# Report - IR Assignment 2

## Pre-Processing Images

1. Opened the csv.file using Pandas dataframe.

2. Importing necessary libraries such as PIL (Python Imaging Library) for image manipulation.

3. Image Processing Functions used are the following -

- `adjust_contrast`: Adjusts the contrast of an image.
- `remove_grain`: Removes noise from an image.
- `change_orientation`: Changes the orientation of an image (rotates it by 90 degrees).
- `resize_image`: Resizes an image to a specified size.
- `change_brightness`: Adjusts the brightness of an image.
- `random_flips`: Performs random horizontal and vertical flips on an image.

4. Using Pretrained ResNet Model:
- It imports PyTorch and loads a pretrained ResNet-50 model.
- It removes the classification layer from the ResNet model, leaving only the feature extraction layers.
- For each image:
  - It loads the image, applies transformations, and generates an embedding using the ResNet model.
  - It flattens the embedding tensor and converts it to a numpy array.
  - It **normalizes** the embedding vector.

5. Storing Image Embeddings:
- We have stored the processed images and their embeddings in a dictionary named `store_dict`.
- I serialized the dictionary using pickle and saved it to a file named "Image Embeddings.pkl".

6. Loading Image Embeddings:
- It loads the serialized dictionary from the file "Image Embeddings.pkl" back into the `store_dict`.
- It retrieves the `images_reviews_pairs` data from the dictionary.

Overall, this code downloads images, processes them, extracts embeddings using a pretrained ResNet model, and stores the embeddings along with image paths for later use. It's a pipeline for image feature extraction and storage.

## Relevant Images:

1. Loading Image Embeddings:
   - The code first imports the `pickle` module, which is used for serializing and deserializing Python objects.
   - It opens the file "Image Embeddings.pkl" in binary read mode (`'rb'`) using a `with` statement and assigns it to the file object `f`.
   - Then, it uses `pickle.load()` to deserialize the data from the file object `f` and load it into the variable `store_dict`.
   - From `store_dict`, it retrieves the dictionary containing image embeddings and assigns it to the variable `images_reviews_pairs`.

2. Loading Pretrained ResNet-50 Model:
   - The code imports necessary modules from the `torchvision` library for working with computer vision tasks in PyTorch.
   - It loads a pretrained ResNet-50 model using `models.resnet50(pretrained=True)`. This model has been pre-trained on the ImageNet dataset.
   - After loading the model, it sets the model to evaluation mode by calling `.eval()` method. This is important because it deactivates dropout layers and batch normalization layers during evaluation.
   - Next, it removes the last fully connected layer of the ResNet model. This layer is typically responsible for classification, but in this case, the model is used for feature extraction, so the classification layer is not needed.
   - The modified ResNet model without the classification layer is assigned back to the variable `resnet_model`.

3. Defining Image Transformations:
   - It defines a series of image transformations using `torchvision.transforms.Compose()`. These transformations will be applied to input images before passing them to the ResNet model.
   - The transformations include resizing the image to 224x224 pixels, converting the PIL Image to a PyTorch tensor, and normalizing the image with mean and standard deviation values provided. These mean and standard deviation values are calculated from the ImageNet dataset and are used for normalization.

Overall, this code segment prepares the ResNet-50 model for feature extraction by loading the pretrained model, removing the classification layer, and defining image transformations for preprocessing the input images before feeding them into the model. It also loads the previously stored image embeddings for further processing or analysis.

# Q-2 : Pre- Processing Reviews:

**1. Importing Libraries:**
- `pandas` (imported as `pd`): Used for data manipulation and analysis.
- `numpy` (imported as `np`): Useful for numerical computing.
- `nltk`: Natural Language Toolkit, used for text processing tasks.
- `spacy`: An open-source library for advanced natural language processing tasks.

**2. Loading Dataset:**
- `df = pd.read_csv('A2_Data.csv')`: Loads a dataset from a CSV file named 'A2_Data.csv' into a pandas DataFrame called `df`.

**3. Loading NLTK resources:**
- Downloads necessary NLTK resources using `nltk.download()`.
- Loads the spaCy English language model using `spacy.load()`.

**4. Checking for Missing Values:**
- Drops rows with missing values in the 'Review Text' column using `df.dropna(subset=['Review Text'])`.

**5. Text Preprocessing:**
- Lowercasing:
  - `df['processed_text'] = df['Review Text'].apply(lambda x: x.lower())`: Converts all text in the 'Review Text' column to lowercase.
- Tokenization:
  - `df['processed_text'] = df['processed_text'].apply(lambda x: word_tokenize(x))`: Splits the text into individual words (tokens) using NLTK's word tokenizer.
- Removing Punctuation:
  - `df['processed_text'] = df['processed_text'].apply(lambda x: [word for word in x if word not in punctuation])`: Removes punctuation from the tokenized text using Python's `string.punctuation`.
- Stopword Removal:
  - `df['processed_text'] = df['processed_text'].apply(lambda x: [word for word in x if word not in stop_words])`: Removes stopwords (commonly occurring words like 'the', 'is', 'are') using NLTK's stopwords list.
- Stemming (using NLTK's Porter Stemmer):
  - `df['processed_text'] = df['processed_text'].apply(lambda x: [stemmer.stem(word) for word in x])`: Reduces words to their base or root form using stemming.
- Removing Punctuation Again:
  - `df['processed_text'] = df['processed_text'].apply(lambda x: [word for word in x if word not in punctuation])`: Removes punctuation again after lemmatization.
- Joining Tokens:
  - `df['processed_text'] = df['processed_text'].apply(lambda x: ' '.join(x))`: Joins the processed tokens back into a single string for each row.

**6. Saving Preprocessed Data:**
- `df.to_csv('preprocessed_data.csv', index=False)`: Saves the preprocessed DataFrame into a new CSV file named 'preprocessed_data.csv' without including the index column.

Overall, this code prepares text data for further analysis or modeling by converting it into a cleaner and more structured format through various preprocessing steps.

## Extract Relevant Images

**1. Imports:**
- `import pandas as pd`: Imports the pandas library, which provides data structures and data analysis tools.
- `import numpy as np`: Imports the numpy library, which provides support for numerical operations.
- `from collections import Counter`: Imports the Counter class from the collections module, which is used for counting hashable objects.
- `import math`: Imports the math module, which provides mathematical functions.

**2. Load Preprocessed Data:**
- Loads preprocessed data from a CSV file named 'preprocessed_data.csv' into a pandas DataFrame called `df`.

**3. Tokenization:**
- Splits the preprocessed text in the DataFrame column 'processed_text' into tokens (words) and stores them in a new column named 'tokens'.

**4. TF Calculation:**
- Defines a function `calculate_tf(tokens)` to calculate the term frequency (TF) for a list of tokens.
- Iterates over the tokens in a document, counts the occurrences of each word, and calculates its TF by dividing its count by the total number of tokens in the document.

**5. IDF Calculation:**
- Defines a function `calculate_idf(documents)` to calculate the inverse document frequency (IDF) for a collection of documents.
- Counts the number of documents containing each word and calculates its IDF using the formula `IDF = log(N / n)`, where `N` is the total number of documents and `n` is the number of documents containing the word.

**6. TF-IDF Calculation:**
- Defines a function `calculate_tfidf(tf, idf)` to calculate TF-IDF scores for each word in a document by multiplying its TF by its IDF.

**7. TF-IDF Calculation for Each Document:**
- Iterates over each document in the DataFrame, calculates its TF-IDF scores using the TF and IDF functions, and appends the results to a list called `tfidf_scores`.

**8. Conversion to DataFrame:**
- Converts the list of TF-IDF scores (`tfidf_scores`) into a pandas DataFrame called `tfidf_df`.

**9. Handling NaN Values:**
- Fills NaN values in the DataFrame with 0. NaN values occur for words that are not present in a document.

**10. Concatenation:**
- Concatenates the TF-IDF scores DataFrame (`tfidf_df`) with the original DataFrame (`df`) along the columns axis.

**11. Saving or Using the Result:**
- Saves the final DataFrame containing TF-IDF scores to a CSV file named 'tfidf_scores.csv' without including the index.

Overall, this code preprocesses text data, calculates TF-IDF scores for each document, and combines the results with the original data for further analysis or storage.

## Q-3: Extraction Using Images and Reviews

1. Image Embedding Functions:
- `get_embedding(image)`: Takes an image as input, applies transformations, generates its embedding using a pre-trained ResNet50 model, and returns the embedding as a numpy array.
- `get_similar_image(image_embedding)`: Computes the cosine similarity between the input image embedding and embeddings of all images in the dataset. Returns a dictionary where keys represent the indices of images, and values represent their cosine similarity with the input image.

2. User Input:
- Prompts the user to input an image URL and a review text.
- Downloads the image from the URL and retrieves its embedding.
- Computes the similarity between the input image embedding and all images in the dataset.
- Identifies the top 3 most similar images and prints their URLs, associated reviews, and cosine similarity scores.

3. Text Preprocessing:
- Loads a CSV file containing TF-IDF scores for text reviews into a pandas DataFrame.
- Defines a function `preprocess_input(input_review)` to preprocess the input review text:
  - Converts the text to lowercase.

- Tokenizes the text using NLTK.
- Removes punctuation and stop words.
- Stems words using NLTK's Porter Stemmer.
- Lemmatizes words using spaCy.
- Joins the processed tokens back into text.
- Preprocesses the input review text using the defined function.

4. TF-IDF Calculation:
- Defines a function `calculate_tf(tokens)` to calculate term frequency (TF) for a list of tokens.
- Splits the preprocessed input review text into tokens and calculates its TF.
- Loads TF-IDF scores for all reviews from the DataFrame and computes cosine similarity between the input review and all reviews in the dataset.
- Sorts the similarities and retrieves the indices of the top 3 most similar reviews.

5. Output:
- Prints the top 3 most similar reviews along with their cosine similarity scores.

Overall, this code allows users to input an image and a review text, calculates their embeddings, and finds the most similar images and text reviews based on cosine similarity.

## Q4 Composite scores:

1. Function Definition:
- `def comp_score(ind_store, ret_method)`: Takes two arguments: `ind_store`, which is a set of indices representing either image or review similarity scores, and `ret_method`, a string indicating whether the comparison is for image retrieval or text retrieval.

2. Initialization:
- Prints a header indicating the retrieval method being used.
- Initializes an empty list `cosine` to store composite similarity scores.
- Initializes variables `img_avg` and `rev_avg` to calculate average cosine similarity scores for images and reviews, respectively.

3. Composite Similarity Calculation:
- Iterates over the indices in `ind_store` (representing either images or reviews).
- For each index, retrieves the review text and its cosine similarity score.
- Retrieves the associated image URLs and their cosine similarity scores.
- Calculates the composite similarity score as the average of the review and image cosine similarity scores.
- Updates `img_avg` and `rev_avg` with the cosine similarity scores of images and reviews, respectively.
- Appends the composite score, review text, image URLs, and individual similarity scores to the `cosine` list.

4. Sorting and Printing:
   - Sorts the `cosine` list based on composite similarity scores in descending order.
   - Prints each entry in the sorted list, displaying the composite score along with the associated image and review information.

5. Average Similarity Scores:
   - Calculates the average cosine similarity scores for images (`img_avg`) and reviews (`rev_avg`).
   - Prints the average scores.

6. Comparison:
   - Compare the average similarity scores of images and reviews.
   - Prints a conclusion based on whether choosing images or reviews provides better similarity scores.

7. Function Invocation:
   - Calls `comp_score()` twice, once for image retrieval (`img_store`) and once for text retrieval (`rev_store`), specifying the retrieval method in each case.

Overall, this code allows for a comparison of the similarity scores between images and text reviews, providing insights into which type of content (images or text) yields better similarity results based on the provided cosine similarity scores.

## Results

**USING IMAGE RETRIEVAL**
Review Number: 758
Image URLs: ['https://images-na.ssl-images-amazon.com/images/I/71bztfqdg+L._SY88.jpg']
Review: I have been using Fender locking tuners for about five years on various strats and teles. Definitely helps with tuning stability and way faster to restring if there is a break.
Image Score: 0.9846782684326172
Review Score: 0.9539979727109786
Composite Score: 0.9693381205717979

Review Number: 655
Image URLs: ['https://images-na.ssl-images-amazon.com/images/I/719-SDMiOoL._SY88.jpg']
Review: These locking tuners look great and keep tune.  Good quality materials and construction.  Excellent upgrade to any guitar.  I had to drill additions holes for installation.  If your neck already comes with pre-drilled holes, then they should drop right in, otherwise you will need to buy a guitar tuner pin drill jig, also available from Amazon.
Image Score: 0.842109739780426
Review Score: 0.14237474642246664

Composite Score: 0.4922422431014463

Review Number: 253
Image URLs: ['https://images-na.ssl-images-amazon.com/images/I/71IleN3UCjL._SY88.jpg', 'https://images-na.ssl-images-amazon.com/images/I/71iM1a+NXBL._SY88.jpg', 'https://images-na.ssl-images-amazon.com/images/I/71dOXPF0p1L._SY88.jpg']
Review: <input type="hidden" name="" value="https://images-na.ssl-images-amazon.com/images/I/D1BiM3frkvS.mp4" class="video-url"><input type="hidden" name="" value="https://images-na.ssl-images-amazon.com/images/I/A1m2Qo-1z%2BS.png" class="video-slate-img-url"> My first Yamaha and I am loving it. Becoming my go to guitar.

This thing had various tonal choices due to pickups and tone pots, plus the neck feel good. Works great with those 60s and 70s Rock & Roll style tunes. Also do a search on YouTube for my unboxing and demo, look up title is:  Yamaha RS 720B Revstar guitar.
Image Score: 0.7838665644327799
Review Score: 0.031712325860091976
Composite Score: 0.40778944514643595

USING REVIEW RETRIEVAL
Review Number: 758
Image URLs: ['https://images-na.ssl-images-amazon.com/images/I/71bztfqdg+L._SY88.jpg']
Review: I have been using Fender locking tuners for about five years on various strats and teles. Definitely helps with tuning stability and way faster to restring if there is a break.
Image Score: 0.9846782684326172
Review Score: 0.9539979727109786
Composite Score: 0.9693381205717979

Review Number: 622
Image URLs: ['https://images-na.ssl-images-amazon.com/images/I/61DvLcapd8L._SY88.jpg']
Review: I went from fender chrome non-locking to fender gold locking. It made my guitar look beautiful and play beautiful. I think locking tuners are the way to go. If you are new to locking tuners look on YouTube for instructions.
Image Score: 0.6966647505760193
Review Score: 0.30326444135269737
Composite Score: 0.4999645959643583

Review Number: 938
Image URLs: ['https://images-na.ssl-images-amazon.com/images/I/61hEWhycZ9L._SY88.jpg']
Review: Nice tuners.  Installed on a strat neck and they are working great. Nice and smooth and has stayed in tune very well. Nothing wrong with these.
Image Score: 0.7675447463989258
Review Score: 0.2010589301950042
Composite Score: 0.484301838296965

## Analysis

**Avg Image cosine similarity score:** 0.6789599428089653
**Avg Review cosine similarity score:** 0.023050059026232403

By comparing the average of the similarity scores of the Image to that of the reviews we can say that:
Choosing images gives a better similarity score in this case.

## How are multiple images for a review handled?

Since one review may have multiple images-

- Separate pairs for all the images and reviews were created.
- If two images for the same review were ranked in top 3, then they were taken out of the rankings.
- After finding the best image, review pair, all of the images for a particular review was returned, with cosine similarities for all the images with the input image.
- For compositie similarity score, the similarity scores for all the images of a review were averaged and then averaged with review similarity score.
- **All the images for a review found relevant were returned for reference, even thought the (image, review) pairs were evaluating individually for relevance.**

## Challenges:

While the code provided lays out a framework for calculating cosine similarity between images and text reviews and ranking them based on relevance, there are several challenges one might face when working on such a project. Here are some potential challenges:

1. Data Quality:
   - The quality of image and text data can significantly impact the accuracy of cosine similarity calculations. Noisy or irrelevant data may lead to misleading similarity scores and rankings.

2. Scalability:
   - Processing a large volume of images and text reviews efficiently can be computationally intensive, especially when calculating cosine similarity scores for all possible pairs. Optimizing

the code for performance and considering scalable solutions (e.g., distributed computing) may be necessary. Utilizing parallelization becomes necessary in such cases.

4. Dimensionality:
   - High-dimensional feature representations, especially in the case of images, can lead to the curse of dimensionality. Dealing with high-dimensional data requires careful feature selection, dimensionality reduction techniques, or using specialized algorithms designed for high-dimensional spaces.

5. Evaluation Metrics:
   - Assessing the effectiveness of the relevance ranking system requires appropriate evaluation metrics. Simply relying on cosine similarity scores may not capture the true relevance of image-text pairs. Incorporating user feedback, human evaluation, or domain-specific metrics can provide more meaningful insights.

6. Semantic Gap:
   - The semantic gap between the low-level features extracted from images/text and the high-level concepts or meanings they represent can pose challenges. Ensuring that similarity calculations capture semantic relevance accurately requires bridging this semantic gap, which may involve advanced techniques like semantic embedding models.

7. Privacy and Ethics:
   - Handling sensitive image or text data requires adherence to privacy regulations and ethical considerations. Ensuring data anonymization, obtaining appropriate consent, and implementing robust security measures are essential for protecting user privacy and maintaining trust.

## Improvements:

While the provided code offers a foundation for retrieving pairs of images and reviews ranked by relevance using cosine similarity, there are several potential challenges and areas for improvement in the retrieval process:

1. Feature Representation:
   -Utilizing more advanced features: While the provided code uses pre-trained ResNet-50 embeddings for images and TF-IDF scores for text, exploring other feature representations such as deep learning-based embeddings for text (e.g., Word2Vec, GloVe) or fine-tuned image embeddings (for example, fine tuning for music instruments in this case) might capture richer semantics and improve retrieval accuracy.
   - Fusion of multimodal features: Integrating both image and text features into a single multimodal embedding space using techniques like late fusion or early fusion can capture correlations between different modalities more effectively.

2. Relevance Ranking:
   - Considering diverse similarity measures: Besides cosine similarity, exploring other similarity measures such as Euclidean distance, Jaccard similarity, or semantic similarity metrics might offer different perspectives on relevance and improve ranking accuracy.

3. Data Preprocessing and Normalization:
   - Standardizing text preprocessing: Ensuring consistent preprocessing steps (e.g., lowercasing, tokenization, stop word removal, stemming/lemmatization) across all text data can mitigate variations and improve the quality of similarity computations.
   - Normalizing feature vectors: Normalizing image and text embeddings to have unit norms can enhance the comparability of different vectors and mitigate the influence of vector magnitudes on similarity scores.

4. Evaluation and Validation:
   - Conducting thorough evaluation: Designing appropriate evaluation metrics (e.g., precision, recall, F1-score, Mean Average Precision) and performing systematic evaluations using held-out datasets or cross-validation techniques are crucial for assessing the effectiveness of the retrieval system and identifying areas for improvement.
   - Addressing biases and fairness: Ensuring the retrieval system is fair and unbiased across different demographic groups by monitoring and mitigating biases in the data and algorithms is essential for providing equitable results.

By addressing these challenges and implementing potential improvements, one can enhance the effectiveness, efficiency, and user satisfaction of the image-text retrieval process.