## load\_check.ipynb

import os  
import sys  
  
# Add project root  
project\_root = os.path.abspath(os.path.join(os.getcwd(), ".."))  
if project\_root not in sys.path:  
 sys.path.append(project\_root)  
  
from src.utils import load\_raw\_k9\_data  
  
# Define absolute paths  
data\_path = os.path.join(project\_root, "data", "raw", "K9.data")  
tag\_path = os.path.join(project\_root, "data", "raw", "K9.instance.tags")  
  
print("Data path exists:", os.path.exists(data\_path))  
print("Tag path exists:", os.path.exists(tag\_path))

This code initializes the environment by setting up the project root and checking whether the required raw dataset files exist in the expected paths. It ensures that both the .data and .tags files are accessible before proceeding.

import os  
  
raw\_data\_dir = os.path.join(project\_root, "data", "raw")  
print("Raw Data Folder Exists:", os.path.exists(raw\_data\_dir))  
  
# List contents of the raw folder  
print("\nFiles inside raw folder:")  
print(os.listdir(raw\_data\_dir))

This block verifies the existence of the data/raw directory and lists all files inside it. This is useful for confirming that all necessary raw data files are available for loading and inspection.

# Call the function with full paths  
data, tags = load\_raw\_k9\_data(data\_path=data\_path, tag\_path=tag\_path)  
data.head()

This code block uses the load\_raw\_k9\_data utility function to load the mutation feature data and its corresponding tags from the specified files. It then previews the top rows of the dataset for verification.

# How many unique tags?  
print(f"Unique tags: {tags.nunique()}")  
  
# Show top 10 most frequent tags  
print("\n Top 10 tags:")  
print(tags.value\_counts().head(10))

This code provides insights into the diversity of tags in the dataset. It shows how many unique mutation tag combinations exist and displays the 10 most common ones.

# How many mutations are there in each tag?  
tag\_lengths = tags.apply(lambda x: len(x.strip('%').split('\_')))  
print(" Mutation counts per tag:")  
print(tag\_lengths.value\_counts().sort\_index())

This block calculates how many mutations are combined in each tag string by splitting on underscores and shows the distribution of tag lengths.

import os  
import pandas as pd  
  
project\_root = os.path.abspath(os.path.join(os.getcwd(), ".."))  
tag\_path = os.path.join(project\_root, "data", "raw", "K9.instance.tags")  
output\_path = os.path.join(project\_root, "data", "processed", "cleaned\_tags.csv")  
  
tags = []  
  
with open(tag\_path, "r") as f:  
 for line in f:  
 tag = line.strip().strip('%') # Remove newline and %  
 tags.append(tag)  
  
cleaned\_tags\_df = pd.DataFrame({'id': list(range(len(tags))), 'tags': tags})  
cleaned\_tags\_df.to\_csv(output\_path, index=False)  
  
print(f"Cleaned tags saved to {output\_path}")  
print("Preview:")  
print(cleaned\_tags\_df.head())

This code reads the original .tags file, cleans each line by removing % symbols and newlines, and saves it as a CSV with an added id column. This structured format is easier to merge with feature data later.

## data\_cleaning.ipynb

import os  
import pandas as pd  
  
project\_root = os.path.abspath(os.path.join(os.getcwd(), ".."))  
tag\_path = os.path.join(project\_root, "data", "raw", "K9.instance.tags")  
output\_path = os.path.join(project\_root, "data", "processed", "cleaned\_tags.csv")  
  
tags = []  
  
with open(tag\_path, "r") as f:  
 for line in f:  
 tag = line.strip().strip('%')  
 tags.append(tag)  
  
cleaned\_tags\_df = pd.DataFrame({'id': list(range(len(tags))), 'tags': tags})  
cleaned\_tags\_df.to\_csv(output\_path, index=False)  
  
print(f"Cleaned tags saved to {output\_path}")  
print("Preview:")  
print(cleaned\_tags\_df.head())

This is a repeated cleaning process, ensuring a consistent and well-structured tag file is saved in the processed directory.

import pandas as pd  
import os  
  
# Step 1: Locate project root  
project\_root = os.path.abspath(os.path.join(os.getcwd(), ".."))  
  
# Step 2: Define paths  
data\_path = os.path.join(project\_root, "data", "raw", "K9.data")  
tag\_path = os.path.join(project\_root, "data", "processed", "cleaned\_tags.csv")  
  
# Step 3: Load raw feature data (no header in K9.data)  
X = pd.read\_csv(data\_path, header=None)  
  
# Step 4: Load cleaned tag data  
tags\_df = pd.read\_csv(tag\_path)  
  
# Step 5: Add ID and tags to feature data  
X['id'] = tags\_df['id']  
X['tags'] = tags\_df['tags']  
  
# Step 6: Preview  
print("Loaded data shape:", X.shape)  
X.head()

This code loads both the raw mutation data and the cleaned tags file. It appends tag information as new columns to the dataset and previews the combined structure.

# Re-load with better handling of missing values and memory usage  
chunk\_list = []  
for chunk in pd.read\_csv(data\_path, header=None, na\_values='?', low\_memory=False, chunksize=1000):  
 chunk\_list.append(chunk)  
X = pd.concat(chunk\_list, ignore\_index=True)  
  
# Drop the last two columns if they are misaligned  
if X.shape[1] > 5410:  
 X = X.iloc[:, :5410]  
  
# Re-attach id and tags from cleaned\_tags.csv  
X['id'] = tags\_df['id']  
X['tags'] = tags\_df['tags']  
  
print("Cleaned and realigned data shape:", X.shape)  
X.head()

This code improves data loading by using chunked reading, handles missing values (marked by ‘?’), and ensures correct feature alignment. It reattaches the tags and ID after cleaning.

# Count how many tags per instance (split by underscore)  
X['num\_tags'] = X['tags'].apply(lambda x: len(x.split('\_')) if isinstance(x, str) else 0)  
  
# Preview the distribution  
print("Distribution of tag counts per instance:")  
print(X['num\_tags'].value\_counts())  
  
# Flatten the tag list to analyze frequency  
from collections import Counter  
  
# Combine all tags across rows into one flat list  
all\_tags = [tag for row in X['tags'].dropna() for tag in row.split('\_')]  
  
# Count tag frequency  
tag\_counts = Counter(all\_tags)  
  
# Convert to DataFrame for display  
tag\_freq\_df = pd.DataFrame(tag\_counts.items(), columns=['tag', 'count']).sort\_values(by='count', ascending=False)  
  
print("Top 10 most common mutation tags:")  
print(tag\_freq\_df.head(10))

This block calculates the number of mutations per instance, and also flattens all tags into a single list to compute mutation tag frequencies. It helps understand tag diversity and dominance.

import matplotlib.pyplot as plt  
import seaborn as sns  
  
# Plot top 15 tags (clean version)  
plt.figure(figsize=(12, 6))  
sns.barplot(data=tag\_freq\_df.head(15), x='tag', y='count', hue='tag', palette='magma', legend=False)  
plt.title('Top 15 Most Frequent Mutation Tags') # Removed emoji  
plt.xlabel('Mutation Tag')  
plt.ylabel('Count')  
plt.xticks(rotation=45)  
plt.tight\_layout()  
plt.show()

This code visualizes the top 15 mutation tags using a bar chart. The visualization aids in identifying the most frequent genetic mutations in the dataset.

from sklearn.preprocessing import MultiLabelBinarizer  
  
# Step 5.1: Split tags into a list of individual mutations  
X['tag\_list'] = X['tags'].str.split('\_')  
  
# Step 5.2: Initialize binarizer and fit  
mlb = MultiLabelBinarizer()  
Y = mlb.fit\_transform(X['tag\_list'])  
  
# Optional: Store the tag names for later use (e.g., inverse transform)  
tag\_classes = mlb.classes\_  
  
# Display the encoded matrix shape  
print("Encoded label matrix shape:", Y.shape)  
print("Sample encoded vector (row 0):", Y[0])

This block transforms the multi-label tag information into a binary matrix using MultiLabelBinarizer, which is essential for training machine learning models that handle multi-label classification.

import os  
import numpy as np  
  
# Define the required folder path ending  
REQUIRED\_PATH\_SUFFIX = os.path.join("USD-Term1-AppliedAI-GroupSynergy", "data", "processed")  
  
def find\_correct\_processed\_path():  
 current\_path = os.getcwd()  
 while True:  
 # Check if this directory ends with the required suffix  
 candidate = os.path.join(current\_path, REQUIRED\_PATH\_SUFFIX)  
 if os.path.isdir(candidate):  
 return candidate  
 # Move up one directory  
 parent = os.path.dirname(current\_path)  
 if parent == current\_path:  
 raise FileNotFoundError(f"Couldn't find required path ending with '{REQUIRED\_PATH\_SUFFIX}'")  
 current\_path = parent  
  
# Find the correct data/processed path  
processed\_dir = find\_correct\_processed\_path()  
os.makedirs(processed\_dir, exist\_ok=True)  
  
# Clean tag\_list column if exists  
if 'tag\_list' in X.columns:  
 X = X.drop(columns=['tag\_list'])  
  
# Save everything in the correct location  
X.to\_csv(os.path.join(processed\_dir, 'cleaned\_features.csv'), index=False)  
np.save(os.path.join(processed\_dir, 'encoded\_labels.npy'), Y)  
  
with open(os.path.join(processed\_dir, 'tag\_classes.txt'), 'w') as f:  
 f.write('\n'.join(tag\_classes))  
  
print(f"All files saved to: {processed\_dir}")

This block ensures the processed dataset and associated artifacts (features, labels, and tag classes) are saved in the correct directory regardless of where the script is run. It improves portability and downstream consistency.