Customers (According to Random Forest)

This visualization shows the distribution of genders among customers predicted to be **SUSPENDED** by the Random Forest model. Understanding the gender distribution can provide insights into whether certain gender groups are more likely to be flagged for suspension based on their risk profile.


Gender Distribution of Positive Predcition over Status
Frequency
0
200
400
600
800
1000
1200
1400
1600
M
F
Gender

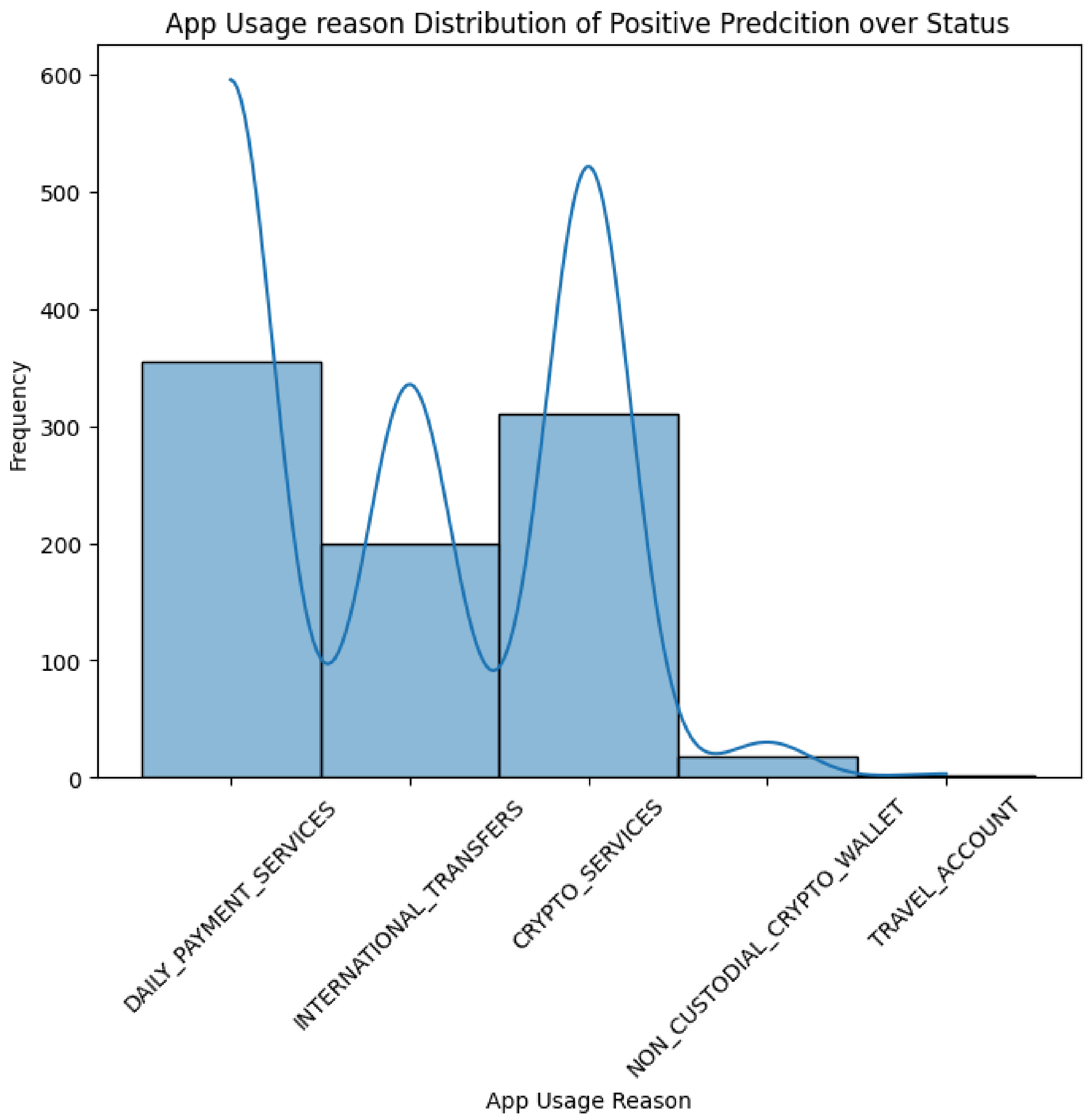
2. Transaction Direction Distribution of SUSPENDED Customers (According to Random Forest)

This plot examines the **transaction direction** (e.g., inbound or outbound) for customers who were predicted to be **SUSPENDED**. By analyzing the direction of transactions, we can understand if customers with a certain type of transaction pattern are



3. Reason for Using the Application for SUSPENDED Customers (According to Random Forest)

This visualization breaks down the **reason for using the application** among customers predicted to be SUSPENDED by the Random Forest model. It highlights whether certain user motivations correlate more strongly with suspension, providing valuable insights for customer risk management strategies.



F. Deploying the Model on Google Cloud Platform (GCP)

1. Containerization and Packaging

Before deploying the machine learning model to a production environment on GCP, it is important to containerize the application. Docker is used to ensure that the model and all its dependencies are packaged together, making deployment easier and scalable. Below are the steps involved in setting up the containerization.

Docker Setup

The provided **Dockerfile** ensures that the necessary dependencies (e.g., Python, Flask, Scikit-learn) and the application code are packaged into a single Docker image. The key steps to containerize this service include:

1. **Base Image**: We use a lightweight Python base image (`python:3.12-slim`), which is optimized for running Python applications. Using a slim version reduces the image size, which improves load times and deployment speed.
2. **Adding Requirements**: The `requirements.txt` file is added to the Docker container, and the required Python libraries such as `numpy`, `pandas`, `flask`, and `scikit-learn` are installed using `pip`. This ensures that all the necessary dependencies are installed in the container.

dockerfile

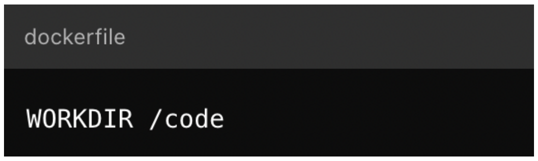
```
# Add the requirements.txt to the container
ADD requirements.txt /.

# Install the required Python packages
RUN pip install --upgrade pip && pip install -r /requirements.txt
```

3. **Adding Code**: The application code and database are copied to the container using the `ADD . /code/` command. This ensures that the Python scripts, SQLite database, and configuration files are available inside the container.

dockerfile

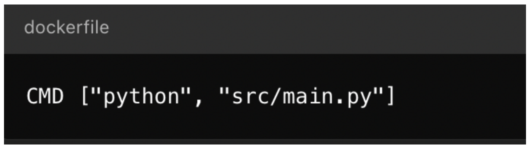
```
# Add the code and database to the container
ADD . /code/
```

4. **Setting the Working Directory**: The `WORKDIR` command is used to define the directory where the container will execute the application. Here, we set the directory to `/code` to run the main application file (`main.py`).



```
WORKDIR /code
```

5. **Running the Application**: The **CMD** directive specifies the command to run the Flask application when the container starts. The application listens on the host `0.0.0.0` and port `1001` as defined in the `config.ini` file.



```
CMD ["python", "src/main.py"]
```

2. Best Practices for Docker

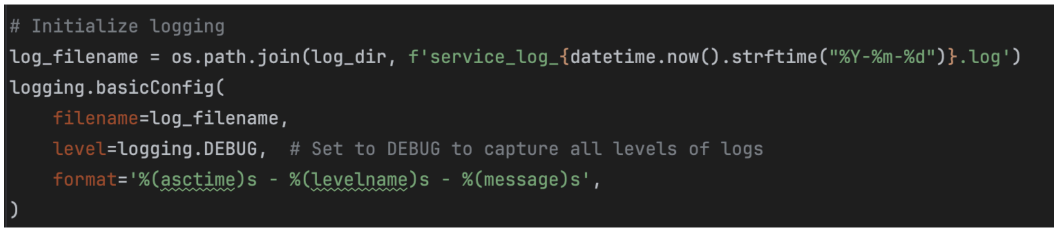
- **Lightweight Image**: The use of `python:3.12-slim` ensures that the Docker image remains small, reducing the overhead in deployment and scaling on GCP.
- **Caching Dependencies**: The `ADD` command is used for `requirements.txt` and is followed by `pip install` to cache dependencies, which optimizes the build process and ensures that dependencies are only downloaded when there are changes.
- **Health Checks**: Implementing a health check in the Dockerfile is recommended for production environments. This can be done using the `HEALTHCHECK` directive to monitor the status of the Flask service.

3. Flask Service and Logging

In the Flask application, we have set up two GET endpoints:

1. `/get_all_results`: Returns a list of customer results, limited to 100 entries.
2. `/search_user_result`: Takes a `user_id` parameter and returns the specific results for the requested user.

Logging has been integrated into the application using Python's built-in logging library. Logs are generated in a `logs` directory, ensuring that any issues during the API calls or database access are recorded.



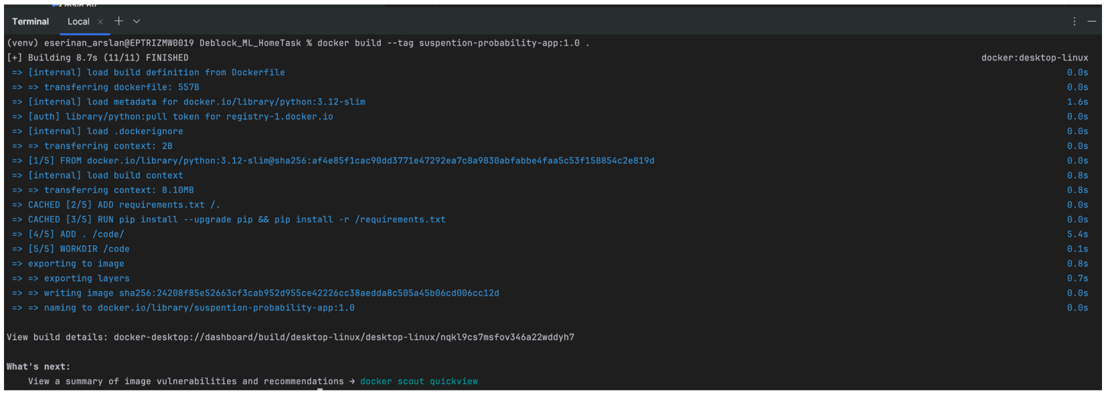
```python
# Initialize logging
log_filename = os.path.join(log_dir, f'service_log_{datetime.now().strftime("%Y-%m-%d")}.log')
logging.basicConfig(
    filename=log_filename,
    level=logging.DEBUG,  # Set to DEBUG to capture all levels of logs
    format='%(asctime)s - %(levelname)s - %(message)s',
)
```

This logging mechanism ensures that any errors or successful operations during the execution of the service are documented. Additionally, logs help in diagnosing issues that may occur during deployment or when the service is in production on GCP.

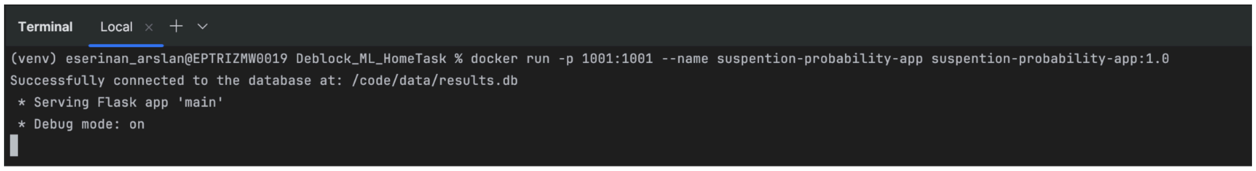
4. Packaging the Docker Image

After writing the **Dockerfile** and setting up the Flask application with the SQLite database, the Docker image can be built and tested locally using:

Bash:

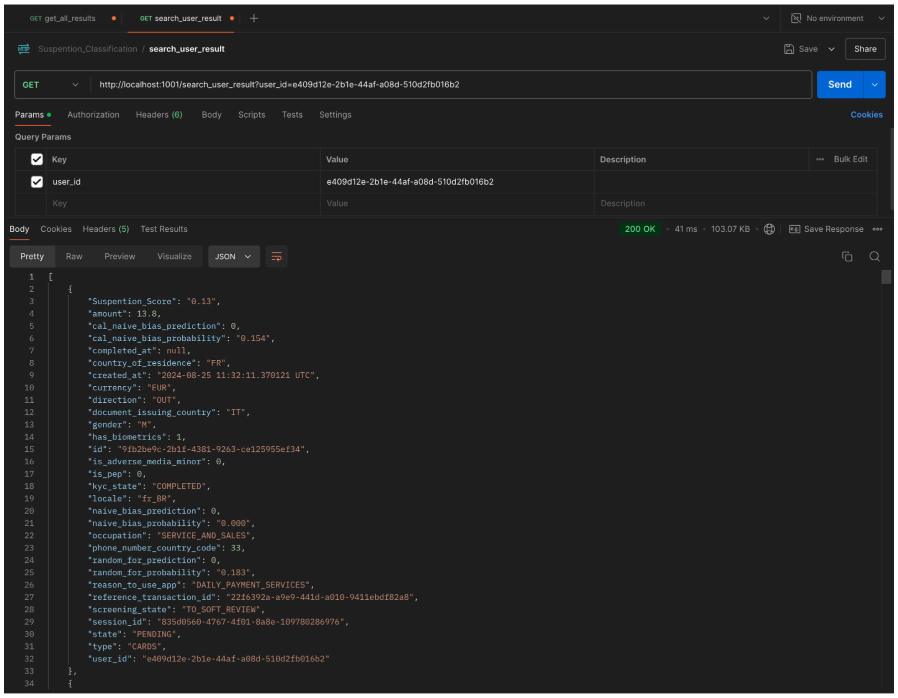


```
(venv) eserinan_arslan@EPTRIZMW0019 Deblock_ML_HomeTask % docker build --tag suspention-probability-app:1.0 .
[+] Building 8.7s (11/11) FINISHED                                                                              docker:desktop-linux
 => [internal] load build definition from Dockerfile                                                                            0.0s
 => => transferring dockerfile: 557B                                                                                            0.0s
 => [internal] load metadata for docker.io/library/python:3.12-slim                                                             1.6s
 => [auth] library/python:pull token for registry-1.docker.io                                                                   0.0s
 => [internal] load .dockerignore                                                                                               0.0s
 => => transferring context: 2B                                                                                                 0.0s
 => [1/5] FROM docker.io/library/python:3.12-slim@sha256:af4e85f1cac90dd3771e47292ea7c8a9830abfabbe4faa5c53f158854c2e819d       0.0s
 => [internal] load build context                                                                                               0.8s
 => => transferring context: 8.10MB                                                                                             0.8s
 => CACHED [2/5] ADD requirements.txt /.                                                                                        0.0s
 => CACHED [3/5] RUN pip install --upgrade pip && pip install -r /requirements.txt                                              0.0s
 => [4/5] ADD . /code/                                                                                                          5.4s
 => [5/5] WORKDIR /code                                                                                                         0.1s
 => exporting to image                                                                                                          0.8s
 => => exporting layers                                                                                                         0.7s
 => => writing image sha256:24208f85e52663cf3cab952d955ce42226cc38aedda8c505a45b06cd006cc12d                                    0.0s
 => => naming to docker.io/library/suspention-probability-app:1.0                                                               0.0s

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/nqkl9cs7msfov346a22wddyh7

What's next:
    View a summary of image vulnerabilities and recommendations → docker scout quickview
```



```
(venv) eserinan_arslan@EPTRIZMW0019 Deblock_ML_HomeTask % docker run -p 1001:1001 --name suspention-probability-app suspention-probability-app:1.0
Successfully connected to the database at: /code/data/results.db
 * Serving Flask app 'main'
 * Debug mode: on
```

5. Rest Service



Suspention_Classification / search_user_result

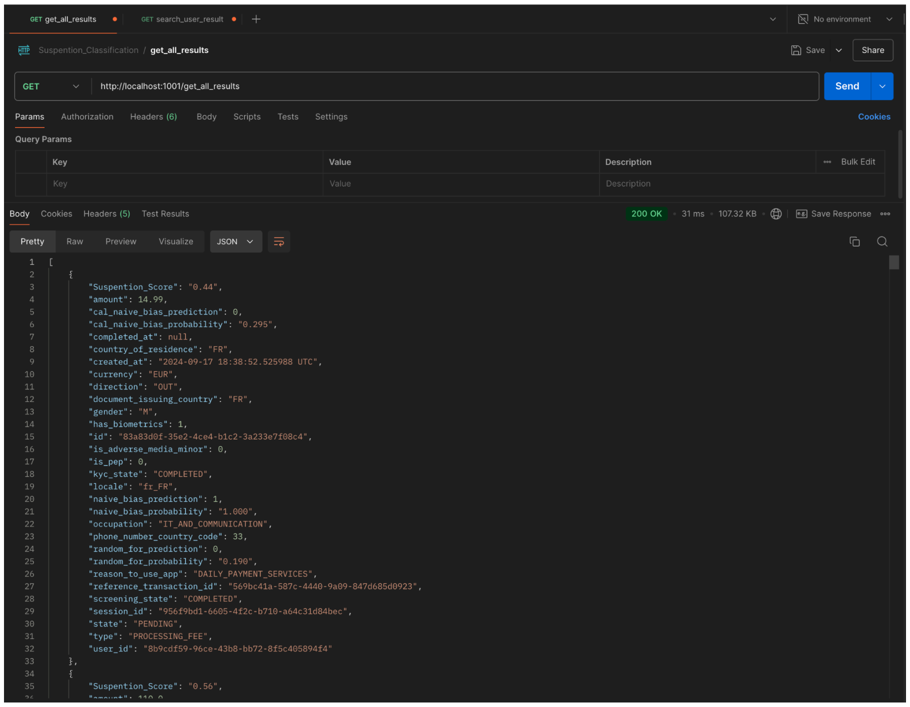
GET http://localhost:1001/search_user_result?user_id=e409d12e-2b1e-44af-a08d-510d2fb016b2

Query Params

✓	Key	Value	Description
✓	user_id	e409d12e-2b1e-44af-a08d-510d2fb016b2	
	Key	Value	Description

200 OK • 41 ms • 103.07 KB

```
[
    {
        "Suspention_Score": "0.13",
        "amount": 13.8,
        "cal_naive_bias_prediction": 0,
        "cal_naive_bias_probability": "0.154",
        "completed_at": null,
        "country_of_residence": "FR",
        "created_at": "2024-08-25 11:32:11.370121 UTC",
        "currency": "EUR",
        "direction": "OUT",
        "document_issuing_country": "IT",
        "gender": "M",
        "has_biometrics": 1,
        "id": "9fb2be9c-2b1f-4381-9263-ce125955ef34",
        "is_adverse_media_minor": 0,
        "is_pep": 0,
        "kyc_state": "COMPLETED",
        "locale": "fr_BR",
        "naive_bias_prediction": 0,
        "naive_bias_probability": "0.000",
        "occupation": "SERVICE_AND_SALES",
        "phone_number_country_code": 33,
        "random_for_prediction": 0,
        "random_for_probability": "0.183",
        "reason_to_use_app": "DAILY_PAYMENT_SERVICES",
        "reference_transaction_id": "22f6392a-a9e9-441d-a010-9411ebdf82a8",
        "screening_state": "TO_SOFT_REVIEW",
        "session_id": "835d0560-4767-4f01-8a8e-109780286976",
        "state": "PENDING",
        "type": "CARDS",
        "user_id": "e409d12e-2b1e-44af-a08d-510d2fb016b2"
    },
    {
```



Suspention_Classification / get_all_results

GET http://localhost:1001/get_all_results

Query Params

	Key	Value	Description
	Key	Value	Description

200 OK • 31 ms • 107.32 KB

```
[
    {
        "Suspention_Score": "0.44",
        "amount": 14.99,
        "cal_naive_bias_prediction": 0,
        "cal_naive_bias_probability": "0.295",
        "completed_at": null,
        "country_of_residence": "FR",
        "created_at": "2024-09-17 18:38:52.525988 UTC",
        "currency": "EUR",
        "direction": "OUT",
        "document_issuing_country": "FR",
        "gender": "M",
        "has_biometrics": 1,
        "id": "83a83d0f-35e2-4ce4-b1c2-3a233e7f08c4",
        "is_adverse_media_minor": 0,
        "is_pep": 0,
        "kyc_state": "COMPLETED",
        "locale": "fr_FR",
        "naive_bias_prediction": 1,
        "naive_bias_probability": "1.000",
        "occupation": "IT_AND_COMMUNICATION",
        "phone_number_country_code": 33,
        "random_for_prediction": 0,
        "random_for_probability": "0.190",
        "reason_to_use_app": "DAILY_PAYMENT_SERVICES",
        "reference_transaction_id": "569bc41a-587c-4440-9a09-847d685d0923",
        "screening_state": "COMPLETED",
        "session_id": "956f9bd1-6605-4f2c-b710-a64c31d84bec",
        "state": "PENDING",
        "type": "PROCESSING_FEE",
        "user_id": "8b9cdf59-96ce-43b8-bb72-8f5c405894f4"
    },
    {
        "Suspention_Score": "0.56",
```

6. Next Steps: Deploying to GCP

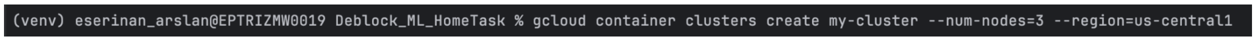
Once the Docker image is created and tested locally, the next steps will involve pushing the Docker image to Google Container Registry (GCR) and deploying it to Google Kubernetes Engine (GKE) or Cloud Run, depending on the deployment strategy you prefer for scaling and managing traffic.

G. Deploying with GCP Services

To deploy the model using **Google Kubernetes Engine (GKE)**, follow these steps:

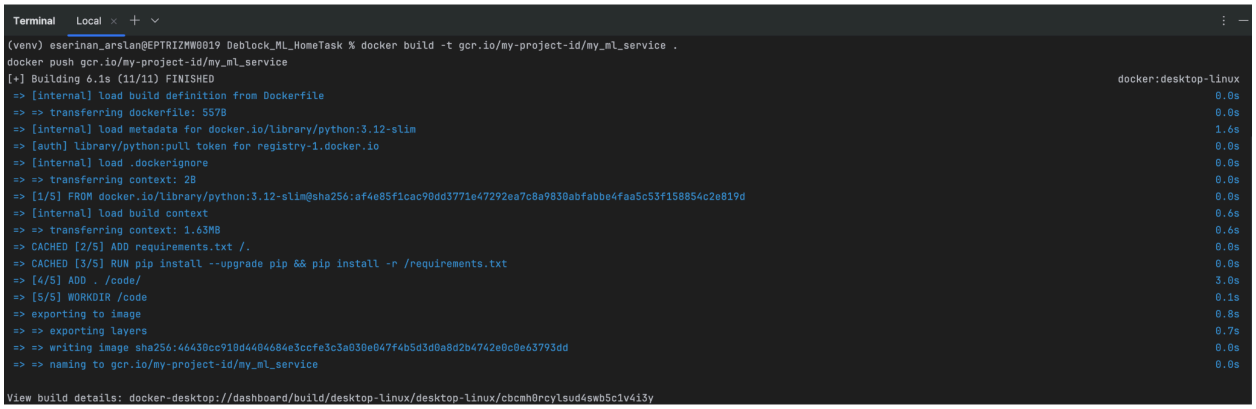
1. Create a Kubernetes Cluster

- **Setup GKE**: First, create a GKE cluster via the Google Cloud Console or using the `gcloud` CLI:



```
(venv) eserinan_arslan@EPTRIZMW0019 Deblock_ML_HomeTask % gcloud container clusters create my-cluster --num-nodes=3 --region=us-central1
```

- **Push Docker Image to GCR**: Build the Docker image and push it to Google Container Registry (GCR):



```
(venv) eserinan_arslan@EPTRIZMW0019 Deblock_ML_HomeTask % docker build -t gcr.io/my-project-id/my_ml_service .
docker push gcr.io/my-project-id/my_ml_service
[+] Building 6.1s (11/11) FINISHED                                                                      docker:desktop-linux
 => [internal] load build definition from Dockerfile                                                                  0.0s
 => => transferring dockerfile: 557B                                                                                  0.0s
 => [internal] load metadata for docker.io/library/python:3.12-slim                                                   1.6s
 => [auth] library/python:pull token for registry-1.docker.io                                                         0.0s
 => [internal] load .dockerignore                                                                                     0.0s
 => => transferring context: 2B                                                                                       0.0s
 => [1/5] FROM docker.io/library/python:3.12-slim@sha256:af4e85f1cac90dd3771e47292ea7c8a9830abfabbe4faa5c53f158854c2e819d  0.0s
 => [internal] load build context                                                                                     0.6s
 => => transferring context: 1.63MB                                                                                   0.6s
 => CACHED [2/5] ADD requirements.txt /.                                                                              0.0s
 => CACHED [3/5] RUN pip install --upgrade pip && pip install -r /requirements.txt                                    0.0s
 => [4/5] ADD . /code/                                                                                                3.0s
 => [5/5] WORKDIR /code                                                                                               0.1s
 => exporting to image                                                                                                0.8s
 => => exporting layers                                                                                               0.7s
 => => writing image sha256:46430cc910d4404684e3ccfe3c3a030e047f4b5d3d0a8d2b4742e0c0e63793dd                          0.0s
 => => naming to gcr.io/my-project-id/my_ml_service                                                                   0.0s

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/cbcmh0rcylsud4swb5c1v4i3y
```

2. Deploy the Model on GKE

- **Create Deployment**: Write a Kubernetes deployment YAML file to define how the Docker container will run on the cluster.

Yaml:

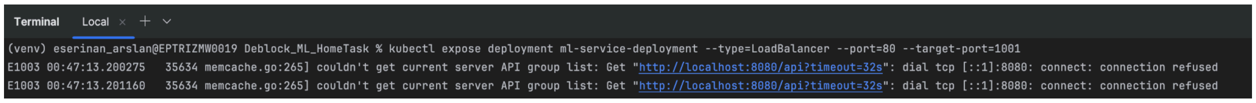
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ml-service-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: ml-service
  template:
    metadata:
      labels:
        app: ml-service
    spec:
      containers:
      - name: ml-service
        image: gcr.io/my-project-id/my_ml_service
        ports:
        - containerPort: 1001
```

Apply the deployment:

```
kubectl apply -f deployment.yaml
```

3. Expose the Deployment

- **Service Exposure**: Create a service to expose the deployment as an external load-balanced endpoint:



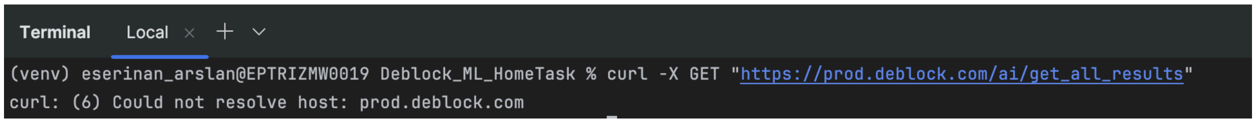
```
(venv) eserinan_arslan@EPTRIZMW0019 Deblock_ML_HomeTask % kubectl expose deployment ml-service-deployment --type=LoadBalancer --port=80 --target-port=1001
E1003 00:47:13.200275   35634 memcache.go:265] couldn't get current server API group list: Get "http://localhost:8080/api?timeout=32s": dial tcp [::1]:8080: connect: connection refused
E1003 00:47:13.201160   35634 memcache.go:265] couldn't get current server API group list: Get "http://localhost:8080/api?timeout=32s": dial tcp [::1]:8080: connect: connection refused
```

GKE will automatically assign a public IP to the service, which can be accessed via the root URL.

4. Create a RESTful API

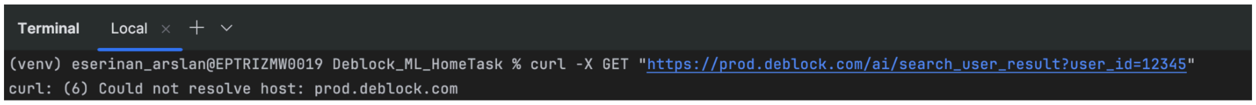
- **API Structure**: Using Flask (as in the previous step), the RESTful API will have endpoints like:
  - `/get_all_results`: Returns results for batch inference.
  - `/search_user_result`: Takes a `user_id` parameter and returns the result for that user.

Example API call for batch inference:



```
(venv) eserinan_arslan@EPTRIZMW0019 Deblock_ML_HomeTask % curl -X GET "https://prod.deblock.com/ai/get_all_results"
curl: (6) Could not resolve host: prod.deblock.com
```

Example API call for real-time inference:



```
(venv) eserinan_arslan@EPTRIZMW0019 Deblock_ML_HomeTask % curl -X GET "https://prod.deblock.com/ai/search_user_result?user_id=12345"
curl: (6) Could not resolve host: prod.deblock.com
```

H. Data Input and Output Handling

1. Google Cloud Storage (GCS)

For Batch Processing: Store inputs/outputs in **Google Cloud Storage** for scalability and low cost, and use **BigQuery** for analyzing results or querying large datasets.

- **Use Case**: GCS is ideal for storing large datasets, input files, and model predictions (outputs). It supports a wide variety of data formats (CSV, JSON, Parquet, etc.) and can handle both structured and unstructured data.

- **Advantages**:
    - **Scalability**: GCS can handle virtually unlimited amounts of data, making it suitable for batch input/output data.
    - **Integration**: GCS integrates seamlessly with GKE, Dataflow, and other GCP services.
    - **Low Cost**: It offers cost-effective storage, especially for infrequent access (via coldline/archival storage).

- **Example Usage**:
    - Input files (e.g., transaction history or user details) can be uploaded to a GCS bucket (`gs://my_bucket/inputs/`).
    - Output predictions (e.g., customer risk scores) can be saved to a GCS location (`gs://my_bucket/outputs/`).

2. Google Cloud Firestore or Cloud SQL

For Real-Time Processing: Use **Firestore** or **Cloud SQL** for low-latency storage of prediction outputs, and consider **Pub/Sub** for streaming inputs in a real-time environment.

- **Use Case**: For real-time application scenarios, where the model needs to frequently read or update small, structured data (e.g., customer profiles or risk statuses), Firestore (NoSQL) or Cloud SQL (relational SQL) are suitable options.
- **Advantages**:
    - **Low-Latency Access**: Ideal for serving real-time predictions or transactional data that need to be quickly retrieved or updated.
    - **Structured Storage**: Use Firestore for flexible, schema-less data storage, or Cloud SQL for traditional relational database use cases.
- **Example Usage**:
    - Customer profiles and predictions can be stored in Cloud Firestore (`projects/my_project/databases/(default)/documents/users`).
    - Cloud SQL can be used to track model prediction results and customer offboarding decisions (`my_project.customer_risk`).

I. Monitoring and Logging

1. Collecting Logs

To track logs in real-time, I recommend using **Google Cloud Logging**. It seamlessly captures logs from GKE, Cloud Run, or VMs and stores them for analysis. You should enable logging for your project and integrate it into your application code. Using Python’s `logging` module, you can easily track errors, predictions, and events. These logs are automatically aggregated, and you can query them in real-time using the **Logs Explorer**.

2. Monitoring Model Performance

For performance monitoring, **Google Cloud Monitoring** is key. It lets you create custom dashboards and set up alerts. Critical metrics to monitor include:

- **Latency**: Track prediction response time to ensure real-time performance.
- **Prediction Accuracy**: Measure precision, recall, and F1-score, and store these metrics in **BigQuery** for detailed analysis.
- **Request Volume**: Monitor API request trends to anticipate scaling needs.
- **Error Rate**: Keep an eye on failed requests, as they can signal infrastructure or model issues.
- **Resource Utilization**: Watch CPU/memory usage to ensure your model scales smoothly.

3. Custom Dashboards & Alerts

You can create custom dashboards in **Cloud Monitoring** to visualize key metrics like latency and accuracy. I'd recommend setting alerts for conditions like high latency (>1 second) or rising error rates, ensuring you're notified in real-time. These alerts can be delivered via email, Slack, or SMS.

4. Automated Response

Finally, use **Google Cloud Functions** or **Pub/Sub** to automate responses to incidents, such as scaling the model when necessary or notifying the team of critical issues.

J. Scaling & Monitoring

1. Auto-scaling

Configure Kubernetes Horizontal Pod Autoscaler (HPA) to automatically scale pods based on CPU/memory usage.

2. Monitoring

Use **Google Cloud Monitoring** and **Cloud Logging** to monitor API performance, set up alerts, and track traffic spikes or failures.

K. CI/CD Pipeline

CI/CD pipelines in GCP streamline the development and deployment process by automating code integration, testing, and delivery. Tools like **Cloud Build** and **Cloud Source Repositories** are used to create and manage pipelines. With **Cloud Build**, you can automate building, testing, and deploying your code to environments like GKE or Cloud Run. **Cloud Deploy** further automates the delivery of applications to different environments (dev, staging, production). The key advantage is faster, more reliable releases with automated checks, ensuring code quality and minimizing manual interventions.