Faculty of Informatics and Computer Science

Artificial Intelligence

EPIC: Egyptian Personal Images Classifier

By: Ashraf Adel Gamil Haress

Supervised By Associate Professor Nahla Barakat

June 2023

Abstract

Managing one's photos is often a monotonous task; one must browse through images in their storage device to distinguish personal images (PIs) from irrelevant ones (IRIs) and keep only the former. To avoid this repetitive process, this project proposes an efficient method utilizing deep and/or other machine learning techniques that will automatically classify the images in question into relevant and irrelevant ones. After running experiments with various variations of Convolutional Neural Network (CNN) architectures, mainly CNN with metadata (MCNN), Hierarchical CNN (HCNN), and XGBoost trained on images' feature representations extracted from a CNN (XCNN), we found that HCNN achieved the best results, with an average f1-score of 0.871, and is therefore considered as the final methodology used for the presented task and given the name of *EPIC: Egyptian Personal Images Classifier*.

Turnitin Report

Ashraf 196280 - EPIC - GP Dissertation

ORIGINALITY REPORT

3%	3%	1%	1%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

MATCH ALL SOURCES (ONLY SELECTED SOURCE PRINTED)

1%

★ "Artificial Intelligence Research and Development", IOS Press, 2022

Publication

Exclude quotes	On	Exclude matches	Off
Exclude bibliography	On		

turnitin

Digital Receipt

This receipt acknowledges that Turnitin received your paper. Below you will find the receipt information regarding your submission.

The first page of your submissions is displayed below.

Submission author:	Ashraf 196280
Assignment title:	Graduation project report-AI Part 1 (Moodle TT)
Submission title:	Ashraf 196280 - EPIC - GP Dissertation
File name:	243_Ashraf_196280_Ashraf_196280_-_EPIC_-_GP_Dissertation_...
File size:	8.36M
Page count:	126
Word count:	20,238
Character count:	122,668
Submission date:	10-Jun-2023 11:55PM (UTC+0200)
Submission ID:	2113272059

Acknowledgment

I would like to take this opportunity to express my heartfelt gratitude to the individuals and organizations who have contributed to the successful completion of my graduation project's dissertation. First and foremost, I offer my sincerest thanks to God for providing me with the strength, wisdom, and guidance throughout this journey. Your blessings and grace have been instrumental in my achievements. Secondly, I am heavily indebted to Prof. Nahla for her unwavering support, guidance, and invaluable expertise. Her mentorship and encouragement have played a pivotal role in shaping my research, refining my ideas, and pushing me to exceed my own expectations. I am also indebted to Dr. Walid Hussein, who guided me and patiently answered all of my questions throughout the data preparation phase. I would also like to extend my gratitude to my family for their unconditional love, unwavering belief in me, and constant encouragement. Their support has been the foundation of my success, and I am immensely grateful for their sacrifices and understanding throughout my academic journey. Then, I would like to thank my friends, mainly Mohamed Negm, Hanin, Adam, and Ali, who helped me obtain academic images to be used in the dataset mentioned in section 3.1. Lastly, I would like to express my appreciation to all the developers of the third-party libraries that I utilized in my project, especially Mr. Rajiv Sarvepalli, who took the time and effort to explain to me his library “simple-hierarchy”, which will be mentioned later in section 3.2.3.

Table of Contents

List of Figures

List of Tables

1 Introduction

In this section, we will introductory aspects of the project including motivation and overview, problem statement, scope and objective, report organization, and work plan presented through a Gantt chart. Firstly, we aim to provide the motivation behind this study, highlighting the reasons why this research is significant and relevant. Secondly, we will outline the problem statement, identifying the specific issue or challenge that this report seeks to address. then, we will define the scope and objective of our study, setting clear boundaries and goals for our research. Next, we will discuss the organization of this report, providing an overview of how the information is structured and presented to ensure coherence and readability. Lastly, we will present a detailed work plan in the form of a Gantt chart, illustrating the timeline and sequence of activities to be undertaken throughout the research process, enabling efficient project management and execution.

1.1 Motivation and Overview

Ever since the inception of mobile storage devices, people have been filling their devices with personal images (PIs) that include family, friends, or selfies. In addition to PIs, these devices have also been cluttered with irrelevant images (IRIs) such as internet memes, and images celebrating a specific event like a holiday or a birthday, which we will refer to as occasional images (OIs). However, IRIs are subjective; one may see screenshots of chat messages as relevant, therefore PIs, while the other thinks they are IRIs. Therefore, we'll first subjectively define what constitutes as PI and IRI, then discuss the methodology that classifies images based on the established criteria.

1.2 Problem Statement

Given certain criteria that defines PIs and IRIs, correctly classify images so that they could be later lumped into one of the two categories.

1.3 Scope and Objectives

We will limit the scope of our project by establishing the following criteria for image classification:

1. Personal images (PIs) contain the following:
    a. Family
    b. Friends
    c. Selfies
2. Irrelevant images (IRIs) contain the following:

a. Internet memes
b. Social media posts
c. Chat screenshots
d. Occasional images. For example, celebrating a certain holiday or birthday.
e. Academic photos and screenshots

However, due to the difficulty of collecting private personal photos, we use only selfies in the PIs category, which will be discussed further in methodology and implementation sections. Moreover, the academic images are grouped into IRIs as we assume students would normally want to get rid of them once their academic semester is over. In any case, the objective of this project is to establish a methodology using Deep learning and/or machine learning techniques to automate the classification of images based on the specified criteria.

1.4 Report Organization (Structure)

In section 2, we will mention the related work done on subsets of the criteria used to classify images, namely face recognition and meme detection. In section 3, we will elaborate on the proposed solution; the detailed architecture related to pre-processing and classification of images. In section 4, we will mention how the proposed solution was implemented. In section 5, we will display the results of the trained models, analyse, and compare them. In section 6.1, we will give general recommendations based on our experiments and conclude our established methodology and any findings inferred while working on the project. In section 6.2, we will mention how the project contributes to the community of A.I developers. Finally, in section 6.3, we will state difficulties faced while implementing the proposed solution and suggest future work to solve these shortcomings.

1.5 Work Plan (Gantt Chart)

Figure 1 represents the projected Gantt Chart for the project which was made before working on the project's implementation.

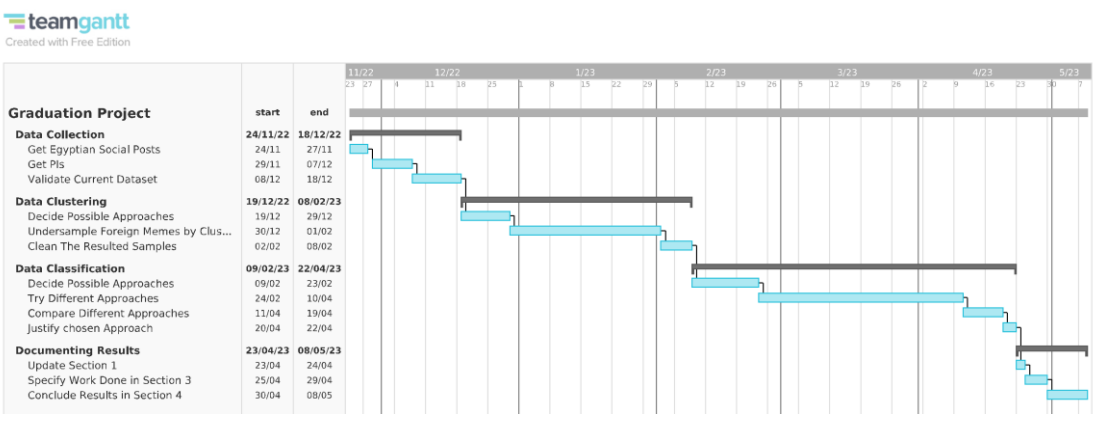


Figure 1. Gantt Chart for the PIC project.

2 Related Work (State-of-The-Art)

This section will review previous work done and the state of the art of deep and/or machine learning models related to the meme classification and facial recognition problem.

2.1 Background

Due to the exponential increase of online interactions between internet users, there has been rapid research developments on automatically analysing online content, especially memes sent on various social media platforms. However, since there is no standard format on how memes should look like, researchers re-define what constitutes a meme, based on the problem that they are aiming to solve. For example, Ferrara et al. [1] consider a meme as similar tweets (i.e., message streams) that are clustered together based on similarity, while Chang [2] only considers memes as visual images with superimposed text.

In general, there are common techniques used to solve online-content clustering and classification problems, which are relevant to the problem presented in this project and therefore will be discussed in the upcoming sections.

2.2 Meme Clustering

2.2.1 Memes as Features Extracted from Text-Only Posts

Ferrara et al.'s research in [1], which is one of the earliest research about memes, focused on clustering grass rooted memes, which are memes that have a role in orchestrated political campaigns, based on textual features such as the hashtag and the post's text but **not including the post's image**. The authors refer to these features as *protomemes*. In essence, their proposed clustering framework used similarity measures over proto-memes to aggregate them into broader memes. Figure 2 shows that these similarity measures are defined by considering the distance between the two protomeme's corresponding features.

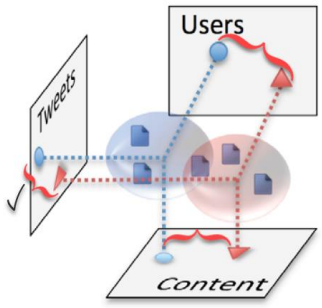


Figure 2. Projection of proto-memes based on their features that inform how they are similar. In the given example, the tweet content is the most similar aspect of the two proto-memes. Adapted from [1].

Moreover, they compared different strategies of combining different similarity measures, and found that the *pairwise maximization strategy* is the most effective strategy (i.e., obtained the best results statistically), where this strategy chooses the similarity measure with the highest value for every two compared proto-memes and based on this chosen measure, it decides the overall similarity between the two proto-memes.

Regarding the clustering algorithm, they compared *hierarchical clustering* with *K-means clustering* algorithms. Furthermore, to determine the similarity among two clusters, the authors adopted the *average-linkage* method, which is the average distance between all pairs, such that each pair consists of a proto-meme from the first and second cluster.

Hierarchical clustering is designed to span a range of granularities, where granularity means the level of details of each cluster, so whether it consists of tightly or broadly related proto-memes is based on changing the similarity threshold to produce varying number of clusters. Meanwhile, K-means clustering is more computationally efficient if the desired number of clusters is known in advance.

The result of this comparison shows that K-means performs better only when the number of clusters is relatively few (less than 100), while the hierarchical clustering algorithm has overall better performance over average and large number of clusters. Thus, they use hierarchical clustering for the rest of their experiments.

2.2.2 Memes as Features Extracted from Image and Text

The following sub-sections discuss two feature extraction methods based on both image and text of a meme.

2.2.2.1 Using DeepCluster

More recently, Chang [2] also analysed and clustered political memes vs non-political ones, but unlike Ferrara et al., he did so by using *DeepCluster* [3]. DeepCluster is a self-supervised architecture that does the following:

1. Take unlabelled images and augment them.
2. Use a Convolutional Network (*ConvNet*) architecture, for example AlexNet or VGG-16, to extract features from the augmented images. Noting that in this case, the author chose VGG-16
3. Use *Principle Component Analysis (PCA)* to reduce the dimensions of the feature vector.
4. Pass the reduced feature vector to *K-Means* clustering algorithm to assign each image (in feature vector form) to a cluster.

5. Generate *pseudo-labels* for each feature vector from these cluster assignments.
6. Train the ConvNet architecture to predict these clusters and update its performance by minimizing the multinomial logistic loss (i.e., cross entropy) using mini-batch gradient descent and backpropagation to compute the gradient.

Regarding the implementation of K-means, Chang set the number of clusters hyper-parameter (K) to 100 and used Euclidean distance to cluster the feature vector representation of the images. Based on the architectural setup mentioned above, he was able to visualise the difference between political and non-political (i.e., authentic) memes as shown in Figure 3.

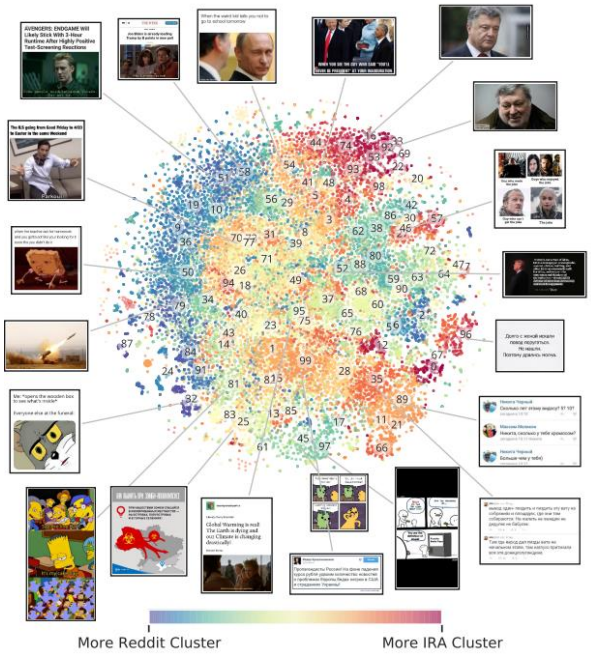


Figure 3. t-SNE projection of IRA (i.e., political) and Reddit (i.e., non-political) memes, where each colour indicates a cluster on the embedding space (i.e., feature vectors) learned from DeepCluster [2].

Regarding step 6. In the DeepCluster architecture, he used logistic regression to obtain a classification F1-score of 0.84.

2.2.2.2 Using pHash

Focusing on the nature of how memes propagate on social platforms, Zannettou et al. [4] grouped various types of memes (including political ones) into clusters to gain insights about what types of memes are present in each social platform, and the influence of such platforms on each other in terms of propagating memes. More specifically, the steps done to group images into clusters and then check which of these clusters have similar images to assign them to higher level groups are illustrated in Figure 4.

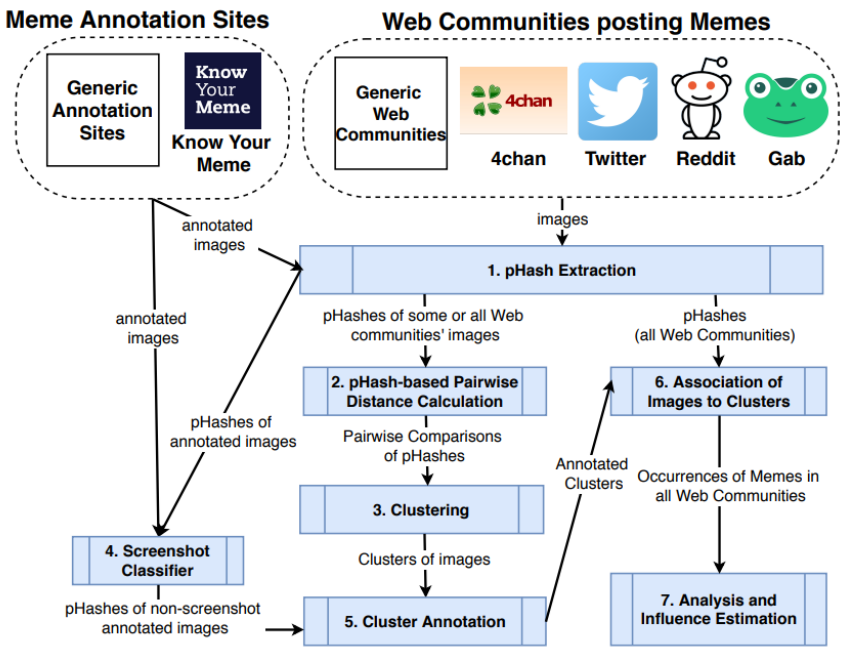


Figure 4. The processing pipeline used by Zannettou et al. [4].

They collected images of memes from various mainstream platforms and fringe web communities, then they used the *Perceptual Hashing (pHash)* algorithm [5] to obtain a fingerprint of each image so that visually similar images map to similar hash values by computing the discrete cosine transform to transform the image from spatial domain into frequency domain, where arithmetic operations are applied on the latter domain to obtain the pHash value. An example of pHashing is provided in Figure 5.

Figure 5. Variants of Smug Frog meme where the computed pHash values for these images are 55352b0b8d8b5b53, 55952b0bb58b5353, and 55952b2b9da58a53 respectively. Adapted from [4].

They then used a custom distance metric for the *DBSCAN* algorithm [6] [7] that is used to cluster the images. This metric uses the following features as similarity measures:

1. Perceptual: It is the similarity of images from a perceptual viewpoint, and it is calculated by getting the Hamming distance, which is the total count of 1 bit in the XOR result of the 2 medoid pHashed images of the clusters that are being compared.
2. Meme, culture, people: They refer to the meme's given name, associated culture, and people included in the meme respectively.

Noting that only the former feature is used as the distance metric if one or both medoid memes are not annotated from generic annotation sites like know your meme (KYM), while all features are used if both medoid memes are annotated.

After using DBSCAN on their dataset, they obtained clusters visualized in Figure 6.

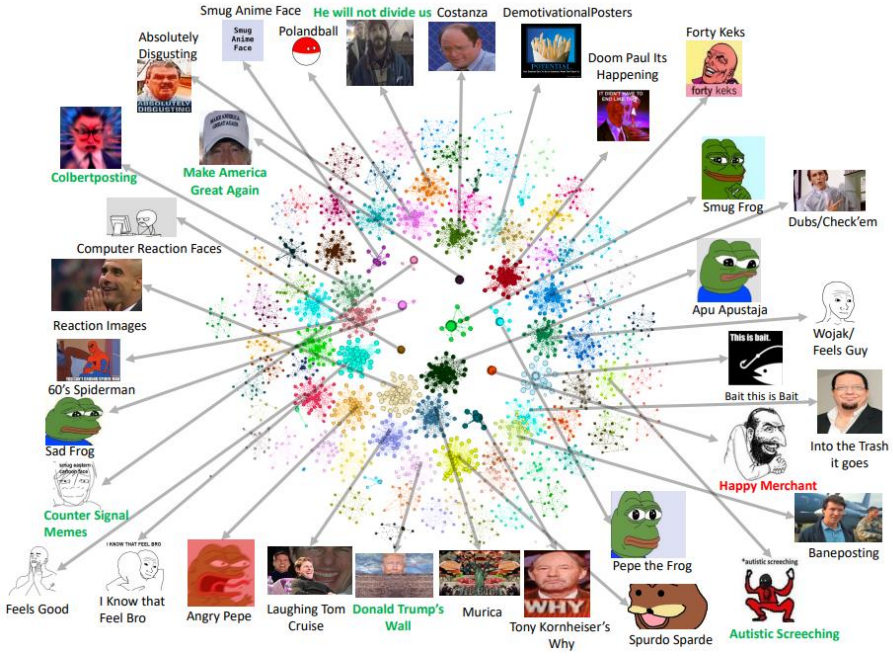


Figure 6. Visualization of the clusters from fringe web communities [4].

After obtaining the clusters, they found insights about the nature of memes and their propagation from these both fringe web communities and mainstream platforms. Finally, they used the Hawkes statistical model to see the social platforms responsible for making certain types of memes viral.

2.2.3 Memes as Features Extracted from Image Only

Onielfa et al. [8] were also interested in how memes propagate. Unlike Zannettou et al. [4], they focused on specific Instagram users' influence on other users, instead of entire web communities' influence. More specifically, Figure 7 illustrates the steps they used for creating a meme influence graph that they used to know the most influential Instagram users, where this information can be leveraged when selecting candidates for marketing campaigns using memes.

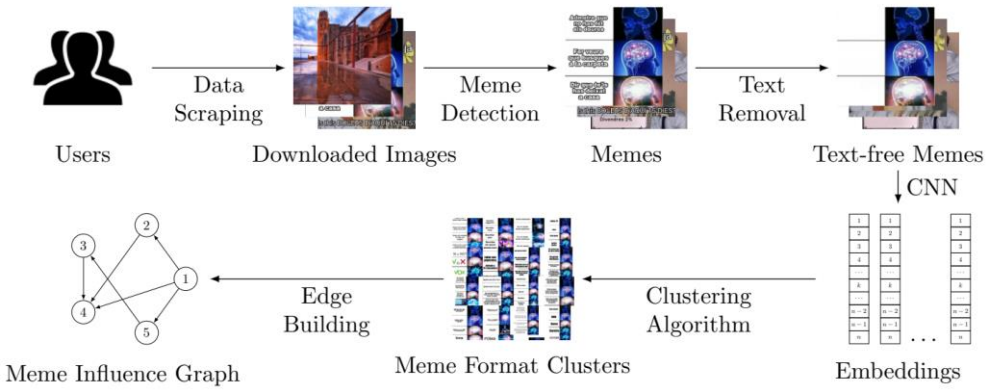


Figure 7. Flow diagram for the creation of a meme influence graph [8].

They first used Instagram's API to extract posts from 91 users and obtained around 457,101 images. After that, they used a *meme detection algorithm* to discard images with no text or with a large amount of text such that it mostly covers the underlying background image such that 342,984 images (75%) were classified as memes. The algorithm does the following steps:

1. Detect areas containing text using CRAFT text detector [9].
2. Compute text-to-image ratio.
3. If this ratio is not within manually-set lower and upper bounds, then the image has either only text or no text, so the algorithm terminates with the image not being a meme.
4. Otherwise, inpaint the image using Navier–Stokes algorithm [10] such that the text areas obtained from CRAFT are used as inpainting mask.
5. If the standard deviation of the inpainted image's grayscale values is lower than a manually-set threshold, then there was no content (i.e., meme template) left after removing the text, so the algorithm terminates with the image not being a meme.
6. Otherwise, the algorithm will have extracted the meme template and will terminate with the image being a meme.

Examples of inpainted images classified as memes or not is illustrated in Figure 8.

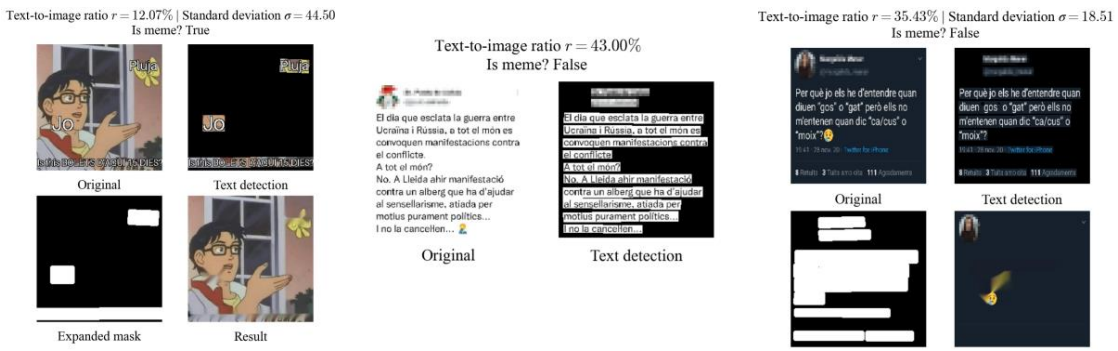


Figure 8. Left image classified as meme, while middle and right images are not. Adapted from [8].

They then used *VGG-16* [11] pre-trained with weights from the ImageNet challenge and its second-to-last fully connected layer to extract a feature vector of 4096 from the text-free meme (i.e., meme template).

After extracting the features, they applied PCA to reduce the dimensionality of these features to 1024, and then they used the resulted features as input to the DBSCAN algorithm which was able to group memes into 13,663 clusters containing 82,801 memes (24%) out of the 342,984 images, where the rest of images were determined by DBSCAN to be noise.

2.3 Meme Classification

Aside from Facebook's DeepCluster [3] classification step and the meme detection algorithm proposed by Onielfa et al. [8], there are other research that specifically tackled the problem of meme classification.

2.3.1 Using Visual Features on ML/NN Techniques

Wanting to identify an image as a meme, sticker, or non-meme, Perez et al. [12] tested *Histogram of Oriented Gradient (HOG)* and *ResNet* as feature descriptors to extract features from the image, and applied various classification models on these features, mainly decision tree, K-Nearest Neighbour (KNN), Support Vector Machine (SVM), and Neural Networks (NN). Figure 9 Shows examples of the 3 considered image classes.

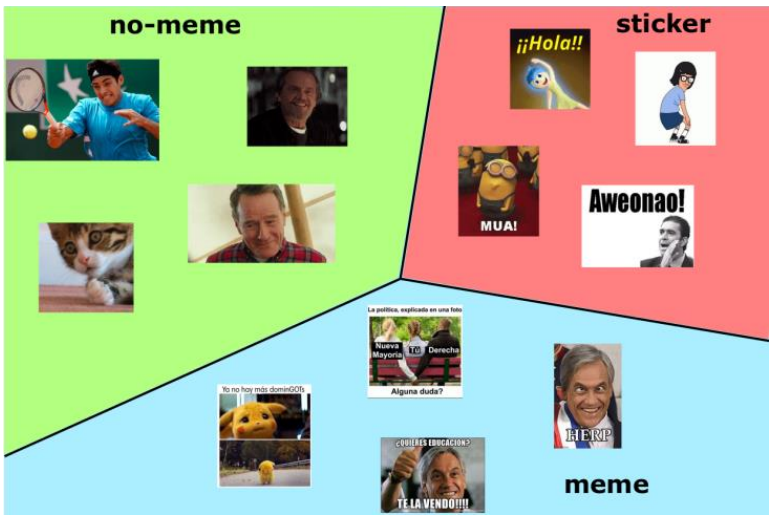


Figure 9. No-meme, sticker, and meme as classes of images [12].

Regarding the dataset, they had unequally distributed classes, so they performed under-sampling technique to equally distribute the classes as shown in Figure 10.

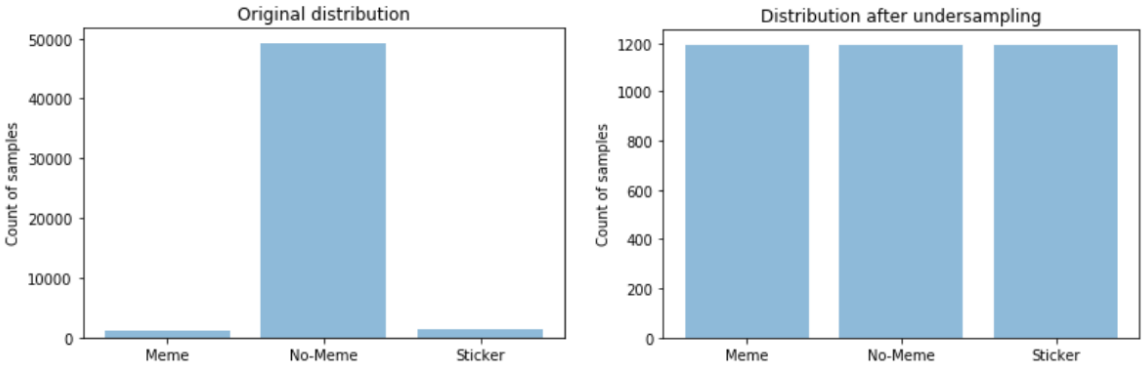


Figure 10. Left and right plots are the data distribution before and after under sampling [12].

Going back to the classification problem, after trying multiple combinations of feature descriptors and learning models, they found that ResNet and Linear SVM had the best results. Therefore, they were able to classify and obtain the images that were considered memes from the dataset from which they were able to implement a system for retrieving memes using textual queries.

2.3.2 Using Visual and Textual Features on Multi-Modal Models

Relying on both visual and textual elements of an image, Sharma et al., [13] have used a dataset that is created by downloading 20K images from public domains, then they differentiated between meme and non-meme images, where samples are shown in Figure 11.

Figure 11. Samples of non-meme and meme images. Adapted from [13].

After preparing the dataset, they did the following steps:

1. For the visual features, they tested both image processing and deep learning techniques on an SVM classifier:
    a. Image processing techniques: HOG, *colour histogram, Local Binary Pattern (LBP)*, *Scale Invariant Feature Transform (SIFT)*, *Haar technique* which achieved F1-scores of 0.8, 0.56, 0.49, 0.75, and 0.91 respectively.
    b. Deep learning techniques: pre-trained models for ResNet, *AlexNet*, *InceptionNet*, and VGG-16, where the last model achieved the highest F1-score of 0.94. Therefore, VGG-16 was subsequently used to get the visual-based features of an image.
2. For the textual features, they tested *N-Gram, Glove Embedding,* and *Sentence Encoder* and obtained F1-scores of 0.51, 0.9, and 0.95 respectively. They therefore used Sentence Encoder to get the textual-based features of an image.

As can be seen, they have already achieved good scores using an SVM classifier with one of the two modalities (visual or textual). However, due to the multi-modal nature of meme content, they believed that combining these feature descriptors using a multi-modal would yield even better results. They therefore tested the dataset on 2 multi-modal models: *Siamese Network (SN)* [14] and *Canonical Correlation Analysis (CCA)* [15].

Regarding Siamese Network (SN), it checks if two input vectors of the same modality are similar by passing each input vector to a deep learning model, such that the models are equivalent in terms of architecture and weights, and then measuring the similarity according to the Euclidean distance of the resulted 2 feature vectors. However, Sharma et al., [13] applied the theory of SN on two different models: the VGG-16 and Sentence Encoder models, as can be shown in Figure 12.

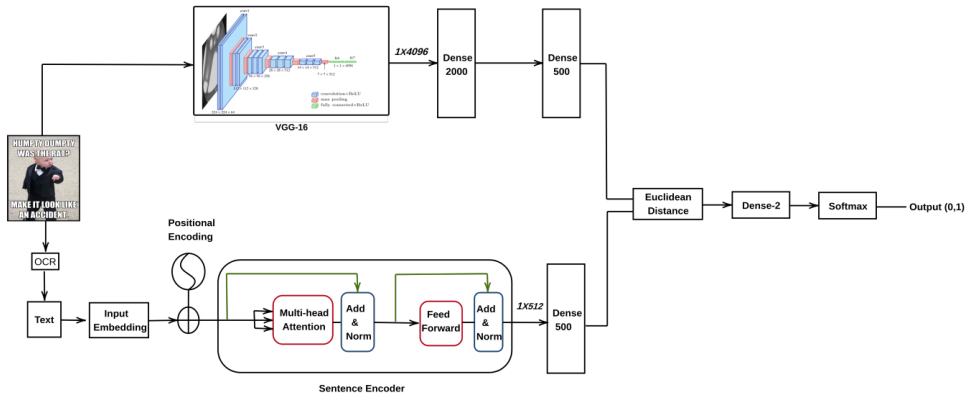


Figure 12. SN using VGG-16 and Sentence Encoder models, such that Dense 500 is applied on both models' outputs in order to calculate the Euclidean distance [13].

Regarding the Canonical Correlation Analysis (CCA) model and in the context of the given visual and textual feature vectors V and T, the CCA model will find linear combinations of V

and *T* that maximizes their Pearson correlation value. Then, the highly correlated canonical features are projected into a correlated semantic space and are considered the final feature vector that is passed to an SVM classifier to predict whether an image is a meme or not. The process of CCA is illustrated in Figure 13.

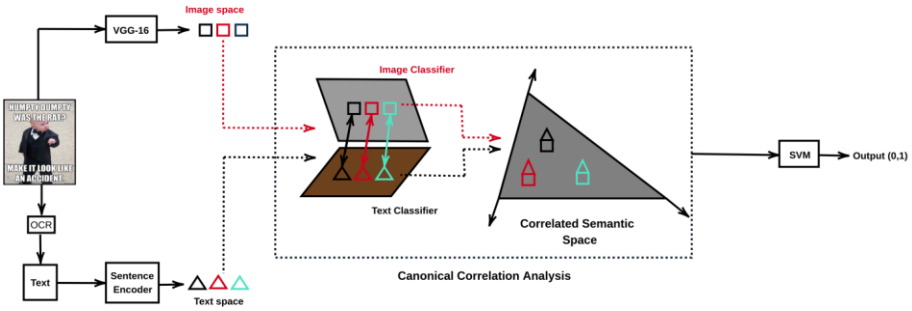


Figure 13. CCA using feature vectors from VGG-16 and Sentence Encoder models, such that the final highly canonically correlated features are passed to SVM classifier [13].

As for the results, SN and CCA obtained 0.98 and 0.99 F1-scores respectively, which proves, at least for the authors' case, that classifying memes using multi-modal features yields better results than classifying based on single-modal features.

2.3.3 Using Hierarchical Image Classification on Multi-Modal Models

While all the aforementioned research is about detecting memes vs non-memes, Das and Mandal [16] focused on memes only, but they also classified the connotation of different types of memes as shown in Figure 14.

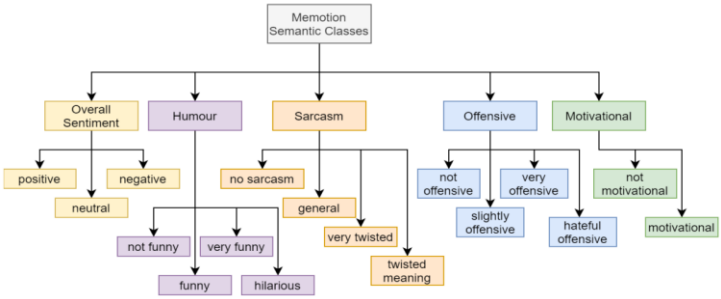


Figure 14. Semantic class hierarchy [16].

Specifically, they accomplished 3 tasks:

1. Task A: To classify if a meme is positive, neutral, or negative (i.e., "Overall Sentiment" sub-labels).
2. Task B: classify humour type (i.e., the other 4 higher labels presented in Figure 14)
3. Task C: classify the scale of each type of the labels from task B (i.e., the sub-labels in Figure 14)

Therefore, they applied a hierarchical classification architecture, such that its structure is expressed in Figure 15.

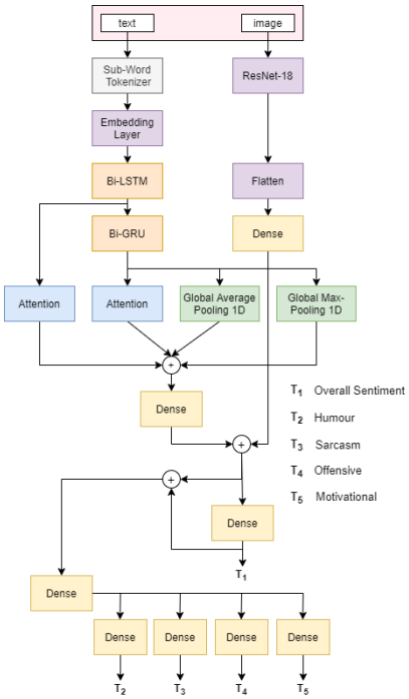


Figure 15. Multi-Modal Architecture [16].

For image feature extraction, they first resized the input image, which is from the memotion dataset on Kaggle, into 224x224 sized images. These images are used as input to ResNet for feature extraction. Meanwhile, a recurrent deep neural network (DNN) model consisting of bidirectional *Long-Short-Term-Memory (LSTM)* and *Gated Recurrent Unit (GRU)* is used for text feature extraction. For the text to be used as input in the DNN model, it went through the following steps:

1. Pre-process the text (e.g., converting text to lowercase).
2. Get an initial sub-word token and its embedding by feeding the text into a sub-word tokenizer.
3. Get the vector of sub-word embedding for each token using *SentencePiece Processor* [17].
4. Generate a more detailed embedding vector for each sub-word token by feeding the vectors obtained in the previous step into another trainable embedding layer. That generated vector is now used as input in the DNN model.

The text feature vector resulted from the DNN model is then passed to global average and max pooling layer to get a reduced text feature vector that is finally passed by a dense layer to obtain a further dimensionally reduced text feature vector.

After getting both visual and textual feature vectors, they were concatenated to get an aggregated feature vector that will be used for the multi-label hierarchical meme classification, by predicting the overall sentiment of the meme. That prediction is then concatenated with the aggregated feature vector, using a dense layer with ReLU activation function, to possibly obtain other labels and their fine-grained counterparts using dense layers with sigmoid activation functions for each of the other labels. Refer to Figure 14 and Figure 15 to review the labels and their respective dense layers.

Finally, the model got the following highest scores on tasks A, B, and C respectively:

1. Macro F1-scores: 0.3488 (SE-ResNet18), 0.5112 (ResNet34), 0.3240 (ResNet34)
2. Micro F1-scores: 0.5022, 0.6685, 0.4402 (all with ResNet18)

Where the authors assumed data imbalance and intraclass correlation among fine grained class labels was the cause of poor performance results, especially for task C.

2.4 Face Recognition

Du et al. [18] have done an intensive survey over the methods of each step towards face recognition (FR). However, in this project, we'll mention only the state-of-the-art models of the face representation step, which is the last step that recognizes faces based on the extracted discriminative features from the image pre-processed from previous steps of the FR steps-pipeline.

Out of the models used for the FR task and based on overall performance on the datasets presented in [18], the *GroupFace* [19] model with backbone of ResNet-100 obtained the best scores between the classification models, while *VGG Face* [20] and *GridFace* [21] achieved the best scores between the embedding models, and *AFRN* [22] had the best scores between the hybrid models. Noting that each subcategory of models means the following:

1. Classification: Considers the face representation learning as a classification task.
2. Feature Embedding: Optimizes the feature distance according to the label of sample pair.
3. Hybrid: Applying the above two subcategories together as the supervisory signals.

2.5 Analysis of the Related Work

Table 1 represents a summary of the discussed methodologies in section 2 and their performance.

Task	Model Type	Feature Extraction Method	Dataset	Performance/Results
Meme Clustering	HC [1]	Proto-memes	Twitter: 5.5K tweets (excluding image)	LFK-NMI better than K-Means when #Clusters ≥ 100
	K-Means			LFK-NMI better than HC when #Clusters < 100
	DBSCAN [6] (Used by [4])	pHash	#pHashes: Twitter: 74M Reddit: 30M 4Chan: 3.6M Gab: 193K	Noise / non-noise Ratio[1]: Twitter: - Reddit: 0.64 (19.2M/30M) 4Chan: 0.63 (2.27M/3.6M) Gab: 0.69 (133.17K/193K)
	DBSCAN [6] (Used by [8])	VGG-16	Instagram: 343K Memes	Noise / non-Noise Ratio: 0.76 (260.7K/343K)
Meme Classification	Meme Detection Algorithm [8]	None	Instagram: 457K Images	Evaluation Results: - Classification Results: 342.75K memes, 114.25K non-memes
	Linear-SVM (Used by [12])	ResNet-152	Twitter Images: 1.2K Memes 1.4K Stickers 49.3K non-Memes	Average F1-score: 0.73
	SN [14] (Used by [13])	IF: VGG-16 TF: Sentence Encoder	Flicker8K & Twitter Images: 7K Memes 7K non-Memes	F1-score: 0.98
	CCA [15] (Used by [13])			F1-score: 0.99
Multi-Label Hierarchical Meme Classification	Multi-Modal Architecture of CNN on Aggregated Features [16]	IF: ResNet (SE18, 18, 34) TF: SentencePiece Processor [17] then DNN of LSTM & GRU	Memotion Dataset: 7K Memes [23]	Average-Macro F1-score: 0.3946 Average-Micro F1-score: 0.53697
Meme Clustering & Classification	DeepCluster [3] (Used by [2] with LR)	VGG-16	IRA Memes: 26K Reddit Memes: 26K	Clustering Evaluation: - Classification F1-score: 0.84
FR: Classification	GroupFace [19]	ResNet-100	10 Face Datasets [18]	Avg Acc: 97.265%
FR: Embedding	VGG Face	CNN-36	Datasets 1-3, 6	Avg. Acc.: 84.84%
	GridFace	GoogLeNet-22	Datasets 1 and 6	Avg. Acc.: 97.65%
FR: Hybrid	AFRN	ResNet-101	Datasets 1, 4-10	Avg. Acc.: 95%

Table 1. Summary of the Related work's Architectures, where HC: Hierarchical Classification, "-": the information was not provided by the author, "IF": Image features, "TF": Text Features, "FR": Face Recognition

[1] Noise refers to the percentage of images not clustered. Zannettou et al. [4] state that these images are "one-off" images that are not considered memes.

It can be shown that the evaluation metrics for the clustering models are not consistent, as the authors were focusing on inferences from the generated clusters. Meanwhile, above-average scores for classification models were achieved, except for the multi-modal architecture. However, it is important to state that research on hierarchical meme classification is not as extensive as research on single-level meme classification. Moreover, it appears that there are no tasks related to classifying memes from other classes under the context of relevancy, like what we mentioned in section 1.3), and there are also no models trained on Egyptian memes specifically. Therefore, this project aims to fill these research gaps.

3 Methodology

This section will discuss the general pipeline used to reach the objective of the project.

3.1 Dataset Preparation Phase

This sub-section discusses how the dataset was obtained, analysed, and pre-processed.

3.1.1 Data Distribution

We show the image classes considered for the project, which include the image categories mentioned in section 1.3, in Figure 16Figure 15 as folder structure and Figure 17 as visualization plot of the data distribution.

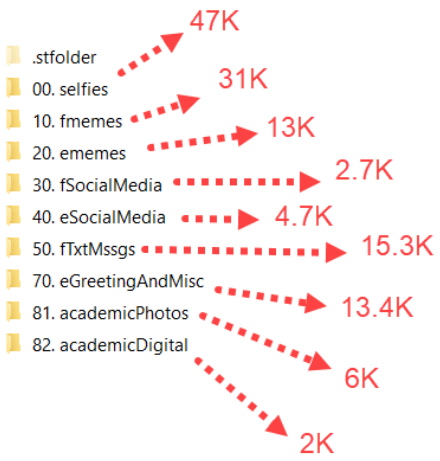


Figure 16. Classes of the dataset inside “dataset” folder.

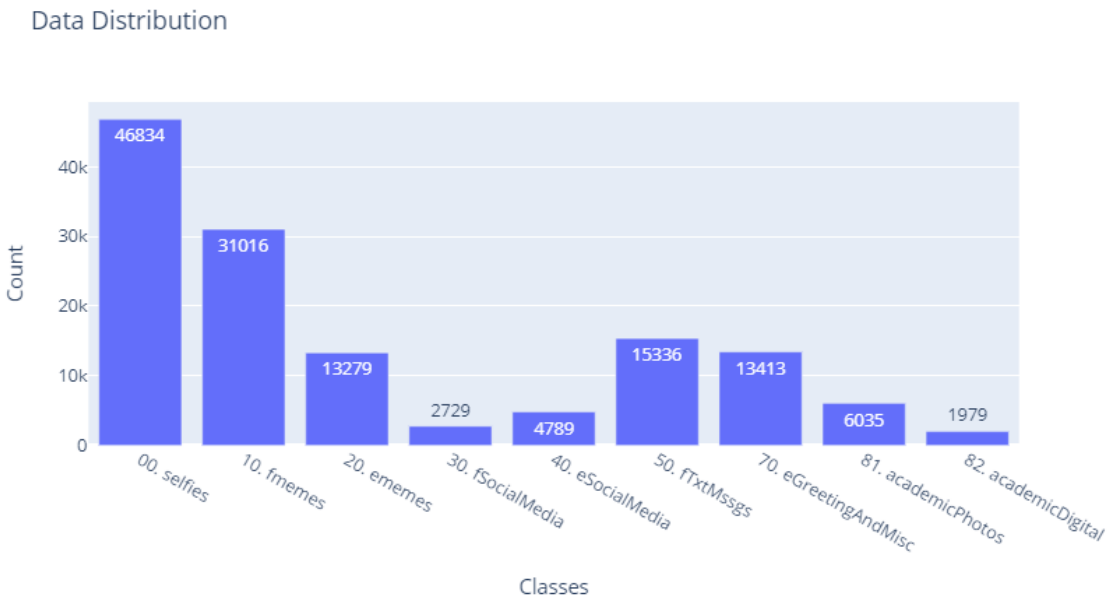


Figure 17. Visualizing the distribution of classes.

We clearly have an imbalanced dataset problem. However, the main problem does not arise from the sparsity of data in a specific class, but rather from possible overlapping of samples between different classes [24], which are also called misplaced labels. The reason of why such imbalance occurs will be mentioned for each imbalanced class in section 3.1.2. Moreover, the indexing of each class seen in Figure 16 was to act as a code for the class name for abbreviating it, but in actuality, the classes were given labels 0 to 8 respectively when loading the images. As a side note, more details about Syncthing application are mentioned in the Appendix section.

3.1.2 Data Collection

The dataset used in this project is a combination of benchmark datasets obtained online and datasets manually scraped and prepared to suit the need of the project.

3.1.2.1 Benchmark Data

The classes "selfies" and "fmemes" (foreign memes) were obtained from the University of Central Florida [25] and Kaggle [26] respectively.

3.1.2.2 Scraped Data

All the other 9 classes were scraped manually through different methods:

- "ememes" (Egyptian memes) and "eSocialMedia" (Egyptian social media) were obtained by writing functions for crawling on specific Facebook pages and public groups related to memes [27]–[31]. In addition, some images were obtained by scraping an image album on Imgur [32].
- "fSocialMedia" (foreign social media) and "fTxtMssgs" (foreign text messages) were obtained by scraping images from Facebook [33] and Twitter [34] subreddits for the former class and chat-screenshots subreddits [35]–[37] for the latter class, where Pushshift's API [38] was used in both cases. It is important to note that initially, the Facebook and Twitter images were in their own separate classes "fFbPosts" and "fTwtrPosts" respectively, but due to the data cleaning process done on these images, their total number of samples massively reduced to the point where it was more sufficient to group them into one class "fSocialMedia".
- "eGreetingAndMisc" (Egyptian greeting and miscellaneous images) were obtained by writing functions for crawling on google search engine, specifically, the images tab.
- "academicPhotos" (taken by phone) and "academicDigital" (taken by mobile or desktop screenshots) were obtained from various university colleagues who volunteered to hand off academic images which they took while studying.

Naturally, the images were all in one folder at first, and cleaning was required to separate them into the aforementioned classes.

3.1.3 Data Analysis and Cleaning

Figure 18 illustrates a sample of the images in all of the 9 classes as 9 by 7 grid, where each row represents a class's samples sorted based on appearance on Figure 16 and Figure 17. The images underlined in red are misplaced images, the images underlined in purple are correctly in their class but could confuse the model because of a visual resemblance to another class or high dissimilarity between its neighbouring samples of the same class, while images underlined in both colours are ones that aren't misplaced per se, but could logically be placed in another class as well. An example of this is the second image in the 5th row regarding "eSocialMedia" class; even though that image contains the Facebook page name and date of the post at the top of the image, the actual content of the image is an image of a chat history, which should technically allow this class to be in text message class (if it were not for the fact that "fTxtMssgs" is for foreign messages, not Arabic).

Now, since most of the dataset was scraped manually, there were naturally other errors and problems than the ones highlighted above, including corrupted images and images retrieved from dead URLs, where Figure 19 and Figure 20 illustrate these problems respectively. However, these problems were easily solved by attempting to open all images in the dataset and deleting ones that were not accessible in code, which solved the former problem, and since the images from Figure 20 are consistent, the latter problem was solved by deleting images that had very specific width and height combination (60 and 130 respectively).

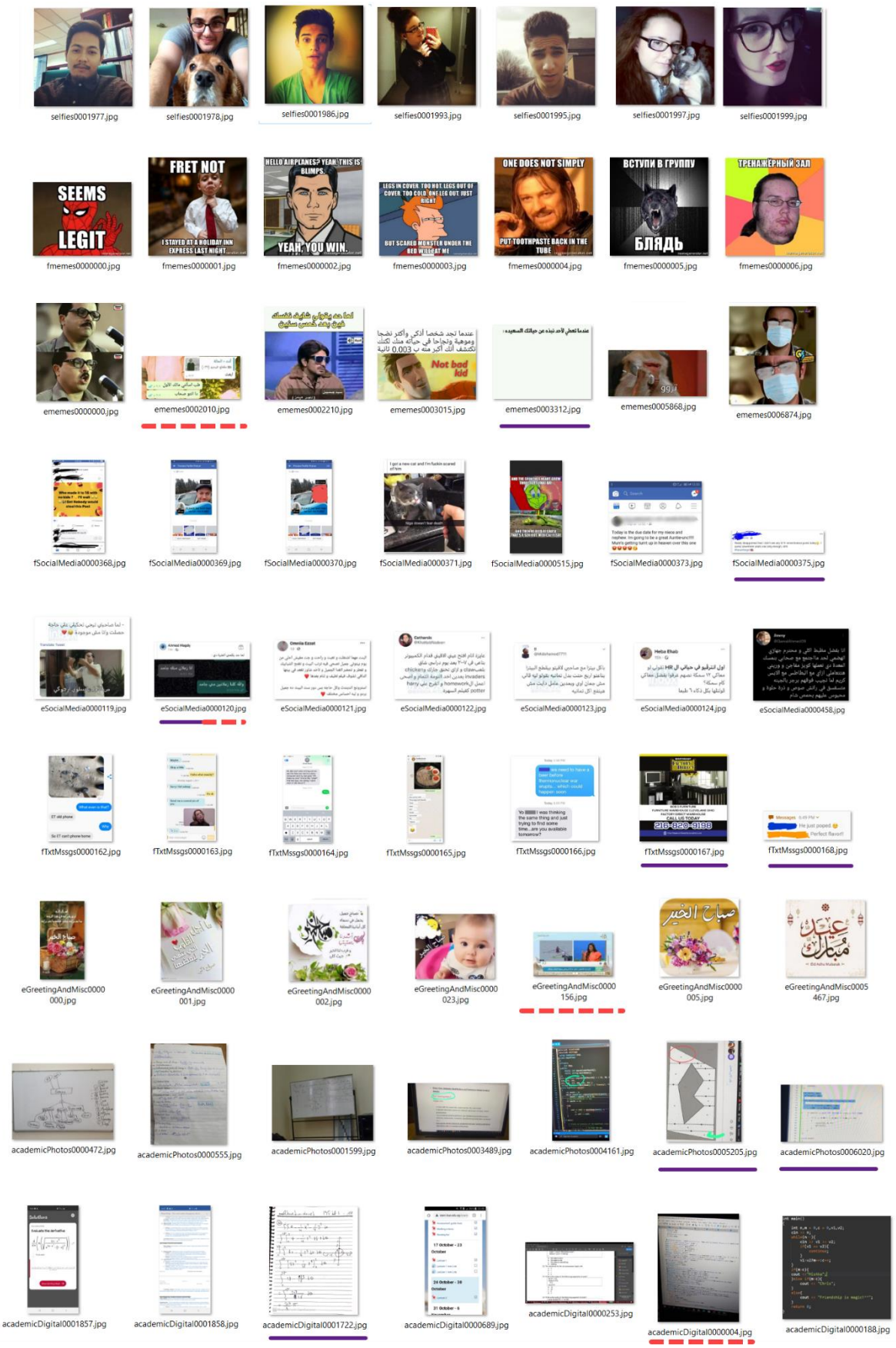


Figure 18. Nine rows each representing a class's samples. Red lines indicate mislabelled images, purple lines indicates outliers, both lines indicate the latter and valid labelling in another class.

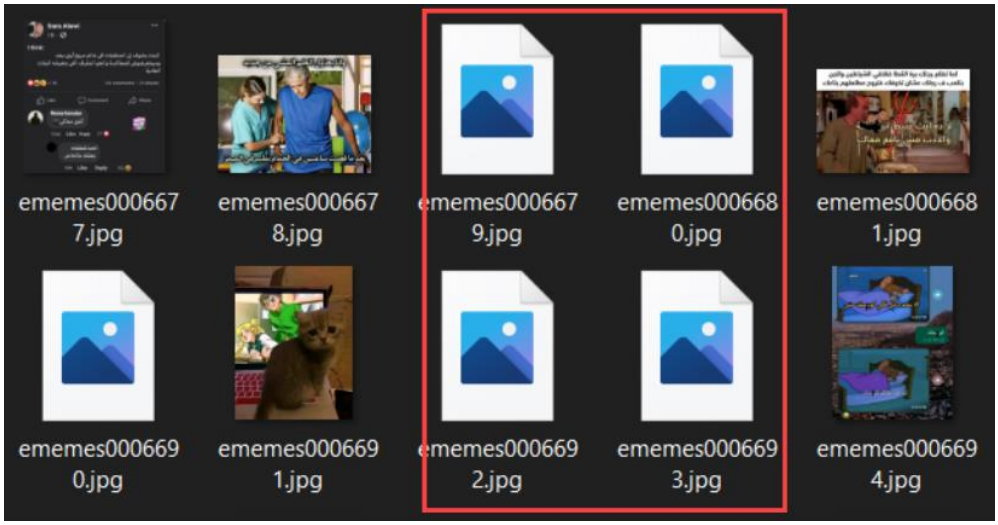


Figure 19. Example of corrupted images.

Figure 20. Images that were downloaded from dead URLs on Reddit.

However, the problems introduced earlier and illustrated in were harder to solve, as we had to manually go through the data and remove these images due to the absence of a consistent format in the images which posed these problems. However, functions were implemented which helped in sorting out the images in each class based on manually defined criteria which helps in weeding out the unwanted images. Specifically, we sorted the images by obtaining a certain scalar value which is calculated for each image based on logic which we can define. For example, for "v1" (version 1) of the dataset which is used throughout most of this project, we were able to separate "academicPhotos" from "academicDigital" images by assigning a scalar value for each image representing its maximum occurrence of a unique colour, so since digitally taken images are pure by nature (i.e., approximately no noise unlike photos taken by camera), the color white or black for example would identically appear frequently throughout the image, and thus these images will have a high score and will therefore appear at the top of the folder while photos taken by camera would appear at the end of the folder; allowing for easier removal. Figure 21, Figure 22, and Figure 23 show illustrations for the previous example, noting that the actual code logic behind the removal of these images is mentioned in the implementation section. There's also another example which uses the number to detected text boxes in an image to sort text messages from "eSocialMedia" and "eMemes" classes, but unfortunately, we did not screenshot the output of this sorting while cleaning those images.

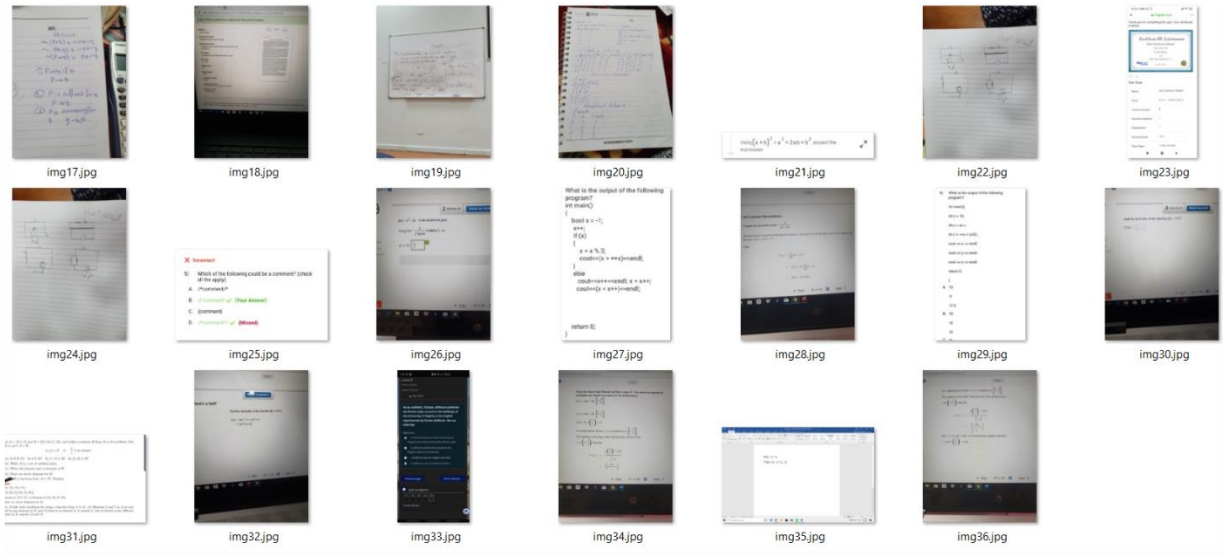


Figure 21. Folder “tmp” containing both “academicDigital” and “academicPhotos”.

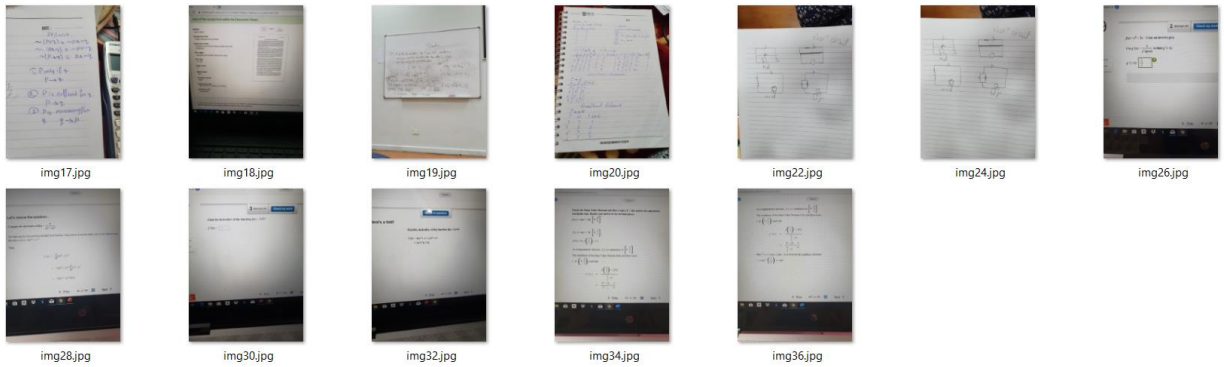


Figure 22. Folder “tmp” containing “academicPhotos” only.

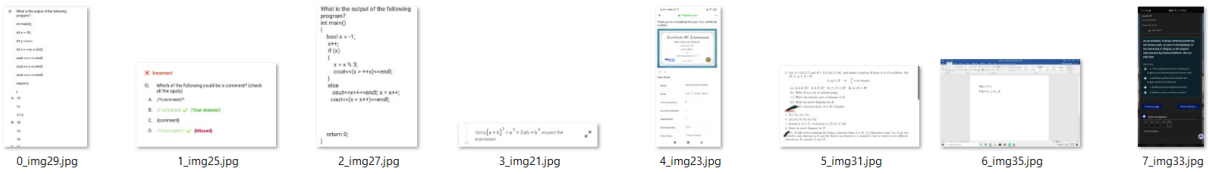


Figure 23. Folder “tmp2” containing “academicDigital” only, sorted from highest to lowest unique color frequency.

In addition to the problems above, there was an issue while scraping Facebook images for “eSocialMedia” and “eMemes” classes: Attempting to scrape images while not logged into Facebook, will eventually cause the image URLs’ queries used in each post scrolled by to be altered so that only a centre-cropped part of the image is loaded, which is done to reduce server overload on Facebook’s Content Delivery Network (FBCDN). This can be seen in Figure 24, thus images retrieved from such URLs are centre cropped as can be seen in Figure 25. Also, in Figure 26, and you can see that if you are logged into Facebook while scraping, then the URLs appear such that the whole image is loaded. But by the time this insight was discovered, several

of the Facebook pages that were used to fetch the images were unfortunately taken down, so the following cleaning procedure took place:

- For images in "eMemes", they were left out, as most of the time, the main essential content remained in the image, as can be seen in Figure 27.
- For images in "eSocialMedia" like the ones illustrated in Figure 25 and Figure 26, they were manually removed, which wasn't hard, as most of these images were found next to each other in the "40. eSocialMedia" directory.

```
'https://scontent.fcai2-1.fna.fbcdn.net/v/t39.30808-6/316428164_5951088171679463_1198526055046785576_n.jpg?stp=c157.0.206.206a_cp1_dst-jpg_p206x206&_nc_cat=103&ccb=1-7&_nc_sid=8bfeb9&_nc_ohc=EHJOTaD6z84AX8W1Dm2&_nc_ht=scontent.fcai2-1.fna&oh=00_AfC7EcfRZc_6m59w9bbMua7wzPu-dUFDeKSo8Nin-2rg5Q&oe=6386054D'

'https://scontent.fcai2-1.fna.fbcdn.net/v/t39.30808-6/316428164_5951088171679463_1198526055046785576_n.jpg?stp=cp6_dst-jpg&_nc_cat=103&ccb=1-7&_nc_sid=8bfeb9&_nc_eui2=AeGNRVuHNXKzqe-YZH7nEw6Rj5k1qtj2ihmPmTWq2PaKGbSWcW31noKzXBXE_ywfTeIF4ORkJQEhIaGX8rdyCP8&_nc_ohc=EHJOTaD6z84AX8W1Dm2&_nc_ht=scontent.fcai2-1.fna&oh=00_AfBGkO-6Cvs1bIVxRM1XVJwOkELKABqAcY86SStc0qLuSQ&oe=6386054D'
```

Figure 24. FB Image URL when logged out (top) vs when logged in (bottom).

Figure 25. Example 1 of cropped (highlighted) vs non-cropped images.

Figure 26. Example 2 of cropped vs non-cropped images.

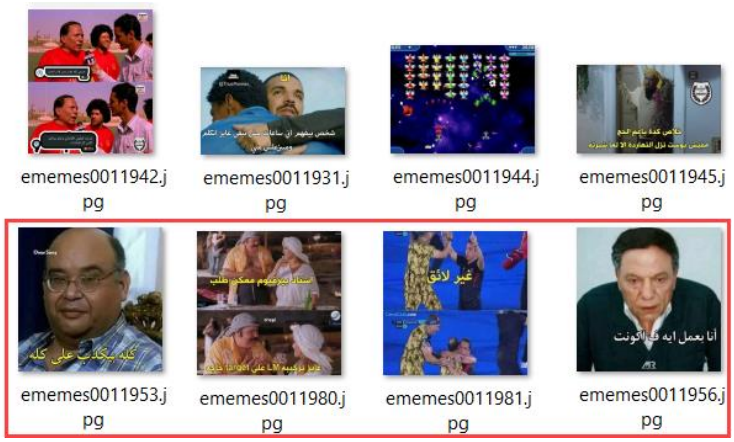


Figure 27. Example 3 of cropped (highlighted) vs non-cropped images.

On the more technical side of issues, after storing the scraped images, we've found that some of them were of type PNG even though they were explicitly stored as JPEG images, which is illustrated in Figure 28. This was fixed by using a third-party Python library which detects the type of the image without relying on its extension.

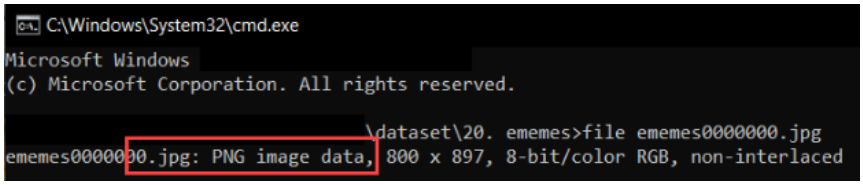


Figure 28. Example of image with ".jpg" extension that are actually PNG images.

Another minor issue that occurred was that some images in "academicPhotos" and "academicDigital" were incorrectly oriented as shown in Figure 29. Moreover, since all of these images were sent through social media and cloud platforms, most of them have the EXIF metadata stripped away from them, so there is no way to certainly know to which degree these images should be rotated to get the correct orientation. However, this is considered a minor issue since upon manual inspection, very few images suffer from this issue.

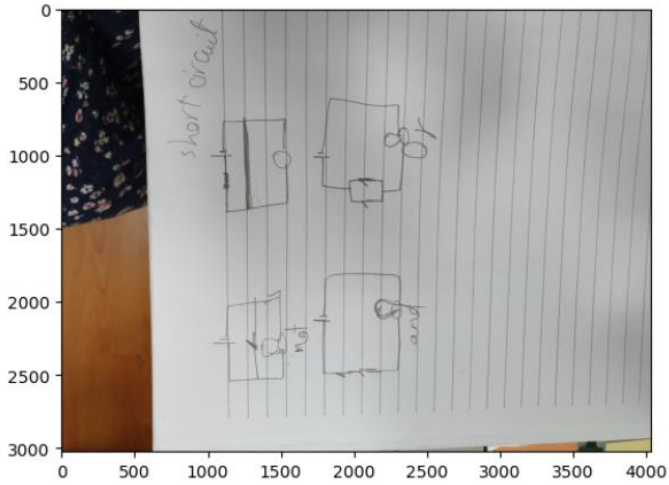


Figure 29. Example of image with wrong orientation.

Finally, after the entire cleaning process is done, we visualized the image's dimensions to get an estimate of the image size to be fixed while training the models. The results are shown in Figure 30, noting that in the references, you can access the notebook online to click on the legend and show or hide any of the classes [39]. In any case, due to the dataset size and time constraint of running these models locally, and due to the majority of the samples lying in the "selfies" class, which is already pre-processed and resized to 306x306, we will choose that size for resizing images that are fed into the deep learning models mentioned later.

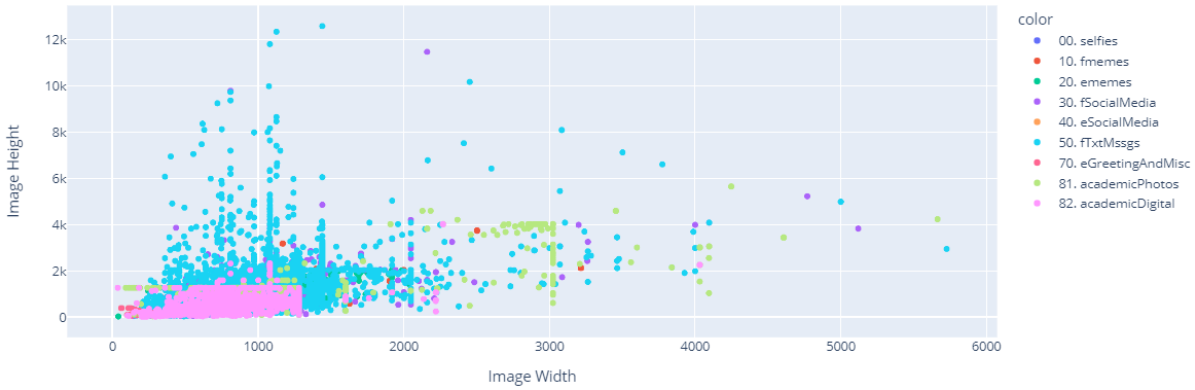


Figure 30. Example of image with wrong orientation.

3.1.4 Data Pre-processing

Very basic pre-processing steps have been made to the images, such as resizing the images into 306x306 using nearest interpolation method and randomly augmenting the dataset while training certain models. However, a crucial pre-processing step was done for "m02" (model version 2) variants, which is extracting certain features from each image and feeding these features as additional input to "m02.x". These features are explained in the upcoming sub-section.

3.1.4.1 Metadata Extraction

Figure 31 shows the list of features extracted from each image. Some of these features were directly extracted from the image's RGB channels, while other features required third-party libraries to be extracted.

```
Index(['file_name', 'relative_path', 'aspect_ratio', 'area', 'width', 'height',
       'dominant_color_1', 'color_to_image_ratio_1', 'dominant_color_2',
       'color_to_image_ratio_2', 'dominant_color_3', 'color_to_image_ratio_3',
       'dominant_color_4', 'color_to_image_ratio_4', 'dominant_color_5',
       'color_to_image_ratio_5', 'total_faces_count', 'faces_to_image_ratio',
       'text_to_image_ratio', 'lines_per_text_block',
       'total_lines_of_text_count', 'text_blocks_count', 'total_en_boxes',
       'total_ar_boxes', 'en_words_original', 'ar_words_original',
       'en_words_lemmatized', 'en_words_segmented_and_lemmatized',
       'ar_words_lemmatized', 'en_words_including_separators',
       'ar_words_including_separators', 'en_words_to_boxes',
       'ar_words_to_boxes', 'en_avg_score', 'ar_avg_score',
       'math_digits_or_symbols_count', 'class'],
      dtype='object')
```

Figure 31. Features of "imgsPropsv1.csv" extracted from the dataset's images.

Upon inspecting Figure 31, we'll notice that obviously not all features were used as input to the model, such as "file_name", "relative_path", and "class". However, there are other features that weren't used throughout this project due to time and resource constraints of integrating an NLP model with our chosen model (CNN). Succinctly, the features that were actually used by "m02.x" variants are within the range of features starting from "dominant_color_1" until "total_ar_boxes", and will be mentioned in section 3.2.2.

Some of these features are self-explanatory, such as "aspect_ratio", "area", "width", and "height", which were obtained by simply getting the dimensions of the image. Moreover, the colour-related features were obtained by analysing the RGB values of each image; "dominant_color_x" represent the top "x" dominant colours in an image, where a dominant colour means the colour with the highest frequency but represented as 24 bits. For example, if we consider Figure 30 an actual image, then its most dominant colour is white, which has RGB values of (255, 255, 255), so it is combined into a single 24 bit value of "11111111 11111111 11111111", which is stored as its base 10 representation, which is "16777215". Correspondingly, the second most dominant colour in Figure 30 is light blue, and so on. Regarding "color_to_image_ratio_x", it refers to the ratio of the "x" most dominant colour over the total image area, so if we take Figure 30 as the same example as before, then by eye-balling, the ratio of white colour over image area is approximately the third of the image.

Next, we have the face-related features "total_faces_count" and "faces_to_image_ratio", which are self-explanatory, and are obtained by passing each image to a pre-trained face detection model called RetinaFace [40], which returns the bounding boxes of each face in an image.

Finally, we have the text-related features, which are obtained by using passing each image to an Optical Character Recognition (OCR) library called Paddle OCR [41] which returns a text

boxes around each detected word along with a confidence score, as well as the detected word itself. Most of the features are self-explanatory, except for a few, namely the following:

- "lines_per_text_block": "total_lines_of_text_count" / "text_blocks_count"
- "text_blocks_count": The number of text blocks. A series of words are considered a "text block" if their bounding boxes are sufficiently close to each other vertically and horizontally within the image.
- "en" and "ar": English and Arabic respectively
- "en_words_segmented_and_lemmatized": A string of English words, where each English word is first segmented, lemmatized, then added to the string of words. A segmented word means a word that was possibly a combination of multiple words then got segmented into multiple words. For example, "Helloworld" is segmented into words "hello world" using a third-party library [42].
- "en_words_including_separators": A string of English words including separators ("||") and ("|||") which were used to indicate the start of a new line and block of text respectively.
- "en_words_to_boxes": A dictionary mapping each detected word to all the boxes that bounded that word (in case that word occurred more than one time in the image).

3.2 Model Development Phase

After the dataset have been prepared, we tried multiple deep learning (DL) models on it to see which of them achieved the best results. The following sub-sections describe the general architecture of each of the attempted models, ordered by the model versions mentioned in the code[1]. As a side note, whenever a question mark "?" appears in any of the architecture figures, then this refers to the batch size, which is changed based on the version of the trained model.

3.2.1 VCNN: Vanilla CNN

Convolutional Neural Networks (CNNs) are a type of deep learning architecture specifically designed for analysing visual data, such as images, and are composed of convolutional layers which perform local operations, called convolutions, on patches (small regions) of the input image to generate feature maps, which are relevant features of the image, then these are passed to pooling layers which aggregate these features to obtain less but more important features a smaller dimensional space which are finally passed to a fully connected (dense) layer in order

[1] However, the HCNN was actually the last model that we tried to implement and test, not the XCNN, due to the complexity of this model both in coding and training time.

to perform the required task (classification or regression). Figure 32 shows the architecture of the base (i.e., vanilla) CNN model chosen in all "m01.x" model versions.

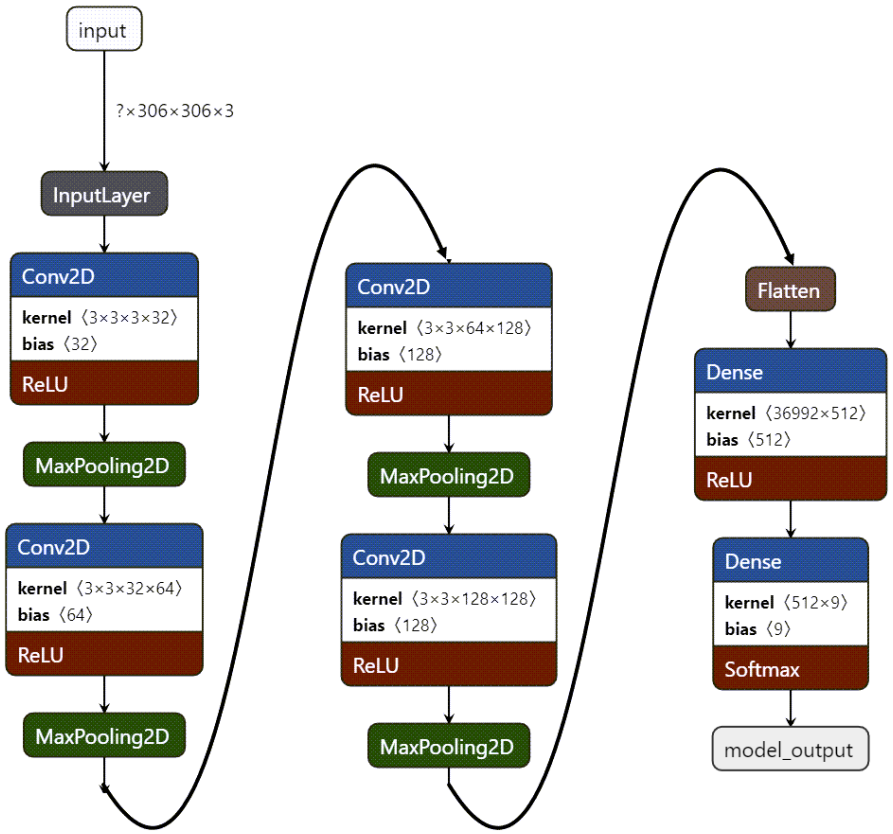


Figure 32. VCNN Architecture.

The parameters chosen for the architecture of the model are reasoned in the following way:

- The number of output feature maps per convolutional layer (i.e., 32, 64, 128, 128) are to capture varying levels of abstraction within the image; by sequentially increasing the feature map size, we're allowing the network to capture increasingly complex and high-level features by learning hierarchical representations, where lower-level features are combined to form higher-level abstractions. Moreover, we didn't reverse this order, because if we did, we'll risk overfitting the training data, as the model may memorize specific details instead of learning more generalizable features.
- The filter size (3x3) is a common choice for CNNs, since it is small enough to capture local features and patterns while being computationally efficient.
- Choosing rectified linear unit (ReLU) activation function introduces non-linearity, which will allow the model to learn complex relationships between the input data and the feature maps while remaining a computationally efficient function which helps alleviate the vanishing gradient problem.

- The choice of max pooling over other pooling strategies is solely based on the popularity of max pooling strategy [43]. Same logic applies for choosing "512" as the number of dense units in the dense layer.

Regarding the other hyperparameters used on "m01.x" model variants, they will be mentioned in section 4.2.1. Same logic applies to the rest of the models in the upcoming sub-sections. Finally, the model's trainable parameters were 19,185,865.

3.2.2 MCNN: CNN Incorporating Metadata

This model has the same base architecture as VCNN, but the differences lie in the structured input which as been added as well as the original 306x306 image input. This structured input is like a table of information (dataframe) which contains metadata about each image discussed in section 3.1.4.1. Figure 33 shows the architecture of MCNN version "m02". The main difference between this and other "m02.x" versions is that the others have different input shape for the structured table of information based on the number of features chosen, which is considered on the hyperparameters that will be discussed in section 4.2.2.

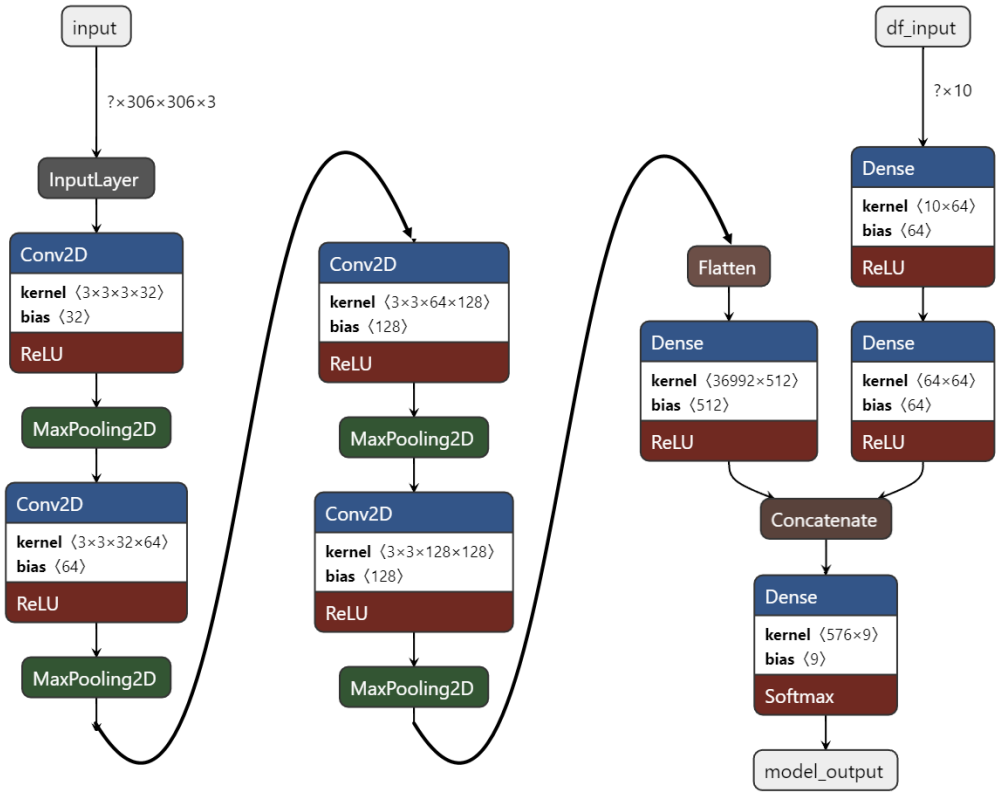


Figure 33. MCNN Architecture.

As we can see, the structured input is passed to two dense layers each of size 64, then the result is concatenated to the output of the VCNN architecture's 1st dense layer, then that

concatenation is passed to a final dense layer. Moreover, the model's trainable parameters were 19,191,305.

3.2.3 HCNN: Hierarchical CNN

HCNNs are designed to handle hierarchical structures in data. They are useful in multi-classification problems where the classes have a hierarchical relationship or a nested structure, which could be applied to our 9 classes in many forms; Figure 34 and Figure 35 are examples of possible hierarchy trees; they are named single and two level hierarchy respectively, as we must have at least one output layer, so we don't count that root output layer. However, given the constraint on time and resources, we've only had time to train one version of HCNN (called "m03") which used the single level hierarchy tree shown in Figure 34.

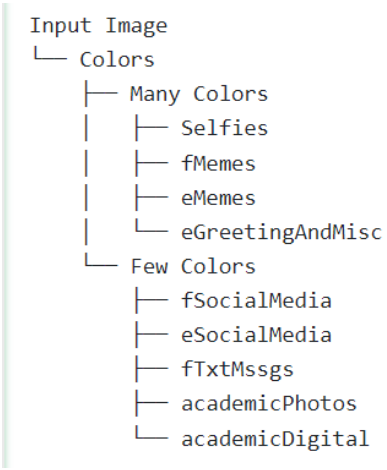


Figure 34. A single level hierarchy tree.

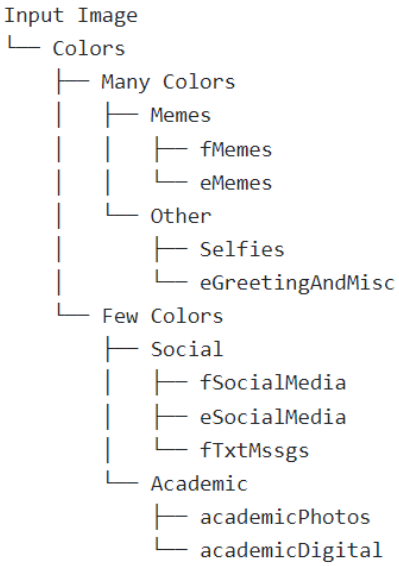


Figure 35. A two-level hierarchy tree.

Moreover, a third-party library called “simple-hierarchy” [44] was used to construct the basic structure of the HCNN shown as shown in Figure 36 and visualized in Figure 37 (1 depth layer) and Figure 38 (2 depth layers). Also, in case of confusion, consider Figure 39 which resembles Figure 38 but with arbitrary number of layers just to illustrate the concept of HCNNs.

```
HierarchalModelPL(
  (hierarchical_model): HierarchalModel(
    (base_model): Sequential(
      (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
      (1): ReLU()
      (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
      (4): ReLU()
      (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (6): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
      (7): ReLU()
      (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (9): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1))
      (10): ReLU()
      (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (12): Flatten(start_dim=1, end_dim=-1)
      (13): Linear(in_features=36992, out_features=512, bias=True)
      (14): ReLU()
    )
    (last_layers): ModuleDict(
      (('Colors', 2)): Sequential(
        (0): Linear(in_features=512, out_features=512, bias=True)
        (1): Linear(in_features=512, out_features=128, bias=True)
        (2): Linear(in_features=128, out_features=64, bias=True)
        (3): Linear(in_features=64, out_features=2, bias=True)
      )
      (('Classes', 9)): Sequential(
        (0): Linear(in_features=640, out_features=512, bias=True)
        (1): Linear(in_features=512, out_features=128, bias=True)
        (2): Linear(in_features=128, out_features=64, bias=True)
        (3): Linear(in_features=64, out_features=9, bias=True)
      )
    )
  )
)
```

128 + 512 = 640

Figure 36. HCNN Architecture: depth of 1 and visualized by code.

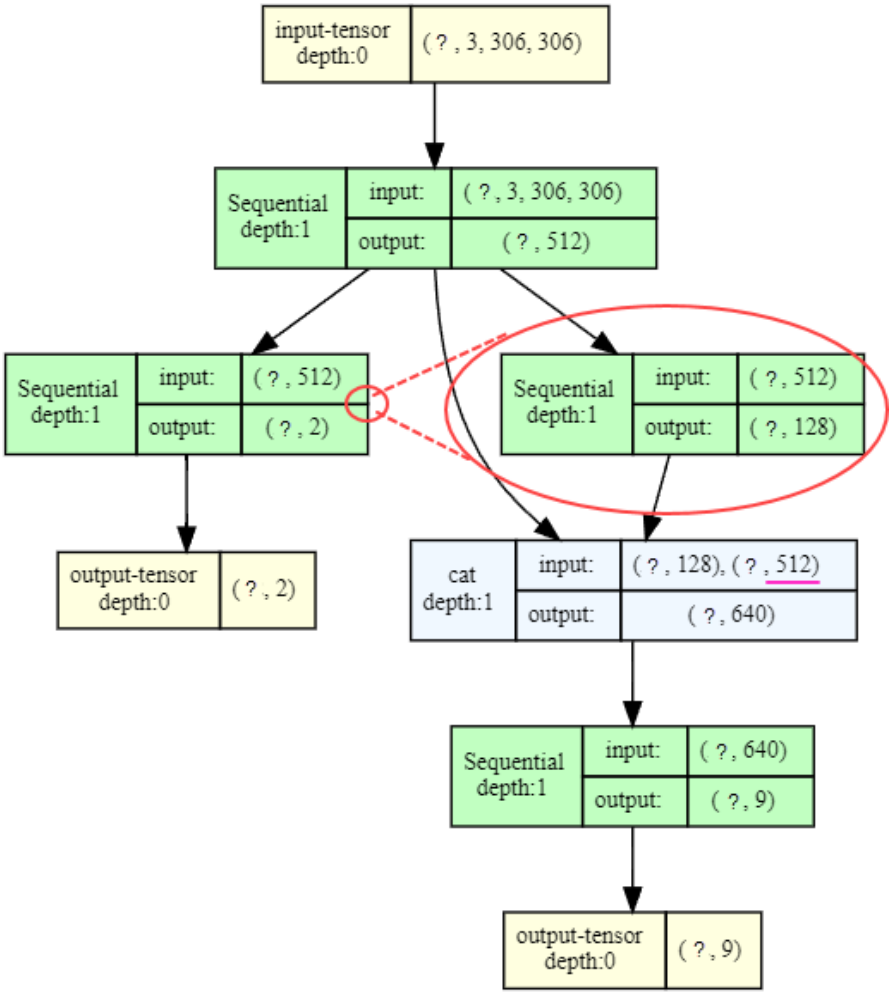


Figure 37. HCNN Architecture: depth of 1.

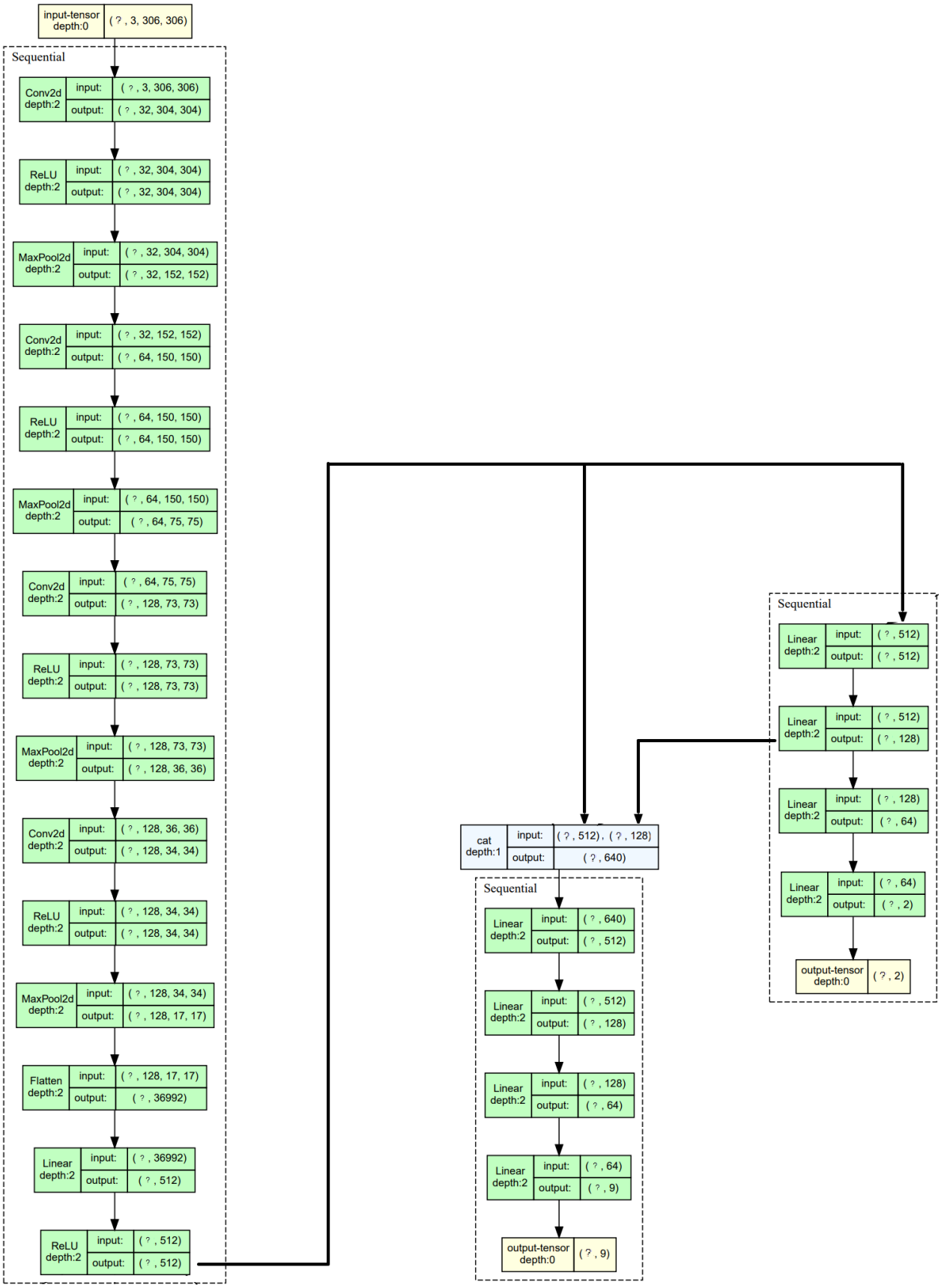


Figure 38. HCNN Architecture: depth of 2.

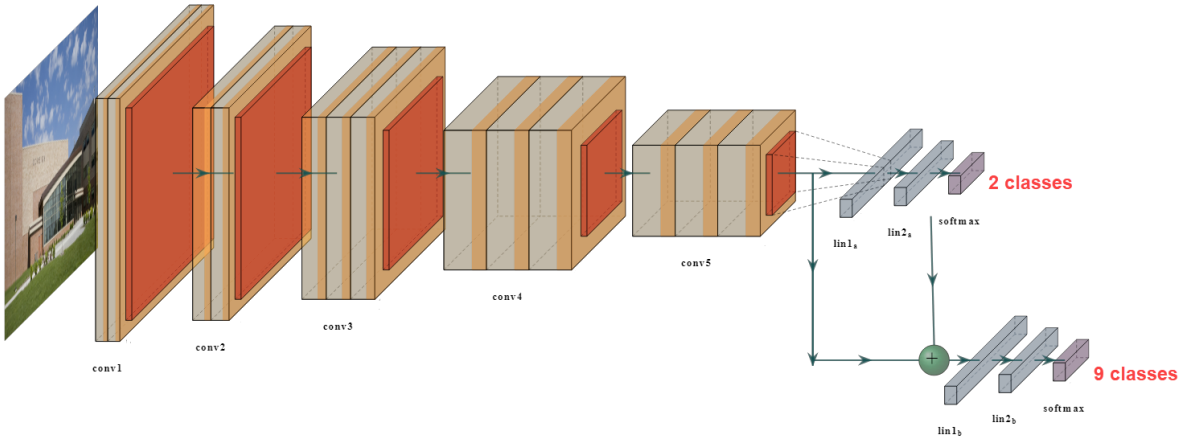


Figure 39. Arbitrary HCNN Architecture to grasp concept. Adapted from [44].

The following is a summary of the logical flow of HCNN with reference to Figure 38:

- The left sequential part (LSP) in the figure has the same architecture as VCNN introduced in section 3.2.1. Any image must be initially forwarded through this part regardless of the required classification.
- The right sequential part (RSP) is for classifying whether an image has many or few colours. It receives 512 "features" from the LSP representing general features extracted from the image and so the RSP will extract high level features based on the chosen class.
- The middle sequential part (MSP) is for classifying the original 9 classes in the dataset. In addition to the 512 features received from the LSP, it also receives 128 features from the MSP via concatenation. These added features essentially tell the MSP that the image it is trying to predict is either an image with many colours or few ones, which should help MSP rule out the classes under the "Few Colors" parent class if the image has many colours and vice versa.
- Note that even though both MSP and RSP are trained and backpropagated at the same time, the MSP should theoretically learn faster than RSP due to the coarseness of its classification task.

To conclude, the model's trainable parameters were 19,846,731.

3.2.4 XCNN: XGBoost and CNN

The basic idea behind the machine learning (ML) algorithm called XGBoost (short for Extreme Gradient Boosting) is to iteratively add decision trees to an ensemble of decision trees (weak learners), where each of these newly added trees is trained to correct the mistakes made by the previous trees using gradient boosting technique. Boosting is essentially building model

1 on dataset 1, then model 2 based on itself and model 1's classification but on dataset 2 (another part of the original dataset), and so on [45].

Now, since it is difficult for an ML algorithm like XGBoost to train directly on images, we first used the overall best model from "m01.x" series of models, which was "m01.1" to extract 512 features from the images (output of second to last dense layer), and stored these features in a csv file, which was then used to train the XGBoost model.

3.3 Model Evaluation Phase

Loss, accuracy, f1 score metrics and confusion matrices were logged for all models. Obviously, due to the nature of the problem at hand, a multi classification problem with an imbalanced dataset, we'll display the other metrics' results, but we'll be focusing on the f1 score metric and confusion matrices during section 5's analysis.

4 Implementation

This section demonstrates how the project's pipeline, which was introduced in section 3, is implemented using Python language.

4.1 Dataset Preparation Phase

This sub-section discusses the main functions and logical sequence of obtaining, analysing, and pre-processing the dataset. In general, all of the upcoming sub-sections can be found in "dataset_preprocessing_part_(1/2/3)" Jupyter notebook files along with some notebook files in "gp_related" directory.

4.1.1 Data Collection and Scraping

As mentioned in section 3.1.2, classes "selfies" and "fmemes" were directly obtained from online sources, however, the files downloaded for "fmemes" class were mainly csv files that contained URLs for the actual foreign meme images, so we've written "downloadImgs()" function in "download_imgs.py" file which takes in URLs of images and requests then downloads said images into the designated class directory. Also, since writing images unto disk is an independent task per image, we've written "getAsyncImgFunctions()" and "runInParallel()" functions which create a list of "downloadImgs()" functions and then run each one of them parallelly; each with a different URL passed as an argument.

4.1.1.1 Scraping from Imgur and Facebook

For scraping "Imgur" albums, we simply used its API service. For scraping Facebook images, we mainly used "Helium" library [46] to easily simulate actions that a user normally does on a browser to access the images on Facebook such as "write(my_password, into='password')" and "press(PAGE_DOWN)" which are self-explained functions. Either way, the function " fbPageImgScraper()" was the main function used to scrap images from Facebook by logging into facebook, opens the i^{th} page from the pages mentioned in section 3.1.2.2, attempts to scroll till the end of the page in order to keep loading javascript into html, then in that final html obtained, uses "fbExtractImgsLinks()" function which uses regex to obtain all URLs of images in that html.

4.1.1.2 Scraping from Reddit

For scraping images from subreddit communities, we used the function "getSubredditImgsLinks ()" which uses Pushshift's API in "getPushshiftData()" to get the data related to images found the subreddit which are created in a specific time stamp, then, the date of the oldest post is used as the new start date to be passed in "getPushshiftData()" again, and

so on until it does not return any new data. The reason for iterating like this is that Pushshift's API has a limit to the posts it can retrieve, so simply passing a timeframe of posts that end at the initial creation date of the subreddit will not give us all the posts in that subreddit, thus we iterate as previously discussed.

4.1.1.3 Scraping from Google Images

For scraping "eGreetingAndMisc" images on Google images' search tab, we've written the function "getGooglePagesImgsLinks()" which uses "getGoogleSinglePageImgsLinks()" on each search term to get all of the images' URLs for that single search term, then runs that function again for each similar search term obtained from "getSimilarTerms()". Figure 40 Illustrates the difference between a search term and a similar search term.

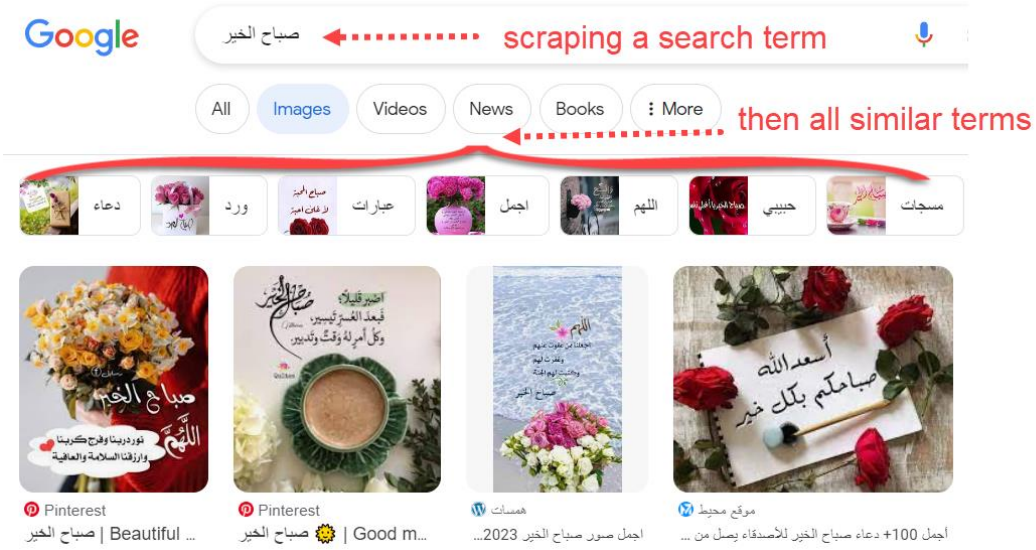


Figure 40. Google search term vs similar search term (search chips).

Moreover, the library "BeautifulSoup" is used in "getGoogleSinglePageImgsLinks()" to get the images' URLs by searching for the images' class name, which Google changes every now and then along with other class names, so these should be updated if the functions are reused. For example, at the time of writing this report, the class name "rg_i Q4LuWd" is used on each of the images as can be seen in Figure 41.

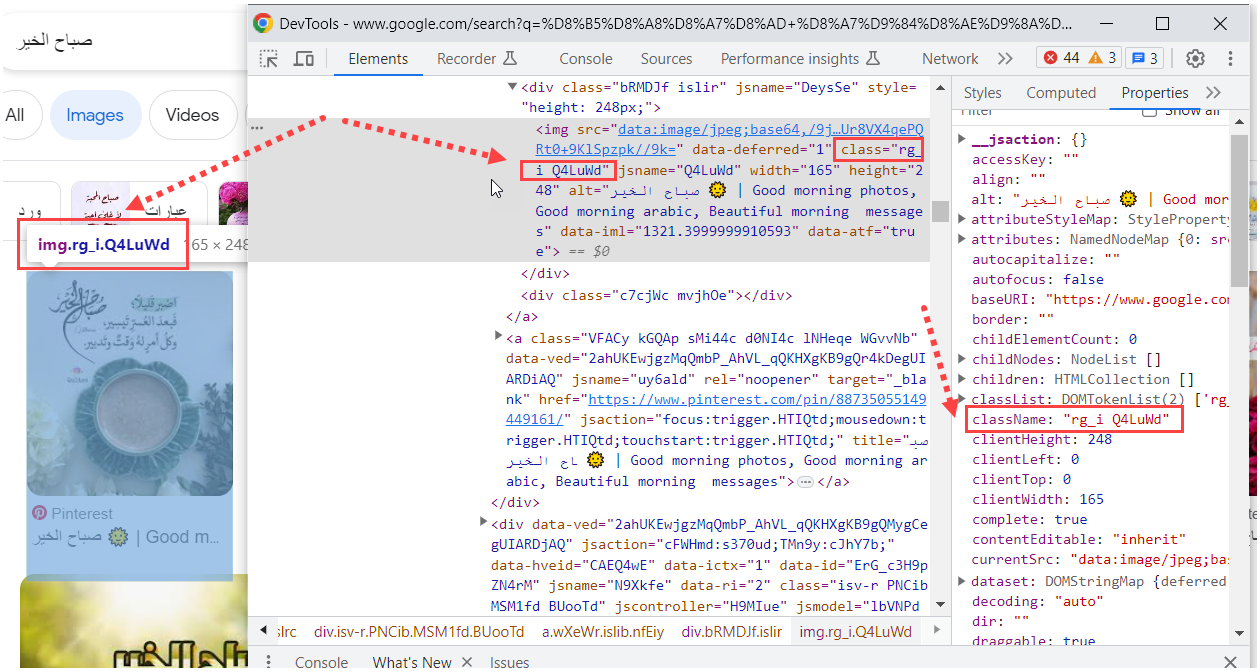


Figure 41. Class name of an image found by searching on Google.

4.1.2 Data Cleaning

The function “cleanImgs()” is responsible for corrupted images or images from dead URLs (recall Figure 20 for illustration). Also, recalling the first problem mentioned in section 3.1.3 and Figure 18, the logic of the function “sortImgsBasedOnFunctionValue()” is used to get a scalar value for each image from either function “getHighestColorCounts()” or “getTextBoxesNum()”, then, “ipywidgets” library is used to visualize the sorted images in order to manually pick an index from which we manually set in code to move images before that index into another directory. The aforementioned logic can be demonstrated again by observing Figure 21, Figure 22, Figure 23, then observing Figure 42 and Figure 43.

Figure 42. Image at index 8 marks the start of the “academicPhotos” class.

```
# debugging:
# by manually checking the image's displays,
# we see that most of the images until index 8 are screenshots,
# so we'll move these images
for i, filename in enumerate([filename_and_dict[0] for filename_and_dict in image_list]):
    if i == 8:
        break
    src_path = joinPaths('../academicimagestemp/tmp/', filename)
    dest_path = joinPaths('../academicimagestemp/tmp2/', f'{i}_{filename}')
    shutil.move(src_path, dest_path)
```

Figure 43. Manually changing the index at which we stop moving "academicPhotos" files.

Up to this point, all data preparation steps mentioned in the report was for dataset version 1 ("v01"). However, only two models "m01.5" and "m01.6" use dataset "v01.2" and "v01.1" respectively. Now, the comments in "model_v01_vcnn" notebook succinctly explain the difference between these versions so they are illustrated in Figure 44.

```
* 01 is the train/val/test split of the original dataset

* 01.1 is the train/val/test split of 01 but removed from it
    images cleaned in dataset_preprocessing_part_3.ipynb

* 02 is the train/val/test split of 01.1 but with a new random split
    (so test set is completely different)

Therefore, only 01 and 0.1 should be compared due to
near-similarity between their test sets
Moreover, 01.1 has 2008, 428, and 447 less images in
train/val/test sets respectively than 01
```

Figure 44. Difference between datasets "v01", "v01.1", and "v02".

Essentially, "dataset_preprocessing_part_3" is the notebook responsible for creating "v01.1" and "v02" datasets, where similar logic to what was described at the start of this section and illustrated by Figure 21, Figure 22, Figure 23 is used to clean images from "academicDigital", "fSocialMedia", "eMemes", and "academicPhotos" classes since they had the lowest scores among other classes when training most of the models. The outline of the aforementioned notebook is shown in Figure 45.

Figure 45. Outline of cleaning done in "dataset_preprocessing_part_3" notebook.

Along with the logic of "getTextBoxNum()" mentioned at the start of this section, we've also used new functions in this notebook: "get_image_ratio()", "run_kmeans()", and "create_kmeans_clusters_gallery()". The first function simply scores and sorts each image based on its ratio, so the first few sorted images are very wide, which can be seen in Figure 46, while last few sorted images are very long[1]. Moreover, the other two functions are from "fastdup" library [47], where the former runs K-means algorithm, with a passed number of clusters as argument, on the dataset and stores the results in a csv "fastdup_work_dir_{class_name}" directory which is then used by the latter function to create an HTML page for visualizing these clusters. Examples of this are in Figure 47 and Figure 48.

[1] "dataset_preprocessing_part_3.ipynb" decreases the manual work required, by automatically renaming all the image files in the directory (shown in Figure 46) so that they are sorted and you can directly remove them using your operating system's GUI.

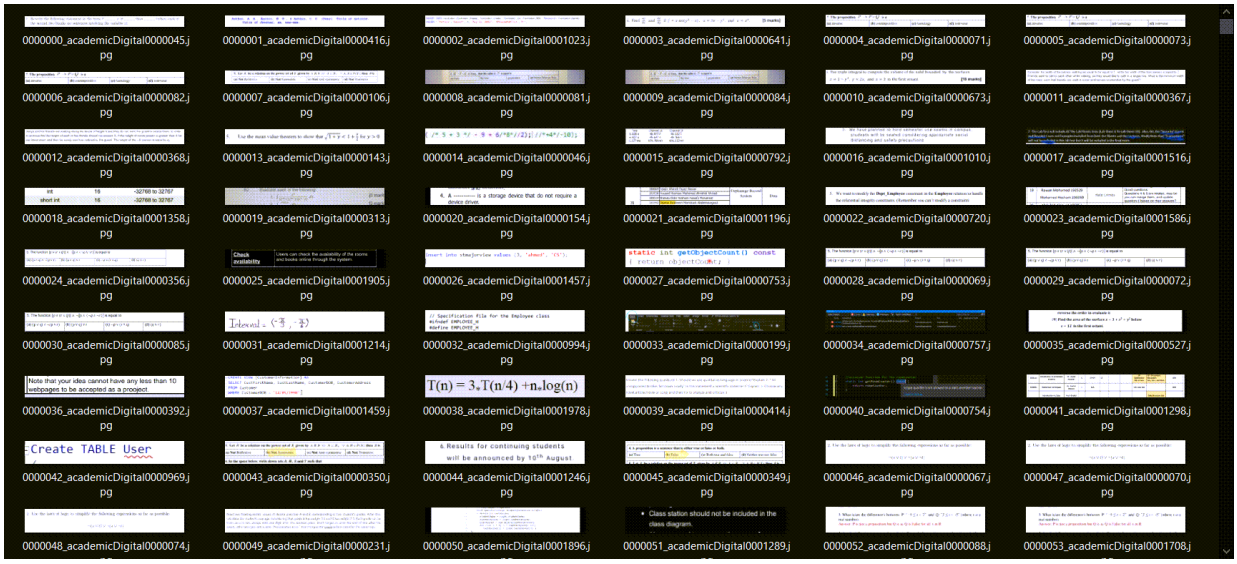
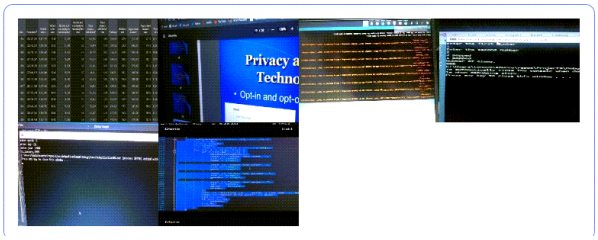


Figure 46. Very wide “academicDigital” images.

Info	
component	6
num_images	48
mean_distance	106.078857



Info	
component	9
num_images	6
mean_distance	146.94873

Figure 47. K-mean clusters of “academicDigital” images which should be in “academicPhotos”.

Info	
component	6
num_images	18
mean_distance	204.656982

Figure 48. K-mean cluster of outlier images in “academicPhotos”.

4.1.3 Data Pre-processing

Metadata explained in section 3.1.4.1 and in Figure 31 were extracted for each image using "getImgsPropsDf()" function. Now, since the relationships between the defined helper functions and "getImgsPropsDf()" is more complicated than the logic of the functions themselves, we've drawn the dependency between functions in Figure 49 with a brief description of each function. It is also important to note that "nltk" library's "WordNetLemmatizer" module [48] and "qalsadi" library's "Lemmatizer" module [49] were used in CF7 and CF8 respectively.

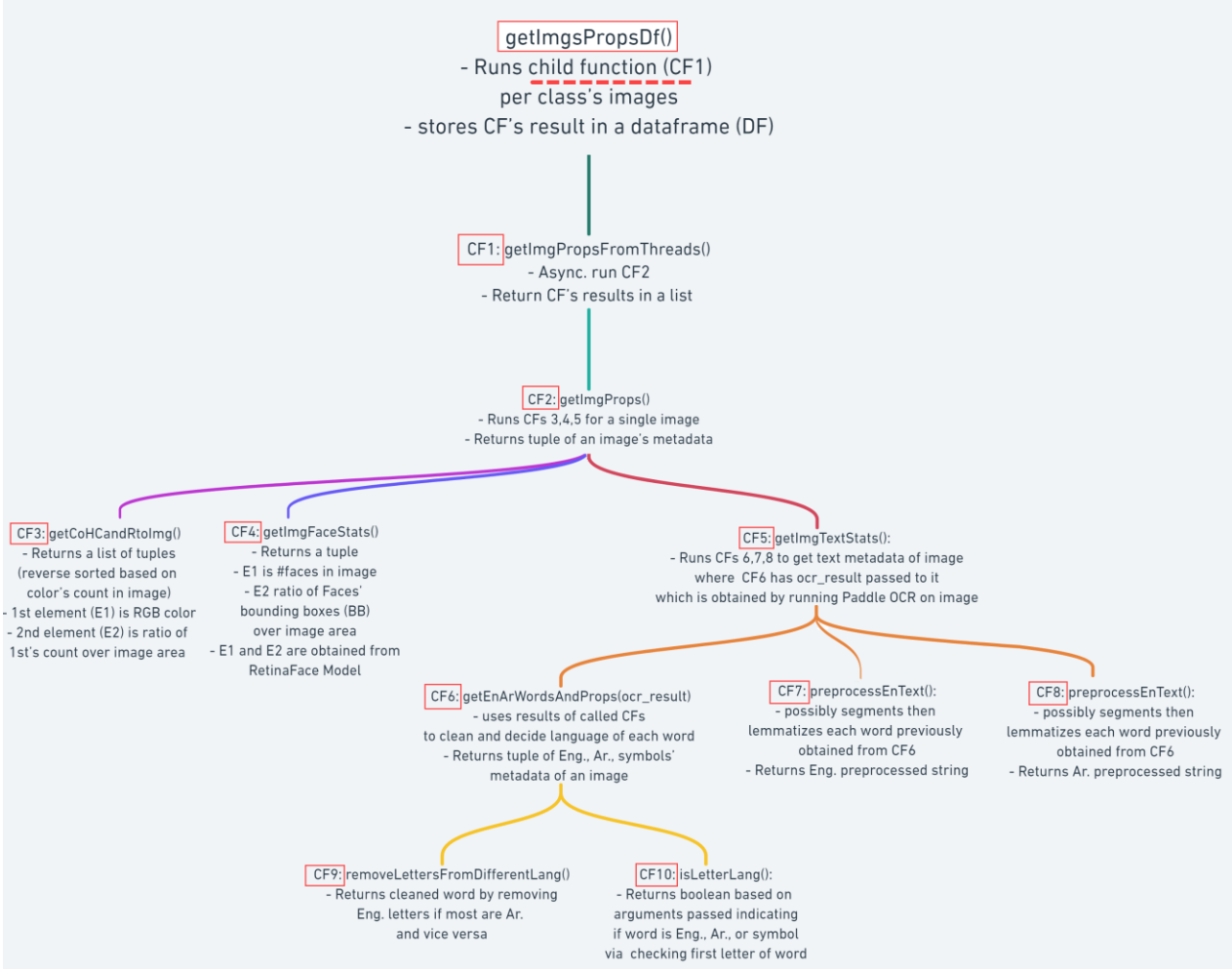


Figure 49. Relationship between functions used for getting metadata mentioned in Figure 31.

Now, Figure 50, Figure 51[1], and Figure 52 are used to illustrate the outputs of "RetinaFace" model and "Paddle OCR" tool respectively. From the last two mentioned figures, we can see that the order of Arabic words is not always correctly reflecting the order of words in the image, and that is because "Paddle OCR" detects words from left to right and from top to bottom, so

[1] Unfortunately, we weren't able to visualize the bounding boxes of the bottom image, so the original image is used instead.

we had to reverse the order of the words retrieved when a line of text ends in CF6, which did not always yield accurate results.

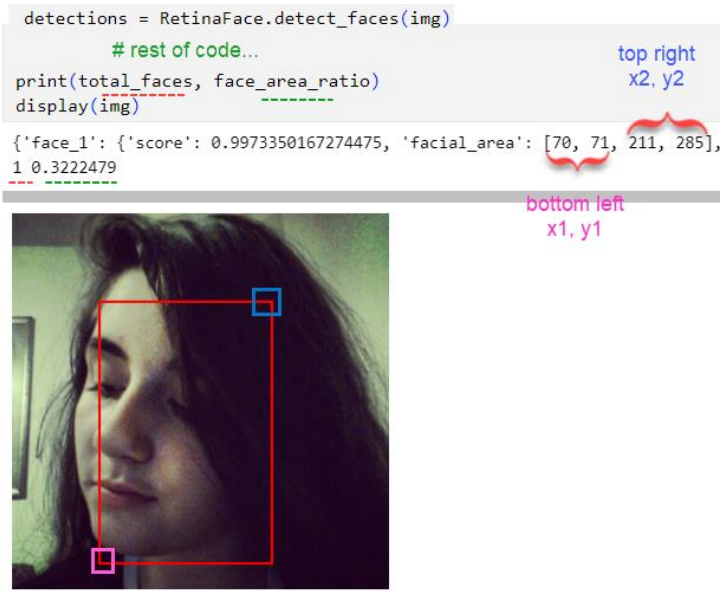
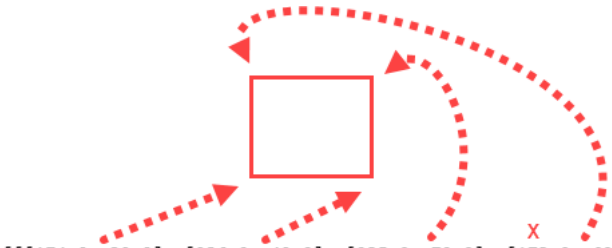


Figure 50. Output of "RetinaFace" model.

Figure 51. Two images were tested on "Paddle OCR" tool along with their "ar_words_original" strings, where words underlined in green are detected correctly.




```
[[[174.0, 39.0], [236.0, 42.0], [235.0, 72.0], [173.0, 69.0]], ('Boe', 0.998548686504364)]
[[[174.0, 85.0], [359.0, 85.0], [359.0, 118.0], [174.0, 118.0]], ('adepressdo', 0.9775846600532532)]
[[[106.0, 189.0], [150.0, 189.0], [150.0, 235.0], [106.0, 235.0]], ('٧٧', 0.5993947982788086)]
[[[158.0, 193.0], [238.0, 193.0], [238.0, 233.0], [158.0, 233.0]], ('0.9999750256538391 ,كنت')]
[[[243.0, 191.0], [330.0, 188.0], [331.0, 242.0], [245.0, 245.0]], ('0.9361270070075989 ,الايام')]
[[[341.0, 199.0], [397.0, 204.0], [393.0, 244.0], [337.0, 238.0]], ('0.7058418989181519 ,به')]
[[[466.0, 194.0], [518.0, 194.0], [518.0, 244.0], [466.0, 244.0]], ('0.9999740123748779 ,في')]
[[[523.0, 192.0], [580.0, 192.0], [580.0, 244.0], [523.0, 244.0]], ('0.5853405594825745 ,قمر')]
[[[587.0, 193.0], [703.0, 193.0], [703.0, 240.0], [587.0, 240.0]], ('0.9999850392341614 ,متخيل')]
[[[403.0, 206.0], [462.0, 206.0], [462.0, 244.0], [403.0, 244.0]], ('0.9999723434448242 ,يوم')]
[[[145.0, 255.0], [199.0, 255.0], [199.0, 303.0], [145.0, 303.0]], ('0.9043794274330139 ,اني')]
[[[211.0, 256.0], [293.0, 259.0], [291.0, 295.0], [209.0, 291.0]], ('0.999965250492096 ,نعمة')]
[[[301.0, 261.0], [391.0, 259.0], [392.0, 297.0], [302.0, 300.0]], ('0.9999805688858032 ,مقدر')]
[[[611.0, 256.0], [708.0, 256.0], [708.0, 300.0], [611.0, 300.0]], ('0.9999946355819702 ,وكنت')]
[[[506.0, 265.0], [598.0, 261.0], [600.0, 297.0], [507.0, 301.0]], ('0.9945564866065979 ,حزين')]
[[[721.0, 314.0], [791.0, 322.0], [787.0, 358.0], [717.0, 350.0]], ('0.9999902248382568 ,سنة')]
[[[227.0, 338.0], [474.0, 335.0], [474.0, 362.0], [227.0, 365.0]], ('foinboxertu', 0.8441101312637329)]
```

Figure 52. Detailed output of "Paddle OCR" tool when used on top image in Figure 51.

Another caveat to note is that the csv file which stores all of the dataset's extracted metadata, named "imgsPropsv1.csv", displays Arabic words as symbols due to Microsoft Excel using a different encoding system than UTF-8 [50]. However, if you read that csv file in python using Pandas library, you'll see that the Arabic words are rendered properly

4.2 Model Development Phase

The following sub-sections discuss the details of implementing the models used throughout this project along with the hyperparameters per model variant. Moreover, there are hyperparameters that are consistent throughout most models, **unless explicitly stated otherwise**, such as using RGB images instead of grey scaling them, training on 30 epochs with a batch size of 32 on the entire dataset without augmenting or any preprocessing other than resizing the images to 306x306, using Adam optimizer with default hyperparameter values specified in Tensorflow's documentation [51], specifically an initial learning rate of 0.001, beta-1 of 0.9, and beta-2 of 0.999. Moreover, the categorical cross entropy loss function was used as the models' objective function. In addition, there are also functions which are applied throughout most of the models. For example, "get_generators_v{x}()" function[1] returns train, validation, and test generators which pass a certain number of images, based on chosen batch size, to the model per single step of an epoch, where each epoch contains a number of steps equal to the

[1] The "x" refers to the model version, so "m02" variants use "get_generators_v2()", while "m03" uses "get_dataloaders()".

number of samples over the batch size, and illustrates this with an example. Also in , the numbers 94787, 20311, and 20312 are the number of images in train, validation, and test sets respectively. As you can notice, we've split the 135,410 total images into 75%, 15%, and 15% splits which are found in "train_datav01.csv", "val_datav01.csv", and "test_datav01.csv" respectively in "dataset_related" directory.

```
Found 94787 validated image filenames belonging to 9 classes.
Found 20311 validated image filenames belonging to 9 classes.
Found 20312 validated image filenames belonging to 9 classes.
11848 training steps
2538 validation steps
2539 testing steps
```

Figure 53.Example of returned steps by the data generators assuming a batch size of 8.

Other examples of common functions used are "trainModel()" and "createTestResultsDf()"; the former is self-explained, while the latter creates 2 HTML pages which contain tables exemplified Figure 54 in Figure 55 and respectively; one of them sorts the table descendingly based on "prediction probability" column, while the other sorts the table based on the "actual class name" column then ascendingly by "prediction probability" column. Noting that the file names are prefixed with "m03hnn_ol_1 test - probs sorting" and "m03hnn_ol_1 test - class and probs sorting" respectively in "dataset_related" directory.

	image name	image	prediction probability (Desc.)	predicted class name	actual class name
0	academicDigital0001448.jpg		1.00000	30. fSocialMedia	82. academicDigital
1	fTxtMssgs0005582.jpg		1.00000	30. fSocialMedia	50. fTxtMssgs
2	eGreetingAndMisc0008586.jpg		1.00000	20. ememes	70. eGreetingAndMisc
3	fTxtMssgs0005877.jpg		1.00000	82. academicDigital	50. fTxtMssgs
4	eGreetingAndMisc0008938.jpg		1.00000	50. fTxtMssgs	70. eGreetingAndMisc
5	academicPhotos0000208.jpg		1.00000	00. selfies	81. academicPhotos

Figure 54. Sample of test results of "m03" sorted descendingly by prediction probability.

	image name	image	prediction probability	predicted class name	actual class name
0	selfies0035586.jpg		② (Asc.) 0.43843	00. selfies	① (Asc.) 00. selfies
1	selfies0034865.jpg		0.45134	81. academicPhotos	00. selfies
2	selfies0033391.jpg		0.46368	70. eGreetingAndMisc	00. selfies
3	selfies0038112.jpg		0.49094	70. eGreetingAndMisc	00. selfies
4	selfies0022526.jpg		0.50350	70. eGreetingAndMisc	00. selfies
5	selfies0015627.jpg		0.51503	70. eGreetingAndMisc	00. selfies

Figure 55. Sample of test results of "m03" sorted ascendingly by actual class name then prediction probability.

The other common functions are related to evaluation and logging metrics and will therefore be mentioned in section 4.3. Moreover, all models were trained using TensorFlow and Keras libraries except "m03" (HCNN), which was trained using PyTorch and the XBoost model "m04" (XCNN) was trained using scickit-learn library. Finally, a brief description of each of the model variants can be found in Table 2 of section 4.3.

4.2.1 VCNN: Vanilla CNN

Figure 56 represents the markdown's outline of "model_v01_vcnn" notebook file indicating the hyperparameters' description of "m01.x" models. It is important to note that "m01.5" will not be considered while comparing the results, as that model uses an entirely different train, validation, and test split outcome (dataset v02) than the other datasets (v01 and v01.1)

m01: Model on Original Dataset with no Metadata, Batch Size of 8, and 30 Epochs
m01.1: m01 but Batch Size of 32 not 8
m01.3: m01 but Batch Size of 16 not 8
m01.4: m01.3 but Rotated (i.e., Augmented) Dataset not Original
m01.5: m01.1 but v02 Dataset not Original (i.e., v01)
m01.6: m01.1 but v01.1 Dataset not Original (i.e., v01)

Figure 56. Outline of "m01.x" models.

4.2.2 MCNN: CNN Incorporating Metadata

Figure 57 represents the markdown's outline of "model_v02_mcnn" notebook file indicating the hyperparameters' description of "m02.x" models. However, the "get_generators_v2()" now uses "CustomDataGeneratorForV02" class which inherits Keras's "Sequence" module, which incorporates the metadata in "imgsPropsv01.csv" along with the 306x306 image as input to the model.

m02cnn: Model on Original Dataset with Color-Related Metadata, Batch Size of 16, and 30 Epochs
m02.1cnn: m02cnn with Metadata: Color
m02.2cnn: m02cnn with Metadata: Face & Color
m02.3cnn: m02cnn with Metadata: Face & Normalized Color
m02.4cnn (Corrupted): m02cnn with Metadata: Text & Face & Color
m02.5cnn: m02cnn with Metadata: Normalized (Text & Face & Color)

Figure 57. Outline of "m02.x" models.

4.2.3 HCNN: Hierarchical CNN

Due to time constraints, there is only one variant of this model which is "m03", and it uses the one-level hierarchy introduced in Figure 34. The hierarchy structure is passed to "HierarchalModel" class of "simple-hierarchy" [44] library to generate the structure of the model, as previously shown in Figure 36, Figure 37, and Figure 38. Now, Figure 58 shows the required hierarchy mappings for the model.

```
hierarchy_1level = {
    ("Colors", 2) : [("Classes", 9)],
}

# mapping flat labels to hierarchical labels
labelToHierarchy_1level = {
    0 : [
            0, # diversity of colors: many or few (so classes: 0,1)
            0, # class label
        ],
    1 : [0, 1],
    2 : [0, 2],
    # assumes fSocialMedia has few colors
    # (fewer than selfies and memes at least)
    3 : [1, 3],
    4 : [1, 4],
    # assumes fTxtMssgs has few colors
    # (fewer than selfies and memes at least)
    5 : [1, 5],
    6 : [0, 6],
    7 : [1, 7],
    8 : [1, 8],
}

# mapping hierarchical labels (integers) to hierarchical names (strings)
hierLayerIdxToClassNames_1level = {
    # diversity of colors
    0 : ['Many Colors', 'Few Colors'],
    # classes
    1 : ['00. selfies', '10. fmemes', '20. ememes', '30. fSocialMedia', '40. eSocialMedia',
         '50. fTxtMssgs', '70. eGreetingAndMisc', '81. academicPhotos', '82. academicDigital'],
}
```

Figure 58. Code for creating class groupings, hierarchical labels, and hierarchical names.

The “labelToHierarchy_1level” (LTH) variable is the most important variable in Figure 58, as it is used by “HierarchalModelPL”, a PyTorch Lightning wrapper around “HierarchalModel”, to properly change the flat y labels, which are given in batches of size 32 by the “get_dataloaders()” returned dataloaders, into hierarchical labels, then matrix and array operations are performed to be able to correctly compare and compute loss, accuracy and f1 scores of the y predictions forwarded from “HierachalModel” with true y labels per each output layer. These “matrix and array operations” are illustrated in Figure 59.

Additionally, a “CustomStratifiedSampler” class was created to attempt to stratify each batch of 32 images passed by the dataloader. Of course this did not work for a lot of the batches, due to the imbalanced dataset problem which imposed some batches to be dominated by the majority class either way.

Finally, regarding the loss function; since we have multiple output layers; each one will have its own loss value, so the loss function we’re trying to optimize will be to minimize a loss value comprised of the sum of the two output layers’ loss values.

```
# assume a batch size of 4
a = torch.rand(4, 3,*img_size)
hier_model(a)
```
✓ 0.2s

```
(tensor([[0.0549, 0.1794],
         [0.0547, 0.1795],
         [0.0548, 0.1793],
         [0.0547, 0.1795]], grad_fn=<AddmmBackward0>),
 tensor([[-0.0745, -0.1007,  0.0128, -0.1221,  0.0113,  0.0724, -0.0560, -0.0426,
           0.0023],
         [-0.0744, -0.1005,  0.0128, -0.1221,  0.0114,  0.0723, -0.0557, -0.0427,
           0.0025],
         [-0.0744, -0.1006,  0.0129, -0.1222,  0.0112,  0.0725, -0.0558, -0.0427,
           0.0022],
         [-0.0743, -0.1006,  0.0129, -0.1220,  0.0113,  0.0724, -0.0557, -0.0426,
           0.0024]], grad_fn=<AddmmBackward0>))
```

```
flat_y = [0,1,2,3]
hier_y = torch.tensor([labelToHierarchy_1level[int(flat_y[i])]
                       for i in range(len(flat_y))], dtype=int)
hier_y
```
✓ 0.1s

```
tensor([[0, 0],
        [0, 1],
        [0, 2],
        [1, 3]])
```

```
hier_y = hier_y.T # transposing hier_y
for out_layer_idx, output_layer_vals in enumerate(hier_model(a)):
        # 'cur' refers to current output layer
        y_cur = hier_y[out_layer_idx]
        y_pred_cur = F.softmax(output_layer_vals, dim=1)
        print(('many vs few colors' if out_layer_idx == 0
               else 'flat labels') + ' output layer')
        print('y true:', y_cur, 'y pred:', y_pred_cur,
              sep='\n', end='\n\n')
```
✓ 0.4s

```
many vs few colors output layer
y true:
tensor([0, 0, 0, 1])
y pred:
tensor([[0.4689, 0.5311],
        [0.4688, 0.5312],
        [0.4689, 0.5311],
        [0.4689, 0.5311]], grad_fn=<SoftmaxBackward0>)

flat labels output layer
y true:
tensor([0, 1, 2, 3])
y pred:
tensor([[0.1064, 0.1037, 0.1161, 0.1015, 0.1159, 0.1232, 0.1084, 0.1099, 0.1149],
        [0.1064, 0.1037, 0.1161, 0.1015, 0.1159, 0.1232, 0.1084, 0.1098, 0.1149],
        [0.1064, 0.1037, 0.1161, 0.1014, 0.1159, 0.1233, 0.1084, 0.1098, 0.1149],
        [0.1064, 0.1037, 0.1161, 0.1015, 0.1159, 0.1232, 0.1084, 0.1098, 0.1149]]
       grad_fn=<SoftmaxBackward0>)
```

Figure 59. Preparing true and predicted y labels (assuming LTH variable from Figure 58).

4.2.4 XCNN: XGBoost & CNN

Figure 61 represents the markdown's outline of "model_v04_xcnn" notebook file indicating the hyperparameters' description of "m04.x" models. Moreover, Figure 61 shows the hyperparameters used on both "m04" and "m04.1" models. The only difference between them is that the latter incorporates sample weighting as an attempt to overcome the imbalanced dataset problem by assigning higher weights to the minority class or samples to make them more influential during training, where this influence comes from the fact that the model adjusts its parameters to minimize the weighted sum of the loss function, considering the assigned weights. This means that samples with higher weights contribute more to the overall loss and, therefore, have a larger impact on the model's training. We can also see from Figure 61 that the objective is "multi:softprob", which is equivalent to minimize the negative log likelihood of the predicted probabilities. Moreover, the "merror" and "mlogloss" metrics are essentially the accuracy subtracted from 1 and the negative log-likelihood of the predicted probabilities respectively. Note that this objective function (negative log-likelihood) is very similar to the multi class cross entropy loss used in other models, except that it doesn't penalize the model if it assigns a high probability to a class other than the true class [52].

```
m04: XGBoost Model on m01.1 CNN Model, and 100 Epochs (a.k.a, rounds, trees)
m04.1: m04 but with Sample Weights
m04.2: m04 but with Sample Weights & Bayesian Optimization
```

Figure 60. Outline of "m04.x" models.

```
# used on m04 and m04.1
params = {
    'booster' : 'gbtree',
    'learning_rate' : 0.3, # alternatively, 'eta'
    'max_depth' : 6,
    'min_child_weight' : 1,
    'seed' : 42,
    'verbosity' : 1,
    'num_class' : len(list(classNameToNum.keys())),
    'objective' : 'multi:softprob',
    'eval_metric' : ['merror', 'mlogloss']
}
```

Figure 61. Hyperparameters chosen for "m04" and "m04.1".

However, regarding "m04.2" hyperparameters, a dictionary of possible hyperparameter configurations were set up in "params_bayes" variable shown in which are then passed along with the XGBoost model to "skopt" library's "BayesSearchCV" class [53]: a class which uses Bayesian optimization to efficiently explore the hyperparameter space and choose hyperparameter configurations with a high predicted performance per iteration by developing a surrogate model (a Gaussian process) which statistically approximates the probability

distribution of the model's actual objective function, such that this approximation gets more close to the actual objective function's distribution over multiple iterations of hyperparameter sampling.

```
# used on m04.2
n_iter = 50 # Number of times we sample a parameter configuration and train a new model.
params_bayes = {
    f'{model_type}__booster' : ['gbtree', 'gblinear', 'dart'],
    f'{model_type}__learning_rate' : Real(0.01, 1.0, 'uniform'),
    f'{model_type}__max_depth' : Integer(2, 12),
    f'{model_type}__min_child_weight' : Integer(1, 10),
    f'{model_type}__subsample': Real(0.1, 1.0, 'uniform'),
    f'{model_type}__colsample_bytree': Real(0.1, 1.0, 'uniform'), # subsample ratio of columns by tree
    f'{model_type}__reg_lambda': Real(1e-9, 100., 'uniform'), # L2 regularization
    f'{model_type}__reg_alpha': Real(1e-9, 100., 'uniform'), # L1 regularization
    f'{model_type}__n_estimators': Integer(50, 100), # same as n_rounds (or "epochs" argument in trainModel())
}
```

```
pipe = Pipeline([
    (model_type, XGBClassifier(seed=42, verbosity=1,
                               num_class=len(list(classNameToNum.keys())),
                               objective='multi:softprob',
                               eval_metric=['merror', 'mlogloss']))
])

xgb_bayes = BayesSearchCV(estimator=pipe,
                          search_spaces=params_bayes,
                          n_iter=50,
                          cv=5,
                          n_jobs=6,
                          scoring='f1_macro',
                          random_state=42,
                          refit=True,
                          return_train_score=True)
```

```
xgb_bayes.best_params_
```

```
OrderedDict([('xgb_classifier__booster', 'gbtree'),
             ('xgb_classifier__colsample_bytree', 0.1),
             ('xgb_classifier__learning_rate', 0.4351418818131793),
             ('xgb_classifier__max_depth', 12),
             ('xgb_classifier__min_child_weight', 10),
             ('xgb_classifier__n_estimators', 100),
             ('xgb_classifier__reg_alpha', 1e-09),
             ('xgb_classifier__reg_lambda', 1e-09),
             ('xgb_classifier__subsample', 1.0)])
```

Figure 62. Sampling 50 hyperparameter configurations from "params_bayes" variable on "m04.2".

Note that the evaluation metric used in "BayesSearchCV" object is "f1_macro" which is the average of the classes' f1 scores, so at first, the model will train for "n_estimators" rounds while trying to minimize the loss ("multi:softprob" objective function), and then the resulting model will be evaluated by "BayesSearchCV" based on the average of f1-scores, such that for the next iteration, hyperparameters will be chosen that are aimed to increase that average of f1-scores. Outlines all of the models mentioned in this section and a brief description of each model variant.

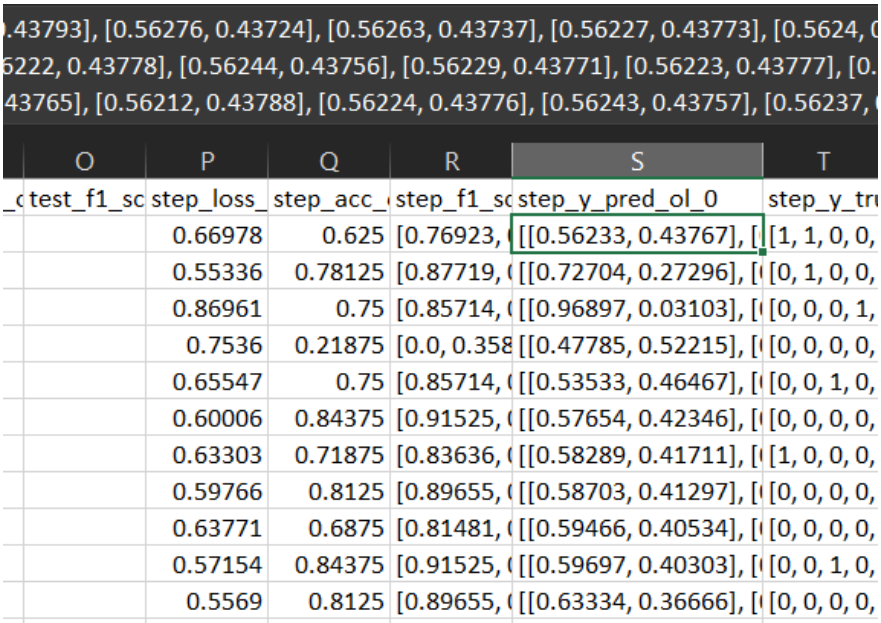
Model Type	Model Version	Description
VCNN	01	no metadata, 8 batch size, 30 epochs, no augmentation.
	01.1	Same as m01, but with 32 batch size
	01.3	Same as m01, but with 16 batch size
	01.4	Same as m01.3, but with augmentation by rotating images
	01.6	Same as m01.1, but on dataset "v01.1" instead of "v01"
MCNN	02	color metadata, 16 batch size, 30 epochs, no augmentation
	02.1	Same as m02, but color metadata is scaled to [0, 1]
	02.2	Same as m02, but also with face metadata
	02.3	Same as m02.1, but also with face metadata
	02.5	Same as m02.1, but scaling face and text metadata as well
HCNN	03	1 level, stratified, no metadata, 32 batch size, 30 epochs, no augmentation
XCNN	04	no metadata, all batch, 100 epochs, no aug, no es, used m01.1
	04.1	Same as m04, but with sample weights
	04.2	Same as m04.1, but with Bayesian optimization and cross validation on train set

Table 2. Models' description table

4.3 Model Evaluation Phase

This section will discuss how metrics were logged and visualized for comparison. First, since most of the models were developed with TensorFlow, the "history" variable retrieved from training the Keras models had accuracy and loss results for each of the 30 epochs. However, TensorFlow did not have the functionality of storing non-scalar values per epoch, so we had to use a custom defined metric class "F1Score" found in "tf_f1score.py" file which was passed to the TensorFlow models to get the f1 score of the 9 classes for each epoch. Moreover, even though no history variable was returned from training an XGBoost model, logging the metric scores was still not difficult as the "evals_result" parameters of xgb's "train()" method allowed us to pass an empty dictionary which will be later filled with metric results per round. The only problem was that the library did not allow storing scalar values even with custom functions, so we had to create a custom function which gets the average of f1 scores instead of the 9 f1 scores themselves. This only applied to the train and validation though; since we had the test set's predictions, we just passed the predicted and true labels to scikit learn's "classification_report()" function which gave us useful metrics including the 9 f1 scores.

The real problem was in “m03” (HCNN), as we wanted to mimic the functionality of other models and get a history variable containing the metrics for all epochs instead of relying on native logging solutions which stored only scalar metrics like “TensorBoardLogger” or “CSVLogger”. Thus, we defined our own logger class called “CustomCSVLogger” which is similar to “CSVLogger”, but allows for storing non scalar values in csv files; all the metric scores passed to this class’s “log_metrics()” method are converted into lists or floats (rounded to 5 decimals) and the string representations of these values are stored in a csv (while training the model, the csv’s name was “m03_metrics.csv” and was found in “models/my_logger” directory). Then the “finalize()” method of this class would run only after the training was done to retrieve the content of this csv file as strings, and use “ast” library’s “literal_eval()” method to evaluate that string into a float, list, or list of lists, then store that into “history” variable, which can be later accessed via the “CustomCSVLogger” object passed as the model’s logger. Rows from “m03_metrics.csv” are shown in . From that figure we can also see that some columns are empty; that is because they are metrics which are stored at each step (i.e., batch) that is evaluated, and there are other metrics that are stored at each epoch (i.e., after all steps are gone through in that epoch). Additionally, some metrics are stored later anyways like validation and testing related metrics. Comprises all the “history” dictionary’s keys, where “ol” stands for “output layer” and the absence of “step_” prefix indicates epoch related metrics, and the absence of “_ol” suffix indicates that the metric is an aggregation of all output layers’ scores for that metric.



.43793], [0.56276, 0.43724], [0.56263, 0.43737], [0.56227, 0.43773], [0.5624, (
6222, 0.43778], [0.56244, 0.43756], [0.56229, 0.43771], [0.56223, 0.43777], [0.
43765], [0.56212, 0.43788], [0.56224, 0.43776], [0.56243, 0.43757], [0.56237, (

	O	P	Q	R	S	T
_(	test_f1_sc	step_loss_	step_acc_(	step_f1_sc	step_y_pred_ol_0	step_y_tru
		0.66978	0.625	[0.76923, (	[[0.56233, 0.43767], [(	[1, 1, 0, 0,
		0.55336	0.78125	[0.87719, (	[[0.72704, 0.27296], [(	[0, 1, 0, 0,
		0.86961	0.75	[0.85714, (	[[0.96897, 0.03103], [(	[0, 0, 0, 1,
		0.7536	0.21875	[0.0, 0.358	[[0.47785, 0.52215], [(	[0, 0, 0, 0,
		0.65547	0.75	[0.85714, (	[[0.53533, 0.46467], [(	[0, 0, 1, 0,
		0.60006	0.84375	[0.91525, (	[[0.57654, 0.42346], [(	[0, 0, 0, 0,
		0.63303	0.71875	[0.83636, (	[[0.58289, 0.41711], [(	[1, 0, 0, 0,
		0.59766	0.8125	[0.89655, (	[[0.58703, 0.41297], [(	[0, 0, 0, 0,
		0.63771	0.6875	[0.81481, (	[[0.59466, 0.40534], [(	[0, 0, 0, 0,
		0.57154	0.84375	[0.91525, (	[[0.59697, 0.40303], [(	[0, 0, 1, 0,
		0.5569	0.8125	[0.89655, (	[[0.63334, 0.36666], [(	[0, 0, 0, 0,

Figure 63. “m03_metrics.csv” created from “CustomCSVLogger” class.

```
'epoch',
'modelNumber',
'loss',
'val_loss',
'test_loss',
'step_loss',
'step_val_loss',
'step_test_loss',
'loss_ol_0',                                'loss_ol_1',
'acc_ol_0',                                 'acc_ol_1',
'f1_score_ol_0',                            'f1_score_ol_1',
'val_loss_ol_0',                            'val_loss_ol_1',
'val_acc_ol_0',                             'val_acc_ol_1',
'val_f1_score_ol_0',                        'val_f1_score_ol_1',
'test_loss_ol_0',                           'test_loss_ol_1',
'test_acc_ol_0',                            'test_acc_ol_1',
'test_f1_score_ol_0',                       'test_f1_score_ol_1',
'step_loss_ol_0',                           'step_loss_ol_1',
'step_acc_ol_0',                            'step_acc_ol_1',
'step_f1_score_ol_0',                       'step_f1_score_ol_1',
'step_y_pred_ol_0',                         'step_y_pred_ol_1',
'step_y_true_ol_0',                         'step_y_true_ol_1',
'step_val_loss_ol_0',                       'step_val_loss_ol_1',
'step_val_acc_ol_0',                        'step_val_acc_ol_1',
'step_val_f1_score_ol_0',                   'step_val_f1_score_ol_1',
'step_val_y_pred_ol_0',                     'step_val_y_pred_ol_1',
'step_val_y_true_ol_0',                     'step_val_y_true_ol_1',
'step_test_loss_ol_0',                      'step_test_loss_ol_1',
'step_test_acc_ol_0',                       'step_test_acc_ol_1',
'step_test_f1_score_ol_0',                  'step_test_f1_score_ol_1',
'step_test_y_pred_ol_0',                    'step_test_y_pred_ol_1',
'step_test_y_true_ol_0',                    'step_test_y_true_ol_1',
'test_support_ol_0',                        'test_support_ol_1',
'test_confusion_matrix_ol_0',               'test_confusion_matrix_ol_1',
'test_confusion_matrix_normalized_ol_0',    'test_confusion_matrix_normalized_ol_1'
```

Figure 64. The keys of "history" dictionary obtained from "CustomCSVLogger" class and "getMetricsHNN()" function.

Finally, regarding logged metrics' visualization, "create_cm()", "plotTypeSingleModel()" function and its variant "plotTypeSingleModelHNN()" were called to utilize "Plotly" library's [54] heatmap, bar and line plots to visualize the confusion matrices, loss, accuracy, and f1 score metrics for each model variant. However, when we wanted to compare models with each other, as in "comparing_models_results" notebook, we also defined "plotTypeMultipleModels()" which plotted line and bar charts incorporating all of the metric results of the models' test sets for easier comparison in addition to returning a table to visualize the best f1 score for each of the 9 classes and the model version responsible for that score. Finally, "get_description_table()" was used to retrieve the descriptions (hyperparameters, etc) of each model variant from the HTML file names mentioned previously in section 4.2 and shown previously in Table 2 (with alterations).

5 Results

This section will focus on showing all the results obtained from training the models, First, the training, validation, and test results of each model will be shown, then comparisons between variants of each major model type and then between all models will be made. Finally, we'll analyse and summarize the results.

5.1 Train, Validation, and Test Results of Each Model

The following sub sections will contain figures from Figure 65 to Figure 132 displaying the metrics of each model respectively. Now, due to the large number of visualizations per model, we will only display the loss, validation, test f1-scores, and test set's normalized confusion matrix for each model except the "m03" HCNN model which obtained the best evaluation scores throughout our experimentation; we will display all recorded metrics for that model. Another exceptions are "m01.6" where we displayed the training f1 scores as well due to the abnormal overshooting of scores which appears while training this model, and "m04" where we also display the training f1 scores to show how XGBoost can easily overfit over the dataset. It is also important to note that model "m01.6" was trained on dataset "v01.1", which was previously explained in Figure 44. Moreover, all of the figures displayed in this section can be found in notebook files that start with "model_" and the "comparing_models_results" notebook. In addition to these files, there are also the HTML files that were mentioned in section 4.2, but because of the large sizes of these tables, we will not be able to show them here. Finally, it is important to note that the **"y axis" ranges of some of the upcoming figures are auto-scaled** based on the given data in each figure.

5.1.1 VCNN: "m01"

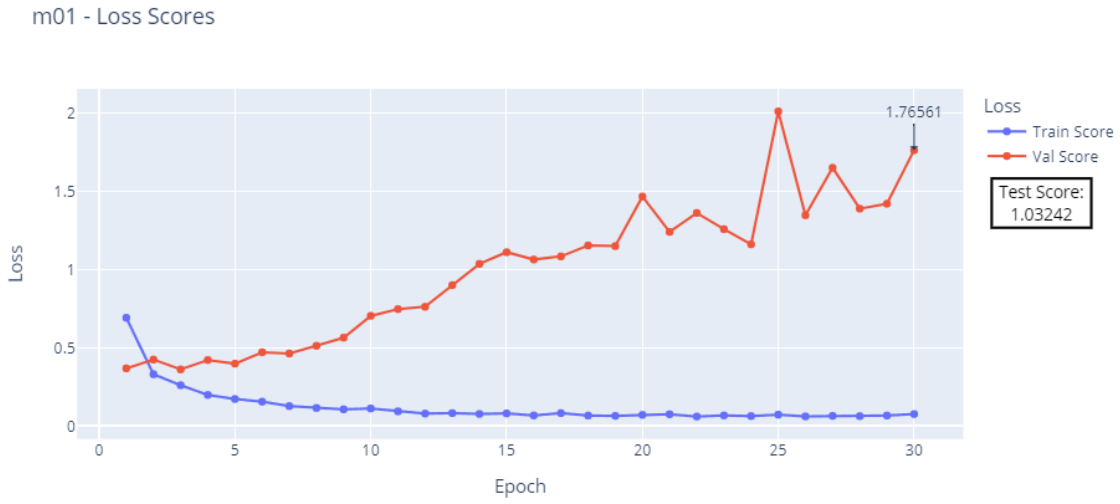


Figure 65. "m01" loss: train, validation, and test scores.

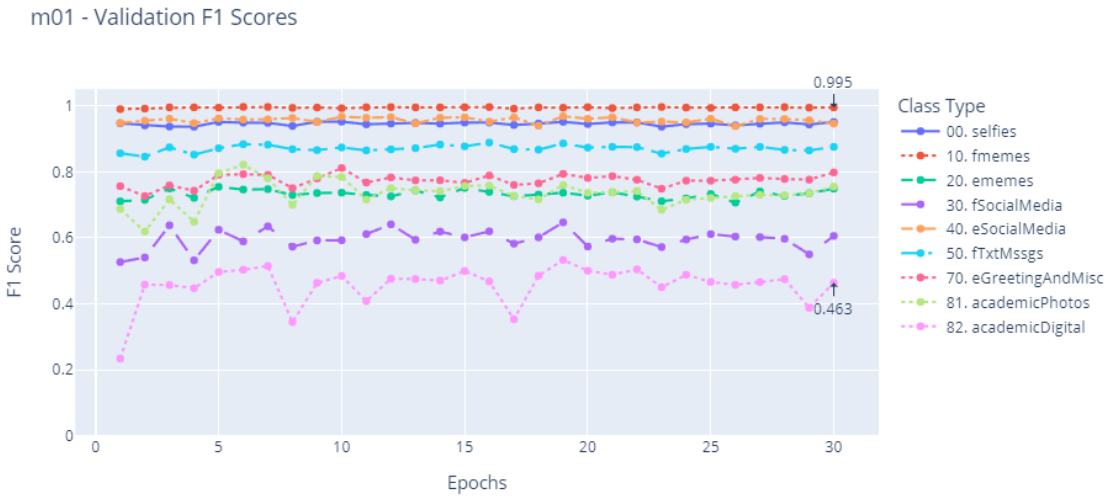


Figure 66. “m01” validation f1-scores.

Figure 67. “m01” test f1-scores.

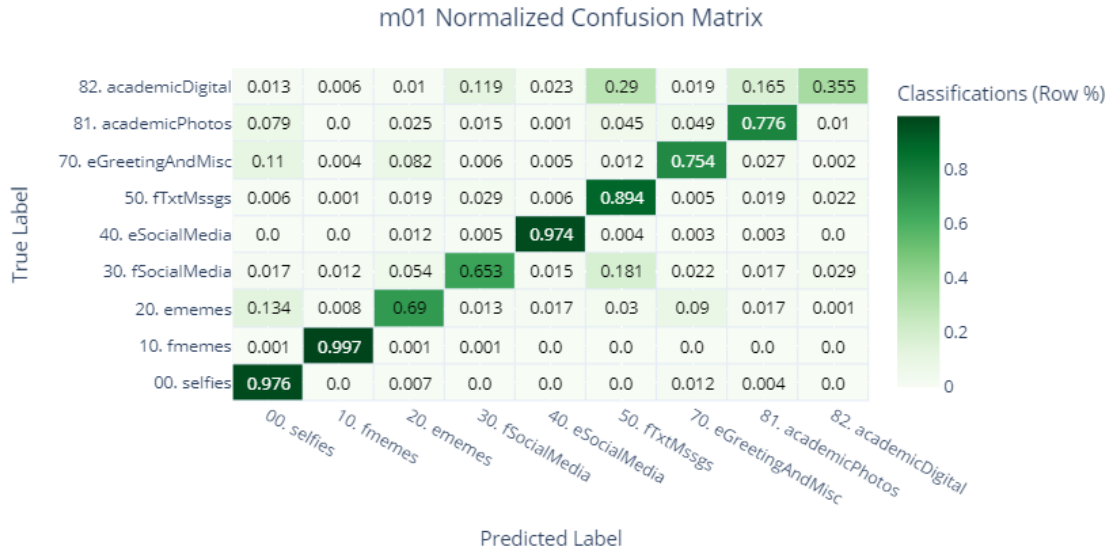


Figure 68. “m01” test set’s normalized confusion matrix.

5.1.2 VCNN: “m01.1”

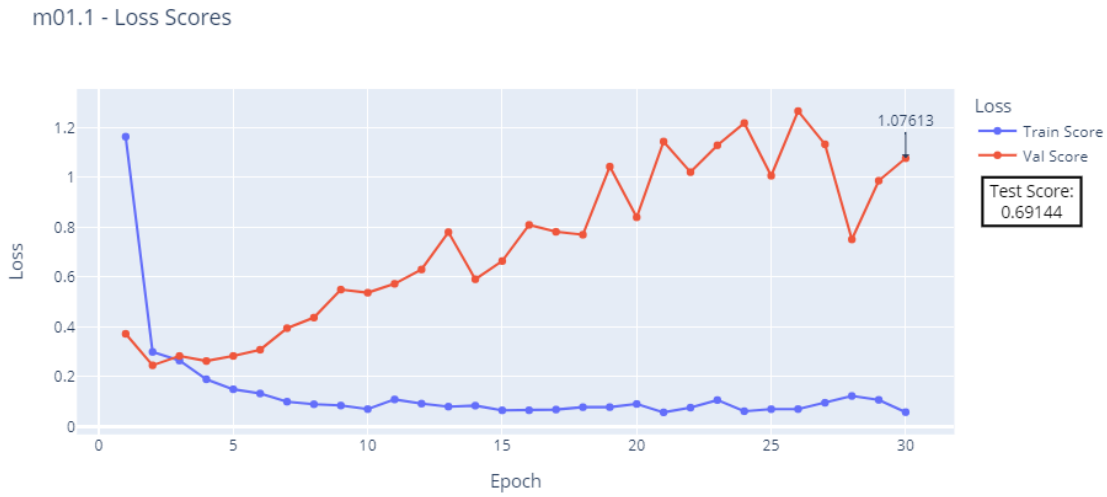


Figure 69. “m01.1” loss: train, validation, and test scores.

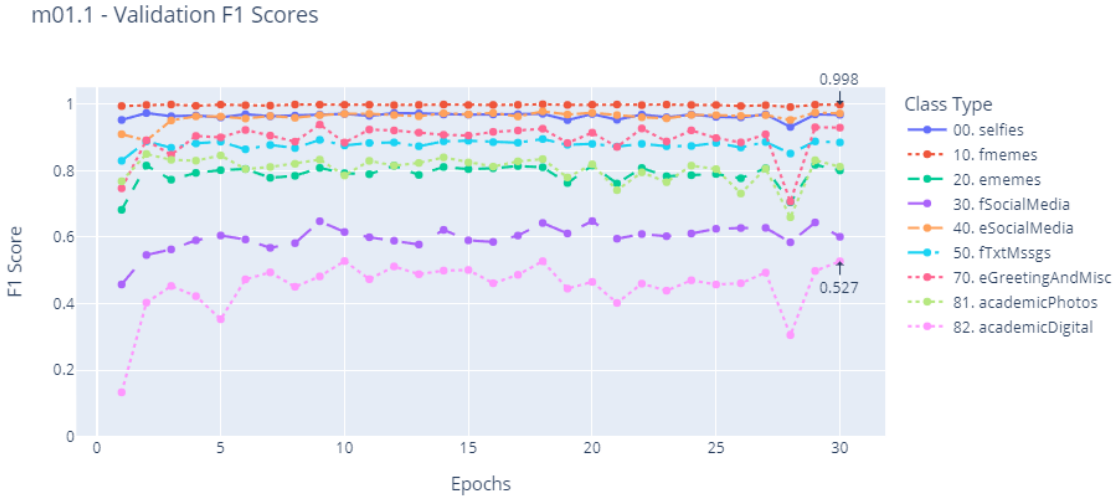


Figure 70. “m01.1” validation f1-scores.

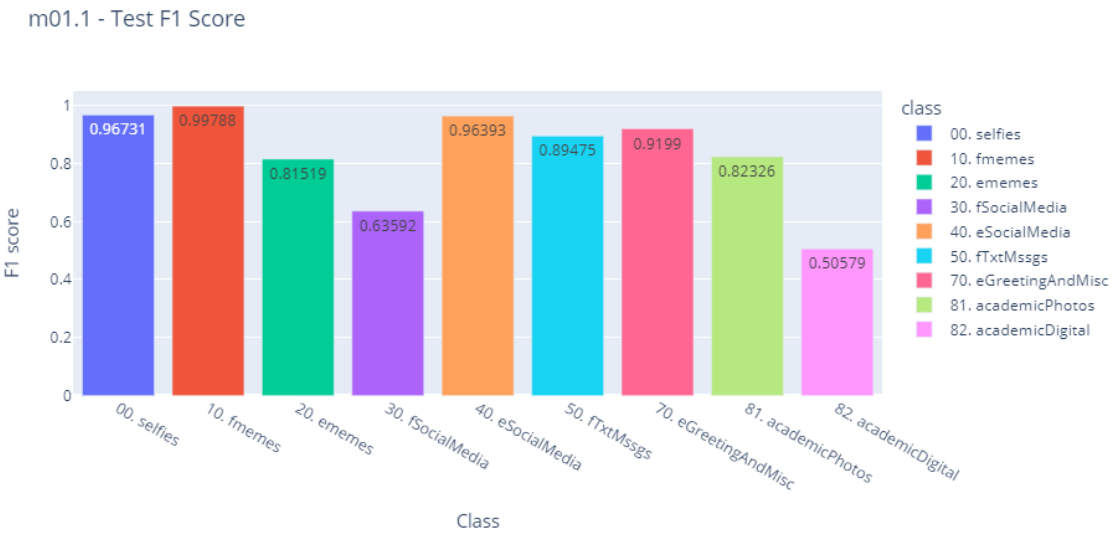


Figure 71. “m01.1” test f1-scores.

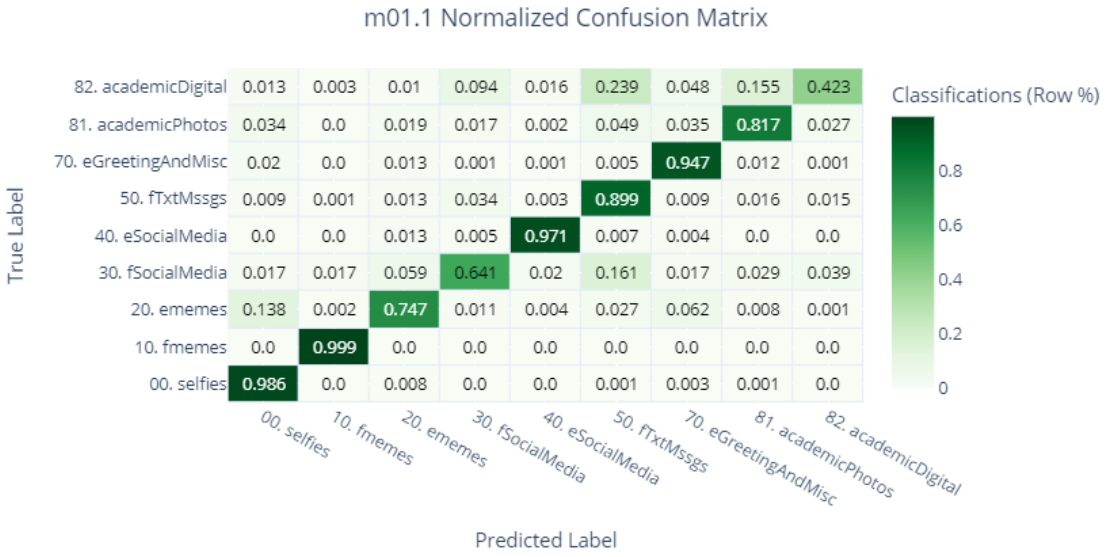


Figure 72. “m01.1” test set’s normalized confusion matrix.

5.1.3 VCNN: “m01.3”

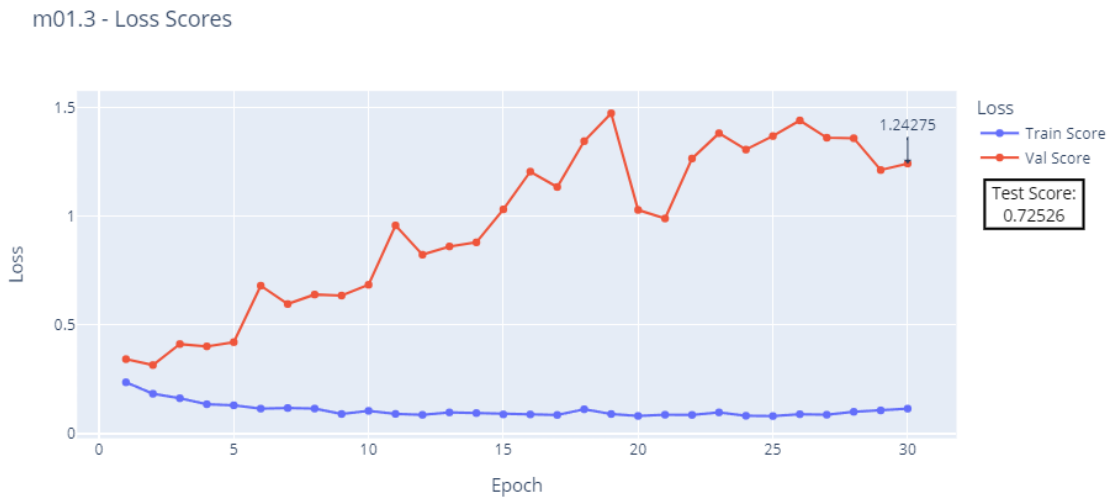


Figure 73. “m01.3” loss: train, validation, and test scores.

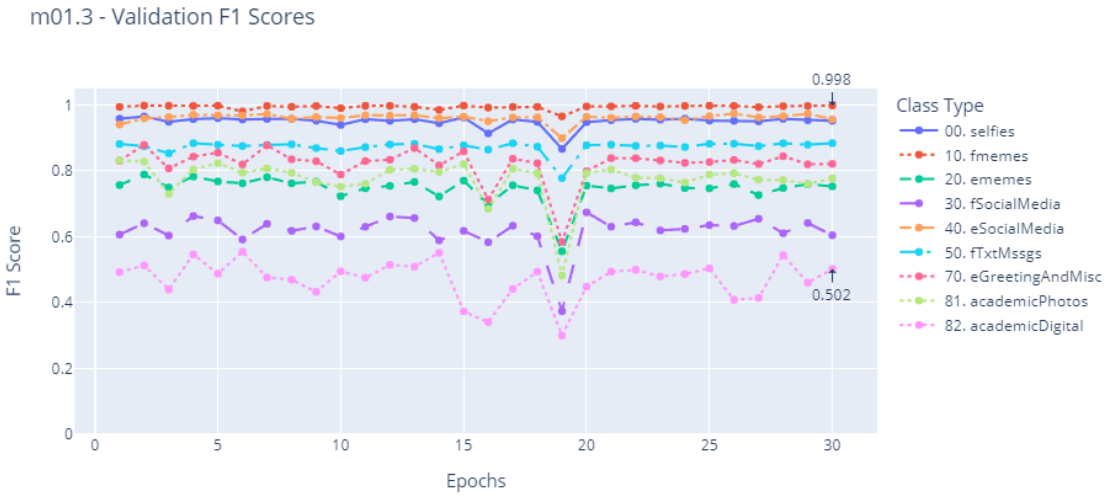


Figure 74. “m01.3” validation f1-scores.

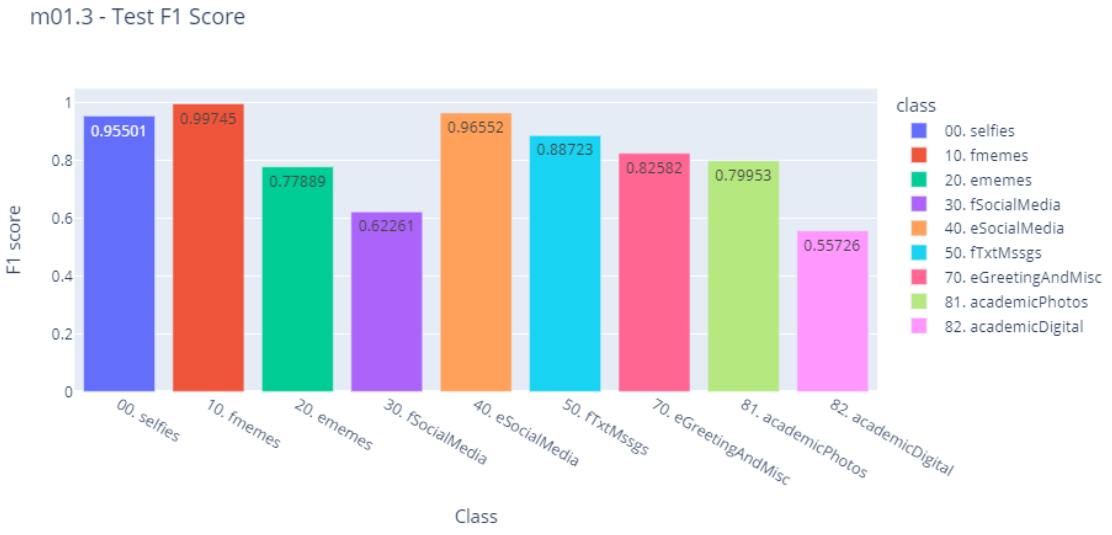


Figure 75. “m01.3” test f1-scores.

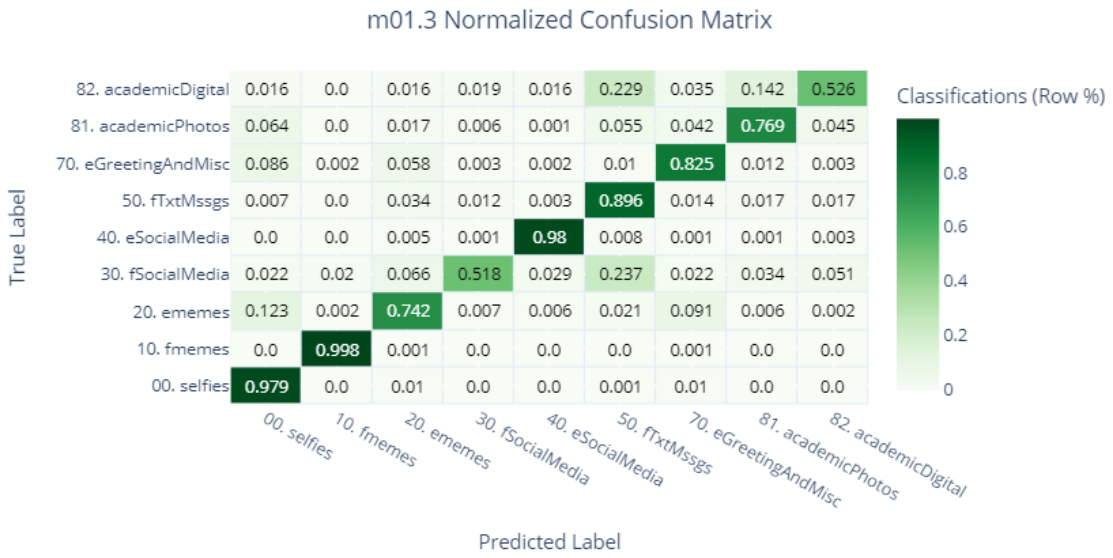


Figure 76. “m01.3” test set’s normalized confusion matrix.

5.1.4 VCNN: “m01.4”

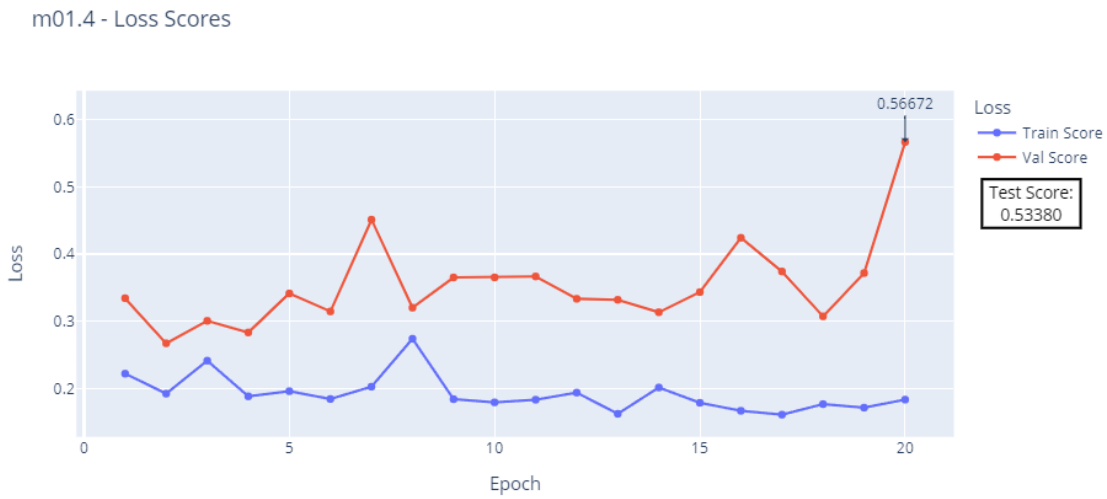


Figure 77. “m01.4” loss: train, validation, and test scores.

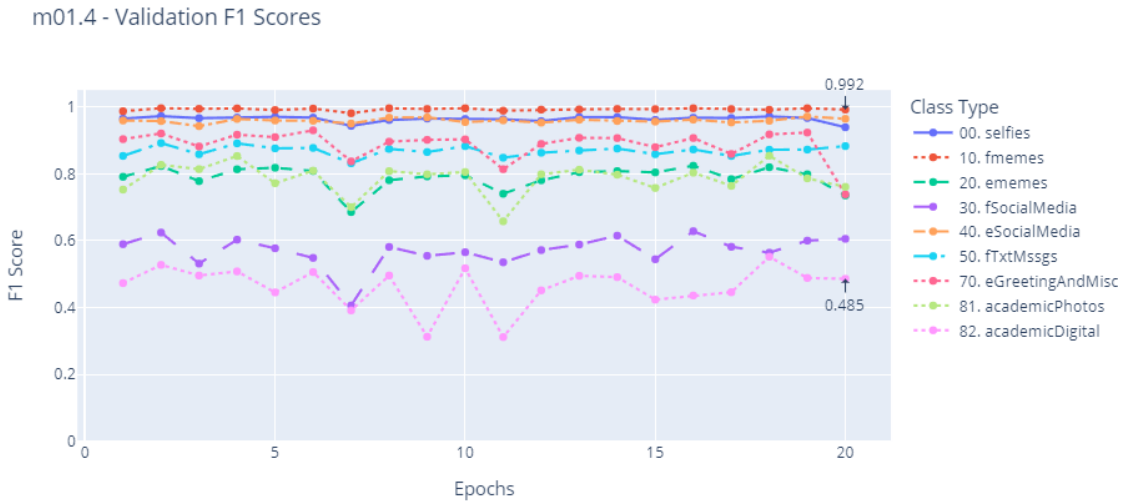


Figure 78. “m01.4” validation f1-scores.

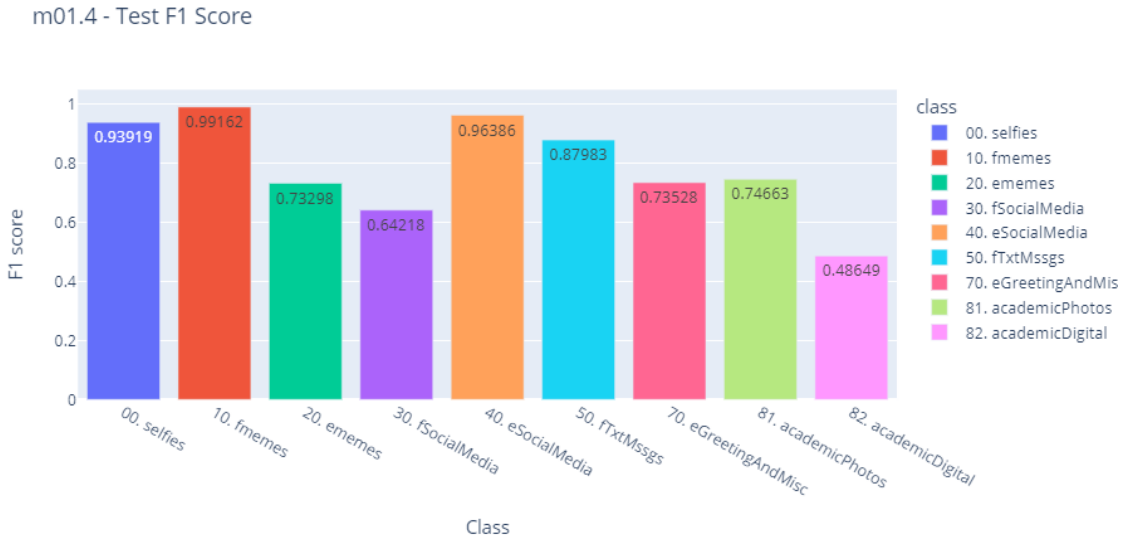


Figure 79. “m01.4” test f1-scores.

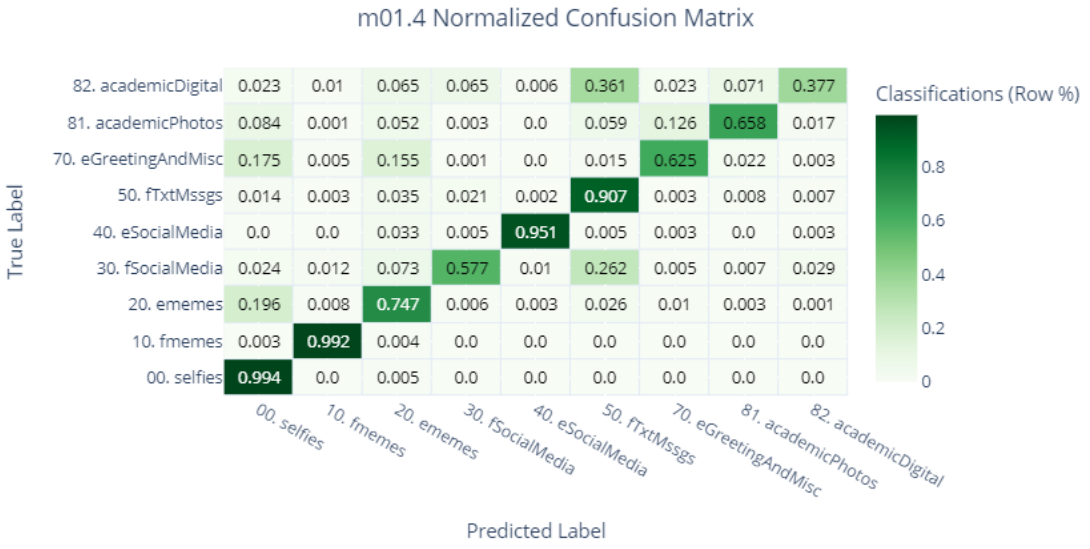


Figure 80. “m01.4” test set’s normalized confusion matrix.

5.1.5 VCNN: “m01.6” on “v1.1” dataset

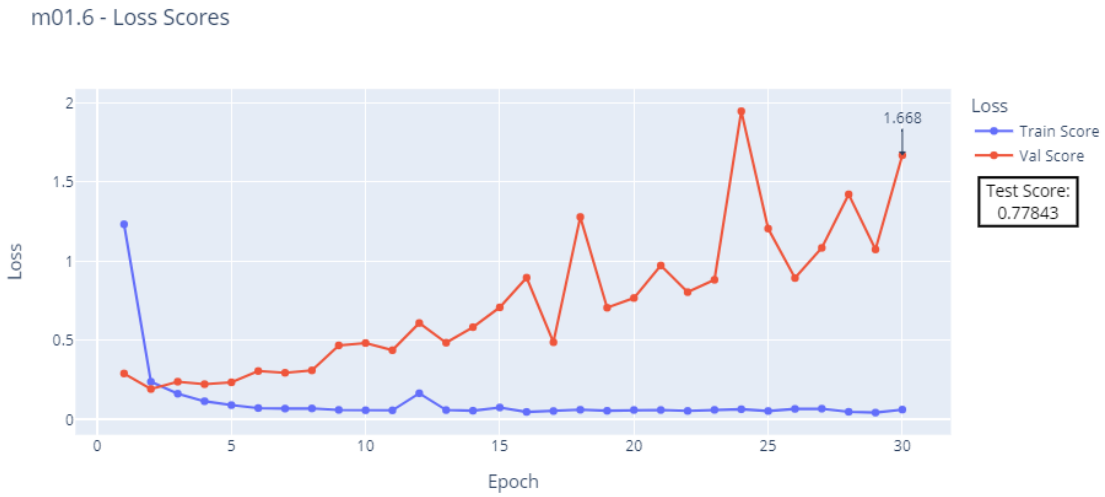


Figure 81. “m01.6” loss: train, validation, and test scores.

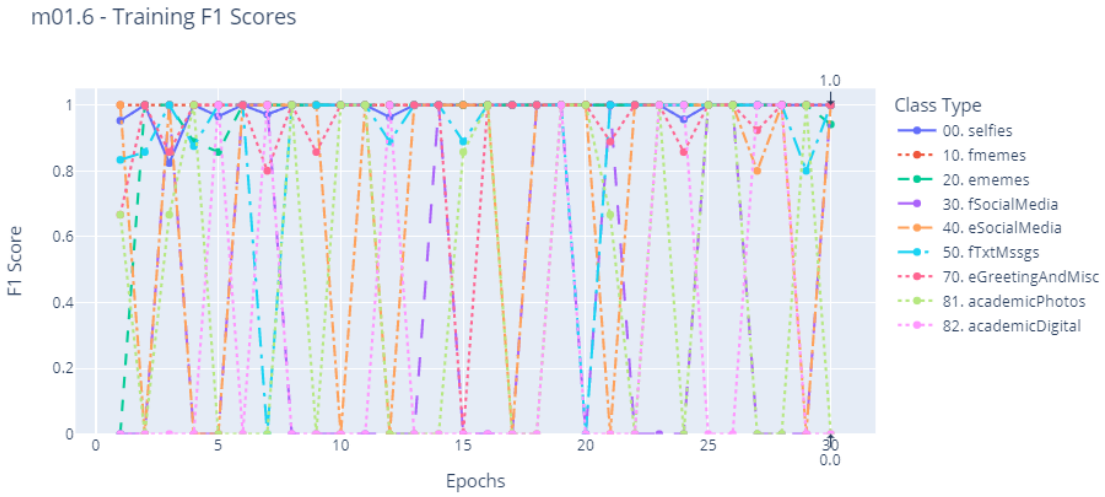


Figure 82. “m01.6” training f1-scores.

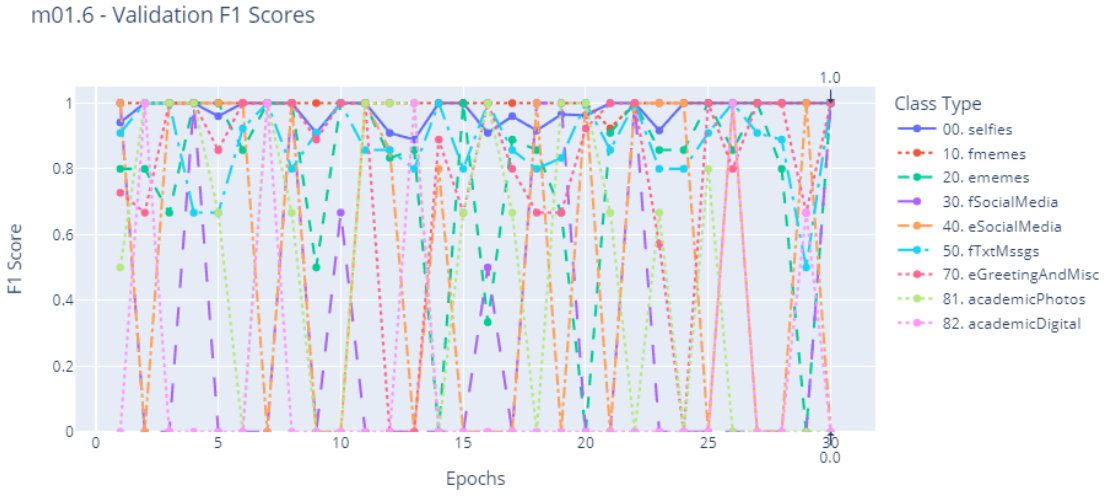


Figure 83. “m01.6” validation f1-scores.

Figure 84. “m01.6” test f1-scores.

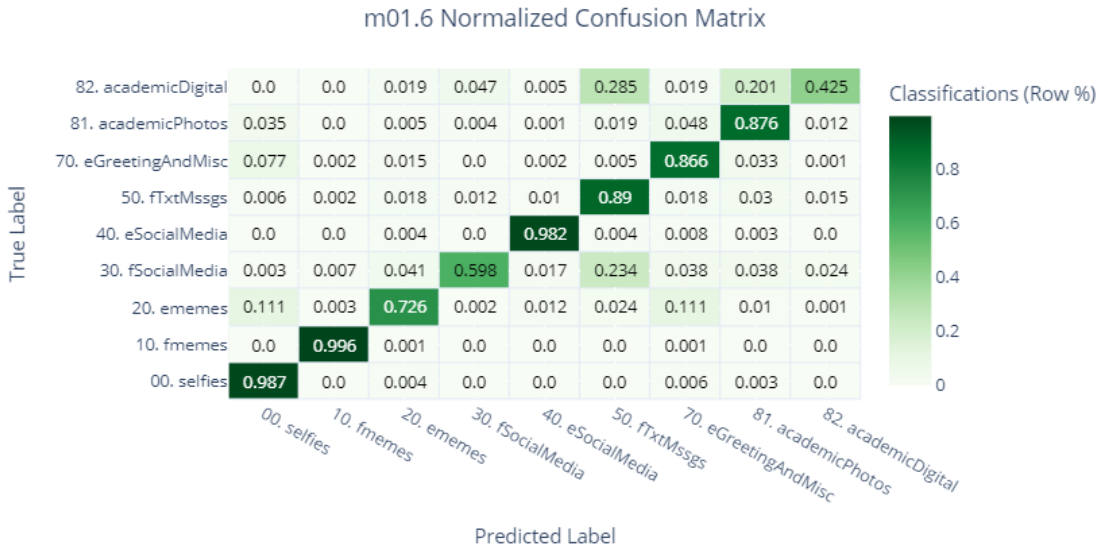


Figure 85. “m01.6” test set’s normalized confusion matrix.

5.1.6 MCNN: “m02”

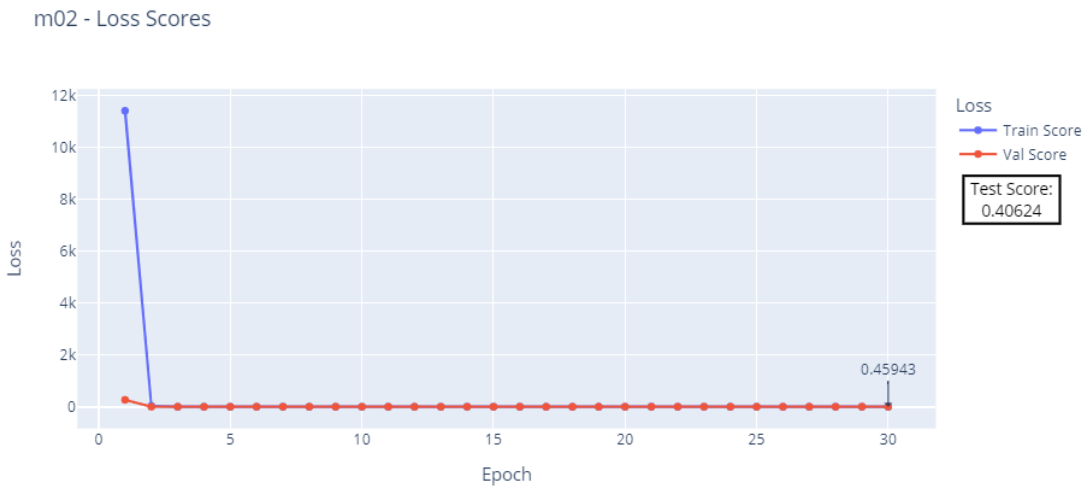


Figure 86. “m02” loss: train, validation, and test scores.

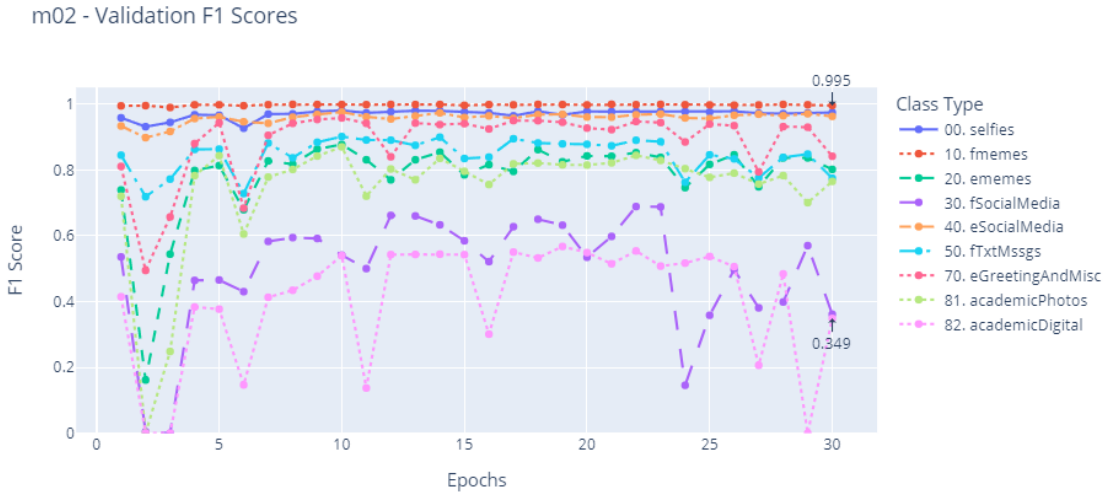


Figure 87. “m02” validation f1-scores.

Figure 88. “m02” test f1-scores.

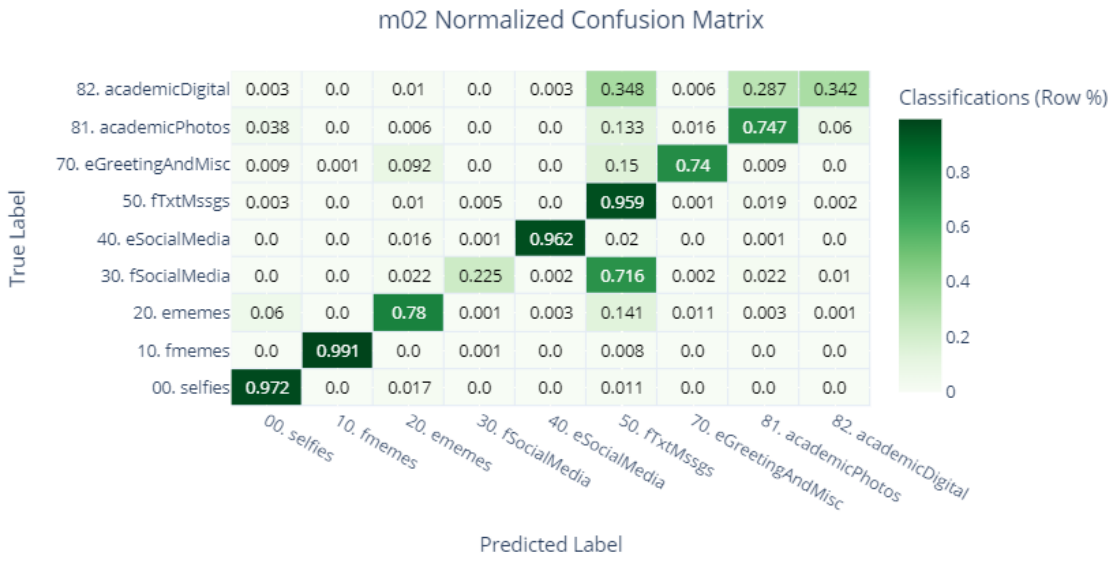


Figure 89. “m02” test set’s normalized confusion matrix.

5.1.7 MCNN: “m02.1”

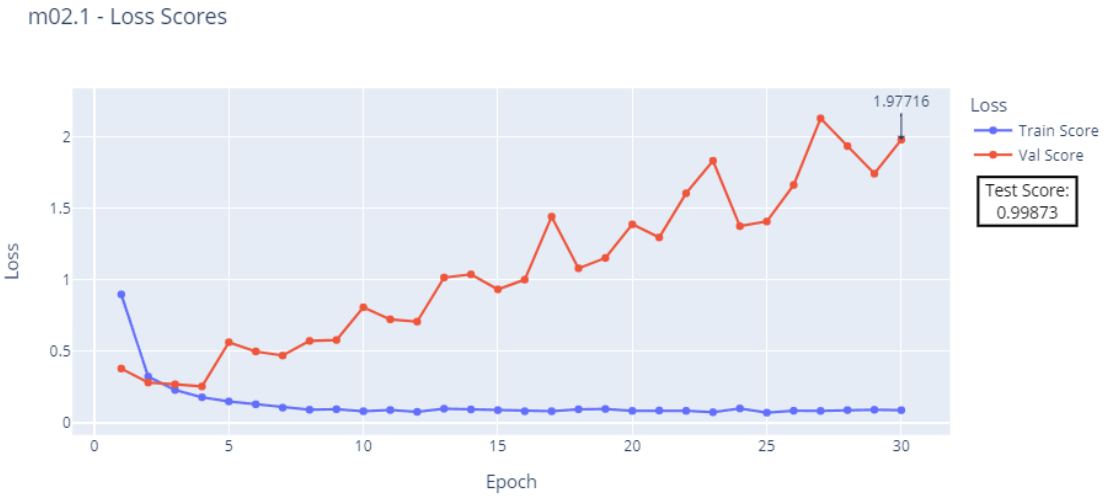


Figure 90. “m02.1” loss: train, validation, and test scores.

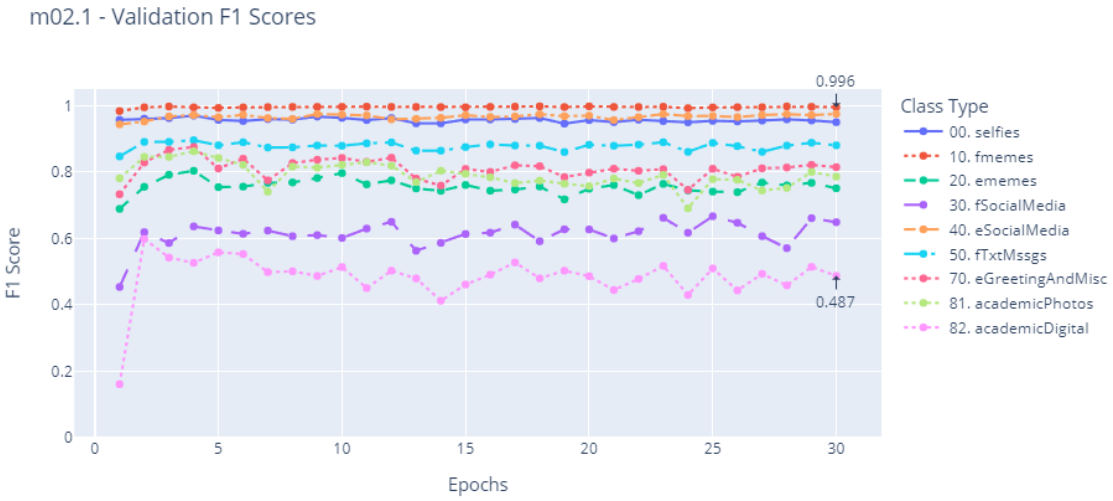


Figure 91. “m02.1” validation f1-scores.

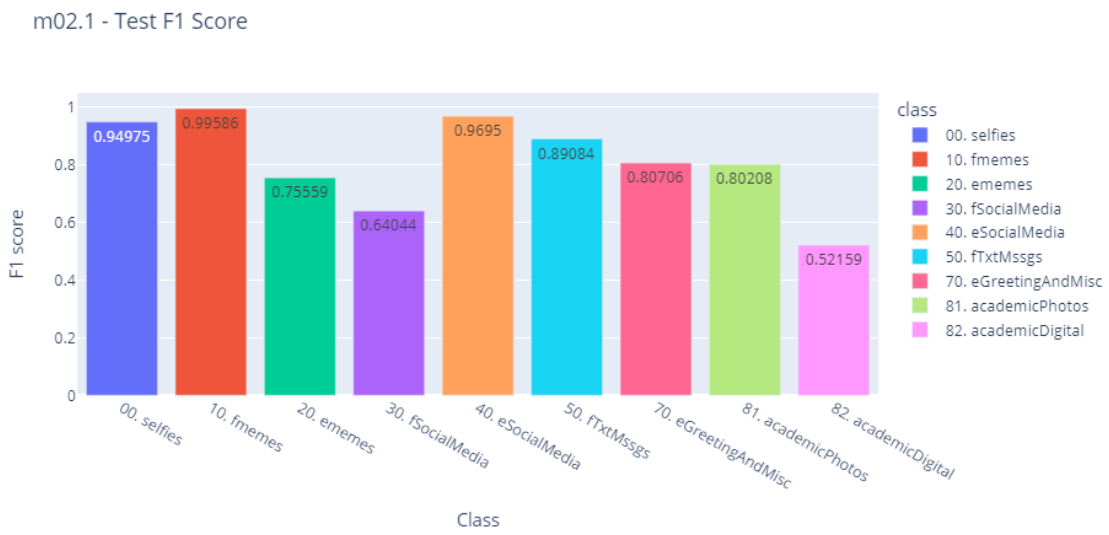


Figure 92. “m02.1” test f1-scores.

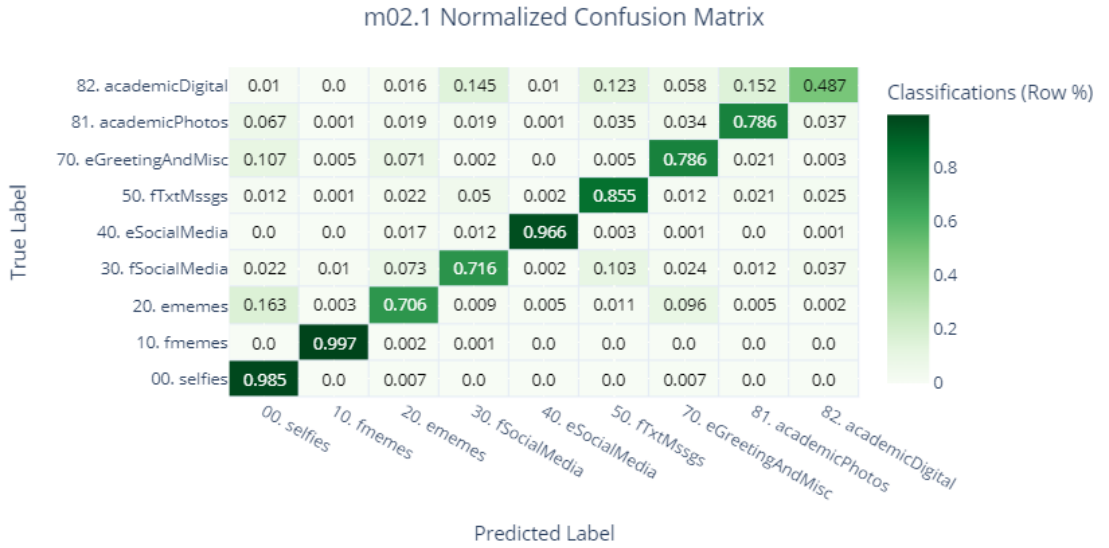


Figure 93. “m02.1” test set’s normalized confusion matrix.

5.1.8 MCNN: “m02.2”

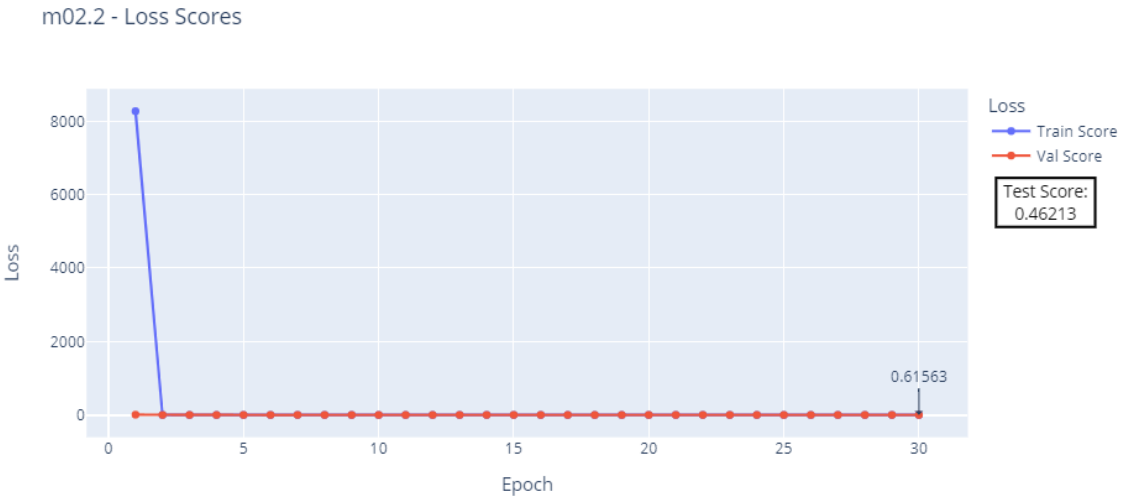


Figure 94. “m02.2” loss: train, validation, and test scores.

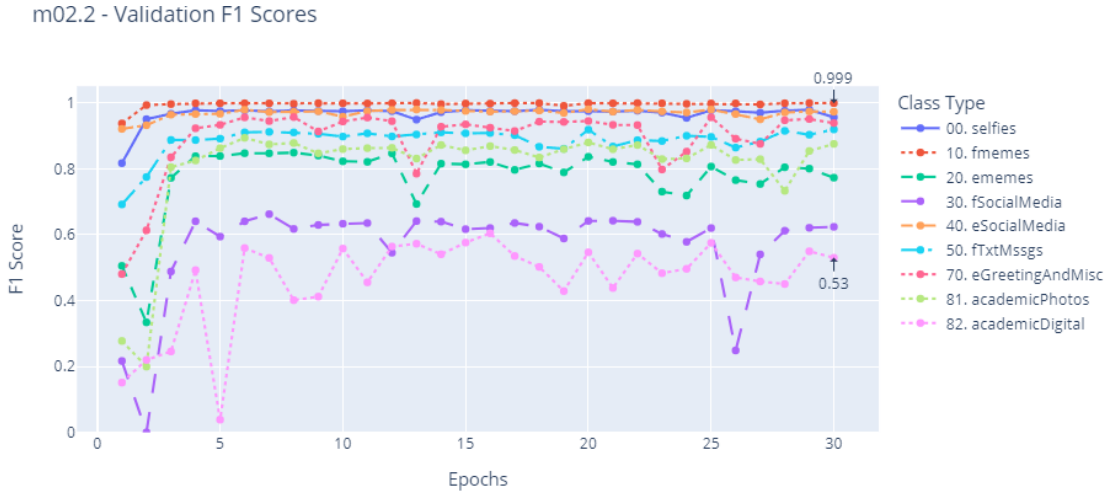


Figure 95. “m02.2” validation f1-scores.

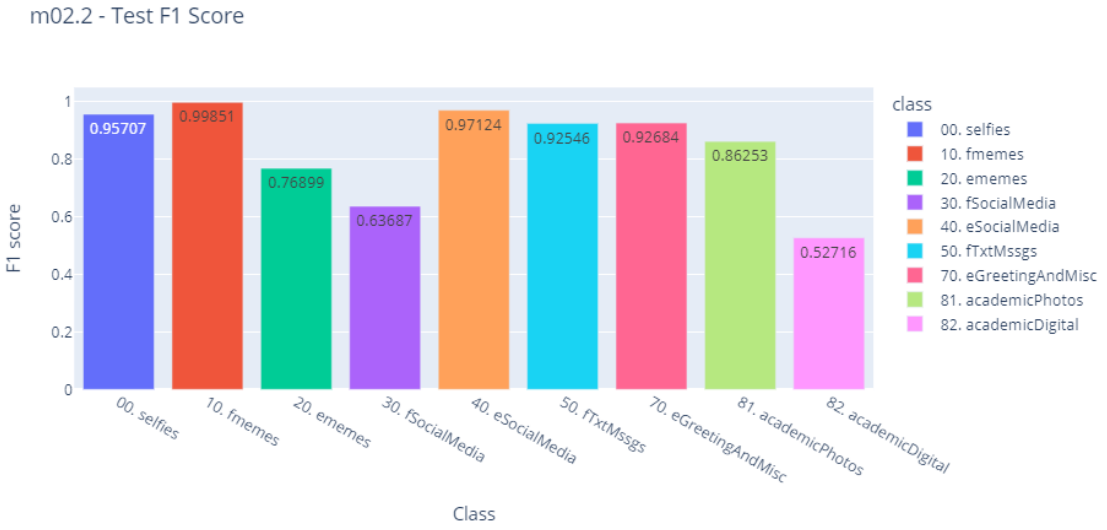


Figure 96. “m02.2” test f1-scores.

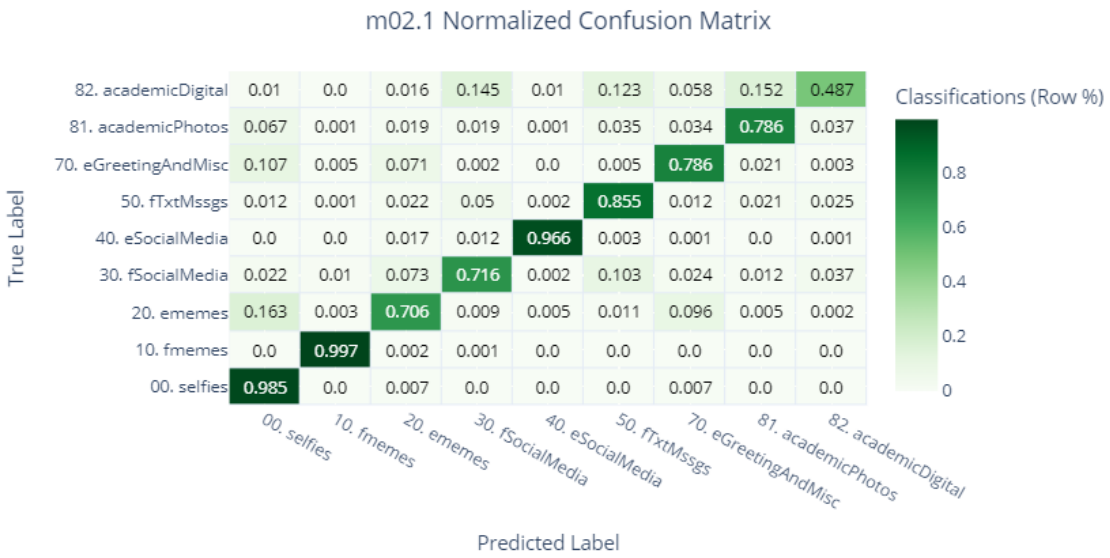


Figure 97. “m02.2” test set’s normalized confusion matrix.

5.1.9 MCNN: “m02.3”

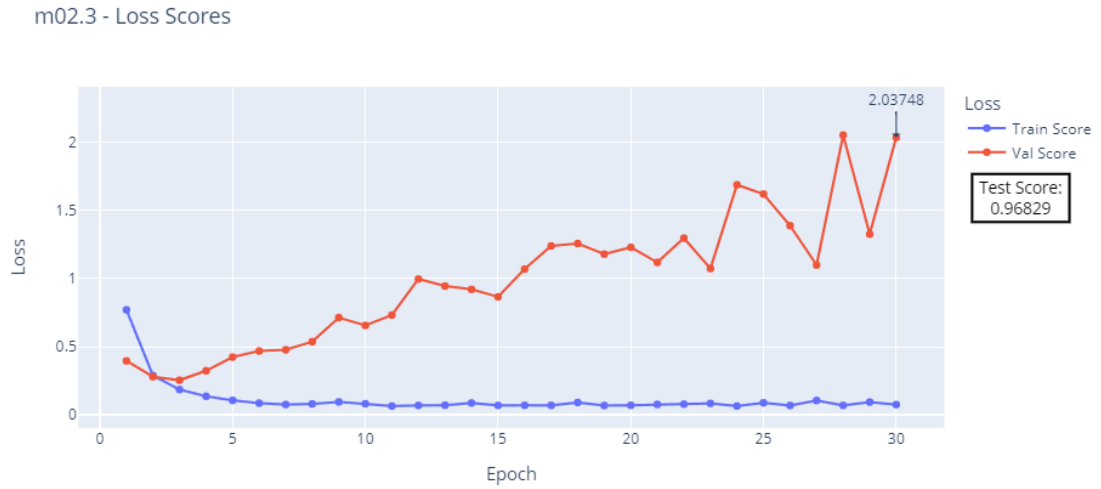


Figure 98. “m02.3” loss: train, validation, and test scores.

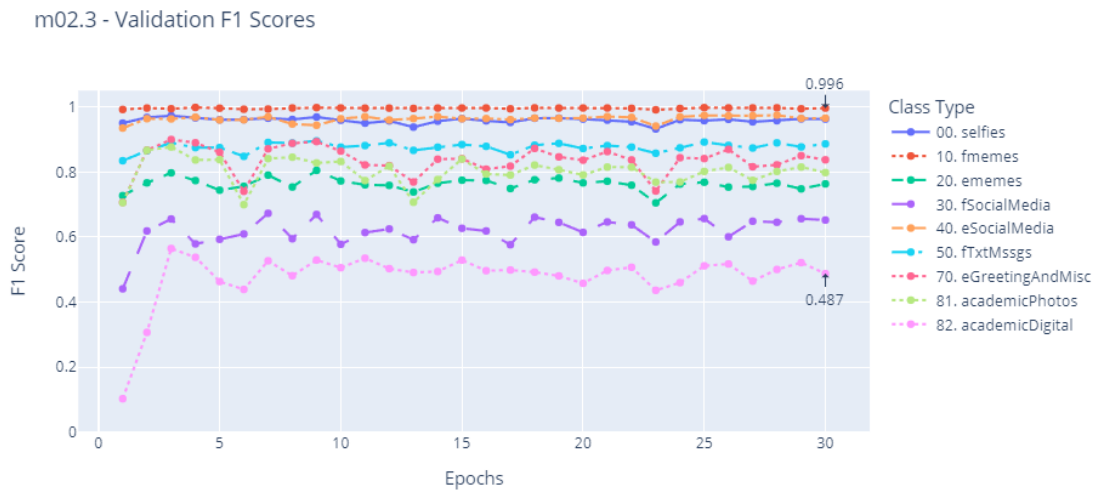


Figure 99. “m02.3” validation f1-scores.

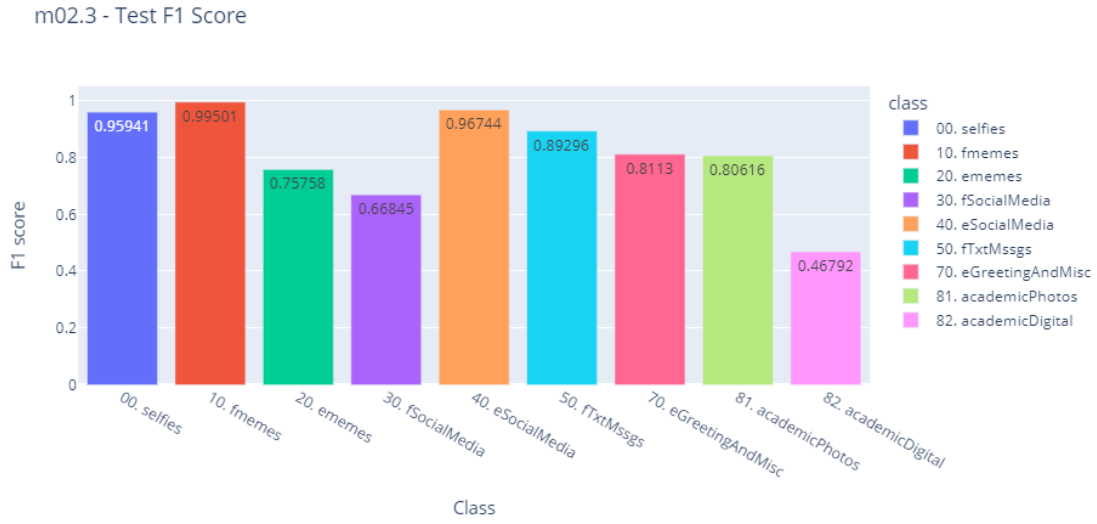


Figure 100. “m02.3” test f1-scores.

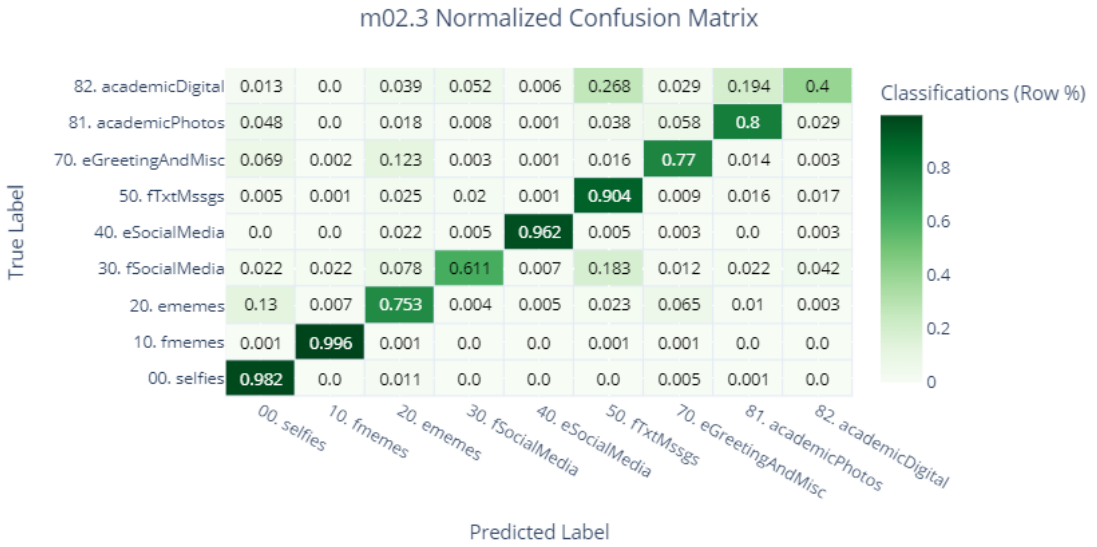


Figure 101. “m02.3” test set’s normalized confusion matrix.

5.1.10 MCNN: “m02.5”

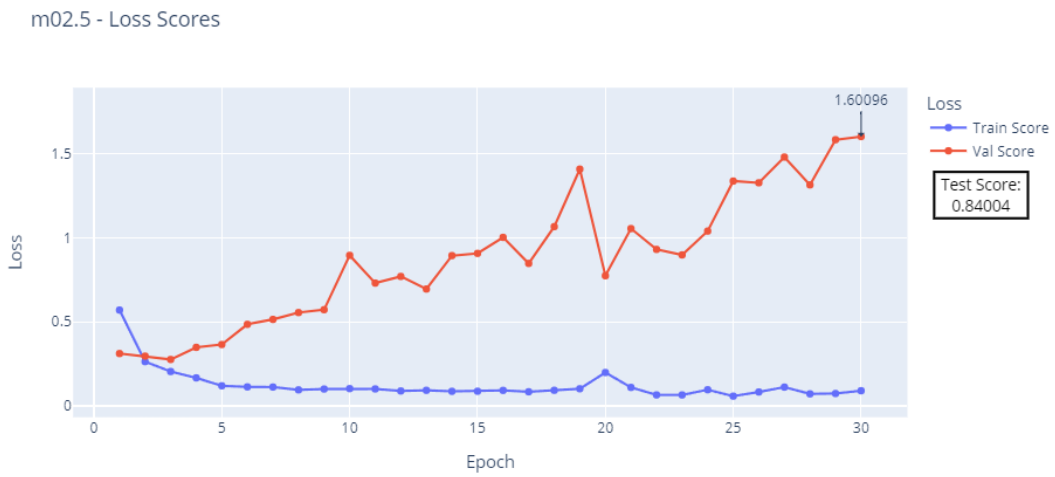


Figure 102. “m02.5” loss: train, validation, and test scores.

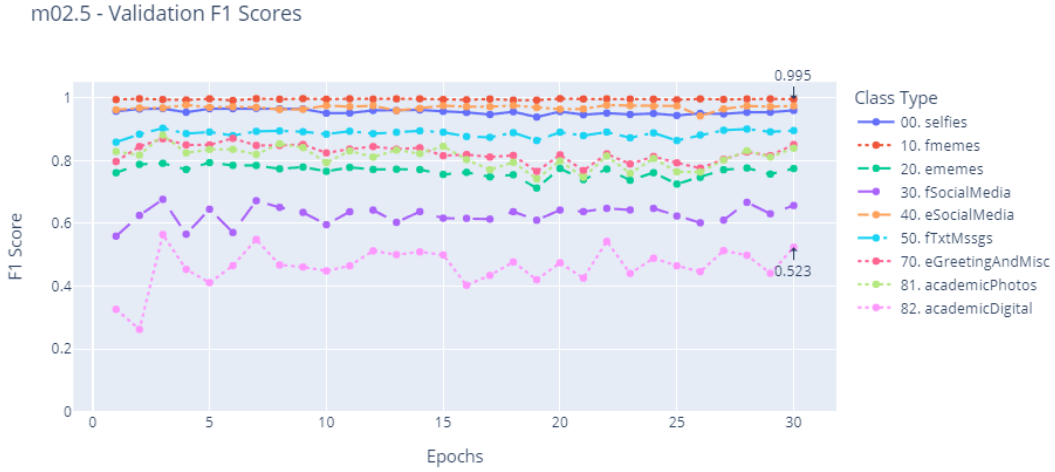


Figure 103. “m02.5” validation f1-scores.

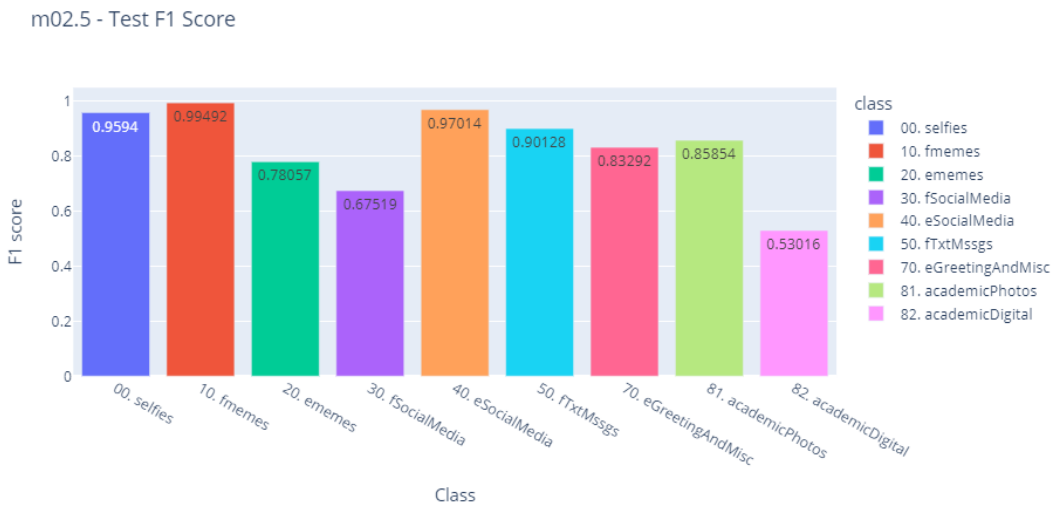


Figure 104. “m02.5” test f1-scores.

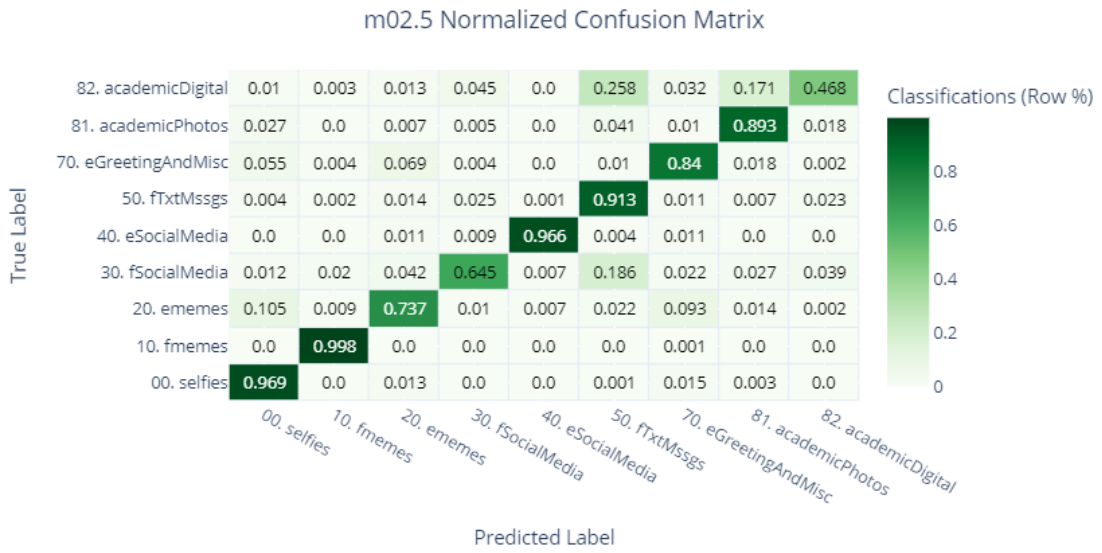


Figure 105. “m02.5” test set’s normalized confusion matrix.

5.1.11 HCNN: “m03”

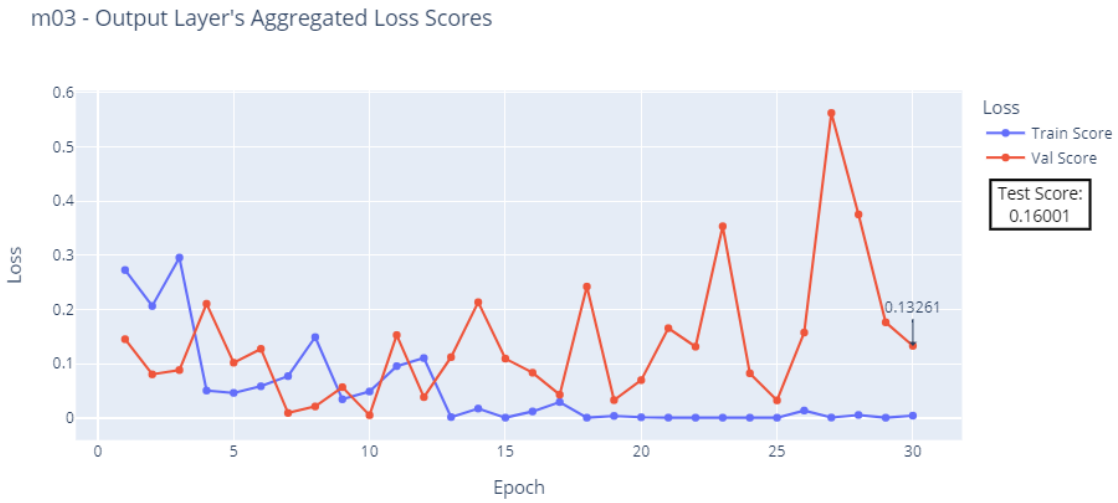


Figure 106. “m03” aggregated loss: train, validation, and test scores.

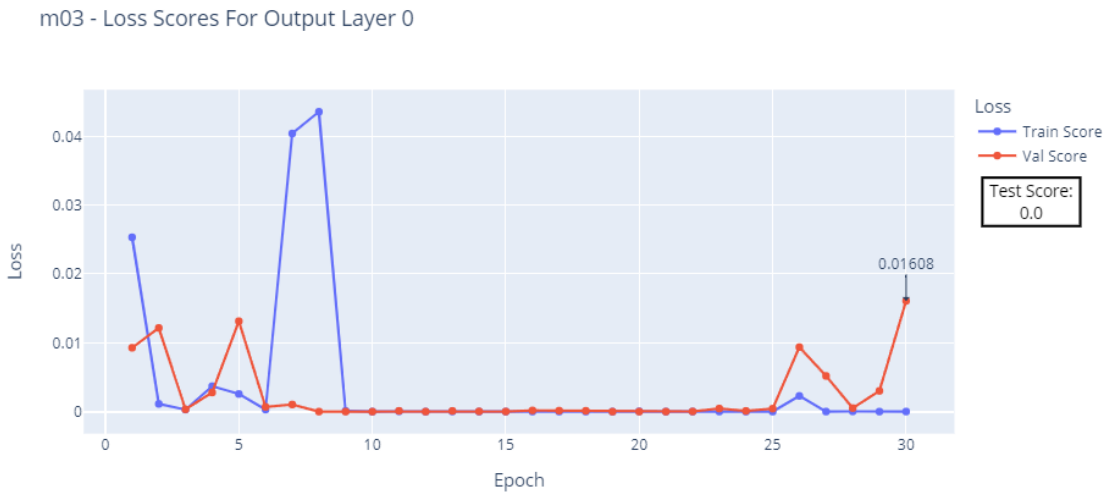


Figure 107. “m03” output layer 0’s loss: train, validation, and test scores

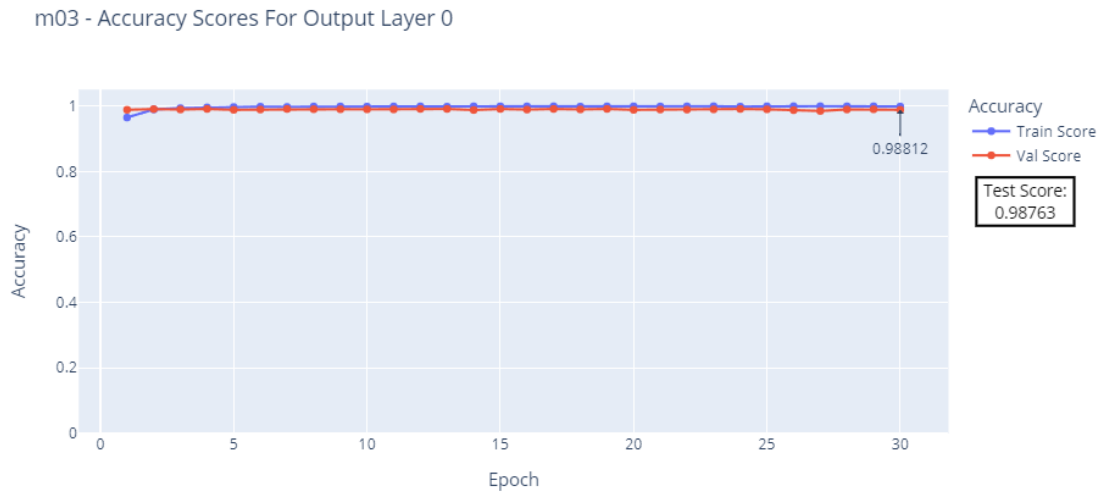


Figure 108. “m03” output layer 0’s accuracy: train, validation, and test scores.

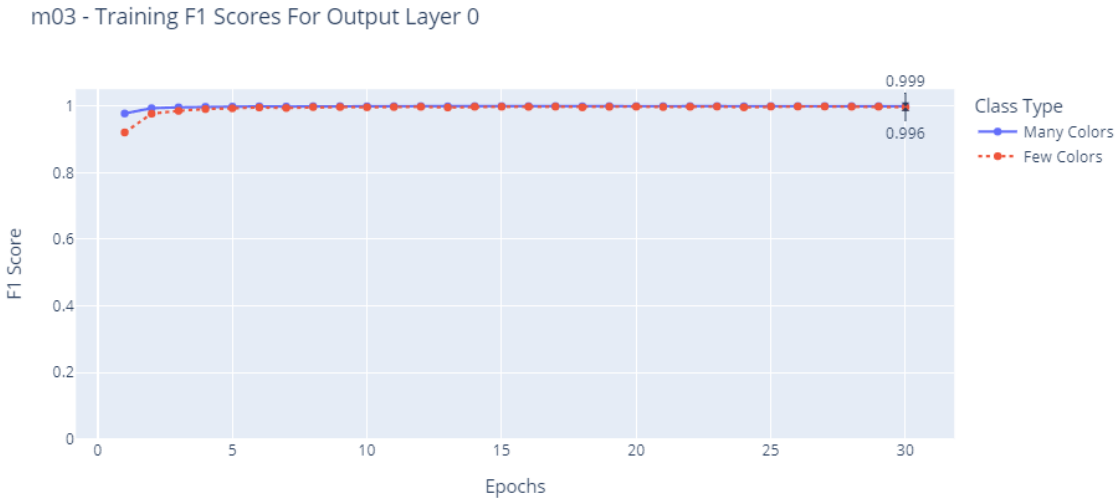


Figure 109. “m03” output layer 0’s train f1-scores.

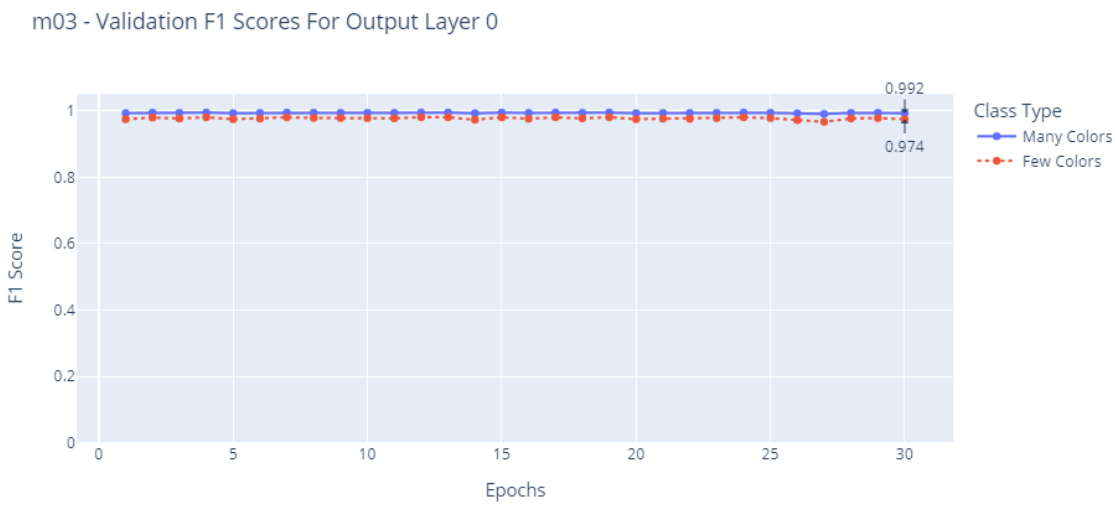


Figure 110. “m03” output layer 0’s validation f1-scores.

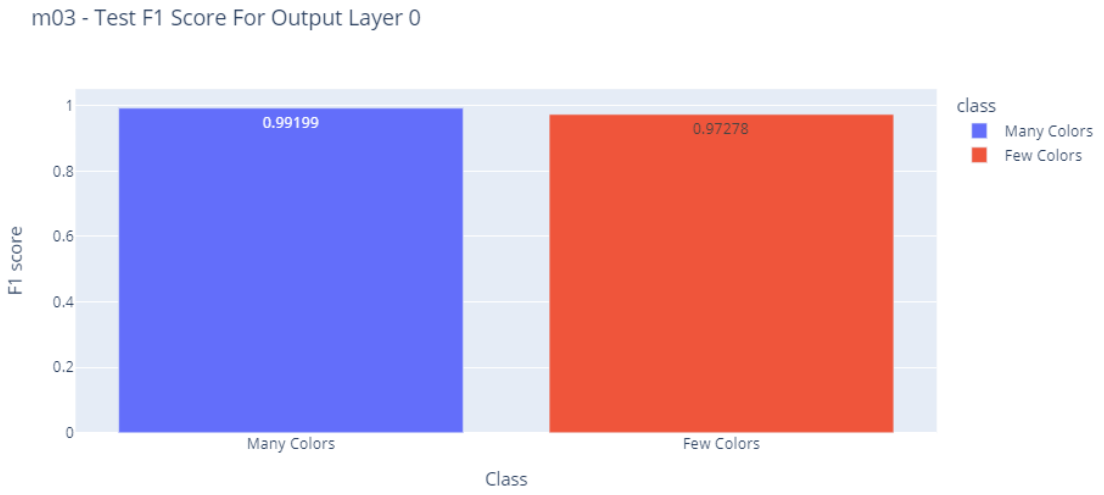


Figure 111. “m03” output layer 0’s test f1-scores.

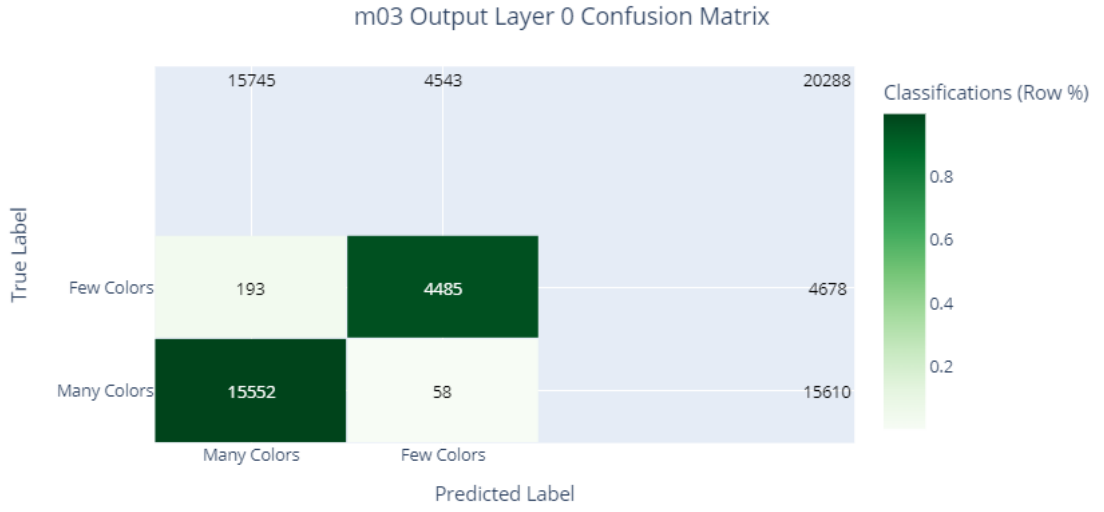


Figure 112. “m03” output layer 0’s test set’s raw confusion matrix.

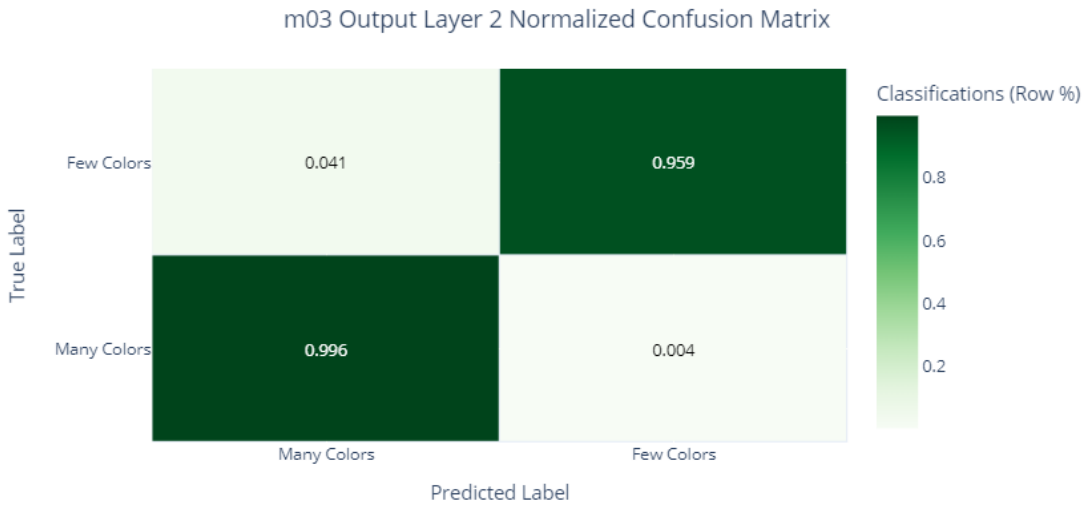


Figure 113. “m03” output layer 0’s test set’s normalized confusion matrix.

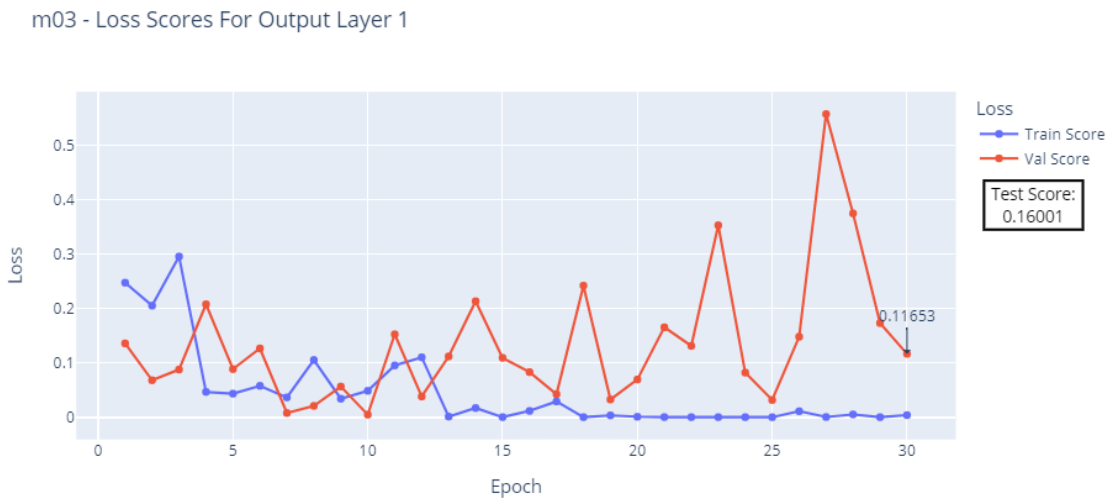


Figure 114. “m03” output layer 1’s loss: train, validation, and test scores.

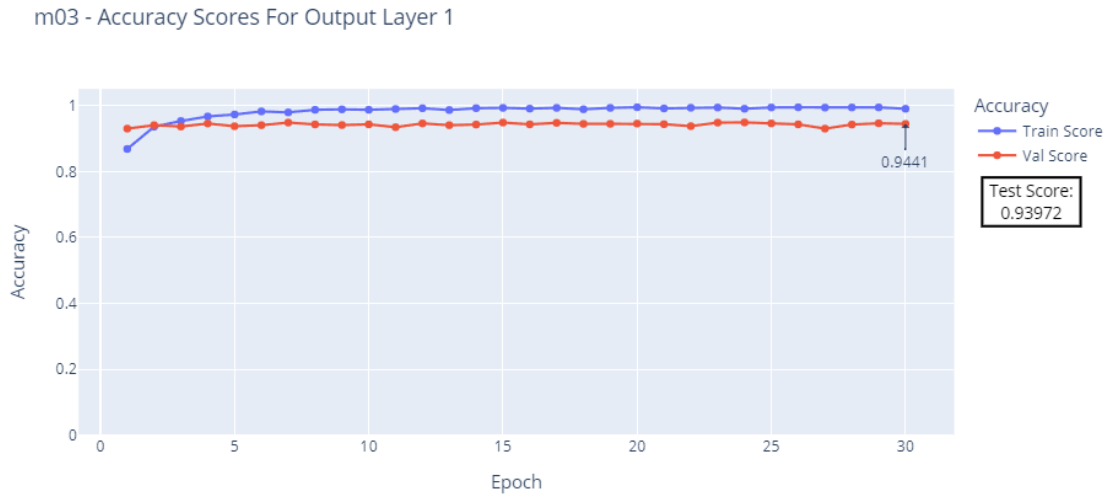


Figure 115. “m03” output layer 1’s accuracy: train, validation, and test scores.

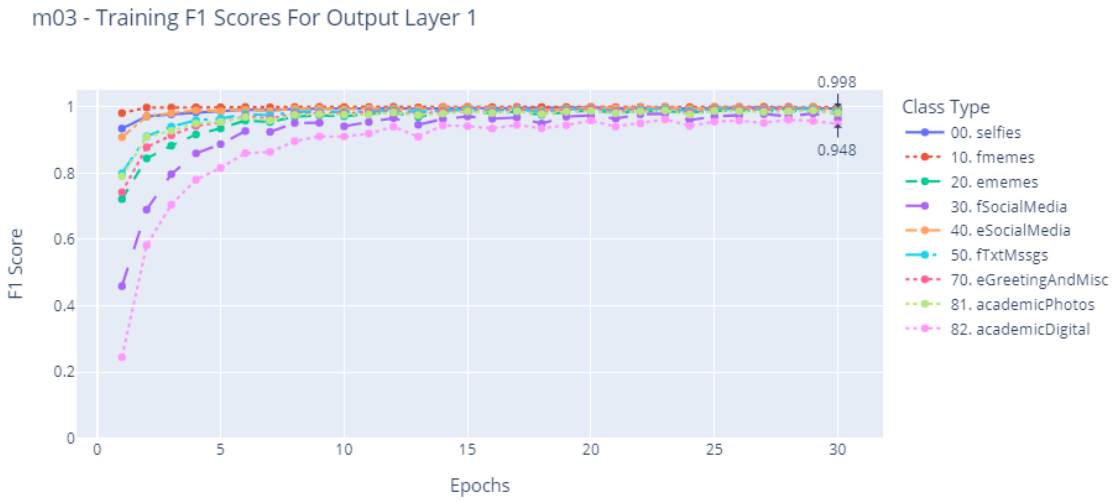


Figure 116. “m03” output layer 1’s train f1-scores.

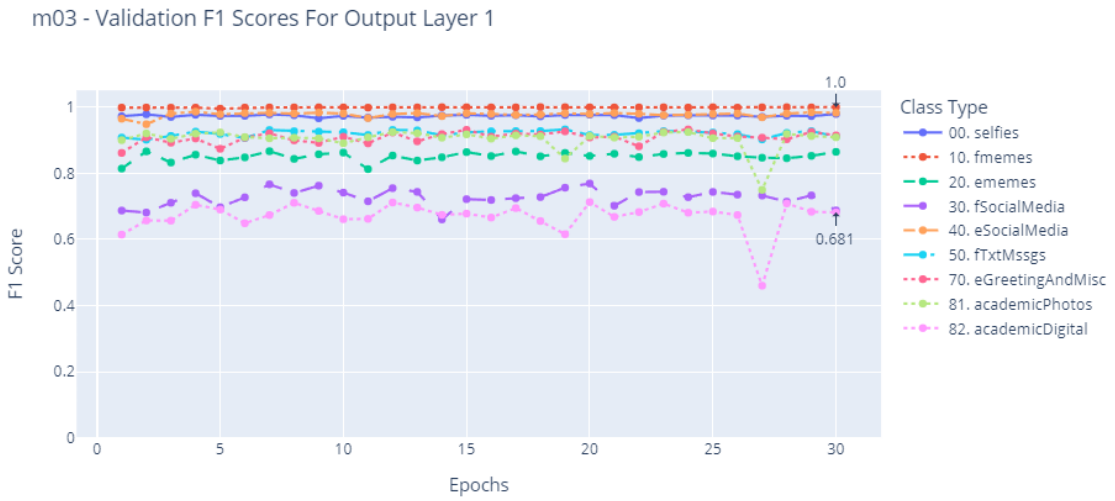


Figure 117. “m03” output layer 1’s validation f1-scores.

Figure 118. “m03” output layer 1’s test f1-scores.

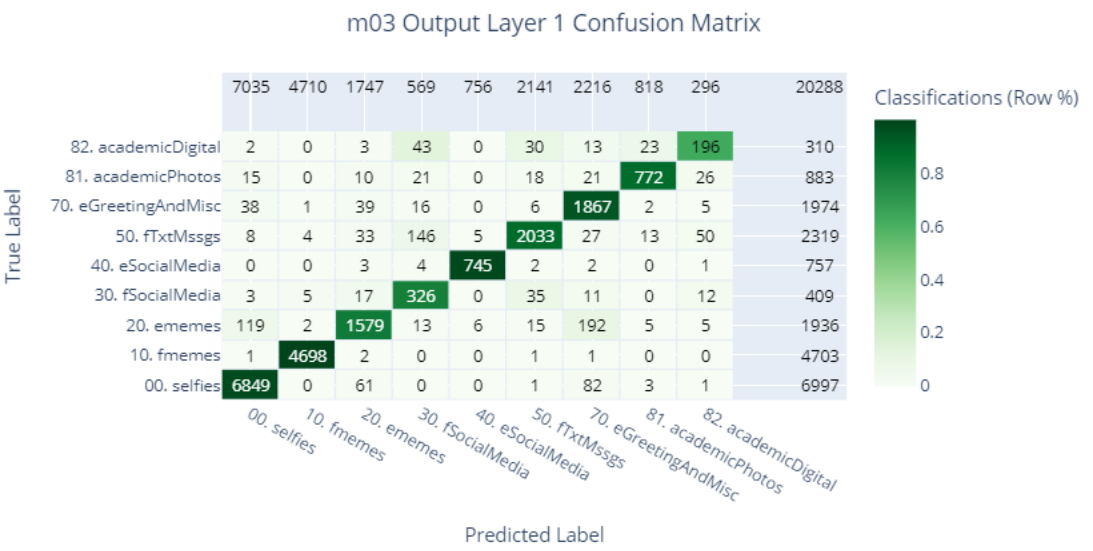


Figure 119. “m03” output layer 1’s test set’s raw confusion matrix.

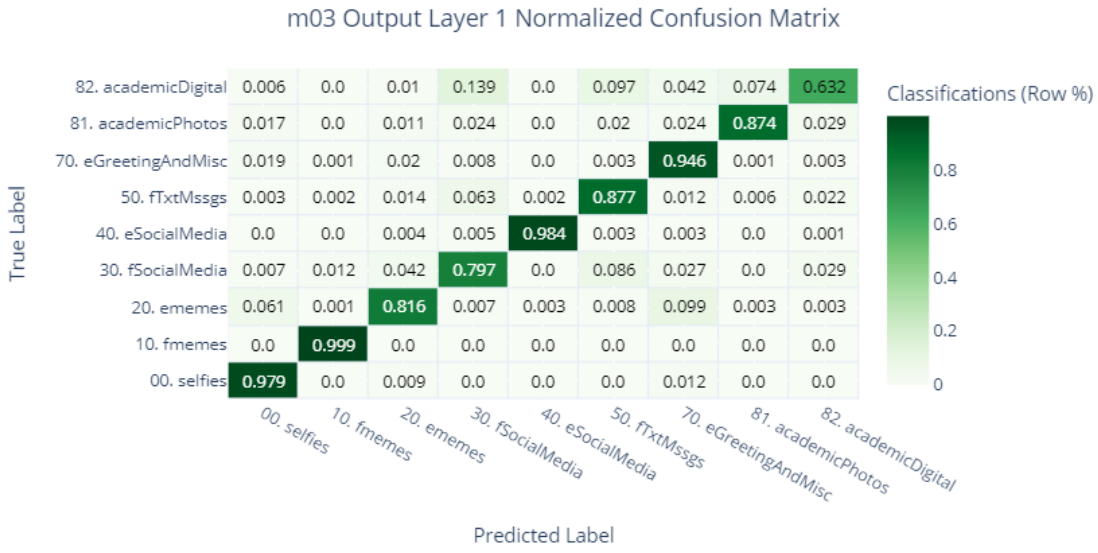


Figure 120. “m03” output layer 1’s test set’s normalized confusion matrix.

5.1.12 MCNN: “m04”

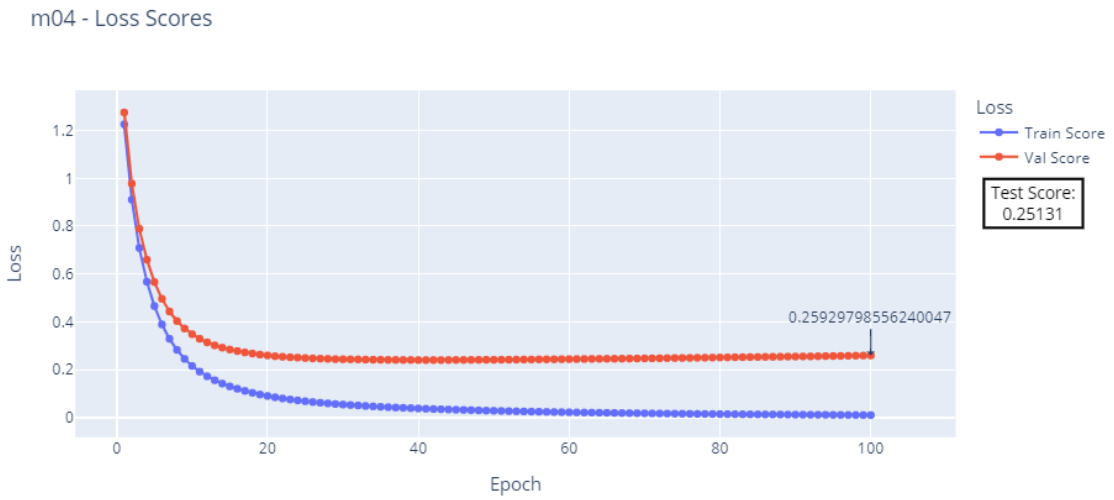


Figure 121. “m04” loss: train, validation, and test scores.

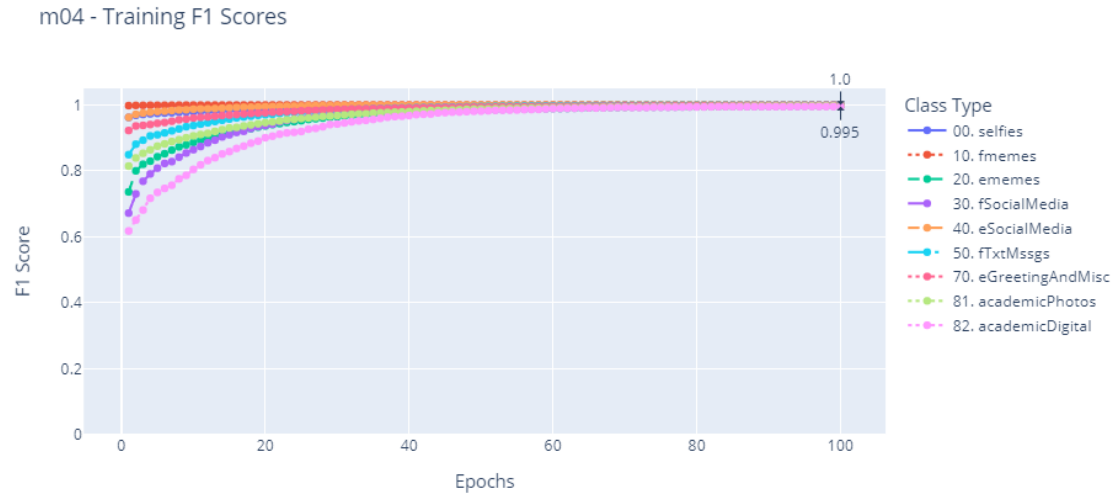


Figure 122. “m04” training f1-scores.

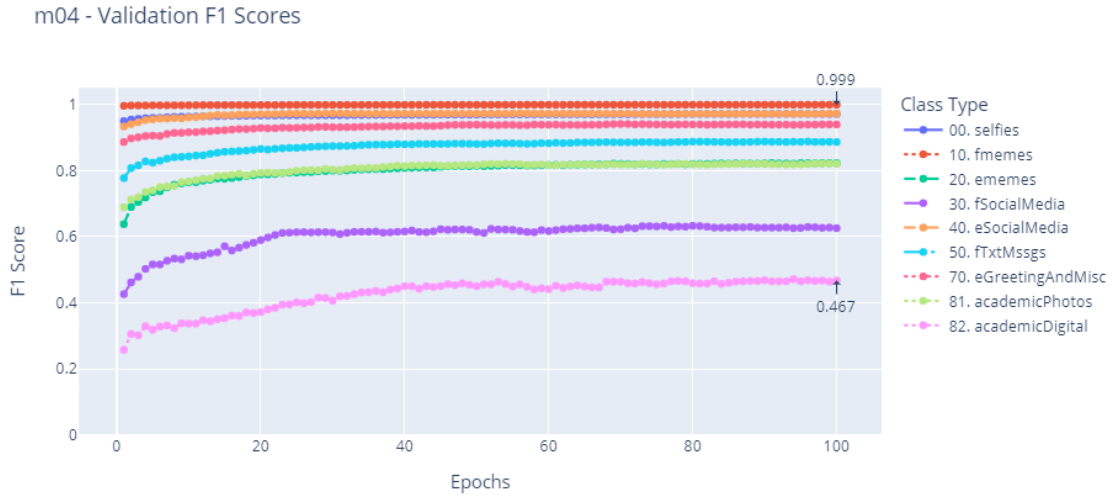


Figure 123. “m04” validation f1-scores.

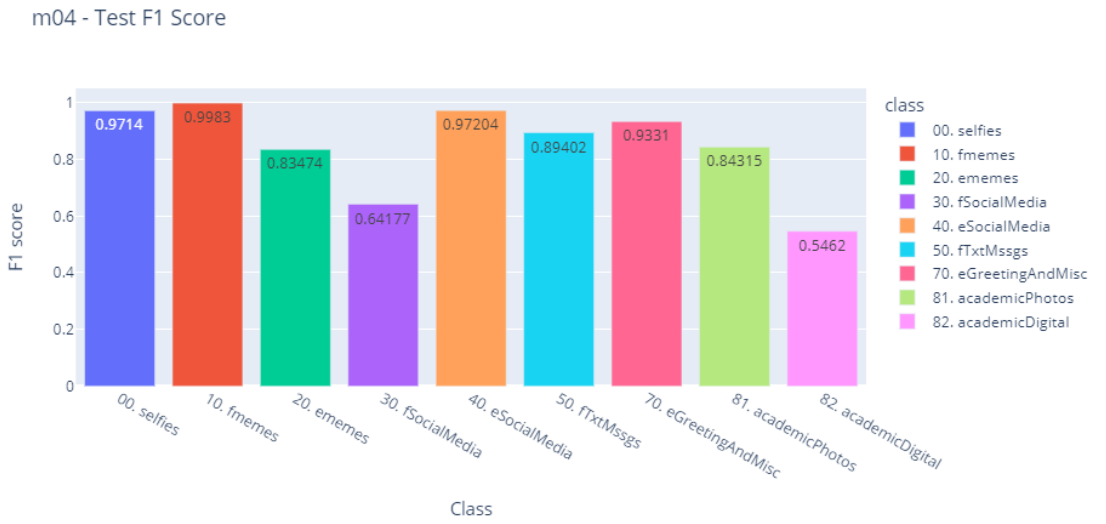


Figure 124. “m04” test f1-scores.

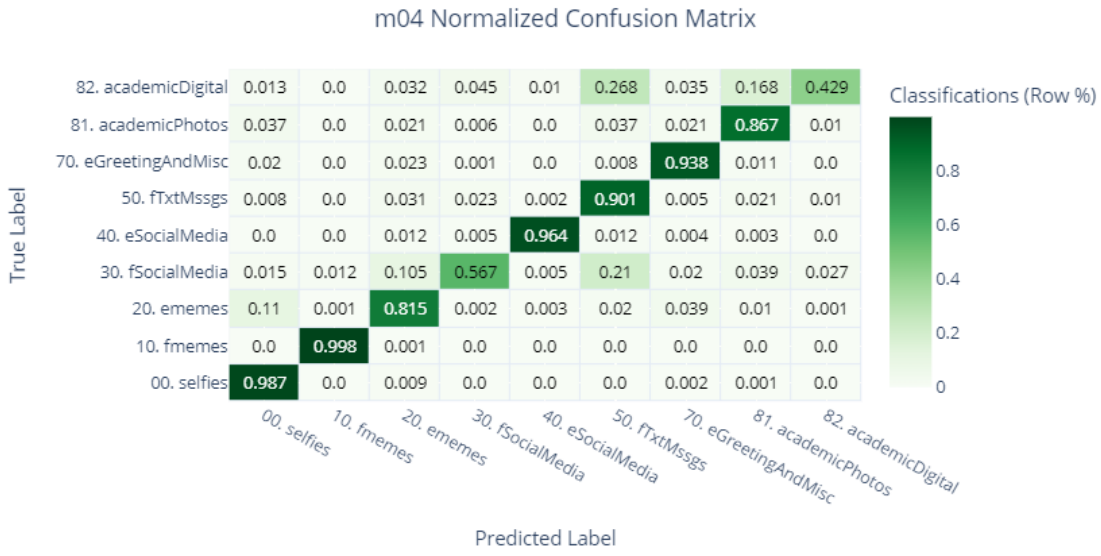


Figure 125. “m04” test set’s normalized confusion matrix.

5.1.13 MCNN: “m04.1”

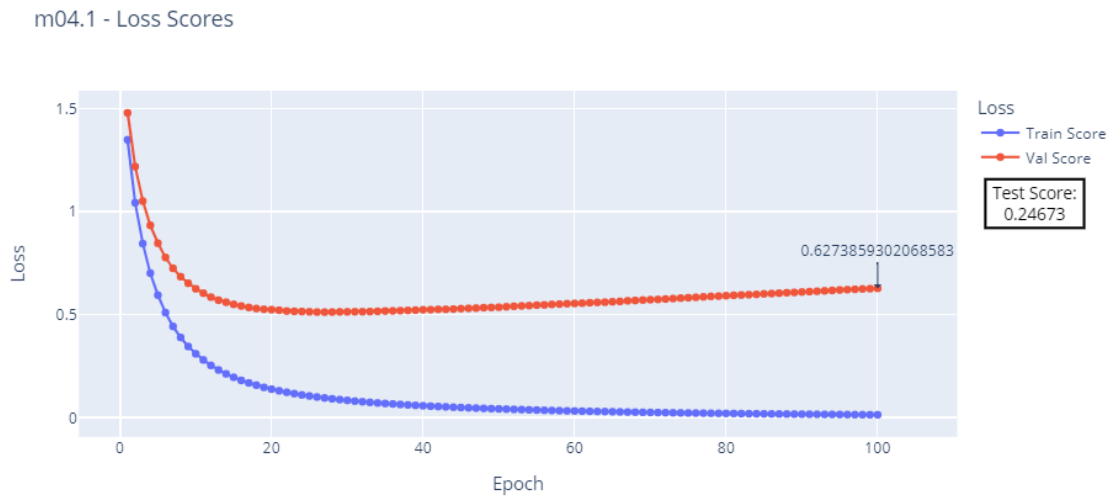


Figure 126. “m04.1” loss: train, validation, and test scores.

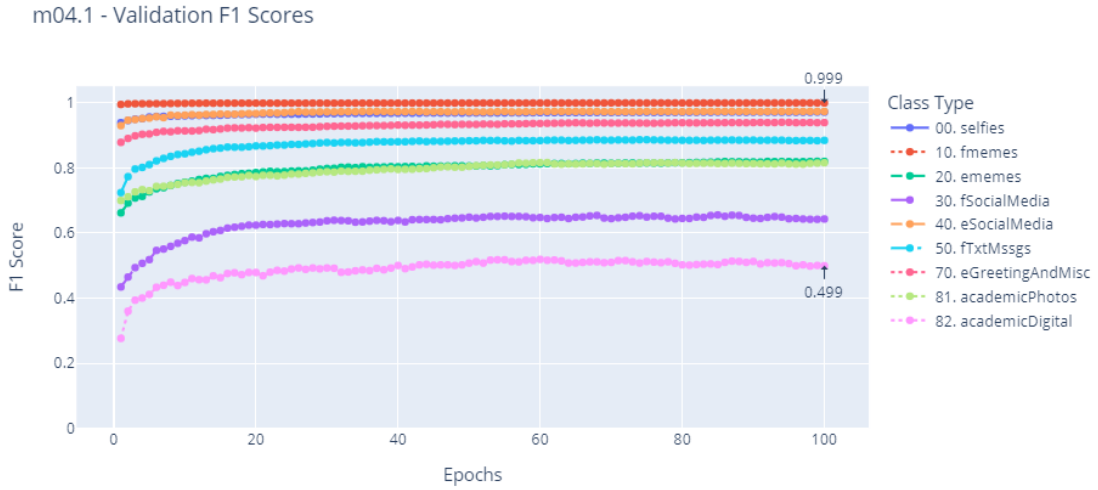


Figure 127. “m04.1” validation f1-scores.

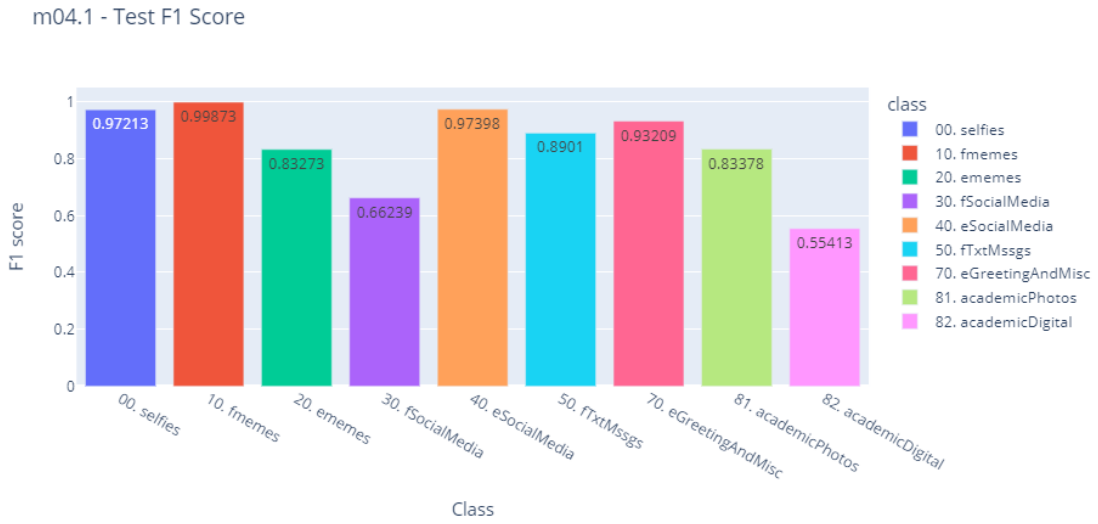


Figure 128. “m04.1” test f1-scores.

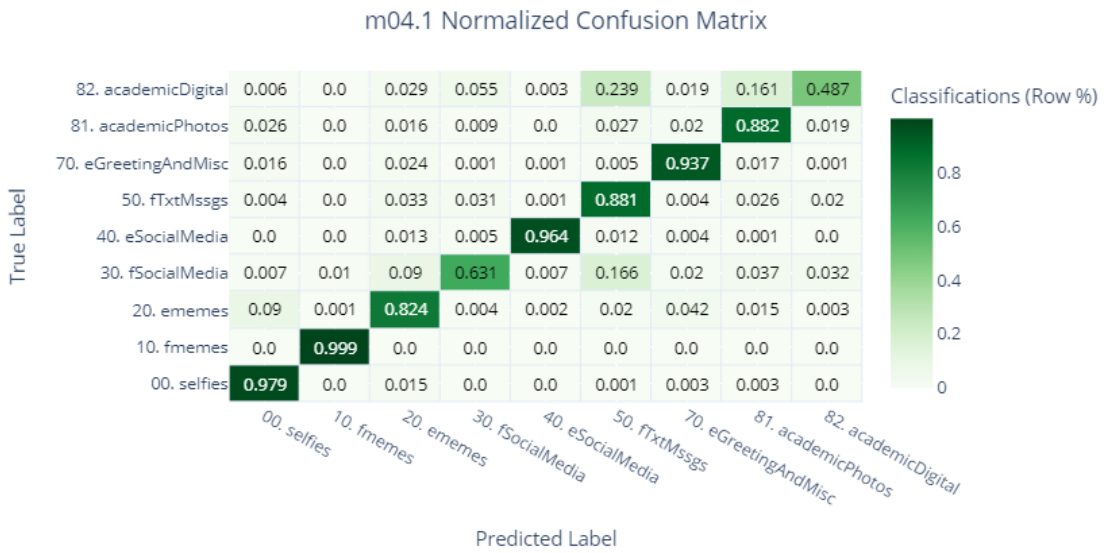


Figure 129. “m04.1” test set’s normalized confusion matrix.

5.1.14 XCNN: “m04.2”

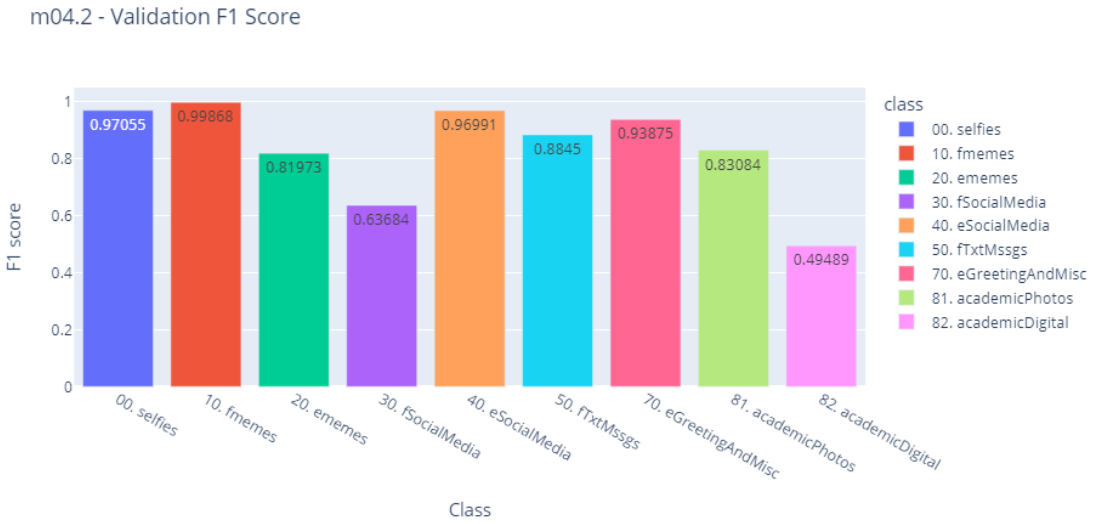


Figure 130. “m04.2” validation f1-scores without epochs (i.e., direct prediction on validation set).

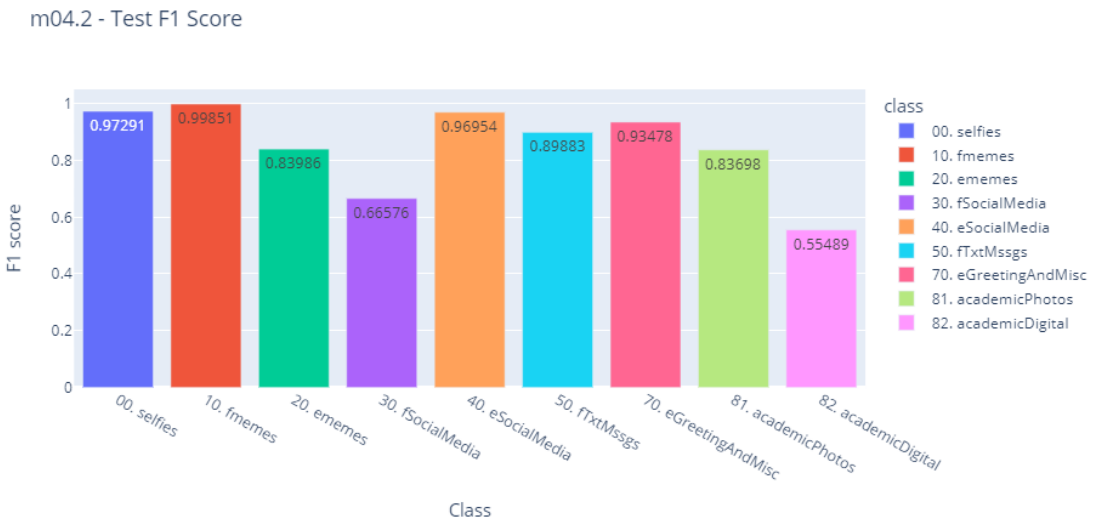


Figure 131. “m04.2” test f1-scores.

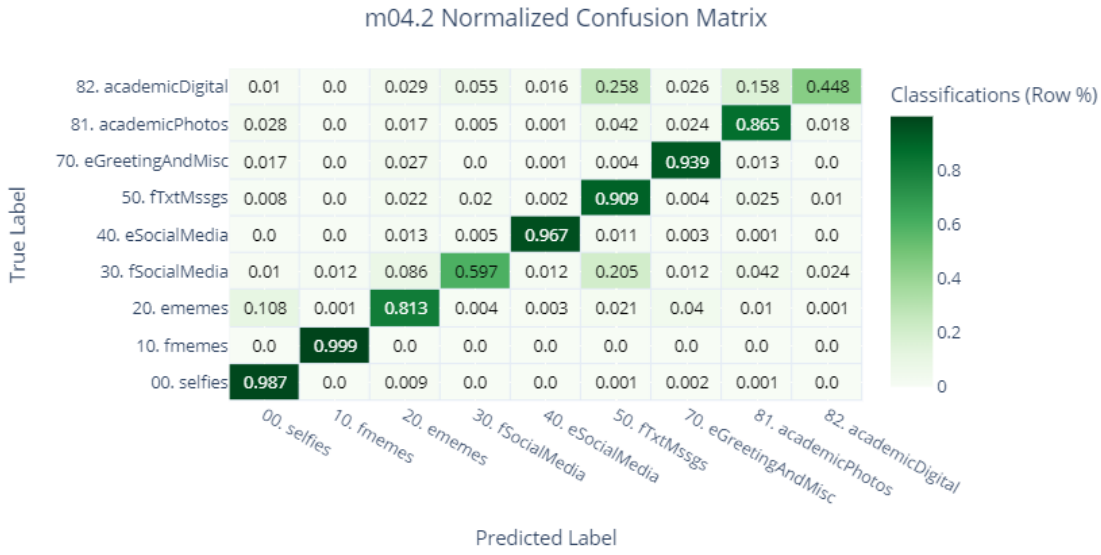


Figure 132. “m04.2” test set’s normalized confusion matrix.

5.2 Comparing Different Variants of the Same Model Type

This sub section will show test set’s f1 scores directly compared between variants of each model type, except for HCNN, as we’ve only trained one variant for this model by the name of “m03”. Now, shows the list of 6 main comparisons between models of the same type, and the rest of the visualizations in this section are tables and plots for each of these 6 comparisons.

```
model_comparisons = [
    # Comp 1: Batch Size Effect: `m01` vs `m01.1` `m01.3`
    ["01", "01.1", "01.3"],

    # Comp 2: `m01.1` (dataset v1) Vs `m01.6` (dataset v1.1)
    ["01.1", "01.6"],

    # Comp 3: Scaling colors features (m02.1) or not (m02)
    ["02", "02.1"],

    # Comp 4: effect of adding face metadata (m02.2) to color metadata (02)
    ["02", "02.2"],

    # Comp 5: effect of adding scaled face and text metadata (m02.5) to
    # scaled color metadata (m02.1)
    ["02.1", "02.5"],

    # Comp 6: effect of adding sample weights (m04.1) on
    # xgboost trained on m01.1 features (m04) then effect when
    # adding Bayesian optimization technique
    # on cross validated sets (m04.2)
    ["04", "04.1", "04.2"]
]
```

Figure 133. The variants chosen for comparison per model type make a total of 6 comparisons.

	class	best model	f1 score
1	00. selfies	1.1	0.96731
2	10. fmemes	1.1	0.99788
3	20. ememes	1.1	0.81519
4	30. fSocialMedia	1	0.63647
5	40. eSocialMedia	1.3	0.96552
6	50. fTxtMssgs	1.1	0.89475
7	70. eGreetingAndMisc	1.1	0.9199
8	81. academicPhotos	1.1	0.82326
9	82. academicDigital	1.3	0.55726

Table 3. Comparison 1: best model in each class.

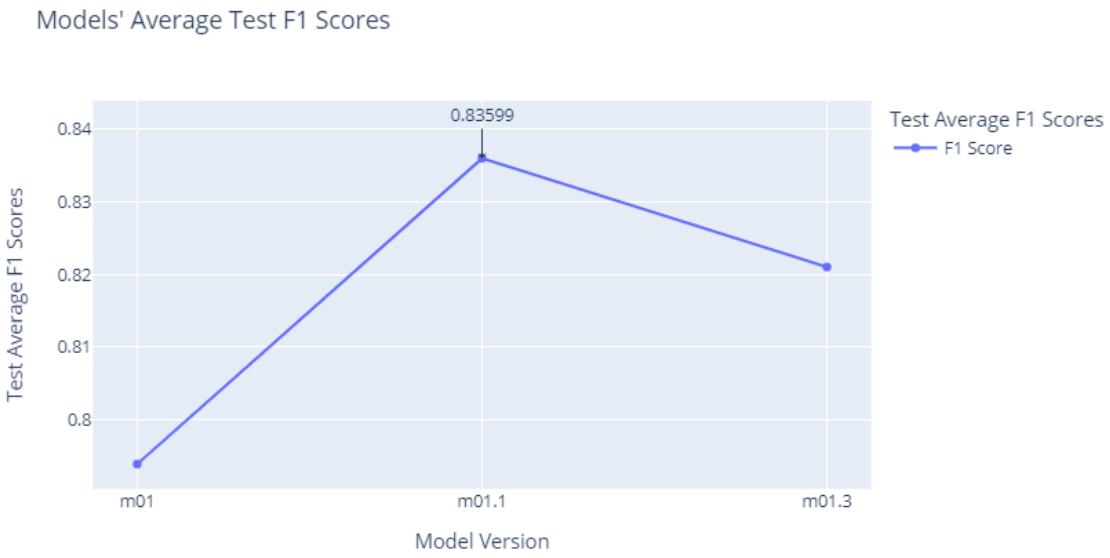


Figure 134. Comparison 1: average of test f1 scores.

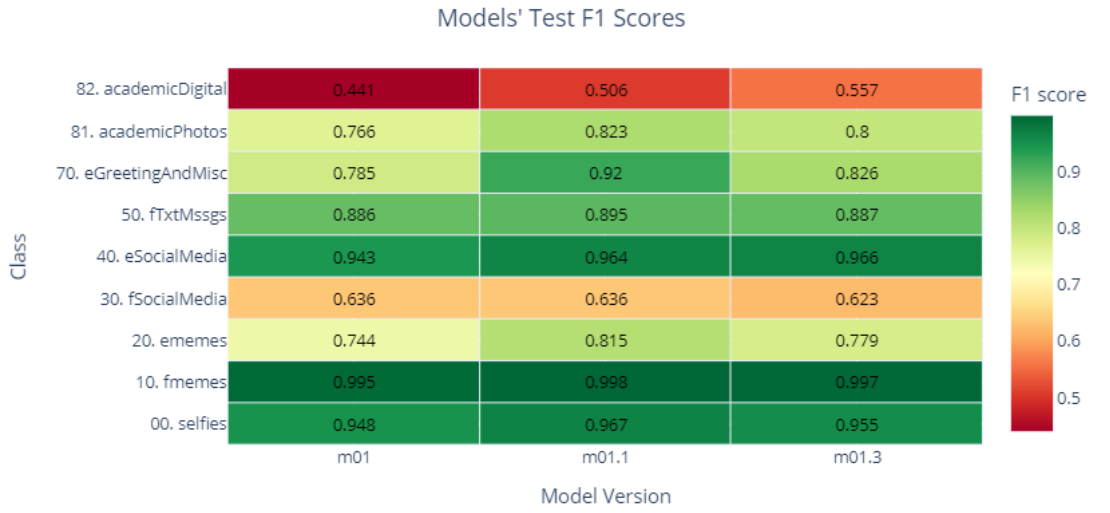


Figure 135. Comparison 1: heatmap of raw test f1 scores.

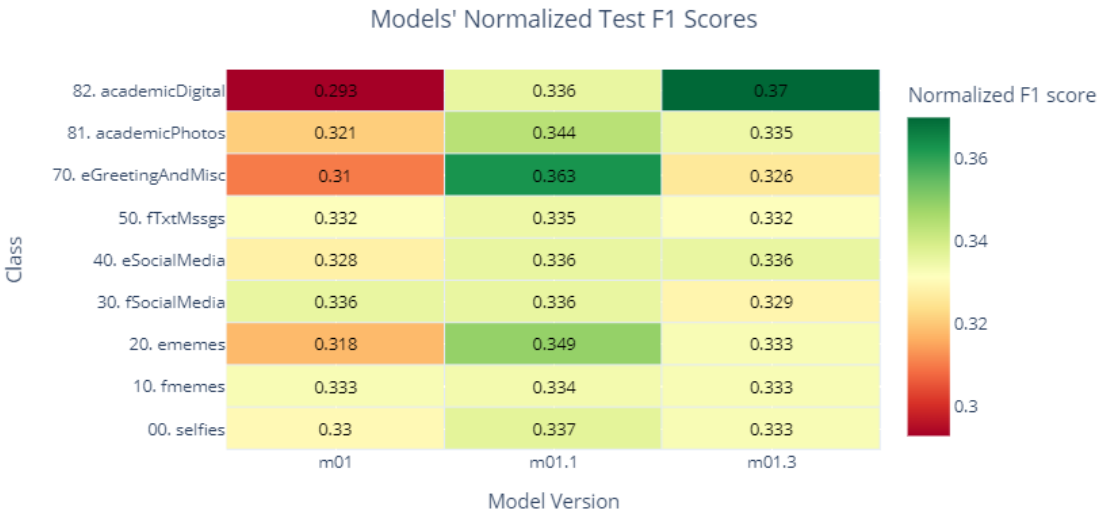


Figure 136. Comparison 1: heatmap of normalized test f1 scores.

	class	best model	f1 score
1	00. selfies	1.1	0.96731
2	10. fmemes	1.1	0.99788
3	20. ememes	1.1	0.81519
4	30. fSocialMedia	1.6	0.68235
5	40. eSocialMedia	1.1	0.96393
6	50. fTxtMssgs	1.6	0.90072
7	70. eGreetingAndMisc	1.1	0.9199
8	81. academicPhotos	1.1	0.82326
9	82. academicDigital	1.1	0.50579

Table 4. Comparison 2: best model in each class.

Figure 137. Comparison 2: average of test f1 scores.

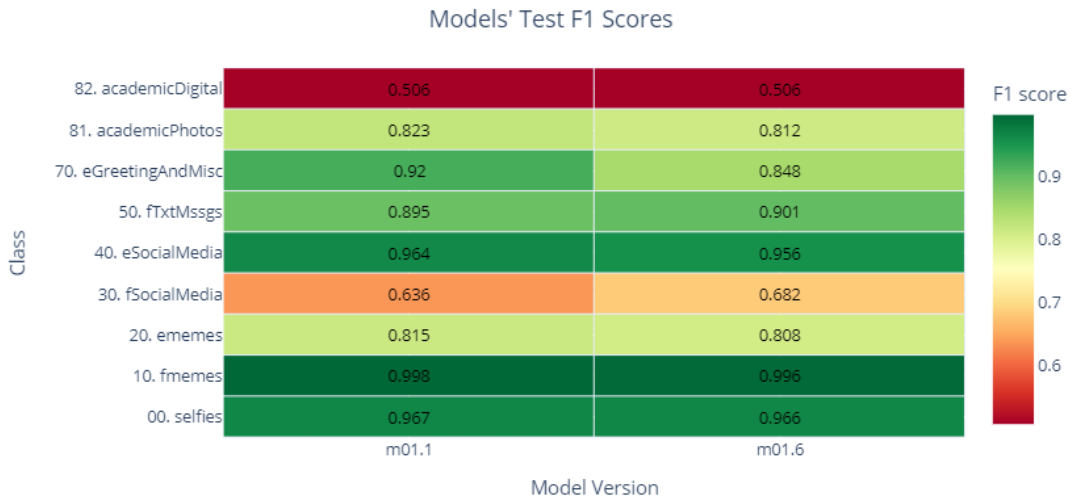


Figure 138. Comparison 2: heatmap of raw test f1 scores.

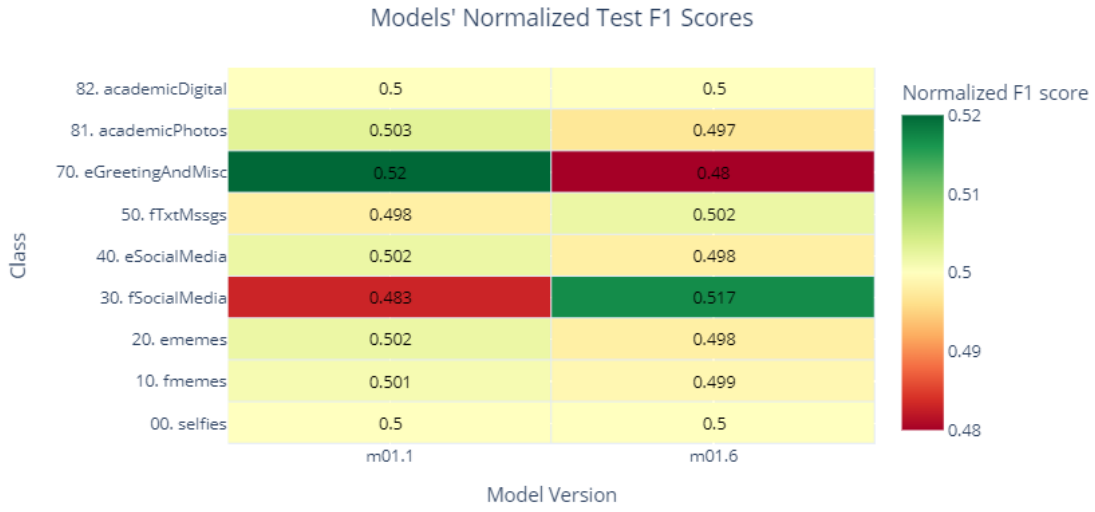


Figure 139. Comparison 2: heatmap of normalized test f1 scores.

	class	best model	f1 score
1	00. selfies	2	0.97317
2	10. fmemes	2.1	0.99586
3	20. ememes	2	0.79558
4	30. fSocialMedia	2.1	0.64044
5	40. eSocialMedia	2	0.97456
6	50. fTxtMssgs	2.1	0.89084
7	70. eGreetingAndMisc	2	0.8396
8	81. academicPhotos	2.1	0.80208
9	82. academicDigital	2.1	0.52159

Table 5. Comparison 3: best model in each class.

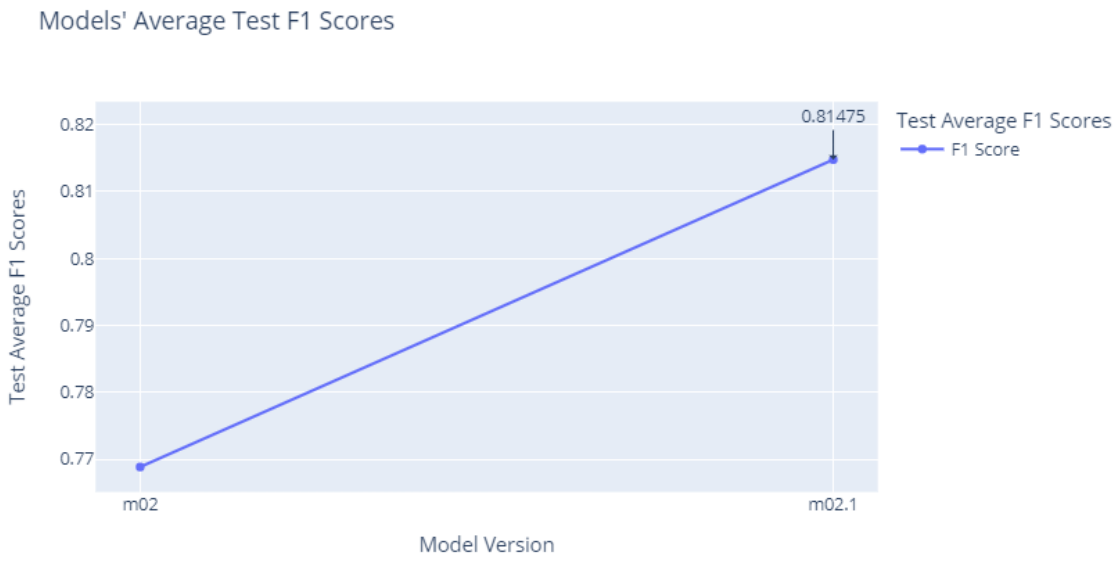


Figure 140. Comparison 3: average of test f1 scores.

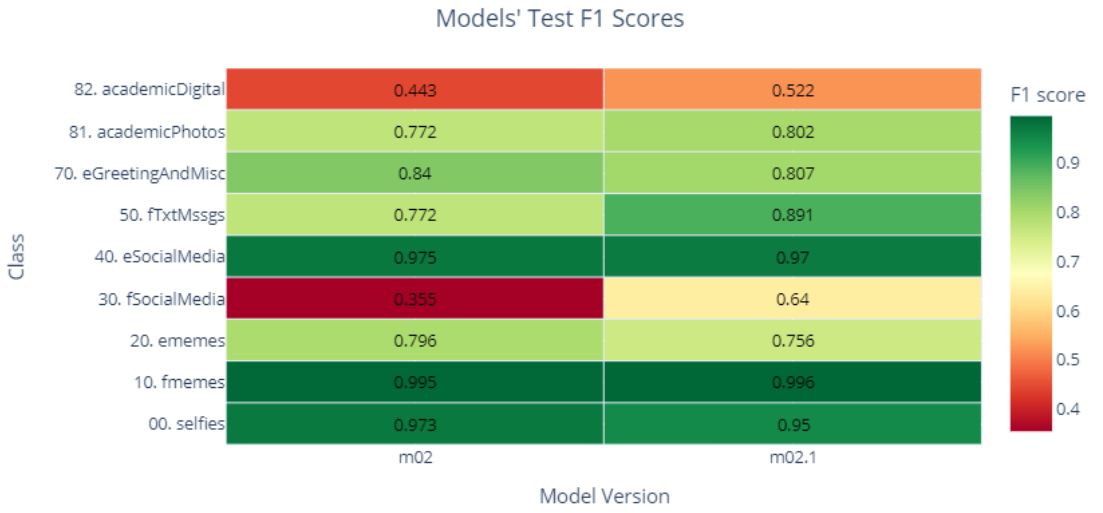


Figure 141. Comparison 3: heatmap of raw test f1 scores.

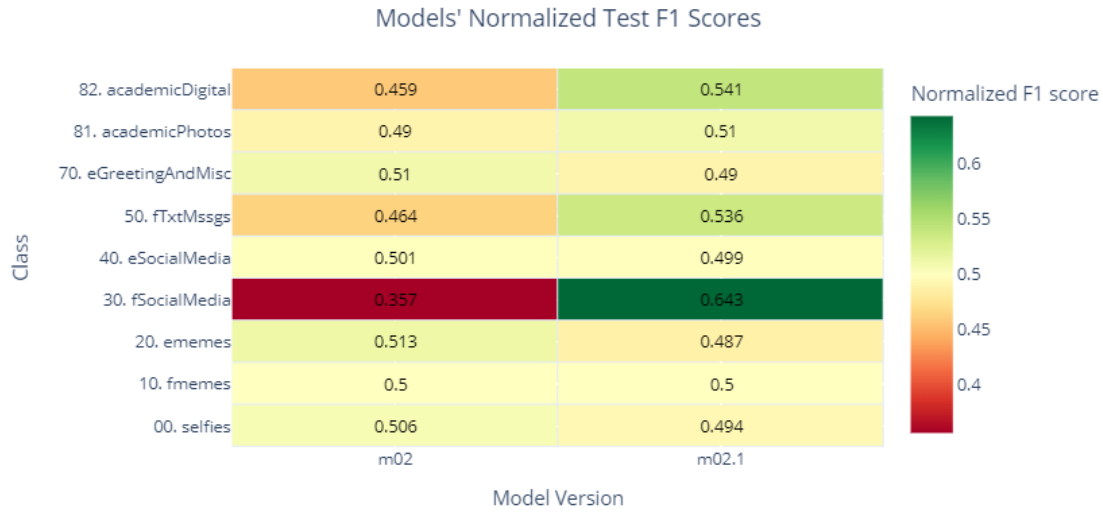


Figure 142. Comparison 3: heatmap of normalized test f1 scores.

	class	best model	f1 score
1	00. selfies	2	0.97317
2	10. fmemes	2.2	0.99851
3	20. ememes	2	0.79558
4	30. fSocialMedia	2.2	0.63687
5	40. eSocialMedia	2	0.97456
6	50. fTxtMssgs	2.2	0.92546
7	70. eGreetingAndMisc	2.2	0.92684
8	81. academicPhotos	2.2	0.86253
9	82. academicDigital	2.2	0.52716

Table 6. Comparison 4: best model in each class.

Figure 143. Comparison 4: average of test f1 scores.

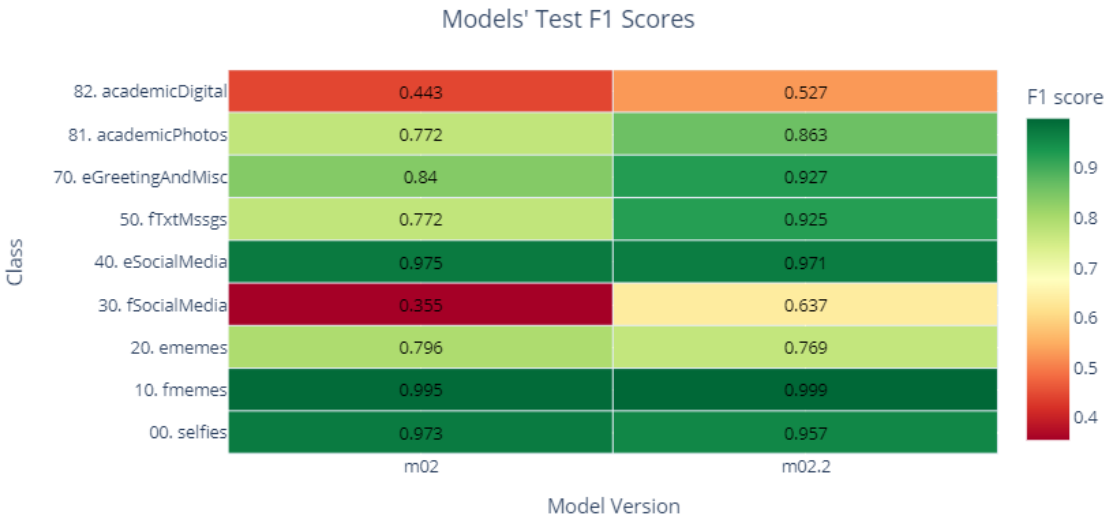


Figure 144. Comparison 4: heatmap of raw test f1 scores.

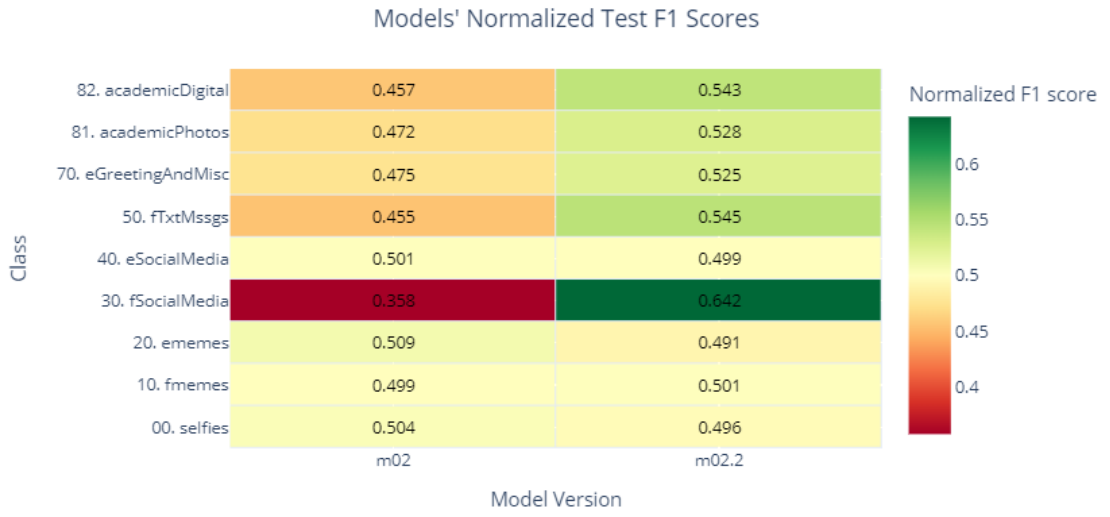


Figure 145. Comparison 4: heatmap of normalized test f1 scores.

	class	best model	f1 score
1	00. selfies	2.5	0.9594
2	10. fmemes	2.1	0.99586
3	20. ememes	2.5	0.78057
4	30. fSocialMedia	2.5	0.67519
5	40. eSocialMedia	2.5	0.97014
6	50. fTxtMssgs	2.5	0.90128
7	70. eGreetingAndMisc	2.5	0.83292
8	81. academicPhotos	2.5	0.85854
9	82. academicDigital	2.5	0.53016

Table 7. Comparison 5: best model in each class.

Figure 146. Comparison 5: average of test f1 scores.

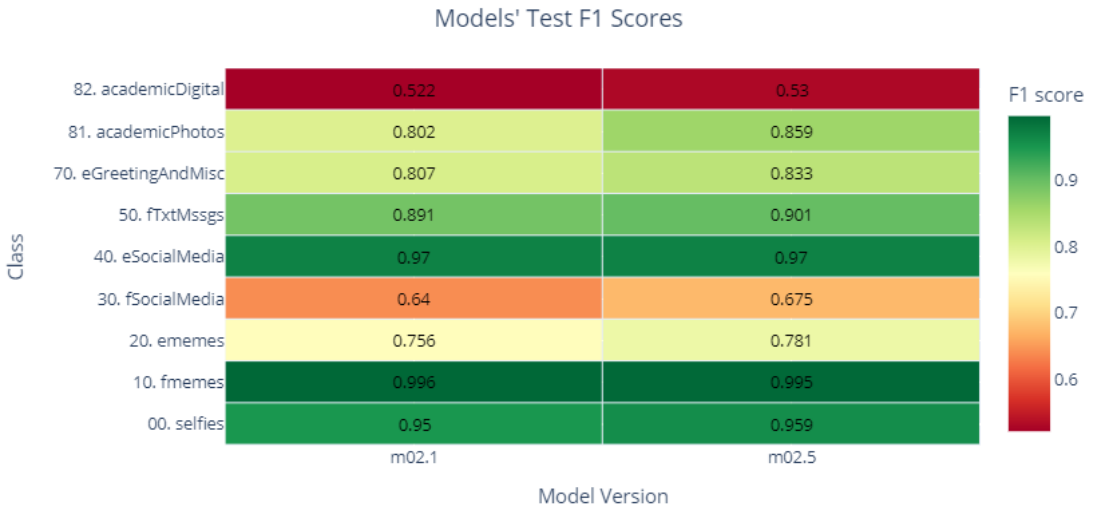


Figure 147. Comparison 5: heatmap of raw test f1 scores.

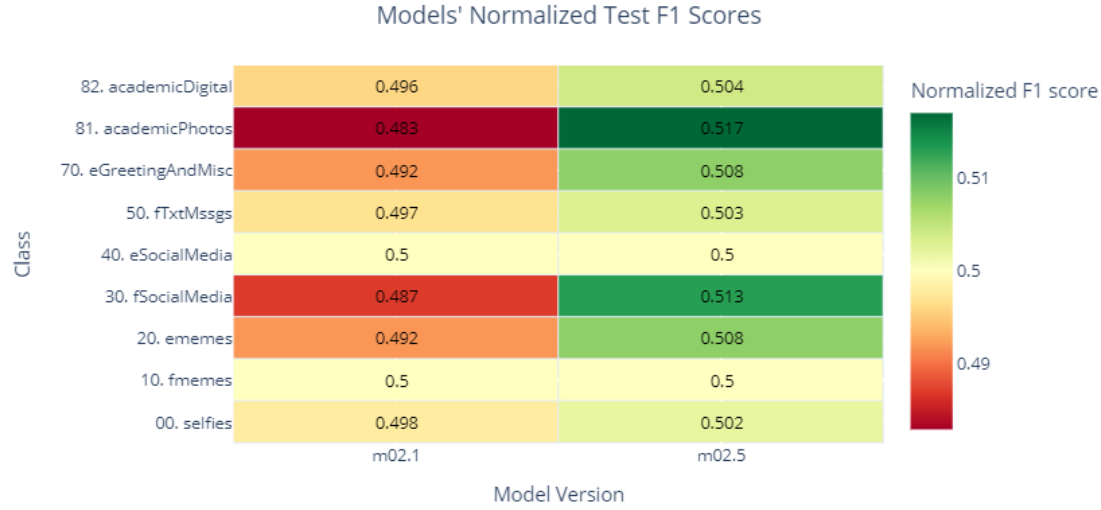


Figure 148. Comparison 5: heatmap of normalized test f1 scores.

	class	best model	f1 score
1	00. selfies	4.2	0.97291
2	10. fmemes	4.1	0.99873
3	20. ememes	4.2	0.83986
4	30. fSocialMedia	4.2	0.66576
5	40. eSocialMedia	4.1	0.97398
6	50. fTxtMssgs	4.2	0.89883
7	70. eGreetingAndMisc	4.2	0.93478
8	81. academicPhotos	4	0.84315
9	82. academicDigital	4.2	0.55489

Table 8. Comparison 6: best model in each class.

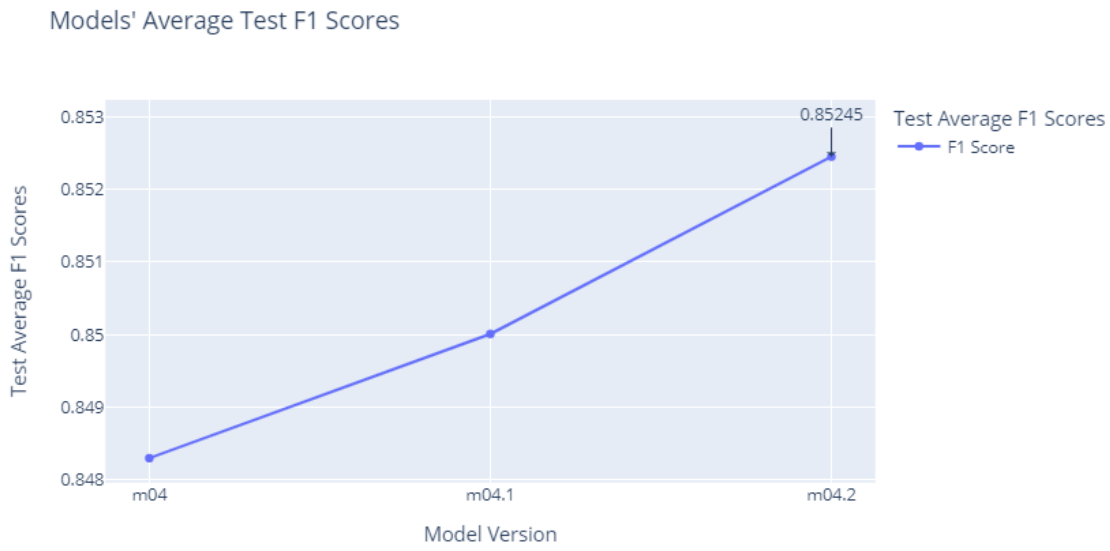


Figure 149. Comparison 6: average of test f1 scores.

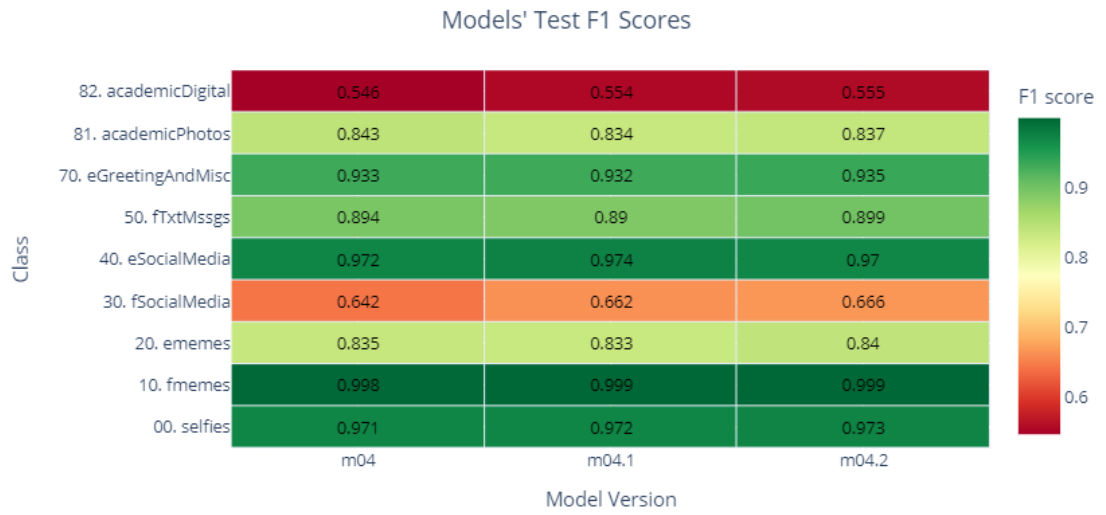


Figure 150. Comparison 6: heatmap of raw test f1 scores.

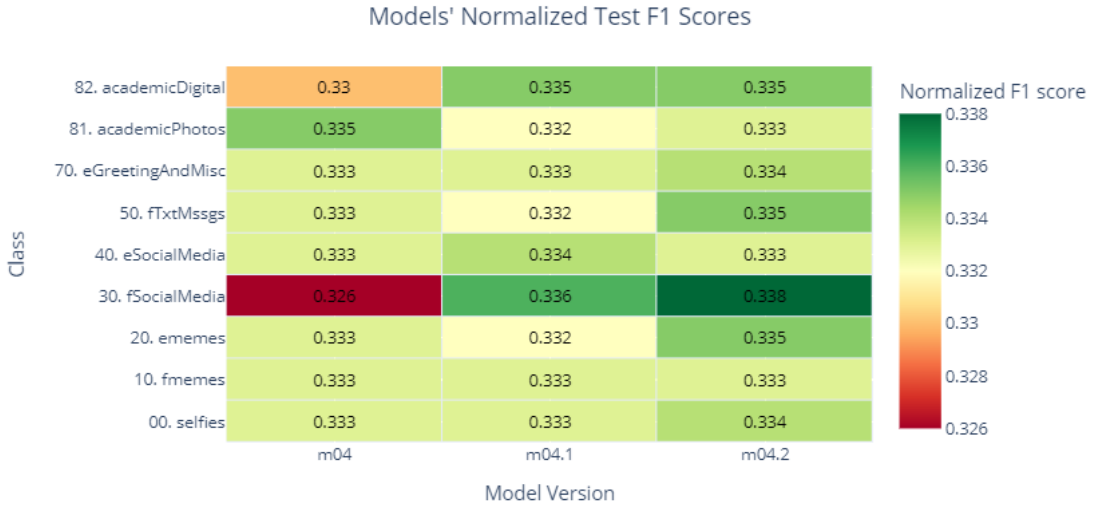


Figure 151. Comparison 6: heatmap of normalized test f1 scores.

5.3 Comparing All Models

We'll include all the previously mentioned models in the plots and table found in this section. This comparison between all models will be called comparison 7.

	class	best model	f1 score
1	00. selfies	3	0.9762
2	10. fmemes	4.1	0.99873
3	20. ememes	3	0.85745
4	30. fSocialMedia	1.6	0.68235
5	40. eSocialMedia	3	0.9848
6	50. fTxtMssgs	2.2	0.92546
7	70. eGreetingAndMisc	4.2	0.93478
8	81. academicPhotos	3	0.9077
9	82. academicDigital	3	0.64686

Table 9. Comparison 7: best model in each class.

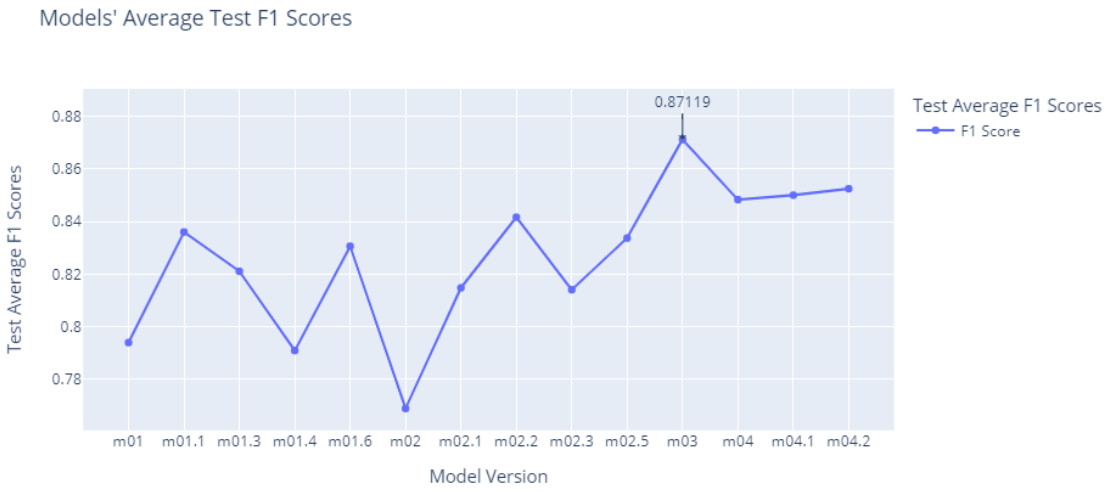


Figure 152. Comparison 7: average of test f1 scores.

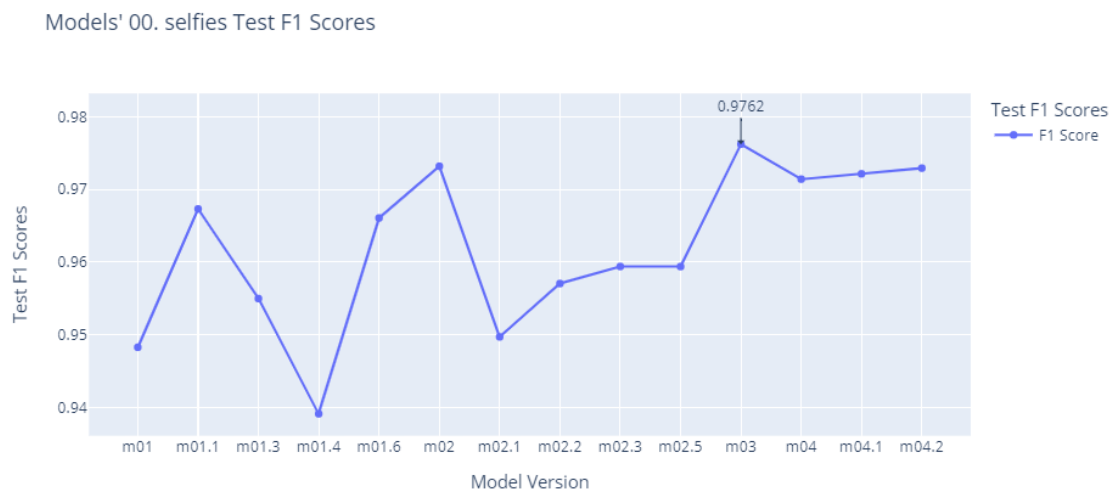


Figure 153. Comparison 7: “selfies” class test f1 scores.

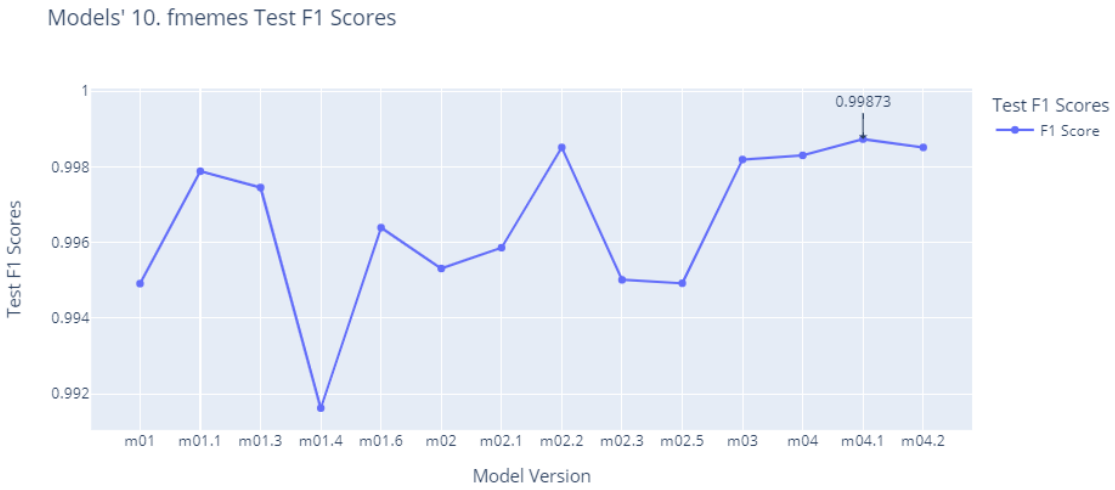


Figure 154. Comparison 7: “fMemes” class test f1 scores.

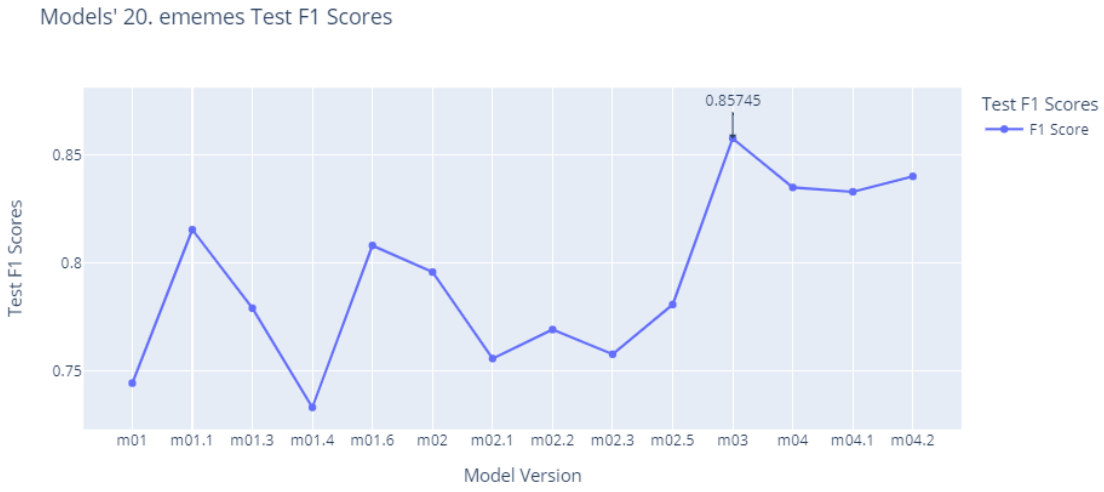


Figure 155. Comparison 7: “eMemes” class test f1 scores.

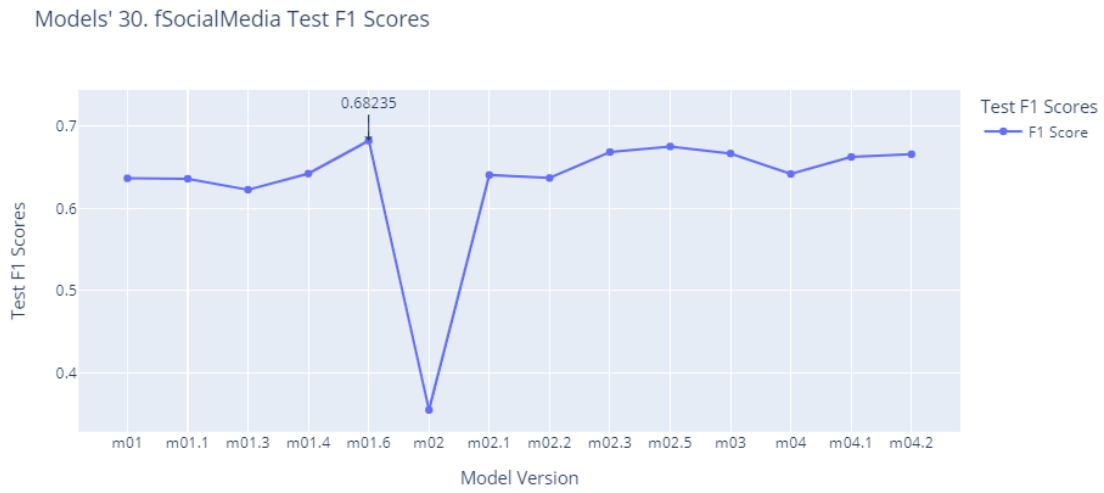


Figure 156. Comparison 7: “fSocialMedia” class test f1 scores.

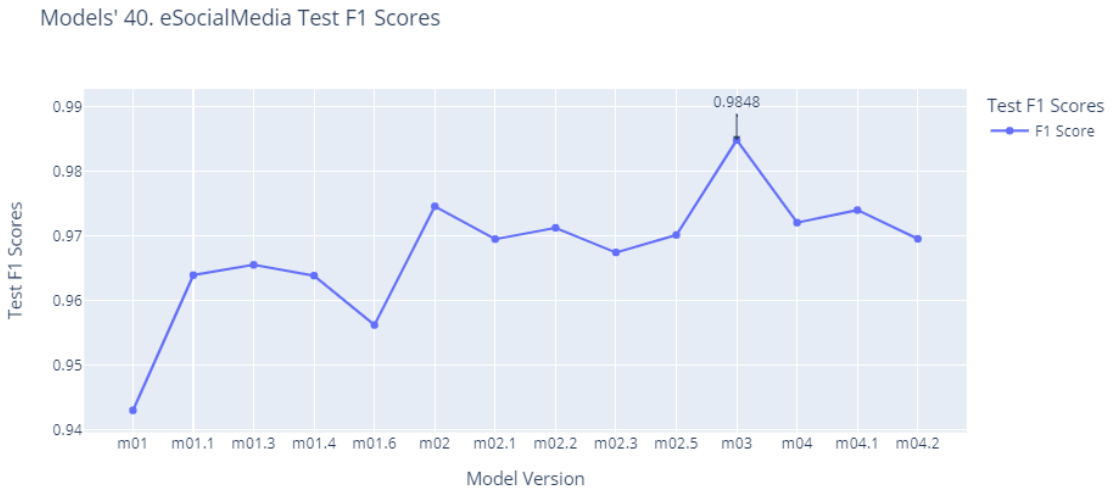


Figure 157. Comparison 7: “eSocialMedia” class test f1 scores.

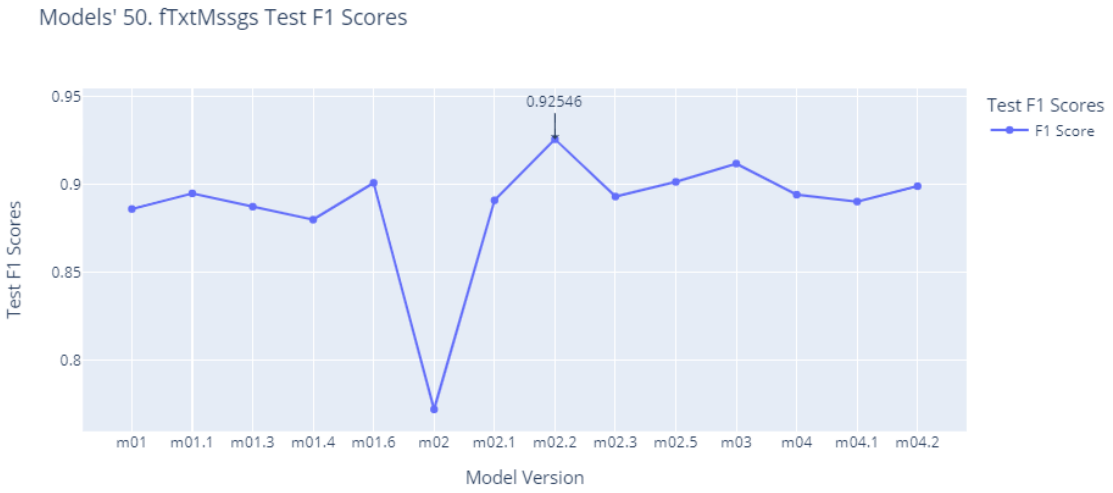


Figure 158. Comparison 7: “fTxtMssgs” class test f1 scores.

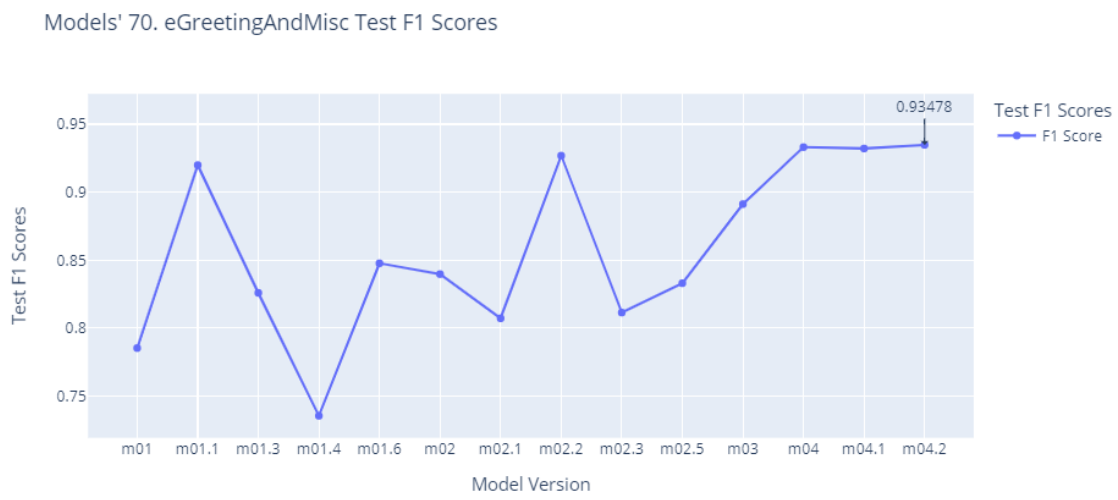


Figure 159. Comparison 7: "eGreetingAndMisc" class test f1 scores.

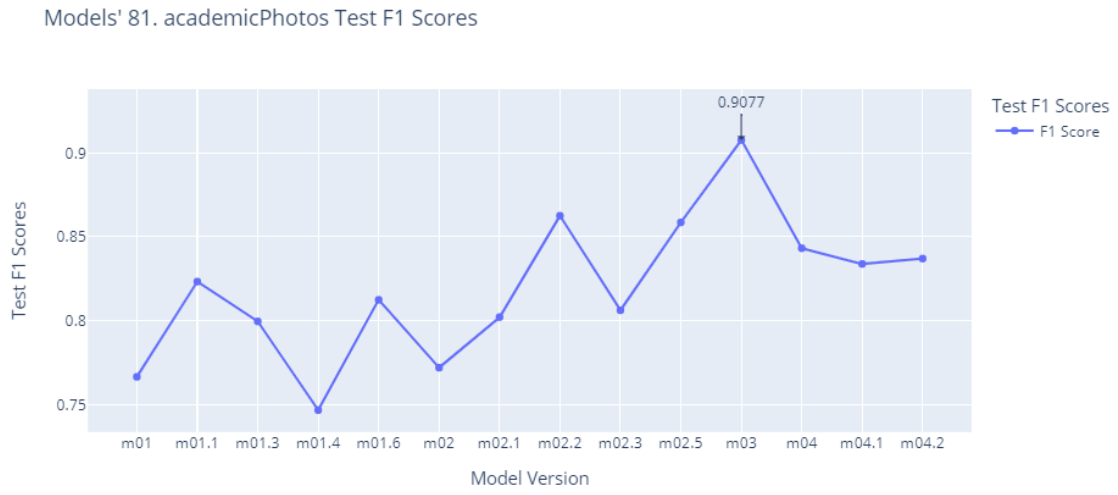


Figure 160. Comparison 7: "academicPhotos" class test f1 scores.

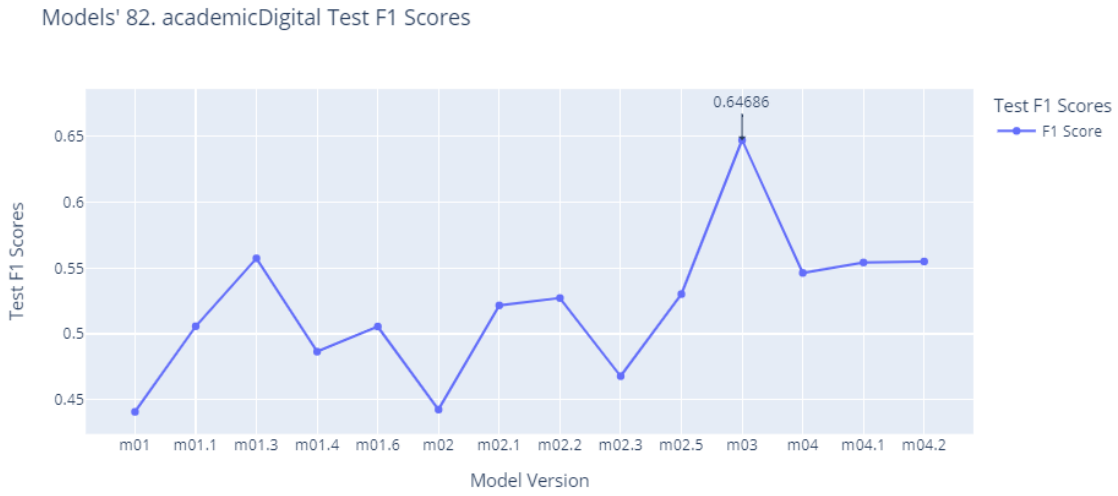


Figure 161. Comparison 7: "academicDigital" class test f1 scores.

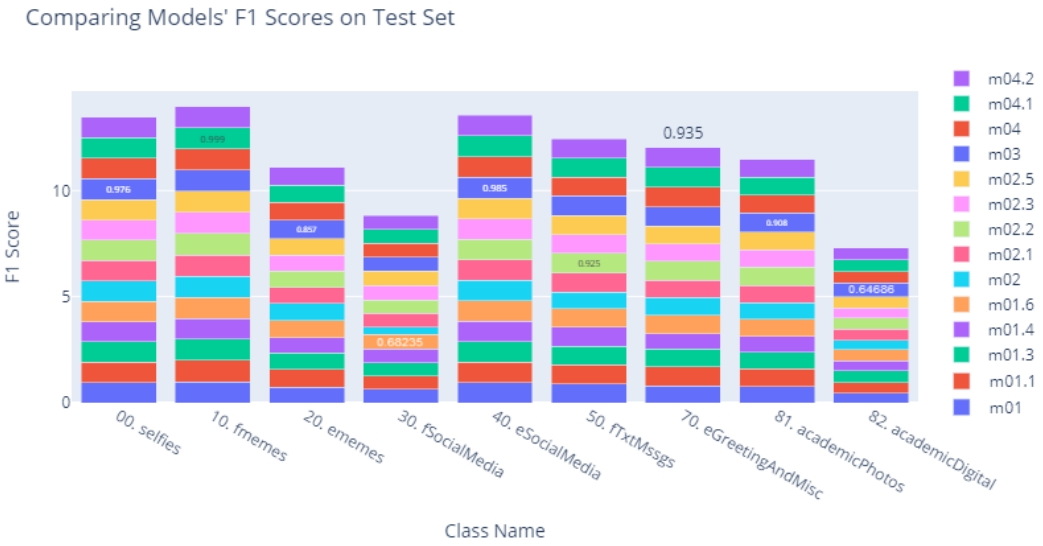


Figure 162. Comparison 7: stacked bar plot of all test f1 scores, where the highest score is written per class.

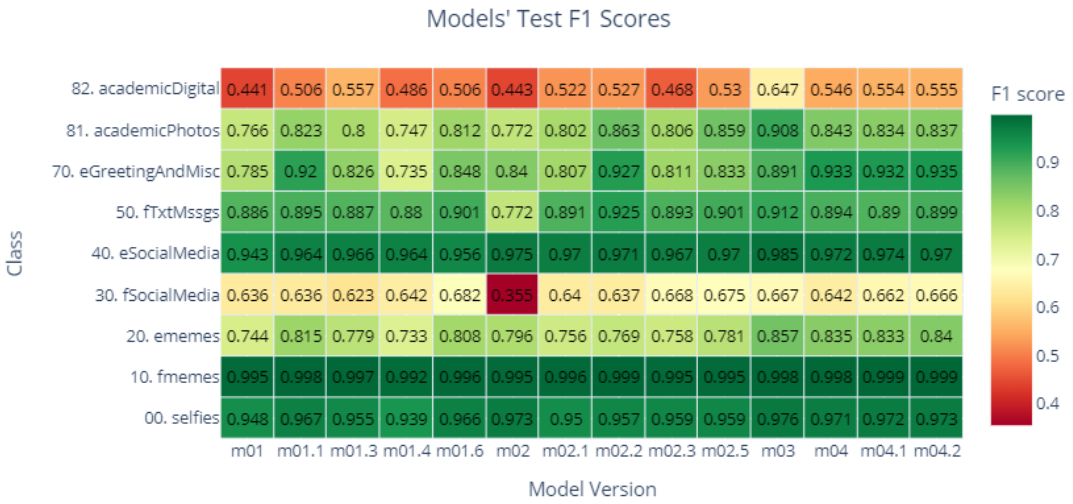


Figure 163. Comparison 7: heatmap of all test f1 scores.

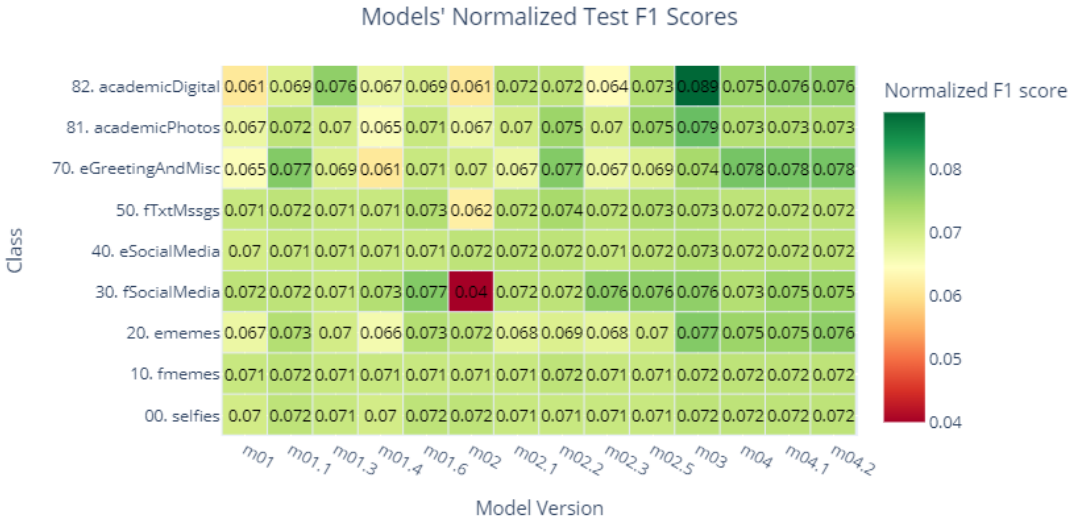


Figure 164. Comparison 7: heatmap of all normalized test f1 scores.

5.4 Analysis

First, regarding section 5.1, we can see from most of the models' loss graphs that the validation loss keeps rising while the training loss stagnates at a low value, which means that these models are overfitting the data. However, this phenomenon decreases in the HCNN's (m03) loss graphs, as can be seen in figures Figure 106, Figure 107, Figure 114, and it also decreases in XCNN's (m04) loss graphs, as can been seen in Figure 121, but as previously discussed in section 4.2.4, the loss function used to evaluate the XGBoost models does not penalize the model for highly predicting incorrect values. So, this should be considered when comparing the loss functions of "m04.x" with other model types. Now, regarding the validation f1 score plots, there are two abnormal phenomena that occur:

Some validation figures like epoch 28 in Figure 70 (m01), epoch 19 in Figure 74 (m01.3), and epoch 26 in Figure 117 (m03) experience a sudden "dip" of performance in most of the classes.

The training and validation figures Figure 82 and Figure 83 respectively (m01.6) seem to experience consistent spikes in performance.

The reason for the former abnormality might be because Adam optimizer's learning decay rate is too quick or abrupt to the point of hindering the model's ability to converge, or because of insufficient use of regularization techniques such as batch normalization which may have led to model instability while training due to feeding unnormalized data to the model. That latter hypothesis is most likely the case, because if we, for example, compare the validation figures Figure 91 and Figure 99 for "m02.1" and "m02.3" with the figures Figure 87 and Figure 95 for "m02" and "m02.2", we'll clearly see that the former two models, which apply normalization to the metadata features input, do not have these spikes in performance and the training is thus more stable, unlike the two latter models, which do not apply normalization.

As for the latter abnormality, clearly using dataset "v01.1" instead of "v01" is the issue, as "m01.6" is the only model which used the former dataset, and it is the only model that experienced very large spikes in performance. Now, recalling Figure 45, we see that each of the 4 minority classes have been further trimmed down as an attempt to remove images which we thought would confuse the model (like very wide images which will not be clear when resizing to 306x306). Even though the test set was trimmed down from 20312 images to 19865, still the trimming down of images in minority classes which already contained relatively few images could have led the model to overshoot its parameters' update leading the model to propagate around the objective function's minima without properly converging. In other words, the decrease of minority classes in addition to a high initial learning rate could have been the reason for the spiked performances we see in these aforementioned figures.

Now, regarding Figure 133, we can see the following results and interpretations for each comparison that was made in section 5.2:

- Comparison 1 results: From Table 3, we can see that "m01" (8 batch size) was not chosen once as best model, and that "m01.1" (32 batch size) dominated most of the classes' best scores than "m01.3" (16 batch size), indicating the importance of increasing the batch size in order for the gradient step done per batch to be a decent approximation for the gradient step done on the entire dataset[1].
- Comparison 2 results: Even though Table 4 shows that "m01.1" (dataset v01) has the lead on best f1 scores over "m01.6" (v01.1), Figure 139 illustrates that the difference between the models' score is relatively minute, thus cleaning images from minority classes might not be an effective method as obtaining more images for these minority classes, augmenting only the minority classes to oversample them, etc.
- Comparison 3 results: Other than the increased training stability discussed earlier in this section, "m02.1" (scaled colour features) also has a better overall f1 score than "m02" (unaltered colour features) as shown in Figure 140.
- Comparison 4 results: As we can see from Figure 144, adding face-related metadata (m02.2) gives a noticeable increase in most of the classes' f1 scores.
- Comparison 5 results: Even though using a fully normalized metadata of colours, face, and text-related information (m02.5) yields better results than using only normalized colour-related metadata (m02.1) as shown in Figure 146, we see that the average f1 score of "m02.5" (0.83368) is slightly less than that of "m02.2" (0.84163), which only used colour and face-related metadata, which could indicate that correlation between text features and the image's class is less than that between face-related features and the image's class. This is uncertain though, as the features in the former model were normalized, while those of the latter model were not normalized.
- Comparison 6 results: It can be shown from Figure 150 that the classes' f1 scores are very similar in "m04", "m04.1" (sample weights), and "m04.2" (sample weights and Bayesian optimization), so the slight respective average f1 score increases shown in Figure 149 is not worth the extra training time offered by "m04.2" to find more suitable hyperparameters, and that using sample weights is not enough to overcome the imbalanced dataset problem, and that in all 3 cases,

[1] However, given that difficulties related to computational resources occurred during the time frame of training "m02.x" models, we'll see that batch size 16 was still used in these model variants.

and as mentioned at the start of section 5.1, the XGBoost model variations are all hindered by overfitting problem.

Finally, regarding section 5.3, we can see from Table 9 that "m03" has the best f1 score in 5 classes: "selfies", "eMemes", "eSocialMedia", "academicPhotos", and "academicDigital". However, by looking at Figure 163, we see that "m03" was very close to the best score achieved by other models in each of the other classes. This is illustrated in Table 10, where the only noticeable difference in score is in "eGreetingAndMisc" class. Moreover, it is worth noting that "m03" achieved very high f1 scores for training, validation, test f1 scores and confusion matrix at its first output layer (output layer 0) as previously illustrated in figures Figure 109, Figure 110, Figure 111, and Figure 112 where the task of that layer was to predict if an image contained relatively many or few colours. This implies that trying a deeper hierarchy, like the one presented in Figure 35, might yield better results.

5.4.1 Results Summary in Relation to HCNN

Moreover, seeing that the only score which was noticeably higher than the HCNN's score was given by XGBoost model "m04.2", as seen in Table 10, one could assume that using an XGBoost model on top of HCNN might also yield better results.

Class	Best Model	Best Model's F1 Score	HCNN's (m03) F1 Score
fMemes	m04.1	0.99873	0.99819
fSocialMedia	m01.6	0.68235	0.66667
fTxtMssgs	m02.2	0.92546	0.91166
eGreetingAndMisc	m04.2	0.93478	0.89117

Table 10. Comparison 8: f1 scores of best models in other classes vs f1 score of "m03".

6 Conclusions

In this section, we present the conclusions drawn from our project along with recommendations for future endeavours. Firstly, we discuss the project's conclusion and recommendation, restating the work done in this project and providing insights for further improvements and potential areas of exploration. Secondly, we emphasize the project contributions, highlighting the novel aspects, methodologies, or techniques developed during the course of our work. Lastly, we acknowledge the limitations encountered during the project and propose avenues for future work. We identify areas that require further investigation, or improvements, to enhance the project's scope and effectiveness.

6.1 Project's Conclusion and Recommendations

we shed light on the daily-life problem of manually classifying images to differentiate between important images from unimportant ones, created a dataset which aims to resemble images found on an Egyptian storage device, elaborated on how it was obtained and curated, explained deep learning and machine learning architectures which could be trained on such dataset in order to classify the images, mentioned how they were implemented under the hood, then analysed their performance metrics' scores in order to deduce possible best practices and considerations for designing model architecture and for choosing their hyperparameters, all in the hopes of eventually developing a mobile application to group, and therefore effectively clean irrelevant images. Hopefully you now realize the importance of keeping your phone's media clean!

Due to the many hardships that we faced during dataset preparation and model development/evaluation phases; we have prepared a few suggestions which should serve as guidance in projects with a task similar to this one:

- Training on datasets created manually, or datasets that have not been properly pre-processed by their authors, should follow a "data-centred" approach first, not a "model-centred" approach, as mentioned later in section 6.3.
- When creating a deep learning pipeline, trying simpler options is preferred. For example, under sampling an imbalanced dataset before moving on to more complex solutions like augmenting or applying sample weights. In short, the KISS rule (keep it simple, stupid) is a relevant concept which should always be considered.
- Larger batch sizes are preferable, as they decrease the objective function's fluctuation around minima points.

- Normalizing the input data is preferable, as that helps the model avoid overfitting, as previously discussed at the start of section 5.4.
- Profiling models after training is important to know where the bottleneck lies. Figure 165 Is a demonstration of this.
- Finally, we personally prefer using "mamba" environment manager over "conda" purely because of its speed of installing and managing dependencies. Moreover, it is preferable not to install any pip packages unless these packages are not provided by mamba, as mamba cannot track pip dependencies, so dependency issues between mamba and pip installed packages may arise.

Action	Mean duration (s)	Num calls	Total time (s)	Percentage %
Total	-	436836	1.127e+04	100 %
run_training_epoch	3605.7	3	1.0817e+04	95.984
[Callback]TQDMProgressBar.on_train_batch_end	0.40905	8886	3634.9	32.254
[TrainingEpochLoop].train_dataloader_next	0.407	8886	3616.6	32.092
run_training_batch	0.2537	8886	2254.4	20.004
[LightningModule]HierarchalModelPL.optimizer_step	0.25334	8886	2251.1	19.975
[Strategy]SingleDeviceStrategy.training_step	0.24873	8886	2210.2	19.612
[EvaluationEpochLoop].val_dataloader_idx_0_next	0.40664	1902	773.43	6.863
[Strategy]SingleDeviceStrategy.validation_step	0.27732	1902	527.46	4.6804
run_test_evaluation	421.12	1	421.12	3.7368

Figure 165. PyTorch Lightning's simple profile option reveals the biggest bottleneck is in "training epoch" logic and in loading the data.

6.2 Project Contributions

Due to the novelty of the problem presented in this report, the most important contributions made are regarding the dataset:

- To the best of our knowledge, this is the first relatively large dataset to include Egyptian memes and social media posts, in addition to images usually sent between family members and relatives ("eGreetingAndMisc" class), and images scraped from Reddit related to social media or text messages. In addition to this, "imgsPropsv1.csv" file which holds metadata, previously mentioned in Figure 31, for the images in the dataset.
- The possibility of reusing scraping functions to scrape from Facebook, Reddit, and google images (given small tweaks to the functions in case of HTML change in the aforementioned platforms)
- To the best of our knowledge, we also established novel pipeline to remove unwanted images in a class, which was introduced in "separating_screenshots_from_photos.ipynb" then refined later to involve less manual work in "dataset_preprocessing_part_3.ipynb". The basic premise of this method was explained near the end of section 3.1.3 and the refined version was mentioned in section 4.1.2.

Moreover, contributions were made regarding model development and evaluation phases:

- To the best of our knowledge, this is the first utilization of PyTorch Lightning to implement an HCNN model (using "simple-hierarchy" library as PyTorch backbone).
- Extensive functions were created to log very specific metrics of a model while it's training. An example of such metrics was illustrated in Figure 64.
- Visualization functions can be used to display plotly figures that, to our knowledge, were not displayed in a unique way like we did in some of the plots. For example, we did not encounter any code online which displayed the sum of rows and columns in the confusion matrix, like in Figure 119.

Finally, contributions were made in the form of python scripts which helps in managing a virtual environment which is initially managed by conda/mamba and pip:

- "pip_install_add_to_environment_yml.py": automatically adds a package installed via pip to the pip's section of "environment.yml" file created by conda/mamba.
- "pip_uninstall_remove_from_environment_yml.py": it is similar to the first file, but for uninstallation.
- "conda_update_from_pip_list.py": It overwrite the pip section in "environment.yml" file with the output from "pip freeze" command. However, this is only suggested if you install all the libraries using pip, not conda/mamba.
- "package_dep_info.py": the most important script; it uses "pipdeptree" library [55] to generate a json-like string to the console output containing all the packages installed by mamba and/or pip (example shown in Figure 166), and recursively list the dependencies of each of these packages. The python file takes this output and reorder it such that when you pass a package name to this python script, it tells you a list of direct (or recursive) dependencies, and a list of direct (or recursive) required-by packages, along with the package and/or environment manger used to install each of these packages. Example of this is illustrated in Figure 166 and Figure 167.

```
>pipdeptree -p pandas

pandas==1.5.3
  - numpy [required: >=1.20.3, installed: 1.24.3]
  - python-dateutil [required: >=2.8.1, installed: 2.8.2]
    - six [required: >=1.5, installed: 1.16.0]
  - pytz [required: >=2020.1, installed: 2022.7]
```

Figure 166. Example of using "pipdeptree" library to get information on "pandas" library.

```
>python package_dep_info.py -h
usage: package_dep_info.py [-h] [-p PACKAGES [PACKAGES ...]] [-ld] [-lrb] [-dd] [-drb] [-spi] [-u]

optional arguments:
  -h, --help            show this help message and exit
  -p PACKAGES [PACKAGES ...], --packages PACKAGES [PACKAGES ...]
                        passing x package(s) will output x requires/required-by descriptions
  -ld, --leaves-dependencies
                        display all leaf packages that don't have dependencies
  -lrb, --leaves-required-by
                        display all leaf packages which are not required by any packages
  -dd, --deep-dependencies
                        recursively output all packages which package(s) requires
  -drb, --deep-required-by
                        recursively output all packages which depend on package(s)
  -spi, --show-package-installer
                        (slower) in the output, displays the packages installed with pip vs mamba (conda)
  -u, --uninstall-packages
                        uninstalls all -p packages from mamba/pip (if -spi, recommended) or pip only (else) along with their unused
                        dependencies
```

Figure 167. Example of using "package_dep_info" script to get information on the script's arguments.

```
>python package_dep_info.py -spi -p pandas simple-hierarchy
package name (installed via mamba/conda):  pandas          package name (installed via pip):  simple-hierarchy

*****required by:*****                                     *****required by:*****
pip installed:                                             pip installed:
['fastdup']                                                []
mamba/conda installed:                                     mamba/conda installed:
['cufflinks']                                              []

*****dependencies:*****                                    *****dependencies:*****
pip installed:                                             pip installed:
[]                                                         []
mamba/conda installed:                                     mamba/conda installed:
['numpy', 'python-dateutil', 'pytz']                       ['numpy', 'torch']

*****dependencies not required by other packages:*****     *****dependencies not required by other packages:*****
None                                                       None
```

Figure 168. Example of using "package_dep_info" script to get information on "pandas" and "simple-hierarchy" libraries, as dependencies, and required-by packages.

6.3 Limitations and Future Work

This section recalls some of the difficulties and shortcomings mentioned throughout this project and brings up other unmentioned hardships and then suggests where future work should be headed with respect to these difficulties.

The most obvious difficulty faced was curating a custom dataset to train the models on; Manual inspection and the few tools and methods mentioned in sections 3.1 and 4.1 were not enough to fully prepare this dataset for model training. Moreover, it is obvious that we chose a "model-centred" approach instead of a "data-centred" approach when attempting to improve the models' performance, even though the dataset was mostly manually curated, and thus needed constant cleaning and preprocessing according to the models' results. Therefore, one of the main directions of any future research on this topic is to apply a "data-centred" approach to further prepare the custom dataset according to decisions made by exploratory data analysis (EDA) and models' results. An example of the former is to apply a clustering algorithm on the entire dataset to see the bigger picture of classes' inter and intra similarity, while and example of the latter is

to fix a specific model and keep changing the dataset itself while monitoring the results of that specific model [56].

We also had difficulties in our "model-centred" approach, where we had limited time and semi-powerful free computational resources to train our models; most models took around 12 hours to finish training, while the HCNN model alone took around 33 hours to finish training on a Nvidia RTX 2060 Max Q graphic card. Consequently, we didn't get to try a lot of hyperparameter and model combinations, for example, as mentioned at the end of section 5.4. Therefore, further model architecture optimization is another future research possibility.

Regarding section 4, the code is written in multiple notebook and python files which could mostly be refactored into succinct functions called from a single or a few sources of truth (i.e., python files), and is not the most optimal. For example, all the data loaders used to get the images in batches did not utilize the CPU parallelly as this required migrating most of the code from notebook files to python files, which was difficult to do. Moreover, future work could be dedicated to replacing outdated third-party libraries like "RetinaFace" face detection model and "PaddleOCR" tool into more updated counterparts, as using these outdated libraries caused a problem in managing the project's virtual environment; since PyTorch Lightning was introduced later in the project for the HCNN model, it, along with TensorFlow, "RetinaFace", and "PaddleOCR" required dependency libraries which had conflicting versions. Therefore, the project's current virtual environment only supports PyTorch Lightning and TensorFlow, so that the two latter libraries along with other libraries used for extracting the images' metadata (as mentioned in sections 3.1.4.1 and 4.1.3) are no longer present in the current virtual environment. Managing this dependency issue, and/or creating another virtual environment dedicated for dataset preprocessing is one of the possible directions for future implementation.

Regarding model evaluation methods, another suggestion is to better utilize the HTML files which display the models' test results, and are previously mentioned in section 4.2, by grouping the images into a grid of small images for each (predicted label, actual label) pair so that we have something similar to what was visualized in Figure 47 instead of a long table like what was shown in Figure 54.

Finally, the most important area to focus on in the future is creating a gallery mobile application which will be able to group images based on one of the 9 classes presented in section 3.1.1. In addition, finer classifications can be made for each class. For example, the model could perform further classification to recognize the person in the image by face after "selfies" class have been chosen as the predicted class for the image.

References

[1] E. Ferrara, M. JafariAsbagh, O. Varol, V. Qazvinian, F. Menczer, and A. Flammini, "Clustering memes in social media," in *2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2013)*, Aug. 2013, pp. 548–555. doi: 10.1145/2492517.2492530.

[2] K.-C. Chang, "Mapping Visual Themes among Authentic and Coordinated Memes." arXiv, Jun. 05, 2022. doi: 10.48550/arXiv.2206.02306.

[3] M. Caron, P. Bojanowski, A. Joulin, and M. Douze, "Deep Clustering for Unsupervised Learning of Visual Features." arXiv, Mar. 18, 2019. doi: 10.48550/arXiv.1807.05520.

[4] S. Zannettou *et al.*, "On the Origins of Memes by Means of Fringe Web Communities." arXiv, Sep. 22, 2018. Accessed: Nov. 22, 2022. [Online]. Available: http://arxiv.org/abs/1805.12512

[5] V. Monga and B. L. Evans, "Perceptual Image Hashing Via Feature Points: Performance Evaluation and Tradeoffs," *IEEE Transactions on Image Processing*, vol. 15, no. 11, pp. 3452–3465, Nov. 2006, doi: 10.1109/TIP.2006.881948.

[6] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, in KDD'96. Portland, Oregon: AAAI Press, Aug. 1996, pp. 226–231.

[7] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN," *ACM Trans. Database Syst.*, vol. 42, no. 3, pp. 1–21, Aug. 2017, doi: 10.1145/3068335.

[8] C. Onielfa, C. Casacuberta, and S. Escalera, "Influence in Social Networks Through Visual Analysis of Image Memes," 2022. doi: 10.3233/FAIA220317.

[9] Y. Baek, B. Lee, D. Han, S. Yun, and H. Lee, "Character Region Awareness for Text Detection." arXiv, Apr. 03, 2019. doi: 10.48550/arXiv.1904.01941.

[10] M. Bertalmio, A. L. Bertozzi, and G. Sapiro, "Navier-stokes, fluid dynamics, and image and video inpainting," in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, Kauai, HI, USA: IEEE Comput. Soc, 2001, p. I-355-I–362. doi: 10.1109/CVPR.2001.990497.

[11] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition.” arXiv, Apr. 10, 2015. doi: 10.48550/arXiv.1409.1556.

[12] J. Perez-Martin, B. Bustos, and M. Saldana, “Semantic Search of Memes on Twitter.” arXiv, May 20, 2020. doi: 10.48550/arXiv.2002.01462.

[13] C. Sharma, V. Pulabaigari, and A. Das, “Meme vs. Non-meme Classification using Visuo-linguistic Association,” in *Proceedings of the 16th International Conference on Web Information Systems and Technologies*, Budapest, Hungary: SCITEPRESS - Science and Technology Publications, 2020, pp. 353–360. doi: 10.5220/0010176303530360.

[14] G. Koch, R. Zemel, and R. Salakhutdinov, “Siamese Neural Networks for One-shot Image Recognition,” p. 8, 2015.

[15] D. R. Hardoon, S. Szedmak, and J. Shawe-Taylor, “Canonical Correlation Analysis: An Overview with Application to Learning Methods,” *Neural Computation*, vol. 16, no. 12, pp. 2639–2664, Dec. 2004, doi: 10.1162/0899766042321814.

[16] S. D. Das and S. Mandal, “Team Neuro at SemEval-2020 Task 8: Multi-Modal Fine Grain Emotion Classification of Memes using Multitask Learning.” arXiv, May 21, 2020. Accessed: Nov. 23, 2022. [Online]. Available: http://arxiv.org/abs/2005.10915

[17] T. Kudo and J. Richardson, “SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing.” arXiv, Aug. 19, 2018. doi: 10.48550/arXiv.1808.06226.

[18] H. Du, H. Shi, D. Zeng, X.-P. Zhang, and T. Mei, “The Elements of End-to-end Deep Face Recognition: A Survey of Recent Advances.” arXiv, Dec. 27, 2021. Accessed: Nov. 23, 2022. [Online]. Available: http://arxiv.org/abs/2009.13290

[19] Y. Kim, W. Park, M.-C. Roh, and J. Shin, “GroupFace: Learning Latent Groups and Constructing Group-based Representations for Face Recognition.” arXiv, May 25, 2020. doi: 10.48550/arXiv.2005.10497.

[20] O. M. Parkhi, A. Vedaldi, and A. Zisserman, “Deep Face Recognition,” in *Procedings of the British Machine Vision Conference 2015*, Swansea: British Machine Vision Association, 2015, p. 41.1-41.12. doi: 10.5244/C.29.41.

[21] E. Zhou, Z. Cao, and J. Sun, “GridFace: Face Rectification via Learning Local Homography Transformations.” arXiv, Aug. 19, 2018. doi: 10.48550/arXiv.1808.06210.

[22] B.-N. Kang, Y. Kim, B. Jun, and D. Kim, “Attentional Feature-Pair Relation Networks for Accurate Face Recognition.” arXiv, Aug. 17, 2019. doi: 10.48550/arXiv.1908.06255.

[23] W. Scott, “Memotion Dataset 7k,” 2020. https://www.kaggle.com/datasets/abe12cc1fbe90bb29b63a0438028e1a7c989789bf217c9d54dc05aca186cbd4c (accessed Nov. 24, 2022).

[24] V. H. Barella, L. P. F. Garcia, M. C. P. de Souto, A. C. Lorena, and A. C. P. L. F. de Carvalho, “Assessing the data complexity of imbalanced datasets,” *Information Sciences*, vol. 553, pp. 83–109, Apr. 2021, doi: 10.1016/j.ins.2020.12.006.

[25] “CRCV | Center for Research in Computer Vision at the University of Central Florida.” https://www.crcv.ucf.edu/data/Selfie/ (accessed Jun. 07, 2023).

[26] “Meme Generator Data Set.” https://www.kaggle.com/datasets/electron0zero/memegenerator-dataset (accessed Jun. 07, 2023).

[27] “ميمز | Facebook.” https://www.facebook.com/MemesYard/photos/?ref=page_internal (accessed Jun. 07, 2023).

[28] “ميمز مسروقه بس عظمه يسطا | Facebook.” https://www.facebook.com/memes.Stolen.1.0 (accessed Jun. 07, 2023).

[29] “كلاسيكال ارت ميمز | Facebook.” https://www.facebook.com/arabicclassicalartmemes/photos/?ref=page_internal (accessed Jun. 07, 2023).

[30] “ميمز و كوميكس طازة معبرة و ترو | Facebook.” https://www.facebook.com/True.Memes.Comics/photos/?ref=page_internal&_rdc=1&_rdr (accessed Jun. 07, 2023).

[31] “Society Sarcasm | Facebook.” https://www.facebook.com/societyforsarcasm/photos/?ref=page_internal&_rdc=1&_rdr (accessed Jun. 07, 2023).

[32] “Facebook Templates HQ Post for Imgur Albums.” https://ar-ar.facebook.com/TemplatesHQ/posts/pfbid02ykyVfQpp62LAXwNXbzFMVFQD5f2iENdz3TR38sjiSR9cJb6MB8aDieVgYqiFnVyil (accessed Jun. 07, 2023).

[33] “r/FacebookCringe.” https://www.reddit.com/r/FacebookCringe/ (accessed Jun. 07, 2023).

[34] “r/Best of Twitter.” https://www.reddit.com/r/bestoftwitter/ (accessed Jun. 07, 2023).

[35] “r/texts.” https://www.reddit.com/r/texts/ (accessed Jun. 07, 2023).

[36] “r/GoodFakeTexts.” https://www.reddit.com/r/GoodFakeTexts/ (accessed Jun. 07, 2023).

[37] “r/Badfaketexts.” https://www.reddit.com/r/Badfaketexts/ (accessed Jun. 07, 2023).

[38] “Pushshift.io API Documentation,” *pushshift.io*. https://pushshift.io/api-parameters/ (accessed Jun. 07, 2023).

[39] “Image Dimension Visualization Jupyter Notebook Viewer.” https://nbviewer.org/github/OdyAsh/graduation_project/blob/4b0b4d00022d412b5179ff34da234f1eab78e157/dataset_preprocessing_part_2.ipynb#:~:text=ac3ddf010a288be5ce0725854099702355adcfea/dataset_preprocessing.ipynb-,0,As,-we%20can%20see (accessed Jun. 07, 2023).

[40] S. I. Serengil, “RetinaFace.” Jun. 06, 2023. Accessed: Jun. 07, 2023. [Online]. Available: https://github.com/serengil/retinaface

[41] “PaddlePaddle/PaddleOCR: Awesome multilingual OCR toolkits based on PaddlePaddle (practical ultra lightweight OCR system, support 80+ languages recognition, provide data annotation and synthesis tools, support training and deployment among server, mobile, embedded and IoT devices).” https://github.com/PaddlePaddle/PaddleOCR (accessed Jun. 07, 2023).

[42] G. Jenks, “Python Word Segmentation.” Apr. 26, 2023. Accessed: Jun. 07, 2023. [Online]. Available: https://github.com/grantjenks/python-wordsegment

[43] A. Zafar *et al.*, “A Comparison of Pooling Methods for Convolutional Neural Networks,” *Applied Sciences*, vol. 12, no. 17, Art. no. 17, Jan. 2022, doi: 10.3390/app12178643.

[44] R. Sarvepalli, “simple-hierarchy.” Oct. 03, 2021. Accessed: Jun. 07, 2023. [Online]. Available: https://github.com/rajivsarvepalli/SimpleHierarchy

[45] R. Bhutaiya, “In a hurry! XGBoost: Six Easy Steps,” *Analytics Vidhya*, Feb. 06, 2020. https://medium.com/analytics-vidhya/in-a-hurry-xgboost-six-easy-steps-b410b56ef85b (accessed Jun. 07, 2023).

[46] M. Herrmann, “Selenium-python but lighter: Helium.” Jun. 08, 2023. Accessed: Jun. 08, 2023. [Online]. Available: https://github.com/mherrmann/selenium-python-helium

[47] “visual-layer/fastdup: fastdup is a powerful free tool designed to rapidly extract valuable insights from your image & video datasets. Assisting you to increase your dataset images & labels quality and reduce your data operations costs at an unparalleled scale.” https://github.com/visual-layer/fastdup (accessed Jun. 09, 2023).

[48] “NLTK :: Natural Language Toolkit.” https://www.nltk.org/ (accessed Jun. 08, 2023).

[49] Taha Zerrouki, “Qalsadi Arabic Morphological Analyzer and Lemmatizer for Python.” Jun. 02, 2023. Accessed: Jun. 08, 2023. [Online]. Available: https://github.com/linuxscout/qalsadi

[50] mohd4482, “Answer to ‘CSV file with Arabic characters is displayed as symbols in Excel,’” *Stack Overflow*, Feb. 15, 2020. https://stackoverflow.com/a/60243234 (accessed Jun. 08, 2023).

[51] “tf.keras.optimizers.Adam | TensorFlow v2.12.0,” *TensorFlow*. https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam (accessed Jun. 08, 2023).

[52] user650654, “Answer to ‘Cross-Entropy or Log Likelihood in Output layer,’” *Cross Validated*, Jan. 17, 2020. https://stats.stackexchange.com/a/445298 (accessed Jun. 08, 2023).

[53] “scikit-optimize: sequential model-based optimization in Python — scikit-optimize 0.8.1 documentation.” https://scikit-optimize.github.io/stable/ (accessed Jun. 08, 2023).

[54] “Plotly: Low-Code Data App Development.” https://plotly.com/ (accessed Jun. 08, 2023).

[55] “pipdeptree.” tox development team, Jun. 09, 2023. Accessed: Jun. 10, 2023. [Online]. Available: https://github.com/tox-dev/pipdeptree

[56] MIT, “Data-Centric AI vs. Model-Centric AI,” *Introduction to Data-Centric AI*. https://dcai.csail.mit.edu/lectures/data-centric-model-centric/ (accessed Jun. 10, 2023).

[57] “FAQ — Syncthing v1.23.4 documentation.” https://docs.syncthing.net/users/faq.html (accessed Jun. 07, 2023).

Appendix

the hidden folder ".stfolder" specified in Figure 16 is a folder marker required by Syncthing that manages "dataset" folder's metadata to be visible by Syncthing [57]. Syncthing is an open-source file synchronization tool designed to keep files and folders in sync across multiple devices. It allows you to easily share and synchronize data between computers, smartphones, tablets, and other devices connected to the same network or even across different networks. It was used when we were temporarily switching between laptop and PC devices to simultaneously train models, store results, and have these results synchronize across the devices.