CS156 Final Project: Predicting Fanart Popularity using Artist Attributes and Image Features

Huey Ning Lok

April 26, 2019

1 Problem Statement

Predicting image popularity is an interesting and potentially profitable task considering that many users post images on social media platforms to increase social and commercial influence - the more popular the image, the greater the influence (De Veirman, Cauberghe, & Hudders, 2017). Analyzing image popularity also provides insight into the mindset and proclivity of the audience, and prompts us to consider the various factors involved in promoting image popularity.

In this project, I aim to use machine learning techniques to predict the popularity of artwork scraped from DeviantArt.com - specifically fanart for the Sherlock Holmes fandom. Both artist attributes and the images themselves will be considered when building the predictive model(s).

1.1 Deviantart

Deviantart.com is "the world's largest online social community for artists and art enthusiasts" (Deviantart, 2019). The website was launched on August 7, 2000, and is still up and running with an active community of artists and viewers alike. According to their About page, they have over 44 million registered members and attract over 45 million unique visitors per month. Members upload tens of thousands of original pieces of artwork every day (Deviantart, 2019).

I chose to scrape Deviantart for the data since a large portion of the user-submitted images consist of fanart.

1.2 Fanart

Fan art is artwork created by fans of a work of fiction and derived from a series character or other aspect of that work (Wikipedia contributors, 2019). It is an interesting subcategory of art to explore since it is explicitly derived from a specific fictional work - which has its own artistic style - and seeks to replicate, or at least pay tribute to some elements of the original work.

1.3 Sherlock Holmes

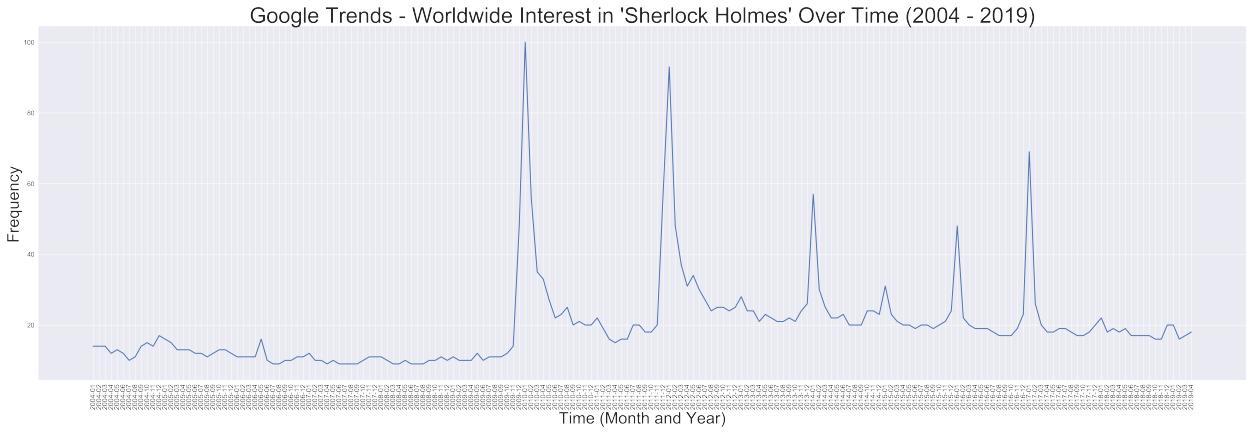
Sherlock Holmes is a fictional private detective created by British author Sir Arthur Conan Doyle(Wikipedia contributors, 2019) According to the Guinness World Records, Sherlock Holmes has the "world record for the most portrayed literary human character in film & TV", having been depicted on screen 254 times (Guinness World Record, 2012). The most recent and arguably most popular adaptations include the 2009 movie adaptation staring Robert Downey Jr. and the 2010 - 2017 BBC One series staring Benedict Cumberbatch.

The graph below depicts the worldwide relative frequency of a Google search for the "Sherlock Holmes" keyword over time (Google Trends, 2019). The popularity of the term peaks at the tail end of 2009 and 2011, which aligns with the release dates of the movie and tv series.

I chose Sherlock Holmes as the target fandom on the basis that it had a large fanbase.

```
In [220]: google_trend_df = pd.read_csv('sherlock-holmes-trend.csv', skiprows=1)
          plt.figure(figsize=(40,12), dpi=250)
          plt.title("Google Trends - Worldwide Interest in 'Sherlock Holmes' Over Time (2004 -
          2019)", size=40)
          plt.xticks(fontsize=13, rotation=90)
          plt.xlabel("Time (Month and Year)", size=30)
          plt.ylabel("Relative Frequency", size=30)
          plt.plot(google_trend_df.iloc[:,0], google_trend_df.iloc[:,1])
```

```
Out[220]: [<matplotlib.lines.Line2D at 0x24c6eb9320>]
```



2 Solution Specification

2.1 Features

2.1.1 Predictors

Prior research on predicting image popularity showed that leveraging both image features as well as social cues allowed the reliable prediction of "the normalized view count of images with a rank correlation of up to 0.81" (Khosla, Das Sarma, & Hamid, 2014).

Intuitively, we are aware that image popularity cannot be predicated on solely social cues or image aesthetics. It is trivial to see that the more popular the content-creator, the more popular their content will be. As for image aesthetics, Dufour (2014) makes a neat statement in his paper on image popularity prediction: "it's clear, at least trivially, that [image popularity] cannot be due entirely to exogenous features (i.e., features that are not contingent on the image itself). As proof, we offer the fact that the majority of imagespace is images of noise, yet very few images of noise are widely popular. Furthermore, images that are objectively aesthetic or attractive—either due to composition or content—are plainly more likely to be popular."

The predictors will thus consist of social cues in the form of artist attributes, and image features extracted via deep learning. A detailed breakdown of the extracted features is shown in the Data Importing section.

2.1.2 Response variable

While there are various metrics to consider when judging image popularity (number of views, comments, favorites, downloads, etc.), I will simply be using number of image favorites as a proxy for popularity. In their study, Khosla et al. used number of image views as a proxy (2014). In future work, one could consider how a mixture of various metrics may lead to the "best" popularity measure.

2.2 Models

2.2.1 Artist Attributes

Four linear regression models are trained as well as a neural network model to compare their performances.[1] The models and their explanations are shown below:

Ordinary Least Squares (OLS) Regression

OLS regression determines parameter values by minimizing the sum of the squared residuals. In a 2D space, the regression can be seen as finding the best-fit line between the datapoints. In higher-dimensional spaces, the best-fit hyperplane is found through minimizing squared residuals. Since there is no bias introduced through a penalty parameter, OLS regression models can have a large amount of variance.

[1] **#regression:** While sklearn's linear regression library makes using various linear regression models a trivial task, it is still important to have an understanding of the differences between each linear regression method used. In my explanations, I detail these differences, suggest using cross-validation for determining penalty terms, and also compare linear regression models to using a neural network model.

Ridge Regression

Ridge regression is only known as Penalized Least Squares (PLS) regression. It introduces a small amount of bias into how the line is fit to the data to significantly drop the variance of the model and minimize overfitting. Ridge regression determines parameter values by minimizing (sum of squared residuals) + (λ x the slope^2) where λ is the regularization parameter and can be determined via cross-validation using sklearn's `RidgeCV()` function.

Lasso Regression

Similar to ridge regresssion, lasso regression introduces bias into the training procedure by minimizing (sum of squared residuals) + (λ x the slope). As seen from the minimizing equation, ridge regression can only shrink the slope asymptotically close to 0, while lasso regression can shrink the slope all the way to 0. As such, lasso regression is superior to ridge regression at reducing variance in models that contain a lot of useless variables. λ can be determined via cross-validation using sklearn's `LassoCV()` function.

Elastic Net Regression

Elastic net regression minimizes the sum of squared residuals + lasso regression penalty + ridge regression penalty. Elastic net regression combines elements of both ridge and lasso by grouping and shrinking the parameters associated with correlated variables, and leaving them in the equation or removing them all at once. The λ's for both the ridge and lasso penalty components can be determined via cross-validation using sklearn's `ElasticNetCV()` function.

Neural Network Model

While the regression models listed above all assume underlying linear relationships within the data, a neural network with a single hidden layer with nonlinear activation functions is considered to be a Universal Function Approximator, i.e. capable of learning any function. Since neural networks are capable of learning highly complex functions, they should perform better if there are many non-linear relationships within the data. Due to the high number of parameters, neural networks can be prone to overfitting, and so a 0.2 cross-validation split is specified for continuous validation throughout the training process.

2.2.2 Image Features

Convolutional Neural Network (CNN)

A convolutional neural network (CNN) is trained to predict image popularity based on image features.

CNNs are appropriate for processing images vs conventional fully-connected neural networks since it enforces a sparse local connectivity pattern between neurons of adjacent layers, i.e. each neuron is connected to only a small region of the input volume (Karpathy, 2016). This takes advantage of the spatial information embedded within the image, and also reduces the number of parameters needed at each layer, allowing deeper networks to be constructed with fewer parameters (Karpathy, 2016). In short, CNNs take advantage of the spatial relationships between an image's pixels to deliver more efficient and more accurate performance.

Transfer Learning

The CNN will be an adaptation of the VGG16 model designed by Simonyan and Zisserman (2014). Distinctive features of the VGG16 model include very deep convolutional layers (up to 19 weight layers) and small convolution filters (3 x 3), which allowed the model to achieve 92.7% top-5 test accuracy in ImageNet Large Scale Visual Recognition Competition (ILSVRC), thus allowing the authors to conclude that "representation depth is beneficial for the classification accuracy" (Simonyan & Zisserman, 2014).

The model also generalises well to a wide range of tasks and datasets. The authors tested the model on a variety of classification tasks outside of ImageNet (which mostly consisted of 10K + images with labels) and found consistent average precision (AP) scores of ~89%.

Since Khosla et al. (2014) built on Krizhevsky, Sutskever, and Hinton's (2012) CNN model (which has about 1 - 5 weight layers) to predict image popularity, and VGG16 has outperformed their original CNN model in various ILSVRC competitions, it would appear to be a good choice to adapt the VGG16 model for my own use in predicting image popularity.

The downside of using VGG16 is that it takes a long time to train (Hassan, 2018), which makes sense considering the deep convolutional layer architecture with roughly 14 million trainable parameters. To reduce the training time, I implement transfer learning by freezing the VGG16 model with 'ImageNet' weights - the weights which were used to classify ImageNet images - removing the top fully-connected layer and subtituting it with my own, and only training the weights in my newly added layers. This reduces the trainable parameters to roughly 2.5 million instead. Due to time constraints, I will also only be training the model for 50 epochs with a batch size of 30 images.

It should be noted that the original model was trained for image classification, while I am using it for a regression problem. While a new output layer with a linear activation function has been added to make continuous predictions, the model's main architecture was still designed with a classification task in mind, as opposed to predicting the value of some metric (in this case, the number of favorites an image will receive). Furthermore, the model was meant to be used on real-life images, while I am using it on fanart, which can be life-like or cartoonish in nature depending on the artist's style.

Since the CNN mostly operates as a black-box, i.e. the predictive power of the image features as calculated by the CNN are hard to interpret. In future work, I could implement techniques of extracting high-level image featues as well such as color intensity, color hue, color contrast, etc. and using them as inputs in more interpretable algorithms such as linear regression, support vector machines, etc. as performed by Khosla et al. (2014).[2]

[2] **#decisionselection:** I elaborated on the decision framework that I followed to decide on using a CNN for predicting image popularity based on image features. Various factors were considered such as the CNN's architectural suitability (which makes the training process both more efficient and accurate), the impressive results from previous deep CNNs built for image classification, the usage of CNNs in a previous task similar to mine (Khosla et al., 2014), and the techniques of transfer-learning and layer-freezing that can be used to save on computational time.

3 Data Collection

To gather the data, I scraped Deviantart for images categorized under the 'sherlock' search query. The exact url used was: https://www.deviantart.com/popular-all-time/?q=sherlock&offset=0. The images were supposedly ordered according to "popular of all time", but I found a mix of images with both high favorite and low favorite counts within my sample size of 5638 images out of the ~190,000 results available. Alongside the images themselves, social metrics in the form of views, downloads, favorites, and various artist attributes were also scraped from the image and artist pages.

The web-scraper was built using a free and open-source Python web-scraping framework, scrapy. I scraped only image posts, and made sure to avoid scraping posts of other formats, e.g. literature. I set a download delay of 250 ms - which had inbuilt scrapy randomization applied to it - as a form of web-scraping etiquette by not downloading a bulk of images continuously, and also to avoid my web-scraper's behavior being obviously detectable as non-human (just in case).

3.1 Number of images downloaded

The reiteration process to fix bugs took time, and my scraper also crashed even on a few bug-free runs, so I settled on downloading 5638 images for now. The scraped images were stored in my local drive under the folder, 'DA-images-2'. The image data was stored in a json file, 'image-data-2'.

A full breakdown of the scraped data can be found below in the Data Importing section. The web-scraper code can be found in the Appendix A.[3]

[3] **#algorithms:** I learned how to use the python web-scraping package, scrapy, to write my own web-scraper for scraping data from DeviantArt. As such, I demonstrated a competence in reading documentation, self-learning, and exploration in order to write an algorithm to automate the scraping process so that data can be collected in high-volume in an efficient and convenient manner for this machine learning project.

4 Data Importing

In [1]:
```
#import libraries
import numpy as np
import pandas as pd
pd.set_option('display.max_colwidth', -1) #for displaying the full image urls so that
they are clickable (if needed)
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(font_scale=1.25)
```

In [2]:
```
df = pd.read_json('image-data-2.json')
df.head(10)
```

Out[2]:

	account_age	artist_asl	artist_comments	\
0	Deviant for 5 Years	Female/France	468	
1	Deviant for 8 Years	Female/Russia	169	
2	Deviant for 6 Years	Female/Japan	23	
3	Deviant for 13 Years	Female/France	322	
4	Deviant for 7 Years	Female/South Korea	436	
5	Deviant for 10 Years	United Kingdom	17	
6	Deviant for 7 Years	28/Female/Russia	8,264	
7	Deviant for 12 Years	France	376	
8	Deviant for 6 Years	Female/United Kingdom	128	
9	Deviant for 7 Years	Female/United States	32	

	artist_critiques	artist_deviations	artist_dob	artist_faves	\
0	0.0	183	March 5	1,830	
1	0.0	36	March 5	216	
2	0.0	28	September 9	53	
3	0.0	55	March 29	396	
4	0.0	73	September 17	37	
5	0.0	77	NaN	6	
6	0.0	150	April 21, 1990	3,760	
7	0.0	13	NaN	114	
8	0.0	97	NaN	0	
9	0.0	8	March 27	30	

	artist_forum_posts	artist_page_views	artist_scraps	...	comments	\
0	0	10,634	0	...	22.0	
1	0	11,377	2	...	45.0	
2	0	9,907	0	...	102.0	
3	0	36,431	3	...	35.0	
4	0	45,119	0	...	8.0	
5	0	18,474	0	...	21.0	
6	0	68,208	0	...	46.0	
7	0	6,566	0	...	5.0	
8	0	29,442	0	...	22.0	
9	0	2,958	2	...	38.0	

	date_posted	downloads	faves	hashtags	\

```
0  January 17, 2014   92.0      142.0   NaN
1  December 30, 2011  186.0     1135.0  NaN
2  August 11, 2012    225.0     1959.0  NaN
3  January 21, 2012   48.0      551.0   NaN
4  August 19, 2011    34.0      91.0    NaN
5  August 1, 2010     59.0      696.0   NaN
6  February 20, 2012  46.0      218.0   NaN
7  January 27, 2012   14.0      96.0    NaN
8  July 13, 2013      8.0       66.0    NaN
9  December 26, 2011  39.0      877.0   NaN


0  https://www.deviantart.com/get-sherlock/art/Sherlock-427568754
1  https://www.deviantart.com/masterhalfling/art/Sherlock-276657827
2  https://www.deviantart.com/donperico/art/SHERLOCK-320465780
3  https://www.deviantart.com/cheeky-bee/art/Sherlock-Of-the-Dead-280821239
4  https://www.deviantart.com/hahaaaaaaaaaaaa/art/Sherlock-253839329
5  https://www.deviantart.com/1stclassstamps/art/Sherlock-Holmes-173600158
6  https://www.deviantart.com/feyjane/art/Sherlock-286116362
7  https://www.deviantart.com/elsias/art/Sherlock-281791771
8  https://www.deviantart.com/sherlockthegame/art/Sherlock-The-Game-Is-On-Character-Cont
9  https://www.deviantart.com/fractionofadot/art/Sherlock-276038401

                                          image_paths  \
0  [full/c11741fda71c47368b3ecc42de439de62c48cf30.jpg]
1  [full/09fbbe05751a1b08cf03445e0821ca44dd5b547b.jpg]
2  [full/0f13dd8004eb7f0ce2b45c826f4e76df5978ac13.jpg]
3  [full/2d3083ae52b6ea7b69fdd83c0f74c5fd8525a489.jpg]
4  [full/be06d405f58195ef9d4c5c42c1c6cb2edb673754.jpg]
5  [full/11dc8450503f97976fba168972410295b76c72ed.jpg]
6  [full/587559fae93b85cbfa8802d39e7a2e1d3cbe415c.jpg]
7  [full/9507a844991ab37afec9bc0a0f619bfa0d033b04.jpg]
8  [full/45901b4a9092329595f2e6739d213b2d169e45f9.jpg]
9  [full/c752fa217f32f5510a49c4f17f3980db9a442c68.jpg]


0  [https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/e9437807-b7f1-4c5b-a26b-f9
1  [https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/dee153a3-1985-4a20-9c5a-87
2  [https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/c0acf0b0-e7e6-4c7c-8ff6-77
3  [https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/bacca111-4256-4446-90e2-4a
4  [https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/75a9b8f3-6cde-4380-a50b-53
5  [https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/0725e00d-6f41-4f48-899c-5c
6  [https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/4da4813c-23ac-4c24-a4af-94
7  [https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/06502dcd-0aae-4239-847b-a3
8  [https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/bff32934-e726-4d2c-85f9-2e
9  [https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/c5b2ce3d-fee1-43ef-b7cd-6c

                                    titles    views
```

```
0  Sherlock                                          1606.0
1  Sherlock                                          14153.0
2  SHERLOCK                                          21163.0
3  Sherlock Of the Dead                              6988.0
4  Sherlock                                          2742.0
5  Sherlock Holmes                                   14566.0
6  Sherlock                                          3139.0
7  Sherlock                                          2208.0
8  Sherlock: The Game Is On (Character Contest)      2983.0
9  Sherlock                                          10189.0

[10 rows x 23 columns]
```

```
In [3]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5638 entries, 0 to 5637
Data columns (total 23 columns):
account_age           5621 non-null object
artist_asl            5621 non-null object
artist_comments       5621 non-null object
artist_critiques      5621 non-null float64
artist_deviations     5621 non-null object
artist_dob            4373 non-null object
artist_faves          5621 non-null object
artist_forum_posts    5621 non-null object
artist_page_views     5621 non-null object
artist_scraps         5621 non-null object
artist_urls           5621 non-null object
artist_watchers       5621 non-null object
artists               5638 non-null object
comments              5629 non-null float64
date_posted           5638 non-null object
downloads             4088 non-null float64
faves                 5636 non-null float64
hashtags              653 non-null object
image_links           5638 non-null object
image_paths           5638 non-null object
image_urls            5638 non-null object
titles                5638 non-null object
views                 5636 non-null float64
dtypes: float64(5), object(18)
memory usage: 1013.2+ KB
```

	Columns	Description	Missing Value Count	Reason for Missing Values
0	account_age	Artist's account age	17	Some images have banned artists, so the artist's account does not exist
1	artist_asl	Artist's age/sex/location	17	Some images have banned artists, so the artist's account does not exist
2	artist_comments	Total number of comments recieved by artist	17	Some images have banned artists, so the artist's account does not exist
3	artist_critiques	Total number of critiques given by artist (Critique is designed to be a thorough evaluation of a deviation on a number of different qualities)	17	Some images have banned artists, so the artist's account does not exist
4	artist_deviations	Artist's Number of posts	17	Some images have banned artists, so the artist's account does not exist
5	artist_dob	Artist's Date of Birth	1265	Some artist's keep their Date of Birth hidden
6	artist_faves	Total number of favorites received by artist	17	Some images have banned artists, so the artist's account does not exist
7	artist_forum_posts	Total number of forum posts made by artist	17	Some images have banned artists, so the artist's account does not exist
8	artist_page_views	Total number of page views received by artist	17	Some images have banned artists, so the artist's account does not exist
9	artist_scraps	Total number of scraps made by artist (Scraps are Works in Progress or Archieved Work)	17	Some images have banned artists, so the artist's account does not exist
10	artist_urls	Link to the artist's account page	17	Some images have banned artists, so the artist's account does not exist
11	artist_watchers	Artist's number of watchers (followers)	17	Some images have banned artists, so the artist's account does not exist
12	artists	Artist's username	0	NA
13	comments	Number of image comments	9	Some images have comments disabled
14	date_posted	The date the image was posted	0	NA
15	downloads	Number of image downloads	1550	Some images have the download function disabled
16	faves	Number of image favorites	2	Oddity - image itself was not downloaded but artist was
17	hashtags	Number of image hashtags	4985	Some images don't have hashtags
18	image_links	Link to the full image page	0	NA
19	image_paths	Image path in local storage	0	NA
20	image_urls	Link to image download	0	NA
21	titles	Image title	0	NA
22	views	Number of image views	2	Oddity - image itself was not downloaded but artist was

5 Data Cleaning and Preprocessing

The dataframe info reveals that some of the columns are not in the right format, and there are also some missing values. Most - if not all - of the missing values can be rationally explained, as shown in the table above. While I would ideally have cleaned all the columns, some of the columns required non-trivial cleaning methods, and so they were omitted due to time constraints. Given additional time and resources, I believe that the "ignored" columns should be cleaned as they could potentially provide significant predictive power to the model, bearing in mind that those with a significant number of missing values may skew the model if an appropriate filtering method is not chosen.

The following section details the cleaning process of all the "cleaned" columns, and the section below this lists the "ignored" columns as well as brief explanations as to why their cleaning process would be non-trivial. In some cases, the portion of non-null objects is too small to justify cleaning the column, e.g. hashtags only has 653 data points out of 5638 within the entire dataset.[4]

[4]**#decisionselection:** The data cleaning and preprocessing step requires the heavy use of appropriate heuristics and frameworks for making decisions. For example, I had to decide on which columns I would keep, would clean, would ignore, etc. I had to weigh the costs and benefits of cleaning columns based on the difficulty of the process, the

5.1 Cleaned Columns

Cleaned columns are columns that were included in the cleaning process. This means that the columns fit the following criteria:

a) need to be cleaned due to improper format;

b) have strong justification for being cleaned (sufficient data points, cleaning process does not take too long, etc.).

Do note that columns which need not be cleaned may still be included in the final dataframe to be analyzed, even if they are not listed. Cleaned columns: account_age, artist_comments, artist_deviations, artist_faves, artist_forum_posts, artist_page_views, artist_scraps, artist_watchers, date_posted.

5.1.1 Account Age

The account age column is cleaned from a string format of "Deviant for -- Years/Months" to be the time delta of the account age in days. There are 17 NaN values leftover after the cleaning process since there are 17 banned artists, aka artists whose accounts have been disabled, and so do not have a valid artist_url to extract data from.

```
In [5]: #cleaning account_age column
        print("Original calue counts in account_age column")
        print(df['account_age'].value_counts(dropna=False))

        #use test_df first since cleaning process is a bit tricky - avoid overwriting original
        df
        test_df = pd.DataFrame()

        test_df['aa'] = df['account_age'].replace(r'^\s*$', np.nan, regex=True) #replace empty
        spaces with nan
        test_df['value'] = test_df['aa'].str.extract('(\d+)').astype(float) #extract time value
        test_df['time_unit'] = test_df['aa'].str[-6:] #extract time unit
        test_df['time_unit'] = test_df['time_unit'].str.strip() #strip time unit of any blank
        spaces

        #if time unit is months, convert value to month time delta, else convert to year time
        delta
        test_df['value'] = np.where(test_df['time_unit']=='Months',
                                    pd.to_timedelta(test_df['value'], unit='M'),
                                     pd.to_timedelta(test_df['value'], unit='Y'))

        print()
        print("Cleaned value counts")
        print(test_df['value'].value_counts(dropna=False).sort_values(ascending=False))

        #lastly, assign cleaned column back to original df
        df['account_age'] = test_df['value']
```

significance of the column, the number of data points, etc. Each choice I made was carefully reasoned and justified so that my decision path is transparent and understandable, both for myself and the audience.

```
Original value counts in account_age column
Deviant for 7  Years      873
Deviant for 8  Years      830
Deviant for 9  Years      691
Deviant for 10  Years     617
Deviant for 11  Years     570
Deviant for 6  Years      465
Deviant for 12  Years     412
Deviant for 5  Years      281
Deviant for 13  Years     256
Deviant for 14  Years     192
Deviant for 15  Years     157
Deviant for 4  Years      111
Deviant for 3  Years      53
Deviant for 16  Years     50
Deviant for 2  Years      39
NaN                       17
Deviant for 1  Year       13
Deviant for 17  Years     6
Deviant for 18  Years     1
Deviant for 7  Months     1
Deviant for 9  Months     1
Deviant for 10  Months    1
Deviant for 5  Months     1
Name: account_age, dtype: int64

Cleaned value counts
2556 days 16:44:24    873
2921 days 22:33:36    830
3287 days 04:22:48    691
3652 days 10:12:00    617
4017 days 16:01:12    570
2191 days 10:55:12    465
4382 days 21:50:24    412
1826 days 05:06:00    281
4748 days 03:39:36    256
5113 days 09:28:48    192
5478 days 15:18:00    157
1460 days 23:16:48    111
1095 days 17:27:36    53
5843 days 21:07:12    50
730 days 11:38:24     39
NaT                   17
365 days 05:49:12     13
6209 days 02:56:24    6
6574 days 08:45:36    1
213 days 01:23:42     1
304 days 08:51:00     1
273 days 22:21:54     1
152 days 04:25:30     1
Name: value, dtype: int64
```

5.1.2 Date posted

The date posted column is cleaned by converting it from string to a datetime object. A new column, "image_age" (whose unit is in days) is created by taking the difference between the date of the latest image posted in the dataset and the image's own posted date. This means that older images will have larger "image_age" values, while newer images will have smaller "image_age" values, with the newest image having an "image_age" of 0.

The "image_age" column has 14 NaN values since some of the dates posted do not contain years, and so an error is produced when trying to calculate the image age. These errors are coerced to a Nan format.

```
In [6]: #cleaning date_posted column by converting dates to datetime
        #a few dates don't have years, which makes their "age" quite meaningless.
        #The errors are coerced so produce NaN outputs, which will later be removed from the
        dataset.
        test_df['date_posted'] = pd.to_datetime(df['date_posted'], errors='coerce')

        #get number of days between image post date and newest post date in dataset
        test_df['image_age'] = (test_df['date_posted'].max() - test_df['date_posted']).dt.days
        print(f"Newest date in dataset: {test_df['date_posted'].max()}")
        print(f"Number of Nans: {sum(test_df['image_age'].isna())}")

        #assign image_age and date_posted column to actual df
        df['image_age'] = test_df['image_age']
        df['date_posted'] = test_df['date_posted']

Newest date in dataset: 2018-12-30 00:00:00
Number of Nans: 14
```

5.1.3 Artist attributes, Image paths and urls

The artist attribute columns and image paths and url columns have a rather trivial cleaning process. For the artist attribute columns, I just removed all commas and converted the strings to floats. The image_paths and image_urls columns consisted of a single element within a list, so I just extracted the element for easier processing later on.

```
In [7]: #cleaning artist attributes - done in bulk because trivial
        df['artist_comments'] = df['artist_comments'].str.replace(',', '').astype(float)
        df['artist_deviations'] = df['artist_deviations'].str.replace(',', '').astype(float)
        df['artist_faves'] = df['artist_faves'].str.replace(',', '').astype(float)
        df['artist_forum_posts'] = df['artist_forum_posts'].str.replace(',', '').astype(float)
        df['artist_page_views'] = df['artist_page_views'].str.replace(',', '').astype(float)
        df['artist_scraps'] = df['artist_scraps'].str.replace(',', '').astype(float)
        df['artist_watchers'] = df['artist_watchers'].str.replace(',', '').astype(float)

        print('Artist attributes cleaned.')

        #clean image_paths and image_urls by extracting the string from list
        df['image_paths'] = df['image_paths'].str[0]
        df['image_urls'] = df['image_urls'].str[0]

        print('Image path and url cleaned.')

Artist attributes cleaned.
Image path and url cleaned.
```

5.2 Ignored columns

Ignored columns are columns that were omitted from the cleaning process despite needing to be cleaned, usually because the cleaning process is too time-consuming (or beyond my skill-level), or the column has too few data points. There are some columns that will be ultimately ignored in the final process, but are not listed here since the data was in the right format (aka they do not need to be cleaned).

'artist_asl'

Data type: Array of the age, gender, and country of the artist.

This is a non-trivial cleaning process since artists may choose to list none to all of the categories listed above, and so trying to parse the list order and assign elements to the proper categories of age, gender, and country can be challenging. Some of the values are also 'Unknown'.

'artist_dob'

Data type: A string of the artist's date of birth.

The column was ignored for two reasons. First, there are only 2678 non-null entries. Second, a good amount of entries do not contain a year within the birthdate, which one would assume to have the most potential significant influence.

'hashtags'

Data type: Array of hashtags posted with the image entry.

The column has only 653 non-null entries, and so was omitted. I also realized later on that I did not scrape the hashtag field properly, and so when the hashtags exceed a certain number, the "(show more)" string is extracted instead of the additional hidden hashtags.

In future work, it would be interesting to use the number of hashtags as a potential feature, or use natural language processing techniques to identify patterns in the hashtags.

```
In [8]: #dropna to demonstrate the ignore columns data formats
        df[['artist_asl','artist_dob','hashtags']].dropna().head(10)
```

Out[8]:

	artist_asl	artist_dob	\
25	30/Female/Russia	February 26, 1989	
51	Female/People's Republic of China	June 3	
61	Male/United Kingdom	August 10	
64	Other/Spain	May 8	
77	Female/Spain	November 11	
85	Male/Sweden	June 26	
87	France	July 19	
90	Unknown	October 23	
97	Female/United States	November 18	
101	19/Female/South Korea	September 11, 1999	

25	[tv, thelastproblem, art, amazing, artdigital, bbc, beautiful, blackandwhite, (show
51	[sherlock, sherlockholmes, johnlock, bbcsherlock]
61	[portrait, sherlock, sherlockholmes, benedictcumberbatch, fanarttraditionalart]
64	[bbc, benedict, portrait, sherlock, sweet, sherlockbbc, benedictcumberbach, sherloc
77	[sherlock, sherlockholmes, watson, martinfreeman, benedictcumberbatch, sherlockbbc,
85	[bbc, design, holmes, max, sherlock, statue, grecke]
87	[fanart, sherlock, sherlockholmes, benedictcumberbatch]
90	[sherlock, sherlockholmes, benedictcumberbatch, sherlockbbc, sherlockfanart, benedi
97	[dark, drawing, gimp, krita, male, man, mysterious, painting, (show more)]
101	[01, edit, kpop, mmd, sherlock, shinee, tda, vocaloid, (show more)]

5.3 Final Dataframe

In the final iteration of dataframe cleaning, the 'Ignored' columns were removed. The 'downloads' column was also removed since it has only 4088 non-null data points.

'faves' is intended as the predicted variable. Although some of the columns such as 'comments' and 'views' can't fairly be used in the predictive process (since one does not know the number of views and comments of an image beforehand), they are still retained for analysis of their distributions and correlations.

```
In [9]: #final df
        final_df = df[['account_age','artist_comments','artist_critiques','artist_deviations','a
        rtist_faves',
                       'artist_forum_posts', 'artist_page_views', 'artist_scraps',
        'artist_watchers', 'artists',
                       'comments', 'faves', 'titles', 'views', 'date_posted','image_age',
                       'image_links','image_paths','image_urls','artist_urls']]
        final_df.head(10)
```

Out[9]:

	account_age	artist_comments	artist_critiques	artist_deviations	\
0	1826 days 05:06:00	468.0	0.0	183.0	
1	2921 days 22:33:36	169.0	0.0	36.0	
2	2191 days 10:55:12	23.0	0.0	28.0	
3	4748 days 03:39:36	322.0	0.0	55.0	
4	2556 days 16:44:24	436.0	0.0	73.0	
5	3652 days 10:12:00	17.0	0.0	77.0	
6	2556 days 16:44:24	8264.0	0.0	150.0	
7	4382 days 21:50:24	376.0	0.0	13.0	
8	2191 days 10:55:12	128.0	0.0	97.0	
9	2556 days 16:44:24	32.0	0.0	8.0	

	artist_faves	artist_forum_posts	artist_page_views	artist_scraps	\
0	1830.0	0.0	10634.0	0.0	
1	216.0	0.0	11377.0	2.0	
2	53.0	0.0	9907.0	0.0	
3	396.0	0.0	36431.0	3.0	
4	37.0	0.0	45119.0	0.0	
5	6.0	0.0	18474.0	0.0	
6	3760.0	0.0	68208.0	0.0	
7	114.0	0.0	6566.0	0.0	
8	0.0	0.0	29442.0	0.0	
9	30.0	0.0	2958.0	2.0	

	artist_watchers	artists	comments	faves	\
0	54.0	get-sherlock	22.0	142.0	
1	135.0	masterHalfling	45.0	1135.0	
2	198.0	DonPerico	102.0	1959.0	
3	1101.0	Cheeky-Bee	35.0	551.0	
4	1118.0	HAHAAAAAAAAAAAA	8.0	91.0	
5	47.0	1stClassStamps	21.0	696.0	
6	1190.0	Feyjane	46.0	218.0	

```
7  13.0             Elsias           5.0       96.0
8  706.0            SherlockTheGame  22.0      66.0
9  22.0             fractionofadot   38.0      877.0

                                          titles    views date_posted  \
0  Sherlock                                       1606.0  2014-01-17
1  Sherlock                                       14153.0 2011-12-30
2  SHERLOCK                                       21163.0 2012-08-11
3  Sherlock Of the Dead                           6988.0  2012-01-21
4  Sherlock                                       2742.0  2011-08-19
5  Sherlock Holmes                                14566.0 2010-08-01
6  Sherlock                                       3139.0  2012-02-20
7  Sherlock                                       2208.0  2012-01-27
8  Sherlock: The Game Is On (Character Contest)   2983.0  2013-07-13
9  Sherlock                                       10189.0 2011-12-26

   image_age  \
0  1808.0
1  2557.0
2  2332.0
3  2535.0
4  2690.0
5  3073.0
6  2505.0
7  2529.0
8  1996.0
9  2561.0


0  https://www.deviantart.com/get-sherlock/art/Sherlock-427568754
1  https://www.deviantart.com/masterhalfling/art/Sherlock-276657827
2  https://www.deviantart.com/donperico/art/SHERLOCK-320465780
3  https://www.deviantart.com/cheeky-bee/art/Sherlock-Of-the-Dead-280821239
4  https://www.deviantart.com/hahaaaaaaaaaaaa/art/Sherlock-253839329
5  https://www.deviantart.com/1stclassstamps/art/Sherlock-Holmes-173600158
6  https://www.deviantart.com/feyjane/art/Sherlock-286116362
7  https://www.deviantart.com/elsias/art/Sherlock-281791771
8  https://www.deviantart.com/sherlockthegame/art/Sherlock-The-Game-Is-On-Character-Cont
9  https://www.deviantart.com/fractionofadot/art/Sherlock-276038401

                                          image_paths  \
0  full/c11741fda71c47368b3ecc42de439de62c48cf30.jpg
1  full/09fbbe05751a1b08cf03445e0821ca44dd5b547b.jpg
2  full/0f13dd8004eb7f0ce2b45c826f4e76df5978ac13.jpg
3  full/2d3083ae52b6ea7b69fdd83c0f74c5fd8525a489.jpg
4  full/be06d405f58195ef9d4c5c42c1c6cb2edb673754.jpg
5  full/11dc8450503f97976fba168972410295b76c72ed.jpg
6  full/587559fae93b85cbfa8802d39e7a2e1d3cbe415c.jpg
```

```
7  full/9507a844991ab37afec9bc0a0f619bfa0d033b04.jpg
8  full/45901b4a9092329595f2e6739d213b2d169e45f9.jpg
9  full/c752fa217f32f5510a49c4f17f3980db9a442c68.jpg


0  https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/e9437807-b7f1-4c5b-a26b-f9e
1  https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/dee153a3-1985-4a20-9c5a-872
2  https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/c0acf0b0-e7e6-4c7c-8ff6-776
3  https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/bacca111-4256-4446-90e2-4ae
4  https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/75a9b8f3-6cde-4380-a50b-53a
5  https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/0725e00d-6f41-4f48-899c-5c8
6  https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/4da4813c-23ac-4c24-a4af-948
7  https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/06502dcd-0aae-4239-847b-a33
8  https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/bff32934-e726-4d2c-85f9-2ed
9  https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/c5b2ce3d-fee1-43ef-b7cd-6c8

                                artist_urls
0  https://www.deviantart.com/get-sherlock
1  https://www.deviantart.com/masterhalfling
2  https://www.deviantart.com/donperico
3  https://www.deviantart.com/cheeky-bee
4  https://www.deviantart.com/hahaaaaaaaaaaaa
5  https://www.deviantart.com/1stclassstamps
6  https://www.deviantart.com/feyjane
7  https://www.deviantart.com/elsias
8  https://www.deviantart.com/sherlockthegame
9  https://www.deviantart.com/fractionofadot
```

In [10]:
```
final_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5638 entries, 0 to 5637
Data columns (total 20 columns):
account_age           5621 non-null timedelta64[ns]
artist_comments       5621 non-null float64
artist_critiques      5621 non-null float64
artist_deviations     5621 non-null float64
artist_faves          5621 non-null float64
artist_forum_posts    5621 non-null float64
artist_page_views     5621 non-null float64
artist_scraps         5621 non-null float64
artist_watchers       5621 non-null float64
artists               5638 non-null object
comments              5629 non-null float64
faves                 5636 non-null float64
titles                5638 non-null object
views                 5636 non-null float64
date_posted           5624 non-null datetime64[ns]
image_age             5624 non-null float64
image_links           5638 non-null object
image_paths           5638 non-null object
image_urls            5638 non-null object
artist_urls           5621 non-null object
dtypes: datetime64[ns](1), float64(12), object(6), timedelta64[ns](1)
memory usage: 881.0+ KB
```

5.4 Filling NaNs with Column Means

As a final step in the cleaning process, all missing values are filled in with the column mean. This leaves 14 NaN values in the date_posted column and 17 NaN values in the artist_urls columns since there is no "mean" for these columns. These NaN values can be safely ignored since the columns will not be used in the regression analysis.

```
In [11]: print(f"Number of Nans in DataFrame:\n\n{final_df.isna().sum()}")

         #fill all NaN values with the column's mean amount
         final_df = final_df.fillna(final_df.mean())
```

```
Number of Nans in DataFrame:

account_age           17
artist_comments       17
artist_critiques      17
artist_deviations     17
artist_faves          17
artist_forum_posts    17
artist_page_views     17
artist_scraps         17
artist_watchers       17
artists                0
comments               9
faves                  2
titles                 0
views                  2
date_posted           14
image_age             14
image_links            0
image_paths            0
image_urls             0
artist_urls           17
dtype: int64
```

```
In [12]: print(f"Final Number of Nans in DataFrame after filling missing values with column
         mean:\n\n{final_df.isna().sum()}")
```

```
Final Number of Nans in DataFrame after filling missing values with column mean:

account_age           0
artist_comments       0
artist_critiques      0
artist_deviations     0
artist_faves          0
artist_forum_posts    0
artist_page_views     0
artist_scraps         0
artist_watchers       0
artists               0
comments              0
faves                 0
titles                0
views                 0
date_posted          14
image_age             0
image_links           0
image_paths           0
image_urls            0
artist_urls          17
dtype: int64
```

6 Data Analysis

The following analyses wil be conducted to gain a better understanding of the data that is being handled:

- Analyzing descriptive statistics.
- Analyzing the distributions and correlations in the data.
- Analyzing any variables which contain a time-series component. In this case, only date_posted has a time-series component.

6.1 Descriptive Statistics

The descriptive statistics dataframes below allow a more detailed insight into the data.[5] Some of the more salient issues are listed below:

- The artist_critiques, artist_forum_posts, and artist_scraps columns contain a lot of zeros. This suggests that most people on DeviantArt - or at least those within the current sample - do not make much use of these functionalities. The lack of use of these functionalities can be explained below:
  - 'artist_critique': The regular user would use DeviantArt to browse art, comment, and favorite it. Making a critique takes additional effort since one has to adhere to several guidelines to do so.
  - 'artist_forum_posts': Forums are not that popular since most people come to the site to view and make art vs having discussions.
  - 'artist_scraps': Most users aren't inclined to post their works-in-progress or to archive their art in "Scraps", and instead choose to leave everything in the main gallery.
- There are 17 banned artists, which explains the missing artist links above.

[5] **#descriptivestats:** Analyzing descriptive statistics of the data is an important step to gain a better understanding of the data, unveil insights, and detect anomalies. In this case, analyzing the descriptive statistics of the artist attribute columns allowed me to point out some salient issues, and also provided insight into the missing artist links. A brief glance at the mean, min, and max also provides us with a good outline of the basic statistics of the data.

```
In [13]: final_df.describe().round(2)
```

Out[13]:

	account_age	artist_comments	artist_critiques	\
count	5638	5638.00	5638.00	
mean	3283 days 18:38:14.697384	4959.48	0.26	
std	1029 days 09:17:57.753093	16813.99	1.47	
min	152 days 04:25:30	0.00	0.00	
25%	2556 days 16:44:24	384.25	0.00	
50%	3287 days 04:22:48	1481.50	0.00	
75%	4017 days 16:01:12	4812.50	0.00	
max	6574 days 08:45:36	987561.00	34.00	

	artist_deviations	artist_faves	artist_forum_posts	artist_page_views	\
count	5638.00	5638.00	5638.00	5638.00	
mean	308.08	3889.57	57.73	113944.25	
std	713.35	18723.56	1112.49	801727.25	
min	0.00	0.00	0.00	336.00	
25%	69.25	228.25	0.00	11109.00	
50%	164.00	869.00	0.00	25035.50	
75%	344.00	2887.75	2.00	64305.25	
max	34267.00	906201.00	79536.00	44584418.00	

	artist_scraps	artist_watchers	comments	faves	views	\
count	5638.00	5638.00	5638.00	5638.00	5638.00	
mean	20.35	1567.65	23.85	145.73	3375.18	
std	141.22	9501.20	43.72	396.01	7041.45	
min	0.00	0.00	0.00	7.00	22.00	
25%	0.00	55.00	5.00	33.00	687.00	
50%	1.00	208.00	12.00	57.00	1435.00	
75%	12.00	821.00	26.00	127.00	3204.50	
max	8678.00	555719.00	1374.00	13066.00	165273.00	

	image_age
count	5638.00
mean	2225.46
std	658.55
min	0.00
25%	1824.00
50%	2345.00
75%	2539.00
max	5568.00

```
In [14]: final_df.describe(include=['O'])
```

```
Out[14]:         artists    titles  \
          count    5638    5638
          unique   5622    4073
          top      Banned  Sherlock
          freq     17      644

                                                                                image_links  \
          count    5638
          unique   5638
          top      https://www.deviantart.com/jbshlover/art/JB-as-Sherlock-Holmes-163778585
          freq     1

                                                          image_paths  \
          count    5638
          unique   5638
          top      full/ffcdbdde1b0ff5e313500a82eb1f98193bbb0f54.jpg
          freq     1


          count    5638
          unique   5638
          top      https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/33c52cee-2f10-42d7-9f
          freq     1

                                       artist_urls
          count    5621
          unique   5621
          top      https://www.deviantart.com/leafbreeze7
          freq     1
```

6.2 Distributions and Correlations

6.2.1 Predictor Features

The final predictors to be used in the regression analysis are 'artist_comments', 'artist_critiques', 'artist_deviations', 'artist_faves', 'artist_forum_posts', 'artist_page_views', 'artist_scraps', and 'artist_watchers'. 'faves' - the number of image favorites - is used as a proxy for popularity and represents the response variable.

An unnormalized pairplot reveals that all of the predictors - and the response variables - have an exponential distribution. As such, all the variables are log-normalized so that there is less skew in their distributions. This should improve the model-training process since the model will not be misled by stark outliers.[6]

The log-normalized pairplot shows that most of the distributions have been successfully normalized to resemble a Gaussian distribution, although the artist_critiques, forum_posts, and artist_scraps still skew heavily to 0, as expected due to the large number of zeros within those columns.

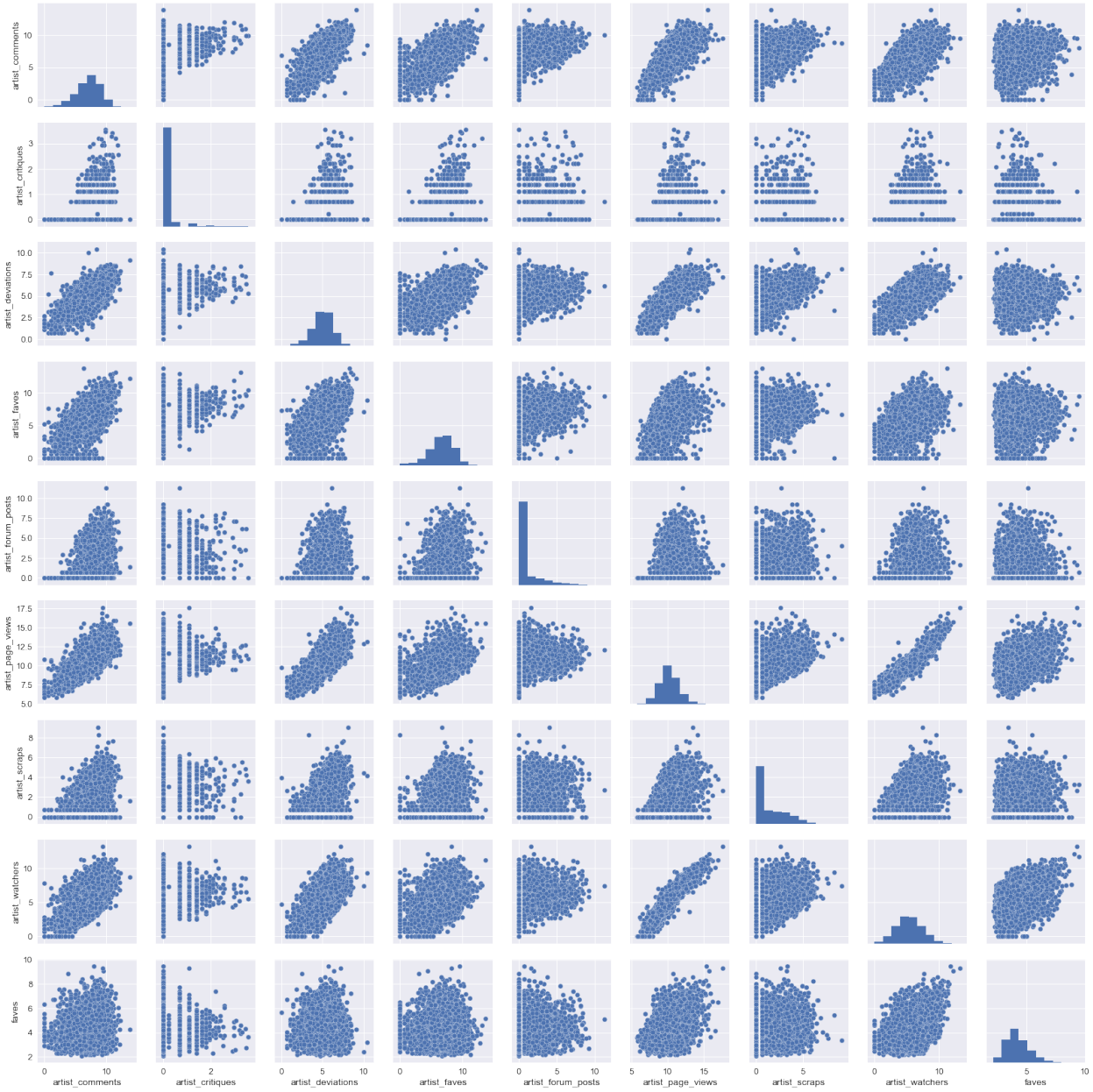
[6] **#distributions:** By analyzing the distributions of the data, we can quickly see whether the data is normal, heavily skewed, etc. In this case, all the predictor columns - including the response column - turn out to be exponentially distributed, and so the data was log-normalized to resemble normal distributions instead. Analyzing distributions and normalizing those that are heavily skewed are vital steps in the machine learning process to help increase the model's accuracy.

In [15]:
```
#predictors are features that the artist knows before posting an image (faves is also
included as the response variable)
predictors = ['artist_comments','artist_critiques','artist_deviations','artist_faves','a
rtist_forum_posts',
              'artist_page_views','artist_scraps','artist_watchers','faves']

sns.pairplot(final_df[predictors])
```

Out[15]: <seaborn.axisgrid.PairGrid at 0x1a170aedd8>

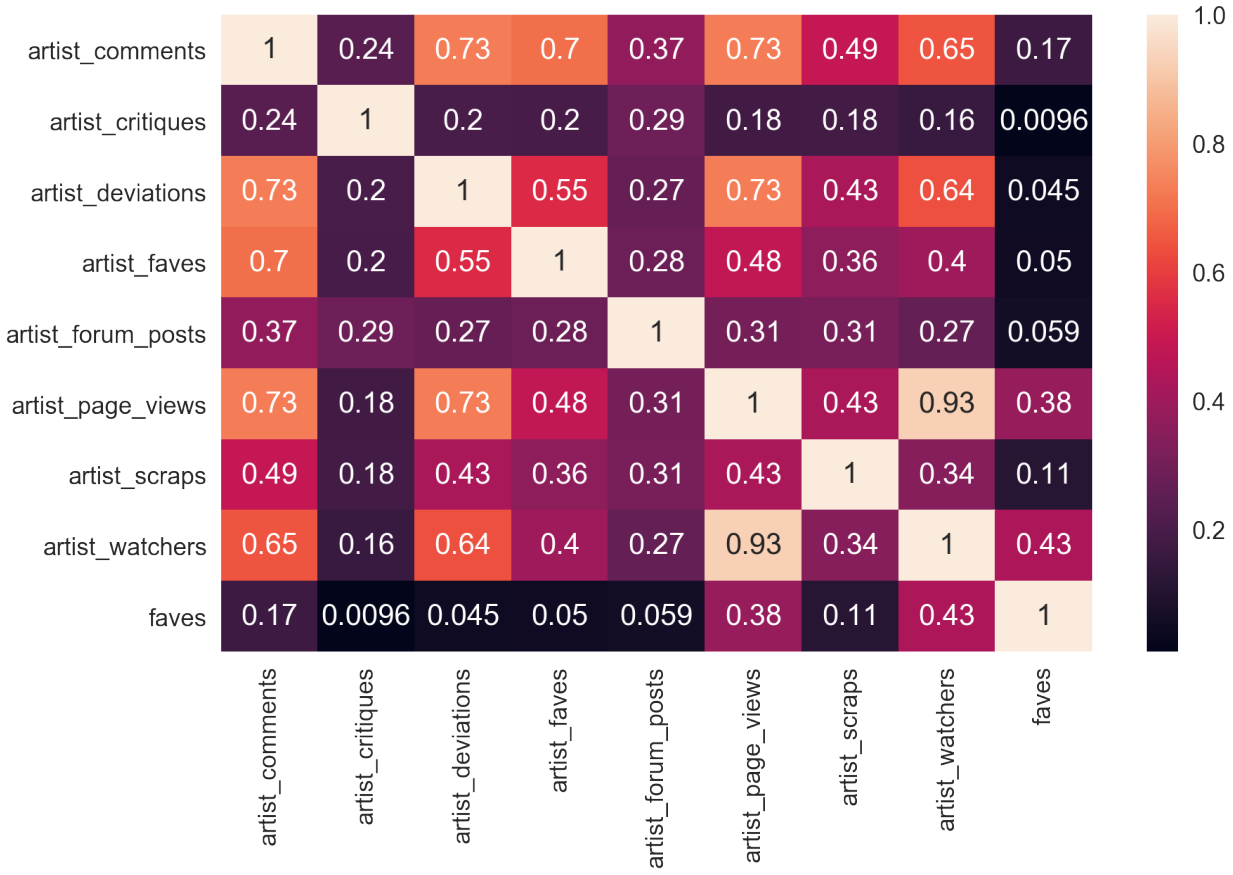
In [16]:
```
log_predictors_df = np.log(final_df[predictors] + 1)

sns.pairplot(log_predictors_df)
```

Out[16]: <seaborn.axisgrid.PairGrid at 0x1a18d28b38>



```
In [17]: corr = log_predictors_df.corr()
         plt.figure(figsize = (10,6), dpi=200)
         sns.heatmap(corr,
                     xticklabels=corr.columns.values,
                     yticklabels=corr.columns.values,
                     annot=True, annot_kws={"size": 15})
```

```
Out[17]: <matplotlib.axes._subplots.AxesSubplot at 0x1a26cdbb70>
```



6.2.2 Correlation Heatmap Analysis

- The number of watchers and total page views has the highest correlations with the number of favorites received by a particular image, with correlation coefficients of 0.43 and 0.38 respectively. This makes sense since the more watchers (aka followers) an artist has and the higher the frequency their account page is viewed, the higher the probability of them being a "popular" artist whose art is appreciated and favorited.

- Although the number of deviations (images posted) has a high correlation with number of page views (0.73), the correlation between number of deviations and number of favorites on a given image is only 0.045. This suggests that when artists produce a lot of artwork, their pages tend to be viewed very often, but this doesn't necessarily mean that people like their work, i.e. quantity does not mean quality.

- The number of total favorites received by an artist across all their work has a weak correlation (0.05) with the number of favorites for a given image. This is probably because although an artist has a high amount of total favorites, it could be due to them accruing favorites through image quantity. More deviations - albeit with less favorites per deviation - could possibly accrue as many likes as a handful of high-favorite deviations.

There are many other insights to be gathered from the heatmap, and most make intuitive sense as to what one would expect from the data.

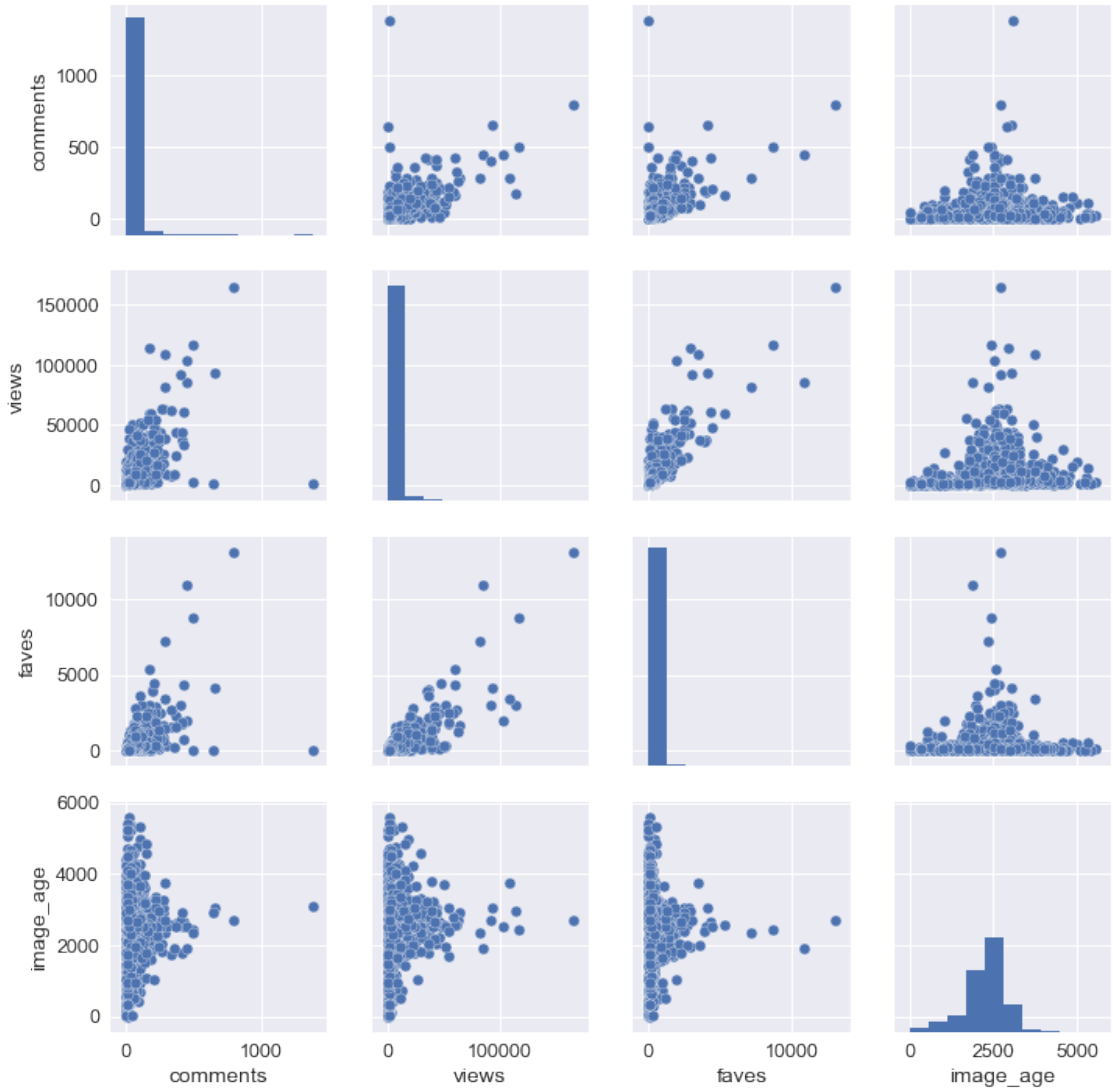
6.2.3 Other Features

Other features are social metrics that should not be used for prediction since they are unknown at the time of image posting, but are nonetheless interesting to examine for insights. These consist of the number of comments, number of views, the image age, and once again, the number of image favorites.

The pairplots show that the comments, views, and favorites and exponentially distribution, while the image age is normally distributed. As such, the comments, views, and favorites are log-normalized to resemble Gaussian distributions before building the correlation heatmap.

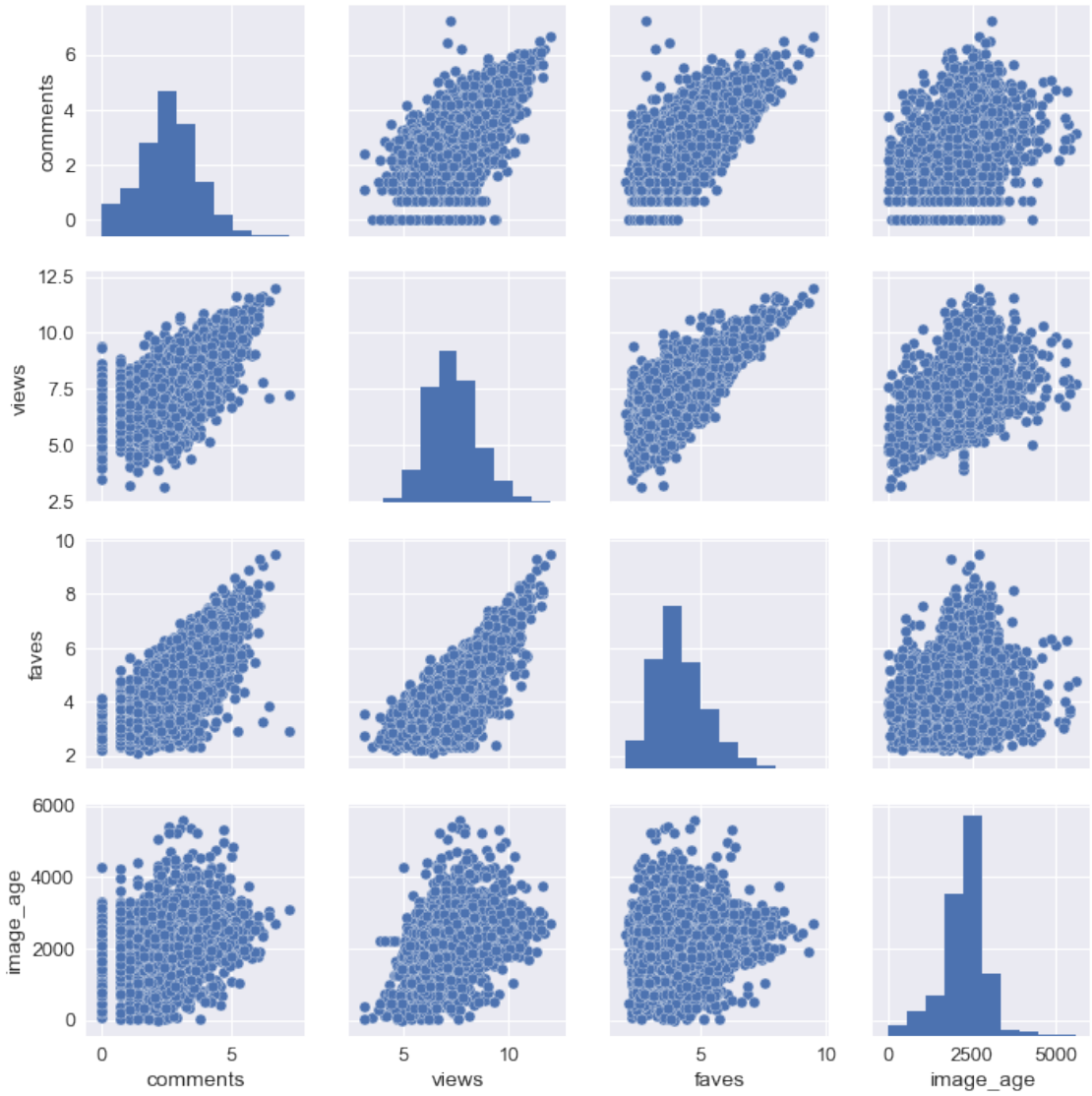
```
In [18]: #these other features are only known after posting
         others = ['comments','views','faves','image_age']
         sns.pairplot(final_df[others])
```

```
Out[18]: <seaborn.axisgrid.PairGrid at 0x1a2733f438>
```



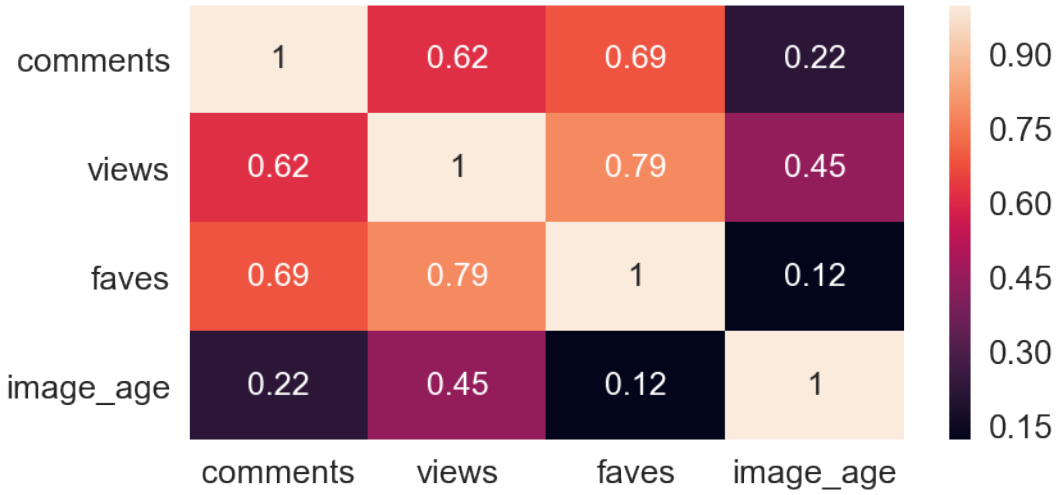
In [19]:
```
#only log comments, faves, and views since image_age is already normally distributed
log_others_df = np.log(final_df[['comments','views','faves']] + 1)
log_others_df = pd.concat([log_others_df, final_df['image_age']], axis = 1)
sns.pairplot(log_others_df)
```

Out[19]: <seaborn.axisgrid.PairGrid at 0x1a29acc4e0>



```
In [49]: corr = log_others_df.corr()
         plt.figure(figsize = (6,3), dpi=150)
         sns.heatmap(corr,
                     xticklabels=corr.columns.values,
                     yticklabels=corr.columns.values,
                     annot=True, annot_kws={"size": 12})
```

```
Out[49]: <matplotlib.axes._subplots.AxesSubplot at 0x1a3cf08ac8>
```



6.2.4 Correlation Heatmap Analysis

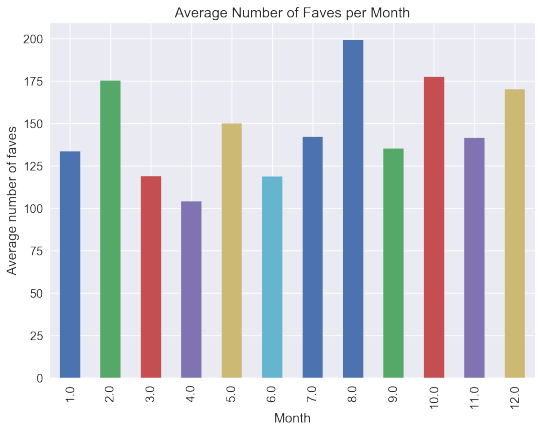
The correlation heatmap shows that there are high correlations between comments, views, and faves; and a moderate correlation between image age and views, but low correlations between image age and comments and faves. This suggests that although older images tend to accrue views over time, this does not necessarily translate to additional comments and favorites.

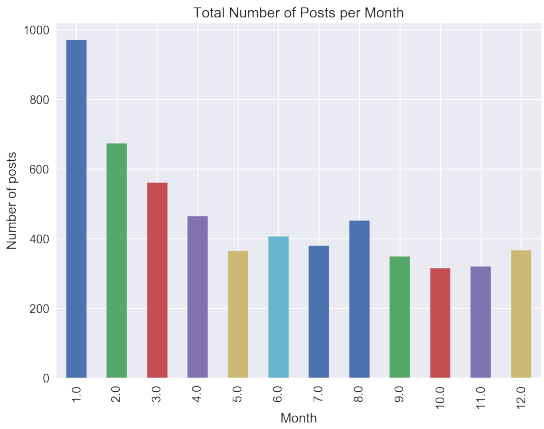
6.3 Time-series of Date Posted

The time-series of the dates the images were posted are analyzed to see whether there are any significant time-dependent trends within the data. All observations conducted are constrained in their generalizability due to the small sample size of data (5638 images).

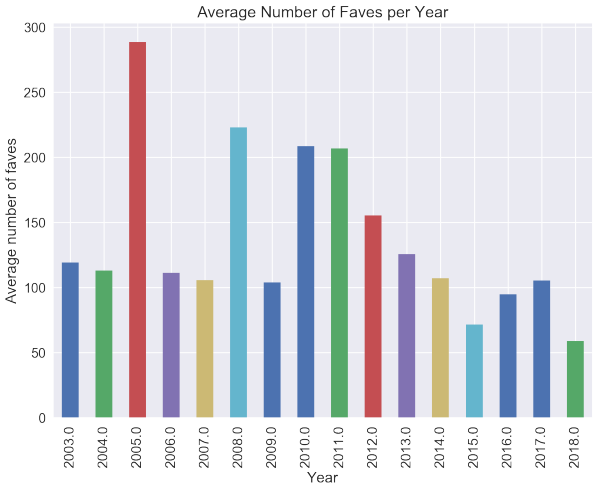
```
In [203]: #create month-year column to get monthly time periods
          final_df['month_posted'] = final_df['date_posted'].dt.month
          final_df['year_posted'] = final_df['date_posted'].dt.year
          final_df['month_year_posted'] = final_df['date_posted'].dt.to_period('M')

          def create_time_series(timescale, title, label_size=14, title_size=15, x_tick_size=13,
          side_plots = True):

              '''
              Inputs:
                  timescale (str): name of timescale column to be used in groupby function
                                   to get average faves per post and number of posts

                  title (str): name of timescale to appear in the title and x-axis label

                  label_size (int): x-axis and y-axis font size. Default: 13

                  title_size (int): title font size. Default: 15

                  x_tick_size (int): x-ticks font size. Default: 13

                  side_plots (bool): if True, plots will be displayed side by side.
                                     If False, plots will be displayed top and bottom.
                                     Default: True

              Outputs:
                  Two bar plots showing the average number of faves and number of posts
                  over the given timescale.
              '''
              if side_plots:
                  a = 221
                  b = 222
              else:
                  a = 211
                  b = 212

              #make df of average faves and number of posts over timescale
              avg_faves = final_df.groupby(final_df[timescale]).mean()
              no_of_posts = final_df.groupby(final_df[timescale]).count()

              #plot the dfs
              plt.subplot(a)
              avg_faves['faves'].plot(kind='bar')
              plt.title(f"Average Number of Faves per {title}", fontsize=title_size)
              plt.xlabel(f"{title}", fontsize=label_size)
              plt.xticks(fontsize=x_tick_size, rotation=90)
              plt.ylabel("Average number of faves", fontsize=label_size)

              plt.subplot(b)
              no_of_posts['faves'].plot(kind='bar')
              plt.title(f"Total Number of Posts per {title}", fontsize=title_size)
              plt.xlabel(f"{title}", fontsize=label_size)
              plt.xticks(fontsize=x_tick_size, rotation=90)
              plt.ylabel("Number of posts", fontsize=label_size)
```

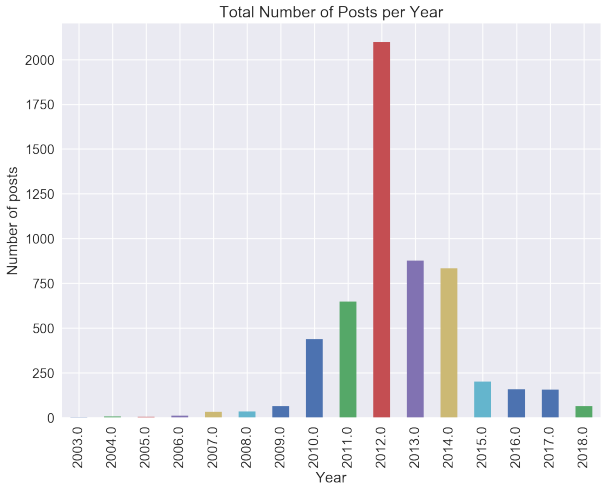
In [204]:
```
plt.figure(figsize=(20,15), dpi=200)
create_time_series('month_posted', 'Month')
```



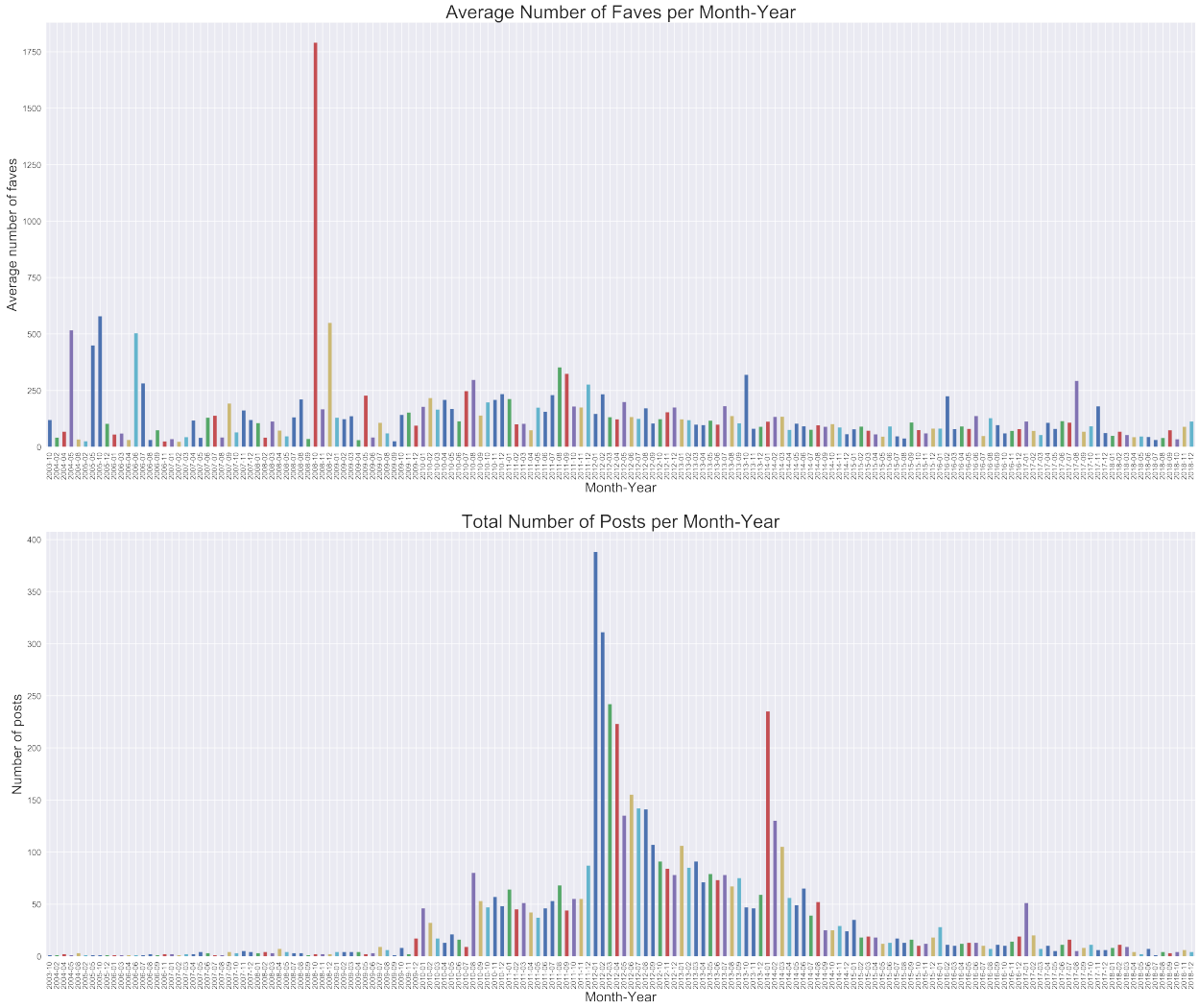


In [205]:
```
plt.figure(figsize=(20,15), dpi=200)
create_time_series('year_posted', 'Year')
```





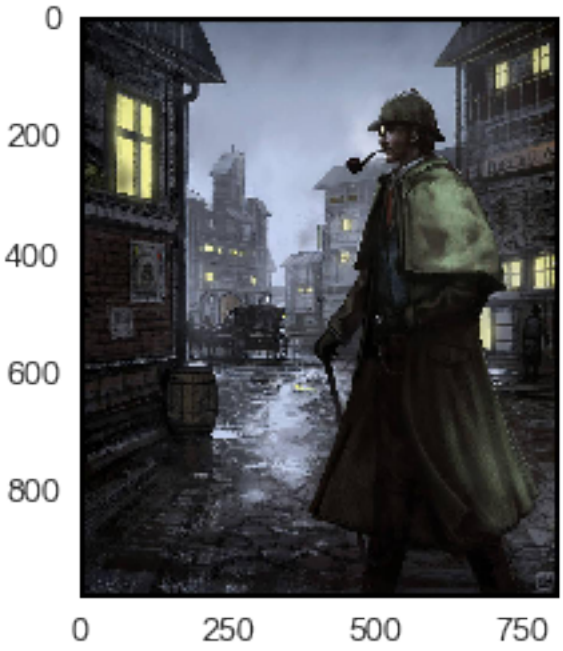
In [206]:
```
plt.figure(figsize=(30,25), dpi=300)
create_time_series('month_year_posted', 'Month-Year', label_size=20, title_size=27,
x_tick_size=11, side_plots=False)
```



```
In [25]: from datetime import datetime
         import matplotlib.image as mpimg
         from IPython.core.display import HTML

         #find the outlier
         outlier_date = datetime.strptime('2008-10','%Y-%m')
         outlier_df =
         final_df[['faves','artists','titles','image_paths','image_links']].loc[np.where((
             final_df['year_posted'] == outlier_date.year) & (final_df['month_posted'] ==
         outlier_date.month))]
         display(HTML(outlier_df.to_html()))

         print(outlier_df['image_links'])
         img = mpimg.imread('DA-images/'+ 'full/cf5fd090c1f8cb96bef89b500abb6974e66cb600.jpg')
         plt.imshow(img)
         plt.grid(False)
```

<IPython.core.display.HTML object>

```
5498    https://www.deviantart.com/halconrojo2006/art/Sherlock-Holmes-Museum-
London-101940689
5587    https://www.deviantart.com/hideyoshi/art/Sherlock-Holmes-99996840
Name: image_links, dtype: object
```



6.3.1 Time-series Analysis

Number of posts

From the time-series plots, we see that the posts are distributed from the years 2003 to 2018. The highest number of posts was in 2012, and this is reflected in the month-year chart as well, where we see that 2012-01 has the highest number of posts; the timeframe from 2012 - 2014 in general appears to contribute the most amont of posts. This aligns with the Google Trends graph which showed that interest in the 'Sherlock Holmes' term peaked worldwide in 2012 and 2014, which is pretty neat. In my graph, we also start to see that the number of posts start to increase around 2010, which was when the Sherlock Holmes movie and tv series were both high in popularity. This suggests that there are a good amount of fans who produce fanart when they gain interest in a show.

Month-wise, it appears that the highest number of posts is in January. This could simply be due to the show's popularity around January 2012, which skewed the rest of the January data.

Average number of favorites - finding the outlier

The average number of favorites across time appear to be relatively similar, though there is one huge outlier on the date of 2008-10. A closer examination of this particular date reveals that there were only 2 images posted on this date, but as one of them accrued a high amount of favorites of 3462, the average number of favorites per post is skewed on that date.

7 Model-Building

7.1 Predicting number of favorites based on social metrics

As explained earlier, various linear regression models and a neural network model are trained to predict the number of favorites based on social metrics. The dataset is split with a test-set ratio of 0.2, giving us 4510 training samples and 1128 testing samples.

7.1.1 Feature selection

Sklearn's Recursive Feature Elimination (RFE) function is used to select features by recursively considering smaller and smaller sets of features and their effect on the model's performance. The importance of each feature is obtained using the coef_ function. I used the `RFECV()` function as it has inbuilt cross-validation to determine the optimum number of features that should be inlcuded when training the model (vs having to choose the number of features myself using the basic `RFE()` function).

Interestingly, the `RFECV()` function shows that all the features are important for model performance (Rank 1). This is especially interesting considering that when I did this earlier with a smaller sample size (3000 points), at least one or two features would get dropped. This suggests that as more data is collected, the predictive power of the features, even when small, are still able to contribute towards improving the model's accuracy.

7.1.2 Linear regression models

The linear regression models used were OLS regression, ridge, lasso and elastic net regression. Ridge, lasso, and elastic net regression were cross-validated using the inbuilt sklearn CV functions. Since OLS regression did not come with an inbuilt CV function, I ommited the cross-validation step on the basis that the other regressions should be more accurate due to being less prone to overfitting anyway, and so I was more concerned with their performance.

7.1.3 Neural network

A very simple neural network consisting of 1 input layer, 2 hidden layers, and 1 output layer was built. The first 3 layers had "relu" activation functions to learn any non-linearities within the data. The output layer had a "linear" activation function so that a continuous value for regression purposes was produced. The total trainable parameters was 321, and the model was trained with a mean squared error loss and adam optimizer for 100 epochs, with a batch size of 20 and validation split of 0.2.

In [178]:
```
#linear models for social metric prediction
from sklearn.linear_model import LinearRegression, RidgeCV, LassoCV, ElasticNetCV
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import RFECV #recursive feature elimination with cross-
validation

X_log = log_predictors_df.drop('faves',1)

y_log = log_predictors_df['faves']

X_train, X_test, y_log_train, y_log_test = train_test_split(X_log, y_log, test_size=0.2,
random_state = 10)

print(X_train.shape, X_test.shape, y_log_train.shape, y_log_test.shape)
```

```
(4510, 8) (1128, 8) (4510,) (1128,)
```

In [179]:
```
def selector_model(estimator, X_train, y_train):

    '''
    Inputs:
        X_train (arr): Feature array of data.
        y_train (arr): Response array of data.

    Outputs:
        selector: estimator with feature selection applied

    Attributes:
        ranking_: The feature ranking, such that ranking_[i] corresponds to the ranking position of the i-th feature.
    '''

    selector = RFECV(estimator, step = 1, scoring='neg_mean_squared_error').fit(X_train,
y_train)
    return selector

def align_coefficients(selector):
    '''
    Simple function to adjust coefficient layout in dataframe to correspond with
    selected features from the given selector.
    '''
    coefs = []
    j = 0
    for i in range(len(selector.ranking_)):
        if selector.ranking_[i] != 1:
            coefs.append(np.nan)
        else:
            coefs.append(selector.estimator_.coef_[j])
            j += 1
    return coefs

#linear regressions
linear_reg = selector_model(LinearRegression(), X_train, y_log_train)
ridge_reg = selector_model(RidgeCV(), X_train, y_log_train)
lasso_reg = selector_model(LassoCV(), X_train, y_log_train)

feature_df = pd.DataFrame({"Features":X_log.columns,
                           "Linear Regression Feature Ranking": linear_reg.ranking_,
                           "Linear Regression Coefficients":
align_coefficients(linear_reg),
                           "Ridge Regression Feature Ranking": ridge_reg.ranking_,
                           "Ridge Regression Coefficients":
align_coefficients(ridge_reg),
                           "Lasso Regression Feature Ranking": lasso_reg.ranking_,
                           "Lasso Regression Coefficients":
align_coefficients(lasso_reg)})

feature_df
```

	Features	Linear Regression Feature Ranking	Linear Regression Coefficients	Ridge Regression Feature Ranking	Ridge Regression Coefficients	Lasso Regression Feature Ranking	Lasso Regression Coefficients
0	artist_comments	1	-0.010462	1	-0.010513	1	-0.009281
1	artist_critiques	1	-0.045695	1	-0.045067	1	-0.031428
2	artist_deviations	1	-0.372919	1	-0.371093	1	-0.369114
3	artist_faves	1	-0.018536	1	-0.018648	1	-0.018670
4	artist_forum_posts	1	-0.016153	1	-0.016106	1	-0.015937
5	artist_page_views	1	0.249748	1	0.247587	1	0.243305
6	artist_scraps	1	0.033023	1	0.032958	1	0.031467
7	artist_watchers	1	0.220165	1	0.220805	1	0.221667

In [180]:
```
from keras.models import Sequential
from keras.layers import Input, Flatten, Dense

def create_mlp(dim):
    # define our MLP network
    model = Sequential()
    model.add(Dense(16, input_dim=dim, activation="relu"))
    model.add(Dense(8, activation="relu"))
    model.add(Dense(4, activation="relu"))
    model.add(Dense(1, activation="linear"))

    return model

nn_model = create_mlp(X_train.shape[1])
nn_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_14 (Dense)             (None, 16)                144
_________________________________________________________________
dense_15 (Dense)             (None, 8)                 136
_________________________________________________________________
dense_16 (Dense)             (None, 4)                 36
_________________________________________________________________
dense_17 (Dense)             (None, 1)                 5
=================================================================
Total params: 321
Trainable params: 321
Non-trainable params: 0
_________________________________________________________________
```

In [181]:
```
nn_model.compile(loss="mse", optimizer='adam', metrics=['mae','mse','mape'])
nn_model.fit(X_train, y_log_train, epochs=100, batch_size=20, validation_split = 0.2,
verbose=1)
```

```
Train on 3608 samples, validate on 902 samples
Epoch 1/100
3608/3608 [==============================] - 1s 218us/step - loss: 7.6398 -
mean_absolute_error: 2.2461 - mean_squared_error: 7.6398 -
mean_absolute_percentage_error: 52.4107 - val_loss: 1.1765 - val_mean_absolute_error:
0.8438 - val_mean_squared_error: 1.1765 - val_mean_absolute_percentage_error: 20.1149
Epoch 2/100
3608/3608 [==============================] - 0s 75us/step - loss: 1.0149 -
mean_absolute_error: 0.7847 - mean_squared_error: 1.0149 -
mean_absolute_percentage_error: 18.9972 - val_loss: 0.7903 - val_mean_absolute_error:
0.6935 - val_mean_squared_error: 0.7903 - val_mean_absolute_percentage_error: 16.9284
```

```
Epoch 3/100
3608/3608 [==============================] - 0s 71us/step - loss: 0.8754 -
mean_absolute_error: 0.7358 - mean_squared_error: 0.8754 -
mean_absolute_percentage_error: 17.9443 - val_loss: 0.7399 - val_mean_absolute_error:
0.6590 - val_mean_squared_error: 0.7399 - val_mean_absolute_percentage_error: 15.5673
Epoch 4/100
3608/3608 [==============================] - 0s 76us/step - loss: 0.8609 -
mean_absolute_error: 0.7259 - mean_squared_error: 0.8609 -
mean_absolute_percentage_error: 17.7648 - val_loss: 0.7258 - val_mean_absolute_error:
0.6495 - val_mean_squared_error: 0.7258 - val_mean_absolute_percentage_error: 15.2035
Epoch 5/100
3608/3608 [==============================] - 0s 76us/step - loss: 0.8373 -
mean_absolute_error: 0.7146 - mean_squared_error: 0.8373 -
mean_absolute_percentage_error: 17.4934 - val_loss: 0.6828 - val_mean_absolute_error:
0.6449 - val_mean_squared_error: 0.6828 - val_mean_absolute_percentage_error: 15.8655
Epoch 6/100
3608/3608 [==============================] - 0s 78us/step - loss: 0.8184 -
mean_absolute_error: 0.7091 - mean_squared_error: 0.8184 -
mean_absolute_percentage_error: 17.3911 - val_loss: 0.6933 - val_mean_absolute_error:
0.6593 - val_mean_squared_error: 0.6933 - val_mean_absolute_percentage_error: 16.5276
Epoch 7/100
3608/3608 [==============================] - 0s 69us/step - loss: 0.8242 -
mean_absolute_error: 0.7117 - mean_squared_error: 0.8242 -
mean_absolute_percentage_error: 17.4397 - val_loss: 0.6864 - val_mean_absolute_error:
0.6567 - val_mean_squared_error: 0.6864 - val_mean_absolute_percentage_error: 16.5001
Epoch 8/100
3608/3608 [==============================] - 0s 65us/step - loss: 0.8222 -
mean_absolute_error: 0.7113 - mean_squared_error: 0.8222 -
mean_absolute_percentage_error: 17.4590 - val_loss: 0.6669 - val_mean_absolute_error:
0.6404 - val_mean_squared_error: 0.6669 - val_mean_absolute_percentage_error: 15.8930
Epoch 9/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.8096 -
mean_absolute_error: 0.7049 - mean_squared_error: 0.8096 -
mean_absolute_percentage_error: 17.3490 - val_loss: 0.6611 - val_mean_absolute_error:
0.6264 - val_mean_squared_error: 0.6611 - val_mean_absolute_percentage_error: 15.0521
Epoch 10/100
3608/3608 [==============================] - 0s 70us/step - loss: 0.8108 -
mean_absolute_error: 0.7042 - mean_squared_error: 0.8108 -
mean_absolute_percentage_error: 17.3184 - val_loss: 0.7347 - val_mean_absolute_error:
0.6529 - val_mean_squared_error: 0.7347 - val_mean_absolute_percentage_error: 14.9523
Epoch 11/100
3608/3608 [==============================] - 0s 84us/step - loss: 0.8051 -
mean_absolute_error: 0.7018 - mean_squared_error: 0.8051 -
mean_absolute_percentage_error: 17.2520 - val_loss: 0.7396 - val_mean_absolute_error:
0.6553 - val_mean_squared_error: 0.7396 - val_mean_absolute_percentage_error: 14.9820
Epoch 12/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.8139 -
mean_absolute_error: 0.7057 - mean_squared_error: 0.8139 -
mean_absolute_percentage_error: 17.3485 - val_loss: 0.6608 - val_mean_absolute_error:
0.6380 - val_mean_squared_error: 0.6608 - val_mean_absolute_percentage_error: 15.8114
Epoch 13/100
3608/3608 [==============================] - 0s 74us/step - loss: 0.7948 -
mean_absolute_error: 0.6971 - mean_squared_error: 0.7948 -
mean_absolute_percentage_error: 17.1241 - val_loss: 0.6534 - val_mean_absolute_error:
0.6285 - val_mean_squared_error: 0.6534 - val_mean_absolute_percentage_error: 15.3458
Epoch 14/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.8038 -
mean_absolute_error: 0.7030 - mean_squared_error: 0.8038 -
mean_absolute_percentage_error: 17.2636 - val_loss: 0.6590 - val_mean_absolute_error:
```

```
0.6363 - val_mean_squared_error: 0.6590 - val_mean_absolute_percentage_error: 15.7888
Epoch 15/100
3608/3608 [==============================] - 0s 86us/step - loss: 0.8076 -
mean_absolute_error: 0.7029 - mean_squared_error: 0.8076 -
mean_absolute_percentage_error: 17.2692 - val_loss: 0.6752 - val_mean_absolute_error:
0.6301 - val_mean_squared_error: 0.6752 - val_mean_absolute_percentage_error: 14.8686
Epoch 16/100
3608/3608 [==============================] - 0s 77us/step - loss: 0.7935 -
mean_absolute_error: 0.6968 - mean_squared_error: 0.7935 -
mean_absolute_percentage_error: 17.1416 - val_loss: 0.6814 - val_mean_absolute_error:
0.6317 - val_mean_squared_error: 0.6814 - val_mean_absolute_percentage_error: 14.8395
Epoch 17/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.8006 -
mean_absolute_error: 0.7002 - mean_squared_error: 0.8006 -
mean_absolute_percentage_error: 17.1774 - val_loss: 0.6596 - val_mean_absolute_error:
0.6380 - val_mean_squared_error: 0.6596 - val_mean_absolute_percentage_error: 15.8298
Epoch 18/100
3608/3608 [==============================] - 0s 75us/step - loss: 0.8039 -
mean_absolute_error: 0.7021 - mean_squared_error: 0.8039 -
mean_absolute_percentage_error: 17.2947 - val_loss: 0.6581 - val_mean_absolute_error:
0.6244 - val_mean_squared_error: 0.6581 - val_mean_absolute_percentage_error: 14.9656
Epoch 19/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.7999 -
mean_absolute_error: 0.7017 - mean_squared_error: 0.7999 -
mean_absolute_percentage_error: 17.2280 - val_loss: 0.6524 - val_mean_absolute_error:
0.6294 - val_mean_squared_error: 0.6524 - val_mean_absolute_percentage_error: 15.4832
Epoch 20/100
3608/3608 [==============================] - 0s 77us/step - loss: 0.7924 -
mean_absolute_error: 0.6998 - mean_squared_error: 0.7924 -
mean_absolute_percentage_error: 17.1904 - val_loss: 0.6522 - val_mean_absolute_error:
0.6300 - val_mean_squared_error: 0.6522 - val_mean_absolute_percentage_error: 15.5306
Epoch 21/100
3608/3608 [==============================] - 0s 89us/step - loss: 0.7939 -
mean_absolute_error: 0.6977 - mean_squared_error: 0.7939 -
mean_absolute_percentage_error: 17.1475 - val_loss: 0.6499 - val_mean_absolute_error:
0.6254 - val_mean_squared_error: 0.6499 - val_mean_absolute_percentage_error: 15.2426
Epoch 22/100
3608/3608 [==============================] - 0s 87us/step - loss: 0.7978 -
mean_absolute_error: 0.6966 - mean_squared_error: 0.7978 -
mean_absolute_percentage_error: 17.1316 - val_loss: 0.6505 - val_mean_absolute_error:
0.6262 - val_mean_squared_error: 0.6505 - val_mean_absolute_percentage_error: 15.3209
Epoch 23/100
3608/3608 [==============================] - 0s 76us/step - loss: 0.7887 -
mean_absolute_error: 0.6939 - mean_squared_error: 0.7887 -
mean_absolute_percentage_error: 17.0469 - val_loss: 0.6543 - val_mean_absolute_error:
0.6355 - val_mean_squared_error: 0.6543 - val_mean_absolute_percentage_error: 15.7981
Epoch 24/100
3608/3608 [==============================] - 0s 77us/step - loss: 0.7961 -
mean_absolute_error: 0.6989 - mean_squared_error: 0.7961 -
mean_absolute_percentage_error: 17.1992 - val_loss: 0.6656 - val_mean_absolute_error:
0.6258 - val_mean_squared_error: 0.6656 - val_mean_absolute_percentage_error: 14.8280
Epoch 25/100
3608/3608 [==============================] - 0s 74us/step - loss: 0.7868 -
mean_absolute_error: 0.6939 - mean_squared_error: 0.7868 -
mean_absolute_percentage_error: 17.0604 - val_loss: 0.6520 - val_mean_absolute_error:
0.6301 - val_mean_squared_error: 0.6520 - val_mean_absolute_percentage_error: 15.5300
Epoch 26/100
3608/3608 [==============================] - 0s 71us/step - loss: 0.7828 -
mean_absolute_error: 0.6931 - mean_squared_error: 0.7828 -
```

```
mean_absolute_percentage_error: 17.0444 - val_loss: 0.6532 - val_mean_absolute_error:
0.6344 - val_mean_squared_error: 0.6532 - val_mean_absolute_percentage_error: 15.7533
Epoch 27/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.7887 -
mean_absolute_error: 0.6944 - mean_squared_error: 0.7887 -
mean_absolute_percentage_error: 17.1120 - val_loss: 0.6496 - val_mean_absolute_error:
0.6275 - val_mean_squared_error: 0.6496 - val_mean_absolute_percentage_error: 15.4346
Epoch 28/100
3608/3608 [==============================] - 0s 71us/step - loss: 0.7848 -
mean_absolute_error: 0.6941 - mean_squared_error: 0.7848 -
mean_absolute_percentage_error: 17.0886 - val_loss: 0.6535 - val_mean_absolute_error:
0.6225 - val_mean_squared_error: 0.6535 - val_mean_absolute_percentage_error: 14.9719
Epoch 29/100
3608/3608 [==============================] - 0s 70us/step - loss: 0.7929 -
mean_absolute_error: 0.6963 - mean_squared_error: 0.7929 -
mean_absolute_percentage_error: 17.1241 - val_loss: 0.6536 - val_mean_absolute_error:
0.6237 - val_mean_squared_error: 0.6536 - val_mean_absolute_percentage_error: 15.0720
Epoch 30/100
3608/3608 [==============================] - 0s 77us/step - loss: 0.7850 -
mean_absolute_error: 0.6942 - mean_squared_error: 0.7850 -
mean_absolute_percentage_error: 17.0982 - val_loss: 0.6858 - val_mean_absolute_error:
0.6322 - val_mean_squared_error: 0.6858 - val_mean_absolute_percentage_error: 14.7814
Epoch 31/100
3608/3608 [==============================] - 0s 72us/step - loss: 0.7928 -
mean_absolute_error: 0.6965 - mean_squared_error: 0.7928 -
mean_absolute_percentage_error: 17.0794 - val_loss: 0.6563 - val_mean_absolute_error:
0.6242 - val_mean_squared_error: 0.6563 - val_mean_absolute_percentage_error: 14.9820
Epoch 32/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.7855 -
mean_absolute_error: 0.6957 - mean_squared_error: 0.7855 -
mean_absolute_percentage_error: 17.1048 - val_loss: 0.6588 - val_mean_absolute_error:
0.6238 - val_mean_squared_error: 0.6588 - val_mean_absolute_percentage_error: 14.9114
Epoch 33/100
3608/3608 [==============================] - 0s 72us/step - loss: 0.7864 -
mean_absolute_error: 0.6932 - mean_squared_error: 0.7864 -
mean_absolute_percentage_error: 17.0517 - val_loss: 0.6581 - val_mean_absolute_error:
0.6243 - val_mean_squared_error: 0.6581 - val_mean_absolute_percentage_error: 14.9701
Epoch 34/100
3608/3608 [==============================] - 0s 67us/step - loss: 0.7790 -
mean_absolute_error: 0.6894 - mean_squared_error: 0.7790 -
mean_absolute_percentage_error: 16.9497 - val_loss: 0.6516 - val_mean_absolute_error:
0.6246 - val_mean_squared_error: 0.6516 - val_mean_absolute_percentage_error: 15.1783
Epoch 35/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.7795 -
mean_absolute_error: 0.6888 - mean_squared_error: 0.7795 -
mean_absolute_percentage_error: 16.9439 - val_loss: 0.6974 - val_mean_absolute_error:
0.6368 - val_mean_squared_error: 0.6974 - val_mean_absolute_percentage_error: 14.7828
Epoch 36/100
3608/3608 [==============================] - 0s 82us/step - loss: 0.7918 -
mean_absolute_error: 0.6970 - mean_squared_error: 0.7918 -
mean_absolute_percentage_error: 17.1023 - val_loss: 0.8574 - val_mean_absolute_error:
0.7602 - val_mean_squared_error: 0.8574 - val_mean_absolute_percentage_error: 20.0613
Epoch 37/100
3608/3608 [==============================] - 0s 76us/step - loss: 0.7895 -
mean_absolute_error: 0.6926 - mean_squared_error: 0.7895 -
mean_absolute_percentage_error: 17.0192 - val_loss: 0.6589 - val_mean_absolute_error:
0.6238 - val_mean_squared_error: 0.6589 - val_mean_absolute_percentage_error: 14.9419
Epoch 38/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.7753 -
```

```
mean_absolute_error: 0.6872 - mean_squared_error: 0.7753 -
mean_absolute_percentage_error: 16.9117 - val_loss: 0.6619 - val_mean_absolute_error:
0.6242 - val_mean_squared_error: 0.6619 - val_mean_absolute_percentage_error: 14.8402
Epoch 39/100
3608/3608 [==============================] - 0s 75us/step - loss: 0.7737 -
mean_absolute_error: 0.6871 - mean_squared_error: 0.7737 -
mean_absolute_percentage_error: 16.9143 - val_loss: 0.6625 - val_mean_absolute_error:
0.6252 - val_mean_squared_error: 0.6625 - val_mean_absolute_percentage_error: 14.8421
Epoch 40/100
3608/3608 [==============================] - 0s 72us/step - loss: 0.7837 -
mean_absolute_error: 0.6925 - mean_squared_error: 0.7837 -
mean_absolute_percentage_error: 17.0062 - val_loss: 0.6529 - val_mean_absolute_error:
0.6224 - val_mean_squared_error: 0.6529 - val_mean_absolute_percentage_error: 14.9709
Epoch 41/100
3608/3608 [==============================] - 0s 65us/step - loss: 0.7825 -
mean_absolute_error: 0.6921 - mean_squared_error: 0.7825 -
mean_absolute_percentage_error: 16.9949 - val_loss: 0.7163 - val_mean_absolute_error:
0.6821 - val_mean_squared_error: 0.7163 - val_mean_absolute_percentage_error: 17.5851
Epoch 42/100
3608/3608 [==============================] - 0s 71us/step - loss: 0.7795 -
mean_absolute_error: 0.6896 - mean_squared_error: 0.7795 -
mean_absolute_percentage_error: 16.9810 - val_loss: 0.6640 - val_mean_absolute_error:
0.6250 - val_mean_squared_error: 0.6640 - val_mean_absolute_percentage_error: 14.8126
Epoch 43/100
3608/3608 [==============================] - 0s 76us/step - loss: 0.7766 -
mean_absolute_error: 0.6895 - mean_squared_error: 0.7766 -
mean_absolute_percentage_error: 16.9793 - val_loss: 0.6533 - val_mean_absolute_error:
0.6336 - val_mean_squared_error: 0.6533 - val_mean_absolute_percentage_error: 15.6088
Epoch 44/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.7840 -
mean_absolute_error: 0.6913 - mean_squared_error: 0.7840 -
mean_absolute_percentage_error: 16.9916 - val_loss: 0.6481 - val_mean_absolute_error:
0.6231 - val_mean_squared_error: 0.6481 - val_mean_absolute_percentage_error: 15.1227
Epoch 45/100
3608/3608 [==============================] - 0s 74us/step - loss: 0.7750 -
mean_absolute_error: 0.6882 - mean_squared_error: 0.7750 -
mean_absolute_percentage_error: 16.9569 - val_loss: 0.6516 - val_mean_absolute_error:
0.6222 - val_mean_squared_error: 0.6516 - val_mean_absolute_percentage_error: 14.9164
Epoch 46/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.7788 -
mean_absolute_error: 0.6884 - mean_squared_error: 0.7788 -
mean_absolute_percentage_error: 16.9091 - val_loss: 0.6503 - val_mean_absolute_error:
0.6298 - val_mean_squared_error: 0.6503 - val_mean_absolute_percentage_error: 15.5498
Epoch 47/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.7821 -
mean_absolute_error: 0.6917 - mean_squared_error: 0.7821 -
mean_absolute_percentage_error: 17.0244 - val_loss: 0.6670 - val_mean_absolute_error:
0.6251 - val_mean_squared_error: 0.6670 - val_mean_absolute_percentage_error: 14.7334
Epoch 48/100
3608/3608 [==============================] - 0s 76us/step - loss: 0.7722 -
mean_absolute_error: 0.6863 - mean_squared_error: 0.7722 -
mean_absolute_percentage_error: 16.8488 - val_loss: 0.6645 - val_mean_absolute_error:
0.6476 - val_mean_squared_error: 0.6645 - val_mean_absolute_percentage_error: 16.3037
Epoch 49/100
3608/3608 [==============================] - 0s 72us/step - loss: 0.7763 -
mean_absolute_error: 0.6871 - mean_squared_error: 0.7763 -
mean_absolute_percentage_error: 16.8933 - val_loss: 0.6521 - val_mean_absolute_error:
0.6364 - val_mean_squared_error: 0.6521 - val_mean_absolute_percentage_error: 15.8662
Epoch 50/100
```

```
3608/3608 [==============================] - 0s 75us/step - loss: 0.7706 -
mean_absolute_error: 0.6852 - mean_squared_error: 0.7706 -
mean_absolute_percentage_error: 16.8519 - val_loss: 0.6577 - val_mean_absolute_error:
0.6412 - val_mean_squared_error: 0.6577 - val_mean_absolute_percentage_error: 16.0950
Epoch 51/100
3608/3608 [==============================] - 0s 72us/step - loss: 0.7762 -
mean_absolute_error: 0.6870 - mean_squared_error: 0.7762 -
mean_absolute_percentage_error: 16.8881 - val_loss: 0.7151 - val_mean_absolute_error:
0.6813 - val_mean_squared_error: 0.7151 - val_mean_absolute_percentage_error: 17.5635
Epoch 52/100
3608/3608 [==============================] - 0s 72us/step - loss: 0.7734 -
mean_absolute_error: 0.6867 - mean_squared_error: 0.7734 -
mean_absolute_percentage_error: 16.9103 - val_loss: 0.6426 - val_mean_absolute_error:
0.6226 - val_mean_squared_error: 0.6426 - val_mean_absolute_percentage_error: 15.1597
Epoch 53/100
3608/3608 [==============================] - 0s 67us/step - loss: 0.7894 -
mean_absolute_error: 0.6932 - mean_squared_error: 0.7894 -
mean_absolute_percentage_error: 17.0412 - val_loss: 0.7597 - val_mean_absolute_error:
0.7104 - val_mean_squared_error: 0.7597 - val_mean_absolute_percentage_error: 18.5070
Epoch 54/100
3608/3608 [==============================] - 0s 81us/step - loss: 0.7757 -
mean_absolute_error: 0.6880 - mean_squared_error: 0.7757 -
mean_absolute_percentage_error: 16.9401 - val_loss: 0.6464 - val_mean_absolute_error:
0.6204 - val_mean_squared_error: 0.6464 - val_mean_absolute_percentage_error: 14.8974
Epoch 55/100
3608/3608 [==============================] - 0s 65us/step - loss: 0.7692 -
mean_absolute_error: 0.6860 - mean_squared_error: 0.7692 -
mean_absolute_percentage_error: 16.9003 - val_loss: 0.6517 - val_mean_absolute_error:
0.6200 - val_mean_squared_error: 0.6517 - val_mean_absolute_percentage_error: 14.7674
Epoch 56/100
3608/3608 [==============================] - 0s 71us/step - loss: 0.7844 -
mean_absolute_error: 0.6940 - mean_squared_error: 0.7844 -
mean_absolute_percentage_error: 17.0533 - val_loss: 0.6580 - val_mean_absolute_error:
0.6444 - val_mean_squared_error: 0.6580 - val_mean_absolute_percentage_error: 16.2114
Epoch 57/100
3608/3608 [==============================] - 0s 75us/step - loss: 0.7725 -
mean_absolute_error: 0.6870 - mean_squared_error: 0.7725 -
mean_absolute_percentage_error: 16.8932 - val_loss: 0.6501 - val_mean_absolute_error:
0.6350 - val_mean_squared_error: 0.6501 - val_mean_absolute_percentage_error: 15.8699
Epoch 58/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.7708 -
mean_absolute_error: 0.6857 - mean_squared_error: 0.7708 -
mean_absolute_percentage_error: 16.8863 - val_loss: 0.6407 - val_mean_absolute_error:
0.6198 - val_mean_squared_error: 0.6407 - val_mean_absolute_percentage_error: 15.0010
Epoch 59/100
3608/3608 [==============================] - 0s 75us/step - loss: 0.7710 -
mean_absolute_error: 0.6880 - mean_squared_error: 0.7710 -
mean_absolute_percentage_error: 16.9569 - val_loss: 0.6475 - val_mean_absolute_error:
0.6195 - val_mean_squared_error: 0.6475 - val_mean_absolute_percentage_error: 14.7975
Epoch 60/100
3608/3608 [==============================] - 0s 79us/step - loss: 0.7732 -
mean_absolute_error: 0.6866 - mean_squared_error: 0.7732 -
mean_absolute_percentage_error: 16.8717 - val_loss: 0.6405 - val_mean_absolute_error:
0.6201 - val_mean_squared_error: 0.6405 - val_mean_absolute_percentage_error: 15.0508
Epoch 61/100
3608/3608 [==============================] - 0s 70us/step - loss: 0.7789 -
mean_absolute_error: 0.6909 - mean_squared_error: 0.7789 -
mean_absolute_percentage_error: 17.0159 - val_loss: 0.6684 - val_mean_absolute_error:
0.6271 - val_mean_squared_error: 0.6684 - val_mean_absolute_percentage_error: 14.7729
```

```
Epoch 62/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.7707 -
mean_absolute_error: 0.6861 - mean_squared_error: 0.7707 -
mean_absolute_percentage_error: 16.8917 - val_loss: 0.6383 - val_mean_absolute_error:
0.6204 - val_mean_squared_error: 0.6383 - val_mean_absolute_percentage_error: 15.1429
Epoch 63/100
3608/3608 [==============================] - 0s 74us/step - loss: 0.7696 -
mean_absolute_error: 0.6856 - mean_squared_error: 0.7696 -
mean_absolute_percentage_error: 16.8568 - val_loss: 0.6389 - val_mean_absolute_error:
0.6253 - val_mean_squared_error: 0.6389 - val_mean_absolute_percentage_error: 15.3605
Epoch 64/100
3608/3608 [==============================] - 0s 72us/step - loss: 0.7683 -
mean_absolute_error: 0.6845 - mean_squared_error: 0.7683 -
mean_absolute_percentage_error: 16.8670 - val_loss: 0.6465 - val_mean_absolute_error:
0.6194 - val_mean_squared_error: 0.6465 - val_mean_absolute_percentage_error: 14.8502
Epoch 65/100
3608/3608 [==============================] - 0s 75us/step - loss: 0.7781 -
mean_absolute_error: 0.6903 - mean_squared_error: 0.7781 -
mean_absolute_percentage_error: 16.9981 - val_loss: 0.6392 - val_mean_absolute_error:
0.6243 - val_mean_squared_error: 0.6392 - val_mean_absolute_percentage_error: 15.2604
Epoch 66/100
3608/3608 [==============================] - 0s 75us/step - loss: 0.7759 -
mean_absolute_error: 0.6882 - mean_squared_error: 0.7759 -
mean_absolute_percentage_error: 16.9158 - val_loss: 0.6410 - val_mean_absolute_error:
0.6187 - val_mean_squared_error: 0.6410 - val_mean_absolute_percentage_error: 14.9516
Epoch 67/100
3608/3608 [==============================] - 0s 80us/step - loss: 0.7688 -
mean_absolute_error: 0.6831 - mean_squared_error: 0.7688 -
mean_absolute_percentage_error: 16.8240 - val_loss: 0.6426 - val_mean_absolute_error:
0.6186 - val_mean_squared_error: 0.6426 - val_mean_absolute_percentage_error: 14.8280
Epoch 68/100
3608/3608 [==============================] - 0s 83us/step - loss: 0.7764 -
mean_absolute_error: 0.6872 - mean_squared_error: 0.7764 -
mean_absolute_percentage_error: 16.9149 - val_loss: 0.6994 - val_mean_absolute_error:
0.6366 - val_mean_squared_error: 0.6994 - val_mean_absolute_percentage_error: 14.6619
Epoch 69/100
3608/3608 [==============================] - 0s 78us/step - loss: 0.7780 -
mean_absolute_error: 0.6927 - mean_squared_error: 0.7780 -
mean_absolute_percentage_error: 17.0179 - val_loss: 0.6537 - val_mean_absolute_error:
0.6436 - val_mean_squared_error: 0.6537 - val_mean_absolute_percentage_error: 16.1619
Epoch 70/100
3608/3608 [==============================] - 0s 77us/step - loss: 0.7688 -
mean_absolute_error: 0.6851 - mean_squared_error: 0.7688 -
mean_absolute_percentage_error: 16.8999 - val_loss: 0.6378 - val_mean_absolute_error:
0.6270 - val_mean_squared_error: 0.6378 - val_mean_absolute_percentage_error: 15.5348
Epoch 71/100
3608/3608 [==============================] - 0s 79us/step - loss: 0.7718 -
mean_absolute_error: 0.6841 - mean_squared_error: 0.7718 -
mean_absolute_percentage_error: 16.8377 - val_loss: 0.6344 - val_mean_absolute_error:
0.6231 - val_mean_squared_error: 0.6344 - val_mean_absolute_percentage_error: 15.3256
Epoch 72/100
3608/3608 [==============================] - 0s 79us/step - loss: 0.7743 -
mean_absolute_error: 0.6862 - mean_squared_error: 0.7743 -
mean_absolute_percentage_error: 16.8985 - val_loss: 0.6613 - val_mean_absolute_error:
0.6230 - val_mean_squared_error: 0.6613 - val_mean_absolute_percentage_error: 14.6635
Epoch 73/100
3608/3608 [==============================] - 0s 74us/step - loss: 0.7636 -
mean_absolute_error: 0.6832 - mean_squared_error: 0.7636 -
mean_absolute_percentage_error: 16.8008 - val_loss: 0.6348 - val_mean_absolute_error:
```

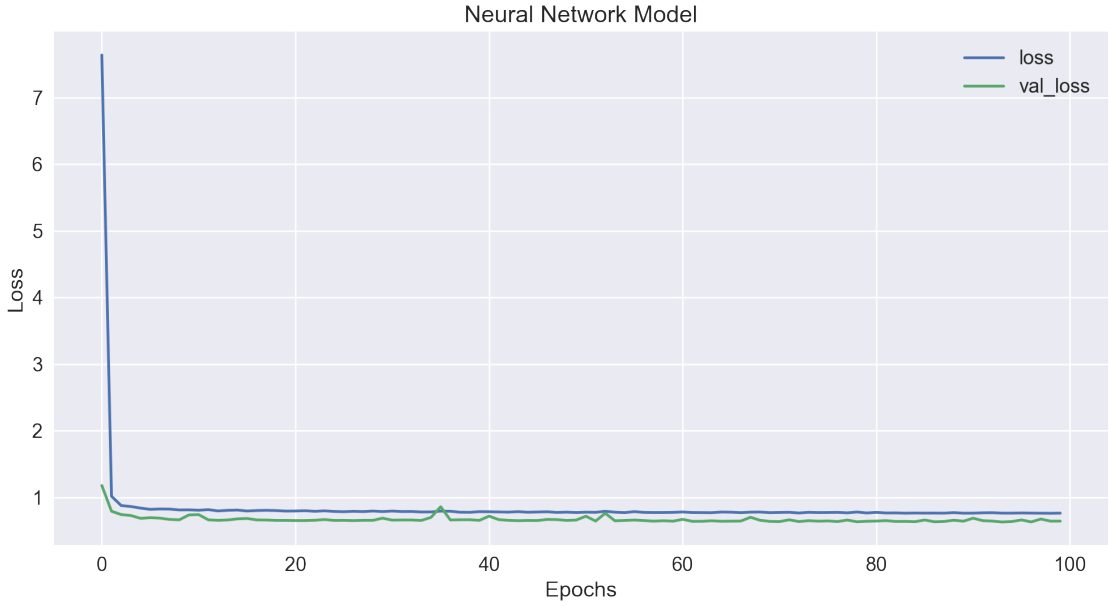
```
0.6262 - val_mean_squared_error: 0.6348 - val_mean_absolute_percentage_error: 15.4772
Epoch 74/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.7739 -
mean_absolute_error: 0.6866 - mean_squared_error: 0.7739 -
mean_absolute_percentage_error: 16.8875 - val_loss: 0.6484 - val_mean_absolute_error:
0.6400 - val_mean_squared_error: 0.6484 - val_mean_absolute_percentage_error: 16.0957
Epoch 75/100
3608/3608 [==============================] - 0s 83us/step - loss: 0.7703 -
mean_absolute_error: 0.6856 - mean_squared_error: 0.7703 -
mean_absolute_percentage_error: 16.9059 - val_loss: 0.6407 - val_mean_absolute_error:
0.6168 - val_mean_squared_error: 0.6407 - val_mean_absolute_percentage_error: 14.7856
Epoch 76/100
3608/3608 [==============================] - 0s 65us/step - loss: 0.7710 -
mean_absolute_error: 0.6856 - mean_squared_error: 0.7710 -
mean_absolute_percentage_error: 16.8540 - val_loss: 0.6443 - val_mean_absolute_error:
0.6169 - val_mean_squared_error: 0.6443 - val_mean_absolute_percentage_error: 14.6883
Epoch 77/100
3608/3608 [==============================] - 0s 66us/step - loss: 0.7731 -
mean_absolute_error: 0.6862 - mean_squared_error: 0.7731 -
mean_absolute_percentage_error: 16.8779 - val_loss: 0.6351 - val_mean_absolute_error:
0.6220 - val_mean_squared_error: 0.6351 - val_mean_absolute_percentage_error: 15.2425
Epoch 78/100
3608/3608 [==============================] - 0s 71us/step - loss: 0.7659 -
mean_absolute_error: 0.6827 - mean_squared_error: 0.7659 -
mean_absolute_percentage_error: 16.7972 - val_loss: 0.6578 - val_mean_absolute_error:
0.6477 - val_mean_squared_error: 0.6578 - val_mean_absolute_percentage_error: 16.4076
Epoch 79/100
3608/3608 [==============================] - 0s 67us/step - loss: 0.7786 -
mean_absolute_error: 0.6899 - mean_squared_error: 0.7786 -
mean_absolute_percentage_error: 16.9437 - val_loss: 0.6318 - val_mean_absolute_error:
0.6172 - val_mean_squared_error: 0.6318 - val_mean_absolute_percentage_error: 15.0224
Epoch 80/100
3608/3608 [==============================] - 0s 71us/step - loss: 0.7640 -
mean_absolute_error: 0.6824 - mean_squared_error: 0.7640 -
mean_absolute_percentage_error: 16.7999 - val_loss: 0.6392 - val_mean_absolute_error:
0.6326 - val_mean_squared_error: 0.6392 - val_mean_absolute_percentage_error: 15.8111
Epoch 81/100
3608/3608 [==============================] - 0s 68us/step - loss: 0.7731 -
mean_absolute_error: 0.6879 - mean_squared_error: 0.7731 -
mean_absolute_percentage_error: 16.9616 - val_loss: 0.6426 - val_mean_absolute_error:
0.6165 - val_mean_squared_error: 0.6426 - val_mean_absolute_percentage_error: 14.7122
Epoch 82/100
3608/3608 [==============================] - 0s 71us/step - loss: 0.7638 -
mean_absolute_error: 0.6836 - mean_squared_error: 0.7638 -
mean_absolute_percentage_error: 16.8187 - val_loss: 0.6490 - val_mean_absolute_error:
0.6387 - val_mean_squared_error: 0.6490 - val_mean_absolute_percentage_error: 16.0842
Epoch 83/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.7649 -
mean_absolute_error: 0.6832 - mean_squared_error: 0.7649 -
mean_absolute_percentage_error: 16.8338 - val_loss: 0.6365 - val_mean_absolute_error:
0.6231 - val_mean_squared_error: 0.6365 - val_mean_absolute_percentage_error: 15.3364
Epoch 84/100
3608/3608 [==============================] - 0s 71us/step - loss: 0.7610 -
mean_absolute_error: 0.6808 - mean_squared_error: 0.7610 -
mean_absolute_percentage_error: 16.7460 - val_loss: 0.6374 - val_mean_absolute_error:
0.6309 - val_mean_squared_error: 0.6374 - val_mean_absolute_percentage_error: 15.7225
Epoch 85/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.7632 -
mean_absolute_error: 0.6820 - mean_squared_error: 0.7632 -
```

```
mean_absolute_percentage_error: 16.8080 - val_loss: 0.6330 - val_mean_absolute_error:
0.6169 - val_mean_squared_error: 0.6330 - val_mean_absolute_percentage_error: 14.9450
Epoch 86/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.7618 -
mean_absolute_error: 0.6808 - mean_squared_error: 0.7618 -
mean_absolute_percentage_error: 16.7490 - val_loss: 0.6585 - val_mean_absolute_error:
0.6491 - val_mean_squared_error: 0.6585 - val_mean_absolute_percentage_error: 16.3947
Epoch 87/100
3608/3608 [==============================] - 0s 72us/step - loss: 0.7623 -
mean_absolute_error: 0.6819 - mean_squared_error: 0.7623 -
mean_absolute_percentage_error: 16.7731 - val_loss: 0.6319 - val_mean_absolute_error:
0.6179 - val_mean_squared_error: 0.6319 - val_mean_absolute_percentage_error: 15.0404
Epoch 88/100
3608/3608 [==============================] - 0s 74us/step - loss: 0.7614 -
mean_absolute_error: 0.6802 - mean_squared_error: 0.7614 -
mean_absolute_percentage_error: 16.7213 - val_loss: 0.6362 - val_mean_absolute_error:
0.6153 - val_mean_squared_error: 0.6362 - val_mean_absolute_percentage_error: 14.8482
Epoch 89/100
3608/3608 [==============================] - 0s 69us/step - loss: 0.7691 -
mean_absolute_error: 0.6827 - mean_squared_error: 0.7691 -
mean_absolute_percentage_error: 16.8109 - val_loss: 0.6551 - val_mean_absolute_error:
0.6195 - val_mean_squared_error: 0.6551 - val_mean_absolute_percentage_error: 14.5960
Epoch 90/100
3608/3608 [==============================] - 0s 72us/step - loss: 0.7608 -
mean_absolute_error: 0.6806 - mean_squared_error: 0.7608 -
mean_absolute_percentage_error: 16.7607 - val_loss: 0.6406 - val_mean_absolute_error:
0.6337 - val_mean_squared_error: 0.6406 - val_mean_absolute_percentage_error: 15.8574
Epoch 91/100
3608/3608 [==============================] - 0s 72us/step - loss: 0.7613 -
mean_absolute_error: 0.6829 - mean_squared_error: 0.7613 -
mean_absolute_percentage_error: 16.8414 - val_loss: 0.6856 - val_mean_absolute_error:
0.6300 - val_mean_squared_error: 0.6856 - val_mean_absolute_percentage_error: 14.5480
Epoch 92/100
3608/3608 [==============================] - 0s 69us/step - loss: 0.7664 -
mean_absolute_error: 0.6845 - mean_squared_error: 0.7664 -
mean_absolute_percentage_error: 16.7957 - val_loss: 0.6483 - val_mean_absolute_error:
0.6414 - val_mean_squared_error: 0.6483 - val_mean_absolute_percentage_error: 16.1801
Epoch 93/100
3608/3608 [==============================] - 0s 87us/step - loss: 0.7678 -
mean_absolute_error: 0.6830 - mean_squared_error: 0.7678 -
mean_absolute_percentage_error: 16.8405 - val_loss: 0.6421 - val_mean_absolute_error:
0.6160 - val_mean_squared_error: 0.6421 - val_mean_absolute_percentage_error: 14.7362
Epoch 94/100
3608/3608 [==============================] - 0s 71us/step - loss: 0.7614 -
mean_absolute_error: 0.6818 - mean_squared_error: 0.7614 -
mean_absolute_percentage_error: 16.7564 - val_loss: 0.6284 - val_mean_absolute_error:
0.6162 - val_mean_squared_error: 0.6284 - val_mean_absolute_percentage_error: 15.0525
Epoch 95/100
3608/3608 [==============================] - 0s 72us/step - loss: 0.7615 -
mean_absolute_error: 0.6803 - mean_squared_error: 0.7615 -
mean_absolute_percentage_error: 16.7422 - val_loss: 0.6354 - val_mean_absolute_error:
0.6176 - val_mean_squared_error: 0.6354 - val_mean_absolute_percentage_error: 14.8683
Epoch 96/100
3608/3608 [==============================] - 0s 77us/step - loss: 0.7641 -
mean_absolute_error: 0.6809 - mean_squared_error: 0.7641 -
mean_absolute_percentage_error: 16.7629 - val_loss: 0.6596 - val_mean_absolute_error:
0.6213 - val_mean_squared_error: 0.6596 - val_mean_absolute_percentage_error: 14.6465
Epoch 97/100
3608/3608 [==============================] - 0s 75us/step - loss: 0.7621 -
```

```
mean_absolute_error: 0.6788 - mean_squared_error: 0.7621 -
mean_absolute_percentage_error: 16.6708 - val_loss: 0.6295 - val_mean_absolute_error:
0.6181 - val_mean_squared_error: 0.6295 - val_mean_absolute_percentage_error: 15.1627
Epoch 98/100
3608/3608 [==============================] - 0s 73us/step - loss: 0.7604 -
mean_absolute_error: 0.6807 - mean_squared_error: 0.7604 -
mean_absolute_percentage_error: 16.7753 - val_loss: 0.6730 - val_mean_absolute_error:
0.6254 - val_mean_squared_error: 0.6730 - val_mean_absolute_percentage_error: 14.5506
Epoch 99/100
3608/3608 [==============================] - 0s 78us/step - loss: 0.7593 -
mean_absolute_error: 0.6787 - mean_squared_error: 0.7593 -
mean_absolute_percentage_error: 16.7135 - val_loss: 0.6411 - val_mean_absolute_error:
0.6189 - val_mean_squared_error: 0.6411 - val_mean_absolute_percentage_error: 14.9029
Epoch 100/100
3608/3608 [==============================] - 0s 80us/step - loss: 0.7615 -
mean_absolute_error: 0.6810 - mean_squared_error: 0.7615 -
mean_absolute_percentage_error: 16.7354 - val_loss: 0.6416 - val_mean_absolute_error:
0.6348 - val_mean_squared_error: 0.6416 - val_mean_absolute_percentage_error: 15.8757
```

Out[181]: <keras.callbacks.History at 0x24bee33710>

In [182]:
```
plt.figure(figsize=(12,6),dpi=200)
plt.plot(nn_model.history.history['loss'], label='loss')
plt.plot(nn_model.history.history['val_loss'], label='val_loss')
plt.title("Neural Network Model")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



In [183]:
```
from sklearn.metrics import r2_score, explained_variance_score, mean_squared_error,
mean_absolute_error, median_absolute_error

def predict_score(model, X_test, y_test):

    predictions = np.array(model.predict(X_test)).reshape(-1, 1)
    y_test = np.array(y_test).reshape(-1, 1)
```

```
        r2 = r2_score(y_test, predictions)
        expl_var = explained_variance_score(y_test, predictions)
        mse = mean_squared_error(y_test, predictions)
        mae = mean_absolute_error(y_test, predictions)
        mede = median_absolute_error(y_test, predictions)
        scores = [r2, expl_var, mse, mae, mede]

        regressor = LinearRegression().fit(y_test, predictions)
        regr_line = regressor.predict(predictions)

        return predictions, scores, regr_line

    lin_pred, lin_score, lin_line = predict_score(linear_reg, X_test, y_log_test)
    ridge_pred, ridge_score, ridge_line = predict_score(ridge_reg, X_test, y_log_test)
    lasso_pred, lasso_score, lasso_line = predict_score(lasso_reg, X_test, y_log_test)
    nn_pred, nn_score, nn_line = predict_score(nn_model, X_test, y_log_test)
```

In [184]:
```
fig, ax = plt.subplots(nrows=2, ncols=2, sharex=True, sharey=True, figsize=(20, 15),
dpi=200)
fig.suptitle('Predicted vs Observed Values', size=30)
fig.text(0.5, 0.04, 'Observed Values', ha='center', size=30)
fig.text(0.04, 0.5, 'Predicted Values', va='center', rotation='vertical', size=30)

plt.subplot(2,2,1)
plt.title('Linear regression predictions')
plt.scatter(y_log_test, lin_pred, label= 'linear regression', alpha = 0.5)
plt.plot(lin_pred, lin_line, label = 'Linear regression fit', color = 'coral')
plt.legend(loc='best')

plt.subplot(2,2,2)
plt.title('Ridge regression predictions')
plt.scatter(y_log_test, ridge_pred, label= 'ridge regression', alpha = 0.5)
plt.plot(ridge_pred, ridge_line, label = 'Ridge regression fit', color = 'coral')
plt.legend(loc='best')

plt.subplot(2,2,3)
plt.title('Lasso regression predictions')
plt.scatter(y_log_test, lasso_pred, label= 'lasso regression', alpha = 0.5)
plt.plot(lasso_pred, lasso_line, label = 'Lasso regression fit', color = 'coral')
plt.legend(loc='best')

plt.subplot(2,2,4)
plt.title('Neural Network predictions')
plt.scatter(y_log_test, nn_pred, label= 'neural network', alpha = 0.5)
plt.plot(nn_pred, nn_line, label = 'neural network fit', color = 'coral')
plt.legend(loc='best')
```

Out[184]: <matplotlib.legend.Legend at 0x24bf2de5c0>


```
In [185]: def create_table_row(index):
              return [lin_score[index], ridge_score[index], lasso_score[index], nn_score[index]]

          scores_df = pd.DataFrame(
                      {"Algorithm": ["Linear Regression","Ridge Regression","Lasso
          Regression","Neural Network"],
                      "r2 score": create_table_row(0),
                      "Explained variance score":create_table_row(1),
                       "Mean Squared Error": create_table_row(2),
                      "Mean Absolute Error": create_table_row(3),
                       "Median Absolute Error": create_table_row(4)})

          scores_df.round(4)
```

	Algorithm	r2 score	Explained variance score	Mean Squared Error	Mean Absolute Error	Median Absolute Error
0	Linear Regression	0.3254	0.3254	0.8119	0.6975	0.5512
1	Ridge Regression	0.3253	0.3253	0.8120	0.6975	0.5520
2	Lasso Regression	0.3252	0.3252	0.8121	0.6975	0.5533
3	Neural Network	0.3180	0.3267	0.8208	0.7094	0.5742

7.1.4 Summary

From the coefficients of the linear regression models, we see that the number of page views and watchers has the largest positive coefficients (0.24 and 0.22 respectively); this suggests that if the artist previously has a huge following and attracts a lot of visitors to their account page, they probably are a popular artist whose art is well-liked (and so get a high number of favorites on their image). The number of deviations has the largest negative coefficient (-0.37); this suggests that artists with a high number of posts tend to have less favorites on their posts - perhaps because it is quicker to produce many pieces of lower-quality work vs high-quality work that receives more favorites but also takes more time to make.

The other predictors have rather weak coefficient values within the range of [-0.05, 0.05].

From the graphs and the scoring dataframe, we see that all the models have roughly the same scores. This means that given a particular image's log-normalized artist attributes, the models will predict the resulting number of favorites with about a 69% - 70% error rate. The explained variance scores of roughly 0.32 suggests that given the log-normalized social metrics, the model is able to account for roughly 32% of the variation within the data. Since the neural network is not outperforming the linear regression models, we can assume that the data has an underlying linear relationship that can be sufficiently modeled by linear models.

Overall, I would say that models perform relatively well. The prediction vs observed values graphs reveal that the predictions at least tend to be within the same neighborhood of the observed values. However, it also appears that the predictions are more accurate when the observed number of likes for a given image are around the average range (3 - 5 number of favorites on the log-normalized scale), while more popular images (7 - 9 on the scale) tend to receive less accurate predictions.

Since the performance of the linear models and the neural network model are quite similar, I would ultimately choose the linear models for prediction of image favorites based based on artist attributes since the linear models have a simpler setup in terms of code and training time, and have more interpretable results since they represent less complex functions.

7.1.5 Unsupervised Learning - K-Means Clustering

As a fun experiment, I decided to use K-Means clustering to see whether there were any discernible clusters that could be formed from the artist attributes.

K-means clustering is an unsupervised learning method commonly used to identify clusters within a dataset. The algorithm works by randomly selecting cluster centers (known as centroids), assigning each data point to their nearest centroid by choosing the one with the smallest Euclidean distance, and then updating the centroid value based on the mean of the data points. The process repeats until a convergence is obtained, i.e. when the centroid values no longer change, thus indicating that "stable" clusters have been found.

Choosing Optimal K using Elbow Plot

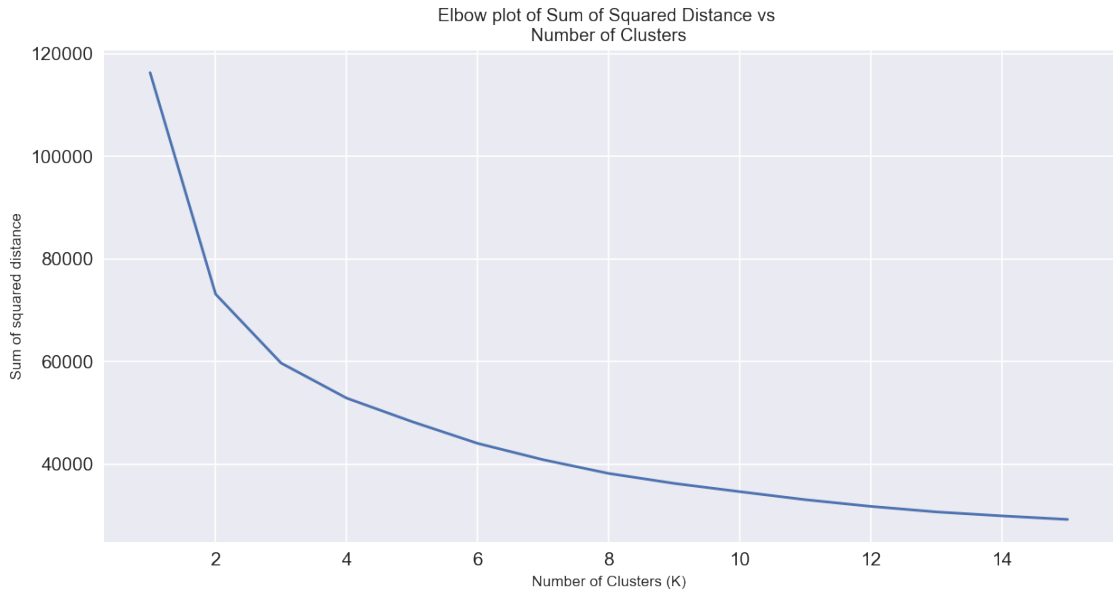
To choose an appropriate number of clusters (K), I tested out the K-means algorithm with K values in the range of 1 to 15 and plotted an elbow plot of sum of squared distances within clusters, i.e. the total variation within each cluster, vs the number of clusters.

The elbow plot is a useful way to determine the optimal K value to choose since it allows us to observe the point at which the within-cluster sum of squares (inertia) starts slowing down in its decrease. While we aim to have clusters with low inertia, when increasing K no longer decreases the inertia significantly, that indicates that the optimal cluster value - the K amount of clusters that contributed to the most inertia decrease - has already been found.

The plot shows a dip in the distances around a K value of 3, and so I clustered the data into 3 clusters.

In [188]:
```
from sklearn.cluster import KMeans

sum_of_squared_distances = []

K = range(1,16)
for k in K:
    km = KMeans(n_clusters=k).fit(X_log)
    sum_of_squared_distances.append(km.inertia_)
```

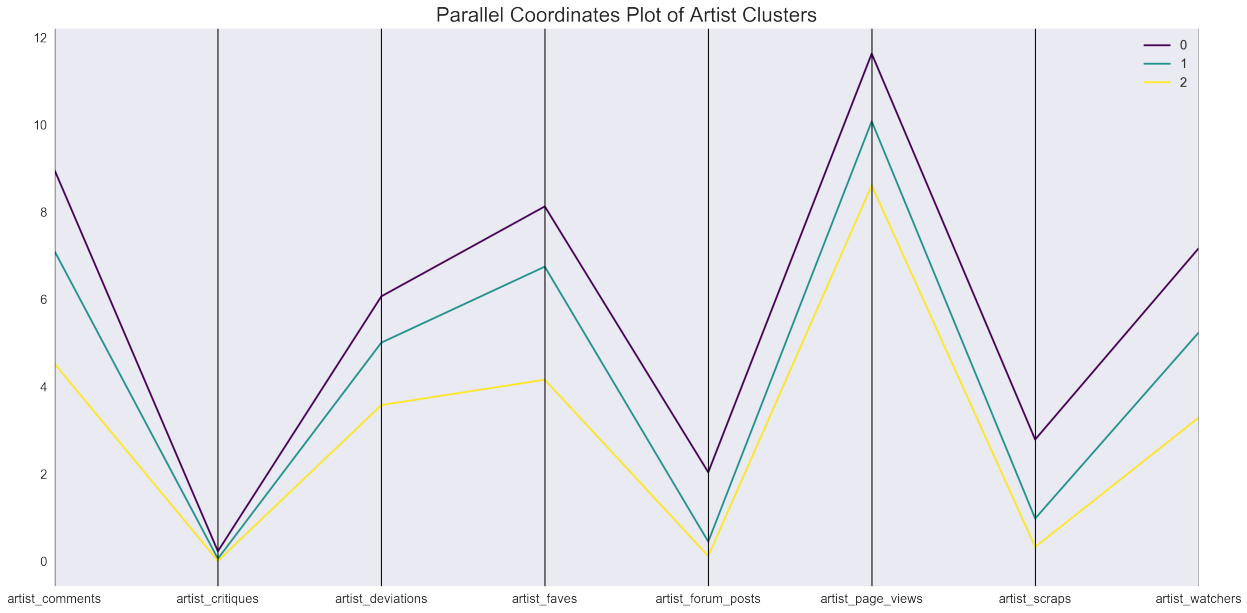
In [189]:
```
plt.figure(figsize=(12,6), dpi=120)
plt.plot(K, sum_of_squared_distances)
plt.title("Elbow plot of Sum of Squared Distance vs \nNumber of Clusters", size=12)
plt.xlabel("Number of Clusters (K)", size=10)
plt.ylabel("Sum of squared distance", size=10)
plt.show()

kmeans = KMeans(n_clusters=3).fit(X_log)
cluster_df = pd.DataFrame(kmeans.cluster_centers_).round(2)
cluster_df.columns = predictors[:-1] #remove 'faves'
cluster_df.reset_index(level=0, inplace=True)
cluster_df
```



	index	artist_comments	artist_critiques	artist_deviations	artist_faves	artist_forum_posts	artist_page_views	artist_scraps	artist_watchers
0	0	8.97	0.23	6.07	8.13	2.04	11.63	2.79	7.17
1	1	7.11	0.06	5.01	6.75	0.45	10.08	0.98	5.24
2	2	4.53	0.01	3.58	4.16	0.12	8.61	0.33	3.29

```
In [236]: plt.figure(figsize=(20,10), dpi=200)
          pd.plotting.parallel_coordinates(cluster_df, "index", colormap='viridis')
          plt.title("Parallel Coordinates Plot of Artist Clusters", size=20)
```

```
Out[236]: Text(0.5,1,'Parallel Coordinates Plot of Artist Clusters')
```

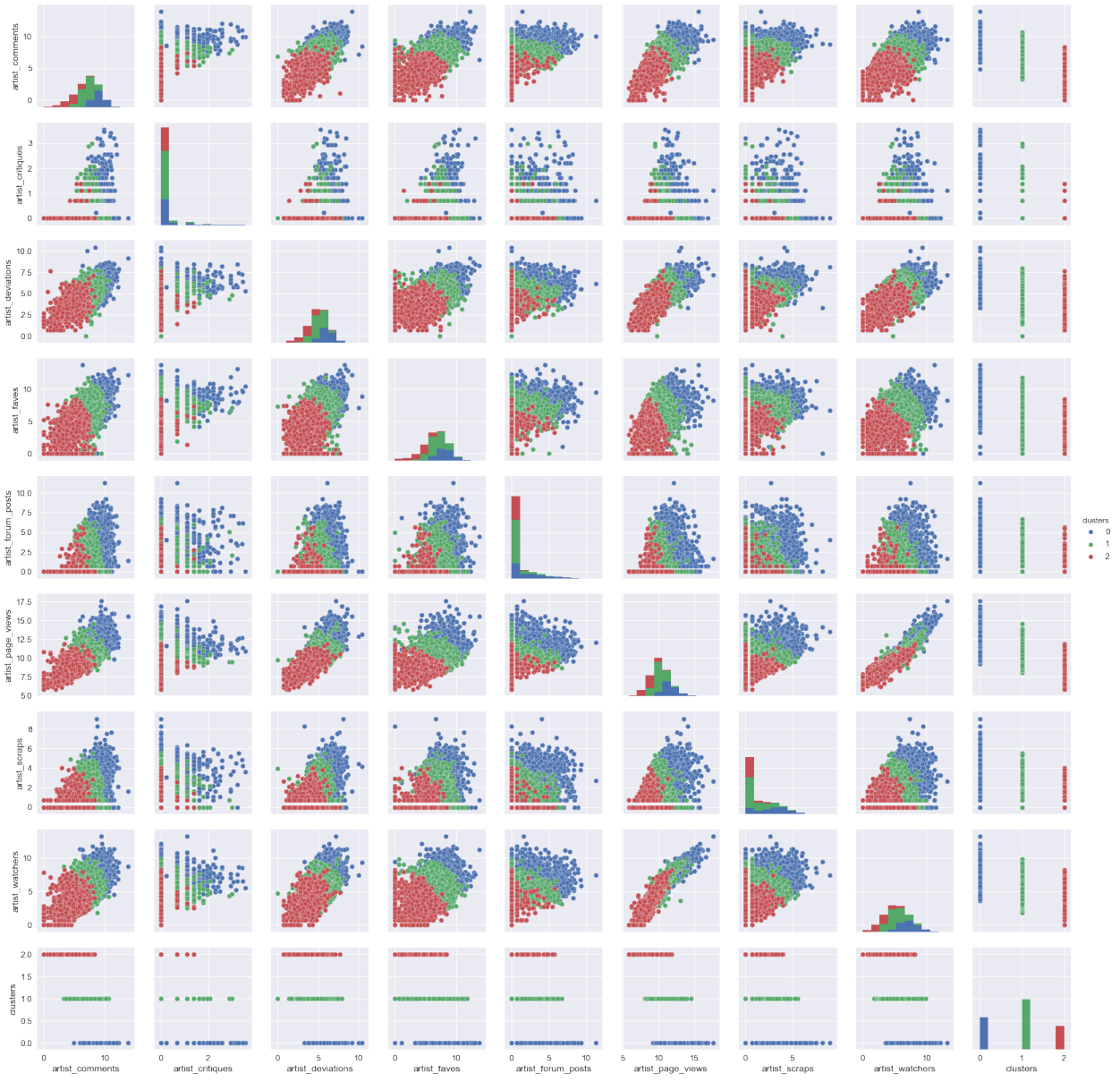


Parallel Coordinates Plot

The parallel coordinates plot shows that the 3 different artist groups identified appear to differ in the number of comments, critiques, deviations, faves, forum posts, page views, scraps, and followers in a rather holistic manner. That is, there are 3 distinct groups separated by the number of attributes that they have in each domain, with one group having the highest count in all attributes (group 0), one group having the lowest count in all attributes (group 2), and one group having a moderate count in all attributes (group 1). This suggests that having a high value in each domain is complementary to accumulating popularity via increased numbers of watchers and faves.

```
In [262]: # assign clusters to each data point using the kmeans model and view pairplot
          cluster_preds = pd.Series(kmeans.predict(X_log))
          X_log['clusters'] = cluster_preds

          sns.pairplot(X_log, hue='clusters')
```

```
Out[262]: <seaborn.axisgrid.PairGrid at 0x24c879db00>
```



Pairplot of Cluster Distributions across the Columns

The pairplot confirms the analysis obtained from the parallel coordinates plot, i.e. there are 3 distinct artist groups that have low, medium, and high values across all domains. This makes sense when we consider the correlations between the artist attributes as shown previously in the heatmap.

7.2 Predicting number of favorites based on image features

As explained earlier, an adapted version of the VGG16 convolutional neural network is used to predict the number of image favorites based on image features. The VGG16 network is preset with 'ImageNet' weights, and the weights are frozen so that they do not have to be retrained, thus saving computational power, though at the cost of less accuracy. The original fully-connected layers are replaced with custom fully-connected layers, with an output layer consisting of a linear activation function with one output so that a regression prediction can be implemented.

After freezing the VGG16 layers, the total trainable weights in the model is around 2.5 million. The model is compiled with mean squared error as the loss function and an adam optimizer.

The images are resized to a dimension of 250 x 250 x 3 for the adapted VGG16 model input, and the data is split with a test-set ratio of 0.3, thus giving us 3946 training data points and 1692 testing data points.

7.2.1 Training the model

Due to time constraints, the model is trained for only 50 epochs with a batch size of 30 and a validation split of 0.2. Due to the small size of the training data and the data's inherent inconsistency (the variation between one artwork and the next is significant), it is expected that the model will probably overfit.

```
In [37]: from skimage.transform import resize

         #create empty image lists
         image_resized_lst = []
         image_original_lst = []

         #height and width of resized image
         new_height = 250
         new_width = 250

         #image loading has to be done one-by-one to ensure that the images are loaded in the
         corresponding order to the
         #data collected in the dataframe
         for i in range(len(final_df)):
             my_dir = 'DA-images-2/'
             img_path = final_df['image_paths'][i]
             #load image
             img = mpimg.imread(my_dir + img_path)
             image_original_lst.append(img)
             #resize image
             img_resized = resize(img,(new_height,new_width),
                                  mode='constant', anti_aliasing=True, anti_aliasing_sigma=None)
             image_resized_lst.append(img_resized)

         original_arr = np.array(image_original_lst)
         resized_arr = np.array(image_resized_lst)

In [38]: print(f"Original images array shape: {original_arr.shape}\nResized images array shape:
         {resized_arr.shape}")

Original images array shape: (5638,)
Resized images array shape: (5638, 250, 250, 3)


In [39]: #peak at the resized images vs original images
         fig, ax = plt.subplots(2, 6, figsize=(20, 10), dpi=120,
```

```
                            subplot_kw={'xticks':[], 'yticks':[]},
                            gridspec_kw=dict(hspace=0.1, wspace=0.1))
        for i in range(6):
            ax[0, i].imshow(original_arr[i])
            ax[1, i].imshow(resized_arr[i])
            ax[0, i].set_title(f"Faves: {final_df.faves[i]}", size=15)

        ax[0, 0].set_ylabel('Images Original',size=15)
        ax[1, 0].set_ylabel('Images Resized',size=15)
```

Out[39]: Text(0,0.5,'Images Resized')

In [40]:
```
from keras.applications.vgg16 import VGG16
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input

#Get back the convolutional part of a VGG network trained on ImageNet
vgg16 = VGG16(weights='imagenet',
              input_shape=(250, 250, 3),
              include_top = False)
vgg16.summary()

#freeze the lower level layers
vgg16.trainable = False

#Add the fully-connected layers
vgg16_adapted = Sequential()
vgg16_adapted.add(vgg16)
vgg16_adapted.add(Flatten(name='flatten'))
vgg16_adapted.add(Dense(100, activation='relu', name='fc1'))
vgg16_adapted.add(Dense(32, activation='relu', name='fc2'))
vgg16_adapted.add(Dense(1, activation='linear'))

#In the summary, weights and layers from VGG part will be hidden, but they will be fit
during the training
vgg16_adapted.summary()

#compile model
vgg16_adapted.compile(loss='mse', optimizer='adam', metrics=['mse','mae','mape'])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 250, 250, 3)       0
```

```
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 250, 250, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 250, 250, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 125, 125, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 125, 125, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 125, 125, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 62, 62, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 62, 62, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 62, 62, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 62, 62, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 31, 31, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 31, 31, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 31, 31, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 31, 31, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 15, 15, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 15, 15, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 15, 15, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 15, 15, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 7, 7, 512)         14714688
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
fc1 (Dense)                  (None, 100)               2508900
_________________________________________________________________
fc2 (Dense)                  (None, 32)                3232
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 33
=================================================================
Total params: 17,226,853
Trainable params: 2,512,165
Non-trainable params: 14,714,688
_________________________________________________________________
```

```
In [41]: cnn_X = resized_arr
         y = np.array(log_predictors_df['faves'])
         cnn_X_train, cnn_X_test, y_train, y_test = train_test_split(cnn_X, y, test_size=0.3)

         print(cnn_X_train.shape,cnn_X_test.shape,y_train.shape,y_test.shape)
```

```
(3946, 250, 250, 3) (1692, 250, 250, 3) (3946,) (1692,)
```

```
In [42]: #training the adapted vgg16 model
         vgg16_adapted.fit(cnn_X_train, y_train,
                           batch_size=30,
                           epochs=50,
                           validation_split=0.2,
                           verbose=1)
```

```
Train on 3156 samples, validate on 790 samples
Epoch 1/50
3156/3156 [==============================] - 1531s 485ms/step - loss: 2.8535 -
mean_squared_error: 2.8535 - mean_absolute_error: 1.1057 -
mean_absolute_percentage_error: 27.0137 - val_loss: 1.0406 - val_mean_squared_error:
1.0406 - val_mean_absolute_error: 0.8087 - val_mean_absolute_percentage_error: 19.9656
Epoch 2/50
3156/3156 [==============================] - 1440s 456ms/step - loss: 0.8879 -
mean_squared_error: 0.8879 - mean_absolute_error: 0.7348 -
mean_absolute_percentage_error: 17.9010 - val_loss: 1.0832 - val_mean_squared_error:
1.0832 - val_mean_absolute_error: 0.7914 - val_mean_absolute_percentage_error: 18.4118
Epoch 3/50
3156/3156 [==============================] - 1450s 460ms/step - loss: 0.6372 -
mean_squared_error: 0.6372 - mean_absolute_error: 0.6173 -
mean_absolute_percentage_error: 15.0267 - val_loss: 1.4883 - val_mean_squared_error:
1.4883 - val_mean_absolute_error: 1.0268 - val_mean_absolute_percentage_error: 27.4544
Epoch 4/50
3156/3156 [==============================] - 1488s 471ms/step - loss: 0.5004 -
mean_squared_error: 0.5004 - mean_absolute_error: 0.5545 -
mean_absolute_percentage_error: 13.5420 - val_loss: 1.1679 - val_mean_squared_error:
1.1679 - val_mean_absolute_error: 0.8418 - val_mean_absolute_percentage_error: 20.1998
Epoch 5/50
3156/3156 [==============================] - 1447s 458ms/step - loss: 0.3206 -
mean_squared_error: 0.3206 - mean_absolute_error: 0.4386 -
mean_absolute_percentage_error: 10.7223 - val_loss: 1.2091 - val_mean_squared_error:
1.2091 - val_mean_absolute_error: 0.8746 - val_mean_absolute_percentage_error: 21.7492
Epoch 6/50
3156/3156 [==============================] - 1417s 449ms/step - loss: 0.2263 -
mean_squared_error: 0.2263 - mean_absolute_error: 0.3671 -
mean_absolute_percentage_error: 9.0110 - val_loss: 1.4216 - val_mean_squared_error:
1.4216 - val_mean_absolute_error: 0.9604 - val_mean_absolute_percentage_error: 24.6864
Epoch 7/50
3156/3156 [==============================] - 1335s 423ms/step - loss: 0.1903 -
mean_squared_error: 0.1903 - mean_absolute_error: 0.3392 -
mean_absolute_percentage_error: 8.3342 - val_loss: 1.3693 - val_mean_squared_error:
1.3693 - val_mean_absolute_error: 0.9038 - val_mean_absolute_percentage_error: 21.1029
Epoch 8/50
3156/3156 [==============================] - 1386s 439ms/step - loss: 0.2013 -
mean_squared_error: 0.2013 - mean_absolute_error: 0.3540 -
mean_absolute_percentage_error: 8.6425 - val_loss: 1.2561 - val_mean_squared_error:
1.2561 - val_mean_absolute_error: 0.8854 - val_mean_absolute_percentage_error: 21.8220
Epoch 9/50
3156/3156 [==============================] - 1519s 481ms/step - loss: 0.1588 -
mean_squared_error: 0.1588 - mean_absolute_error: 0.3126 -
```

```
mean_absolute_percentage_error: 7.6218 - val_loss: 1.2844 - val_mean_squared_error:
1.2844 - val_mean_absolute_error: 0.8988 - val_mean_absolute_percentage_error: 22.3626
Epoch 10/50
3156/3156 [==============================] - 1530s 485ms/step - loss: 0.1194 -
mean_squared_error: 0.1194 - mean_absolute_error: 0.2633 -
mean_absolute_percentage_error: 6.4775 - val_loss: 1.3049 - val_mean_squared_error:
1.3049 - val_mean_absolute_error: 0.8935 - val_mean_absolute_percentage_error: 21.3603
Epoch 11/50
3156/3156 [==============================] - 1627s 515ms/step - loss: 0.0983 -
mean_squared_error: 0.0983 - mean_absolute_error: 0.2358 -
mean_absolute_percentage_error: 5.7299 - val_loss: 1.3811 - val_mean_squared_error:
1.3811 - val_mean_absolute_error: 0.9148 - val_mean_absolute_percentage_error: 21.6038
Epoch 12/50
3156/3156 [==============================] - 1425s 451ms/step - loss: 0.0882 -
mean_squared_error: 0.0882 - mean_absolute_error: 0.2288 -
mean_absolute_percentage_error: 5.5620 - val_loss: 1.6550 - val_mean_squared_error:
1.6550 - val_mean_absolute_error: 1.0449 - val_mean_absolute_percentage_error: 27.2221
Epoch 13/50
3156/3156 [==============================] - 1433s 454ms/step - loss: 0.1038 -
mean_squared_error: 0.1038 - mean_absolute_error: 0.2517 -
mean_absolute_percentage_error: 6.1282 - val_loss: 1.3538 - val_mean_squared_error:
1.3538 - val_mean_absolute_error: 0.9073 - val_mean_absolute_percentage_error: 21.6787
Epoch 14/50
3156/3156 [==============================] - 1485s 470ms/step - loss: 0.0963 -
mean_squared_error: 0.0963 - mean_absolute_error: 0.2375 -
mean_absolute_percentage_error: 5.7705 - val_loss: 1.3314 - val_mean_squared_error:
1.3314 - val_mean_absolute_error: 0.9068 - val_mean_absolute_percentage_error: 21.9359
Epoch 15/50
3156/3156 [==============================] - 1379s 437ms/step - loss: 0.0776 -
mean_squared_error: 0.0776 - mean_absolute_error: 0.2152 -
mean_absolute_percentage_error: 5.1893 - val_loss: 1.3625 - val_mean_squared_error:
1.3625 - val_mean_absolute_error: 0.9218 - val_mean_absolute_percentage_error: 22.7891
Epoch 16/50
3156/3156 [==============================] - 1355s 429ms/step - loss: 0.0677 -
mean_squared_error: 0.0677 - mean_absolute_error: 0.2009 -
mean_absolute_percentage_error: 4.8694 - val_loss: 1.3488 - val_mean_squared_error:
1.3488 - val_mean_absolute_error: 0.9197 - val_mean_absolute_percentage_error: 22.9609
Epoch 17/50
3156/3156 [==============================] - 1347s 427ms/step - loss: 0.0492 -
mean_squared_error: 0.0492 - mean_absolute_error: 0.1661 -
mean_absolute_percentage_error: 4.0210 - val_loss: 1.3188 - val_mean_squared_error:
1.3188 - val_mean_absolute_error: 0.8971 - val_mean_absolute_percentage_error: 21.2710
Epoch 18/50
3156/3156 [==============================] - 1352s 428ms/step - loss: 0.0589 -
mean_squared_error: 0.0589 - mean_absolute_error: 0.1847 -
mean_absolute_percentage_error: 4.4815 - val_loss: 1.5370 - val_mean_squared_error:
1.5370 - val_mean_absolute_error: 0.9943 - val_mean_absolute_percentage_error: 25.6795
Epoch 19/50
3156/3156 [==============================] - 1351s 428ms/step - loss: 0.0751 -
mean_squared_error: 0.0751 - mean_absolute_error: 0.1994 -
mean_absolute_percentage_error: 4.7945 - val_loss: 1.3511 - val_mean_squared_error:
1.3511 - val_mean_absolute_error: 0.9101 - val_mean_absolute_percentage_error: 22.0959
Epoch 20/50
3156/3156 [==============================] - 1347s 427ms/step - loss: 0.0584 -
mean_squared_error: 0.0584 - mean_absolute_error: 0.1849 -
mean_absolute_percentage_error: 4.4394 - val_loss: 1.3051 - val_mean_squared_error:
1.3051 - val_mean_absolute_error: 0.8975 - val_mean_absolute_percentage_error: 22.1280
Epoch 21/50
3156/3156 [==============================] - 1355s 429ms/step - loss: 0.0450 -
```

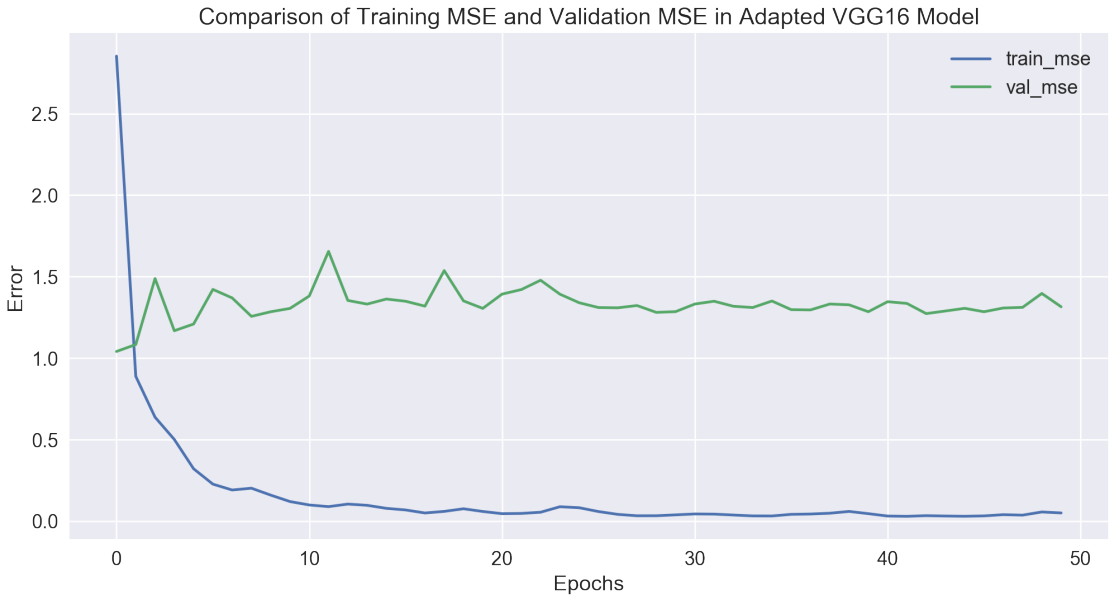
```
mean_squared_error: 0.0450 - mean_absolute_error: 0.1627 -
mean_absolute_percentage_error: 3.9105 - val_loss: 1.3927 - val_mean_squared_error:
1.3927 - val_mean_absolute_error: 0.9323 - val_mean_absolute_percentage_error: 23.6012
Epoch 22/50
3156/3156 [==============================] - 1350s 428ms/step - loss: 0.0464 -
mean_squared_error: 0.0464 - mean_absolute_error: 0.1605 -
mean_absolute_percentage_error: 3.8682 - val_loss: 1.4206 - val_mean_squared_error:
1.4206 - val_mean_absolute_error: 0.9458 - val_mean_absolute_percentage_error: 24.1697
Epoch 23/50
3156/3156 [==============================] - 1350s 428ms/step - loss: 0.0538 -
mean_squared_error: 0.0538 - mean_absolute_error: 0.1781 -
mean_absolute_percentage_error: 4.3033 - val_loss: 1.4779 - val_mean_squared_error:
1.4779 - val_mean_absolute_error: 0.9645 - val_mean_absolute_percentage_error: 24.8907
Epoch 24/50
3156/3156 [==============================] - 1352s 428ms/step - loss: 0.0875 -
mean_squared_error: 0.0875 - mean_absolute_error: 0.2275 -
mean_absolute_percentage_error: 5.4662 - val_loss: 1.3924 - val_mean_squared_error:
1.3924 - val_mean_absolute_error: 0.9276 - val_mean_absolute_percentage_error: 23.3362
Epoch 25/50
3156/3156 [==============================] - 1360s 431ms/step - loss: 0.0815 -
mean_squared_error: 0.0815 - mean_absolute_error: 0.2182 -
mean_absolute_percentage_error: 5.2428 - val_loss: 1.3398 - val_mean_squared_error:
1.3398 - val_mean_absolute_error: 0.9015 - val_mean_absolute_percentage_error: 22.0077
Epoch 26/50
3156/3156 [==============================] - 1348s 427ms/step - loss: 0.0580 -
mean_squared_error: 0.0580 - mean_absolute_error: 0.1824 -
mean_absolute_percentage_error: 4.3500 - val_loss: 1.3103 - val_mean_squared_error:
1.3103 - val_mean_absolute_error: 0.9012 - val_mean_absolute_percentage_error: 22.5496
Epoch 27/50
3156/3156 [==============================] - 1348s 427ms/step - loss: 0.0407 -
mean_squared_error: 0.0407 - mean_absolute_error: 0.1524 -
mean_absolute_percentage_error: 3.6618 - val_loss: 1.3086 - val_mean_squared_error:
1.3086 - val_mean_absolute_error: 0.8980 - val_mean_absolute_percentage_error: 22.1823
Epoch 28/50
3156/3156 [==============================] - 1347s 427ms/step - loss: 0.0321 -
mean_squared_error: 0.0321 - mean_absolute_error: 0.1374 -
mean_absolute_percentage_error: 3.2948 - val_loss: 1.3226 - val_mean_squared_error:
1.3226 - val_mean_absolute_error: 0.8963 - val_mean_absolute_percentage_error: 21.6126
Epoch 29/50
3156/3156 [==============================] - 1350s 428ms/step - loss: 0.0323 -
mean_squared_error: 0.0323 - mean_absolute_error: 0.1366 -
mean_absolute_percentage_error: 3.2644 - val_loss: 1.2806 - val_mean_squared_error:
1.2806 - val_mean_absolute_error: 0.8844 - val_mean_absolute_percentage_error: 21.5928
Epoch 30/50
3156/3156 [==============================] - 1356s 430ms/step - loss: 0.0377 -
mean_squared_error: 0.0377 - mean_absolute_error: 0.1458 -
mean_absolute_percentage_error: 3.4816 - val_loss: 1.2850 - val_mean_squared_error:
1.2850 - val_mean_absolute_error: 0.8874 - val_mean_absolute_percentage_error: 21.5136
Epoch 31/50
3156/3156 [==============================] - 1353s 429ms/step - loss: 0.0432 -
mean_squared_error: 0.0432 - mean_absolute_error: 0.1565 -
mean_absolute_percentage_error: 3.7519 - val_loss: 1.3319 - val_mean_squared_error:
1.3319 - val_mean_absolute_error: 0.8984 - val_mean_absolute_percentage_error: 21.6392
Epoch 32/50
3156/3156 [==============================] - 1342s 425ms/step - loss: 0.0422 -
mean_squared_error: 0.0422 - mean_absolute_error: 0.1567 -
mean_absolute_percentage_error: 3.7381 - val_loss: 1.3490 - val_mean_squared_error:
1.3490 - val_mean_absolute_error: 0.9169 - val_mean_absolute_percentage_error: 23.2494
Epoch 33/50
```

```
3156/3156 [==============================] - 1342s 425ms/step - loss: 0.0367 -
mean_squared_error: 0.0367 - mean_absolute_error: 0.1468 -
mean_absolute_percentage_error: 3.5103 - val_loss: 1.3181 - val_mean_squared_error:
1.3181 - val_mean_absolute_error: 0.8954 - val_mean_absolute_percentage_error: 21.8857
Epoch 34/50
3156/3156 [==============================] - 1346s 427ms/step - loss: 0.0312 -
mean_squared_error: 0.0312 - mean_absolute_error: 0.1337 -
mean_absolute_percentage_error: 3.2084 - val_loss: 1.3103 - val_mean_squared_error:
1.3103 - val_mean_absolute_error: 0.8976 - val_mean_absolute_percentage_error: 22.3398
Epoch 35/50
3156/3156 [==============================] - 1343s 425ms/step - loss: 0.0308 -
mean_squared_error: 0.0308 - mean_absolute_error: 0.1341 -
mean_absolute_percentage_error: 3.2157 - val_loss: 1.3503 - val_mean_squared_error:
1.3503 - val_mean_absolute_error: 0.9126 - val_mean_absolute_percentage_error: 22.7862
Epoch 36/50
3156/3156 [==============================] - 1347s 427ms/step - loss: 0.0409 -
mean_squared_error: 0.0409 - mean_absolute_error: 0.1541 -
mean_absolute_percentage_error: 3.6624 - val_loss: 1.2973 - val_mean_squared_error:
1.2973 - val_mean_absolute_error: 0.8907 - val_mean_absolute_percentage_error: 21.8468
Epoch 37/50
3156/3156 [==============================] - 1354s 429ms/step - loss: 0.0428 -
mean_squared_error: 0.0428 - mean_absolute_error: 0.1596 -
mean_absolute_percentage_error: 3.8293 - val_loss: 1.2957 - val_mean_squared_error:
1.2957 - val_mean_absolute_error: 0.8892 - val_mean_absolute_percentage_error: 21.7912
Epoch 38/50
3156/3156 [==============================] - 1395s 442ms/step - loss: 0.0477 -
mean_squared_error: 0.0477 - mean_absolute_error: 0.1672 -
mean_absolute_percentage_error: 4.0116 - val_loss: 1.3319 - val_mean_squared_error:
1.3319 - val_mean_absolute_error: 0.8999 - val_mean_absolute_percentage_error: 21.6848
Epoch 39/50
3156/3156 [==============================] - 1356s 430ms/step - loss: 0.0587 -
mean_squared_error: 0.0587 - mean_absolute_error: 0.1869 -
mean_absolute_percentage_error: 4.5054 - val_loss: 1.3270 - val_mean_squared_error:
1.3270 - val_mean_absolute_error: 0.8961 - val_mean_absolute_percentage_error: 21.3831
Epoch 40/50
3156/3156 [==============================] - 1421s 450ms/step - loss: 0.0450 -
mean_squared_error: 0.0450 - mean_absolute_error: 0.1597 -
mean_absolute_percentage_error: 3.8111 - val_loss: 1.2846 - val_mean_squared_error:
1.2846 - val_mean_absolute_error: 0.8894 - val_mean_absolute_percentage_error: 21.5802
Epoch 41/50
3156/3156 [==============================] - 1438s 456ms/step - loss: 0.0304 -
mean_squared_error: 0.0304 - mean_absolute_error: 0.1295 -
mean_absolute_percentage_error: 3.0991 - val_loss: 1.3462 - val_mean_squared_error:
1.3462 - val_mean_absolute_error: 0.9131 - val_mean_absolute_percentage_error: 22.8258
Epoch 42/50
3156/3156 [==============================] - 1379s 437ms/step - loss: 0.0288 -
mean_squared_error: 0.0288 - mean_absolute_error: 0.1285 -
mean_absolute_percentage_error: 3.0693 - val_loss: 1.3356 - val_mean_squared_error:
1.3356 - val_mean_absolute_error: 0.9080 - val_mean_absolute_percentage_error: 22.6378
Epoch 43/50
3156/3156 [==============================] - 1549s 491ms/step - loss: 0.0327 -
mean_squared_error: 0.0327 - mean_absolute_error: 0.1381 -
mean_absolute_percentage_error: 3.2934 - val_loss: 1.2731 - val_mean_squared_error:
1.2731 - val_mean_absolute_error: 0.8818 - val_mean_absolute_percentage_error: 21.1893
Epoch 44/50
3156/3156 [==============================] - 1485s 471ms/step - loss: 0.0305 -
mean_squared_error: 0.0305 - mean_absolute_error: 0.1338 -
mean_absolute_percentage_error: 3.1919 - val_loss: 1.2890 - val_mean_squared_error:
1.2890 - val_mean_absolute_error: 0.8852 - val_mean_absolute_percentage_error: 21.1778
```

```
Epoch 45/50
3156/3156 [==============================] - 1451s 460ms/step - loss: 0.0291 -
mean_squared_error: 0.0291 - mean_absolute_error: 0.1301 -
mean_absolute_percentage_error: 3.1152 - val_loss: 1.3055 - val_mean_squared_error:
1.3055 - val_mean_absolute_error: 0.8914 - val_mean_absolute_percentage_error: 21.8665
Epoch 46/50
3156/3156 [==============================] - 1607s 509ms/step - loss: 0.0313 -
mean_squared_error: 0.0313 - mean_absolute_error: 0.1358 -
mean_absolute_percentage_error: 3.2478 - val_loss: 1.2845 - val_mean_squared_error:
1.2845 - val_mean_absolute_error: 0.8828 - val_mean_absolute_percentage_error: 21.1241
Epoch 47/50
3156/3156 [==============================] - 2373s 752ms/step - loss: 0.0389 -
mean_squared_error: 0.0389 - mean_absolute_error: 0.1493 -
mean_absolute_percentage_error: 3.5678 - val_loss: 1.3075 - val_mean_squared_error:
1.3075 - val_mean_absolute_error: 0.9029 - val_mean_absolute_percentage_error: 22.5372
Epoch 48/50
3156/3156 [==============================] - 1583s 502ms/step - loss: 0.0360 -
mean_squared_error: 0.0360 - mean_absolute_error: 0.1415 -
mean_absolute_percentage_error: 3.3586 - val_loss: 1.3113 - val_mean_squared_error:
1.3113 - val_mean_absolute_error: 0.8990 - val_mean_absolute_percentage_error: 22.1704
Epoch 49/50
3156/3156 [==============================] - 1389s 440ms/step - loss: 0.0554 -
mean_squared_error: 0.0554 - mean_absolute_error: 0.1824 -
mean_absolute_percentage_error: 4.3745 - val_loss: 1.3967 - val_mean_squared_error:
1.3967 - val_mean_absolute_error: 0.9286 - val_mean_absolute_percentage_error: 23.2085
Epoch 50/50
3156/3156 [==============================] - 1405s 445ms/step - loss: 0.0494 -
mean_squared_error: 0.0494 - mean_absolute_error: 0.1706 -
mean_absolute_percentage_error: 4.0780 - val_loss: 1.3150 - val_mean_squared_error:
1.3150 - val_mean_absolute_error: 0.8972 - val_mean_absolute_percentage_error: 22.1429
```

```
Out[42]: <keras.callbacks.History at 0x22bed63940>
```

```
In [237]: plt.figure(figsize=(12,6),dpi=200)
          plt.plot(vgg16_adapted.history.history['mean_squared_error'], label='train_mse')
          plt.plot(vgg16_adapted.history.history['val_mean_squared_error'], label='val_mse')
          plt.title("Comparison of Training MSE and Validation MSE in Adapted VGG16 Model")
          plt.xlabel("Epochs")
          plt.ylabel("Error")
          plt.legend()
          plt.show()
```



7.2.2 Analysis of Training Loss vs Validation Loss

The training loss decreases as the epochs increase, but the validation loss flucatues around 1.0 - 1.5. This indicates that the model has overfit to the data, which as previously explained, was expected due to the small sample size and variation between each data point (artwork).

While overfit could also be due to the number of epochs being too high, in which case an early-stopping mechanism could be implemented, this is probably not likely given that the validation error is not dipping and rising again after a certain point, but does not even start to go down at any point. As such, while the CNN is overfitting, an early-stopping criterion will not assist in this case.

7.2.3 Making Predictions on Training and Testing Data

Also the results above show that the model has overfitted to the training data, I decided to evaluate the model's predictions on the training data and unseen testing data anyway to gain more insight as to how poorly it is performing on the testing data.

Training Data Predictions

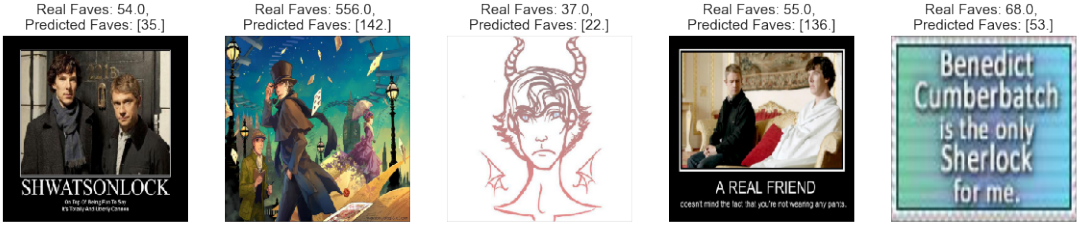
The 5 examples below show that the training predictions are performing well, even guessing the exact number at some points. Unfortunately, this is a bad sign since it just cements the fact that the model has overfitted.

```
In [123]: cnn_train_pred, cnn_train_scores, cnn_train_line = predict_score(vgg16_adapted,
          cnn_X_train, y_train)

In [124]: cnn_train_pred_exp = np.exp(cnn_train_pred)
          y_train_exp = np.exp(y_train)

          plt.figure(figsize=(20,6))
          for i in range(5):
              plt.subplot(1,5,i+1)
              plt.grid(False)
              plt.axis('off')
              plt.title(f"Real Faves: {np.floor(y_train_exp[i])}, \nPredicted Faves:
          {np.floor(cnn_train_pred_exp[i])}")
              plt.imshow(cnn_X_train[i])
```

Testing Data Predictions

While the 5 testing data predictions below are just examples and not representative of the model's performance on the entire dataset, we can still see that the model's predictions are not performing as well as on the training data.

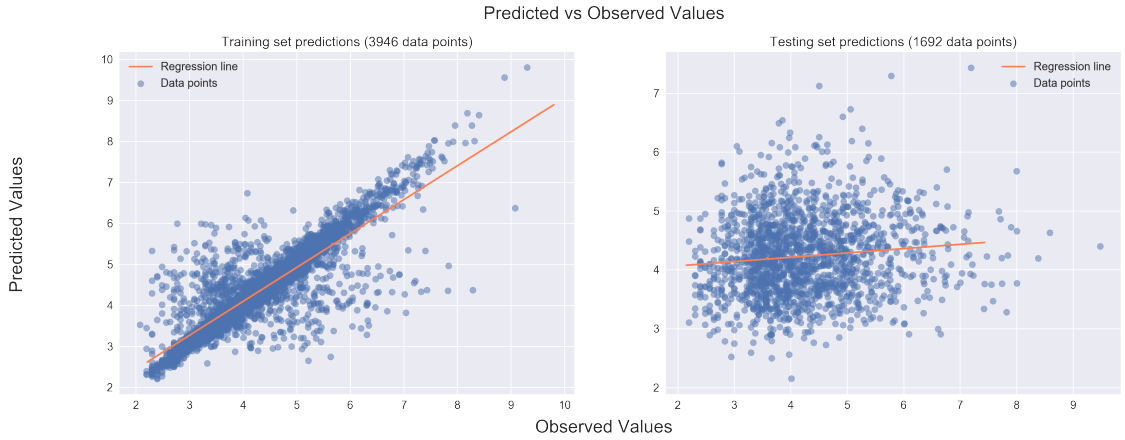
```
In [125]: cnn_test_pred, cnn_test_scores, cnn_test_line = predict_score(vgg16_adapted, cnn_X_test,
          y_test)

In [126]: cnn_test_pred_exp = np.exp(cnn_test_pred)
          y_test_exp = np.exp(y_test)

          plt.figure(figsize=(20,6))
          for i in range(5):
              plt.subplot(1,5,i+1)
              plt.grid(False)
              plt.axis('off')
              plt.title(f"Real Faves: {np.floor(y_test_exp[i])}, \nPredicted Faves:
          {np.floor(cnn_test_pred_exp[i])}")
              plt.imshow(cnn_X_test[i])
```



7.2.4 Analysis of Predicted vs Observed Values

```
In [129]: fig, ax = plt.subplots(nrows=1, ncols=2, sharex=True, sharey=True, figsize=(20, 7),
          dpi=200)
          fig.suptitle('Predicted vs Observed Values', size=20)
          fig.text(0.5, 0.04, 'Observed Values', ha='center', size=20)
          fig.text(0.04, 0.5, 'Predicted Values', va='center', rotation='vertical', size=20)

          plt.subplot(1,2,1)
          plt.title(f'Training set predictions ({len(y_train)} data points)')
          plt.scatter(y_train, cnn_train_pred, label='Data points', alpha = 0.5)
          plt.plot(cnn_train_pred, cnn_train_line, label='Regression line', color = 'coral')
          plt.legend(loc='best')

          plt.subplot(1,2,2)
          plt.title(f'Testing set predictions ({len(y_test)} data points)')
          plt.scatter(y_test, cnn_test_pred, label='Data points', alpha = 0.5)
          plt.plot(cnn_test_pred, cnn_test_line, label='Regression line', color = 'coral')
          plt.legend(loc='best')
```

```
Out[129]: <matplotlib.legend.Legend at 0x1a3473f898>
```




```
In [239]: cnn_results_df = pd.DataFrame({
              'Data':['Training set','Testing set'],
              'r2 score': [cnn_train_scores[0], cnn_test_scores[0]],
              'Explained variance score': [cnn_train_scores[1], cnn_test_scores[1]],
              'Mean Squared Error': [cnn_train_scores[2], cnn_test_scores[2]],
              'Mean Absolute Error': [cnn_train_scores[3], cnn_test_scores[3]],
              'Median Absolute Error': [cnn_train_scores[4], cnn_test_scores[4]],
          })

          cnn_results_df
```

	Data	r2 score	Explained variance score	Mean Squared Error	Mean Absolute Error	Median Absolute Error
0	Training set	0.727882	0.730327	0.292173	0.294817	0.139657
1	Testing set	-0.282316	-0.282142	1.380767	0.919549	0.772398

7.2.5 Summary

The MSE comparison plot shows that the model has most likely overfitted the data since the training MSE has successfully decreased whereas the validation MSE remains consistently around 1.0 - 1.5 with no sign of decrease. This is also reflected in the predicted vs observed values regression plots. The model's training data predictions have a good regression line fit to the observed values, whereas the model's testing data predictions have pretty much no pattern to them. The results dataframe also shows this in the explained variance score and errors. While the model has 0.7 explained variance on the training data, it has -0.28 explained variance for the testing data. Furthermore, while the MAE is relatively low on the training data (0.29), the MAE on the testing data is 0.91.

The overfitting is probably a result of the small sample size. Aside from simply collecting more data, some other suggestions for improvement are listed below.[7]

7.2.6 Potential Improvements for CNN Model

Selective Filtering of Images

While most of the images look fine when resized to 250 x 250, there are some images with very unbalanced height x width dimensions that look severely misrepresented after resizing. While this is probably not a significant contributer to the model's poor performance on unseen data, it would still be good to remove such images beforehand.

Using PCA to Reduce Dimensionality vs Naive Resizing

To make my images have similar input sizes, I simply used `skimage.transform` to resize my images to 250 x 250. By using a more sophisticated method for dimensionality reduction that actually seeks to maximize the explained variance, such as PCA, I could potentially have retained more information in my images after reducing their dimensionality.

Data Augmentation

Data augmentation is a technique for increasing the dataset size and also improving the robustness of the CNN model by making it invariant to different forms of translations and transformations (Raj, 2018). Some data augmentation methods include flipping, rotating, scaling, cropping, etc. Since I am dealing with art popularity and not object classification, the augmentation will have to carried out within reasonable constraints since the same piece of art, cropped, will most definitely not represent the original art accurately. Minor augmentation such as flipping and rotation should be fine and may help the model since we do not expect an artwork's aesthetic appeal to significantly decrease if it was viewed from a different orientation.

[7] **#creativeheuristics:** While an obvious solution to overfitting would be to collect more data, there are various other steps that can be implemented to try to minimize the model's overfitting behavior. By doing research and exploring various different concepts (augmenting the data, reducing dimensionality, using neural codes, including dropout layer, etc.), I demonstrate an integration of multiple different heuristics in order to solve the problem of overfitting. By exploring various methods for combatting overfit, I gained a better understanding of why overfit happens, why it could be happening to my data, and also am now aware of various steps I can try to reduce overfit in my model.

Include Dropout Layer

Dropout is a regularization method that randomly ignores various units in the neural network during the training process; it has been shown to correct overfitting in convolutional neural networks (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014). By randomly dropping nodes during training, the other nodes within the layer are forced to probabilistically take on more or less responsibility for the inputs. This minimizes the ocurrence of complex co-adaptations within the network, which can be hard to generalize to unseen data (Srivastava et al., 2014).

While the original VGG16 model has dropout regularization incorporated (Simonyan & Zisserman, 2014), the Keras VGG16 does not. As such, I could try incorporating dropout layers in the future to reduce overfitting.

Create Image Representation using Lower-Layer Neural Codes

Neural Codes are "the activations invoked by an image within the top layers of a large convolutional neural network [that] provide a high-level descriptor of the visual content of the image" (Babenko, Slesarev, Chigorin, & Lempitsky, 2014). In a study on the efficacy and applicability of neural codes for image classification in different tasks, Babenko et al. (2014) found that neural codes trained on a CNN for a particular task could transfer well to classification performance on a different task.

Of particular note, they found that "the best performance is observed not on the very top of the network, but rather at the layer that is two levels below the outputs ... because the very top layers are too much tuned for the classification task, while the bottom layers do not acquire enough invariance to nuisance factors" (Babenko et al., 2014). Their findings suggest that when implementing transfer-learning, I should consider using lower-layer neural codes instead and train a separate model on those neural codes to avoid the VGG16 model from overfitting to various noise factors within my images (since VGG16 was originally built for a different task).

Babenko et al. (2014) also found that PCA could be used to reduce the dimensionality of their neural codes to as low as 250 or even 180 dimensions without loss in model performance. Thus, in the future, I could consider using neural codes combined with PCA dimensionality reduction to create image representations that hopefully retain the most variance while discarding the most noise to build a model that does not overfit.

Early-stopping (Not Applicable in this Case)

Early-stopping is a method for preventing overfit by stopping the training as soon as the validation error is higher than the last time that it was checked (Prechelt, 1998). The weights of the model at that point in time are then used as the final model weights for testing. While this is a useful method to make note of in general, it does not apply in this specific case since the validation error does not show a pattern of decrease followed by increase.

8 Conclusion

While my results show that the artist attributes model has a better performance than the image features model, I would say that comparing the different models is not quite fair since the adapted VGG16 model has much more parameters vs the amount of data available, which leads to a much higher chance of overfitting - as we see in the results. In the future, I would need to collect more data as well as incorporate various other strategies to combat overfitting, e.g. data augmentation, dropout.

That being said, the preliminary results obtained from the artist attributes models are quite encouraging as they align with our intuitions that the more followers and page views an artist has, the more favorites they will receive on their image. Although the mean absolute error is ~70%, I think given the use case, a highly accurate model is not necessary, and we would instead like to gain an intuition as to the most influential predictors for fanart popularity.

By knowing the artist attributes with the most predictive power, we see that the underlying behavior within fans of fictional works is not too different from that of social media, i.e. more followers == more likes. In future iterations of this work, gathering more data may reveal other social metrics that are good predictors, and allow those in the media and entertainment industry to know which social metrics they should prioritize when looking to collaborate with a particular artist for advertising, to increase the exposure and popularity towards a certain show, movie, etc.[8]

8.1 Potential Improvements and Extensions

While specific improvements for the CNN model were listed above, the improvements and extensions listed here are more general and apply to the project as a whole.

8.1.1 Potential Improvements

- Normalize the number of favorites to account for the effect of account age. According to Khosla et al. (2014), "visual media tends to receive views over some period of time. To normalize for this effect, we divide the number of views by the duration since the upload date of the given image."
- Use a more nuanced "popularity indicator" instead of just using the number of favorites. While using the number of favorites is easy and convenient for analyzing popularity, it is possible that the number of views may be more suggestive of popularity since to favorite an image, one has to be a member of the site, whereas anyone can view the image. As such, a more nuanced popularity metric consisting of some mixture of the two may be ideal; perhaps even comments could be included since they indicate engagement with the image.

[8] **#regression:** This whole project consisted of solving a regression task. In order to predict the popularity of fanart, the number of favorites is used as a response variable, and various models - from linear regression models to adapted CNNs - are used to solve the regression problem by testing out different types of predictors (artist attributes vs image features).

8.1.2 Potential Extensions

- Analyze high-level color features by extracting the images color distribution, hues, intensity, etc. This would provide a more interpretable indication of 'popular' image features vs using a CNN.
- Analyze the different types of comments on the image. It would be interesting to use NLP techniques to analyze the different comment types and try to categorize them into good, bad, or neutral comments - which can then be used to predict image popularity.
- Analyze the image titles. While the image titles were not used as a predictor this time (mainly because I did not think that the length of the image title would be a very good predictor), they could be used in the future.
- Analyze the hashtags. As previously explained, I could not use the hashtags in the predictive process this time since there were so few non-null values, and I also made a mistake while scraping the hashtags, thus corrupting the data. That being said, I think hashtags would be a potential powerful predictor - one could use the number of hashtags as a predictor, and also the type of hashtags if they are processed using NLP methods.
- Analyze image description. Every image posted on Deviantart comes with a description written by the artist. I did not scrape or analyze image descriptions in this project since the descriptions often contained irregular elements such as images, emojis, gifs, etc. which made them difficult to parse into a proper format. That being said, descriptions - much like hashtags - would also be a significant feature to consider in the future in terms of both length and content.
- Analyze age, sex, location. While I did scrape this data, the column format was deemed to challenging to parse this time around. In the future, I would work to parse the column properly to gain insights from the artist's more personal attributes.
- Analyze artist activity. Artist activity, such as how often they post, comment, etc. could also be a strong predictor. This feature would be quite hard to obtain since most artists tend to keep their activity hidden on DeviantArt.

References

[1] Babenko, A., Slesarev, A., Chigorin, A., & Lempitsky, V. (2014, September). Neural codes for image retrieval. In European conference on computer vision (pp. 584-599). Springer, Cham.

[2] De Veirman, M., Cauberghe, V., & Hudders, L. (2017). Marketing through Instagram influencers: the impact of number of followers and product divergence on brand attitude. International Journal of Advertising, 36(5), 798-828.

[3] DeviantArt. (2019). Copyright Policy. Retrieved March 16, 2019, from https://about.deviantart.com/policy/copyright/

[4] Dufour, N. (2014). Will it play in Peoria? Predicting Image Popularity with Convolutional Neural Networks. Retrieved March 25, 2019, from http://cs231n.stanford.edu/reports/2015/pdfs/dufour.pdf

[5] Google Trends (2019) Sherlock Holmes Worldwide Interest Over time. Retrieved April 23, 2019, from https://trends.google.com/trends/explore?date=all&q=%2Fm%2F06rkl

[6] Guinness World Record. (2012, May 14). Sherlock Holmes awarded title for most portrayed literary human character in film & TV. Retrieved April 21, 2019, from http://www.guinnessworldrecords.com/news/2012/5/sherlock-holmes-awarded-title-for-most-portrayed-literary-human-character-in-film-tv-41743/

[7] Hassan, M. U. (2018, November 21). VGG16 - Convolutional Network for Classification and Detection. Retrieved April 21, 2019, from https://neurohive.io/en/popular-networks/vgg16/

[8] Karpathy, A. (2016). Cs231n convolutional neural networks for visual recognition. Neural networks, 1. Retrieved April 25, 2019, from https://cs231n.github.io/convolutional-networks/

[9] Khosla, A., Das Sarma, A., & Hamid, R. (2014, April). What makes an image popular?. In Proceedings of the 23rd international conference on World wide web (pp. 867-876). ACM.

[10] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

[11] Prechelt, L. (1998). Early stopping-but when?. In Neural Networks: Tricks of the trade (pp. 55-69). Springer, Berlin, Heidelberg.

[12] Raj, B. (2018, April 11). Data Augmentation | How to use Deep Learning when you have Limited Data - Part 2. Retrieved April 23, 2019, from https://medium.com/nanonets/how-to-use-deep-learning-when-you-have-limited-data-part-2-data-augmentation-c26971dc8ced

[13] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

[14] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The Journal of Machine Learning Research, 15(1), 1929-1958.

[15] Wikipedia contributors. (2019, April 13). Fan art. In Wikipedia, The Free Encyclopedia. Retrieved 03:11, April 21, 2019, from https://en.wikipedia.org/w/index.php?title=Fan_art&oldid=892258552

[16] Wikipedia contributors. (2019, April 20). Sherlock Holmes. In Wikipedia, The Free Encyclopedia. Retrieved 16:10, April 21, 2019, from https://en.wikipedia.org/w/index.php?title=Sherlock_Holmes&oldid=893342542

A Appendix: Scraper Code

https://nbviewer.jupyter.org/github/hueyning/DA-scraper/blob/master/python-notebook/DA-webscraper.ipynb

```
In [1]: #import basic libraries
        import pandas as pd
        import re
        import logging
        import scrapy
        from scrapy.crawler import CrawlerProcess

In [2]: from scrapy.pipelines.images import ImagesPipeline
        from scrapy.exceptions import DropItem

        class ImageItem(scrapy.Item):

            #direct link to image file for downloading via ImagePipeline
            image_urls = scrapy.Field()

            #link to specific image page for scraping more stats
            image_links = scrapy.Field()

            #image attributes
            titles = scrapy.Field() #image title
            date_posted = scrapy.Field() #date posted
            hashtags = scrapy.Field() #hashtags

            #image stats
            views = scrapy.Field() #number of views of the current image
            faves = scrapy.Field() #number of faves of the current image
            comments = scrapy.Field() #number of comments of the current image
            downloads = scrapy.Field() #number of downloads of the current image

            #artist details
            artists = scrapy.Field() #artist's name
            artist_urls = scrapy.Field() #link to the artist's account
            artist_deviations = scrapy.Field() #number of deviations (images) posted
            artist_comments = scrapy.Field() #number of total comments received
            artist_page_views = scrapy.Field() #number of total page views received
            artist_scraps = scrapy.Field() #number of scraps (WIPs or archived art)
            artist_watchers = scrapy.Field() #number of watchers (followers)
            artist_critiques = scrapy.Field() #number of critiques given
            artist_forum_posts = scrapy.Field() #number of forum posts made
            artist_faves = scrapy.Field() #number of total faves received
            artist_asl = scrapy.Field() #age, sex, location
            artist_dob = scrapy.Field() #date of birth
            account_age = scrapy.Field() #how old the account is

            # to be filled in by ImagePipeline
            image_paths = scrapy.Field() #location of image in local storage


        class MyImagesPipeline(ImagesPipeline):
            '''
            Image pipeline for downloading images.
            '''
            def get_media_requests(self, item, info):
                for image_url in item['image_urls']:
                    yield scrapy.Request(image_url)

            def item_completed(self, results, item, info):
                image_paths = [x['path'] for ok, x in results if ok]
                if not image_paths:
                    raise DropItem("Item contains no images")
                item['image_paths'] = image_paths
                return item
```

```
In [3]: class ImageSpider(scrapy.Spider):

            name = 'images'

            start_urls = ['https://www.deviantart.com/popular-all-time/?q=sherlock&offset=0']

            #initialize offset at 0
            offset = 0
            #set offset limit to control the amount of images downloaded
            offset_limit = 16000


            custom_settings = {
                'LOG_LEVEL': logging.INFO,
                'ITEM_PIPELINES': {'__main__.MyImagesPipeline': 1}, #enable image download
                'IMAGES_STORE': 'DA-images-2', #store images in DA-images-2 folder
                'FEED_FORMAT':'json',
                'FEED_URI': 'image-data-2.json', #store image data in image-data-2.json
                'DOWNLOAD_FAIL_ON_DATALOSS': False, #if image download fails (due to various
        issues), don't send error message, just flag it.
                'DOWNLOAD_DELAY': 0.25 #250 ms download delay, with inbuilt scrapy randomization
            }

            def parse(self, response):

                #get page body
                page = response.css('div.page-results span.thumb')

                for img in page:

                    #thumbnail link
                    thumbnail = img.css('::attr(data-super-img)').get()

                    #full link that leads to the individual image post
                    img_link = img.css('::attr(href)').get()

                    #if there is a thumbnail, aka the post is an image, follow url to scrape
        image details
                    if thumbnail:
                        yield scrapy.Request(img_link, callback = self.parse_image)

                #go to next page
                while self.offset < self.offset_limit:
                    self.offset += 24 #DA's natural offset scroll is set at increments of 24
                    next_page = f'https://www.deviantart.com/popular-all-
        time/?q=sherlock&offset={self.offset}'
                    yield scrapy.Request(next_page, callback=self.parse)


            def parse_image(self, response):

                #initialize image item
                image = ImageItem()

                #get image url (for downloading via ImagePipeline)
                image["image_urls"] = [response.css('div.dev-view-deviation img
        ::attr(src)').get()]
                #get other image info
                image["image_links"] = response.url
                image['titles'] = response.xpath("//a[@class='title']/text()").extract()[0]
                image['date_posted'] = response.xpath("//div[@class='dev-right-bar-content dev-
        metainfo-content dev-metainfo-details']/dl/dd/span/text()").extract()[0]

                #check whether image has hashtags (some don't)
                hashtag = response.xpath("//div[@class='dev-about-tags-cc dev-about-
        breadcrumb']/a/text()").extract()
                if hashtag: image['hashtags'] = hashtag
```

```
        #get image stats
        stats =  response.xpath("//div[@class='dev-right-bar-content dev-metainfo-
content dev-metainfo-stats']/dl/dd/text()").extract()

        #check that stats list only contains numbers (sometimes irregular data falls in)
        stats = [re.sub("\D", "", s) for s in stats]

        #remove any None types from list
        stats = list(filter(None, stats))

        #the responses are ordered in: views, faves, comments, downloads
        #sometimes comments are disabled, sometimes downloads are disabled
        headers = ['views','faves','comments','downloads']

        #if comments/downloads are disabled, they will not be looped over for a given
image
        for i in range(len(stats)):
            image[headers[i]] = stats[i]

        #get artist info
        artist_name = response.xpath("//small[@class='author']/span/a/text()").extract()

        if artist_name:
            image['artists'] =
response.xpath("//small[@class='author']/span/a/text()").extract()[-1]
            image['artist_urls'] =
response.xpath("//small[@class='author']/span/a/@href").extract()[-1]
            request = scrapy.Request(image['artist_urls'], callback=self.parse_artist,
meta={'image':image})
            yield request
        else: #if no artist name (sometimes artists are banned), just yield the image
            image['artists'] = 'Banned'
            yield image

    def parse_artist(self, response):

        #get image item for the higher-level parser
        image = response.meta['image']

        #get artist account stats
        artist_stats = response.xpath("//div[@id='super-secret-
stats']/div/div/div/strong/text()").extract()

        headers = ['artist_deviations','artist_comments','artist_page_views','artist_scr
aps','artist_watchers','artist_critiques','artist_forum_posts','artist_faves']

        for i in range(len(artist_stats)):
            image[headers[i]] = artist_stats[i]

        #get account age and membership details
        age_membership = response.xpath("//a[@href='#super-secret-
activity']/div/text()").extract()

        #sometimes the age is wrapped up in a span
        if len(age_membership) == 0:
            age = response.xpath("//a[@href='#super-secret-
activity']/span/div/text()").extract()[0]
            age_membership.append(age)
        image['account_age'] = age_membership[0]


        #get artist personal details
        artist_details = response.xpath("//div[@id='super-secret-
why']/div/div/div/dl/dd/text()").extract()
        details = ['artist_asl','artist_dob'] #some artists do not share their dobs
        for i in range(len(artist_details)):
            image[details[i]] = artist_details[i]
```

```
            return image

    process = CrawlerProcess()
    process.crawl(ImageSpider)
    process.start()
```

B Appendix: Analysis Code

https://nbviewer.jupyter.org/github/hueyning/DA-scraper/blob/master/python-notebook/DA-analysis.ipynb