Machine Learning and Content Analytics
Face Mask Detection

Konstantinos Alexakis
p2821901

Panagiotis Kosmidis
p2821907

Vaggelis Stathis
p2821925

October 2020

Table of Contents

Face Mask

Since March 2020, our lives have entered into a new reality, the reality of COVID – 19. This situation has triggered lots of changes into people's daily routine. A number of outrageous restrictions where introduced into everyday life, as containment measures, in order to "flatten the curve" and minimize the losses. Citizen where forced to work from home, older people where advised not to leave their houses, schools where closed. The whole economy changed.

One of the restrictive measures is that people are obliged to wear protective masks (either medical or no medical). The World Health Organization (WHO) has changed its advice on face masks, saying they should be worn in public where social distancing is not possible to help stop the spread of coronovirus. The WHO had previously said there was not enough evidence to say that healthy people should wear masks. This contradiction raised many considerations throughout the globe. Many people started questioning such a decision and were not happy to accept such an advice. Governments had to take action against those who were not willing to were face coverings. In many countries it is now illegal not to were mask in public places and an individual could get a fine. Furthermore, the owner of the facility (store, company, private school) could get finned as well or even face a severe punishment such as the complete shut down of his business.

It is now a common strategy for facility owners to employ people to check whether a person wears a mask or not. These employs are most commonly located at the gates of the facility to prevent an individual from entering the facility if he is not wearing face covering. How many employees are needed for such a task? Is there a guarantee that a customer that enters a shop will keep wearing his mask after entering the facility?

1 Description - Mission

1.1 Description

Having in mind the new regulations that were globally forced during the pandemic, a monitoring system to automatically detect individuals who are not wearing a mask is necessary. The new reality obligates additional security personnel on public places in order to observe and impose individuals to wear protective masks. However, let's imagine crowded places such as airports, ports, public services, shopping malls etc. Is it possible on such places where hundreds of people are gathered to be regulated from a small number of security personnel? Reasonable the answer is no. This turns us to solutions provided by technology.

Our objective is to build a model that will detect whether a person wears a mask or not. The model will take a photo of the person, detect the face and will make a decision about face covering. Then, this response will be passed to an automated system that will be able to close the doors and not let the person enter the facility (if the person is outside) or to inform the guards that an individual does not wear mask (if the person entered and removed his mask). Such a system will help the employees focus on more profitable tasks.

The last problem that needs to be solved is what will be the use of such a system when the pandemic ends. In this case the model will be used in the opposite way. It's use will be to prevent a person that is wearing a mask to enter the facility. This will be a serious obstacle if someone wants to make a robbery since he will have to reveal his face in order to enter.

1.2 Mission

In that phase, we should stand and think what it is done on similar cases. For example, we may think airports or courts on which security measures are extensive. On these places it is common practice to use close circuits television (CCTV) cameras in order to detect gun possession or illegal substances. Having this mentality, the task that we are trying to complete is to use CCTVs on places where big number of groups are gathered and detect whether they wear their mask. Our task is going to be divided into two major categories:

- Face recognition
- Classification whether somebody wears mask or not

Each category will be treated separately and finally we will combine them in order to generate integrated results. Using the face recognition algorithm, we are going to detect people from images taken from cameras and then the detected faces will be used as input on another algorithm that decides if the individual wears mask.

For the first part of our project, we will use a pretrained model (will be explained later on detail) and we will utilize its prediction capabilities. The second part will be created from scratch. Briefly, we will present three different models, two CNNs with different image pre-processing as input and model configuration and one SVM classifier (in reality the models which have been tested are a lot more). Finally, we will present results for each model, we will pick the best one and we will use it on a “virtual” application which will simulate its use.

2 Models

2.1 Data

The most important is to find the appropriate data. Thankfully, there are lot of sources to find the data needed. In fact, we are going to use images for training. These images will be separated into two categories, one category will be images with people wearing masks and images with people not wearing masks. Because we want to test our application on real time video and take good results, we decided to combine images from two sources, in order to have diversity on our data. These sources are:

- MaskedFace-Net (link will be provided on bibliography section)
- Kaggle

Now let us see what data provided each source in detail.

2.1.1 MaskedFace - Net

This source provided to us the biggest part of our data. The maskedFace-Net dataset it is consisted of 1024x1024 70000 colored images, showing people wearing a mask. Also, there is another batch of 70000 images (same size) showing people without mask. Since each batch of images is about 20GB we did not use all of them. So, we chose 3000 images of each category in order to use them during training phase. Also, we used 1000 images of each category that will be used during test phase. It is important to mention here that images that will be used during training were downloaded and saved into two separate folders which are:

- dataset/with_mask
- dataset/without_mask

Dataset is the folder which contains the two subfolders, which contain the images. This separation is done to be easier for us to combine them with images downloaded from Kaggle.

2.1.2 Kaggle

As mentioned above we would like to have different kind of data into our dataset. So, we included images that are different from images used from MaskedFace – Net. Images used from MaskedFace – Net are focused one the face of each person, so we found images that show the whole body of people on different angles. In contrast with previous source, Kaggle images where all saved into one folder so, we had to separate them and save them into corresponding folder (/with_mask, /without_mask). Alongside with images Kaggle provided to us a .csv file which contained the image name and the corresponding class name of each image. Due to the fact that class names contained and classes that we would not like to include into training phase we have to clean our data and use only the needed images. So, using pandas from python and the .csv file mentioned above we kept into our dataset only images that belonged to classes **face_with_mask** and **face_no_mask**. The processed images were saved into folders mentioned above (dataset/with_mask and dataset/without_mask).

2.1.3 Pre – processing

After collecting our data we had to split them into training dataset, which will be used during training phase, and on validation dataset, which will be used during training phase in order to validate the results from training. The data were divided on a percentage of 70% for training and 30% for validation. The result is that we ended up with 7713 images for training and 3308 images for validation divided into the following folders:

- training_data (main folder for training)
  - with_mask (subfolder which contains images showing people wearing a mask)
  - without_mask (subfolder which contains images showing people not wearing a mask)
- validation_data (main folder for validation phase)
  - with_mask (subfolder which contains images showing people wearing a mask)
  - without_mask (subfolder which contains images showing people not wearing a mask)

Finally, keep in mind that we kept from the beginning, about 2000 images in order to test the results from final model, which are saved into a folder named test_data with subfolder with_mask and without_mask.

2.2 Methodology

2.2.1 Face Detection Algorithms

As we already mentioned in previous chapters, for the face-detection part we used two different pre-trained models.

2.2.1.1 Object Detection using Haar feature-based cascade classifiers

Object Detection using Haar feature-based cascade classifiers is an effective object detection method proposed by Paul Viola and Michael Jones in their paper, "Rapid Object Detection using a Boosted Cascade of Simple Features" in 2001. It is a machine learning based approach where a cascade function is trained from a lot of positive and negative images. It is then used to detect objects in other images.

Here we will work with face detection. Initially, the algorithm needs a lot of positive images (images of faces) and negative images (images without faces) to train the classifier. Then we need to extract features from it. For this, Haar features shown in Figure 1 are used. Each feature is a single value obtained by subtracting sum of pixels under the white rectangle from sum of pixels under the black rectangle.

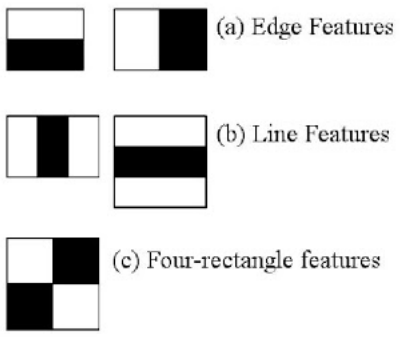

Figure 1: Haar features for Face Detection

The final classifier is a weighted sum of these weak classifiers. It is called weak because it alone can't classify the image, but together with others forms a strong classifier. In an image, most of the image is non-face region. So it is a better idea to have a simple method to check if a window is not a face region.

For this they introduced the concept of Cascade of Classifiers. Instead of applying all 6000 features on a window, the features are grouped into different stages of classifiers and applied one-by-one. (Normally the first few stages will contain fewer features). If a window fails the first stage, will be discarded. We don't consider the remaining features on it. The window which passes all stages is a face region.

2.2.1.2 Face Recognition Project

Paul Viola and Michael Jones invented a way to detect faces that was fast enough to run on cheap cameras. However, much more reliable solutions exist now. We're going to introduce a method invented in 2005 called Histogram of Oriented Gradients — or just HOG for short.

Figure 2: Grayscale Image

To find faces in an image, we'll start by making our image black and white (Figure 2) because we don't need color data to find faces. Then we'll look at every single pixel in our image one at a time. For every single pixel, we want to look at the pixels that directly surrounding it (Figure 3).

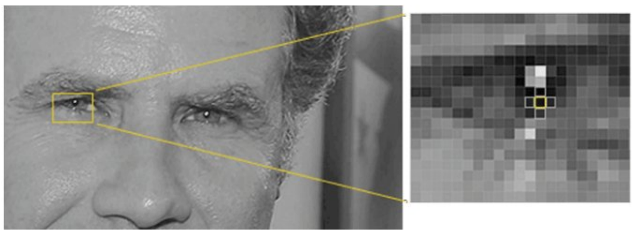

Figure 3: Comparison Between Pixels

Our goal is to figure out how dark the current pixel is compared to the pixels directly surrounding it. Then we want to draw an arrow showing in which direction the image is getting darker as shown in Figure 4. If you repeat that process for every single pixel in the image, you end up with every pixel being replaced by an arrow. These arrows are called gradients and they show the flow from light to dark across the entire image.

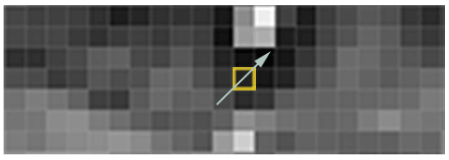

Figure 4: The image is getting darker towards the upper right

Saving the gradient for every single pixel gives us way too much detail. We end up missing the forest for the trees. It would be better if we could just see the basic flow of lightness/darkness at a higher level so we could see the basic pattern of the image.

To do this, we'll break up the image into small squares of 16x16 pixels each. In each square, we'll count up how many gradients point in each major direction (how many point up, point up-right, point right, etc...). Then we'll replace that square in the image with the arrow directions that were the strongest.

The end result is we turn the original image into a very simple representation that captures the basic structure of a face in a simple way:

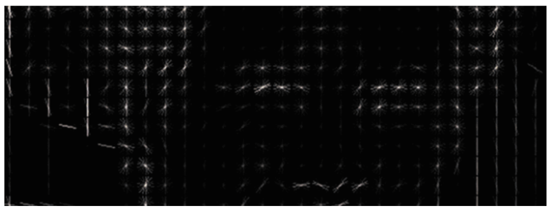

Figure 5: The original image is turned into a HOG representation that captures the major features of the image regardless of image brightnesss.

To find faces in this HOG image, all we have to do is find the part of our image that looks the most similar to a known HOG pattern that was extracted from a bunch of other training faces:

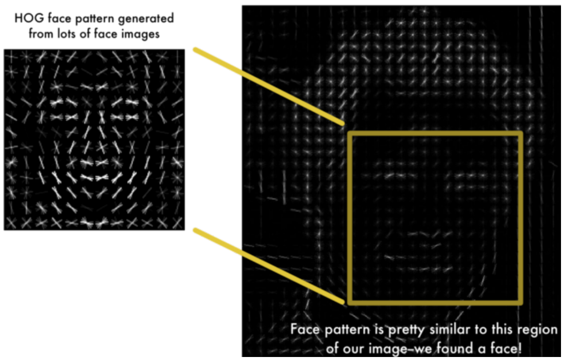
HOG face pattern generated from lots of face images
Face pattern is pretty similar to this region of our image–we found a face!

Figure 6: Comparing Our Image With HOG face Pattern

2.2.2 SVM Model

The first model we create will not be related with neural networks. At this first point we will work with a conventional machine learning algorithm; Support Vector Machine (SVM).

SVM is a supervised learning model with associated learning algorithms that analyze data used for classification. This means that we will train a model in order to be able to classify images (in our case image with faces as will be described later) and decide whether the depicted person wears a mask or not.

The first thing we need to do is to load our dataset. As described on Data section, our images are divided into three different folders for training, validating and testing the model. Each folder contains two subfolders, dividing images to these with mask and these without mask. This structure is extremely useful for the convolutional neural networks, which will be described later, but it is not for SVM.

On this model we need to merge all images into a single structure. This means we will create a single structure that contain train and validation images with and without mask. Of course, as always with images, these will be saved as arrays. In our case, these arrays have the following shape (x, y, 3). ‘x’ represents height, ‘y’ represents width and ‘3’ represents depth. Images are of several sizes, but they are all colorful thus depth dimension is ‘3’.

While we are loading images, we will do two additional things. We will convert them on grayscale mode resulting image’s shape to be converted to (x, y) and we will resize them on (128, 64). Both conversions are targeted. We just have to look at the scale of the problem. Having approx. 8000 images for training and 3500 images for validation, we would create a structure with size 11500 x (x, y, 3), which is enormous. Easily someone can understand that we need to be smart and process our data in a way that will enable us to manage them better.

Additionally, after testing many different models and making several pre-processing tests to our data, we found that greyscale images provide better results. This may be related to the fact that masks can be of very different colors and/ or having many patterns. Thus, making images to greyscale, we lower contrast between pixels and how colors affect algorithm.

One more thing we have not commented is why we used shape (128, 64). This is related with HOG technique. Our task on this project is related with unstructured data, such as images, and how we will extract features from these in order to classify them. For this reason, we will use a popular feature extraction technique for images, HOG, which is described on a previous chapter. What HOG actually does is to remove the unnecessary and heavy information from images, such as shape of objects, color, edges, background etc. and keeps only the most important information.

Steps for HOG:

1. We will divide the image to 8*8 pieces to extract features, image shape (128, 64) makes calculation pretty simple.
2. Calculate gradients for both directions, x and y.
3. Calculate magnitude and orientation.
4. Calculate histogram of gradients for 8x8 cells.
5. Normalize gradients in 16x16 cells.
6. Calculate features for the complete image.

The above process will be performed for all of our images. However, we should note that this process is not performed while loading images. It will happen when training model (will discuss later).

Up until now, we have loaded train and validation images and converted them on grayscale mode and with (128,64) shape. Before moving on, we will shuffle data in order not to bias our model.

Now we are ready to train our model. Let's discuss the general process we will follow.

We will create a pipeline on which the on the first stage we will pass our data from a custom transformer and on the second stage from an SVM classifier.

- Custom Transformer: We will use HOG technique as described above to extract features from our images and flatten them too.
- Classifier: We will define two type of SVM classifiers, one with linear kernel and one with non-liner kernel.

Then we will define some parameters for our classifier to search for the best model. Due to the fact that SVM with linear and non-linear kernel have different hyperparameters to search for, this process will be separated to two different phases. Thus, we will end up with the best model for linear SVM and the best model for non-liner SVM and we will keep the best one.

We continue defining a grid search process for the pipeline and hyperparameters. We will use 10-fold CV in order to find the best model upon the defined hyperparameters.

Here we should stress that due to this versatile functionality of grid search, there was no reason to previously divide our data to train and validation. Thus, we have inserted them all to the same structure. Finally, we train our models and keep the best ones. Below are presented the results.

```
>>>>Non-Linear SVM Model<<<<
Accuracy: 0.9122535723037661
Best parameters: {'clf-svm-rbf__C': 5, 'clf-svm-rbf__gamma': 0.001}
>>>>Linear SVM Model<<<<
Accuracy: 0.8972460939150422
Best parameters: {'clf-svm-lin__C': 0.1}
```

Listing 1: Results for SVM Parameters

We can easily understand from Listing 1 that non-linear SVM with C=5 and gamma=0.001 provide the best results. At this point it would be valuable to take a deeper look on the best model. For this reason, we will train one more model with the best hyperparameters found above. Below we present the results from each run of the 10-fold CV.

```
Details for each iteration:
   accuracy  precision  recall     f1
0     0.930      0.939   0.921  0.927
1     0.929      0.938   0.920  0.926
2     0.924      0.935   0.914  0.921
3     0.933      0.944   0.923  0.930
4     0.919      0.933   0.908  0.916
5     0.924      0.935   0.915  0.922
6     0.934      0.945   0.924  0.931
7     0.924      0.935   0.914  0.920
8     0.920      0.934   0.909  0.916
9     0.921      0.933   0.910  0.917
```

Listing 2: 10 Fold CV for Best SVM Model

Finally, we take present mean values of the above 10 runs as seen in Listing 2.

```
accuracy     0.9258
precision    0.9371
recall       0.9158
f1           0.9226
```

Listing 3: Mean Values of Performance Metrics

The metrics in Listing 3 give us a complete perspective of the model. We can understand that accuracy is relatively high which means that on 92.58% of the cases the classification is correct.

Precision, which talks about how many of the predictive positives are actually positive, is on 93.51% while recall, which talks about how many of the actual positives have been predicted as such, is on 91.58%. Both metrics are relatively high, thus F1 is also high.

Generally, the above metrics points to a model that works very well. However, as always, we must evaluate it on the test set which is mirror of the reality. On the next chapter we will check how model performs there.

Before moving to the next model, let's point out the existence of a log functionality which is used in order to keep record of the results. This is extremely useful in order to permanently store information about the models and their results.

2.2.3 Convolutional model Trained with Cropped Face Images

After reading about the face recognition algorithms, we came up with the idea of creating a model that will be pre-processing images by detecting and keeping only the faces. We first used Open CV library. The results where not what as we would have liked. The algorithm failed in many cases to detect the faces correctly and many no-face images where produced (Figure 7). Then we tried out the face-recognition library which is based in HOG. This time, there where rare false detections but the algorithm failed to find masked faces in many cases. We knew from the beginning of our test, that due to the fact that the two pre-trained algorithms we used where trained with photos of faces without masks, the results where not going to be as expected.

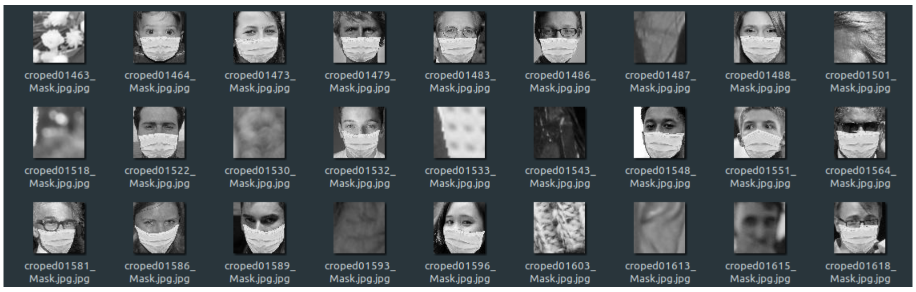
croped01463_ Mask.jpg.jpg
croped01464_ Mask.jpg.jpg
croped01473_ Mask.jpg.jpg
croped01479_ Mask.jpg.jpg
croped01483_ Mask.jpg.jpg
croped01486_ Mask.jpg.jpg
croped01487_ Mask.jpg.jpg
croped01488_ Mask.jpg.jpg
croped01501_ Mask.jpg.jpg
croped01518_ Mask.jpg.jpg
croped01522_ Mask.jpg.jpg
croped01530_ Mask.jpg.jpg
croped01532_ Mask.jpg.jpg
croped01533_ Mask.jpg.jpg
croped01543_ Mask.jpg.jpg
croped01548_ Mask.jpg.jpg
croped01551_ Mask.jpg.jpg
croped01564_ Mask.jpg.jpg
croped01581_ Mask.jpg.jpg
croped01586_ Mask.jpg.jpg
croped01589_ Mask.jpg.jpg
croped01593_ Mask.jpg.jpg
croped01596_ Mask.jpg.jpg
croped01603_ Mask.jpg.jpg
croped01613_ Mask.jpg.jpg
croped01615_ Mask.jpg.jpg
croped01618_ Mask.jpg.jpg

Figure 7: Open CV Algorithm Producing No-Face Images

In Listing 4 we can see the summary of this model.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 32)        9248
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 14, 14, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 6272)              0
_________________________________________________________________
dense (Dense)                (None, 128)               802944
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 258
=================================================================
Total params: 812,770
Trainable params: 812,770
Non-trainable params: 0
```

Listing 4: Summary of Convolutional Model With Pre-Processed Images

The training of this model was very fast due to the fact that the images where of dimensions 64x64. This, resulted an early stopping.

While, in Listing 5, we can see the training of the model.

```
Epoch 1/100
92/92 [==============================] - 3s 29ms/step - loss: 0.3381 -
    accuracy: 0.8623 - val_loss: 0.1333 - val_accuracy: 0.9526
Epoch 2/100
92/92 [==============================] - 3s 29ms/step - loss: 0.1217 -
    accuracy: 0.9612 - val_loss: 0.0861 - val_accuracy: 0.9679
Epoch 3/100
92/92 [==============================] - 3s 28ms/step - loss: 0.0738 -
    accuracy: 0.9772 - val_loss: 0.0709 - val_accuracy: 0.9771
Epoch 4/100
92/92 [==============================] - 3s 28ms/step - loss: 0.0624 -
    accuracy: 0.9789 - val_loss: 0.0695 - val_accuracy: 0.9755
Epoch 5/100
92/92 [==============================] - 3s 28ms/step - loss: 0.0514 -
    accuracy: 0.9844 - val_loss: 0.0317 - val_accuracy: 0.9908
Epoch 6/100
92/92 [==============================] - 3s 28ms/step - loss: 0.0383 -
    accuracy: 0.9869 - val_loss: 0.0321 - val_accuracy: 0.9908
Epoch 7/100
92/92 [==============================] - 3s 29ms/step - loss: 0.0288 -
    accuracy: 0.9913 - val_loss: 0.0319 - val_accuracy: 0.9878
Epoch 8/100
92/92 [==============================] - 3s 29ms/step - loss: 0.0300 -
    accuracy: 0.9898 - val_loss: 0.0330 - val_accuracy: 0.9908
Epoch 9/100
92/92 [==============================] - 3s 29ms/step - loss: 0.0218 -
    accuracy: 0.9937 - val_loss: 0.0250 - val_accuracy: 0.9908
Epoch 10/100
92/92 [==============================] - 3s 29ms/step - loss: 0.0195 -
    accuracy: 0.9940 - val_loss: 0.0267 - val_accuracy: 0.9878
Epoch 11/100
92/92 [==============================] - 3s 30ms/step - loss: 0.0177 -
    accuracy: 0.9951 - val_loss: 0.0278 - val_accuracy: 0.9893
Epoch 12/100
92/92 [==============================] - 3s 30ms/step - loss: 0.0185 -
    accuracy: 0.9937 - val_loss: 0.0312 - val_accuracy: 0.9908
Epoch 13/100
92/92 [==============================] - 3s 30ms/step - loss: 0.0159 -
    accuracy: 0.9940 - val_loss: 0.0382 - val_accuracy: 0.9893
Epoch 14/100
91/92 [============================>.] - ETA: 0s - loss: 0.0115 -
   accuracy: 0.9969Restoring model weights from the end of the best
   epoch.
92/92 [==============================] - 3s 30ms/step - loss: 0.0115 -
    accuracy: 0.9969 - val_loss: 0.0295 - val_accuracy: 0.9862
Epoch 00014: early stopping
```

Listing 5: Training of Convolutional Model With Pre-Processed Images

In Figures 8 and 9 we can see the loss and accuracy diagrams.

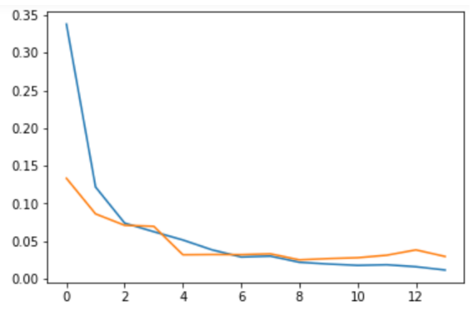

Figure 8: Training and Validation Loss

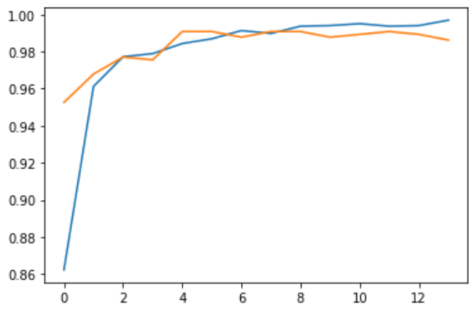

Figure 9: Training and Validation Accuracy

2.2.4 Convolutional model using Keras Image Data Generator

Another model that we would like to try is CNN (Convolutional Neural Network) but here instead of processing the image data of our own, we are going to use **Image Data Generator** from Keras. Using Image Data Generator will help us process the images easier and add some extra functionalities needed. Furthermore, in this model the input of the images will be larger, thus, more Convolutional layers are going to be used in order to apply more filters on images.

Let us get started by explaining the parameters that we have to pass on image data generator and why we are using these specific parameters. In order, for our model, to read images as input we have to transform them into an array. The size of the array will be $300 \cdot 300 \cdot 1$ because we have 300 pixels height, 300 pixels width and 1 channel for color due to the fact that all images will be in a grayscale format. The value for each pixel inside the image lies in range 0-255, so we have to rescale them into range 0-1. In order to achieve this, we have to set **rescale** parameter of Image Data Generator into **rescale = 1.0/255**.

Another benefit about image data generator is that we can easily perform data augmentation to our data. We would like to do this in order to give more information to our model and include instances that are missing from our training dataset. Specifically, we will use two parameters which are:

- zoom range

  This parameter will choose randomly some images and it will zoom into them in order to focus on some details that in another way would be ignored. Zoom augmentation randomly zooms the image in and either adds new pixel values around the image or interpolates pixel values respectively. The zoom amount is uniformly randomly sampled from the zoom region for each dimension (width, height) separately. Here by setting this parameter on [0.5, 1.5] we achieve to make the faces in the image 50% larger or closer with 0.5 and with 1.5 we achieve to zoom out by 50%, so make the faces smaller or be further away.

- brightness range

  This parameter helps us to generate images with different brightness. This means that the brightness of the image can be augmented by either randomly darkening images, brightening images, or both. The intent is to allow the model to generalize across images trained on different lighting levels. Values less than 1.0 darken the image, e.g. [0.5, 1.0], whereas values larger than 1.0 brighten the image, e.g. [1.0, 1.5]. In our case we set this parameter on [0.5, 1.5] in order to generate both brighter and lighter images.

These parameters are assigned only on Image Data Generator object of training dataset, on validation dataset the only parameter that we are using is rescale=1.0/255. So, by setting these parameters we are specifying that in any case our input images will be rescaled into 0-1 range and more examples will be generated. The next step is to specify the location of our data. As mentioned on Data section our training data is divided into two subfolders (/with_mask, /without_mask), so in order to help image data generator to locate our images we have to set as parameter the path of the folder that contains these two subfolders. By using **flow_from_directory** function we set the first parameter with the appropriate path and image data generator locates the images and uses the names of the subfolders in order to set the label of each image. The rest parameters, for **flow_from_directory** function, are:

- **color_mode = 'grayscale'**: transforms all the images into grayscale
- **target_size = (300,300)**: resizes all images to 300x300 pixels
- **batch_size = 128**: during training phase, images will be given into model into batches of 128 at each step
- **class_mode='binary'**: this is used because we have only two classes to predict

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 298, 298, 16)      160
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 149, 149, 16)      0
_________________________________________________________________
dropout (Dropout)            (None, 149, 149, 16)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 147, 147, 32)      4640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 73, 73, 32)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 73, 73, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 71, 71, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 35, 35, 64)        0
_________________________________________________________________
dropout_2 (Dropout)          (None, 35, 35, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 33, 33, 64)        36928
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 16, 16, 64)        0
_________________________________________________________________
```

```
dropout_3 (Dropout)          (None, 16, 16, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 14, 14, 64)        36928
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
dropout_4 (Dropout)          (None, 7, 7, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 3136)              0
_________________________________________________________________
dense (Dense)                (None, 512)               1606144
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 513
=================================================================
Total params: 1,703,809
Trainable params: 1,703,809
Non-trainable params: 0
```

Listing 6: Model Summary

Same values will be used for both training and validation phase. The difference is that on validation phase we set the parameter batch_size=64 because we have fewer images on validation dataset, also we have to set the correct path for validation folder. By setting these parameters we have defined the input shape of the model.

The next step is to define our model. This model's input shape is 300x300x1 because we have 300x300 grayscale images. Moreover, we will use five Conv2D layers, each one of them will take different parameters, five MaxPooling2D 2x2 layers, one Flatten layer and two Dense layers. In addition, dropout will be used to avoid overfitting. Each Conv2D and the first Dense layer will use relu as activation function and the last layer, which will produce the final output, will use sigmoid as activation function because we have a binary classification problem and it is the appropriate activation function for this kind of tasks. Finally, before we fit the model we must go through compilation phase. We will count the loss of the model by using 'binary_crossentropy' (we have binary classification problem), we will use adam optimizer and accuracy metric to measure the performance. In Listing 6 you can see the summary of the model.

The first layer, corresponds to Conv2d layer that we use on our data. Basically, this layer means that we pass a total of 16 3x3 filters on each image. MaxPooling2 is used in order to reduce the dimension of the output. In our case we use 2x2, thus, the output will be reduced by half. You can notice that second Conv2d uses 32 3x3 filters on its input and the rest 64 3x3 filters. All MaxPooling2 layers are of the same size so each one of them reduces its output by half. After these layers we are using a Flatten layer in order to flat the remaining dimension. Finally, we ended up with a 7x7 image with 64 filters and as a result, flatten layer will produce 7x7x64 = 3136 neurons as input for the next Dense layer. The first dense layer has an output of 512 neurons which will be used from the final Dense layer which will produce the result.

Now that we have defined all the above, we are ready to fit our model with our training data. Moreover, we are defining two callbacks to feed them into our model. The first is called EarlyStopping and it is used to stop the training procedure before all epochs are finished. This callback is monitoring val_loss (Validation loss) and stops the training procedure in case that validation loss is not improved for five consecutive epochs. Furthermore, we are using ModelCheckpoint in order to save the best version of the model by monitoring val_loss. We will use 15 epochs to fit our model. Bellow, in Listing 7 you can see the result alongside with some accuracy and loss plots from training and validation phase.

```
Epoch 1/15
61/61 [==============================] - ETA: 0s - loss: 0.5771 - accuracy: 0.6843
Epoch 00001: val_loss improved from inf to 0.36183, saving model to model.h5
61/61 [==============================] - 4198s 69s/step - loss: 0.5771 - accuracy: 0.6843 - val_loss: 0.3618 - val_accuracy: 0.8232
Epoch 2/15
61/61 [==============================] - ETA: 0s - loss: 0.3506 - accuracy: 0.8418
Epoch 00002: val_loss improved from 0.36183 to 0.24610, saving model to model.h5
61/61 [==============================] - 1811s 30s/step - loss: 0.3506 - accuracy: 0.8418 - val_loss: 0.2461 - val_accuracy: 0.8606
Epoch 3/15
61/61 [==============================] - ETA: 0s - loss: 0.2712 - accuracy: 0.8797
Epoch 00003: val_loss improved from 0.24610 to 0.22240, saving model to model.h5
61/61 [==============================] - 1814s 30s/step - loss: 0.2712 - accuracy: 0.8797 - val_loss: 0.2224 - val_accuracy: 0.8818
Epoch 4/15
61/61 [==============================] - ETA: 0s - loss: 0.2290 - accuracy: 0.8952
Epoch 00004: val_loss improved from 0.22240 to 0.18943, saving model to model.h5
61/61 [==============================] - 1856s 30s/step - loss: 0.2290 - accuracy: 0.8952 - val_loss: 0.1894 - val_accuracy: 0.9184
Epoch 5/15
61/61 [==============================] - ETA: 0s - loss: 0.2184 - accuracy: 0.9052
Epoch 00005: val_loss improved from 0.18943 to 0.18069, saving model to model.h5
61/61 [==============================] - 1850s 30s/step - loss: 0.2184 - accuracy: 0.9052 - val_loss: 0.1807 - val_accuracy: 0.9166
Epoch 6/15
61/61 [==============================] - ETA: 0s - loss: 0.2004 - accuracy: 0.9135
Epoch 00006: val_loss did not improve from 0.18069
61/61 [==============================] - 1865s 31s/step - loss: 0.2004 - accuracy: 0.9135 - val_loss: 0.1825 - val_accuracy: 0.9160
Epoch 7/15
61/61 [==============================] - ETA: 0s - loss: 0.1972 - accuracy: 0.9149
Epoch 00007: val_loss improved from 0.18069 to 0.17713, saving model to model.h5
61/61 [==============================] - 1862s 31s/step - loss: 0.1972 - accuracy: 0.9149 - val_loss: 0.1771 - val_accuracy: 0.9178
Epoch 8/15
61/61 [==============================] - ETA: 0s - loss: 0.1889 - accuracy: 0.9204
```

```
Epoch 00008: val_loss improved from 0.17713 to 0.16396, saving model to model.h5
61/61 [==============================] - 1870s 31s/step - loss: 0.1889 - accuracy: 0.9204 - val_loss: 0.1640 - val_accuracy: 0.9247
Epoch 9/15
61/61 [==============================] - ETA: 0s - loss: 0.1828 - accuracy: 0.9223
Epoch 00009: val_loss did not improve from 0.16396
61/61 [==============================] - 1888s 31s/step - loss: 0.1828 - accuracy: 0.9223 - val_loss: 0.1650 - val_accuracy: 0.9253
Epoch 10/15
61/61 [==============================] - ETA: 0s - loss: 0.1783 - accuracy: 0.9231
Epoch 00010: val_loss improved from 0.16396 to 0.16234, saving model to model.h5
61/61 [==============================] - 1874s 31s/step - loss: 0.1783 - accuracy: 0.9231 - val_loss: 0.1623 - val_accuracy: 0.9281
Epoch 11/15
61/61 [==============================] - ETA: 0s - loss: 0.1824 - accuracy: 0.9199
Epoch 00011: val_loss improved from 0.16234 to 0.15734, saving model to model.h5
61/61 [==============================] - 1909s 31s/step - loss: 0.1824 - accuracy: 0.9199 - val_loss: 0.1573 - val_accuracy: 0.9281
Epoch 12/15
61/61 [==============================] - ETA: 0s - loss: 0.1786 - accuracy: 0.9214
Epoch 00012: val_loss did not improve from 0.15734
61/61 [==============================] - 1988s 33s/step - loss: 0.1786 - accuracy: 0.9214 - val_loss: 0.1580 - val_accuracy: 0.9274
Epoch 13/15
61/61 [==============================] - ETA: 0s - loss: 0.1705 - accuracy: 0.9251
Epoch 00013: val_loss improved from 0.15734 to 0.15505, saving model to model.h5
61/61 [==============================] - 1928s 32s/step - loss: 0.1705 - accuracy: 0.9251 - val_loss: 0.1551 - val_accuracy: 0.9287
Epoch 14/15
61/61 [==============================] - ETA: 0s - loss: 0.1672 - accuracy: 0.9269
Epoch 00014: val_loss improved from 0.15505 to 0.15220, saving model to model.h5
61/61 [==============================] - 1997s 33s/step - loss: 0.1672 - accuracy: 0.9269 - val_loss: 0.1522 - val_accuracy: 0.9314
Epoch 15/15
61/61 [==============================] - ETA: 0s - loss: 0.1660 - accuracy: 0.9279
Epoch 00015: val_loss did not improve from 0.15220
61/61 [==============================] - 1943s 32s/step - loss: 0.1660 - accuracy: 0.9279 - val_loss: 0.1533 - val_accuracy: 0.9290
```

Listing 7: Model Training

And the corresponding plots are:

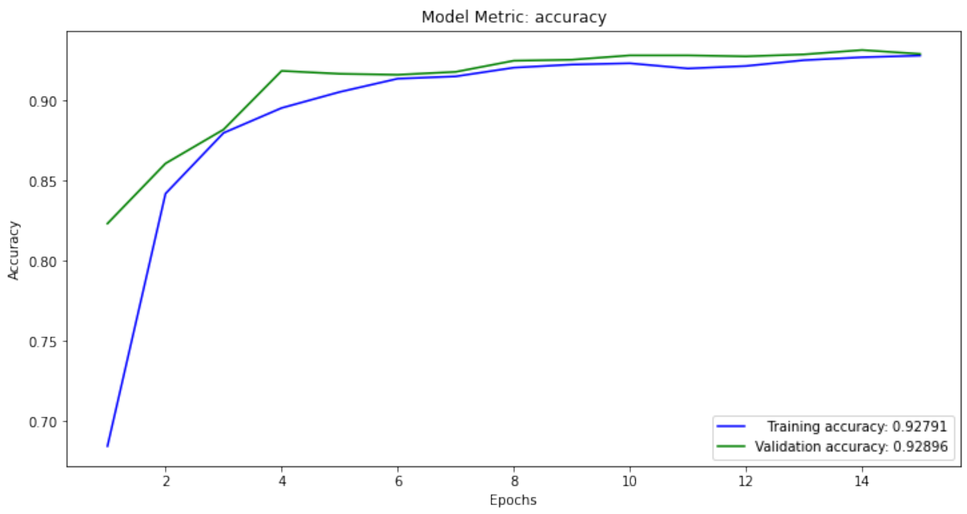

Figure 10: Training - Validation Accuracy Plot

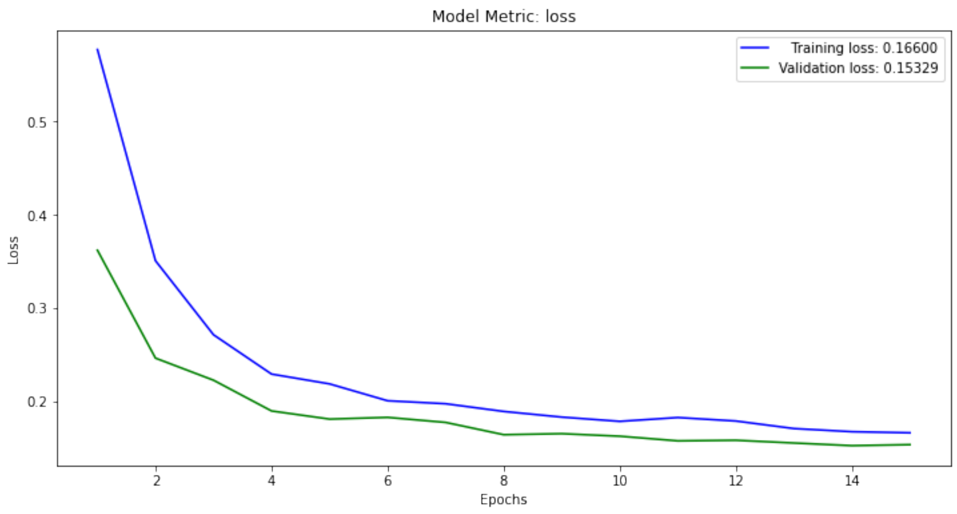

Figure 11: Training - Validation Loss Plot

2.3 Results

On this section we are going to test how our models really perform. To accomplish this, we will use test set. In reality this is not completely correct, but due to the fact that models use dataset with different ways, other use (300, 300) images, other use cropped images, other user (128, 64) images, and divide dataset on train and validation with different ways, it is impossible to compare algorithms on validation set.

That is, we will make a compromise and we will check how model go on real cases and we will pick the best one.

2.3.1 SVM Model

Non-linear SVM model gives accuracy of 99.9% which is surprisingly good for a test set. This is presented in the confusion matrix in Listing 8.

```
Confusion matrix:
[[953   1]
 [  1 999]]
```

Listing 8: Confusion Matrix on Test Set

From the table we can understand that from 1954 (954+1000) images, 1952 are predicted correct while a small subset of two images where misclassified.

Finally, below we may check the ROC curve which is a probability curve and AUC represents degree or measure of separability. It tells how much model is capable of distinguishing between classes. Higher the AUC, better the model is at predicting 0s as 0s and 1s as 1s. The ROC curve is plotted with TPR against the FPR where TPR is on y-axis and FPR is on the x-axis.

For our case, we can see Figure 12.

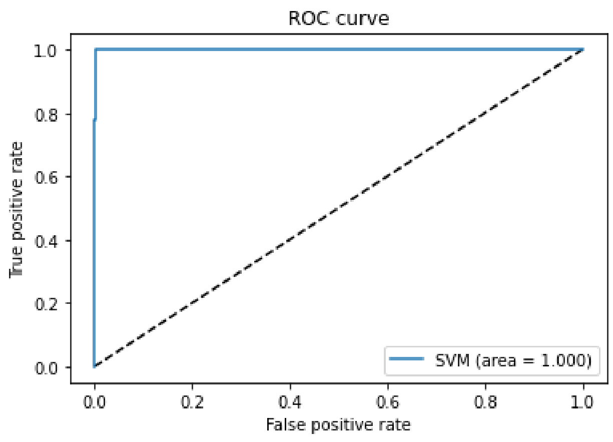

Figure 12: Roc Curve for the SVM Model

2.3.2 Convolutional model Trained with Cropped Face Images

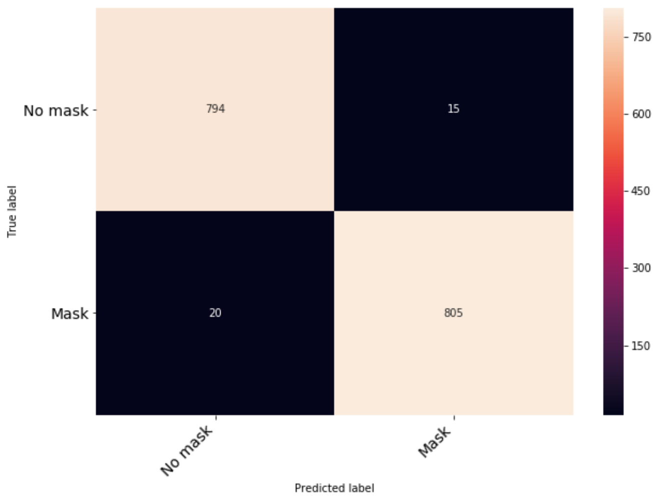

Figure 13: Confusion Matrix

The idea behind this algorithm was to train a model in the same way the video app functions. The problem is that the pretrained algorithms that we used, didn't manage to capture all the faces from the images. In addition some of the faces that the algorithm found, were cropped in an unnatural way. This caused worst predictions for this model compared with the other two algorithms, as we can see in Figure 13.

2.3.3 Convolutional model using Keras Image Data Generator

Bellow, in Listing 9 you can see the classification report of this model for the test dataset.

```
Found 1954 images belonging to 2 classes.
              precision    recall  f1-score   support

   with_mask       1.00      1.00      1.00       954
without_mask       1.00      1.00      1.00      1000

    accuracy                           1.00      1954
   macro avg       1.00      1.00      1.00      1954
weighted avg       1.00      1.00      1.00      1954
```

Listing 9: Classification Report

As you can see the results are perfect. Furthermore, in Figure 14 you can see the confusion matrix of this model.

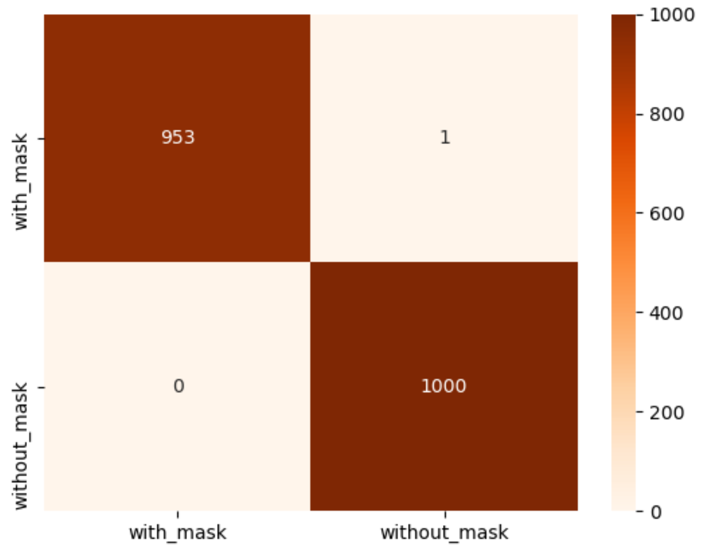

Figure 14: Confusion Matrix

The model classifies correctly all images except for one and this is why precision, recall, accuracy and F1 score have such great values. Finally, in Figure 15 you can see the ROC curve for this model:

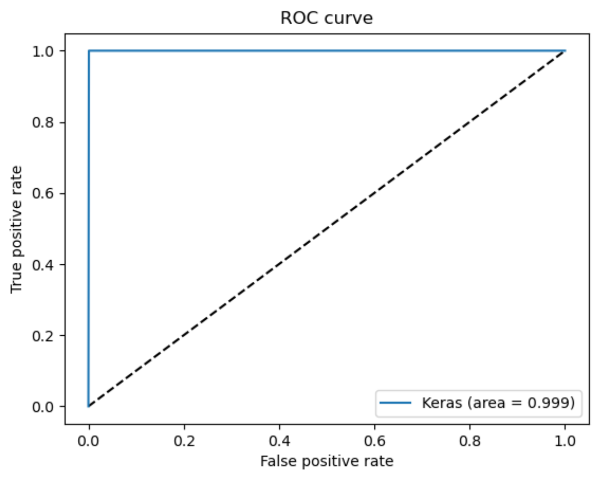

Figure 15: Roc Curve on CNN Model

3 General Information

3.1 Web App

After having described and explained how we created our model, everything comes down to the question how these can be useful in real world cases. For this reason, we have created a simple yet enlightening application which illustrates how these models could be meaningful for humanity.

As said before, our goal is to create an application on which we will automatically identify whether an individual wear their mask or not. We will try to simulate CCTV functionality.

To accomplish that, we have created an HTML application using Flask web framework. Application's functionality is simple. There are two buttons, "Start Recording" and "Stop Recording", which start and stop live video correspondingly. When camera opens, we perform face recognition and mask detection in real time, returning green and red rectangles around individual's face based on classification results. All the results are also written to a log file for future reference.

There is no need to describe how Flask application is set up as it is out of the scope of project. We could briefly mention two things:

1. Due to the real time results we want to achieve, we use multipart generator functions. Basically, when video starts, we divide it into several frames. On each frame models are applied and then are returned as jpg images in bytes. These images are constantly returning to user's screen with classification results constructing video. The above process could be related to streaming processes.

2. Functionalities on application have been deployed with the use of Javascript.

Now that we have described the outline of the application, we are ready to discuss how our models are put in action. As said above, video is divided into frames. For each frame we perform face recognition. After having applied this model, we take as result a list with the position of the faces in the frame. Then, for each face on the frame, we apply the following steps:

1. We crop the frame around the face. This is a very interesting point we should discuss. Our model has been trained with images on which only one individual is depicted and their relative distance from camera is close. However, in real world this would not be the case. Camera will probably be recording human beings from distance while there would be more than one individual on the scene. Cutting the images around each individual's face, we bring the prediction frame to the same context as the training sets.
2. Pre-process the cropped frame (resize, normalize, grayscale).
3. Make prediction.
4. Based on the prediction, create a red or green rectangle around individual's face on the frame.

When we have finished with every face, we return the frame as jpg image with bytes (here are applied generator functions discussed above).

3.2 Who We Are

Our team consists of three members. Each one of us developed one of the models mentioned on previous sections. All three members contributed in each step of this assignment by providing code and ideas on how to make the models.

- Alexakis Konstantinos. Responsible for the 'Convolutional model using Keras Image Data Generator' and the video application.
- Kosmidis Panagiotis. Responsible for the 'SVM model' and the video web application.
- Stathis Evangelos. Responsible for the 'Convolutional model Trained with Cropped Face Images', the Dataset and the report.

3.3 Contact

- Alexakis Konstantinos
  - Mob: 6972301563
  - Email: kalexakis104@gmail.com
- Kosmidis Panagiotis
  - Mob: 6932889352
  - Email: kosmidispanos@gmail.com
- Stathis Evangelos
  - Mob: 6998159567
  - Email: vagstathis@gmail.com

3.4 Timeplan

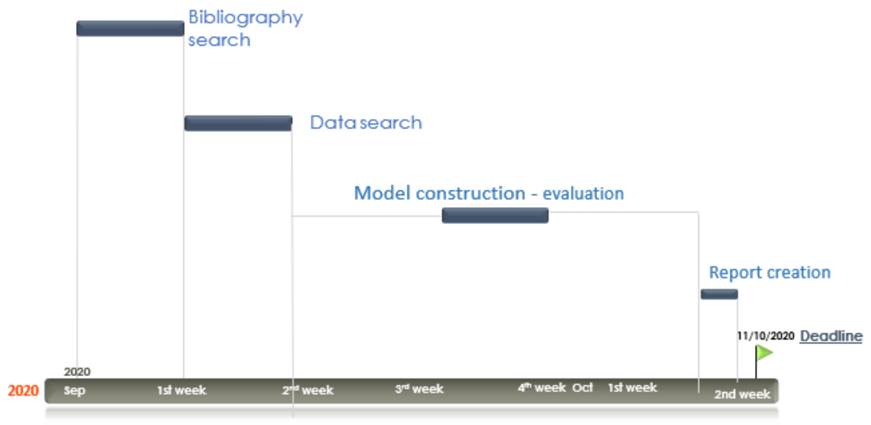

Figure 16: Project's Timeplan

List of Figures

Listings

References

[1] opencv.org: Cascade Classifier,
`https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html`

[2] medium.com: Adam Geitgey, Modern Face Recognition with Deep Learning,
`https://medium.com/@ageitgey/`
`machine-learning-is-fun-part-4`
`-modern-face-recognition-with-deep-learning-c3cffc121d78`

[3] analyticsvidhya: Aishwarya Singh, Feature Engineering for Images: A Valuable Introduction to the HOG Feature Descriptor,
`https://www.analyticsvidhya.com/blog/2019/09/`
`feature-engineering-images-introduction-hog-feature-descriptor/`

[4] towardsdatascience.com: SVM Parameter Tuning in Scikit Learn using GridSearchCV,
`https://towardsdatascience.com/`
`svm-hyper-parameter-tuning-using-gridsearchcv-49c0bc55ce29`

[5] Vincent Dumoulin and Francesco Visin, *A guide to convolution arithmetic for deep learning*, (2018).

[6] Sebastian Ruder, *An overview of gradient descent optimization algorithms*, (2016).