MLOPS Assignment 1

Student name: Ahmed Boray
Student ID: 2309015841

Date: 11/18/2025

Q1 part

A. Implementing Hashed Feature

1. Code: Hashed Feature Implementation

The Hashed Feature, or Feature Hashing, design pattern maps the categorical string feature to a numerical index (bucket) using a hash function and the modulo operator.

We'll use a standard hash function (like Python's built-in hash()) and the modulo operator (%) to map the large set of unique IATA codes to a fixed number of buckets (100 in this case).

```
Python
import hashlib

def hashed_feature_mapping(iata_code: str, num_buckets: int = 100) -> int:
    """
    Maps a categorical string (IATA code) to a bucket index using Feature Hashing.

    Args:
        iata_code: The string value of the categorical feature (e.g., "JFK").
        num_buckets: The fixed number of buckets (dimension of the output vector).

    Returns:
        The bucket index (an integer between 0 and num_buckets - 1).
    """
    # 1. Encode the string to bytes (required for the hash function)
    encoded_string = iata_code.encode('utf-8')

    # 2. Apply a cryptographic hash function (SHA-256 for stability/security,
    #    or Python's built-in hash for simplicity/speed)
    #    Here, we use Python's built-in hash for simplicity.
    hash_value = hash(encoded_string)

    # 3. Apply the modulo operator to map the hash value to a bucket index
    #    This constrains the output to the range [0, num_buckets - 1].
    bucket_index = hash_value % num_buckets

    # We ensure the result is non-negative by taking the absolute value,
    # though Python's hash() generally handles this well with the % operator.
    return abs(bucket_index)
```

```
# --- Example Usage ---
# Use 100 buckets as specified
NUM_BUCKETS = 100

# Known Airports
airport_1 = "JFK"  # John F. Kennedy International Airport
airport_2 = "LAX"  # Los Angeles International Airport
airport_3 = "SFO"  # San Francisco International Airport

# Unseen (Cold Start) Airport
airport_new = "IST"  # Istanbul Airport (might not be in the original training data)

print(f"IATA Code '{airport_1}' maps to Bucket: {hashed_feature_mapping(airport_1,
NUM_BUCKETS)}")
print(f"IATA Code '{airport_2}' maps to Bucket: {hashed_feature_mapping(airport_2,
NUM_BUCKETS)}")
print(f"IATA Code '{airport_3}' maps to Bucket: {hashed_feature_mapping(airport_3,
NUM_BUCKETS)}")
print(f"IATA Code '{airport_new}' (Cold Start) maps to Bucket:
{hashed_feature_mapping(airport_new, NUM_BUCKETS)}")
```

---

2. Analysis: Addressing Challenges and Trade-offs

Addressing Challenges

Feature Hashing effectively tackles the challenges of the "Departure Airport" feature by:

- High Cardinality: Instead of creating a massive One-Hot Encoding (OHE) vector where the dimension is equal to the number of unique airports (potentially thousands), Hashing fixes the feature dimension to a manageable size (e.g., 100 buckets). This drastically reduces the memory footprint and the size of the model weights, making the model more compact and faster to train.
- Incomplete Vocabulary / Cold Start: This is the most significant benefit. Since the mapping relies purely on a hash function and the modulo operator, there is no need for a pre-defined vocabulary (a dictionary mapping airport $\rightarrow$ index).
    - Any new, unseen IATA code (a cold-start airport like "IST") is immediately processed by the function and mapped to a valid bucket index. It guarantees a non-zero representation in the feature space, allowing the model to make predictions without breaking, which is impossible with standard OHE.

Key Trade-off: Bucket Collision

The core trade-off of Feature Hashing is the risk of Bucket Collision (or Hash Collision).

- Explanation of Collision: Since we are mapping a potentially infinite set of strings (airport codes) to a finite and small number of buckets (100), it is highly probable that two or more different IATA codes will be mapped to the exact same bucket index.
    - *Example:* Airport "JFK" might map to index 42, and Airport "LHR" (London Heathrow) might also map to index 42.
- Impact on Model Accuracy: This collision leads to a loss of model accuracy because the model can no longer perfectly distinguish between the colliding features.
    - If the model learns that airport "JFK" is associated with a high delay probability, and "LHR" collides into the same bucket, the model will incorrectly associate the same high delay probability with "LHR," even if "LHR" historically has very low delays. The features become aliased or "smeared" together in the feature space, making it harder for the model to capture the unique predictive power of each airport.
- Mitigation: The primary way to mitigate collisions is by increasing the number of buckets (N). As N increases, the probability of collision decreases, but the trade-off with memory and model size is re-introduced. The choice of N balances model complexity/speed against the acceptable level of feature collision and resulting accuracy loss.

Code/Description: Embedding Layer Implementation

For the "Departure Airport" feature, the implementation involves three main steps within a neural network architecture:

1. Preprocessing (Vocabulary Mapping)

Unlike Feature Hashing, Embeddings require a pre-defined vocabulary.

- All unique "Departure Airport" IATA codes (e.g., JFK, LAX, SFO, etc.) present in the training data must be mapped to a unique integer ID, starting from 0.
    - Example: \{ \text{'JFK'} \rightarrow 0, \text{'LAX'} \rightarrow 1, \text{'SFO'} \rightarrow 2, \dots, \text{'YUL'} \rightarrow N-1 \}.
- When a data point is fed into the network, the string "JFK" is replaced by its integer ID, \mathbf{0}.

2. The Embedding Layer

The integer ID is then passed to an Embedding Layer. This layer is essentially a lookup table (a weight matrix, W_{emb}) with the dimensions:

W_{emb} \in \mathbb{R}^{V \times D}
Where:

- V: The Vocabulary Size (the total number of unique airports, e.g., 3,000).
- D: The Embedding Dimension (the chosen size of the vector, typically small, e.g., 10).

When the integer ID \mathbf{0} (representing 'JFK') is input, the Embedding Layer simply retrieves the 0-th row of the weight matrix W_{emb}. The output is a dense vector of size D:

\text{Embedding}(\text{'JFK'}) = \text{Row}_0 \text{ of } W_{emb} = [e_1, e_2, \dots, e_D]

3. Training and Learning

Crucially, the values within the W_{emb} matrix (the e_i values) are not hand-designed. They are learned during the neural network's training process via backpropagation, just like any other weights in the network. The network learns a representation where airports with similar characteristics (e.g., large hubs, regional airports, or airports with similar delay patterns) have their embedding vectors placed closer together in the D-dimensional space.

---

2. Analysis: Embeddings vs. One-Hot Encoding (OHE)

Embeddings fundamentally solve the issues associated with traditional One-Hot Encoding (OHE) for high-cardinality features.

Issue with OHE	How Embeddings Solve It
Sparse Matrices	Embeddings map the feature to a dense vector of fixed size D. A sparse OHE vector of size V \approx 3,000 becomes a dense embedding vector of size D \approx 10. This drastically reduces memory usage and improves computational efficiency.

Independence Assumption	OHE treats every airport as entirely independent, resulting in a zero dot product for any two different airport vectors. Embeddings learn a semantic relationship between airports. Airports with similar functions or geographical locations (e.g., "JFK" and "EWR") will have their embedding vectors closer together (higher dot product), allowing the model to generalize knowledge between similar categories.

The Critical Trade-off: Embedding Dimension (D)

The critical trade-off when using embeddings lies in the choice of the Embedding Dimension (D).

- Low D (e.g., D=4):
    - Pros: Very memory efficient and less likely to overfit to the training data.
    - Cons: The limited dimension might not provide enough capacity to capture all the relevant, independent characteristics and patterns of the airports. The model might fail to differentiate between airports whose differences are subtle but important for the prediction task.
- High D (e.g., D=100):
    - Pros: Provides a rich feature space, offering the model high capacity to learn fine-grained, independent representations for every airport.
    - Cons: Overfitting becomes a significant risk. The model may memorize the training data's airport-specific patterns, and the large number of parameters (V \times D) increases model size, training time, and memory consumption, mitigating the advantage over OHE.

Practical Rule of Thumb: A common heuristic for determining D is D \approx \sqrt[4]{V} or D \approx \log_2(V), where V is the vocabulary size, leading to small dimensions (often between 5 and 50) that balance information capture with model efficiency.

2. Analysis: Reframing the Task

The original problem is Regression: Predict the continuous value of "Arrival Delay" (e.g., 5.3 minutes, 18.0 minutes).

The reframed task is Classification: Predict the category or level of delay (e.g., On-Time, Short Delay, Long Delay).

This conversion is achieved by defining discrete, non-overlapping ranges (bins or buckets) for the continuous "Arrival Delay" values. Instead of predicting Delay=x, the model now predicts the probability that the flight belongs to CategoryA, CategoryB, or CategoryC.

---

2. Bucketing the Continuous Output

The transformation is done by bucketing the continuous "Arrival Delay" (in minutes) into discrete categories. The choice of buckets should be driven by the business or regulatory significance of the delay times.

Here are three example buckets for the delay times, which would become the new class labels:

Bucket Index	Class Label	Delay Time Range (Minutes)	Interpretation
0	On-Time/Early	$(-\infty,10]$	The flight arrived within the FAA-defined "On-Time" window (typically under 15 minutes, here set generously to 10 for safety).
1	Medium Delay	$(10,45]$	The flight incurred a noticeable, but manageable delay.
2	Significant Delay	$(45,\infty)$	The flight incurred a major delay that likely caused missed connections or significant passenger inconvenience.

Export to Sheets

Transformation Process: During the data preparation phase, the original continuous label (e.g., `Arrival Delay = 35 minutes`) is replaced by its corresponding categorical label (e.g., `Class Label = 1`). The model is then trained to predict one of these three classes.

---

3. Justification: Advantage of Discrete Probability Distribution

The primary advantage gained by reframing this problem from regression to classification is the model's ability to natively capture and express the inherent uncertainty and probabilistic nature of flight delays.

- Regression Output: A regression model provides a single numerical output (e.g., 8.2 minutes). This is a point estimate, and it gives the user no information about the confidence or alternative outcomes. If a flight has historically had delays of 5, 10, and 20 minutes under the same conditions, the regression model is forced to predict an average (e.g., 11.7 minutes).
- Classification Output (Probability Distribution Function - PDF): A classification model, using the Softmax activation function, outputs a discrete probability distribution (PDF) over the buckets. This directly captures the probabilistic reality:

Prediction=[P(On-Time),P(Medium),P(Significant)]

For example, the model might predict:

Prediction=[0.40,0.45,0.15]

This output is far more informative than a single number. It tells the airline or passenger that there is a 45% chance of a Medium Delay and a 40% chance of being On-Time. This allows the decision-maker to understand the risk profile associated with the prediction, which is crucial for operational planning (e.g., crew scheduling, gate allocation).

3. Stateless Serving Function Design Pattern

The Stateless Serving Function design pattern is the standard approach for deploying machine learning models in a scalable, high-throughput, and fault-tolerant manner. It treats the model prediction step as a self-contained, side-effect-free function call.

The core idea is that the prediction service holds no memory of past requests (i.e., it is *stateless*), allowing any instance of the serving function to handle any request, which is key for horizontal scaling.

Three Main Steps of the Solution

1. Export Model Artifact:
    - The fully trained model (weights and architecture) must be saved into a programming-language-agnostic format (e.g., ONNX, PMML, or SavedModel). This decoupling ensures the model can be served by high-performance, optimized serving engines (like TensorFlow Serving or

    TorchServe) regardless of the original training framework (e.g., PyTorch or Keras).

2. Containerization and Deployment:
    - The model artifact, along with the necessary dependencies (pre-processing logic, serving library, etc.), is packaged into a container (e.g., using Docker).
    - This container is deployed to a serverless or container orchestration platform (e.g., Kubernetes, Google Cloud Run, AWS Lambda). These platforms automatically manage the horizontal scaling (adding more identical instances) required to handle varying levels of real-time traffic.
3. API Gateway / Endpoint Creation:
    - A REST or gRPC API endpoint is created to expose the prediction function. Client applications send an input request (e.g., JSON payload of flight features) to this endpoint, and the function processes the request using the loaded model artifact and returns the prediction without storing any intermediate state.

---

2. Resilience and Monitoring

The Continued Model Evaluation design pattern is crucial for maintaining model performance and resilience in a production environment.

Purpose of Continued Model Evaluation

The purpose is to proactively detect and alert when the predictive performance of the deployed model degrades over time. Models trained on historical data inevitably face changes in real-world data distributions, leading to decreased accuracy. This pattern formalizes the continuous, automated measurement of key metrics (e.g., accuracy, precision, AUC) against collected ground truth data.

Three Types of Data for Continuous Monitoring

To monitor performance and identify degradation issues like concept drift or data drift, three essential data streams must be collected and aligned:

1. Prediction Data (Model Output):
    - The actual outputs of the deployed model for every request (e.g., the predicted delay category or probability distribution).

    - *Used for:* Analyzing the distribution of model predictions for Data Drift (i.e., is the model now predicting more "Long Delays" than before?) and comparing against incoming features.
2. Feature Data (Model Input):
    - The raw or pre-processed input features sent by the user for prediction (e.g., the 'Departure Airport', time of day, weather conditions).
    - *Used for:* Detecting Data Drift in the input features (i.e., are we suddenly seeing a massive surge of flights from a new airport or different time slots than the training data?).
3. Ground Truth Data (Actual Outcome):
    - The true, observed outcome for the prediction request, often collected days or weeks after the prediction was made (e.g., the actual arrival delay minutes recorded after the flight landed).
    - *Used for:* Calculating the true performance metrics (e.g., accuracy, ROC) to detect Concept Drift (i.e., the relationship between inputs and outputs has changed, even if the input data hasn't). This is the definitive measure of degradation.