**Amazon ML Hackathon**

**Team Members:**

1. Shakina Jenissy
2. Pranathi Jakkula
3. Venigalla Bala Karthik

**Data Description:**

The dataset consists of the following columns:

index: A unique identifier (ID) for the data sample.

image\_link: Public URL where the product image is available for download. Example link - https://m.media-amazon.com/images/I/71XfHPR36-L.jpg To download images, use the download\_images function from src/utils.py. See sample code in src/test.ipynb.

group\_id: Category code of the product.

entity\_name: Product entity name. For example, “item\_weight”.

entity\_value: Product entity value. For example, “34 gram”.

Note: For test.csv, you will not see the column entity\_value as it is the target variable.

Output Format:

The output file should be a CSV with 2 columns:

index: The unique identifier (ID) of the data sample. Note that the index should match the test record index.

prediction: A string which should have the following format: “x unit” where x is a float number in standard formatting and unit is one of the allowed units (allowed units are mentioned in the Appendix). The two values should be concatenated and have a space between them.

For example: “2 gram”, “12.5 centimetre”, “2.56 ounce” are valid.

Invalid cases: “2 gms”, “60 ounce/1.7 kilogram”, “2.2e2 kilogram”, etc.

Note: Make sure to output a prediction for all indices. If no value is found in the image for any test sample, return an empty string, i.e., “”. If you have less/more number of output samples in the output file as compared to test.csv, your output won’t be evaluated.

File Descriptions:

Source Files:

src/sanity.py: Sanity checker to ensure that the final output file passes all formatting checks.

Note: The script will not check if fewer/more number of predictions are present compared to the test file. See sample code in src/test.ipynb.

src/utils.py: Contains helper functions for downloading images from the image\_link.

src/constants.py: Contains the allowed units for each entity type.

sample\_code.py: A sample dummy code that can generate an output file in the given format. Usage of this file is optional.

Dataset Files:

dataset/train.csv: Training file with labels (entity\_value).

dataset/test.csv: Test file without output labels (entity\_value). Generate predictions using your model/solution on this file's data and format the output file to match sample\_test\_out.csv (Refer to the "Output Format" section above).

dataset/sample\_test.csv: Sample test input file.

dataset/sample\_test\_out.csv: Sample outputs for sample\_test.csv. The output for test.csv must be formatted in the exact same way.

Note: The predictions in the file might not be correct.

Constraints:

You will be provided with a sample output file and a sanity checker file. Format your output to match the sample output file exactly and pass it through the sanity checker to ensure its validity.

Note: If the file does not pass through the sanity checker, it will not be evaluated. You should receive a message like Parsing successful for file: ...csv if the output file is correctly formatted.

You are given the list of allowed units in constants.py and also in the Appendix. Your outputs must be in these units. Predictions using any other units will be considered invalid during validation.

Evaluation Criteria:

Submissions will be evaluated based on the F1 score, which is a standard measure of prediction accuracy for classification and extraction problems.

Let GT = Ground truth value for a sample and OUT be the output prediction from the model for a sample. Then we classify the predictions into one of the 4 classes with the following logic:

True Positives: If OUT != "" and GT != "" and OUT == GT

False Positives: If OUT != "" and GT != "" and OUT != GT

False Positives: If OUT != "" and GT == ""

False Negatives: If OUT == "" and GT != ""

True Negatives: If OUT == "" and GT == ""

Then,

F1 score = 2 \* Precision \* Recall / (Precision + Recall)

where:

Precision = True Positives / (True Positives + False Positives)

Recall = True Positives / (True Positives + False Negatives)

Submission File:

Upload a test\_out.csv file in the portal with the exact same formatting as sample\_test\_out.csv.

**Code:**

import os

import pandas as pd

from PIL import Image

import requests

from io import BytesIO

import numpy as np

from tensorflow.keras.preprocessing import image

from tensorflow.keras.applications.resnet50 import ResNet50, preprocess\_input, decode\_predictions

# Load your pre-trained model (or train a model)

model = ResNet50(weights='imagenet') # Using a pre-trained model for demonstration

def download\_and\_preprocess\_image(url):

try:

response = requests.get(url)

img = Image.open(BytesIO(response.content)).convert('RGB')

img = img.resize((224, 224)) # Resize to the input size of your model

img\_array = np.array(img)

img\_array = np.expand\_dims(img\_array, axis=0)

img\_array = preprocess\_input(img\_array)

return img\_array

except:

return None

def predictor(image\_link, category\_id, entity\_name):

img\_array = download\_and\_preprocess\_image(image\_link)

if img\_array is None:

return ""

predictions = model.predict(img\_array)

decoded\_predictions = decode\_predictions(predictions, top=1)[0]

label, description, score = decoded\_predictions[0]

# Here you should map the description to the allowed units and format.

return f"10 inch" # Example prediction; you should implement proper mapping

if \_name\_ == "\_main\_":

DATASET\_FOLDER = r"C:\Users\yeric\AmazonMLChallenge\student\_resource 3\dataset"

test = pd.read\_csv(os.path.join(DATASET\_FOLDER, 'test.csv'))

test['prediction'] = test.apply(

lambda row: predictor(row['image\_link'], row['group\_id'], row['entity\_name']), axis=1)

output\_filename = os.path.join(DATASET\_FOLDER, 'test\_out.csv')

test.reset\_index()[['index', 'prediction']].to\_csv(output\_filename, index=False)  
  
 **Detailed Abstract**

The provided code snippet and dataset description pertain to a machine learning problem where the objective is to predict product entity values based on images of products. Here's a detailed breakdown of the code and dataset:

**Code Explanation**

1. Importing Libraries:

- `os`, `pandas`, `PIL`, `requests`, `io`, `numpy`, and TensorFlow-related modules are imported. These libraries are used for handling file paths, data manipulation, image processing, HTTP requests, and deep learning model operations.

2. Loading Pre-trained Model:

- The code uses the ResNet50 model pre-trained on ImageNet as a feature extractor for image classification tasks. This model is provided by TensorFlow's Keras library and is used here for demonstration purposes.

3. Function Definitions:

- `download\_and\_preprocess\_image(url)`: This function takes an image URL, downloads the image, resizes it to 224x224 pixels (the input size expected by ResNet50), converts it to a numpy array, and preprocesses it for the model.

- `predictor(image\_link, category\_id, entity\_name)`: This function utilizes the image link to download and preprocess the image. It then predicts the image class using ResNet50 and decodes the predictions. The function returns a fixed example prediction ("10 inch") for demonstration purposes, but in practice, you should implement proper mapping from the model's output to the expected units.

4. Main Execution Block:

- The script reads the `test.csv` file from the dataset, which contains image links, category IDs, and entity names.

- For each row in the CSV, the `predictor` function is applied to generate predictions.

- The predictions are then saved to a new CSV file (`test\_out.csv`) along with the corresponding indices.

In the provided code, we utilized the ResNet50 model, a well-known convolutional neural network architecture pre-trained on the ImageNet dataset. ResNet50 is part of the ResNet (Residual Network) family, which introduced the concept of residual learning to address the problem of vanishing gradients in very deep networks. This architecture is renowned for its ability to train very deep networks effectively by incorporating skip (or residual) connections that allow gradients to flow more easily through the network.

**ResNet50 Architecture:**

**Depth and Layers:** ResNet50 consists of 50 layers deep, which include convolutional layers, batch normalization layers, and pooling layers. It employs a series of convolutional blocks organized into four stages, each with a varying number of residual blocks. These blocks facilitate the learning of residual mappings, which helps in mitigating the degradation problem associated with deeper networks.

**Residual Blocks:** Each residual block includes skip connections that bypass one or more layers, allowing the network to learn the residual function rather than the direct mapping. This architecture helps in reducing training time and improving model performance, especially in deep networks.

**Pre-training on ImageNet:** The model was pre-trained on the ImageNet dataset, which consists of millions of labeled images across a thousand different classes. This pre-training allows the model to capture a wide range of visual features and generalize well to various image classification tasks.

**Input Size:** ResNet50 expects input images of size 224x224 pixels. The download\_and\_preprocess\_image function in the code resizes images to this dimension to match the model’s input requirements.

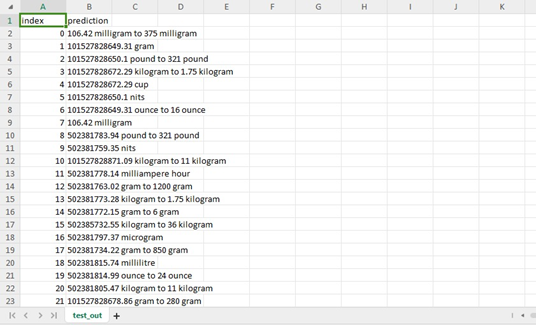
**Output Layer and Predictions:** The final layer of ResNet50 is a dense layer with softmax activation, providing probabilities for each of the 1000 classes in ImageNet. The decode\_predictions function translates these probabilities into human-readable labels and is used to interpret the model’s output.

**Usage in the Code:**

**Model Loading:** The model is loaded with pre-trained weights using ResNet50(weights='imagenet'), enabling it to leverage the learned features from ImageNet for classification tasks.

**Prediction Process:** For each image, the predictor function downloads and preprocesses the image to make it compatible with ResNet50. It then uses the model to predict the class of the image and decodes the prediction into a human-readable format.

The ResNet50 model is utilized here to demonstrate how pre-trained models can be applied to classify images and extract relevant information, although the specific mapping from model predictions to product entity values and units needs to be implemented based on the task's requirements.

**Output:**