Face Recognition

João Nogueira - 40%
DETI, TAA, Pétia Georgieva
Universidade de Aveiro
Aveiro, Portugal
joaonogueira20@ua.pt
89262

Tiago Melo - 60%
DETI, TAA, Pétia Georgieva
Universidade de Aveiro
Aveiro, Portugal
tiagomelo@ua.pt
89005

***Abstract*—The goal of this paper is to study and compare different Machine Learning models and their performances in the context of the classical problem of face recognition. The benchmarked models include Linear Regression, Logistic Regression, Support Vector Machines (SVM), Neural Networks and Decision Trees. After implementing and testing each of these, one is chosen and its hyper-parameters are finely tuned. We aim to present a system that is able to identify and recognize a subject's head in near-real-time without too much added complexity. In order to do this, every trained model will be shortly explained, as well as the corresponding development strategy. Furthermore, the data are gonna be studied and analyzed beforehand in order to more easily explain and troubleshoot problems when discussing results. This pre-processing consists of, essentially, plan ahead what classes are similar and more likely to be mislabelled and, in turn, lower the model's accuracy, recall and precision. Lastly, we will make use of Eigenfaces to reduce example dimensions and have a more efficient and faster system that is able to recognize faces via Principal Component Analysis. By doing this in the Exploratory Data Analysis, we will also have a better grasp of the data and the images that are most prone to being wrongly labelled by being too similar.**

***Index Terms*—face-recognition, machine learning, models, fine-tuning, linear regression, logistic regression, neural networks, support vector machines, decision trees**

I. Introduction

Face recognition has proven to be of more and more use as technology advances. Real-time identification of individuals, further security when unlocking personal devices or implementing a social credit point system. Unlike face detection, which consists of identifying where in a picture can faces be spotted [1], face recognition focuses on understanding who the face in a picture belongs to; in other words, identifying people. As mentioned before, this project aims to propose and develop a well-performing Machine Learning model that is able to recognize and identify faces with acceptable efficiency. This paper has the goal of explaining the thought process behind the development of said project, the obtained results when testing, validating and comparing different models and how feature extraction via PCA can prove to be of extreme relevance on such projects.

This article is structured in the following manner: we will start by briefly reviewing and analysing past work done in similar such problems using similar, or even the same, data sets in the State of the Art Review section. In the Data section, we will go through the content of the data set, as well as explain its main features, how it was loaded, processed and split before it could be fed to our models. Additionally, we will also perform a short Exploratory Data Analysis by building the mean faces and Eigenfaces, which will allow us to find similar subjects and labels. Eigenfaces essentially consist of extracting the most relevant features regarding given subjects by means of Principal Component Analysis [2]. Once EDA has been completed, we will proceed towards the experimentation between different models for solving the task at hand in the Picking a Model section. Results will be briefly discussed and compared with other model's. In the Fine Tuning the Model section, we will pick a model and, as the name suggests, tweak the hyper-parameters in order to maximize the model's performance. Lastly, we will discuss and display the results and conclusions obtained from this project, in the Results and Conclusions sections, respectively.

II. State of the Art

For this project, we researched and studied 5 papers that we considered relevant for our goal: to find a well performing machine learning model for face recognition. We tried to cover the main models we implemented throughout the project as well as papers that used the same data as ours: the ORL face database.

M. Turk and A. Pentland [2] in their paper "Face Recognition Using Eigenfaces" treat face recognition as a two-dimensional recognition problem, taking advantage of the fact that faces are normally upright and thus may be described by a small set of 2-D characteristic views. They then project faces into a "face space", defined by the eigenvectors of the set of faces.

F. S. Samaria and A. C. Harter [3] in the paper "Parameterization of a stochastic model" for human face identification propose a top-bottom Hidden Markov Model that allows feature encoding and focus on the parameterisation of such a model, resulting on a testing accuracy of 87%.

Guodong Guo, Stan Z. Li, and Kapluk Chan [4] in the paper "Face Recognition by Support Vector Machines" study the performance of Support Vector Machines with a binary tree recognition strategy using the ORL database and an extended version of such. They also compare SVM performance with the standard Eigenface approach using Nearest Center Classification. They achieved a total accuracy of 91.21%.

Steve Lawrence, C. Lee Giles, Ah Chung Tsoi and Andrew D. Back [5] in the paper "Face Recognition: A Convolutional Neural-Network Approach" present a hybrid neural-network solution for Face Recognition and compare it with Hidden Markov Model based approaches. Their final model obtained a total of 96.2% test accuracy on the ORL face dataset.

Ratnesh Kumar Shukla and Arvind Kumar Tiwari [6] in the paper "Machine Learning approaches for Face Identification Feed Forward Algorithms" discuss the use of Convolutional Neural Networks and Deep Learning for face identification.

Stephen Baladan [7] focuses on discussing the state of the Art of Deep Learning the field of Face Recognition in his paper "Deep learning and face recognition: the state of the art", arguing the best Model is Google FaceNet with a total accuracy of 0.9963 using a dataset of 200 million images.

III. Data

A. Data description

The data used for the development of this project is the ORL Face Database. It consists of pictures of 40 people, where each person has had 10 photographs taken of them. The images have a height of 112 pixels by 92. They have been cropped and centered, so that the system can focus more on Facial Recognition, rather than an also popular problem in Machine Learning, Face Detection, as explained in the Introduction. Besides the people present in the original data set, a kind contributor has also added his pictures to the file system. Hence, our data consist of 410 112 by 92 grey scale images of 41 subjects, having 10 pictures each.

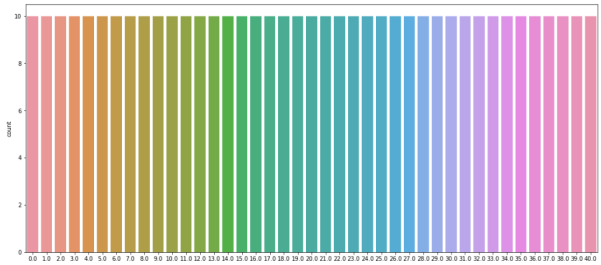

Fig. 1. Histogram displaying the frequency of each label.

That being said, the data were loaded into a 410 * (112 * 92 + 1) matrix. We want to have a matrix that holds a flattened version of the image on each row and has the name of the subject on its last column. Since our data set, unfortunately, did not come with named subjects, we simply labelled each individual with the folder number the picture was stored in. An example of the resulting matrix can be seen in Figure 1.

	0	1	2	3	4	5	6	7	8	9	...
0	0.403922	0.407843	0.403922	0.407843	0.407843	0.396078	0.411765	0.400000	0.400000	0.411765	...
1	0.125490	0.145098	0.125490	0.145098	0.133333	0.149020	0.133333	0.141176	0.137255	0.145098	...
2	0.141176	0.141176	0.149020	0.152941	0.121569	0.160784	0.141176	0.145098	0.149020	0.137255	...
3	0.458824	0.470588	0.478431	0.470588	0.474510	0.470588	0.474510	0.458824	0.474510	0.474510	...
4	0.396078	0.407843	0.407843	0.400000	0.403922	0.415686	0.400000	0.411765	0.407843	0.419608	...

Fig. 2. Samples of the first 10 pixels of the first 5 images in the data set.

The images were stored in the Portable Graymap format. In other words, each pixel corresponded to an integer, ranging from 0 to 255, where the closer to 0 the darker the represented color is. After loading the data, it was normalized. By normalizing the information that we are working with, we are assuring we prune out any anomaly that we might find as well as ease the system performance and data load. The chosen method to normalize our data was to simply divide each pixel by 255, which resulted in each of these being converted into a number between 0 and 1.

To make sure out data was properly loaded and parsed, we can randomly sample and display images in our data set. Each of these has the corresponding label below them.

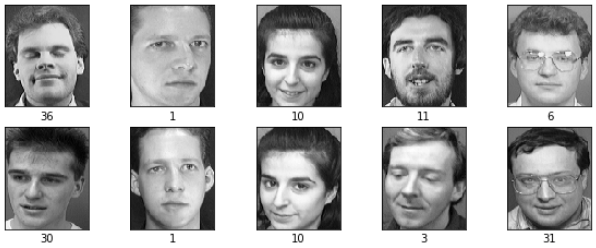

Fig. 3. Random examples of the images used for this project.

In order to load the data from the local folders into the script, we made use of Python's "os" library, which can traverse through the paths passed on as arguments recursively and retrieve the files in them. This resulted in most of the data being poorly mixed. In other words, same label images were bundled together in subsequent rows. Hence, when we split the data, the training set lacked a lot of information about other training examples. In order to prevent this, we shuffled the data beforehand, to dramatically increase the odds of such a scenario being near impossible. Afterwards, splitting is done normally: 60% (246 examples) for the training set, 20% (82 examples) for validation and testing sets.

B. Exploratory Data Analysis

When performing EDA in Face Recognition problems, we considered relevant to plot the mean face for each class and inspect similar results, in order to have a better grasp on which classes are more likely to be wrongly labelled when making predictions as well as total average face, to understand how well formatted images are. From the figure, we can immediately infer most of the images in the data set have been either poorly cropped or centered, since we do not have a clean, understandable face, and rather a blur.

As to the average faces per class, when calculating the MSE between each classes we obtain rather noisy and unclear results. Our explanation for such is that since most images are "alike", given they are all faces and follow the same structure, MSE is not the best approach for doing this kind of EDA. We can also notice that the first subject seems to differ a lot from others, but this is likely due to different photography conditions since, as we can recall, our first subject is the individual that made their own contribution to the data set.

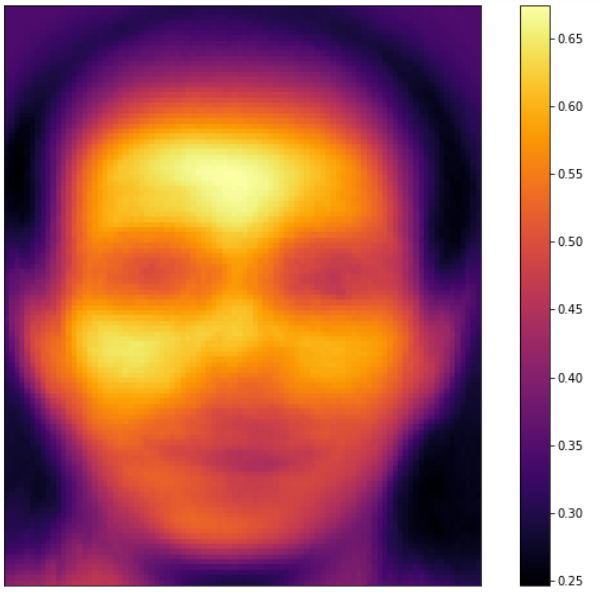

Fig. 4. Average face overall.

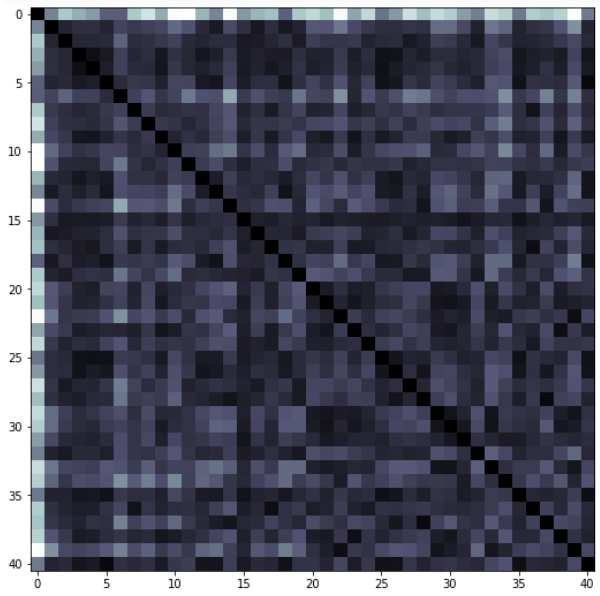

Fig. 6. Similarity matrix between the previously calculated average faces.

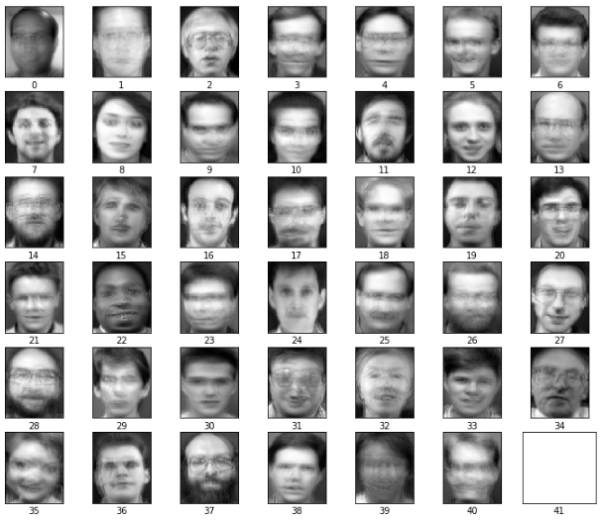

Fig. 5. Average face per class.

IV. Picking a Model

Like explained above, we set out to try a set of machine learning models, pick the best-performing one and fine tune it. The models were either implemented by hand, using libraries or both, in the case of Logistic Regression. In order to more accurately pick a best-performing model, instead of simply training and validating one instance with randomly selected hyper-parameters, we tried to increase the odds of developing and picking a good model by creating and comparing three different sets of hyper-parameters. This way, we can pick the model that behaves the best based on its best set of hyper-parameters.

A. Linear Regression

One of the simplest Machine Learning models, Linear Regression consists of tweaking a set of parameters (Thetas) until they converge into a set of values that best translates the real world. This is called training. Since in our example we have 10304 pixels, our model will consist of a one by 10304 values, each corresponding to the correct weight each pixel should have in order to accurately predict classes. Since this model is usually used in continuous contexts and outputs a real number, we validated it by casting the output to an integer and comparing it with the right folder number, also cast to an integer.

In order to do accomplish this, we focus on minimizing a function that allows us to understand how ”far” our model is from reality. By finding the set of thetas that allows this minimum cost function, we achieve the parameters that bring our model the closest to reality. In our case, the cost function consists of multiplying our set of parameters by the input, resulting in matrix called Hypothesis. We then calculate the element-wise difference between our Hypothesis and the real values it should assume, y, square it, to ensure positive values and calculate the mean. This is called the Mean Squared Error method and is one of the most popular choices for cost calculation.

There is a number of algorithms that aim to reduce this cost function in the best way possible. These are called optimizers. The optimizer chosen for this model was the Gradient Descent. In Calculus, the gradient can be interpreted as the direction or rate of the fastest increase. Hence, it’s inverse can be used to find a minimum of any function, namely the Cost Function. Naturally, this choice has a few caveats, such as local minima.

Linear Regression was implemented by hand. Based on course code, we can pick a number of iterations (i. e. how

many times the gradient will ”descend”) and a learning rate, how much it does descend every iteration. As explained, on every iteration we measure the cost function of the current thetas, calculate the gradient for the current position and tweak them in the direction told by the gradient, iteratively updating our parameters and, hopefully, dragging them closer to an accurate, real world prediction model. However, it is worth mentioning this model performs best for continuous values and not classification problems.

We can see the variation of the each set of hyper-parameters' cost function in the plot in Fig. 1. We can naturally infer the best choice for our Linear Regression candidate would be model 4. However, as expected, since Linear Regression isn't fit for a classification problem, this model performed poorly predicting classes, obtaining only at best 2.1% accuracy on the validation set.

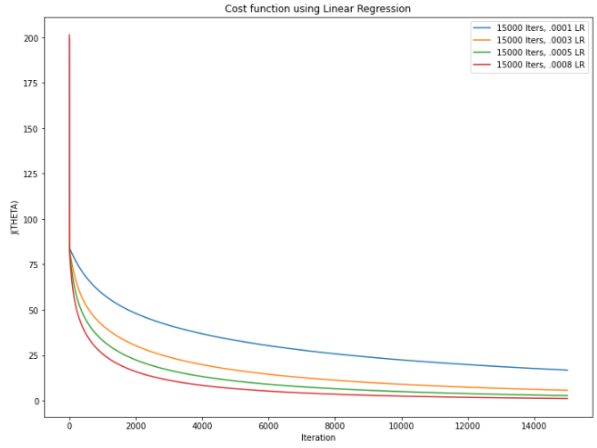

Fig. 7. Cost function evolution after training four distinct Linear Regression models with different learning rates.

B. Logistic Regression

For a classification problem, a much more suitable choice for a model would be Logistic Regression. It differs only on a few key concepts when compared to Linear Regression, such as the cost function calculation. For instance, whereas in Linear Regression the cost is calculated as explained above, in logistic regression the cost function is calculated by increasing the value of the cost the closer it is to the opposite label. This assumes a binary model, however. Since the task at hand consists of a multiclass classification problem, we must adopt a fitting strategy, such as One Versus All. This consists of training one binary classifier for each existing class. Later, for predictions, the winner takes all: the binary classifier with the highest output score assigns the class.

The rest is similar. We have a set of parameters we aim to reduce. When making predictions we multiply the resulting theta vector by our parameters and it outputs a list of numbers that can be translated to probabilities using a function such as SoftMax, where each represents the probability that a given input corresponds to a given label.

Much like Linear Regression, Logistic Regression was also implemented by hand, based on course code. There is a set of customizable hyper-parameters, such as the number of iterations and learning rate, that can be finely tuned. The one-vs-all strategy was also based on course code. We trained four Logistic Regression models with different sets of hyper-parameters. It is worth mentioning that, since we are making use of the One Vs All strategy, we will run all iterations per classifier. In other words, the total of iterations for a given set of hyper-parameters will be equal to the number of classes (in our case 41) times the number of iterations.

That being said, the best performing model got a training accuracy of 11.38%. Each model's cost function can be seen in the plot below, with the corresponding set of hyper-parameters chosen. Although we can naturally infer all models converge to nearly the same value, we will pick the one that converges to the lowest cost function, regardless of how small the change.

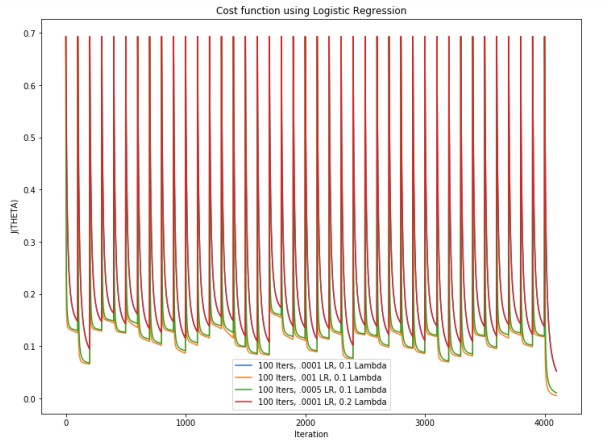

Fig. 8. Cost function evolution after training four distinct Logistic Regression models with different learning rates.

On the other hand, the scikit-learn library [9] contains an already built-in Logistic Regression model, we need only insert the initial parameters. For this case, we chose to only use the regularization parameter C, set to 1, which achieved a reasonably high validation accuracy: 95.16%. Currently, this is the best candidate for further fine-tuning and, lastly, testing.

C. Support Vector Machines

An also fitting choice for a classification problem would be a Support Vector Machine (SVM). A SVM classifies data by finding the most adequate boundary that separates classes. This can be done by finding the boundary that maximizes the margin between the closest data point of each class to the border line. It is worth mentioning that only the closest points of each class, denominated by support vectors are used to determine the optimum margin between each label. In other words, for a two-class classification problem, the goal is to separate the two classes by a function which is induced from available examples. Consider an example where there are many possible linear classifiers that can separate the data, but there is only one that maximizes the margin (the distance

between the hyper-plane and the nearest data point of each class). This linear classifier is termed the optimal separating hyper-plane (OSH). Intuitively, we would expect this boundary to generalize well as opposed to the other possible boundaries [4].

The cost function used in this model typically consists of an adaptation of the Logistic Regression cost function. It adds a regularization parameter C, which adjusts the penalty for misclassified training examples. Much like past models, the optimization objective of a SVM consists of finding a set of Thetas that brings the hypothesis as close as possible to reality, i. e., minimizes the cost function.

The library scikit-learn [9] also offers a built-in SVM model, which needs only the initial hyper-parameters, such as C. For our model, we chose the default value for the penalization attribute: one. This resulted in a final validation accuracy of 98.78%. The customizable hyper-parameters this model has will be further explained later on.

D. Neural Networks

Similarly to other ML models, given a set of initial Thetas, our Neural Network will try to find the combination of Thetas that achieves the lowest cost function possible with a few caveats. A cost function, in its core, calculates how ”far” the combination of Thetas is from the real world. By using the Vector Theta to calculate, in our example, the values of each pixel and calculating the Mean Square Error we will then have a useful metric that we wish to reduce to as low as possible, in order to have ”edges” that allow our Network to learn and is able to accurately represent the real world.

There are a number of algorithms that aim to reduce the forementioned cost function, called optimizers. In this paper, we will be using the Gradient Descent. A gradient, in Calculus, can be interpreted as the direction or rate of the fastest increase in a given plot. Hence, it’s inverse can be used to find a minimum of our Cost Function [10]. So we will be tweaking our Theta Vector in the ”direction” of the most profitable Thetas, i. e., the ones that minimize J. This method, however, is still vulnerable to be stuck at local minima, since there would be no direction in which the algorithm could ”move” to lower the cost function. Furthermore, since this is a multiclass classification problem, we also need to employ the One Vs All strategy in this case, just like we did in Logistic Regression.

A Neural Network needs an activation function, in order to map results. Common choices are usually the Sigmoid (or Logistic Function) or Tanh, that map any real number to a number between 0 and 1, which proves to be extremely useful in classification problems, which is our case. Our activation function will be responsible for mapping the result of the Cost Function to a probability of outcome of each label. For instance, in the context of ASL, we will use the Sigmoid to achieve the vector of the model predictions for all training examples, which afterwards will be used to calculate the Cost Function and adapt Thetas accordingly. Additionally, since we have a Classification problem with multiple labels, we will use the One versus All strategy, which essentially consists in mapping each of the desired outputs to a vector in which the index that corresponds to the desired label is 1 and the rest is 0. Lastly, with the resulting vector of costs, we will use the SoftMax function, to get how likely a given label is to correspond to an example [11].

Just like in other models, we defined three different sets of hyper-parameters, to increase the chances of finding a well representative model of a Neural Network’s performance in the context of this problem. The variation of the cost function of each developed model can be seen on the plot on Figure X. Naturally, let us pick the set of hyper-parameters that converges to the lowest cost function the fastest. For this model, we observed a train accuracy of 3.25% for every the three sets of hyper-parameters.

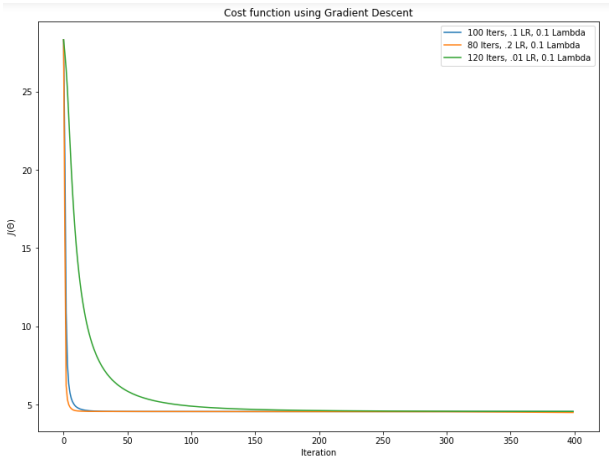

Fig. 9. Cost function evolution after training three distinct Neural Network models with different learning rates.

E. Convolutional Neural Networks

A common variation of a Neural Network consists of a Convolutional Neural Network (CNN). In our project, a simple CNN was developed and used, with the help of libraries such as Tensorflow and Keras. Convolutional Neural Networks are neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

In this project, we implemented a CNN using third-party libraries such as TensorFlow and Keras. Keras allows building a Neural Network using a sequential model. In other words, the network itself is built by putting together layers. In our model we used Conv2D, MaxPooling and Dropout, developing a similar model to the one used in previous papers [11].

Our Neural Network performed fairly well for the training data, achieving an overall training accuracy of 98.37%. However, when validating it, the accuracy fell short of 91% with a total score of 90.24%. We can naturally see this model is overfit for the data, but achieved a nevertheless good score.

V. Decision Trees

In decision analysis, a decision tree can be used to visually and explicitly represent decisions and decision making. As the

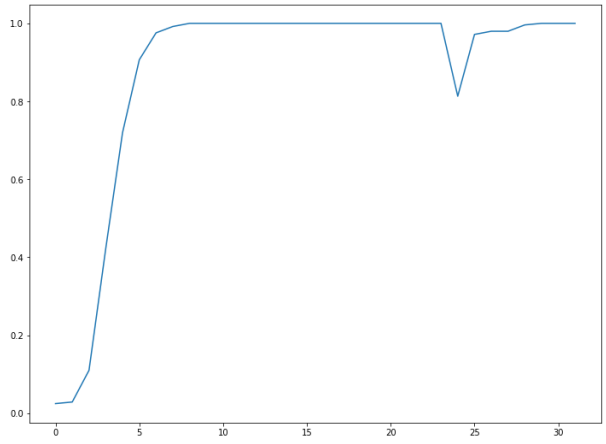

Fig. 10. Accuracy evolution after 32 epochs when training the Convolutional Neural Network model.

name goes, it uses a tree-like model of decisions. Though a commonly used tool in data mining for deriving a strategy to reach a particular goal, it is also widely used in machine learning [12].

In our project, a straight-forward Decision Tree Classifier was used, using the class provided by the scikit-learn library. This model allows us to select a few hyper-parameters, such as the criterion used for the split, the maximum depth of the generated tree and the mininum number of samples required to split an internal node. On a first approach, we trained our model with the "entropy" criterion, along with maximum tree depth of 3 and a minimum number of collected samples of 5.

This resulted in a validation accuracy of 8.16%. After researching and reading further documentation [9], we can conclude our model is fairly overfitting the training data. This might be due to the high feature over training examples ratio. A possible workaround for this problem would be either through feature selection or data augmentation. However, since we already have well-performing models, we chose to discard this option for fine-tuning.

VI. Fine-tuning the model

Bearing in mind these results, we opted to fine-tune the Support Vector Machine model, since it was the one which presented the best accuracy. However, instead of using a Holdout validation set like we did when choosing a model, we opted to implement K-fold cross validation.

There is a number of ways to perform model validation. The reason behind our choice lies vastly on the results discussed by Rodríguez J. and Lazaro J [13]. According to the paper, K-fold validation represents a good sensitive measure to properly validate models. This way we can be sure of the results obtain as we fine-tune our model using the aforementioned algorithm.

Our model hyper-parameter selection was hence done by generating a logarithmically spaced range of values for both C and gamma. C controls the trade-off between a smooth decision boundary and classifying training points correctly [10]. In other words, a larger C will mean a lower bias but, in turn, result in a higher variance for our model. A lower C generally implies higher bias but less varying results. On the other hand, we also fine-tuned the gamma hyper-parameter, is the parameter of a Gaussian Kernel, which is used to handle non-linear classification. A common solution for non-linearly separable data when using Support Vector Machines is to raise the data to a higher dimension, so that it can be separated using hyper-planes. Usually for such cases an RBF kernel is used, we obtained far more stable results using a linear one on both with and without feature extraction, as discussed in the Results section.

The method for finding the best hyper-parameters was done using the Grid Search algorithm. Grid-searching is the process of scanning the data to configure optimal parameters for a given model. Depending on the type of model utilized, certain parameters are necessary, such as, in our case, C and gamma. Since Grid-searching can be applied across machine learning to calculate the best parameters to use for any given model, such an algorithm is vastly used when it comes to fine-tuning machine learning models. It is important to note, however, that Grid-searching can be extremely computationally expensive and may take quite a long time to run, since it iterates through every parameter combination and stores a model for each combination. In other words, Grid Search will build a model for each parameter combination possible and test it [14].

Each of these combinations is then tested and validated against one of the K-fold training-validation sets. In our method we used K = 3, meaning this rotation was done three times for each combination of parameters. The average of results was then calculated and plotted, as we will discuss in the Results section.

VII. Results

The selection of the best hyper-parameters for a model is done automatically, once the grid-search is completed. Below are the obtained results for this experiment with and without feature extraction and their discussion. In the former, feature extraction was done using Principal Component Analysis and generating Eigenfaces, following the methodology developed by Turk and Pentland [2]. Further details will also be explained in that subsection.

A. Without feature extraction

When using unregularized parameters and no Principal Component Analysis or other means of feature extraction, the two models (using Linear and RBF kernels) take a total time of 1176.039 seconds to be finely tuned and trained using Grid-Search.

As to making predictions, once it has been tuned, it is able to predict 82 testing examples in under 0.5 seconds, which results in a total of 164 predictions per second or, in other words, 0.006 seconds per prediction. We obtained a total accuracy of 100% on the testing set, which results in a F1, Precision and Recall scores of 1.00. The obtained confusion matrix can be seen below. It is worth mentioning however that, since the sets have been shuffled, some classes do not show up in the test

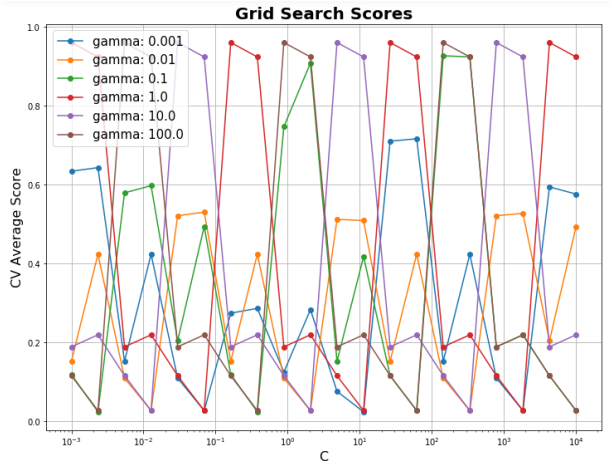

Fig. 11. Obtained average accuracy of 3-fold cross-validations for each combination of hyper-parameters using a RBF kernel for the Support Vector Machine model.

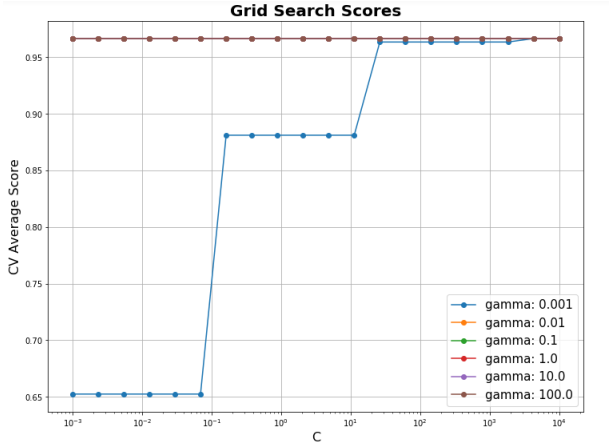

Fig. 12. Obtained average accuracy of 3-fold cross-validations for each combination of hyper-parameters using a Linear kernel for the Support Vector Machine model.

set and are only contained in the training set. For this reason, some rows and columns in the diagonal can be seen blank.

If we randomly select a few testing samples, we notice the model performs well at predicting the represented subjects.

Comparing the results obtained with those stated by [4], we already achieve a better model accuracy and overall metrics, given they obtained less than 92%. Let us now analyze how well our model does when using feature extraction.

B. With feature extraction using Principal Component Analysis

As proposed in [2], by projecting face images into a feature space that best encodes the variation among known face images, we can achieve a much superior performance through means of feature reduction/extraction. The feature space is defined by the eigenvectors of each face: the Eigenfaces, which do not necessarily correspond to real facial features, such as eyes, noses or ears. These eigenvectors are obtained using PCA.

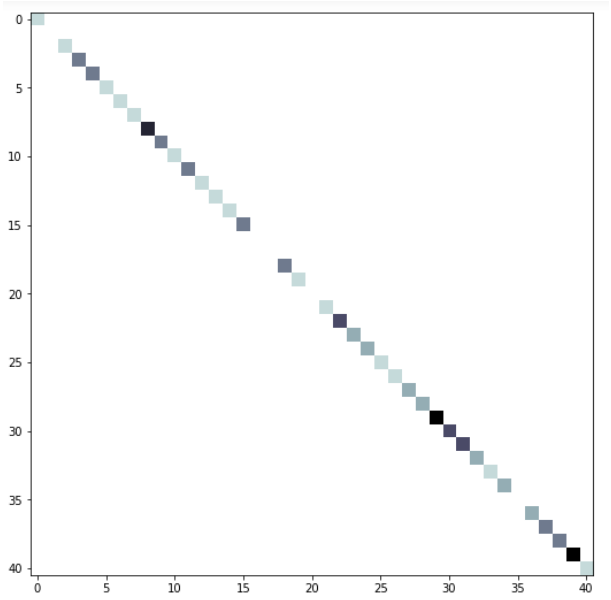

Fig. 13. Confusion matrix for the linear kernel. Some points are darker than others since shuffling results in uneven sets. Yet, there are no points outside the diagonal.

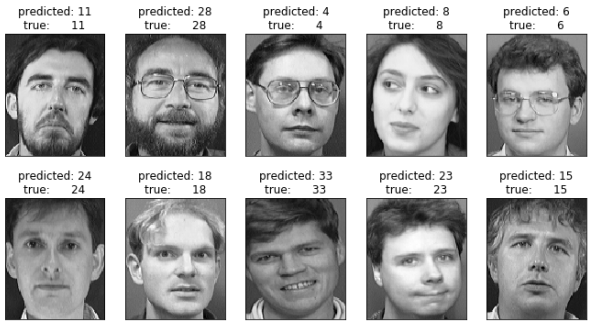

Fig. 14. Identifying subjects.

As to when these methods are applied and the facial features are extracted through Principal Component Analysis, the two models (once again, with Linear and RBF kernels) display much higher performance. In this example, we opted to reduce the feature space to 100 features, from the original 10304. Some of the obtained Eigenfaces can be seen in Figure X.

Fine-tuning our Support Vector Machines with this reduced number of features takes a total time of 7.665 seconds using Grid-Search, about 153 times less compared to the model fine-tuning without any feature extractions.

The quantitative evaluation of the model quality on the test set shows nevertheless similar results. When using PCA, we obtained a total accuracy of 96.34% on the testing set, as well as an F1 Score, Precision and Recall scores of 93.90%, 93.33% and 89.76%, respectively. The obtained confusion matrix can also be seen below.

The qualitative comparison between the two approaches can be seen in Table 1, along with the obtained metrics.

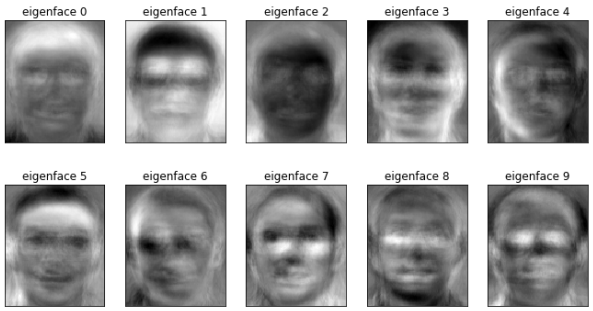

Fig. 15. Some of the resulting Eigenfaces in the data set.

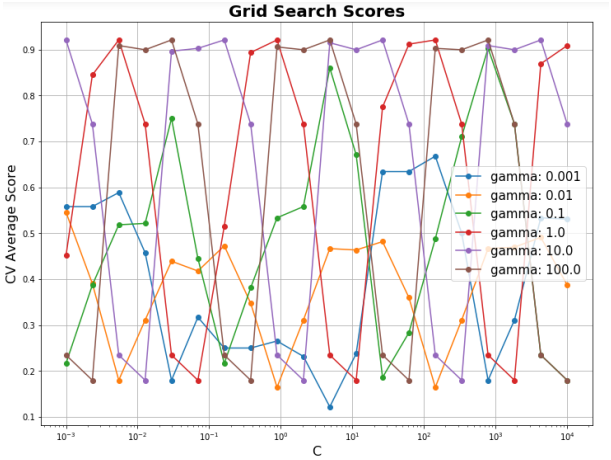

Fig. 16. Obtained average accuracy of 3-fold cross-validations for each combination of hyper-parameters using a RBF kernel for the Support Vector Machine model.

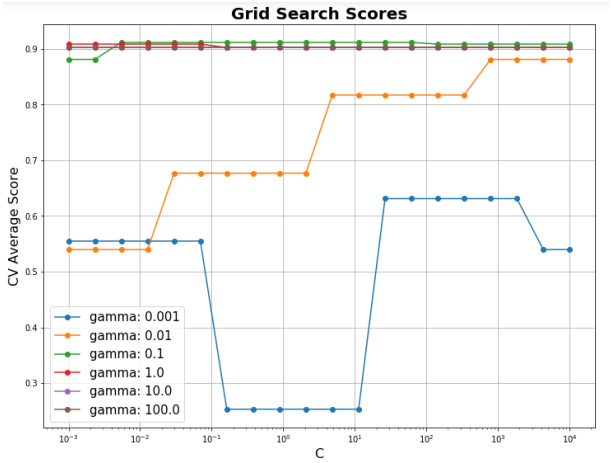

Fig. 17. Obtained average accuracy of 3-fold cross-validations for each combination of hyper-parameters using a linear kernel for the Support Vector Machine model.

TABLE I
SCORES

Feature Extraction	Accuracy	F1 Score	Time (s)
None	1.0000	1.0000	1176.0390
PCA	0.9634	0.9588	7.6650

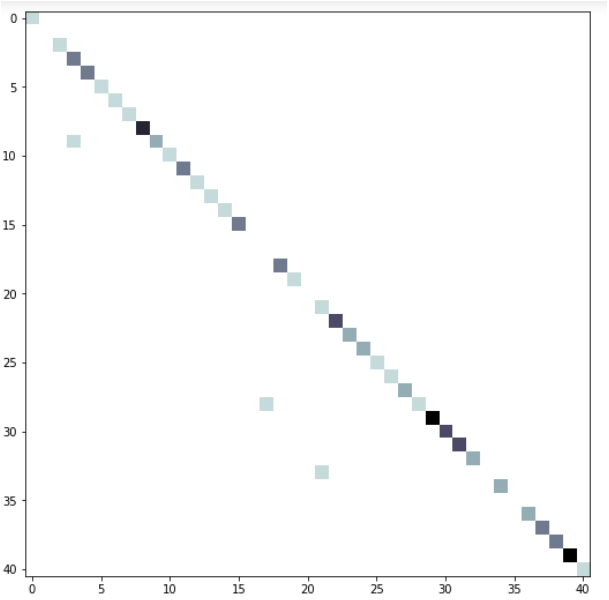

Fig. 18. Resulting confusion matrix using Principal Component Analysis for feature extraction.

VIII. Conclusions

Thanks to this project we can have a much better understand of the underlying foundations of the current State of the Art when solving Face Recognition problems. Approaches such as HMM and using Eigenfaces played a major role in the development of solutions in this field. In this project, we also had the chance to understand and study advanced mathematical concepts such as Eigenvectors and how they can be applied in Machine Learning and real world problems, by implementing and visualizing Eigenfaces. Albeit not implemented, thanks to this paper we learned more about Hidden Markov Models as seen in [1].

As to results, when picking a model we immediately spot Support Vector Machines constitute a great choice for pattern recognition even in complex problems as such. Furthermore, we can infer feature extraction drastically increases model performance at a very reasonable accuracy price. We also learned feature extraction from faces using Principal Component Analysis, as well as using the Grid Search algorithm to fine tune machine learning models.

References

[1] O. Sakhi. Face Recognition Guide using Hidden Markov Models and Singular Value Decomposition. 2012.
[2] M. Turk and A. Pentland. Eigenfaces for recognition. J. Cognitive Neurosci., vol. 3, pp. 71–86, 1991.
[3] F. S. Samaria and A. C. Harter. Parameterization of a stochastic model for human face identification. Proc. 2nd IEEE Wkshp. Applicat. Comput. Vision, Sarasota, FL, 1994.
[4] Guo, Guodong Li, Stan Chan, Kapluk. (1970). Face Recognition by Support Vector Machines. Proceedings of the Fourth IEEE International Conference on Automatic Face and Gesture Recognition.
[5] S. Lawrence, C. L. Giles, Ah Chung Tsoi and A. D. Back, "Face recognition: a convolutional neural-network approach," in IEEE Transactions on Neural Networks, vol. 8, no. 1, pp. 98-113, Jan. 1997, doi: 10.1109/72.554195.

[6] Shukla, Ratnesh. (2019). Machine Learning approaches for Face Identification Feed Forward Algorithms.
[7] Balaban, Stephen. (2015). Deep learning and face recognition: the state of the art. 94570B. 10.1117/12.2181526.
[8] H. Zulkifli (2018). Understanding Learning Rates and How It Improves Performance in Deep Learning
[9] Pedregosa, F., Varoquaux, Ga"el, Gramfort, A., Michel, V., Thirion, B., Grisel (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12(Oct), 2825–2830.
[10] G. Sanderson (2017). But what is a neural nework?.
[11] J. Nogueira, T. Melo (2020). Neural Networks for Sign Language Recognition.
[12] P. Gupta (2017). Decision Trees in Machine Learning
[13] Rodríguez, Juan Pérez, Aritz Lozano, J.A.. (2010). Sensitivity Analysis of k-Fold Cross Validation in Prediction Error Estimation. Pattern Analysis and Machine Intelligence, IEEE Transactions on. 32. 569 - 575.
[14] E. Lutins (2017). Grid Searching in Machine Learning: Quick Explanation and Python Implementation.

REFERENCES