Evaluation and pricing of reusable fashion products

Research for Machine Learning Solutions

Group members

Anna Margrét Jónasdóttir

Karl Arnar Ægisson

Margrét Vala Björgvinsdóttir

External Examiner: Gunnar Sigurðsson
Instructor: Sigurjón Ingi Garðarsson

TABLE OF CONTENTS

1 Introduction

In this research paper we will analyze different solutions to try to find the best one for our application. One of the requirements we got from our Product Owner at K3, was that the application should be able to automatically tag images of used clothing taken by the users of our application.

The given requirements were that as an A requirement, the application should be able to automatically tag both the clothing type and the colour of the clothing. The B requirements were that the application should be able to automatically tag the brand of the clothing and the condition of the clothing. The C requirements were that the application should be able to automatically tag the style of the clothing and the size of the clothing.

The main challenges that we face when we are looking at solutions to automatically tag the images is that most of the clothing has no visible logo of the brand and the condition of the clothing would be extremely difficult to determine due to that some clothing styles have distressed clothing as a style.

The Product Owner gave us a list of three general solutions that we could test and check if they give an acceptable rate of correct predictions. These were Google Vision AI, Microsoft Azure Computer Vision, Amazon Rekognition. We will test these providers to determine the correct prediction rate and if the rate is not acceptable, we will look at other solutions, such as automated machine learning models.

Automated machine learning models (AutoML) was chosen as a backup instead of making a machine learning model from the ground up as we didn't have the necessary knowledge building our own model. The AutoML works by feeding the model a predefined dataset with tagged images which the model can use to learn and make a prediction on similar images based on the images in the dataset.

Due to the AutoML needing a dataset, we talked to our Product Owner early in the process if he had a dataset available that we could use in case the general solutions didn't work. The reasoning for this is that it would take a fair amount to tag the images manually and it would be better served to spend our valuable hours working on other tasks.

When it became clear that there were no datasets available that fitted our needs, we decided that instead of manually tagging the images we would rather try to find websites that offered used clothing for sale and had images that fitted our needs and then build a small script in Python to scrape the images along with the tags that were input for those images. This would save us time to build our dataset in case the general solutions didn't work for our needs.

With all of this in mind, we had a plan for the automated tagging requirements. We would first test the general solutions to check if they actually worked for our application, if so, we would use the general solution. In case the general solutions didn't work, we would look at AutoML solutions and build our own dataset by scraping websites that offered used clothing with similar images that our users would use to get an automated tagging.

2 Analyzing the requirements

Once we had made a Product Backlog together with our Product Owner, we noted that there were four user stories in our Product Backlog that were related to the outcome of this research.

As we can see in Table 2.1, there is one A requirement, a requirement that we must implement, that had an estimated 55 hours to complete. Then we had two B requirements and one C requirement, requirements that we would only implement if we had enough time, that had an estimated 209 hours to complete.

Number	User Story	Priority	Estimated hours
5a	As a user I want the system to recognize the type and colour of the product	A	55
5b	As a user I want the system to attempt to recognize the brand of the product	B	5
5c	As a user I want the system to attempt to recognize the condition of the product	B	128
5d	As a user I want the system to attempt to recognize the style and size of the product	C	21

Table 2.1: Subset of our Product Backlog listing the requirements for the automated tagging of images

User story 5a is a must implement requirement. We will need our solution to successfully recognize both the type and the colour of the clothing item to be able to use the solution.

Since user story 5b is only estimated to take 5 hours to complete, we will combine it with the user story 5a in this research and try to find a solution that will correctly predict on the type, colour and brand. However the brand might be a problem since most of the clothing does not have a logo on the clothing, e.g. H&M, Next, etc., so if the brand gives us skewed results, we will drop it and focus on the type and colour.

User story 5c is estimated to take 128 hours to complete. The significant time estimated for the requirement is given because we thought that implementing this requirement would be almost impossible, given the different styles of clothing that are new but have a distressed style along with giving us a lot of false positives for clothing that has a bad condition but it is not shown on the image we would use to predict the condition of the clothing.

User story 5d is a C requirement, so we will only look at it if we have significant time left before the project deadline.

3 Vision AI Providers

In the beginning of the project, we were given a list of three potential providers that could provide a solution for our problem. These were Google Cloud Vision AI, Microsoft Azure Computer Vision, Amazon Rekognition.

3.1 Google Cloud Vision AI

Google Cloud's Vision AI offers powerful pre-trained machine learning models through REST and RPC APIs. A user can assign labels to images and quickly classify them into millions of predefined categories.[1]

The benefits of the Vision API is that a user can classify images using pre-trained models and predefined labels. A user can also compare photos to images in his product catalog, and return a ranked list of similar items. The API can also identify well-known product logos.

Google Cloud Vision AI offers an API online which can be used to test images to check the prediction it makes on the images.

Pricing is based on which features we would most likely need, in our case we would need label detection, logo detection and image properties, but also on the volume of predictions. The first 1000 units/month are free, but it will be charged from the 1001th image and onwards, as we can see in Table 3.1. Web detection might also be useful, and we will look into if it will be useful in our case.

Feature	Price per 1000 units		
	First 1000 units/month	Units 1001 - 5,000,000 / month	Units 5,000,001 - 20,000,000 / month
Label Detection	Free	$1.50	$1.00
Text Detection	Free	$1.50	$0.60
Logo Detection	Free	$1.50	$0.60
Image Properties	Free	$1.50	$0.60
Web Detection	Free	$3.50	Contact Google

Table 3.1: Google Vision AI pricing tiers[2]

[1] "Vision AI | Derive Image Insights via ML ...." https://cloud.google.com/vision. Accessed 25 Feb. 2020.
[2] "Pricing | Cloud Vision API | Google Cloud." https://cloud.google.com/vision/pricing. Accessed 25 Feb. 2020.

3.2 Microsoft Azure Computer Vision

Microsoft Azure Computer Vision API is a service that enables developers to analyze and retrieve information from images in a very simple way.[3]

Azure has a “detect brands” option which can be used to detect brands. This option could be beneficial with recognizing clothing logos.

Azure offers an API online, which can be used for testing and we will utilize this test API to check what predictions the API gives us for various images.

Pricing is based both on the features and the volume, as we can see in Table 3.2. Although it doesn’t offer us a free tier like Google Cloud Vision AI, the pricing is cheaper.

Product	Features	Price
Computer Vision S1 up to 10 requests per second	Tag Color	0-1M transactions - $1 per 1,000 transactions 1M-5M transactions - $0.80 per 1,000 transactions 5M+ transactions - $0.65 per 1,000 transactions
	Brand	0-1M transactions - $1.50 per 1,000 transactions 1M-5M transactions - $1 per 1,000 transactions 5M+ transactions - $0.65 per 1,000 transactions

Table 3.2: Microsoft Azure Cognitive Services pricing tiers[4]

[3] "Computer Vision | Microsoft Azure." https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/. Accessed 25 Feb. 2020.

[4] "Cognitive Services Pricing—Computer Vision API - Microsoft ...." https://azure.microsoft.com/en-us/pricing/details/cognitive-services/computer-vision/. Accessed 25 Feb. 2020.

3.3 Amazon Rekognition

Amazon Rekognition, an industry leader when it comes to providing highly accurate facial analysis, uses deep learning technology to analyze both images and videos. This technology can be utilized with little to no machine learning experience. With Amazon Rekognition, you can identify objects, people, text, scenes, and activities.[5]

Rekognition does not offer an API online which can be used for testing, so we will not be able to test the service against other providers unless we sign up to the service.

Pricing is based on volume, with steep discounts for each tier, as we can see in Table 3.3.

Cost type	Pricing	Price per 1,000 images
First 1M images processed* per month	$0.00116 per image	$1.16
Next 9 million images processed* per month	$0.000928 per image	$0.928
Next 90 million images processed* per month	$0.000696 per image	$0.696
Over 100 million images processed* per month	$0.000464 per image	$0.464
**Each API that accepts 1 or more input images, counts as 1 image processed.*		

Table 3.3: Amazon Rekognition pricing tiers[6]

[5] "Amazon Rekognition - Wikipedia." https://en.wikipedia.org/wiki/Amazon_Rekognition. Accessed 25 Feb. 2020.
[6] "Amazon Rekognition – Pricing - AWS." https://aws.amazon.com/rekognition/pricing/. Accessed 25 Feb. 2020.

4 Testing Vision AI Providers

Out of the three vision providers, two offered an online API test to check how their general service stacks up against the competition. Amazon Rekognition did not offer an API test unless signing up to their service, and since they are more geared towards facial recognition[7], we decided against looking at their service and will only be comparing the two other providers, Google Vision AI and Microsoft Azure Computer Vision.

When we were looking for potential images to use as a test, our Product Owner told us that ASOS had quite a large selection of images and we decided to take a closer look at the site. We decided to download five images from their site and sent each image to the API on the providers we were testing to check how well they fared in predicting each image.

The results from Google Cloud Vision AI were really good but when we were going through the process that the user would go through, we noticed that the images that the user would take did not match the images we downloaded from ASOS. Before choosing Google Cloud Vision AI as our solution, we wanted to check how the providers predicted on images that closely resembled the images our users of the application would use to get a prediction.

We found five such images while browsing Craigslist.org, which contains images of used clothing that a regular person is trying to sell. These images will closely resemble the images taken by our users and testing these images on the providers will give us a more accurate results on the predictions and therefore we can be more confident that the solution that we choose will work when we start the coding phase.

The test results from the ten images, five from ASOS and five from Craigslist, can be found in Tables 4.1 to 4.10 in Chapter 4.1.

[7] "We Built A Powerful Amazon Facial Recognition Tool ... - Forbes." 6 Jun. 2018, https://www.forbes.com/sites/thomasbrewster/2018/06/06/amazon-facial-recognition-cost-just-10-and-was-worryingly-good/. Accessed 25 Feb. 2020.

4.1 Vision AI Providers Test Reports

ASOS-1 Test Report

Name: ASOS-1

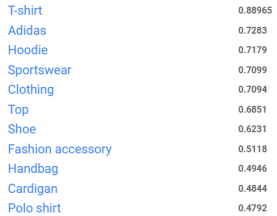
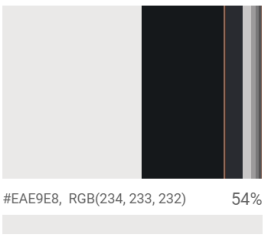
Type: Hoodie
Colour: Black
Brand: Adidas

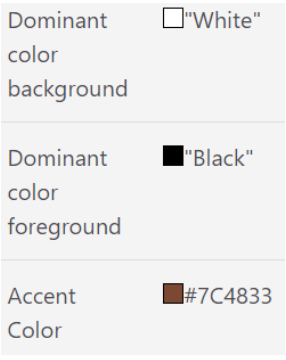
Description: A male model wearing a black Adidas hoodie. The Adidas logo is visible.

Elements to ignore: Person, cap, jeans, background

Tags from source: Adidas Training Mountain Logo Hoodie Black

Figure 4.1: ASOS-1 Image

Google Labels	Google Web	Google Colour	Google Conclusion
			The model accurately labeled the picture as a hoodie with a 98% confidence and the color black with 97%. The model accurately gave the type as a hoodie with a ~0.72 score and the brand as Adidas with a ~0.73 score in Web results. The color prediction was wrong.
Figure 4.2: Google Labels	Figure 4.3: Google Web	Figure 4.4: Google Colour	

Microsoft Labels	Microsoft Colours	Microsoft Conclusion
		The model accurately gave the type as hoodie with ~74% confidence, although it gave inaccurately the type as “sleeve”, “active shirt”, “outerwear”, “shirt”, “top”, “jacket”, “sweatshirt” and “t-shirt” before giving the type as a hoodie. The model didn’t give the brand as Adidas. The color was correct as “Black”
Figure 4.5: Microsoft Labels	Figure 4.6: Microsoft Colour	

Prediction Summary

Provider	Type	Brand	Colour
Google	✓	✓	✓
Microsoft	✗	✗	✓

Table 4.1: ASOS-1 Test Summary

ASOS-2 Test Report

Name: ASOS-2

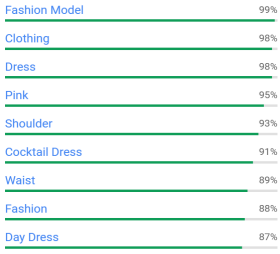
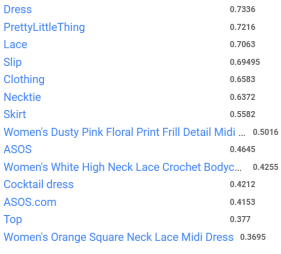
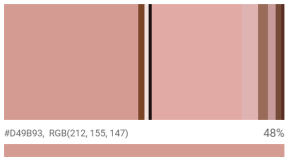
Type: Dress
Colour: Pink
Brand: PrettyLittleThing

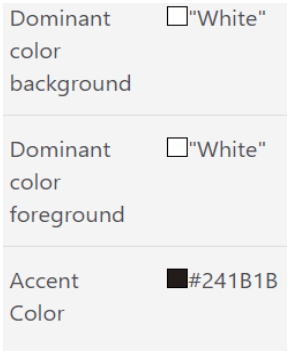
Description: A female model wearing a pink PrettyLittleThing dress, with black laces. The logo is not visible.

Elements to ignore: Person, background, heels

Tags from source: prettylittlething lace midi dress with tie detail in pink

Figure 4.7: ASOS-2 Image

Google Labels	Google Web	Google Colour	Google Conclusion
			The model accurately labeled the picture as a dress with a 98% confidence and the color pink with 95%. The model also accurately gave the type as a dress with a ~0.73 score and the brand as PrettyLittleThing with a ~0.72 score in Web results. The color was correct.
Figure 4.8: Google Labels	Figure 4.9: Google Web	Figure 4.10: Google Colour	

Microsoft Labels	Microsoft Colours	Microsoft Conclusion
		The model accurately gave the type as a day dress with ~83% confidence and as a cocktail dress with ~76% confidence. The model didn't predict the brand and the color was incorrectly predicted as "White".
Figure 4.11: Microsoft Labels	Figure 4.12: Microsoft Colour	

Prediction Summary

Provider	Type	Brand	Colour
Google	✓	✓	✓
Microsoft	✓	✗	✗

Table 4.2: ASOS-2 Test Summary

ASOS-3 Test Report

Name: ASOS-3

Type: Hoodie
Colour: White/Black
Brand: Puma

Description: A male model wearing a white/black Puma hoodie. The logo is visible.

Elements to ignore: Person, background, pants

Tags from source: Puma XTG Colour Block Hoody White

Figure 4.13: ASOS-3 Image

Google Labels

Figure 4.14: Google Labels

Google Web

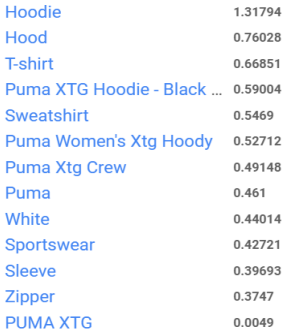


Figure 4.15: Google Web

Google Colour

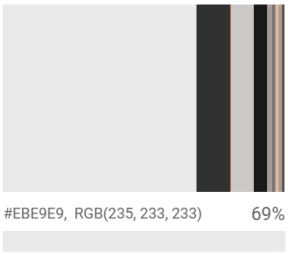


Figure 4.16: Google Colour

Google Conclusion

The model made a prediction that the image contained a Hoodie with a 79% confidence, however, it made several inaccurate predictions about the type that had a higher confidence. The web entities made a correct prediction about the type with a ~1.32 score, the brand with a ~0.46 score and the colour with a ~0.44 score. It gave the wrong prediction for the colour.

Microsoft Labels

Figure 4.17: Microsoft Labels

Microsoft Colours

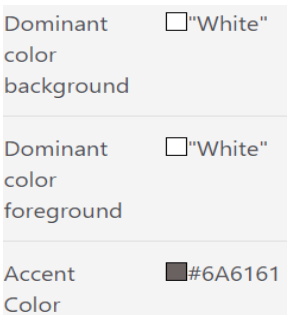


Figure 4.18: Microsoft Colour

Microsoft Conclusion

The model predicted the type as a hoodie with a ~77% confidence, but it had already made several other predictions for the type with a higher confidence. It didn't make a prediction on the brand but made a correct prediction on the colour as white.

Prediction Summary

Provider	Type	Brand	Colour
Google	✓	✓	✓
Microsoft	✓	✗	✗

Table 4.3: ASOS-3 Test Summary

ASOS-4 Test Report

Name: ASOS-4

Type: Top
Colour: Black/Light Blue
Brand: Vero Moda

Description: A female model wearing a light blue/black top from Vero Moda. The logo is not visible in the photo.

Elements to ignore: Person, pants, woman background

Tags from source: Vero Moda Floral Square Neck Top LightBlue&Black

Figure 4.19: ASOS-4 Image

Google Labels

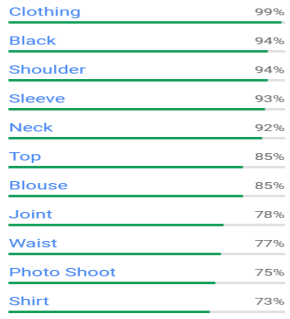


Figure 4.20: Google Labels

Google Web

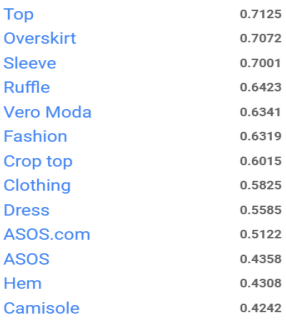


Figure 4.21: Google Web

Google Colour

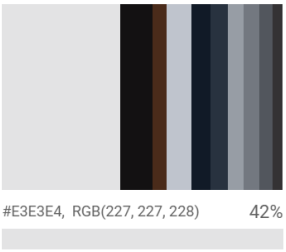


Figure 4.22: Google Colour

Google Conclusion

The model made a prediction that the image contained a top with 85% confidence, however, it also made a prediction that it was a blouse with 85% confidence. The web entities made a correct prediction about the type with a ~0.71 score, the brand with a ~0.63 score. It gave the wrong prediction for the colour.

Microsoft Labels

Figure 4.23: Microsoft Labels

Microsoft Colours

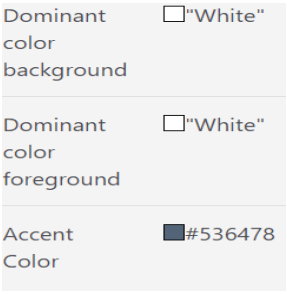


Figure 4.24: Microsoft Colour

Microsoft Conclusion

The model didn't predict the type as a top, however it made several other incorrect predictions about the type. It didn't make a prediction on the brand and made an incorrect prediction about the colour.

Prediction Summary

Provider	Type	Brand	Colour
Google	✓	✓	✕
Microsoft	✕	✕	✕

Table 4.4: ASOS-4 Test Summary

ASOS-5 Test Report

Name: ASOS-5

Type: Jeans
Colour: Blue
Brand: Vero Moda

Description: A female model wearing blue jeans from Vero Moda. The logo is not visible.

Elements to ignore: Person, heels, blouse, background

Tags from source: Vero Moda Skinny Jeans Blue

Figure 4.25: ASOS-5 Image

Google Labels

Figure 4.26: Google Labels

Google Web

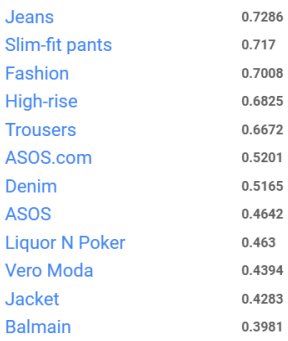


Figure 4.27: Google Web

Google Colour

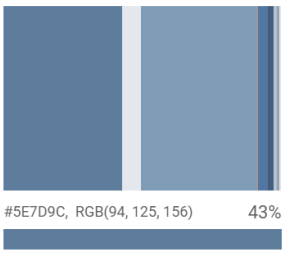


Figure 4.28: Google Colour

Google Conclusion

The model made a correct prediction that the type was jeans with a 99% confidence, and a correct prediction that the colour was blue. In web entities, the model made a correct prediction that the type was jeans with a ~0.73 score and the brand as Vero Moda with a ~0.44 score. It predicted the colour correctly as blue.

Microsoft Labels

Figure 4.29: Microsoft Labels

Microsoft Colours

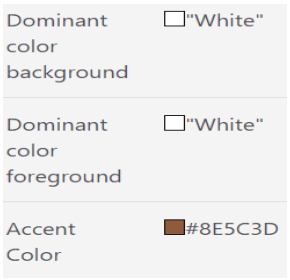


Figure 4.30: Microsoft Colour

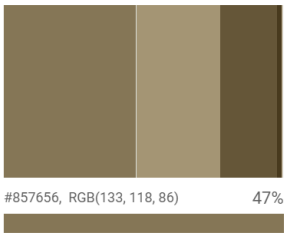
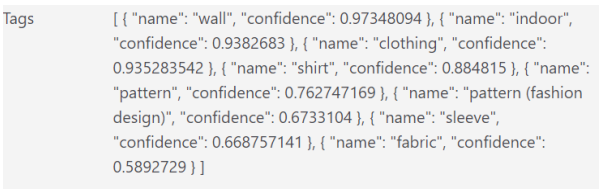
Microsoft Conclusion

The model made a correct prediction of the type as jeans with ~100% confidence, however it also made the prediction that the type was trousers with ~99% confidence. It didn't make a prediction on the brand and made an incorrect prediction about the colour.

Prediction Summary

Provider	Type	Brand	Colour
Google	✓	✓	✓
Microsoft	✓	✗	✗

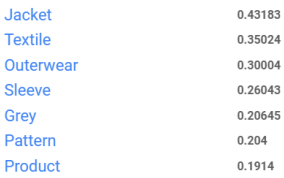
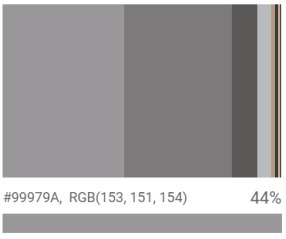
Table 4.5: ASOS-5 Test Summary

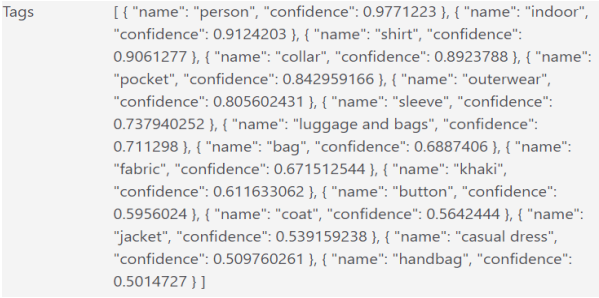
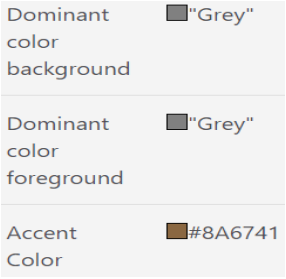
CRAIGSLIST-1 Test Report			
Name: CRAIGSLIST-1 **Type:** Shirt Long Sleeve **Colour:** Tan **Brand:** Knightsbridge **Description:** A Knightsbridge dress shirt hung on a hanger. The logo is not visible. **Elements to ignore:** Hanger, background **Tags from source:** Knightsbridge Dress Shirt NEW 100perc cotton Tan with black vertical and horizontal stripes			 Figure 4.31: CRAIGSLIST-1 Image
Google Labels	**Google Web**	**Google Colour**	**Google Conclusion**
 Figure 4.32: Google Labels	 Figure 4.33: Google Web	 Figure 4.34: Google Colour	The model didn't make an accurate prediction as a shirt, the web entities only made two predictions, one was a blouse and the other was plaid, both incorrect. The model didn't make a prediction on the brand but it made the correct prediction on the colour.
Microsoft Labels		**Microsoft Colours**	**Microsoft Conclusion**
 Figure 4.35: Microsoft Labels		 Figure 4.36: Microsoft Colour	The model made a correct prediction of the type as a shirt with ~88% confidence. It didn't make a prediction on the brand and made an incorrect prediction about the colour.

Prediction Summary			
Provider	**Type**	**Brand**	**Colour**
Google	✗	✗	✓
Microsoft	✓	✗	✗

Table 4.6: CRAIGSLIST-1 Test Summary

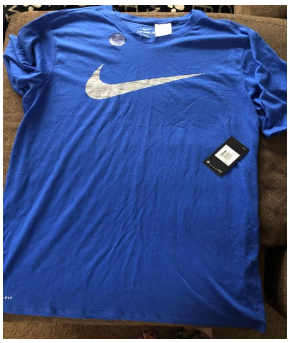
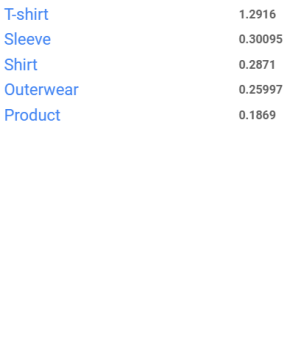
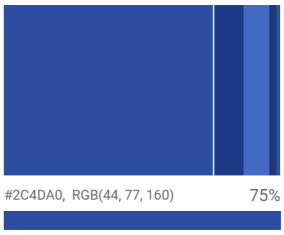
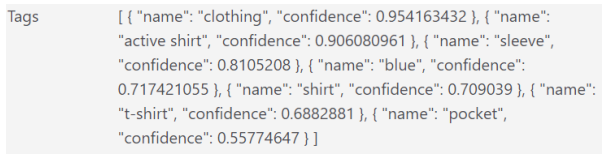
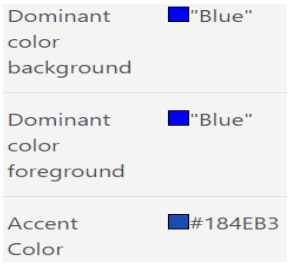
CRAIGSLIST-2 Test Report	
Name: CRAIGSLIST-2 **Type:** Long sleeve top **Colour:** Gray **Brand:** Nike **Description:** A gray Nike long sleeved top. The logo is visible. **Elements to ignore:** Background **Tags from source:** New Mens Nike DriFit top long-sleeve gray	 Figure 4.37: CRAIGSLIST-2 Image

Google Labels	Google Web	Google Colour	Google Conclusion
			The model didn't make an accurate prediction as a top, the web entities made a correct prediction about the colour as grey, although only with a ~0.21 score. The model didn't make a prediction on the brand but it made the correct prediction on the colour.
Figure 4.38: Google Labels	Figure 4.39: Google Web	Figure 4.40: Google Colour	

Microsoft Labels	Microsoft Colours	Microsoft Conclusion
		The model didn't make a correct prediction on the type, and it didn't make a prediction about the brand. However, it made a correct prediction about the colour being grey.
Figure 4.41: Microsoft Labels	Figure 4.42: Microsoft Colour	

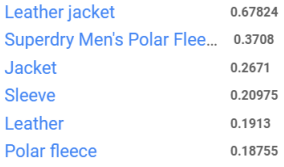
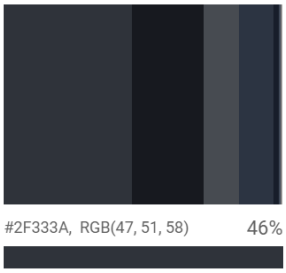
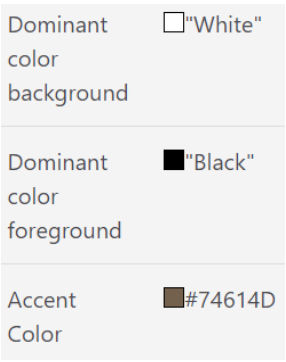
Prediction Summary			
Provider	Type	Brand	Colour
Google	✗	✗	✓
Microsoft	✗	✗	✓

Table 4.7: CRAIGSLIST-2 Test Summary

CRAIGSLIST-3 Test Report			
Name: CRAIGSLIST-3 **Type:** T-Shirt **Colour:** Blue **Brand:** Nike **Description:** A blue Nike T-shirt with Nike logo on front. The logo is visible. **Elements to ignore:** Background, sale tags **Tags from source:** Nike T-Shirt new blue with tags			 Figure 4.43: CRAIGSLIST-3 Image
Google Labels	**Google Web**	**Google Colour**	**Google Conclusion**
 Figure 4.44: Google Labels	 Figure 4.45: Google Web	 Figure 4.46: Google Colour	The model made an accurate prediction about the type as a T-shirt with 98% confidence and the colour as Blue with 98% confidence. In web entities, it made a correct prediction about the type as a T-shirt with a ~1.29 score. It didn't make a prediction about the brand but it made a correct prediction about the colour being blue.
Microsoft Labels		**Microsoft Colours**	**Microsoft Conclusion**
 Figure 4.47: Microsoft Labels		 Figure 4.48: Microsoft Colour	The model made a prediction about the type as a t-shirt with a ~69% confidence, although it predicted the type being several other types with higher prediction, and it made a prediction about the colour being blue with ~72% prediction. It didn't make a prediction about the brand, and it predicted the colour correctly.

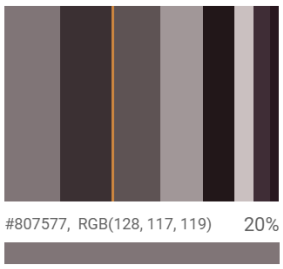
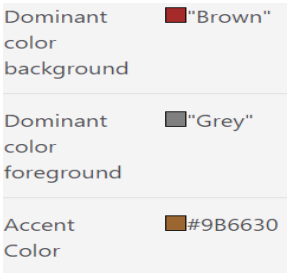
Prediction Summary			
Provider	**Type**	**Brand**	**Colour**
Google	✓	✗	✓
Microsoft	✗	✗	✓

Table 4.8: CRAIGSLIST-3 Test Summary

CRAIGSLIST-4 Test Report			
Name: CRAIGSLIST-4 **Type:** Jacket **Colour:** Black **Brand:** North Face **Description:** A black Denali jacket from North Face. The logo is visible. **Elements to ignore:** Background			 Figure 4.49: CRAIGSLIST-4 Image
Google Labels	**Google Web**	**Google Colour**	**Google Conclusion**
 Figure 4.50: Google Labels	 Figure 4.51: Google Web	 Figure 4.52: Google Colour	The model made an accurate prediction about the type as a jacket with 97% confidence. In web entities, it made an incorrect prediction about the type being a leather jacket with a score of ~0.68 and the brand being Superdry with a score of ~0.37. It predicted the colour right as black.
Microsoft Labels		**Microsoft Colours**	**Microsoft Conclusion**
 Figure 4.53: Microsoft Labels	 Figure 4.54: Microsoft Colour	The model made a prediction about the type as a jacket with ~100% confidence. It didn't make a prediction about the brand, and it predicted correctly about the colour being black.

Prediction Summary			
Provider	**Type**	**Brand**	**Colour**
Google	✓	✕	✓
Microsoft	✓	✕	✓

Table 4.9: CRAIGSLIST-4 Test Summary

CRAIGSLIST-5 Test Report			
Name: CRAIGSLIST-5 **Type:** Long sleeve shirt **Colour:** White/Black **Brand:** Under Armour **Description:** A white/black Under Armour plaid shirt. The logo is not visible. **Elements to ignore:** Background **Tags from source:** Under Armor Shirt Long Sleeve plaid			 Figure 4.55: CRAIGSLIST-5 Image
Google Labels	**Google Web**	**Google Colour**	**Google Conclusion**
 Figure 4.56: Google Labels	 Figure 4.57: Google Web	 Figure 4.58: Google Colour	The model made an accurate prediction about the type as a shirt with 94% confidence. The model didn't make a prediction about the brand but it predicted the colour right.
Microsoft Labels		**Microsoft Colours**	**Microsoft Conclusion**
 Figure 4.59: Microsoft Labels		 Figure 4.60: Microsoft Colour	The model made a prediction about the type as a shirt with ~100% confidence. It didn't make a prediction about the brand, and it didn't predict the colour correctly.

Prediction Summary			
Provider	**Type**	**Brand**	**Colour**
Google	✓	✗	✓
Microsoft	✓	✗	✗

Table 4.10: CRAIGSLIST-5 Test Summary

4.2 Vision AI Providers Test Conclusion

Using a pre-trained model at these two providers yielded different results as we can see in Table 4.11.

Dataset	# Types Google	# Brands Google	# Colours Google	# Types Microsoft	# Brands Microsoft	# Colours Microsoft
ASOS Dataset (5)	5	5	4	3	0	1
CRAIGSLIST Dataset (5)	3	0	5	3	0	3

Table 4.11: Number of correct predictions for each dataset

Microsoft didn't recognize the brand on any of the images, which probably needs to be tested with an account from Microsoft, however it recognized the colour on fourimages, three of them from Craigslist, and one from ASOS. It fared a little better on recognizing the type and successfully recognized the type on six images, three of them from Craigslist and three of them from ASOS. According to this small test, Microsoft predicts the colour correctly more often on images from Craigslist and predicts the same amount of types on both of the datasets.

Google however, fared much better than Microsoft. It recognized the brand on five images, all of them from ASOS. It also recognized the colour on nine images, five of them from Craigslist and four of them from ASOS. It recognized the type accurately on eight images, five of them from ASOS and three from Craigslist.

Google was much better at recognizing the colour than Microsoft and was the only provider to recognize any brand. However, it only recognized the brand through the ASOS images, which leads to the conclusion that the image must be on the internet for Google to capture any text around the image. Most likely, Google performs extremely well combined with a product catalog with images and features, according to the tests.

Since none of the providers actually offered good enough results on the images from Craigslist, we decided to start looking at other solutions that could potentially offer better results, but our backup approach if the providers did not give good enough results was to look at automated machine learning solutions.

5 Other Solutions

Since the vision models, that were pre-built by the providers, performed poorly on the Craigslist dataset, we decided that we needed to build our own model to make better predictions.

Our contingency plan for this was to look at automated machine learning solutions (AutoML), but auto machine learning has a high degree of automation which allows non-experts in the field of machine learning to make use of machine learning models and techniques.[8]

After looking at various AutoML providers, we decided to focus on two of them, Google AutoML and Microsoft Azure AutoML. There were many other providers in the field of AutoML, but since the nature of how AutoML works, that meant that we needed to sign up and provide a dataset for each provider to test them and compare the providers which would be too time consuming and possibly too pricey.

After consultation with our Product Owner and a quick chat with a machine learning engineer, we decided to choose Google AutoML and build a dataset to ultimately use a model there.

[8] "Automated machine learning - Wikipedia." https://en.wikipedia.org/wiki/Automated_machine_learning. Accessed 11 Mar. 2020.

5.1 Google AutoML

Google offers a service called AutoML to cater to developers that need a custom machine learning model with minimal effort and machine learning expertise. It relies on Google's transfer learning and neural architecture search technology.[9]

AutoML offers four options to integrate machine learning vision models, as we can see in Table 5.1. We decided that we would use AutoML Vision Classification as it enables us to train our own, custom model to classify our images according to the labels we define, such as "T-Shirt", "Blue" and "Adidas" for a blue Adidas t-shirt.

AutoML Vision Services	
AutoML Vision Classification	AutoML Vision Classification enables you to train your own custom machine learning models to classify your images according to labels that you define.
AutoML Vision Edge - Image Classification	AutoML Vision Edge enables you to build fast, high-accuracy custom models to classify images at the edge, and trigger real-time actions based on local data. Deploy these models to a variety of edge devices.
AutoML Vision Object Detection	AutoML Vision Object Detection enables you to train your own custom machine level models to detect and extract multiple objects, and provide information about each object including its position within the image.
AutoML Vision Edge - Object Detection	AutoML Vision Edge enables you to build fast, high-accuracy custom models to detect multiple objects in images at the edge, and trigger real-time actions based on local data. Deploy these models to a variety of edge devices.

Table 5.1: Google AutoML vision services[10]

As we can see in Table 5.2, the pricing is $3.15 for each node hour to train a dataset. The timeframe for each training varies from 1 node hour to 500 node hours. The significant spread is due to the varied amount of rows in the dataset along with the amount of columns. The first 40 node hours are free for training and the same for deploying the model to get a prediction.

Image Classification	Free	Paid
Training	First 40 node hours are free (one time)	$3.15 per node hour
Deployment and Online (Individual) Prediction	First 40 node hours are free (one time)	$1.25 per node hour
Batch Prediction	First node hour is free (one time)	$2.02 per node hour

Table 5.2: Google AutoML pricing tiers[11]

[9] "Cloud AutoML - Google Cloud." https://cloud.google.com/automl. Accessed 25 Feb. 2020.
[10] "AutoML Vision documentation | Cloud AutoML Vision ...." https://cloud.google.com/vision/automl/docs. Accessed 25 Feb. 2020.
[11] "Pricing | Cloud AutoML Vision | Google Cloud." https://cloud.google.com/vision/automl/pricing. Accessed 25 Feb. 2020.

The process for deploying a model on Google AutoML and being able to make a prediction is a few steps.

As we can see in Fig. 5.1, Google AutoML needs a dataset to be able to work. Once the dataset is provided, we can train the dataset, and deploy a model based on the dataset and finally serve predictions we send the deployed model. We can see the steps broken further down in Table 5.3.

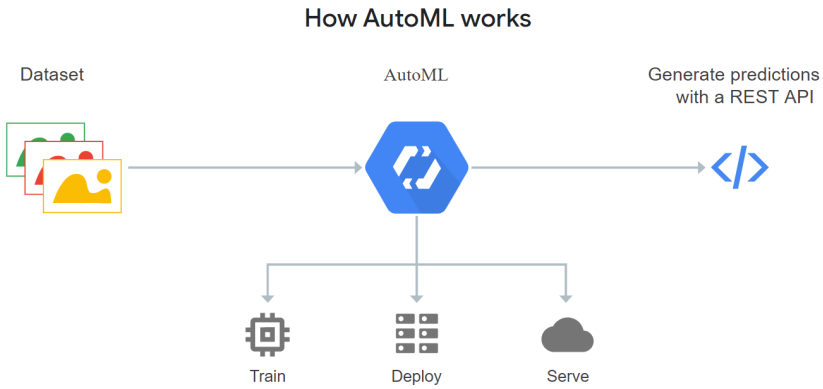


Figure 5.1: How Google AutoML works

Google AutoML Process	
Compiling a list of images	We need to compile a list of images that we will use to train a dataset.We will need to find either a source that has tagged images in a dataset or build our own by scraping a site.
Copying the images to Google Cloud	We will need to copy the images to a storage bucket in Google Cloud.
Creating a .csv document	We will need to create a .csv image that contains the link to the image location on our storage bucket, along with the type, colour and brand.
Create a dataset	We will need to create a dataset on Google Cloud that uses the .csv document we created in the previous step.
Train the dataset	Once we create a dataset that contains the image location and the tags associated with each of the images, we will need to train the dataset. We will need to find the most optimal node hour to train the dataset to keep the cost down.
Validate the model	Once the data training is done, we will need to check the score of the model and determine if the model is good enough. If not, we will need to either clean our dataset better or find another source of data.
Deploy the model	If the score on the model is good enough, we will need to deploy the model and test the model before we decide to use the model. The tests will be similar to the tests we did in Chapter 4.
Use the model	Once we deployed the model and tested the model, we will use the model to generate predictions through our application.

Table 5.3: Google AutoML process[12]

[12] "AutoML Vision Beginner's guide - Google Cloud." https://cloud.google.com/vision/automl/docs/beginners-guide. Accessed 25 Feb. 2020.

5.2 Steps we need to take to deploy a successful model

We then further broke down the process into steps that we needed to take to be able to successfully deploy a model that met our needs, these steps can be viewed in Table 5.4 and we will follow this process in the next chapters.

Our process	
Step 1	We need to either find a dataset with tagged images or we need to find websites that we can scrape that have images of used clothing along with the tags that we require. The images need to closely resemble the images that we tested in Tables 4.6 to 4.10, or images that do not have a model wearing clothes. If we find a dataset with tagged images, we can go to step 5, otherwise we will need to go to step 2.
Step 2	We need to analyze the websites we found in step 1 and build a strategy on how we can successfully scrape the images and tags associated with the images on these websites.
Step 3	Once we have a strategy, we can start building the scraper code. We have experience with building a scraper using Python and Python modules “requests” and “BeautifulSoup” and we will use this experience to build the scraper code. We can then reuse parts of the code for the first website to build code for the other websites.
Step 4	When step 3 is completed, we will need a server to run our code and scrape the data. Scraping the data can take several days so it will not be possible for us to scrape the data through our personal laptops. The server will scrape the tags into a .csv file along with the image url and then download the images based on a list of the image urls.
Step 5	We will then use Jupyter Notebooks to view our dataset and clean the dataset if needed. This is done so that we will get low amounts of false positives due to a “dirty” dataset.
Step 6	We will need to talk to our Product Owner about getting an account at Google Cloud to proceed with the next steps. An account at Google Cloud is essential to start the next steps, so we will need to get an account as soon as possible.
Step 7	Once we have an account at Google Cloud, we will need to copy the images to a Google Cloud bucket according to step 2 in Table 5.3.
Step 8	We will then need to get the image locations in our bucket to create a .csv file containing the image locations along with the labels we want to use to train the dataset.
Step 9	Once we created the .csv file in step 8, we will use it to start training a new model.
Step 10	Once the model is trained, we will need to validate the results. If the results don’t deliver a good enough score, we will need to either go back to step 5 and clean the dataset better or back to step 1 and find new sources to build a new dataset.
Step 11	If the model shows good enough results, we can start deploying the model. Before we start using the model for our application, we will need to test it using images that closely resemble the images in Chapter 4. If the tests are successful, we will move on to step 12, else we need to go back to step 5 and if that fails, back to step 1.
Step 12	The final step is to use the deployed model to predict the images we send it. This will be the final model we will use and no further testing for other models will be done unless we have a significant time before the deadline on the project to tune the model to make better predictions.

Table 5.4: Our process to successfully build a deployable model on Google AutoML

6 Finding websites to scrape

According to step 1 in Table 5.4, we either needed to find a dataset that contained tagged images or we needed to build our own dataset. We consulted with our Product Owner to check if he had any tagged images in a dataset that could be useful for us since we would save a significant amount of time using a known dataset instead of building our own. After consultation with his colleagues, he came to the conclusion that there was no dataset that would be useful for us. We therefore decided to start looking at potential websites to start scraping images and tags.

A quick search on Google yielded two potential websites that we could potentially scrape. These websites were Oxfam.org.uk and Craigslist.org. Both of these websites offered the potential to build a dataset containing tagged images that do not contain a model wearing the clothing, which resembled the images that our users would use.

When we were looking at a strategy to scrape Craigslist.org, we noticed that only a small percentage had filled in tags. Building a dataset would therefore take a tremendous amount of time due to the need of cleaning and filling in the missing values. We therefore ultimately decided against scraping Craigslist.org.

This reason along with the fact that the images we scraped from Oxfam were extremely low quality led us to look at other potential sources. This search led us to two more sites, Netflea.com and Swap.com. Both of these sites fit the profile of the websites we were looking at and both offer a significant amount of products that can be scraped.

However Swap.com was built using React and our scraping tool, that we built, only worked on websites where pagination was in the url. We could have used an automated browser tool such as Selenium, but since we were using an headless server to run our scraping scripts, this was not possible. This resulted in us scrapping the idea of scraping Swap.com.

We therefore had two websites that we successfully built datasets with images and tags associated with the images, Oxfam.org.uk and Netflea.com.

When training models based on the Oxfam.org.uk dataset and the Netflea.com dataset it became clear that the datasets were not good enough. Both of them yielded subpar precision scores and upon further inspection, we noticed that a lot of the images were incorrectly labeled from the source.

After consultation with our instructor, who told us that we could make the solution work at a subset of clothing items, we decided to manually build a dataset containing two brands, “Adidas” and “Nike”, three types, “Hoodie”, “Sweatshirt” and “T-Shirt”, and five colours, “Black”, “Blue”, “Gray”, “Red” and “White”. The reasoning for this was that if we would have a small subset, it would mean that we could get more images for each label which would result in higher precision scores.

With this in mind, we decided to look at websites that offered a significant amount of images that fitted our dataset. This search led us to further four websites, Grailed.com, Shpock.com, Picclick.com and Poshmark.com so in total we looked at eight websites.

6.1 Oxfam.org.uk

Oxfam.org.uk is a confederation of 19 independent charitable organizations focusing on the alleviation of global poverty, founded in 1942. Oxfam.org.uk has shops all over the world, which sell many fairtrade and donated items, one of these items is used clothing and the items can be viewed at their online shop.[13]

The clothing items are categorized in three subcategories, women's clothing, men's clothing and kids clothing (which is further subcategorized into girls clothing and boys clothing). The site for each product is well documented with information related to the item and we will use this to build our dataset.

As we can see in Fig. 6.1, each product listing has several tags associated with each item along with a picture of the item, "Size", "Colour", "Brand", "Brand type", "Condition", "Exact fabric", "Garment care" and "Type". Out of these we are most interested in the "Colour", "Brand" and "Type" tags.

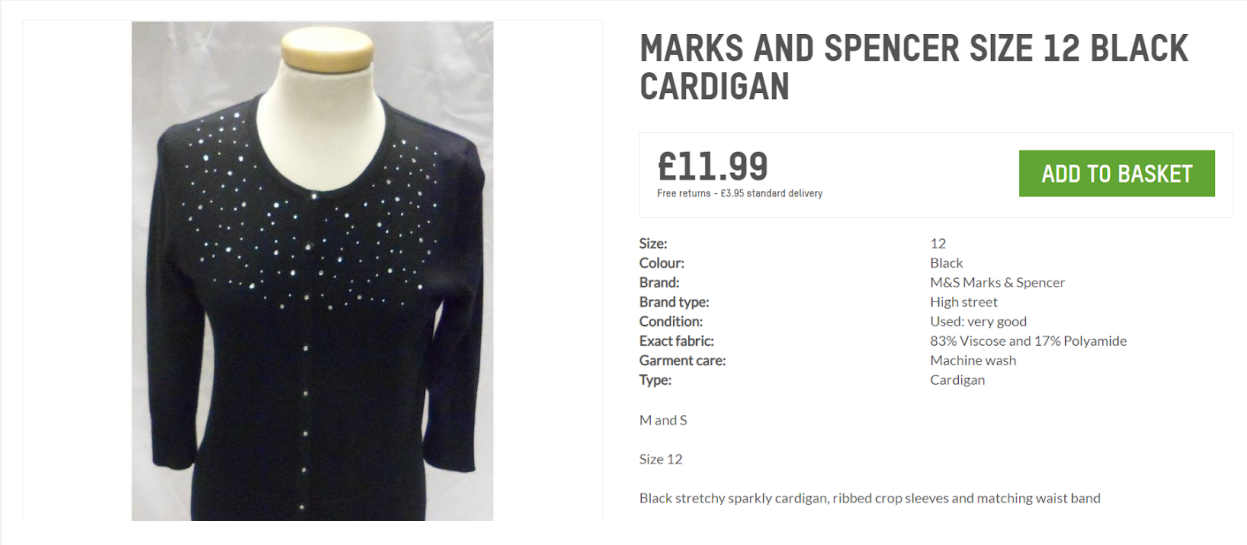


Figure 6.1: Oxfam.org.uk product listing example

The image is hosted on the website, so we can save the image location and download the image at a later date.

Upon further inspection of other product pages, we found two more tags that could potentially benefit us when we are cleaning the data. These are "Exact Colour" and "Title". We plan on using the "Exact Colour" tag to fill in missing values in the "Colour" column and "Title" column to fill in any of the three columns if they have a missing value. This could potentially be of great value as we won't be needing to drop rows that contain a missing value, since we can try to fill in the missing value and therefore keep the row.

[13] "Oxfam's Online Shop - Oxfam GB." https://www.oxfam.org.uk/shop. Accessed 25 Feb. 2020.

6.2 Craigslist.org

Craigslist.org is an American classified advertisements website which was founded in 1995.[14] It offers various sections devoted to miscellaneous items, but we will concentrate on the “clothes+acc” section.

Each listing offers the option to add an image, tags and a text associated with the listing.

Our main objective is to scrape the image url so that we can download the image, and scrape the tags which can be seen on the bottom right in Fig. 6.2. Since there aren’t many tags, we will also be looking at scraping both the title and the description and try to use them to fill in missing values.

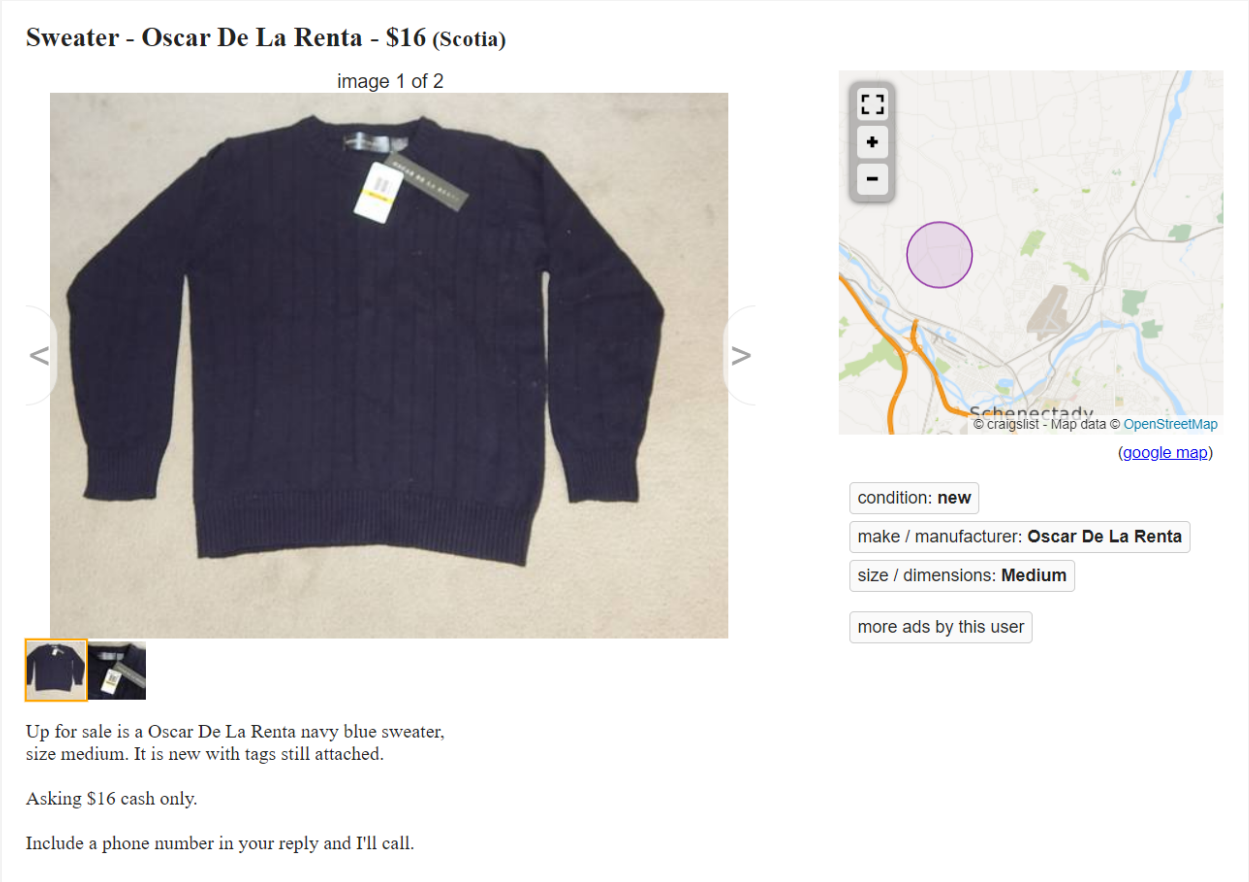


Figure 6.2: Craigslist.org.uk product listing example

Since there are many locations within the United States, we will focus on scraping the cities with the most population, such as San Francisco, Los Angeles, Dallas, Miami, etc, in order to minimize the amount of different cities to scrape.

[14] "Craigslist - Wikipedia." https://en.wikipedia.org/wiki/Craigslist. Accessed 25 Feb. 2020.

6.3 Netflea.com

Netflea is an online fleamarket where items are sold by consumers to consumers, where Netflea acts as a trusted third party between the consumers. Founded in 2013, Netflea has been a leader in online flea markets in Europe.[15]

As we can see in Figs. 6.3 to 6.6, the product listings all have the information we are looking for, that is "Type", "Brand" and "Colour". The other tags don't offer any valuable information that we could potentially use if we needed to fill in missing values so we will be focusing on the three tags mentioned.

DRESS - POLA - L

€11.90

Category	Women's dresses and skirts	Type	Dress	Size	L	Brand	Pola
Rating	4. Barely used	Color	Gray	Item ID	61309-10		

Figure 6.3: Sample #1 from Netflea

QUILTED JACKET - H&M - 34

€11.00

Category	Women's coats and jackets	Type	Quilted jacket	Size	34
Brand	H&M	Rating	4. Barely used	Color	Olive green
Additional Information	Zipper front	Item ID	59844-78		

Figure 6.4: Sample #2 from Netflea

WOOL COAT - SARA KELLY BY ELLOS - 38

€18.90

Category	Women's coats and jackets	Type	Wool coat	Size	38
Brand	Sara Kelly By Ellos	Rating	3. Good	Color	Red
Additional Information	Villa	Item ID	14647-2122		

Figure 6.5: Sample #3 from Netflea

RUNNING JACKET - NEWLINE - L

€10.90

Category	Women's sportswear	Type	Running jacket	Size	L
Brand	Newline	Rating	4. Barely used	Color	Black / Red
Item ID	69101-20				

Figure 6.6: Sample #4 from Netflea

[15] "About Us -Netflea.com." https://www.netflea.com/about-us. Accessed 2 Mar. 2020.

6.4 Swap.com

Swap.com was founded in 2012 and is an online consignment store offering used clothing. Sellers price their own items and Swap.com handles the fulfillment and returns process for items sold.[16]

As we can see in Table 6.4, the product listings all have tags that we are interested in scraping. These are "Type", "Brand" and "Colour". The other tags don't offer any valuable information that we could potentially use if we needed to fill in missing values so we will be focusing on the three tags mentioned.

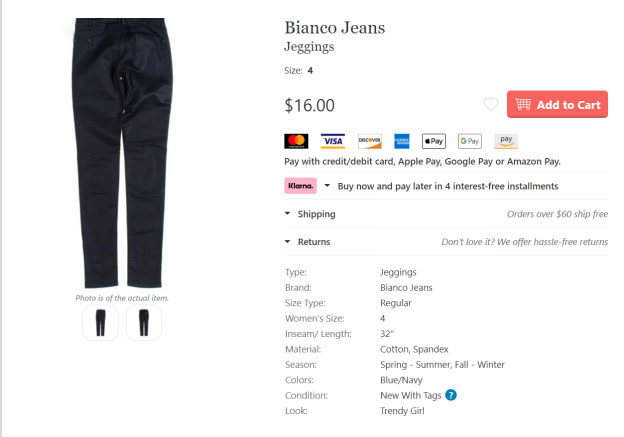


Figure 6.7: Sample #1 from Swap.com

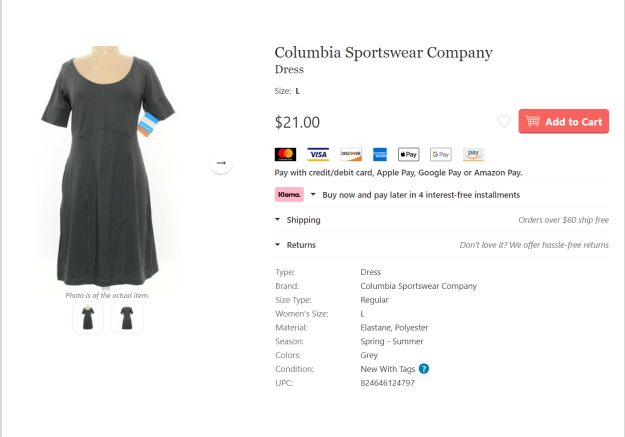


Figure 6.8: Sample #2 from Swap.com

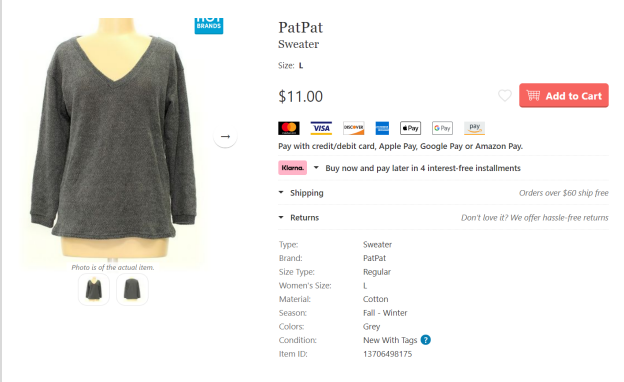


Figure 6.9: Sample #3 from Swap.com

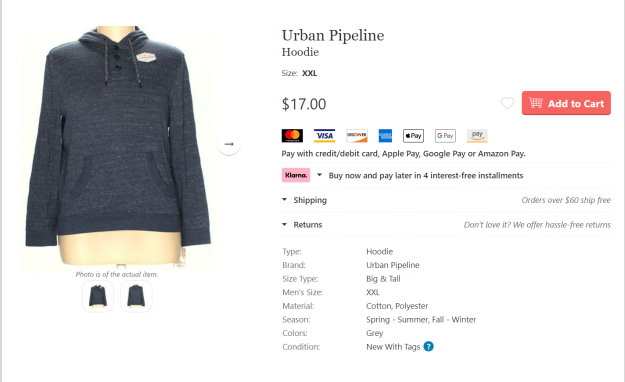


Figure 6.10: Sample #4 from Swap.com

[16] "Swap.com - Wikipedia." https://en.wikipedia.org/wiki/Swap.com. Accessed 2 Mar. 2020.

6.5 Grailed.com

Grailed.com is a community driven marketplace for men's fashion and streetwear where users can flip their wardrobe on the site.[17]

Founded in 2013, Grailed.com also offers a sister site for women's clothing called Heroine.com[18], although the brands Adidas and Nike do not have as high a volume on the site as Grailed.com.

Since we will be manually extracting the information and manually entering the tags based on the image, we will be focusing on the category page as opposed to the product page as we did with the previous four sites.

As we can see in Fig. 6.11, the category page for Nike sweatshirts and hoodies contain an image along with a title, description, size and the price. We will be focusing on getting images that closely resembles the images that the user of our application will take and we can see that two images from the category page matches our wishes, that is the first image which is a grey Nike hoodie and the third one which is a blue Nike sweatshirt.

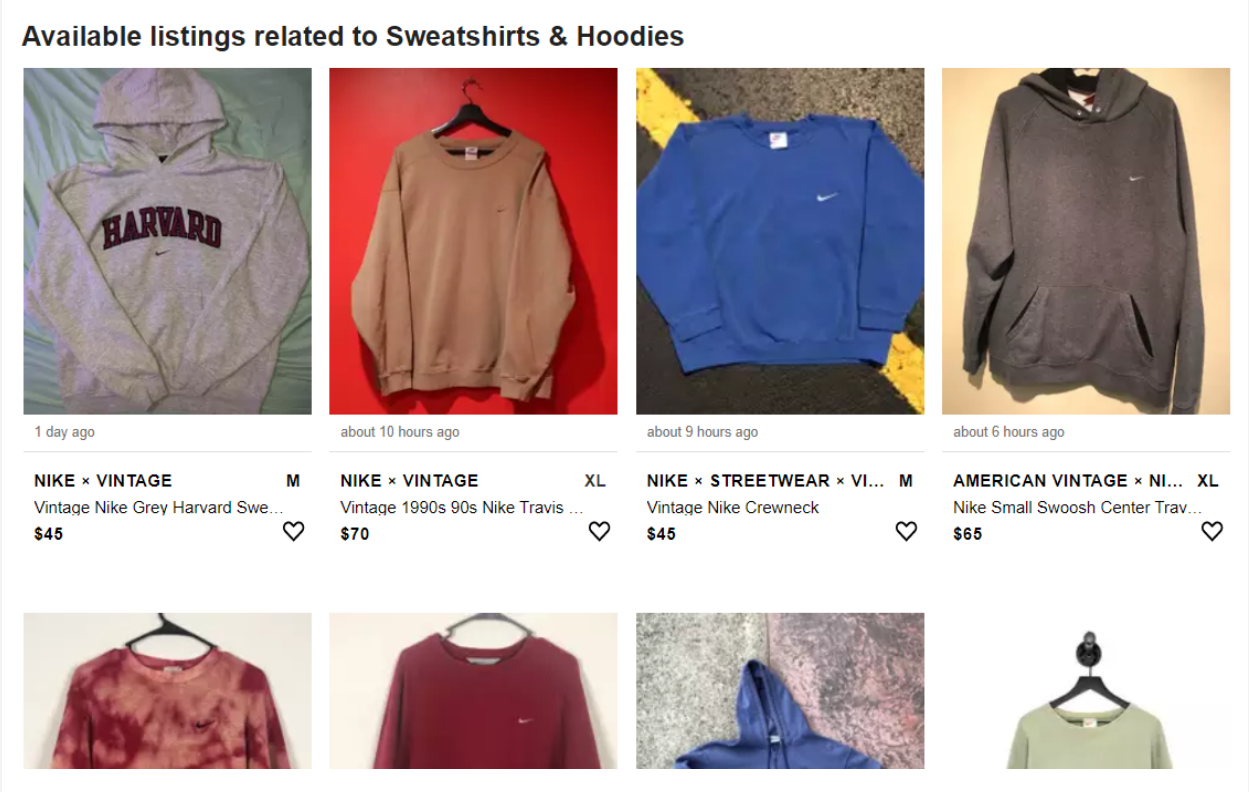


Figure 6.11: Nike sweatshirts & hoodies at Grailed.com

[17] "About Us | Grailed." https://www.grailed.com/about. Accessed 3 Apr. 2020.
[18] "Curated Fashion Marketplace Grailed Raises $15 Million in ...." 21 Jun. 2018, https://www.businesswire.com/news/home/20180621005084/en/Curated-Fashion-Marketplace-Grailed-Raises-15-Million. Accessed 3 Apr. 2020.

6.6 Shpock.com

Shpock.com is an online marketplace platform that uses a mobile app or a browser for private buying and selling in an area.[19]

Launched in 2012, Shpock is short for "**Sh**op in your **pock**et" and is consistently ranked as one of Europe's most popular shopping apps.[20]

Since we will be using this site to build our backup manual dataset, we will only be looking at how the categories list their products on the site. Although Scpock.com doesn't have category pages, you can search the site for the desired products, e.g. "Adidas T-Shirt".

As we can see in Fig. 6.12, the image we would be interested in getting is image #4, or a white Adidas t-shirt. Image #1 would usually be included but it is multi-coloured and since we can't be sure if the t-shirt is blue or black, we will exclude it.

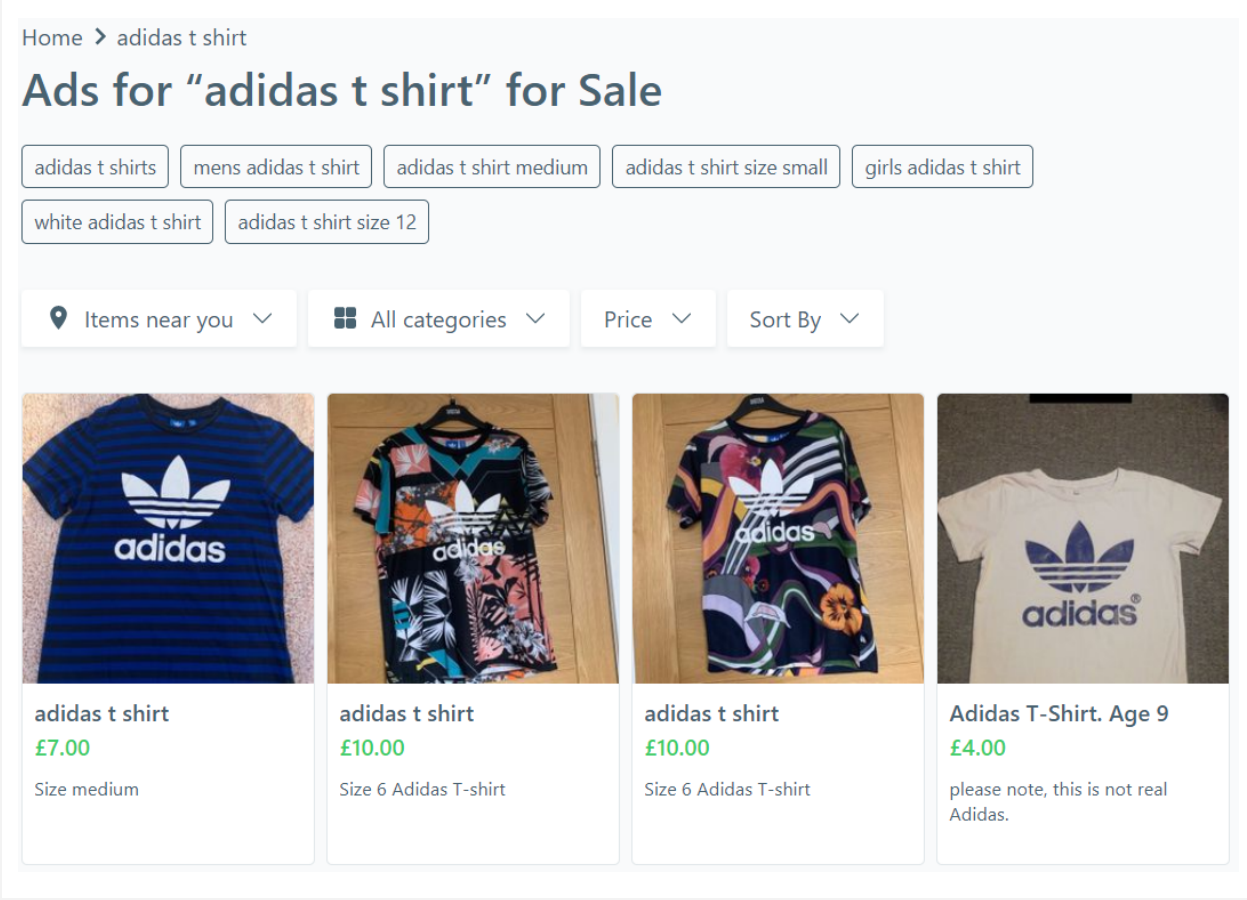


Figure 6.12: Adidas T shirts at Shpock.com

[19] "Shpock - Wikipedia." https://en.wikipedia.org/wiki/Shpock. Accessed 6 Apr. 2020.
[20] "Our Values - Jobs - Shpock." https://jobs.shpock.com/shpock/. Accessed 6 Apr. 2020.

6.7 Picclick.com

Picclick.com is a website that allows a user to search eBay.com faster by optimizing the website for visual browsing. The result is that it makes it much easier to see thousands of products by showing all the products in a thumbnail gallery.[21]

Founded in 2008, Picclick.com is primarily used by eBay power buyers, but we can use the site for our purpose of gathering the correct images for our dataset.[22]

Since we are only interested in getting the image, not the tags, we will manually get the image url by looking at the images and determine if the image closely resembles the images our users will use to make a prediction.

As we can see in Fig. 6.13, the category page for Adidas T-shirts offers a lot of images that we can quickly scroll through to find the images we want. We can see that of the 16 first images in the category, we would be interested in getting image #1, image #9 and image #11.

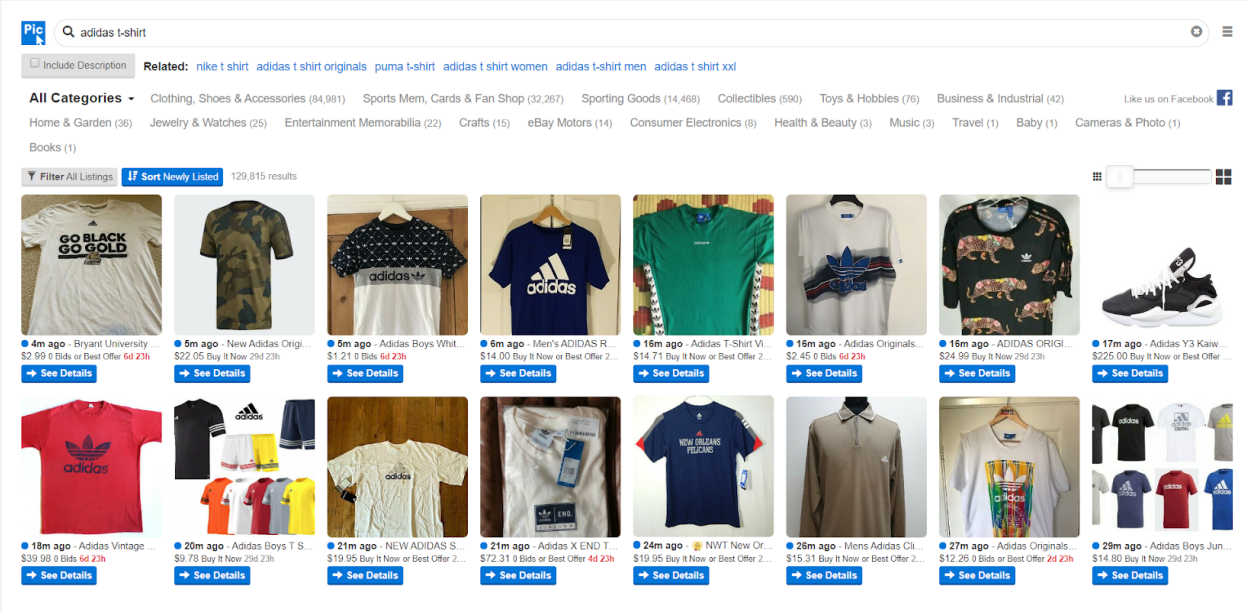


Figure 6.13: Adidas T shirts at Picclick.com

[21] "Meet the company: PicClick - Tamebay." 26 Feb. 2018, https://tamebay.com/2018/02/meet-the-company-picclick.html. Accessed 7 Apr. 2020.
[22] "PicClick About Us." https://picclick.com/pages/about.html. Accessed 7 Apr. 2020.

6.8 Poshmark.com

Poshmark.com is a social commerce marketplace where people in the United States can buy and sell new or used clothing, shoes and accessories. The company was launched in 2011.[23]

Poshmark.com enables people to buy and sell items to each other. The site has over 25 million items and 5.000 brands.[24]

Since we are only interested in getting the image and not the tags, we will only be looking at the category pages as opposed to looking at the product pages.

As we can see in Fig. 6.14, the women category for Adidas T-shirts contains one image that we might be interested in adding to our dataset. Image #6, a white t-shirt, might be included since we can instantly recognize the brand as Adidas and the colour as white. Image #1 would be included if the image only contained the t-shirt and image #3 would be included if we could quickly determine that the brand is Adidas, which we can't in this case.

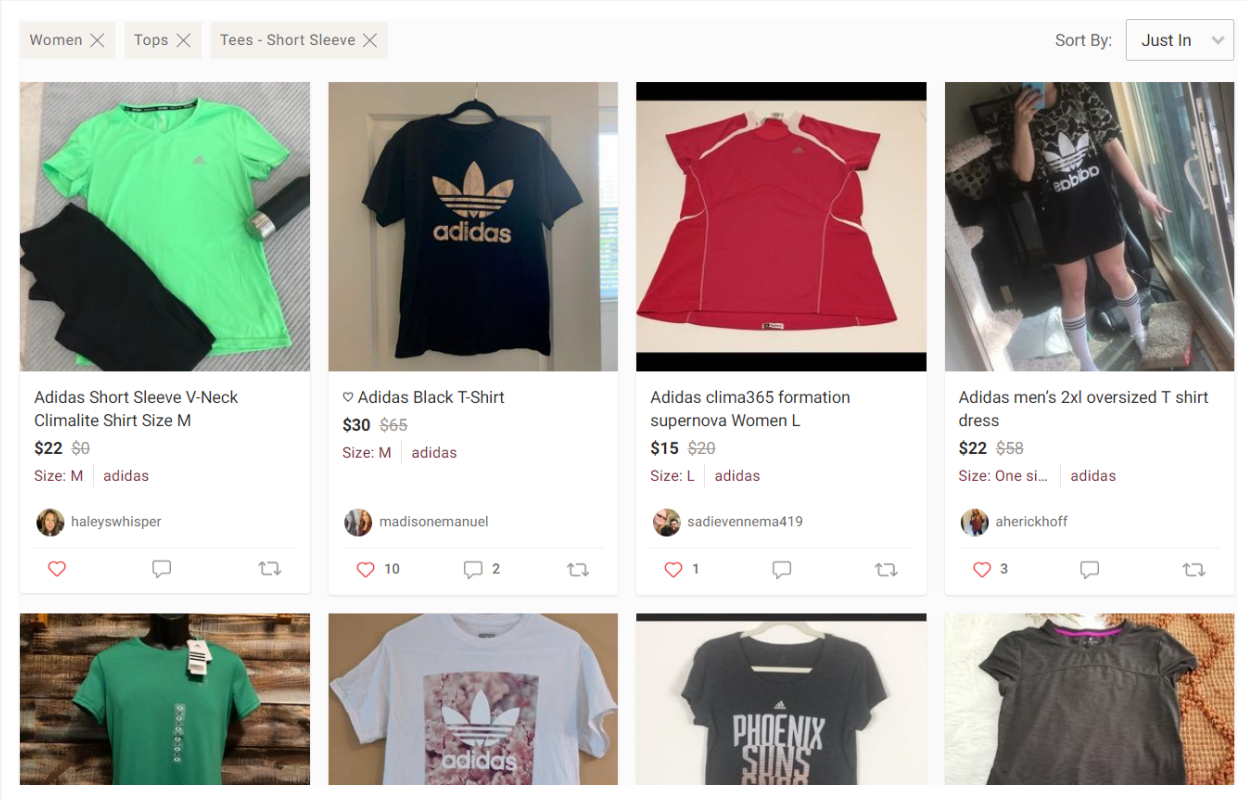


Figure 6.14: Adidas T shirts in women's category at Poshmark.com

[23] "Poshmark - Wikipedia." https://en.wikipedia.org/wiki/Poshmark. Accessed 7 Apr. 2020.
[24] "About Poshmark." https://poshmark.com/about. Accessed 7 Apr. 2020.

7 Analyzing the websites

We had two sites we were planning on scraping in the beginning, Oxfam.org.uk and Craigslist.org. When we were thinking about a strategy to scrape the sites, we decided to first go through Oxfam.org.uk and build a skeleton code which could be reused to scrape Craigslist.org.

When we had built the skeleton code based on the analysis of the Oxfam.org.uk, we started to analyze the Craigslist.org website. This analysis showed that the product pages on Craigslist.org differed a lot in terms of if the seller of the product had filled in the tags. Most of the information related to the product was contained either within the title or within the description field. This led us to two options. We could either not scrape the Craigslist.org site or we could scrape the site and later try to fill the missing values with information contained in the title or description, but ultimately decided against it, as it would be too time consuming.

Since we decided to not scrape Craigslist.org and the low quality images from Oxfam.org.uk, we decided to expand our search for other sites. This search led us to Netflea.com and Swap.com.

When we were analyzing the second set of websites, we decided that we would expand our due diligence of the sites to include looking at several product pages to make sure that we wouldn't get the same problem as with Craigslist.org, that is, too much difference in how tags are presented on the site. We also decided to check random images to check the image size to make sure that the image quality would be good, which was not the case with Oxfam.org.uk.

Both sites in our second set passed these tests, although later analysis of the Swap.com revealed that the site was coded in React and therefore the complexity of building a code would be significantly higher as we could not reuse the skeleton code we had, and the fact that our server that we used to scrape the sites was headless, and scraping React websites would require an automated browser.

Ultimately, from these four sites, we decided to build a scraping tool for two of them, Oxfam.org.uk and Netflea.com.

As we explained in the beginning of chapter 6, the two datasets that we had from Oxfam.org.uk and Netflea.com produced subpar precision scores when training models based on them, which lead us to building a manual dataset containing images from four websites, Grailed.com, Shpock.com, Picclick.com and Poshmark.com. Although we didn't need to build a scraper for these sites, as we were going to manually go through them, we still needed to analyze them in order to get the most optimal pictures from the sites.

7.1 Finding a strategy to scrape Oxfam.org.uk

Before writing the code for scraping the Oxfam.org.uk website, we looked at the site to get a better overview on what we needed to code and how we would extract the information from the website. This information was used to build a skeleton code that would be reused for scraping Netflea.com

First, we noted that there were 3 categories we wanted to scrape, women's clothing, men's clothing and kids clothing.

In women's clothing there is a subcategory where we can view all the items currently listed for sale, as we can see in Fig. 7.1.

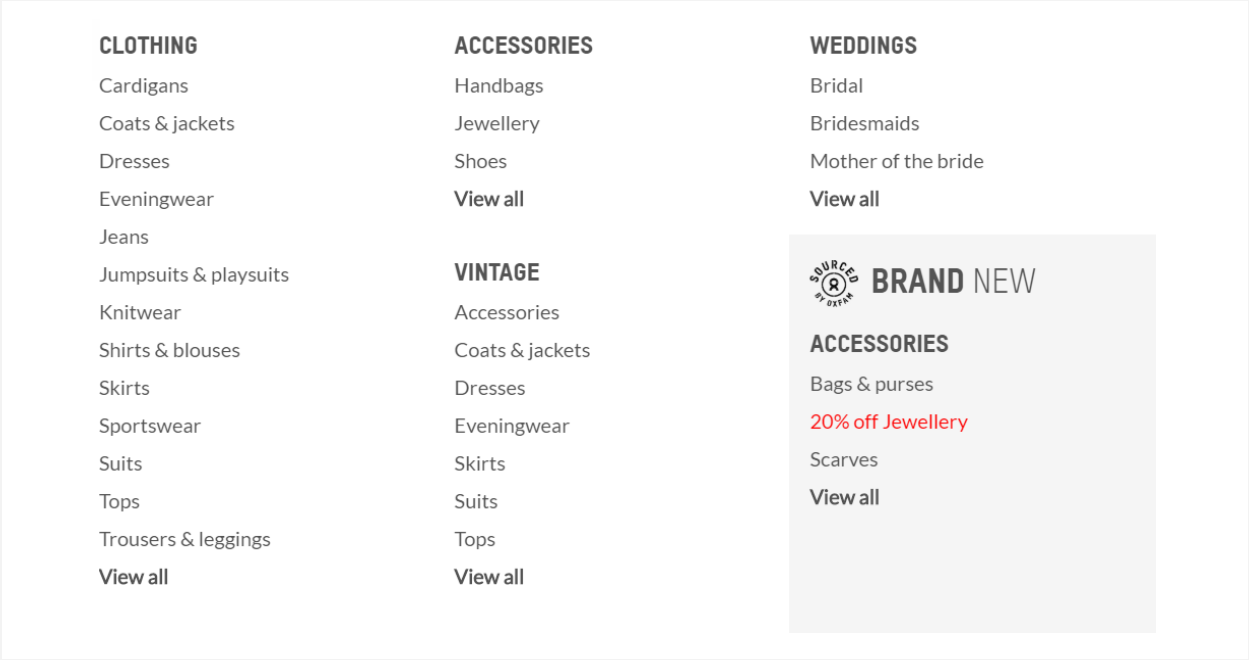


Figure 7.1: Oxfam.org.uk subcategories for women's clothing[25]

We therefore decided to scrape all the product listings contained within the "View all" subcategory instead of looping through each individual subcategory.

The same thing was applicable within the men's category and we decided to scrape the product listings contained within the "View all" subcategory in that category as well.

[25] "Buy Women's Second-Hand Clothing & Accessories - Oxfam GB." https://www.oxfam.org.uk/shop/womens-clothing. Accessed 12 Mar. 2020.

However, within the kids category, the subcategories are split into a boys section and a girls section, as we can see in Fig. 7.2. We decided to further split it into two categories, girls and boys and use the same strategy for both of them as we used for the women's section and the men's section, that is concentrate on the "View all" subcategory instead of scraping every subcategory individually.

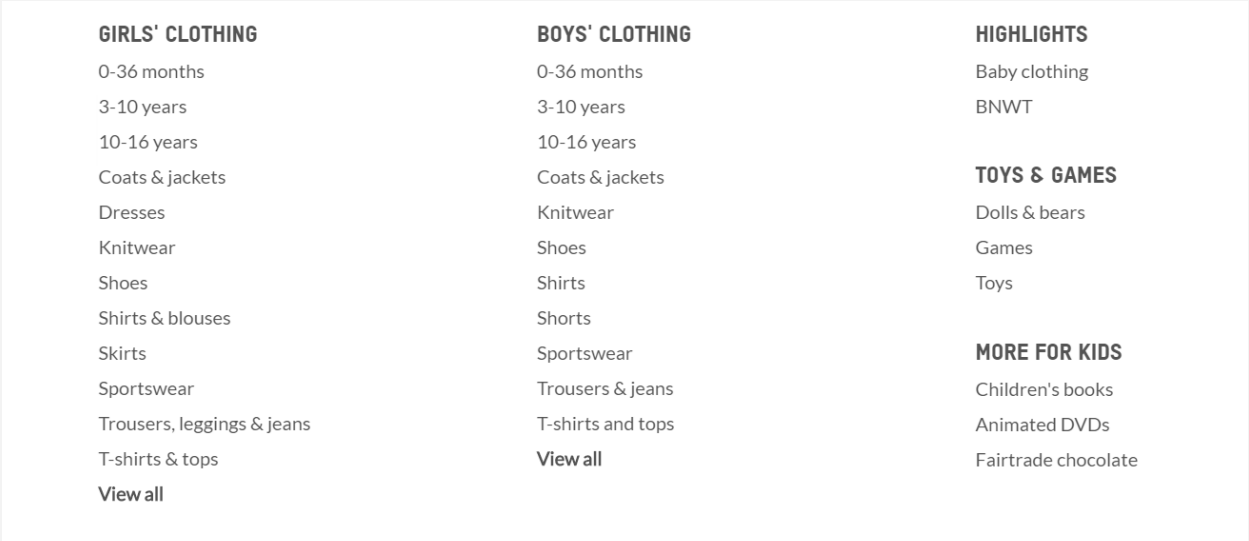


Figure 7.2: Oxfam.org.uk subcategories for kid's clothing[26]

Within the View all subcategory within the women's category, there are currently 31.986 items listed in a total of 267 pages as noted in Fig. 7.3.

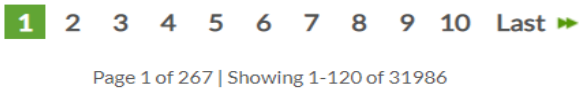


Figure 7.3: Oxfam.org.uk page navigation for all products in women's clothing

Instead of manually going through every page and listing it's url, we figured it was better to check the url on two different pages and check if we could utilize the urls to loop through every page to get the product urls.

[26] "Great Value & Second-Hand Kids' Clothes - Oxfam GB." https://www.oxfam.org.uk/shop/kids-clothing. Accessed 12 Mar. 2020.

Comparing Fig. 7.4 and Fig. 7.5, we can see that the only difference is that on page 2, the url has "...page=2.." in the url, while page 3, the url has "...page=3..." in the url.

```
oxfam.org.uk/shop/womens-clothing/all?i=1;m_sort_shops=FirstMadeLive;page=2;q=*;q1=Women%27s;show_all=products;x1=secondary_cat_1
```

Figure 7.4: Oxfam.org.uk url for page 2 for all products in women's clothing

```
oxfam.org.uk/shop/womens-clothing/all?i=1;m_sort_shops=FirstMadeLive;page=3;q=*;q1=Women%27s;show_all=products;x1=secondary_cat_1
```

Figure 7.5: Oxfam.org.uk url for page 3 for all products in women's clothing

With this information in mind, we decided to split the url in two parts, one part contains the url up to the page number, "oxfam.org.uk/shop/womens-clothing/all?i=1;m_sort_shops=FirstMadeLive:page=", and the other part, the url from the page number, ";q=*;q1=Women%27s;show_all=products;x1=secondary_cat_1". We can then loop from 1 to 267 to build our own list of urls, which saved us a significant time instead of doing this manually. The following strategy could also be used for the other categories we wanted to scrape.

When we have all the urls we needed to scrape all the product items, we need to find a solution to only scrape the urls that were actually a url to a product item. To do so, we used the inspect tool in Google Chrome. As we can see in Fig. 7.6, all the urls for a product listing were within a list with the class "product-result-item" assigned to them. This enabled us to only look at html code where a "li" item that contained the class that we wanted to extract the url from to exclude any url that isn't a product listing.

```
▼<div class="sm sm-product-search-results">
  ▼<ul>
    ▶<li class="product-result-item">…</li>
    ▶<li class="product-result-item">…</li>
    ▶<li class="product-result-item">…</li>
    ▶<li class="product-result-item result-row-end">…</li>
    ▶<li class="product-result-item">…</li>
    ▶<li class="product-result-item">…</li>
    ▶<li class="product-result-item">…</li>
    ▶<li class="product-result-item result-row-end">…</li>
    ▶<li class="product-result-item">…</li>
```

Figure 7.6: Subset of Oxfam.org.uk html code for all product listings within a page

By only finding urls that contain a product listing enables us to loop through all the pages of a category and extract all product listings within that category into a list.

The next step was about how we could extract only the information we wanted from the product page. As we can see in the Fig. 7.7, the product listing contains some information we are not interested in scraping.

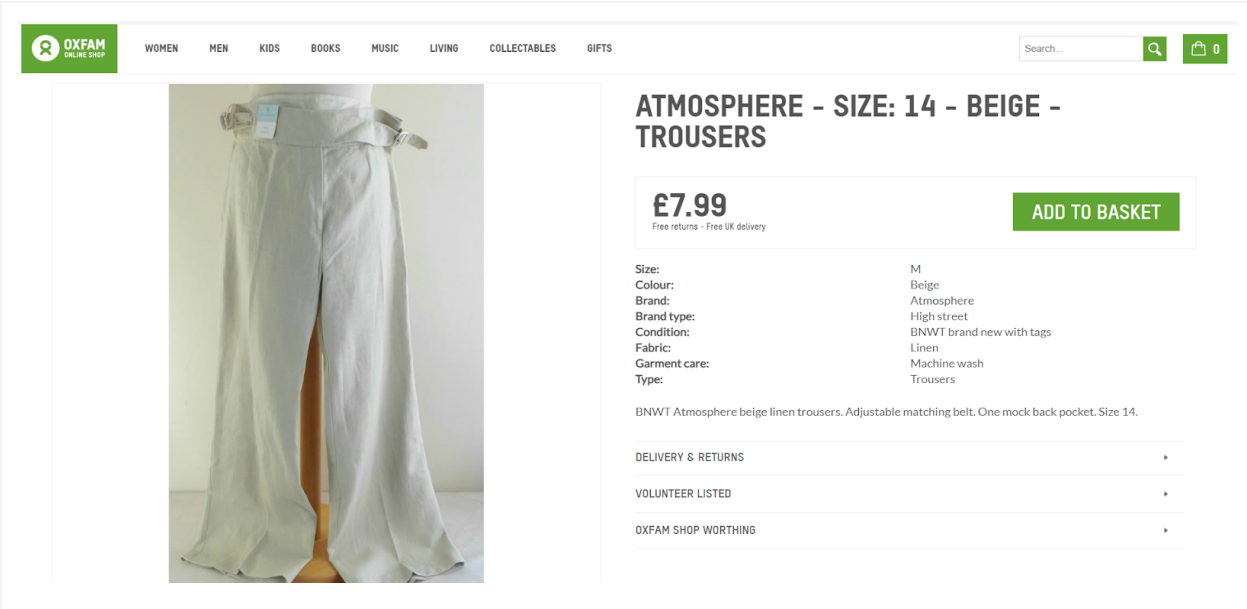


Figure 7.7: Oxfam.org.uk product listing

On this page we were only interested in getting a couple of the tags along with the product image url. We decided to use the inspect tool again to get a better understanding of the html of the page and check if we could pinpoint the class of each of the tags and pinpoint where the image url is located within the html.

Fig. 7.8 shows that all tags are stored within a description list in html with the class “product-attributes”. We will only be looking to extract information from this class.

Fig. 7.9 shows that the image url is contained within the class “sm-product-details-gallery-v2”, and we will be extracting the href from that class. We will also extract the image name, in this case HD_101888708.

```
▼<dl class="product-attributes clearfix">
    <dt>Size:</dt>
    <dd>M</dd>
    <dt>Colour:</dt>
    <dd>Beige</dd>
    <dt>Brand:</dt>
    <dd>Atmosphere</dd>
    <dt>Brand type:</dt>
    <dd>High street</dd>
    <dt>Condition:</dt>
    <dd>BNWT brand new with tags</dd>
    <dt>Fabric:</dt>
    <dd>Linen</dd>
    <dt>Garment care:</dt>
    <dd>Machine wash</dd>
    <dt>Type:</dt>
    <dd>Trousers</dd>
```

Figure 7.8: Oxfam.org.uk code for tags

```
▼<div class="sm sm-product-details-gallery-v2 clearfix image-gallery zoomable">
  ▼<div id="container_0_contentprimary_1_ProductDetailGalleryControl_MainImageContainer"
   class="main-image-container second-hand">
      <div class="roundel"></div>
    ▼<a id="container_0_contentprimary_1_ProductDetailGalleryControl_MainImageLink"
     class="main-image" href="//yatra8exe7uvportalprd.blob.core.windows.net/images/
     products/HighStDonated/Zoom/HD_101888708_01.jpg?v=1" rel="undefined" title style=
     "outline-style: none; text-decoration: none;" onclick="s_objectID="https://
     yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Zoom/
     HD_1018887_1";return this.s_oc?this.s_oc(e):true">
```

Figure 7.9: Oxfam.org.uk code for image url location

7.2 Finding a strategy to scrape Craigslist.org

Before writing the code for scraping the Craigslist.org website, we needed to do the same due diligence as we did before scraping Oxfam.org.uk. Since Craigslist.org is divided into cities, we decided to look at listings in the San Francisco Bay Area to see how we could scrape the information we wanted.

Craigslist.org offers a lot of categories for each city, “community”, “housing”, “jobs”, “services”, “for sale” and “discussion forums”. We will look into a subcategory within the “for sale” category called “clothes+acc”, which lists clothing items and accessories currently for sale by sellers located in the area.

As we can see in Fig. 7.10, Craigslist.org offers a few filters that we can use to filter out listings. We will use the filter “has image”, as we are only interested in listings that offer images. Other filters are of no use to us.

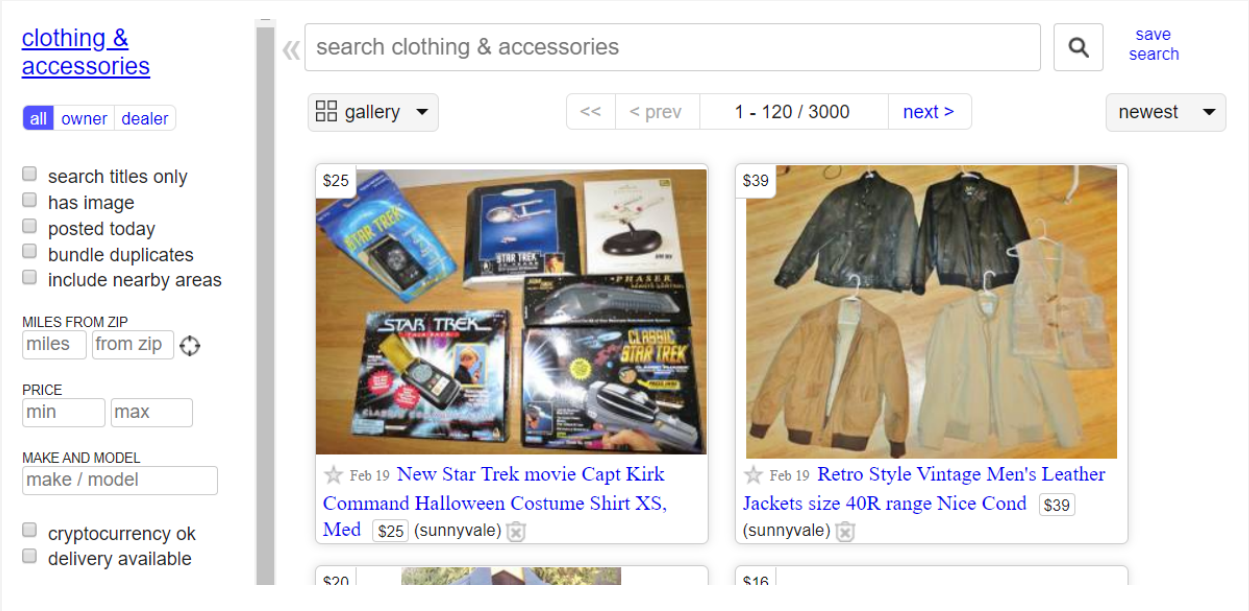


Figure 7.10: Craigslist.org Clothing & Accessories category

Next, we needed some type of a strategy to get all the product listings. Our strategy was the same as we used for scraping Oxfam.org.uk. Here we will look at the paging system to see if we can find a strategy to scrape product listings by looping through the pages.

As we can see in Fig. 7.11, the paging system differs a bit from Oxfam.org.uk. Here, Craigslist.org shows 120 items on each page, and up to 3.000 items, which means that either we must find a way to go beyond 3.000 items or look at other cities within Craigslist.org to build a sizable dataset.

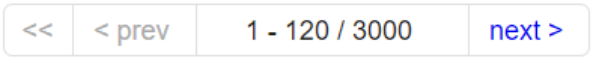


Figure 7.11: Craigslist.org pagination

Next, we needed a way to automatically go from products 1-120 to the next 120 products. We looked at the url to see if we could manipulate the url to loop through each iteration of 120 products.

sfbay.craigslist.org/search/cla?hasPic=1&s=0

Figure 7.12: First 120 listings on Craigslist.org

sfbay.craigslist.org/search/cla?s=120&hasPic=1

Figure 7.13: Next 120 listings on Craigslist.org, starting from listing 120

sfbay.craigslist.org/search/cla?hasPic=1&s=240

Figure 7.14: Next 120 listings on Craigslist.org, starting from listing 240

Looking at Fig. 7.12, 7.13 and 7.14, we can see how the urls change when looking at different sets of listings. The only thing that seems to change is the s=X, where X is the starting number of listings and shows the 119 listings after the starting number along with that listing. We can iterate through all the pages by increasing the s=X, by 120 for each iteration.

Once we have all the listings, we will need a way to scrape all the product urls from each page. Here we will use the same trick as we used in scraping the product urls for each page for Oxfam.org.uk, that is, we will look at the html and check if we can pinpoint where the url can be scraped.

In Fig. 7.15, we can see that all listings on a page are contained within a list, where each list item has the class name “result-row”. In Fig. 7.16, we can see that each item within the list has a href that lists the url where a specific listing can be found. We will use this to scrape only the product urls from each page.

Figure 7.15: Html for all listings on Craigslist.org

Figure 7.16: Html for each listing on Craigslist.org

Once, we have all the urls for all the products we want to scrape, we need to find a strategy to only extract the information we want. We will take four sample product listings and check how the website lists the tags and how we can extract the information we want to extract.

As we can see in Figs. 7.17 to 7.20, the listings differ a lot. The only thing that is consistent through all listings is the image. Some listings have all the information in either the Title or the Description field.

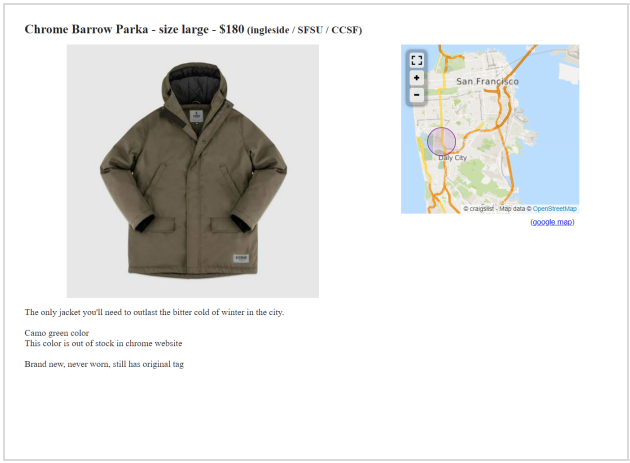


Figure 7.17: Craigslist.org product listing #1

Figure 7.18: Craigslist.org product listing #2

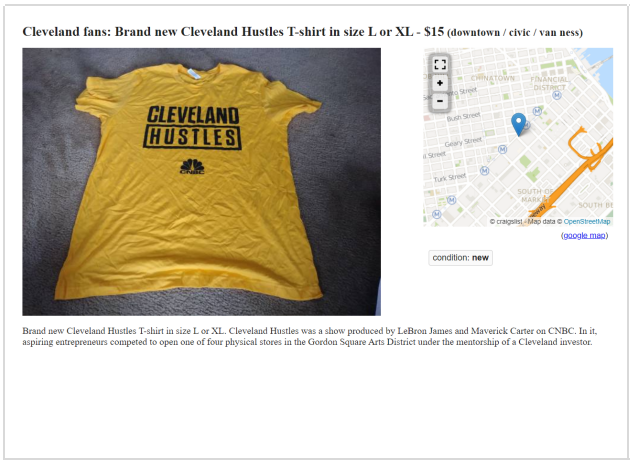


Figure 7.19: Craigslist.org product listing #3

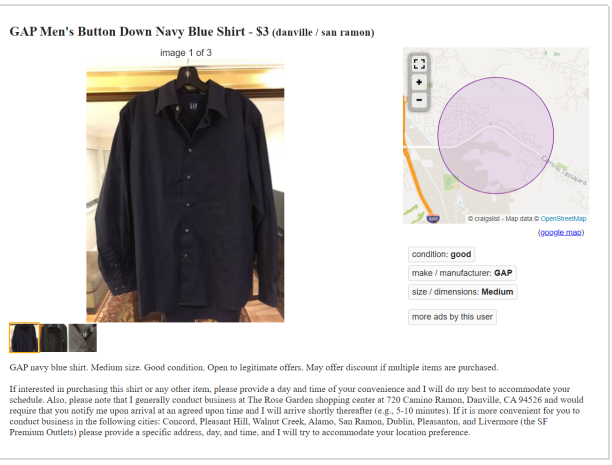


Figure 7.20: Craigslist.org product listing #4

Since this data will be extremely difficult to clean and get consistency for all the listings, we decided against scraping Craigslist.org. However, if needed, Craigslist.org can be scraped with human tagging, that is, pay someone to manually tag images according to title and/or description. This can be done using a cheap workforce from e.g. Freelancer.com.

7.3 Finding a strategy to scrape Netflea.com

Our strategy for scraping Netflea.com is using the same strategy as we used for scraping Oxfam.org.uk. The sites are very much alike, except for minor differences in how each product listing is listed.

Netflea.com offers a lot of categories, and as we can see in Fig. 7.21, there are a lot of categories that we want to ignore. We are only interested in “Children’s clothing”, “Women’s clothing” and “Men’s clothing”.

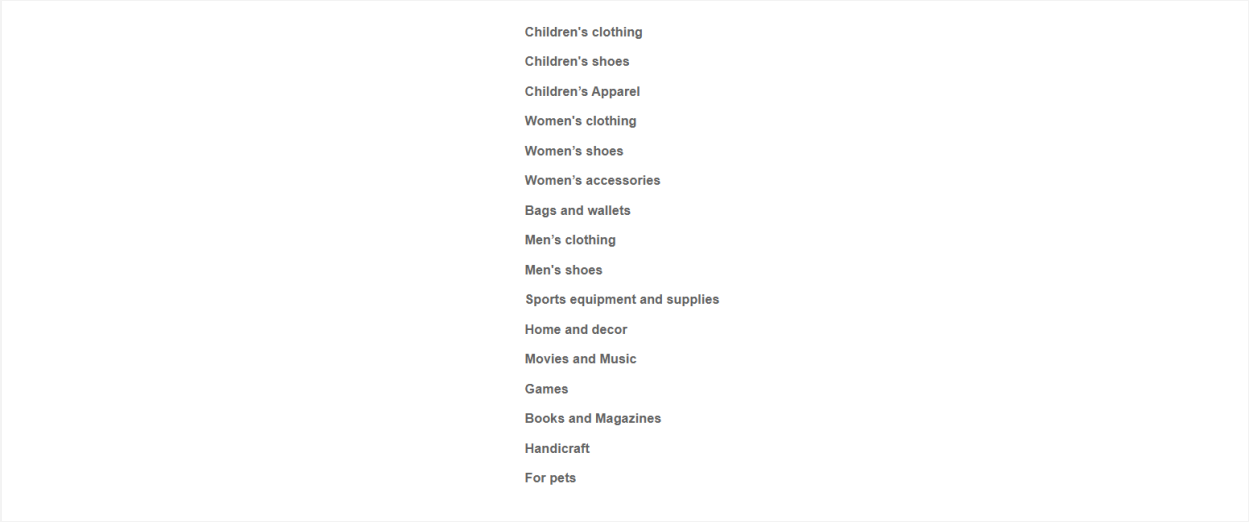


Figure 7.21: Product categories within Netflea.com

Next we want to find a way to loop through all pages of the product listings within each of the three categories. We will use the same strategy used when looping through Oxfam.org.uk, that is, look at the pagination at the website and find a way to manipulate the url to loop through all the pages.

As we can see in Fig. 7.22, the pagination of the Children’s clothing category contains 38.405 total listings. In total there should be 385 pages (38.405/100) with 100 product listings for each page except for the last one, which should contain 5 product listings..

Figure 7.22: Netflea.com pagination

As we can see in Fig. 7.23 and Fig. 7.24, the url lists the number of the page at the end. In this case we would only change the number after &p= to reflect on the current iteration through all the pages. The “limit=100” is done so that we will get maximum listings, in this case 100, for each page.

netflea.com/childrens-clothing.html?limit=100&p=1

Figure 7.23:Url for page 1 at Netflea.com

netflea.com/childrens-clothing.html?limit=100&p=2

Figure 7.24: Url for page 2 at Netflea.com

The next step in the process would be to extract the product listing url. The html for each page of product listings contain a lot of hrefs, so we will need to find a way to pinpoint which urls we will extract from each html page. To do this we will find the hrefs in the html and find out the class for the listings.

We can see the html for a particular product listing within the html in Fig. 7.25. We can see that the product listings are within a list with the class names “products-grid”, “first”, “last” and “odd”.
We can also see that each listing is a list item within the list, having the class names “item”, “last”, “col-xs-12” and “col-sm-3” and within each list item there is a href that we will be extracting.

```
▼<ul class="products-grid first last
odd">
    <!--        <ul class="products-grid
    row">-->
  ▼<li class="item last col-xs-12 col-sm-
  3" style="overflow: inherit;">
    ▼<a href="https://www.netflea.com/
    baby-warm-unipussi-other-brand-size-
    other-size.html" title="Baby warm
    unipussi - OTHER BRAND - Size OTHER
    SIZE" class="product-image" data-id=
    "643-731" data-name="Baby warm
    unipussi - OTHER BRAND - Size OTHER
    SIZE" data-price="10.9" data-category=
    "Children's clothing" data-brand=
    "Other Brand" data-quantity="1" data-
    click data-position="1" data-event=
```

Figure 7.25: Using inspect tool in Chrome to view html for product listings within each page

We can see how the product listing is shown in Fig. 7.26. Here, we will need to pinpoint the information that is located on the right that contains the Type, Brand and Colour, and the url to the image located on the left.

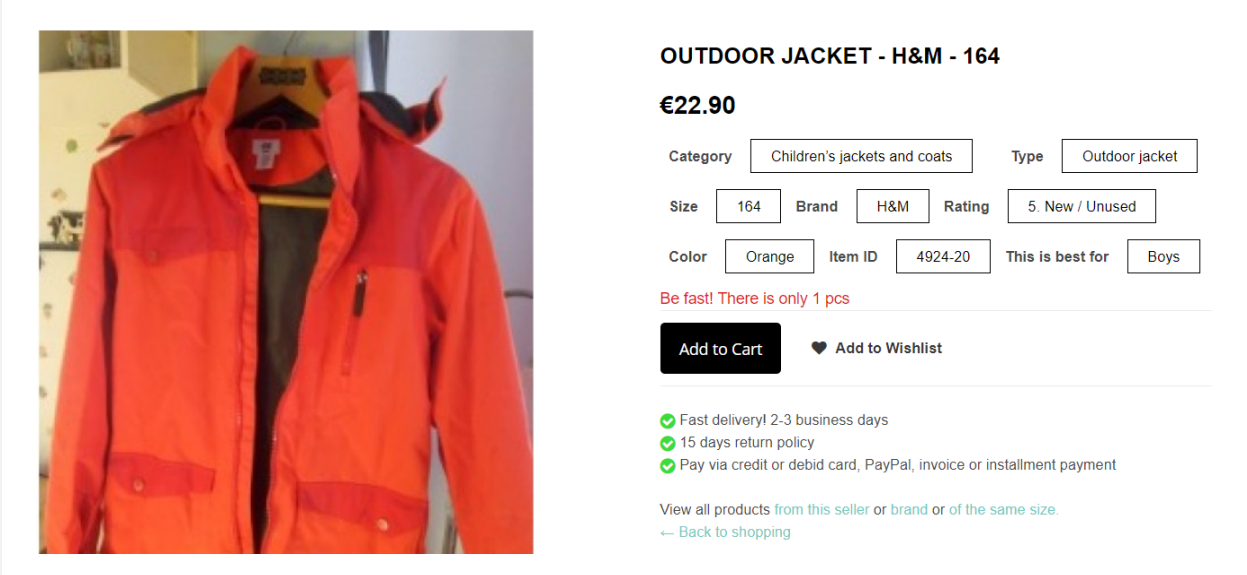


Figure 7.26: Using inspect tool in Chrome to view html for product listings within each page

Using the inspector tool in Chrome, we can see that the information that we will be focusing on is wrapped in a with the class “product-attributes”, as we can see in Fig. 7.27. Each attribute has the class “attribute-wrapper” assigned to it. Taking a closer look, we can see in Fig. 7.28, that each attribute has two classes, the class “label” tells us the attribute category and the class “outeratc” tells us the attribute value.

The image url is contained in a different section of the html. As we can see in Fig. 7.29, the image href is contained within a class called “product-image”. We will use this to our advantage when we are extracting the image url. Along with the url, we will also be extracting the image name, so that we can link each row of information to a picture, which is the name after the last backslash in the url.

```
▼<div class="product-
attributes">
  ▶<span class="attribute-
  wrapper">…</span> == $0
  ▶<span class="attribute-
  wrapper">…</span>
  ▶<span class="attribute-
  wrapper">…</span>
  ▶<span class="attribute-
  wrapper">…</span>
  ▶<span class="attribute-
  wrapper">…</span>
  ▶<span class="attribute-
  wrapper">…</span>
  ▶<span class="attribute-
  wrapper">…</span>
  ▶<span class="attribute-
  wrapper">…</span>
  ▶<span class="attribute-
  wrapper">…</span>
```

Figure 7.27: Html for attributes #1

```
▼<span class="attribute-
wrapper">
  <div class="label"></div>
  ▼<span class="attribute-
  wrapper"> == $0
    <div class="label">
    Category</div>
    ▼<span class="outeratc">
      <span class="inneratc2">
      Children's jackets and
      coats</span>
    </span>
  </span>
</span>
```

Figure 7.28: Html for attributes #2

```
▼<p class="product-image">
  ▼<div id="wrap" style="top:0px;
  z-index:auto;position:relative;">
    ▶<a href="https://
    www.netflea.com/media/catalog/
    product/cache/2/image/750x7…
    6e5fb8d27136e95/B/E/BE6E0938-
    9D2D-4B0C-BD00-
    2A66C355FF711569051273.jpeg"
```

Figure 7.29: Html for image location

7.4 Finding a strategy to scrape Swap.com

The strategy for scraping Swap.com will be the same we used for scraping Oxfam.org.uk and Netflea.com, except when we looked at the website, we noticed that the product listings were not contained within a page that could be accessed through an url, instead it was loaded as an endless scroll feature. We needed to take that into account when we were analyzing the page and finding a way to loop through the product listing pages.

We will use the same steps for analyzing Netflea.com as we did with the other sites, but we will make sure that we have a way to loop through the pages once we are at that section, before analyzing further.

Swap.com offers a couple of categories but we will focus on three of them, Women, Men and Kids. The Kids category is then further divided into two subcategories, Girls' Apparel and Boys' Apparel.

As we can see Figs. 7.31 to 7.33, each category offers a link to view all the products in each category. This is the same approach that we will take with Oxfam.org.uk and saves us a bit of time with not needing to build an url to loop through each subcategory.

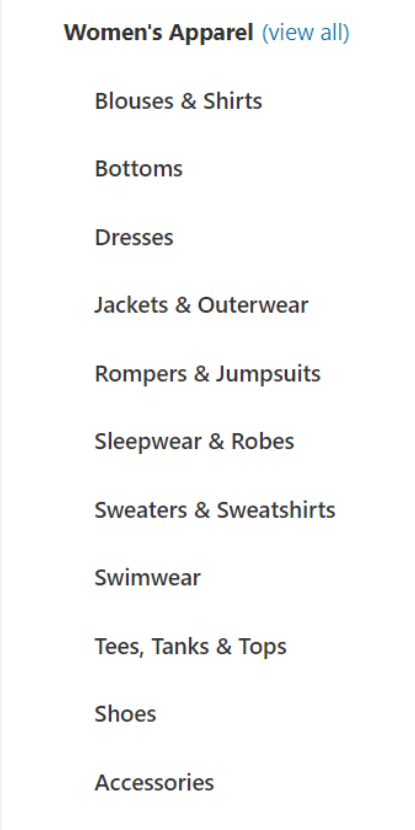


Figure 7.30: Women's categories

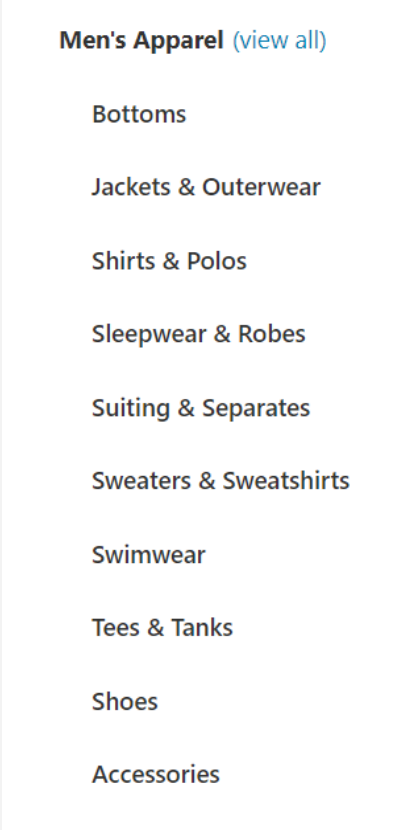


Figure 7.31: Men's categories

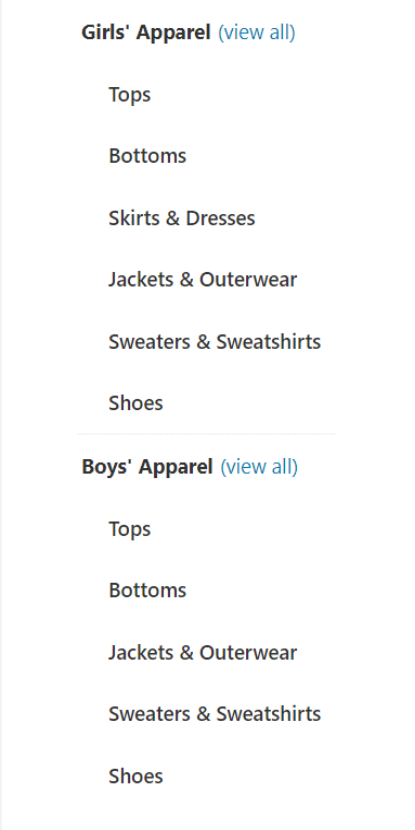


Figure 7.32: Kid's categories

Each page with product listings within a subcategory differs from the previous websites we looked at. Here the product listings appear as an "endless scrolling" and the urls don't show us anything useful that we could use to build a code to loop through, as we can see in Fig. 7.33 and Fig. 7.34, which shows the url at when we are at the top of the page and the url when we scroll down.

swap.com/shop/womens-apparel/blouses-shirts/

Figure 7.33: Url for "Blouses" within women's category

swap.com/shop/womens-apparel/blouses-shirts/?cursor=CgRVeWpligEuCioKE1JSVV83MzIwMjI0NDA4MDM2NTASE1JSU180ODIzMDI3NDM2MDExMTUgHg

Figure 7.34: Url for "Blouses" within women's category when scrolling down

We need to find a way where we can loop through all the product listings to extract the product urls when we are building the code to scrape the website.

Once we have figured out a way to extract all the product urls, we will need to look at how we can extract information from a given product page.

As we can see in Fig. 7.35, the product listing contains all the information that we want. The attributes are all contained within a specified section on the website, so it could be easy to extract the information.

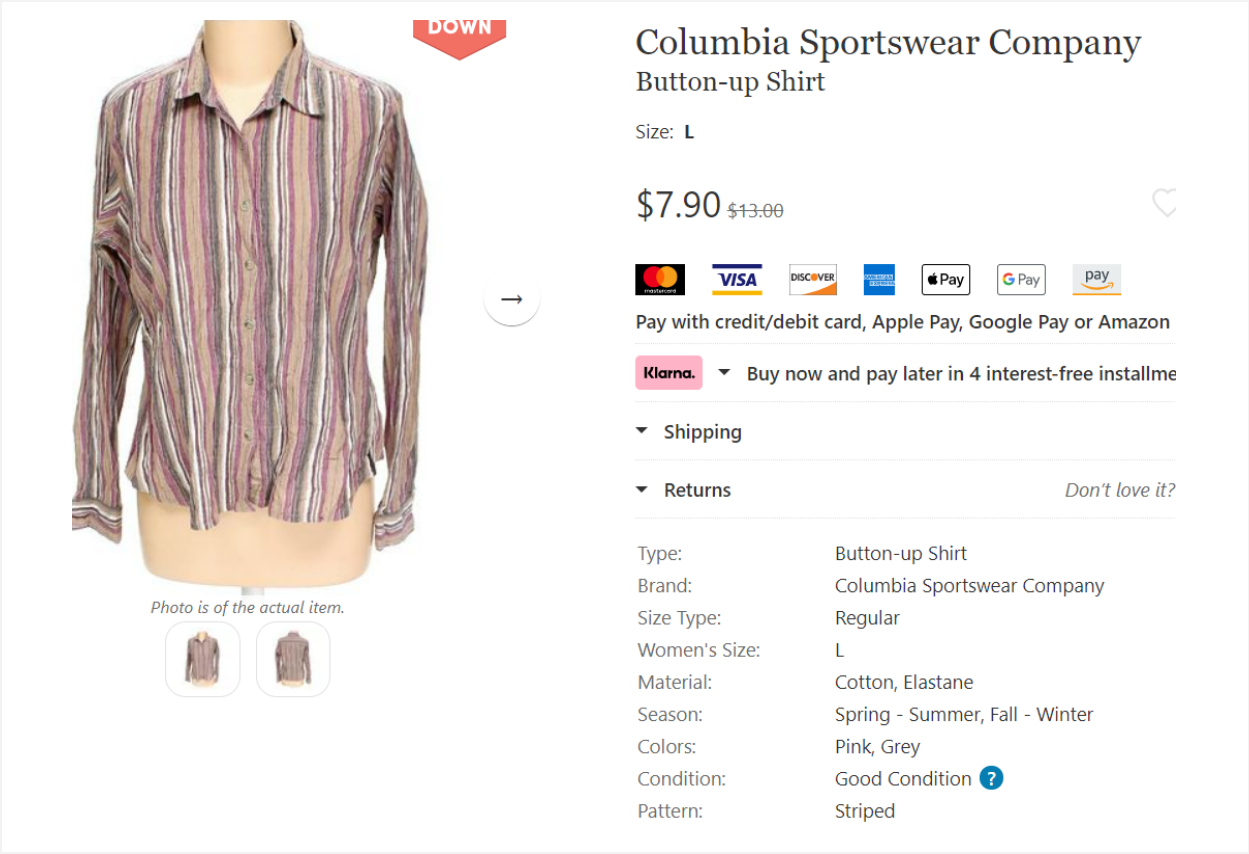


Figure 7.35: Swap.com product listing

As we can see in Fig. 7.36, the attributes are contained within a table using the class "ItemInfoTable_table_2cNr" and the attribute name and the value are contained within a table row. The attribute name has the class name "TableCell_property_16NeX" and the attribute value has the class name "TableCell_value_2Ifn0". Both are a <td> property.

As we can see in Fig. 7.37, the image is contained within a using the class name "ItemImage_image_Qi6dL". However, upon further inspection, we noticed another url contained within the class name "ItemImage_thumbnails_2lHP2", as we can see in Fig. 7.38, that had two hrefs, either listing the location of the front side of the image or the back side. Upon reviewing both images, we noticed that the images contained within the thumbnail class were of higher quality than the ones contained in the image class. We decided to scrape both urls, but with the intention of downloading the images contained in the thumbnails class. We will also be scraping the image name, which is the name in the url after the last backsplash. This is done so that we can link the images to a row in the dataset.

Figure 7.36: html for attributes on product listing page

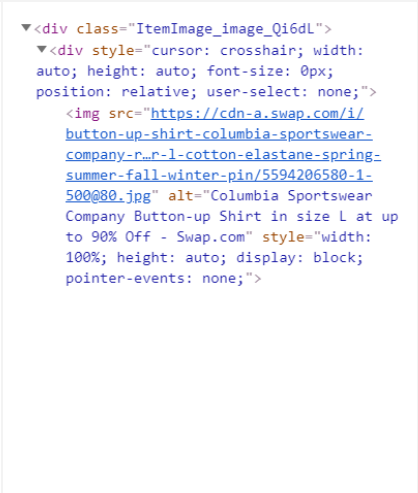


Figure 7.37: Image location for product listing

Figure 7.38: Thumbnail location for product listing

Upon further investigation into how we could scrape the site, we noticed that the site is built using React, which complicates the scraping process significantly. We therefore decided against scraping Swap.

Scraping the site would require the use of an automated browser, such as Selenium, and scrolling to capture product listings. We will have this option as a backup if Oxfam.org.uk and Netflea.com don't produce good enough models with good enough predictions.

7.5 Finding a strategy to build a backup dataset

To build the backup dataset, we don't need to figure out a way on how we would scrape the websites as we will be hand picking the images that we are going to download. Our strategy here is to find a way to get the most high quality image available.

To build our backup dataset, we will use four websites to gather the images that we want. These websites are Grailed.com, Shpock.com, Picclick.com and Poshmark.com.

Our process will be the same for all websites, that is to go to a category page, e.g. Adidas T-Shirts, copy the image address for an image we want to include in our dataset and download the image. If the image is of low quality, e.g. a thumbnail picture, we will try to figure out a way to find the higher quality image by looking at the address of the lower quality image and the higher quality image and see if we can spot anything that enables us to mass edit all the links to get the higher quality image. Going into each product listing to copy the address to the higher quality image would be too time consuming since we are manually building the dataset instead of scraping the data.

Once we have all the links that we want, we will use the wget tool in Linux to download all the images.

7.5.1 Strategy for Grailed.com

Our strategy for finding images from Grailed.com is to go to each category of the clothing we want to accumulate pictures of, e.g. Adidas T-shirts. We will then pick the pictures that closely resemble the pictures that our users will take and have the colour that we wanted.

As we can see in Fig. 7.39, a sample of the product listings of Adidas T-shirts yields one picture that we might be interested in having in our dataset. Picture #4 is a black Adidas T-shirt that closely resembles the picture our users will take.

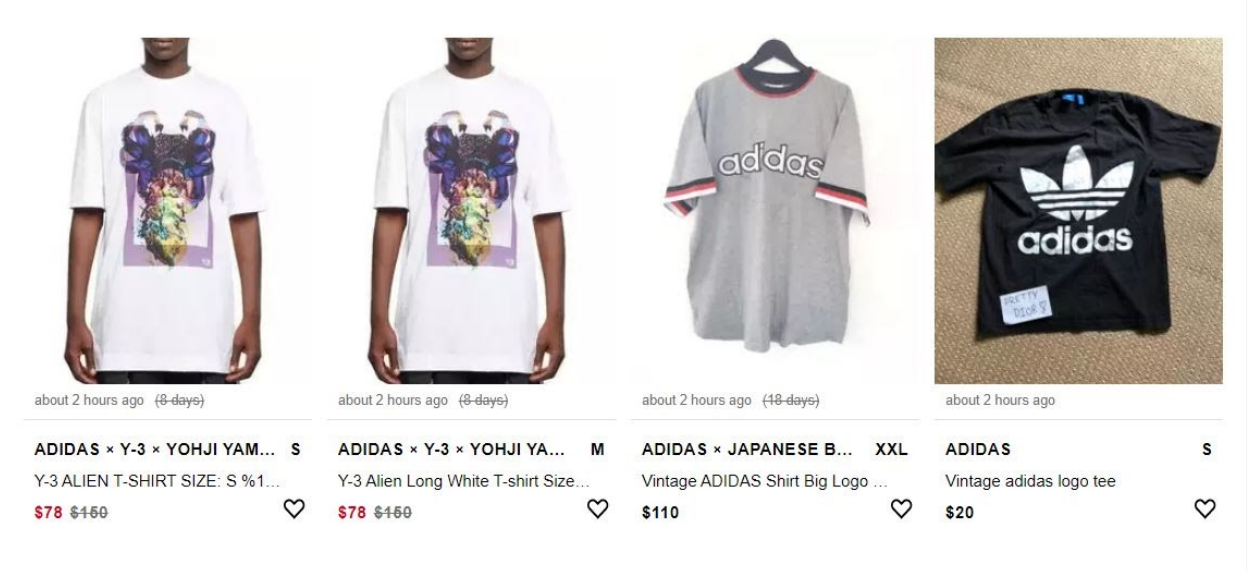


Figure 7.39: Adidas T-shirts at Grailed.com

Using Google Chrome we can right click on the image and select “copy image address” to get the address to the image. Doing so, yields us this link:

https://process.fs.grailed.com/AJdAgnqCST4iPtnUxiGtTz/auto_image/cache=expiry:max/rotate=deg:exif/resize=height:320,width:240,fit:crop/output=quality:70/compress/https://cdn.fs.grailed.com/api/file/pGKbazMTDC6hgi7SioVM

Looking at the url, we can see that the url is compressed and resized into a lower dimension image. We can therefore cut the process part of the url which gives us this url to the uncompressed image:

https://cdn.fs.grailed.com/api/file/pGKbazMTDC6hgi7SioVM

As we can see in Table 7.1, this will give us an image which is ~750KB vs. the ~16KB image which we got by copying the image address. This yields an image which is almost 47 times the size of the compressed image.

Image Properties	Compressed Image	Uncompressed Image
Image Dimensions	240 x 320	1200 x 1600
Image Size (in bytes)	16.256	749.725

Table 7.1: Comparison of images at Grailed.com

Our strategy is therefore going through each category, copying the image url of the images we want in our dataset to an Excel document. These urls will yield us the compressed image url.

Once we have all the image urls that we want in our Excel document, we will split the urls by going into the "Text to Columns" function within Excel and split it by "/". After removing the columns that we don't want, we are left with 5 columns where each row has "https:", "cdn.fs.grailed.com", "api", "file" and the image name, which in our example is "pGKbazMTDC6hgi7SioVM". Combining these cells into a single cell using "&" with an extra cell that contains a "/" will yield us with the url to the uncompressed image:

https://cdn.fs.grailed.com/api/file/pGKbazMTDC6hgi7SioVM

Once we have done this to all the rows containing the urls, we will export the list into a .txt document and run wget for the .txt document to download all the images.

7.5.2 Strategy for Shpock.com

Our strategy for Shpock.com is to look for a category of a clothing item that we want to include in our dataset and try to find the most high quality image available.

As we can see in Fig. 7.40, a sample of the product listings of Adidas T-shirts yields three pictures that we might be interested in having in our dataset.

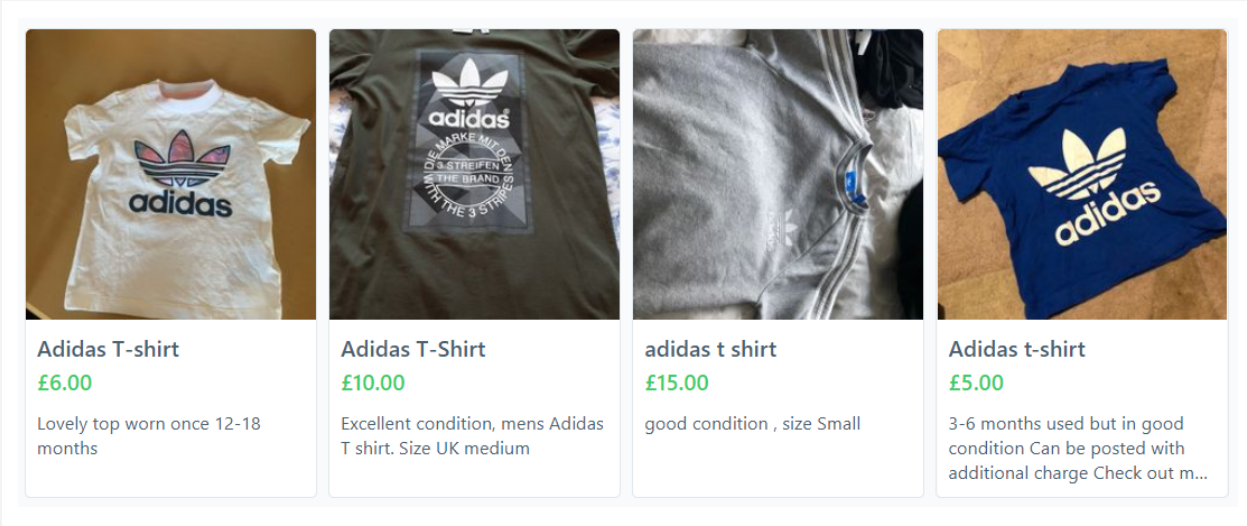


Figure 7:40: Adidas T-shirts at Shpock.com

Using Google Chrome we can right click on the image and select “copy image address” to get the address to the image. Doing so, yields us this link:

https://webimg.secondhandapp.com/w-i-m/5e974cb79015f91abfcce828

We can’t see anything that could yield us a higher quality image by looking at the url, so we will try to find the higher quality image url by going into the listing and finding the url for the image within the listing.

As we can see in Fig. 7.41, by going to the product listing, we can see the full image as compared to some sort of a thumbnail image as we did in Fig. 7.40.

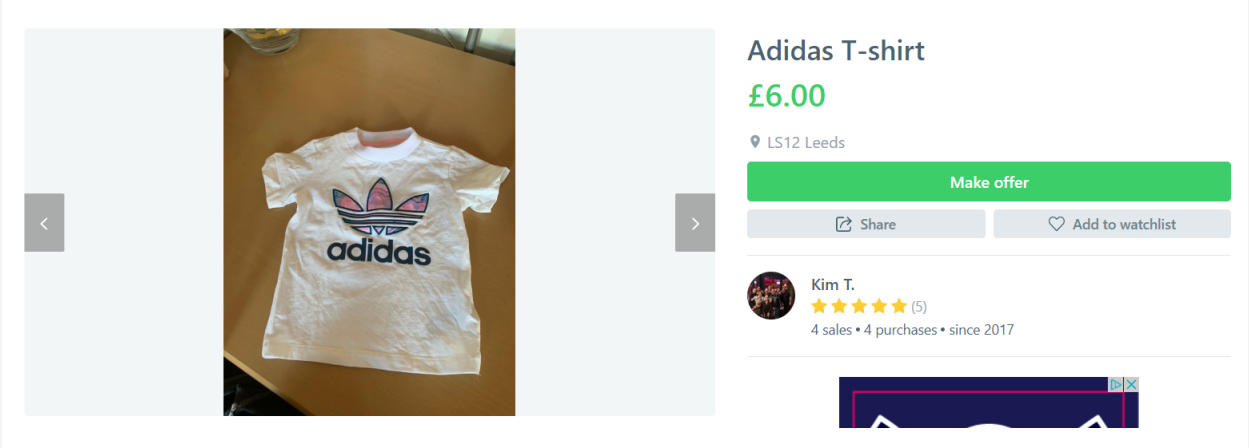


Figure 7.41: Product listing for particular clothing at Shpock.com

Using Google Chrome again to right click on the image to get the image address, we get the following link:

https://webimg.secondhandapp.com/w-i-mgl/5e974cb79015f91abfcce828

By comparing the two urls, we can see that the only difference here is that the first url was "../w-i-m/.." and the second one was "../w-i-mgl/..". We further looked at several other images and noticed the same trend.

As we can see in Table 7.2, this will give us an image which is ~153KB vs. the ~10KB image which we got by copying the image address on the category listing. This yields an image which is almost 16 times the size of the lower quality image.

Image Properties	Lower Quality Image	Higher Quality Image
Image Dimensions	237 x 237	1200 x 1600
Image Size (in bytes)	9.786	153.172

Table 7.2: Comparison of images at Shpock.com

Our strategy is therefore going through each category, copying the image url of the images we want in our dataset to a .txt document. These urls will yield us the compressed image url.

Once we have all the image urls to the lower quality image we will replace the text where "w-i-m" is with "w-i-mgl" which will change all the links to the higher quality image for each image we found.

We will then run wget to download all the images that are in our .txt document.

7.5.3 Strategy for Picclick.com

Our strategy for Shpock.com is to look for a category of a clothing item that we want to include in our dataset and try to find the most high quality image available.

As we can see in Fig. 7.42, a sample of the product listings of Adidas T-shirts yields a lot of potential pictures we might be interested in adding to our dataset, e.g. image #3.

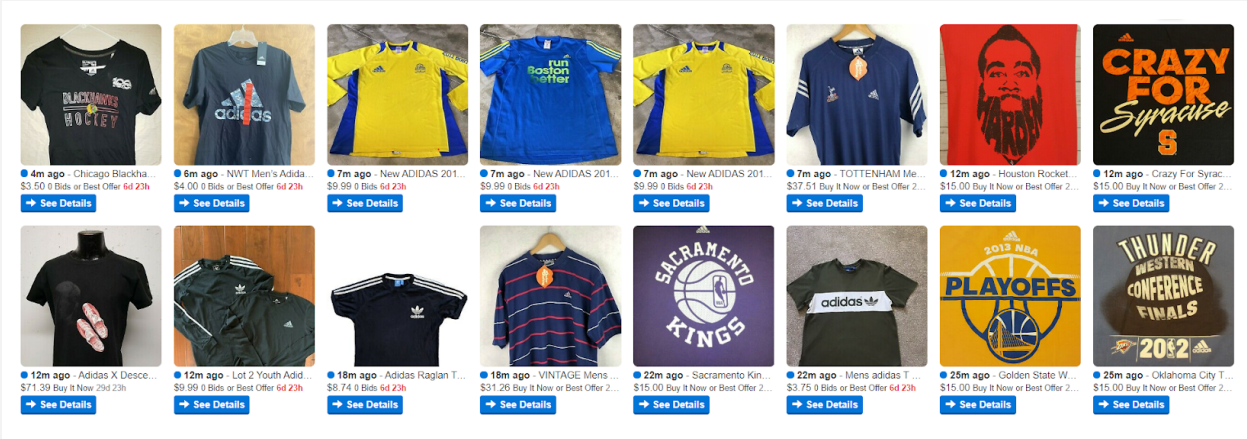


Figure 7:42: Adidas T-shirts at Picclick.com

Using Google Chrome we can right click on the image and select “copy image address” to get the address to the image. Doing so, yields us this link:

https://www.picclickimg.com/d/l400/pict/153904024955_/New-ADIDAS-2011-BOSTON-MARATHON-Long-Sleeve-T-Shirt.jpg

We can’t see anything that could yield us a higher quality image by looking at the url, so we will try to find the higher quality image url by going into the listing and finding the url for the image within the listing.

As we can see in Fig. 7.43, by going to the product listing, we can choose between five images and one of these images is the same image as we saw in the category view.

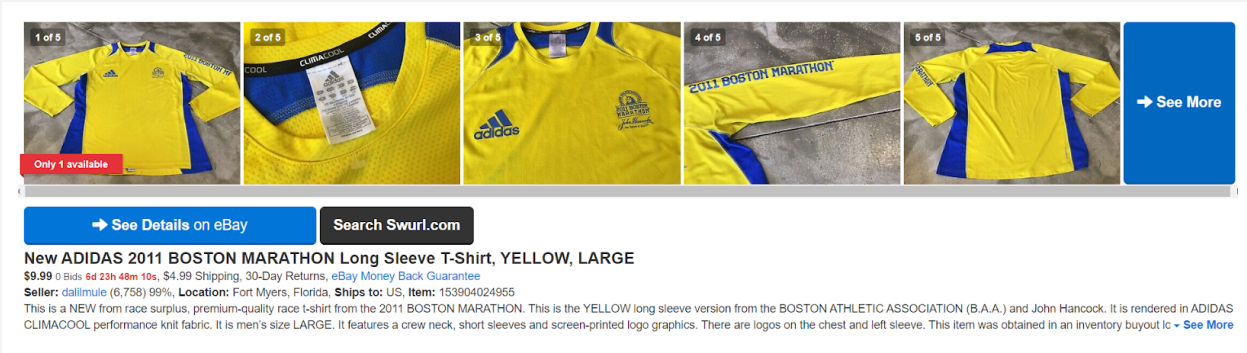


Figure 7:43: Product listing for particular clothing at Picclick.com

By clicking on the image and using Google Chrome again to right click on the image to get the image address, we get the following link:

https://www.picclickimg.com/d/w1600/pict/153904024955_/New-ADIDAS-2011-BOSTON-MARATHON-Long-Sleeve-T-Shirt.jpg

By comparing the two urls, we can see that the only difference here is that the first url was "../l400/.." and the second one was "../w1600/..". We further looked at several other images and noticed the same trend.

As we can see in Table 7.3, this will give us an image which is ~316KB vs. the ~25KB image which we got by copying the image address on the category listing. This yields an image which is almost 13 times the size of the lower quality image.

Image Properties	Lower Quality Image	Higher Quality Image
Image Dimensions	400 x 300	1600 x 1200
Image Size (in bytes)	24.937	316.344

Table 7.3: Comparison of images at Picclick.com

Our strategy is therefore going through each category, copying the image url of the images we want in our dataset to a .txt document. These urls will yield us the lower quality image url.

Once we have all the image urls to the lower quality image we will replace the text where "l400" is with "w1600" which will change all the links to the higher quality image for each image we found.

We will then run wget to download all the images that are in our .txt document.

7.5.4 Strategy for Poshmark.com

Our strategy for Poshmark.com is to look for a category of a clothing item that we want to include in our dataset and try to find the most high quality image available.

As we can see in Fig. 7.42, a sample of the product listings of Adidas T-shirts yields two potential pictures we might be interested in adding to our dataset, i.e. image #1 and image #2.

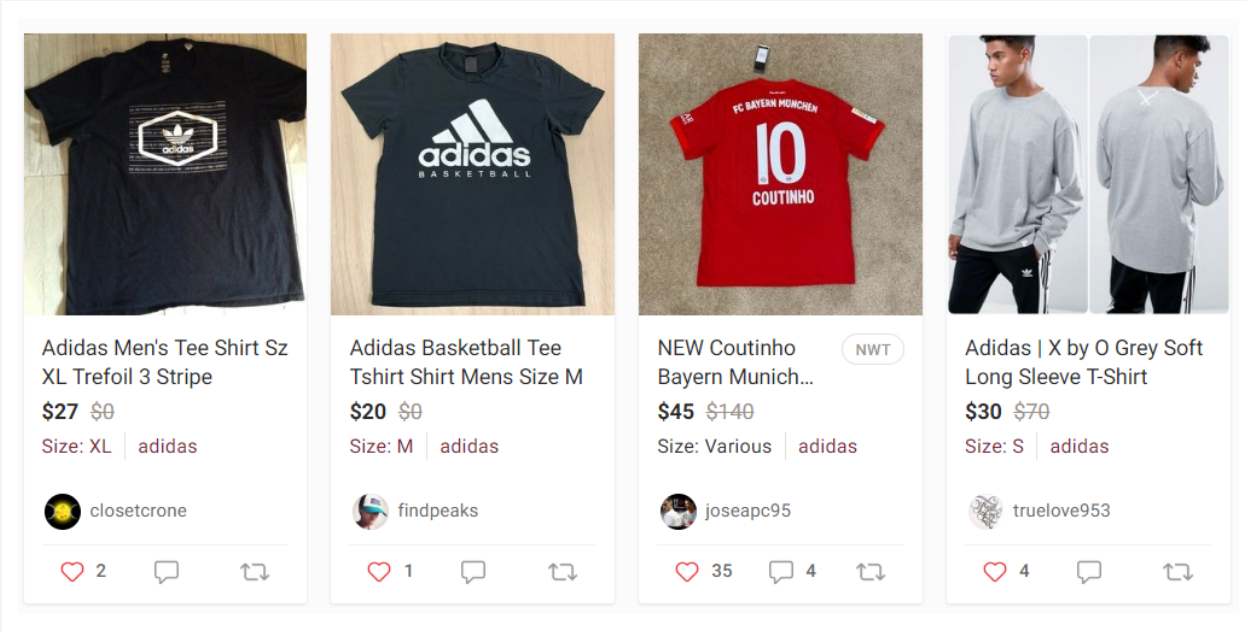


Figure 7:44: Adidas T-shirts at Poshmark.com

Using Google Chrome we can right click on the image and select "copy image address" to get the address to the image. Doing so, yields us this link:

https://di2ponv0v5otw.cloudfront.net/posts/2019/05/19/5ce1906b7bc3607289897700/s_5ce1906f264a55b2b1a7ea09.jpeg

We can't see anything that could yield us a higher quality image by looking at the url, so we will try to find the higher quality image url by going into the listing and finding the url for the image within the listing.

As we can see in Fig. 7.44, by going to the product listing, we can choose between four images and one of these images is the same image as we saw in the category view.

Figure 7:45: Adidas T-shirts at Poshmark.com

By clicking on the image and using Google Chrome again to right click on the image to get the image address, we get the following link:

https://di2ponv0v5otw.cloudfront.net/posts/2019/05/19/5ce1906b7bc3607289897700/m_5ce1906f264a55b2b1a7ea09.jpeg

By comparing the two urls, we can see that the only difference here is that the first url was "../s_5ce1906.." and the second one was "../m_5ce1906..". We further looked at several other images and noticed the same trend. By looking at the difference we can also see that there is an "s" and an "m", this resulted in us looking at if it was maybe a "l" for large, that is if "s" stands for small and "m" stands for "medium". This yielded a higher quality image than the "m" image.

As we can see in Table 7.4, this will give us an image which is ~457KB vs. the ~158KB and ~14KB images which we got by copying the image address on the category listing and the product listing. This yields an image which is almost 33 times the size of the "s" image and almost 3 times the size of the "m" image.

Image Properties	"s" Image	"m" Image	"l" image
Image Dimensions	300 x 300	580 x 580	1000 x 1000
Image Size (in bytes)	14.285	158.297	457.047

Table 7.4: Comparison of images at Poshmark.com

Our strategy is therefore going through each category, copying the image url of the images we want in our dataset to a .txt document. These urls will yield us the highest quality image url.

Once we have all the image urls to the lower quality image we will replace the text where "s_" is with "l_" which will change all the links to the higher quality image for each image we found and then we will run wget to download all the images that are in our .txt document, like we did with the previous four websites.

8 Scraping code

When deciding on the programming language to build the scraping tool, we decided on using the Python programming language. All team members had some experience with the Python language and one member of the team had successfully built a scraping tool using the Python language.

To build the code using the Python language, we needed to use a couple of modules, these were “requests”, “BeautifulSoup”, “csv”, “pathlib” and “os”. “requests” and “BeautifulSoup” can be installed through pip tools and the other modules are most often included in the Python setup. We can see in Table 8.1 how we are planning on using each module.

Module	How we will use it
requests	Send a request to an url and get back a response containing the html of the url
BeautifulSoup	View the html, search the html and extract information from the html
csv	Open the .csv file containing the dataset and add new rows to the .csv file
pathlib	Create an images folder in the system running the code to store the images
os	Send the wget command to the system running the code to download the images

Table 8.1: Listing the modules we are going to use and how we are going to use them

We had already built a skeleton code to start scraping when we were making a strategy to scrape the Oxfam.org.uk site. We built the scraping tool for Oxfam.org.uk using that skeleton, and we used the same skeleton to build a scraping tool for the Netflea.com site.

Lessons learnt from the Oxfam.org.uk scraping were applied to the Netflea.com code. Those were that instead of storing the product urls and image urls in .txt document, we went through the process of getting everything we needed in one go, that is, we downloaded the image and added the tags along with the image name as a row in the dataset .csv document. Another lesson we learnt from Oxfam.org.uk, was that we needed to have if statements in case something went wrong. If that happened, we wanted to write empty values as a row in the dataset and try again for the next picture.

8.1 Oxfam.org.uk scraping code

We used the skeleton we built previously, when we were figuring out a strategy to scrape Oxfam.org.uk, as a starting point to create the code to scrape the website.

First we made variables for the class names that we would be looking to extract information from, the variables are hardcoded, as can be seen in Listing 8.1.

```
pagesClass = "product-result-item"
attributesClass = "product-attributes clearfix"
imagesClass = "sm sm-product-details-gallery-v2 clearfix image-gallery zoomable"
```

Listing 8.1: Hardcoded class names that we will be focusing on when scraping information

Then we made variables that we needed to build the url to loop through, these are also hardcoded, as can be seen in Listing 8.2. We also wanted to save the extracted urls, in case the program exited unexpectedly, so we wouldn't have to scrape everything again. The same code reflects on each of the four categories, changing only to reflect on each category.

```
womenFirstPartLink = \
"https://www.oxfam.org.uk/shop/womens-clothing/all?i=1;m_sort_shops=FirstMadeLive;page="
womenSecondPartLink = ";q=*;q1=Women%27s;show_all=products;x1=secondary_cat_1"
womenNumberOfPages = 268
womenPagesName = "women_pages"
```

Listing 8.2: Hardcoded information that will be used when scraping

Once we had all the information we wanted to continue, we made the same code for all categories, where only the variables would be sent reflecting on the category.

As we can see in Listing 8.3, the first thing we do, when we go through each category, is to create a .txt file to save every product listing url, as we can see in Listing 8.4. Next we go into the getPageLinks function which will get all the urls so that we can finally write the urls to our .txt file, which we can see in Listing 8.5, we created earlier.

```
helper_functions.createTXT(womenPagesName)

womenList = scrape_pages.getPageLinks(womenFirstPartLink, \
  womenSecondPartLink, womenNumberOfPages, pagesClass)

helper_functions.writingListToTXT(womenPagesName, womenList)
```

Listing 8.3: Creating a text document, scraping pages and extracting urls to text document

```python
def createTXT(fileName):
    txtName = fileName + ".txt"
    txtPath = Path(txtName)
    txtPath.touch(exist_ok=True)
```

Listing 8.4: Function to create a text document

```python
def writingListToTXT(fileName, lst):
    txtName = fileName + ".txt"
    with open(txtName, "a") as f:
        for l in lst:
            f.write("https://www.oxfam.org.uk%s\n" % l)
```

Listing 8.5: Function to write to a text document

As we can see in Listing 8.6, the function getPageLinks loops through all the pages for a given category and builds a link to a page with the hardcoded information we created earlier. For each page we then send a request to the page and receive a html response. Then we use BeautifulSoup to find the url for all the product listings found on that page which we managed to pinpoint during our analysis of the website.

```python
for page in range(1, numberPages + 1):

        link = firstPartLink + str(page) + secondPartLink

        response = requests.get(link)
        html = response.content
        soup = BeautifulSoup(html, "html.parser")

        for s in soup.find_all("li", {"class": pagesClass}):
            foundHref = s.find("a", href=True)
            retList.append(foundHref["href"])
```

Figure 8.6: Builds a url for each page of products, scrapes the urls of products and returns it

Once we have saved all the product urls for a given category, e.g. women's clothing, to a text document. We want to go to each page and retrieve the data we want to scrape. We created a function for this, which can be seen in Listing 8.7, called scrapeToCSV.

```python
scrape_products.scrapeToCSV(womenPagesName, attributesClass, imagesClass, csvName)
```

Listing 8.7: Builds a url for each page of products, scrapes the urls of products and returns it

To be able to write the data we need to create a .csv document. We will create a .csv with a name that we define and the first row of the document is written when we create the document. This row is the header that we will use to name our columns. The column names can be seen in Listing 8.8.

```python
def firstCSV(csvName):
    f = csv.writer(open(csvName, "w"))
    f.writerow(["Type", "Colour", "Brand", "Exact Colour", "Title", "Img name", "Img src"])
```

Listing 8.8: Creating a .csv document with headers

The scrapeToCSV function is run when we have written all the product listing urls to a .text document. As we can see in Listing 8.9, the first thing we will do in this function is to open the .csv document we created earlier, with the column names as the first row, so that we can store the tags that we will scrape. We will also create a list called lst that will contain all the product listing urls, as we can see in Listing 8.10. We created a function for this called txtToList, which basically just loops through each line in the .text document and then adds each line as a list item in a list.

```
f = csv.writer(open(csvName, "a"))
```

Listing 8.9: Builds a url for each page of products, scrapes the urls of products and returns it

```
lst = helper_functions.txtToList(txtName)
```

Listing 8.10: Builds a url for each page of products, scrapes the urls of products and returns it

We will then loop through every url in the list, send a request to that website and get a html response that we can use to get our data. To find the subset of the html that contains the attributes, we will use our analysis which showed us within which class the attributes were contained in, as we can see in Listing 8.11.

As we can see in Listing 8.12, we created a function that works for all attributes, where the function only needed to get the subset html and the attribute we wanted to extract. According to our analysis, we noticed that each attribute was contained in a “dt” tag, e.g. “Type:”, and each value for that attribute was contained in the next “dd” tag, e.g. “Jacket”. Since the code finds all the next “dd” siblings after the “dt” tag, we only want the first one, which is the value we want to associate with the attribute. Lastly, since the value contains “<dt>” tag before the value and “</dt>” after, we will cut out these tags before returning the value.

```
prodAttr = soup.find(class_=attributeClass)

prodType = helper_functions.get_attribute(prodAttr, "Type:")
prodColour = helper_functions.get_attribute(prodAttr, "Colour:")
prodBrand = helper_functions.get_attribute(prodAttr, "Brand:")
prodExactColour = helper_functions.get_attribute(prodAttr, "Exact colour:")
prodTitle = helper_functions.get_attribute(prodAttr, "Title:")
```

Listing 8.11: Getting product attributes

```
def get_attribute(soup, attr):
    prodAttr = soup.find("dt", string=attr)
    retAttr = ""

    if prodAttr:
        siblings = prodAttr.find_next_siblings("dd")
        sibling = siblings[0]
        retAttr = str(sibling)[4:-5]
    return retAttt
```

Listing 8.12: General code that works with all attributes

The same logic applied for getting the image url. First we will use BeautifulSoup to get a subset of the html that is contained within the class we found out that the image url is located in through our analysis as we can see in Listing 8.13. Then we want to get the url ready so that we can write it to the .csv document.

As we can see in Listing 8.13, we want to have the name of the image as an empty string, in case there is no image found. This will help us when we are cleaning the data so that we can drop those rows that don't have an image.

Next, we want to find the image url that we want to scrape. As we can see in Listing 8.14, we only want the url if it contains "/Main/" and if it does, we will save the url. We also want to get the image name so that we can link the image to the row. To achieve this, we will split the image url and only get the partial string of the url that begins with "HD_" as this will be the image file name when we download the image.

```
prodImages = soup.find(class_=imageClass)
prodImage = prodImages.findAll("img")
prodImageName = ""
```

Listing 8.13: Getting a subset of the html code for the image

```
if prodImage:
    for image in prodImage:
        imgStr = str(image["src"])

        if "/Main/" in imgStr:
            prodImage = "https:" + imgStr
            prodImageName = "HD_" + prodImage.split("HD_")[1]
        else:
            prodImage = ""
```

Listing 8.14: Getting the url for the image we want to scrape

The last step of the program is to write all the data we scraped to a .csv document. If no data is found, we will write an empty string for the column in the row. This will make it so that once we are cleaning the data, it will come up as a NaN value in pandas.

8.2 Netflea.com scraping code

Since we had already scraped the data from Oxfam.org.uk before we built the scraping code for Netflea.com, we could apply some lessons learnt from the Oxfam.org.uk code.

One of the main disadvantages of the Oxfam.org.uk code was that since we were storing the image url to download the images later, it resulted in a high amount of missing images. We will rectify this by downloading the image at the same time we are scraping a particular product page. This will decrease the number of missing images significantly, since most missing images in the Oxfam.org.uk case was due to removal of listings in the time between the scraping of the image url and the download of the image url.

We also noticed that getting all product urls and saving them and then going through them afterwards to scrape was redundant. We therefore decided to go through each page, then through each product listing in that page before going to the next page.

The same skeleton code we used for Oxfam.org.uk was used here. We used the lessons learnt from scraping Oxfam.org.uk and the analysis of NetIfea.com to build the code..

As with Oxfam, we made variables that we could build an url to loop through, as we can see in Listing 8.15, that were hard coded so that we could loop from 1 to womenNumberOfPages and add the iteration number to the womenFirstPartLink.

```
womenFirstPartLink = "https://www.netflea.com/womens-clothing.html?limit=100&p="
womenNumberOfPages = 550
```

Listing 8.15: Hardcoded information we will use when looping through pages

Before we went ahead to begin scraping, we wanted to make a .csv document that would contain the data we will scrape and we also wanted to make a folder called images in the linux system the code would be run, as we can see in Listing 8.16.

```
helper_functions.first_CSV(csvName)
imagesPath = helper_functions.setting_image_path()
```

Listing 8.16: Hardcoded information we will use when looping through pages

The first_CSV function is the same as we used for Oxfam.org.uk. It takes in a given name which will be the file name and then writes the first row as a header, which will be the column names as can be seen in Listing 8.17.

```
def first_CSV(csvName):
    f = csv.writer(open(csvName, "w"))
    f.writerow(["Type", "Brand", "Colour", "ImgName"])
```

Listing 8.17: Creating the .csv document and writing the header row

The setting_image_path function creates a string that contains the path to an images folder in the linux system. It then creates the folder and finally returns the path as we can see in Listing 8.18.

```
def setting_image_path():
    imagesFolder = Path.cwd().joinpath('images')
    imagesFolder.mkdir(parents=True, exist_ok=True)
    return imagesFolder
```

Listing 8.18: Setting the path to our images folder

Next step is to go into a function called getPageLinks. The function does the same as in the Oxfam.org.uk code, that is it builds a link for each page in a given category, sends a request to the page and gets a html response back. We then search that html for a subset of the code that is contained within a class that we found during our analysis. This subset will contain all the product listings urls on that page. The difference in this code and the other code is that we use if statements more, in case there is an empty product listing. If there is an empty product listing, we will go to the next product listing as we can see in Listing 8.19. We also do not want to keep a record of the product listing urls, but rather go to each product listing and begin scraping.

```
prodLink = soup.findAll("h2", {"class":"product-name"})

if prodLink:
    for p in prodLink:
        prodHref = p.find("a", href=True)
        if prodHref:
            scrape_products.scrapeToCSV(prodHref["href"], csvName, imagesPath)
```

Listing 8.19: Finding a subset of the html to get all product listings urls

Once we have established that the page contains product listing urls, we want to loop through each product listing and send the product listing url, the name of the .csv document and the path to the images folder to another function called scrapeToCSV, which handles the data on the product listing.

The scrapeToCSV function is basically the same as the function for Oxfam.org.uk. Here we send a request to the product listing url, get a html response and then get a subset of the html that contains the attributes that we want to scrape. We then go into another function called get_attribute with each of the attributes that we want to scrape as we can see in Listing 8.20.

```
prodAttr = soup.findAll(class_="attribute-wrapper")

prodType = helper_functions.get_attribute(prodAttr, "Type")
prodBrand = helper_functions.get_attribute(prodAttr, "Brand")
prodColour = helper_functions.get_attribute(prodAttr, "Color")
```

Listing 8.20:Searching the html response for a subset of the code within the class “attribute-wrapper”

The get_attribute function also works like the get_attribute function for Oxfam.org.uk. The main difference, as we can see in Listing 8.21, is that the value for each attribute is contained within a different html code and we factored that in when we built the code.

```
def get_attribute(soup, attr):
    if soup:
        for p in soup:
            prodAttr = p.find("div", string=attr)
            if prodAttr:
                attrVal = p.find("span", {"class": "outeratc"})
                if attrVal:
                    return attrVal.text.strip()
                else:
                    return ""
    else:
        return ""
```

Listing 8.21:Getting the attribute for a specific type

Getting the image url also works almost in the same way as it did with Oxfam.org.uk. We get a subset of the html code that contains the image url. According to our analysis we noticed that there were two images and the one that was contained within the class “cloud-zoom” was a better quality image, we therefore will be looking for that image, as we can see in Listing 8.22. We will put the image url as an image name and extract the image name when we are cleaning the data as opposed to get the image file name in this step. Lastly, we will download the image through the function wget_image before we go to the next product listing and we will utilize the wget command in Linux to achieve this.

```
prodImages = soup.find(class_="product-image")
    if prodImages:
        prodImage = prodImages.find("a", {"class": "cloud-zoom"})

        if prodImage:
            imgName = str(prodImage["href"])
            helper_functions.wget_image(imgName, imagesPath)
        else:
            imgName = ""
    else:
        imgName = ""
```

Listing 8.22: Finding the image url within the html

Instead of downloading all the images in one go after we scraped the data, we decided it would be better to download the image for a particular product when we were scraping the data from that product. To do this, we needed to find a way to achieve this in our code. Since we used wget to download the images from Oxfam.org.uk, we decided that we would use the wget with the Python code. Using the os module allows us to run Linux commands within a Python application and as we can see in Listing 8.23, we just need to send it to the os module as a string that contains the command we would use in a Linux terminal.

```
def wget_image(url, imagesPath):
    try:
        wgetStr = "wget -P " + str(imagesPath) + " " + url
        os.system(wgetStr)
    except OSError as e:
        print("Failed to wget " + str(imagesPath), e.strerror)
```

Listing 8.23: Using wget to download the image into the images folder

The last responsibility of the function is to write the attributes along with the image url to a .csv document. This is done so that we can load the .csv document in a pandas dataframe so that we can both clean the data and manipulate the data.

Once all these steps are done, the program will continue for the next product listing and once all product listings are complete, it will go on the next page of the category. Once all pages are done it will go to the next category and then finish.

9 Getting the data

When we were building our scraping code for scraping Oxfam.org.uk, we used our personal laptops to test the code and to successfully scrape a dozen product listings. To scrape all the product listings would take at least a few days and since we couldn't keep our laptops on at all times, we faced the problem on how we were going to run the code over the course of a couple of days.

One team member had access to a server located in Germany. The server was leased from the company Hetzner and had a 1Gbps dedicated up and down bandwidth. The server was running on a headless Ubuntu server distro so it fitted our needs perfectly, as the scraping code could be run through a headless setup.

Running the code for Oxfam.org.uk took a few tries. The code was running for around six hours when it suddenly stopped due to error. We found out that the error was due to a listing being removed and therefore the code couldn't find the attributes. We fixed this in our code and ran it again. Then came another error around one day later. When we had done a test run of the code locally, we forgot to check if the other categories worked as well. This caused the code to fail due to a misspelling when the code was supposed to go to the next category. Since the code kept a log of all the product listings, we fixed the issue and ran the code starting from the category it failed on. In total, the code was running for around three days and managed to exit without a failure at the end. Once we had scraped all the data, we added all the product image urls to a .text document and used wget to download the images. This process took around 20 minutes.

When it came to run the code for Netflea.com, we did tests so that we wouldn't run into the same problems as we did with the Oxfam.org.uk code. The code ran for around nine days without failing and managed to exit without a failure. Since we downloaded the images at the same time we scraped, we didn't need to download the images at the end of the process.

Once it became clear that the datasets built using the data from Oxfam.org.uk and Netflea.com were not good enough, we built a backup dataset. This dataset was built manually by choosing the images we would include in the dataset so we didn't need to scrape the images, but instead only needed to wget the images.

Once we had downloaded the images, we needed to build our own .csv document with the tags for the images. This was done manually by the team and double checked to make sure that the images had the correct tags.

9.1 Oxfam.org.uk Data

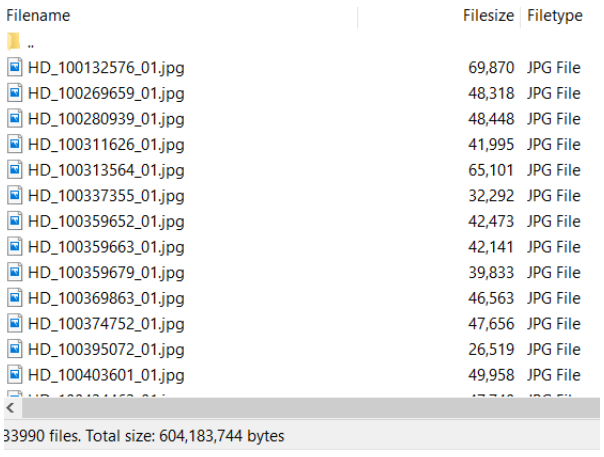
The code was running for three days and managed to scrape 41.281 listings in total (excluding the header).

As we can see in Fig. 9.1, the scraped data contained a lot of missing values and some values are in different cells, although .csv should be delimited by a comma, not cells.

41259	,White,Unbranded ,,,HD_100311626_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_100311626_01.jpg?v=1
41260	,Grey,Heaven sent ,,,HD_100313564_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_100313564_01.jpg?v=1
41261	Trousers,Blue,Planet,,,HD_101655542_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101655542_01.jpg?v=1
41262	Mini skirt,Red,Miss Selfridge,,,HD_101583722_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101583722_01.jpg?v=1
41263	Sleeveless,Grey,Oasis,Grey,,HD_101523071_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101523071_01.jpg?v=1
41264	Scarf,Black,Unbranded,Black,,HD_101769995_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101769995_01.jpg?v=1
41265	Strapless dress,Black,Forever21,,,HD_101356379_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101356379_01.jpg?v=1
41266	Strapless dress,Green,Maxazria,,,HD_101476846_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101476846_01.jpg?v=1
41267	Strapless dress,Black,Yessica,,,HD_101661399_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101661399_01.jpg?v=1
41268	Casual jacket / coat,Red,DKNY ,Terracota Red ,,HD_101475457_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101475457_01.jpg?v=1
41269	Vintage,Black,Debut,,,HD_101432369_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101432369_01.jpg?v=1
41270	Trousers,M S Marks &i Spencer,White,,HD_101712014_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101712014_01.jpg?v=1
41271	Trousers,M S Marks &i Spencer,White,,HD_101712015_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101712015_01.jpg?v=1
41272	Trousers,C S Marks &i Spencer,Ivory,,HD_101713843_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101713843_01.jpg?v=1
41273	Knee length dress,Red,Jack Wills, Red/Pink/Yellow/Blue,,HD_101735931_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101735931_01.jpg?v=1
41274	Blouse,Blue,Whistles,,,HD_101557787_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101557787_01.jpg?v=1
41275	Full length dress,White,A Mere Co,White,,HD_101736165_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101736165_01.jpg?v=1
41276	Mules,Yellow,ASOS,Yellow,,HD_101768207_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101768207_01.jpg?v=1
41277	Cropped trousers,Brown,Savane,Brown,,HD_101636185_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101636185_01.jpg?v=1
41278	Beanie,Blue,Unbranded,Navy,,HD_101791992_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101791992_01.jpg?v=1
41279	Beanie,Black,Unbranded,Black/Burgundy,,HD_101791994_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101791994_01.jpg?v=1
41280	Casual jacket / coat,Cream / ivory,Miss Sixty,,,HD_101367852_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101367852_01.jpg?v=1
41281	Jeans,Black,Liz Claiborne,,,HD_101773208_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101773208_01.jpg?v=1
41282	Casual jacl M,,,HD_101475296_01.jpg?v=1,https://yatra8exe7uvportalprd.blob.core.windows.net/images/products/HighStDonated/Main/HD_101475296_01.jpg?v=1
41283	

Figure 9.1: .csv document with scraped data

Once we had the image urls, we took all the image urls and put them in a .txt document. We then used a tool with Linux called wget to download all the images. In total we managed to download 33.990 images as can be seen in Fig 9.2. Although we managed to scrape so many images, the total file size was only ~600 MB, which indicated that the images were extremely low quality, hence the usability to train a model using them is not good.



Filename	Filesize	Filetype
..		
HD_100132576_01.jpg	69,870	JPG File
HD_100269659_01.jpg	48,318	JPG File
HD_100280939_01.jpg	48,448	JPG File
HD_100311626_01.jpg	41,995	JPG File
HD_100313564_01.jpg	65,101	JPG File
HD_100337355_01.jpg	32,292	JPG File
HD_100359652_01.jpg	42,473	JPG File
HD_100359663_01.jpg	42,141	JPG File
HD_100359679_01.jpg	39,833	JPG File
HD_100369863_01.jpg	46,563	JPG File
HD_100374752_01.jpg	47,656	JPG File
HD_100395072_01.jpg	26,519	JPG File
HD_100403601_01.jpg	49,958	JPG File

33990 files. Total size: 604,183,744 bytes

Figure 9.2: Downloaded images folder viewed with FTP viewer

9.2 Netflea.org.uk Data

The code was running for 9 days straight and managed to scrape 86.900 listings in total (excluding the header).

As we can see in Fig. 9.3, the scraped data contains either a lot of missing data (the empty string between commas indicates missing data), and contains a lot of “Other Brand” for the brand column.

Figure 9.3: .csv document with scraped data for Netflea.com

Since we downloaded the images at the same time as we scraped the information, we expected that we would manage to get more images in comparison with the Oxfam code. As we can see in Fig. 9.4, we managed to scrape 86.655 images out of possible 86.900 (~99.72% success rate vs. ~82.34% at Oxfam).

The image quality was also significantly better than from Oxfam, or ~99KB on average vs. ~18KB.

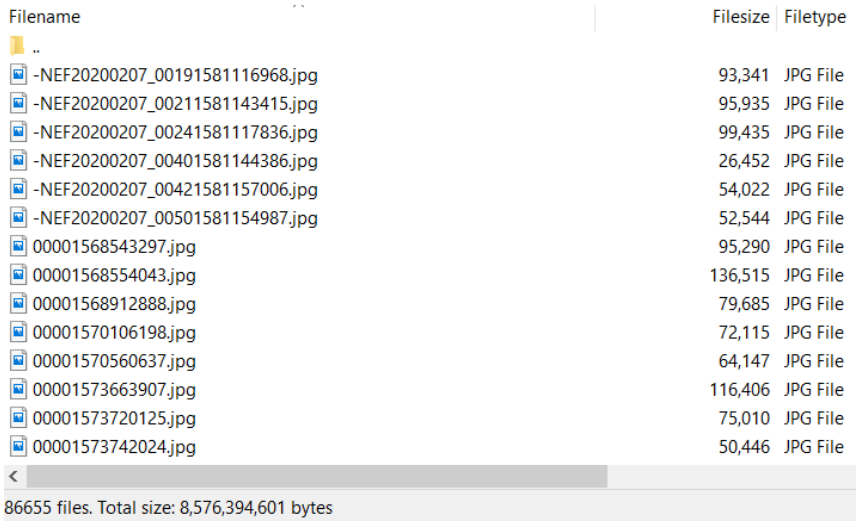


Figure 9.4: Downloaded images folder viewed with FTP viewer

9.3 Backup Data

We spent a total of 45 hours finding images for our backup dataset and manually tagging the images in a .csv document.

In total we managed to find 2.886 images as we can see in Table 9.1.

Source	Total Images	Percentage of Total	Total Size	Percentage of Total	Average Size
Grailed.com	846	~29.31%	504 MB	~45.12%	596 KB
Picclick.com	790	~27.37%	238 MB	~21.31%	301 KB
Poshmark.com	1.121	~38.84%	345 MB	~30.89%	308 KB
Shpock.com	129	~4.47%	30 MB	~2.69%	233 KB
Total	**2.886**	**~100%**	**1.117 MB**	**~100%**	**387 KB**

Table 9.1: Total downloaded images with total size and average size

Grailed.com, Picclick.com and Poshmark.com yielded a respectable amount of images, but the time spent on finding images on Shpock.com yielded only 129 images or ~4.47% of the total images.

Grailed.com had the highest average image size, or 596 KB, while Shpock.com had the lowest average image size, 233 KB.

The total size of the 2.886 images was 1.117 MB or 387 KB on average for each image. In comparison to Oxfam.org.uk and Netflea.com, the quality of these images are much higher as we can see in Table 9.2.

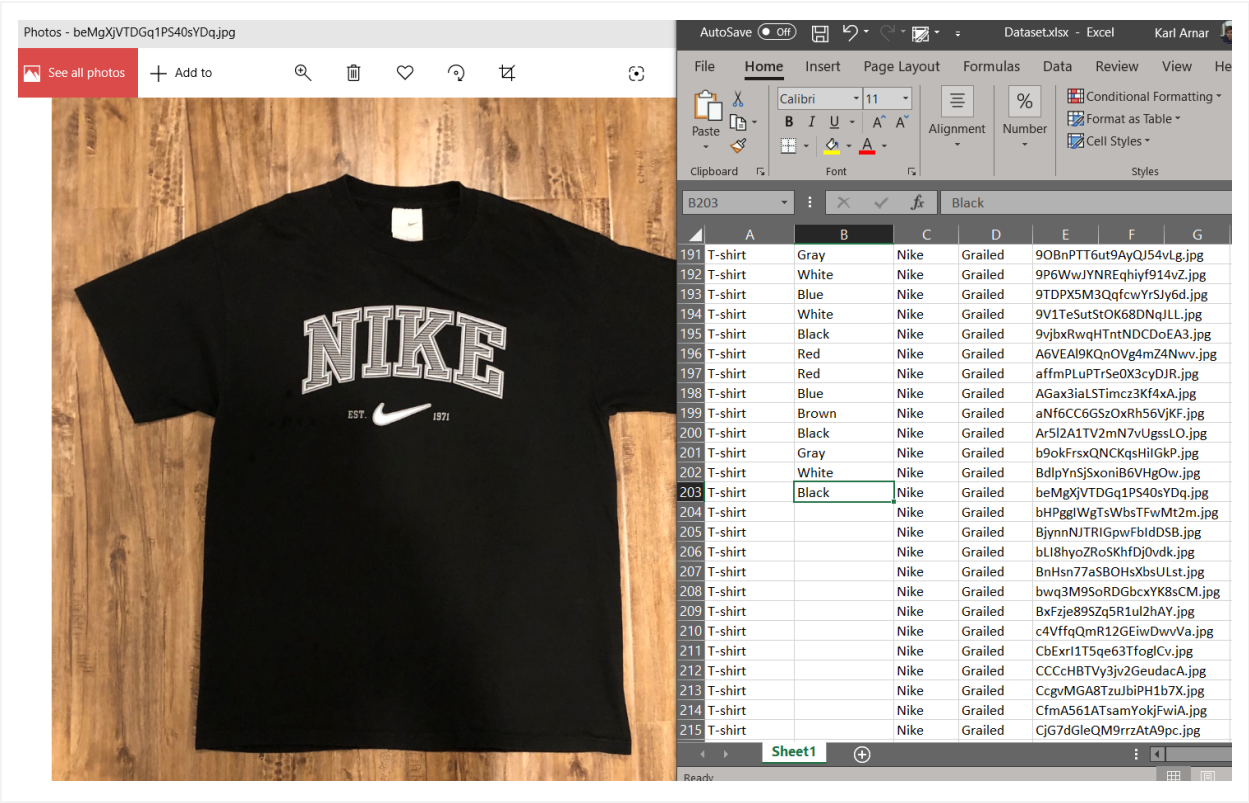
Dataset	Total Images	Total Size	Average Size
Oxfam.org.uk	33.990	604 MB	18 KB
Netflea.com	86.655	8.576 MB	99 KB
Backup	2.886	1.117 MB	387 KB

Table 9.2: Comparison of datasets

The images in the backup dataset are on average almost 22 times larger in size than Oxfam.org.uk and almost 4 times larger in size than Netflea.com.

Although the backup dataset contains a lot less images than the other two datasets, the backup dataset only contains a select few labels as opposed to the other two. For each label in the backup dataset, it contains on average more images than the other two which render the total images a useless metric when comparing the datasets.

Once we had downloaded all the images we needed to manually tag them. To do so, we opened up Microsoft Excel, exported the image names and put them into a column. For each image name, we then added the tags, as can be seen in Fig. 9.5.



	A	B	C	D	E
191	T-shirt	Gray	Nike	Grailed	9OBnPTT6ut9AyQJ54vLg.jpg
192	T-shirt	White	Nike	Grailed	9P6WwJYNREqhiyf914vZ.jpg
193	T-shirt	Blue	Nike	Grailed	9TDPX5M3QqfcwYrSJy6d.jpg
194	T-shirt	White	Nike	Grailed	9V1TeSutStOK68DNqJLL.jpg
195	T-shirt	Black	Nike	Grailed	9vjbxRwqHTntNDCDoEA3.jpg
196	T-shirt	Red	Nike	Grailed	A6VEAl9KQnOVg4mZ4Nwv.jpg
197	T-shirt	Red	Nike	Grailed	affmPLuPTrSe0X3cyDJR.jpg
198	T-shirt	Blue	Nike	Grailed	AGax3iaLSTimcz3Kf4xA.jpg
199	T-shirt	Brown	Nike	Grailed	aNf6CC6GSzOxRh56VjKF.jpg
200	T-shirt	Black	Nike	Grailed	Ar5l2A1TV2mN7vUgssLO.jpg
201	T-shirt	Gray	Nike	Grailed	b9okFrsxQNCKqsHiIGkP.jpg
202	T-shirt	White	Nike	Grailed	BdlpYnSjSxoniB6VHgOw.jpg
203	T-shirt	Black	Nike	Grailed	beMgXjVTDGq1PS40sYDq.jpg
204	T-shirt		Nike	Grailed	bHPgglWgTsWbsTFwMt2m.jpg
205	T-shirt		Nike	Grailed	BjynnNJTRIGpwFbIdDSB.jpg
206	T-shirt		Nike	Grailed	bLI8hyoZRoSKhfDj0vdk.jpg
207	T-shirt		Nike	Grailed	BnHsn77aSBOHsXbsULst.jpg
208	T-shirt		Nike	Grailed	bwq3M9SoRDGbcxYK8sCM.jpg
209	T-shirt		Nike	Grailed	BxFzje89SZq5R1ul2hAY.jpg
210	T-shirt		Nike	Grailed	c4VffqQmR12GEiwDwvVa.jpg
211	T-shirt		Nike	Grailed	CbExrl1T5qe63TfoglCv.jpg
212	T-shirt		Nike	Grailed	CCCcHBTVy3jv2GeudacA.jpg
213	T-shirt		Nike	Grailed	CcgvMGA8TzuJbiPH1b7X.jpg
214	T-shirt		Nike	Grailed	CfmA561ATsamYokjFwiA.jpg
215	T-shirt		Nike	Grailed	CjG7dGleQM9rrzAtA9pc.jpg

Figure 9.5: Tagging images in backup dataset

Although this process was time consuming, it was essential that the images were correctly tagged as it would only slow the process down later when we were preparing the dataset to be trained.

We however did a few things that would make the process much less time consuming. When we downloaded the images, we put them into a folder which was descriptive of the clothing item, e.g. “Grailed-Adidas-Hoodie” and “Grailed-Adidas-Tshirt”. This was done so that we could add the tags in bulk for everything except for the colour, meaning that we would only need to add the colour tag when tagging the images.

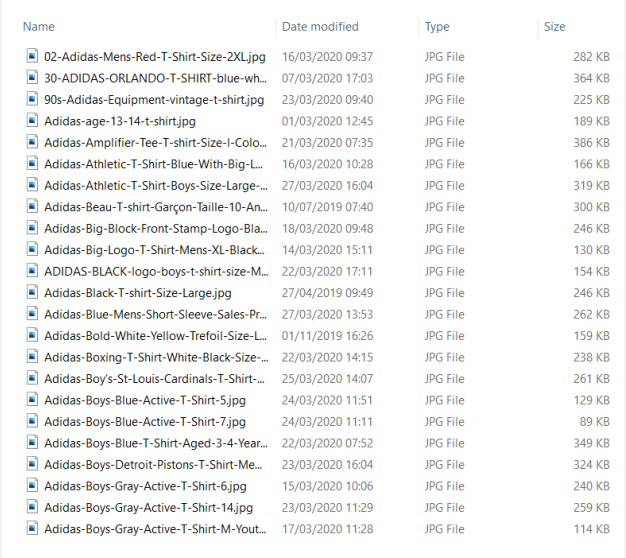
For two of the websites, Picclick.com and Poshmark.com, we noticed that the file names of the images were rather long, so we decided to rename each of all the files with a random number. An example of our code for this can be seen in Listing 9.1.

```
randomNumbers = []
for _ in range(numberOfFiles):
    val = randint(1000000000, 9999999999)
    randomNumbers.append(val)

for index, file in enumerate(files):
    oldName = os.path.abspath(file)
    newName = thisFolderPath + "/" + str(randomNumbers[index]) + ".jpg"
    os.rename(oldName, newName)
```

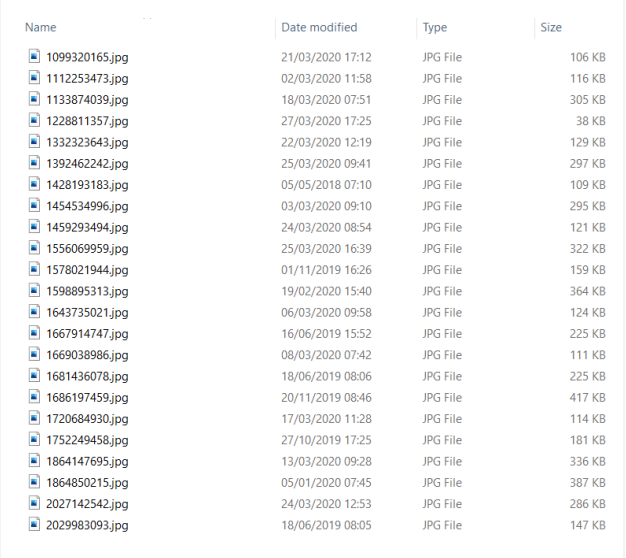
Listing 9.1: Subset of renameFilesPicclick.py code

This resulted in image names which did not contain any of the potential tags and were instead just a random number. As we can see in Fig. 9.6, the image names were rather long before and in Fig. 9.7, we can see how the image names are now just a random number.



Name	Date modified	Type	Size
02-Adidas-Mens-Red-T-Shirt-Size-2XL.jpg	16/03/2020 09:37	JPG File	282 KB
30-ADIDAS-ORLANDO-T-SHIRT-blue-wh...	07/03/2020 17:03	JPG File	364 KB
90s-Adidas-Equipment-vintage-t-shirt.jpg	23/03/2020 09:40	JPG File	225 KB
Adidas-age-13-14-t-shirt.jpg	01/03/2020 12:45	JPG File	189 KB
Adidas-Amplifier-Tee-T-shirt-Size-l-Colo...	21/03/2020 07:35	JPG File	386 KB
Adidas-Athletic-T-Shirt-Blue-With-Big-L...	16/03/2020 10:28	JPG File	166 KB
Adidas-Athletic-T-Shirt-Boys-Size-Large-...	27/03/2020 16:04	JPG File	319 KB
Adidas-Beau-T-shirt-Garçon-Taille-10-An...	10/07/2019 07:40	JPG File	300 KB
Adidas-Big-Block-Front-Stamp-Logo-Bla...	18/03/2020 09:48	JPG File	246 KB
Adidas-Big-Logo-T-Shirt-Mens-XL-Black...	14/03/2020 15:11	JPG File	130 KB
ADIDAS-BLACK-logo-boys-t-shirt-size-M...	22/03/2020 17:11	JPG File	154 KB
Adidas-Black-T-shirt-Size-Large.jpg	27/04/2019 09:49	JPG File	246 KB
Adidas-Blue-Mens-Short-Sleeve-Sales-Pr...	27/03/2020 13:53	JPG File	262 KB
Adidas-Bold-White-Yellow-Trefoil-Size-L...	01/11/2019 16:26	JPG File	159 KB
Adidas-Boxing-T-Shirt-White-Black-Size-...	22/03/2020 14:15	JPG File	238 KB
Adidas-Boy's-St-Louis-Cardinals-T-Shirt-...	25/03/2020 14:07	JPG File	261 KB
Adidas-Boys-Blue-Active-T-Shirt-5.jpg	24/03/2020 11:51	JPG File	129 KB
Adidas-Boys-Blue-Active-T-Shirt-7.jpg	24/03/2020 11:11	JPG File	89 KB
Adidas-Boys-Blue-T-Shirt-Aged-3-4-Year...	22/03/2020 07:52	JPG File	349 KB
Adidas-Boys-Detroit-Pistons-T-Shirt-Me...	23/03/2020 16:04	JPG File	324 KB
Adidas-Boys-Gray-Active-T-Shirt-6.jpg	15/03/2020 10:06	JPG File	240 KB
Adidas-Boys-Gray-Active-T-Shirt-14.jpg	23/03/2020 11:29	JPG File	259 KB
Adidas-Boys-Gray-Active-T-Shirt-M-Yout...	17/03/2020 11:28	JPG File	114 KB

Figure 9.6: Picclick.com names before



Name	Date modified	Type	Size
1099320165.jpg	21/03/2020 17:12	JPG File	106 KB
1112253473.jpg	02/03/2020 11:58	JPG File	116 KB
1133874039.jpg	18/03/2020 07:51	JPG File	305 KB
1228811357.jpg	27/03/2020 17:25	JPG File	38 KB
1332323643.jpg	22/03/2020 12:19	JPG File	129 KB
1392462242.jpg	25/03/2020 09:41	JPG File	297 KB
1428193183.jpg	05/05/2018 07:10	JPG File	109 KB
1454534996.jpg	03/03/2020 09:10	JPG File	295 KB
1459293494.jpg	24/03/2020 08:54	JPG File	121 KB
1556069959.jpg	25/03/2020 16:39	JPG File	322 KB
1578021944.jpg	01/11/2019 16:26	JPG File	159 KB
1598895313.jpg	19/02/2020 15:40	JPG File	364 KB
1643735021.jpg	06/03/2020 09:58	JPG File	124 KB
1667914747.jpg	16/06/2019 15:52	JPG File	225 KB
1669038986.jpg	08/03/2020 07:42	JPG File	111 KB
1681436078.jpg	18/06/2019 08:06	JPG File	225 KB
1686197459.jpg	20/11/2019 08:46	JPG File	417 KB
1720684930.jpg	17/03/2020 11:28	JPG File	114 KB
1752249458.jpg	27/10/2019 17:25	JPG File	181 KB
1864147695.jpg	13/03/2020 09:28	JPG File	336 KB
1864850215.jpg	05/01/2020 07:45	JPG File	387 KB
2027142542.jpg	24/03/2020 12:53	JPG File	286 KB
2029983093.jpg	18/06/2019 08:05	JPG File	147 KB

Figure 9.7: Picclick.com names after

After double checking the tags in the .csv document and fixing incorrect tags, we had our final dataset that contained a tag for “Type”, “Colour”, “Brand”, “Source” and “ImgName” which was ready for uploading to Google Cloud and start training a model based on the dataset.

10 Cleaning the data and exploring the data

To make the data ready for training with Google AutoML, we used Jupyter Notebook that we had access to through the University of Reykjavík[27]. One of the team members had some experience with data cleaning through a class in his studies that used Jupyter Notebook, so the team decided that we would use this to benefit from the experience, as opposed to learning how to clean data through another application.

Jupyter Notebook is a document which can contain both computer code (e.g. Python) and rich text elements (e.g. text in Markdown format, equations, figures, links, etc.). Notebook documents are both human-readable documents containing the analysis description and the results (figures, tables, etc…)[28], although we mostly used it to get the results and decided to keep the analysis description in this paper.

To help us with the data cleaning, we used a few modules. These modules were “pandas”, “re”, “matplotlib” and “seaborn”. The description for each of these modules can be viewed in Table 10.1.

Module	Description
pandas	A fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.[29]
re	A module that is a built-in package in Python, which can be used to work with Regular Expressions.[30]
matplotlib	A comprehensive library for creating static, animated, and interactive visualizations in Python.[31]
seaborn	A Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.[32]

Table 10.1: Description of the modules we are going to use when cleaning the data

Cleaning the Oxfam.org.uk data took a lot more time than cleaning the Netflea.com data. This can be attributed to the fact that the data from Oxfam.org.uk was a lot more “dirty”. The attribute values contained misspellings and were scattered around in a lot of attribute values. This was most likely due to the employees of Oxfam.org.uk having a free text box when inputting the attribute values but this meant that the data needed to be cleaned significantly compared to the Netflea.com data, which had far less misspellings and wasn’t scattered around in a lot of attribute values.

Cleaning the Oxfam.org.uk data was done while the code was running for Netflea.com so it was not a bottleneck per se, although it took a lot more time than anticipated, and was done before finishing scraping Netflea.com was concluded.

[27] "JupyterHub." https://jupyter.ru.is/. Accessed 21 Apr. 2020.

[28] "1. What is the Jupyter Notebook? — Jupyter/IPython Notebook ...." https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html. Accessed 14 Mar. 2020.

[29] "Pandas." https://pandas.pydata.org/. Accessed 14 Mar. 2020.

[30] "Python RegEx - W3Schools." https://www.w3schools.com/python/python_regex.asp. Accessed 14 Mar. 2020.

[31] "Matplotlib." https://matplotlib.org/. Accessed 14 Mar. 2020.

[32] "Seaborn." https://seaborn.pydata.org/. Accessed 14 Mar. 2020.

10.1 Cleaning data from Oxfam.org.uk

To start cleaning the data, we imported the four modules that we listed in Table 10.1. We referred to the pandas module as “pd”, the matplotlib module as “plt” and the seaborn module as “sns”, as we can see in Fig. 10.1. These are universally recognized abbreviations for these modules. Since the re module is already abbreviated, we won't be abbreviating that one.

Next step was to get the dataset on a dataframe form. This was done by using the read_csv function within the pandas module. As we can see in Fig. 10.1, we referred to the dataframe as “data”.

```
import pandas as pd
import re
import matplotlib.pyplot as plt
import seaborn as sns


data = pd.read_csv("Oxfam_Dataset.csv")
```

```
data.sample(10)
```

	Type	Colour	Brand	Exact Colour	Title	Img name	Img src
19762	Gloves	Cream / ivory	Johnsons	Cream	NaN	HD_101844674_01.jpg?v=1	https://yatra8exe7uvportalprd.blob.core.window...
33912	Trousers	Brown	CC PETITE	NaN	NaN	HD_101641681_01.jpg?v=1	https://yatra8exe7uvportalprd.blob.core.window...
22896	Court shoes	Black	New Look	Black	New Look Faux Suede Court Shoes NWOT Black Siz...	HD_200004723_01.jpg?v=1	https://yatra8exe7uvportalprd.blob.core.window...
7580	Long sleeved	White	T. M. Lewin	white and black stripes	NaN	HD_101549532_01.jpg?v=1	https://yatra8exe7uvportalprd.blob.core.window...
33162	Long dress	Pink	Khalil	NaN	NaN	HD_101672045_01.jpg?v=1	https://yatra8exe7uvportalprd.blob.core.window...
17440	Calf length	Black	Desigual	NaN	NaN	HD_101860414_01.jpg?v=1	https://yatra8exe7uvportalprd.blob.core.window...
34955	NaN	Beige	Per Una	NaN	NaN	HD_101605875_01.jpg?v=1	https://yatra8exe7uvportalprd.blob.core.window...
4702	NaN	Brown	M&S Collection	Brown	M&S Collection Slip on Loafers Brown Size: 10	HD_200002815_01.jpg?v=1	https://yatra8exe7uvportalprd.blob.core.window...
31859	Calf length	Blue	Per Una	Navy Blue Different shades of Green Purple	NaN	HD_101708715_01.jpg?v=1	https://yatra8exe7uvportalprd.blob.core.window...
28694	Necklace	Black	costume jewellery	Black cream and silver tonne	NaN	HD_101760550_01.jpg?v=1	https://yatra8exe7uvportalprd.blob.core.window...

Figure 10.1: Importing modules and .csv document to pandas dataframe in Jupyter

We then printed a sample 10 rows from the dataframe to get a quick understanding of the dataset and as we can see in Fig. 10.1, this resulted in a lot of NaN values, which was a missing value for that column. We can also see that the “Type” column contained a lot of values that we were not interested in using in our final dataset, such as “Gloves” and “Necklace”. We can also already see that by including the “Exact Colour” and “Title” attributes helped us when we were filling in the missing data in the other columns. For example, in row 4702, we see that the “Type” column had a missing value but the “Title” value had “Slip on Loafers” in the value which we could use to fill in the missing value.

Another thing that we can see is that the “Img name” column contained “?v=1” at the end of the string for each value. We cut that from the string to be able to link the row to the image file. “Img src” column was of no value to us as we already had the image name already, so we dropped that column.

After dropping the “Img src” column, we wanted to know how many missing values there were for each column. Looking back at Fig. 10.1, we can draw the conclusion that the most missing values would be in the “Exact Colour” and “Title” columns based on the sample data.

However, there is a simple way to check all the missing values for each column in Python, which is a lot better than just drawing conclusions. As we can see in Fig. 10.2, the conclusions that we drew were correct for the “Exact Colour” and “Title” columns, however the “Type” column also contained a lot of missing values or 14.080. This was a significant number or as we found out earlier in Fig. 9.1, that we managed to scrape 41.281 rows, meaning that the type was missing ~34.11% of the total rows. Dropping these rows straight away would mean that our dataset would shrink to 27.201 rows or only ~65.89% of the original dataset.

```
data.isna().sum()
```

```
Type            14080
Colour            634
Brand             121
Exact Colour    19892
Title           34049
Img name            0
dtype: int64
```

Figure 10.2: Listing the sum of NaN values in each column

Luckily we had the “Title” column which was included in the scraping process if this scenario would arise, which it did. We therefore went through the missing values in the “Type” column and checked if there was a known value in the “Title” column. If both columns were empty, we dropped those rows as it was crucial for our final dataset to contain a type.

In Table 10.2, we can see the missing values and their percentage of missing values versus the total dataset. The “Exact Colour” and “Title” columns are only included to fill in missing values in other columns, so we did not look at lowering the amount of missing values in these columns.

Column	# Missing	% Missing	% Known
Type	14.080	~34.11%	~65.89%
Colour	634	~1.54%	~98.46%
Brand	121	~0.29%	~99.71%
Exact Colour	19.892	~48.19%	~51.81%
Title	34.049	~82.48%	~17.52%
Img name	0	0%	100%

Table 10.2: Percentage of missing and known values

10.1.1 Getting rid of unwanted types in “Type” column

Before we started looking at filling in the missing values in the “Type” column, we wanted to remove those types that we were not interested in keeping in the dataset that we noticed in Fig. 10.1, such as “Gloves”.

To do this, we printed out all the unique values in the “Type” column, which corresponds to the types. As we can see in Fig. 10.3, a subset of these unique values showed a lot of types that we weren’t interested in keeping in our final dataset. For example, “Trainers”, “Boots”, “Flat shoes”, “Sandals” and “Pyjamas” are types that we wanted to exclude. Fig. 10.4 shows a subset of all the types that we decided to exclude.

```
data.Type.unique()
```

```
array(['Cardigan', nan, 'Other', 'Full length dress', 'Parka',
       'Quilted jacket', 'Coat', 'Jacket', 'Hoodie', 'Long dress',
       'Summer', 'Calf length', 'Sleeveless', 'Fleece jacket',
       'School uniform', 'Leotard', 'Jeans', 'Dress /  gown',
       'Body warmer', 'Tracksuit top', 'Knee length dress',
       'Long sleeved shirt', 'Gilet', 'Wetsuit', 'Vintage', 'Jumper',
       'Sleeveless top', 'Evening dress', 'Trainers', 'Slip-on shoes',
       'Bikini', 'Trousers', 'Denim jacket', 'T-Shirt', 'Raincoat',
       'Prom dress', 'Smock top', 'Polo shirt', 'One piece', 'Sweater',
       'Boots', 'Knee length skirt', 'Bomber jacket', 'Mini skirt',
       'Casual jacket / coat', 'Flat shoes', 'Hot pants', 'Trenchcoat',
       'Patterned skirt', 'Leather jacket', 'Vest', 'Cargo pants',
       'Jeggings / stretch trousers', 'Cropped trousers', 'Ballet',
       'Blouse', 'Checked skirt', 'Short', 'Smock', 'Short sleeved shirt',
       'Hooded top', 'Mini dress', 'Jeggings / stretch jeans',
       'Pleated skirt', 'Gypsy skirt', 'Quilted coat', 'A-line skirt',
       'Sandals', 'Overcoat', 'Strapless dress', 'Padded jacket',
       'Dance shoes', 'Single breasted blazer', 'Long sleeved',
       'Sweatshirt', 'All in one', 'Pullover', 'Hiking jacket',
       'Surf shorts', 'Ski trousers', 'Waxed jacket', 'Tracksuit bottoms',
       'Short sleeved T-shirt', 'Combats', 'Pyjamas', 'Deck shoes',
       'Short sleeved', '3 piece suit', 'Football socks', 'Hi tops',
       'Single breasted suit jacket', 'Cargo shorts', 'Football top',
       'Plimsolls', 'Tie', 'Scarf', 'Chelsea / ankle boots', 'Lace-ups',
       'Work boots', 'Double breasted blazer', 'Sports top', 'Wallet',
       'Cowboy hat', 'Walking, hiking &amp;amp; trail', 'Single breasted suit',
       'Football', 'Belt', 'Loafers', 'Denim jeans', 'Beanie', 'Rugby top',
       'Manbag', 'Smart jacket / coat', 'Chinos', 'Waistcoat',
       'Tail coat suit jacket', 'Tail coat suit', 'Leather trousers',
```

Figure 10.3: Subsample of the types in the “Type” columns

```
lstToDel = ['Summer', 'School uniform', 'Leotard', 'Body warmer', 'Wetsuit', 'Slip-on shoes',
            'Bikini', 'Prom dress', 'Boots', 'Flat shoes', 'Ballet', 'Sandals', 'Dance shoes',
            'All in one', 'Pyjamas', 'Deck shoes', 'Football socks', 'Plimsolls', 'Tie', 'Scarf',
            'Chelsea / ankle boots', 'Lace-ups', 'Work boots', 'Wallet', 'Cowboy hat',
            'Walking, hiking &amp;amp; trail', 'Football', 'Belt', 'Loafers', 'Beanie', 'Rugby top',
            'Manbag', 'Brogue', 'Baseball shoes', 'Slipper shoes', 'Ski suit', 'Cross body bag',
            'Bow tie', 'Mittens', 'Moccasisns', 'Sports Wristwatch', 'digital watch',
            'Weekend Bag', 'Cuff links', 'Running shoes', 'Baseball cap', 'Briefs', 'Cravat',
            'Rucksack', 'Flip flops', '43924', 'Skate shoes', 'Peaked cap', 'Driving gloves',
            'Slip ons', 'Bowler hat', 'Cuff links', 'Outfit', 'Biker boots', 'Bush hat',
            'Coin purse', 'Woolie hat', 'Boxers', 'Gloves', 'Cuff Links', 'Robe', 'Desert boots',
            'Tie Pin', 'Snow &amp;amp; winter boots', 'Trilby', 'Wrist watches', 'Backpack',
            'College scarf', 'Trunks', 'Golf shoes', 'Tablewear', 'Make up bag', 'Board',
            'Salopettes', 'Strapped sandals', 'Motorcycle trousers', 'Sports sandals',
            'Sports gloves', 'Cowboy boots', 'Hiking trousers', 'Mixed items', 'Kilt',
```

Figure 10.4: Subsample of the types in the “Type” column

Our approach to remove these types was to take the listToDel list from Fig. 10.4 and compare it to our dataset. We kept all the rows that were not contained in the list. As we can see in Fig. 10.5, this approach yielded a reduction of types in the amount of 209. A reduction from 321 to 112 total types. Fig. 10.6 shows the remaining types in the dataset. This lowered the total amount of rows in the dataset to 35.380 or a reduction of 5.901 rows from 41.281 or ~14.29% decrease.

```
before = data.Type.unique()
beforeLen = len(before)
print("Number of unique values in 'Type' before: ", beforeLen)

data = data[~data.Type.isin(lstToDel)]

after = data.Type.unique()
afterLen = len(after)
print("Number of unique values in 'Type' after: ", afterLen)

difference = beforeLen - afterLen
print("Difference is: ", difference)
```

```
Number of unique values in 'Type' before:  321
Number of unique values in 'Type' after:  112
Difference is:  209
```

Figure 10.5: Removing unwanted types in “Type” column

```
data.Type.unique()
```

```
array(['Cardigan', nan, 'Full length dress', 'Parka', 'Quilted jacket',
       'Coat', 'Jacket', 'Hoodie', 'Long dress', 'Calf length',
       'Sleeveless', 'Fleece jacket', 'Jeans', 'Dress /  gown',
       'Knee length dress', 'Long sleeved shirt', 'Gilet', 'Jumper',
       'Sleeveless top', 'Evening dress', 'Trousers', 'Denim jacket',
       'T-Shirt', 'Raincoat', 'Smock top', 'Polo shirt', 'Sweater',
       'Knee length skirt', 'Bomber jacket', 'Mini skirt',
       'Casual jacket / coat', 'Hot pants', 'Trenchcoat',
       'Patterned skirt', 'Leather jacket', 'Vest', 'Cargo pants',
       'Jeggings / stretch trousers', 'Cropped trousers', 'Blouse',
       'Checked skirt', 'Short', 'Smock', 'Short sleeved shirt',
       'Hooded top', 'Mini dress', 'Jeggings / stretch jeans',
       'Pleated skirt', 'Gypsy skirt', 'Quilted coat', 'A-line skirt',
       'Overcoat', 'Strapless dress', 'Padded jacket',
       'Single breasted blazer', 'Long sleeved', 'Sweatshirt', 'Pullover',
       'Hiking jacket', 'Waxed jacket', 'Short sleeved T-shirt', 'Combats',
       'Short sleeved', 'Hi tops', 'Single breasted suit jacket',
       'Cargo shorts', 'Double breasted blazer', 'Sports top',
       'Denim jeans', 'Smart jacket / coat', 'Chinos', 'Waistcoat',
       'Leather trousers', 'PVC trousers', 'Long sleeved T-shirt',
       'Fleece jackets', 'Leather coat', 'Double breasted suit jacket',
       'Duffle coat', 'Waxed coat', 'Tank top', 'Tweed jacket',
       'Vintage skirt', 'Calf length skirt', 'Skirt suit', 'Evening',
       'Cocktail dress', 'Halter-neck dress', 'Long skirt', 'Pencil skirt',
       'Cap sleeved T-shirt', 'Full length', 'Wraparound top', 'Blazer',
       'Shirt', 'Wrap around', 'Asymmetrical dress', 'Dress', 'Jumpsuit',
       'Wrap around dress', 'Playsuit', 'Suit jacket', 'Sweat pants',
       'Cropped jeans', 'Harem pants', 'Leggings', 'Culottes',
       'Boiler suit', 'Puffball skirt', 'Tulip skirt', 'Short jacket',
       'Flared / kick flare jeans'], dtype=object)
```

Figure 10.6: “Type” column after removing unwanted types

Looking at the types, we noticed that we could possibly further reduce the types by combining some types into a single type. We figured that we needed to check the count for each type, and make an educated guess if the types could be merged.

As we can see in Table 10.3, which shows the top 15 types and the bottom 15 types, a lot of the low count types could be merged into a high count type, e.g. “Tulip skirt” could be merged with “Shirt” and “Flared / kick flare jeans” could be merged with “Jeans”.

Top 15 types		Bottom 15 types	
Type	Value	Type	Value
Jacket	1.713	Sweat pants	5
Trousers	1.501	Hooded top	5
Knee length dress	1.264	Tweed jacket	4
Smart jacket / coat	1.147	Short jacket	4
Long sleeved	988	Puffball skirt	4
Casual jacket / coat	932	Fleece jackets	4
Sleeveless	852	PVC trousers	4
Jumper	674	Tulip skirt	3
Blouse	669	Waxed jacket	3
Jeans	592	Boiler suit	2
Single breasted suit jacket	575	Tank top	2
Long sleeved shirt	567	Waxed coat	2
Cardigan	486	Hi tops	2
Mini dress	466	Flared / kick flare jeans	1
Cocktail dress	462	Dress	1

Table 10.3:Types with highest 15 values and lowest 15 values

To achieve this, we made a dictionary in Python named replaceDict which had all the types we wanted to merge as a key and the types it should be merged into as a value. We then looped through the dictionary replacing all values in the dataset that matched the key with the value in the dictionary.

This approach lowered the total number of unique values to 59 from the 112 types we had before, or a reduction of ~47.32%. The lowest value type now had 15 instead of 1, and the 15th lowest value type had a value of 84 instead of 5 previously. We still had 35.380 rows in our dataset since this approach didn’t require removal of rows.

10.1.2 Getting rid of NaN values

Removing unwanted types reduced the rows, but to check how it affected our dataset we needed to check the missing values again. In Table 10.4, we can see the missing values before and missing values after. Since we were removing unwanted types, the only rows that we were removing were from known values, so we had the same amount of missing values in "Type" as before. However, the percentage of missing types of the dataset increased. "Colour" had 22 less missing values, "Brand" had 41 less missing values, "Exact Colour" had 2.586 less missing values and "Title" had 4.627 less missing values.

Column	# Missing Pre	% Missing Pre	# Missing Post	% Missing Post	# Change	% Change
Type	14.080	~34.11%	14.080	~39.80%	0	0%
Colour	634	~1.54%	612	~1.73%	22	~3.47%
Brand	121	~0.29%	80	~0.23%	41	~33.88%
Exact Colour	19.892	~48.19%	17.306	~48.91%	2.586	~13.00%
Title	34.049	~82.48%	29.422	~83.16%	4.627	~13.59%

Table 10.4: Missing values before removing unwanted types and after

Since the "Exact Colour" column was only added for data cleaning purposes, we looked at the possibility that for a missing value in the "Colour" column, there might be a known value in the "Exact Colour" value.

As we can see in Fig. 10.7, the row number 96 had a missing value in "Colour" but a known value in the "Exact Colour" column. We filled in all the missing values in "Colour" with a known value in "Exact Colour".

```
data[data['Colour'].isna()]
```

	Type	Colour	Brand	Exact Colour	Title	Img name
17	NaN	NaN	M&S Kids	Pink	M&S Kids Hello Kitty Zipped Hoodie Pink Si...	HD_200025544_01.jpg?v=1
96	Cardigan	NaN	Unbranded Hand Knit	Lilac	Unbranded Hand Knit Cardigan Lilac Size: 2 - ...	HD_200016415_01.jpg?v=1
100	NaN	NaN	New Look	Blue	New Look Jacket Blue Size: 12 - 13 Years	HD_200015945_01.jpg?v=1
111	NaN	NaN	Monsoon	White	Monsoon Dress White Size: 8-9	HD_200015135_01.jpg?v=1
112	NaN	NaN	DK House	Pink	DK House Dress Pink Size: 90	HD_200015133_01.jpg?v=1
113	NaN	NaN	H&M	Black	H&M Jacket Black Size: 12-13Y	HD_200015118_01.jpg?v=1
169	NaN	NaN	M&S Marks & Spencer	red	M&S knitted dress red Size: 2 - 3 Years	HD_200010604_01.jpg?v=1

Figure 10.7: Subset of rows with missing values in the "Colour" column before filling in with "Exact Colour"

As we can see in Table 10.5, we had a total of 612 rows with a missing value in the "Colour" column. Filling in the missing values with known values in the "Exact Colour" lowered the missing values in the "Colour" column to 32, or a reduction of 580 rows or ~94.77%.

Column	# Pre Fill	# Post Fill	# Post Removal	# Change	% Change
Type	14.080	14.080	14.048	32	100%
Colour	612	32	0	32	100%
Brand	80	80	57	23	~71.88%
Exact Colour	17.306	17.306	17.274	32	100%
Title	29.422	29.422	29.421	1	~3.13%

Table 10.5: Missing values pre filling, missing post filling and how many rows removed

This left us with 32 missing values in the “Colour” column. Since both “Colour” and “Exact Colour” columns have missing values in these rows we dropped these rows from the dataset.

We now had 0 missing values in two columns, “Colour” and “Img name”. “Exact Colour” was of no use to us anymore and we dropped it when we had filled in missing values using the “Title” column.

The “Brand” column had 57 missing values. As we can see in Fig. 10.7, looking at the “Title” column, all these rows are some sort of an accessory, such as cuff links or earrings. We therefore dropped them.

```
data[data['Brand'].isna()]
```

	Type	Colour	Brand	Exact Colour	Title	Img name
2131	NaN	Grey	NaN	NaN	NaN	HD_200014675_01.jpg?v=1
2193	NaN	Silver	NaN	NaN	Sterling silver set of cuff links and tie pin	HD_200014145_01.jpg?v=1
3406	NaN	Metallics	NaN	NaN	2 Pairs of Cuff Links	HD_200008857_01.jpg?v=1
9263	NaN	Red	NaN	NaN	NWOT Marks & Spencer 2 Thin Leather Belts ...	HD_200028991_01.jpg?v=1
9271	NaN	Red	NaN	NaN	NWOT Marks & Spencer 2 Thin Leather Belts ...	HD_200028976_01.jpg?v=1
9429	NaN	Multi-coloured	NaN	NaN	New Poppy multicoloured rectangular scarf	HD_200028506_01.jpg?v=1
9941	NaN	Metallics	NaN	NaN	Silver Bracelet	HD_200027517_01.jpg?v=1
9942	NaN	Metallics	NaN	NaN	Silver engraved hinged bangle	HD_200027510_01.jpg?v=1
10077	NaN	Red	NaN	NaN	BURGUNDY KOKO WIG LONG BOB WITH FRINGE	HD_200027017_01.jpg?v=1
10078	NaN	Cream	NaN	NaN	BLONDE KOKO WIG LONG HIGHLIGHTED BOB WITH CENT...	HD_200027028_01.jpg?v=1
10888	NaN	Multi-coloured	NaN	NaN	Mackintosh inspired enamelled brooch	HD_200024750_01.jpg?v=1
11121	NaN	Metallics	NaN	Gold	Women's Gold Plated Pyramid Brooch	HD_200024053_01.jpg?v=1
11125	NaN	Cream / ivory	NaN	NaN	Women's Washer Drop Earrings	HD_200024042_01.jpg?v=1
11129	NaN	Multi-coloured	NaN	NaN	Women's Bead Hoop Earrings	HD_200024037_01.jpg?v=1
11130	NaN	Metallics	NaN	NaN	Women's Brass Chain Hoop Earrings	HD_200024026_01.jpg?v=1
11437	NaN	Multi-coloured	NaN	NaN	Pineapple Earrings	HD_200023127_01.jpg?v=1
11682	NaN	Black	NaN	NaN	Hat Pin 9cm Black End	HD_200022406_01.jpg?v=1
11741	NaN	Multi-coloured	NaN	NaN	Pretty Multi-coloured Wrap Bracelet	HD_200022301_01.jpg?v=1
11858	NaN	Metallics	NaN	NaN	Black and White Wrap Bracelet	HD_200022050_01.jpg?v=1
11964	NaN	Metallics	NaN	NaN	Silver bracelet	HD_200021715_01.jpg?v=1

Figure 10.7: Subset of rows with missing values in the “Brand” column

As we can see in Table 10.6, dropping these rows resulted in a reduction of 57 in the column “Type” which means that every missing value in “Brand” also had a missing value in “Type”. Also, we can notice that the “Exact Colour” dropped by 55, which means that there were 2 rows that had a value in the column.

Column	# Before	# After	# Change	% Change	% Known	% Missing
Type	14.048	13.991	57	100%	~60.36%	~39.64%
Colour	0	0	0	0%	100%	0%
Brand	57	0	57	100%	100%	0%
Exact Colour	17.274	17.219	55	~96.49%	~51.21%	~48.79%
Title	29.421	29.418	3	~5.26%	~16.64%	~83.36%

Table 10.6: Missing values before and after removing for “Brand”

After dropping these values, our dataset contained 35.291 rows and the “Type” column still had a missing value in 39.64% of the rows. The other columns that we planned on keeping in our final dataset all had 100% known values and therefore 0% missing values.

To fill in the remaining 13.991 missing values in the “Type” column we used the “Title” column. We eliminated the rows that both had a missing value in the “Type” and “Title” columns as we used the known value in the “Title” column to fill in the missing value in the “Type” column.

Doing this lowered down the missing values in the “Type” column to 2.350 or a reduction of 11.641, as we can see in Table 10.7.

Column	# Before	# After	# Change	% Change	% Known	% Missing
Type	13.991	2.350	11.641	100%	~90.06%	~9.94%
Exact Colour	17.219	8.939	8.280	71.13%	~62.20%	~37.80%
Title	29.418	17.777	11.641	100%	~24.83%	~75.17%

Table 10.7: Missing values before and after removing missing values in both “Type” and “Title”

The total rows were now 23.650 and as we can see in Fig. 10.8, we could look at the “Title” to fill in the missing values in the “Type” column.

```
data[data['Type'].isna()]
```

	Type	Colour	Brand	Exact Colour	Title	Img name
1	NaN	Pink	Patagonia	Pink and Yellow	Patagonia Rain Coat Pink and Yellow Size: S	HD_200022754_01.jpg?v=1
2	NaN	Multi-coloured	Sidewalk Sports	Black/Pink	Sidewalk Sports Wheeled Trainers Black/Pink Si...	HD_200027757_01.jpg?v=1
4	NaN	Grey	Alpinestars	NaN	Alpinestars Biker jacket blue Size: S	HD_200027000_01.jpg?v=1
5	NaN	Pink	Shredz	Pink	Shredz Ski Trousers Pink Size: 14 - 15 Years	HD_200027087_01.jpg?v=1
7	NaN	Purple	Trespass	Purple	Trespass Kids Ski Trousers Purple Size: 5 - 6 ...	HD_200026907_01.jpg?v=1
10	NaN	Red	Gap	Red	Gap Polka Dot Top Red Size: 12 - 13 Years	HD_200026539_01.jpg?v=1
11	NaN	Brown	Miss Evie	Brown	Miss Evie Faux Suede Jacket Brown Size: 12 - 1...	HD_200026403_01.jpg?v=1
12	NaN	Red	Other	NaN	BNWT Hudson's Bay Olympics Youth Red Mittens	HD_200025698_01.jpg?v=1
17	NaN	Pink	M&S Kids	Pink	M&S Kids Hello Kitty Zipped Hoodie Pink Si...	HD_200025544_01.jpg?v=1
19	NaN	Red	M&S Girls	Red	M&S Girls Cashmere Dress with Sequin Bow R...	HD_200024851_01.jpg?v=1
22	NaN	Brown	M&S Marks & Spencer	Brown	NWOT Marks & Spencer Suede Loafer Brown Si...	HD_200021338_01.jpg?v=1
23	NaN	Brown	M&S Marks & Spencer	Brown	NWOT Marks & Spencer Leather Lace Up Shoes...	HD_200021311_01.jpg?v=1
24	NaN	Cream	M&S Marks & Spencer	Cream	NWOT Marks & Spencer Bridesmaid / Party Sh...	HD_200021474_01.jpg?v=1
25	NaN	Cream	M&S Marks & Spencer	Cream	NWOT Marks & Spencer Bridesmaid/Party Shoe...	HD_200021442_01.jpg?v=1
26	NaN	Brown	M&S Marks & Spencer	Brown	NWOT Marks & Spencer Kids Slip on Shoes Br...	HD_200021268_01.jpg?v=1
27	NaN	Pink	Lelli Kelly	Pink	Lelli Kelly Shoes Pink Size: 8.5	HD_200021394_01.jpg?v=1
28	NaN	White	Little Miss Pink	White	Little Miss Pink Satin Party Shoes White Size: 1	HD_200023263_01.jpg?v=1
29	NaN	Pink	M&S Kids	Rose pink	M&S Kids Pumps Rose pink Size: 1	HD_200023320_01.jpg?v=1
30	NaN	Grey	Tu	Grey	Tu Trainers Grey Size: 1	HD_200023296_01.jpg?v=1
31	NaN	Black	Converse All star	Black	Converse All star Kids Shoes/Trainers Black Si...	HD_200023187_01.jpg?v=1
32	NaN	Grey	Vertbaudet	Charcoal grey	Vertbaudet Child's wool-mix lined coat Charco...	HD_200022982_01.jpg?v=1

Figure 10.8: Subset of rows with a missing value in the “Type” column

Since going through each of these 2.350 rows and manually finding which value should be in the “Type” column would be too time-consuming, we made a strategy to fill the missing values automatically.

Our approach was to make a list of the types we removed and a list of types we have, then loop through the “Title” column and either remove the row or fill in the missing value in the “Type” column by which value is found in the “Title” column.

As we can see in Fig. 10.9, our approach to remove rows that belonged to an unwanted type was to create a list of unwanted types that we knew that we did not want, then loop through these items and check if the “Title” column had a partial match with the item. Here we used the re module to ignore the case. If the “Title” contained any of these items, then we excluded it from the dataset.

```
subsetOfItemsToRemove = ['Shoe', 'Tie', 'Ski Trouser', 'Trainer', 'Scarf', 'Links', 'Mittens', 'Loafer', 'Slipper',
                         'Sandal', 'Bag', 'Bracelet', 'Belt', 'Boot', 'slip on', 'Watch', 'Pumps', 'glasses',
                         'Chain', 'suit', 'Salopettes', "Gloves", "Cap", "Gaiters", "Handkerchief", "Joggers",
                         "Brogues", "Pouch", "Heels", "Earrings", "Pyjamas", "Beige", "Poncho", "Purse", "Necklace",
                         "Mules", "Stilettoes", "Parka", "Gown", "Ring", "Hat", "Brooch", "Bra", "Uniform", "Heel"]

for item in subsetOfItemsToRemove:
    data = data[~data.Title.str.contains(item, na=False, flags=re.IGNORECASE)]
```

Figure 10.9: List of types to search in “Title” to remove from the dataset

Checking if the “Title” contained any of the types that we wanted to include in our dataset worked almost the same way. As we can see in Fig. 10.10, there were a few differences though.

```
lst = ["Evening dress", "Long dress", "Full length dress", "Overcoat", "Trenchcoat", "Long sleeved T-shirt","Leggings",
       "Gilet", "Raincoat", "Sweatshirt", "Waistcoat", "Full length dress", "Sleeveless", "Knee length dress", "Sweater",
       "Hoodie", "Smock", "Chinos", "T-Shirt", "Cardigan", "Trousers", "Pants", "Dress", "Skirt", "Shirt", "Vest", "Blazer",
       "Coat", "Top", "Short", "Jeans", "Blouse", "Jumper", "Jacket"]

dataCopy = data[data['Type'].isna()]

for item in lst:
    idxLst = dataCopy[dataCopy.Title.str.contains(item, na=False, flags=re.IGNORECASE)].index.values
    data.loc[idxLst, 'Type'] = item
```

Figure 10.10: List of types to search in “Title” to fill in missing values in “Type”

First, we only wanted to search the rows that had a missing value in the “Type” column. Then we stored a list of all rows that contained a partial match of the item so that we could change these missing values to the type it contained.

As we can see in Table 10.8, the method to remove rows lowered the amount of missing values by 908 rows and the method to fill in the missing value further lowered the amount by 1.386 rows, bringing the total to 2.294. This meant that our missing values in the “Type” column was only 56 or 0.25% of the dataset.

Column	# Before	# After Removal	# After Filling	# Change	% Change	% Known	% Missing
Type	2.350	1.442	56	2.294	~97.62%	99.75%	0.25%

Table 10.8: Missing values before and after removal and then after filling in missing values

By looking at the remaining 56 rows, we noticed that these either contained an accessory in the “Title” or it didn’t provide us with anything useful. We therefore decided to drop these rows from the dataset bringing the total rows in the dataset to 22.304 rows.

Since “Exact Colour” and “Title” were of no use to us anymore, we dropped those columns from the dataset.

10.1.3 Cleaning the “Colour” column

Since we already consolidated the “Type” column in an earlier step, our next step was to look at the “Colour” column and check if we could lower the amount of unique values in the “Colour” column.

Using describe() with pandas enabled us to get a quick overview of the statistics of the dataset. As we can see in Table 10.9, we had 164 unique colours with the Black colour being the most frequent with 4.745. 164 unique colours are much more than we needed so we took a better look at it.

Column	Count	Unique	Top	Freq
Type	22.304	59	Jacket	2.716
Colour	22.304	164	Black	4.745
Brand	22.304	6.339	M&S Marks & Spencer	2.364
Img name	22.304	22.239	HD_101852076_01.jpg?v=1	4

Table 10.9: Using describe() to get statistics about the dataset

Table 10.10 shows us the top 10 colours and the bottom 10 colours. Looking at the bottom 10 colours we can see that Black/White had a few colour tags although the same colour such as “black and white” and “Black/white”. Our research before indicated that the employees of Oxfam.org.uk probably had a free text choice when they input the tags which resulted in the tags being scattered around in many tags.

Top 10 colours		Bottom 10 colours	
Colour	Value	Colour	Value
Black	4.745	black and white	1
Blue	3.819	checked	1
Grey	2.333	Black Silver	1
Multi-coloured	2.114	Black & Ivory	1
Brown	1.500	navy	1
Pink	1.349	Black/white	1
Green	1.077	Camel	1
White	950	olive green	1
Red	895	Navy & White	1
Purple	887	Grey-Green	1

Table 10.10: Colours with highest 10 values and lowest 10 values

Our approach to lower the amount of unique colours was similar to the approach to lower the amount of types. We first removed the colours that we didn't want in the final dataset, such as "Wool", "Not specified", etc., as we can see in Fig. 10.11.

```
subsetOfItemsToRemove = ["striped ", "checked", "Leopard Print", "Not specified", "Tweed", "Wool", "Turquoise", "Colour",
                         "Olive", "Teal", "Peach", "Putty", "Khaki", "Stone", "Taupe", "Oyster", "Cerise", "Coral", "Aqua"]

for item in subsetOfItemsToRemove:
    data = data[~data.Colour.str.contains(item, na=False, flags=re.IGNORECASE)]
```

Figure 10.11: List of colours to search in "Colour" to remove from the dataset

Next step was to consolidate colours together that belonged together. As we can see in Fig. 10.12, an example of this would be that "Cream / ivory" belonged in the "Cream" colour and "/" belonged in the "Multi-coloured" colour.

```
colourLstBefore = ["Cream / ivory", "amp;", "/", "and", "Multi", "Mutli", "Silver", "Gold", "Bronze", "Metallic", "Black",
                   "Blue", "Grey", "Brown", "Pink", "Green", "White", "Cream", "Purple", "Red", "Lemon", "Navy", "Mustard",
                   "Denim", "Plum", "Bleu", "Indigo", "Lilac", "Burgundy", "Merlot", "Camel"]

colourLstAfter = ["Cream", "Multi-coloured", "Multi-coloured", "Multi-coloured", "Multi-coloured", "Multi-coloured",
                  "Metallics", "Metallics", "Metallics", "Metallics", "Black", "Blue", "Grey", "Brown", "Pink", "Green",
                  "White", "Cream", "Purple", "Red", "Green", "Blue", "Yellow", "Blue", "Purple", "Blue", "Blue", "Blue",
                  "Red", "Red", "Brown"]

for index, item in enumerate(colourLstBefore):
    rowIDs = data[data.Colour.str.contains(item, na=False, flags=re.IGNORECASE)].index.values
    data.loc[rowIDs, 'Colour'] = colourLstAfter[index]
```

Figure 10.12: List of colours to search in "Colour" to fill in missing values in "Colour"

We achieved this by finding a partial match and consolidated it in the colour that we had found that best matched with that colour.

As we can see in Table 10.11, the removal and consolidating colours together yielded a list of 14 unique colours. A reduction of 150 colours from the previous 164. The dataset now contained 20.145 rows.

Colour	Value	Colour	Value
Black	4.755	White	951
Blue	3.864	Purple	890
Grey	2.345	Beige	844
Brown	1.506	Cream	816
Pink	1.354	Orange	302
Green	1.082	Metallics	266
Red	978	Yellow	192

Table 10.11: Colours after consolidating

10.1.4 Cleaning the “Brand” column

The last column we wanted to clean before the dataset would be ready for training was the “Brand” column. We used the same approach as we did with the “Type” column and the “Colour” column, that is, lower the amount of unique brands significantly by consolidating brands together that belonged together.

Looking at Table 10.12, we can see that the “Brand” column contained a unique 5.855 brands or on average each brand contained ~3.44 unique rows which was too low for training a dataset.

Column	Count	Unique	Top	Freq
Type	20.145	59	Jacket	2.633
Colour	20.145	14	Black	4.755
Brand	20.145	5.855	M&S Marks & Spencer	2.208
Img name	20.145	20.082	HD_101852076_01.jpg?v=1	4

Table 10.12: Using describe() to get statistics about the dataset

Top 10 brands		Bottom 10 brands	
Brand	Value	Brand	Value
M&S Marks & Spencer	2.208	Grenfell	1
Next	455	Blue Star	1
Per Una	364	Butler & Wilson	1
Unbranded	346	Camanchi Leathers London	1
Ted Baker	303	Julian Taylor	1
Monsoon	257	McKenzy outerwear	1
ASOS	247	CONRAD	1
Jaeger	222	Bravissimo	1
Coast	208	Holmewood	1
Laura Ashley	199	Madeleine Press	1

Table 10.13: Brands with highest 10 values and lowest 10 values

With this in mind, we printed all the brands with the total amount and loaded the data into Microsoft Excel. We then sorted the data alphabetically and looked for all the brands that could be consolidated into a single brand. We ignored brands that would only tally up to 9 since the model would need a substantial amount of brands to accurately predict on a brand

As we can see in Fig. 10.3, we found some brands that our approach would match on a partial match with a different brand, so we decided to consolidate these brands into the “Unknown” brand. All of these brands had under 10 total rows so they would have been consolidated together later on.

```
brandChangeToUnknown = ["AZARA", "Velvet by Graham &amp;amp; Spencer", "East Coast", "Green Coast", "North Coast",
                        "Tadashi Collection", "Kardashian Kollection for Lipsy", "Dolce Vita", "Big Star", "Arrogant",
                        "Handmade", "Jacques Britt", "Jacques Sac", "Mylene Klass", "Oaklands", "Rowlands", "Urban Landscape",
                        "Larry Levine", "Miss levi", "Tahari Arthur S.Levine", "Tahari, Arthur S Levine", "Linea Raffaelli",
                        "Linea Tesini", "axcess (a Claiborne company)", "Kim &amp;amp; Co", "Love Mango Basic", "Monsoon Fusion",
                        "Kiwistuff", "Vilagallo", "Vila Joy", "J. Toyner Eromann", "Roman Carter"]

for item in brandChangeToUnknown:
    rowIDs = data[data.Brand.str.contains(item, na=False, flags=re.IGNORECASE)].index.values
    data.loc[rowIDs, 'Brand'] = "Unknown"
```

Figure 10.13: Consolidating some brands into the same “Unknown” brand

Our next step was to find brands that could be consolidated into a single brand. We only looked at brands that would tally up to at least 10 after consolidation.

As we can see in Fig. 10.14, we made a list of all the brands that we could consolidate into a single brand as we can see in Fig. 10.15. For example, “M&S”, “M & amp; S”, “M and S” and “Marks & Spencer” could all be consolidated into the brand “Marks & Spencer”. The method is shown in Fig. 10.16.

```
brandLstBefore = ["M&amp;amp;S", "M &amp;amp; S", "M and S", "Marks &amp;amp; Spencer", "Unbranded", "H&amp;amp;M", "Armani", "Calvin",
                  "Laura Ashley", "Jack &amp;amp; Jones", "Marks and Spencer", "Abercombie", "Abercrombie", "Abercrombe", "Zara",
                  "Dominguez", "Adolfo", "Adidas", "All Saints", "allsaints", "Apricot", "Aquascutum", "ASOS", "Atmosphere",
                  "Austin Reed", "Autograph", "AX Paris", "AX (Paris)", "Ax, Paris", "Banana", "Barbour", "Ben Sherman",
                  "Bench", "Benetton", "Betty Barclay", "Betty Jackson", "BHS", "Biba", "Boden", "Boohoo", "Brook Taverner",
                  "Burberry", "Burton", "Charles Tyrwhitt", "Coast", "Country Casuals", "Cotton Traders", "Daks", "Damart",
                  "Daniel Hechter", "Daniel Hetcher", "Dannimac", "Dash", "Debenhams", "Debut", "Début", "Furstenberg",
                  "Diane Von", "Diesel", "DKNY", "Karan", "Gabbana", "Dolce", "Dorothy Perkins", "Eastex", "Edina Ronay",
                  "Edinburgh", "Escada", "Evans", "EWM", "F&amp;amp;F", "F &amp;amp; F", "F + F", "F+F", "Fat Face", "FatFace",
                  "Fenn", "Fire Trap", "Forever 21", "Frank Usher", "French Connection", "G Star", "G-Star", "Gant", "Darel",
```

Figure 10.14: Subset of brands that we found that could be consolidated into another brand

```
brandLstAfter = ["Marks & Spencer", "Marks & Spencer", "Marks & Spencer", "Marks & Spencer", "Unknown", "H&M", "Armani",
                 "Calvin Klein", "Laura Ashley", "Jack & Jones", "Marks & Spencer", "Abercrombie & Fitch",
                 "Abercrombie & Fitch", "Abercrombie & Fitch", "Zara", "Adolfo Dominguez", "Adolfo Dominguez", "Adidas",
                 "All Saints", "All Saints", "Apricot", "Aquascutum", "ASOS", "Atmosphere", "Austin Reed", "Autograph",
                 "AX Paris", "AX Paris", "AX Paris", "Banana Republic", "Barbour", "Ben Sherman", "Bench", "Benetton",
                 "Betty Barclay", "Betty Jackson", "BHS", "Biba", "Boden", "Boohoo", "Brook Taverner", "Burberry", "Burton",
                 "Charles Tyrwhitt", "Coast", "Country Casuals", "Cotton Traders", "Daks", "Damart", "Daniel Hechter",
                 "Daniel Hechter", "Dannimac", "Dash", "Debenhams", "Debut", "Début", "Diane Von Furstenberg",
                 "Diane Von Furstenberg", "Diesel", "DKNY", "DKNY", "Dolce & Gabbana", "Dolce & Gabbana", "Dorothy Perkins",
                 "Eastex", "Edina Ronay", "Edinburgh Woollen Mill", "Escada", "Evans", "EWM", "F&F", "F&F", "F&F", "F&F",
```

Figure 10.15: Subset of brands that the brands from Fig. 10.14 would be consolidated into

```
for index, item in enumerate(brandLstBefore):
    rowIDs = data[data.Brand.str.contains(item, na=False, flags=re.IGNORECASE)].index.values
    data.loc[rowIDs, 'Brand'] = brandLstAfter[index]
```

Figure 10.16: Approach to consolidate brands together

When we looked at brands that we could potentially consolidate, we noticed a brand “TU” that we needed to do manually as the partial match of “TU” would consolidate a lot of unrelated brands together. We took the three unique values for the brand, “TU”, “TU ” and “Tu” and consolidated them into the “TU” brand.

The next step was to consolidate all the brands that had a total number of rows less than 20 into the “Unknown” brand. This was done because the model would be able to make a better prediction on a brand if it had a higher amount of total images to train on. The number 20 was used, but it might need to be substantially higher if the model doesn’t predict well on the brand.

Our approach for this was that we made a dictionary with all the brands along with the total number of rows each brand had. The key in the dictionary was the brand itself and the value was the number of total rows. We then looped through the dictionary and consolidated any brand that had less than 20 total rows into the brand “Unknown”.

As we can see in Table 10.14, the “Unknown” brand had a significant amount of rows or 8.041. This might be a problem when we train the model and we might need a better solution to predict on a brand if the model will not be able to predict accurately on brands.

This method managed to bring the unique values of the brand column down to 129, a reduction of 5.726 or ~97.80%. Since we didn’t remove any rows, but rather consolidated them into the “Unknown” brand, our total rows in the dataset is still 20.145.

Top 10 brands		Bottom 10 brands	
Brand	Value	Brand	Value
Unknown	8.041	Rohan	21
Marks & Spencer	2.441	Mint Velvet	21
Next	475	Apricot	21
Per Una	375	Adidas	21
Ted Baker	312	G Star	20
ASOS	299	Prada	20
Monsoon	268	Lakeland	20
Jaeger	231	Dannimac	20
Coast	214	Brook Taverner	20
Zara	212	Alex & Co	20

Table 10.14: Brands with highest 10 values and lowest 10 values

10.1.5 Final touches

Before the dataset was ready for Google AutoML, there were a few things that we needed to address. First, the “Img name” column contained a few duplicates. As we can see in Table 10.15, the total rows in the dataset was 20.145 but the unique values in the “Img name” column was 20.082, which leads to the conclusion that there were 63 duplicate values in the “Img name” column.

Column	Count	Unique	Top	Freq	% Freq
Type	20.145	59	Jacket	2.633	~13.07%
Colour	20.145	14	Black	4.755	~23.60%
Brand	20.145	129	Unknown	8.041	~39.92%
Img name	20.145	20.082	HD_101852076_01.jpg?v=1	4	~0.02%

Table 10.15: Using describe() to get statistics about the dataset

We also knew that, from Chapter 9.1, the scraper managed to download 33.900 images but had 41.281 rows in the dataset. This meant that 7.381 rows were missing a downloaded image. The odds that our data cleaning eliminated these rows from the dataset were extremely low, so we needed to compare the images that we have downloaded and the image names that we have in the dataset.

Our plan to address these issues was to export the cleaned dataset as a .csv file. Then open the .csv file in excel and extract the image names. Delete any image that was not within the extracted list of image names and then create a .txt document that contained all image names that were left and compare them to the image names contained within the .csv document.

As we can see in Fig. 10.17, the raw .csv document contained an ID row that we would not need so we removed the ID row and we also removed the trailing “?v=1” text in the image name. As we can see in Fig. 10.18, the “Img name” now contained the image names that we were interested in extracting and using to remove any images that were not contained within that list.

	A	B	C	D	E	F
1	,Type,Colour,Brand,Img name					
2	0,Cardigan,Grey,Boden,HD_200028434_01.jpg?v=1					
3	1,Coat,Pink,Unknown,HD_200022754_01.jpg?v=1					
4	4,Jacket,Grey,Unknown,HD_200027000_01.jpg?v=1					
5	8,Full length dress,White,Unknown,HD_200026637_01.jpg?v=1					
6	10,Top,Red,Gap,HD_200026539_01.jpg?v=1					
7	11,Jacket,Brown,Unknown,HD_200026403_01.jpg?v=1					
8	15,Jacket,Pink,Barbour,HD_200025532_01.jpg?v=1					
9	17,Hoodie,Pink,Marks & Spencer,HD_200025544_01.jpg?v=1					
10	19,Dress,Red,Marks & Spencer,HD_200024851_01.jpg?v=1					

Figure 10.17: .csv document before

	A	B	C	D	E
1	Type	Colour	Brand	Img name	
2	Cardigan	Grey	Boden	HD_200028434_01.jpg	
3	Coat	Pink	Unknown	HD_200022754_01.jpg	
4	Jacket	Grey	Unknown	HD_200027000_01.jpg	
5	Full length	White	Unknown	HD_200026637_01.jpg	
6	Top	Red	Gap	HD_200026539_01.jpg	
7	Jacket	Brown	Unknown	HD_200026403_01.jpg	
8	Jacket	Pink	Barbour	HD_200025532_01.jpg	
9	Hoodie	Pink	Marks & S	HD_200025544_01.jpg	
10	Dress	Red	Marks & S	HD_200024851_01.jpg	

Figure 10.18: .csv document after

After we extracted the image names to a .txt document called “removeImages.txt”, we needed a way to remove all the images that were not contained within that document. Our approach can be seen in Fig. 10.19, which was importing all image names into a set, which only kept unique values, and if the file name was not within that set, it would be removed using the os module in Python.

```
keep = set()

with open('./removeImages.txt') as f:
     for line in f:
         keep.add(line.strip())

for f in os.listdir('.'):
    if f not in keep:
        os.unlink(f)
```

Figure 10.19: removeImages.py file to remove images not in the dataset

This approach resulted in 16.543 images being left in our images folder. We however had 20.082 unique image names in our dataset, which meant that 3.539 rows in the dataset didn’t have an image attached.

Our next step was to extract all image names within the folder with the images that were left and compare them to the .csv document and remove both any row that didn’t have an image and any duplicate rows.

To get a better understanding on what rows to remove, we used the conditional formatting tool in Excel to check which rows were a duplicate with the images we had in the folder. We then sorted the rows by the cell colour as we can see in Fig. 10.20. This allowed us to quickly remove the rows that didn’t have the cell colour that indicated that the row was a duplicate with an image in the folder.

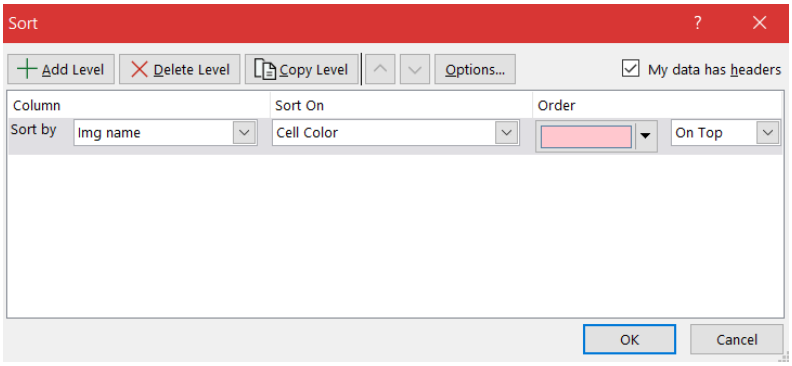


Figure 10.20: Sorting by Cell Colour in Excel

Once we removed all rows that were either a duplicate or didn’t have an image, we were left with 16.543 rows in the dataset. We then made the document ready so that we could import it back into Jupyter.

The Excel document that we used for this purpose can be seen in Fig. 10.21, although it shows the state before we sorted based on cell color.

	A	B	C	D	E	F	G	H	I
1	Type	Colour	Brand	Img name				In folder	
2	Cardigan	Grey	Boden	HD_200028434_01.jpg				HD_100337355_01.jpg	
3	Coat	Pink	Unknown	HD_200022754_01.jpg				HD_100403601_01.jpg	
4	Jacket	Grey	Unknown	HD_200027000_01.jpg				HD_100424462_01.jpg	
5	Full length	White	Unknown	HD_200026637_01.jpg				HD_100443530_01.jpg	
6	Top	Red	Gap	HD_200026539_01.jpg				HD_100452597_01.jpg	
7	Jacket	Brown	Unknown	HD_200026403_01.jpg				HD_100453860_01.jpg	
8	Jacket	Pink	Barbour	HD_200025532_01.jpg				HD_100453881_01.jpg	
9	Hoodie	Pink	Marks & S	HD_200025544_01.jpg				HD_100462918_01.jpg	
10	Dress	Red	Marks & S	HD_200024851_01.jpg				HD_100463822_01.jpg	
11	Jacket	White	Next	HD_101887084_01.jpg				HD_100497065_01.jpg	
12	Coat	Grey	Unknown	HD_200022982_01.jpg				HD_100506878_01.jpg	
13	Long dress	White	Unknown	HD_200019358_01.jpg				HD_100506981_01.jpg	
14	Calf length	Pink	Unknown	HD_200019380_01.jpg				HD_100507752_01.jpg	
15	Sleeveless	Pink	Unknown	HD_200019561_01.jpg				HD_100515248_01.jpg	
16	Jacket	Black	Unknown	HD_200021719_01.jpg				HD_100519241_01.jpg	
17	Jacket	Blue	Barbour	HD_200016887_01.jpg				HD_100537597_01.jpg	
18	Jeans	Metallics	Unknown	HD_200020285_01.jpg				HD_100541079_01.jpg	
19	Dress	Cream	Ted Baker	HD_101884132_01.jpg				HD_100544009_01.jpg	
20	Jeans	Blue	Levi's	HD_101881289_01.jpg				HD_100552383_01.jpg	
21	Cardigan	Purple	Lands' End	HD_200018993_01.jpg				HD_100570399_01.jpg	
22	Sleeveless	Beige	Monsoon	HD_200018778_01.jpg				HD_100573773_01.jpg	
23	Knee lengt	Blue	Marks & S	HD_200016389_01.jpg				HD_100573857_01.jpg	
24	Long sleev	Blue	Laura Ashl	HD_101882177_01.jpg				HD_100575150_01.jpg	
25	Coat	Green	River Islan	HD_200017862_01.jpg				HD_100575643_01.jpg	
26	Coat	Grey	Unknown	HD_200017963_01.jpg				HD_100577697_01.jpg	
27	Jacket	Black	Barbour	HD_101879730_01.jpg				HD_100585054_01.jpg	

Figure 10.21: Comparing “Img name” to image names in folder

As we can see in Table 10.16, our dataset now contained 16.543 unique rows. However, since we removed a significant amount of rows, we needed to check the “Type”, “Brand” and “Colour” columns again.

We can also see that the “Brand” lost 3 unique values, the “Unknown” brand and the “Black” colour both increased its frequency percentage on the dataset but the “Jacket” type had a lower frequency percentage.

Column	Count	Unique	Top	Freq	% Freq
Type	16.543	59	Jacket	2.069	~12.51%
Colour	16.543	14	Black	4.039	~24.42%
Brand	16.543	126	Unknown	7.598	~45.93%
Img name	16.543	16.543	HD_101441949_01.jpg	1	~0.01%

Table 10.16: Using describe() to get statistics about the dataset

As we can see in Table 10.17, “Type” now had 5 types that were less than 20 and no type had less than 10. We kept the column as it was. The colours were all significantly above the lowest number in both the “Type” and the “Brand” columns so we kept it intact.

However, the “Brand” column had some significant changes. There were now 7 brands that were beneath the 20 value threshold and we consolidated them into the “Unknown” brand. We can also see that the brand “Marks & Spencer” now only had 12 rows, a significant drop from the 2.441 rows it had before.

Bottom 10 types		Bottom 10 colours		Bottom 10 brands	
Type	Value	Colour	Value	Brand	Value
Pants	39	Pink	1.151	Lakeland	20
Chinos	28	Green	923	Dannimac	20
Jeggings	27	Red	796	Adidas	20
T-shirt	26	Purple	788	Prada	19
Cargo pants	26	Beige	697	Brook Taverner	19
Trenchcoat	18	Cream	662	Apricot	18
Cargo shorts	17	White	651	Marks & Spencer	12
Gilet	17	Metallics	230	Dolce & Gabbana	8
Leggings	16	Orange	205	Abercrombie & Fitch	8
Long sleeved T-shirt	10	Yellow	165	Alex & Co	4

Table 10.17: Columns with lowest 10 values

As we can see in Table 10.18, the “Brand” column now had 119 unique values instead of 126 and the frequency percentage of the “Unknown” brand had increased a little bit to ~46.46% of the dataset, compared to the ~45.93% it had before.

Column	Count	Unique	Top	Freq	% Freq
Type	16.543	59	Jacket	2.069	~12.51%
Colour	16.543	14	Black	4.039	~24.42%
Brand	16.543	119	Unknown	7.686	~46.46%
Img name	16.543	16.543	HD_101441949_01.jpg	1	~0.01%

Table 10.18: Using describe() to get statistics about the dataset

As we can see in Fig. 10.22, the type countplot shows the distribution between the types was not ideal and that we might need to address this if the model does not predict accurately.

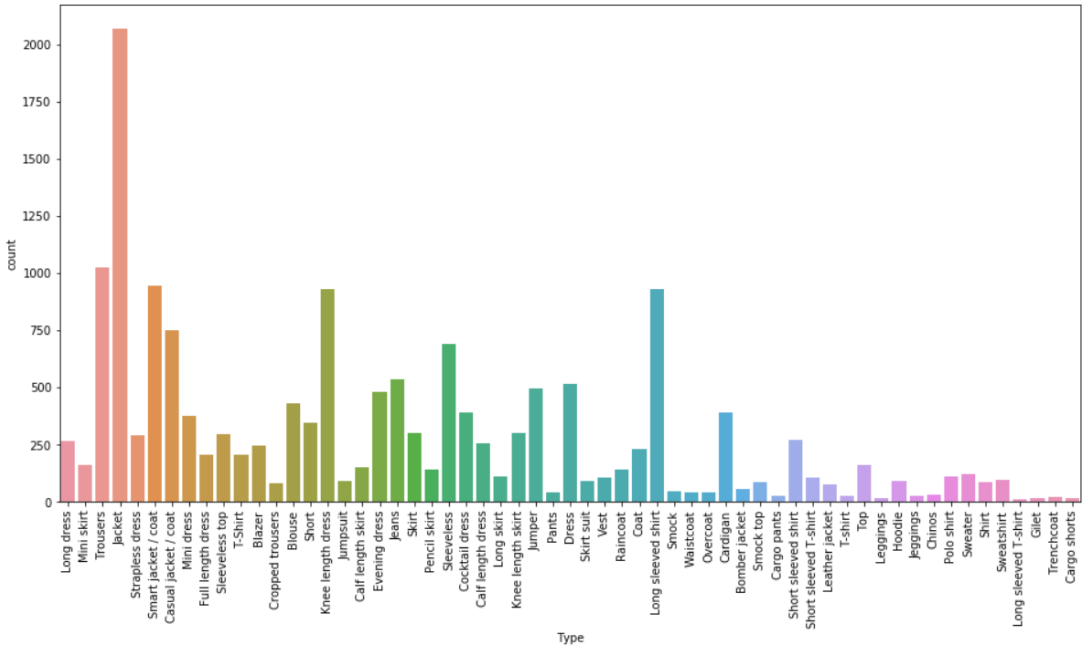


Figure 10.22: Count plot for the Type column

Same with colour as we can see in Fig. 10.23, where “Black” and “Blue” had a significant more amount than the other colours.

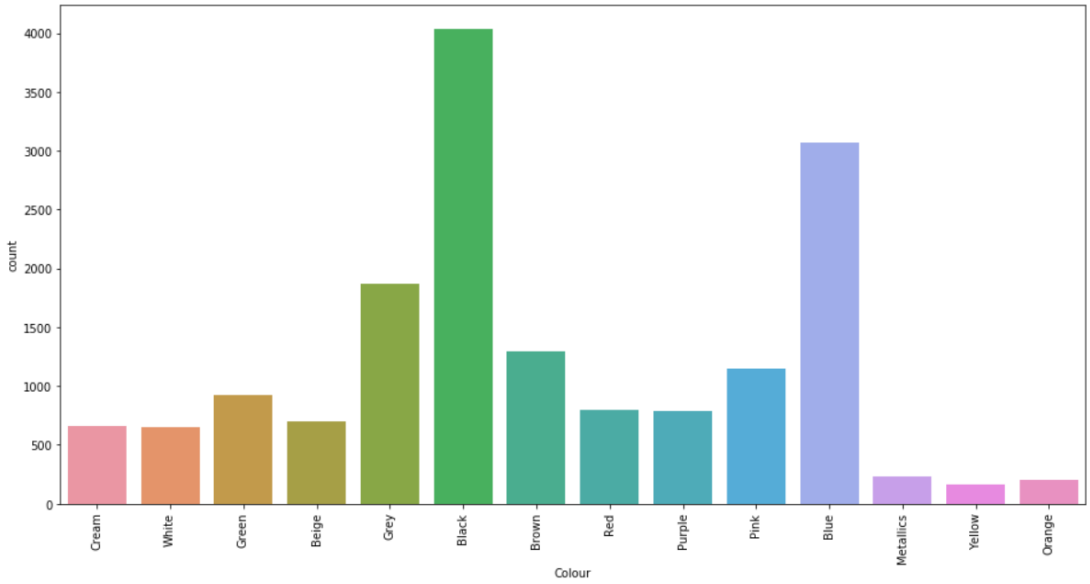


Figure 10.23: Count plot for the Colour column

As we can see in Fig. 10.24, the “Unknown” brand had much more rows than the second highest one, “Next”. But to get a better feeling of the distribution between the known brands, we have to exclude “Unknown”.

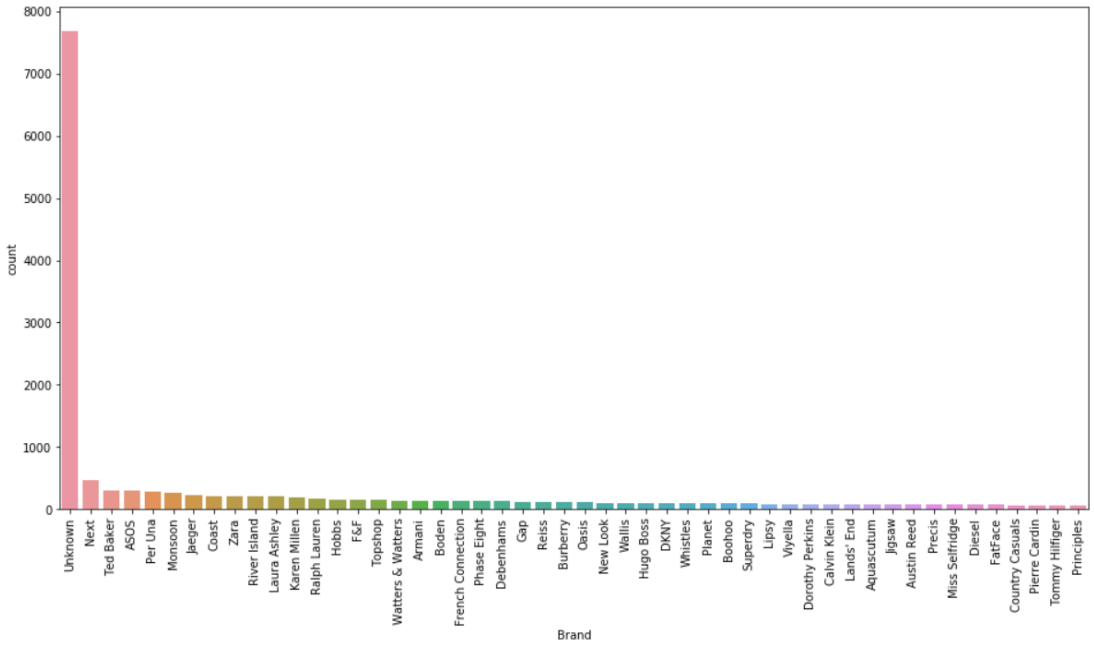


Figure 10.24: Count plot for the Brand column

As we can see in Fig. 10.25, excluding the “Unknown” brand shows a much more even distribution.

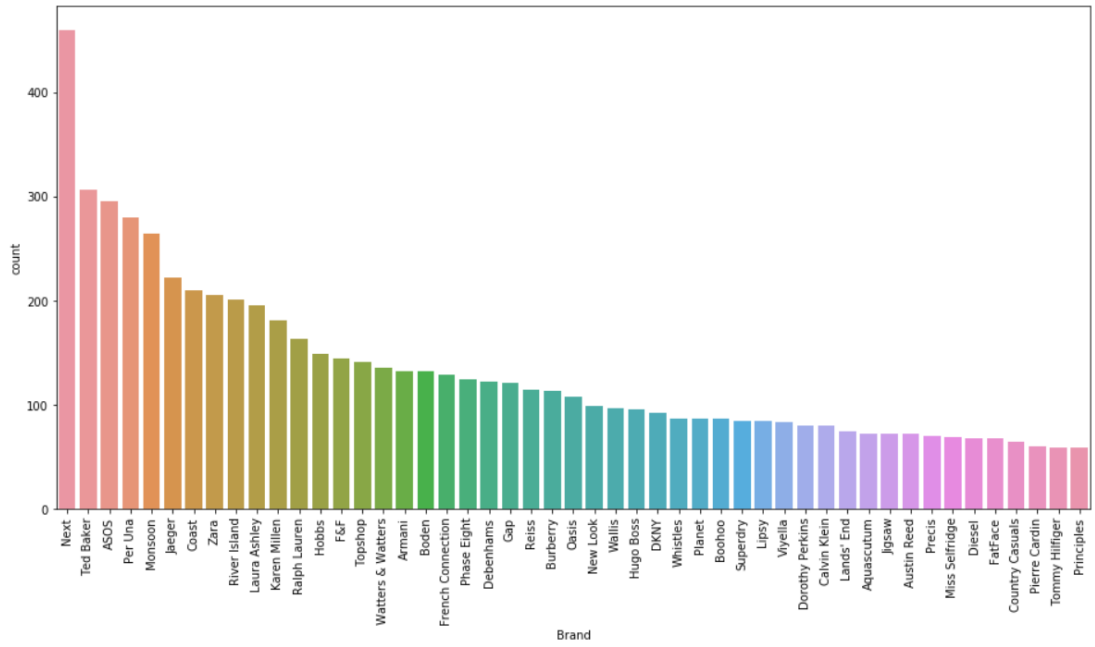


Figure 10.25: Count plot of the Brand column, excluding the “Unknown” brand

10.2 Cleaning data from Netflea.com

To explore and clean the data from Netflea.com, we used the same tools and approach as we used for the Oxfam dataset, that is, pandas to convert the .csv document to a pandas dataframe.

First we started by printing out the shape of the data, but the shape tells us how many rows there are in the dataset and how many columns. Then we printed out 10 sample rows from the dataset to check out the dataset and see what we should expect in the dataset.

As we can see in Fig. 10.26, our dataset contained 86.900 rows and 4 columns. The columns were “Type”, “Brand”, “Colour” and “ImgName”.

Further, a random listing of 10 rows showed a couple of missing values, 1 missing value in “Type”, 1 missing value in “Brand” and 2 missing values in the “Colour” column.

```
data.shape
```

```
(86900, 4)
```

```
data.sample(10)
```

	Type	Brand	Colour	ImgName
7979	T-shirt	H&M	White	https://www.netflea.com/media/catalog/product/...
77500	Socks set	H&M	Blue	https://www.netflea.com/media/catalog/product/...
22600	Sweater	Vero Moda	Green	https://www.netflea.com/media/catalog/product/...
29486	Long-sleeved shirt blouse	L.O.G.G.	Dark gray	https://www.netflea.com/media/catalog/product/...
15492	Jacket coat	NaN	Gray	https://www.netflea.com/media/catalog/product/...
29296	NaN	Kappahl	NaN	https://www.netflea.com/media/catalog/product/...
43350	Skirt	Vero Moda	Black	https://www.netflea.com/media/catalog/product/...
41440	Straight pants	Share	NaN	https://www.netflea.com/media/catalog/product/...
75264	Pants	Hampton Republic	Red	https://www.netflea.com/media/catalog/product/...
66627	Cotton shirt	Name it	Gray	https://www.netflea.com/media/catalog/product/...

Figure 10.26: Shape of dataset and random 10 rows from the dataset

We can also see that the “ImgName” column had the full url as the “ImgName”. We needed to remove the url and keep the image file name so that we could link the images to the rows. We didn’t have two columns, “Img src” and “Img name” as we did with Oxfam.org.uk, as we thought it would be redundant as we can get the image name from the image url.

As we can see in Fig. 10.27, there were a significant amount of missing values in the “Brand” column and in the “Colour” column. We didn’t have any extra columns that we could potentially use to fill in the missing values, so these rows needed to be dropped.

```
data.isna().sum()
```

```
Type         5379
Brand       13011
Colour      22529
ImgName       245
dtype: int64
```

Figure 10.27: listing the sum of missing values in each column

In Table 10.18, we can see how many missing values these attributes had as a percentage of the dataset. We can see that “Colour” had the most missing values or ~25.93%. As we mentioned previously, the dataset contained 86.900 rows, dropping ~25.93% of them would yield 64.371 rows left in the dataset.

Column	# Missing	% Missing	% Known
Type	5.379	~6.19%	~93.81%
Brand	13.011	~14.97%	~85.03%
Colour	22.529	~25.93%	~74.07%
ImgName	245	~0.28%	~99.72%

Table 10.18: Percentage of missing and known values

All the columns had a missing value, the “Type” column had ~93.81% known values, the “Brand” column had ~85.03% known values, the “Colour” column had ~74.07% known values and the “ImgName” column had the most known values or ~99.72%.

10.2.1 Removing missing values in “ImgName” column

First thing we needed to do was to drop the rows that had a missing value in the “ImgName” column, this was done because we needed an image to be able to train the dataset.

As we can see in Fig. 10.28, almost all rows in this subset contained a missing value in all rows. Removing all rows based on having a missing value in all columns resulted in removing 242 rows. The three remaining rows were then dropped as they didn’t have an image attached to the data, thus rendering the data useless for training.

```
data[data['ImgName'].isna()]
```

	Type	Brand	Colour	ImgName
31513	Cardigan	Vero Moda	Light beige	NaN
32324	NaN	NaN	NaN	NaN
32325	NaN	NaN	NaN	NaN
51707	Skirt	H&M	Black	NaN
60433	Pants	Bula	Dark blue	NaN
76161	NaN	NaN	NaN	NaN
76162	NaN	NaN	NaN	NaN
76171	NaN	NaN	NaN	NaN
76172	NaN	NaN	NaN	NaN
76173	NaN	NaN	NaN	NaN
76174	NaN	NaN	NaN	NaN
76175	NaN	NaN	NaN	NaN
76176	NaN	NaN	NaN	NaN
76177	NaN	NaN	NaN	NaN
76178	NaN	NaN	NaN	NaN
76179	NaN	NaN	NaN	NaN
76180	NaN	NaN	NaN	NaN
76181	NaN	NaN	NaN	NaN

Figure 10.28: Subset of rows where there is a missing value in the “ImgName” column

This resulted in the dataset now having 86.655 rows and as we can see in Table 10.19, it lowered the missing values in all columns except for “ImgName” by 242, and 245 in “ImgName”.

Column	# Before	# After	Difference
Type	5.379	5.137	242
Brand	13.011	12.769	242
Colour	22.529	22.287	242
ImgName	245	0	245

Table 10.19: Number of missing values before and after

Since we only needed the image name, which was the part after the last slash, we removed everything before the last slash, along with the last slash.

Our method for doing this can be seen in Fig. 10.29.

```
dataSample = data.sample(1)
sampleIndex = dataSample.index.values.astype(int)[0]

print("Sample row to check: ", sampleIndex)

print("Sample ImgName column before: " + data.ImgName.iloc[sampleIndex])

data.ImgName = data.ImgName.map(lambda x: x.split("/")[-1])

print("Sample ImgName column after: " + data.ImgName.iloc[sampleIndex])
```

```
Sample row to check:  41663
Sample ImgName column before: https://www.netflea.com/media/catalog/product/cache/2/image/750x750/9df78eab33525d08d6e5fb8d27136e95/1/5/15712041693528385201691571204278.jpg
Sample ImgName column after: 15712041693528385201691571204278.jpg
```

Figure 10.29: Fixing the "ImgName" column to only show the image name

First we took a sample row, in this case row #41.663 to check if we were keeping the image name intact or if we were manipulating the image name to something else. Then we took each row and split the url by slashes and renamed the image name to the last item in the splitted url which was the image name.

10.2.2 Removing missing values in “Type” column

As we can see in a subset in Fig. 10.30, all rows with a missing value in the “Type” column also had a missing value in the “Colour” column. We checked the whole set and noticed that for each missing value in the “Type” column there was a missing value in the “Colour” column. Based on this information, we decided to drop all rows that contained a missing value in the “Type” column.

```
data[data['Type'].isna()]
```

	Type	Brand	Colour	ImgName
38	NaN	Vila	NaN	1546763514145590141993657491004515467636 43.jpg
39	NaN	Gina Tricot	NaN	1546198966175280806072222770299415461990 47.jpg
40	NaN	Object	NaN	1546195455883567525735697522176615461956 37.jpg
679	NaN	Other Brand	NaN	vvvvvvvvvv0041544208382.JPG
1287	NaN	Other Brand	NaN	DSC006731544459189.JPG
7179	NaN	Mexx	NaN	DSC080991542482688.JPG
7588	NaN	JSFN	NaN	DSC_03231487191482.JPG
7589	NaN	Kappahl	NaN	DSC_00561487275815.JPG
7598	NaN	Adidas	NaN	688F6C39-348D-47BB-978F-B73FDCAAE21D1540849747...
7611	NaN	-	NaN	image1535008456.jpg
7612	NaN	Sisters Point	NaN	image1534321391.jpg
7613	NaN	Seppälä	NaN	image1534320554.jpg
7614	NaN	Gina Tricot	NaN	image1534319166.jpg

Figure 10.30: Subset of rows with missing values in the “Type” column

This resulted in the dataset now containing 81.518 rows, a reduction of 5.137 rows from 86.655. As we can see in Table 10.19, both “Type” and “Colour” reduced the missing values by 5.137, as expected, however “Brand” had the same amount of missing values which shows that for each row that we dropped, there was a known value in the “Brand” column.

Column	# Before	# After	Difference
Type	5.137	0	5.137
Brand	12.769	12.769	0
Colour	22.287	17.150	5.137

Table 10.19: Number of missing values before and after

We then consolidated the missing values in the “Colour” and “Brand” columns into another brand or colour as we were most interested in getting the type and image name.

10.2.3 Consolidating types together in “Type” column

Once we removed all missing values in the “Type” column, we wanted to finish cleaning it by consolidating types together that belonged together. As we can see in Table 10.20, the “Type” column contained 1.014 unique types with “Shirt” the most frequent one with 5.209.

Column	Count	Unique	Top	Freq	Freq %
Type	81.518	1.014	Shirt	5.209	~6.39%

Table 10.20: Using describe() to get statistics about “Type”

When we were looking to consolidate types together, like we did with Oxfam.org.uk, we noticed that there were no misspellings, as we can see in Table 10.21, which leads to the conclusion that the product was added with predefined labels. With this in mind, we decided to drop all types that had less than 50 rows.

Top 10 types		Bottom 10 types	
Type	Value	Type	Value
Shirt	5.209	Hiking pants	1
T-shirt	4.063	Football	1
Jeans	3.787	Decoration	1
Dress	3.426	Protection	1
Sweater	2.739	Lined rain pants	1
Top	2.712	Toddler stockings	1
Cardigan	2.672	Yoga shirt	1
Pants	2.578	Sports equipment	1
Knitting	2.076	Chalk stripe suit	1
Sweatshirt	1.743	Winter running tights	1

Table 10.21: Types with highest 10 values and lowest 10 values

As we can see in Table 10.22, our dataset now contained 72.743 rows, a reduction of 8.775 from 81.518 and the “Type” column contained 224 unique types, a reduction of 790 from 1.014.

Column	Count	Unique	Top	Freq	Freq %
Type	72.743	224	Shirt	5.209	~7.16%

Table 10.22: Using describe() to get statistics about “Type” after removing

We can visualize the “Type” column by looking at Fig. 10.31 and Fig. 10.32.

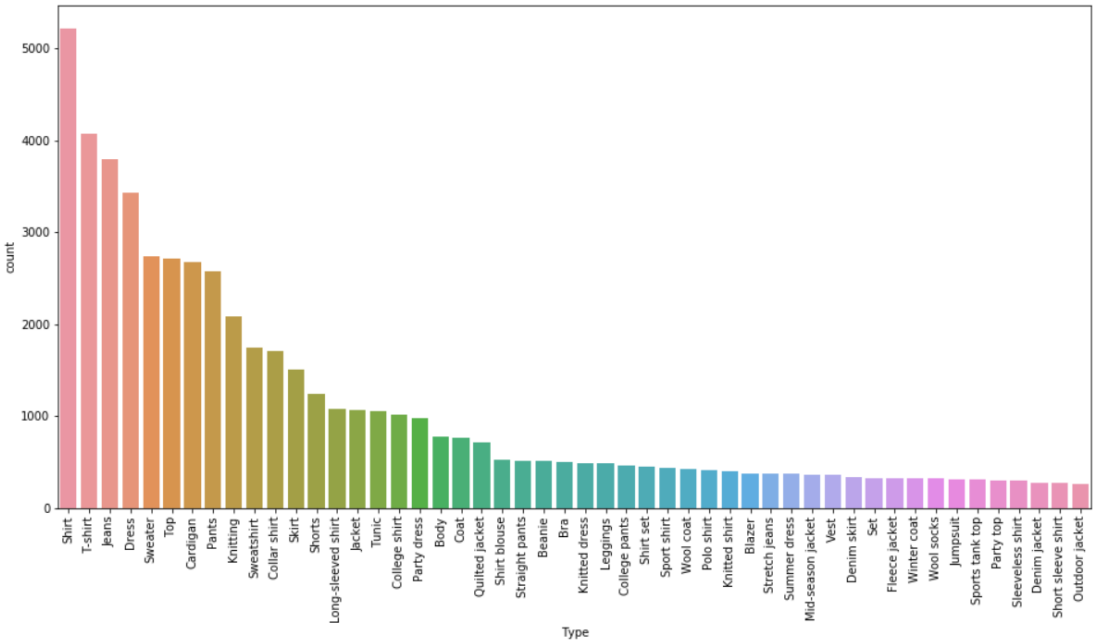


Figure 10.31: Top 50 types

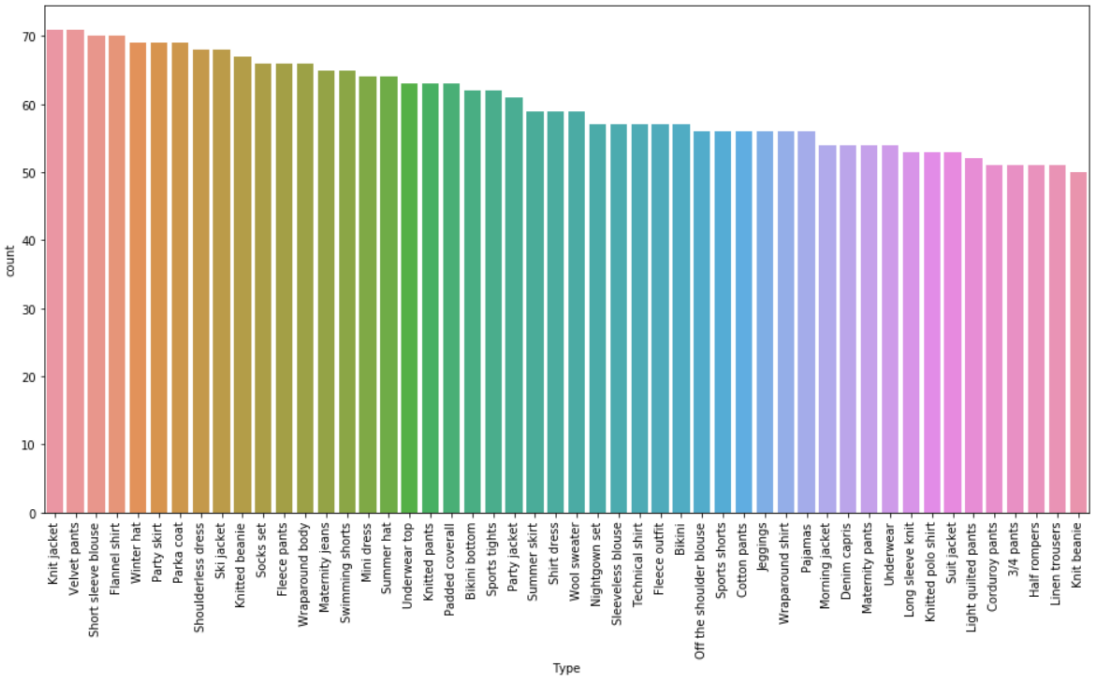


Figure 10.32: Bottom 50 types

10.2.4 Consolidating brands together in the “Brand” column

As we can see in Table 10.23, there were now 11.327 missing values in the “Brand” column, but they had a known image and a known type, so we wanted to keep these rows so that the model could more accurately predict on types, which Google Vision AI had trouble recognizing. The known values were 61.416 and the “Brand” column had 1.656 unique brands, which gave an average of ~37.09 known values for each brand. The most common brand was H&M with ~15.14% of the known values in the dataset.

Column	Count	Unique	Top	Freq	Freq %	# Missing	% Missing
Brand	61.416	1.656	H&M	9.301	~15.14%	11.327	~15.57%

Table 10.23: Using describe() to get statistics about “Brand”

As we can see in Table 10.24, there were several brands that could be consolidated into an “Unknown” brand, such as “Other Brand” and all the bottom brands. We looked at all the brands and to try and figure out which ones could be consolidated under the “Unknown” brand.

Top 10 brands		Bottom 10 brands	
Brand	Value	Brand	Value
H&M	9.301	JD Williams	1
Other Brand	4.399	Villawool	1
Lindex	1.900	RUXARA	1
Vero Moda	1.623	Reino	1
Esprit	1.306	Gerber	1
Gina Tricot	1.258	Pearl	1
Kappahl	1.149	Fairtex	1
Only	1.021	Savannah	1
Vila	806	Carcia Jeans	1
Zara	734	Fitnesstukku	1

Table 10.24: Brands with highest 10 values and lowest 10 values

By looking at all the brands, we noticed that the brands didn’t contain any typos, which gives us the conclusion that the same applied here as in the “Type” column. That is, that the users of Netflea.com could only list a brand from a predefined list of brands. This saved us a significant time when consolidating the brands together.

We started by consolidating all the missing values in the “Brand” column into the “Unknown” brand. By doing this our count of known values went up to 72.743. The unique values of the “Brand” value went up by one, as expected since we were adding a new brand, so our new unique values were 1.657.

Next step was to lower the amount of unique brands. Our approach was to move all the brands that had lower than 50 total rows to the consolidated brand “Unknown”. This would ensure that we wouldn’t miss any types and we could always just drop the “Unknown” brand if we needed to.

As we can see in Fig. 10.33, we found a list of brands that could be consolidated into the “Unknown” brand. These all had more than 50 total rows, except for “Sloggi”, which we included due to it potentially being consolidated into the “L.O.G.G.” brand in the next step, and it would anyway be consolidated into the “Unknown” brand when we consolidated brands that had less than 50 rows.

```
unknownBrandList = ["Other Brand", "-", "Unknown brand", "Self-made",
                    "Brand not available", "Various brands", "Sloggi", "Handmade"]

for item in unknownBrandList:
    rowIDs = data[data.Brand.str.contains(item, na=False, flags=re.IGNORECASE)].index.values
    data.loc[rowIDs, 'Brand'] = "Unknown"
```

Figure 10.33: Consolidating brands into the “Unknown” brand

We then found a couple of brands that could be further consolidated together, such as “Baby Lindex” could be consolidated into “Lindex”. As we can see in Fig. 10.34, we used an approach we used before, that is finding a partial match in the “Brand” column and filling that value with a brand that it matches.

```
brandListBefore = ["Seppälä", "H&M", "Lindex", "Kappahl", "Zara", "Ellos",
                   "ReimaTec", "Benetton", "Reflex", "Premoda", "Dressman",
                   "Hilfiger", "Crivit", "Fransa", "Bull & Bear", "Inspire",
                   "Gap", "LOGG", "M&S Collection"]

brandListAfter = ["Seppälä", "H&M", "Lindex", "Kappahl", "Zara", "Ellos",
                  "ReimaTec", "Benetton", "Reflex", "Premoda", "Dressmann",
                  "Tommy Hilfiger", "Crivit", "Fransa", "Pull & Bear",
                  "Inspire", "Gap", "L.O.G.G.", "Marks & Spencer"]

for index, item in enumerate(brandListBefore):
    rowIDs = data[data.Brand.str.contains(item, na=False, flags=re.IGNORECASE)].index.values
    data.loc[rowIDs, 'Brand'] = brandListAfter[index]
```

Figure 10.34: Consolidating brands together

As we can see in Table 10.25, this method lowered the amount of unique brands to 168, a reduction of 1.488 from the previous 1.656 but the “Unknown” brand now had a frequency of 29.872 or ~41.07% of the dataset. We might therefore need to drop these rows later on if the data training doesn’t predict brands well.

Column	Count	Unique	Top	Freq	Freq %
Brand	72.743	168	Unknown	29.872	~41.07%

Table 10.25: Using describe() to get statistics about “Brand” after consolidating

We can visualize the “Brand” column by looking at Fig. 10.35. Since “Unknown” had much more rows than the other brands, we can see the countplot when we exclude it in Fig. 10.36.

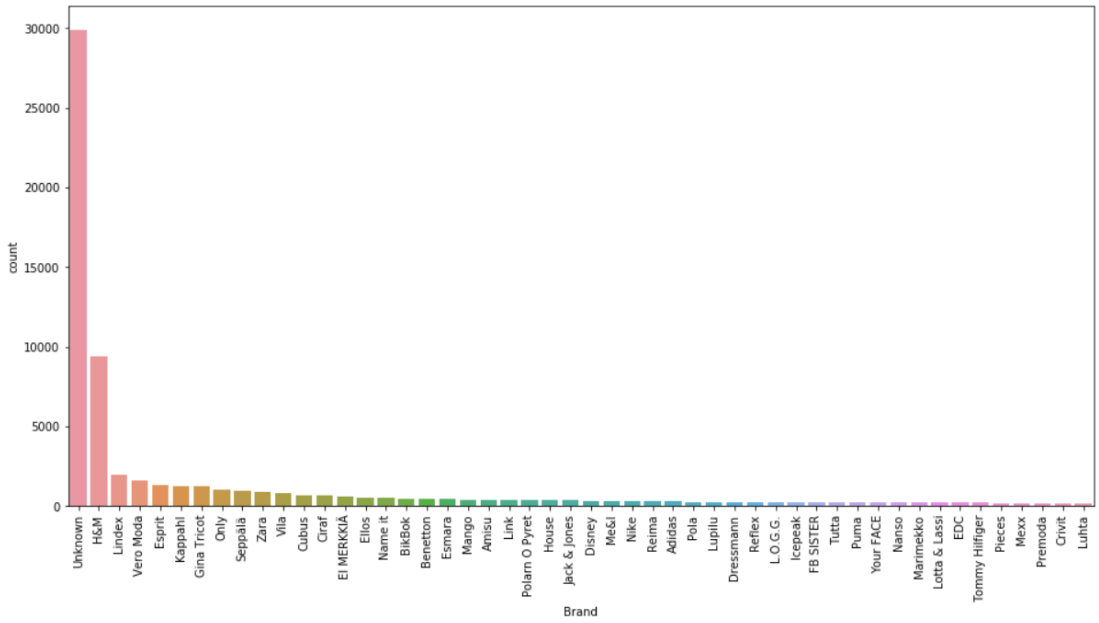


Figure 10.35: Top 50 brands

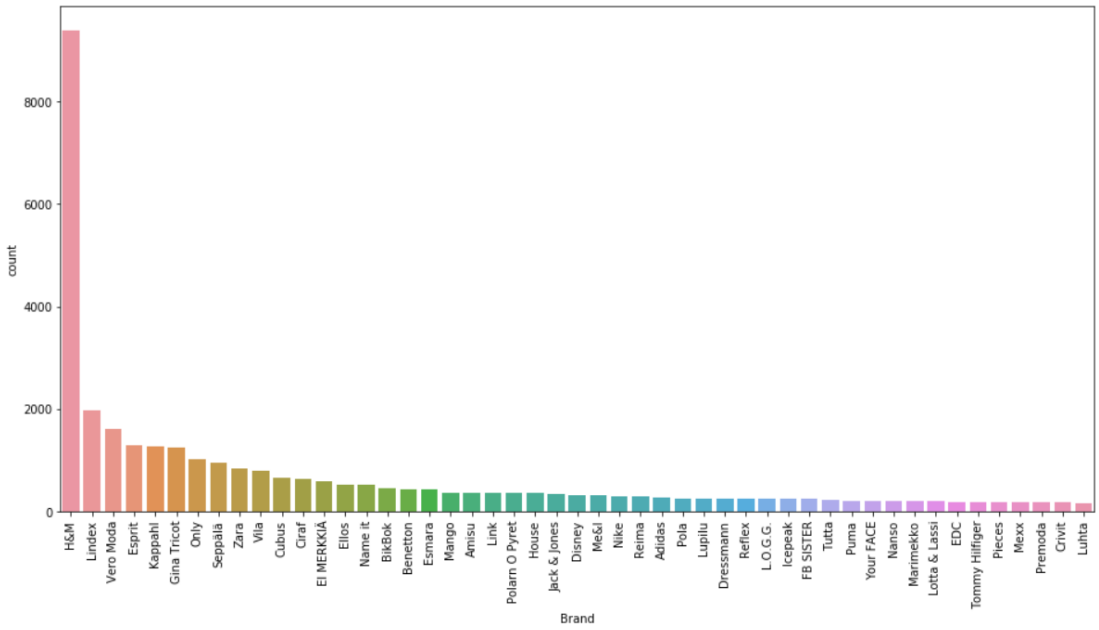


Figure 10.36:Top 2-50 brands

As we can see in Fig. 10.37, the distribution between brands is much better when we exclude “H&M”. The distribution between the bottom 50 brands had a more even distribution as we can see in Fig. 10.38.

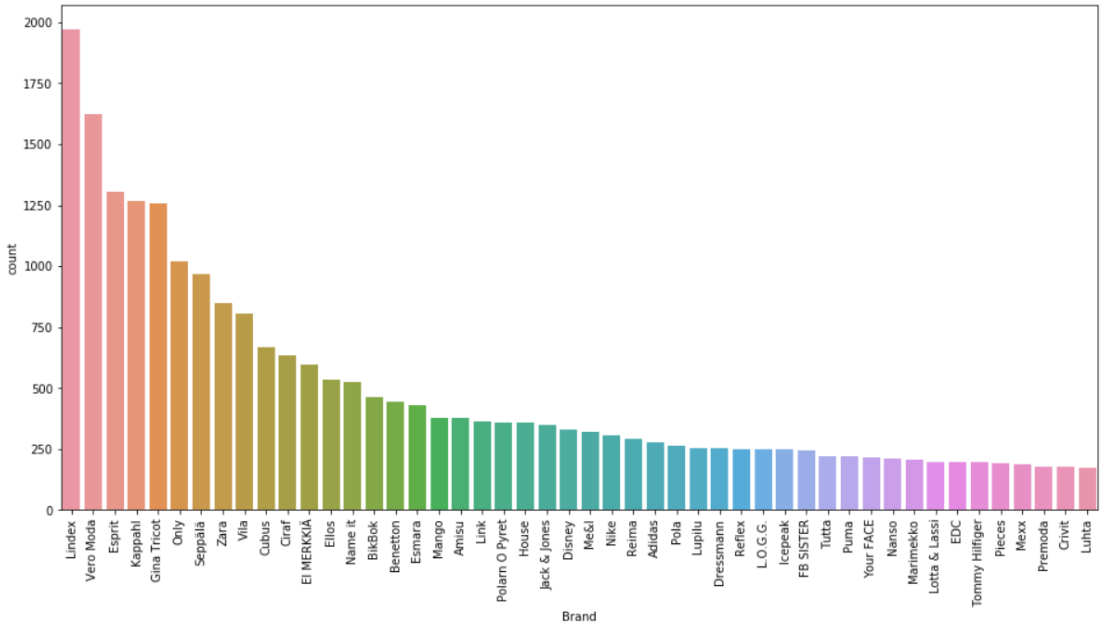


Figure 10.37:Top 3-50 brands

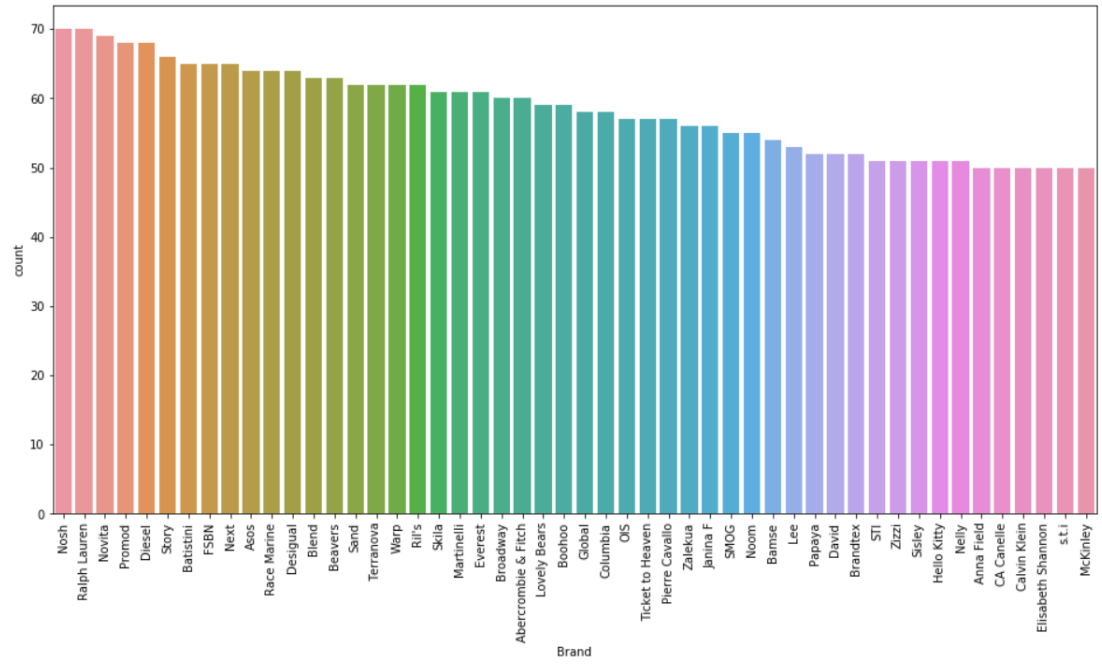


Figure 10.38:Bottom 50 brands

10.2.5 Consolidating colours together in the “Colour” column

When we consolidated the colours together when we were cleaning the Oxfam.org.uk dataset, we were consolidating everything together into a few colours. With this, we wanted to keep the colours that Netflea.com users had put, because as with the “Type” and “Brand”, they could only input the colour from a predefined list, which makes the odd of it being the correct colour more.

As we can see in Table 10.26, there were 15.693 missing values in the “Colour” column. The known values were 57.050 and the “Colour” column had 156 unique colours, which gave an average of ~365.71 known values for each colour. The most common colour was “Black” with ~23.04% of the total known values

Column	Count	Unique	Top	Freq	Freq %	# Missing	% Missing
Colour	57.050	156	Black	13.142	~23.04%	15.693	~21.57%

Table 10.26: Using describe() to get statistics about “Colour”

After looking at the full set of the colours, we decided to consolidate all colours with less than 50 total rows into an “Unknown” colour. Missing values were also consolidated into the “Unknown” colour. The Top 5 colours and the bottom 5 colours can be seen in Table 10.27, which shows that the bottom colours were not a misspelling and can’t be consolidated with another colour of the same colour.

Top 5 colours		Bottom 5 colours	
Colour	Value	Colour	Value
Black	13.142	Brass	1
Blue	4.374	Mauve	1
Gray	4.308	Sienna	1
White	3.175	Tie-dye	1
Dark blue	3.104	Smoky	1

Table 10.27: Colours with highest 5 values and lowest 5 values

As we can see in Table 10.28, after consolidating colours into the “Unknown” colour, it lowered the amount of unique colours to 78, a reduction of 78 from 156. The “Unknown” colour had a frequency of ~23.45% of the dataset and we might need to drop these rows later on as we might with the “Unknown” brand.

Column	Count	Unique	Top	Freq	Freq %
Colour	72.743	78	Unknown	17.058	~23.45%

Table 10.28: Using describe() to get statistics about “Colour” after removing

We can visualize the “Colour” column by looking at Fig. 10.39. Since “Unknown” colour might be dropped like with the “Unknown” brand, we can visualize the dataset if we drop the “Unknown” rows by looking at Fig. 10.40.

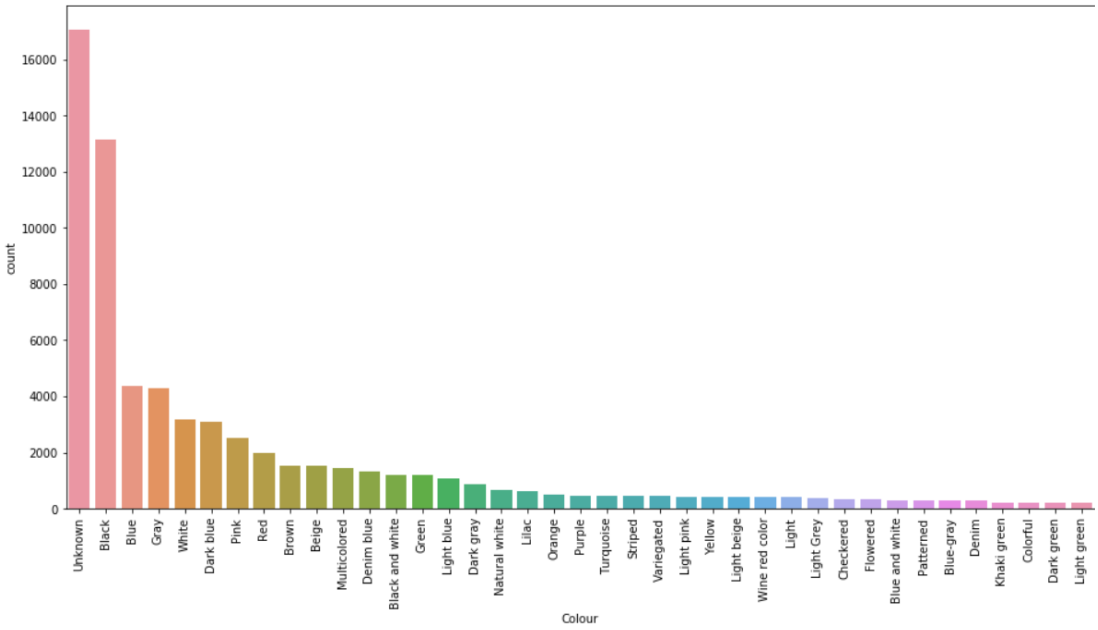


Figure 10.39: Top half of Colours

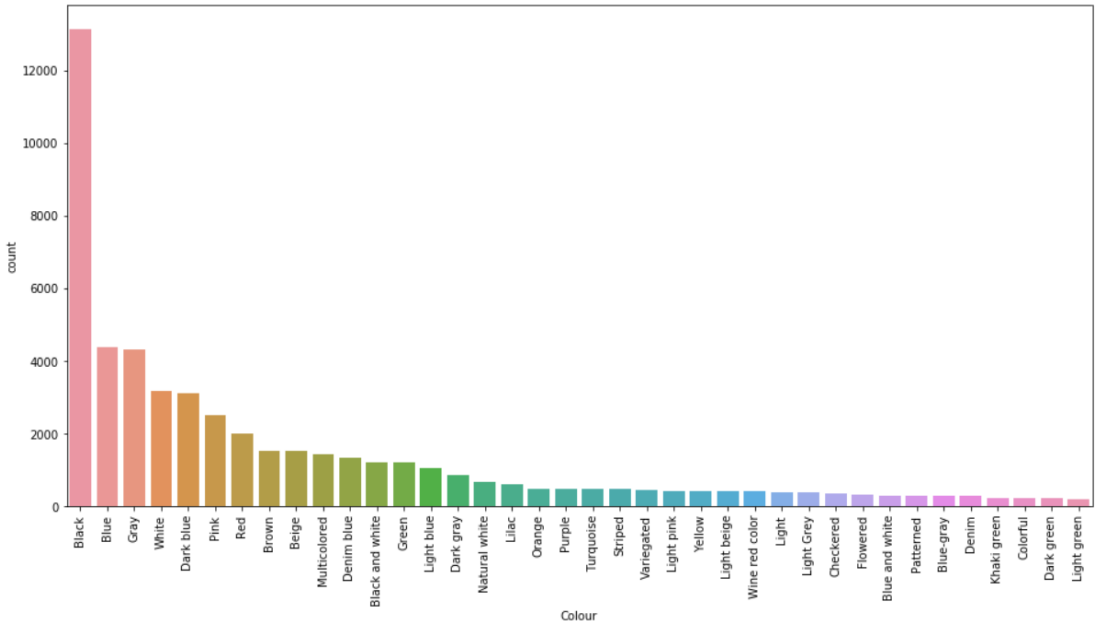


Figure 10.40: Top half, excluding the “Unknown” colour

10.2.6 Final touches

As with the final touches in the Oxfam.org.uk dataset, we needed to address a few things before the dataset was ready. First, we noticed that the "ImgName" contained a few duplicates, or 673, as we can see in Table 10.29.

Column	Count	Unique	Top	Freq	% Freq
Type	72.743	224	Shirt	5.209	~7.16%
Brand	72.743	168	Unknown	29.872	~41.07%
Colour	72.743	78	Unknown	17.058	~23.45%
ImgName	72.743	72.070	4ABA31D1-7734-47D2-95A0....	3	~0.01%

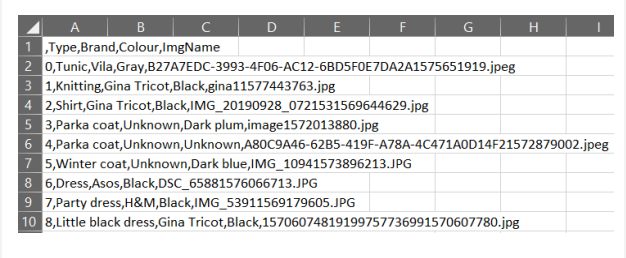
Table 10.29: Using describe() to get statistics about the dataset

We also knew that, from Chapter 9.2, the scraper managed to download 86.655 images but had 86.900 rows in the dataset. That meant that 245 rows didn't have an image in the folder attached to it. We needed to address this so we removed the images from the image folder that were not in our dataset.

Our folder with images contained a total of 86.655 images. We knew that the dataset contained 72.743 rows so we needed to remove 13.912 images that were associated with rows that we removed from the dataset. We also knew that of these 72.743 rows there were a total of 673 duplicates. We also needed to remove these images from our folder.

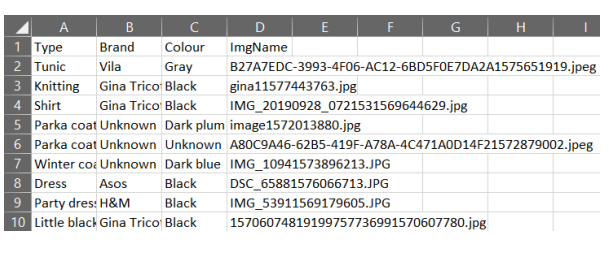
Our plan to remove these images from the folder was the same as we used when we removed images from the Oxfam.org.uk folder. We exported the new dataset as a .csv file. Then we extracted the image names into a .txt file. We then used the .txt file to delete all images that were not in the .txt file. Last step was to remove the duplicates which we did by using Excel and then made the document ready for Jupyter.

As we can see in Fig. 10.31, we removed the ID column, see Fig. 10.30 for comparison. We had already fixed the "ImgName" column on Jupyter Notebooks, so we didn't need to do anything else. Here we only needed to extract the "ImgName" column into a .txt document called removeImages.txt and use the same code we used when we deleted images that were not in the .txt document. The code for this can be seen in Fig. 10.19.



	A	B	C	D	E	F	G	H	I
1	,Type,Brand,Colour,ImgName								
2	0,Tunic,Vila,Gray,B27A7EDC-3993-4F06-AC12-6BD5F0E7DA2A1575651919.jpeg								
3	1,Knitting,Gina Tricot,Black,gina11577443763.jpg								
4	2,Shirt,Gina Tricot,Black,IMG_20190928_072153156964462 9.jpg								
5	3,Parka coat,Unknown,Dark plum,image1572013880.jpg								
6	4,Parka coat,Unknown,Unknown,A80C9A46-62B5-419F-A78A-4C471A0D14F21572879002.jpeg								
7	5,Winter coat,Unknown,Dark blue,IMG_10941573896213.JPG								
8	6,Dress,Asos,Black,DSC_65881576066713.JPG								
9	7,Party dress,H&M,Black,IMG_53911569179605.JPG								
10	8,Little black dress,Gina Tricot,Black,1570607481919975773699157060778 0.jpg								

Figure 10.30: .csv document before



	A	B	C	D	E	F	G	H	I
1	Type	Brand	Colour	ImgName					
2	Tunic	Vila	Gray	B27A7EDC-3993-4F06-AC12-6BD5F0E7DA2A1575651919.jpeg					
3	Knitting	Gina Trico	Black	gina11577443763.jpg					
4	Shirt	Gina Trico	Black	IMG_20190928_0721531569644629.jpg					
5	Parka coat	Unknown	Dark plum	image1572013880.jpg					
6	Parka coat	Unknown	Unknown	A80C9A46-62B5-419F-A78A-4C471A0D14F21572879002.jpeg					
7	Winter coa	Unknown	Dark blue	IMG_10941573896213.JPG					
8	Dress	Asos	Black	DSC_65881576066713.JPG					
9	Party dres	H&M	Black	IMG_53911569179605.JPG					
10	Little black	Gina Trico	Black	1570607481919975773699157060778 0.jpg					

Figure 10.31: .csv document after

This method resulted in the image folder now containing 72.066 images. 4 images less than what we expected, so we needed to remove the rows where those four images were.

We used pandas to work with the data instead of using Excel as we used for Oxfam. Here we needed to remove all duplicate rows from the dataset and then remove any row whose image is not in our folder.

As we can see in Fig. 10.32, we dropped duplicate rows from the dataset which resulted in 72.070 rows being left. As expected, the unique rows for the "ImgName" column were now the same amount as all the rows, or 72.070.

```
data = data.drop_duplicates()
data.describe()
```

	Type	Brand	Colour	ImgName
count	72070	72070	72070	72070
unique	224	168	78	72070
top	Shirt	Unknown	Unknown	20191019_1411001571483557.jpg
freq	5151	29613	16943	1

Figure 10.32: Dropping duplicate rows from the dataset

As we can see in Fig. 10.33, we found the four rows that were missing an image and dropped them from the dataset. This resulted in our dataset now containing 72.066 rows or ~0.006% less.

```
imagesList = [line.rstrip('\n') for line in open("imagesList.txt")]
print("Total images in list: ", len(imagesList))
```

```
Total images in list:  72066
```

```
before = data.ImgName.unique()
beforeLen = len(before)
print("Number of unique values in 'ImgName' before: ", beforeLen)

data = data[~data.ImgName.isin(imagesList)]

after = data.ImgName.unique()
afterLen = len(after)
print("Number of unique values in 'ImgName' after: ", afterLen)

difference = beforeLen - afterLen
print("Difference is: ", difference)
```

```
Number of unique values in 'ImgName' before:  72070
Number of unique values in 'ImgName' after:  4
Difference is:  72066
```

Figure 10.33: Removing rows whose "ImgName" value is not in our folder

Fig. 10.34 shows us the final top 50 rows of the “Type” column. As we can see, there were still a lot of rows that were from the “Shirt” type, or ~25% higher than the second highest type.

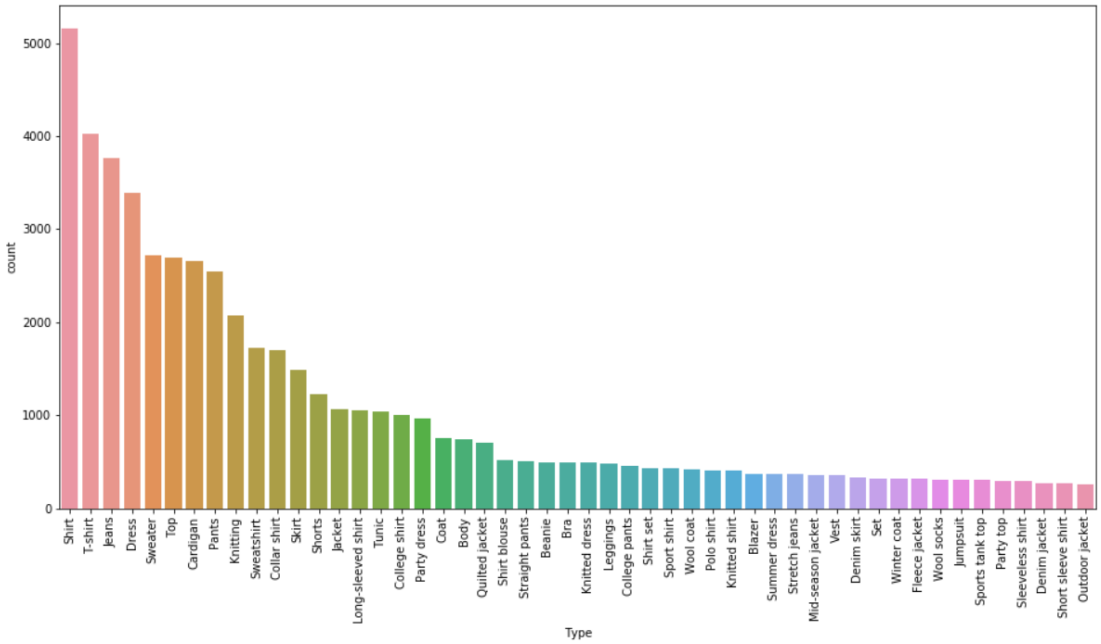


Figure 10.34: Final top 50 rows of “Type”

As we can see in Fig. 10.35, “H&M” was almost 5 times higher than “Lindex” when we exclude “Unknown”.

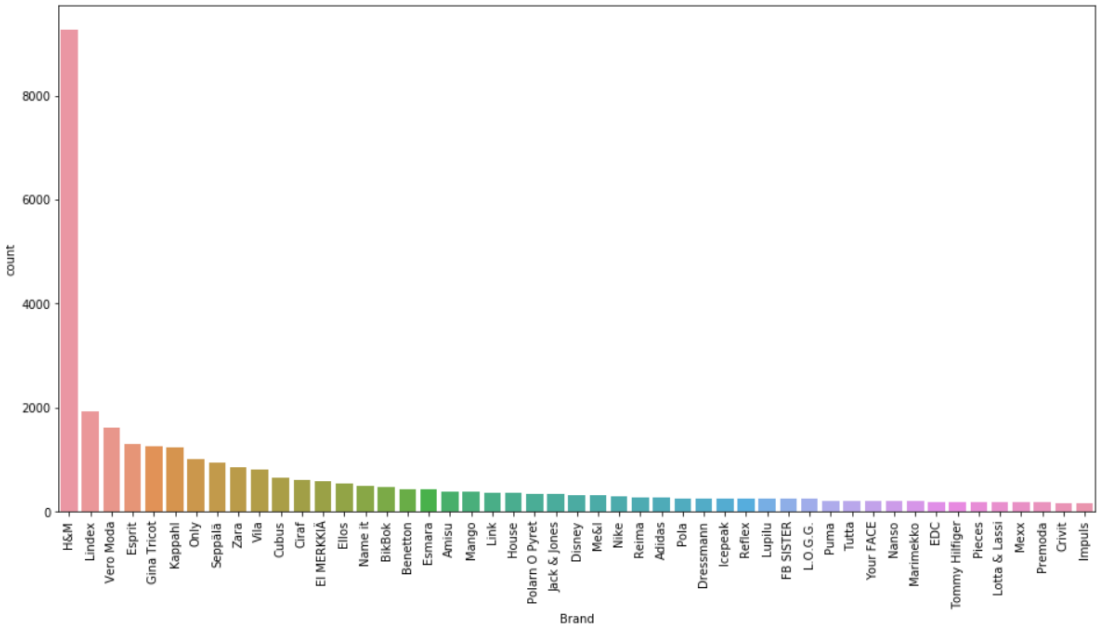


Figure 10.35: Final top 50 rows of “Brand”, excluding “Unknown”

As we can see in Fig. 10.36, The “Black” colour had a little more than three times the amount as “Blue” if we excluded the “Unknown” colour.

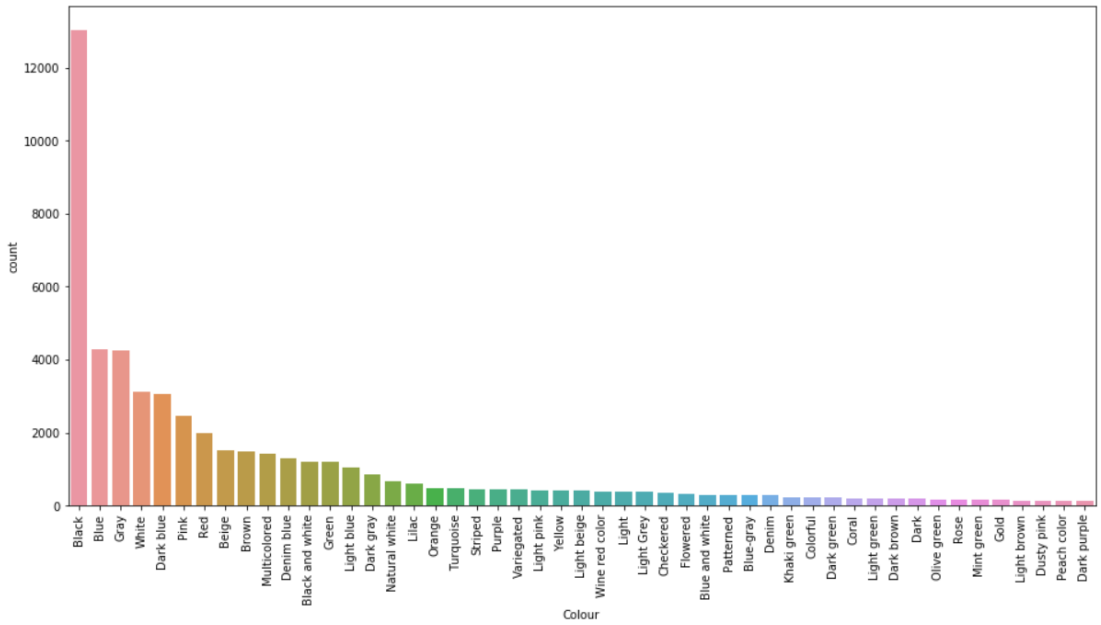


Figure 10.36: Final top 50 rows of “Colour”, excluding “Unknown”

10.3 Exploring the backup dataset

Since the data for the backup dataset was already clean because we manually entered the tags when looking at the pictures, we didn't need to clean the data or wrangle the data. Instead we decided to explore the data to get a better understanding on what is in the dataset and if we needed to add more to the dataset.

To start exploring the data, we imported the four modules that we listed in Table 10.1. We also set the style of the Seaborn plot to the style "darkgrid" and the colour palette to "pastel".

Next step was to get the dataset on a dataframe form. This was done by using the read_csv function within the pandas module. As we can see in Fig. 10.37, we referred to the dataframe as "df", but df stands for dataframe.

Since we didn't need the "ImgName" column for exploring the dataset, we decided to drop it and a subset of the dataset after dropping the "ImgName" column can be viewed in Fig. 10.38.

```
import pandas as pd
import re
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="darkgrid")
sns.set_palette("pastel")

df = pd.read_csv("Backup_Dataset_CSV.csv")
```

```
df.shape
```

```
(2886, 5)
```

```
df.sample(10)
```

	Type	Colour	Brand	Source	ImgName
388	Hoodie	Black	Nike	Picclick	3772129453.jpg
1773	Hoodie	Gray	Adidas	Poshmark	11908.jpg
509	Sweatshirt	Black	Nike	Grailed	EiHYc5cWRLWb3Uilt8br.jpg
109	Sweatshirt	Gray	Nike	Grailed	4PYZ4zk2SaKGAIB1b9GD.jpg
127	T-shirt	Black	Nike	Grailed	64ZqJQgzSTG0Imo3gXB8.jpg
726	Hoodie	Blue	Adidas	Picclick	8855213352.jpg
105	T-shirt	Red	Nike	Grailed	4EdvMFWLR62P4FLRZpLj.jpg
933	Hoodie	Gray	Adidas	Shpock	5e3bf06c955b1c2b611c68a7.jpg
2061	Hoodie	Red	Adidas	Poshmark	84606.jpg
405	Hoodie	Black	Nike	Picclick	6635030265.jpg

Figure 10.37: Shape of dataset and random 10 rows from the dataset

```
df = df.drop(columns=['ImgName'])
df
```

	Type	Colour	Brand	Source
0	T-shirt	Black	Adidas	Grailed
1	T-shirt	Red	Nike	Grailed
2	T-shirt	White	Adidas	Grailed
3	Sweatshirt	Gray	Nike	Grailed
4	T-shirt	Gray	Nike	Grailed
5	T-shirt	White	Adidas	Grailed
6	T-shirt	Black	Nike	Grailed
7	Sweatshirt	Red	Adidas	Grailed
8	T-shirt	Black	Nike	Grailed
9	Sweatshirt	Blue	Adidas	Grailed
10	Hoodie	Black	Nike	Grailed
11	Sweatshirt	Black	Nike	Grailed
12	Hoodie	White	Nike	Grailed
13	Hoodie	Red	Nike	Grailed
14	Hoodie	Blue	Adidas	Grailed
15	Hoodie	Black	Nike	Grailed
16	T-shirt	Black	Nike	Grailed
17	Sweatshirt	Gray	Nike	Grailed
18	T-shirt	Black	Adidas	Shpock
19	T-shirt	Black	Adidas	Shpock

Figure 10.38: Dataset after removing "ImgName"

Value counts of the dataset can be seen in Tables 10.30 to 10.33 along with a comment.

Type	Count
Hoodie	1.225
T-Shirt	1.172
Sweatshirt	489
Total	**2.886**

Table 10:30: Type counts

Comment: We got almost the same amount of hoodies and t-shirts, but sweatshirt is 2-2.5 times lower than the other two. Google AutoML requires at least 100 entries for each tag and we are well above it for each of the tags.

However, Google AutoML also indicates that it would be better to have 1.000 entries for each tag to get as accurate a prediction as possible and here, only hoodies and t-shirts have over 1,000 entries.

Colour	Count
Black	744
Blue	708
Gray	629
Red	434
White	371
Total	**2.886**

Table 10:31: Colour counts

Comment: All the colours are above the 100 threshold that Google AutoML sets for each tag, however all fall short of the 1,000 threshold.

Black, blue and gray all have over 500 while red and white are below 500. Black, blue and grey might be problematic since some images such as dark blue can be either interpreted as black or blue depending on image quality and they are almost indistinguishable from each other.

White might also be somewhat problematic since white could be confused with light gray, but the colour red is very distinguishable so it shouldn't be a problem.

Brand	Count
Adidas	1.481
Nike	1.405
Total	**2.886**

Table 10:32: Brand counts

Comment: The two brands have almost the same amount of entries and both are above the 1,000 threshold. The model should be able to accurately predict the brands.

However, the brand logos are not always the same, e.g. Nike can have the swoosh logo, the swoosh logo along with the text Nike and only the text. We might need to address this by adding more images of the brand in different contexts that address this.

Source	Count
Poshmark.com	1.121
Grailed.com	846
Picclick.com	790
Shpock.com	129
Total	**2.886**

Table 10.33: Source counts

Comment: The source has no effect on the training of a model since we will drop this row when importing the .csv document to Google AutoML.

This column was included to check how the different sources could work in future expansion of the dataset.

Although the tables give an insight into the dataset, visualizing it would be more insightful.

As we can see in Fig. 10.39, sweatshirts don't have close to the same amount of rows as hoodies and t-shirts. To rectify this in the future, we will need to add sweatshirts to the dataset to bring it over 1,000.

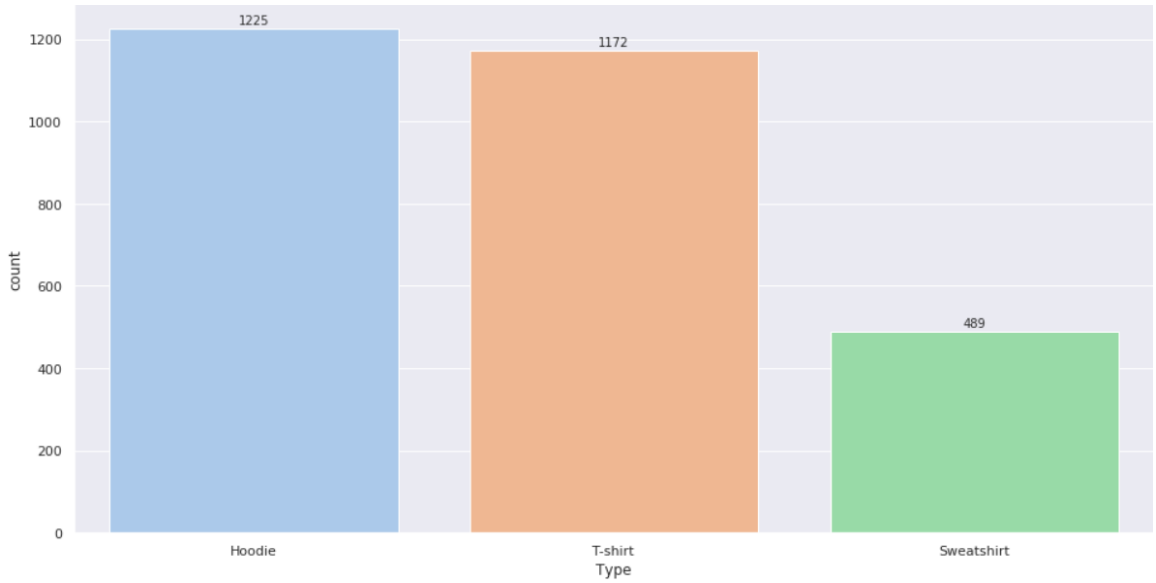


Figure 10.39: Visualizing the count of types

As we can see in Fig. 10.40, both brands have almost the same amount of rows.

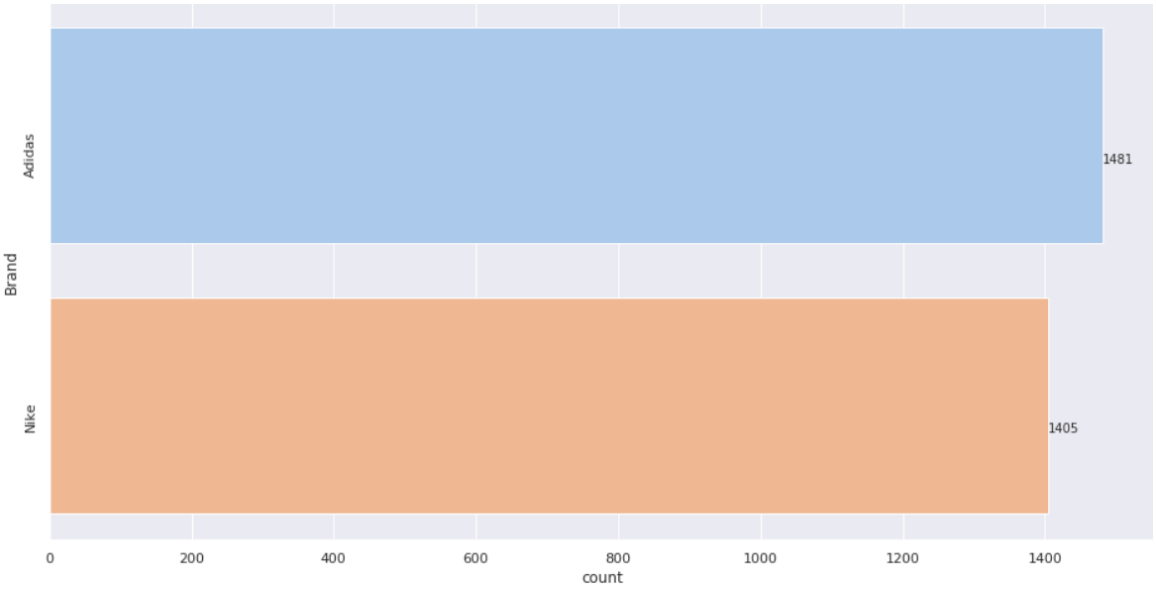


Figure 10.40: Visualizing the count of brands

As we can see in Fig. 10.41, black, blue and gray have much more rows than red and white. We might need to add more images with the tags red and white, although we might need to add more images with all colours to bring the total rows to above 1,000 for all of them.

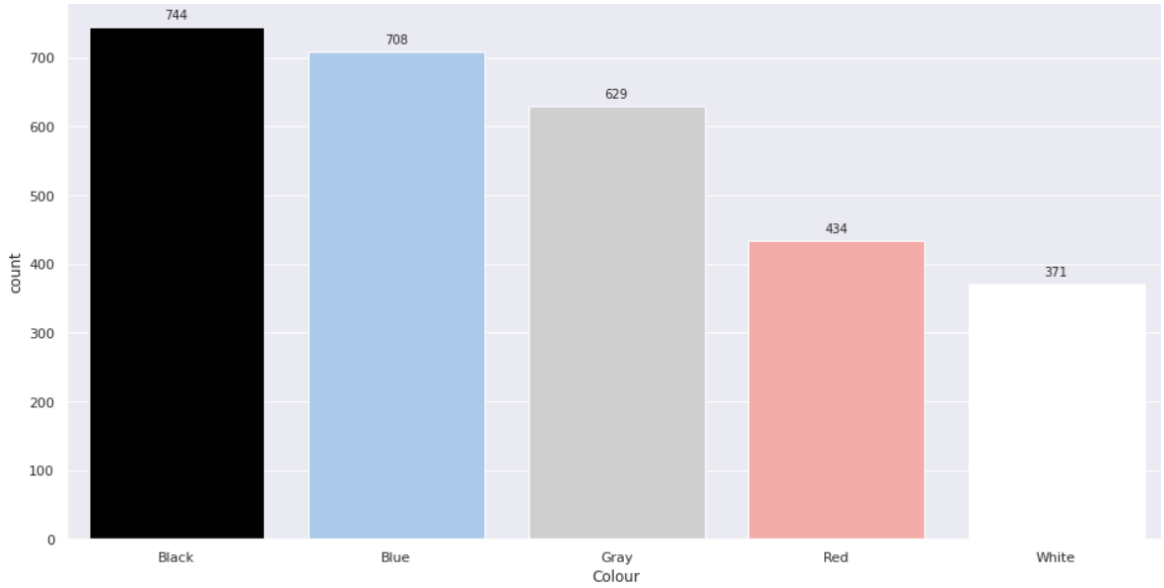


Figure 10.41: Visualizing the count of colours

As we can see in Fig. 10.42, we got the most images from Poshmark.com, while the least from Shpock.com, which only contributed 129 images to the dataset. In the future we will not look at getting images from Shpock.com as it yields a significantly lower amount of images than the other sources.

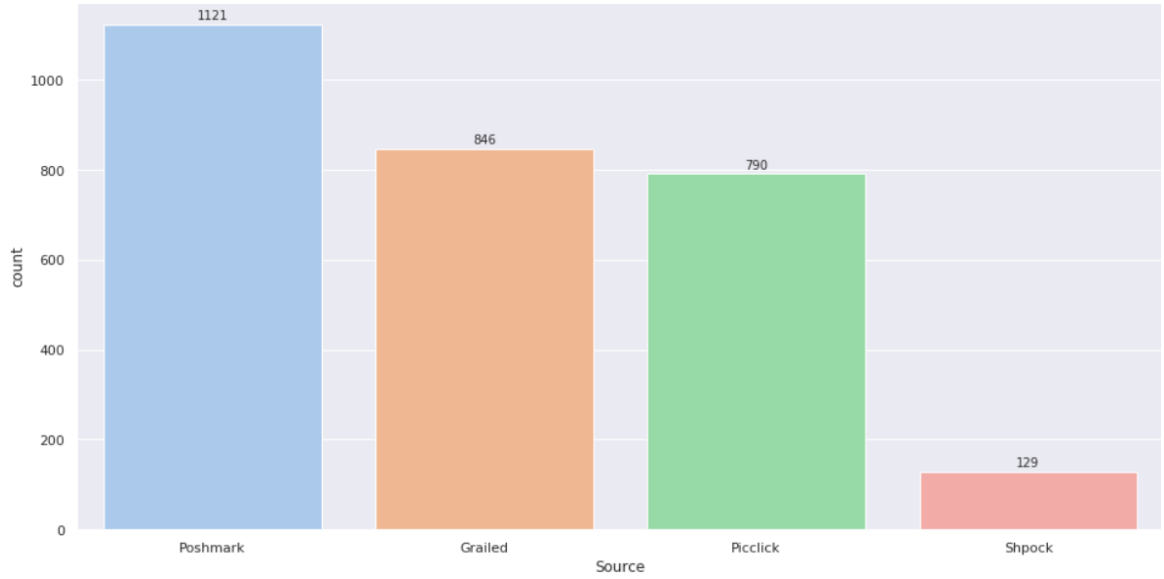


Figure 10.42: Visualizing the count of sources

As we can see in Fig. 10.43, Picklick.com and Poshmark.com yielded the most images of hoodies, while Poshmark.com and Grailed.com yielded the most images of t-shirts. Picclick yielded an extremely low amount of sweatshirts as compared to Grailed.com and Poshmark.com, which yielded most of the sweatshirt images. Shpock.com yielded mostly t-shirts.

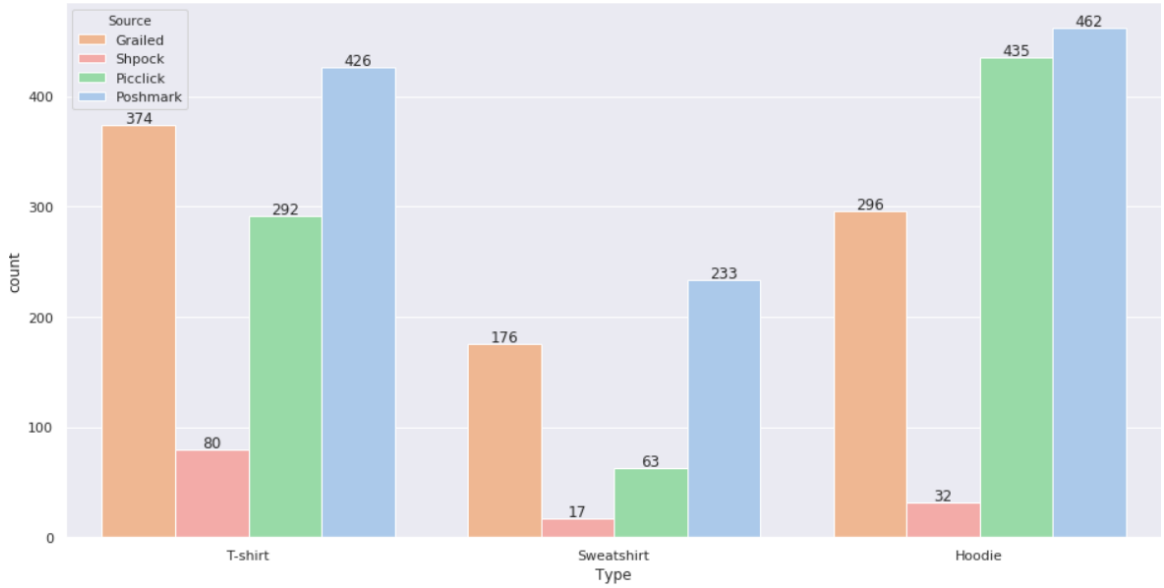


Figure 10.43: Visualizing the count of types from each source

Visualizing the types of each brand from each source can be seen in Fig. 10.44. We can see that Grailed.com yielded more images from Nike, while Shpock.com yielded no images from Nike.

Poshmark.com yielded around the same amount of images of sweatshirts and hoodies from both brands but yielded almost double as many t-shirts from Adidas than Nike.

Picklick.com yielded around the same amount of types from each brand, although it yielded almost twice as many t-shirts from Adidas than Nike.

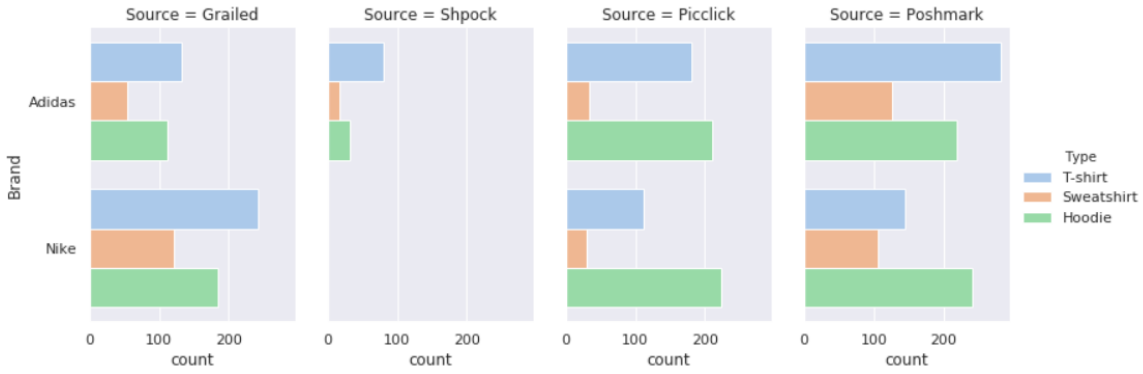


Figure 10.44: Visualizing the count of types in each brand from each source

11 Training Models

We decided to use Google AutoML to train a model that we would use to serve predictions for our application once it became clear that the Vision AI solutions were too generalized. Our Product Owner set up a project in Google Cloud[33], which enabled us to start importing the datasets and images to Google and eventually start training models. All data is stored within this project.

In the beginning we had two datasets that we trained, Oxfam.org.uk dataset and Netflea.com dataset. Our initial idea was to try both of them and use the one that yielded the highest precision of predictions.

Once it became clear that the models were not giving us the results that we wanted, we shifted our focus on the backup dataset that we trained.

[33] "Google Cloud." https://cloud.google.com/. Accessed 23 Apr. 2020.

11.1 Preparing Google AutoML

To work with Google Cloud, we used Google Cloud SDK, which is a set of command-line tools for developing with Google Cloud. These tools can be used to access Compute Engine, Cloud Storage, BigQuery, and other Google Cloud services directly from the command line.[34]

Once we set up the SDK and initialized the SDK to work with our project[35], we could use commands to work with Google Cloud. The commands that we used along with a description can be seen in Table 11.1.

Command	Description
gsutil	The python application that lets us access Cloud Storage from the command line.[36]
mb	Indicating that we want to make a new bucket.[37]
-b <on|off>	Specifies the uniform bucket-level access setting. Default is "off".[38]
-l <location>	Specifying the location of the bucket.[39] We use us-east1 because AutoML Vision currently requires that location.[40]
gs://<bucket_name>	gs:// is the location of the buckets within Google Cloud and it requires a name of a bucket.[41]
cp	Allows us to copy data between our local file system and the cloud.[42]
-m	Allows us to perform a parallel (multi-threaded/multi-processing) copy.[43]
-R	Causes directories, buckets, and bucket subdirectories to be copied recursively.[44]

Table 11.1: Commands in Google Cloud SDK

[34] "Cloud SDK - Google Cloud." https://cloud.google.com/sdk. Accessed 23 Apr. 2020.

[35] "Quickstart for Windows - Google Cloud." https://cloud.google.com/sdk/docs/quickstart-windows. Accessed 23 Apr. 2020.

[36] "gsutil tool - Google Cloud." https://cloud.google.com/storage/docs/gsutil. Accessed 23 Apr. 2020.

[37] "mb - Make buckets | Cloud...." https://cloud.google.com/storage/docs/gsutil/commands/mb#description. Accessed 23 Apr. 2020.

[38] Options | Cloud Storage..." https://cloud.google.com/storage/docs/gsutil/commands/mb#options. Accessed 23 Apr. 2020.

[39] "Bucket locations | Cloud.." https://cloud.google.com/storage/docs/gsutil/commands/mb#bucket-locations. Accessed 23 Apr. 2020.

[40] "AutoML Vision API Tutorial | Cloud AutoML Vision....." https://cloud.google.com/vision/automl/docs/tutorial. Accessed 23 Apr. 2020.

[41] "mb - Make buckets | Cloud...." https://cloud.google.com/storage/docs/gsutil/commands/mb. Accessed 23 Apr. 2020.

[42] "cp - Copy files and objects...." https://cloud.google.com/storage/docs/gsutil/commands/cp#description. Accessed 23 Apr. 2020.

[43] Description | Cloud Storage.." https://cloud.google.com/storage/docs/gsutil/commands/cp#description. Accessed 23 Apr. 2020.

[44] Options | Cloud Storage | Google.." https://cloud.google.com/storage/docs/gsutil/commands/cp#options. Accessed 23 Apr. 2020.

The first thing we needed to do was to create buckets for our Oxfam.org.uk and Netflea.com datasets along with a bucket to hold our .csv documents. We created three buckets, “oxfam_bucket”, “netflea_bucket” and “csv_bucket_99”. Later on, we also added a bucket for our backup dataset called “backup_dataset_1”. An example of how we created the buckets in the SDK can be seen in Listing 11.1.

```
$ gsutil mb -b on -l us-east1 gs://oxfam_bucket
```

Listing 11.1: Creating a storage bucket with Google SDK for the Oxfam.org.uk dataset

Once we created the buckets, we needed to upload the images and .csv documents to our buckets. To do so we used the “cp” command. An example of the command we used in SDK can be seen in Listing 11.2.

```
$ gsutil -m cp -R "C:\Users\Kalli\Desktop\LOKA\OxfamImages" gs://oxfam_bucket
```

Listing 11.2: Uploading images from local address to bucket

Once of the main disadvantages of the command that we used to upload the images to our buckets, was that it uploaded the folder into the buckets so the location of an image was: “gs://oxfam_bucket/OxfamImages/IMAGE.jpg” instead of “gs://oxfam_bucket/IMAGE.jpg”. When we uploaded the backup dataset, we rectified this by adding “\.” at the end of our local address so that the command would only upload the images, not the folder itself, as we can see in Listing 11.3.

```
$ gsutil -m cp -R "C:\Users\Kalli\Desktop\BackupDataset\." gs://backup_dataset_1
```

Listing 11.3: Uploading images within local folder to bucket

The last step we had to do before creating a dataset that could be used to start training models, was to create a .csv document that could be used within Google Cloud. To use a multi-label dataset, we needed to set the .csv document up in the format Google wanted, “gs://BUCKET/IMAGE.jpg,LABEL1,LABEL2,..”[45].

To do so, we added the location within Google Cloud for the images in a specific bucket in front of the image name as we can see by comparing Fig. 11.1 and Fig. 11.2. We then utilized the “&” within Excel to make the document ready as a .csv document by adding columns together with a comma between them. Uploading the .csv documents to our “csv_bucket_99” worked in the same way as it did with the images.

	A	B	C	D	E
1	Type	Colour	Brand	Img name	
2	Long dress	Cream	Unknown	HD_100337355_01.jpg	
3	Mini skirt	White	Unknown	HD_100403601_01.jpg	
4	Trousers	Green	Unknown	HD_100424462_01.jpg	
5	Jacket	Green	Unknown	HD_100443530_01.jpg	
6	Jacket	Beige	Unknown	HD_100452597_01.jpg	
7	Jacket	Grey	Unknown	HD_100453860_01.jpg	
8	Strapless dress	Black	ASOS	HD_100453881_01.jpg	
9	Smart jacket / coat	Brown	Unknown	HD_100462918_01.jpg	
10	Smart jacket / coat	Red	Unknown	HD_100463822_01.jpg	

Figure 11.1: .xlsx Oxfam.org.uk dataset

	A	B	C	D	E	F	G	H	I
1	gs://oxfam_bucket/OxfamImages/HD_100337355_01.jpg,Long dress,Cream,Unknown								
2	gs://oxfam_bucket/OxfamImages/HD_100403601_01.jpg,Mini skirt,White,Unknown								
3	gs://oxfam_bucket/OxfamImages/HD_100424462_01.jpg,Trousers,Green,Unknown								
4	gs://oxfam_bucket/OxfamImages/HD_100443530_01.jpg,Jacket,Green,Unknown								
5	gs://oxfam_bucket/OxfamImages/HD_100452597_01.jpg,Jacket,Beige,Unknown								
6	gs://oxfam_bucket/OxfamImages/HD_100453860_01.jpg,Jacket,Grey,Unknown								
7	gs://oxfam_bucket/OxfamImages/HD_100453881_01.jpg,Strapless dress,Black,ASOS								
8	gs://oxfam_bucket/OxfamImages/HD_100462918_01.jpg,Smart jacket - coat,Brown,Unknown								
9	gs://oxfam_bucket/OxfamImages/HD_100463822_01.jpg,Smart jacket - coat,Red,Unknown								
10	gs://oxfam_bucket/OxfamImages/HD_100497065_01.jpg,Casual jacket - coat,Purple,Unknown								

Figure 11.2: .csv Oxfam.org.uk dataset ready for Google

[45] "Preparing your training data | Cloud AutoML ...." https://cloud.google.com/vision/automl/docs/prepare. Accessed 24 Apr. 2020.

11.2 Creating datasets and training datasets

Next step was to create a dataset within Google. To do so, we decided to use the web UI as opposed to using a python script or cli, as it was straightforward to create it through the UI and rather fast.

Within the Vision tab in Google Cloud[46], we went to “Datasets” and added new multi-label datasets. For the Oxfam.org.uk dataset we named it “dataset_1”, for the Netflea.com dataset, we named it “dataset_2” and for the backup dataset, we named it “Dataset_15_04_2020__22_36”. We then selected the .csv documents in our “csv_bucket_99” to link the images for each dataset to the labels we assigned for each image.

Oxfam.org.uk dataset yielded 16.347 tagged images, a reduction of 196 images from the 16.543 tagged images that we had, which can be attributed to low quality images or duplicate images.

Netflea.com dataset yielded 71.448 tagged images, a reduction of 618 images from 72.066 images.

The backup dataset yielded 2.864 tagged images, a reduction of 22 images from 2.886 images.

As we can see in Table 11.2, we trained a total of 3 datasets with a total of 90.659 images and the train cost was a total 48 node hours, on average 16 node hours.

To get a better view of the train cost, which we listed at $3.15 p/ node hour in Chapter 5.1, we also listed the train cost by node hour for each 1.000 images. As we can see the node hours varied greatly. The main reasoning for the low node hours for Oxfam.org.uk and Netflea.com datasets as opposed to the backup dataset lies in the fact that the default node hours as recommended by Google was upwards to 400 hours for Netflea.com and upwards to 100 hours for Oxfam.org.uk as opposed to 16 for the backup dataset. The cost of ordering 400 node hours was for example $1.260, which we deemed inappropriate use of the funds provided by K3.

The total cost for training these 3 datasets was a total of 48 hours, where 40 hours were included as free by Google, as we listed in Chapter 5.1, with the remaining 8 hours paid by K3 at a cost of $25.2.

Dataset	Number of images	Train cost (node hours)	Node hour p/ 1.000 images	Train cost ($)	$ cost p/ 1.000 images
Oxfam.org.uk	16.347	12	~0.73	$37.80	~$2.31
Netflea.com	71.448	20	~0.28	$63.00	~$0.88
Backup	2.864	16	~5.59	$50.40	~$17.60
Total	**90.659**	**48**	**~0.53**	**$151.2**	**~$1.67**

Table 11.2: Training cost comparison for the three datasets we trained

[46] "Vision - Google Cloud Console." https://console.cloud.google.com/vision. Accessed 24 Apr. 2020.

12 Google AutoML Evaluation

Once we had trained the models, the next step was to evaluate the performance of the models. First we had two models, “dataset_1_model”, based on “dataset_1” and “dataset_2_model”, based on dataset_2. Once it became clear that the datasets had subpar precision scores, we trained a new model based on the dataset “backup_dataset”.

The reason for the subpar performances of the first two datasets can be attributed to the fact that there were too many labels which meant that for each label, there were too few images. We also checked the tagging of the images and noticed that a lot of the images were wrongly tagged by employees of the sites, as can be seen in Fig. 12.1, where a subsample of the tag Green in “dataset_2” shows various clothing items tagged Green although it is in other colours.

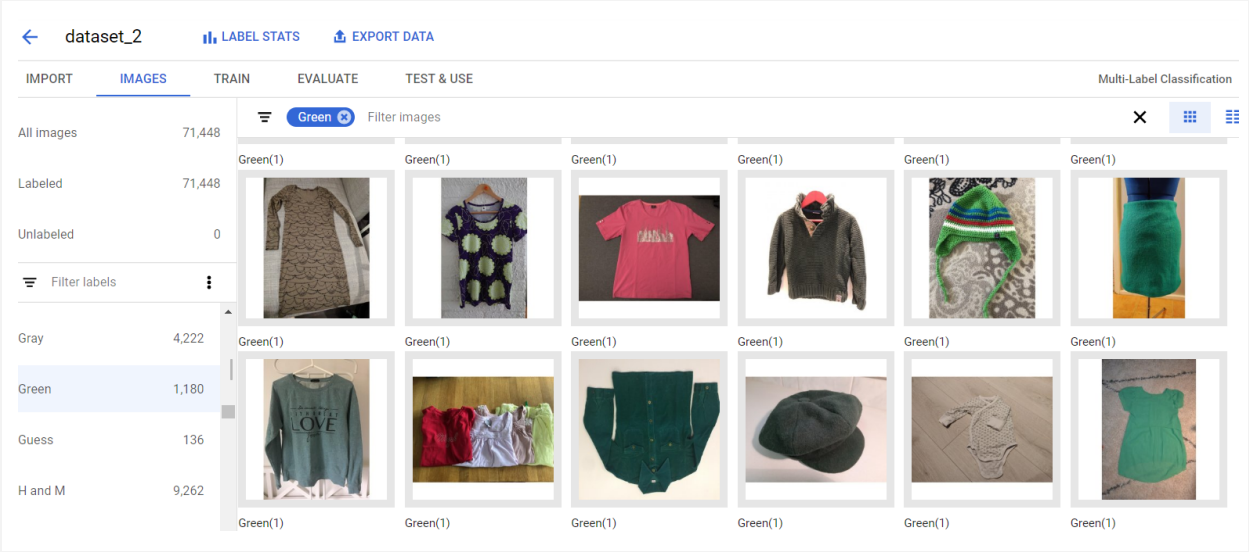


Figure 12.1: Subsample of items tagged “Green” in dataset_2

Initially when we scraped the sites, we were under the impression that either the employees or the users of the sites manually entered the tags that were associated with each picture. It however became clear that this might not be the case or that the items were tagged incorrectly due to various reasons.

We therefore came to the conclusion that scraping entire websites didn’t yield the results that we wanted, and going through each and every image was going to be more time consuming than building our own dataset containing hand picked images that fitted our tags that we wanted to include.

There are a few things that we need to have knowledge of before evaluating the performance of each model. Luckily Google has a good documentation on what each of these metrics mean.

When training the data, Google automatically splits the data into three sets, training set, validation set and test set, with a split of 0.8/0.1/0.1, although it is possible to manually split the dataset with a different split than the automatic split. Definition for each set is explained in Table 12.1.

Term	Definition
Training Set	80% of the dataset is automatically in the training set. This is the data the model sees during training, that is, it's used to learn the parameters of the model, namely the weights of the connections between the nodes of the neural network.[47]
Validation Set	10% of the dataset is automatically in the validation set. The validation set is also used during the training process. After the model learning framework incorporates training data during each iteration of the training process, it uses the model's performance on the validation set to tune the model's hyperparameters, which are variables that specify the model's structure. It is wise to not use the training set to tune the hyperparameters, since it is quite likely that the model would have a hard time generalizing to examples that don't exactly match it.[48]
Test Set	10% of the dataset is automatically in the test set. The test set is not involved in the training process at all. Once the model has completed its training entirely, the test set is used to give an idea of how well the model would perform on real-world data.[49]

Table 12.1: Explaining training set, validation set and test set

Once the model is trained, Google offers a summary of its performance. Definitions for each of the terms that come up in the evaluation of the performance can be viewed in Table 12.2.

Term	Definition
Model Output	A number that the model outputs that indicates how high confidence the model has that the label should be applied to the example given, e.g. a test image.[50]
Confidence Threshold	The score threshold refers to the level of confidence the model must have to assign a category to a test item. For example if the score threshold is set at 80%, it will not include any predictions that are lower than the threshold. A low threshold score will result in more classified images, but at the risk of misclassifying images. A high threshold score will result in less classified images, but also less risk of misclassifying images.[51]
Precision and recall	Precision and recall helps us understand how well our model is capturing information, and how much it's leaving out. Precision tells us, from all the test examples that were assigned to a label, how many actually were supposed to be categorized with that label. Recall tells us, from all the test examples that should have had the label assigned, how many were actually assigned the label.[52]

Table 12.2: Explaining terms that come up in the evaluation phase

[47] "AutoML Vision training set..." https://cloud.google.com/vision/automl/docs/beginners-guide#training_set. Accessed 27 Apr. 2020.
[48] "AutoML Vision validation.." https://cloud.google.com/vision/automl/docs/beginners-guide#validation_set. Accessed 27 Apr. 2020.
[49] "AutoML Vision test set..." https://cloud.google.com/vision/automl/docs/beginners-guide#test_set. Accessed 27 Apr. 2020.
[50] "AutoML Vision Beginner's guide - Google Cloud." https://cloud.google.com/vision/automl/docs/beginners-guide#how_do_i_interpret_the_model%E2%80%99s_output. Accessed 27 Apr. 2020.
[51] "AutoML Visi.." https://cloud.google.com/vision/automl/docs/beginners-guide#what_is_the_score_threshold. Accessed 27 Apr. 2020.
[52] "AutoML Vision Beginner's guide - Google Cloud." https://cloud.google.com/vision/automl/docs/beginners-guide#what_are_precision_and_recall. Accessed 27 Apr. 2020.

12.1 Oxfam.org.uk Model Evaluation

The Oxfam.org.uk model delivered subpar results. As we can see in Table 12.3, at the 80% confidence threshold that we planned on using for our application resulted in a 89.97% precision rate but only 11.15% for recall. The most optimal confidence threshold in regards to precision and recall was 25%, delivering a precision rate of 53.91% and 54.30% recall rate.

Confidence	Precision	Recall
25%	53.91%	54.30%
50%	70.61%	31.16%
75%	88.32%	13.42%
80%	89.97%	11.15%

Table 12.3: Precision and recall for Oxfam.org.uk

In total we had 192 labels, 119 for brand, 59 for type and 14 for colour as we found out in Chapter 10.1.5. To get a better view of how the model works in regards with images that are above the recommended threshold, we looked at the precision and recall of the top five labels, by total images, to check out the scores.

As we can see in Table 12.4, when we set the confidence threshold at 80%, all the five labels have a high precision score with “Blue” the highest with 92.31% and “Grey” the lowest with 70.00%. The recall scores are however more interesting to look at, with “Black” and “Blue” having much higher scores than “Jacket”, “Grey” and “Brown”. “Grey” had the lowest recall score of 3.76% which combined with a precision score of 70.00% was extremely low.

Label	Total	Train	Validation	Test	Precision	Recall
Black	3.976	3.178	401	397	89.17%	35.26%
Blue	3.039	2.434	302	303	92.31%	39.60%
Jacket	2.052	1.640	204	208	85.00%	8.17%
Grey	1.854	1.483	185	186	70.00%	3.76%
Brown	1.290	1.032	129	129	84.62%	8.53%

Table 12.4: Precision and recall rates for top 5 labels in Oxfam.org.uk by total images

A more detailed analysis of the most optimal precision-recall tradeoff for the full model and the labels in Table 12.4 revealed that the tradeoffs recurred at a low precision score for all of them as we can see in Figs. 12.2 to 12.13, so we decided that this model would not be good enough.

Full Model precision-recall tradeoff

The full model had the most optimal tradeoff at 25% confidence level as we can see in Fig. 12.2.

At 80% threshold: Precision: 89.97%, Recall: 11.15%

At 25% threshold: Precision: 53.91%. Recall: 54.30%

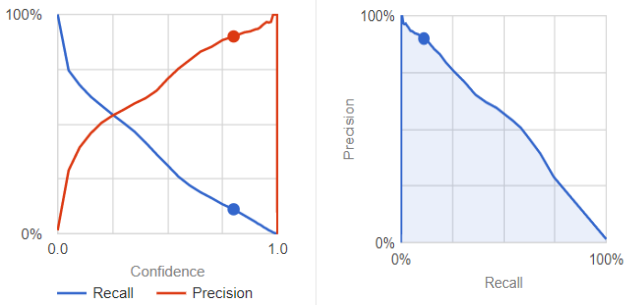


Figure 12.2: Full Model Recall/Precision

"Black" precision-recall tradeoff

The "Black" label had the most optimal tradeoff at 40% confidence level as we can see in Fig. 12.3.

At 80% threshold: Precision: 89.17%, Recall: 35.26%

At 40% threshold: Precision: 77.26%. Recall: 79.60%

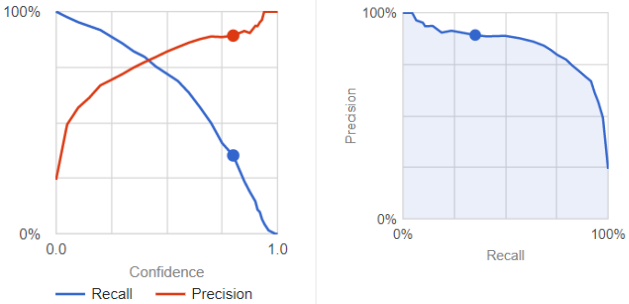


Figure 12.3: "Black" Recall/Precision

"Blue" precision-recall tradeoff

The "Blue" label had the most optimal tradeoff at 35% confidence level as we can see in Fig. 12.4.

At 80% threshold: Precision: 92.31%, Recall: 39.60%

At 35% threshold: Precision: 75.76%. Recall: 74.26%

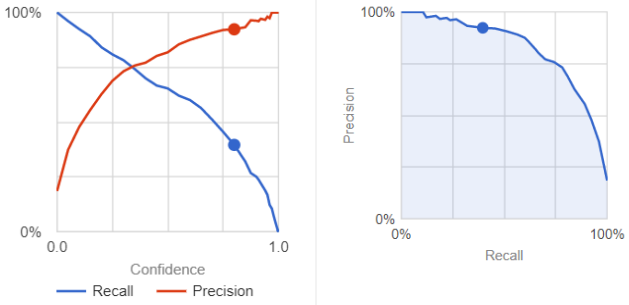


Figure 12.4: "Blue" Recall/Precision

"Jacket" precision-recall tradeoff

The "Jacket" label had the most optimal tradeoff at 40% confidence level as we can see in Fig. 12.5.

At 80% threshold: Precision: 85.00%, Recall: 8.17%

At 40% threshold: Precision: 63.18%. Recall: 61.06%

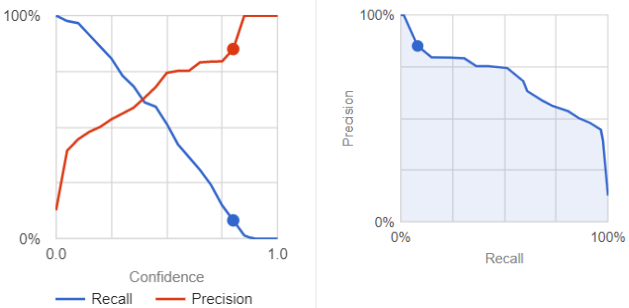


Figure 12.5: "Jacket" Recall/Precision

"Grey" precision-recall tradeoff

The "Grey" label had the most optimal tradeoff at 30% confidence level as we can see in Fig. 12.6.

At 80% threshold: Precision: 70.00%, Recall: 3.76%

At 30% threshold: Precision: 61.88%. Recall: 60.22%

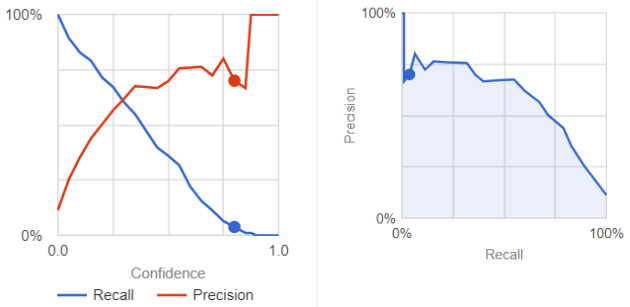


Figure 12.6: "Grey" Recall/Precision

"Brown" precision-recall tradeoff

The "Brown" label had the most optimal tradeoff at 25% confidence level as we can see in Fig. 12.7.

At 80% threshold: Precision: 84.62%, Recall: 8.53%

At 25% threshold: Precision: 56.03%. Recall: 50.39%

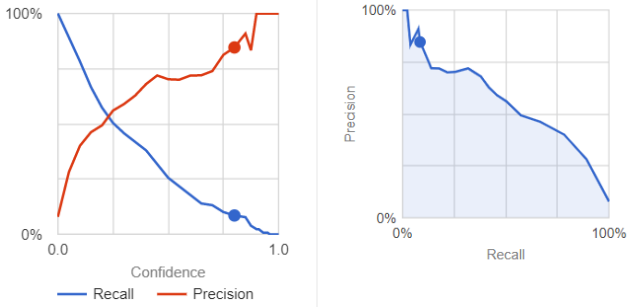


Figure 12.7: "Brown" Recall/Precision

12.2 Netflea.com Model Evaluation

The Netflea.com model also delivered subpar results. As we can see in Table 12.5, at the 80% confidence threshold that we planned on using for our application resulted in a 88.32% precision rate but only 4.70% for recall. The most optimal confidence threshold in regards to precision and recall was 20%, delivering a precision rate of 48.32% and 46.13% recall rate.

Confidence	Precision	Recall
25%	52.49%	41.68%
50%	67.16%	24.31%
75%	86.63%	6.81%
80%	88.32%	4.70%

Table 12.5: Precision and recall for Netflea.com

In total we had 470 labels, 168 for brand, 224 for type and 78 for colour as we found out in Chapter 10.2.6. To get a better view of how the model works in regards with images that are above the recommended threshold, we looked at the precision and recall of the top five labels, by total images, to check out the scores like we did with Oxfam.org.uk.

As we can see in Table 12.6, when we set the confidence threshold at 80%, only one label managed to return a prediction that had a higher confidence of 80%, that is the label “Black” with a 88.31% precision score and a 11.59% recall score. The other four had all correct predictions that returned a confidence score of <80%.

Label	Total	Train	Validation	Test	Precision	Recall
Black	12.938	10.345	1.290	1.303	85.31%	11.59%
H and M	9.262	7.410	926	926	100%	0%
Shirt	5.118	4.094	513	511	100%	0%
Blue	4.348	3.478	435	435	100%	0%
Gray	4.222	3.378	421	423	100%	0%

Table 12.6: Precision and recall rates for top 5 labels in Netflea.com by total images

A more detailed analysis of the most optimal precision-recall tradeoff for the full model and the labels in Table 12.6 revealed that the tradeoffs recurred at a low precision score for all of them as we can see in Figs. 12.14 to 12.25, so we decided that this model would not be good enough like we did with the Oxfam.org.uk model. This meant that we needed to focus on making a manual backup dataset.

Full Model precision-recall tradeoff

The full model had the most optimal tradeoff at 20% confidence level as we can see in Fig. 12.8.

At 80% threshold: Precision: 88.32%, Recall: 4.7%

At 20% threshold: Precision: 48.32%. Recall: 46.13%

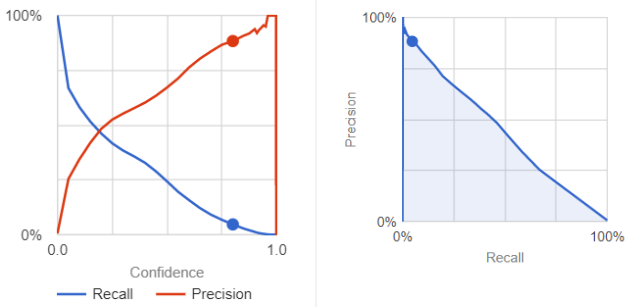


Figure 12.8: Full Model Recall/Precision

"Black" precision-recall tradeoff

The "Black" label had the most optimal tradeoff at 40% confidence level as we can see in Fig. 12.9.

At 80% threshold: Precision: 85.31%, Recall: 11.59%

At 40% threshold: Precision: 69.19%. Recall: 67.38%

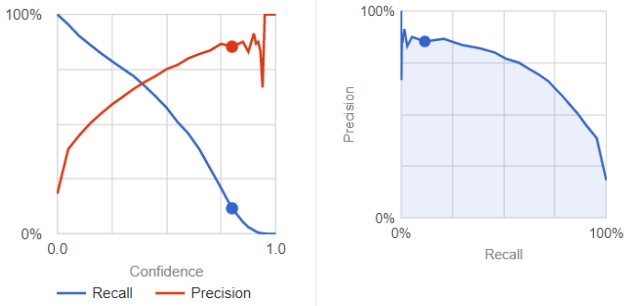


Figure 12.9: "Black" Recall/Precision

"H and M" precision-recall tradeoff

The "H and M" label had the most optimal tradeoff at 20% confidence level as we can see in Fig. 12.10.

At 80% threshold: Precision: 100%, Recall: 0%

At 20% threshold: Precision: 25.74%. Recall: 23.54%

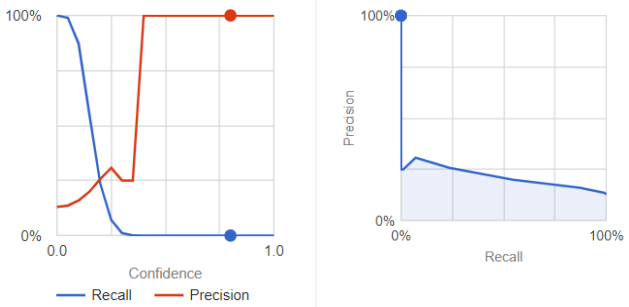


Figure 12.10: "H and M" Recall/Precision

"Shirt" precision-recall tradeoff

The "Shirt" label had the most optimal tradeoff at 25% confidence level as we can see in Fig. 12.11.

At 80% threshold: Precision: 100%, Recall: 0%

At 25% threshold: Precision: 38.54%. Recall: 38.16%

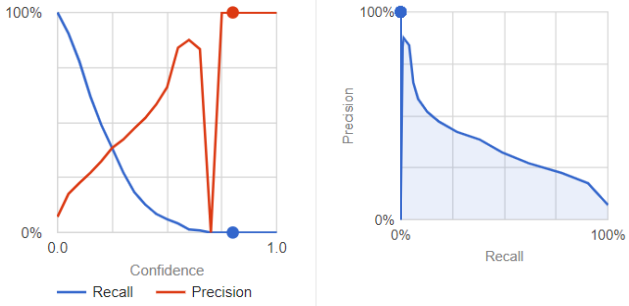


Figure 12.11: "Shirt" Recall/Precision

"Blue" precision-recall tradeoff

The "Blue" label had the most optimal tradeoff at 25% confidence level as we can see in Fig. 12.12.

At 80% threshold: Precision: 100%, Recall: 0%

At 25% threshold: Precision: 41.22%. Recall: 35.63%

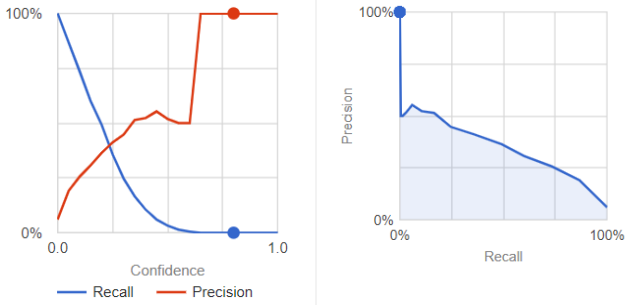


Figure 12.12: "Blue" Recall/Precision

"Gray" precision-recall tradeoff

The "Gray" label had the most optimal tradeoff at 20% confidence level as we can see in Fig. 12.13.

At 80% threshold: Precision: 100%, Recall: 0%

At 20% threshold: Precision: 45.71%. Recall: 45.39%

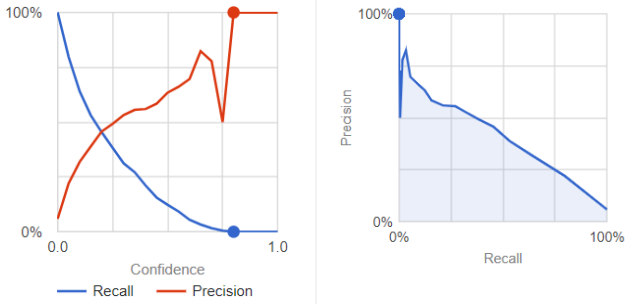


Figure 12.13: "Gray" Recall/Precision

12.3 Backup Model Evaluation

When it became clear that neither the Oxfam.org.uk model nor the Netflea.com model didn't work due to low precision scores and low recall scores, we decided to build a backup dataset. The backup model delivered great results, as we can see in Table 12.7, which shows the scores at the 80% confidence threshold.

Confidence	Precision	Recall
25%	96.86%	96.75%
50%	97.98%	95.82%
75%	98.90%	93.73%
80%	99.13%	92.8%

Table 12.7: Precision and recall for backup model

In total we had 10 labels, 2 for brand, 3 for type and 5 for colour as we found out in Chapter 10.3 Since this model only had a few labels, we wanted to make sure that it worked well for those labels.

Instead of looking at the top five labels in terms of images as we did with Netflea.com and Oxfam.org.uk, we decided to look at all the labels, since this model had quite high scores. The scores can be viewed in Table 12.8, which shows the precision score and recall score for each label at 80% confidence threshold.

Label	Total	Train	Validation	Test	Precision	Recall
Adidas	1.469	1.174	149	146	99.32%	99.32%
Nike	1.395	1.112	142	141	99.30%	100%
Hoodie	1.212	966	123	123	100%	78.05%
T-shirt	1.166	930	119	117	100%	98.29%
Black	740	590	76	74	100%	82.43%
Blue	699	558	71	70	100%	81.43%
Gray	623	498	63	62	93.85%	98.39%
Sweatshirt	486	390	49	47	97.78%	93.62%
Red	431	343	43	45	100%	100%
White	371	297	38	36	100%	94.44%

Table 12.8: Precision and recall rates for all 10 labels in backup model

A full detail of precision-recall tradeoff for the full model and labels can be viewed in Figs. 12.14 to 12.24.

Full Model precision-recall tradeoff

The full model had the most optimal tradeoff at 25% confidence level as we can see in Fig. 12.14.

At 80% threshold: Precision: 99.13%, Recall: 92.80%

At 25% threshold: Precision: 96.86%. Recall: 96.75%

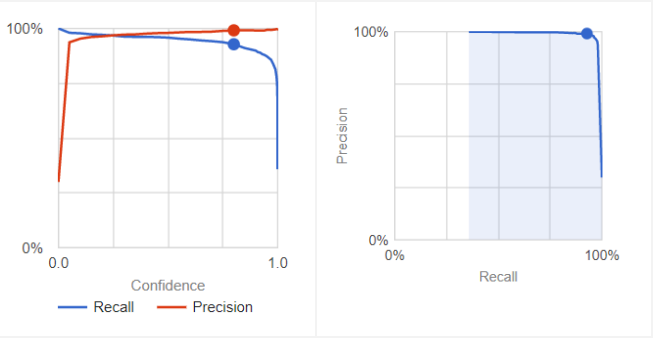


Figure 12.14: Full model Recall/Precision

"Adidas" precision-recall tradeoff

The "Adidas" label had the most optimal tradeoff at 55% to 99% confidence level as we can see in Fig. 12.15.

At 80% threshold: Precision: 99.32%, Recall: 99.32%

At 99% threshold: Precision: 99.32%. Recall: 99.32%

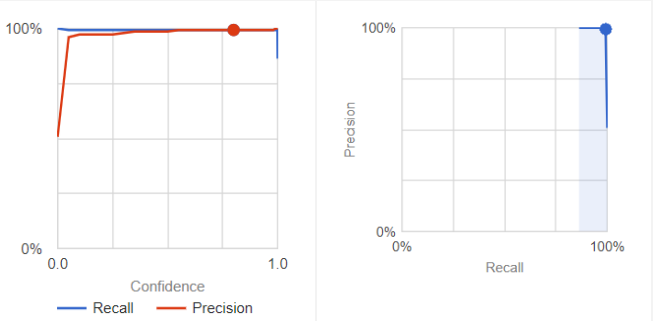


Figure 12.15: "Adidas" Recall/Precision

"Nike" precision-recall tradeoff

The "Nike" label had the most optimal tradeoff at 80% to 99% confidence level as we can see in Fig. 12.16.

At 80% threshold: Precision: 99.30%, Recall: 100%

At 99% threshold: Precision: 99.30%. Recall: 100%

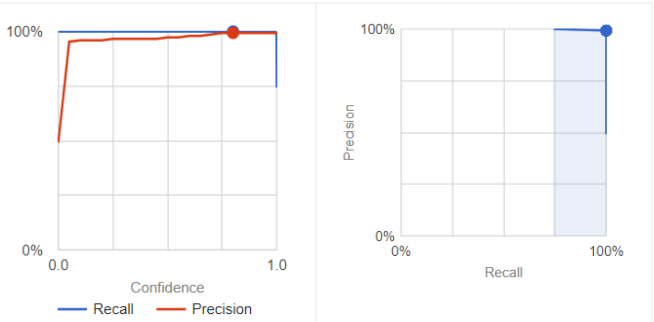


Figure 12.16: "Nike" Recall/Precision

"Hoodie" precision-recall tradeoff

The "Hoodie" label had the most optimal tradeoff at 5% confidence level as we can see in Fig. 12.17.

At 80% threshold: Precision: 100%, Recall: 78.05%

At 5% threshold: Precision: 96.75%. Recall: 100%

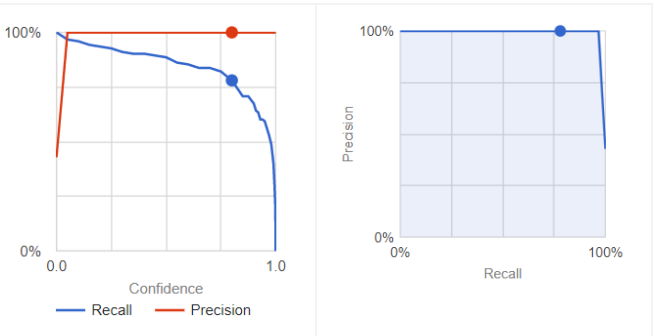


Figure 12.17: "Hoodie" Recall/Precision

"T-Shirt" precision-recall tradeoff

The "T-Shirt" label had the most optimal tradeoff at 5% to 60% confidence level as we can see in Fig. 12.18.

At 80% threshold: Precision: 100%, Recall: 98.29%

At 60% threshold: Precision: 99.15%. Recall: 99.15%

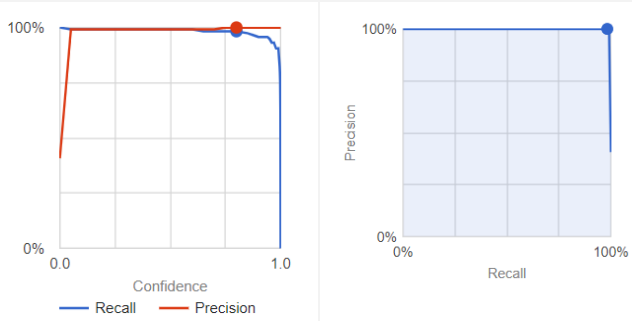


Figure 12.18: "T-Shirt" Recall/Precision

"Black" precision-recall tradeoff

The "Black" label had the most optimal tradeoff at 5% confidence level as we can see in Fig. 12.19.

At 80% threshold: Precision: 100%, Recall: 82.43%

At 5% threshold: Precision: 98.61%. Recall: 95.65%

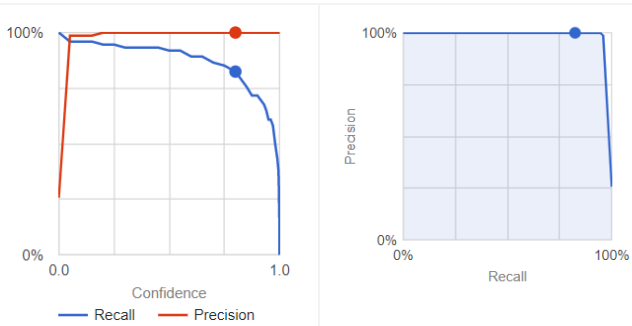


Figure 12.19: "Black" Recall/Precision

"Blue" precision-recall tradeoff

The "Blue" label had the most optimal tradeoff at 5% confidence level as we can see in Fig. 12.20.

At 80% threshold: Precision: 100%, Recall: 81.43%

At 5% threshold: Precision: 98.46%. Recall: 91.43%

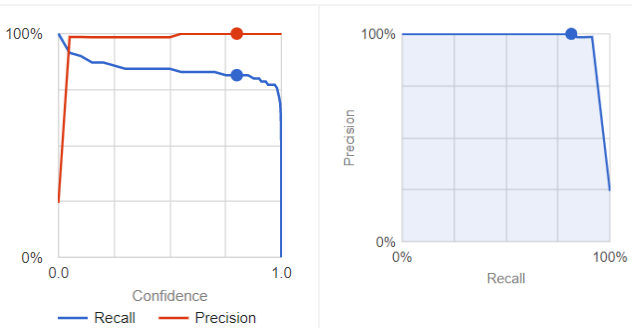


Figure 12.20: "Blue" Recall/Precision

"Gray" precision-recall tradeoff

The "Gray" label had the most optimal tradeoff at 97% confidence level as we can see in Fig. 12.21.

At 80% threshold: Precision: 93.85%, Recall: 98.39%

At 97% threshold: Precision: 96.77%. Recall: 96.77%

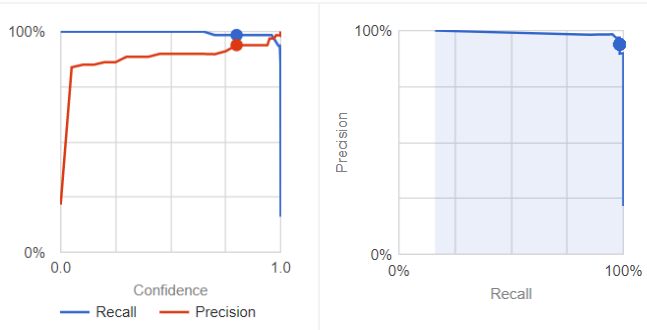


Figure 12.21: "Gray" Recall/Precision

"Sweatshirt" precision-recall tradeoff

The "Sweatshirt" label had the most optimal tradeoff at 45% to 55% confidence level as we can see in Fig. 12.22.

At 80% threshold: Precision: 97.78%, Recall: 93.62%

At 55% threshold: Precision: 95.74%. Recall: 95.74%

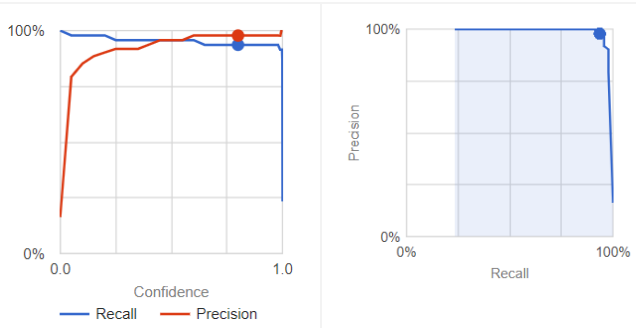


Figure 12.22: "Sweatshirt" Recall/Precision

"Red" precision-recall tradeoff

The "Red" label had the most optimal tradeoff at 40% to 99% confidence level as we can see in Fig. 12.23.

At 80% threshold: Precision: 100%, Recall: 100%

At 99% threshold: Precision: 100%. Recall: 100%

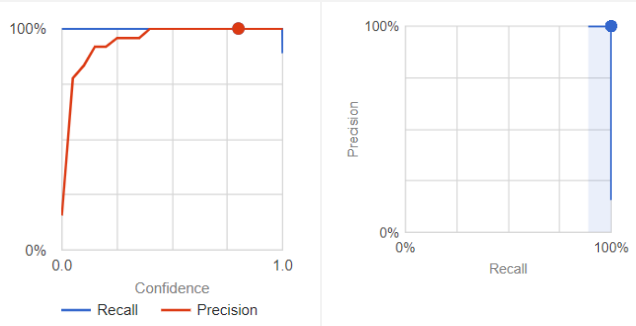


Figure 12.23: "Red" Recall/Precision

"White" precision-recall tradeoff

The "White" label had the most optimal tradeoff at 10% confidence level as we can see in Fig. 12.24.

At 80% threshold: Precision: 100%, Recall: 94.44%

At 10% threshold: Precision: 100%. Recall: 97.22%

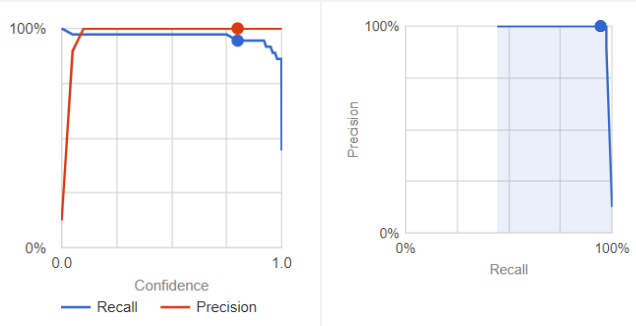


Figure 12.24: "White" Recall/Precision

The low optimal tradeoff in the "White" label can be attributed to some gray colours being close to the colour white, likewise the low optimal tradeoff in "Gray" label can be attributed to some white colours being close to the colour gray as well as the black colour being close to dark gray.

"Blue" and "Black" also fall in this category with dark blue being close to black.

One way to fix this is to gather more images for each shade of colour so that the model can accurately distinguish between them. We can see an example of how this works in the "Red" label. No colour is close to the colour "Red" so it has a 100% precision score and a 100% recall score.

The "Hoodie" also had a low optimal tradeoff. Many of the predictions fell beneath the 80% threshold so it would be wise to add more images to the dataset that contains hoodies to get a higher confidence score.

13 Testing Google AutoML Models

In the beginning we had two models that we wanted to test, Oxfam.org.uk model and Netflea.com model. We wanted to check how they stacked up against the general solutions that we tested in Chapter 4.

We used the same images to test the models that we used in Chapter 4, but after testing it became clear that the models weren't good enough, thus confirming our findings in Chapter 10 where we decided that the models weren't good enough based on the evaluation of them.

The evaluation of the backup model did exceptionally well, although it only worked on a small subset of clothing items, that is, the brands Adidas and Nike, the types hoodie, sweatshirt and t-shirt and the colours black, blue, gray, red and white.

We also wanted to see how the backup model stacked up against the general solutions, although it might only return for example "T-Shirt" for a yellow Puma t-shirt.

To test the models, we used the code that Google provided for testing in Python[53] where we plugged in the model id, the project id and changed the path to the local image we wanted to test at a given time.

We made a modification to the code so that we could authenticate with Google through the code which entailed a JSON document with a private key which we could use to authenticate, as we can see in Listing 13.1. This was done so that we wouldn't need to add an environment variable and thus making it easier to test the code on different computers.

```
credentials = service_account.Credentials.from_service_account_file("privatekey.json")

prediction_client = automl.PredictionServiceClient(credentials=credentials)
```

Listing 13.1: Modification to code given by Google to send a prediction request

In addition to the ten images we tested in Chapter 4, we also wanted to check how well our backup model worked on known labels, e.g. a black Adidas t-shirt or white Nike hoodie, so we added ten images that we found from Picclick.com that closely resembled the images our users would take and did testing on them as well. Images that were uploaded in the last 5 days were chosen so we wouldn't accidentally test an image that was in our dataset.

The test results from the ten images, five from ASOS and five from Craigslist, and the ten images from Picclick.com can be found in Tables 13.1 to 13.20.

[53] "Making individual predictions | Cloud AutoML Vision | Google ...." https://cloud.google.com/vision/automl/docs/predict#automl_vision_classification_predict-python. Accessed 26 Apr. 2020.

13.1 AutoML Test Reports

ASOS-1 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.1: ASOS-1	**Type:** Hoodie **Colour:** Black **Brand:** Adidas	No results	No results	Hoodie: 99% Black: 99% Adidas: 99%
	Description: A male model wearing a black Adidas hoodie. The Adidas logo is visible.	**Comment:** No results obtained by model	**Comment:** No results obtained by model	**Comment:** Correctly tagged the type as hoodie, colour as black and brand as Adidas, all with 99% confidence.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		✕	✕	✕
Netflea.com		✕	✕	✕
Backup		✓	✓	✓

Table 13.1: ASOS-1 AutoML Test Report

ASOS-2 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.2: ASOS-2	**Type:** Dress **Colour:** Pink **Brand:** PrettyLittleThing	No results	No results	White: 99% Adidas: 99% Red: 89%
	Description: A female model wearing a pink dress, with black laces. The logo is not visible.	**Comment:** No results obtained by model	**Comment:** No results obtained by model	**Comment:** Incorrectly tagged all the tags wrong with 89% - 99%.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		✕	✕	✕
Netflea.com		✕	✕	✕
Backup		✕	✕	✕

Table 13.2: ASOS-2 AutoML Test Report

ASOS-3 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.3: ASOS-3	**Type:** Hoodie **Colour:** White / Black **Brand:** Puma	No results	No results	White: 99% Adidas: 99%
	Description: A male model wearing a white/black Puma hoodie. The logo is visible.	**Comment:** No results obtained by model	**Comment:** No results obtained by model	**Comment:** Correctly tagged the colour whitewith 99% confidence but iIncorrectly tagged the brand as Adidas with 99% confidence.

Prediction Summary			
Model	Type	Brand	Colour
Oxfam.org.uk	×	×	×
Netflea.com	×	×	×
Backup	×	×	✓

Table 13.3: ASOS-3 AutoML Test Report

ASOS-4 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.4: ASOS-4	**Type:** Top **Colour:** Black / Light Blue **Brand:** Vero Moda	No results	No results	Adidas: 99%
	Description: A female model wearing a light blue/black top from Vero Moda. The logo is not visible.	**Comment:** No results obtained by model	**Comment:** No results obtained by model	**Comment:** Incorrectly tagged the brand as Adidas with 99% confidence.

Prediction Summary			
Model	Type	Brand	Colour
Oxfam.org.uk	×	×	×
Netflea.com	×	×	×
Backup	×	×	×

Table 13.4: ASOS-4 AutoML Test Report

ASOS-5 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.5: ASOS-5	**Type:** Jeans **Colour:** Blue **Brand:** Vero Moda	Blue: 99%	No results	Sweatshirt: 99% Adidas: 99%
	Description: A female model wearing blue jeans from Vero Moda. The logo is not visible.	**Comment:** Correctly tagged the colour as blue with 99% confidence.	**Comment:** No results obtained by model	**Comment:** Incorrectly tagged the type as Sweatshirt and incorrectly tagged the brand as Adidas with 99% confidence.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		✗	✗	✓
Netflea.com		✗	✗	✗
Backup		✗	✗	✗

Table 13.5: ASOS-5 AutoML Test Report

CRAIGSLIST-1 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.6: CRAIGSLIST-1	**Type:** Shirt Long Sleeve **Colour:** Tan **Brand:** Knightsbridge	No results	No results	Gray: 93% Sweatshirt: 99% Adidas: 99%
	Description: A Knightsbridge dress shirt hung on a hanger. The logo is not visible.	**Comment:** No results obtained by model	**Comment:** No results obtained by model	**Comment:** Incorrectly tagged the colour as gray, the type as sweatshirt and the brand as Adidas with 99% confidence.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		✗	✗	✗
Netflea.com		✗	✗	✗
Backup		✗	✗	✗

Table 13.6: CRAIGSLIST-1 AutoML Test Report

CRAIGSLIST-2 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.7: CRAIGSLIST-2	**Type:** Long sleeve top **Colour:** Gray **Brand:** Nike	No results	No results	Gray: 99% Nike: 100% Sweatshirt: 99%
	Description: A gray Nike long sleeved top. The logo is visible.	**Comment:** No results obtained by model	**Comment:** No results obtained by model	**Comment:** Correctly tagged the colour and brand with 99% to 100% confidence.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		✗	✗	✗
Netflea.com		✗	✗	✗
Backup		✗	✓	✓

Table 13.7: CRAIGSLIST-2 AutoML Test Report

CRAIGSLIST-3 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.8: CRAIGSLIST-3	**Type:** T-Shirt **Colour:** Blue **Brand:** Nike	Blue: 98%	No results	T-Shirt: 99% Nike: 100% Blue: 99%
	Description: A blue Nike T-shirt with Nike logo on front. The logo is visible.	**Comment:** Correctly tagged the colour as blue with 98% confidence.	**Comment:** No results obtained by model	**Comment:** Correctly tagged the type, colour and brand with 99% to 100% confidence.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		✗	✗	✓
Netflea.com		✗	✗	✗
Backup		✓	✓	✓

Table 13.8: CRAIGSLIST-3 AutoML Test Report

CRAIGSLIST-4 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
	**Type:** Jacket **Colour:** Black **Brand:** North Face	No results	No results	Hoodie: 99% Adidas: 99%
Figure 13.9: CRAIGSLIST-4	**Description:** A black Denali jacket from North Face. The logo is visible.	**Comment:** No results obtained by model	**Comment:** No results obtained by model	**Comment:** Incorrectly tagged the type as hoodie and the brand as Adidas with 99% confidence.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		×	×	×
Netflea.com		×	×	×
Backup		×	×	×

Table 13.9: CRAIGSLIST-4 AutoML Test Report

CRAIGSLIST-5 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
	**Type:** Long sleeve shirt **Colour:** White/Black **Brand:** Under Armour	No results	No results	Adidas: 99%
Figure 13.10: CRAIGSLIST-5	**Description:** A white/black Under Armour plaid shirt. The logo is not visible.	**Comment:** No results obtained by model	**Comment:** No results obtained by model	**Comment:** Incorrectly tagged the brand as Adidas with 99% confidence.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		×	×	×
Netflea.com		×	×	×
Backup		×	×	×

Table 13.10: CRAIGSLIST-5 AutoML Test Report

BACKUP-1 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.11: BACKUP-1	**Type:** Hoodie **Colour:** Black **Brand:** Adidas	No results	Sweatshirt: 91%	Adidas: 99%
	Description: A black hoodie with Adidas logo on front. The logo is visible.	**Comment:** No results obtained by model	**Comment:** Incorrectly tagged the type as sweatshirt with 91% confidence.	**Comment:** Correctly tagged the brand as Adidas with 99% confidence.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		✗	✗	✗
Netflea.com		✗	✗	✗
Backup		✗	✓	✗

Table 13.11: BACKUP-1 AutoML Test Report

BACKUP-2 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.12: BACKUP-2	**Type:** T-Shirt **Colour:** White **Brand:** Adidas	No results	T-Shirt: 95%	T-Shirt: 99% White: 99% Adidas: 100%
	Description: A white Adidas T-shirt with logo on front. The logo is visible.	**Comment:** No results obtained by model	**Comment:** Correctly tagged the type as t-shirt with 95% confidence.	**Comment:** Correctly tagged the type, colour and brand with 99% to 100% confidence.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		✗	✗	✗
Netflea.com		✓	✗	✗
Backup		✓	✓	✓

Table 13.12: BACKUP-2 AutoML Test Report

BACKUP-3 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.13: BACKUP-3	**Type:** Sweatshirt **Colour:** Red **Brand:** Adidas	Red: 95%	Sweatshirt: 91%	Sweatshirt: 99% Adidas: 99% Red: 100%
	Description: A red Adidas sweatshirt with a logo on front. Logo is visible.	**Comment:** Correctly tagged the colour as red with 95% confidence.	**Comment:** Incorrectly tagged the type as sweatshirt with 91% confidence.	**Comment:** Correctly tagged the type, colour and brand with 99% to 100% confidence.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		✗	✗	✓
Netflea.com		✗	✗	✗
Backup		✓	✓	✓

Table 13.13: BACKUP-3 AutoML Test Report

BACKUP-4 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.14: BACKUP-4	**Type:** Sweatshirt **Colour:** Blue **Brand:** Adidas	Blue: 92%	No results	Sweatshirt: 99% Blue: 99% Adidas: 99%
	Description: A blue Adidas sweatshirt with Adidas logo on front. The logo is visible.	**Comment:** Correctly tagged the colour as blue with 92% confidence.	**Comment:** No results obtained by model	**Comment:** Correctly tagged the type, colour and brand with 99% confidence.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		✗	✗	✓
Netflea.com		✗	✗	✗
Backup		✓	✓	✓

Table 13.14: BACKUP-4 AutoML Test Report

BACKUP-5 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.15: BACKUP-5	**Type:** Hoodie **Colour:** Gray **Brand:** Adidas	No results	No results	Gray: 99% Hoodie: 99% Adidas: 100%
	Description: A gray Adidas hoodie with Adidas logo on front. The logo is visible.	**Comment:** No results obtained by model	**Comment:** No results obtained by model	**Comment:** Correctly tagged the type, colour and brand with 99% to 100% confidence.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		✗	✗	✗
Netflea.com		✗	✗	✗
Backup		✓	✓	✓

Table 13.15: BACKUP-5 AutoML Test Report

BACKUP-6 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.16: BACKUP-6	**Type:** T-Shirt **Colour:** Blue **Brand:** Nike	Blue: 96%	No results	T-Shirt: 99% Nike: 100% Blue: 99%
	Description: A blue Nike t-shirt with Nike logo on front. The logo is visible.	**Comment:** Correctly tagged the colour as blue with 96% confidence.	**Comment:** No results obtained by model	**Comment:** Correctly tagged the type, colour and brand with 99% to 100% confidence.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		✗	✗	✓
Netflea.com		✗	✗	✗
Backup		✓	✓	✓

Table 13.16: BACKUP-6 AutoML Test Report

BACKUP-7 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.17: BACKUP-7	**Type:** T-Shirt **Colour:** White **Brand:** Nike	No results	T-Shirt: 81%	T-Shirt: 99% Nike: 100% White: 100%
	Description: A white Nike t-shirt with Nike logo on front. The logo is visible.	**Comment:** No results obtained by model	**Comment:** Correctly tagged the type as t-shirt with 81% confidence.	**Comment:** Correctly tagged the type, colour and brand with 99% to 100% confidence.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		✕	✕	✕
Netflea.com		✓	✕	✕
Backup		✓	✓	✓

Table 13.17: BACKUP-7 AutoML Test Report

BACKUP-8 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.18: BACKUP-8	**Type:** Sweatshirt **Colour:** Blue **Brand:** Nike	Blue: 96%	No results	Hoodie: 99% Nike: 100% Blue: 99%
	Description: A blue Nike sweatshirt with Nike logo on front. The logo is visible.	**Comment:** Correctly tagged the colour as blue with 96% confidence.	**Comment:** No results obtained by model	**Comment:** Correctly tagged the type, colour and brand with 99% to 100% confidence.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		✕	✕	✓
Netflea.com		✕	✕	✕
Backup		✓	✓	✓

Table 13.18: BACKUP-8 AutoML Test Report

BACKUP-9 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.19: BACKUP-9	**Type:** Sweatshirt **Colour:** Blue **Brand:** Nike	Blue: 97%	No results	Nike: 100% Sweatshirt: 99% Blue: 99%
	Description: A blue Nike sweatshirt with a logo on front. Logo is visible.	**Comment:** Correctly tagged the colour as blue with 97% confidence.	**Comment:** No results obtained by model	**Comment:** Correctly tagged the type, colour and brand with 99% to 100% confidence.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		✗	✗	✓
Netflea.com		✗	✗	✗
Backup		✓	✓	✓

Table 13.19: BACKUP-9 AutoML Test Report

BACKUP-10 AutoML Test Report				
Image	Correct Tags	Oxfam.org.uk	Netflea.com	Backup
 Figure 13.20: BACKUP-10	**Type:** Hoodie **Colour:** Black **Brand:** Nike	No results	No results	Black: 99% Nike: 100%
	Description: A black Nike hoodie with Nike logo on front. The logo is visible.	**Comment:** No results obtained by model	**Comment:** No results obtained by model	**Comment:** Correctly tagged the colour as black with 99% confidence and the brand as Nike with 100% confidence.
Prediction Summary				
Model		Type	Brand	Colour
Oxfam.org.uk		✗	✗	✗
Netflea.com		✗	✗	✗
Backup		✗	✓	✓

Table 13.20: BACKUP-10 AutoML Test Report

13.1 AutoML Test Conclusion

Testing the first two models, Oxfam.org.uk and Netflea.com, showed that the models were not good enough to give an accurate prediction when the confidence threshold was 80% or above, which all the tests required. The backup model did well, given that the images were in the same format as the images in our dataset. It did however give false positives with high confidence, e.g. 99%.

As we can see in Table 13.21, predicting the five images from ASOS did not go well. Netflea.com wasn't able to predict a single label. Oxfam.org.uk managed to correctly predict a single colour label and the backup model managed to predict one type, one brand and two colours.

Model	Correct Types	Correct Brands	Correct Colours
Oxfam.org.uk	0/5	0/5	1/5
Netflea.com	0/5	0/5	0/5
Backup	1/5	1/5	2/5

Table 13.21: Prediction Summary for ASOS tests

However the dataset contained one image, Fig. 13.1, which was a black Adidas hoodie and we wanted to know how the backup model would compare against the vision AI providers. The backup model was able to accurately predict all the labels, whereas Google only managed to predict it correctly with a lot of false positives and Microsoft only managed to predict the colour and the type with false positives.

Predicting the five images from Craigslist didn't fare better as we can see in Table 13.22. Netflea.com again didn't manage to make a single prediction and Oxfam.org.uk again managed only to make a single correct prediction on the colour. The backup model managed to predict one type, two brands and two colours.

Model	Correct Types	Correct Brands	Correct Colours
Oxfam.org.uk	0/5	0/5	1/5
Netflea.com	0/5	0/5	0/5
Backup	1/5	2/5	2/5

Table 13.22: Prediction Summary for Craigslist tests

Here we were most interested in seeing how the backup model would predict on two images, Fig. 13.7 and Fig. 13.8 as they contained images that closely resembled the images our users would take. The backup model managed to predict all the labels correctly on the second image, while predicting the colour and brand correctly on the first image. The type in the first image was not a label in the dataset which makes it understandable that it didn't manage to predict correctly on it. In comparison with the Vision AI providers, both Google and Microsoft only managed to predict the colour correctly in the first image and in the second image, Google managed to predict the type and colour correctly while Microsoft only managed the colour.

Predicting the ten images we downloaded from Craigslist to test the backup model fared much better for all models, although as predicted, it fared exceptionally well for the backup model as we can see in Table 13.23. Oxfam.org.uk managed to predict the colour five times out of ten, although it didn't manage to predict a single label in types and brands, and Netflea finally was able to predict something, although only the correct type on two occasions and unable to return a prediction for a brand or a colour. The backup model however managed to predict the type on eight out of ten occasions, the brand in all ten images and the colour in nine out of the ten images.

Model	Correct Types	Correct Brands	Correct Colours
Oxfam.org.uk	0/10	0/10	5/10
Netflea.com	2/10	0/10	0/10
Backup	8/10	10/10	9/10

Table 13.23: Prediction Summary for Craigslist backup tests

Since we already decided to use the backup model, we wanted to know what it was not being able to predict. Here it was unable to come up with a prediction for the type on two occasions, for Fig. 13.11 and Fig. 13.20. Interestingly, both of the figs. are hoodies and looking into why it wasn't able to predict a hoodie on those two images, we noticed that it was giving out a prediction as a hoodie with sub 80% confidence, which means that we need to add more images of hoodies to our dataset to increase the precision score for hoodies.

We also noticed that the model gave out correct predictions for hoodies in Figs. 13.15 and 13.18. We then noticed that the difference between the images that it gave a prediction with sub 80% confidence and above 80% confidence was the distance that the pictures were taken from. From this, we can assume that we need to add more images of hoodies from different distances, especially far away so it will give a prediction with a higher confidence score.

The model also didn't manage to predict the colour on one of the images, for Fig. 13.11. The correct prediction is black and we already concluded in the evaluation process of the model that it would perhaps have difficulty predicting the color black as it was close to the colour dark gray.

Putting in a lower confidence threshold, the model managed to accurately predict the colour as black with a sub 80% confidence score, which strengthens our conclusion that the model had a problem distinguishing between the colour black and dark gray.

Interestingly, the model managed to predict correctly on the brand on all occasions, which leads to the conclusion that the model has enough images with tagged brands to accurately predict it correctly with an above 80% confidence score.

14 The Past and the Future

14.1 The Past

In the beginning of the project, we thought that the machine learning part of the app would be plug and play and somewhat straightforward by using either Microsoft Azure or Google Cloud Vision AI. But to make sure that they worked before we started coding the app, we tested them to check if they were able to make correct predictions as they were thought to be used in a general sense, not in a niche sense as we were planning on using them. We decided to test five images from ASOS which contained five professional models wearing the clothes and five images from Craigslist, which contained images of used clothing uploaded by sellers. The rationale for including the images from Craigslist was that even if the providers provided accurate predictions for the images from ASOS, it was perhaps not as well equipped to make accurate predictions on images that were in our niche, that is images that were taken by users with the clothing item laid down.

The testing phase of the providers showed that Google was well equipped to make accurate predictions for the clothing items from the ASOS dataset, however it didn't fare as well for the Craigslist dataset. Since Microsoft didn't fare better, we were in kind of a dilemma. We thought that we would be able to use a general service provided by either Google or Microsoft but the testing showed that this would not be the case, which meant that either we could use the general service that provided inaccurate results or we could try to find a solution that worked for our needs. The latter meant that we would need to allocate significant more time to the machine learning research than we originally estimated, and meant that this research paper, which was part of the hand in to K3, would be significantly larger than previously thought.

After consultation with both our instructor and our Project Owner, we decided that we would try to utilize a new technology called Automated Machine Learning (AutoML). AutoML doesn't require the machine learning knowledge that we lacked and we could build a model to make predictions built on a dataset that related to our niche of used clothing articles.

We decided to utilize Google AutoML for building a model since we had experience with working with Google products compared with none with Microsoft products and after consultation with a machine learning engineer that concluded that the Google AutoML solution would fit us better.

To build a dataset that worked with our niche, we first planned on using a known dataset that contained tagged images of used clothing. After consultation with our Product Owner asking him if he knew of such a dataset, we came to the conclusion that there were no such dataset that fitted our needs. This led us to start looking at websites that we could scrape to build our dataset.

This search led us to two websites, Oxfam.org.uk and Netflea.com, which both offered used clothing for sale and had tags of each clothing on the site. Instead of choosing to scrape either of the sites to build a dataset, we decided that we would scrape both sites and build separate datasets that we could compare and use the dataset that gave more accurate predictions.

Scraping both of these sites went well and after cleaning the data and then importing the images and a .csv document with tags for each image so we could train the images, it became clear in the evaluation of the training of the datasets that the models built on these datasets were not good enough. After further inspection we noticed that both datasets had incorrect tags on several occasions and both datasets had too few images for each label than recommended by Google.

This led us to another dilemma, we could finetune the models by going through each image to check if they were correctly tagged and lower the labels by combining labels that fitted together, but we ultimately deemed that this would take too much time, and ultimately weren't sure if it would even work.

After consultation with our instructor, he told us that we could try and make the app work with a subset of clothing items first instead of trying to make the app work on the full set of clothing as we envisioned in the beginning. We decided that we would try to make it work on a few brands, a few colours and a few types.

For this we decided that we would make the app work on two brands, "Adidas" and "Nike", on three types, "Hoodie", "Sweatshirt" and "T-Shirt", and five colours, "Black", "Blue", "Gray", "Red" and "White". The focus on building a dataset for this small subset enabled us to find a large amount of images for each label, which would potentially mean that Google would be able to make accurate predictions based on this subset.

To find the images that we needed for our dataset, we decided that we would focus on images that resembled the images that would be taken by our users as close as possibly, that is only including images that were of clothing that were laid down, thus eliminating the images that were hanged up for example on a hanger or the seller were wearing them.

We decided that we would find websites that had images in that format and would be downloading them and then manually tag them to minimize the risk of the same thing happening in regards with incorrect tags when we scraped Oxfam.org.uk and Netflea.com. The search of finding these images led us to four websites, Grailed.com, Shpock.com, Picclick.com and Poshmark.com, although Shpock.com only delivered a few images. In total we gathered 2.886 manually tagged images for our dataset and after importing the images and a .csv document with the tags for each image, we started training a model based on the dataset.

The evaluation of the backup model was really encouraging with a 99.13% precision score and a 92.80% recall score at 80% confidence threshold, compared to 89.97% precision score and a 11.15% recall score for the Oxfam.org.uk model and a 88.32% precision score and a meager 4.70% recall score for the Netflea.com model.

Testing the backup model with ten images showed that the model was well equipped to work on the niche that we had, and only missed the type on two occasions of the ten images and the colour once. It managed to accurately predict the brand on all occasions.

These findings resulted in us using the backup model to serve predictions for our app.

14.2 The Future

Currently we have a working model that works on two brands, “Adidas” and “Nike”, three types, “Hoodie”, “Sweatshirt” and “T-Shirt”, and five colours, “Black”, “Blue”, “Gray”, “Red” and “White”.

However, the model does not work perfectly on a few of these tags. As we noted in the testing phase, the model does not give a prediction on the type “Hoodie” with a high enough confidence score. This can be improved by importing more images of hoodies.

The model also has some trouble distinguishing between a few colours. Black closely resembles the dark gray colour, black also closely resembles the dark blue colour and light gray closely resembles the white colour. This can be improved by importing images with each colour to bring the total number for each colour that closely resembles each other higher than it currently is, thus making the model able to make a prediction with a higher confidence score and enabling it to distinguish between closely resembled colours better.

Once the model has been improved to work better on the current labels, we can add to the labels, e.g. adding the brand “Puma” to the model. Doing so would however require over 1.000 images of Puma clothing to the dataset.

We would recommend building on the dataset label by label, thus making the model work for each added label before we add on the next label.

One of the main disadvantages of having the model predicting on brands is that a large subset of clothing items don’t have a brand visible. For example if we look at the “H&M” brand, most of their clothing don’t have the H&M brand visible on their clothing, thus making it near impossible for a machine learning model to learn to recognize the brand. If it is decided to use the brand, the contingency plan must be that the brand will come up empty for a large number of predictions that the model makes on brands.