FocusedFashion Group Report

Mary Gibbs and Jessica Fogerty

Introduction

We all have our favorite pieces of clothing that we consistently wear. Over time, these clothing items may not fit anymore or degrade in quality. Subsequently, it may be difficult to find the same clothing item or time-intensive to find similar clothing items due to the vast amount of clothing websites and clothing retail stores. For example, the clothing retailer H&M has hundreds of fashion options available on their website and a subpar filtering system, making it difficult to find clothing pieces of interest. This is a common issue for both of us, and it is something we attempted to streamline by using deep learning techniques. Our solution involves a two-step approach. First, we trained convolutional neural networks (CNN) on fashion images in order to extract the feature maps associated with different clothing items. Second, we used those feature maps as inputs to a k-nearest neighbors (KNN) model that found the five closest neighbors to a given query image that served as recommendations.

Prior research has found that utilizing deep learning techniques in fashion recommendation systems has proven to be better than traditional recommendation techniques. Le (2019) trained a ResNet neural network on tops to classify these clothing images into six classes and then obtain feature maps. Then, the author implemented a nearest-neighbor based search on the feature maps. Similarly, Tuinhof, Pirker, & Haltmeier (2018) trained fashion images using AlexNet and BN-inception to extract feature maps. They used those feature maps to implement a KNN to return ranked recommendations. Our solution is similar to the previous literature except that we experimented with different network architectures and KNN implementations/distance metrics in the hopes of achieving efficient and superior fashion recommendations.

The remainder of the paper is structured as follows. Dataset includes a description of the dataset and preprocessing techniques. Experimental Setup details the project set up and an overview of the project strategy. Models provides a detailed comparative analysis of neural networks trained during this project and the KNN model recommendation results. Results provides a numerical analysis of model results and an evaluation of the clothing recommendations. Conclusion summarizes the project and discusses limitations and future work.

Dataset

The dataset used for this project comes from the Kaggle competition iMaterialist Challenge (Fashion) at FGVC5. It comprises 1,014,000 train images, 10,586 validation images, and 42,590 test images with each image possessing multiple labels constituting a total of 228 unique labels. The dataset comes in the format of three JSON files that each contain an image_id, url, and label_id (iMaterialist Challenge (Fashion) at FGVC5, 2018; Visipedia, 2019). To download the dataset, we adapted a dataset downloading script from the competition website that loops through the image urls and downloads the corresponding images (Nlecoy, 2018). We decided to subset the dataset to 500,000 train images, 10,000 validation images, and 30,000 test images. Subsequently, it took around 6 hours to download all of the images. After training our initial baseline CNN model, we decided to further subset the dataset to 300,000 train images to reduce training time (Figure 1).

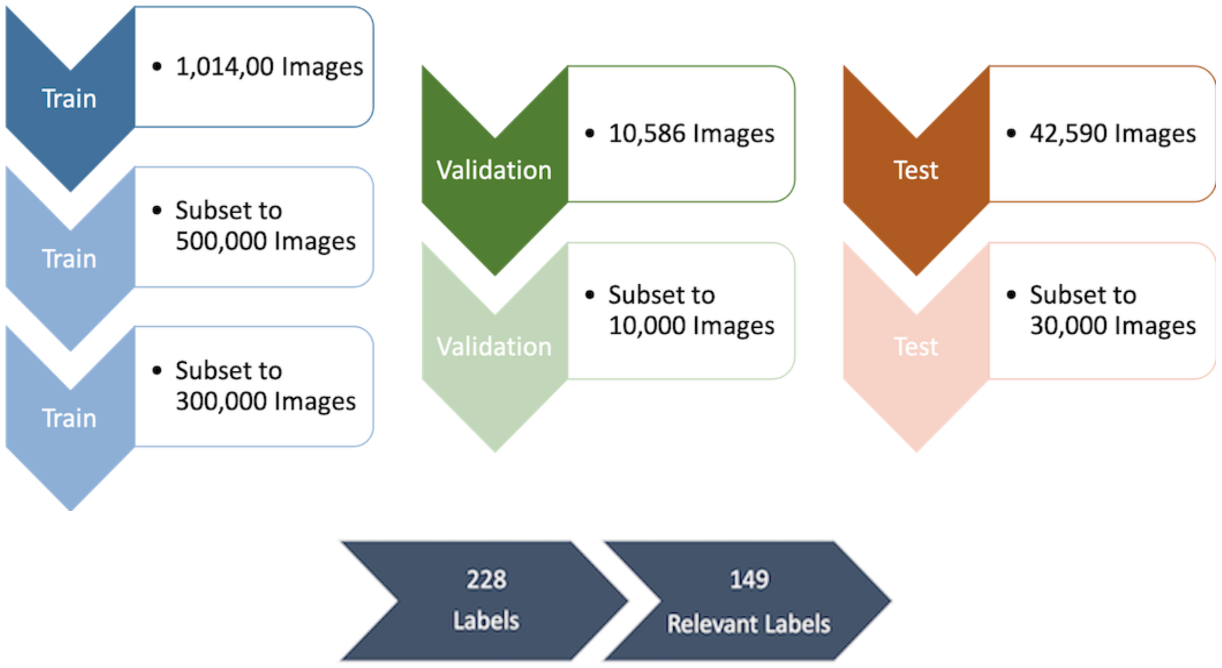


Figure 1. A summary of dataset preprocessing.

Additionally, we removed labels that were irrelevant to the clothing items that we were interested in recommending, such as shoelaces and wedding dresses, for a total of 149 unique labels. We ensured that all 1abels were present in the train, validation, and test sets. To gain some insights regarding the prevalence and importance of labels, we performed an exploratory data analysis. We decided to assess the distribution of labels per image within the training set (Figure 2).

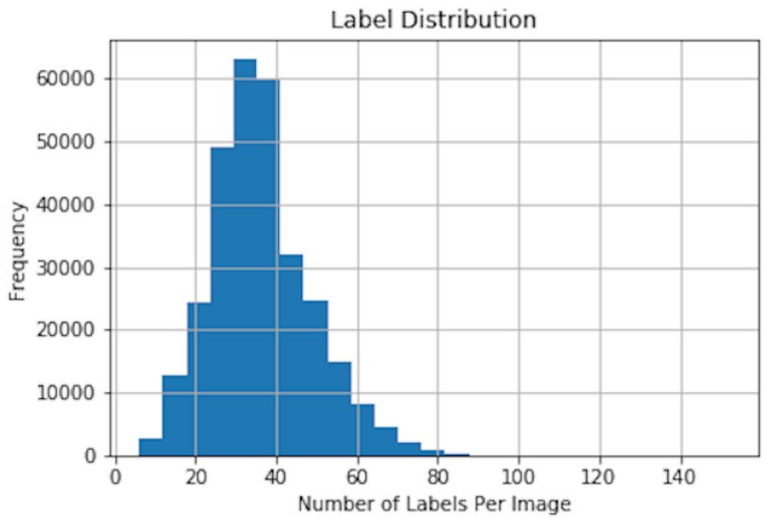


Figure 2. The distribution of labels per image within the training set.

The average number of labels per image is 36, while the maximum is 142 and the minimum is 6. Then, we created a word cloud in order to evaluate the frequency of the labels within the training set (Figure 3).

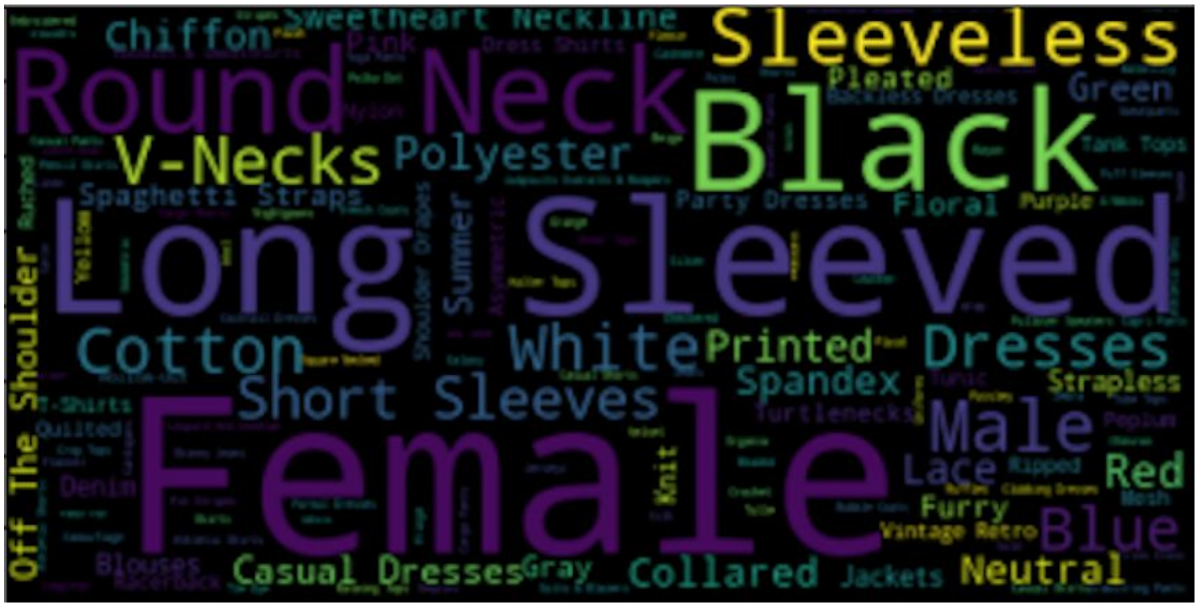


Figure 3. A word cloud, highlighting the frequency of the labels within the training set.

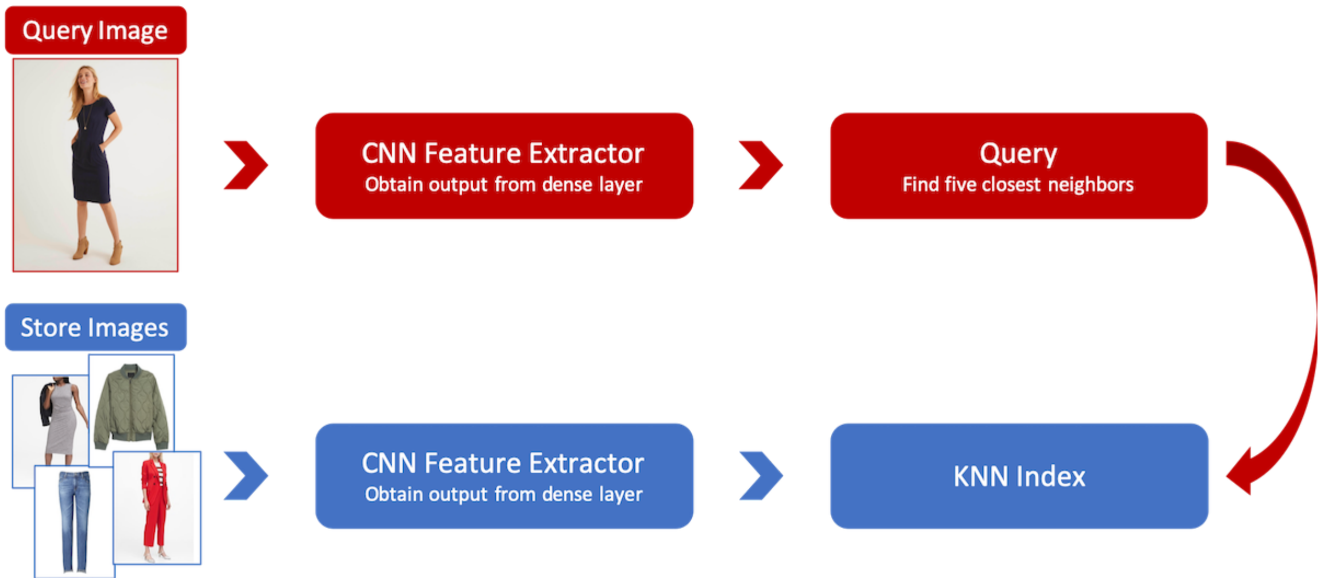
The size of the label text indicates the frequency of the label. From this word cloud, it is apparent that the labels female, long sleeved, black, round neck, and sleeveless appear to be the most prevalent labels, while purple, tank tops, and t-shirts appear to be less prevalent labels.

Ultimately, this word cloud indicates that there is a label imbalance within the training set that may potentially influence our fashion recommendations.

Experimental Setup

As previously described, the first step in our approach involves creating and training CNNs that will be utilized for feature extraction and subsequent KNN ranked recommendations. For this purpose, we decided to use an NVIDIA Tesla P100 GPU on Google Cloud Platform and the PyTorch deep learning framework due to its ease of use and implementation of dynamic computation graphs that make it a more flexible framework. A discussion of CNN model architecture and hyperparameter tuning can be found in the Models section below. To evaluate the CNN models, we decided to assess binary cross-entropy with logits loss on the validation set and micro-averaged F1-score on the test set since these metrics have been previously cited in the literature for multilabel classification problems and used for evaluation in the Kaggle competition iMaterialist Challenge (Fashion) at FGVC5 (iMaterialist Challenge (Fashion) at FGVC5, 2018). Binary cross-entropy with logits loss combines a sigmoid activation plus cross-entropy loss. It is ideal for multilabel classification because it is independent, meaning that it is calculated for every vector in the CNN output layer, so that the calculation pertaining to one label does not influence the calculation pertaining to another label (Gómez, 2018). Micro-averaged F1-score takes into consideration the distribution of the labels in order to calculate the average F1-score, making it a reasonable metric for our dataset since there is a label imbalance (sklearn.metrics.f1_score, n.d.). The CNN models with the best scores in regards to both of these metrics were utilized in the recommendation process.

As previously mentioned, the second step in our approach involves using the trained CNNs and KNN to find the five closest neighbors to a given query image and return recommendations. In order to do this, we acquired an example fashion image that we were interested in obtaining recommendations. Then, we scraped fashion images from a store of interest's website, in this case Banana Republic's website, using selenium. We passed all of these images through our trained CNN model and obtained a vector from the first dense layer of the CNN model that is representative of the feature maps for each image. Afterwards, we made use of these feature map vectors by experimenting with two different methods to create a KNN index: the scikit-learn BallTree method or annoy, an open-source Python package from Spotify for approximate nearest neighbors, methods (sklearn.neighbors.BallTree, n.d.; Spotify, 2019). Then, we used the example image feature map vector to query the KNN index and found the five closest neighbors based on various distance metrics, including Euclidean distance and cosine similarity (Le, 2019; Tuinhof, Pirker, & Haltmeier, 2018). This recommendation process is summarized below (Figure 4). To assess the quality of the recommendations, we created baseline recommendations by transforming the example and store images into vectors and performing a KNN ranking recommendation.


Query Image
CNN Feature Extractor
Obtain output from dense layer
Query
Find five closest neighbors
Store Images
CNN Feature Extractor
Obtain output from dense layer
KNN Index
Figure 4. A summary of the fashion image recommendation process utilized for this project.

Models

In order to obtain the best model, we built multiple simple CNN models (Figures CL7-8). Each of the simple CNN models vary in terms of architecture and hyperparameters (Figure 4a-b). Of note, Jessica 2 was removed due to a corrupted model file. Overall, we tried a variety of different hyperparameter combinations (Figure 4a-b). We decided to experiment with the learning rate by increasing and decreasing it in pursuit of better results. Jessica experimented with a smaller dropout rate, while Mary implemented a larger dropout rate. We found that the smaller dropout rate was more advantageous in this case for obtaining a higher micro-averaged F1-score. Also, we agreed that due to the nature of fashion recommendation, overfitting may not be a high-priority issue as it is in classification tasks. We determined that increasing the number of epochs was beneficial in obtaining a higher micro-averaged F1-score. Each of the simple CNN models utilize the Adam optimizer, which was chosen based on the previous literature regarding deep learning-based fashion recommendation (Tuinhof, Pirker, & Haltmeier, 2018). Experimenting with the batch size proved to be difficult due to computational constraints. In our case, the smallest batch size was 200 images and the largest batch size was 1024 images. Besides adjusting hyperparameters, we also experimented with resizing the images ranging from sizes of 32x32 to 120x120 and performing a variety of image transformations (Figures CL 1-6).

HyperParameters	**Model: Baseline [Template]**	**Model: Baseline [Jessica 1]**	**Model: Baseline [Jessica 3]**	**Model: Baseline [Jessica 4]**	**Model: Baseline [Jessica 5]**
Learning Rate	1.00E-02	5.00E-03	5.00E-05	5.00E-07	5.00E-05
Dropout	0.5	0.1	0.1	0.005	0.01
Epochs	5	5	10	10	25
Optimizer	Adam	Adam	Adam	Adam	Adam
Loss	BCEWithLogitsLoss()	BCEWithLogitsLoss()	BCEWithLogitsLoss()	BCEWithLogitsLoss()	BCEWithLogitsLoss()
Activation function	Relu	Relu	Relu	Relu	Relu
Batch Size	256	256	200	280	200
Layers	4 (2 conv, 2 linear)	4 (2 conv, 2 linear)	4 (2 conv, 2 linear)	4 (2 conv, 2 linear)	4 (2 conv, 2 linear)

Figure 4a. A table of hyperparameters for the baseline models Template and Jessica 1-5.

HyperParameters	**Model: Baseline [Jessica 6]**	**Model: Baseline [Jessica 7]**	**Model: Baseline [Mary 1]**	**Model: Baseline [Mary 2]**
Learning Rate	5.00E-05	5.00E-05	0.1	0.01
Dropout	0.01	0.01	0.45	0.5
Epochs	50	25	5	10
Optimizer	Adam	Adam	Adam	Adam
Loss	BCEWithLogitsLoss()	BCEWithLogitsLoss()	BCEWithLogitsLoss()	BCEWithLogitsLoss()
Activation function	Relu	Relu	Relu	Relu
Batch Size	200	200	1024	1024
Layers	4 (2 conv, 2 linear)	4 (2 conv, 2 linear)	3 (2 conv, 1 linear)	4 (3 conv, 2 linear)

Figure 4b. A table of hyperparameters for the baseline models Jessica 6-7 and Mary 1-2.

Ultimately, the best simple CNN model was Jessica 5 (Figure CL8). It consists of two convolutional layers using a kernel size of 3, a stride of 1, and a padding of 1. After each convolutional layer, there is a batch normalization layer and max pooling layer that utilizes a kernel size of 2 and a stride of 2. Then, the two convolutional layers are followed by a linear layer with a relu activation function, batch normalization layer, and a dropout layer. The output layer is a linear layer that outputs a 1-D array of size 149, which is the total number of unique labels (Figure 5). The total number of parameters that this simple CNN model includes is 2,642,851 (Figure 6).

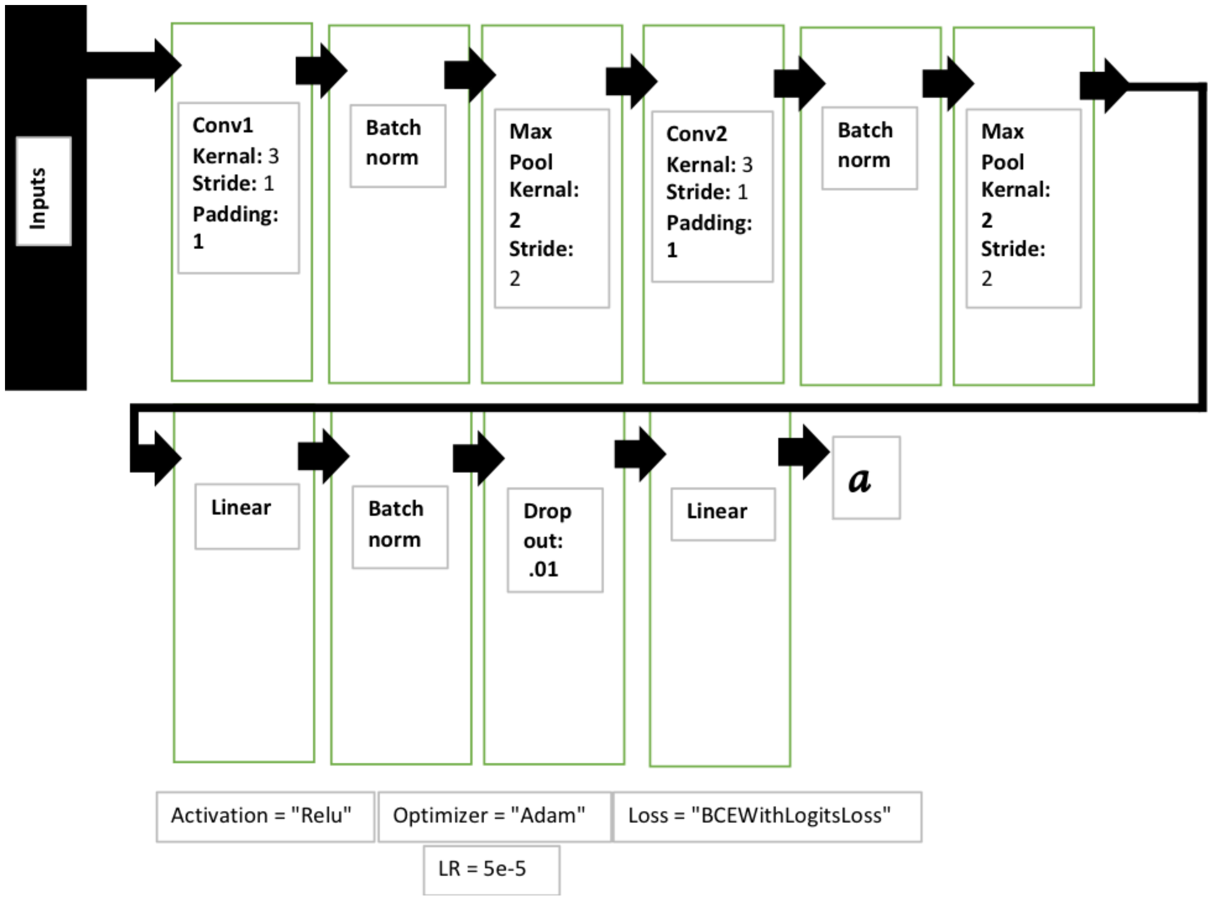
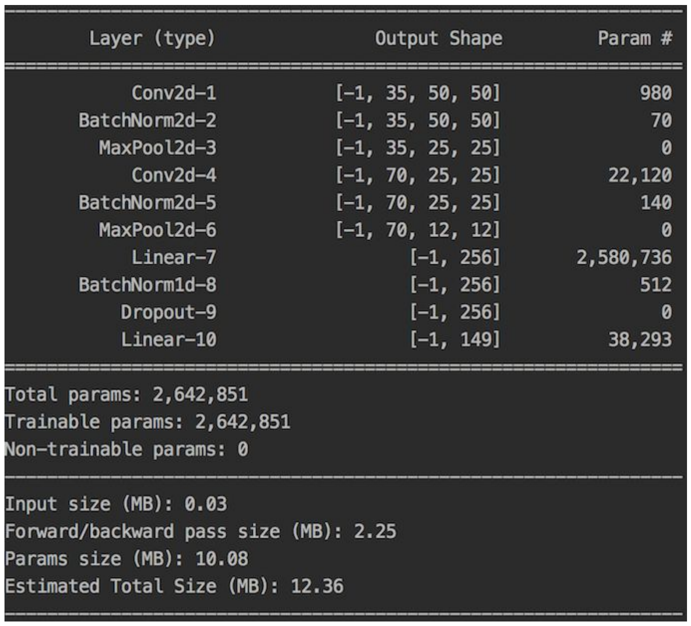


Figure 5. The architecture of the best baseline model Jessica 5.



```
================================================================
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 35, 50, 50]             980
       BatchNorm2d-2           [-1, 35, 50, 50]              70
         MaxPool2d-3           [-1, 35, 25, 25]               0
            Conv2d-4           [-1, 70, 25, 25]          22,120
       BatchNorm2d-5           [-1, 70, 25, 25]             140
         MaxPool2d-6           [-1, 70, 12, 12]               0
            Linear-7                  [-1, 256]       2,580,736
       BatchNorm1d-8                  [-1, 256]             512
           Dropout-9                  [-1, 256]               0
           Linear-10                  [-1, 149]          38,293
================================================================
Total params: 2,642,851
Trainable params: 2,642,851
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.03
Forward/backward pass size (MB): 2.25
Params size (MB): 10.08
Estimated Total Size (MB): 12.36
----------------------------------------------------------------
```

Figure 6. The trainable parameters and output sizes of the best baseline model Jessica 5.

In addition to the simple CNN models, we experimented with pretrained CNN models that were trained on the ImageNet dataset. First, we decided to try DenseNet-161 due to the fact that it has been previously mentioned in the literature in regards to feature extraction (Renatobmlr, 2018). DenseNet is a Densely Connected Convolutional Network model, in which each layer is connected to every other layer in a feed-forward fashion. As a result of the compactness of the model, the feature maps learned in any of the models layers can be accessed by all of the layers succeeded by it (Figure 8). “Recent work has shown that convolutional networks can be substantially deeper, more accurate, and efficient to train if they contain shorter connections between layers close to the input and those close to the output” (Renatobmlr, 2018). DenseNet has several advantages. The most important one to us is that it encourages feature map reuse, ensuring that we are only obtaining relevant feature maps. Additionally, this feature map reuse significantly reduces the number of parameters in the model. In addition this model addresses the vanishing-gradient problem and strengthens feature propagation (Huang, Liu, & van der Maaten, 2018). We did not train DenseNet-161 on our fashion dataset due to time constraints, but we used it to make predictions on our test set.

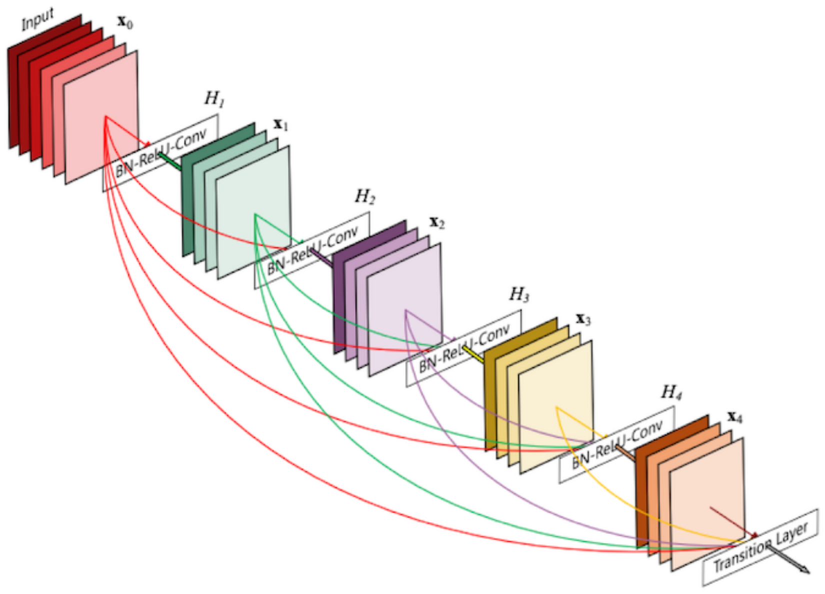


Figure 8. An illustration of DenseNet architecture.

Next, we decided to assess the pre-trained model MobileNetV2, which is a neural network architecture designed for efficiency and use on mobile devices (Hollemans, 2017). Our reasoning for choosing this pre-trained model was that ideally this project could be turned into a mobile application, so that users can upload fashion images of interest on-the-go and receive recommendations on their mobile devices. MobileNetV2 makes use of a depthwise convolution to act as a filter followed by a pointwise convolution to create new features. Unlike a traditional convolution, a depthwise convolution performs convolution on each channel separately, allowing each channel to receive its own set of weights. Then, the depthwise convolution is followed by a pointwise convolution that combines all of the channels (Figure 9). Together, this process is known as a depthwise separable convolution and is computationally more efficient than traditional convolution (Hollemans, 2017).

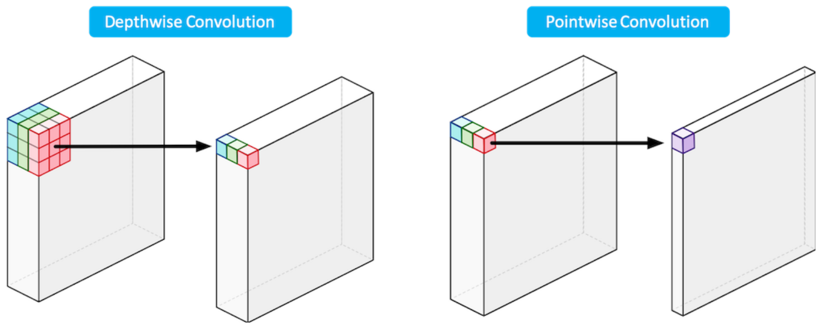


Figure 9. An illustration of a depthwise separable convolution followed by a pointwise convolution.

MobileNetV2's architecture consists of blocks containing three convolutional layers. It combines a 1x1 convolution to expand the number of channels prior to a depthwise separable convolution that filters and then reduces the number of channels. Also, it makes use of residual connections as seen in ResNet to aid with gradient flow (Figure 10) (Hollemans, 2018).

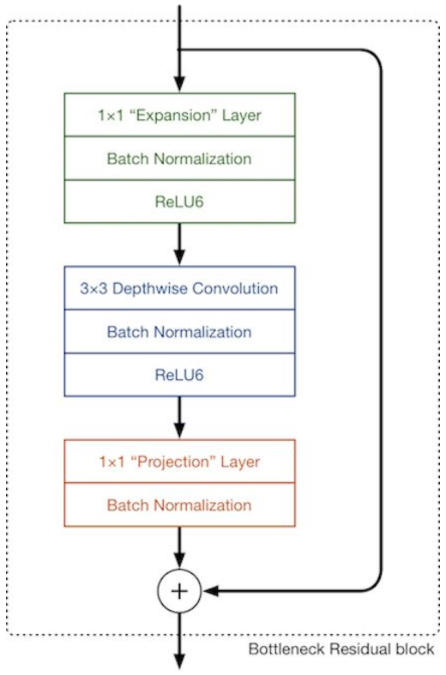
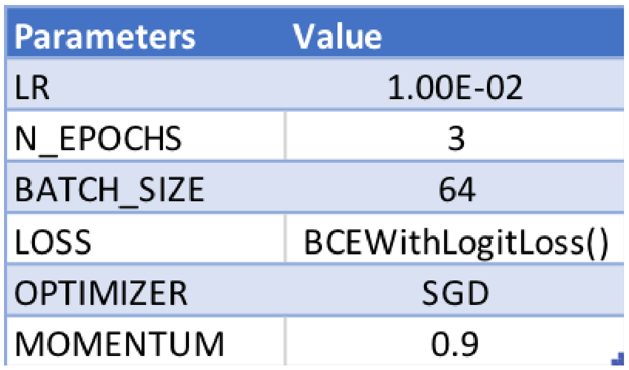


Figure 10. An illustration of a MobileNetV2 block consisting of three convolutional layers.

Regarding our use of MobileNetV2, we froze the layers within the first child attribute to take advantage of the pretrained parameters. We ran the frozen layers sequentially followed by a single linear layer, trained on our fashion dataset, that outputs a 1-D array of size 149, which is the total number of unique labels. The hyperparameters utilized for training can be found below (Figure 11) (Figure CL9).



Parameters	Value
LR	1.00E-02
N_EPOCHS	3
BATCH_SIZE	64
LOSS	BCEWithLogitLoss()
OPTIMIZER	SGD
MOMENTUM	0.9

Figure 11. The hyperparameters used for training MobileNetV2.

Since MobileNetV2 took around 11 hours to train, we did not experiment to the extent that we would have liked to with the hyperparameters. Regarding fashion recommendations, we decided to try different KNN methods. First, we tried the scikit-learn BallTree method with Euclidean distance, which appeared to perform well (Figure CL10) (sklearn.neighbors.BallTree, n.d.; Tuinhof, Pirker, & Haltmeier, 2018). Second, we attempted to use annoy for approximate nearest neighbors with cosine similarity. However, this method did not work well. For different example images, this method would return the exact same store image recommendations (Spotify, 2019). Therefore, we decided to utilize the scikit-learn BallTree method with Euclidean distance.

Results

Overall, the best simple CNN model was Jessica 5. During training, as the number of batches increased, the training the loss decreased. The tail of the training loss plot appears to continually decrease, indicating that the training loss may still continue to decrease with more batches and could potentially increase model performance (Figure 9). On the test set, the average micro-averaged F1-score was 0.390, the maximum was 0.410 and the minimum was 0.360 (Figure 10).

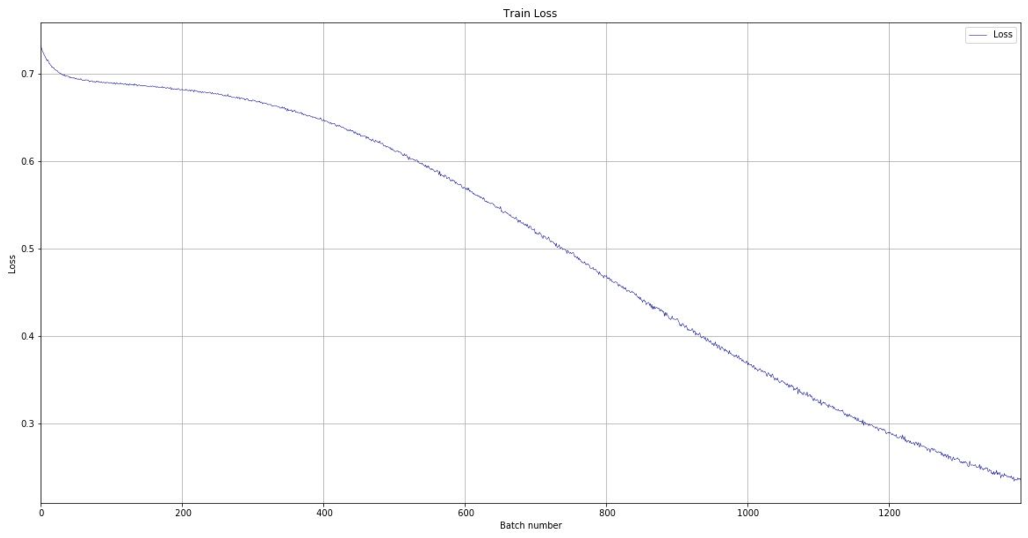


Figure 9. The training loss for the simple CNN model Jessica 5.

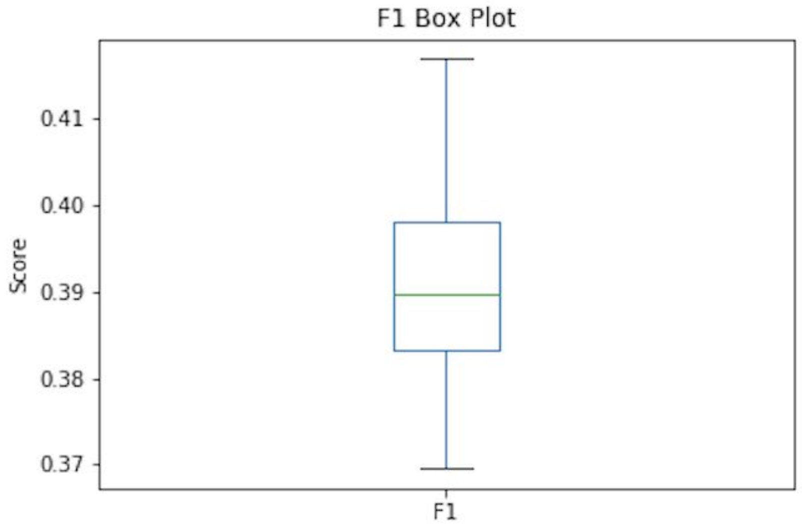


Figure 10. Micro-averaged F1-score distribution for the simple CNN model Jessica 5.

In addition the testing the simple CNN models we tested DenseNet-161 on our test set. DenseNet performed the worst out of all ten models that we tested. The micro-averaged average F1 score was 0.060, the maximum was 0.070 and the minimum was 0.050 (Figure 11). We can assume the poor test results are due to the fact that this model was not trained on our fashion dataset.

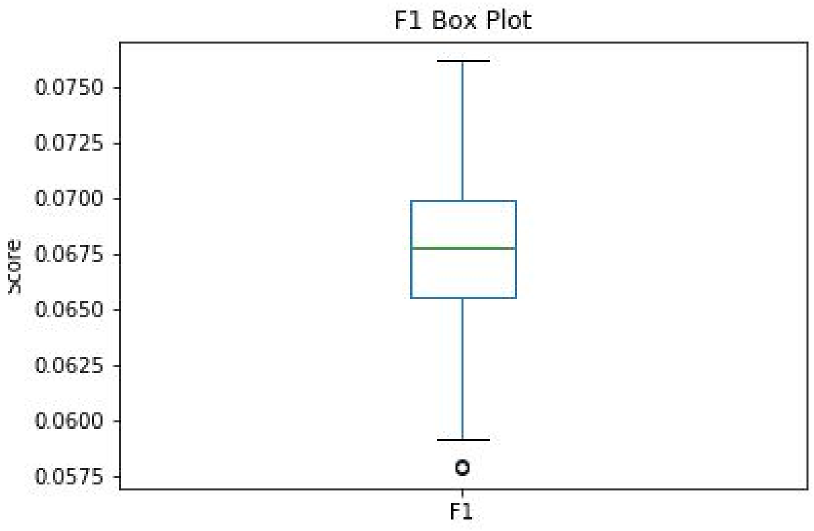


Figure 11. Micro-averaged F1-score distribution for DenseNet-161.

Regarding MobileNetV2, the training loss indicates that the loss significantly decreased in the beginning of training and flattened out for the rest of training (Figure 12).

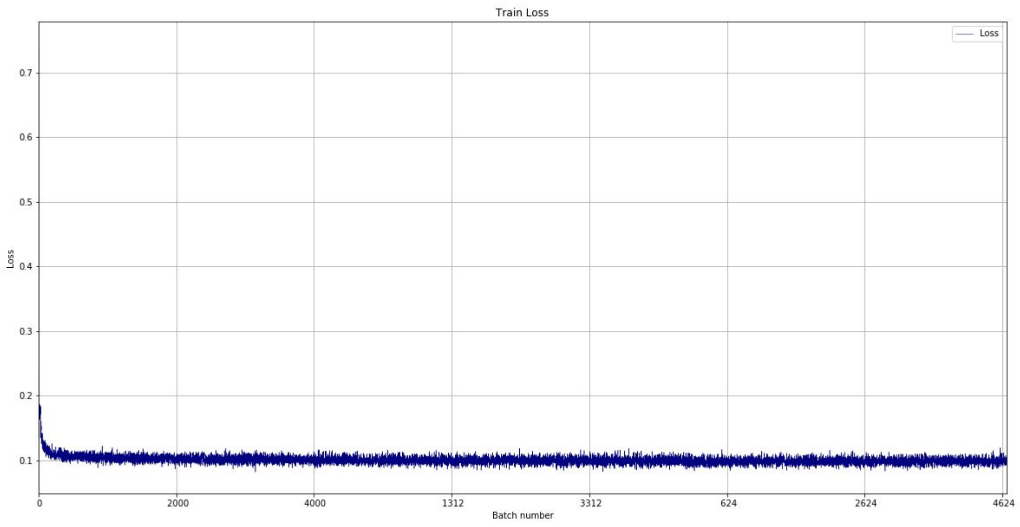


Figure 12. The training loss for MobileNetV2.

This suggests that our MobileNetV2 may possibly be overfit, but for our use case of fashion recommendation, overfitting may not be a major issue. MobileNetV2 performed slightly better than the simple CNN model Jessica 5 in terms of multilabel classification. The average F1 score was 0.396, the maximum was 0.417 and the minimum was 0.368. (Figure 13).

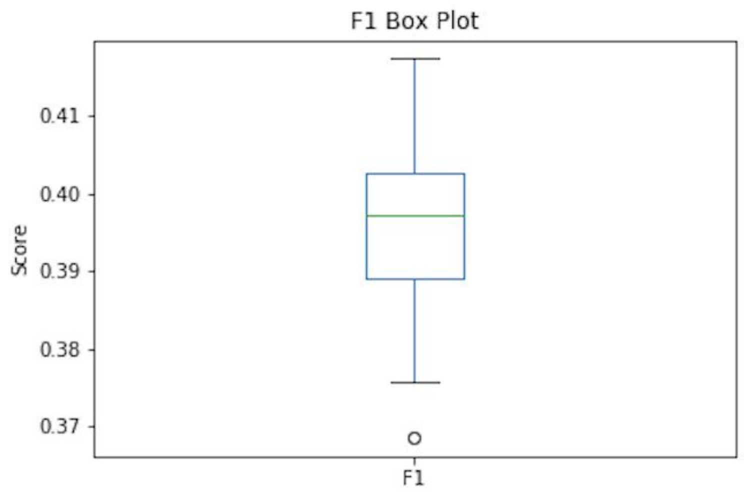


Figure 13. Micro-averaged F1-score distribution for MobileNetV2..

Below, are examples of recommendations using the scikit-learn BallTree method with Euclidean distance. The first example query image that we tried was a pair of jeans (Figure 14).

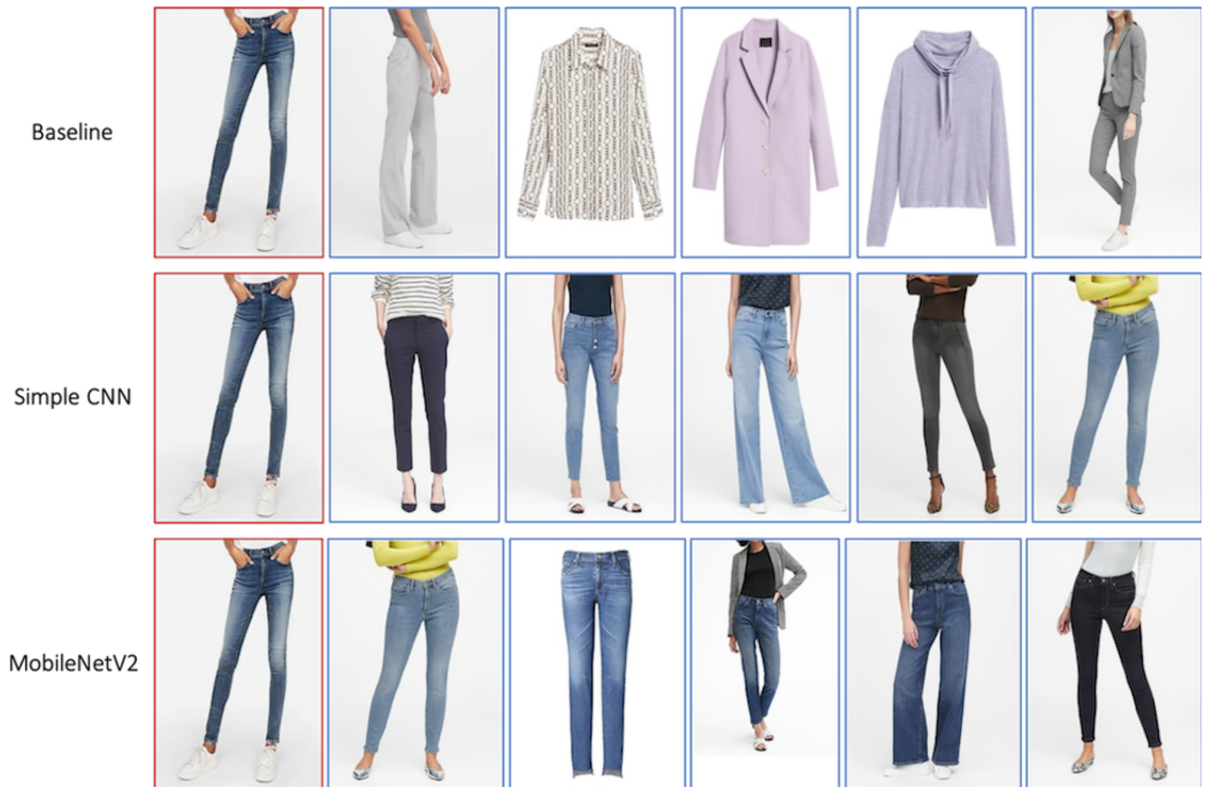


Figure 14. Jeans recommendations every model.

From this example, it is apparent that the baseline KNN recommendations do not look similar to our example query image. However, the simple CNN model Jessica 5 and MobileNetV2 feature extraction followed by KNN ranked recommendations appeared to perform well, recommending store images containing jeans. MobileNetV2 performed better than the simple CNN because it picked up on the faded, light pattern near the knees. The second example image that we tried was a skirt (Figure 15).

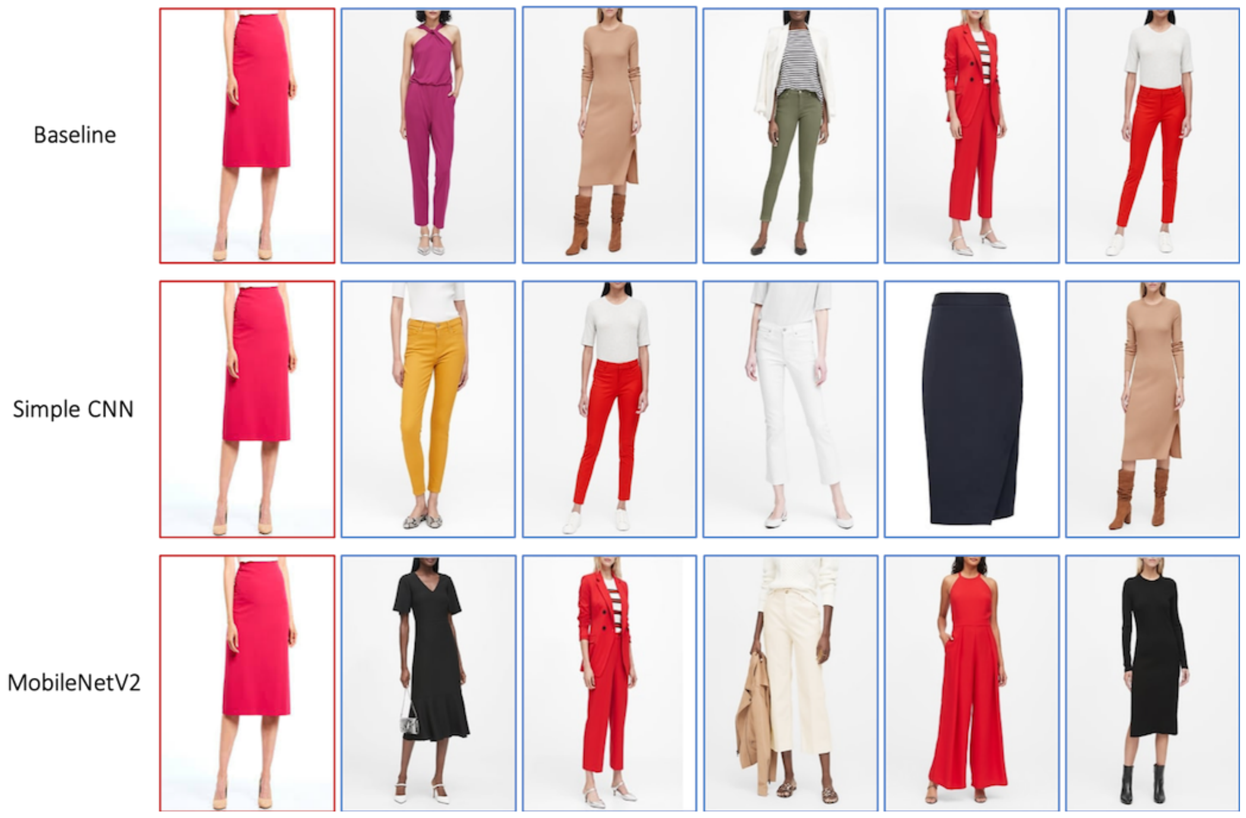


Figure 15. Skirt recommendations for every model.

From this example, it is evident that all of the models picked up on the color of the pink skirt and the nude shoes. However, the baseline KNN and the simple CNN model Jessica 5 did not pick up on the cut and the length of the skirt well, and provided recommendations of pants. The MobileNetV2 recommendations are superior to the others since they not only picked up on the colors within the example query image, but also the cut and length of the skirt well. For the last example, we decided to observe how well our fashion recommendation models would transfer to real-life images. Therefore, we decided to use an example query image consisting of Jessica and her dog, Conrad (Figure 16).

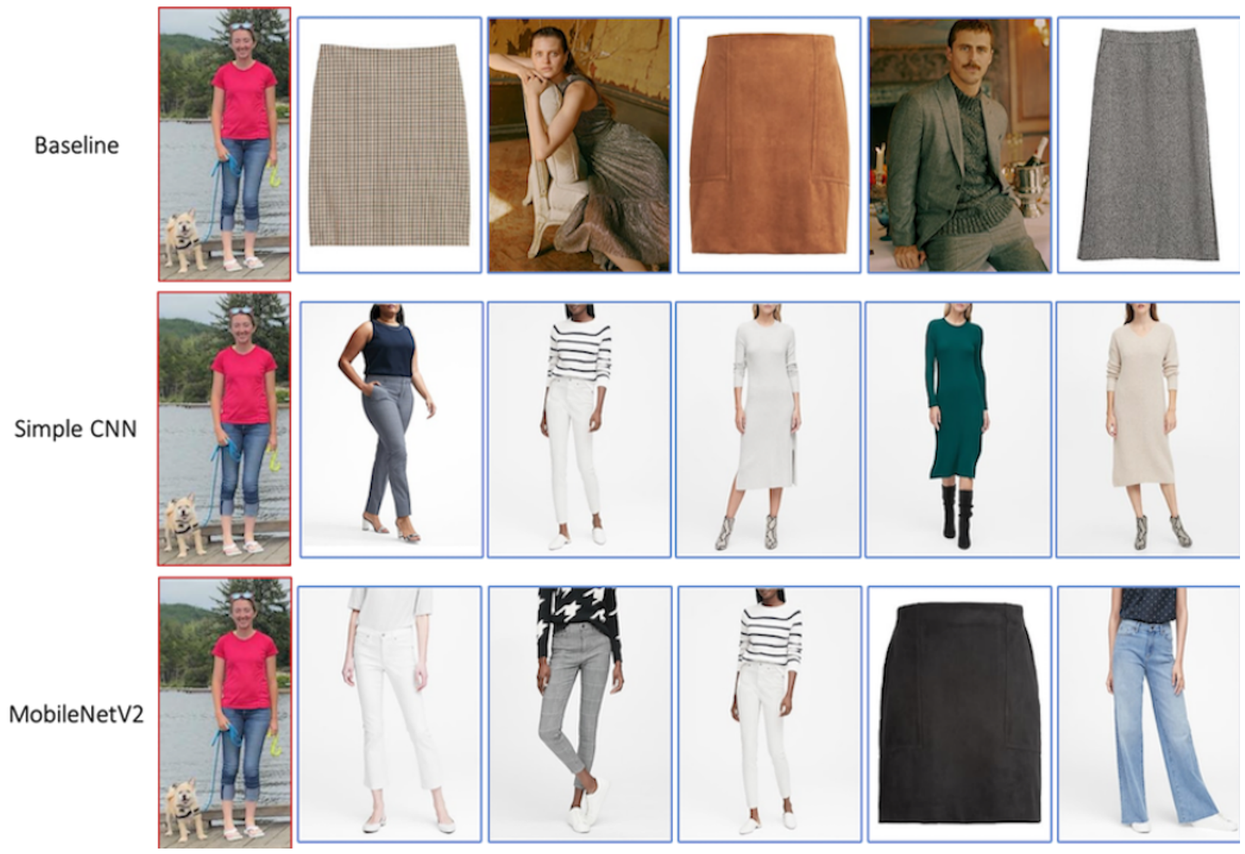


Figure 16. Jessica recommendations for every model.

It is apparent that our models did not perform extremely well on the Jessica example image. Based on the recommended images, it is evident that the models picked up on the colors within the example image, such as the green trees, bluish-grey water, and black harness on Conrad. We want to point out that the simple CNN model Jessica 5 and MobileNetV2 made a lot of recommendations related to white clothing, picking up on the white shoes within the example query image. Based on the word cloud, white is one of the majority labels in our training set, indicating that the label imbalance within our training set might be a cause for concern and something that we should address. Also, of note, the recommended images consisted of skirts, dresses, and pants, but not tops, which suggests the need for bounding boxes and pose estimation. Overall, we feel that MobileNetV2 performed the best for this task since it recommended pants that appear to be a similar length to the example query image.

Summary & Conclusions

We have presented a deep learning-based fashion recommendation system. Based on the proposed two-step approach, we first trained CNN models on fashion images in order to extract the feature maps associated with different clothing items. Second, we used those feature maps to create a KNN model that returns the five nearest neighbors to a given query image as clothing recommendations. We found that the CNN model MobileNetV2 produced the best results in terms of micro-averaged F1-score and when coupled with the the scikit-learn Ball-Tree KNN model using Euclidean distance produced the best fashion recommendations.

Throughout this project, we were presented with several limitations regarding our approach. We feel that a major limitation was computational power. Due to memory constraints, we could only use a small image size and a small batch size when training and testing the CNN models. Another major limitation was time. With 300,000 images, training the CNN models was a time-intensive process with some of the models taking up to 12 hours to train. Lastly, the fashion dataset in itself was a limitation due to the fact that it contained a class imbalance, which affected our clothing recommendations. Also, the fashion dataset did not contain bounding boxes or pose estimation, which could have greatly improved our clothing recommendations.

As this is a project we are both passionate about, we intend to continue to improve it. The main motivation behind this project was to create a sophisticated recommendation system that can eventually be used in the context of a mobile/web application. To further improve upon this project, we plan on working with the DeepFashion and DeepFashion2 datasets. Both of these datasets include bounding boxes, which we would like to use to extract feature maps for individual clothing items, and pose estimations, which will aid in dealing with different poses within fashion images that may help to better generalize our approach to real-life images. Moreover, we would like to assess future models by utilizing Top-k accuracy since we are only interested in the first k items being correct and relevant in order to produce a recommendation.

References

Anqitu. (2018, April 11). For Starter: JSON to MultiLabel in 24 seconds. Retrieved from https://www.kaggle.com/anqitu/for-starter-json-to-multilabel-in-24-seconds.

Aspris, M. (2018, September 15). Simple Implementation of Densely Connected Convolutional Networks in PyTorch. Retrieved from https://towardsdatascience.com/simple-implementation-of-densely-connected-convolutional-networks-in-pytorch-3846978f2f36.

Gómez, Raúl. (2018). Retrieved from https://gombru.github.io/2018/05/23/cross_entropy_loss/.

Guérin, J., & Boots, B. (2018). Improving Image Clustering With Multiple Pretrained CNN Feature Extractors, 1–10.

He, T., & Hu, Y. (2018). FashionNet: Personalized Outfit Recommendation with Deep Neural Network, 1–8.

Hollemans, M. (2017). Retrieved from https://machinethink.net/blog/googles-mobile-net-architecture-on-iphone/.

Hollemans, M. (2018). Retrieved from https://machinethink.net/blog/mobilenet-v2/.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., … Adam, H. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.

Huang, G., Liu, Z., & van der Maaten, L. (2018). Densely Connected Convolutional Networks. Densely Connected Convolutional Networks, 1–8. Retrieved from https://arxiv.org/pdf/1608.06993.pdf

iMaterialist Challenge (Fashion) at FGVC5. (2018). Retrieved from https://www.kaggle.com/c/imaterialist-challenge-fashion-2018.

Le, J. (2019, August 16). Recommending Similar Fashion Images with Deep Learning. Retrieved from https://blog.floydhub.com/similar-fashion-images/.

Mratsim. (2017, October 4). Starting Kit for PyTorch Deep Learning. Retrieved from https://www.kaggle.com/mratsim/starting-kit-for-pytorch-deep-learning.

Nazi, Z. A., & Abir, T. A. (2018). Automatic Skin Lesion Segmentation and Melanoma Detection: Transfer Learning approach with U-Net and DCNN-SVM. International Joint Conference on Computational Intelligence. Retrieved from https://www.researchgate.net/publication/330564744_Automatic_Skin_Lesion_Segmentation_and_Melanoma_Detection_Transfer_Learning_approach_with_U-Net_and_DCNN-SVM

Nlecoy. (2018, April 4). iMaterialist downloader util. Retrieved from https://www.kaggle.com/nlecoy/imaterialist-downloader-util.

PyTorch. (2017, December 12). DenseNet-161. Retrieved from https://www.kaggle.com/pytorch/densenet161.

Renatobmlr. (2018, November 11). [PyTorch] DenseNet as Feature Extractor. Retrieved from https://www.kaggle.com/renatobmlr/pytorch-densenet-as-feature-extractor.

sklearn.metrics.f1_score. (n.d.). Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html.

Sharma, P. (2019, August 1). Build your First Multi-Label Image Classification Model in Python. Retrieved from https://www.analyticsvidhya.com/blog/2019/04/build-first-multi-label-image-classification-model-python/.

Shrimali, V. (2019, June 3). Home. Retrieved from https://www.learnopencv.com/pytorch-for-beginners-image-classification-using-pre-trained-models/.

sklearn.neighbors.BallTree. (n.d.). Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.BallTree.html#sklearn.neighbors.BallTree.

Spotify. (2019, October 24). spotify/annoy. Retrieved from https://github.com/spotify/annoy.

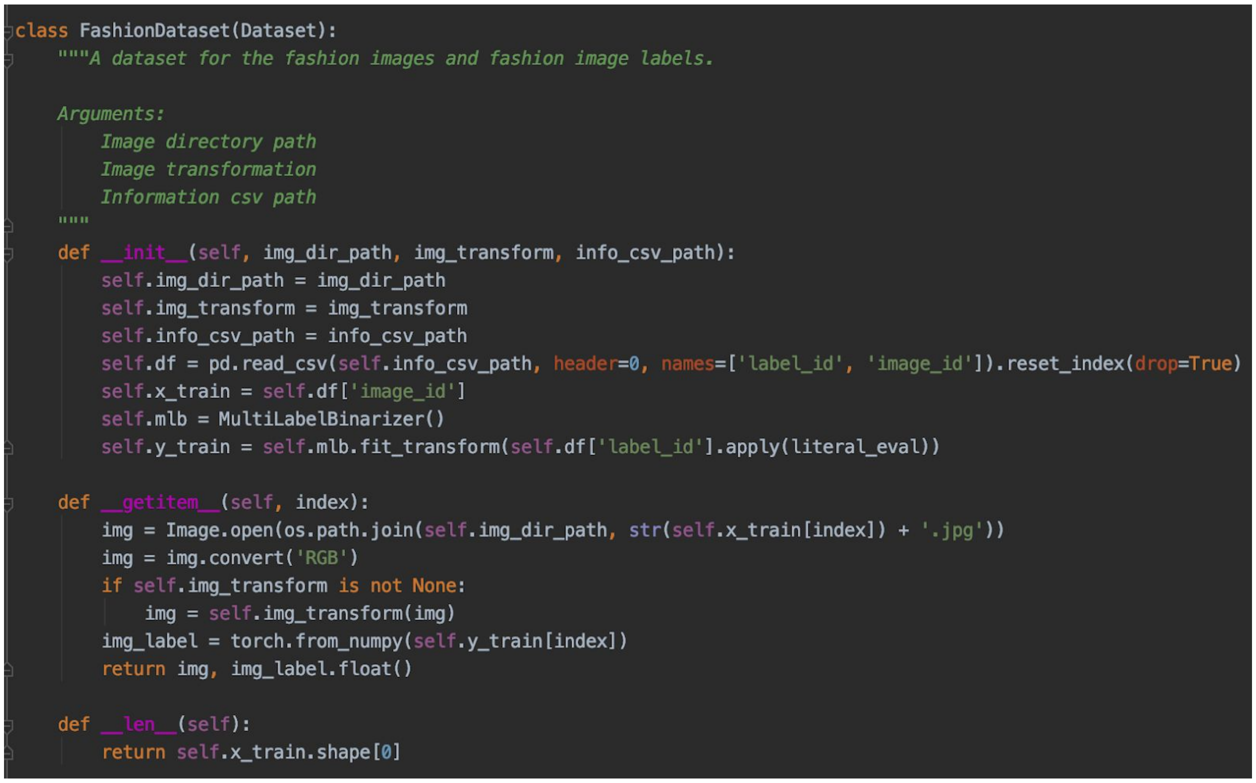
Tsang, S.-H. (2019, March 20). Review: DenseNet - Dense Convolutional Network (Image Classification). Retrieved from https://towardsdatascience.com/review-densenet-image-classification-b6631a8ef803.

Tuinhof, H., Pirker, C., & Haltmeier, M. (2018). Image Based Fashion Product Recommendation with Deep Learning, 1–10.doi.org/10.1007/978-3-030-13709-0_40

Utkuozbulak. (2018, July 15). utkuozbulak/pytorch-custom-dataset-examples. Retrieved from https://github.com/utkuozbulak/pytorch-custom-dataset-examples.

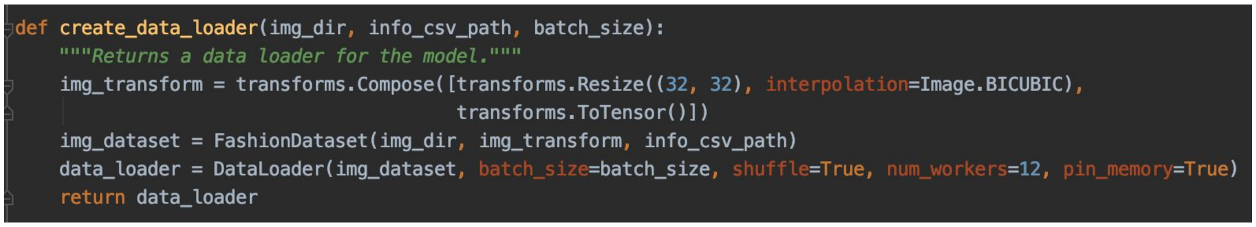
Visipedia. (2019, June 26). visipedia/imat_fashion_comp. Retrieved from https://github.com/visipedia/imat_fashion_comp.

Computer Listings



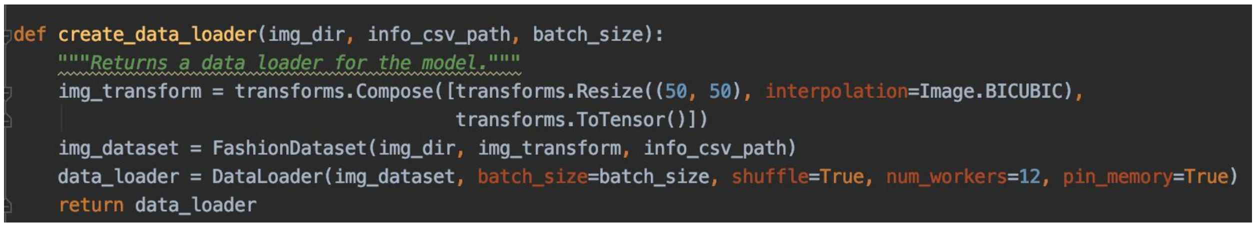
```python
class FashionDataset(Dataset):
    """A dataset for the fashion images and fashion image labels.

    Arguments:
        Image directory path
        Image transformation
        Information csv path
    """
    def __init__(self, img_dir_path, img_transform, info_csv_path):
        self.img_dir_path = img_dir_path
        self.img_transform = img_transform
        self.info_csv_path = info_csv_path
        self.df = pd.read_csv(self.info_csv_path, header=0, names=['label_id', 'image_id']).reset_index(drop=True)
        self.x_train = self.df['image_id']
        self.mlb = MultiLabelBinarizer()
        self.y_train = self.mlb.fit_transform(self.df['label_id'].apply(literal_eval))

    def __getitem__(self, index):
        img = Image.open(os.path.join(self.img_dir_path, str(self.x_train[index]) + '.jpg'))
        img = img.convert('RGB')
        if self.img_transform is not None:
            img = self.img_transform(img)
        img_label = torch.from_numpy(self.y_train[index])
        return img, img_label.float()

    def __len__(self):
        return self.x_train.shape[0]
```

Figure CL1. Fashion Dataset



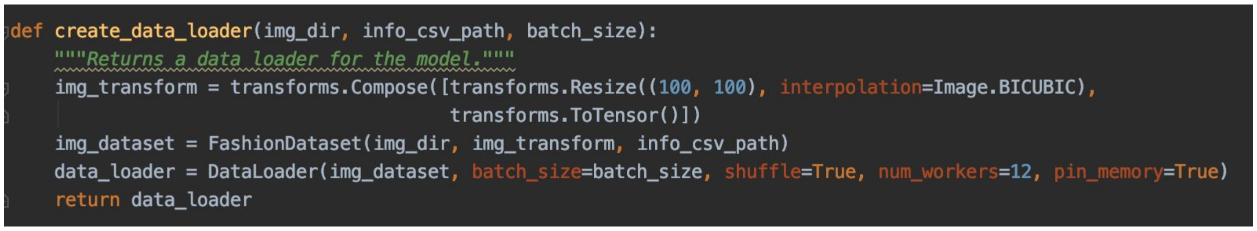
```python
def create_data_loader(img_dir, info_csv_path, batch_size):
    """Returns a data loader for the model."""
    img_transform = transforms.Compose([transforms.Resize((32, 32), interpolation=Image.BICUBIC),
                                        transforms.ToTensor()])
    img_dataset = FashionDataset(img_dir, img_transform, info_csv_path)
    data_loader = DataLoader(img_dataset, batch_size=batch_size, shuffle=True, num_workers=12, pin_memory=True)
    return data_loader
```

Figure CL2. Jessica Model 1



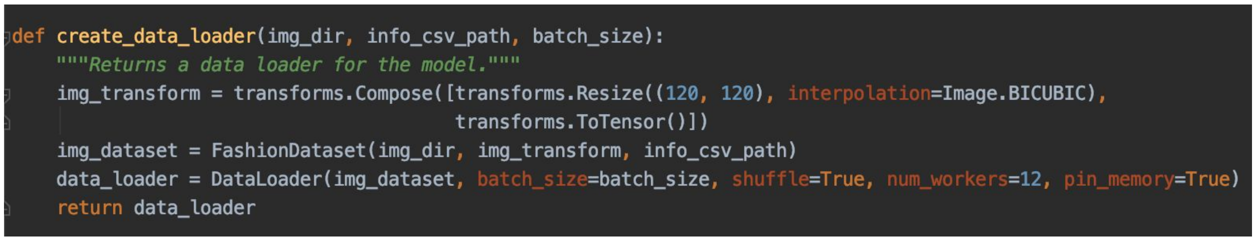
```python
def create_data_loader(img_dir, info_csv_path, batch_size):
    """Returns a data loader for the model."""
    img_transform = transforms.Compose([transforms.Resize((50, 50), interpolation=Image.BICUBIC),
                                        transforms.ToTensor()])
    img_dataset = FashionDataset(img_dir, img_transform, info_csv_path)
    data_loader = DataLoader(img_dataset, batch_size=batch_size, shuffle=True, num_workers=12, pin_memory=True)
    return data_loader
```

Figure CL3. Jessica Models 3-7



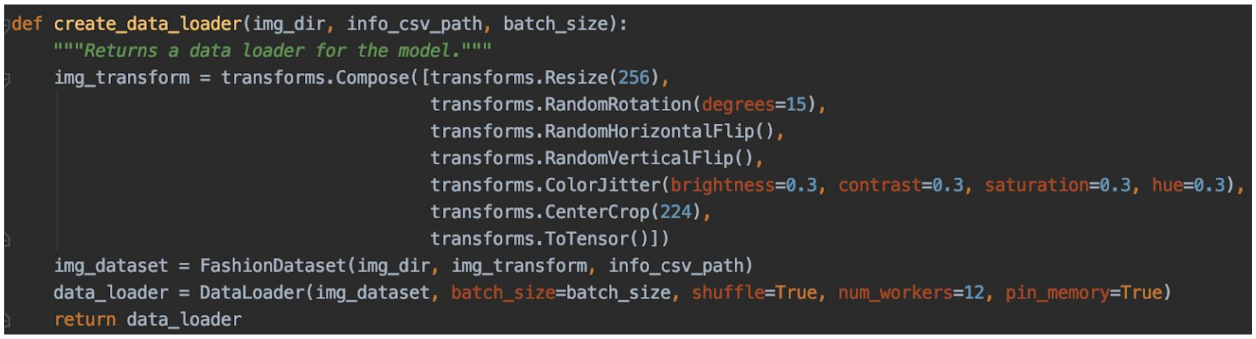
```python
def create_data_loader(img_dir, info_csv_path, batch_size):
    """Returns a data loader for the model."""
    img_transform = transforms.Compose([transforms.Resize((100, 100), interpolation=Image.BICUBIC),
                                        transforms.ToTensor()])
    img_dataset = FashionDataset(img_dir, img_transform, info_csv_path)
    data_loader = DataLoader(img_dataset, batch_size=batch_size, shuffle=True, num_workers=12, pin_memory=True)
    return data_loader
```

Figure CL4. Mary Model 1



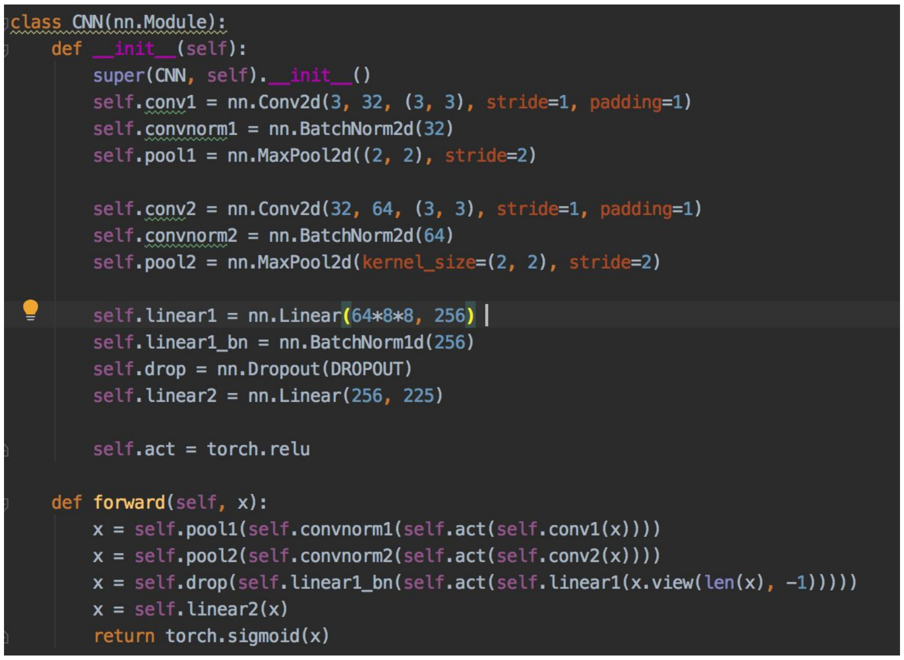
```python
def create_data_loader(img_dir, info_csv_path, batch_size):
    """Returns a data loader for the model."""
    img_transform = transforms.Compose([transforms.Resize((120, 120), interpolation=Image.BICUBIC),
                                        transforms.ToTensor()])
    img_dataset = FashionDataset(img_dir, img_transform, info_csv_path)
    data_loader = DataLoader(img_dataset, batch_size=batch_size, shuffle=True, num_workers=12, pin_memory=True)
    return data_loader
```

Figure CL5. Mary Model 2



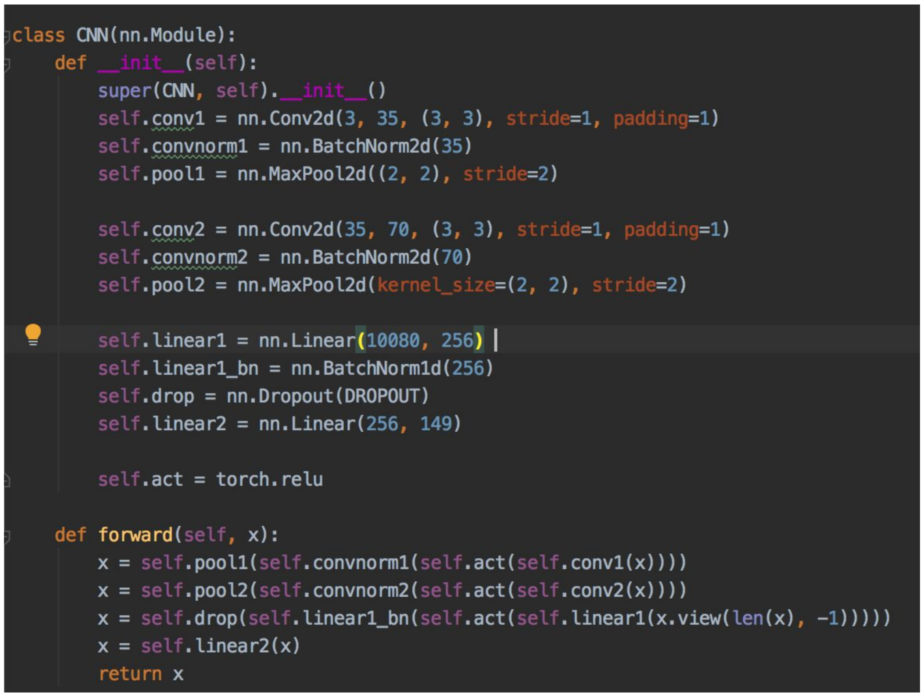
```python
def create_data_loader(img_dir, info_csv_path, batch_size):
    """Returns a data loader for the model."""
    img_transform = transforms.Compose([transforms.Resize(256),
                                        transforms.RandomRotation(degrees=15),
                                        transforms.RandomHorizontalFlip(),
                                        transforms.RandomVerticalFlip(),
                                        transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation=0.3, hue=0.3),
                                        transforms.CenterCrop(224),
                                        transforms.ToTensor()])
    img_dataset = FashionDataset(img_dir, img_transform, info_csv_path)
    data_loader = DataLoader(img_dataset, batch_size=batch_size, shuffle=True, num_workers=12, pin_memory=True)
    return data_loader
```

Figure CL6. MobileNetV2



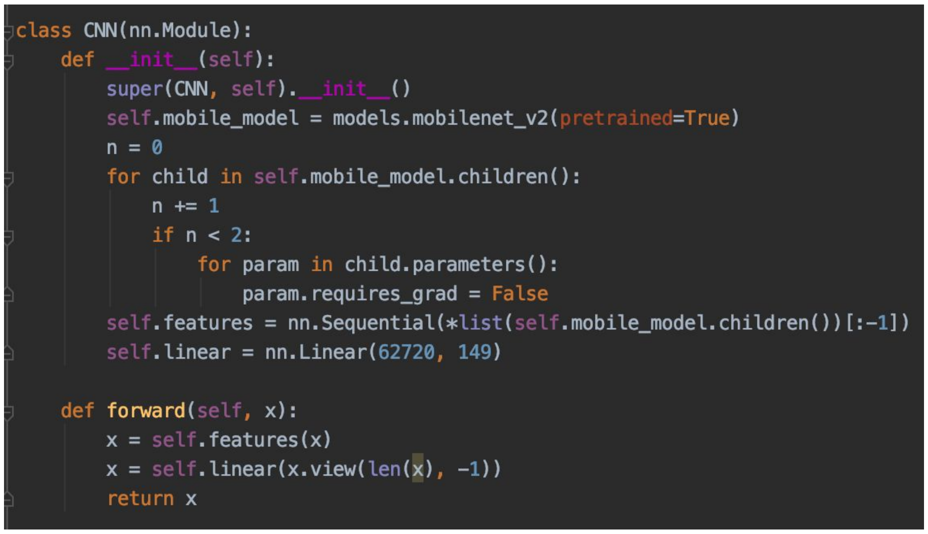
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, (3, 3), stride=1, padding=1)
        self.convnorm1 = nn.BatchNorm2d(32)
        self.pool1 = nn.MaxPool2d((2, 2), stride=2)

        self.conv2 = nn.Conv2d(32, 64, (3, 3), stride=1, padding=1)
        self.convnorm2 = nn.BatchNorm2d(64)
        self.pool2 = nn.MaxPool2d(kernel_size=(2, 2), stride=2)

        self.linear1 = nn.Linear(64*8*8, 256)
        self.linear1_bn = nn.BatchNorm1d(256)
        self.drop = nn.Dropout(DROPOUT)
        self.linear2 = nn.Linear(256, 225)

        self.act = torch.relu

    def forward(self, x):
        x = self.pool1(self.convnorm1(self.act(self.conv1(x))))
        x = self.pool2(self.convnorm2(self.act(self.conv2(x))))
        x = self.drop(self.linear1_bn(self.act(self.linear1(x.view(len(x), -1)))))
        x = self.linear2(x)
        return torch.sigmoid(x)
```

Figure CL7. Baseline CNN Model



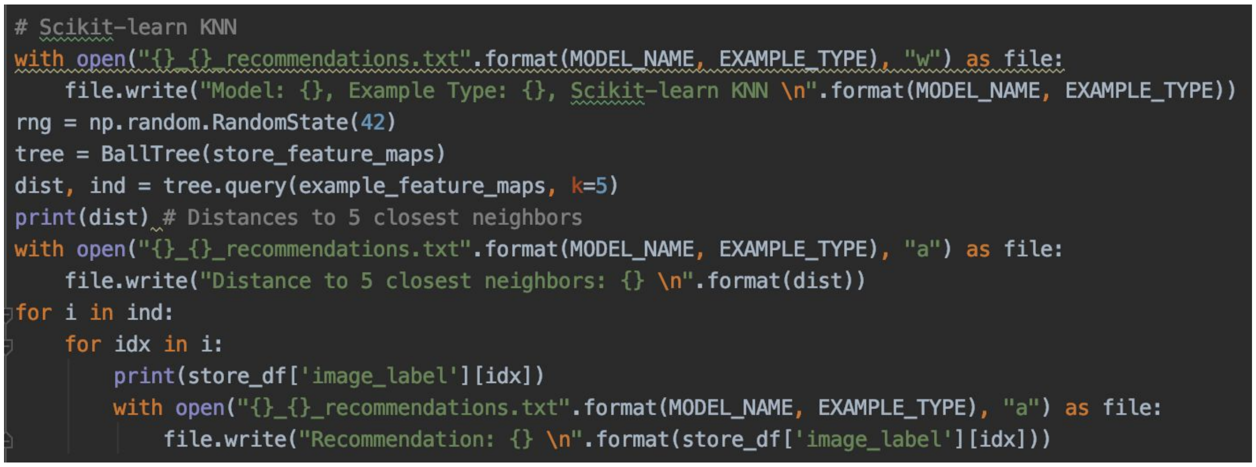
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 35, (3, 3), stride=1, padding=1)
        self.convnorm1 = nn.BatchNorm2d(35)
        self.pool1 = nn.MaxPool2d((2, 2), stride=2)

        self.conv2 = nn.Conv2d(35, 70, (3, 3), stride=1, padding=1)
        self.convnorm2 = nn.BatchNorm2d(70)
        self.pool2 = nn.MaxPool2d(kernel_size=(2, 2), stride=2)

        self.linear1 = nn.Linear(10080, 256)
        self.linear1_bn = nn.BatchNorm1d(256)
        self.drop = nn.Dropout(DROPOUT)
        self.linear2 = nn.Linear(256, 149)

        self.act = torch.relu

    def forward(self, x):
        x = self.pool1(self.convnorm1(self.act(self.conv1(x))))
        x = self.pool2(self.convnorm2(self.act(self.conv2(x))))
        x = self.drop(self.linear1_bn(self.act(self.linear1(x.view(len(x), -1)))))
        x = self.linear2(x)
        return x
```

Figure CL8. Simple CNN Model Jessica 5



```python
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.mobile_model = models.mobilenet_v2(pretrained=True)
        n = 0
        for child in self.mobile_model.children():
            n += 1
            if n < 2:
                for param in child.parameters():
                    param.requires_grad = False
        self.features = nn.Sequential(*list(self.mobile_model.children())[:-1])
        self.linear = nn.Linear(62720, 149)

    def forward(self, x):
        x = self.features(x)
        x = self.linear(x.view(len(x), -1))
        return x
```

Figure CL9. MobileNetV2



```python
# Scikit-learn KNN
with open("{}_{}_recommendations.txt".format(MODEL_NAME, EXAMPLE_TYPE), "w") as file:
    file.write("Model: {}, Example Type: {}, Scikit-learn KNN \n".format(MODEL_NAME, EXAMPLE_TYPE))
rng = np.random.RandomState(42)
tree = BallTree(store_feature_maps)
dist, ind = tree.query(example_feature_maps, k=5)
print(dist) # Distances to 5 closest neighbors
with open("{}_{}_recommendations.txt".format(MODEL_NAME, EXAMPLE_TYPE), "a") as file:
    file.write("Distance to 5 closest neighbors: {} \n".format(dist))
for i in ind:
    for idx in i:
        print(store_df['image_label'][idx])
        with open("{}_{}_recommendations.txt".format(MODEL_NAME, EXAMPLE_TYPE), "a") as file:
            file.write("Recommendation: {} \n".format(store_df['image_label'][idx]))
```

Figure CL10. Scikit-learn Ball-Tree KNN using Euclidean Distance