Building, Deploying and Serving a Sticker Sales Forecasting Machine Learning Model with FastAI and Modal

[Silver Rubanza](https://www.rubanzasilver.com/)

May 15, 2025

### Background

In January 2025, Kaggle ran a playground series [competition](https://www.kaggle.com/competitions/playground-series-s5e1/overview) to forecast multiple years’ worth of sticker sales in different countries. For each id row, you have to predict the num\_sold which represents the number of stickers sold for each product type per store in each country.

This notebook walks through building, deploying and serving a machine learning model for forecasting sticker sales.

I do my tabular data preprocessing and build my model using [fastai](https://docs.fast.ai/), I then push my training, test and validation data to modal and run a script which runs the training job on modal cloud.

Now we can proceed to run some python scripts which deploy and serve the same model to a web endpoint in modal as we shall see below. I can then pass new data without a sales column to this API and expect it to return the same data with new sales predictions in a new num\_sold column.

[Modal](https://modal.com/) is a serveless cloud platform that enables us to run and execute any python code in the cloud without having to manage infrastructure. Modal makes it easy to attach GPU’S with just one line of code and can serve our functions as web endpoints.

Modal makes deploying ML models simple with:

* Containerized environments defined in code
* Seamless scaling
* GPU support (when needed)
* Easy endpoint creation

I previously showed how to train, serve and deploy a machine learning model to a live API endpoint with bentoml and bentocloud [here](https://nbsanity.com/static/ac40bb062434c7446906d3eb8875e061/load.html).

The solution we are building below will allow us to:

* Train a gradient boosting model to predict sticker sales
* Use Modal for serverless deployment
* Create an API endpoint for predictions
* Visualize results with a Streamlit dashboard.

After we are done deploying and serving our machine learning model, we build a UI where we pass in the API link. This can be used to make calls to the API from our streamlit dashboard.

[Streamlit](%28https%3A//streamlit.io/%29) is an open source framework that we can use to quickly build web applications.

Once we are done building our web application, we can pass in our data for a single prediction using a form etc.

We can also do batch prediction by passing in a csv file of the new data you want it will make predictions on.

Our batch prediction solution looks like below when we are done.

|  |
| --- |
| Sticker sales forecasting dashboard built with streamlit |

As you can see above, once we pass in previously unseen data, our model returns new predictions for the number of stickers to be sold.

### Evaluation

The competition submissions were evaluated on the [Mean Absolute Percentage Error (MAPE)](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_absolute_percentage_error.html) metric which expresses the average absolute error as a percentage of the actual values, making it easy to understand the relative size of errors.

For example, if you predict a value of 90 when the actual is 100, the percentage error is 10%, but if you predict 110 when the actual is 100, the percentage error is also 10%. MAPE averages these absolute percentage errors across all observations.

NB: The data for this competition was synthentically generated but was made to contain real world patterns and effects that you might see in real world data such as seasonality, weekend and holiday effect, etc.

Let’s get started!

### Project Structure

We start by creating our project file structure which should follow the structure below. Follow this and create the neccesary directories and files.

The training data, train.csv shall automatically be loaded into the data folder as we shall see in the Extract and Load section below hence we just need to create the data folder.

sticker\_sales\_model\_deployment/
├── data/
 └── train.csv # Training and test data
 └── test.csv # Test data
 └── sample\_submission.csv # sample submission data for kaggle
 └── transformed\_data.csv # Training data extracted from airbyte data - Approach 2
 └── local\_filename.csv # airbyte data from s3 - ELT job
 └── output.csv # Training data extracted from airbyte data - Approach 1
├── data\_upload.py # Upload data to modal cloud
├── train.py # Modal model training
├── serve.py # Create API service with Modal
├── test\_modal\_api.py # Modal API test
└── ui/
 └── streamlit\_ui.py # Streamlit dashboard

### Setting Up Your Environment

Let’s start by setting up our environment. We start by logging into modal by running modal setup via our terminal.

We then install the libraries we need for this project by running the below pip installations via your terminal.

You can do the same installation by running pip install catboost seaborn xgboost lightgbm fastkaggle -Uqq fastbook polars tqdm gradio dash streamlit plotly requests boto3 modal bentoml pandas via your terminal in your home directory which in this case would be sticker\_sales\_model\_deployment/.

%pip install catboost
#%pip install optuna
#%pip install optuna\_distributed
#%pip install openfe
%pip install seaborn
%pip install xgboost
%pip install lightgbm
%pip install fastkaggle
#%pip install h2o
%pip install -Uqq fastbook
%pip install polars
%pip install tqdm
#%pip install wandb
#%pip install sweetviz
#%pip install --upgrade scipy
%pip install gradio
%pip install dash
%pip install streamlit plotly requests
%pip install boto3
%pip install modal bentoml
%pip install streamlit pandas requests plotly

### Imports

import fastbook
fastbook.setup\_book()
from fastbook import \*
from fastai.tabular.all import \*
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from numpy import random
from tqdm import tqdm
from ipywidgets import interact

from fastai.imports import \*
#from fastkaggle import setup\_comp
np.set\_printoptions(linewidth=130)

from pathlib import Path
import os

from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean\_absolute\_percentage\_error
from sklearn.ensemble import VotingRegressor,StackingRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.model\_selection import KFold, cross\_val\_score,train\_test\_split
from sklearn.linear\_model import LinearRegression

#transformers and pipeline
from sklearn.compose import ColumnTransformer, make\_column\_transformer
from sklearn.pipeline import Pipeline, make\_pipeline
from sklearn import set\_config

import xgboost as xgb
from xgboost import plot\_importance
from xgboost import XGBClassifier
import lightgbm as lgb
from lightgbm import LGBMClassifier
from catboost import CatBoostRegressor,Pool, metrics, cv

import warnings

matplotlib.rc('image', cmap='Greys')
import scipy.stats as stats

#import h2o
#from h2o.automl import H2OAutoML

import gc
#import wandb

import pickle
from joblib import dump, load
#import sweetviz as sv
#from IPython.display import FileLink

import dash
from dash import dcc, html, dash\_table

import boto3
import io
import getpass
import json

from joblib import dump, load
import typing as t
import bentoml
from bentoml.validators import DataframeSchema

import modal

## Extract and Load with Airbyte

|  |
| --- |
| Airbyte - An open source data movement infrastructure for building extract and load (EL) data pipelines. |

To replicate/mirror how we would build & work with systems in real life, where data sources might be in different data sources such as ERP systems, spreadsheet software, social media apps, websites, etc.

I use [Airbyte](https://airbyte.com/), an open source tool that can be used to extract and load data from one source to another to build data pipelines.

This shows how one would go about extracting data from different sources to a single data warehouse / lake. With this we can always easily pull our neccesary data into our native training enviroment.

To demonstrate this in our example, we uploaded our training csv data into a google sheet and then pulled it [AWS S3](https://aws.amazon.com/pm/serv-s3/?trk=6b57822f-2fb8-4067-ab88-bac5e4381331&sc_channel=ps&ef_id=CjwKCAjwiezABhBZEiwAEbTPGMl3dan1CR4wYsinMeyynJXXo1n2ucMlT0VrRjc6t15aU9FN3BDqIhoCjJ8QAvD_BwE:G:s&s_kwcid=AL!4422!3!645125274608!p!!g!!aws%20s3%20web%20hosting!19574556923!145779858632&gad_campaignid=19574556923&gbraid=0AAAAADjHtp843MDwcEXA-az1hu6WnKFcn&gclid=CjwKCAjwiezABhBZEiwAEbTPGMl3dan1CR4wYsinMeyynJXXo1n2ucMlT0VrRjc6t15aU9FN3BDqIhoCjJ8QAvD_BwE). Airbyte is well suited to handle this.

I show the whole process in this [Video showing and Extract and Load job with Airbyte](https://drive.google.com/file/d/1oHN74oTUawj-WFLNgN8h2LqxzkWkEtrD/view?usp=sharing).

Once the data has been extracted from google sheets to AWS S3, we run the code below to pull the data from AWS S3 into our notebook and native training enviroment.

# Method 1: Download file to local and then read
s3\_client.download\_file('flexible-functions', 'sticker-sales/train.csv/ss\_train.csv', 'data/local\_filename\_ss.csv')
airbyte\_df = pd.read\_csv('data/local\_filename\_ss.csv')

Our data is in the \_airbyte\_data column, so lets go ahead and extract it. We have a few ways we can do this.

### Approach 1

# Extract the JSON blobs from the '\_airbyte\_data' column
data = airbyte\_df['\_airbyte\_data'].apply(json.loads)

# Convert the extracted JSON data into a DataFrame
extracted\_df = pd.json\_normalize(data)

# Save the extracted DataFrame to a CSV file in the data folder
extracted\_df.to\_csv("data/output.csv", index=False)
print("Data successfully extracted and saved to data/output.csv")

output\_df = pd.read\_csv('data/output.csv')
output\_df.head()

### Approach 2

We can load our data in a more optimal way. Lets do that below

def transform\_airbyte\_data(input\_file, output\_file):
 """
 Transform Airbyte raw data format back to original tabular format

 Parameters:
 input\_file (str): Path to the input file (CSV or JSON) from Airbyte
 output\_file (str): Path to save the transformed CSV file

 Returns:
 pd.DataFrame: The transformed DataFrame
 """
 file\_ext = os.path.splitext(input\_file)[1].lower()

 # Read the data based on file type
 if file\_ext == '.csv':
 df = pd.read\_csv(input\_file)
 elif file\_ext == '.json':
 df = pd.read\_json(input\_file, lines=True) # Assuming JSONL format
 else:
 raise ValueError(f"Unsupported file extension: {file\_ext}. Use .csv or .json")

 # Check if the data is in Airbyte format
 airbyte\_columns = [col for col in df.columns if col.startswith('\_airbyte\_')]

 if '\_airbyte\_data' in df.columns:
 # If \*airbyte\*data is a string column, parse it to dictionaries
 if df['\_airbyte\_data'].dtype == 'object' and isinstance(df['\_airbyte\_data'].iloc[0], str):
 df['\_airbyte\_data'] = df['\_airbyte\_data'].apply(json.loads)

 # Extract the data from \*airbyte\*data column
 extracted\_data = pd.json\_normalize(df['\_airbyte\_data'])

 # Convert numeric columns if needed
 for col in extracted\_data.columns:
 if col in ['id', 'num\_sold']:
 try:
 extracted\_data[col] = pd.to\_numeric(extracted\_data[col])
 except:
 pass # Keep as string if conversion fails

 # Save the result
 extracted\_data.to\_csv(output\_file, index=False)
 print(f"Transformed data saved to {output\_file}")

 return extracted\_data
 else:
 print("Data doesn't appear to be in Airbyte format. No transformation needed.")
 df.to\_csv(output\_file, index=False)
 return df

# Example usage
if \_\_name\_\_ == "\_\_main\_\_":
 # Use paths in the data folder
 input\_file = "data/local\_filename\_ss.csv" # or .json
 output\_file = "data/transformed\_data.csv"

 transformed\_df = transform\_airbyte\_data(input\_file, output\_file)
 transformed\_df.head()

### Load data

We only did an ELT job for the training dataset for demonstration

For the test and sample submission file, we shall load them from the local files already available in the data/ folder.

path = Path('')
path

Path('.')

!ls

train\_df = pd.read\_csv(path/'data/transformed\_data.csv',index\_col='id')
test\_df = pd.read\_csv(path/'data/test.csv',index\_col='id')
sub\_df = pd.read\_csv(path/'data/sample\_submission.csv')

## Exploratory Data Analysis

train\_df.columns

train\_df.shape,test\_df.shape

# For a specific column (e.g., column 'A')
missing\_values\_count = train\_df['num\_sold'].isnull().sum()
missing\_values\_count

train\_df['num\_sold'].isnull().mean() \* 100

## Data Upload - data\_upload.py

We are now going to start by uploading our data to modal using volumes. To quote the modal documentation

Modal Volumes provide a high-performance distributed file system for your modal applications. They are designed for write-once, read-many I/O workloads, like creating machine learning model weights and distributing them for inference.

Uploading our data will enable our training function that we run later to access the data it will need to train our machine learning model.

NB: You can achieve the same data upload functionality using modal Images by using the image.add\_local\_dir and image.add\_local\_file image builder methods. This can be done by creating an image that has our data like below

sticker\_data\_image = (
 modal.Image.debian\_slim()
 .pip\_install(["pandas", "numpy", "xgboost", "bentoml"])
 # Add your local data directory to the image
 .add\_local\_dir("./data", remote\_path="/data")
)

For this example, I shall be uploading my data using modal volumes as opposed to using images.

Navigate to your data\_upload.py file and paste the below code which whose purpose we shall explain in detail.

import modal
import sys
from pathlib import Path

# Create an app for the data upload
app = modal.App("sticker-data-upload")

Here we are initializing a modal application named sticker-data-upload.

Modal is a cloud platform that lets you run Python functions in the cloud. The App class is used to define a Modal application which is just a group of functions and classes that are deployed together.

# Create a volume to persist data
volume = modal.Volume.from\_name("sticker-data-volume", create\_if\_missing=True)

We then define our volume with modal.Volume.from\_name which creates a persistent storage volume in modal named sticker-data-volume. The create\_if\_missing=True flag means it will create the volume if it doesn’t already exist.

Volumes in modal are like shared disk space that can persist between function runs.

@app.function(volumes={"/data": volume})
def upload\_data(local\_data\_path):
 import shutil
 import os

 # Ensure the destination directory exists
 os.makedirs("/data", exist\_ok=True)

 # Copy all files from the local data directory to the volume
 for file in Path(local\_data\_path).glob("\*"):
 dest = f"/data/{file.name}"
 if file.is\_file():
 shutil.copy(file, dest)
 print(f"Copied {file} to {dest}")

 # List files to confirm upload
 print("\nFiles in Modal volume:")
 for file in Path("/data").glob("\*"):
 print(f" - {file}")

Remember our modal app can consist of various functions and classes. To explicitly register an object with the app, we use the @app.function() decorator.

We can now define the function, upload\_data to upload our data. The function takes in one argument, local\_data\_path which is the path to the directory on our local machine that contains the files we would like to upload.

Inside the function, start by creating the /data directory if it does not exist. We then iterate through our local file directory.

For each file, we create a destination path in our modal volume, and if it is a file, copy it from our local directory to our previously defined modal volume and then it prints a confirmation message.

Finally, It prints a list of all files in the volume.

@app.local\_entrypoint()
def main():
 if len(sys.argv) > 1:
 data\_path = sys.argv[1]
 else:
 data\_path = "./data" # Default path

 print(f"Uploading data from {data\_path}")
 upload\_data.remote(data\_path)

The @app.local\_entrypoint() decorator defines a Command Line entry point for our modal app and it marks a function, in this case main to be executed locally.

This function then parses command-line arguments such as the data path, print a local message, and then triggers the actual data uploading process.

Our function, main triggers our function upload\_data to be executed remotely in the modal cloud by calling upload\_data.remote(data\_path) .

To upload the data, run python data\_upload.py ./data via the terminal to trigger the data upload.

## Model Training - train.py

Next, we’ll create a script to train our forecasting model. Navigate to your train.py file and paste the code in the below model training section whose code i shall explain.

#import modal
#import pandas as pd
#import numpy as np
#from fastai.tabular.all import \* # Move this import to the top level
#import xgboost as xgb
#import bentoml
#from pathlib import Path
#import os
#import pickle

# Define Modal resources
app = modal.App("sticker-sales-forecast")

Above, we define a modal app called sticker-sales-forecast.

image = modal.Image.debian\_slim().pip\_install([
 "fastai",
 "xgboost",
 "bentoml",
 "scikit-learn",
 "pandas",
 "numpy",
 "torch"
])

We then define an image which is a snapshot of our container’s filesystem state.

We can easily add any third-party packages like torch, pandas by passing in all the packages we need to the pip\_install method of an image as shown above.

volume = modal.Volume.from\_name("sticker-data-volume")

After defining our dependencies, we call and attach an existing volume called sticker-data-volume which we previously defined when doing our data upload.

This contains the data needed for our model training in the train\_model function defined below.

We define our @app.function decorator passing in our specified image and volume mounted at “/data”.

# Define paths for pickle-based model saving
MODEL\_PATH = "/data/sticker\_sales\_model.pkl"
PREPROC\_PATH = "/data/sticker\_sales\_preproc.pkl"

@app.function(image=image, volumes={"/data": volume})
def train\_model():
 # No need to import fastai.tabular.all here since we moved it to the top

 # Set up paths
 path = Path('/data/')

 # Check if data files exist
 print("Files available in volume:")
 for file in path.glob("\*"):
 print(f" - {file}")

 # Load data
 print("Loading data...")
 train\_df = pd.read\_csv(path/'train.csv', index\_col='id')
 test\_df = pd.read\_csv(path/'test.csv', index\_col='id')

 # Data preprocessing
 print("Preprocessing data...")
 train\_df = train\_df.dropna(subset=['num\_sold'])
 train\_df = add\_datepart(train\_df, 'date', drop=False)
 test\_df = add\_datepart(test\_df, 'date', drop=False)

 # Feature preparation
 cont\_names, cat\_names = cont\_cat\_split(train\_df, dep\_var='num\_sold')
 splits = RandomSplitter(valid\_pct=0.2)(range\_of(train\_df))
 to = TabularPandas(train\_df, procs=[Categorify, FillMissing, Normalize],
 cat\_names=cat\_names,
 cont\_names=cont\_names,
 y\_names='num\_sold',
 y\_block=CategoryBlock(),
 splits=splits)
 dls = to.dataloaders(bs=64)

 # Prepare training data
 X\_train, y\_train = to.train.xs, to.train.ys.values.ravel()
 X\_test, y\_test = to.valid.xs, to.valid.ys.values.ravel()

 # Train XGBoost model
 print("Training XGBoost model...")
 xgb\_model = xgb.XGBRegressor()
 xgb\_model = xgb\_model.fit(X\_train, y\_train)

 # Save model with BentoML
 print("Saving model with BentoML...")
 model\_tag = bentoml.xgboost.save\_model(
 "sticker\_sales\_v1",
 xgb\_model,
 custom\_objects={
 "preprocessor": {
 "cont\_names": cont\_names,
 "cat\_names": cat\_names
 }
 }
 )

 # Save model with pickle
 print(f"Saving model with pickle to {MODEL\_PATH}...")
 with open(MODEL\_PATH, 'wb') as f:
 pickle.dump(xgb\_model, f)

 # Save preprocessing info separately
 print(f"Saving preprocessing info to {PREPROC\_PATH}...")
 preproc\_info = {
 "cont\_names": cont\_names,
 "cat\_names": cat\_names,
 "procs": [Categorify, FillMissing, Normalize]
 }
 with open(PREPROC\_PATH, 'wb') as f:
 pickle.dump(preproc\_info, f)

 # Ensure changes are committed to the volume
 volume.commit()

 print(f"Model saved: {model\_tag} and to pickle files")
 return str(model\_tag)

In our code above, I am defining a function train\_model.

In this function, I start by setting up the paths and loading the training and test data into their respective data frames, train\_df and test\_df.

I then do some basic preprocessing steps on our data frames.

First, from our exploratory data analysis above, I noticed that our target column, num\_sold has missing values which would intefere with our model training, so first we handle this.

We can deal with these missing values with several techniques like filling in with the mean, median, or a random value. From testing, I noticed the best strategy for this is to drop all the rows with missing values by running train\_df = train\_df.dropna(subset=['num\_sold']).

We then use the [add\_datepart](https://docs.fast.ai/tabular.core.html#add_datepart) helper function from [fast.ai](https://docs.fast.ai/) to add columns/features relevant to the date column in a data frame if available. This is defined using add\_datepart (df, field\_name, prefix=None, drop=True, time=False).

For example, if we have a date column with a row that has 2019-12-04. We can derive new columns from that date such as Year, Month, Day, Dayofweek, Is\_month\_start, Is\_quarter\_end, etc.

Our machine learning model expects our training data to be in a certain format. Luckily, fastai has functions which we shall see below that can help transform this raw data into a format that can be efficiently and effectively processed by a neural network.

We then go on to define categorical and continuous variables, I use the fastai [cont\_cat\_split](https://docs.fast.ai/tabular.core.html#cont_cat_split) function to separate my dataset variables into categorical and continuous variables based of the cardinality of my column values.

cont\_cat\_split takes an argument max card whose default is 20. If the number of unique values exceeds 20 (max\_card value) for a particular column, that column is considered continuous, and vice versa.

I then use [RandomSplitter](https://docs.fast.ai/data.transforms.html#randomsplitter), a fastai function that splits our training data into new training and validation sets.

It separates the training dataset into training and validation sets based on the value of the argument valid\_pct.

We can now use fastai’s TabularPandas class to create a TabularPandas object that applies given preprocessing steps to our data.

This creates a data frame wrapper that takes in different arguments and knows which columns are categorical and continuous. I also define the target variable, y\_name, the type of target, the problem we are dealing with such as a regression problem in this case, and the way to split our data which was previously defined in the splits above.

I define a list of preprocessing steps, Procs, to be taken on our data which we pass to our TabularPandas object. Procs contains the below preprocessing steps.

* Categorify deals with the categorical variables and converts each category into a list of indexable numerical integers, creating numerical input which is required by our model. Each category corresponds to a different number.
* FillMissing as its name suggests, fills in the missing values in columns with continuous values. This can be filled in with the median, mode of that column, or a constant, with the default being the median value for that particular column.

FillMissing supports using the mode and a constant as strategies for dealing with missing values. We can do this by changing the FillMissing argument fill\_strategy to mode or constant.

* Normalize puts the continuous variables between a standardized scale without losing important information by subtracting the mean and dividing by the standard deviation.

We can now define a DataLoader which is an extension of PyTorch’s DataLoaders class albeit with more functionality. This takes in our data above from the TabularPandas object and prepares it as input for our model passing it in batches which we defined by our batch size set by the bs argument.

The DataLoaders and TabularPandas objects allow us to build data objects we can use for training without specifically changing the raw input data.

The dataloader then acts as input for our models.

* To use other libraries with fastai, I extract the x’s and y’s from my TabularPandas object which I used to preprocess the data. I can now directly use the training and validation set values I extracted above as direct input for decision trees and gradient-boosting models.

An instance of XGBRegressor is then created with default parameters, and a model is trained by calling .fit() with the training features (X\_train) and target values (y\_train). The model is finally serialized and saved to the bentoML model store.

### Entry Point

@app.local\_entrypoint()
def main():
 # Train the model remotely
 print("Starting model training on Modal...")
 model\_tag = train\_model.remote()
 print(f"Model training completed. Model tag: {model\_tag}")
 print(f"Model and preprocessing info also saved as pickle files at {MODEL\_PATH} and {PREPROC\_PATH}")

Above, we define our entry point.

As explained before the @app.local\_entrypoint() decorator, declared above makes this function the entry point when you run the script locally and it triggers the remote execution of the train\_model function in modal cloud by calling train\_model.remote() within our main function.

@app.local\_entrypoint()
def main():
 # Train the model remotely
 print("Starting model training on Modal...")
 model\_tag = train\_model.remote()
 print(f"Model training completed. Model tag: {model\_tag}")

Now run python train.py via the terminal to trigger the machine learning model training and saving in modal cloud.

## Trained model serving and deployment - serve.py

After training our model, we can proceed to deploy and serve our trained machine learning model.

To do this, go to the serve.py file and paste all the code in the below model serving and deployment section.

Just like before, we create a modal app called sticker-sales-api which acts as the container for all the functions that will be deployed.

#import modal
#import pandas as pd
#import numpy as np
#from fastapi import File, UploadFile, Form, HTTPException
#import io

# Create app definition
app = modal.App("sticker-sales-api")

# Define base image with all dependencies
base\_image = (modal.Image.debian\_slim()
 .pip\_install("pydantic==1.10.8")
 .pip\_install("fastapi==0.95.2")
 .pip\_install("uvicorn==0.22.0")
 .pip\_install("bentoml==1.3.2")
 .pip\_install([
 "xgboost==1.7.6",
 "scikit-learn==1.3.1",
 "pandas",
 "numpy",
 ]))

# Create the fastai image by extending the base image
fastai\_image = (base\_image
 .pip\_install(["fastai", "torch"]))

We then go ahead and define two separate container images.

The main image, base\_image includes dependencies for the API (FastAPI, pydantic, uvicorn) plus modal-related packages (BentoML, XGBoost, scikit-learn).

A separate fastai\_image is created to avoid dependency conflicts, as fastai has specific requirements for torch and other packages.

# Create volume to access data
data\_volume = modal.Volume.from\_name("sticker-data-volume")

Just like before, we call and attach an exisiting volume called sticker-data-volume.

# Simple health endpoint
@app.function(image=base\_image)
@modal.fastapi\_endpoint(method="GET")
def health():
 """Health check endpoint to verify the API is running"""
 return {"status": "healthy", "service": "sticker-sales-api"}

We define a health endpoint to provide a simple way to check if our API service is alive and functioning correctly. We can use this to verify our service is available without needing to test the full prediction functionality.

# Function to train and save model
@app.function(image=fastai\_image, volumes={"/data": data\_volume})
def serve\_model():
 """Load or train an XGBoost model"""
 import xgboost as xgb
 from fastai.tabular.all import add\_datepart, TabularPandas, cont\_cat\_split
 from fastai.tabular.all import Categorify, FillMissing, Normalize, CategoryBlock, RandomSplitter, range\_of
 from pathlib import Path
 import pickle
 import os
 import bentoml

 # Model tag used in train.py
 model\_tag = "sticker\_sales\_v1"

 # Create a path to save the model for future use
 model\_path = "/data/sticker\_sales\_model.pkl"

 try:
 # First attempt: Try loading from BentoML
 print(f"Attempting to load model from BentoML with tag '{model\_tag}'...")
 try:
 bento\_model = bentoml.xgboost.load\_model(model\_tag)
 print(f"Successfully loaded model from BentoML.")
 return bento\_model
 except Exception as e:
 print(f"Could not load from BentoML: {str(e)}")

 # Second attempt: Try loading from pickle
 if os.path.exists(model\_path):
 print(f"Loading existing model from pickle at {model\_path}")
 with open(model\_path, 'rb') as f:
 model = pickle.load(f)
 return model

 # Third attempt: Train a new model if neither option worked
 print("No existing model found. Training new model...")
 # Load and preprocess training data
 path = Path('/data/')

 print("Loading training data...")
 train\_df = pd.read\_csv(path/'train.csv', index\_col='id')

 # Drop rows with missing target values
 train\_df = train\_df.dropna(subset=['num\_sold'])

 # Add date features
 print("Preprocessing data...")
 train\_df = add\_datepart(train\_df, 'date', drop=False)

 # Feature preparation
 cont\_names, cat\_names = cont\_cat\_split(train\_df, dep\_var='num\_sold')
 splits = RandomSplitter(valid\_pct=0.2)(range\_of(train\_df))

 # Create TabularPandas processor
 to = TabularPandas(train\_df,
 procs=[Categorify, FillMissing, Normalize],
 cat\_names=cat\_names,
 cont\_names=cont\_names,
 y\_names='num\_sold',
 y\_block=CategoryBlock(),
 splits=splits)

 # Prepare training data
 X\_train, y\_train = to.train.xs, to.train.ys.values.ravel()

 # Train a simple XGBoost model
 print("Training XGBoost model...")
 xgb\_model = xgb.XGBRegressor(n\_estimators=100)
 xgb\_model.fit(X\_train, y\_train)

 # Save model to both formats

 # 1. Save to BentoML
 print(f"Saving model to BentoML with tag '{model\_tag}'...")
 bentoml.xgboost.save\_model(
 model\_tag,
 xgb\_model,
 custom\_objects={
 "preprocessor": {
 "cont\_names": cont\_names,
 "cat\_names": cat\_names
 }
 }
 )

 # 2. Save to pickle
 print(f"Saving model to pickle at {model\_path}")
 with open(model\_path, 'wb') as f:
 pickle.dump(xgb\_model, f)

 # Save preprocessing info separately
 preproc\_path = "/data/sticker\_sales\_preproc.pkl"
 print(f"Saving preprocessing info to {preproc\_path}...")
 preproc\_info = {
 "cont\_names": cont\_names,
 "cat\_names": cat\_names,
 "procs": [Categorify, FillMissing, Normalize]
 }
 with open(preproc\_path, 'wb') as f:
 pickle.dump(preproc\_info, f)

 # Ensure changes are committed to the volume
 volume.commit()

 print("Model training and saving complete!")
 return xgb\_model

 except Exception as e:
 import traceback
 print(f"Error loading/training model: {str(e)}")
 print(traceback.format\_exc())
 raise

Above we define a function serve\_model

First, it checks if a trained model exists in our current model\_path in our modal volume.

If found, it loads the existing model; if not, it trains a new one. This is a fallback mechanism built into the serving API that first checks if a model exists at a specific path on the volume.

This ensures the API endpoint can always return predictions, even if the scheduled training hasn’t run yet.

# CSV upload endpoint
@app.function(image=fastai\_image, volumes={"/data": data\_volume})
@modal.fastapi\_endpoint(method="POST")
async def predict\_csv(file: UploadFile = File(...)):
 """API endpoint for batch predictions from a CSV file"""
 import xgboost as xgb
 import io
 import pickle
 from fastai.tabular.all import add\_datepart, TabularPandas, cont\_cat\_split
 from fastai.tabular.all import Categorify, FillMissing, Normalize, CategoryBlock, RandomSplitter, range\_of
 from pathlib import Path

 try:
 # First, train or load model
 model = serve\_model.remote()

 # Read uploaded CSV file content
 contents = await file.read()

 # Parse CSV data
 try:
 test\_df = pd.read\_csv(io.BytesIO(contents))
 except Exception as e:
 return {
 "success": False,
 "error": f"Failed to parse uploaded CSV: {str(e)}"
 }

 # Load the training data for preprocessing
 path = Path('/data/')
 train\_df = pd.read\_csv(path/'train.csv', index\_col='id')
 train\_df = train\_df.dropna(subset=['num\_sold'])

 # Add date features to both datasets
 train\_df = add\_datepart(train\_df, 'date', drop=False)
 test\_df = add\_datepart(test\_df, 'date', drop=False)

 # Feature preparation
 cont\_names, cat\_names = cont\_cat\_split(train\_df, dep\_var='num\_sold')
 splits = RandomSplitter(valid\_pct=0.2)(range\_of(train\_df))

 # Create TabularPandas processor
 to = TabularPandas(train\_df,
 procs=[Categorify, FillMissing, Normalize],
 cat\_names=cat\_names,
 cont\_names=cont\_names,
 y\_names='num\_sold',
 y\_block=CategoryBlock(),
 splits=splits)

 # Create a test dataloader
 dls = to.dataloaders(bs=64)
 test\_dl = dls.test\_dl(test\_df)

 # Make predictions using our model
 predictions = model.predict(test\_dl.xs)

 # Return the predictions as a simple list
 return predictions.tolist()

 except Exception as e:
 import traceback
 return {
 "success": False,
 "error": f"Error processing CSV: {str(e)}",
 "traceback": traceback.format\_exc()
 }

We then define a predict\_csv function which creates a REST API endpoint that enables batch predictions for multiple sticker sales records via a CSV file upload.

predict\_csv starts by calling the serve\_model function previously defined to be executed. This loads up our machine learning model if we have one available or trains a new one as we saw before.

We then read in the uploaded file csv file as bytes, wrap it in an in-memory buffer using BytesIO, and safely parse it into a pandas DataFrame with error handling.

Above, we load our training data to enable us to recreate the preprocessing steps using TabularPandas. This is necessary because Fastai needs the original transformations to correctly preprocess the new test data and requires this to generate the test\_dl from the incoming test data

When making predictions with tabular machine learning models, we must apply the exact same preprocessing transformations to new data that were used during training.

With this we can now add date-related features with add\_datepart, apply the same preprocessing transformations ( Categorify, FillMissing,Normalize) that were used during training and finally create a test dataloader, test\_dl with the proper format expected by the model.

We can now make predictions by running the model on the prepared data which returns the sales predictions as a JSON array that can be consumed by clients.

The above ensures consistent processing between training and inference, but creates unnecessary overhead and is probably not the most optimal way of handling the preprocessing as it requires us to reload and reprocess the training data for each prediction.

So we can try out another implementation and try to leverage our previously saved preprocessing information at inference time as opposed to loading the training dataset everytime.

We previously saved our steps in the serve\_model function where we did this

preproc\_info = {
 "cont\_names": cont\_names,
 "cat\_names": cat\_names,
 "procs": [Categorify, FillMissing, Normalize]
}

To do this, we shall redefine predict\_csv to add the use of the saved preprocessing steps with the previous implementation as a backup if the preprocessing information is not available.

# CSV upload endpoint - with debugging info (commented out)
@app.function(image=fastai\_image, volumes={"/data": data\_volume})
@modal.fastapi\_endpoint(method="POST")
async def predict\_csv(file: UploadFile = File(...)):
 """API endpoint for batch predictions from a CSV file using cached preprocessing"""
 import xgboost as xgb
 import io
 import pickle
 import os
 import traceback
 from fastai.tabular.all import add\_datepart, TabularPandas, cont\_cat\_split
 from fastai.tabular.all import Categorify, FillMissing, Normalize, CategoryBlock, RandomSplitter, range\_of
 from pathlib import Path

 # Uncomment for debugging
 # response\_data = {"success": False, "debug\_info": {}}

 try:
 # Debug information
 # response\_data["debug\_info"]["step"] = "Starting prediction process"

 # First, load or train model
 model = serve\_model.remote()
 # response\_data["debug\_info"]["model\_loaded"] = True

 # Read uploaded CSV file content
 contents = await file.read()

 # Parse CSV data
 try:
 test\_df = pd.read\_csv(io.BytesIO(contents))
 # response\_data["debug\_info"]["test\_columns"] = test\_df.columns.tolist()
 # response\_data["debug\_info"]["test\_shape\_before"] = test\_df.shape
 except Exception as e:
 return {
 "success": False,
 "error": f"Failed to parse uploaded CSV: {str(e)}"
 }

 # Add date features to the test dataset
 test\_df = add\_datepart(test\_df, 'date', drop=False)
 # response\_data["debug\_info"]["test\_shape\_after\_datepart"] = test\_df.shape
 # response\_data["debug\_info"]["test\_columns\_after\_datepart"] = test\_df.columns.tolist()

 # Load the full training data to ensure proper preprocessing
 path = Path('/data/')
 train\_df = pd.read\_csv(path/'train.csv', index\_col='id')
 train\_df = train\_df.dropna(subset=['num\_sold'])

 # response\_data["debug\_info"]["train\_columns"] = train\_df.columns.tolist()
 # response\_data["debug\_info"]["train\_shape"] = train\_df.shape

 # Add date features to training data
 train\_df = add\_datepart(train\_df, 'date', drop=False)
 # response\_data["debug\_info"]["train\_columns\_after\_datepart"] = train\_df.columns.tolist()

 # Feature preparation
 cont\_names, cat\_names = cont\_cat\_split(train\_df, dep\_var='num\_sold')
 # response\_data["debug\_info"]["categorical\_features"] = cat\_names
 # response\_data["debug\_info"]["continuous\_features"] = cont\_names

 # Check if test data has all required columns
 missing\_cols = []
 for col in cat\_names + cont\_names:
 if col not in test\_df.columns:
 missing\_cols.append(col)

 if missing\_cols:
 # response\_data["debug\_info"]["missing\_columns"] = missing\_cols

 # Add missing columns with default values
 for col in missing\_cols:
 if col in cat\_names:
 test\_df[col] = "unknown" # Default value for categorical
 else:
 test\_df[col] = 0.0 # Default value for continuous

 # response\_data["debug\_info"]["columns\_added"] = missing\_cols

 # Create TabularPandas processor
 splits = RandomSplitter(valid\_pct=0.2)(range\_of(train\_df))
 to = TabularPandas(train\_df,
 procs=[Categorify, FillMissing, Normalize],
 cat\_names=cat\_names,
 cont\_names=cont\_names,
 y\_names='num\_sold',
 y\_block=CategoryBlock(),
 splits=splits)

 # Create dataloaders
 dls = to.dataloaders(bs=64)

 # Process the test data and make predictions
 test\_dl = dls.test\_dl(test\_df)
 # response\_data["debug\_info"]["test\_xs\_shape"] = test\_dl.xs.shape

 # Make predictions
 predictions = model.predict(test\_dl.xs)

 # Return predictions in the format expected by test\_modal\_api.py
 return predictions.tolist()

 # To return structured response with debug info, use this instead:
 # response\_data["success"] = True
 # response\_data["predictions"] = predictions.tolist()
 # return response\_data

 except Exception as e:
 import traceback
 return {
 "success": False,
 "error": f"Error processing CSV: {str(e)}",
 "traceback": traceback.format\_exc()
 }

@app.local\_entrypoint()
def main():
 """Local entrypoint for testing the API"""
 print("Starting sticker-sales-api...")

 # Pre-train the model to ensure it exists
 print("Training model...")
 serve\_model.remote()
 print("Model training complete!")

 print("\nAPI is ready for use at:")
 print("- Health check: https://flexible-functions-ai--sticker-sales-api-health.modal.run")
 print("- CSV predictions: https://flexible-functions-ai--sticker-sales-api-predict-csv.modal.run")

The @app.local\_entrypoint() carries out a similar function as before where it triggers the remote execution of the serve\_model function in modal cloud when we run modal deploy serve.py.

We also print the endpoint information such as the URLs where the deployed API endpoints can be accessed, making it easier for developers to know where to send requests.

This pattern creates a smoother deployment experience where your model is prepared and ready before any user makes their first API call. It’s particularly helpful during initial deployment and testing phases when you want immediate feedback on whether your endpoints are working correctly.

After, we are done running, deploying and serving our app on modal cloud. We can see the App and its API’s in the modal dashboard.

|  |
| --- |
| Live sticker sales forecasting app in the Modal dashboard |

Above, we essentially create a convenient batch prediction service, allowing our Streamlit dashboard (or any client) to upload a CSV file and immediately get back predictions without having to handle the preprocessing or model loading logic themselves.

We then build a dashboard below with streamlit that uses these predictions to create visualizations, calculate KPIs, and enable interactive filtering of the results but first we shall test our API with a simple script.

## Testing the API - test\_modal\_app.py

Navigate to the home directory and paste the below code in this section into test\_modal\_api.py. We shall use this to test that our API is working as expected.

The below code makes a request to an API that predicts sticker sales.

import requests
import pandas as pd
import io

# The URL of your CSV prediction endpoint
url = "https://flexible-functions-ai--sticker-sales-api-predict-csv.modal.run"

# Create a sample CSV with test data
test\_data = pd.DataFrame([
 {
 "date": "2023-01-15",
 "country": "US",
 "store": "Store\_001",
 "product": "Sticker\_A"
 },
 {
 "date": "2023-01-15",
 "country": "Canada",
 "store": "Discount Stickers",
 "product": "Holographic Goose"
 },
 {
 "date": "2023-01-16",
 "country": "UK",
 "store": "Sticker World",
 "product": "Kaggle"
 }
])

# Save the test data to a CSV file in memory
csv\_buffer = io.StringIO()
test\_data.to\_csv(csv\_buffer, index=False)
csv\_bytes = csv\_buffer.getvalue().encode()

# Prepare the file for upload
files = {'file': ('test\_data.csv', csv\_bytes, 'text/csv')}

# Make the prediction request
print(f"Sending request to {url}...")
response = requests.post(url, files=files)

# Print the result
print("Status code:", response.status\_code)

# Try to parse the JSON response
try:
 prediction = response.json()
 print("Prediction:", prediction)

 # If the prediction is a list as expected
 if isinstance(prediction, list):
 # Create a DataFrame with predictions
 result\_df = test\_data.copy()
 result\_df['predicted\_sales'] = prediction
 print("\nPrediction results:")
 print(result\_df)
except Exception as e:
 print("Error parsing response:", e)
 print("Response text:", response.text[:500])

To run the api test, run python test\_modal\_api.py via the terminal.

This returns predictions, something like below.

|  |
| --- |
| Result after running python test\_modal\_api.py |

## User Interface with Streamlit - streamlit\_ui.py

Finally, let’s create a dashboard with streamlit to visualize our predictions. Streamlit as described before,

is an open-source Python framework for data scientists and AI/ML engineers to deliver dynamic data apps with only a few lines of code.

We navigate to our home directory and navigate to our streamlit code folder named ui. We then paste the code in this section into a file named streamlit\_ui.py.

This code enables us to create an interactive dashboard using streamlit that allows users to upload sales data, get predictions from a remote API, and visualize the results.

# Import required libraries
import streamlit as st
import pandas as pd
import requests
import json
import plotly.express as px
from datetime import datetime
import io

def load\_and\_predict\_data(csv\_path):
 """
 Sends the test CSV to the API endpoint and gets predictions
 """
 try:
 # Read the CSV file
 test\_df = pd.read\_csv(csv\_path)

 # Set the API URL
 api\_url = "https://flexible-functions-ai--sticker-sales-api-predict-csv.modal.run"

 # Prepare the file for upload using proper multipart/form-data format
 with open(csv\_path, 'rb') as f:
 files = {'file': ('test\_data.csv', f, 'text/csv')}

 # Make the request
 st.info(f"Sending data to API at {api\_url}...")
 response = requests.post(api\_url, files=files)

 # Check if the request was successful
 if response.status\_code == 200:
 try:
 result = response.json()

 # Check if the result is an error message
 if isinstance(result, dict) and not result.get('success', True):
 st.error(f"Error from API: {result.get('error', 'Unknown error')}")
 st.error("Using dummy predictions for demonstration")
 test\_df['predicted\_sales'] = 100 # Dummy predictions
 else:
 # Assume the result is a list of predictions
 if isinstance(result, list) and len(result) == len(test\_df):
 st.success("Successfully received predictions from API")
 test\_df['predicted\_sales'] = result
 else:
 st.warning(f"Unexpected response format. Using dummy predictions.")
 st.json(result) # Show the actual response
 test\_df['predicted\_sales'] = 100 # Dummy predictions
 except Exception as e:
 st.error(f"Error parsing API response: {str(e)}")
 st.text(f"Response text: {response.text[:500]}") # Show first 500 chars
 test\_df['predicted\_sales'] = 100 # Dummy predictions
 else:
 st.error(f"API returned status code: {response.status\_code}")
 st.text(f"Response text: {response.text[:500]}") # Show first 500 chars
 test\_df['predicted\_sales'] = 100 # Dummy predictions

 # Convert date column to datetime
 test\_df['date'] = pd.to\_datetime(test\_df['date'])

 return test\_df

 except Exception as e:
 st.error(f"Error processing prediction request: {str(e)}")
 # Return a dummy dataframe for demonstration
 test\_df = pd.read\_csv(csv\_path)
 test\_df['predicted\_sales'] = 100 # Dummy prediction values
 test\_df['date'] = pd.to\_datetime(test\_df['date'])
 return test\_df

Above,we define our load\_and\_predict\_data(csv\_path) function, which powers the sales forecasting feature in the Streamlit app.

The function reads a test CSV, sends it to an external API for prediction, handles the response, and returns a DataFrame with predicted sales all while implementing proper error handling.

In our function, load\_and\_predict\_data, we

1. **Read the Input CSV**

Reads the uploaded CSV file into a pandas DataFrame.

test\_df = pd.read\_csv(csv\_path)

1. **Define the API Endpoint**

Set the URL of the deployed prediction API.

api\_url = "https://flexible-functions-ai--sticker-sales-api-predict-csv.modal.run"

1. **Send the CSV to the API**

Send the CSV file using a POST request with multipart/form-data.

with open(csv\_path, 'rb') as f:
 files = {'file': ('test\_data.csv', f, 'text/csv')}
 response = requests.post(api\_url, files=files)

1. **Handle the API Response**

Check the response status and process the JSON result.

if response.status\_code == 200:
 result = response.json()
else:
 test\_df['predicted\_sales'] = 100 # Dummy fallback

1. **Process and Validate Predictions** Add valid predictions to the DataFrame or use dummy values if the format is invalid.

if isinstance(result, list) and len(result) == len(test\_df):
 test\_df['predicted\_sales'] = result
else:
 test\_df['predicted\_sales'] = 100 # Dummy fallback

1. **Convert the Date Column** Ensure the 'date' column is in datetime format.

test\_df['date'] = pd.to\_datetime(test\_df['date'])

1. **Return the Final DataFrame** Return the DataFrame with the predictions appended.

return test\_df

1. **Catch and Handle Unexpected Errors** Use try-except to prevent app crashes and provide dummy output in case of failure.

except Exception as e:
 test\_df = pd.read\_csv(csv\_path)
 test\_df['predicted\_sales'] = 100
 test\_df['date'] = pd.to\_datetime(test\_df['date'])
 return test\_df

load\_and\_predict\_datafunction enables seamless interaction between the Streamlit app and a backend machine learning model. It is: - **Robust**: Handles network and response errors gracefully. - **User-friendly**: Uses Streamlit to notify users of progress or issues. - **Fail-safe**: Provides dummy predictions if the API fails or returns unexpected results.

def create\_dashboard():
 """
 Creates the Streamlit dashboard with enhanced filters, KPI cards, and visualizations
 """
 st.title("Sales Prediction Dashboard")

 # Add custom CSS for dark theme cards
 st.markdown("""
 <style>
 .metric-card {
 background-color: #2C3333;
 padding: 20px;
 border-radius: 10px;
 margin: 10px 0;
 }
 .metric-label {
 color: #718096;
 font-size: 0.875rem;
 }
 .metric-value {
 color: white;
 font-size: 1.5rem;
 font-weight: bold;
 }
 .trend-positive {
 color: #48BB78;
 }
 .trend-negative {
 color: #F56565;
 }
 </style>
 """, unsafe\_allow\_html=True)

 # File uploader for the test CSV
 uploaded\_file = st.file\_uploader("Upload test CSV file", type=['csv'])

 if uploaded\_file is not None:
 # Save the uploaded file temporarily
 with open('temp\_test.csv', 'wb') as f:
 f.write(uploaded\_file.getvalue())

 # Load data and get predictions
 df = load\_and\_predict\_data('temp\_test.csv')

 # Convert date column to datetime if not already
 df['date'] = pd.to\_datetime(df['date'])

 # Creating filters in a sidebar
 st.sidebar.header("Filters")

 # Time period filter
 time\_periods = {
 'All Time': None,
 'Last Month': 30,
 'Last 3 Months': 90,
 'Last Year': 365
 }
 selected\_period = st.sidebar.selectbox('Select Time Period', list(time\_periods.keys()))

 # Country filter
 countries = ['All'] + sorted(df['country'].unique().tolist())
 selected\_country = st.sidebar.selectbox('Select Country', countries)

 # Store filter
 stores = ['All'] + sorted(df['store'].unique().tolist())
 selected\_store = st.sidebar.selectbox('Select Store', stores)

 # Product filter
 products = ['All'] + sorted(df['product'].unique().tolist())
 selected\_product = st.sidebar.selectbox('Select Product', products)

 # Apply filters
 filtered\_df = df.copy()

 # Apply time filter
 if time\_periods[selected\_period]:
 max\_date = filtered\_df['date'].max()
 cutoff\_date = max\_date - pd.Timedelta(days=time\_periods[selected\_period])
 filtered\_df = filtered\_df[filtered\_df['date'] >= cutoff\_date]

 if selected\_country != 'All':
 filtered\_df = filtered\_df[filtered\_df['country'] == selected\_country]
 if selected\_store != 'All':
 filtered\_df = filtered\_df[filtered\_df['store'] == selected\_store]
 if selected\_product != 'All':
 filtered\_df = filtered\_df[filtered\_df['product'] == selected\_product]

 # Calculate metrics for KPI cards
 total\_sales = filtered\_df['predicted\_sales'].sum()
 avg\_daily\_sales = filtered\_df.groupby('date')['predicted\_sales'].sum().mean()

 # Calculate period-over-period changes
 if time\_periods[selected\_period]:
 previous\_period = filtered\_df['date'].min() - pd.Timedelta(days=time\_periods[selected\_period])
 previous\_df = df[df['date'] >= previous\_period]
 previous\_df = previous\_df[previous\_df['date'] < filtered\_df['date'].min()]

 prev\_total\_sales = previous\_df['predicted\_sales'].sum()
 sales\_change = ((total\_sales - prev\_total\_sales) / prev\_total\_sales \* 100
 if prev\_total\_sales != 0 else 0)
 else:
 sales\_change = 0

 # Create KPI cards using columns
 col1, col2, col3, col4 = st.columns(4)

 with col1:
 st.markdown(f"""
 <div class="metric-card">
 <div class="metric-label">Total Predicted Sales</div>
 <div class="metric-value">${total\_sales:,.0f}</div>
 <div class="{'trend-positive' if sales\_change >= 0 else 'trend-negative'}">
 {sales\_change:+.1f}% vs previous period
 </div>
 </div>
 """, unsafe\_allow\_html=True)

 with col2:
 st.markdown(f"""
 <div class="metric-card">
 <div class="metric-label">Average Daily Sales</div>
 <div class="metric-value">${avg\_daily\_sales:,.0f}</div>
 </div>
 """, unsafe\_allow\_html=True)

 with col3:
 top\_store = (filtered\_df.groupby('store')['predicted\_sales']
 .sum().sort\_values(ascending=False).index[0])
 store\_sales = (filtered\_df.groupby('store')['predicted\_sales']
 .sum().sort\_values(ascending=False).iloc[0])

 st.markdown(f"""
 <div class="metric-card">
 <div class="metric-label">Top Performing Store</div>
 <div class="metric-value">{top\_store}</div>
 <div class="metric-label">${store\_sales:,.0f} in sales</div>
 </div>
 """, unsafe\_allow\_html=True)

 with col4:
 top\_product = (filtered\_df.groupby('product')['predicted\_sales']
 .sum().sort\_values(ascending=False).index[0])
 product\_sales = (filtered\_df.groupby('product')['predicted\_sales']
 .sum().sort\_values(ascending=False).iloc[0])

 st.markdown(f"""
 <div class="metric-card">
 <div class="metric-label">Best Selling Product</div>
 <div class="metric-value">{top\_product}</div>
 <div class="metric-label">${product\_sales:,.0f} in sales</div>
 </div>
 """, unsafe\_allow\_html=True)

 # Group by date and calculate daily total predicted sales
 daily\_sales = filtered\_df.groupby('date')['predicted\_sales'].sum().reset\_index()

 # Create the line chart using Plotly with dark theme
 fig = px.line(
 daily\_sales,
 x='date',
 y='predicted\_sales',
 title='Predicted Daily Sales Over Time'
 )

 # Update layout for dark theme
 fig.update\_layout(
 template="plotly\_dark",
 plot\_bgcolor='rgba(0,0,0,0)',
 paper\_bgcolor='rgba(0,0,0,0)',
 xaxis\_title="Date",
 yaxis\_title="Predicted Sales",
 hovermode='x unified',
 showlegend=True,
 legend=dict(
 orientation="h",
 yanchor="bottom",
 y=1.02,
 xanchor="right",
 x=1
 )
 )

 # Add trend line
 fig.add\_scatter(
 x=daily\_sales['date'],
 y=daily\_sales['predicted\_sales'].rolling(7).mean(),
 name='7-day trend',
 line=dict(dash='dash', color='#48BB78'),
 visible='legendonly'
 )

 # Display the plot
 st.plotly\_chart(fig, use\_container\_width=True)

 # Display detailed data view
 st.subheader("Detailed Data View")
 st.dataframe(
 filtered\_df.sort\_values('date'),
 hide\_index=True
 )

if \_\_name\_\_ == "\_\_main\_\_":
 # Set page configuration at the very beginning
 st.set\_page\_config(
 page\_title="Sales Prediction Dashboard",
 page\_icon="📊",
 layout="wide",
 initial\_sidebar\_state="expanded"
 )
 create\_dashboard()

Above, we define another function named create\_dashboard() where we build a rich, interactive sales prediction dashboard using Streamlit with support for uploading data, filtering, KPI cards, and time series visualizations.

Here we are doing the following

1. **Dashboard Title**

st.title("Sales Prediction Dashboard")

* Sets the main dashboard title at the top.
1. **Custom Dark-Themed KPI Card Styling**

st.markdown("""<style> ... </style>""", unsafe\_allow\_html=True)

* Injects custom CSS to style the KPI cards with a modern dark theme.
1. **File Upload**

uploaded\_file = st.file\_uploader("Upload test CSV file", type=['csv'])

* Allows users to upload a test CSV file containing sales data to be predicted.
1. **Load and Predict**

df = load\_and\_predict\_data('temp\_test.csv')

* Saves the uploaded CSV to a temporary file and uses the load\_and\_predict\_data() function to get predictions from the API.
1. **Sidebar Filters**

st.sidebar.selectbox(...)

* **Time Period Filter**: All Time, Last Month, Last 3 Months, Last Year
* **Country, Store, Product Filters**: Dynamically populated from the dataset
1. **Filter the DataFrame**

filtered\_df = df.copy()
# Apply time, country, store, and product filters

* Applies all selected filters to the data, narrowing down what’s visualized and summarized.
1. **KPI Calculations**

total\_sales = ...
avg\_daily\_sales = ...
sales\_change = ...

* Calculates total predicted sales, average daily sales, and period-over-period % change for comparison.
1. **KPI Card Layout**

col1, col2, col3, col4 = st.columns(4)

* Displays four cards:
	+ **Total Predicted Sales**
	+ **Average Daily Sales**
	+ **Top Performing Store**
	+ **Best Selling Product**
1. **Interactive Time Series Chart**

fig = px.line(...)

* Uses Plotly to show a line chart of daily predicted sales.
* Includes a 7-day rolling trend line (initially hidden).
* Styled for a dark background using plotly\_dark.
1. **Detailed Data View**

st.dataframe(filtered\_df.sort\_values('date'), hide\_index=True)

* Renders the filtered DataFrame in a table format for user inspection.

### Main Entry Point

if \_\_name\_\_ == "\_\_main\_\_":
 st.set\_page\_config(...)
 create\_dashboard()

* Sets the Streamlit app layout and starts the dashboard when the script is run directly.

The create\_dashboard() function integrates: - **Data upload** and **real-time API predictions** - **User-driven filtering** of predictions - **Insightful KPIs** with stylish dark-themed cards - **Interactive time series plots** and a **raw data table**

It’s a modular and user-friendly interface that enables stakeholders to explore predictive insights across time, location, and product dimensions.

Now you can navigate to the terminal, and run streamlit run streamlit\_ui.py to run the streamlit app.

It brings up an option to upload a csv file. We upload our test set csv file should looks like the test\_df below.

test\_df.head()

|  | date | country | store | product |
| --- | --- | --- | --- | --- |
| id |  |  |  |  |
| 230130 | 2017-01-01 | Canada | Discount Stickers | Holographic Goose |
| 230131 | 2017-01-01 | Canada | Discount Stickers | Kaggle |
| 230132 | 2017-01-01 | Canada | Discount Stickers | Kaggle Tiers |
| 230133 | 2017-01-01 | Canada | Discount Stickers | Kerneler |
| 230134 | 2017-01-01 | Canada | Discount Stickers | Kerneler Dark Mode |

Once we upload our file, the data is sent to the batch prediction api and it returns predictions for the number of sticker sold per day for each product type in each country and for each shop as we can see below.

|  |
| --- |
| Sticker sales forecasting dashboard built with streamlit KPI Cards and plot |

|  |
| --- |
| Sticker sales forecasting dashboard built with streamlit with detailed data view |

## Running the Entire Pipeline

Here’s how to run the entire pipeline:

1. Upload data to Modal:

python data\_upload.py data/

1. Train the model on Modal:

python train.py

1. Deploy the API on Modal:

python serve.py

1. Run the Streamlit dashboard:

cd ui
streamlit run streamlit\_ui.py

## Key Benefits of This Approach

1. **Serverless Training and Deployment**: Modal handles all infrastructure, scaling, and container management.
2. **Production-Ready API**: The API is automatically served with proper endpoints, error handling, and authentication.
3. **Separation of Concerns**: Data preprocessing, model training, and serving are cleanly separated.
4. **Interactive Dashboard**: Stakeholders can visualize predictions without technical knowledge.
5. **Reproducibility**: The entire pipeline is defined in code, making it easy to reproduce.

## Improvements

Adding a feature store - I shall be adding a feature store in v2 of this blog. This will help with consistent feature engineering and avoid duplicating the same processing logic in different scripts, etc.

Improving the underlying model - I used a basic XGBoost model for this for purposes of ease and speed. I expect to update this to use an ensemble of gradient boosting machines (XGBoost, CatBoost and LightGBM), stacking, etc so as to get improved model predictions.

Improving latency when filtering in the streamlit ui.

## Conclusion

In this notebook, we’ve built an end-to-end machine learning system for forecasting sticker sales.

We’ve used FastAI for preprocessing, model building, modal to run our training job, deploy and serve our model, and streamlit for visualization.

This approach demonstrates how modern tools can simplify the MLOps process.