Credit Card Default Prediction

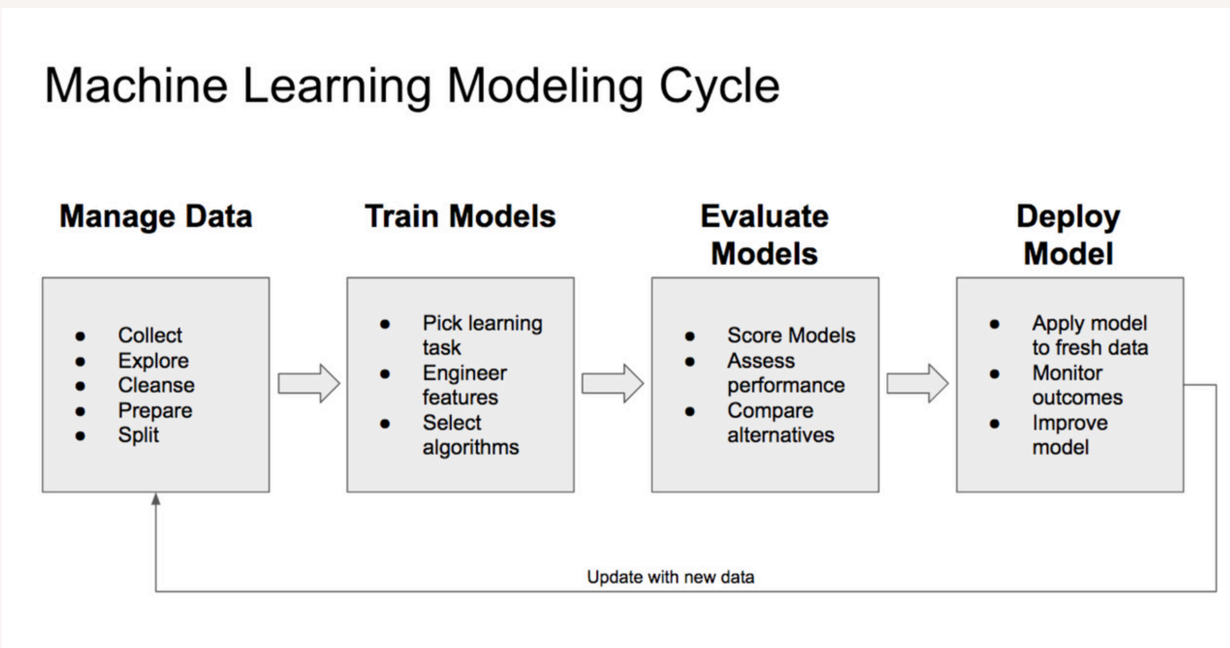
Low-Level Design

Ritik Patel

Low-Level Design Document for Credit Card Default Prediction with Machine Learning

1. Introduction

This document provides a detailed low-level design for the credit card default prediction system. It outlines the specific components, their interactions, and the technologies used.



2. Component Overview

2.1. Data Ingestion

- Source: Transactional data, customer profiles, historical payment information.

- Technologies: `aiofiles`, `dnspython`, `pymongo`
- Functionality:
    - Extract data from various sources.
    - Store raw data in a MongoDB database.
    - Log data ingestion activities.

2.2. Data Processing

- Technologies: `pandas`, `seaborn`, `evidently`, `imbalanced-learn`
- Functionality:
    - Clean data: Handle missing values, and remove duplicates.
    - Feature engineering: Create new features and normalize data.
    - Exploratory Data Analysis (EDA): Visualize data distributions and relationships.

2.3. Model Training

- Technologies: `scikit-learn`, `xgboost`, `dill`
- Functionality:
    - Split data into training and testing sets.
    - Train models using various algorithms (e.g., Logistic Regression, Random Forest, XGBoost).
    - Serialize trained models for deployment using `dill`.

2.4. Model Evaluation

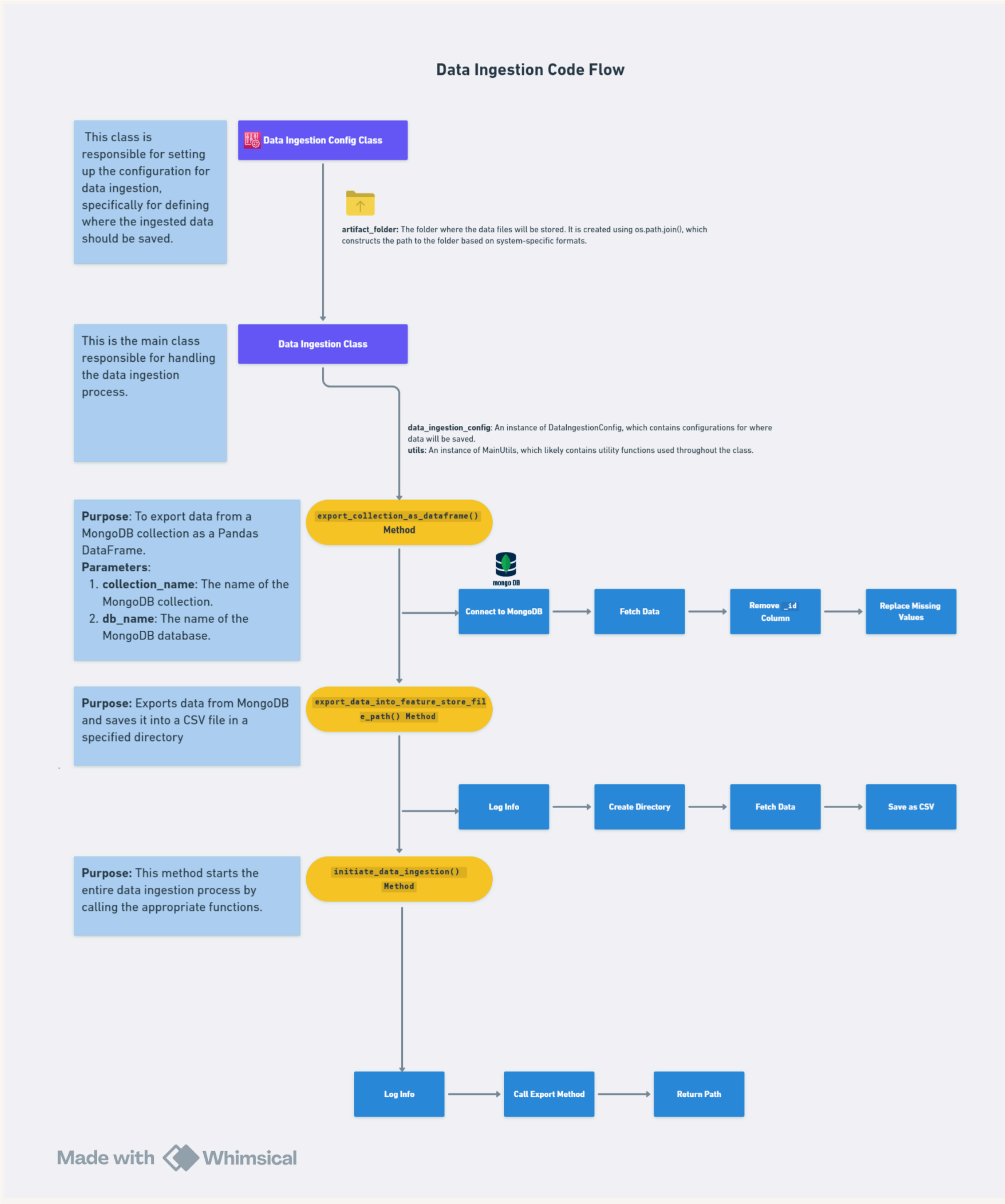
- Technologies: `scikit-learn`, `matplotlib`
- Functionality:
    - Evaluate models using metrics like Accuracy, Precision, Recall, F1 Score, AUC-ROC.
    - Visualize performance metrics and ROC curves.
    - Select the best model based on evaluation results.

2.5. Model Deployment

- Technologies: `Flask`, `Uvicorn`, `FastAPI`, `Docker`
- Functionality:
    - Create RESTful API endpoints for model prediction.
    - Dockerize the application for easy deployment.
    - Deploy the Docker container on a cloud platform (e.g., AWS, Azure).

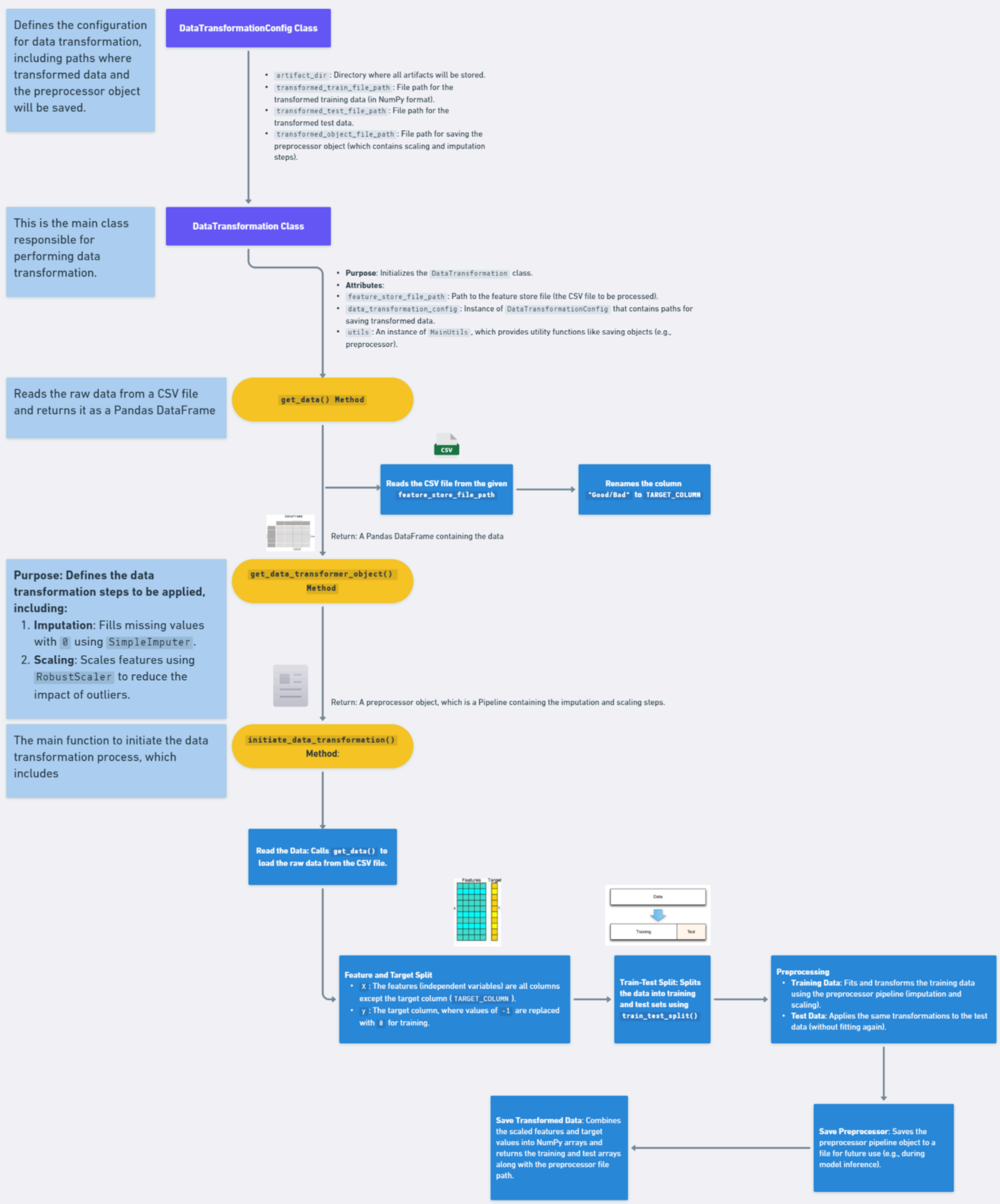
2.6. Monitoring and Maintenance

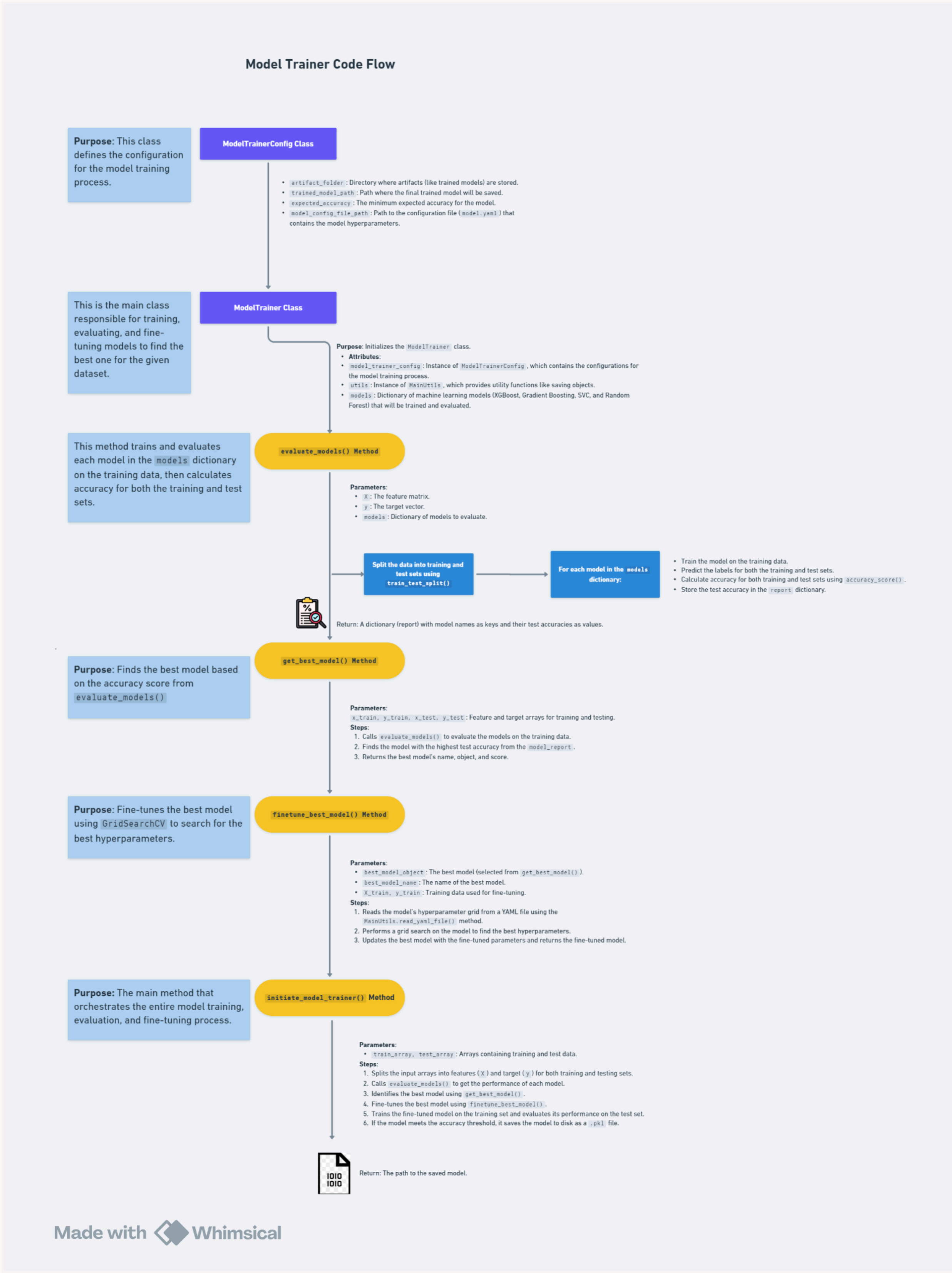
- Technologies: `Prometheus`, `Grafana`, `AWS CloudWatch`
- Functionality:
    - Monitor model performance and detect data drift.
    - Track system health and resource utilization.
    - Set up alerts for anomalies and performance issues.

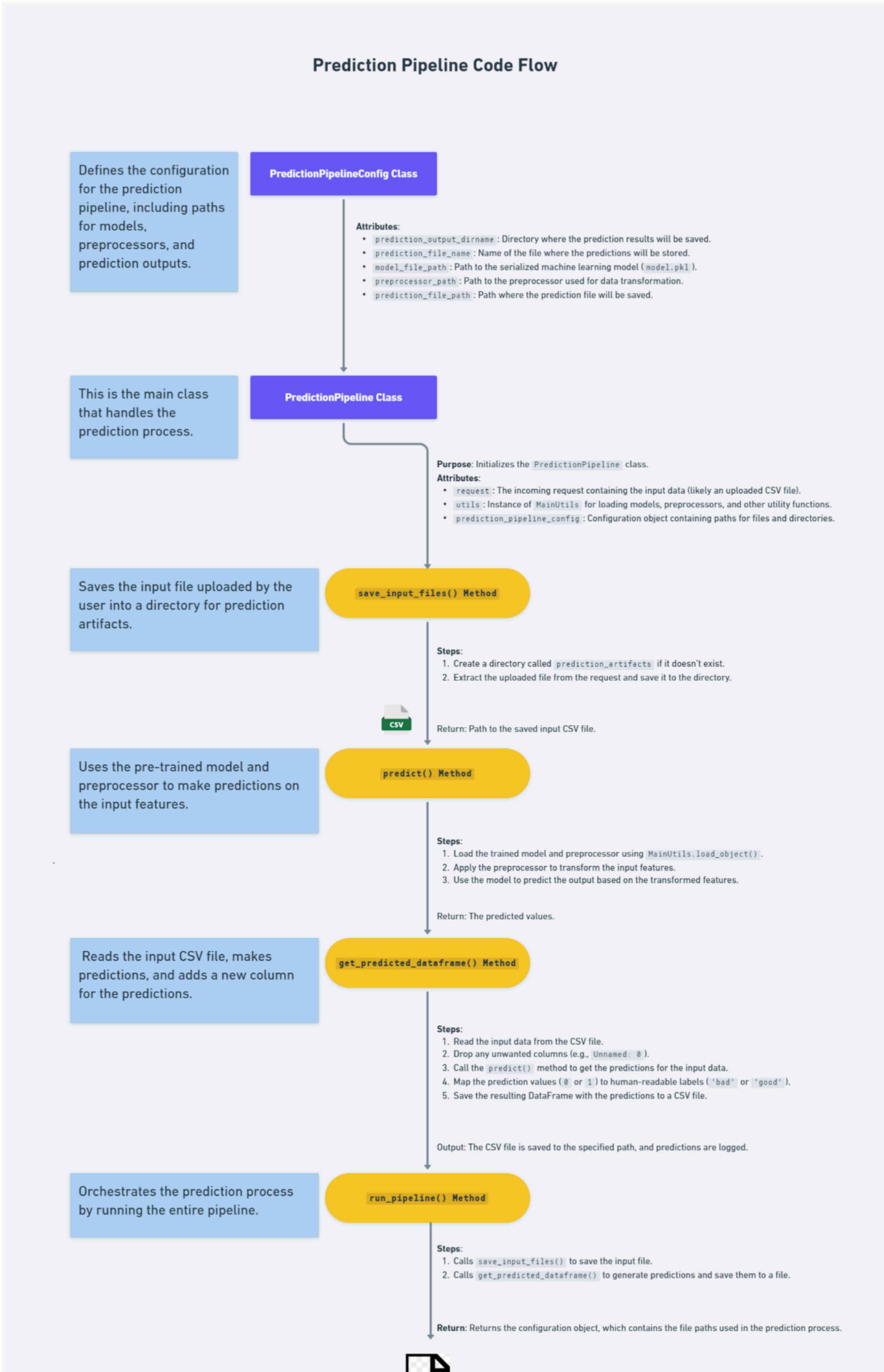


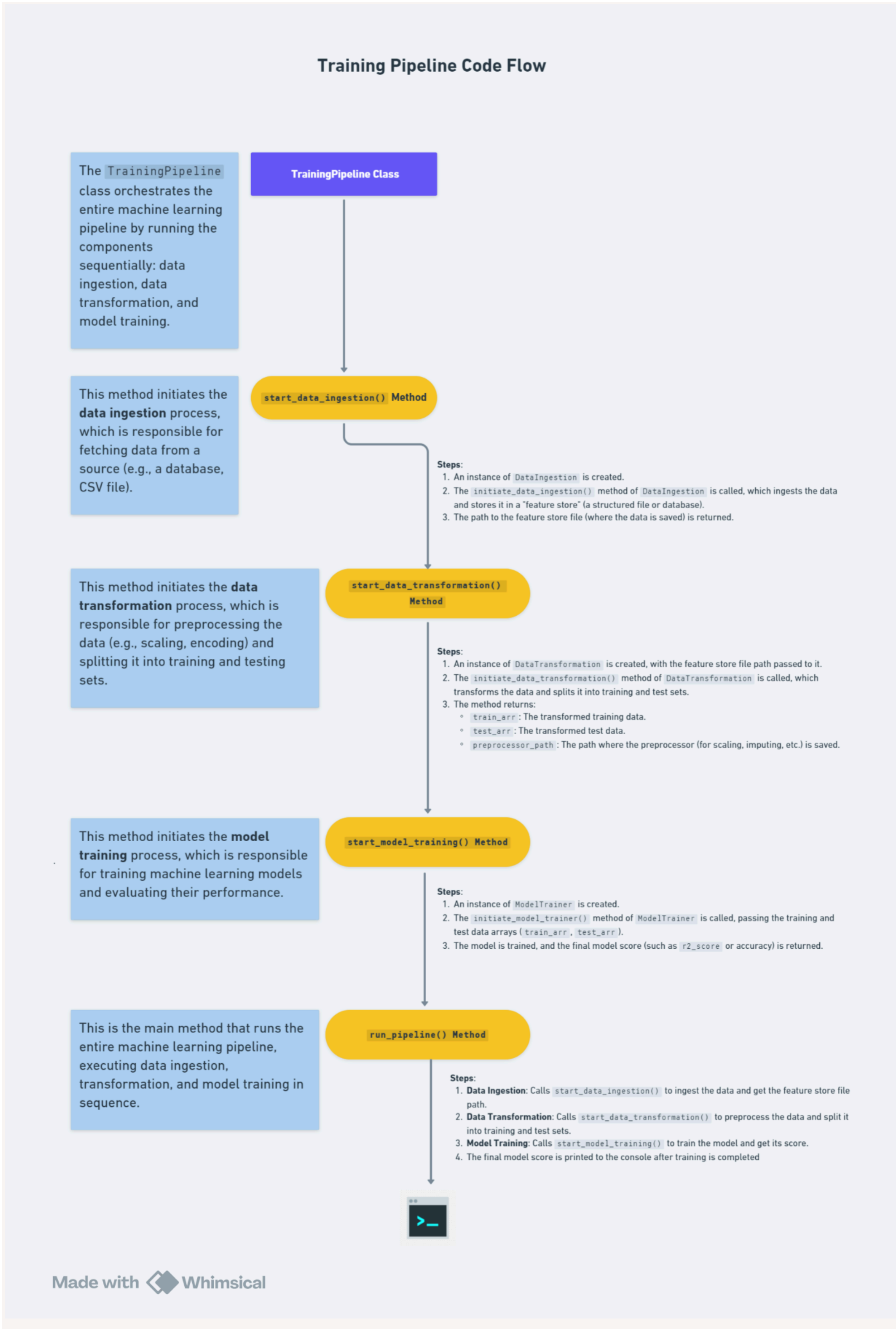
Data Ingestion Code Flow
This class is responsible for setting up the configuration for data ingestion, specifically for defining where the ingested data should be saved.
Data Ingestion Config Class
artifact_folder: The folder where the data files will be stored. It is created using os.path.join(), which constructs the path to the folder based on system-specific formats.
This is the main class responsible for handling the data ingestion process.
Data Ingestion Class
data_ingestion_config: An instance of DataIngestionConfig, which contains configurations for where data will be saved.
utils: An instance of MainUtils, which likely contains utility functions used throughout the class.
Purpose: To export data from a MongoDB collection as a Pandas DataFrame.
Parameters:
1. collection_name: The name of the MongoDB collection.
2. db_name: The name of the MongoDB database.
export_collection_as_dataframe() Method
mongo DB
Connect to MongoDB
Fetch Data
Remove _id Column
Replace Missing Values
Purpose: Exports data from MongoDB and saves it into a CSV file in a specified directory
export_data_into_feature_store_file_path() Method
Log Info
Create Directory
Fetch Data
Save as CSV
Purpose: This method starts the entire data ingestion process by calling the appropriate functions.
initiate_data_ingestion() Method
Log Info
Call Export Method
Return Path
Made with Whimsical

Data Transformation Code Flow




Model Trainer Code Flow
Purpose: This class defines the configuration for the model training process.
ModelTrainerConfig Class
artifact_folder : Directory where artifacts (like trained models) are stored.
trained_model_path : Path where the final trained model will be saved.
expected_accuracy : The minimum expected accuracy for the model.
model_config_file_path : Path to the configuration file (model.yaml) that contains the model hyperparameters.
This is the main class responsible for training, evaluating, and fine-tuning models to find the best one for the given dataset.
ModelTrainer Class
Purpose: Initializes the ModelTrainer class.
Attributes:
model_trainer_config : Instance of ModelTrainerConfig , which contains the configurations for the model training process.
utils : Instance of MainUtils , which provides utility functions like saving objects.
models : Dictionary of machine learning models (XGBoost, Gradient Boosting, SVC, and Random Forest) that will be trained and evaluated.
This method trains and evaluates each model in the models dictionary on the training data, then calculates accuracy for both the training and test sets.
evaluate_models() Method
Parameters:
X : The feature matrix.
y : The target vector.
models : Dictionary of models to evaluate.
Split the data into training and test sets using train_test_split()
For each model in the models dictionary:
Train the model on the training data.
Predict the labels for both the training and test sets.
Calculate accuracy for both training and test sets using accuracy_score() .
Store the test accuracy in the report dictionary.
%
Return: A dictionary (report) with model names as keys and their test accuracies as values.
Purpose: Finds the best model based on the accuracy score from evaluate_models()
get_best_model() Method
Parameters:
x_train, y_train, x_test, y_test : Feature and target arrays for training and testing.
Steps:
1. Calls evaluate_models() to evaluate the models on the training data.
2. Finds the model with the highest test accuracy from the model_report .
3. Returns the best model's name, object, and score.
Purpose: Fine-tunes the best model using GridSearchCV to search for the best hyperparameters.
finetune_best_model() Method
Parameters:
best_model_object : The best model (selected from get_best_model()).
best_model_name : The name of the best model.
X_train, y_train : Training data used for fine-tuning.
Steps:
1. Reads the model's hyperparameter grid from a YAML file using the MainUtils.read_yaml_file() method.
2. Performs a grid search on the model to find the best hyperparameters.
3. Updates the best model with the fine-tuned parameters and returns the fine-tuned model.
Purpose: The main method that orchestrates the entire model training, evaluation, and fine-tuning process.
initiate_model_trainer() Method
Parameters:
train_array, test_array : Arrays containing training and test data.
Steps:
1. Splits the input arrays into features (X) and target (y) for both training and testing sets.
2. Calls evaluate_models() to get the performance of each model.
3. Identifies the best model using get_best_model() .
4. Fine-tunes the best model using finetune_best_model() .
5. Trains the fine-tuned model on the training set and evaluates its performance on the test set.
6. If the model meets the accuracy threshold, it saves the model to disk as a .pkl file.
1010
1010
Return: The path to the saved model.
Made with Whimsical


Prediction Pipeline Code Flow
Defines the configuration for the prediction pipeline, including paths for models, preprocessors, and prediction outputs.
PredictionPipelineConfig Class
Attributes:
• prediction_output_dirname : Directory where the prediction results will be saved.
• prediction_file_name : Name of the file where the predictions will be stored.
• model_file_path : Path to the serialized machine learning model (model.pkl).
• preprocessor_path : Path to the preprocessor used for data transformation.
• prediction_file_path : Path where the prediction file will be saved.
This is the main class that handles the prediction process.
PredictionPipeline Class
Purpose: Initializes the PredictionPipeline class.
Attributes:
• request : The incoming request containing the input data (likely an uploaded CSV file).
• utils : Instance of MainUtils for loading models, preprocessors, and other utility functions.
• prediction_pipeline_config : Configuration object containing paths for files and directories.
Saves the input file uploaded by the user into a directory for prediction artifacts.
save_input_files() Method
Steps:
1. Create a directory called prediction_artifacts if it doesn't exist.
2. Extract the uploaded file from the request and save it to the directory.
CSV
Return: Path to the saved input CSV file.
Uses the pre-trained model and preprocessor to make predictions on the input features.
predict() Method
Steps:
1. Load the trained model and preprocessor using MainUtils.load_object() .
2. Apply the preprocessor to transform the input features.
3. Use the model to predict the output based on the transformed features.
Return: The predicted values.
Reads the input CSV file, makes predictions, and adds a new column for the predictions.
get_predicted_dataframe() Method
Steps:
1. Read the input data from the CSV file.
2. Drop any unwanted columns (e.g., Unnamed: 0).
3. Call the predict() method to get the predictions for the input data.
4. Map the prediction values (0 or 1) to human-readable labels ('bad' or 'good').
5. Save the resulting DataFrame with the predictions to a CSV file.
Output: The CSV file is saved to the specified path, and predictions are logged.
Orchestrates the prediction process by running the entire pipeline.
run_pipeline() Method
Steps:
1. Calls save_input_files() to save the input file.
2. Calls get_predicted_dataframe() to generate predictions and save them to a file.
Return: Returns the configuration object, which contains the file paths used in the prediction process.


Training Pipeline Code Flow
TrainingPipeline Class
The TrainingPipeline class orchestrates the entire machine learning pipeline by running the components sequentially: data ingestion, data transformation, and model training.
start_data_ingestion() Method
This method initiates the **data ingestion** process, which is responsible for fetching data from a source (e.g., a database, CSV file).
Steps:
1. An instance of DataIngestion is created.
2. The initiate_data_ingestion() method of DataIngestion is called, which ingests the data and stores it in a "feature store" (a structured file or database).
3. The path to the feature store file (where the data is saved) is returned.
start_data_transformation() Method
This method initiates the **data transformation** process, which is responsible for preprocessing the data (e.g., scaling, encoding) and splitting it into training and testing sets.
Steps:
1. An instance of DataTransformation is created, with the feature store file path passed to it.
2. The initiate_data_transformation() method of DataTransformation is called, which transforms the data and splits it into training and test sets.
3. The method returns:
- train_arr : The transformed training data.
- test_arr : The transformed test data.
- preprocessor_path : The path where the preprocessor (for scaling, imputing, etc.) is saved.
start_model_training() Method
This method initiates the **model training** process, which is responsible for training machine learning models and evaluating their performance.
Steps:
1. An instance of ModelTrainer is created.
2. The initiate_model_trainer() method of ModelTrainer is called, passing the training and test data arrays (train_arr , test_arr).
3. The model is trained, and the final model score (such as r2_score or accuracy) is returned.
run_pipeline() Method
This is the main method that runs the entire machine learning pipeline, executing data ingestion, transformation, and model training in sequence.
Steps:
1. **Data Ingestion**: Calls start_data_ingestion() to ingest the data and get the feature store file path.
2. **Data Transformation**: Calls start_data_transformation() to preprocess the data and split it into training and test sets.
3. **Model Training**: Calls start_model_training() to train the model and get its score.
4. The final model score is printed to the console after training is completed
Made with Whimsical

3. Detailed Design

3.1. Data Ingestion

3.1.1. Data Extraction

- Files: `data_ingestion.py`
- Process:
  - Connect to data sources using `dnspython` and `pymongo`.
  - Use `aiofiles` for asynchronous file handling.
  - Extract data from MongoDB and other sources.
  - Store raw data locally and log extraction activities.

```
import aiofiles
import pymongo

async def extract_data():
    client = pymongo.MongoClient("mongodb://localhost:27017/")
    db = client["credit_card"]
    collection = db["transactions"]

    async with aiofiles.open('raw_data.csv', mode='w') as f:
        data = collection.find()
        async for record in data:
            await f.write(f"{record}\n")
```

3.2. Data Processing

3.2.1. Data Cleaning and Feature Engineering

- Files: `data_transformation.py`
- Process:
  - Use `pandas` to clean data and handle missing values.
  - Create new features and normalize using `scikit-learn`.
  - Visualize data using `seaborn`.

python

```
import pandas as pd
from sklearn.preprocessing import StandardScaler

def clean_and_engineer_data(data):
    data = data.dropna()
    data['AGE'] = data['AGE'] / data['AGE'].max()
    scaler = StandardScaler()
    data[['LIMIT_BAL', 'BILL_AMT1']] =
scaler.fit_transform(data[['LIMIT_BAL', 'BILL_AMT1']])
    return data
```

3.3. Model Training

3.3.1. Model Training and Serialization

- Files: `model_trainer.py`
- Process:
  - Split data into training and testing sets.
  - Train models using `xgboost` and other algorithms.
  - Serialize models using `dill`.

python

```
import xgboost as xgb
import dill
from sklearn.model_selection import train_test_split

def train_model(data, target):
    X_train, X_test, y_train, y_test = train_test_split(data,
target, test_size=0.2)
    model = xgb.XGBClassifier()
    model.fit(X_train, y_train)
    with open('model.dill', 'wb') as f:
        dill.dump(model, f)
    return model
```

3.4. Model Evaluation

3.4.1. Model Evaluation and Visualization

- Files: `model_evaluation.py`
- Process:
  - Evaluate model using `scikit-learn` metrics.
  - Visualize ROC curves and other metrics using `matplotlib`.

python

```
from sklearn.metrics import roc_auc_score, roc_curve
import matplotlib.pyplot as plt

def evaluate_model(model, X_test, y_test):
    y_pred = model.predict(X_test)
    auc = roc_auc_score(y_test, y_pred)
    fpr, tpr, _ = roc_curve(y_test, y_pred)

    plt.plot(fpr, tpr, label=f'AUC = {auc:.2f}')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve')
    plt.legend()
    plt.show()
```

3.5. Model Deployment

3.5.1. API Development and Deployment

- Files: `app.py`, `Dockerfile`
- Process:
  - Create Flask API for model prediction.
  - Dockerize the application.
  - Deploy using `Uvicorn` and `FastAPI`.

python

```
from flask import Flask, request, jsonify
```

```python
import dill

app = Flask(__name__)

with open('model.dill', 'rb') as f:
    model = dill.load(f)

@app.route('/predict', methods=['POST'])
def predict():
    data = request.json
    prediction = model.predict([data['features']])
    return jsonify({'prediction': prediction.tolist()})

if __name__ == '__main__':
    app.run(debug=True)
```

3.6. Monitoring and Maintenance

3.6.1. Monitoring Setup

- Files: `monitoring_setup.py`
- Process:
    - Configure `Prometheus` to collect metrics.
    - Set up `Grafana` dashboards for visualization.
    - Use `AWS CloudWatch` for system health monitoring.

yaml

```yaml
# prometheus.yml
scrape_configs:
  - job_name: 'flask'
    static_configs:
      - targets: ['localhost:5000']
```

4. Dataset Information

4.1. Dataset Description

This dataset contains information on default payments, demographic factors, credit data, history of payment, and bill statements of credit card clients in Taiwan from April 2005 to September 2005.

4.2. Content

There are 25 variables:

- ID: ID of each client
- LIMIT_BAL: Amount of given credit in NT dollars (includes individual and family/supplementary credit)
- SEX: Gender (1=male, 2=female)
- EDUCATION:
    - 1: Graduate school
    - 2: University
    - 3: High school
    - 4: Others
    - 5: Unknown
    - 6: Unknown
- MARRIAGE: Marital status
    - 1: Married
    - 2: Single
    - 3: Others
- AGE: Age in years
- PAY_0: Repayment status in September, 2005
    - -1: Pay duly
    - 1: Payment delay for one month
    - 2: Payment delay for two months
    - …
    - 8: Payment delay for eight months
    - 9: Payment delay for nine months and above
- PAY_2: Repayment status in August, 2005 (scale same as above)
- PAY_3: Repayment status in July, 2005 (scale same as above)
- PAY_4: Repayment status in June, 2005 (scale same as above)
- PAY_5: Repayment status in May, 2005 (scale same as above)
- PAY_6: Repayment status in April, 2005 (scale same as above)
- BILL_AMT1: Amount of bill statement in September, 2005 (NT dollar)
- BILL_AMT2: Amount of bill statement in August, 2005 (NT dollar)

- BILL_AMT3: Amount of bill statement in July, 2005 (NT dollar)
- BILL_AMT4: Amount of bill statement in June, 2005 (NT dollar)
- BILL_AMT5: Amount of bill statement in May, 2005 (NT dollar)
- BILL_AMT6: Amount of bill statement in April, 2005 (NT dollar)
- PAY_AMT1: Amount of previous payment in September, 2005 (NT dollar)
- PAY_AMT2: Amount of previous payment in August, 2005 (NT dollar)
- PAY_AMT3: Amount of previous payment in July, 2005 (NT dollar)
- PAY_AMT4: Amount of previous payment in June, 2005 (NT dollar)
- PAY_AMT5: Amount of previous payment in May, 2005 (NT dollar)
- PAY_AMT6: Amount of previous payment in April, 2005 (NT dollar)
- default.payment.next.month: Default payment (1=yes, 0=no)