MLE Interview - Take Home Exercise - Aug.2021

August 20, 2021

1 Order Status Decision System

Our team works in the main company's Order Processing system and created a Machine Learning model that can "recommend" what's the next action to apply to customer orders as they are placed.

All customer new orders need to be assessed by this model in order to trigger further steps in order processing.

This model receives an incoming, already generated `order` (order that have already been placed) and predicts a recommended action what could be:

- DELIVER: Release the order and prepare for delivery.
- HOLD, CHECK AVAILABILITY: Hold the order (this is issued to pause order fulfilment preparation while notifying Supply Chain department to check for product availability). The model doesn't know exactly if the product inventory will ne available or not at the moment of order fulfillment, but there is a high probability of issues that will require intervention from Supply Chain systems, i.e., to place purchase orders, estimate a proper delivery, etc date based on manual or automated processes.
- HOLD, CHECK DELIVERY: Hold the order until delivery can be confirmed. Same as before, the model predicts a high chance that delivery is not possible and that requires intervention from Logistics deparment or systems to confirm, cancel or reschedule the order.
- HOLD, CHECK PAYMENT: Hold the order (this is issued to pause order fulfillment, and notify Payments and Fraud department). The model issues this prediction, when the order is likely associated with past Fraud or Payments issues. This flag indicates that the order will be sent for additional checking (either manual or automated).
- DECLINE: Decline and cancel the order.

The objective of the system is to act as a first step in a workflow for order fulfillment. Based on the recommendations from this system, other processes can be triggered. These predictions are particularly useful to trigger further verification and decision steps in Supply Chain, Logistics and Payments department, as putting the orders ON_HOLD until properly fullfilment can be confirmed by these departments. In the other hand it allows safe orders to be prepared as soon as possible.

> Notice that this checks can also be done using different approaches (i.e. many models, business logic, etc). For the simplicity of the interview case, we've decided to use a single model to make an initial assesment on orders but in real scenarios this can be usually performed by different systems.

The model was trained using historical order information, where the following features were calculated:

```
- order_hour_of_day: int, A number between 0-23 with the hour of the day of
    the order timestamp

- inventory: int, Sku inventory at the time of the order is placed

- payment_status: str, Categorial feature: Payment status of the order
    returned by payment method provider

- zip_code_available: bool, whether the delivery to the zip_code is possible
```

For simplicity we can assume each order contains a single product item

For simplicity we assume the model will return/predict only ONE class.

The features are calculated from orders and inventory information available. The following are examples of the data used in the inital POC version:

And we have as an input a historical list of order information in a csv file:

orders.csv entry examples

```
order_id,customer_id,timestamp,sku_code,zip_code,order_fulfilled
ORDER1,CUST1,2021-05-01T13:21:59.930321+00:00,111,SKU1,TRUE
ORDER2,CUST2,2021-05-02T13:21:59.930321+00:00,222,SKU1,FALSE
ORDER3,CUST3,2021-05-03T13:21:59.930321+00:00,333,SKU2,TRUE
ORDER4,CUST1,2021-05-04T13:21:59.930321+00:00,111,SKU2,FALSE
```

And payment information: payments.csv entry examples

```
order_id,customer_id,timestamp,payment_method,payment_status
ORDER1,CUST1,2021-05-01T13:21:59.930321+00:00,CREDIT_CARD,OK
ORDER2,CUST2,2021-05-02T13:21:59.930321+00:00,CREDIT_CARD,FAILED
ORDER3,CUST3,2021-05-03T13:21:59.930321+00:00,CREDIT_CARD,OK
ORDER4,CUST1,2021-05-04T13:21:59.930321+00:00,ON_DELIVERY,VERIFY_ADDRESS
```

List of availabel delivery zip_codes zip_codes.csv entry examples

```
zip_code,available_from
111,2021-05-01T13:21:59.930321+00:00
222,2021-06-01T13:21:59.930321+00:00
333,2021-05-01T13:21:59.930321+00:00
```

And finally, historical inventory for each SKU: inventory.csv entry examples

```
sku_code,timestamp,inventory
SKU1,2021-05-01T00:00:00+00:00,2
SKU2,CUST3,2021-05-03T00:00:00+00:00,0
```

The model shows very good performance on historical data, specially to prevent cases where the order is not likely to be succesfully fulfilled.

2 Take home exercise

Please read the document `Case Description` before Notice that you need data files (csv) that can be used in this exercise. Check if you have received the data files.

Model Suppose that the model can be mocked with a python function returning a prediction based on certain arbitrary logic:

`model.py`:

```
"""
Module containing a model implementation(s) and `predict` function that
can be invoked with a set of Features to get a Recommendation

Example:
``
from model import Features, model_v1, predict

features = Features(...)
recommendation = predict(model_v1, features)
``
"""
from enum import Enum
import random
from typing import Callable, NamedTuple

__all__ = [
    "Features",
    "Recommendation",
    "Model",
    "model_v1",
    "predict"
]


# Here we choose NamedTuple but you can adapt this if you prefer to use
# a dataclass or libraries like Pydantic (https://pydantic-docs.helpmanual.io/).
class Features(NamedTuple):
    """
    Container for model feature values

    :field: order_hour_of_day: int, A number between 0-23 with the hour of the day of
        the order timestamp
    :field: inventory: int, Sku inventory at the time of the order is placed
    :field: payment_status: str, Categorial feature: Payment status of the order
        returned by payment method provider
    :field: zip_code_available: bool, whether the delivery to the zip_code is possible
    """
    order_hour_of_day: int
```

```python
    inventory: int
    payment_status: str
    zip_code_available: bool


class Recommendation(Enum):
    """
    Possible predicted classes
    """
    DELIVER = 'Deliver'
    HOLD_CHECK_AVAILABILITY = 'HoldCheckAvailability'
    HOLD_CHECK_DELIVERY = 'HoldCheckDelivery'
    HOLD_CHECK_PAYMENT = 'HoldCheckPayment'
    DECLINE = 'Decline'


# Signature for the model function. Not required, but feel free to adapt it or extend it
# if you need it for the next steps
Model = Callable[[Features], Recommendation]


def model_v1(features: Features) -> Recommendation:
    """
    This function is a very simple unrealistic example of what a
    decision tree could predict based on the example features above.
    We can assume that it behaves like a real-world ML model
    for the purposes of this exercise.
    """
    if features.inventory <= 0:
        return Recommendation.HOLD_CHECK_AVAILABILITY
    if not features.zip_code_available:
        return Recommendation.HOLD_CHECK_DELIVERY
    if features.payment_status != "OK":
        return Recommendation.HOLD_CHECK_PAYMENT
    if features.order_hour_of_day < 6:
        return Recommendation.DECLINE
    return Recommendation.DELIVER


def predict(model: Model, features: Features) -> Recommendation:
    """
    Invokes model with the given features to return a recommendation.
    Notice that model is a Callable that receives Features and returns Recommendation.
    """
    return model(features)
```

Here you can see some example predictions in the unit tests for the model function, `test_model_v1.py`:

```python
import pytest

from model import Features, Recommendation, model_v1 as model


def default_features_happy(
    *,
    order_hour_of_day: int = 10,
    inventory: int = 100,
    payment_status: str = "OK",
    zip_code_available: bool = True
) -> Features:
    """
    Create Features with default values for testing purposes.
    Optionally user can override specific features values to create test cases.
    Default values will match the happy case where recommendation should be "DELIVER"
    """
    return Features(
        order_hour_of_day,
        inventory,
        payment_status,
        zip_code_available
    )


def test_check_availability_zero_or_negative_inventory():
    assert model(
        default_features_happy(inventory=0)
    ) == Recommendation.HOLD_CHECK_AVAILABILITY

    assert model(
        default_features_happy(inventory=-1)
    ) == Recommendation.HOLD_CHECK_AVAILABILITY


def test_check_availability_positive_inventory():
    assert model(
        default_features_happy(inventory=1)
    ) == Recommendation.DELIVER


def test_check_delivery_zip_code_not_available():
    assert model(
        default_features_happy(zip_code_available=False)
    ) == Recommendation.HOLD_CHECK_DELIVERY


def test_check_delivery_zip_code_available():
    assert model(
        default_features_happy(zip_code_available=True)
    ) == Recommendation.DELIVER


def test_check_payment_status_not_ok():
    assert model(
        default_features_happy(payment_status="FAILED")
    ) == Recommendation.HOLD_CHECK_PAYMENT
    assert model(
        default_features_happy(payment_status="VERIFY_ADDRESS")
    ) == Recommendation.HOLD_CHECK_PAYMENT
    assert model(
        default_features_happy(payment_status="VERIFY_BANK_DETAILS")
    ) == Recommendation.HOLD_CHECK_PAYMENT


def test_check_payment_status_ok():
    assert model(
        default_features_happy(payment_status="OK")
    ) == Recommendation.DELIVER


def test_decline_bad_hour_of_day():
    assert model(
        default_features_happy(order_hour_of_day=0)
    ) == Recommendation.DECLINE


def test_not_decline_good_hour_of_day():
    assert model(
        default_features_happy(order_hour_of_day=7)
    ) == Recommendation.DELIVER


if __name__ == "__main__":
    pytest.main()
```

Available data We have as an input a historical list of order information in a csv file: orders.csv And payment information: payments.csv List of availabel delivery zip_codes: zip_codes.csv And finally, historical inventory for each SKU: inventory.csv

2.0.1 What you need to do:

1) Write a Python program to read the source files, and calculate the features needed at the timestamp of each order.

The features associated with each order from `orders.csv` should be written to a data file (i.e. `csv` or `jsonlines` http://jsonlines.org)

You will need these feature values to be used in the next step by a service

[Required] add unit tests for feature calculation

2) Write an HTTP server using a Python framework from the list below* to serve predictions on real-time for incoming orders using model_v1. Endpoint: POST /decision/v1 Payload example:

```
{
    "order_id": "ORDER1",
    "customer_id": "CUST1",
    "timestamp": "",
    "sku_code": "SKU1",
    "zip_code": "111"
}
```

Response example:

```
{
    "order_id: "ORDER1",
    "recommendation": "DELIVER"
}
```

This endpoint must invoke the model you created in step 2 and use the features calculated in step 1. You can assume that all additional information necessary to assess the order (payment information, delivery zip codes and sku inventories) can be loaded at the startup of the service.

(*) Allowed Python frameworks to use: FastAPI (https://fastapi.tiangolo.com/), aiohttp (https://docs.aiohttp.org/en/stable/) or Flask (https://flask.palletsprojects.com/en/2.0.x/)

2.1 Deliverable

You should return a zip file that we can run and it contains:

1) The python script that process the given input csv data files and creates `features.csv` file
2) The service with the prediction endpoint
3) A Python script to invoke the service for each entry in the `orders.csv` file, and outputs the results into a file called `results.json` using jsonlines format (https://jsonlines.org) where each line is a json document.

Example output:

```
{"order_id: "ORDER1", "recommendation": "DELIVER"}
{"order_id: "ORDER2", "recommendation": "DECLINE"}
{"order_id: "ORDER3", "recommendation": "ON_HOLD_CHECK_AVAILABILITY"}
```

4) We expect to run the solution using `docker-compose` like this:

```
docker compose up
```

So you need to write a `docker-compose.yml` file that:

1) Read the source csv files and calculate the features (can be done at start of the service)
2) Start the service with the mentioned HTTP endpoint
3) Run the predictions invoking the service for each order
4) Outputs the results to `results.json` file

Input data (provided csv files) should be located in `./data/` folder of your project root.

Results should be output to `./results/` folder of your project root.

You can document your solution in a `README.md` file. If you provide additional functionality to what's required, please mention it there.

Once again, thanks for your time and please reach out in case of questions or suggestions!

We hope this exercise gives you a starting point for a Coding case, and a System Design case. Both interviews will be based on the same problem so this will give you time to think and prepare.

The whole exercise can take about 3-4 hours. If you think you don't have time to solve all the required items above, please contact us with a proposal telling us which parts you would like to tackle first, a first version you can deliver, or an alternative solution.

Thanks a lot!