Camera Image Classification
Written by: Nick Jaton
Fall 2020

Introduction:

In 2018, IEEE ran a Kaggle competition to see who could develop the best camera prediction algorithm[1]. The competitors were given both a training and testing image set comprised of images from 10 different phone cameras. More information on this dataset will be given below in the section labeled as "Data". For this project, a model will be developed to predict which camera took each image in the test set. The goal for this project is to develop an algorithm that will have a score of .95 using the method provided by Kaggle. The function used to determine accuracy is given below. This formula was taken directly from the evaluation page of the competition.

Accuracy Function

$$\text{weightedaccuracy}(y, \hat{y}) = \frac{1}{n} \sum i = 1^n \frac{wi(yi = \hat{y}i)}{\sum wi}$$

n = # of samples
y = true camera label
$\hat{y}$ = predicted camera label
w_i = .7 for unaltered images and .3 for altered images

Data:

The training dataset provided by IEEE contains 275 full images from 10 different cameras. List of cameras used is located below as Figure 1. The images were all taken from a labeled phone. In some instances more than one phone of its model was used in the data collection process. It is important to note that half of the images have been slightly modified in some way. This could be an adjustment to the JPEG compression qualifty factor, resizing via bicubic interpolation, or a gamma correction. The modified images are labled as "_manip" where the the original images are labeled as "_unalt". Figure 2 is an example of one of the iphone 4s images that was given for this competition. Under Properties $->$ Details, the cameras contain information on the cameras used. This includes ISO speed, F-stop, Focal Length, even camera model. This information is only included in the training images. The dataset does not have universal ISO, F-stop, etc values. Instead, the best optimal value was produced by the camera. In other words, the automatic option was used instead of aperture or shutter priority. The only information retained on the test set are resulution (96 dpi), Bit depth (24) and Dimensions (512 x 512). The training set does not have universal dimensions. This means that each image will have to be scaled to a constant value.

Figure 1: List of Cameras Used

1. Sony NEX-7
2. Motorola Moto X
3. Motorola Nexus 6
4. Motorola DROID MAXX
5. LG Nexus 5x
6. Apple iPhone 6
7. Apple iPhone 4s
8. HTC One M7
9. Samsung Galaxy S4
10. Samsung Galaxy Note 3

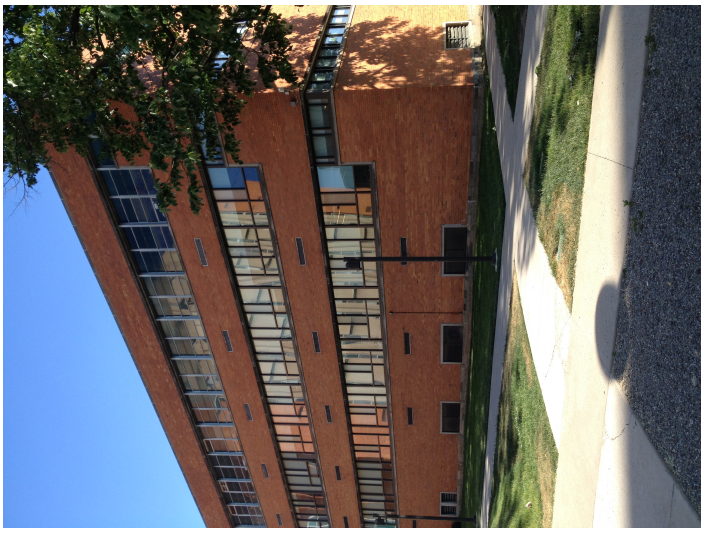

Figure 2: Example of Iphone 4s Image.

Processing was needed to input these images into a machine learning algorithm. For context, python was used for this processing and will be used for future modeling. The training images were given by IEEE in a folder with the same name of the camera that was used to create the image. Therefore, scripts were used to generate a list of all image addresses and the label for each of said

images. Since images hold quite a bit of data, the pillow library was used to reduce the image size to 256 x 256. Originally, the training images are much larger than 256 x 256. For model development, using reduced sizes such as 256 x 256 or 128 x 128 allows for much quicker prototyping. Once a prototype promising model is found, it may be worth while to increase the size of the input. This was done with the hope of reducing the amount of time needed to train neural networks in the future. All of the arrays generated were placed into a single numpy array called x_train. Figure 3. shows the printed output of the 500th image in the testing dataset. The testing data was also placed into a similar array called x_test. For the label values (named y_train), all of the folder names were collected for each testing images and added to a numpy array. Each column of the array represents a different camera and the row represents the image. The y_train array is shown in Figure 4. This data should now be in a format that can be used in libraries such as TensorFlow to develop statistical models[2].

Figure 3: Resulting Image Data

```
>>> print(x_test[500])
[[[0.05490196 0.06666667 0.03529412]
  [0.05882353 0.06666667 0.04313725]
  [0.05882353 0.0627451  0.04313725]
  ...
  [0.1254902  0.16470588 0.08235294]
  [0.1372549  0.18823529 0.07843137]
  [0.18431373 0.21568627 0.11764706]]

 [[0.05490196 0.0627451  0.04313725]
  [0.05490196 0.06666667 0.04313725]
  [0.05098039 0.05882353 0.03921569]
  ...
  [0.10588235 0.15294118 0.06666667]
  [0.16862745 0.22745098 0.10980392]
  [0.21960784 0.26666667 0.1372549 ]]

 [[0.05098039 0.05882353 0.03921569]
  [0.06666667 0.07843137 0.05098039]
  [0.05882353 0.07058824 0.04313725]
  ...
  [0.08627451 0.1372549  0.05098039]
  [0.19215686 0.25098039 0.1372549 ]
  [0.18039216 0.20392157 0.09411765]]

 ...

 [[0.9372549  0.66666667 0.47058824]
  [0.93333333 0.66666667 0.4745098 ]
  [0.94509804 0.68235294 0.48235294]
  ...
  [0.6        0.58431373 0.52156863]
  [0.56470588 0.5372549  0.47058824]
  [0.49803922 0.4745098  0.41176471]]

 [[0.93333333 0.6627451  0.46666667]
  [0.91372549 0.65098039 0.4627451 ]
  [0.9254902  0.66666667 0.46666667]
  ...
  [0.5372549  0.52156863 0.4627451 ]
  [0.58823529 0.56078431 0.49019608]
  [0.55686275 0.5372549  0.46666667]]

 [[0.9254902  0.65098039 0.45882353]
  [0.89019608 0.63137255 0.45490196]
  [0.9254902  0.65882353 0.46274511]
  ...
  [0.48627451 0.4627451  0.41176471]
  [0.58039216 0.55294118 0.49019608]
  [0.56862745 0.54117647 0.46666667]]]
```

Figure 4: Label information for the training table.

```
>>> y_train
array([[1, 0, 0, ..., 0, 0, 0],
       [1, 0, 0, ..., 0, 0, 0],
       [1, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 1, 0, 0],
       [0, 0, 0, ..., 1, 0, 0],
       [0, 0, 0, ..., 1, 0, 0]], dtype=uint8)
>>> len(y_train)
2750
>>> len(y_train[0])
10
```

Methods:

Various methods were used to attempt to predict the camera being used to generate each image. Being that the data used in this analysis is image data, a convolutional neural network was a natural starting point. This lead to implementation of convolutional extreme gradient boosting (ConvXGB) as described in Ramin and I's presentation. Implementations of k-means and random forest were also produced using the extracted features from the aforementioned CNN. Weighted random forest was not used on this dataset because, there is no preferred result.

1. **Convolutional Neural Network**
   The name of the game for CNN's is feature extraction. The CNN architecture generally starts with a combination of convolutional layers and pooling layers. At this point the layers are trying to extract features and reduce the overall size of the resulting data. The method used to calculate the extracted data can differ based of architecture. The conventional layer will take a kernel and move it over the given matrix. During this process the values of the kernel will be multiplied by the values of the matrix section. The summation of these values will become a value in the new matrix that is used by the next layer. Global max pooling and max pooling were both used by the CNN used. The max pooling matrix "moves over" the matrix like the convolutional layer but, will grab the max value in the process. Global max pooling just grabs the largest value of the given matrix. At this point the data is flattened and put into a neural network model. The model is based off of "Keras Simple CNN Starter" found on the Kaggle page[4]. Both 20 & 25 epochs were used for this analysis. Please Figure 5 below for the exact model used.

Figure 5: Convolutional Neural Network Used.

```
Model: "model_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         [(None, 256, 256, 3)]     0
_________________________________________________________________
batch_normalization_4 (Batch (None, 256, 256, 3)       12
_________________________________________________________________
conv2d_24 (Conv2D)           (None, 256, 256, 16)      448
_________________________________________________________________
conv2d_25 (Conv2D)           (None, 256, 256, 16)      2320
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 85, 85, 16)        0
_________________________________________________________________
dropout_12 (Dropout)         (None, 85, 85, 16)        0
_________________________________________________________________
conv2d_26 (Conv2D)           (None, 85, 85, 32)        4640
_________________________________________________________________
conv2d_27 (Conv2D)           (None, 85, 85, 32)        9248
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 28, 28, 32)        0
_________________________________________________________________
dropout_13 (Dropout)         (None, 28, 28, 32)        0
_________________________________________________________________
conv2d_28 (Conv2D)           (None, 28, 28, 64)        8256
_________________________________________________________________
conv2d_29 (Conv2D)           (None, 28, 28, 20)        5140
_________________________________________________________________
global_max_pooling2d_4 (Glob (None, 20)                0
_________________________________________________________________
dropout_14 (Dropout)         (None, 20)                0
_________________________________________________________________
flatten_4 (Flatten)          (None, 20)                0
_________________________________________________________________
dense_10 (Dense)             (None, 20)                420
_________________________________________________________________
get_dense (Dense)            (None, 10)                210
=================================================================
Total params: 30,694
Trainable params: 30,688
Non-trainable params: 6
_________________________________________________________________
```

2. **Conventional Extreme Gradient Boosting**
   Extreme Gradient Boosting is a algorithm based off of boosted trees. This means that the data will generate a series of possible trees. Each of these trees is a set of classification and regression trees. XGBoost will take these trees and optimize these trees to best fit the data that was presented[4]. The implementation of the algorithm was then applied to the "get_dense" layer of the CNN[5][6]. This allows the algorithm to use the image features that were previously discussed. This model was developed using the xgboost module in python and is shown in Figure 6.

Figure 6: The XGBoost Model Generated from the output of the Convolutional layers.

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints='',
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints='()',
              n_estimators=100, n_jobs=0, num_class=10, num_parallel_tree=1,
              objective='multi:softprob', random_state=0, reg_alpha=0,
              reg_lambda=1, scale_pos_weight=None, subsample=1,
              tree_method='exact', validate_parameters=1, verbosity=None)
```

3. **Convolutional Random Forest**
   Convolutional Random Forest (ConvRF) was virtually the same as the last. The big difference here is the argument of bagging vs boosting. The RandomForestClassifier() method in scikit-learn[7] will using bagging by default. This differs from XGBoost which of course, uses boosting. Bagging involves building many trees and generating and average of the trees. The boosting algorithm in XGBoost relies on the ideal of optimizing the loss function as each new tree is created. This model was expected to do worse than the previous but, allowed for an interesting comparison between the methods.

4. **Convolutional K-Means**
   As a curiosity, K-Means was also applied to the feature data of the convolutional neural network. The idea behind K-Means is to split the given samples into clusters by reducing the squared Euclidean distance. During each iteration of the algorithm, the mean of the data under each centriod will be calculated. This will then be used to determine the next centriod location. This will continue until convergence. Since there were ten possible cameras, ten clusters were used. The K-Means implementation was also developed using python's scikit-learn module[8]. The idea was to have the algorithm develop clusters for each possible image label. This did not prove effective. Please reference the results section for more information.

5. **Convolutional Support Vector Machines**
   Support Vector Machines (SVM) was the final model that was attempted for this analysis. The process is the same as the others. The data from "get_dense" was extracted from the CNN model and pumped into scikit-learn's svm function[9]. This model was trained using the polynomial kernel due to its performance over linear and rbf. When testing on the CNN model with 25 epochs, the models were able to produce accuracy scores of .487 for polynomial, .453 for linear, and .475 for rbf. SVM works by optimizing a hyperplane around individual support vectors. The support vectors are the data points closest to the hyperplane. As one would expect, the closest data points would also be the ones that are the hardest to predict. The planes around the support vectors are then used to classify the image data[10].

Results

In the end, the models developed were not nearly as accurate as the top submissions on Kaggle. Instead of using the algorithm given by kaggle, accuracy was used to compare the models. As one would expect, accuracy of the K-Means model is less than the more advanced models such as the Neural Network and extreme gradient boosting. The CNN, ConvXGB, ConvRF , and ConvSVM models all performed fairly similarly. The model based on SVM ended up being able to produce the best results of all algorithms. ConvXGB and ConvRF have a tendency to swap places in the leader board depending on the training result of the CNN. F1-score also tended to follow the same leaderboard as accuracy. It is important to recall that the results below come from one of two CNN models with either 20 or 25 epochs. It is very possible that these models could be improved with a increase in training time. Please view Figure .7 for the exact accuracy data collected. Figures 8 through 17 show the confusion matrix and the classification matrix for each model.

Figure 7: Model Accuracy

Model Name	Accuracy (20 Epochs)	Accuracy(25 Epochs)
CNN	.3971	.4407
ConvXGB	.4504	.4383
ConvRF	.4455	.4649
Conv K-Means	.1332	.1113
ConvSVM	.4891	.4867

Figure 8: CNN Confusion Matrix at 25 Epochs

```
>>> print(confusion_matrix(y_test_sub_argmax, y_vals_cnn_argmax))
[[16  3  6  5  3  1  1  0  0  1]
 [ 6  9  3 12  1  2  0  1  0  2]
 [ 2  0 18  3  6  6  1  1  0  2]
 [ 1  7  1 19  2  4  0  3  0  5]
 [ 6  1  1  3 25  7  1  0  0  5]
 [ 0  0  2  5  2 15 11  0  0  5]
 [ 0  0  0  0  0  6 22  0  1 10]
 [ 1  2  4  5  2  7  0 15  0  0]
 [ 0  0  0  0  0  1  3  0 23 20]
 [ 0  1  1  0  3 12 10  0  2 20]]
```

Figure 9: CNN Classification Report at 25 Epochs

```
>>> print(classification_report(y_test_sub_argmax, y_vals_cnn_argmax))
              precision    recall  f1-score   support

           0       0.50      0.44      0.47        36
           1       0.39      0.25      0.31        36
           2       0.50      0.46      0.48        39
           3       0.37      0.45      0.40        42
           4       0.57      0.51      0.54        49
           5       0.25      0.38      0.30        40
           6       0.45      0.56      0.50        39
           7       0.75      0.42      0.54        36
           8       0.88      0.49      0.63        47
           9       0.29      0.41      0.34        49

    accuracy                           0.44       413
   macro avg       0.49      0.44      0.45       413
weighted avg       0.50      0.44      0.45       413
```

Figure 10: ConvXGB Confusion Matrix at 25 Epochs

```
>>> print(confusion_matrix(new_val, y_pred_xgb))
[[18  2  5  2  3  3  1  2  0  0]
 [ 6 18  2  7  1  0  0  2  0  0]
 [ 1  3 19  1  4  3  1  6  0  1]
 [ 3 10  3 11  6  4  0  3  1  1]
 [ 6  3  2  7 22  5  1  2  0  1]
 [ 1  2  4  5  3  5 11  2  2  5]
 [ 0  0  0  0  1 10 17  0  4  7]
 [ 0  5  5  5  0  0  0 20  0  1]
 [ 0  0  1  0  0  2  1  0 33 10]
 [ 0  2  3  2  1 12  4  1  6 18]]
```

Figure 11: ConvXGBB Classification Report at 25 Epochs

```
>>> print(classification_report(new_val, y_pred_xgb))
              precision    recall  f1-score   support

           0       0.51      0.50      0.51        36
           1       0.40      0.50      0.44        36
           2       0.43      0.49      0.46        39
           3       0.28      0.26      0.27        42
           4       0.54      0.45      0.49        49
           5       0.11      0.12      0.12        40
           6       0.47      0.44      0.45        39
           7       0.53      0.56      0.54        36
           8       0.72      0.70      0.71        47
           9       0.41      0.37      0.39        49

    accuracy                           0.44       413
   macro avg       0.44      0.44      0.44       413
weighted avg       0.44      0.44      0.44       413
```

Figure 12: ConvRF Confusion Matrix at 25 Epochs

```
>>> print(confusion_matrix(new_val,y_pred_rf))
[[16  3  5  2  4  3  1  2  0  0]
 [ 3 22  2  6  1  0  0  1  0  1]
 [ 1  3 20  2  5  2  1  4  0  1]
 [ 1  8  3 18  3  2  1  4  0  2]
 [ 5  5  4  6 20  3  2  1  1  2]
 [ 0  2  5  6  2  8 11  2  1  3]
 [ 0  0  1  0  1  4 25  1  4  3]
 [ 0  2  6  7  0  0  0 21  0  0]
 [ 1  0  1  0  0  2  2  0 28 13]
 [ 0  2  3  2  1 13  7  1  6 14]]
```

Figure 13: ConvRF Classification Report at 25 Epochs

```
>>> print(classification_report(new_val, y_pred_rf))
              precision    recall  f1-score   support

           0       0.59      0.44      0.51        36
           1       0.47      0.61      0.53        36
           2       0.40      0.51      0.45        39
           3       0.37      0.43      0.40        42
           4       0.54      0.41      0.47        49
           5       0.22      0.20      0.21        40
           6       0.50      0.64      0.56        39
           7       0.57      0.58      0.58        36
           8       0.70      0.60      0.64        47
           9       0.36      0.29      0.32        49

    accuracy                           0.46       413
   macro avg       0.47      0.47      0.47       413
weighted avg       0.47      0.46      0.46       413
```

Figure 14: ConvK-Means Confusion Matrix at 25 Epochs

```
>>> print(confusion_matrix(new_val, y_pred_kmeans))
[[12  6  0  1  1  0  8  3  0  5]
 [ 1  7  0  0  0  0 24  2  0  2]
 [ 1  4  1  4  1  0  5  2  4 17]
 [ 0 15  1  0  0  0 18  3  3  2]
 [ 1 11  2 22  0  0  6  0  6  1]
 [ 0 10  3  0  8  0  2  1 14  2]
 [ 0  2  7  0 16  0  0  0 14  0]
 [ 0  6  0  1  0  0  5 19  2  3]
 [ 0  2 19  0  1 19  0  0  6  0]
 [ 0  3 20  2  3  1  0  0 19  1]]
```

Figure 15: ConvK-Means Classification Report at 25 Epochs

```
>>> print(classification_report(new_val, y_pred_kmeans))
              precision    recall  f1-score   support

           0       0.80      0.33      0.47        36
           1       0.11      0.19      0.14        36
           2       0.02      0.03      0.02        39
           3       0.00      0.00      0.00        42
           4       0.00      0.00      0.00        49
           5       0.00      0.00      0.00        40
           6       0.00      0.00      0.00        39
           7       0.63      0.53      0.58        36
           8       0.09      0.13      0.10        47
           9       0.03      0.02      0.02        49

    accuracy                           0.11       413
   macro avg       0.17      0.12      0.13       413
weighted avg       0.15      0.11      0.12       413
```

Figure 16: ConvSVM Confusion Matrix at 25 Epochs

```
>>> print(confusion_matrix(new_val, y_pred_svm))
[[15  5  5  3  2  1  1  4  0  0]
 [ 0 21  2 11  0  0  0  2  0  0]
 [ 1  0 23  5  4  2  1  2  0  1]
 [ 0 11  2 20  1  4  0  3  0  1]
 [ 4  5  4  7 21  5  1  2  0  0]
 [ 0  0  2  9  2 12  8  3  1  3]
 [ 0  0  0  1  0  6 24  0  1  7]
 [ 0  3  5  4  0  1  0 23  0  0]
 [ 0  0  0  1  0  2  3  0 25 16]
 [ 0  1  2  0  3 17  6  0  3 17]]
```

Figure 17: ConvSVM Classification Report at 25 Epochs

```
>>> print(classification_report(new_val, y_pred_svm))
              precision    recall  f1-score   support

           0       0.75      0.42      0.54        36
           1       0.46      0.58      0.51        36
           2       0.51      0.59      0.55        39
           3       0.33      0.48      0.39        42
           4       0.64      0.43      0.51        49
           5       0.24      0.30      0.27        40
           6       0.55      0.62      0.58        39
           7       0.59      0.64      0.61        36
           8       0.83      0.53      0.65        47
           9       0.38      0.35      0.36        49

    accuracy                           0.49       413
   macro avg       0.53      0.49      0.50       413
weighted avg       0.53      0.49      0.49       413
```

Though the models did not produce the strongest results, they do serve as a good starting point for this analysis. Models such as the CNN can require a lot of customization. With further modifications to this model, I suspect that a better accuracy can be derived from all models used. It will also be interesting to see if further modifications could start to show a clear winner between the two models.

Summary

The goal of this analysis was to develop an algorithm that was comparable to the methods used by top Kaggle competitors from the IEEE Camera Model Identification challenge. The dataset given by the competition was a collection of 2750 JPEG images from 10 different phone cameras. The image count was split evenly by each camera type. Each image was collected by an individual member of IEEE. The only modification to each image was bicubic interpolation and gamma correction. This was completed on half of the images. Once the images were collected, they were resized and pushed through a convolutional neural network. At which point, the final density layer of the CNN was pushed into XGBoost, Random Forest (Bagged), K-Means, and SVM in hopes of developing a highly predictive model.

The results section indicates that these models were not able to perform at the desired level. At an accuracy of .4891, ConvSVM was the best performing model. There is still much room for improvement. I am confident that there is more juice to squeeze on the initial CNN model. This could by done by increasing the image size of the training data and increasing the epochs used. SVM also has many tuning parameters in scikit-learn that were not investigated. Such an investigation could lead to more predictive models in the future. When researching the methods used by top contenders, it is clear that increasing the size of the training dataset through means such as Flickr can give a more accurate model. Additional modeling techniques such as XceptionNet[11] & inception_v3[12] should also be researched as they both made it into the top 5 submissions.

With all of that said, there is no reason to end this report on a negative note. By exploring this problem I was able to get a better understanding of image processing, python programming, and all the aforementioned models. Learning to combine convolutional neural networks with other statistical methods has proven to be incredibly useful for image classification. The result for the ConvSVM model is particularly interesting. There was no expectation that it would outperform the random forest methods, but it gathered a clear lead.

Resources

1. "IEEE's Signal Processing Society - Camera Model Identification," Kaggle. [Online]. Available: https://www.kaggle.com/c/sp-society-camera-model-identification/overview. [Accessed: 12-Nov-2020].
2. "Convolutional Neural Network (CNN) :   TensorFlow Core," TensorFlow. [Online]. Available: https://www.tensorflow.org/tutorials/images/cnn. [Accessed: 13-Nov-2020].
3. C. (2017, December 23). Keras Simple CNN Starter. Retrieved November 01, 2020, from https://www.kaggle.com/CVxTz/keras-simple-cnn-starter
4. "Introduction to Boosted Trees," Introduction to Boosted Trees - xgboost 1.3.0-SNAPSHOT documentation. [Online]. Available: https://xgboost.readthedocs.io/en/latest/tutorials/model.html. [Accessed: 12-Nov-2020].
5. S. Thongsuwan et al., ConvXGB: A new deep learning model for classification problems based on CNN and XGBoost, Nuclear Engineering and Technology, https://doi.org/10.1016/j.net.2020.04.008
6. Chekoduadarsh, "Starters Guide: Convolutional XGBOOST," Kaggle, 02-May-2020. [Online]. Available: https://www.kaggle.com/chekoduadarsh/starters-guide-convolutional-xgboost. [Accessed: 13-Nov-2020].
7. "3.2.4.3.1. sklearn.ensemble.RandomForestClassifier," scikit. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html. [Accessed: 13-Nov-2020].
8. "sklearn.cluster.KMeans2," scikit. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html. [Accessed: 12-Nov-2020].
9. "sklearn.svm.SVC," scikit. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html. [Accessed: 15-Dec-2020].
10. R. Berwick, "An Idiot's guide to Support vector machines (SVMs)." Available: https://web.mit.edu/6.034/wwwbob/svm.pdf
11. L. Verdoliva, "IEEE's Signal Processing Society - Camera Model Identification," Kaggle. [Online]. Available: https://www.kaggle.com/c/sp-society-camera-model-identification/discussion/49602. [Accessed: 16-Dec-2020].
12. G. Xu, "IEEE's Signal Processing Society - Camera Model Identification," Kaggle. [Online]. Available: https://www.kaggle.com/c/sp-society-camera-model-identification/discussion/49298. [Accessed: 16-Dec-2020].