Image Filter Recognition

Ben Baker, Matthew Steffen, Steven Fuller

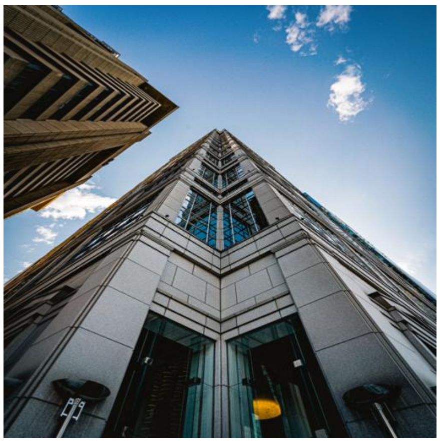
10 December, 2020

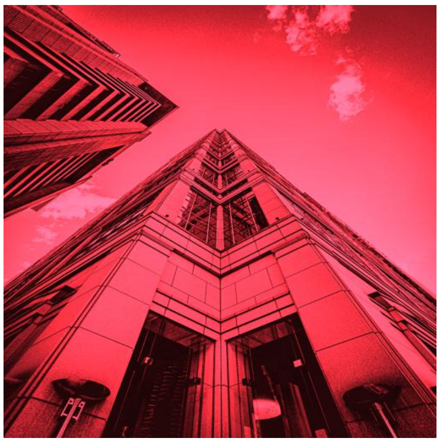
Introduction

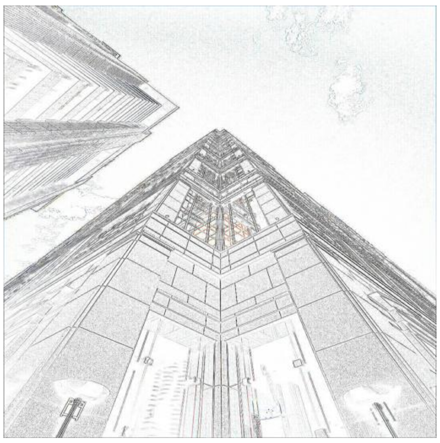
We wish to create an algorithm that detects the presence of filters or the types of filters applied to photos. Filters like those common on social media platforms like Instagram, Facebook, and Snapchat allow pictures to be edited (perturbed) with thematic changes to each pixel creating dramatic effects. We aim to create a model that will accept a photo and outputs which of the trained filters has been placed on the photo.

The final product will be given an image, scale the image to fit our trained data set, and return which filter matches the given photo.

Below demonstrates the sepia red, and contour filters applied to the original photo.


Original


Sepia Red


Contour

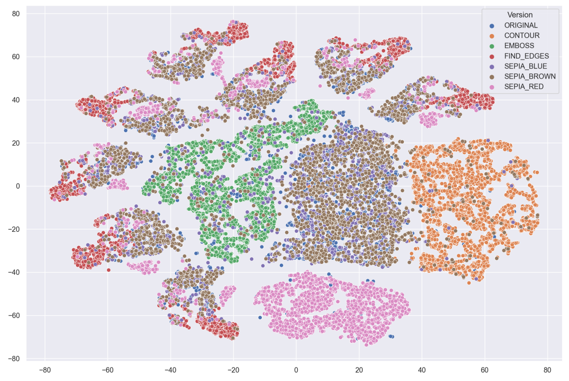


Figure 1: t-SNE of Image Data

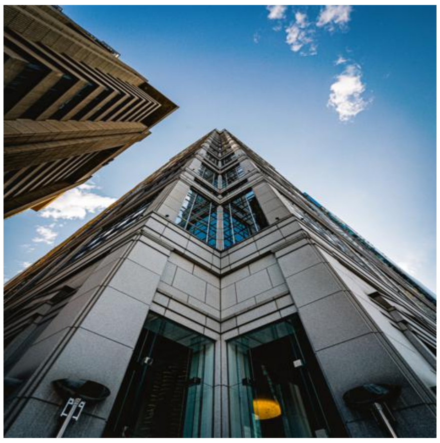
Our methods are based upon two papers published by the journals *Optics & Laser Technology* and *Optics Communications*. The first deals with impact of object recognition on phase shifted photos and the second convolutions impact on photo coloring. The methods we use derived from the papers are described later. It is important to note, that we will use more tools than prescribed in these papers and that they were found in support of what we had already decided to undertake. We use them only to justify the tools applied as they provide scientific backing to our methods.

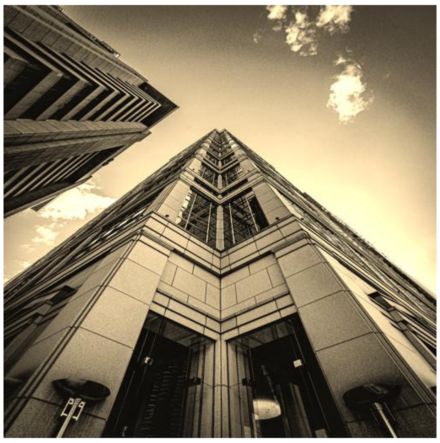
We took a total of 4096 photos and applied 14 initial filters. We reduced the usable filters by only using those filters that provided a large enough change to the picture as measured by MSE between the photos. This left us with 7 filters. Next we developed several features that involved color frequencies and averages. To illustrate further uniqueness we also took the same statistics on the images convoluted. We then visualized the data to determine which clusters would be easy to identify. As seen in Figure 1, there are distinct clusters however several filters, including the non-filtered original image are spread throughout thus will be more difficult to identify (as we see in Figure 4). We then built a random forest model to classify and after optimizing hyper-parameters we have about a 97% accuracy of the model.

Data Cleaning

For our data set, we scraped images from the website unsplash.com which is permissible to common use under the creative commons license (see appendix). We used their random image generator (which draws a random image from their database) to get 4096 images. We then cleaned the data by first applying 14 unique filters, and then resizing every image to a standard 512x512 pixels. The scraping used the requests python library, and the filtering and resizing used the Python Imaging Library (PIL). We used the PIL's standard filters (contour as above), as well as a sepia (standard phase shift) filter to compile the 14 total filters. Each original image and it's filtered counterparts are then stored with a specific photo_id for future reference throughout the project.

Original


Resized


Filtered (Sepia Brown)

Due to the variety of the different filters, it may be equally as difficult to distinguish the presence of a filter as the type of filter itself. We considered only using one type of filter like phase shifting (see appendix for code). However, the other filters offer a huge variety in the appearance of the images thus we have elected to keep and use all applied filters.

While all the photos were taken from the image generator there may already be some filters in a few of the photos which may skew our results. We believe this may be the case since during the scraping, there were a couple pictures that appeared to be very filtered. During the scraping process we eliminated those pictures and replaced them with clearly non-filtered pictures. 4096 photos are a lot to keep up with so we may have missed a few. We hope that in applying the PIL and sepia filters will still create enough of a distinction as to maintain the model's integrity.
Note, we decided to scale the pictures after filtration since this should improve the accuracy of running non-scaled photos through the algorithm.

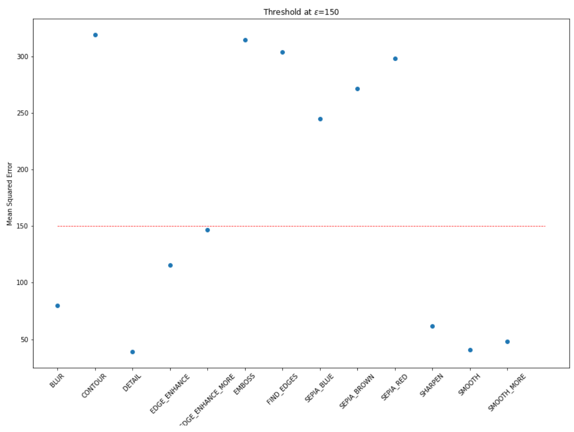


Figure 2: MSE Loss for the different filters

Due to the inaccuracy of the algorithms tested in class we began investigating why the tests would perform so poorly. We looked at the filtered photos and compared them to the originals. We then graphed the MSE between each of the filters and their original photo. Many of the photos were not very different from their original thus we defined an epsilon and determined that any photo who's MSE was below epsilon we would eliminate since they were not filtered enough for our algorithm to detect. The range of MSE was $[25, 325]$ thus by setting $\epsilon = 150$ we keep a majority of the filters that are most detectable. (See appendix for sample code)

Feature Engineering

Our initial database of 4096 photos has been converted to 57,344 photos by applying 14 unique filters to each photo. This filtration are not part of the feature engineering, they are building the data set. We then gathered 24 unique features for each photo (including the original and each of it's filtered counterparts separately).

We have decided on 3 modes of data.

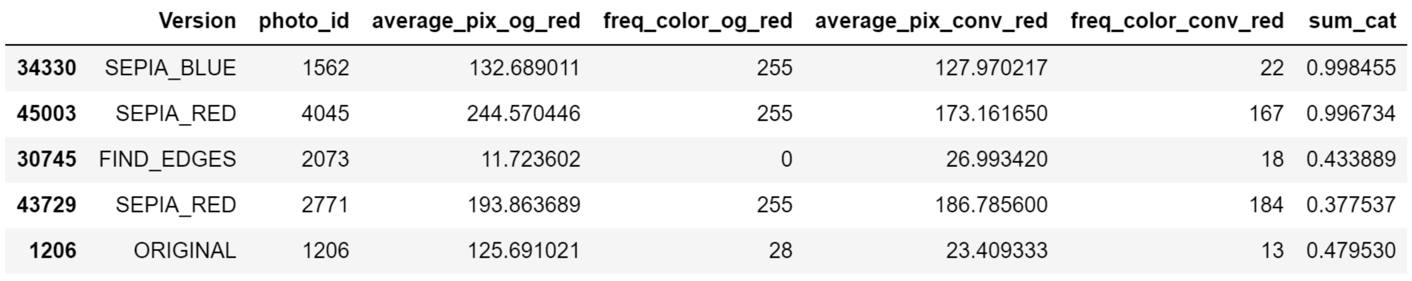
1. The presence of an object that has a general hue different from its surrounding may indicate a phase shift of some kind. We also are considering the idea that if a classifier predicts the presence of an item with less confidence that might be due to some unnoticed noise (a filter). Thus we gathered and implemented an object recognition package from the python package *gluoncv* which does image object recognition for 19 items (bicycle, bird, boat, bottle, bus, car, cat, chair, cow, dining table, dog, horse, motorbike, person, potted plant, sheep, sofa, train, tvmonitor). In our data-set we have these 1-hot-encoded with the probabilities placed in the entries of the columns. Later we combined the average of the non-zero probabilities into one column so that we can measure the impact of filters on the probabilities (see figure 3).

2. Next, we found the average pixel color strengths and the average pixel color frequency. We will use these numbers in conjunction to the next two data points to ascertain if an image has an unusually strong presence (red would be the most prominent in a red-shifted photo).

3. We then used pytorch's nn.conv2d module to go through 1 convolution which isolates average colors in their respective blocks creating a drastically different image. With very discrete color schemes. If too many colors are the same, the image is close to black and white, or a very monotone image this implies a filter is present. Now we use this and the previous tools to determine how much change has occurred during the convolution. We can then answer questions dealing the change in mean pixel colors such as 'If the mean pixel was red-dominant and now is green/blue-dominant then the red was very prevalent on average throughout the picture, does this imply red-shift?'

We have made several modifications to the original data set resulting in this new data set.The original data set incorporated unique probabilities of 10 different object classifications, this new data set combines the average of all the non-zero probabilities giving us a general idea of how the classifier saw the images. This increased the effectiveness of the classifier in the data set. We also split tuples of RGB data (average pixel or frequency of the pixel values) into their own respective columns. These modifications allowed a much faster implementation of the algorithms discussed later.

Below we have a sample of the dataframe we will be using.



	Version	photo_id	average_pix_og_red	freq_color_og_red	average_pix_conv_red	freq_color_conv_red	sum_cat
34330	SEPIA_BLUE	1562	132.689011	255	127.970217	22	0.998455
45003	SEPIA_RED	4045	244.570446	255	173.161650	167	0.996734
30745	FIND_EDGES	2073	11.723602	0	26.993420	18	0.433889
43729	SEPIA_RED	2771	193.863689	255	186.785600	184	0.377537
1206	ORIGINAL	1206	125.691021	28	23.409333	13	0.479530

Note that each 'red' column has a corresponding blue and green not shown here.

The version indicates what type of filter has been placed, with 'ORIGINAL' being non-filtered. Then the two main features we have engineered are classification probabilities 'sum_cat' and the impact of filtration on average pixel colors and pixel color frequencies.

The creation of these features is justified by the two articles mentioned previously.

1. *Optics Communications* published *Matched filter and phase only filter performance in colour image recognition.* This article describes the use of phase shifts (where a specific color like blue is enhanced across the entirety of the image) and the impact that this has on object recognition. To use this paper, we are using an object classifier named *YOLO3* and observing the impact that filters have on the probabilities returned by the classifier [**?**].

2. *Optics & Laser Technology* published *Color component marking and convolution-based encoding for polychromatic pattern recognition.* We decided to observe the impact that convolutions have on images to determine the presence of a filter if the convolutions returned a specific color shift. The methods described in this paper outline the specific effects of convolution in color changes, thus we hope to include their methods as part of our analysis in determining if a filter is present based on the effects a filter should have on the image [**?**].

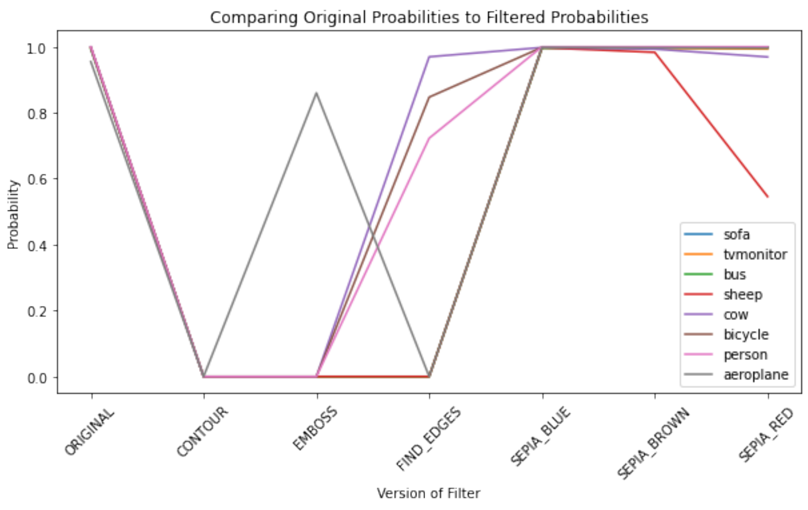


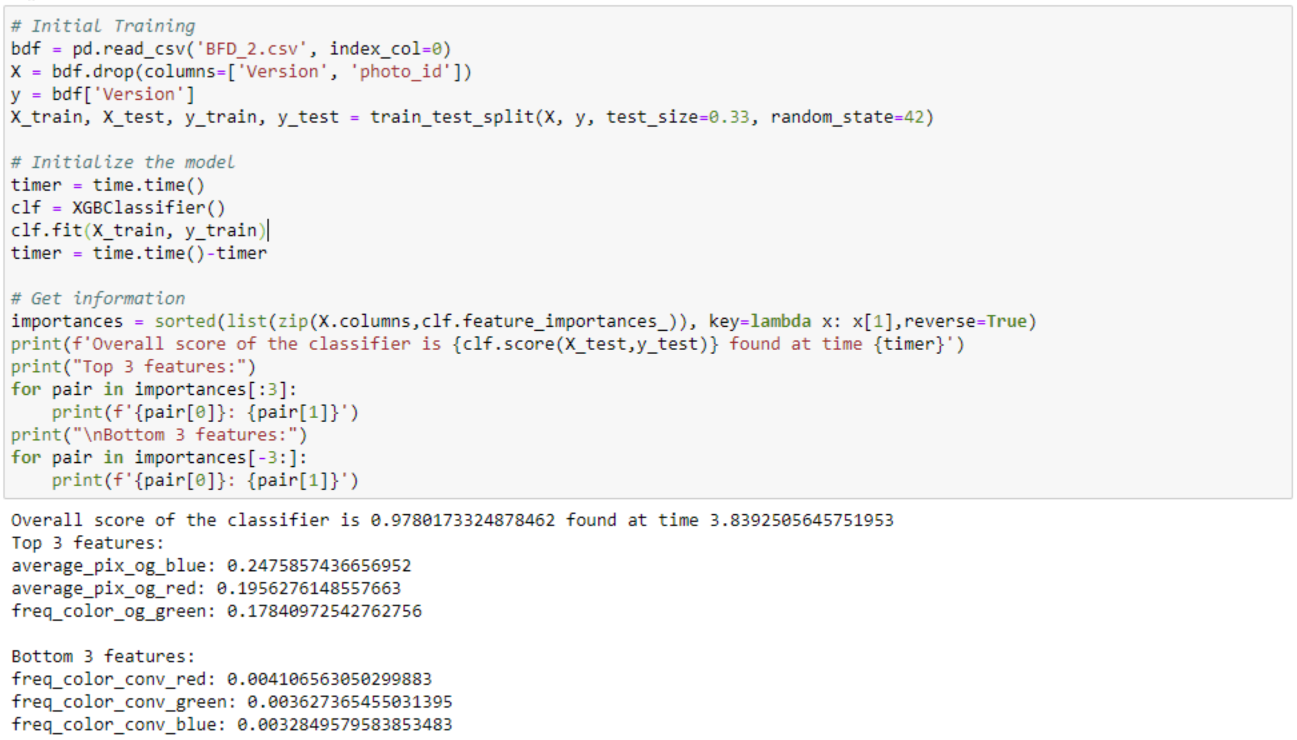
Figure 3: For each filter, the plot is below the line assigned by the original probabilities

In accordance with the *Optics Communications* paper we compare the impacts of filters on the confidence of the predictions of object classification using the YOLO3 network classification algorithm.

Now not all images have a probability so high, so we specifically chose different images to illustrate the differences that were had when the original image had a high enough probability of the object. This is one of the key features in our model (while not as impactful as the actual pixel discoloration itself). We image that in neural networks that are a bit more perceptive to noise (hence the research into adversarial networks) would place much more weight on the probabilities predicted. however according to the article the higher success of the models after incorporating this worth the cost calculations.

Our Model

We tested several different variations of linear regression, logistic regression, gradient boosted trees, random forests, XGBoost and as a final test the unsupervised method of OPTICS. We choose XGBoost after seeing that out of the box it almost outperformed everything else including OPTICS and random forests. We also found that the misclassification rate for each filter was better than the random forest. This held true as we optimized parameters using the sklearn package *GridSearchCV*, unfortunately any parameter search we used decreased the accuracy so we decided to keep the out of box implementation.



```
# Initial Training
bdf = pd.read_csv('BFD_2.csv', index_col=0)
X = bdf.drop(columns=['Version', 'photo_id'])
y = bdf['Version']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)

# Initialize the model
timer = time.time()
clf = XGBClassifier()
clf.fit(X_train, y_train)
timer = time.time()-timer

# Get information
importances = sorted(list(zip(X.columns,clf.feature_importances_)), key=lambda x: x[1],reverse=True)
print(f'Overall score of the classifier is {clf.score(X_test,y_test)} found at time {timer}')
print("Top 3 features:")
for pair in importances[:3]:
    print(f'{pair[0]}: {pair[1]}')
print("\nBottom 3 features:")
for pair in importances[-3:]:
    print(f'{pair[0]}: {pair[1]}')
```

```
Overall score of the classifier is 0.9780173324878462 found at time 3.8392505645751953
Top 3 features:
average_pix_og_blue: 0.2475857436656952
average_pix_og_red: 0.1956276148557663
freq_color_og_green: 0.17840972542762756

Bottom 3 features:
freq_color_conv_red: 0.004106563050299883
freq_color_conv_green: 0.003627365455031395
freq_color_conv_blue: 0.0032849579583853483
```

Now as we compare the score of each of the algorithms we get the following table:

Algorithms:	Linear Regression	Logistic Regression	Gradient Boosted Trees	Random Forest	XGBoost	OPTICS
Out of Box:	.0276	.375	.488	.97	.98	.0004
Regularized:	.419	.86	.8	.96	.97	.0004
Optimized:	.58	.86	.8	.96	.97	.9

Now in the final model we use the default hyper-parameters which trains in 4 seconds and provides the a 98% accuracy. To explore further we wanted to know which filters and what common traits caused the different classification rates. We found the following graph that represents the misclassifications for each filter:

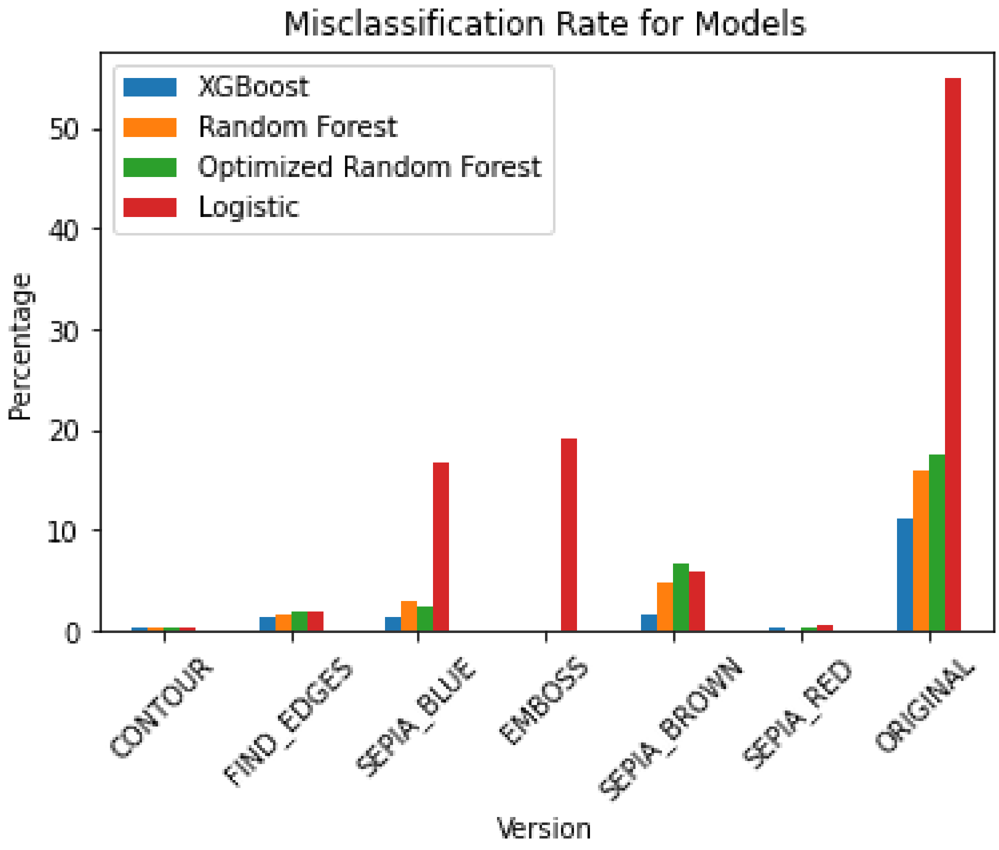


Figure 4: Note that the original was almost always had the poorest classification rates.

To our detriment, we had (until this visualization was observed) planned on using the random forest due to its similar accuracy and lower training costs. However, we found that the cost of calculations of the optimized XGBoost far outweighed the ease and accuracy of the default random forest.

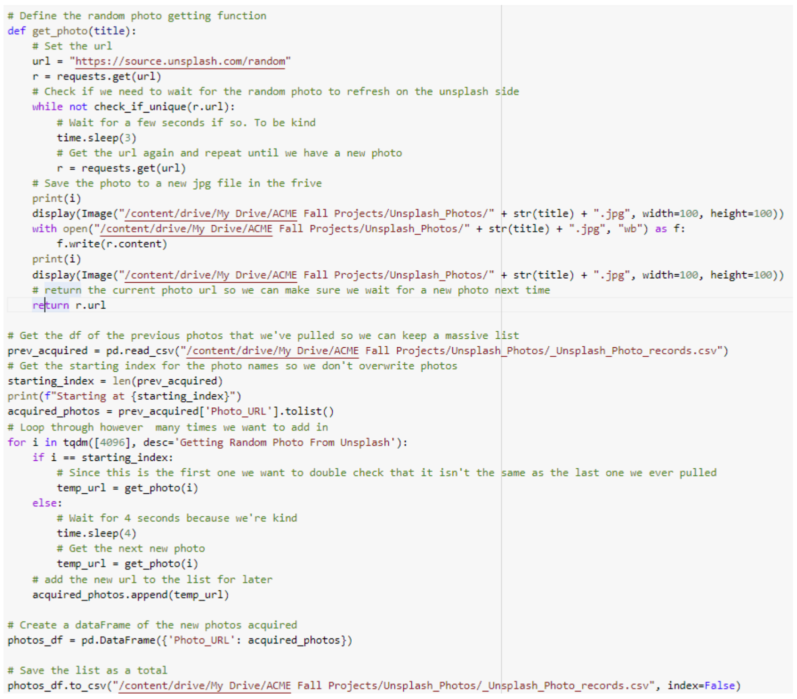
Ethical Discussion

The primary concern in regards to the ethics involved in our project is that of the use of the photos to train. The general implications of our classification do not propose many ethical risks. As seen in the creative commons license that 'unsplash.com' uses all photos we used to train our model are intended for public use. We used a random generator for gathering the data so that our bias against the photos chosen would be eliminated. This would hopefully reduce the risk of the classification model misclassifying due to the race of people or ethnic background of objects in the photos. Thus we see no reason for further ethical concern.

Appendix

Code snippets:

For a complete version of our data cleaning and feature engineering code please visit: Colab Link to Sample Code



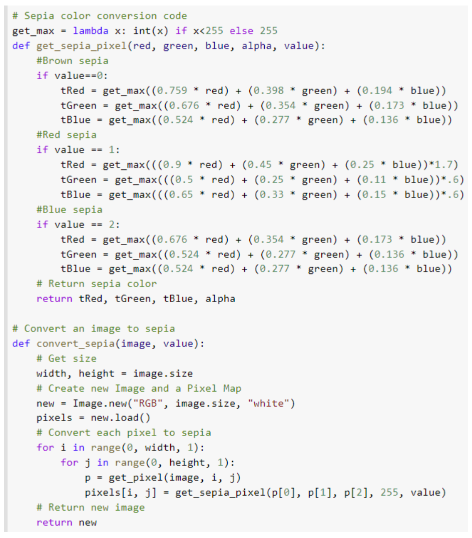
```python
# Define the random photo getting function
def get_photo(title):
    # Set the url
    url = "https://source.unsplash.com/random"
    r = requests.get(url)
    # Check if we need to wait for the random photo to refresh on the unsplash side
    while not check_if_unique(r.url):
        # Wait for a few seconds if so. To be kind
        time.sleep(3)
        # Get the url again and repeat until we have a new photo
        r = requests.get(url)
    # Save the photo to a new jpg file in the frive
    print(i)
    display(Image("/content/drive/My Drive/ACME Fall Projects/Unsplash_Photos/" + str(title) + ".jpg", width=100, height=100))
    with open("/content/drive/My Drive/ACME Fall Projects/Unsplash_Photos/" + str(title) + ".jpg", "wb") as f:
        f.write(r.content)
    print(i)
    display(Image("/content/drive/My Drive/ACME Fall Projects/Unsplash_Photos/" + str(title) + ".jpg", width=100, height=100))
    # return the current photo url so we can make sure we wait for a new photo next time
    return r.url

# Get the df of the previous photos that we've pulled so we can keep a massive list
prev_acquired = pd.read_csv("/content/drive/My Drive/ACME Fall Projects/Unsplash_Photos/_Unsplash_Photo_records.csv")
# Get the starting index for the photo names so we don't overwrite photos
starting_index = len(prev_acquired)
print(f"Starting at {starting_index}")
acquired_photos = prev_acquired['Photo_URL'].tolist()
# Loop through however  many times we want to add in
for i in tqdm([4096], desc='Getting Random Photo From Unsplash'):
    if i == starting_index:
        # Since this is the first one we want to double check that it isn't the same as the last one we ever pulled
        temp_url = get_photo(i)
    else:
        # Wait for 4 seconds because we're kind
        time.sleep(4)
        # Get the next new photo
        temp_url = get_photo(i)
    # add the new url to the list for later
    acquired_photos.append(temp_url)

# Create a dataFrame of the new photos acquired
photos_df = pd.DataFrame({'Photo_URL': acquired_photos})

# Save the list as a total
photos_df.to_csv("/content/drive/My Drive/ACME Fall Projects/Unsplash_Photos/_Unsplash_Photo_records.csv", index=False)
```

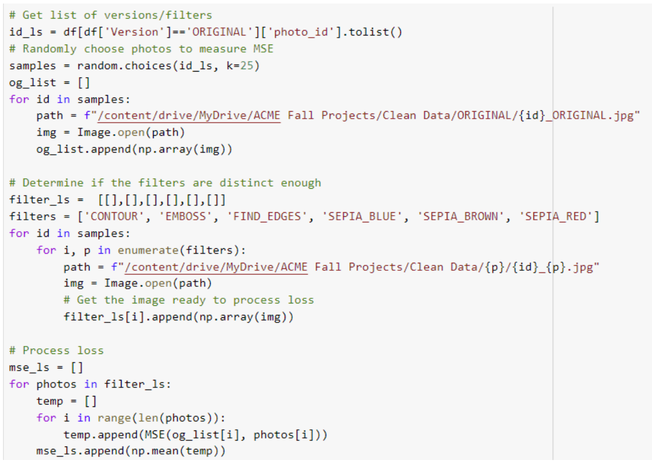
An image of the code used to scrape and save the images.



```python
# Sepia color conversion code
get_max = lambda x: int(x) if x<255 else 255
def get_sepia_pixel(red, green, blue, alpha, value):
    #Brown sepia
    if value==0:
        tRed = get_max((0.759 * red) + (0.398 * green) + (0.194 * blue))
        tGreen = get_max((0.676 * red) + (0.354 * green) + (0.173 * blue))
        tBlue = get_max((0.524 * red) + (0.277 * green) + (0.136 * blue))
    #Red sepia
    if value == 1:
        tRed = get_max(((0.9 * red) + (0.45 * green) + (0.25 * blue))*1.7)
        tGreen = get_max(((0.5 * red) + (0.25 * green) + (0.11 * blue))*.6)
        tBlue = get_max(((0.65 * red) + (0.33 * green) + (0.15 * blue))*.6)
    #Blue sepia
    if value == 2:
        tRed = get_max((0.676 * red) + (0.354 * green) + (0.173 * blue))
        tGreen = get_max((0.524 * red) + (0.277 * green) + (0.136 * blue))
        tBlue = get_max((0.524 * red) + (0.277 * green) + (0.136 * blue))
    # Return sepia color
    return tRed, tGreen, tBlue, alpha

# Convert an image to sepia
def convert_sepia(image, value):
    # Get size
    width, height = image.size
    # Create new Image and a Pixel Map
    new = Image.new("RGB", image.size, "white")
    pixels = new.load()
    # Convert each pixel to sepia
    for i in range(0, width, 1):
        for j in range(0, height, 1):
            p = get_pixel(image, i, j)
            pixels[i, j] = get_sepia_pixel(p[0], p[1], p[2], 255, value)
    # Return new image
    return new
```

An image of the code used to create the various sepia filters (color shifting)



```python
# Get list of versions/filters
id_ls = df[df['Version']=='ORIGINAL']['photo_id'].tolist()
# Randomly choose photos to measure MSE
samples = random.choices(id_ls, k=25)
og_list = []
for id in samples:
    path = f"/content/drive/MyDrive/ACME Fall Projects/Clean Data/ORIGINAL/{id}_ORIGINAL.jpg"
    img = Image.open(path)
    og_list.append(np.array(img))

# Determine if the filters are distinct enough
filter_ls =  [[],[],[],[],[],[]]
filters = ['CONTOUR', 'EMBOSS', 'FIND_EDGES', 'SEPIA_BLUE', 'SEPIA_BROWN', 'SEPIA_RED']
for id in samples:
    for i, p in enumerate(filters):
        path = f"/content/drive/MyDrive/ACME Fall Projects/Clean Data/{p}/{id}_{p}.jpg"
        img = Image.open(path)
        # Get the image ready to process loss
        filter_ls[i].append(np.array(img))

# Process loss
mse_ls = []
for photos in filter_ls:
    temp = []
    for i in range(len(photos)):
        temp.append(MSE(og_list[i], photos[i]))
    mse_ls.append(np.mean(temp))
```

An image of the code used to determine which filters to drop based on MSE loss.