TAA - German Traffic Sign Recognition Benchmark (GTSRB)

Diogo Magalhães - 102470
DETI
University of Aveiro
Tópicos de Aprendizagem Automática
Petia Georgieva
d.magalhaes@ua.pt

Emanuel Marques - 102565
DETI
University of Aveiro
Tópicos de Aprendizagem Automática
Petia Georgieva
emanuel.gmarques@ua.pt

***Abstract*—Nowadays autonomous driving is a hot topic and it is becoming increasingly evident that cars will become more and more automated in the future. To achieve this, it is necessary that cars can analyze the environment around them and make the best decisions based on that. One of the most important aspects of this is the ability to recognize traffic signs. This report shows our approach to the machine learning GTSRB - German Traffic Sign Recognition Benchmark challenge. The GTSRB dataset consists of 43 classes of traffic signs, commonly found on German roads, and has over 50000 images. In this report, we analyze how different machine learning models perform in this task and compare their results. We also analyze the results of the models and try to understand why some models perform better than others.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

More and more we see an increased interest in automated vehicle driving both in the manufacturers as well as in common citizens wanting to buy such products. The recognition of traffic signs is an essential task for ensuring the safety and efficiency of this type of vehicle driving.
With the increasing number of vehicles on the road, the need for accurate, fast, and reliable way of traffic sign recognition is becoming more pressing, challenging, and important than ever.
In recent years, with the development of machine learning techniques, great potential for solving this problem has been shown.
In this report, we will explore the usage of different machine learning algorithms used for recognizing traffic signs in the GTSRB dataset, which is a widely used benchmark dataset for this type of task and problem.
The GTSRB dataset contains over 50000 images of 43 different types of traffic signs commonly found on German roads, as shown in figure 1 in a ordered way.

The task of recognizing these signs is challenging due to the variation in lighting, weather, and other factors that can affect the appearance of the signs.
We will investigate several models, such as Logistic Regression, K-nearest neighbors, Neural Networks, and SVMs.
Our primary goal is to evaluate the performance of these models on the GTSRB dataset and compare their results.

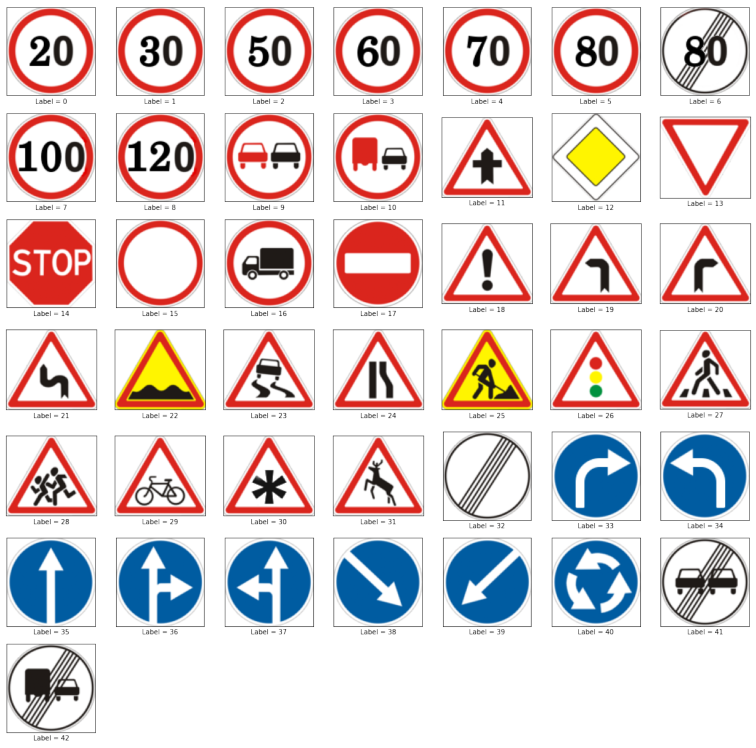


Fig. 1. All 43 different types of signals (classes).

We measure the performance of the models using standard metrics such as accuracy and f1 score. We also analyze the strengths and weaknesses of each model and provide insights into how to improve their performance.
The findings of this report have an important impact on real-world applications, such as autonomous driving as seen before, where accurate and reliable recognition of traffic signs is crucial for ensuring the safety of passengers and pedestrians.
By providing insights into the performance of different machine learning models for traffic sign recognition, this report can help inform the development of more accurate and reliable systems for autonomous driving and other transportation applications.

II. Data Visualization

As said before, we have 43 classes corresponding to a traffic sign. We have a folder with 39209 images to train and a folder with 12630 images to test, this makes around 24.36% of the entire dataset to test.

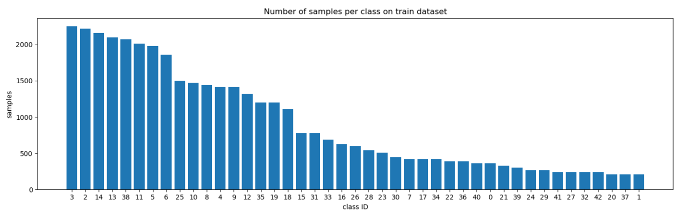


Fig. 2. Quantity of images by class in train dataset

We have a total of 39209 images to train spread across all 43 classes, however, each class does not have the same amount of samples

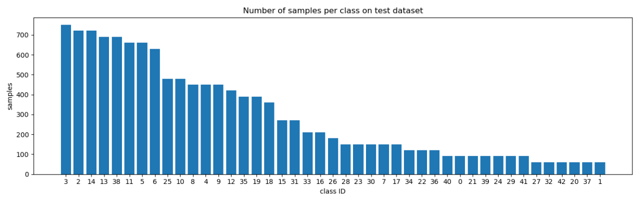


Fig. 3. Quantity of images by class in test dataset

As seen by the graphs above, we have a data imbalance.

We picked 30 random samples from our dataset and it is displayed below in a 6 by 5 grid.

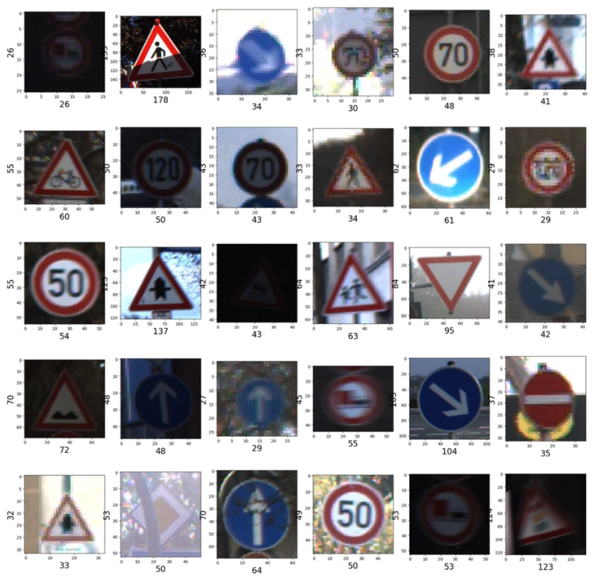


Fig. 4. Random sample of 30 images

III. Data Preprocessing

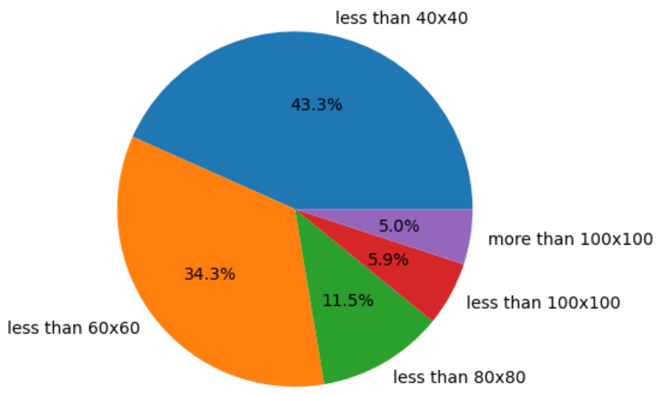


Fig. 5. Pie Graph with image size distribution

Through this pie graph, we can see the distribution of images per format as their sizes are not the same. Most images are small, up to 40 by 40 pixels but we also have some that are bigger reaching 160 pixels width by 160 pixels height, although this represents a small portion of the dataset. This is a problem as machine learning models required the same amount of features across all samples, and in our case, each feature will be a pixel value. The solution was to resize all images to the same size. We chose 30 by 30 pixels as it was the most used proportion, and dimensions bigger than that would be too much to handle and require too much time to train.

When first trying and experimenting with our dataset, we saw the results of some basic models and found out that independently of using pixels with RGB colors or in grayscale, the accuracy of the models wasn't improving or decreasing. Since with RGB in each pixel, the number of features that we would have would be 30*30*3=2700 and with grayscale the number of features would be 30*30=900, we decided to choose grayscale to train our algorithms because they already had large quantity of data and it would take a lot of time to train any of the models.

Since we were working with grayscale pixels, the value of each feature would only vary between 0 and 255. Even though these weren't very big numbers, we decided to always normalize the features. Sometimes we normalized the features by dividing each feature by 255, and some other times we normalized the features with the featureNormalization function of the practical classes that used Z-Score Normalization that is expressed by $Xnorm = (X - Xmean)/Xstd$.

IV. Models

In this report, we investigate the performance of several machine learning models for recognizing traffic signs in the GTSRB dataset.

By comparing the performance of different machine learning models on the traffic sign recognition task, this report can provide valuable insights into the strengths and weaknesses of these models for real-world applications such as autonomous driving.
Due to class imbalance, accuracy is not a good metric, so we are using F1 score.
We decided to leave convolution neural networks for the next project, even though CNNs are now the state of the art in this situation as they usually yield higher accuracy.

A. *Logistic Regression*

Logistic Regression is a classification algorithm used for binary classification problems, that is when the output can only take two possible values. It is a predictive analysis algorithm based on the concept of probability.
Unlike linear regression which outputs continuous number values, the logistic regression calculates a weighted sum of the input features and then applies a sigmoid function that returns a probability value between 0 and 1. This probability represents the likelihood of belonging to a class, basically if the probability is higher than one threshold, the sample is classified as one class, otherwise, it is classified as the other class.
Since we have 43 classes it is not possible to use logistic regression by itself, because it is a binary classifier. To solve this problem we use the one-vs-all approach, which means that we train a model for each class and then choose the class in which it model has the highest probability.

We decided to split the training dataset into a training set and a validation set, so we could evaluate the performance of the model during training and choose the best hyperparameters. We used 80% of the training dataset for training and 20% for validation making sure that the proportion of samples for each class was the same in both sets.

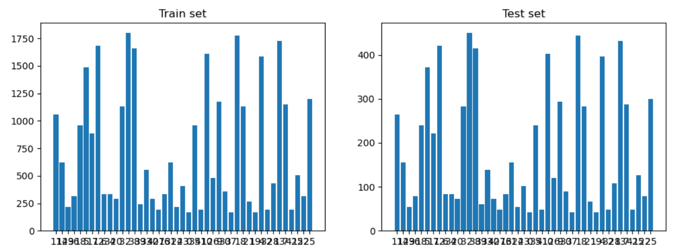


Fig. 6. Logistic Regression Proportional datasets

The logistic regression model is trained by minimizing the loss function, using optimization algorithms like gradient descent. To find the best hyperparameters for our model, we used GridSearch and tried several combinations of the values for the learning rate and the regularization parameter. Several plots were created since sometimes the number of labels would get in front of the results.
Here are two examples of the generated plots:

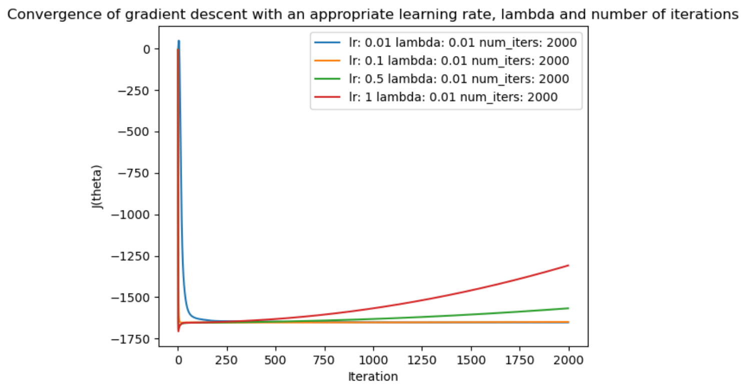


Fig. 7. First Example of Gradient descent with some combination of hyperparameters

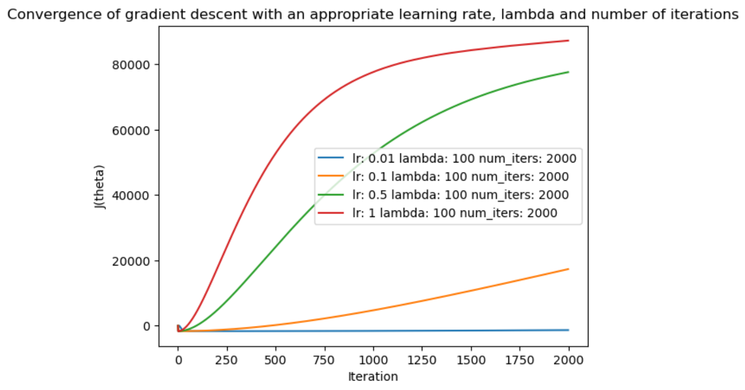


Fig. 8. Second Example of Gradient descent with some combination of hyperparameters

After analyzing the convergence of the gradient descent with each of the hyperparameters combination, we tried to choose the best one and ended up choosing the value 0.01 for the learning rate and 0.01 for the regularization parameter.

After having the hyperparameters chosen, we trained the model with the entire training dataset.

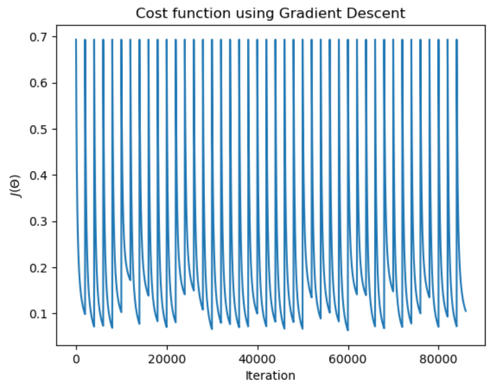


Fig. 9. Cost Function Using Gradient descent of trained model

We started by analyzing the cost function for each class, and we got to the conclusion that, even though it didn't fully converge to 0, it was a good result for this model, because even with other hyperparameters combinations, the results would be similar or worse.
We were already expecting this result because the logistic regression is a simple model, and it is not able to learn many complex patterns, and the dataset is very complex, with a lot of images and a lot of classes.

We then analyzed the confusion matrix for the training dataset.

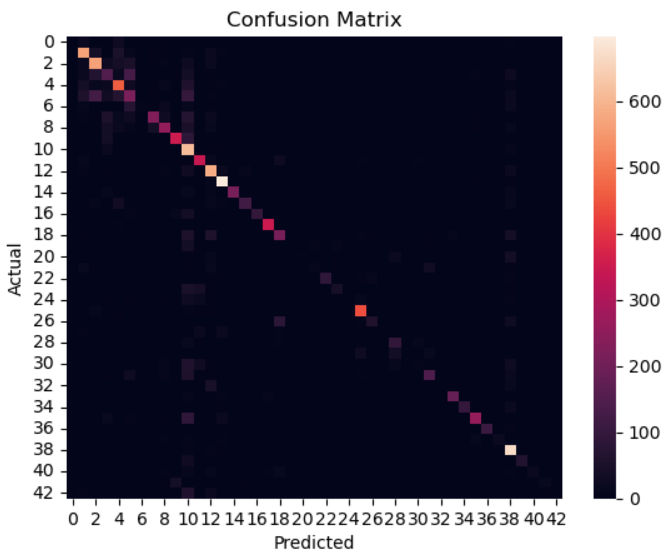


Fig. 10. Confusion Matrix using Logistic Regression

Analyzing the confusion matrix, we saw that some classes were not being chosen at all, for example when the real class was class 0, the model was predicting other classes like class 4. After seeing this bad results we decided to analyze the percentage of accuracy in each class.

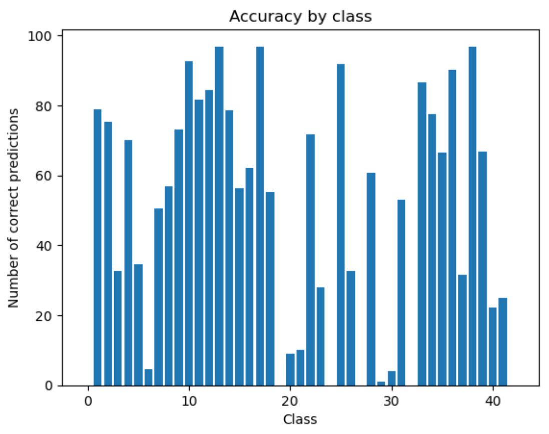


Fig. 11. Accuracy for each Class using Logistic Regression

As we were expecting, some classes, like class 0, had 0% of accuracy, while other classes had much more percentage of accuracy. With this, we realized that this model could not be used to classify all of these traffic signs at all.
Even though the model had an accuracy of nearly 74% overall with the original test dataset, since it is not capable of classifying some classes, it is not a good model for this problem.

We then decided to use sklearn's LogisticRegression model, to see if we could get better results.
The method used was pretty much the same. We started by finding out which were the best hyperparameters using GridSearch.

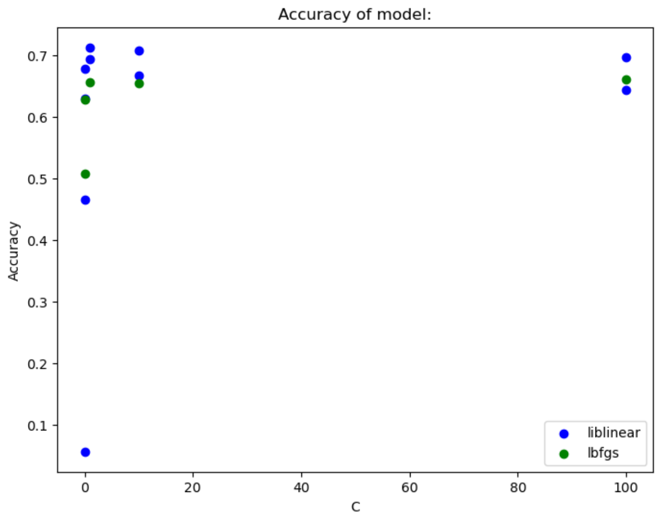
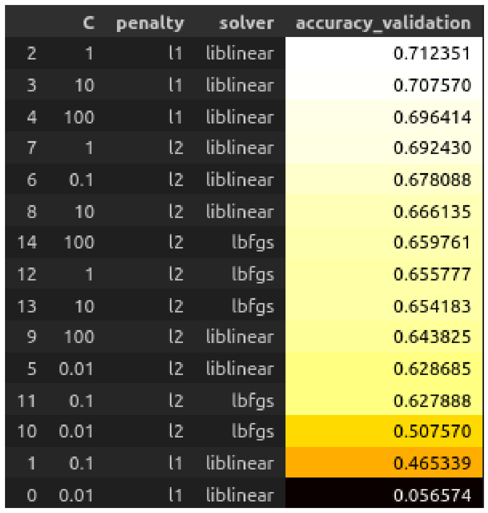


Fig. 12. Accuracy of each model cross-validation



	C	penalty	solver	accuracy_validation
2	1	l1	liblinear	0.712351
3	10	l1	liblinear	0.707570
4	100	l1	liblinear	0.696414
7	1	l2	liblinear	0.692430
6	0.1	l2	liblinear	0.678088
8	10	l2	liblinear	0.666135
14	100	l2	lbfgs	0.659761
12	1	l2	lbfgs	0.655777
13	10	l2	lbfgs	0.654183
9	100	l2	liblinear	0.643825
5	0.01	l2	liblinear	0.628685
11	0.1	l2	lbfgs	0.627888
10	0.01	l2	lbfgs	0.507570
1	0.1	l1	liblinear	0.465339
0	0.01	l1	liblinear	0.056574

Fig. 13. Dataframe of accuracy of each model

After analyzing each model, we concluded that the best model was the one with the hyperparameters C = 0.1 and penalty = l1, and solver = liblinear since those were always in front of the other in terms of validation accuracy.

After finding out which were the best hyperparameter we trained the model with the entire training dataset and once again we analyzed the confusion matrix and the accuracy for each class.

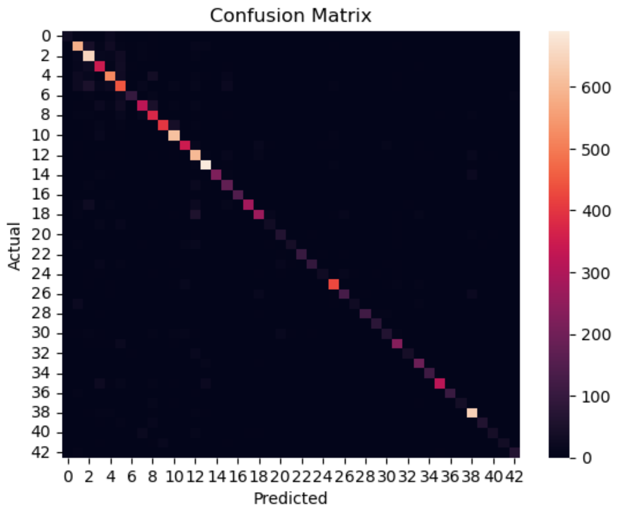


Fig. 14. Confusion matrix using sklearn

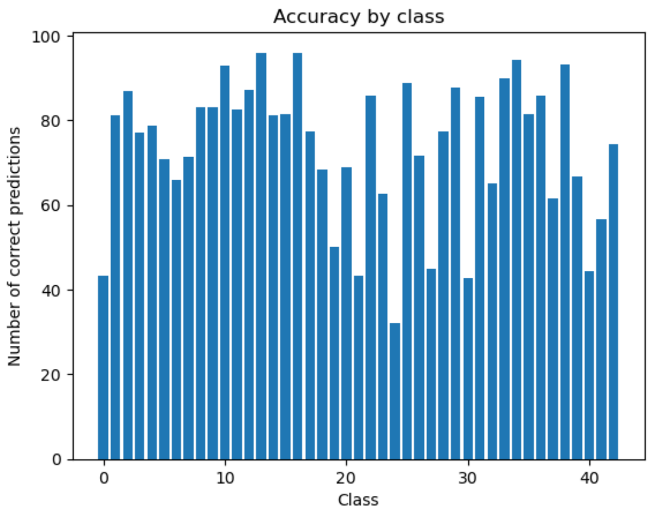


Fig. 15. Accuracy of each class using sklearn (%)

This time the results looked much better, since the model was already able to sometimes classify correctly all classes. This time the model had an accuracy of nearly 81% overall with the original test dataset, which is a much better result than the one we got with our model.

B. Artificial Neural Networks

Artificial Neural Networks try to mimic the learning process of the human brain by utilizing most of the math that is behind logistic regression. This imagines layers of neurons that activate according to the weights of the previous neurons, the first layer is the input layer and its values never change as this layer contains the features of our dataset, then exists a set of hidden layers that can have an arbitrary number of neurons and it is possible to have 1 or more of this layer, and at last, there is the output layer that assigns a probability to each of the labels. In a Dense Connected Layer, each neuron receives all the values from previous neurons multiplied by weights and summed to a bias, then the computed values pass across an activation function and the result is forwarded to the neurons on the next layer. After this process, there is a deviation from the real ground truth and errors but the tweaks to adjust the weights are propagated to the previous layers in a process called backpropagation.
To calculate the error it's crucial to have a set of data that wasn't used in the input layer, so we split the original train dataset to have a validation set that allows our NN models to perform cross-validation which is a practical and reliable way for testing the predicting power of models.

There are many possible architectures for this type of models, 1 hidden layer, 2 hidden layers, or more, and its hard to say *a priori* which parameters will give the best results, so exhaustive experimentation needs to be performed.

We started with a simple model of only one hidden layer but had to figure out what would be the best number of neurons in that layer. For that, we trained the model varying the number of neurons in the hidden layer, from 400 to 1000 with steps of 100.

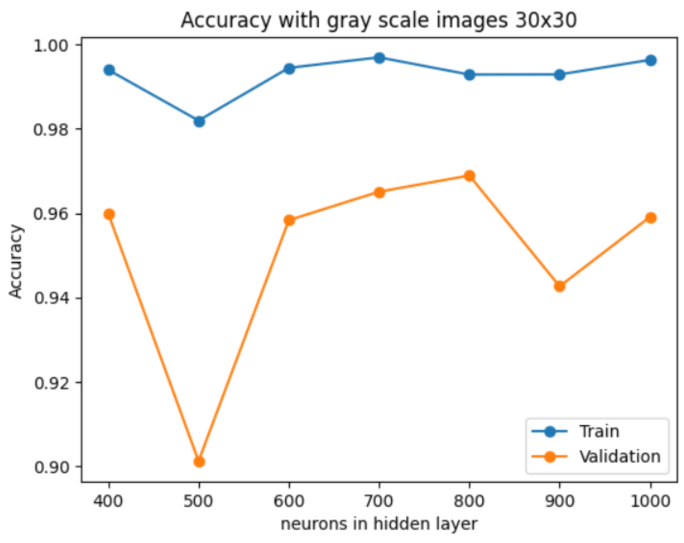


Fig. 16. Accuracy with different quantity of neurons

Through the analysis of this graph, we concluded that the best number of neurons to use in the hidden layer was 800 as it is the value that outputs be the best accuracy for the validation process.

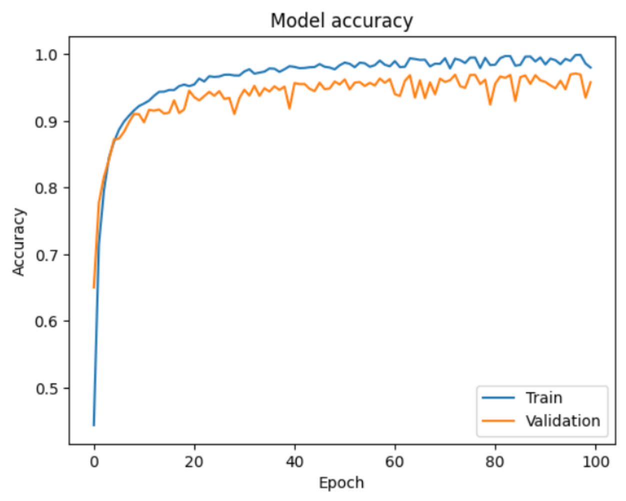


Fig. 17. Accuracy of model with 800 neurons

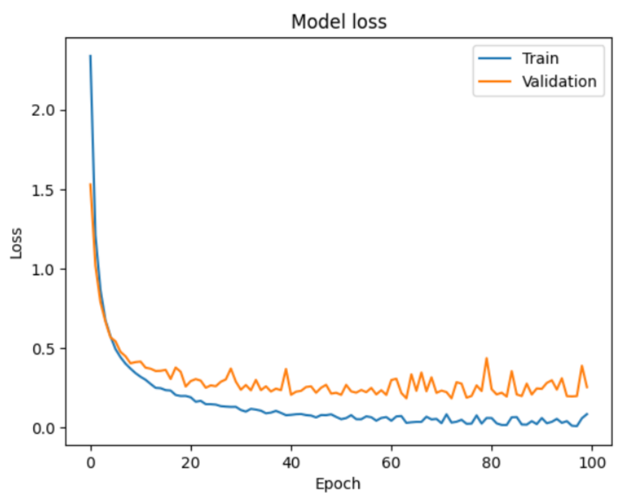


Fig. 18. Loss of model with 800 neurons

These graphs show the accuracy and loss function along epochs. We clearly see that the accuracy improves with the number of epochs, but begins to stagnate at a certain amount.

With the same conditions we trained changing the learning rate which is a parameter that controls how fast the loss function approaches its local minimum. Through this plot, we saw that in our case a value of 0.005 is what works best as its approaches faster a low value of loss function as we can see in figure 19.

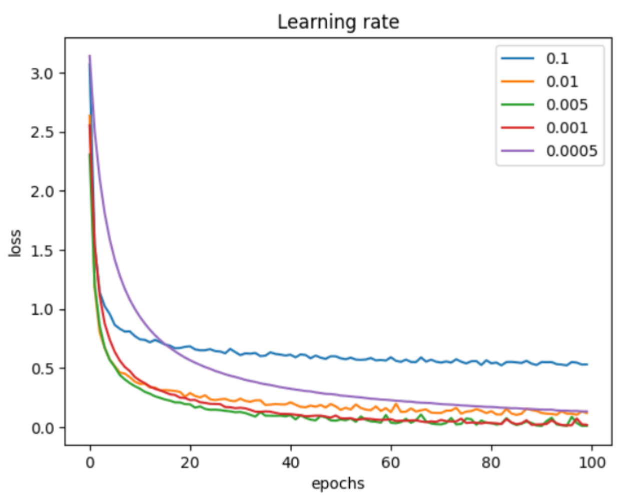


Fig. 19. Loss of each learning rate

After that we tried to discover what would be the best regularization parameters

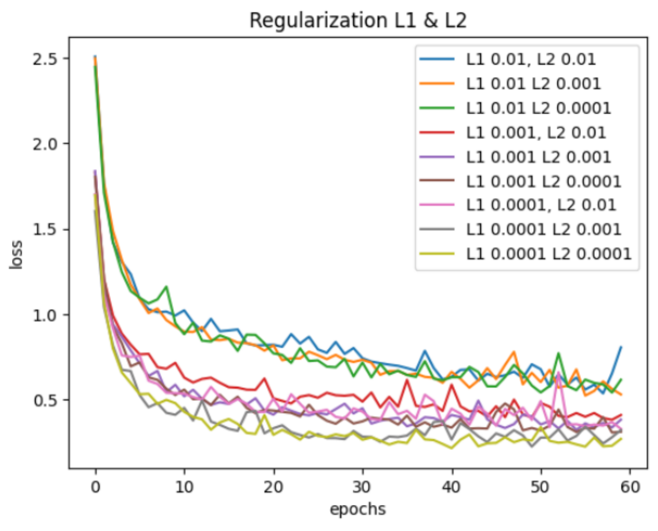


Fig. 20. Loss of each learning rate

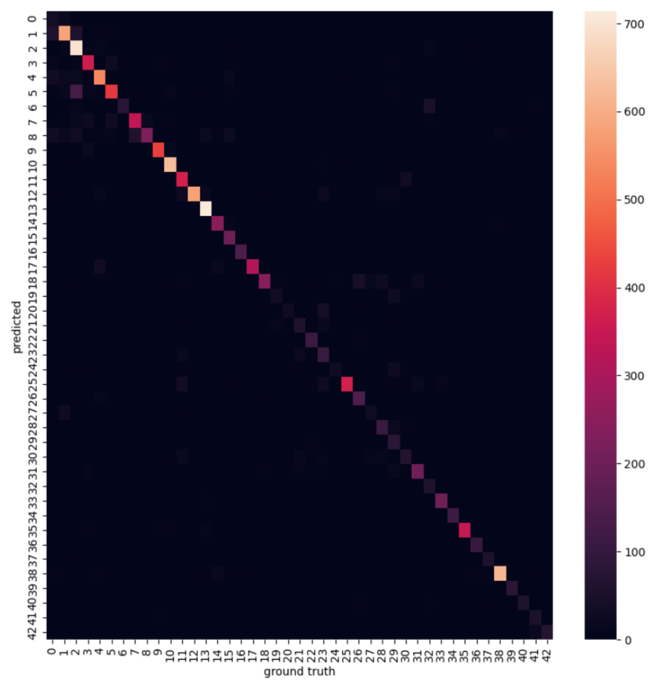


Fig. 21. Confusion Matrix using Neural Network

Analyzing the confusion matrix, we see that mos classes are being well evaluated and predicted. This ended up being our best model with an accuracy of near 84%.

C. Support Vector Machine

Support Vector Machine (SVM) is a type of supervised learning algorithm that can be used for both classification and regression types of analysis.
The objective of the SVM algorithm is to find a hyperplane in N-dimensional space (N — the number of features) that distinctly separates the data points from each class. The data points which are closest to the hyperplane are called support vectors and are used to determine the margin for the classifier. The goal is to find the optimal hyperplane that maximizes the margin between the two classes.
Sometimes classes are not linearly separable by a hyperplane, so we need to transform the data into a higher-dimension space. To solve this it's used the kernel trick. The kernel trick is a method by which we can use a linear classifier to solve a non-linear problem. It transforms the data into another dimension that has a clear dividing margin between classes of data. The kernel trick is used to find a suitable hyperplane in the new dimension.
For binary classification, one hyperplane would be enough to separate the two classes and to predict the class of a new data point. Since we have 43 classes, one unique hyperplane wouldn't be enough to separate all 43 classes so we used a one-vs-one approach, which means that we train a model for each pair of classes and then we predict the class that has the most wins. For this kind of problem, we have to divide the problem into N(N-1)/2 classifiers, in our case, 903 classifiers were needed.

In our approach to the problem, we started by separating the training dataset into two datasets, one for training and another for validation on a scale of 80%/20%, making sure that both datasets had the same proportion of images in each class to avoid some classes being forgotten when training or validating our model.

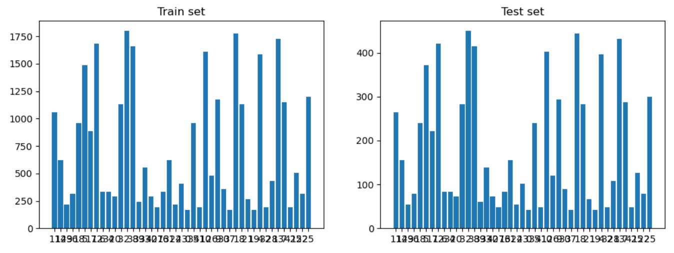


Fig. 22. SVM Proportional Datasets

The first step to finding the best model was to find out which would be the best kernel, C, and gamma values. To do this, we started by using GridSearchCV, which is a method that allows us to find the best parameters for a model by testing all the possible combinations of parameters. The problem was that since our dataset was huge and had a lot of features, GridSearchCV was taking too long to run, so what we did was first create a subset of the training dataset with a small number of images (always maintaining the proportion in each class) and then we used GridSearchCV to find the best parameters for that subset. For each combination of parameters, we used cross-validation with 5 folds to find the hyperparameters with the best accuracy in the validation dataset.

We got the following results:

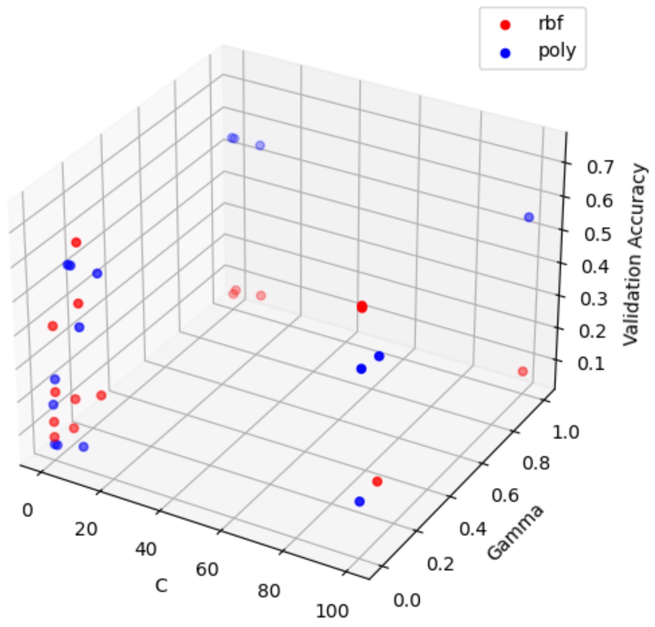
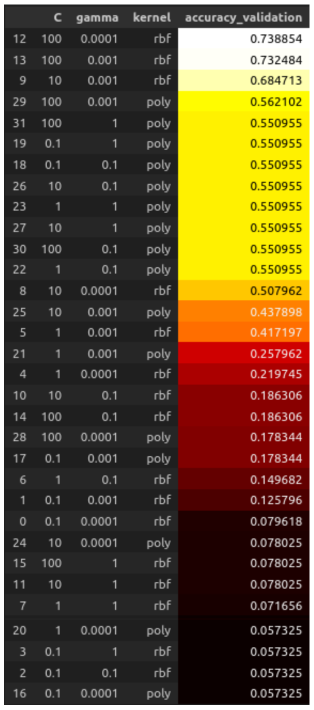


Fig. 23. SVM Hyperparameters accuracy comparation



	C	gamma	kernel	accuracy_validation
12	100	0.0001	rbf	0.738854
13	100	0.001	rbf	0.732484
9	10	0.001	rbf	0.684713
29	100	0.001	poly	0.562102
31	100	1	poly	0.550955
19	0.1	1	poly	0.550955
18	0.1	0.1	poly	0.550955
26	10	0.1	poly	0.550955
23	1	1	poly	0.550955
27	10	1	poly	0.550955
30	100	0.1	poly	0.550955
22	1	0.1	poly	0.550955
8	10	0.0001	rbf	0.507962
25	10	0.001	poly	0.437898
5	1	0.001	rbf	0.417197
21	1	0.001	poly	0.257962
4	1	0.0001	rbf	0.219745
10	10	0.1	rbf	0.186306
14	100	0.1	rbf	0.186306
28	100	0.0001	poly	0.178344
17	0.1	0.001	poly	0.178344
6	1	0.1	rbf	0.149682
1	0.1	0.001	rbf	0.125796
0	0.1	0.0001	rbf	0.079618
24	10	0.0001	poly	0.078025
15	100	1	rbf	0.078025
11	10	1	rbf	0.078025
7	1	1	rbf	0.071656
20	1	0.0001	poly	0.057325
3	0.1	1	rbf	0.057325
2	0.1	0.1	rbf	0.057325
16	0.1	0.0001	poly	0.057325

Fig. 24. SVM Hyperparameters Dataframe

As we can see, the best kernel was clearly the RBF kernel, since it was in all of the Top 3 combinations of parameters and those had much better validation accuracy. High values of C were also better, also winning by a large validation accuracy margin when compared to the other values of C. The gamma values were not so clear, but the best values were 0.0001 and 0.001 when using big values for C. After analyzing those results we decided to use the option with the best validation accuracy, which was kernel='rbf', C=100, and gamma=0.0001.

After finding out which was the best combination of hyperparameters, we used those parameters to train a model with the entire train dataset and then we tested it with the entire validation dataset.

We got the following results:

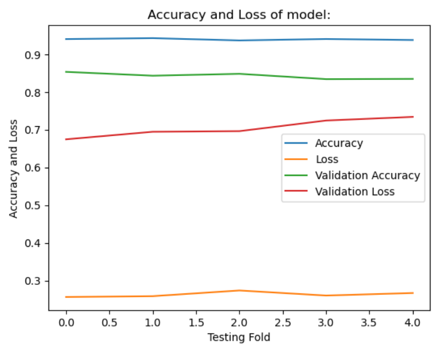


Fig. 25. Accuracy and loss of validation data using SVM

As we can see the model was really good at predicting the accuracy in the training dataset, having an accuracy of nearly 95%. On the other hand, it was not so good at predicting using the validation dataset, having an accuracy of nearly 80%. This means that the model ended up overfitting the training dataset, which means that it learned too much from the training dataset and ended up not being able to generalize well to new data.

There are several explanations to this like:
- This is a common problem when using SVM since it's a model that is very sensitive to the hyperparameters and it's very easy to overfit the data. Even though we trying to check for the best hyperparameters.
- The dataset is very big and has a lot of features, so it's very easy to overfit the data because the model might be learning noise instead of the actual patterns.

Even though the validation loss is not very good, the model still has a good accuracy, when compared to some other models that we tried.

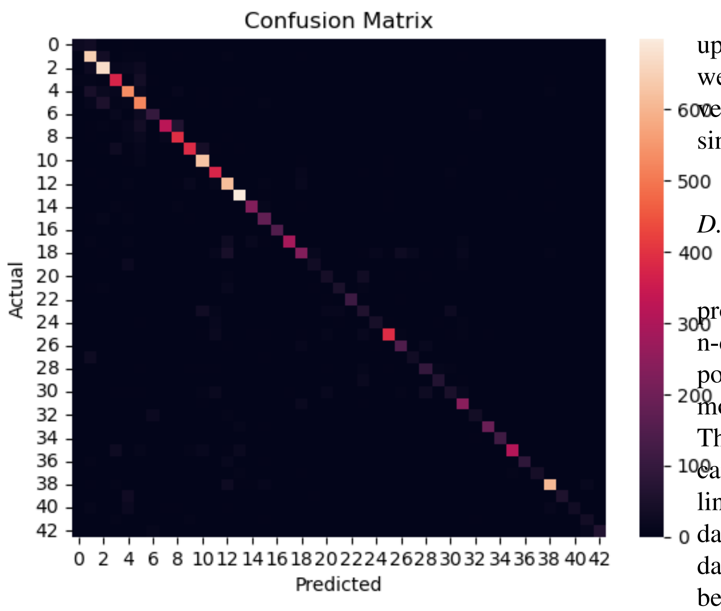


Fig. 26. SVM Confusion matrix

With the confusion matrix, we can see that the model was able to predict most of the classes correctly and that it has a medium accuracy for most cases.

Another way to better view the accuracy of the model, since not all classes have the same amount of images is to check the accuracy for each class in particular.

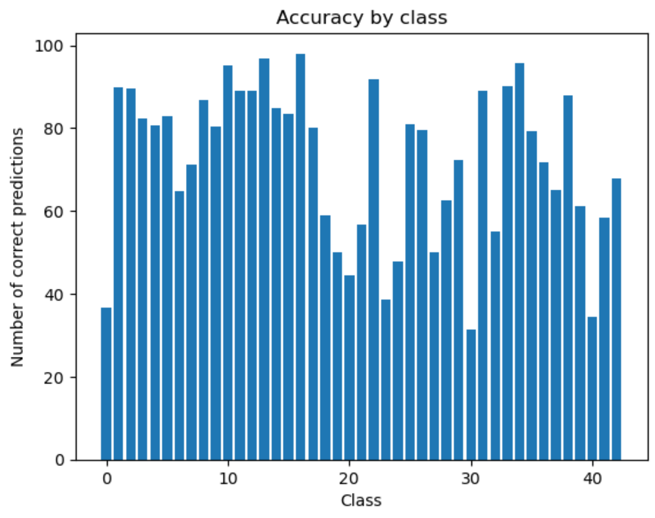


Fig. 27. SVM Accuracy (%) of each class

As we can see, the model has different accuracies for each class, but even though some of them are very high, as near as 98%, others, like classes 0 and 30, have very low accuracies, below 40%.

Overall, the model, when using the real test dataset, ended up having an accuracy of 81%, which is not a bad result, since we are working with a huge dataset and with images that are very similar to each other, but it's also not a very good result, since we were expecting to have better accuracy.

D. *K-nearest neighbors*

This method is very simple and is used in classification problems. It checks the neighbors of a given point in an n-dimension space and by comparing the distances to other points the most k near examples vote and the class that has more votes wins.
This seems a good method but it doesn't learn anything, it cannot improve itself over time. Its computational time scales linearly $O(N)$ as it has to compare each test sample to every data point in the training set making it inefficient with large datasets. Also, there's no deterministic method to choose the best k so a try-and-guess approach is needed.
This method is good when the number of features is low, however, our case is the opposite, so we expect a low performance using this approach.

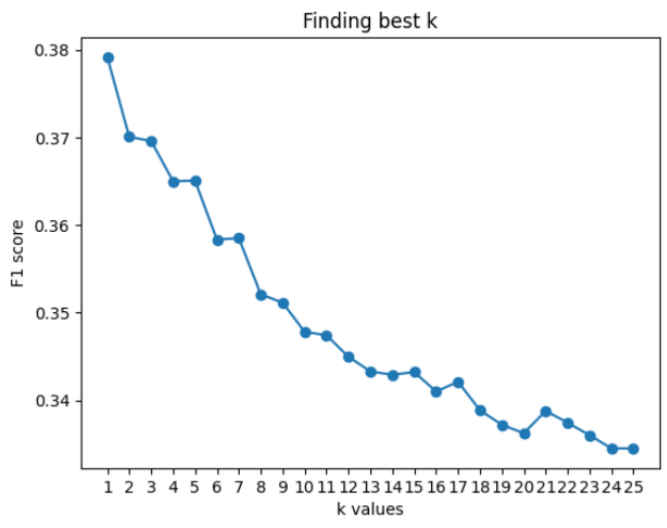
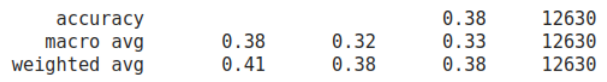


Fig. 28. Trying multiple K to find the best score



```
    accuracy                           0.38     12630
   macro avg       0.38      0.32      0.33     12630
weighted avg       0.41      0.38      0.38     12630
```

Fig. 29. K-nearest neighbors classification report

As we can see from the graph, K-nearest Neighbors doesn't yield good results with any value of k, in fact no more than 37% of the F1 score was possible to achieve. These results were according to our expectations.

V. Results

	Logistic Regression	Artificial Neural Network	K-Nearest Neighbors	SVM	State of the Art
Validation accuracy	85.15%	˜95%	-	˜84%	˜99.99%
Test accuracy	72.88%	˜84%	˜37%	81.08%	˜98%

After analyzing the table above, we got to the conclusion that:
- Our models are worst than the models that already exist, which are almost flawless at predicting the correct class
- Our worst model is the k-nearest neighbors model, which is a very simple model and is bad for images since it is hard to find a specific pixel with a lot of information
- Our Logistic regression was actually a model that performed well compared with the other models, even though we didn't have much expectations on it
- SVM had a good result but takes a lot of time to train, so for this kind of report it was hard to work with
- Neural Networks, as expected were our better model, but they could still be improved if we used Convolutional Neural Network

VI. Conclusion

As referenced before, Convolution Neural Networks would very likely outperform any of the models we tested, but our objective was to apply the models taught in classes so far, and as we will have another project and learn CNN more in-depth in the upcoming class we postponed the exploration of this method. This topic, and this dataset in specific, is very popular and a bit complex so it is natural that our models are weak and professionals in the field get much better results. Data preprocessing could make a substantial difference in our models' performance, like images rotation, changing color, shades, adjust brightness and contrast but due to lack of experience with these kinds of transformations in machine learning data and because some models took a very long time to train, only some transformation were applied such as resizing all images to the dimensions. This project was a good opportunity to apply and experiment with various techniques and machine learning models that we learned in class. This field has captivated our interest, and being able to visualize and work with different models has allowed us to gain valuable experience. Both students contributed equally to the elaboration of this project.

References

[1] 1.4. Support Vector Machines
https://scikit-learn.org/stable/modules/svm.html

[2] sklearn.linear_model.LogisticRegression.
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[3] Mykola, GTSRB - German Traffic Sign Recognition Benchmark
https://www.kaggle.com/datasets/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign?resource=download

[4] 3.1. Cross-validation: evaluating estimator performance
https://scikit-learn.org/stable/modules/cross_validation.html

[5] Menon A., Logistic Regression in Machine Learning using Python
https://towardsdatascience.com/logistic-regression-explained-and-implemented-in-python-880955306060

[6] ChatGPT
https://chat.openai.com/

[7] Use of decision trees for classifying images
https://datascience.stackexchange.com/questions/64289/use-of-decision-trees-for-classifying-images

[8] Arcos-García Á., Álvarez-Garcia J., Soria-Morillo L., Deep neural network for traffic sign recognition systems: An analysis of spatial transformers and stochastic optimisation methods
https://www.sciencedirect.com/science/article/abs/pii/S0893608018300054

[9] GTSRB (German Traffic Sign Recognition Benchmark)
https://paperswithcode.com/dataset/gtsrb