Introduction

In this case study, we will apply everything we have learned in this course to a real world example. In this example we will be creating an image classifier. We will create a model that will be able to look at an image of a certain retail item (eg shoes, watches, makeup) and classify what category that item belongs to based on the items picture. As an overview, we will download the dataset, preprocess the data/convert it into a form that can be read by our machine learning model, create a model that attempts to predict the category each datapoint belongs to, evaluate the accuracy of our model and finally create visualizations that show how exactly the model is treating our dataset. Create a new Jupiter notebook called CaseStudy.ipynb within this repository and type/execute all the code that is needed for this assignment in that notebook. We will grade this assignment based on what is in the notebook and what output is being generated by the code. Lastly if there is another case study project involving another dataset you feel compelled to do instead of this then feel free to do that instead. In that case, at least use the directions in this case study as an example of what you should be including in your own project.

Collecting Data

First, go to https://www.kaggle.com/paramaggarwal/fashion-product-images-small and download the dataset where it says “Download (546 MB)”. This will download a zip file containing all the data. Once you have forked and cloned this repository over to your computer, move the zip file to the repository directory. Then unzip the file. The way to do this will depend on your operating system but for linux/mac you should be able to use the “unzip” command. This will create styles.csv, an images subdirectory, and a myntradataset subdirectory. The myntradataset subdirectory seems to contain all the same contents as the images directory so it shouldn’t be needed.

The Data

Styles.csv is a file that contains all the different retail items. The columns worth noting here are the id column and the masterCategory column. The masterCategory column is the category that each item is associated with, and is the column we will be trying to classify. The id column shows the id of each retail item. With each id, there is a corresponding image in the images folder with the name images/<id>.jpg. The other columns can be disregarded (unless in the future you decide to analyze these other columns).

Preprocessing

Load styles.csv into memory as a pandas dataframe and save it as a variable df. Using the read_csv() method like usual will already cause problems because there seem to be some lines within the csv file where there is a comma within one of the entries. This will create a row that pandas is reading to have more columns than it is supposed to because commas are the default delimiter. One way around this is to have pandas not read the “bad” rows by specifying error_bad_lines=False.

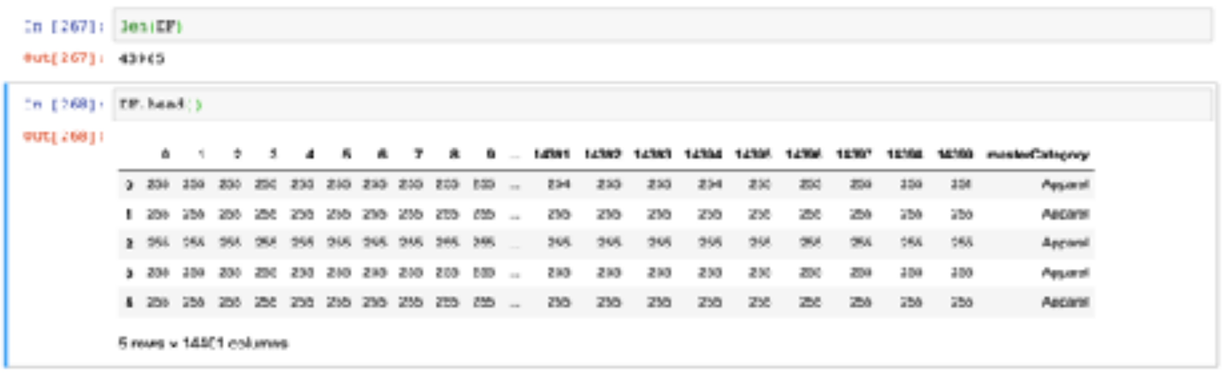
Once you have done this, create a plot that shows the number of items that belong to each distinct category in the masterCategory column. The easiest way to do this is probably to import seaborn and use the countplot() method. This will give you an idea of the fraction of items that go to each category. Some things to think about: Which category has the most items? What fraction of items belong to that category? Knowing this, what would a suitable benchmark be for the accuracy of our image classification model?.

Next we are going to open one of the images in the images folder and convert it to a form that can be used for our model. Pick one of the images in the folder and identify the path to that image. matplotlib.pyplot has a method called imread() that can read in jpg files, and convert them to numpy arrays. Save one of the images as an array and then print out the shape of the array. Unless you got very unlucky (in which case try another image), it should have a shape of (80,60,3). This means that the image is 80 pixels tall, 60 pixels wide, and the third dimension shows the red, green, and blue intensities at every pixel (the color of every pixel is some combination of red, green, and blue). For this array to be in a form that we can treat as a row in a dataset, it needs to be flattened with the flatten() method. This should turn it into an array of shape (14400,). Print out the shape of the flattened array. If you ever want to convert it back to its original shape, you can use the reshape() method.

Create 2 empty lists called arrays and names. Then, by iterating over the id column in df, add the id to the names list, and the flattened numpy array of the corresponding image to the arrays list one by one. You will notice a few issues doing this however. There are a handful of ids in the id column that don't have a corresponding image in the images folder. There are also a few images that don't have a shape of (80, 60, 3), either because the picture has a different height/width, or it is a grayscale image, so it has a shape of (80, 60) instead. Find a way to make sure you are only including images that are in the images folder, and have the correct shape of (80, 60, 3) when appending to the lists. If you like, find a way to take note of the number of entries that don't satisfy these conditions, and justify why it is okay to leave out these entries. You will want to use try/except and if statements.

Once these 2 lists have been created, the variable will be a list of numpy arrays all having a shape of (14400,). Find a way to convert this into a 2 dimensional numpy array with the number of rows being the original number of rows in the variable arrays and the number of columns being 14400. np.concatenate and reshape() may be helpful here. Then convert this into another pandas dataframe and save this dataframe to a variable called imagedf. Add an 'id' column to imagedf which is the names list you created earlier. As long as they are the same length, you shouldn't experience problems with this step.

You now have df, which should have the columns id and masterCategory, and imagedf which should have the columns id, and all of the columns representing the image arrays. Merge these dataframes together with an inner join, and join on the id column. Once you have done this, drop the id column from this new dataframe. Save this new dataframe as the variable DF (or overwrite imagedf if you're concerned with memory). If all went well, you should have a dataframe that has 43965 rows and 14401 columns and should look something like this.

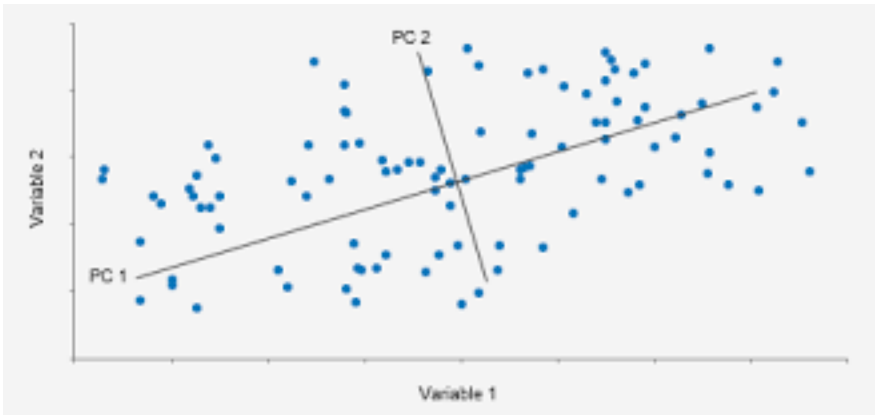


```
In [267]: len(DF)
Out[267]: 43965

In [268]: DF.head()
Out[268]:
     0    1    2    3    4    5    6    7    8    9  ...  14391  14392  14393  14394  14395  14396  14397  14398  14399  masterCategory
0  230  230  230  230  230  230  230  230  230  230  ...    234    230    230    234    230    230    230    230    230         Apparel
1  255  255  255  255  255  255  255  255  255  255  ...    255    255    255    255    255    255    255    255    255         Apparel
2  246  246  246  246  246  246  246  246  246  246  ...    246    246    246    246    246    246    246    246    246         Apparel
3  230  230  230  230  230  230  230  230  230  230  ...    230    230    230    230    230    230    230    230    230         Apparel
4  255  255  255  255  255  255  255  255  255  255  ...    255    255    255    255    255    255    255    255    255         Apparel

5 rows × 14401 columns
```

We now have a dataset that can be used by a sklearn machine learning model.

Modeling

Using DF, create a training set and testing set with the train_test_split() method, using the masterCategory column as y. Use whatever test size you wish. Since there is quite a lot of rows, we can afford to use a relatively small test size, like 0.2. Once you have created X_train, X_test, y_train, and y_test, there is one last preprocessing step we need to take before modeling. Because of the number of rows and columns in this dataset, it is not practical to fit a machine learning model directly to it, even if you're using logistic regression or some other simple model. It will take much too long to fit. Instead, we are going to reduce the dimensionality of X_train and X_test by using a technique called PCA. Within any high dimensional dataset, there are directions that the data has the highest variance along. PCA attempts to find these directions, and reduce all the data points to how far along these specific directions they lie. For example in the picture below,



the data varies most along PC 1 and PC 2 is the direction that is perpendicular to PC 1. In this way, the data points can be reduced to how far along PC 1 they are and how far along PC 2 they are. If we leave out PC 2 and only focus on how far along PC 1 the data points lie, we can reduce the dimensionality of this dataset from 2 to 1, effectively lowering the dimensionality while keeping as much information as possible about each point. Using this technique, we can lower the dimensionality of our dataset from 14400 to as low as we want. See https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html for how to use the PCA module. The principal components will be ordered by the variance in the data along that direction, which in theory should retain the most information about each point. Initialize a PCA instance with n_components = 10 (you can change this in the future if you like but be aware that the model will take longer to fit the more components you use). Fit it to X_train, and then transform it on X_train, and X_test, creating the new variables P_train and P_test.

Now that we have data that has a more reasonable amount of columns, initialize a sklearn classification model, fit it to the training data, and record the accuracy score on the testing data. What score did you get, and how does this compare to the benchmark score you set earlier? Once you have this, try to optimize this process by using a different amount of PCA components, using different models and tuning the hyperparameters for the models you use. What was the highest accuracy score you managed to get on the test set?

Visualization

Having a model that can accurately predict the category of the images is great, but from a clients perspective, it is very important to create visualizations that are easy to look at/follow that convey important information, like how well the model is performing on each category and what it is actually doing to create these predictions.

First create a confusion matrix of the prediction values for P_test vs the actual values. See https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html. This is just an array that represents the confusion matrix. You will have to do some research in order to create a nice looking plot of this confusion matrix. I would recommend using the seaborn heatmap() method. If you use this, you can change the color mapping with the cmap argument, to 'Blues' and you can add x and y labels with plt.xlabel and plt.ylabel (despite using seaborn to create the figure). This way it is possible to distinguish which axis is the predicted values vs the actual values.

Next, we reduced the original data down to their first 10 or so PCA components, but what do these components actually represent? The PCA instance you initialized to reduce the dataset has a components_ attribute. This will return an array of arrays each with shape (14400,) that represent vectors pointed in the direction of highest variance within the original dataset. The number of arrays should be the value you specified for n_components. Convert these vectors back to their original shape of (80, 60, 3), and plot the corresponding images of these principal component vectors using the matplotlib imshow() method. You may have to multiply these vectors by a scalar (eg 100) to get the image to show anything. The higher number you use, the brighter your images will be. You should see pictures that look like something out of a strange painting or science fiction film.

One way to think of this if you are familiar with linear algebra is that these images are basis vectors (or basis images) that live in a subspace of the original vector space for the images. To an approximation every image in our images folder can be expressed as a linear combination of these basis images. The more PCA components you use, the closer this approximation of the images will get to the real thing. By using PCA, we are reducing the dimensionality of these images to the amount of each PCA component is needed to reproduce these images, cut off by the number of components we decided to use.

To make this more believable, you can compare any one of the actual images to the approximation of that image using the first PCA components. Load one of the images to an array using imread(). Then reduce the dimensionality of this array by flattening it, and then applying the transform() method on this array from the PCA instance you initialized earlier to fit to the training set. You may need to use reshape() to change the shape of this array to a 2 dimensional array with one row and 14400 columns (this is not quite the same as a 1D array). The approximation to the original flattened array is $\sum_i reducedarr[i] * components[i]$ where reducedarr is the PCA transformed flattened array, and components is the array of PCA component vectors (pca.components_). Once you have this calculated out, reshape it back to (80, 60, 3) and plot an image of this array using imshow(). Make a plot of the original array which should show the original image right underneath this image for comparison. The first image should look vaguely similar to the original image. The more pca components you use, the more similar this picture will become to the original image.

Once you have completed these steps, you are done with the assignment, though if you feel compelled to create more visualizations/results from your analysis feel free to include them in the notebook. Push the jupyter notebook to github, but do not attempt to push the dataset to github. There is a limit on how much data can be pushed to a github repository.

What we have done here only scratches the surface when it comes to image classification techniques. A more advanced approach is to create a neural network that detects a “target” image from a larger image, and classifies that image by certain features of the image, like the angle and locations of all the edges, etc. See https://towardsdatascience.com/wtf-is-image-classification-8e78a8235acb for an introduction to this topic.