Product Specification Document

MSDS 434

Winter 2020

By
Christine Byron

Citi Bike Customers: Real-Time Recommendations

Overview

Using the agile methodology, our project team developed a real-time recommendation application to increase the user interaction and enrich shopping potential for our Customer user type. In this 10 week MVP, we utilized demographic data from customers, as well as information from previous purchases and user behavior to predict which pass our daily rider should select based on whether their desired ride was estimated to be 30 minutes. Rider gender, day of the week, start station, and end station were used as indicators.

Source Code Stored in Github

GitHub was chosen to host our project's source code. It provided our team the ability to successfully track of the constant revisions over our ten-week development lifecycle.Final code and project documentation has been uploaded into our Project Repository:

https://github.com/christinebyron/MSDS434_Final-Project-MVP/tree/master

Continuous Deployment from CircleCI

Our approach of Continuous Integration encouraged our project team to frequently introduce changes into our codebase. With this practice, our developers followed the process of integrating their code modifications into a shared repository several times a day. Each commit triggered a pipeline that would build and test how new changes performed overall and shared the results. This approach supports version control, efficient feedback, and faster iterations.

To best accomplish this, CircleCI was selected and was connect our GitHub repository. The result of introduced changes can be viewed through the "Status Badge" maintained on our repository.

Data stored in GCP

Our product utilizes the Citi Bike daily ridership and membership data hosted in Google BigQuery. This dataset consists of the trips collected since the ride share program launched in September 2013, and is updated daily. It is important to note that the data has been processed by Citi Bike to remove all trips that are taken by staff to service and inspect the system, as well as any trips below 60 seconds in length, which are considered false starts. It was used to

The data includes:

- Trip Duration (seconds)
- Start Time and Date
- Stop Time and Date
- Start Station Name
- End Station Name
- Station ID
- Station Lat/Long

- Bike ID
- User Type (Customer = 24-hour pass or 3-day pass user; Subscriber = Annual Member)
- Gender (Zero = unknown; 1 = male; 2 = female)
- Year of Birth

Additional fields calculated for exploratory analysis purposes included:

- Distance
- Season
- Day of Week
- Hour
- Minutes
- Age

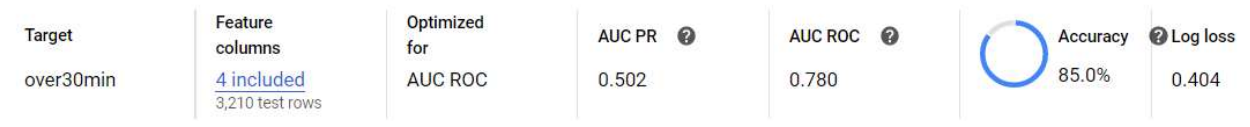
ML Predictions

After a thorough exploratory analysis of the NYC Citibike Customer data, the team chose to train our model, test it, and evaluate its performance using a combination of BigQueryML and AutoMLTables capabilities. This approach encouraged the team to review various model results, adjust the training dataset as needed, and train new models by using our findings. A large assortment of linear regression and binary classification models were evaluated in this process.

In the end, our team selected the following features and target variable:

- Target Variable: 'over30min'
- Features: 'start_station_id', 'end_station_id', 'gender', 'dayofweek'

The AutoML Table evaluation indicated:



Target	Feature columns	Optimized for	AUC PR	AUC ROC	Accuracy	Log loss
over30min	4 included 3,210 test rows	AUC ROC	0.502	0.780	85.0%	0.404

Monitoring

Our team has leveraged Google Cloud's Operations suite (formerly Stackdriver) for monitoring, logging, and diagnostics purposes. This functionality allowed our team, and will offer future product owners, visibility into our application and its integrated infrastructure.

Using the Stackdriver monitoring capabilities, our team:

- Enabled Cloud Debugger so that we could inspect snapshots of the deployed application
- Reviewed application logs.
- Setup metrics and alerts

Lastly, Cloud Billing is used to efficiently manage and report our product costs and usages. We have enabled Cloud Billing data to be exported to a defined BigQuery dataset. This will allow leaders to see all data captured for our defined Billing Account. Data exports have been set up at a regular interval since the day the application was deployed. Additionally, a Cloud Billing report has been created in Data Studio so that the spend over time can be visualized. As ownership of the report is transferred, leaders can choose to slice and dice the overall Google Cloud bill as needed.

Google App Engine serves out HTTP requests via REST API with a JSON payload

Our product uses the simple Logistic Regression model using the NYC Citibike dataset. The model has been defined to predict his model whether a ride will should purchase a single ride or another daily pass given provided information about start station, end station, gender, and day of the week.

After having selected the best model, the scrum team created and pickled our model, and created a Flask application:

```
import pickle
from google.cloud import storage
import pandas as pd
from flask import Flask, jsonify, request
import sklearn

# app
app = Flask(__name__)

# routes
@app.route('/', methods=['POST'])

def predict():
    # get data
    data = request.get_json(force=True)

    # convert data into dataframe
    data.update((x, [y]) for x, y in data.items())
    data_df = pd.DataFrame.from_dict(data)

    # set up access to the GCS bucket
    bucket_name = "citibikenyc_app"
    storage_client = storage.Client()
    bucket = storage_client.get_bucket(bucket_name)

    # download and load the model
    blob = bucket.blob("sMSDS434_Final-Project-MVP")
    blob.download_to_filename("/tmp/lmodel.pkl")
    model = pk.load(open("/tmp/model.pkl", 'rb'))

    # predictions
    result = model.predict(data_df)

    # send back to browser
    output = {'results': int(result[0])}

    # return data
    return jsonify(results=output)

if __name__ == '__main__':
    app.run(port = 5000, debug=True)
```

The structure of the code follows:

1. Load pickled model
2. Name flask app
3. Create a route that receives JSON inputs, uses the trained model to make a prediction, and returns that prediction in a JSON format

* Model files have been programmatically set to store to a defined path within our product's Google Cloud Storage Bucket:

* All files have been committed the GitHub repository.

Deployed Application

After developing application using our environment strategy, we wanted to enabling sharing it with our customers. For this step, Heroku was chosen as the platform to host the API.

Once we finalized building our prediction model on the historic NYC Citibike customer data and created a web application with Flask, we deploy it using Heroku. In order for us to successfully deploy our application to Heroku, we had to add a Procfile to that application:

```
web: gunicorn app:app
```

From our code:

- web is used by Heroku to start a web server for the application
- app:app specifies the module and application name. In our application we have the app module and our flask application is also called app.

Finally, our application was deployed to Heroku by connecting it to our GitHub production branch.