Watch image classification using CNN and KNN

Changanaqui Julian, Bourleau Marc, Khairallah Charlène, Jesudasan Lawrence

Master in Management – Business Analytics Orientation
University of Lausanne
Lausanne, Switzerland
julian.changanaqui@unil.ch, marc.bourleau@unil.ch, charlene.khairallah@unil.ch, lawrence.jesudasan@unil.ch

Abstract – **The objective of this project is to assist customers of Swiss luxury watchmaking brands in selecting their desired timepiece. The primary aim is to alleviate the difficulties associated with making an informed purchase due to the overwhelming amount of available information. The motivation for this project arises from the inadequacy of existing online watch databases, such as *Watchbase*, in fully addressing this issue. To effectively tackle the problem, we have devised a methodology. Our approach involves creating a user-friendly interface that offers watch recommendations based on user-inputted images. We utilise two classification image algorithms, namely Convolutional Neural Networks (CNN) and K-nearest Neighbors (KNN), to generate these recommendations.**
This methodology provides a robust and comprehensive solution to the challenge of assisting customers in selecting their ideal luxury watch.
Index terms – **luxury Swiss watches, web scraping, KNN, CNN, Gradio interface, PyTorch, image classification.**

I. Introduction

Since the inception of the first wristwatch by Patek Philippe in 1868, Swiss luxury watches have witnessed remarkable growth and global popularity. Despite the challenges posed by the recession caused by Covid-19 and inflationary pressures, the global luxury watch market demonstrated significant expansion between 2022 and 2023. The market size escalated from 30.58 billion in 2022 to 34.17 billion in 2023, indicating a compound annual growth rate (CAGR) of 11.7%.

According to the Deloitte Swiss watch industry study, it is anticipated that the proportion of watches purchased online will rise to 30% by 2030, a notable increase from the current approximate level of half that. This surge in e-commerce sales can be attributed to the preference of nearly half of the Millennial and Gen Z population for online shopping over in-store experiences. Additionally, a Bain & Co report predicts that by 2030, younger generations (millennials, Z, and Alpha) will account for 80% of luxury purchases.

As the wealth of information on the internet continues to expand, the task of finding the perfect timepiece has become increasingly challenging for both new and experienced consumers. With an increasing amount of purchases made in online channels, the absence of a centralised and comprehensive database that consolidates information on all watches adds to the complexity of the process. It necessitates individuals to navigate through the websites of numerous watch brands, resulting in a time-consuming and laborious endeavour.

Therefore, as the luxury watch market continues to expand, the development of an interactive interface that facilitates informed purchasing decisions holds significant importance. Such a dashboard would provide consumers with a user-friendly interface with recommended watches.

II. Research question

A. Problem

A current problem is that luxury watch consumers are facing difficulties to find their dream wristwatch. The luxury watch market is characterised by a wide range of brands, each offering their distinctive styles, designs, complications, and craftsmanship. The lack of a centralised resource that aggregates and organises this diverse range of watches makes it challenging for consumers to compare and evaluate different options effectively. They may need to visit multiple sources, navigate various websites, and rely on fragmented information, making the decision-making process more complex and daunting.

Currently, Watchbase stands as one of the few websites that attempts to tackle the aforementioned challenge by providing a centralised platform that consolidates various luxury watch brands. However, it is important to note that while Watchbase addresses the issue of navigating through individual brand websites, it does not fully resolve the core problem at hand. This is primarily due to the lack of a easy-to-use interface that helps consumers find their dream watch using recommendations. Thus, although Watchbase serves as a resource for accessing information on luxury watch brands in a centralised manner, it falls short in terms of offering an intuitive and streamlined solution that effectively addresses the complexities associated with purchasing luxury watches.

B. Objective

Our main goal is to fully address the problem by providing consumers with a comprehensive platform of watches. However, the development of this platform requires the construction of a database of the watch industry. In order to do that, we have scraped the Watchbase website using the available package on Python called Selenium. Then, in order to build the interface, we have used the package Gradio.

C. Scope

In contrast to the Watchbase database, our approach involved selecting a specific number of brands to scrap based on their market share. We were successful in gathering 2065 images along with their accompanying text characteristics, such as price, material, type of glass, diameter, and more. However, there were two primary reasons that prevented us from scraping the entire website. The first reason relates to time constraints. The process of scraping all the watches would have necessitated an extensive loading time, potentially spanning multiple days. Given our project's limitations and time restrictions, it was not feasible to allocate such a significant amount of time to the scraping process. The second reason involves mitigating bias in our data collection. To ensure a fair and unbiased representation, we tried to manually select brands that had a comparable number of watches. This approach would have helped to avoid any potential skewing of the dataset and ensured a more balanced representation across different brands. But, unfortunately, we haven’t been able to have a balanced number of images in each brand.

III. Methodology

A. Database

The core of our project revolves around constructing a database. We will gather images and textual data for each of our watches using a reference website (https://watchbase.com/)). To ensure variety in our data, we have decided to include watches from 20 brands, totaling 2000 watches (approximately 100 watches per brand).

B. Scraping

To retrieve the required data for our database, we utilised the Selenium library for web scraping. This library allows us to target dynamic data on websites, making automation easier. We divided the scraping process into two parts. The first program downloads all the required images, while the second program scraps the associated textual data.

C. Data Cleaning / Mapping

Using mapping operations, we merge all our data together. Since including images directly in a CSV is not practical, we only keep their references in a new variable. The images are stored in a separate folder, and we can retrieve them directly using the references in our data frame. The end result is a data base that contains the desired variables and observations.

D. Image Classification Models

The main objective of our research is to develop accurate image classification algorithms for luxury watches. To achieve this, we utilized two different methods: Convolutional Neural Networks (CNN) and K Nearest Neighbors (KNN).

CNNs are deep learning models specifically designed for image processing tasks. We utilized CNNs to learn and extract meaningful features from the luxury watch images. Transfer learning techniques were employed to leverage pre-trained CNN models such as VGG16, which have been trained on large-scale image datasets. Fine-tuning and adjusting the network architecture were performed to adapt the models to our specific classification task.

We implemented the KNN algorithm, a simple yet effective machine learning method for classification tasks. KNN makes predictions based on the similarity between a given image and its k nearest neighbors in the feature space. By training the KNN model on the preprocessed dataset, we aimed to classify images based on their similarity and identify similar watches. To train this model, we used the standardized images mentioned before and had to extract certain features from these images. These features are grayscale, color histograms and texture features. We needed to extract these features for each image using a package in Python called scikit-image. We then had to train our models using our extracted features and the corresponding labels from the dataset. Finally, we used the trained model to classify new images. Given a reference as an input, the model would return three similar images corresponding to the 3 nearest neighbors from KNN.

E. Visual Interface : Gradio

Once we have trained our models and tested them, we wanted to have an interface where the user could enter an image or the path of his watch. our algorithm would then run and propose three resembling watches. We propose two options in the inputs : the user can either choose to have similar watches from any brand available or from a specific brand. In the case, where any brand is chosen, the KNN algorithm would be working and return the 3 nearest neighbors. In the case of a specific brand, we will use CNN that predicts the brand of a specific image path and then allow KNN to return the 3 nearest neighbors of that specific brand coming from CNN.

To do that, we turned to an open-source Python package called Gradio. This package enabled us to quickly create customizable UI components for our model. Gradio is easy-to-use and well known for visualization of Machine Learning projects. With it, we can create buttons and scrollbars, enter text or inout images. That is what we did. The user is able to enter his image's path and use the scrollbar to choose between the two following options : "Any brand" and "Same brand". After entering his choice, the program will run and return three images according to the results of the model.

IV. Implementation

The project is hosted in a Github repository, which contains all the required files and instructions on a step by step basis. Please refer to the README for the adequate installation of the environment and packages required for the proper replication of the platform. The program runs in Python, version 3.9. and is structured using the following packages:

- Tkinter - version 8.5.9
- Numpy - version 1.24.3
- Pandas - version 2.0.1
- Matplotlib - version 3.7.1
- Selenium - version 4.9.1
- Tqdm - version 4.65.0
- Pillow - version 9.5.0
- Torch - version 2.0.1
- Torchvision - version 0.15.2
- Sklearn - version 0.0.post5
- Gradio - version 3.33.1

A. Program implementation

The platform code has a modular structure presented as follows (we only display the most important executable files for diagramming reasons):

```
ADA_Project/
├── README.md
├── prediction/
├── testing/
├── training/
├── watch_grading_knn.xlsm
├── watches_images/
├── watches_images_no_category/
├── watches_images_processed/
├── 1_images_scraping.py
├── 2a_text_scraping.py
├── 2b_cleaned_watch_text_def.csv
├── 3a_mapping.py
├── 3b_data_with_images.csv
├── 4a_gray_color_texture.py
├── 4b_merged_df.csv
├── 5a_df_all_standardized.py
├── 5b_merged_df_scaled.csv
├── 6a_cnn_training.py
├── 6b_best_checkpoint.model
├── 6c_cnn_inference.py
├── 7_knn.py
└── 8_gradio.py
```

1. Images_Scraping.py: code for scraping watches images.
2a. text_scraping.py: code for scraping watches characteristics.
2b. cleaned_watch_text_def.csv: database with all images characteristics.
3a. mapping.py: code for linking the watches images with its respective text characteristics.
4a. gray_color_texture.py: code for the extraction of image features.
4b. merged_df: database where all the three features are stored, per reference.
5a. df_all_standardised.py: code to standardise all the features stored in the previous dataset
5b. merged_df_scaled.csv: database containing all the features, scaled
6a. cnn_training.py: code for training the CNN.
6b. best_checkpoint.model: CNN model with highest prediction accuracy.
6c. cnn_inference.py: code for testing the CNN on new unlabelled watches.
7. Knn.py: code for the implementation of the KNN model and the testing of three references.
8. gradio.py: code for creating the interface.

B. Images

For our scraping purposes, we use the Selenium library, which allows us to scrape dynamic websites. We start by defining a vector with the names of the 20 brands we are incorporating into our database. The number of images we are planning to download in each category is stored in the variable *images_per_category*, which we have set to 55. After trying various values for this variable, we decided to use 55 because it allows us to have around 2000 images in the end. We've experienced a significant number of image losses, either because the image didn't download correctly or because the program didn't go through every watch.

Using the WebDriver from Selenium, we are able to connect to the website. By examining its structure (cf: figure 1), we see that categories are divided into subparts on each brand's page. Using CSS selectors, we are able to select the common HTML identifier for each category (.family-box.row .col-md-6 h2.title a). We chose to go through 5 categories for each watch brand, which will provide more variety in our database.

We create a for loop that navigates to each category page. We artificially scroll down on each page to trigger image loading and add a wait statement that waits for each image to load properly. We then save the image URLs and store them in a variable called *image_urls*. We replace the prefix 'md' with 'lg' in each link to have higher quality images. Within each category, we search for the <img> tag.

Once we have all the URLs of all the watches, we create a new for loop to save our observations. The end result is a folder named 'Watches_Images' that contains subfolders for each brand and the corresponding reference name for all watches. We

have a total of 2065 images. Unfortunately, not all brands have the same number of watches. For example, Rolex has 114 observations, whereas Audemars Piguet has 68.

Still overall, we were able to obtain approximately 100 images for most of the watches. Please note that these values may vary depending on different factors when running our code, such as the website structure (which can change), image formats, page loading speed, and others.

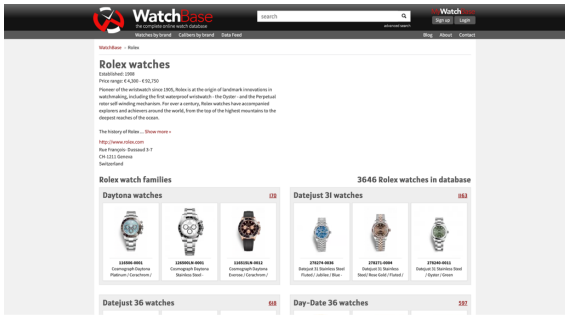

Figure 1: The Rolex page. Sub categories are contained inside separate blocks (Source: https://watchbase.com/rolex)

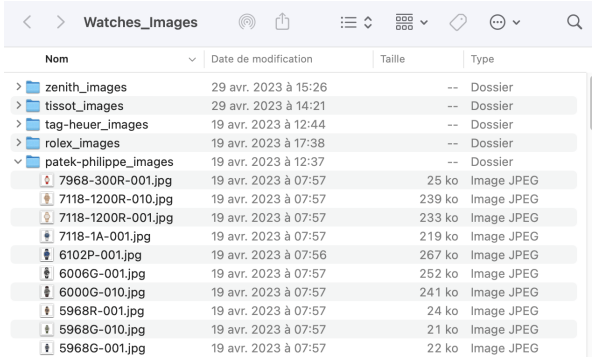

Figure 2: A view of the resulting folder Watches_Images

C. Watch characteristics

To extract the text characteristics of each watch, we employed a systematic scraping process similar to the one used for image scraping. We initiated by creating an empty dataframe called "watch_data" that encompasses columns representing the desired text characteristics, such as materials, price, diameter, and others. Subsequently, we iterated through each brand's webpage, extracting the links for the first five categories on the page. Within this iteration, we nested another loop that iterated through each category and extracted the corresponding text information. To achieve this, we identified three distinct CSS selectors (labelled as "1", "2", and "3" in Figure 2) to locate and retrieve the desired text data. The first selector, "info-table", was used to identify the relevant table containing the name, category and reference of the watch. The second selector, "col-xs-6", enabled the extraction of specific attributes within the table. Lastly, the selector "#pricechart" was employed to locate the URL associated with the price attributes.

To successfully scrape the price, we employed the URL obtained from "#pricechart" and opened it in a new window. Through inspection of the HTML source, we identified the CSS selector "#\/datasets\/0\/data\/1 > td:nth-child(2) > span:nth-child(1) > span:nth-child(1)" to target the price element. Additionally, we incorporated an exception handling mechanism, specifically a NoSuchElementException, to prevent the loop from stopping when a watch did not have a price. Any NaN values in the Price column were subsequently replaced with "NA" to denote missing data.

Throughout the scraping process, we encountered two inconsistencies that required our attention. Firstly, the watch references provided on the website occasionally included an additional AKA (“Also Known As”) following the main reference number.

Secondly, the variable representing the material of the watches exhibited inconsistency in its naming convention on the website. In some instances, the material was listed in plural form, while in others, it was singular. To ensure consistency and avoid data duplication, we incorporated both variations in the dataframe and combined them into a unified column named "Materials."

Finally, we saved the resulting database as "watch_text_.csv," ensuring that the extracted text characteristics were organised and readily accessible for further analysis and utilisation.

Figure 3: Cosmograph Daytona Platinum watch. (Source: https://watchbase.com/rolexdaytona/116506-0001)

D. Data Base

In our final database, "data_with_images," we have a total of 2062 observations. Initially, we collected the text characteristics on 3937 watches. However, during the image scraping process, we encountered an issue where some of the images were found to be black or empty. We used a mapping dictionary, linking the watch reference to its corresponding characteristics, to associate the characteristics with the watches. This mapping automatically dropped the watches from our database for which we had the text characteristics but did not have the respective image. As a result, the final "data_with_images" database contains only the watches for which we successfully obtained both the text characteristics and the corresponding image.

During the mapping process, we encountered a problem where the inclusion of the "price" variable in our final database resulted in a significant reduction in the number of observations from 2062 to 1449. It was observed that at least 20% of the watches did not have a price value available. In order to prioritize a larger number of observations, it was decided to exclude the "price" variable from the "data_with_images" dataset.

Furthermore, during the mapping process, we made some modifications to the "Reference" variable. Specifically, we removed any additional alternative names (AKA) and selected only the variables that would be useful for constructing the interface. These selected variables include: "Reference", "Limited", "Glass", "Shape", "Diameter", and most importantly, "image_file". The "image_file" variable will serve as a crucial link between the text characteristic filters and the corresponding image to be displayed as the output.

E. Convolutional Neural Network (CNN)

After assembling the database, the next step is to apply image classification algorithms, starting with a Convolutional Neural Network (CNN). The CNN is created in two phases. Firstly, we have created "*7a_cnn_training.py*", in which we have trained many models. We have started by defining the network architecture and implementing a transformer function using PyTorch. This function preprocesses the images and converts them into PyTorch tensors. We have created three folders: "testing," "training," and "prediction". The training folder contains 70% of the images and it's used to train the CNN model to learn and extract meaningful features from the data. Each of the remaining folders contain 15% of the remaining images. The training and testing folders contain labelled categories, while the prediction folder is used to evaluate the model's performance on new, unlabeled data.

Secondly, to identify the best CNN model, we conducted a hyperparameter tuning process. The hyperparameters considered were the number of layers, batch size, and number of epochs. The models were compared based on two metrics: testing accuracy and, more importantly, prediction accuracy, as the primary objective is to accurately predict the brand of the watches. The prediction accuracy has been computed in the program: "*7c_cnn_inference.py*". This program has been used to test all the model's ability to predict unlabelled data by computing its accuracy to predict a watches brand based on the image. Our first model had a precision accuracy of 46% but by tweaking the hyperparameters, we have been able to increase it to 77%. This means that almost 8 out of 10 times, our convolutional network model will be able to correctly classify the watch brand based on its image.

The best model chosen for this project exhibits the following characteristics: a batch size of 32, 15 epochs, and 2 convolutional layers. The testing

accuracy achieved was 82%, and the prediction accuracy was 77%.

F. K-Nearest Neighbors (KNN)

In this part, we will detail the implementation of our KNN model. In order to use the KNN model, we had to retrieve features that the algorithm could use to classify the images and return similar ones. Those features are well known in the case of image recognition : grayscale, colour histograms and texture features. Of course, many other features are available but we decided to focus on those three. Grayscale is often used in image feature extraction in order to retrieve several shades of grey, usually ranging from black to white. Unlike colourful images that use colour information as features, grayscale will only represent the intensity values of an image.

We also used colour histograms to extract colour features of an image, as many of our watches' images show different colours in their design. As a last feature, we extracted the texture information from our images. Texture refers to the visual patterns and surface characteristics present in an image. Here we capture and quantify these patterns to extract meaningful information from the images.

First we created three Pandas dataframe, each of them comprising two columns. The first column lists the filename and the second one gives the features values. We have three dataset, one for each feature. The idea is to extract feature values from the images contained in the “watches_images_processed” folder. The code will use a for loop to go through each image that is processed and load it. It will then convert the image into grayscale by applying a function called *convert* from the PIL package, and compute the grayscale histogram features, normalising them so that the values sum to 1. We do the same for the colour features by computing the colour histogram features and normalising them as well. Concerning the texture features, we used two functions from the skimage.features package called *graycomatrix* and *graycoprops*. Then, we had to reshape the feature vectors to 1-dimensional arrays and concatenate them together. The next part was to merge all the features into one dataset that we called “5b_merged_df.csv”.

After that, we had to standardise the features so that all the features may have a similar range in their values. For that, we first extract features into separate arrays. Then we create a *StandardScaler* object that will enable us to scale the individual feature arrays. The last part here is to update the scaled values into our data frame.

Now we are able to train the KNN model, having all the information we need. We start by extracting the gray, color and texture features and concatenate the features together into a matrix. We will then have three variables, that we call “X_gray”, “X_color” and “X_texture”. The next step is to concatenate them into a single matrix and create the target vector for our model. This vector would be the reference names of our watches. After that, we split the data into training (80%) and testing (20%) sets and we can initialize the KNN classifier. Then, we just have to train the model using all features. To test the model, we decided to take 50 random watches from our images folder and computed the 3 nearest neighbors for each of them. As we explained earlier in this report, we used a subjective grading method to evaluate the performance of the model. For each of the 50 watches to test, we graded each of the neighbours with a scaling going from 1 to 5.

G. Gradio

The final step of our project consists of combining our 2 models and implementing them into an user interface. We want to let the user choose a watch image of his choice, and our algorithms will output recommendations based on its visual characteristics. With our ability to predict the watch brand with our CNN model, we will implement a dropdown button which will let the user indicate if he wants results from the same brand as his image, or any other brand. This choice will then guide the behaviour of our KNN model, which will define its outputs accordingly.

To create our interface, we decided to use the Gradio package. It is efficient and easy to use in accordance with machine learning models. The structure we implement has the following characteristics:

On the left panel:

- A text box where the user paste the absolute path of his image
- A dropdown button indicating « same brand » or « any brand »

On the right panel:
- A textbox showing the references of the watches
- The 3 watches images displayed

We start by defining our main function « image analysis ». This function will perform all the operations of both CNN and KNN discussed prior. We extract the relevant features of the input image, we add them to our existing database, and we use standardisation techniques for our data processing. Then, we run our « best model » from CNN. If the user chooses « same brand », then we limit the scope of our database to this specific brand when performing the KNN algorithm. Otherwise, any brand can be chosen from our database. An example of the user interface can be seen below, in figure 4.

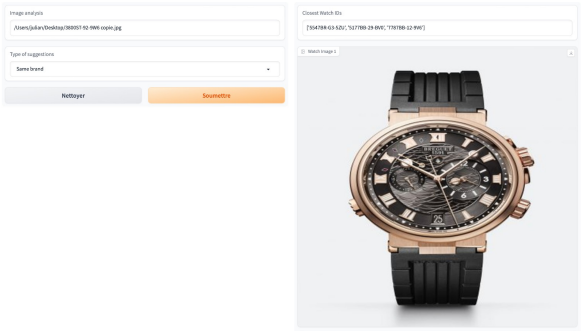

Figure 4: Gradio's interface

V. Results and limitations

Through the process of web scraping, we have successfully obtained a collection of 2065 images along with their associated text characteristics. By extracting the relevant features from these images and processing them, we were able to implement our CNN and KNN models.

Using CNN, we achieved a brand classification accuracy of approximately 82% on our test data set, and 77% on unlabeled watches. This means that when a user inputs a new image of a watch, and if the brand is present in our dataset, we can predict the correct brand almost 8 times out of 10. This model was obtained using a batch size of 32, 15 epochs, and 2 convolutional layers. We are pretty satisfied with these results.

For KNN, we had to develop our own metric as determining visual similarity between images is subjective. We used a 5-point rating scale to classify images, ranging from no resemblance (1) to very similar (5) (figure 5). When comparing our model using only the texture feature versus using all three features, we obtained scores of 2.7 and 2.86, respectively. This demonstrates that our predictions improve when we utilise all of our features.

Watch Reference (input)	Grade 1	Grade 2	Grade 3	Average	
PAM00355	2	3	2	2,33	
298600-3005	4	3	4	3,67	
WAT2112.BA0950	4	1	2	2,33	
116505-0001	2	1	5	2,67	
116505-0010	2	4	2	2,67	
PAM00021	3	2	3	2,67	
110338	2	3	2	2,33	
				2,7	AVERAGE (TEXTURE ONLY)

Figure 5: Grading watches references from KNN

Finally, we wrapped all of our data and models into a Gradio interface, to provide a compelling user experience.

Despite the positive results achieved in our project, there are some limitations that should be acknowledged. One limitation is that the accuracy of the CNN model could be further improved by increasing the size of our dataset and ensuring a more balanced distribution of data, as we don't always have the same number of images per brand. By including more images, especially those representing underrepresented brands or variations, we could enhance the model's ability to classify watches accurately.

Additionally, for the KNN model, there are potential improvements that can be applied. As we saw, we got a rather average rating, even when combining our 3 features. By experimenting a bit with various testing, we often encountered an image that was a very good recommendation, along another that seemed quite different. The situation was worse for watches that contained unusual forms. As for CNN, having more data could help the model by

being able to better learn the features and differences among different brands. Another possibility is to explore alternative features that could contribute to a better assessment of visual similarity between images. We could, for example, incorporate new features, such as shape descriptors or edge features. Additionally, we could experiment with various standardisation techniques, and evaluate if the model's output and predictions can be further enhanced.

By improving our models in such ways, we could further enhance our recommendations and empower customers with the necessary tools and information to make informed decisions when purchasing their new luxury watch.

VI. References

(1) Report: "The Deloitte Swiss Watch Industry Study 2022" https://www2.deloitte.com/content/dam/Deloitte/ch/Documents/consumer-business/deloitte-ch-en-swiss-watch-industry-study-2022.pdf

(2) Report: "THE MILLENNIAL STATE OF MIND" https://www.bain.com/contentassets/0b0b0e19099a448e83af2fb53a5630aa/bain20media20pack_the_millennial_state_of_mind.pdf

(3) Report: "Luxury Goods in Switzerland" https://www.statista.com/outlook/cmo/luxury-goods/switzerland

(4) Report: "Watch Industry statistics" https://www.fhs.swiss/eng/statistics.html

(5) Study: "State of the industry - Swiss watchmaking in 2022" https://watchesbysjx.com/2022/03/morgan-stanley-watch-industry-report-2022.html

(6) Data base: "Watch base" https://watchbase.com/

1_Images_Scraping.py

```
from selenium import webdriver
from selenium.webdriver.common.by import By
import time
import os
import urllib.request
import selenium.common.exceptions
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions
as EC
import ssl

ssl._create_default_https_context =
ssl._create_unverified_context

# Brands and their corresponding URLs
brands = ["Rolex", "Omega", "Patek-philippe", "Audemars-
piguet", "Cartier", "Breitling", "Tag-Heuer", "Iwc",
"Chopard",
          "Jaeger-lecoultre", "Blancpain", "Hublot",
"Zenith", "Certina", "Panerai", "Girard-perregaux",
"Breguet",
          "Montblanc", "Tissot", "Bulgari"]

urls = ["https://watchbase.com/" + brand.lower() for brand
in brands]

# Number of images to download per category per brand
images_per_category = 55
target_per_category = 50
timeToSleep = 2

# Open the browser
driver = webdriver.Firefox()

# Loop through each brand's page
for url in urls:
    driver.get(url)

    # Find the first 5 categories on the page and get their
links
    categories = driver.find_elements(By.CSS_SELECTOR,
".family-box.row .col-md-6 h2.title a")[:5]
    category_links = [category.get_attribute("href") for
category in categories]

    # Loop through each category and extract the image URLs
    total_count = 0
    for link in category_links:
        print("--------------------" + link)
        try:
            driver.get(link)
        except
selenium.common.exceptions.NoSuchWindowException:
            # Reopen the window or tab
            driver.execute_script("window.open('');")
            driver.switch_to.window(driver.window_handles[-
1])
            # Navigate to the URL again
            driver.get(link)

        # Wait for page to load
        time.sleep(timeToSleep)

        # Scroll down the page several times to trigger
image loading
        for i in range(5):
            driver.execute_script('window.scrollTo(0,
document.body.scrollHeight);')
            time.sleep(timeToSleep)

        # Wait for all images to load
        WebDriverWait(driver,
10).until(EC.visibility_of_all_elements_located((By.TAG_NAME
, 'img')))
```

2a_text_scraping.py

```
        # Wait for all images to load
        WebDriverWait(driver,
10).until(EC.visibility_of_all_elements_locat
ed((By.TAG_NAME, 'img')))
        # Find all the image tags on the page
        images =
driver.find_elements(By.TAG_NAME,
'img')[:images_per_category]
        # Find the name of the references
        reference_links =
driver.find_elements(By.TAG_NAME, "strong")
        reference_names = [reference.text for
reference in reference_links]
        # Collect the URLs of the images that
match the expected prefix
        vector = ["speedmaster-automatic-and-
other", "datejust"]
        brand_prefix =
f"https://cdn.watchbase.com/watch/md/{link.sp
lit('/')[-2] + '/' + link.split('/')[-1]}"
        image_urls =
[image.get_attribute('src') for image in
images if

image.get_attribute('src').startswith(brand_p
refix) or

(image.get_attribute('src').startswith(v) for
v in vector)]
        # Replace "md" with "lg" in each URL
to get higher quality images
        modified_urls = [url.replace('/md/',
'/lg/') for url in image_urls]
        # Create a folder on your desktop to
save the images
        folder_path =
os.path.expanduser(f"~/Desktop/{link.split('/
')[-2]}_images")
        if not os.path.exists(folder_path):
            os.makedirs(folder_path)

        # Download the images and save them
to the folder with the reference name as file
name
        count = 0
        for i, url in
enumerate(modified_urls):
            if count == target_per_category:
                break

            reference_name =
reference_names[(i - 3) %
len(reference_names)]
            if '/' in reference_name:
                reference_name =
reference_name.replace('/', '-')
            file_name =
os.path.join(folder_path,
f"{reference_name}.jpg")
            urllib.request.urlretrieve(url,
file_name)
            print(f"Downloaded image {i + 1}:
{file_name}")
            count += 1
            total_count += 1

        if total_count >= images_per_category
* len(category_links):
            break

# Close the browser
driver.quit()
```

```
import csv
import os
import time
import pandas as pd
from selenium import webdriver
from selenium.webdriver.common.by import By
from tqdm import tqdm
from selenium.webdriver.support.ui import
WebDriverWait
from selenium.webdriver.support import
expected_conditions as EC
from selenium.common.exceptions import
NoSuchElementException
from urllib3.util.url import get_host
import ssl

ssl._create_default_https_context =
ssl._create_unverified_context

# Create an empty pandas DataFrame to store the
watch data: we will then add all the values that
are stored in the watch_data_dict
watch_data = pd.DataFrame(
    columns=["Brand", "Family", "Reference",
"Name", "Movement", "Produced", "Limited",
"Materials", "Material",
             "Bezel", "Glass", "Back", "Shape",
"Diameter", "Height", "Lug Width", "W/R",
"Nickname", "Color", "Finish",
             "Indexes", "Hands", "Price"])

# Brands and their corresponding URLs
brands = ["Rolex", "Omega", "Patek-philippe",
"Audemars-piguet", "Cartier", "Breitling", "Tag-
Heuer", "Iwc", "Chopard",
          "Jaeger-lecoultre", "Blancpain", "Hublot",
"Zenith", "Certina", "Panerai","Girard-perregaux",
"Breguet",
           "Montblanc", "Tissot", "Bulgari"]
urls = ["https://watchbase.com/" + brand.lower()
for brand in brands]

# Cleaning the data: sometimes the variable
Material is saved in plural. We added both in the
dataframe watch data and we merged them into
"Materials".
watch_data['merged_column'] =
watch_data['Materials'].fillna(watch_data['Material
'])
watch_data.drop(['Materials', 'Material'], axis=1,
inplace=True)
watch_data.rename(columns={'merged_column':
'Materials'}, inplace=True)

# Open the browser
driver = webdriver.Firefox()

# Loop through each brand's page
for url in urls:
    driver.get(url)

    # Find the first 5 categories on the page and
get their links
    categories =
driver.find_elements(By.CSS_SELECTOR, ".family-
box.row .col-md-6 h2.title a")[:5]
    category_links =
[category.get_attribute("href") for category in
categories]

    # Loop through each category and extract the
text
    for link in tqdm(category_links):
        driver.get(link)
```

```
# Wait for page to load
        time.sleep(2)

        # Scroll down the page several times to
trigger image loading
        for i in range(5):

driver.execute_script('window.scrollTo(0,
document.body.scrollHeight);')
            time.sleep(2)

        # Now we landed on the category website
but we have to be inside each watch website
        watches =
driver.find_elements(By.CSS_SELECTOR, ".watch-
block-container a.item-block")[:50]  # we seek
to take 50 images
        watches_link =
[watch.get_attribute("href") for watch in
watches]

        for each_watch in watches_link:
            driver.get(each_watch)

            # Wait for page to load
            time.sleep(2)

            # Scroll down the page several times
to trigger image loading
            for i in range(5):

driver.execute_script('window.scrollTo(0,
document.body.scrollHeight);')
                time.sleep(2)

            watch_data_dict = {}

            # Extract watch details from the
first table
            first_table =
driver.find_element(By.CLASS_NAME, "info-table")
            rows_first_table =
first_table.find_elements(By.TAG_NAME, "tr")

            for row in rows_first_table:
                row_data = row.text.split(":",
1)  # split on first occurrence of ":"

                if len(row_data) == 2:
                    key = row_data[0].strip()
                    value = row_data[1].strip()
                    watch_data_dict[key] = value
                    print(watch_data_dict)

            # Extract watch details from the
second table
            second_table =
driver.find_element(By.CLASS_NAME, "col-xs-6")
            row_second_table =
second_table.find_elements(By.TAG_NAME, "tr")

            for row in row_second_table:
                row_data = row.text.split(":",
1)  # because reference sometimes has (aka
number)

                if len(row_data) == 2:
                    key = row_data[0].strip()
                    value = row_data[1].strip()
                    watch_data_dict[key] = value
                    print(watch_data_dict)

            try:
                # Find the element with the
"data-url" attribute
                element =
driver.find_element(By.CSS_SELECTOR,
"#pricechart")
                data_url =
element.get_attribute("data-url")

                # Open the "data-url" in a new
tab

driver.execute_script(f"window.open('{data_url}'
)")
```

```
# Switch to the new tab
                # Add an explicit wait to ensure the
new tab is fully loaded before switching
                WebDriverWait(driver,
10).until(EC.number_of_windows_to_be(2))

driver.switch_to.window(driver.window_handles[1])

                # Wait for the new tab to load
                # Add an explicit wait to ensure the
content of the new tab is loaded
                WebDriverWait(driver,
10).until(EC.presence_of_element_located((By.CSS_SELEC
TOR,"#\\/datasets\\/0\\/data\\/1 > td:nth-child(2) >
span:nth-child(1) > span:nth-child(1)")))

                # Find the price
                price_element =
driver.find_element(By.CSS_SELECTOR,
"#\\/datasets\\/0\\/data\\/1 > td:nth-child(2) >
span:nth-child(1) > span:nth-child(1)")
                price_value = price_element.text if
price_element.text != "null" else "NA"

                # Add the price value to the
watch_data_dict
                watch_data_dict["Price"] = price_value

                # Close the new tab
                driver.close()

                # Switch back to the main tab

driver.switch_to.window(driver.window_handles[0])

            except NoSuchElementException:
                    print("No price section found.
Skipping watch.")
            # Add the watch data to the pandas
DataFrame
            watch_data.loc[len(watch_data)] =
watch_data_dict
            print(watch_data)

# Replace "NaN" values in the "Price" column with "NA"
watch_data["Price"].fillna("NA", inplace=True)
# Count the number of missing values
for col in watch_data.columns:
    missing_values = watch_data[col].isna().sum()
    print(f"Number of missing {col} values:
{missing_values}")

# Save the pandas DataFrame to a CSV file
watch_data.to_csv("2b_cleaned_watch_text_def.csv",
index=False)
```

3a_mapping.py

```
import os
import pandas as pd
import re

# Load the characteristics dataset
char_df =
pd.read_csv('2b_cleaned_watch_text_def.csv')
pattern = r' \(aka.*\)'
char_df['Reference'] =
char_df['Reference'].apply(lambda x: re.sub(pattern,
'', x))
char_df['Reference'] =
char_df['Reference'].str.replace('/', '-')

# Create a new column with the image file names
char_df['image_file'] =
char_df['Reference'].apply(lambda x: f"{x}.jpg")
char_df['image_file_processed'] = 'processed_' +
char_df['image_file']

# Define the directory where the image files are
stored
image_dir = './watches_images'
```

```
# Create a dictionary to map the
characteristics to the images
char_to_image = {}
for subdir, _, files in os.walk(image_dir):
    for file in files:
        if file.endswith('.jpg'):
            image_path = os.path.join(subdir,
file)
            reference = file[:-4]  # Remove the
file extension
            if reference in
char_df['Reference'].values:
                char_to_image[reference] =
char_df.loc[char_df['Reference'] ==
reference].iloc[0].to_dict()

mapped_df =
pd.DataFrame.from_dict(char_to_image,
orient='index')

mapped_df.to_csv("4b_data_with_images.csv",
index=False)

# Print the dictionary to verify the mapping
print(char_to_image)
```

4a_gray_color_texture.py

```
import os
from PIL import Image
import numpy as np
import pandas as pd
from skimage.feature import graycomatrix,
graycoprops
from sklearn.preprocessing import StandardScaler

# Set the directory path containing the images
dir_path = "./watches_images"

# Create an empty DataFrame with columns for the
image filename, the grayscale histogram features,
the color
# histogram features, and the texture features
df_gray = pd.DataFrame(columns=["filename",
"gray_features"])
df_color = pd.DataFrame(columns=["filename",
"color_features"])
df_texture = pd.DataFrame(columns=["filename",
"texture_features"])

# Loop through each subfolder in the directory
for subfolder in os.listdir(dir_path):
    subfolder_path = os.path.join(dir_path,
subfolder)
    # Check if the subfolder is actually a
directory
    if not os.path.isdir(subfolder_path):
        continue
    # Loop through each image in the subfolder
    for filename in os.listdir(subfolder_path):
        # Ignore non-image files and the
.DS_Store file
        if not filename.endswith(('.jpg',
'.jpeg', '.png')) or filename == '.DS_Store':
            print(f"Skipping file: {filename}")
            continue
        # Ignore files that don't start with
"processed_"
        if not filename.startswith('processed_'):
            print(f"Skipping file: {filename}")
            continue
        # Extract the reference name by removing
the "processed_" prefix and file extension
        reference =
os.path.splitext(filename[len('processed_'):])[0]
        # Load the image
        img =
Image.open(os.path.join(subfolder_path,
filename))
```

```
        # Compute the grayscale histogram features
        gray_hist = np.array(gray.histogram())
        gray_hist = gray_hist / np.sum(gray_hist)  #
Normalize the histogram so that the values sum to 1
        # Compute the color histogram features
        color_hist = np.array(img.histogram())
        color_hist = color_hist / np.sum(color_hist)  #
Normalize the histogram so that the values sum to 1
        # Compute the texture features using
graycomatrix and graycoprops
        gray_arr = np.array(gray)
        glcm = graycomatrix(gray_arr, distances=[1],
angles=[0], levels=256, symmetric=True, normed=True)
        texture_props = np.array([graycoprops(glcm,
'contrast'), graycoprops(glcm, 'energy'),
graycoprops(glcm, 'homogeneity'), graycoprops(glcm,
'correlation')]).reshape(-1)
        # Reshape the feature vectors to 1D arrays
        gray_features = gray_hist.reshape(-1)
        color_features = color_hist.reshape(-1)
        # Concatenate the feature vectors
        features = np.concatenate([gray_features]),  #
color_features, texture_props])
        # Add the new row to the DataFrame of new data
        df_gray = pd.concat([df_gray,
pd.DataFrame({"filename": [reference], "gray_features":
[gray_features]})], ignore_index=True)
        df_color = pd.concat([df_color,
pd.DataFrame({"filename": [reference],
"color_features":
[color_features]})],ignore_index=True)
        df_texture = pd.concat([df_texture,
pd.DataFrame({"filename": [reference],
"texture_features": [texture_props]})],
ignore_index=True)

        # Print the filename of the processed image
        print(f"Processed image: {reference}")

# Create a merged dataset which contains all the 3
features
merged_df = pd.merge(df_gray, df_color, on="filename")
merged_df = pd.merge(merged_df, df_texture,
on="filename")
merged_df.to_csv("5b_merged_df.csv")
```

5a_df_all_standardized.py

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler

# Read the CSV file into a pandas DataFrame
data = pd.read_csv('5b_merged_df.csv')

# Extract the features into separate arrays
gray_features = data['gray_features'].apply(lambda
x: np.fromstring(x[1:-1], sep=' ')).values
color_features = data['color_features'].apply(lambda
x: np.fromstring(x[1:-1], sep=' ')).values
texture_features =
data['texture_features'].apply(lambda x:
np.fromstring(x[1:-1], sep=' ')).values

print("Shapes:")
print("gray_features shape:", gray_features.shape)
print("color_features shape:", color_features.shape)
print("texture_features shape:",
texture_features.shape)

# Create a StandardScaler object
scaler = StandardScaler()

# Scale the individual feature arrays
gray_features_scaled =
np.vstack(gray features).astype(float)
```

```
gray_features_scaled =
scaler.fit_transform(gray_features_scaled).tolis
t()

color_features_scaled =
np.vstack(color_features).astype(float)
color_features_scaled =
scaler.fit_transform(color_features_scaled).toli
st()

texture_features_scaled =
np.vstack(texture_features).astype(float)
texture_features_scaled =
scaler.fit_transform(texture_features_scaled).to
list()

# Update the data DataFrame with the scaled
features
data['gray_features'] = gray_features_scaled
data['color_features'] = color_features_scaled
data['texture_features'] =
texture_features_scaled

# Save the updated DataFrame to a CSV file
data.to_csv("6b_merged_df_scaled.csv",
index=False)
```

6a_cnn_training.py

```
#Load libraries
import os
import numpy as np
import torch
import glob
import torch.nn as nn
from torchvision.transforms import transforms
from torch.utils.data import DataLoader
from torch.optim import Adam
from torch.autograd import Variable
import torchvision
import pathlib
import random
import shutil
# Checking for device
device = torch.device('cuda' if
torch.cuda.is_available() else 'cpu')
print(device)

# Transforms
transformer = transforms.Compose([
    transforms.Resize((350, 350)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5], [0.5,
0.5, 0.5])
])

# Path to the original image folder
original_folder = './watches_images_cnn' #contains
all the images; warning, the folder is empty after
operation

# Path to the destination folders for validation,
testing, and training
prediction_folder = './prediction'
testing_folder = './testing'
training_folder = './training'

# Create the destination folders if they don't
exist
os.makedirs(prediction_folder, exist_ok=True)
os.makedirs(testing_folder, exist_ok=True)
os.makedirs(training_folder, exist_ok=True)

# List to store alli image paths and their
corresponding labels
image_paths_labels = []
# Get the subdirectories within the original
folder
subdirectories = [subdir for subdir in
os.listdir(original_folder) if
os.path.isdir(os.path.join(original_folder,
subdir))]
```

```
# Iterate over the subdirectories
for category in subdirectories:
    category_folder = os.path.join(original_folder,
category)
    for file in os.listdir(category_folder):
        if file.endswith(('.jpg', '.png', '.jpeg')):
            image_path = os.path.join(category_folder,
file)
            image_paths_labels.append((image_path,
category))

# Print the total number of images
print(f"Total number of images:
{len(image_paths_labels)}")

# Shuffle the image paths randomly
random.shuffle(image_paths_labels)

# Calculate the number of images for each set
total_images = len(image_paths_labels)
prediction_count = int(total_images * 0.15)
testing_count = int(total_images * 0.15)
training_count = total_images - prediction_count -
testing_count

# Split the image paths and labels into validation,
testing, and training sets
prediction_data =
image_paths_labels[:prediction_count]
testing_data =
image_paths_labels[prediction_count:prediction_count +
testing_count]
training_data = image_paths_labels[prediction_count +
testing_count:]
# Make sure that prediction folder images are shuffled
(no pattern)
prediction_images = os.listdir(prediction_folder)
random.shuffle(prediction_images)
# Move the images to the respective output folders
while maintaining the category structure

for image, _ in prediction_data:
    file_name = os.path.basename(image)
    dst_path = os.path.join(prediction_folder,
file_name)
    shutil.move(image, dst_path)


for image, label in testing_data:
    file_name = os.path.basename(image)
    dst_path = os.path.join(testing_folder, label,
file_name)  # Create subfolders for each category
    os.makedirs(os.path.dirname(dst_path),
exist_ok=True)  # Create subfolders if they don't
exist
    shutil.move(image, dst_path)

for image, label in training_data:
    file_name = os.path.basename(image)
    dst_path = os.path.join(training_folder, label,
file_name)  # Create subfolders for each category
    os.makedirs(os.path.dirname(dst_path),
exist_ok=True)  # Create subfolders if they don't
exist
    shutil.move(image, dst_path)
print("Image splitting and shuffling completed
successfully.")
print(f"Number of images in prediction folder:
{len(os.listdir(prediction_folder))}")
testing_image_count = sum(len(files) for _, _, files
in os.walk(testing_folder))
training_image_count = sum(len(files) for _, _, files
in os.walk(training_folder))

print(f"Number of images in testing folder:
{testing_image_count}")
print(f"Number of images in training folder:
{training_image_count}")


#Path for training and testing directory
train_path='./training'
test_path='./testing'
```

```
train_loader=DataLoader(

torchvision.datasets.ImageFolder(train_path,transf
orm=transformer),
    batch_size=32, shuffle=True
)
test_loader=DataLoader(

torchvision.datasets.ImageFolder(test_path,transfo
rm=transformer),
    batch_size=32, shuffle=True
)

# Categories
root=pathlib.Path(train_path)
classes=sorted([j.name.split('/')[-1] for j in
root.iterdir()])

print(classes)


# CNN Network
class ConvNet(nn.Module):
    def __init__(self, num_classes=20):
        super(ConvNet, self).__init__()

        # Output size after convolution filter
        # ((w-f+2P)/s) +1

        # Input shape= (256,3,350,350)

        self.conv1 = nn.Conv2d(in_channels=3,
out_channels=12, kernel_size=3, stride=1,
padding=1) # let's try with 12 channels as our
data set is quite small
        # Shape= (256,12,350,350)
        self.bn1 = nn.BatchNorm2d(num_features=12)
# same number as number of channels; number of
different filters or feature maps produced by that
layer
        # Shape= (256,12,350,350)
        self.relu1 = nn.ReLU() # to bring non-
linearity
        # Shape= (256,12,350,350)

        self.pool = nn.MaxPool2d(kernel_size=2) #
reduces the height and width of convolutional
output while keeping the most salient features
        # Reduce the image size be factor 2
        # Shape= (256,12,175,175)

        self.conv2 = nn.Conv2d(in_channels=12,
out_channels=20, kernel_size=3, stride=1,
padding=1) # add second conv layer to apply more
patterns and increase the number of channels to 20
        # Shape= (256,20,175,175)
        self.relu2 = nn.ReLU()
        # Shape= (256,20,175,175)

        self.conv3 = nn.Conv2d(in_channels=20,
out_channels=32, kernel_size=3, stride=1,
padding=1)
        # Shape= (256,32,175,175)
        self.bn3 = nn.BatchNorm2d(num_features=32)
        # Shape= (256,32,175,175)
        self.relu3 = nn.ReLU()
        # Shape= (256,32,175,175)

        self.fc = nn.Linear(in_features=175 * 175
* 32, out_features=num_classes) # fully connected
layer

    # Feed forward function

    def forward(self, input):
        output = self.conv1(input)
        output = self.bn1(output)
        output = self.relu1(output)

        output = self.pool(output)

        output = self.conv2(output)
        output = self.relu2(output)

        output = self.conv3(output)
        output = self.bn3(output)
        output = self.relu3(output)
```

```
# Above output will be in matrix form, with shape
(256,32,175,175)

        output = output.view(-1, 32 * 175 * 175)

        output = self.fc(output)

        return output

model=ConvNet(num_classes=20).to(device)
#Optmizer and loss function
optimizer=Adam(model.parameters(),lr=0.001,weight_dec
ay=0.0001)
loss_function=nn.CrossEntropyLoss()
num_epochs=15
#calculating the size of training and testing images
train_count=len(glob.glob(train_path+'/**/*.jpg'))
test_count=len(glob.glob(test_path+'/**/*.jpg'))
print(train_count,test_count)

# Model training and saving best model

best_accuracy = 0.0

for epoch in range(num_epochs):

    # Evaluation and training on training dataset
    model.train()
    train_accuracy = 0.0
    train_loss = 0.0

    for i, (images, labels) in
enumerate(train_loader):
        if torch.cuda.is_available():
            images = Variable(images.cuda())
            labels = Variable(labels.cuda())

        optimizer.zero_grad()

        outputs = model(images)
        loss = loss_function(outputs, labels)
        loss.backward()
        optimizer.step()

        train_loss += loss.cpu().data *
images.size(0)
        _, prediction = torch.max(outputs.data, 1)

        train_accuracy += int(torch.sum(prediction ==
labels.data))

    train_accuracy = train_accuracy / train_count
    train_loss = train_loss / train_count

    # Evaluation on testing dataset
    model.eval()

    test_accuracy = 0.0
    for i, (images, labels) in
enumerate(test_loader):
        if torch.cuda.is_available(): # way to use
GPU instead of CPU
            images = Variable(images.cuda())
            labels = Variable(labels.cuda())

        outputs = model(images)
        _, prediction = torch.max(outputs.data, 1)
        test_accuracy += int(torch.sum(prediction ==
labels.data))

    test_accuracy = test_accuracy / test_count

    print('Epoch: ' + str(epoch) + ' Train Loss: ' +
str(train_loss) + ' Train Accuracy: ' + str(
        train_accuracy) + ' Test Accuracy: ' +
str(test_accuracy))

    # Save the best model
    if test_accuracy > best_accuracy:
        torch.save(model.state_dict(),
'7b_best_checkpoint.model')
        best_accuracy = test_accuracy
```

a

6b_cnn_inference.py ...

Remainig code is available in the zip file