Classifying German Traffic Signs

Project 1 Report

Gabriel Silva (85129) & Gonçalo Vítor (85119)

Universidade de Aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Course: Machine Learning

Professor: Pétia Georgieva

Abstract

This project makes use of the German Traffic Sign Recognition Benchmark (GTSRB), a dataset composed of images of traffic signs (one per image) in several illumination, weather, obstruction, distance and rotation conditions. In this project we use the GTSRB[10] dataset to train several classification models and study how each one performs.

This report will start by describing the dataset and the work that was done to prepare it for further use and will continue, then, talking about the models we used and how we trained and optimized them.

The work hereby reported was developed with equal efforts by both authors.

Contents

1 Data

1.1 Data Description

This project uses the GTSRB[10] dataset. This dataset is composed of thousands of images, each one containing one single traffic sign. There are a total of 43 different traffic signs in it.

The dataset provides three main folders: train, test and meta folder. The meta folder contains 43 images, one per each traffic sign that appears in the training and test folders. The test folder is not meant to be used as a test set when training a model. The reason for it is that the images in this test folder are not labeled. The images in this folder are still used in a simple application demo, that shows a model working and making predictions. The train folder is organized in 43 sub-folders. Those 43 folders' names are the number that corresponds to the label of the images inside, and so we know the label of a given image by the name of the folder it resides in.

1.2 Data Preprocessing

The biggest problem faced with the dataset is that the images are of various sizes. By using each pixel of an image as a feature for our models, the fact that the images have different sizes, and therefore a different amount of features, is a problem that must be fixed. The image resizing thecnique used resizes the images to a common size, padding them with grey stripes, if necessary. We reused a solution made by a kaggle user named gauss256[11]. In the previous reference ([11]) we can see a notebook where the components to an image preprocessing algorithm are presented. Besides image resizing it also presents a solution for image normalization, but this problem was faced in a different way in this project.

The normalization is done with a tool from the sklearn library, the StandardScaler. The StandardScaler performs normalization by removing the mean and scaling the the data to unit varience[1]. Any given sample x will be transformed according to the following expression:

$$z = (x - u)/s$$

where u is the mean and s is the standard deviation of the training examples.

Finally, the amount of data also proved to be a problem. Training models on this much data took too long on ordinary machines and so we decided to

work only on 10% of it. To do so, we first collected and processed all the images, and then shuffled them, allowing us to use 10% of the dataset, in a well divided manor, by simple using the first tenth of the samples of the shuffled dataset. The reason why shufflig is necessary is because the simplest way to retrieve the dataset is by traversing all folder and process all images of each folder. By doing so, the images of a particular class (and also, of a particular folder) will be added to our data structures together, which makes it so that just using the first tenth of the dataset will also mean having images of just a few classes, which is a total missrepresentation of the dataset.

Figure 1: A sample of our data, post-processing

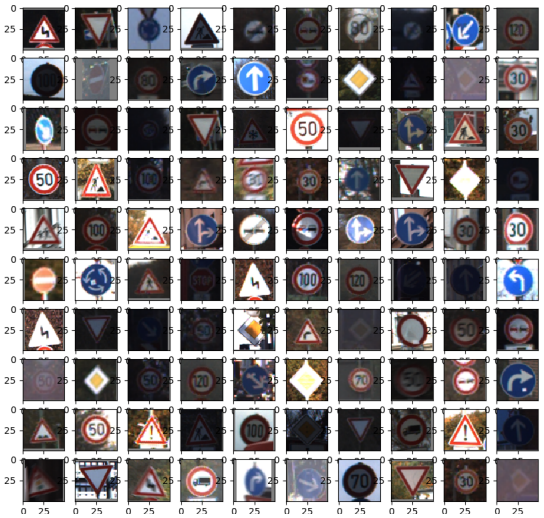
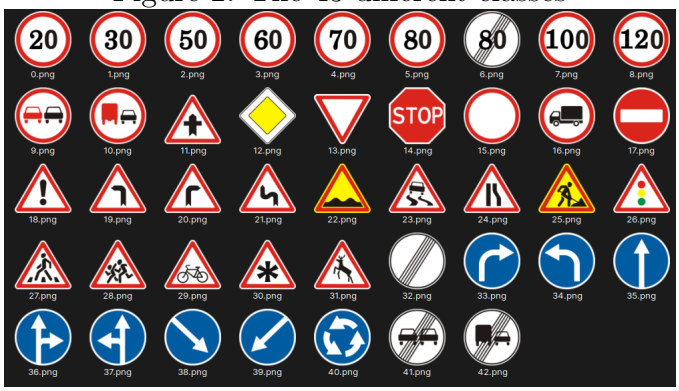


Figure 2: The 43 different classes



2 Models

2.1 Used Models

In this projecct, we used 3 different classification algorithms: Logistic Regression, Support-Vector Machine (SVM) and a Neural Network.

2.1.1 LogisticRegression

The sklearn LogisticRegression[2] class implements a regularized version of a logistic regression that provides several solvers(optimization functions). In this project we used the 'lbfgs' solver. One thing to note is that this is a regularized version of a logistic regression, which uses a parameter C that is the inverse of the regularization strength, meaning the smaller it is, the stronger the regularization is. This means that it's not actually possible to have a 0 regularization factor, but, as C tends to infinity the regularization factor tends to 0, and so, a very high value of C will provide very small regularization. This is how we trained our model to not have regularization. In our case we used $C = 9999$[3]. This class also allows the use of different loss functions. We made use of the L2[3] one.

2.1.2 SVC

The sklearn SVC[4] class implements a Support-Vector Classifier. It has a similar parameter C as the Logistic Regression in subsubsection 2.1.1 that works the same way. We used the same strategy to train a non-regularized version of the model. This model has the particularity that *'the fit time scales at least quadratically with the number of samples and may be impractical beyond tens of thousands of samples.'*[4]. The sklearn documentation recomends using the LinearSVC[5] for larger than around ten thousand sample datasets, but when doing so, we did not notice any meaningful time performance improvements. This was one of the factors that lead us to decide to use a much smaller fraction of the dataset. We decided to use a linear kernel, since it is less time consuming[12].

2.1.3 MLPClassifier

The MLPClassifier[6] class is a multi-layer perceptron classifier. Much like the LogisticRegression classifier, it allows the use of several solvers, including

'lbfgs' and Stochastic Gradient Descent (SGD). We used SGD simply because it was the one we were more familiar with. It gives us the possibility of using various activation functions, like the logistic sigmoid functions, the hyperbolic tangent function or the rectified linear unit function. Once more, we used the logisitc function since it was the one we knew best. It also let's us change the regularization parameter alpha and the learning rate, which we varied in our optimization tests. As to the amount and size of hidden layers used, we used only one hidden layer[13] with 100 neurons, which is a vlaue between the size of the input and output layers. We also noticed that when using a larger hidden layer, the training time increased, and so we decided to settle with only 100 neurons. Finally, we also decided to try out some parameters that were unknown to us until now: momentum and nesterov's momentum. Momentum speeds up gradient descent by trying to make it take a more direct route towards the local optimum.

2.2 Model Training and Hyper-Parameter tunning

To optimize the models we used the GridSearchCV[7] class from sklearn, which uses an estimator(model) and a dictionary of parameters and performs an exhaustive search over all the possible parameter combinations and scores the models using cross-validated optimized parameters. The GridSearchCV can use several scoring metrics, including the accuracy, f1 score and logistic loss that were all used by us. Many more scoring metrics are available[8]. We also used the k-fold cross-validation method with 3 folds[9]. We used a small value because the more folds the longer it takes train each model.

2.2.1 LogisticRegression

For our Logistic Regression model, we tested using multiple values of C : [0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 9999]. Both when optimizing the model and when training the final one to test on the test set we operated on only 10% of the data and used the parameters mentioned in subsubsection 2.1.1. In Figure 3 we show our GridSearchCV results.

In the results of Figure 3, we see the value measured for the three scores given to the GridSearchCV for each value of the regularization parameter C of our model. We can see that most models have the same accuracy and F1 score, around the 77% mark and so we selected, as the best model, the model with the lowest logistic loss, which is the model with $C = 0.03$.

Figure 3: Logistic Regression optimization

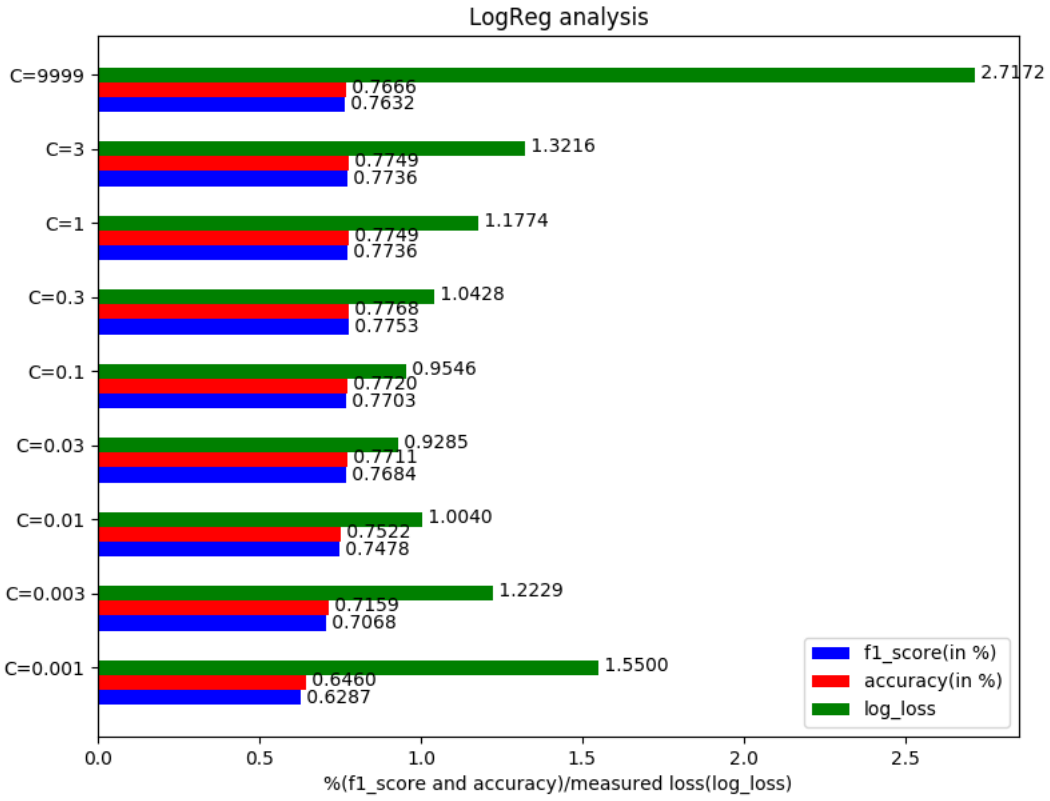


Table 1: Logistic Regression's Scores

Accuracy(%)	F1(%)	Recall(%)	Precision(%)
0.796	0.795	0.796	0.812

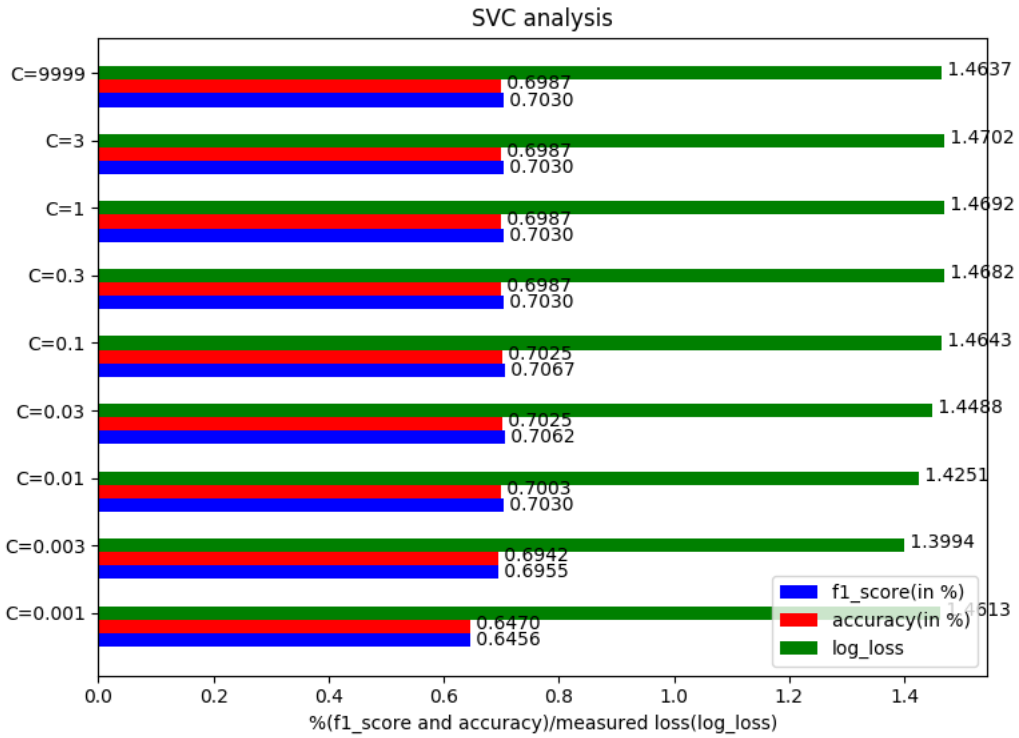
When training the model outside of the GridSearchCV, we reached the resutls in Table 1 with our test set.

The model was trained in 240 seconds (4.0 minutes). The model fitting function, or the model itself don't provide any loss value record through iterations, which made it so we couldn't see the loss function converging, but a warning is raised whenever a model fails to converge. We made sure we gave the model enough iterations so that it would converge, which we are sure happened given the absence of the warning mentioned.

2.2.2 SVC

For our SVC model, we tested, again, using multiple values of C : [0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 9999]. Both when optimizing the model and when training the final one we operated on only 10% of the data and used the parameters mentioned in subsubsection 2.1.2. In Figure 4 we show our

Figure 4: SVC optimization



GridSearchCV results.

In the results of Figure 4, we see the value measured for the three scores given to the GridSearchCV for each value of the regularization parameter C of our model. We can see that both the f1 score and the accuracy peak at 70% and the logistic losses are all very similar. We selected as our best fitting model the one that keeps both the accuracy and the f1 score above 70% and has the lowest loss, which is the model with $C = 0.01$.

Table 2: SVC's Scores

Accuracy(%)	F1(%)	Recall(%)	Precision(%)
0.774	0.781	0.774	0.805

When training the model outside of the GridSearchCV, we reached the results in Table 2 with our test set.

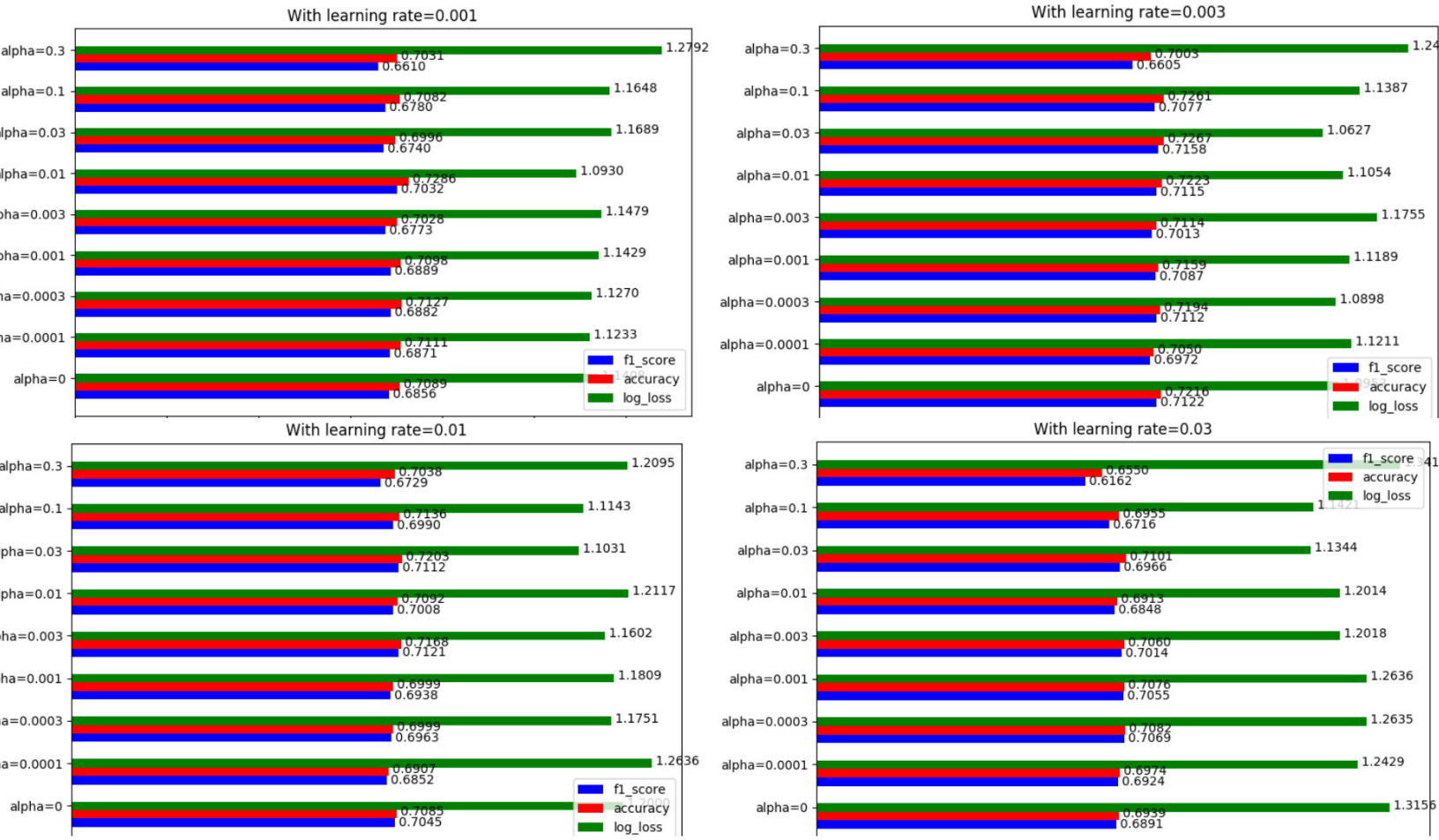
The model was trained in 54 seconds (less than 1 minute). Like in the Logistic Regression model we didn't have access to a loss history, and couldn't see the loss converging, but again, given the absence of the warning that is raised when the model fails to converge, we are pretty sure it did, in fact, converge.

2.2.3 MLPClassifier

For the MLPClassifier we varied two parameters: alpha (the regularization parameter) and the learning rate. We tested the values [0, 0.0001, 0.0003, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3] for alpha and [0.001, 0.003, 0.01, 0.03, 0.1, 0.3] for the learning rate. We also considered experimenting several hidden layer sizes[13] and using more learning rates, in particular, smaller ones. The reason why we decided against these ideas was the amount of time the optimization would take. With the given possibilities for alpha and the learning rate we already have 54 different models, and with 3-fold validation that gives 162 models to fit, each one taking a few minutes (with the exception of the ones with the highest leaning rates, which we found to be too high). If each model were to take 3 minutes (which is very optimistic) to train we are already looking at 162*3=486 minutes(over 5 hours) to finish the exhaustive search. If we add more learning rates, in particular, smaller ones (smaller learning rates make it so that the model takes longer to converge) and also experiment with more hidden layer sizes we are looking at thousands of minutes to finish the search, even with only 10% of the dataset. For this reason we only trained our MLPClassifier with a hidden layer with 100 neurons. Just like in the previous examples, both when optimizing the model and when training the final one we only used 10% of the data.

In Figure 5 we can see the plots for the different scores used for each learning rate and alpha used. In them we notice that the smaller learning rates are the ones that give us the best results. The two highest learning rates are not shown, but their results were the worst. When observing the

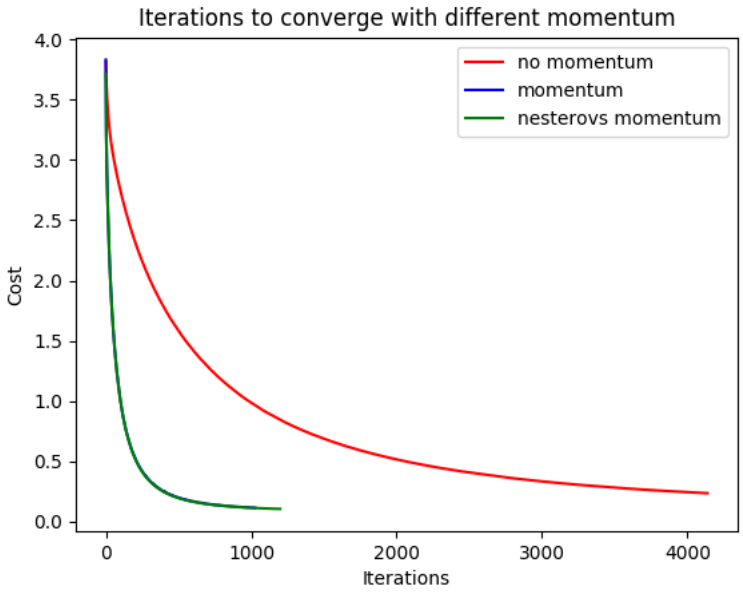
Figure 5: MLPC optimization



two lower learning rates' results (the ones on the up side) we notice that the accuracy was higher with the smaller learning rate (0.001 with alpha 0.01) but the f1 score was the highest and (perhaps more importantly)the loss the lowest with the learning rate at 0.003 and alpha 0.03. This final one is the model we selected as the best one.

Unlike the previous models, the MLPClassifier does provide a loss history. Figure 6 shows a plot of the loss convergence with and without momentum, using the parameters we found as best before. In it we can notice the clear and very meaningful impact of momentum. When using momentum and

Figure 6: MLPC convergence



nesterov's momentum we see an improvement of around 4 times.

Table 3: MPLC's Scores

Accuracy(%)	F1(%)	Recall(%)	Precision(%)
0.881	0.880	0.881	0.890

When training the model outside of the GridSearchCV, we reached the results in Table 3 with our test set.

The model was trained in 432 seconds (a little over 7 minutes) using nesterov's momentum.

3 Conclusions

Table 4: Models' Performances

	Accuracy(%)	F1(%)	Recall(%)	Precision(%)	TimeToTrain(min)
MLPC	0.881	0.880	0.881	0.890	7
SVC	0.774	0.781	0.774	0.805	1
LogReg	0.796	0.795	0.796	0.812	4

When looking at the results in Table 4 we can see that the MPLC got the best results out of all the classifiers, but at the cost of training time. Time-wise the SVC performed the best, but it had the worst scores. The Logistic Regression worked as a middle ground, getting scores a little above the scores of the SVC but still quite far from the MPLC's results and having a training time very close to the middle of the three.

Overall we can say that when looking for the best predictions possible the right model is the MPLC, out of the used models. But the SVC, given it's speed, can be a great way to get fast and good results and demonstrate the feasibility of a given project, giving ensurance that investing in more complex and time consuming models can be productive.

Acronyms

GTSRB German Traffic Sign Recognition Benchmark

SVM Support-Vector Machine

SGD Stochastic Gradient Descent

References

[1] URL: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html.

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

[3] URL: https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression.

[4] URL: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

[5] URL: https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC.

[6] URL: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html.

[7] URL: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html.

[8] URL: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics.

[9] URL: https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation.

[10] *GTSRB - German Traffic Sign Recognition Benchmark*. URL: https://www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign.

[11] https://www.kaggle.com/gauss256. *Image preprocessing exploration 2*. URL: https://www.kaggle.com/gauss256/image-preprocessing-exploration-2.

[12] Hsin-Yuan Huang and Chih-Jen Lin. “Linear and Kernel Classification: When to Use Which?” In: (), pp. 3, 9. URL: https://www.csie.ntu.edu.tw/~cjlin/papers/kernel-check/kcheck.pdf.

[13] Warren S. Sarle. In: (), Part 3. URL: http://www.faqs.org/faqs/ai-faq/neural-nets/part1/preamble.html.