

Machine Learning Operations

Homework 1

Efe Çetin

220901578

1A- Implementing Hashed Feature

```
import hashlib

def hash_airport_code(code, num_buckets=100):

    # convert strings to bytes
    encoded = code.encode("utf-8")

    # md5 hash üret
    hashed = hashlib.md5(encoded).hexdigest()

    # convert to integer
    hashed_int = int(hashed, 16)

    # bucket index
    return hashed_int % num_buckets

# apply it on ORIGIN feature
df["airport_bucket"] = df["ORIGIN"].apply(lambda x: hash_airport_code(str(x)))

df[["ORIGIN", "airport_bucket"]].head()
```

...

| | ORIGIN | airport_bucket |
|---|--------|----------------|
| 0 | FLL | 10 |
| 1 | MSP | 65 |
| 2 | DEN | 77 |
| 3 | MSP | 65 |
| 4 | MCO | 90 |

Analysis:

The “Departure Airport” feature is a high cardinality categorical variable with over 300+ unique airports. Methods like one hot encoding become inefficient because:

- They create hundreds of sparse columns, increasing memory usage and slowing model training.
- The training and test sets often have incomplete vocabularies.
- Never before seen airports cause a cold start problem, because one hot encoding cannot represent them.

The Hashed Feature solves all these problems:

- High Cardinality

Instead of creating hundreds of columns, hashing maps every airport to a small number of fixed buckets (e.g., 100 buckets).

- Incomplete Vocabulary

If the training set lacks certain airports, hashing still assigns them a bucket.

- Cold Start

New airports (unseen categories) still get hashed into a bucket = no model errors.

Trade-Off : Collision Risk

The major trade off is bucket collisions:

Two different airports may fall into the same hash bucket:

"IST" → bucket 12

"AMS" → bucket 12

This causes the model to treat them as the same feature, reducing accuracy.

Hashing reduces dimensionality but introduces information loss.

1B- Implementing Embeddings

```
• # Unique airport list
unique_airports = df["ORIGIN"].unique()
airport_to_id = {a: i for i, a in enumerate(unique_airports)}

# apply Mapping
df["airport_id"] = df["ORIGIN"].map(airport_to_id)

df[["ORIGIN", "airport_id"]].head()

import tensorflow as tf

# create embedding layer using keras
num_categories = len(unique_airports)
embedding_dim = 16

airport_input = tf.keras.Input(shape=(1,), dtype=tf.int32)

embedding_layer = tf.keras.layers.Embedding(
    input_dim=num_categories,
    output_dim=embedding_dim,
    name="airport_embedding"
)(airport_input)

x = tf.keras.layers.Flatten()(embedding_layer)
output = tf.keras.layers.Dense(1)(x)

model = tf.keras.Model(inputs=airport_input, outputs=output)
model.summary()
```

... Model: "functional"

| Layer (type) | Output Shape | Param # |
|---|---------------|---------|
| input_layer (InputLayer) | (None, 1) | 0 |
| airport_embedding (Embedding) | (None, 1, 16) | 6,080 |
| flatten (Flatten) | (None, 16) | 0 |
| dense (Dense) | (None, 1) | 17 |

Total params: 6,097 (23.82 KB)
Trainable params: 6,097 (23.82 KB)
Non-trainable params: 0 (0.00 B)

Analysis:

The “Departure Airport” feature has over 300 unique categories, which makes one-hot encoding inefficient. One-hot vectors become very large and sparse, slow down training, and treat all airports as completely independent, ignoring any meaningful similarities between them.

Embeddings solve these problems by mapping each airport to a dense, low dimensional vector. These vectors are trainable, meaning the model can learn relationships between airports such as hubs behaving similarly improving predictive power while keeping the input compact and efficient.

Trade Off:

The main trade-off is choosing the embedding size.

- Too small = not enough capacity to represent useful information
- Too large = risk of overfitting and unnecessary parameters

A common guideline is to set the embedding dimension to roughly the square root of the number of categories (around 16 – 20 for 347 airports).

2- Problem Representation (Reframing) (Analysis)

1. Turning Regression into Classification

“Arrival Delay” is a continuous number, so the task normally looks like a regression problem. But flight delays do not follow a fixed pattern. Even with the same conditions, one flight might be delayed 5 minutes and another 12 minutes. Because of this randomness, treating the delay as exact numbers is not always meaningful.

With the Reframing approach, we turn the problem into a classification task instead.

Instead of predicting the exact delay in minutes, we group delay values into categories and make the model predict which group the flight belongs to.

2. Creating Delay Buckets

To apply this idea, we divide the continuous delay values into buckets.

For example:

- 0: 0–10 minutes (on time or very small delay)
- 1: 11–30 minutes (medium delay)
- 2: 31+ minutes (large delay)

After this step, the model simply decides which bucket the delay is in.

3. Why This Reframing Helps

The main advantage is that the model can give us probabilities, not just one number. Instead of saying “The delay will be 17 minutes,” it can say something like:

- 60% chance of a small delay
- 30% chance of a medium delay
- 10% chance of a large delay

This is more realistic because real-world delays are uncertain.

Airline operations, weather, and airport traffic change constantly, so predicting an exact number is not always reliable. A probability distribution gives a clearer picture of the risk level.

3- Model Serving and Resilience (Analysis)

1. Stateless Serving

For deploying the model, we use the Stateless Serving Function pattern. “Stateless” means the server does not store any information from previous requests. Every prediction request is handled independently.

This approach has three main steps:

1. Export the model

First, we save the trained model into a format that does not depend on any specific programming language.

2. Load the model inside a lightweight service

A simple API service loads the exported model and uses it to make predictions.

3. Process requests one by one

Each request comes with its features, the model makes a prediction, returns the result, and forgets everything.

This makes the system easy to scale because any server can handle any request.

2. Resilience and Monitoring

After deployment, the model's performance may change over time, especially in real world environments like air travel, where patterns can shift. The Continued Model Evaluation pattern is used to monitor the model continuously and detect any problems early, such as concept drift or data drift.

To do this, the system needs to collect three types of data:

1. Model Inputs (Features)

The actual data being sent to the model during prediction.

Monitoring these inputs helps detect changes in the data distribution.

2. Model Predictions

The outputs the model produces in real time.

Tracking them helps identify unusual behavior or sudden shifts.

3. Actual Outcomes

The real delay values or delay classes, once they become available.

Comparing ground truth with predictions shows if accuracy is dropping.

By collecting these three data types continuously, we can detect when the model starts degrading and decide whether retraining or updating is needed.