VRIJE
UNIVERSITEIT
BRUSSEL

ANIMAL CLASSIFICATION AI

Machine Learning

Lennert Bontinck

January, 2020

Master Computer Science: AI

Sciences and Bio-Engineering Sciences

Abstract

This report documents the development of an *animal classification AI* using a more *old-school approach* of Visual-Bag-of-Words models. This AI is capable of differentiating 12 different animals. These models, and thus the AI, are developed in Python-based Jupyter Notebooks accompanied by this document. This animal classification AI was developed as a fulfilment of the Machine Learning course requirements and was used to compete in the organised Kaggle competition (Rosseau, 2020).

Part I of this report discusses the accompanied code in general. Section 1.1 explains which files are the most important. Section 1.2 describes the ideology used to created the code. To make testing multiple models easier, *a template* for model exploration was created and is discussed in section 1.3.

In part II, the data analysis part of this project is discussed. Section 2.2 talks about the unbalanced data distribution. In the next section, section 2.3, a deeper look is taken into the data and possible preprocessing is discussed. The last few sections of this part discuss how the feature extraction is dealt with and what the numerical representation looks like.

The linear baseline model is discussed in part III. This model is a fine-tuned Logistic Regression model from the SciKit Learn library. This model is often used to compare other models with. Only models that perform better then this baseline model should be considered. This part discusses the parameters used and the road to finding those optimal parameters.

TODO XXX

Finally, don't let the page count scare you. Due to clear separation in parts and large figures the page count of this document doesn't represent the length of it's actual content. Content that isn't crucial or is a repetition of supplied information is given as appendixes.

Contents

I About the code **1**

1.1 Files accompanied by this report 2

1.2 Ideology of the developed code 2

1.3 A typical model exploration 2

1.4 Technical remarks 2

II Data analysis **3**

2.1 About this part 4

2.2 Data distribution 4

2.3 Deeper look at the training data 5

2.4 Feature extraction 5

2.5 The numerical representation 6

III Linear baseline model **8**

3.1 About this part 9

3.2 Scoring used to evaluate the model 9

3.3 Fine-tuning the input 9

3.4 Fine-tuning the validation set 10

3.5 Fine-tuning the model parameters 10

3.6 The optimal settings for this model 12

More figures **13**

References **14**

Appendix A: Layout of the code **15**

Appendix B: LRM parameters **16**

Part I

About the code

1.1 Files accompanied by this report

Since this report discusses the development of an AI, *a lot of code* is discussed as well. This code is not shown inside this document but is available on the GitHub repository (Bontinck, 2020). All code is written in Python-based Jupyter Notebooks.

1.2 Ideology of the developed code

The Jupyter Notebooks have many inline comments and markdown blocks to make reading the code easier. If code is extensively discussed in this report, a reference to the corresponding section is made inside the code. Some of the gathered results come from time-consuming function calls. These can take *multiple hours* to complete. To spare some time, these results are saved in a Pickle file so they can be loaded in without having to do the function call. The Notebooks are written in a way that makes testing multiple models easy, paying extra attention to reusability.

1.3 A typical model exploration

The testing of a model consists of two main parts. Firstly the input of the model has to be optimized. Afterwards, the (hyper)parameters of the model itself can be optimized. These steps are clearly visible with the discussed models in this report since they follow a form of *template*. This template makes testing new models rather easy. After optimizing everything in a standalone fashion, it has to be checked that these newly optimized parameters do not influence previously optimized parameters. The resulting model should perform better than the linear baseline model in order to be considered. Whilst many abstractions were made and creating a *one-call pipeline* is possible, it's chosen to not do so. This is because *human reasoning* can be required in finding truly optimal parameters. It also makes understanding and discussing the model easier. After all, the goal of this report is to gather an understanding on how these models work, not solely to get the best Kaggle score.

1.4 Technical remarks

Most source files, for this report and the created models, are available on GitHub (Bontinck, 2020). Some files, like the used training images, were not included in this GitHub repository. Details about this can be found on the GitHub page (`README` file). Rights to this GitHub repository can be asked from the author. This report was created in LaTeX by modifying the VUB themed template from Ruben De Smet (2020).

Part II

Data analysis

2.1 About this part

Before rigorously testing different models available, it's important to take a look at the data that is supplied. The supplied data consists of two main groups of images, labelled training images and unlabeled test images. As per the requirements of the Kaggle competition, the test images should only be used for evaluating the model. Thus the test images *can not* be used for creating the model in any shape or form. This means that only the labelled training images can be used to create and validate the model in development. To avoid altering the model to perform well on the supplied test data and not in general, only the training data will be analysed. All code used for this part is available under the developed code folder on GitHub, in the Jupyter Notebook `data_analysis.ipynb`. This part will only analyze the data and make suggestions for possible preprocessing, it won't manipulate the data just yet. This notebooks allows for changing parameters easily, which is crucial since this analysis has to be done for multiple descriptors and settings.

2.2 Data distribution

The provided labeled training data consists of 12 different classes. There is a total of 4042 labelled training images supplied, the distribution of which is shown in figure 1. As visible in this figure, the distribution between classes is not balanced. This has to be taken into account when fitting a model since some models will show unwanted behaviour when fitted with unbalanced data. Luckily many solutions exist to minify the impact of this unbalance. This unbalance has to be kept in mind when using a split of the training set as a validation set as well. This is because such split might lead to a test set where some classes have considerably fewer instances in the test set and thus the performance on those classes has less impact on the total score, which may be unwanted.

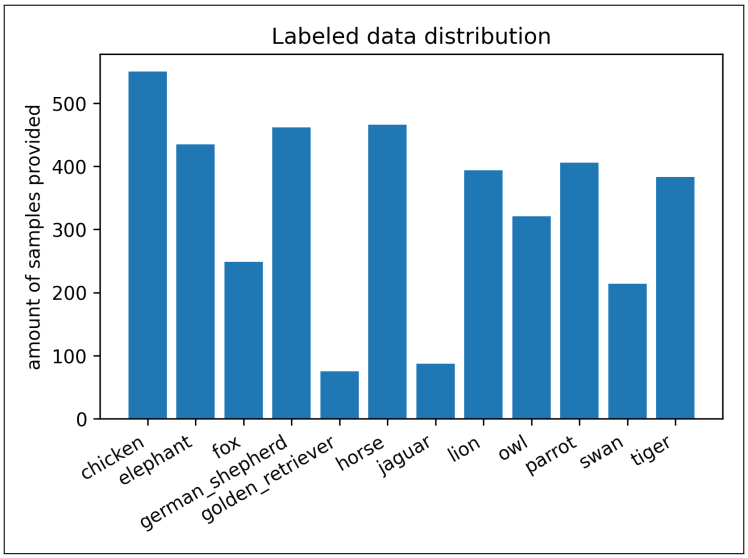


Figure 1: Data distribution training data.

2.3 Deeper look at the training data

Whilst noting that the available data isn't balanced over all the classes is very important, there are also different aspects of the data that need analysing. An overview of the supplied training data is given in figure 8, available in the figures list at the end of this report. This figure shows the first five images of each class. From this, it becomes apparent that multiple factors of the data aren't *optimal*. This knowledge is important since it can aid in better prepossessing and in finding a better model in general. The most noteworthy findings are listed here:

- Images vary in shapes, some are taken in portrait, others in landscape.
- Images vary in size, some are high resolution whilst others are relatively low resolution.
- The framing of the subject(s) varies a lot. Sometimes the labelled animal is completely visible and centred in the frame. In some images there are multiple animals spread across the image, others show a close-up of the animal.
- Some images have a detailed background that makes up for a lot of the image, in others the background is blurry and its impact is presumably less.
- Some images have very vibrant colours in broad daylight, others are black and white in dimly lit environments.

This diversity in the provided training set is expected since it has been scraped from the web. This also means that *noise* can be expected, another important factor to keep in mind when choosing and optimizing models. Many of the listed things can be minified by doing some clever prepossessing of the images.

2.4 Feature extraction

Some feature extraction has already been provided. In short, images are converted from there typical RGB representation to a numerical representation of interesting points, which can be used as input for our model. How this is done and could be optimized is briefly discussed here.

Instead of using the whole image as data, only a select few of *interesting points* of the image are taken into consideration. These interesting points of an image are found by using the *Shi-Tomasi corner detector*. As the name suggest, these interesting points are *strong corners on an image*. The following important parameters for the `features.extractShiTomasiCorners` function call are:

- Maximum number of interesting points = 500
- Minimum Euclidean distance between interesting points = 20

Shown in figure 2 is an example output of interesting points found by the Shi-Tomasi corner detector. It's clear that it's performance varies a lot, but finding interesting points isn't an easy task and thus the results are better then they might seem on first sight. This method might benefit from fine-tuning and earlier preprocessing of the data.

(a) First chicken from train data.

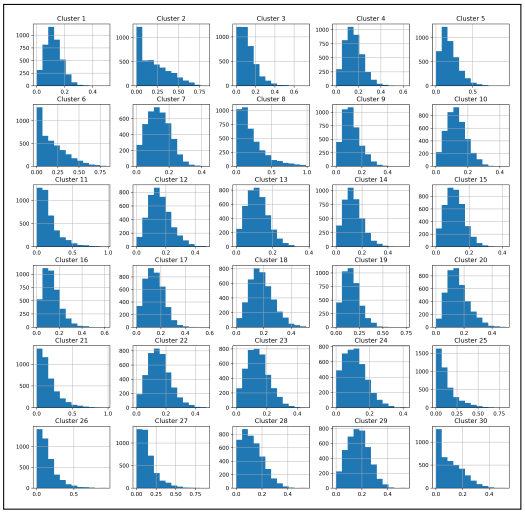
(b) First German shepherd from train data

Figure 2: Points of interest found by the Shi-Tomasi corner detector.

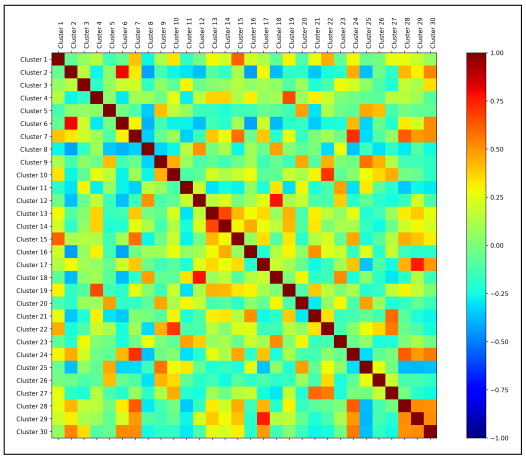
2.5 The numerical representation

Finding interesting points is only half of the work. These interesting points now need to be represented by numerical values that have actual meaning. Remember from section 2.3 that the provided images differ a lot. Thus the numerical representation, generated by a descriptor, has to be so that it minifies the impact of different lighting, scaling... afterwards, these values can be clustered together. Clustering is done by the `createCodebook` function which uses Mini-Batch K-Means clustering from the SciKit Learn library. Opting for a different clustering algorithm and/or fine-tuning its (hyper)parameters might result in better performance of the chosen model.

An overview showing histograms for each word (cluster) and a corresponding correlation matrix is shown in figure 3. From the histogram, it is visible that the values are normalized as was discussed. This would have to be checked for all descriptors used and perhaps some outliers would need to be removed. The correlation between these clusters doesn't seem too dramatic when opting for 30 clusters. A low correlation between clusters is mostly positive for model building since it means each cluster represent a distinct concept. This is again something that would have to be checked for different parameters.



(a) Histogram for each cluster.



(b) Correlation matrix for all clusters.

Figure 3: Data analysis of encoded images using 30 clusters.

Part III

Linear baseline model

3.1 About this part

A good linear model available in the SciKit Learn library is the Logistic Regression model. This model is often used as a linear baseline model to compare other models with. Only models scoring better then this linear baseline model should be considered. This part discusses the parameters found to be optimal for this model in this setting and the road to finding these optimal parameters. The Python-based Jupyter Notebook corresponding with this part is `linear_baseline_model.ipynb`. This notebook will form a *template* for future model exploration.

3.2 Scoring used to evaluate the model

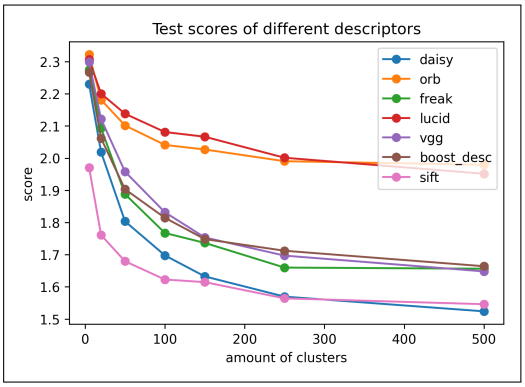
The multi-class Log Loss score of a validation set taken from the training set is used to evaluate models. This scoring strategy is the same as used in the Kaggle competition. An important note to make is that the unbalance, as discussed in section 2.2, might make this score overly dependent on classes which have many instances. This can and will be taken into account for some settings.

3.3 Fine-tuning the input

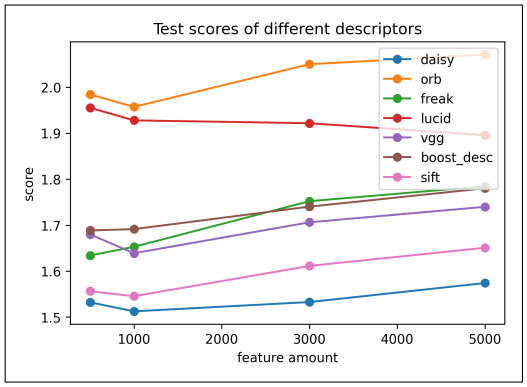
The first step in finding optimal settings for the model is finding optimal settings for the input of the model. In this case, the parameters that can control the input are the number of clusters each image is separated into and the descriptor used as discussed in part II. In general, more clusters often correspond to a better score, however, including many clusters can lead to overfitting of the model. The following values were tried to manually find an optimum:

- Descriptors: DAISY, ORB, FREAK, LUCID, VGG, BoostDesc, SIFT.
- Cluster amounts (small): 5, 20, 50, 100, 150, 250 and 500.
- Cluster amounts (small): 500, 1000, 3000 and 5000.

A comparison was done by averaging the multi-class Log Loss score over 5 trials for each of these cluster amounts and descriptors. Only the test score results are of significance. There are 2 ways of looking at this data. The descriptor that achieves the minimum with a certain cluster amount can be seen as the optimal setting. However, since clustering is used, one can consider the *elbow method* to determine the optimal setting as well. Figure 4 shows the found results for all used descriptors and cluster amounts. When taking into consideration the elbow method, the *SIFT* descriptor performs best with a cluster amount around 100. When looking at the minimum, it seems that the *DAISY* descriptor would come out on top with a small margin. The SIFT approach seems more viable since the difference in score is minimal and the SIFT approach uses less clusters suggesting it is more general. In contrast, the Daisy approach has signs of overfitting.



(a) Scores for small cluster amounts.



(b) Scores for large cluster amounts.

Figure 4: Average multi-class Log Loss score over 5 trials for different descriptors and cluster amounts.

3.4 Fine-tuning the validation set

Since the training data is further split into a training and validation set, this splitting can also be fine-tuned. The parameter that can be fine-tuned is *test_size* and whether or not to take into account that the data set is *unbalanced*. The latter is quite obvious as discussed in section 2.2 and thus the unbalance should be kept in mind. Ideally, there would be enough instances in the training set to make a specific enough model and there would be enough models in the validation set to get a representative score. The following values test sizes were tested for the otherwise optimal settings: 5%, 7.5%, 10%, 12.5%, 15%, 20%, 25%, 30%. Figure 5 shows the result of this experiment using average multi-class Log Loss score over 5 trials. A healthy balance seems to be around 15%.

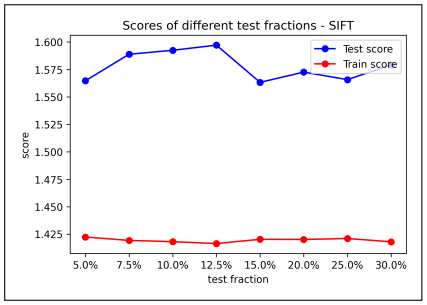


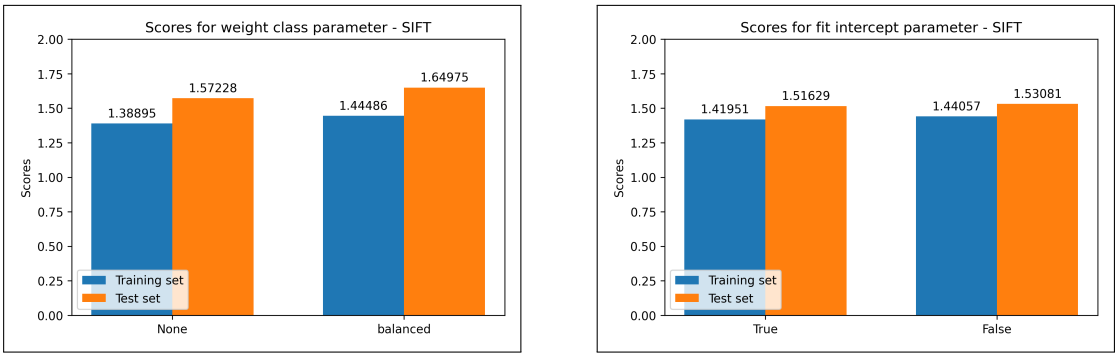
Figure 5: Experimenting with different test sizes.

3.5 Fine-tuning the model parameters

Now that all of the parameters available for the input are fine-tuned, the parameters of the model itself can be optimized. As found in the documentation of the `LogisticRegression` function available in the SciKit Learn library there are multiple (optional) parameters (Pedregosa et al., 2011). The most interesting ones are given in appendix B.

Before experimenting, it was assumed that changing the class weight parameter to balanced would enhance the performance due to the unbalance of the training data. Weirdly, this wasn't the case for the score received from the test set split from the training data nor on the Kaggle page. Due to the unbalance the worse score for the split test set can be expected. However, the fact

that the score is worse on the Kaggle competition, 1.60289 vs 1.67565, is not expected. Perhaps changing this parameter has more impact than was first assumed. Setting the fit intercept parameter to false has a negative impact, albeit minor. These experiments are visualised in figure 6.



(a) Scores for different class_weight parameters.

(b) Scores for different fit_intercept parameters.

Figure 6: Average multi-class Log Loss score over 10 trials for different model settings.

The default value for the maximum allowed iterations is 100. With the current settings, convergence is not always reached after 100 iterations. The following values were tried: 50, 100, 150, 200 and 250. Since convergence is reached after 250 times in all test cases, this value is used for maximum iterations.

Finally the *hyperparameter* C has to be optimized. This was done by using `GridSearchCV` from the Sci Kit Learn library. This performs an exhaustive search over specified parameter values for the model. A similar result should be reached by performing the more manual methods used for previous parameters. In this case the following potential C values were tried: 0.00001, 0.0001, 0.001, 0.01 , 0.1, 0.5, 1.0, 1.5, 3, 5, 10, 100, 1000, 10000 According to Grid Search, 3 is the best value for C, which happens to be close to the default of 1. If doing the same experiment with the manual method used for the other parameters, the same can be concluded. This manual method is visualised in figure 7. This also shows the manual method is most likely just as good and offers greater insight into the working.

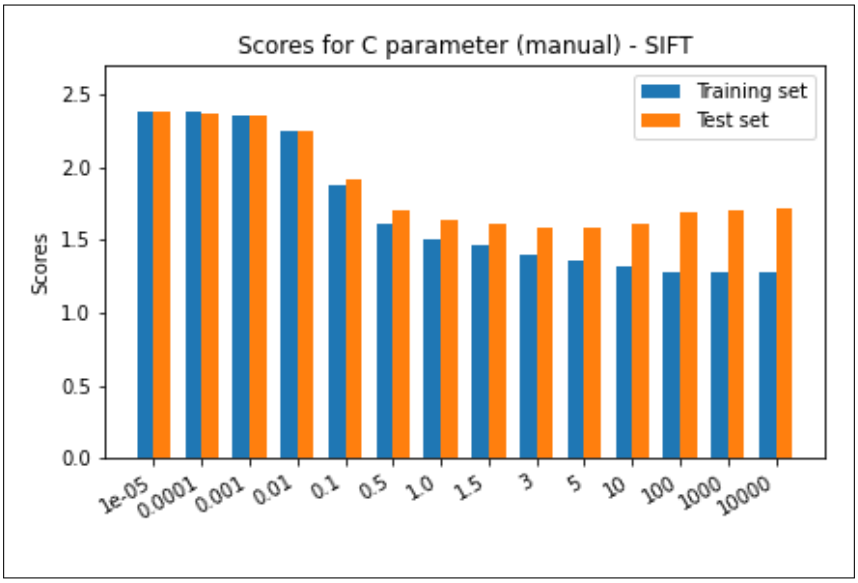


Figure 7: Experimenting with different C values.

3.6 The optimal settings for this model

After all the fine-tuning discussed in the previous sections, an optimal model can be formed. The optimal settings and received score for the SIFT descriptor are:

- Descriptor used: SIFT
- Cluster amounts: 100
- Sample size: 15%
- Class weight: None
- C: 3
- Max it-er: 250
- Fit intercept: false
- Score received from validation set: $\pm$ 1.55
- Score received on Kaggle: 1.60289

More figures

Some figures are referred to in the text but not placed directly under the text. These are included in this list. All figures are high resolution thus zooming in the PDF should be viable to get a clearer view.

Overview of training set

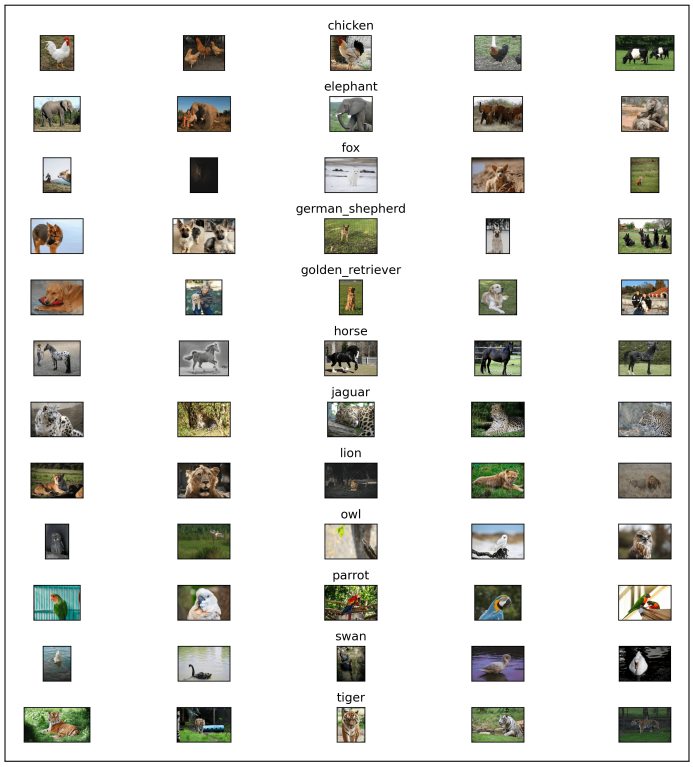


Figure 8: An overview of the supplied data per class.

References

Bontinck, L. (2020). *Machine learning project* [GitHub commit: TODO]. Retrieved December 11, 2020, from https://github.com/VUB-CGT/ml-project-2020-pikawika

De Smet, R. (2020). *Vub latex huisstijl* [GitHub commit: d91f55799abd390a7dac92492f894b9b5fea2f47]. Retrieved November 2, 2020, from https://gitlab.com/rubdos/texlive-vub

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Rosseau, A. (2020). *Vub: Animal classification*. Retrieved November 15, 2020, from https://www.kaggle.com/c/vub-animal-classification-20/

Appendix A: Layout of the code

TODO XXX

Appendix B: LRM parameters

As found in the documentation of the `LogisticRegression` function available in the SciKit Learn library there are multiple (optional) parameters (Pedregosa et al., 2011). The most interesting ones are:

- *solver*
  - Specifies which solver should be used for the optimization problem in the model.
  - *lbfgs* is used as default and whilst a little slow, this parameter doesn't require further fine-tuning.
- *penalty*
  - Since the *lbfgs* solver is used, the default *l2* penalization norm is the only one that can be used.
- *class_weight*
  - This parameter defaults to None but can be set to balanced to take into account the unbalance in our data, as discussed in section 2.2.
  - The results with this parameter set to balanced will be studied.
- *C*
  - The regularisation hyperparameter C defaults to 1. Fine-tuning this could boost performance.
- *max_iter*
  - This parameter can be changed so that convergence might be found, which is not the case right now.
- *fit_intercept*
  - Boolean that specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.
  - The results with this parameter set to true and false should be checked.