Ben Cimini, Blair Bowen, Caleb Hendrix

ciminibb@mail.uc.edu, bowenbv@mail.uc.edu, hendricw@mail.uc.edu

Abstract

Our project applies cloud computing, data engineering, and data science skills to a retail dataset from 84.51°. We leverage tools and services from Azure to derive insights that could, feasibly, enhance the retail experience for customers.

GROUP 39 Final Project

Data Science and Analytics Using Azure Cloud Computing Technologies

# Write-Up on ML Models

**Linear Regression**

Linear Regression is among the simplest ML models. It assumes a linear relationship between input and output variables, classically represented by . In the context of regression modeling, and are sometimes called and respectively. The goal of such modeling is to find the best values of and , by minimization of error in the best-fit line.

**Random Forest**

Random Forest is popular and powerful. It is based on statistical bagging, in which repeated samples are taken from a dataset to yield many decision trees. When a prediction is needed, the result of each decision tree is averaged for a strong estimate. Random Forest introduces randomness into these decision trees, meaning they’re more different from one another than they would be otherwise.

**Gradient Boosting**

Gradient Boosting is powerful but dense. It also uses decision trees, but they depend on one another. Unlike Random Forest, each tree in Gradient Boosting is based on the last in an attempt to fix errors. Ultimately, it aims to create a “strong learner” from the “weak learner” trees. The number of weak learners depends on the dataset. They will keep being iterated upon until a) the training set is predicted perfectly or b) some maximum number of decision trees have been exhausted. The final prediction is a weighted average over the trees.

**Conclusion**

Any of the above would be serviceable for Basket Analysis, at least. So, a choice between them becomes one of explainability versus predictive accuracy. In short, given the nature of this project and dataset, we believe Random Forest is our best, most balanced choice.

# Web Server Setup

Created a resource group for this project, <finalproject-group39-rg-1>, and gave the “Owner” role to all members. A screenshot of a computer

Description automatically generated

Created a Linux VM for the web server, <finalproject-group39-vm-webserver>, using defaults from the “Free Services” page. A screenshot of a computer

Description automatically generated

Added inbound security rules for HTTP and HTTPS, making the VM internet accessible. A screenshot of a computer

Description automatically generated

Updated the VM’s package list before necessary installations. A screenshot of a computer

Description automatically generated

Installed Python 3 and pip on the VM. A screenshot of a computer screen

Description automatically generated

Installed Flask on the VM. A screenshot of a computer screen

Description automatically generated

Created a Flask app on the VM to expose our data to the internet. At this point, the app only displayed a message. A screenshot of a computer

Description automatically generated

This is the message the app displayed at the time of writing. A screenshot of a computer

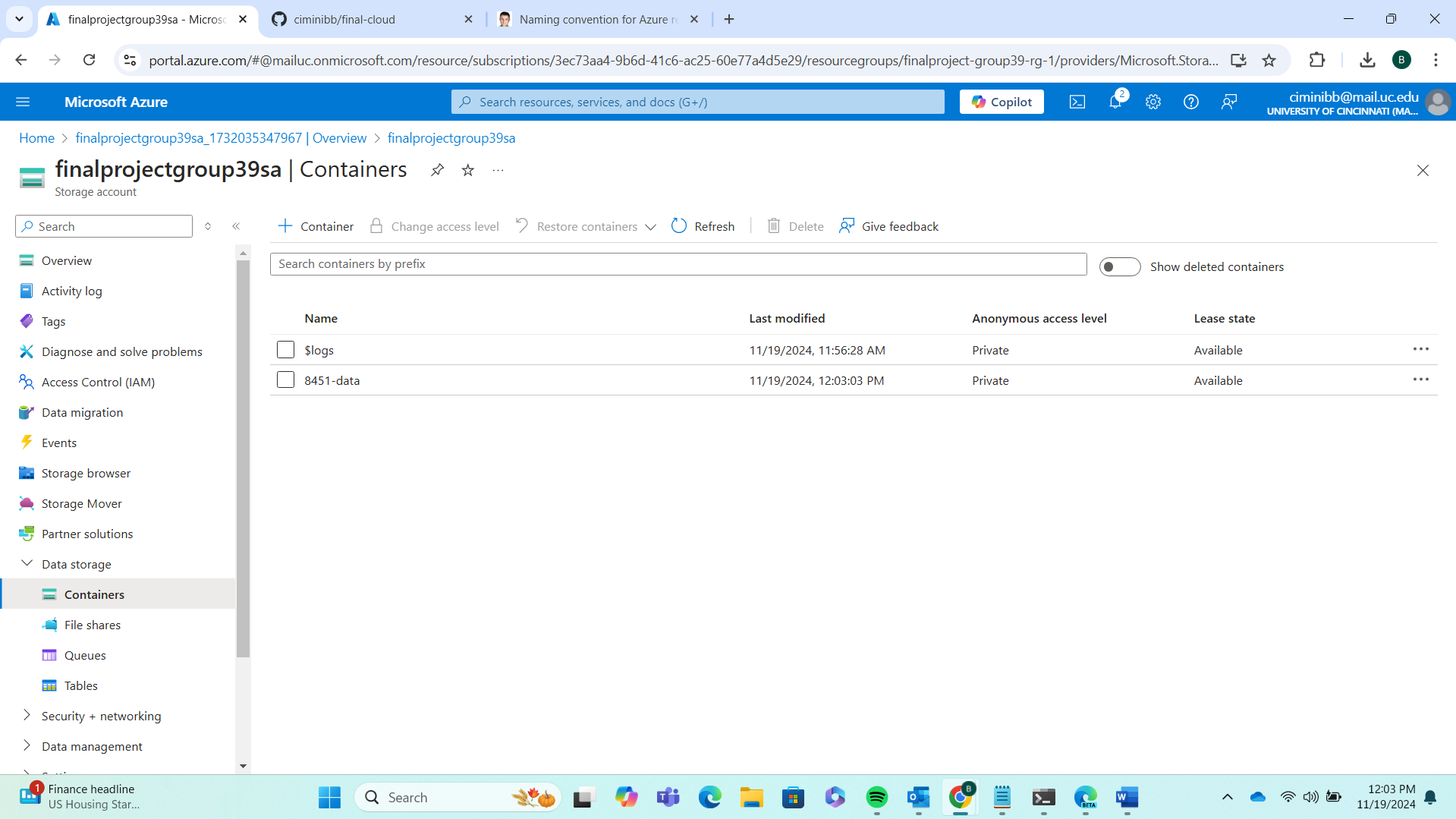
Description automatically generated

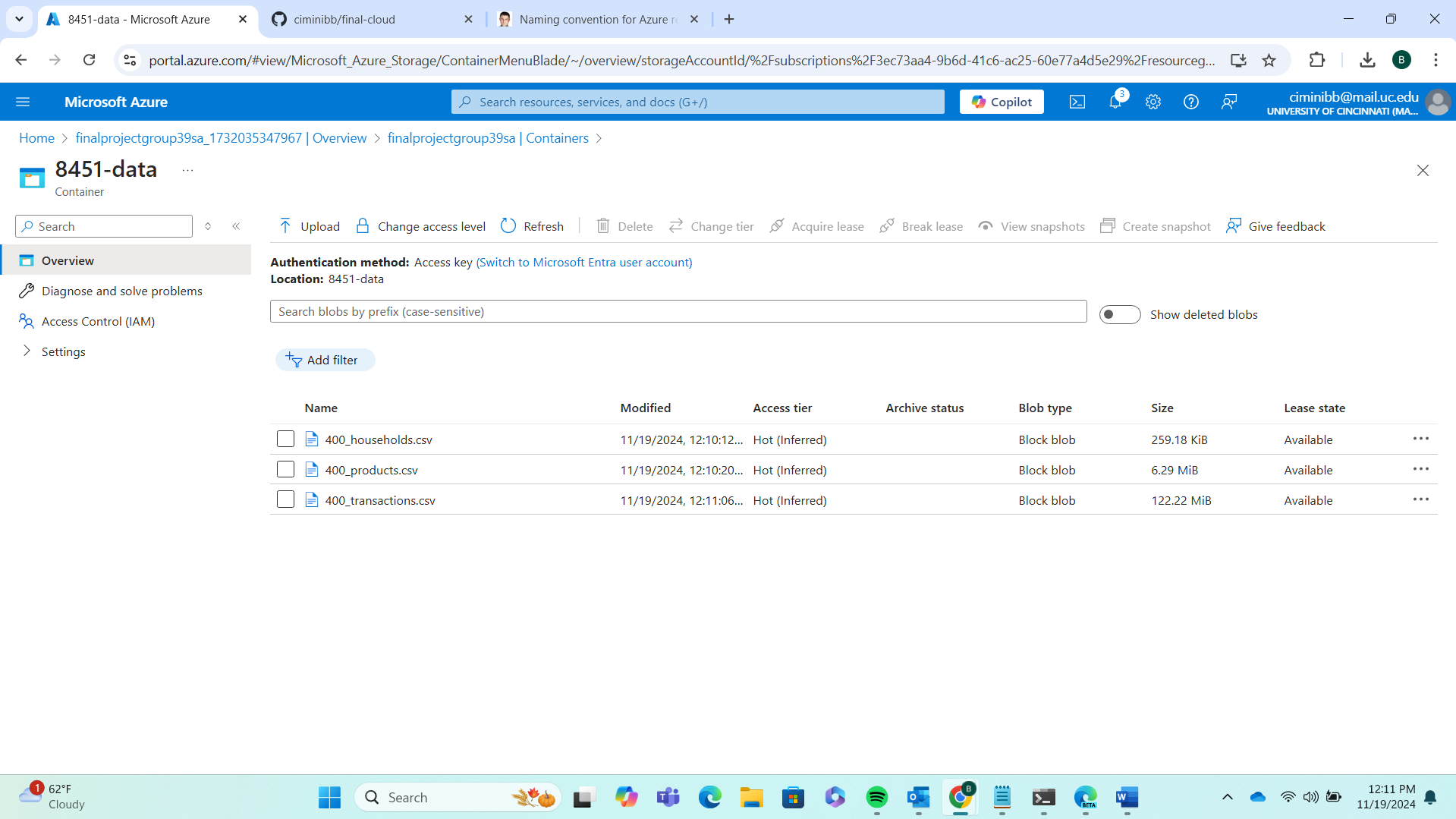
Added to the Flask app so it has a sign-in form on the homepage that, upon submission, redirects the user to a menu. You can see the relevant screenshots below (note that the menu shown is an early version). For the code and file structure involved, visit our repository. A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

# Datastore and Data Loading

Created a storage account and container, <finalprojectgroup39sa> and <8451-data> respectively, since Azure SQL Database does not allow direct uploads, meaning we need blob storage. 

Uploaded 84.51° retail data to the blob storage container. 

Created a SQL database and associated server, <finalproject-group39-sql-db> and <finalproject-group39-sql-server> respectively, to which the 84.51° retail data will be fed. A computer screen shot of a computer

Description automatically generated

Added a firewall rule to the database that lets our VM, <finalproject-group39-vm-webserver>, access the database. Thus allowing its data to be pulled into the Flask app. A screenshot of a computer

Description automatically generated

Created all necessary database tables – Households, Products, and Transactions – in Azure Data Studio. The schemas were defined per the SQL queries below to mimic those of the CSV files. A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Cleaned the CSVs of problematic extra whitespace then created a data factory, linked services for blob storage and database, and a dataset for each CSV file and database table. These were then incorporated into an end-to-end data pipeline that copied the CSVs to our SQL database in one swoop. This was the longest step of the project so far and took lots of debugging. The whitespace issue, for example, was difficult to discover.A computer screen shot of a computer

Description automatically generated

Wrote a query for the sample data pull (Household 10) whose results will be displayed in the Flask app. A screenshot of a computer

Description automatically generated

Integrated the query above into our Flask app, creating a display page for the data pull. Again, this was a rather long process. Rather than explain it here, I’ll, once again, leave you to review the code in our GitHub repository. The resulting display page looked like this. A screenshot of a computer

Description automatically generated

# Interactive Web Page

Here is the base interactive webpage where a user can enter the number of a household and then order the results by the selected item:

A screenshot of a computer

Description automatically generated

When a user clicks the search button then the results will be shown from the database:

A screenshot of a computer

Description automatically generated

# Data Loading Web App

Here is the image of the data loading web app, note that the user must use the file names that are listed for the upload to be successful:

A screenshot of a computer

Description automatically generated

# Web Page with Dashboard

Here is the dashboard created from different data pulls of the database. We explore the effects of different demographic situations on spending behavior:

A screenshot of a graph

Description automatically generated

Here are the SQL queries that were used to create the tables seen above:

Chart 1 –

A screenshot of a computer program

Description automatically generated

Chart 2 –

A screenshot of a computer

Description automatically generated

Chart 3 –

A screen shot of a computer program

Description automatically generated

Chart 4 –

A screenshot of a computer program

Description automatically generated

Chart 5 –

A screenshot of a computer program

Description automatically generated

Chart 6 –

A screenshot of a computer program

Description automatically generated

# Data Loading Web App

# Web Page with Dashboard

# ML Model Application

<https://github.com/ciminibb/final-cloud/blob/main/ml_app/ml_application.ipynb>

This notebook evaluates the performance of machine learning models (Logistic Regression, Random Forest, Gradient Boosting) for predicting high-spending baskets (Spend > $100). It explores feature importance, predicted probabilities, and cross-selling opportunities to drive actionable insights.

A graph of a bar chart

Description automatically generated with medium confidence

# Churn Prediction

<https://github.com/ciminibb/final-cloud/blob/main/ml_app/churn_analysis.ipynb>

This notebook focuses on analyzing customer churn using a dataset containing information about transactions, products, and households. It uses features like Recency, Frequency, Monetary Value, and Product Diversity to predict churn.

A screenshot of a graph

Description automatically generated