Machine Learning Engineer Nanodegree

Capstone Project

Rashid Kamran
May 28, 2018

Proposal https://github.com/rkamran/MLE/blob/master/capstone/proposal.pdf

---

I. Definition

Project Overview

I recently came across an interesting study "A Diverse Benchmark Dataset for Multi-Paradigm Facial Beauty Prediction (FBP)" [1]. The study mainly assessed the facial attractiveness that is consistent with human perception.

In my project I want to replicate the results of the study and take it fun step further to deploy the trained model to an iOS app that would help people capture best of their faces (aka selfies) using an indicator on the screen.

Why use machine learning?

It is very hard if not impossible to codify perceived characteristics and relative measurement such as beauty and attractiveness. A rule-based system will suffer from complicated structures to capture the nuances attached with such problems. Machine learning especially deep learning with its ability to "learn" those nuances hidden in the complicated features is our best bet to even imagine any such system. Unlike for example checkered board where at every stage we can calculate all possible moves and come up with a non-ml solution while for this problem we can only construct handful of patterns in a non-ml way which will also suffer from high bias of the beholder (rule creator). On the other hand, with sufficient labelled data of otherwise perceived "adjective" we can easily turn a very complicated problem into "just another" regression problem.

Problem Statement

This is a classic image analysis problem but rather than classifying we are assigning a numerical score between 0-5 to measure attractiveness based on the faces in the dataset. Higher score perceived to be more attractive based on the score provided in the dataset. The utility of the original regression seems limited as it would be unwise to put a score on a face but by using the technique I will find a way to help people capture their best-looking faces.

Metrics

For a simple regression albeit on image data we will be relying on MSE evaluation metrics. I will try to compare against study's results using 90/10 train/test split and 90/10 train/validation split during training.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2.$$

For anecdotal proofs I have also visualized the results of the model using some celebrities, personal, family and friend's photos.

Our final output will be a number after all the feature extraction and processing. It's usually work better in regression problem to penalize large error relative to their contribution. That's one of the reason I chose MSE as opposed to MAE in hopes to converge faster and better. With this dataset and relative small range of the output I don't see any reason to dismiss MAE but avoiding absolute value for now.

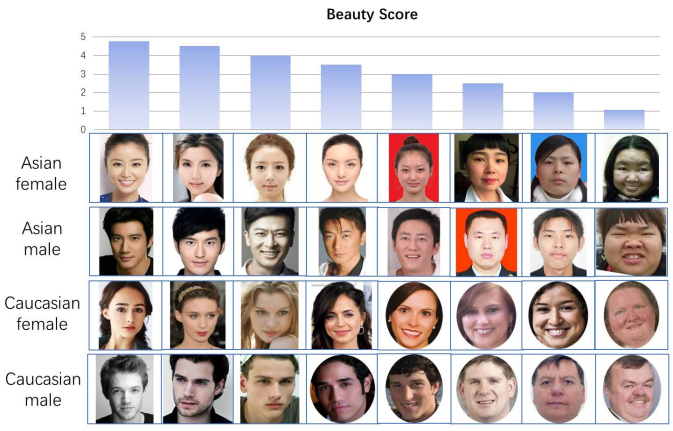
II. Analysis

Data Exploration

For this project I am relying on the original dataset [3] of 5500 face images (Asians and Caucasian male/female). Every face has given a score between 0-5 of perceived attractiveness. All images are 350x350 in shape. Out of 5500 images 4000 faces are Asian(male/female) so it would be interesting at least anecdotally to see the effect on other faces not represented in the dataset.

Here in Fig 1. I am showing an example from the original paper that shows how the dataset is actually distributed.

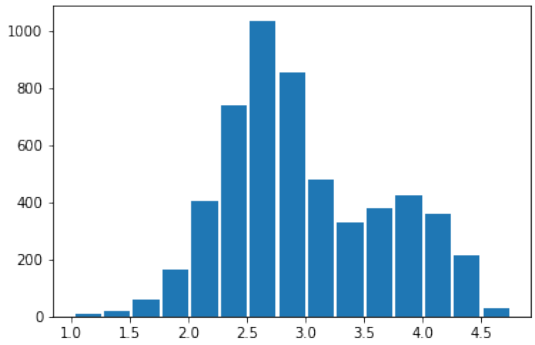
Fig 1. Scoring in the dataset



Exploratory Visualization

Below is a simple histogram of the scores in the original data. This gives us a glimpse of what kind of dataset we have in our hand. It makes as majority scores are regressing to the middle when it comes to attractiveness. It is important to note that there are some celebrity faces in the dataset and that might explain some medium towers around 4.0.

Fig 2. chart represents the score distribution in the data.



Algorithms and Techniques

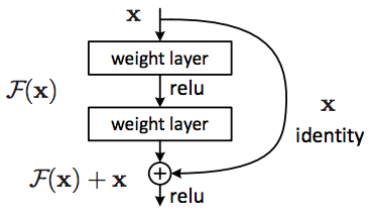
For this particular problem I am going to split it into two problems. First feature extraction and for that I am planning to use fully trained ResNet50. This particular deep residual network is really good in extracting features from image data.

Let's break it down a little bit and discuss some of the key ideas behind this decision.

Transfer learning. With smaller image datasets it's essential to find models that are trained on larger diverse or similar datasets, so they can help us in "feature extraction". It is unusual and not a common practice to train a very deep network end to end for individual project.

ResNet50 – A residual network as feature extractor. Deep convolutional networks are not easy to train specially when depth grows to 50 as in ResNet50. ResNet relies to a micro architecture called residual module [5].

Fig 3. A residual module—the building block of residual networks. [5].



Note the addition of the input to the output of the CONV + RELU + BN. It is called identify mapping and since it's added to the output hence the word residual. Unlike traditional networks that learns a function **y = f(x)** the residual network is learning **y = f(x) + id(x)** where id(x) is the identify function.

With this approach network can go really deep, learn fast and with higher learning rate as input is included with every residual block.

In our case we will take a pretrained network and remove the top layer (classifier) and replace it with a regressor. In our final implementation it will be a dense layer with one output. Conv Nets are great in feature extraction and since ImageNet dataset includes a whole SynSet[7] called person it is perfect for us to consider it extremely close to our dataset and not try to add additional conv layers on top.

Preprocessing Images – Always a good idea to extract and store features when it comes to stationary models in the pipeline like pre-trained ResNet50. In this implementation I will use Keras' image utilities to feed the images and store the last layer activations as feature inputs for regressors.

Regressor - In the second part I am treating this problem as a simple regression problem. That will make it much easier to train for benchmark and custom solution. I will start with one dense layer and will increased he capacity as needed.

Benchmark

I am going to establish a benchmark by using out of the box Support Vector Regressor and use it on top of extracted features. This will give me a baseline to meet or beat.

Please Note: Actual calculation is being done below after data preprocessing.

III. Methodology

Preprocessing

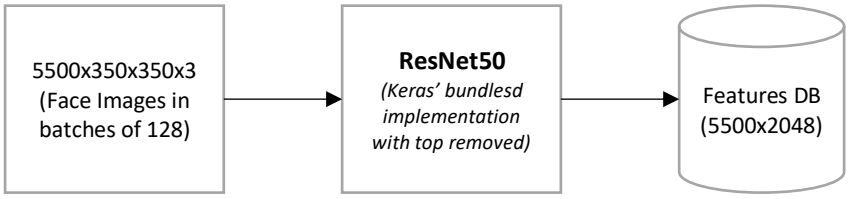
For dataset involving images it's always a good idea to preprocess, extract feature and serialize. This step will result in faster training time later. It would be much easier to create custom generator later.

Preprocessing Pipeline:

Step 1. Create HDF5 database with *images* and corresponding *score* tables

Step 2. Load batches of images *(batch size 128 was used in this implementation)* from the dataset and pass it through ResNet50.

Fig 4. Preprocessing pipeline



Here's a block of code that run the image data through ResNet50. It's a pretty straight forward implementation and used Keras image processing utilities to load and convert images into 4D tensors.

Fig 5. Feature extraction using ResNet50

```
def extract_feature(image_files=[], input_shape=(350, 350, 3)):
    images = []
    for image_file in image_files:
        img = preprocess_input(img_to_array(load_img(image_file)))
        images.append(img)
    images = np.stack(images)
    model = ResNet50(weights="imagenet",
                     include_top=False,
                     input_shape=input_shape,
                     pooling="avg")
    return model.predict(images)
```

Complete implementation can be found here and a GPU is recommended to run this code.

Implementation

The first thing I am doing is to establish a baseline result. The way I thought it's better to process is not involve feature extraction as part of the main solution and put the baseline and actual implementation on top of the extracted features.

Baseline - Support Vector Regressor.

Support Vector Machine is widely used in classification problems and tends to handle high dimensional data very well. The idea behind support vector machine is to represent features in n-dimensional (n=number of features) space as support vectors and find hyper plans separating those points to identify classes. The theory behind SVM is the maximum margin classifier [8] which is a hypothetical classifier that maximize the distance of data points from the decision boundaries.

For establishing baseline, I am using `sklearn.svm.SVR`. As per documentation

"The model produced by support vector classification (as described above) depends only on a subset of the training data, because the cost function for building the model does not care about training points that lie beyond the margin. Analogously, the model produced by Support Vector Regression depends only on a subset of the training data, because the cost function for building the model ignores any training data close to the model prediction." [9]

Since we have already extracted features, so we can run it pretty quickly on a relative small but high dimensional dataset of 2048 dimensions. In our implementation we are using default configuration with the following parameters

`Kernel='rbf', degree=3, epsilon = 0.1 and C=1.0`

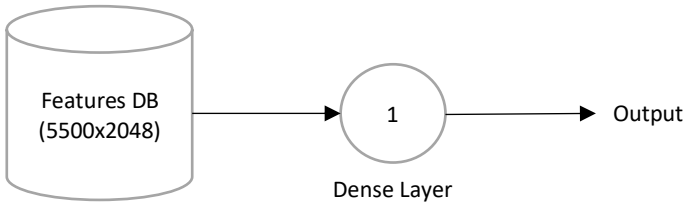
The loss function as I mentioned earlier will be Mean Squared Error for our baseline and proposed solution. It's the same old trusted function and can be described as

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2.$$

Dense Layer implementation

In the final implementation I am adding a dense layer on top of the extracted features for simple regression. After running the ResNet50 feature extractor this problem has become very simple.

Fig 6. Dense Layer Regressor



Step 1. Load feature sets from the DB (2048 features per image)

Step 2. Use a single layer dense network as regressor to assign score between 1 to 5.

Step 3. Use MSE to determine the error and continue improving.

I am also using 90/10 split for taring and testing. 10% data is used as validation set during training. Notice how fast the training is due to the fact that the hard part has already been taking care of. Full implementation can be found here. GPU recommended for running this code.

Refinement

For optimization I have decided to use `Adam` optimization with a learning rate or `0.0001`. The results converge rather quickly probably because of the data size so to keep it going for at least `20` epochs I have also added a learning rate decay or **learning rat/epochs**. Here's a complete compilation of the model.

Fig 7. Model optimization

```
model.compile(loss=loss,
              optimizer=Adam(lr=lr, decay=lr/epochs),
              metrics=["acc"])
```

IV. Results

Model Evaluation and Validation

For training and evaluating the model I initially did a **90/10** split to set aside 10% data. And during training further split the data into 90/10 to have epoch level validation. Both splits shuffle the dataset as well.

Fig 8. Model Compilation

```
trainX, testX, trainY, testY = train_test_split(features, labels, test_size=0.1, shuffle=True)

model = build_model((features.shape[-1],))
model.compile(loss=loss,
              optimizer=Adam(lr=lr, decay=lr/epochs),
              metrics=["acc"])

H = model.fit(trainX, trainY,
              epochs=epochs,
              batch_size=batch_size,
              validation_split=0.1,
              callbacks=[ModelCheckpoint(model_file, monitor="loss", save_best_only=True)])
```

For single dense layer regressor I am using the following hyper parameters and using learning rate decay by using number of epochs.

Table 1. Hyper parameters

Learning Rate	0.0001
Epochs	20
LR Decay	0.0001/20

Observations were made using SVR (as described above in the baseline) and also calculated both Mean Squared error and Mean absolute error for comparison purposes.

$$\mathrm{MAE} = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j| \qquad \mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2.$$

Table 2. Model Results against baseline

Loss	SVR	Dense Layer
MSE	0.0352125165231629 7	0.0315726697444915 8
MAE	0.1318955946471878	0.0328099466860294 34

Also ran the model by introducing a hidden layer with ReLu activation of size 128 to increase capacity but didn't see any significant difference.

Table 3. Model Results single vs 1 hidden

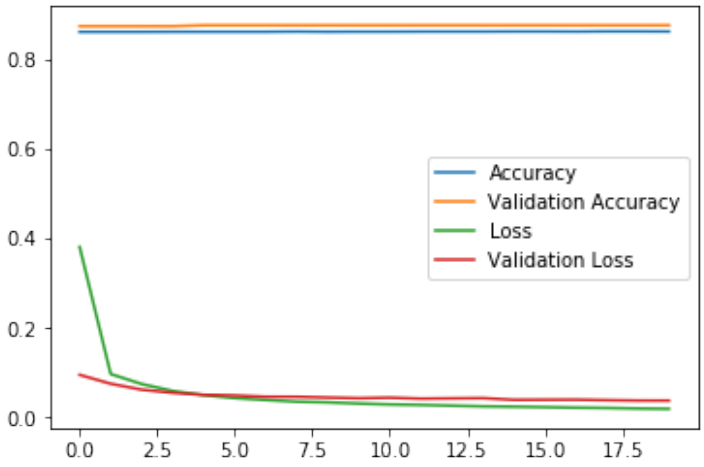
Loss	Dense Layer	Hidden Layer (128 ReLu)
MSE	0.03157266974449158	0.030313672497868538
MAE	0.032809946686029434	0.05008156970143318

Justification

The below graph provides a visualization of how 20 epoch training immediately converged and didn't improve much. I could easily stop the training after 5 epochs.

I believe well-structured data, a good feature extractor like ResNet50 and tilt towards mean immediately get to maximum accuracy which is better than the original study itself [6].

Fig 3. Chart showing the 20 epochs training and validation scores.



V. Conclusion

It's very hard to get similar dataset or create a large split for testing while the actual size is only 5500 so I am resorting to some anecdotal proofs by using some faces which I (eye of the beholder) personally think should get high/low score.

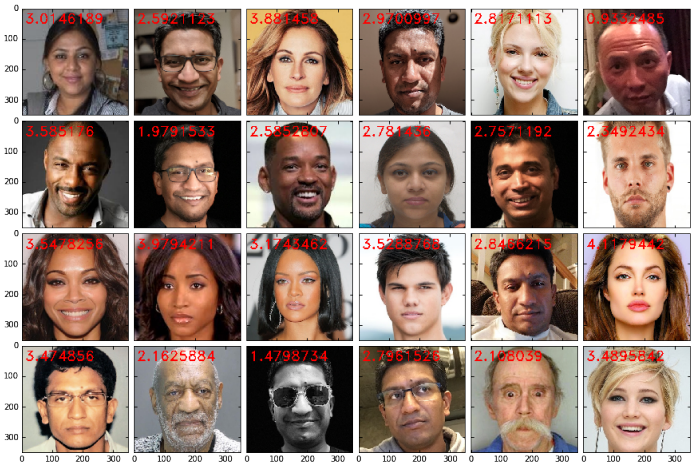
Free-Form Visualization

Anecdotal proof and visual reassurance

Fig 4 shows a grid of some personal, friends, random folks and celebrity photos. Model seems to be assigning high scores to celebrities and regressing back to mean when it comes to normal faces. In some cases, I tried to stylize and was able to get some good score.

Note that how **Angelina Jolie**, **Julia Roberts** and **Idris Elba**'s face got high score although there were no black faces in the dataset.

Fig 4. Scores generated on random faces



iOS App to also validate how model is working

Note: The iPhone application source was also uploaded in the project repo [4]

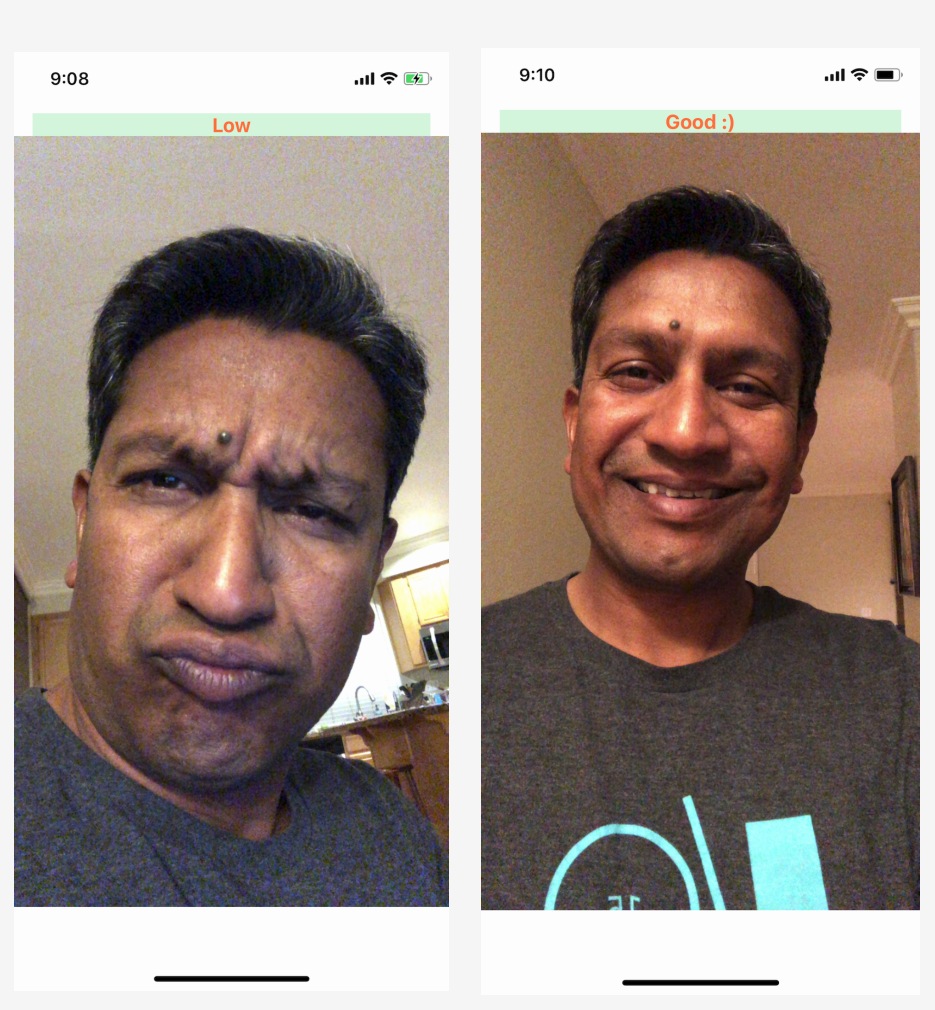
Rather than providing a score I am classifying a selfie pose as `Low`, `GOOD` and `Awesome`. The idea is to help people capture best faces when the beholder agent things it looks attractive.

Here's a breakdown of categories

```
Low < 2.5 2.5 > GOOD < 4.0 Awesome >= 4.0
```

Here I am pretending to be frowny and happy and getting the treatment from the app accordingly. I run the model every 10 frames, so it updates pretty quickly as you adjust faces.

Fig 5. iOS screenshot in real time scoring



Reflection

This is a tough problem, first of all assigning a numeric score to attractiveness looks pretty random but then we can all agree that majority of the world is not Miss Universe and then it kind of make sense.

The second aspect which I found really interesting is that how good ResNet50 is in detecting features from an image. At least in my testing which I ran quite a few times to make sure that the's not a fluke it helped converge the model pretty quickly. I still see it as too good to be true scenario and looking for something I missed because my scores are way better than the study itself. That might be because of the upfront feature extraction.

The third thing which I learnt from Dr. Adrian Rosebrock of pyimagesearch that serializing activations after running through a complex model is way efficient and faster compared to running it in the main pipeline. It enabled me to experiment and play with other SKLearn elements without ever worrying about run time or memory issues.

Improvements

I believe the dataset is way too small. I will take it to next level by doing the following

1. Use similar data to create a more elaborate profile of what a healthy good-looking face should be. Of course, there was absolutely no diversity in the dataset and it was very biased towards what it knew
2. Combine the app with object detection technique and run it on multiple faces at the same time