Alan Wang

Amy Tang

Isaac No

Joy Richardson

Philip Huang

Tim Chen

Math 156

Professor Georg Menz

10 June 2022

Facial Emotion Recognition

Abstract **2**

1. Introduction **2**
1.1 Dataset 2
1.2 Data Preprocessing & Exploratory Data Analysis 4

2. Modeling **6**
2.1 Naive Bayes Classifier 6
2.2 Logistic Regression 7
2.3 K-means + Logistic Regression 8
2.4 K-Nearest Neighbors with PCA + CV 8
2.5 Random Forest 9
2.6 Support Vector Machine with PCA 10
2.7 Artificial Neural Network 11
2.8 Convolutional Neural Network 13

3. Conclusion **15**
3.1 Results 15
3.2 Afterword 16

4. Cited Works **17**

Abstract

With the exponential progress in artificial intelligence, one particular field that has been in development is facial emotion recognition. Much research today revolves around the deep learning process in recognizing something as abstract as human emotions. This process serves to have several types of complications based upon detecting specific facial features within a given set of parameters. There are many applications for this technology including, but not limited to, personalized advertisements based on customer satisfaction, percepting certain behaviors within given environments, and recreating or simulating actions for further purposes. Considering these applications, several classification methods serve to have higher efficiencies in processing given information while reaching a solvable state – a correct recognition and categorization. This research report aims to explore these differences, primarily focusing on the process of implementing the models and their achieved results.

Our group builds a facial emotion recognition machine learning model. We explore classification methods such as Naive Bayes Classifier, Logistic Regression, KNN, Random Forest, and SVM using scikit-learn. In addition, we perform cross validation to find the optimal hyperparameters in increasing efficiency or decreasing the margin of error. In the end, we implement an artificial neural network and a convoluted neural network in TensorFlow for such classification. Overall, we will utilize each of our models on the validation set and the testing set and use accuracy as the metric of the performance of each model.

---

1. Introduction

1.1 Dataset

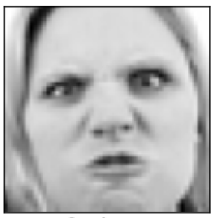
Our task is therefore to train a multi-class classification model for facial emotion recognition.

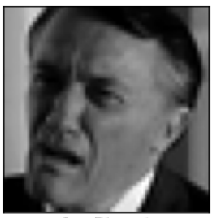
For our purposes, we use a dataset of grayscale faces, from a Kaggle competition dataset by Aaditya Singhal. Our dataset consists of 28709 faces, with each face consisting of 48 by 48 pixels, and each pixel is a value from 0 to 255. Each face is also centered and occupies similar space throughout the dataset.

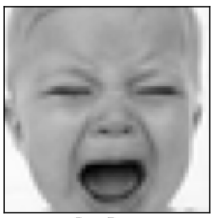
There are seven facial expressions covered in this dataset, represented with numeric labels, and the expressions for the labels are shown in the table below:

Label	**Emotion**
0	Angry
1	Disgust
2	Fear
3	Happy
4	Sad
5	Surprise
6	Neutral

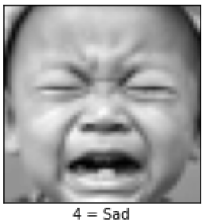
The following are one image from each of the seven categories:

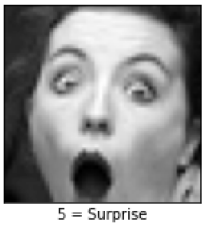

0 = Angry


1 = Disgust


2 = Fear


3 = Happy


4 = Sad


5 = Surprise


6 = Neutral

1.2 Data Preprocessing & Exploratory Data Analysis

The pixel information is stored as string in the original dataset, which is not trainable by any classifier. Thus, we cast each of them to an array of integers of length 2304, which represents all 48 by 48 pixels. Therefore, the number of features we have is 2304. For the purpose of training a convolutional neural network in Section 2.8 of this report, we will also keep a copy of the dataset with each image information stored as a 3-dimension 48 by 48 by 1 array of integers.

Normalization is one trick widely used in machine learning to improve the performance of certain models, as the great difference in the scale of the numbers could cause problems when attempting to combine the values as features during modeling, and the problem is especially serious when training a neural network. We therefore normalize all data points to mean 0 and standard deviation 1.

To discuss the performance of a model, we need to define the metrics first. Although metrics such as F-score more accurately captures the performance of a model by looking at both precision and recall, for a multi-class classification problem, we have to compute the per-class F-score: for the i-th class, F-score is defined as:

$$F_i = 2 \times (precision_i \times recall_i) \,/\, (precision_i + recall_i) ,$$

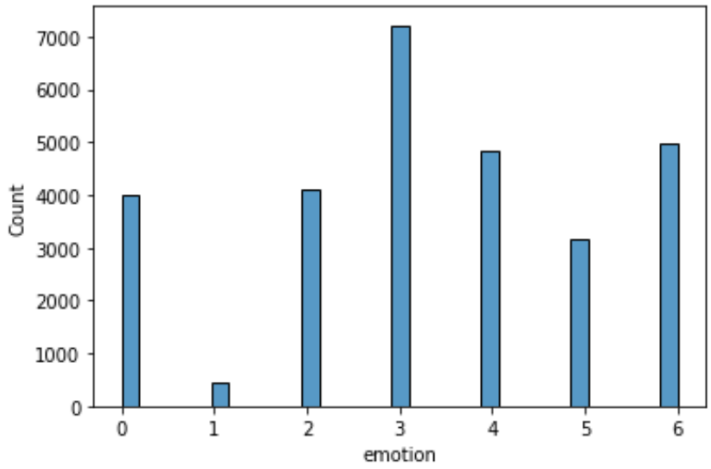
and then take the average of all per-class F-scores to get the macro F-score:

$$macro\ F = \frac{1}{k} \sum_{i=1}^{k} F_i$$

For deep learning libraries such as Tensorflow, such definition is enough for the moment as we can simply add an additional metric `macro f1`, but for traditional machine learning algorithms, macro F-score is not integrated: by default, calling the `score` function would calculate the accuracy. Thus, for convenience, we decide to use accuracy as our metrics, which is defined as:

$$accuracy = \frac{\#\ of\ correct\ predictions}{\#\ of\ total\ predictions}$$

Although accuracy is easier to calculate and implement in practice, we also need to ensure that we have a balanced dataset – in the case of a extremely imbalanced dataset, the model can just predict any input to the most dominant label and reach a high accuracy, even when the model is not good at all under any other metrics. Therefore, we first plot a histogram of count of labels:



The dataset is quite balanced, except for label 1 “Disgust” and label 3 “Happy”. We can overcome this issue by either oversampling the low-cost classes or subsampling the high-cost classes. For our project, since label 1 only has 436 instances, applying subsampling would result in insufficient training data. However, we only do this for the training set - we separate the validation set and the test set out before this step - as we have to keep the validation set and the test set “unseen” by the model when training. We apply both to the training set: either oversample or subsample each class so that each label has 3,000 instances. The validation set is used for preventing overfitting. We will use the same validation and testing data throughout the project to compare the performance on unseen data of various models.

2. Modeling

2.1 Naive Bayes Classifier

Naive Bayes is a set of methods using the Bayes classifier to perform supervised learning. The Bayes classifier is a probabilistic model of the form

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y)P(x_1, \ldots, x_n \mid y)}{P(x_1, \ldots, x_n)}$$

,

where y is the class variable and x_1 through x_n are various features. The "naive" assumption of Bayes is based on strong independence between the features x_1, ..., x_n. In other words, the covariances are assumed to be 0. It does not take into account the weight of one variable over another. For example, if trying to classify an orange, the classifier may take into account color, shape, and size. So a round, orange object with a diameter of approximately 3 inches would be considered an orange. The features color, shape, and size are assumed to contribute equally to the probability that this object is an orange. In this sense, all x variables are equally important to determining y.

Despite the simplicity of the classifier, Naive Bayes works well in real world situations as a document classifier and spam filter. It only requires a small amount of training data and is very fast. On the other hand, though it works well as a classifier, Naive Bayes performs poorly as an estimator, which is why we use it for classification purposes only.

Using scikit-learn in Python, we performed Gaussian Naive Bayes on our dataset. The model performed poorly, with training accuracy of 23% and validation and test accuracy at 20 and 21%, respectively. The confusion matrix also shows that the classifier did not work very well, with low values across the main diagonal. The values that were predicted with the best accuracy of 0.61

and 0.38 were expressions of surprise and sadness. Overall, the Naive Bayes classifier performed poorly on our dataset.

2.2 Logistic Regression

Logistic regression is a linear model for classification. In particular, it is a resourceful supervised machine learning algorithm used for binary classification problems, in which the target can be categorized. Not only is it an easy method to use to model the probability of a discrete outcome given an input, as opposed to linear regression, this method does not need to have a linear relationship between inputs and corresponding outputs. This is particularly due to its characteristic of utilizing a logistic function, which is bounded between 0 and 1, to which a nonlinear log transformation is applied to the odds ratio.

The logistic regression model can be represented by

$$p(C_1|\phi) = y(\phi) = \sigma(w^{-T}\phi) \text{ such that } \sigma(a) = \frac{1}{1+exp(-a)},$$

which is known as the logistic sigmoid function.

Furthermore, logistic regression minimizes the following cost function:

$$min_{w,c} \frac{1}{2} w^T w + C \sum_{i=1}^{n} log(exp(- y_i(X_i^T w + c)) + 1)$$

We use scikit-learn log regression package to run our data after cross validation separation. Logistic regression enables us to find the probability of new test data belonging to each class (in this case, supervised since we already have 7 groups of emotions). In general, we have 7 probability results for each new test data, and we put the new datapoint to the group of the highest probability.

This process continues for all test data, and we get a result for our final analysis. The score reported by constructing the code and functions imported in the library, we can also compute the normalized confusion matrix, which is a direct graphical representation of how well the model is. The overall accuracy score of 0.35 means logistic regression is probably not the best model, but still an improvement compared to random guessing.

2.3 K-means + Logistic Regression

K-means is a clustering algorithm which splits certain data into k amounts of clusters. As the number of clusters can be changed by the user, for our purposes, we use this method to divide the data as a parameter for the logistic regression method, which was defined earlier.

Much like that of what was described for the logistic regression model, the process is similar with the exception that we set a finite value to split the data into k number of clusters before moving onto calling logistic regression. K-means was also implemented through the usage of scikit-learn through Python. Since there are approximately five different ways in which people express each emotion, we get k = 5 * 7 = 35 to split the data for logistic regression.

However, the results were drastically different to that of logistic regression. In fact, in our case the accuracy of classifying the images correctly(under the “accuracy” metric) was significantly lower than the latter. Beside the factor of randomness, one other assumption for this result may have been due to the split in data that occured before implementing logistic regression, in which the sets of images were truncated within the process.

2.4 K-Nearest Neighbors with PCA + CV

K-nearest neighbors, or KNN, is a supervised learning method used for both classification and regression. The training data is labeled with a class, then the algorithm iterates through each point to find the closest neighboring points. Distance can be measured using Euclidean,

kernelized, and other methods of measuring distance. The classes of the k closest points are considered, and a majority-rule method of voting occurs. For example, say a point is waiting to be classified. Its 2 closest neighbors are classified as red. However, its next 3 closest neighbors are all classified as blue. Choosing k = 2 would result in the classification “red”, while k = 5 would classify the point as blue. Choice of k is arbitrary, but a common method is to use k = sqrt(n), where n is the number of observations in the data. A smaller k is more sensitive to noise in the data, while a larger k is more computationally expensive and often results in lower complexity (a “smoother” boundary).

We use principal component analysis (PCA) to reduce the dimensions of the data. KNN is computationally expensive with large datasets, as distances must be computed and sorted. PCA is a statistical method that uses an orthogonal transformation to convert a set of observations of many correlated variables into a smaller set of linearly uncorrelated variables called principal components. It is used for dimensionality reduction by projecting each data point onto only the principal components. This way, the dimensionality of the data is reduced while still preserving as much variation of the data as possible. PCA attempts to keep only the most “important” parts of the data while discarding as much fluff as possible.

2.5 Random Forest

The random forest model consists of a large number of individual decision trees, where it altogether operates as an ensemble. Based on each prediction produced by the decision trees, the greatest value becomes the entire model’s prediction. In other sense, the random forest model operates as a set of individualized – not necessarily correlated – tasks that come together in the end. In order for the model to perform well, it must be implemented with specified parameters that prevent random guessing and each individual prediction must not have a lot of correlation to

another. As it has many decision trees by its core, the random forest model is unable to find patterns that allow it to extrapolate values that do not exist in the training set.

As explained in the definition of the random forest model, it consists of many decision trees. Through scikit-learn using Python, we are able to implement the model by considering the Gini impurity – an important measure that is used to construct multiple decision trees. Not only is it simple to implement, it works better with bigger distributions of values that may or may not be diverse. The following is the equation:

$\sum_{i=1}^{C} p_i(1 - p_i)$, where p is the frequency of the ith node and C is the value relating to unique nodes.

However, despite our expectation that multiple decision trees would be constructed well within the random forest model, the model unfortunately produced a result that fell below our expectations. One assumption attributing to its overall inaccuracy is due to the fact that, by definition, a decision tree follows a binary pattern in discerning one part of the data from the other. For this reason, even if each individual decision tree within the random forest model operated separately without any bias or with a significant amount of correlation between all trees, the one with the best prediction is still limited by the reason mentioned. It is also noted here that even if we tried to use a decision tree with CV, it would take forever to process through all of the data.

2.6 Support Vector Machine with PCA

The Support Vector Machine, also known as the SVM, is a way to classify data points by using a linear boundary. It tries to maximize the margin of the support vectors and is very useful along with PCA. Sometimes, to create a linear boundary for non-linear data, we utilize kernelized SVM:

$$K(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\sigma^2}\right)$$

where we project the data into a higher dimension to then be able to separate data points with a linear hyperplane.

In our case, covariance matrix is (performed by scikit-learn) calculated by

$$\frac{1}{n-1}\sum_{i=1}^{n}\left(x_{ij} - \bar{x}_j\right)\left(x_{ik} - \bar{x}_k\right)$$

.

Selecting eigenvectors of this covariance matrix means getting a reduced version (eigenimage) of the original faces, and we project each instance to such an eigenspace spanned by 50 (we used a rough estimate of 50) eigenvectors, hugely reduced the dimension from 2304 (48 by 48) to 50.

We can use the reconstruction step to project the 50-dimension dataset back to the original space to see the before and after effect of PCA. After we retrieve the 50-D dataset, we implement the SVM, which projects features to higher dimensions, using the default radial basis function kernel algorithm.

Radial Basis function in general form: with denominator replaced by (- n_features * var_x) instead in our case.

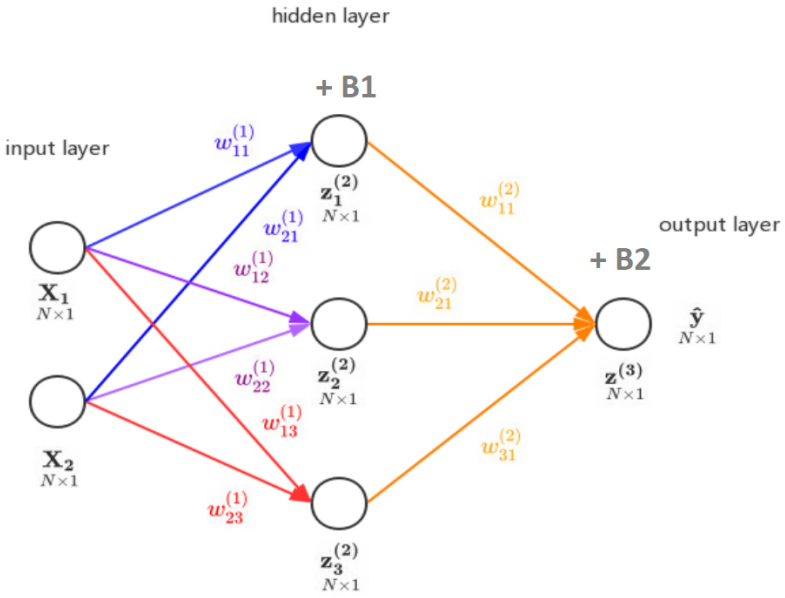
2.7 Artificial Neural Network

The Artificial Neural Network (ANN) is a deep learning algorithm that emerged and evolved from the idea of biological neural networks of human brains. An attempt to simulate the workings of the human brain culminated in the emergence of ANN. ANN works very similar to the biological neural networks.

The basic unit of computation in a neural network is the neuron , often called a node or unit. It receives input from some other nodes, or from an external source and computes an output. Each

input has an associated weight (w), which is assigned on the basis of its relative importance to other inputs. The node applies a function to the weighted sum of its inputs.

We define a neural network through multiple layers of nodes, each layer also encodes an activation function: a non-linear transformation to allow for more complexities in the relationship. For instance, we can represent the following neural network:



with the following set of equations:

$$\underset{N\times4}{\mathbf{X}}\ \underset{4\times3}{\mathbf{W}^{(1)}} + \underset{N\times1}{\mathbf{J}}\ \underset{1\times3}{\mathbf{B}^{(1)T}} = \underset{N\times3}{\mathbf{z}^{(2)}}$$

$$f(\underset{N\times3}{\mathbf{z}^{(2)}}) = \underset{N\times3}{\mathbf{a}^{(2)}}$$

$$\underset{N\times3}{\mathbf{a}^{(2)}}\ \underset{3\times3}{\mathbf{W}^{(2)}} + \underset{N\times1}{\mathbf{J}}\ \underset{1\times3}{\mathbf{B}^{(2)T}} = \underset{N\times3}{\mathbf{z}^{(3)}}$$

$$f(\underset{N\times3}{\mathbf{z}^{(3)}}) = \underset{N\times3}{\hat{\mathbf{y}}}$$

This step is referred to as “forward propagation”. By defining a loss function, we can then calculate the loss between the predicted values y hat and the true values y:

$$\mathcal{J} = \frac{1}{2}\sum_{n}(y_n - \hat{y}_n)^2$$

We can then perform "backward propagation", find the gradient of the loss function with respect to each weight and bias matrix, and perform gradient descent. For our example, the gradient are as follows:

$$\begin{aligned}
\frac{\partial J}{\partial W^{(2)}} &= \mathbf{a}^{(2)T}\delta^{(3)} \\
\frac{\partial J}{\partial B^{(2)}} &= \delta^{(3)T}1_{N\times 1} \\
\frac{\partial J}{\partial W^{(1)}} &= \mathbf{X}^{T}\delta^{(2)} \\
\frac{\partial J}{\partial B^{(1)}} &= \delta^{(2)T}1_{N\times 1}
\end{aligned},$$

where

$$\begin{aligned}
\delta^{(3)} &= -(\mathbf{y} - \hat{\mathbf{y}}) \odot f'\left(\mathbf{z}^{(3)}\right) \\
\delta^{(2)} &= \delta^{(3)}\left(\mathbf{W}^{(2)}\right)^{T} \odot f'\left(\mathbf{z}^{(2)}\right)
\end{aligned},$$

where $\odot$ represents element wise multiplication.

By iterating forward and backward propagation, the model learns the best parameters.

In our dataset, we feed the flattened normalized pixels into the neural network with 7 output nodes, where each node represents the probability of expressing one facial expression. We take the argmax of the 7 output to determine the most likely facial expression, which is our result.

2.8 Convolutional Neural Network

All the methods discussed from section 2.1 to 2.6 are more generalized classification models, but the convolutional neural network is specifically designed for image classification. Compared to other models, CNN looks at the relationship between pixels and determines if a certain feature

exists at a certain location through the use of convolution kernels. Suppose we have a picture of an “X”, binary and 7 by 7, represented by the matrix below:

$$X = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \in \mathbb{R}^{7 \times 7}$$

If we want to detect the feature dignonal, we can represent the feature with a 3 * 3 identity matrix, called convolution kernel:

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \in \mathbb{R}^{3 \times 3}$$

We can then detect the abundance of this feature in this matrix, call it A:

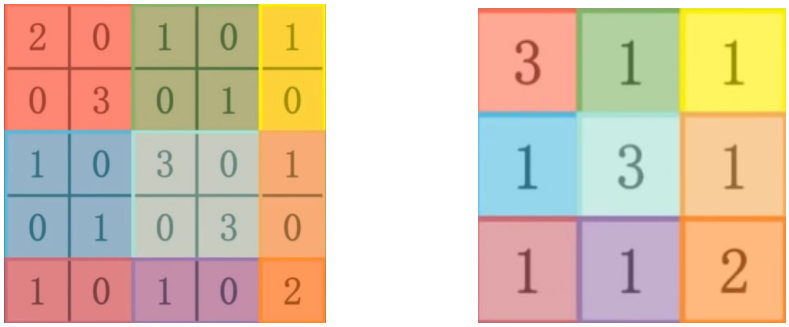
$$A \in \mathbb{R}^{5 \times 5}, \text{where } A_{i,\,j} = X[i : i + 2, j : j + 2] \cdot I, \text{where } i, j \in \{1, 2, 3, 4, 5\}$$

where $\cdot$ represents matrix dot product.

$$A = \begin{pmatrix} 2 & 0 & 1 & 0 & 1 \\ 0 & 3 & 0 & 1 & 0 \\ 1 & 0 & 3 & 0 & 1 \\ 0 & 1 & 0 & 3 & 0 \\ 1 & 0 & 1 & 0 & 2 \end{pmatrix}$$

The greater the value is, the more abundant this feature exists in the original picture. We can of course use more than one convolution kernel to detect various features; however, the resulting matrix (or tensor if we use more than one) might become too large, resulting in a slow training speed. Therefore, one way to solve this problem is through pooling. In practice, max pooling and average pooling is most often used. It maintains the original feature distribution while

significantly reducing the size of the matrix/tensor. The result of performing max pooling on our matrix A can is shown below:



As discussed in Section 1.2, normalization is a trick widely used in machine learning. We can therefore normalize the pooled matrix by sigmoid function and obtain:

$$B = \text{sigmoid}(\text{pooled}(\text{A})) = \begin{pmatrix} 0.95 & 0.73 & 0.73 \\ 0.73 & 0.95 & 0.73 \\ 0.73 & 0.73 & 0.88 \end{pmatrix}$$

This step is referred to as “activation”. Various other activation functions such as “ReLU” can also be adopted.

Through iteration of applying convolution kernels, pooling, and activation, we are able to capture nearly all features of an image, and through the process of flattening, we transform the matrix/tensor to a one-dimensional vector and feed it to the artificial neural network discussed in Section 2.7.

In actual implementation, the process of choosing the optimal convolution kernel is as well learned from the training set.

---

3. Conclusion

3.1 Results

We obtain a table of results of the eight models above:

Model	Training Accuracy	Validation Accuracy	Test Accuracy
Naive Bayes	0.232	0.204	0.209
Logistic Regression	0.489	0.364	0.365
K-Means + Logistic Reg	0.228	0.197	0.197
KNN with PCA + CV	0.663	0.646	0.638
Random Forest	0.401	0.287	0.281
SVM with PCA	0.631	0.459	0.469
Artificial NN	0.903	0.619	0.629
Convolutional NN	0.977	0.728	0.718

As expected, using CNN yields the highest accuracy on both validation and test sets, and it should therefore be used for future predictions.

3.2 Afterword

In this dataset, we have around 30,000 instances, which might not be sufficient for image classification training, given the similarity among several facial expressions. To solve this problem, we get the idea from the paper “Convolutional Neural Networks for Facial Expression Recognition” by Shima Alizadeh and Azar Fazel that we can flip all pictures so that we can double our dataset, as human faces are symmetrical. We do not have the time and computational power to retrain all our models on a dataset of 60,000 instances. In the future, we will explore this approach through using cloud computing services such as Google Cloud or Amazon Web Services.

Between the classification methods that were covered in this entire project, one hope was to see little to no difference in the inaccuracies of recognizing emotions, despite contrasting efficiencies. From this, a future possible implementation is being able to recognize human

expressions from various angles and even with color. Furthermore, real-time recognition could be another application to look forward to, as such implementation in many areas of marketing, health, education, and public safety (and more) would allow constant feedback between the client and the service provided. Hence, from this project, it would be a step in various possibilities of further practical application that can be of service to many different fields for convenience and necessity.

---

4. Cited Works

"1.1. Linear Models." *Scikit-Learn*, scikit-learn.org/stable/modules/linear_model.html#logistic-regression.

Alizadeh, Shima, and Azar Fazel. "Convolutional Neural Networks for Facial Expression Recognition". https://arxiv.org/abs/1704.06756.

Daumé, Hal, III. "A Course in Machine Learning".

Geron. "Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow".

"Introduction to Convolutional Neural Networks CNNs". https://aigents.co/data-science-blog/publication/introduction-to-convolutional-neural-networks-cnns.

Singhal, Aaditya. "Facial Expression Dataset." Kaggle, 26 Dec. 2020, https://www.kaggle.com/datasets/aadityasinghal/facial-expression-dataset.

Yiu, Tony. "Understanding Random Forest - Towards Data Science." *Medium*, 10 Dec. 2021, towardsdatascience.com/understanding-random-forest-58381e0602d2.

FacialEmotionRecognition

June 7, 2022

[1]:
```
# Basics
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from pandas.core.common import random_state

# Sklearn
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.cluster import KMeans
from sklearn.pipeline import Pipeline
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay


# Tensorflow
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import tensorflow.keras.layers as layers
from tensorflow.keras.utils import to_categorical


# Graphing Style
%matplotlib inline
```

[2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

1 Data Cleaning and Exploratory Data Analysis

[3]:
```
face = pd.read_csv("/content/drive/MyDrive/Colab Notebooks/face.csv")
```

[4]:
```
face.shape
```

[4]:
```
(28709, 2)
```

[5]:
```
face["pixels"] = face.pixels.apply(lambda x: np.array(tuple(map(int, x.
↪split()))))
```

[6]:
```
face.head()
```

[6]:

	emotion	pixels
0	0	[70, 80, 82, 72, 58, 58, 60, 63, 54, 58, 60, 4…
1	0	[151, 150, 147, 155, 148, 133, 111, 140, 170, …
2	2	[231, 212, 156, 164, 174, 138, 161, 173, 182, …
3	4	[24, 32, 36, 30, 32, 23, 19, 20, 30, 41, 21, 2…
4	6	[4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 15, 23…

[7]:
```
emo_di = {0: "Angry", 1:"Disgust", 2: "Fear", 3: "Happy", 4: "Sad", 5:␣
↪"Surprise", 6: "Neutral"}
```

[8]:
```
plt.figure(figsize = (20,20))
start_index = 0
for i in range(7):
    plt.subplot(1,7,i+1)
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(face[face.emotion == i].pixels.iloc[20].reshape(48,48),␣
↪cmap="gray")
    plt.xlabel("{} = {}".format(i, emo_di[i]))
```

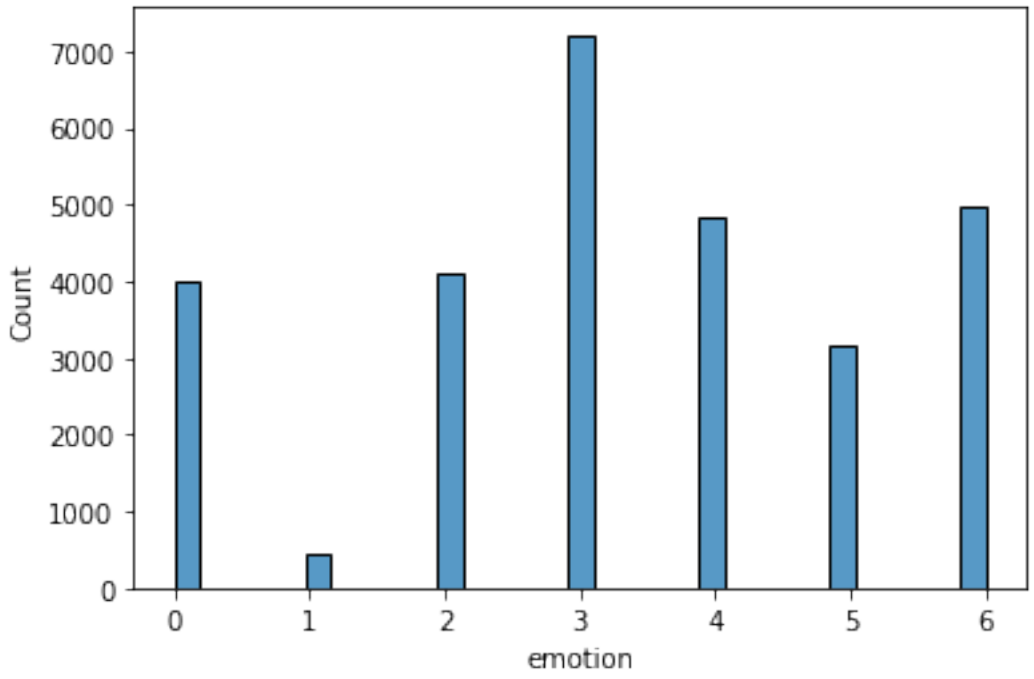
[9]:
```
face.emotion.value_counts()
```

[9]:
```
3    7215
6    4965
4    4830
```

```
2    4097
0    3995
5    3171
1     436
Name: emotion, dtype: int64
```

[10]:
```
sns.histplot(face.emotion)
```

[10]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f43872e6ed0>
```



[11]:
```
train, test = train_test_split(face, train_size=0.6, random_state=1)
val, test = train_test_split(test, test_size=0.5, random_state=1)

# Oversample and subsample
faces_balanced = None
for i in range(7):
    if faces_balanced is None:
        faces_balanced = train[train.emotion == i].sample(n=3000, replace=True,␣
 ↪random_state=100)
    else:
        faces_balanced = pd.concat([faces_balanced, train[train.emotion == i].
 ↪sample(n=3000, replace=True, random_state=100)], sort = False)
faces_balanced = faces_balanced.sample(frac=1, random_state=49)
```

```
train_x = np.concatenate(np.asarray(faces_balanced["pixels"])).reshape(-1, 48 *␣
↪48)
val_x = np.concatenate(np.asarray(val["pixels"])).reshape(-1, 48 * 48)
test_x = np.concatenate(np.asarray(test["pixels"])).reshape(-1, 48 * 48)
train_y = faces_balanced.emotion
val_y = val.emotion
test_y = test.emotion

x_mean = np.mean(train_x)
x_std = np.std(train_x) + 1e-10
train_x = (train_x - x_mean) / x_std
val_x = (val_x - x_mean) / x_std
test_x = (test_x - x_mean) / x_std
```

[12]:
```
train_y.value_counts()
```

[12]:
```
4    3000
5    3000
2    3000
0    3000
1    3000
3    3000
6    3000
Name: emotion, dtype: int64
```

[13]:
```
val_y.value_counts()
```

[13]:
```
3    1420
6    1035
4     963
0     811
2     796
5     625
1      92
Name: emotion, dtype: int64
```

[14]:
```
test_y.value_counts()
```

[14]:
```
3    1494
6     976
4     946
2     833
0     792
5     627
1      74
Name: emotion, dtype: int64
```

2 Modeling

2.1 Naive Bayes

[15]:
```
nb = GaussianNB()
nb.fit(train_x, train_y)
```

[15]:
```
GaussianNB()
```

[16]:
```
nb.score(train_x, train_y)
```

[16]:
```
0.23157142857142857
```

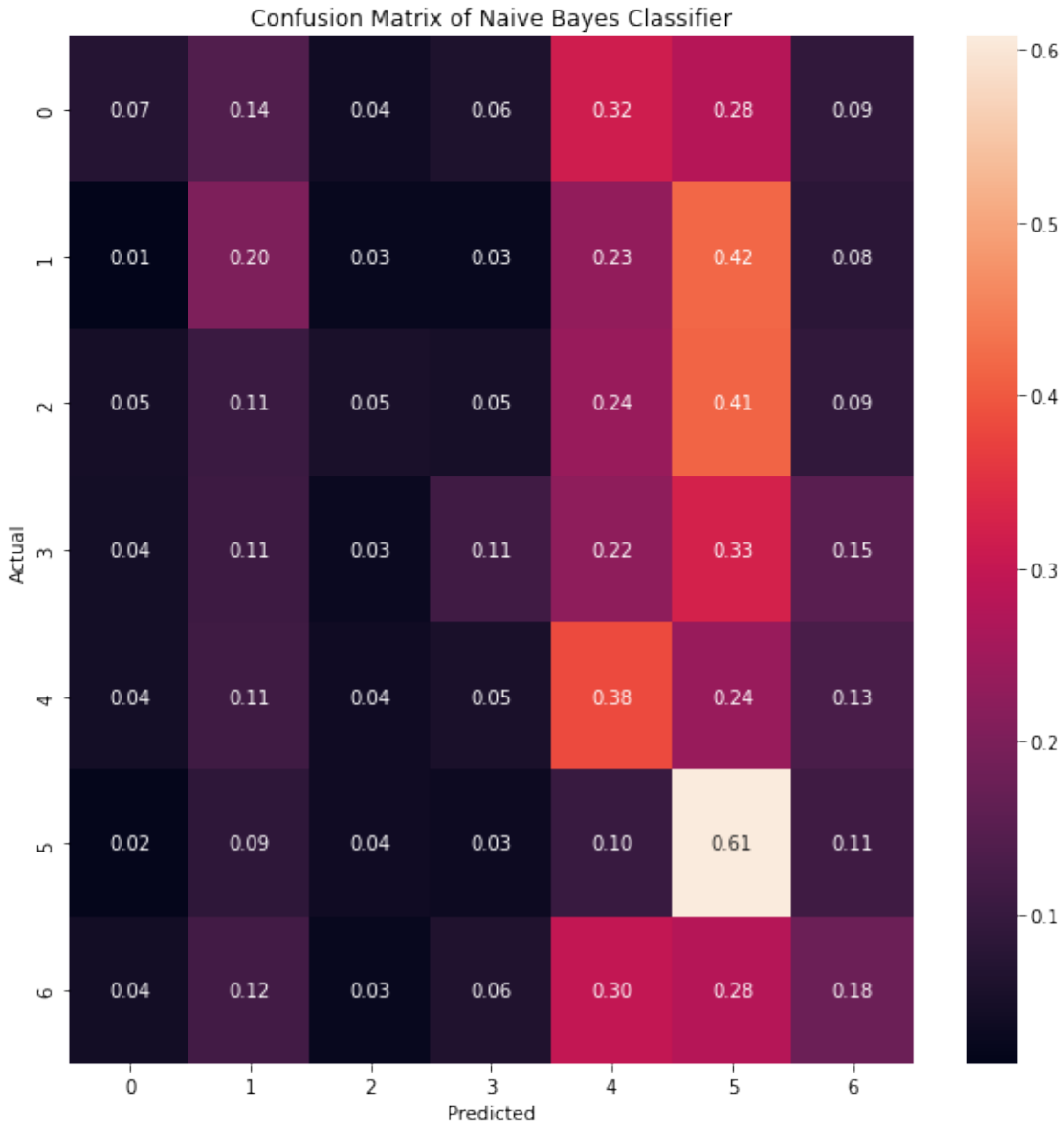
[17]:
```
nb.score(val_x, val_y)
```

[17]:
```
0.20358760013932428
```

[18]:
```
nb.score(test_x, test_y)
```

[18]:
```
0.20968303726924417
```

[19]:
```
cm = confusion_matrix(test_y, nb.predict(test_x))
cmn = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
fig, ax = plt.subplots(figsize=(10,10))
sns.heatmap(cmn, annot=True, fmt='.2f', xticklabels=nb.classes_, yticklabels=nb.
 ↪classes_)
plt.title('Confusion Matrix of Naive Bayes Classifier')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show(block=False)
```



2.2 Logistic Regression

[20]:
```
lr = LogisticRegression(penalty='l2', tol=0.1, solver='saga')
lr.fit(train_x, train_y)
```

[20]:
```
LogisticRegression(solver='saga', tol=0.1)
```

[21]:
```
lr.score(train_x, train_y)
```

[21]:
```
0.4888571428571429
```

[22]:
```
lr.score(val_x, val_y)
```

[22]:
```
0.36363636363636365
```

[23]:
```
lr.score(test_x, test_y)
```

[23]:
```
0.36450714036920934
```

[24]:
```
cm = confusion_matrix(test_y, lr.predict(test_x))
cmn = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
fig, ax = plt.subplots(figsize=(10,10))
sns.heatmap(cmn, annot=True, fmt='.2f', xticklabels=nb.classes_, yticklabels=nb.
 ↪classes_)
plt.title('Confusion Matrix of Logistic Regression')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show(block=False)
```

2.3 K-Means + Logistic Regression

[25]:
```
pl = Pipeline([
    ("kmeans", KMeans(n_clusters=35)),
    ("logisreg", LogisticRegression(penalty='none', tol=0.1, solver='saga'))
])
```

[26]:
```
pl.fit(train_x, train_y)
```

[26]:
```
Pipeline(steps=[('kmeans', KMeans(n_clusters=35)),
                ('logisreg',
                 LogisticRegression(penalty='none', solver='saga', tol=0.1))])
```

[27]:
```
pl.score(train_x, train_y)
```

[27]:
```
0.2277142857142857
```

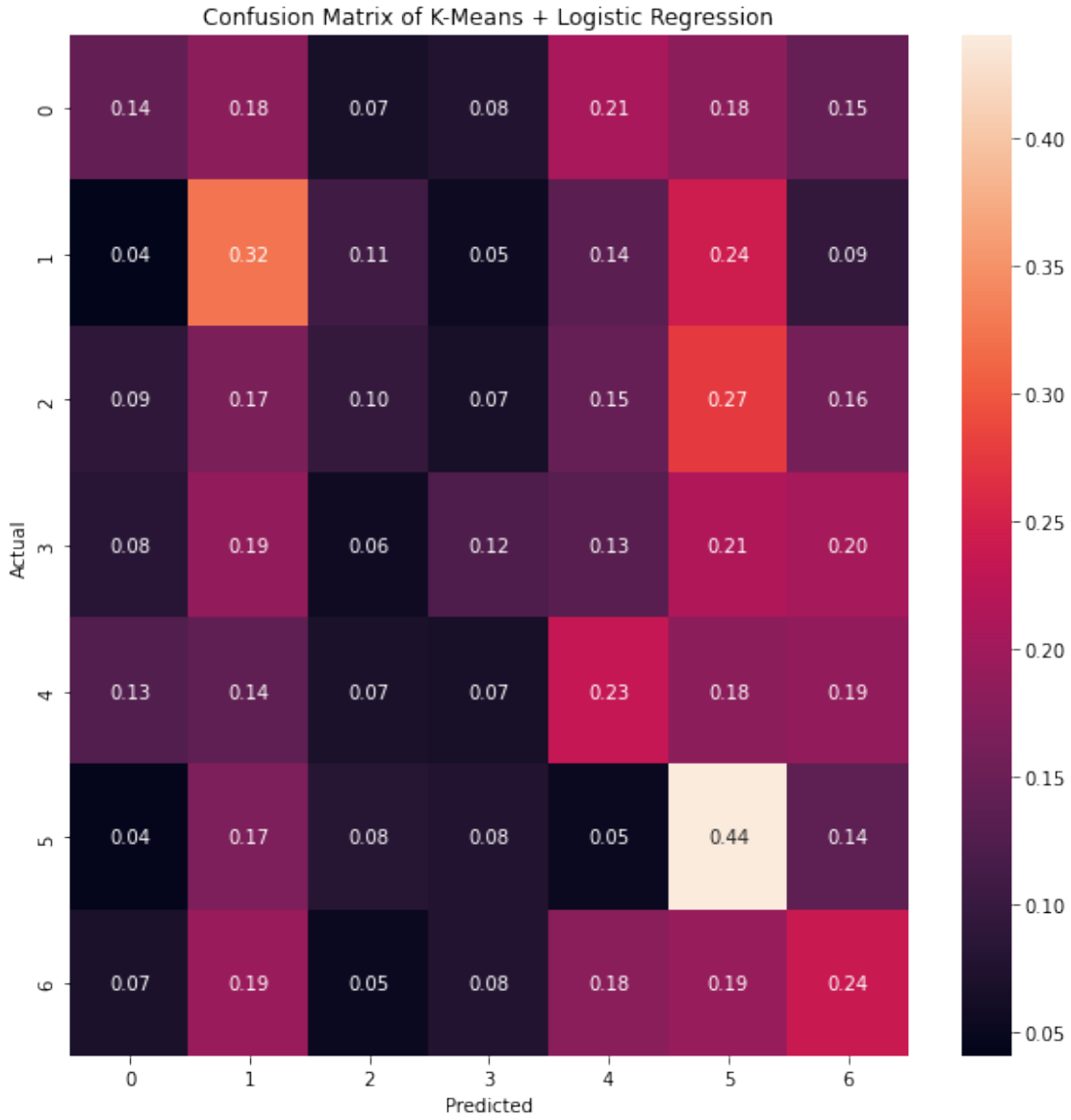
[28]:
```
pl.score(val_x, val_y)
```

[28]:
```
0.19696969696969696
```

[29]:
```
pl.score(test_x, test_y)
```

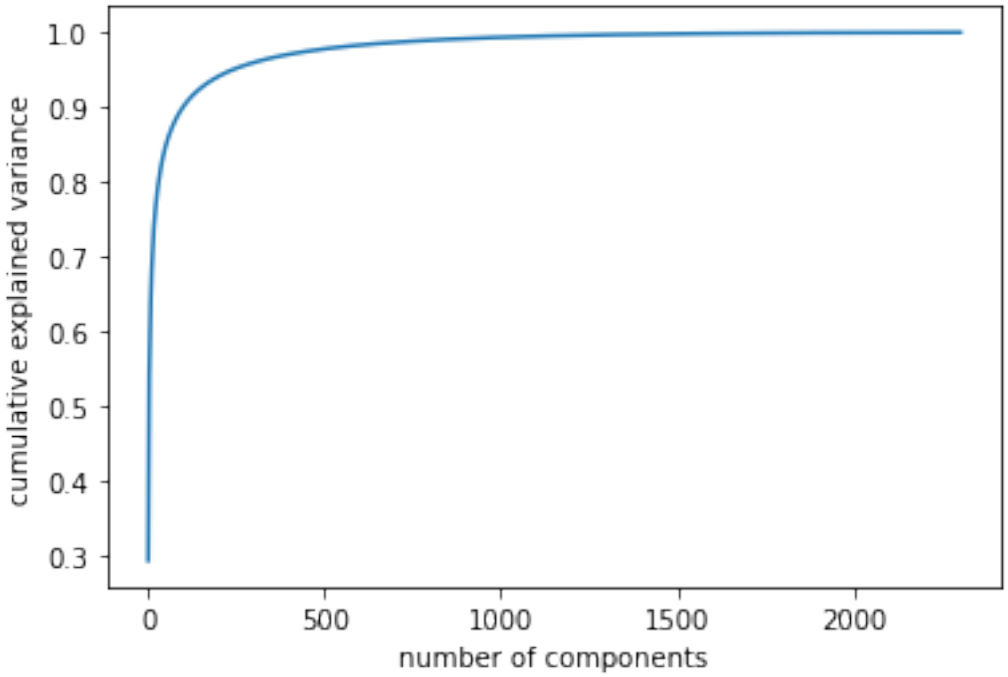
[29]:
```
0.1966213862765587
```

[30]:
```
cm = confusion_matrix(test_y, pl.predict(test_x))
cmn = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
fig, ax = plt.subplots(figsize=(10,10))
sns.heatmap(cmn, annot=True, fmt='.2f', xticklabels=nb.classes_, yticklabels=nb.
 ↪classes_)
plt.title('Confusion Matrix of K-Means + Logistic Regression')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show(block=False)
```



2.4 KNN with PCA + CV

[31]:
```
pca = PCA().fit(train_x)
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
```



[32]:
```
pl1 = Pipeline([
    ("PCA", PCA(n_components=100)),
    ("knn", KNeighborsClassifier())
])
params = {"knn__n_neighbors": [1, 3, 5, 7, 12, 15, 20]}
grids = GridSearchCV(pl1, params, cv=5)
grids.fit(train_x, train_y)
```

[32]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('PCA', PCA(n_components=100)),
                                       ('knn', KNeighborsClassifier())]),
             param_grid={'knn__n_neighbors': [1, 3, 5, 7, 12, 15, 20]})
```

[33]:
```
grids.best_params_
```

[33]:
```
{'knn__n_neighbors': 1}
```

[34]:
```
grids.best_score_
```

[34]:
```
0.6629047619047619
```

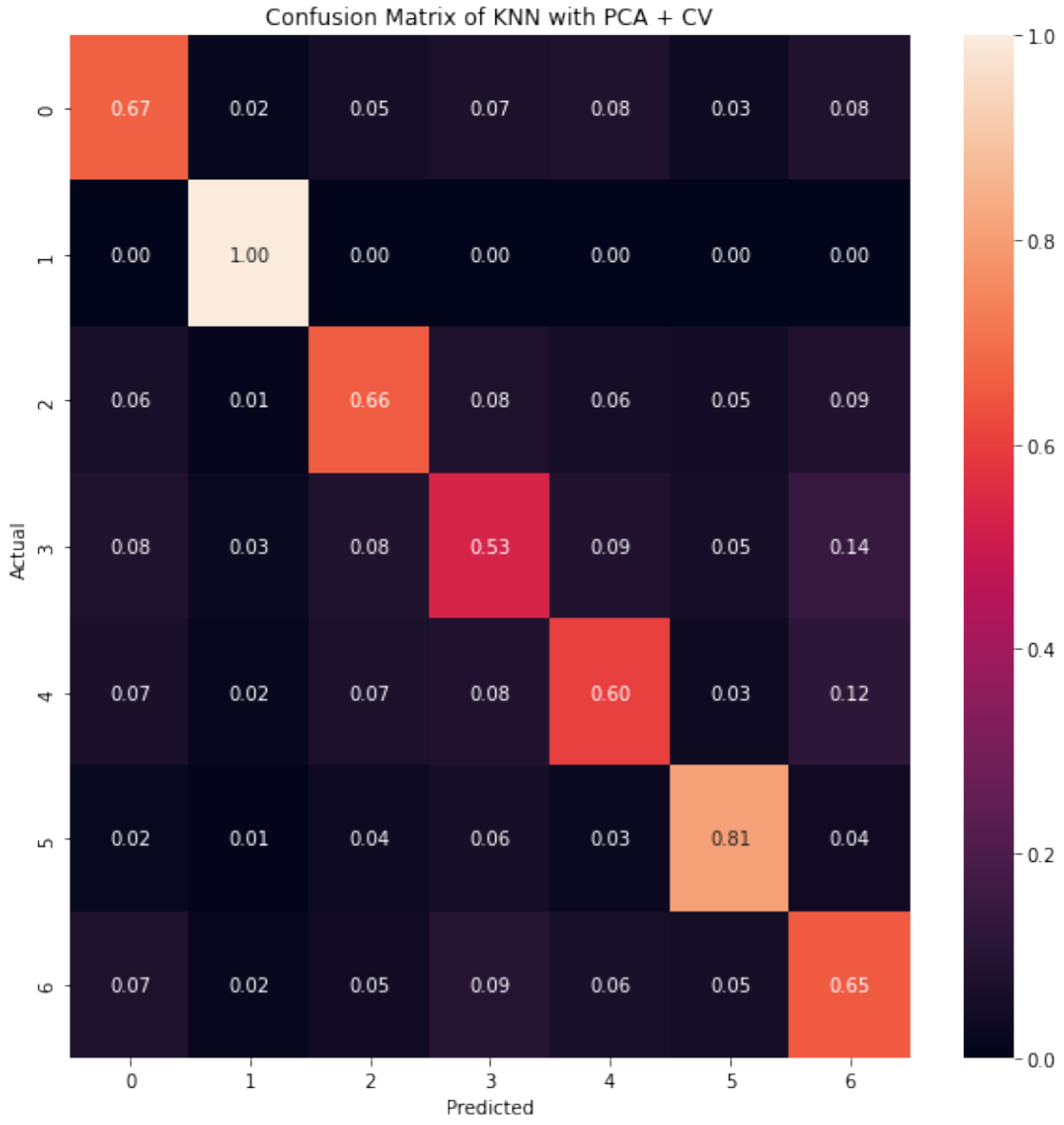
[35]:
```
grids.best_estimator_.score(val_x, val_y)
```

[35]:
```
0.6461163357715082
```

[36]:
```
grids.best_estimator_.score(test_x, test_y)
```

[36]:
```
0.6379310344827587
```

[37]:
```
cm = confusion_matrix(test_y, grids.best_estimator_.predict(test_x))
cmn = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
fig, ax = plt.subplots(figsize=(10,10))
sns.heatmap(cmn, annot=True, fmt='.2f', xticklabels=nb.classes_, yticklabels=nb.
 ↪classes_)
plt.title('Confusion Matrix of KNN with PCA + CV')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show(block=False)
```



2.5 Random Forest

[38]:
```
rfc = RandomForestClassifier(max_depth=5)
rfc.fit(train_x, train_y)
```

[38]:
```
RandomForestClassifier(max_depth=5)
```

[39]:
```
rfc.score(train_x, train_y)
```

[39]:
```
0.4005714285714286
```

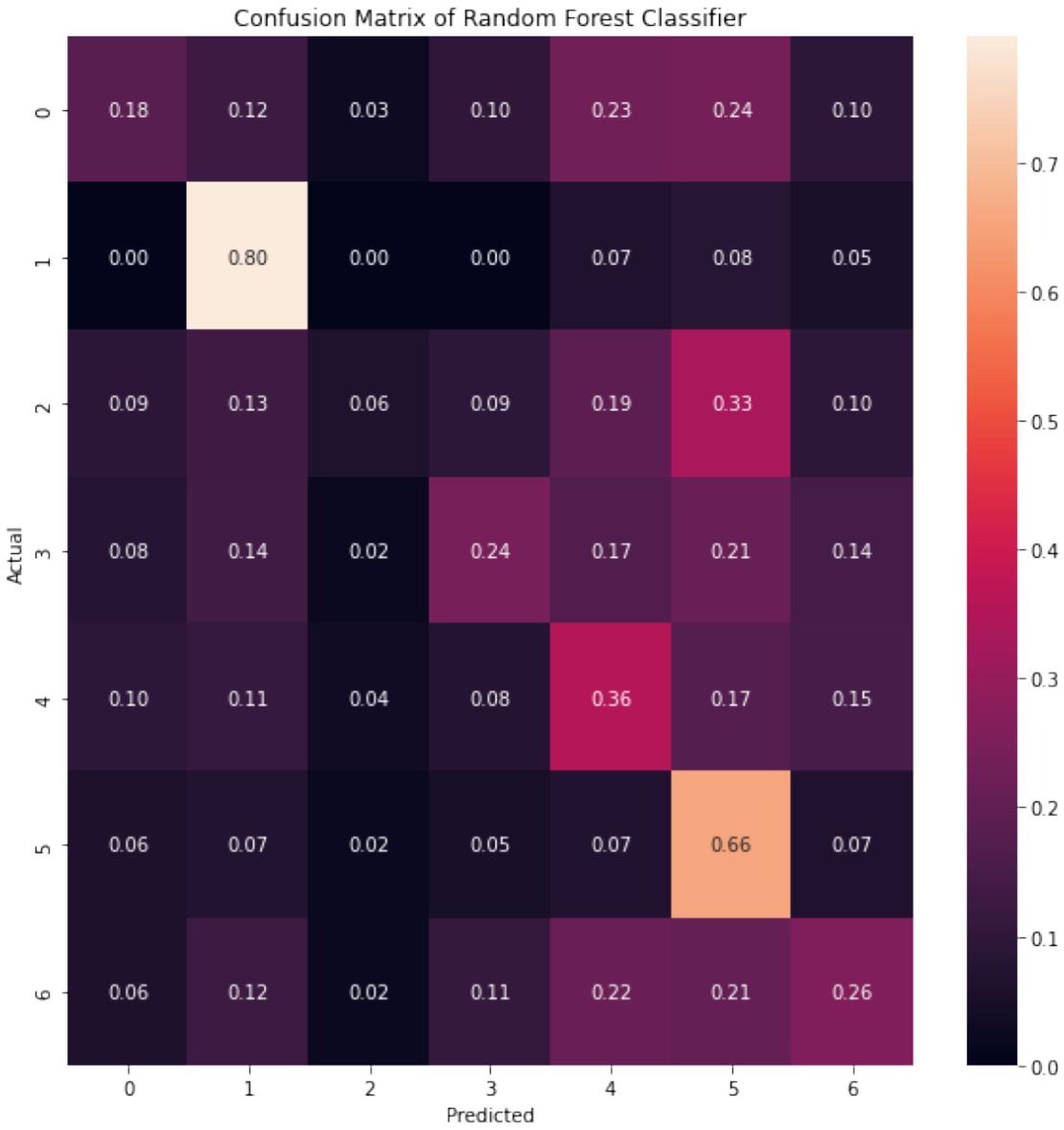
[40]:
```
rfc.score(val_x, val_y)
```

[40]:
```
0.2871821664925113
```

[41]:
```
rfc.score(test_x, test_y)
```

[41]:
```
0.28126088470916055
```

[42]:
```
cm = confusion_matrix(test_y, rfc.predict(test_x))
cmn = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
fig, ax = plt.subplots(figsize=(10,10))
sns.heatmap(cmn, annot=True, fmt='.2f', xticklabels=nb.classes_, yticklabels=nb.
 ↪classes_)
plt.title('Confusion Matrix of Random Forest Classifier')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show(block=False)
```



2.6 Support Vector Machine with PCA

[43]:
```
pl2 = Pipeline([
                ("PCA", PCA(n_components=50)),
                ("svm", SVC())
])
pl2.fit(train_x, train_y)
```

[43]:
```
Pipeline(steps=[('PCA', PCA(n_components=50)), ('svm', SVC())])
```

[44]:
```
pl2.score(train_x, train_y)
```

[44]:
```
0.6312857142857143
```

[45]:
```
pl2.score(val_x, val_y)
```

[45]:
```
0.45942180424939044
```

[46]:
```
pl2.score(test_x, test_y)
```

[46]:
```
0.46900034831069315
```

[47]:
```
cm = confusion_matrix(test_y, pl2.predict(test_x))
cmn = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
fig, ax = plt.subplots(figsize=(10,10))
sns.heatmap(cmn, annot=True, fmt='.2f', xticklabels=nb.classes_, yticklabels=nb.
 ↪classes_)
plt.title('Confusion Matrix of SVM with PCA')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show(block=False)
```

2.7 Artificial Neural Network

[99]:
```
train_y_cat = to_categorical(train_y)
val_y_cat = to_categorical(val_y)
test_y_cat = to_categorical(test_y)
```

[100]:
```
model = Sequential()
model.add(Dense(256, activation = 'relu', input_shape = (48 * 48,)))
model.add(Dense(256, activation = 'relu'))
model.add(Dense(256, activation = 'relu')),
model.add(Dense(256, activation = 'relu')),
```

```
model.add(Dense(7, activation = 'softmax'))

model.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics =␣
 ↪['accuracy'])
model.summary()
```

```
Model: "sequential_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_22 (Dense)            (None, 256)               590080

 dense_23 (Dense)            (None, 256)               65792

 dense_24 (Dense)            (None, 256)               65792

 dense_25 (Dense)            (None, 256)               65792

 dense_26 (Dense)            (None, 7)                 1799

=================================================================
Total params: 789,255
Trainable params: 789,255
Non-trainable params: 0
_________________________________________________________________
```

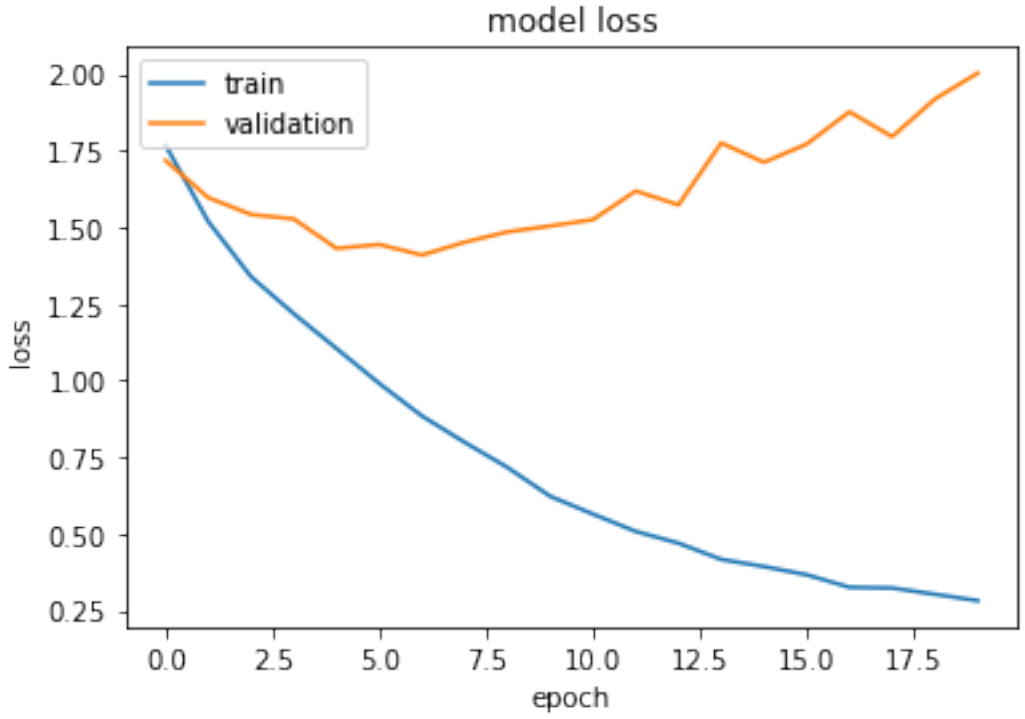
[101]:
```
history = model.fit(train_x, train_y_cat, epochs = 20, validation_data =␣
 ↪(val_x, val_y_cat))
```

```
Epoch 1/20
657/657 [==============================] - 4s 5ms/step - loss: 1.7671 -
accuracy: 0.3019 - val_loss: 1.7173 - val_accuracy: 0.3022
Epoch 2/20
657/657 [==============================] - 3s 5ms/step - loss: 1.5181 -
accuracy: 0.4083 - val_loss: 1.5960 - val_accuracy: 0.3661
Epoch 3/20
657/657 [==============================] - 3s 5ms/step - loss: 1.3385 -
accuracy: 0.4810 - val_loss: 1.5412 - val_accuracy: 0.3931
Epoch 4/20
657/657 [==============================] - 3s 5ms/step - loss: 1.2176 -
accuracy: 0.5320 - val_loss: 1.5267 - val_accuracy: 0.4072
Epoch 5/20
657/657 [==============================] - 3s 4ms/step - loss: 1.1043 -
accuracy: 0.5774 - val_loss: 1.4307 - val_accuracy: 0.4702
Epoch 6/20
657/657 [==============================] - 3s 5ms/step - loss: 0.9904 -
accuracy: 0.6308 - val_loss: 1.4433 - val_accuracy: 0.4805
```

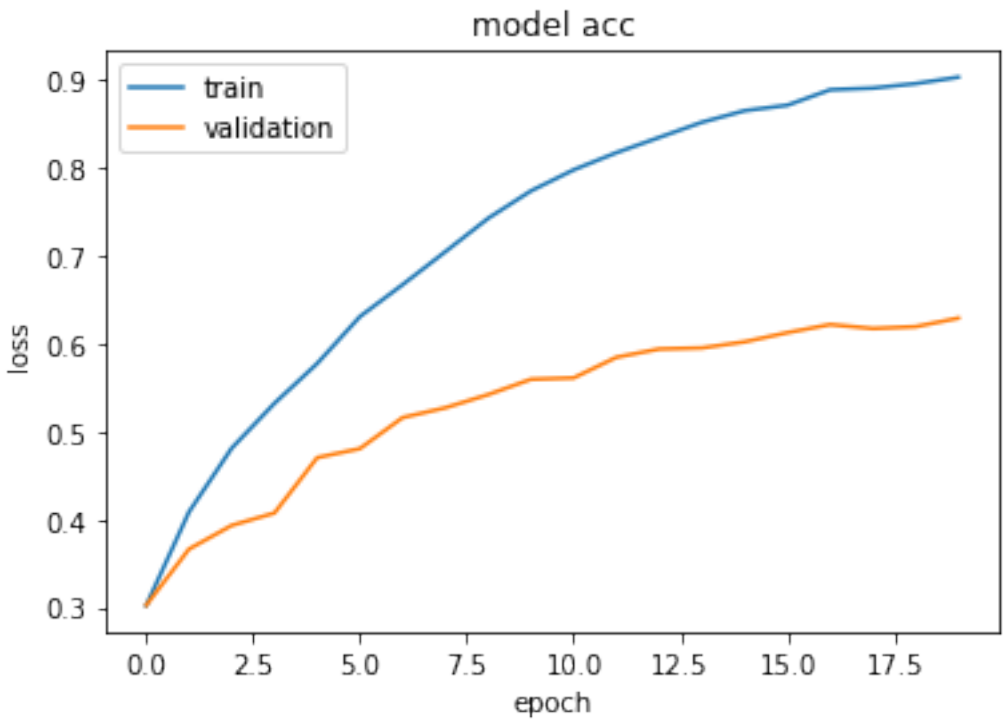
```
Epoch 7/20
657/657 [==============================] - 3s 5ms/step - loss: 0.8841 -
accuracy: 0.6672 - val_loss: 1.4089 - val_accuracy: 0.5158
Epoch 8/20
657/657 [==============================] - 3s 5ms/step - loss: 0.7984 -
accuracy: 0.7047 - val_loss: 1.4509 - val_accuracy: 0.5270
Epoch 9/20
657/657 [==============================] - 3s 5ms/step - loss: 0.7165 -
accuracy: 0.7427 - val_loss: 1.4845 - val_accuracy: 0.5420
Epoch 10/20
657/657 [==============================] - 3s 5ms/step - loss: 0.6224 -
accuracy: 0.7739 - val_loss: 1.5038 - val_accuracy: 0.5594
Epoch 11/20
657/657 [==============================] - 3s 5ms/step - loss: 0.5644 -
accuracy: 0.7979 - val_loss: 1.5244 - val_accuracy: 0.5610
Epoch 12/20
657/657 [==============================] - 3s 5ms/step - loss: 0.5080 -
accuracy: 0.8174 - val_loss: 1.6178 - val_accuracy: 0.5845
Epoch 13/20
657/657 [==============================] - 3s 5ms/step - loss: 0.4697 -
accuracy: 0.8347 - val_loss: 1.5725 - val_accuracy: 0.5939
Epoch 14/20
657/657 [==============================] - 3s 5ms/step - loss: 0.4167 -
accuracy: 0.8520 - val_loss: 1.7744 - val_accuracy: 0.5951
Epoch 15/20
657/657 [==============================] - 3s 5ms/step - loss: 0.3934 -
accuracy: 0.8652 - val_loss: 1.7112 - val_accuracy: 0.6021
Epoch 16/20
657/657 [==============================] - 3s 5ms/step - loss: 0.3665 -
accuracy: 0.8714 - val_loss: 1.7708 - val_accuracy: 0.6125
Epoch 17/20
657/657 [==============================] - 3s 5ms/step - loss: 0.3258 -
accuracy: 0.8890 - val_loss: 1.8759 - val_accuracy: 0.6217
Epoch 18/20
657/657 [==============================] - 3s 5ms/step - loss: 0.3235 -
accuracy: 0.8909 - val_loss: 1.7949 - val_accuracy: 0.6172
Epoch 19/20
657/657 [==============================] - 3s 5ms/step - loss: 0.3030 -
accuracy: 0.8960 - val_loss: 1.9177 - val_accuracy: 0.6193
Epoch 20/20
657/657 [==============================] - 3s 5ms/step - loss: 0.2823 -
accuracy: 0.9033 - val_loss: 2.0023 - val_accuracy: 0.6290
```

[102]:
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
```

```
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```



[103]:
```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model acc')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```

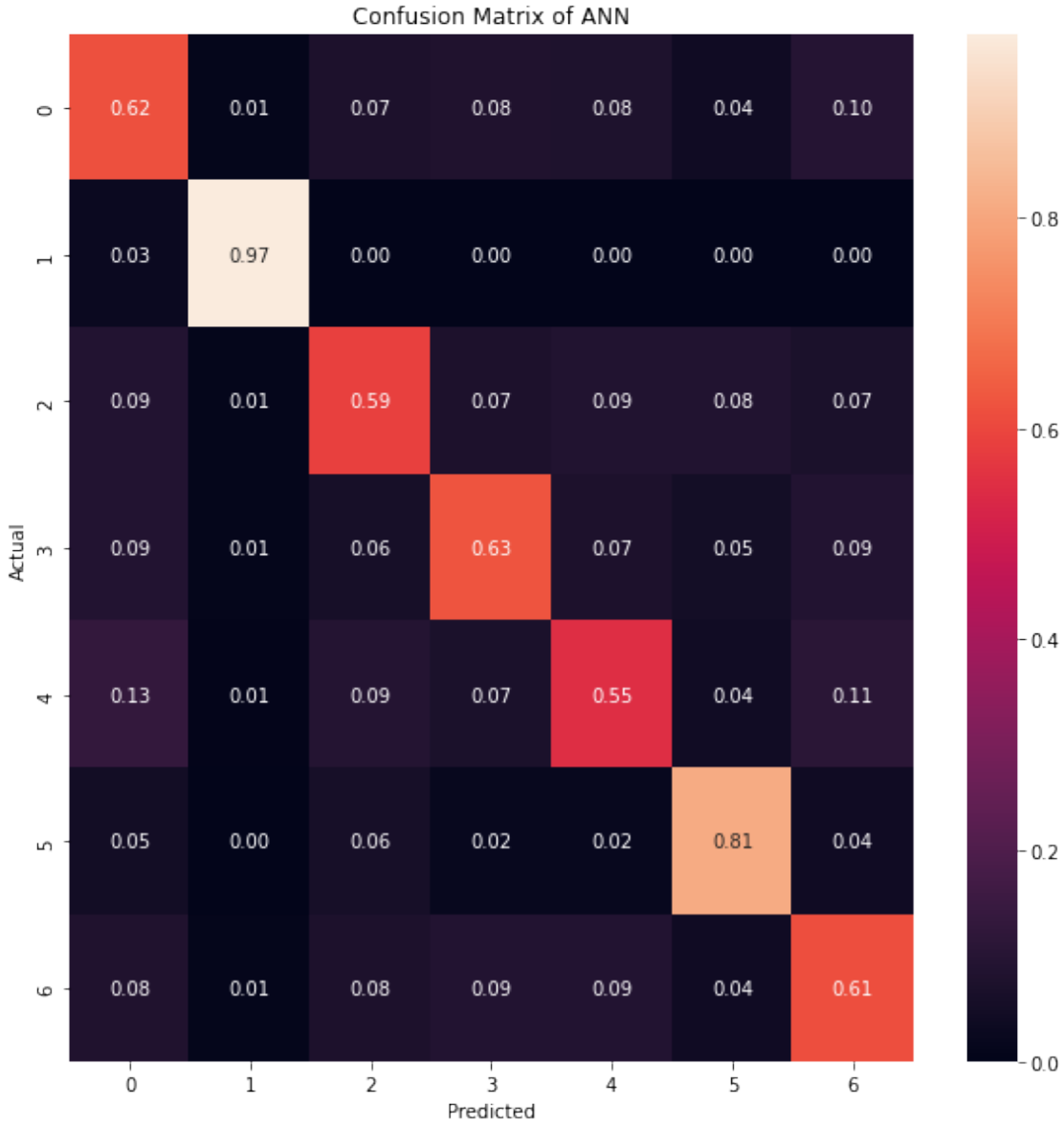


[104]:
```
model.evaluate(test_x, test_y_cat)
```

```
180/180 [==============================] - 0s 2ms/step - loss: 1.9723 -
accuracy: 0.6287
```

[104]:
```
[1.972302794456482, 0.6287007927894592]
```

[105]:
```
cm = confusion_matrix(test_y, np.apply_along_axis(np.argmax, 1, model.
 ↪predict(test_x)))
cmn = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
fig, ax = plt.subplots(figsize=(10,10))
sns.heatmap(cmn, annot=True, fmt='.2f', xticklabels=nb.classes_, yticklabels=nb.
 ↪classes_)
plt.title('Confusion Matrix of ANN')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show(block=False)
```



2.8 Convolutional Neural Network

[86]:
```
train_x_cnn = train_x.reshape(-1, 48, 48, 1)
val_x_cnn = val_x.reshape(-1, 48, 48, 1)
test_x_cnn = test_x.reshape(-1, 48, 48, 1)
```

[2]:
```
model3 = Sequential([
  layers.Conv2D(32, (3, 3), activation='relu', input_shape=(48, 48, 1)),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
```

```
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(7)
])
```

[4]:
```
model3.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 46, 46, 32)        320

 conv2d_1 (Conv2D)           (None, 46, 46, 16)        4624

 max_pooling2d (MaxPooling2D  (None, 23, 23, 16)       0
 )

 conv2d_2 (Conv2D)           (None, 23, 23, 32)        4640

 max_pooling2d_1 (MaxPooling  (None, 11, 11, 32)       0
 2D)

 conv2d_3 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_2 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 flatten (Flatten)           (None, 1600)              0

 dense (Dense)               (None, 128)               204928

 dense_1 (Dense)             (None, 7)                 903

=================================================================
Total params: 233,911
Trainable params: 233,911
Non-trainable params: 0
_________________________________________________________________
```

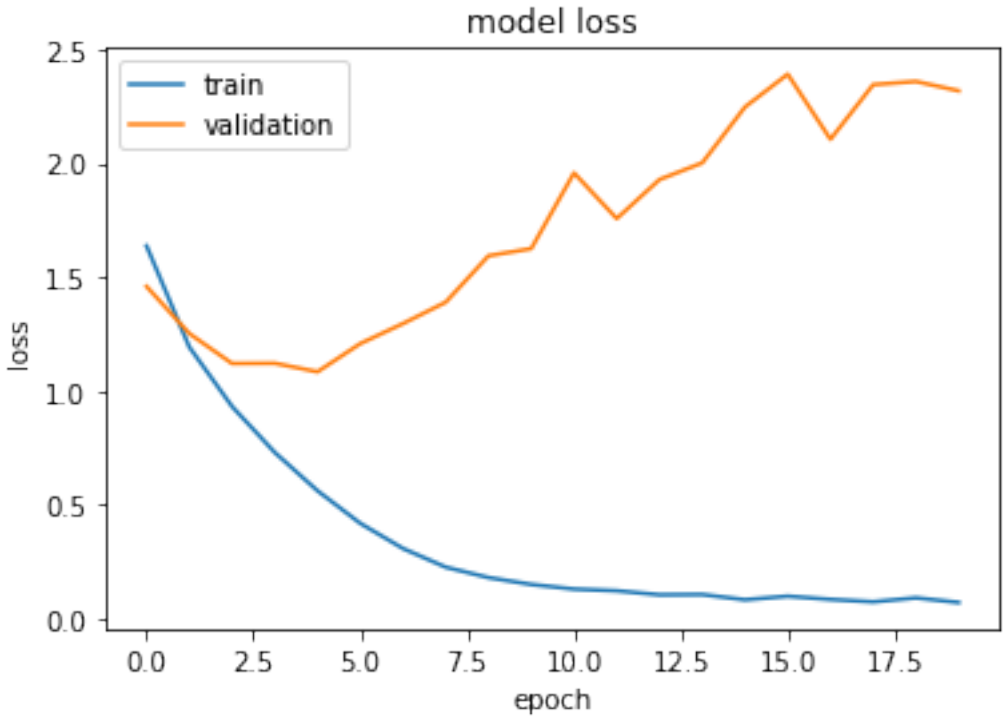
[3]:
```
model3.compile(optimizer='adam',
               loss=tf.keras.losses.
 ↪SparseCategoricalCrossentropy(from_logits=True),
               metrics=['accuracy'])
```

[95]:
```
history3 = model3.fit(train_x_cnn, train_y, epochs = 20, validation_data =␣
↪(val_x_cnn, val_y))
```

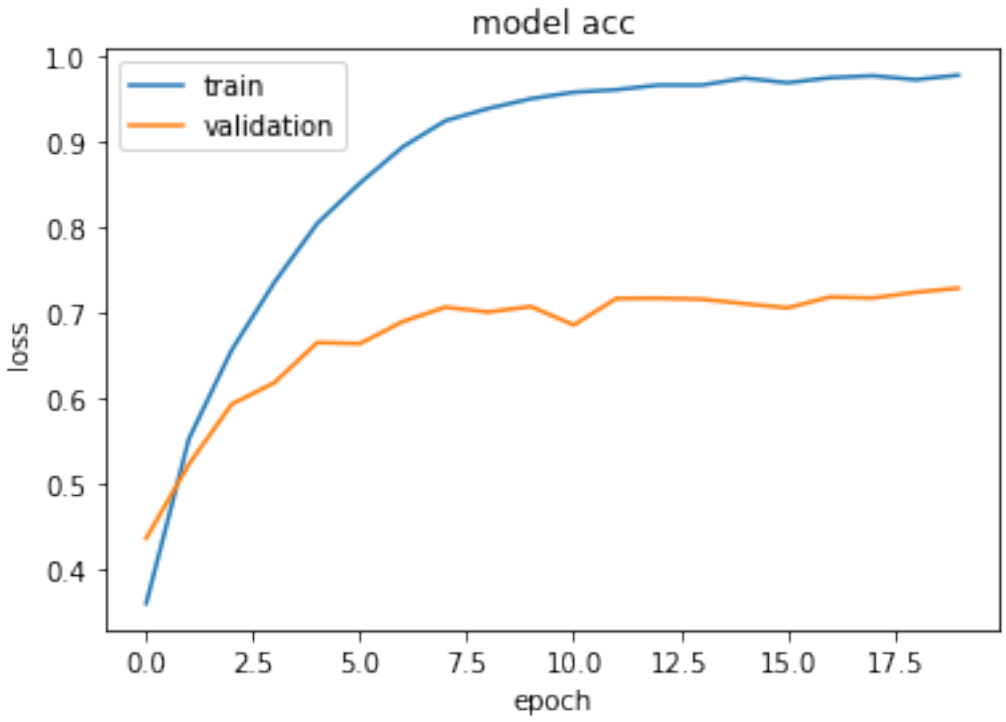
```
Epoch 1/20
657/657 [==============================] - 16s 24ms/step - loss: 1.6371 -
accuracy: 0.3590 - val_loss: 1.4592 - val_accuracy: 0.4352
Epoch 2/20
657/657 [==============================] - 16s 24ms/step - loss: 1.1933 -
accuracy: 0.5517 - val_loss: 1.2526 - val_accuracy: 0.5219
Epoch 3/20
657/657 [==============================] - 16s 24ms/step - loss: 0.9327 -
accuracy: 0.6554 - val_loss: 1.1211 - val_accuracy: 0.5920
Epoch 4/20
657/657 [==============================] - 16s 24ms/step - loss: 0.7309 -
accuracy: 0.7345 - val_loss: 1.1220 - val_accuracy: 0.6172
Epoch 5/20
657/657 [==============================] - 16s 24ms/step - loss: 0.5627 -
accuracy: 0.8034 - val_loss: 1.0846 - val_accuracy: 0.6641
Epoch 6/20
657/657 [==============================] - 16s 24ms/step - loss: 0.4200 -
accuracy: 0.8506 - val_loss: 1.2082 - val_accuracy: 0.6628
Epoch 7/20
657/657 [==============================] - 16s 24ms/step - loss: 0.3095 -
accuracy: 0.8928 - val_loss: 1.2962 - val_accuracy: 0.6886
Epoch 8/20
657/657 [==============================] - 16s 24ms/step - loss: 0.2275 -
accuracy: 0.9235 - val_loss: 1.3900 - val_accuracy: 0.7055
Epoch 9/20
657/657 [==============================] - 16s 24ms/step - loss: 0.1819 -
accuracy: 0.9378 - val_loss: 1.5935 - val_accuracy: 0.6998
Epoch 10/20
657/657 [==============================] - 16s 24ms/step - loss: 0.1518 -
accuracy: 0.9492 - val_loss: 1.6249 - val_accuracy: 0.7062
Epoch 11/20
657/657 [==============================] - 16s 24ms/step - loss: 0.1315 -
accuracy: 0.9568 - val_loss: 1.9570 - val_accuracy: 0.6846
Epoch 12/20
657/657 [==============================] - 16s 24ms/step - loss: 0.1240 -
accuracy: 0.9598 - val_loss: 1.7568 - val_accuracy: 0.7156
Epoch 13/20
657/657 [==============================] - 16s 24ms/step - loss: 0.1064 -
accuracy: 0.9652 - val_loss: 1.9280 - val_accuracy: 0.7160
Epoch 14/20
657/657 [==============================] - 16s 24ms/step - loss: 0.1075 -
accuracy: 0.9651 - val_loss: 2.0010 - val_accuracy: 0.7147
Epoch 15/20
657/657 [==============================] - 16s 24ms/step - loss: 0.0847 -
```

```
accuracy: 0.9733 - val_loss: 2.2462 - val_accuracy: 0.7095
Epoch 16/20
657/657 [==============================] - 16s 24ms/step - loss: 0.1002 -
accuracy: 0.9679 - val_loss: 2.3902 - val_accuracy: 0.7046
Epoch 17/20
657/657 [==============================] - 16s 24ms/step - loss: 0.0861 -
accuracy: 0.9738 - val_loss: 2.1045 - val_accuracy: 0.7173
Epoch 18/20
657/657 [==============================] - 16s 24ms/step - loss: 0.0756 -
accuracy: 0.9760 - val_loss: 2.3444 - val_accuracy: 0.7161
Epoch 19/20
657/657 [==============================] - 16s 24ms/step - loss: 0.0930 -
accuracy: 0.9715 - val_loss: 2.3581 - val_accuracy: 0.7229
Epoch 20/20
657/657 [==============================] - 16s 24ms/step - loss: 0.0727 -
accuracy: 0.9768 - val_loss: 2.3171 - val_accuracy: 0.7276
```

[106]:
```
plt.plot(history3.history['loss'])
plt.plot(history3.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```



[107]:
```
plt.plot(history3.history['accuracy'])
plt.plot(history3.history['val_accuracy'])
plt.title('model acc')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```



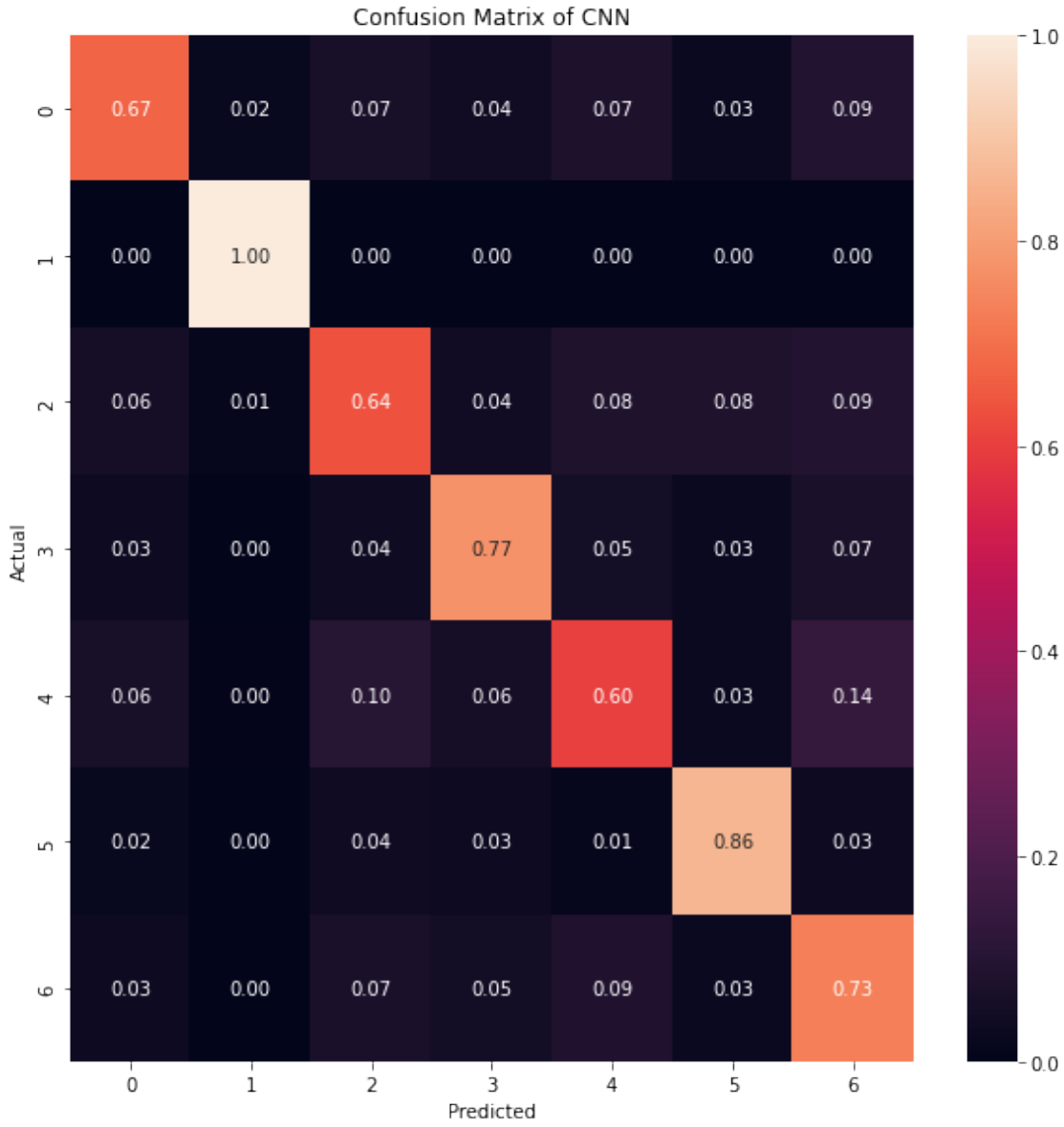
[96]:
```
model3.evaluate(test_x_cnn, test_y)
```

```
180/180 [==============================] - 1s 7ms/step - loss: 2.3736 -
accuracy: 0.7177
```

[96]:
```
[2.3736231327056885, 0.7176941633224487]
```

[97]:
```
cm = confusion_matrix(test_y, np.apply_along_axis(np.argmax, 1, model3.
 ↪predict(test_x_cnn)))
cmn = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
fig, ax = plt.subplots(figsize=(10,10))
sns.heatmap(cmn, annot=True, fmt='.2f', xticklabels=nb.classes_, yticklabels=nb.
 ↪classes_)
plt.title('Confusion Matrix of CNN')
plt.ylabel('Actual')
```

```
plt.xlabel('Predicted')
plt.show(block=False)
```



[108]:
```
model.save('saved_model/ann.h5')
model3.save('saved_model/cnn.h5')
```