VRIJE
UNIVERSITEIT
BRUSSEL

ANIMAL CLASSIFICATION AI

Machine Learning

Lennert Bontinck

December 11, 2020

Master Computer Science: AI
Sciences and Bio-Engineering Sciences

Abstract

This intermediate report documents the development of an animal classification AI using a more "old-school" approach of Visual-Bag-of-Words models. This AI should be 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 2 sections of this part, 2.4 and 2.5, 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 getting those optimal parameters.

Finally, part IV discusses future plans for this project. Section 4.1 lists possible topics that can be explored to create a better animal classification AI. In the last section, section 4.2, some open issues are discussed.

Contents

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 interception can be required in finding truly optimal parameters. It also makes understanding and discussing the model easier.

1.4 Technical remarks

This report was created in LaTeX by modifying the excellent and well-known VUB themed template from Ruben De Smet (2020). BibLaTeX was used for reference management and natbib was used for more citation control.

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.

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 2 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 on the Kaggle page by submitting a CSV of the prediction results. Thus the test images can't 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 describes how data can be analysed with the given code. This code can be easily changed to analyse other variations of the data, e.g. using another descriptor.

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.

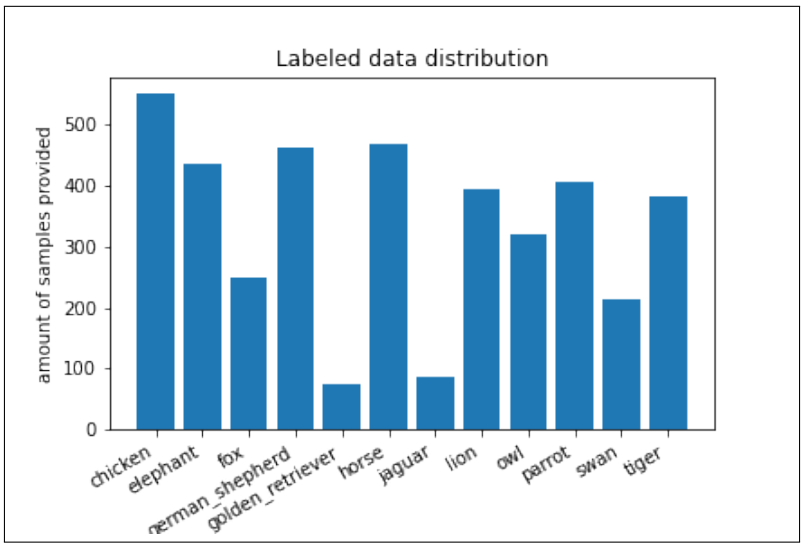


Figure 1: The data distribution of the supplied training set.

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

Since the focus of this competition is on developing great models and not necessarily on data prepossessing and feature extraction, some feature extraction has already been provided. More info on the prepossessing and feature extraction provided is available in the provided notebook *creating_vbow.ipynb*. 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 will briefly be 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*. The following important parameters for the *features.extractShiTomasiCorners* function call are:

- number of interesting points = 500
- minimum 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 this is far from optimal, 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 perform better after fine-tuning.

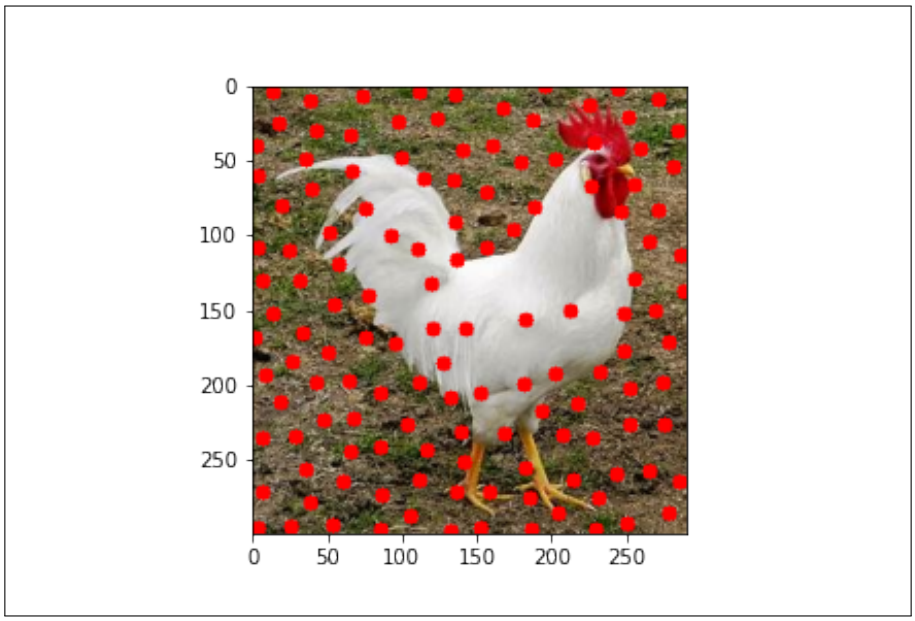


Figure 2: Example of points of interest found by Shi-Tomasi corner detector.

Finding interesting points is only half of the work. These interesting points now need to be represented by numerical values that have actual meaning, afterwards, they're clustered using a helper function. 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... The following descriptors are used and their outputs are provided: DAISY, ORB, FREAK, LUCID, VGG, BoostDesc, SIFT. Whilst SURF is another great descriptor, it's not provided nor is the license available for this project. SIFT is often referred to as the most famous and successful of these descriptors, but all of them should be explored. Clustering is done by the *createCodebook* function which uses Mini-Batch K-Means clustering from the SciKit Learn library. This is also something that might be fine-tuned.

2.5 The numerical representation

As discussed in section 2.4, the images are stored as numerical representations using descriptors. To save time, these numerical representations for all the descriptors are stored in a separate *Pickle* file. Since these representations form the input of a model, it's important to get a grip on how these look. The provided *createCodebook* function is used for clustering this data, which was also discussed in the previous section. This function allows specifying how many clusters

should be created for clustering the interesting points. These different clusters can be thought of as different *features*. The function returns 2 lists. The first contains the labels for the image represented at a certain index. The other contains information about each image at that index. This information is the output of the clustering done for that image and thus corresponds to an array which size equals the requested cluster amount.

An overview of the data from such clusters/features given by the SIFT descriptor is given in figure 8, available in the figures list at the end of this report. From this, it is visible that the values seem to be normalized. This would have to be checked for all descriptors used and perhaps some outliers would need to be removed.

In figure 3 the correlation matrix is shown for 30 clusters of the SIFT descriptor. The correlation between these values doesn't seem too dramatic, which is mostly positive for our model building. This is again something that would have to be checked for different parameters.

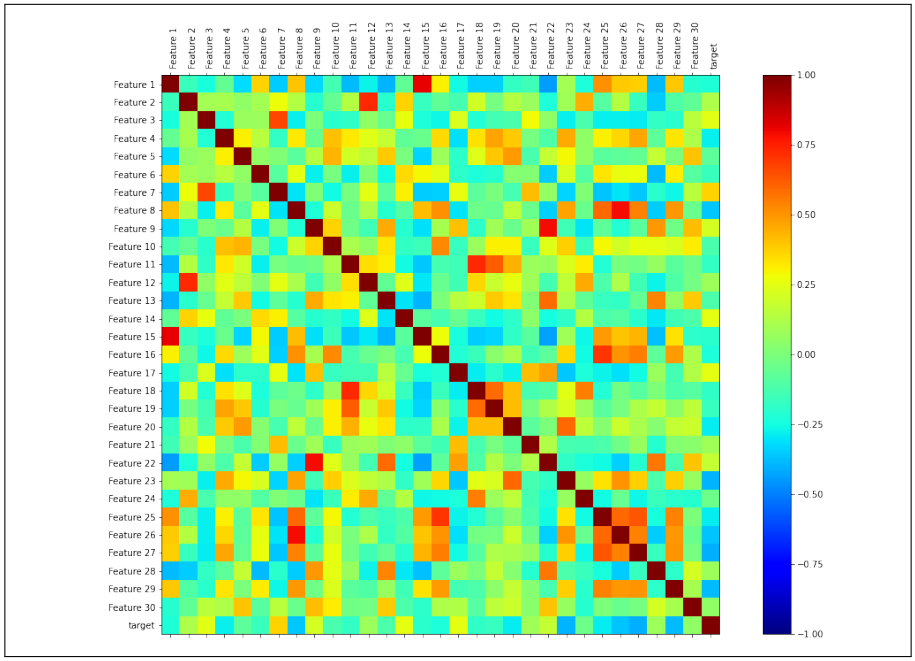


Figure 3: Correlation matrix of 30 clusters made from the SIFT descriptor.

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 Layout of the code

The code is split into different steps. Each step has some explanation in a comment or refers to a section in this report. This makes the code easy to read. At the end of the Jupyter Notebook, optimisation of the model is performed. This optimisation includes all previous steps, but these are now taken care of all in once. This is a bit more complicated, hence why the code is split into steps at the beginning of the notebook. The optimal parameters found are used in the steps in the beginning. This means the resulting model from the code split into steps is the optimal model.

3.3 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 a model. 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 is an open issue discussed in section 4.2.

3.4 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 features each image has and the descriptor used as discussed in part II. SIFT is often referred to as the most famous and successful of the descriptor, but all of them should be explored. In general, more features often correspond to a better score, however, including many features can lead to overfitting of the model. The following values were tried initially:

- Descriptors: DAISY, ORB, FREAK, LUCID, VGG, BoostDesc, SIFT.
- Feature amounts: 5, 20, 50, 100, 150, 250 and 500.

A comparison was done by averaging the multi-class Log Loss score over 5 iterations for each of these feature amounts and descriptors. Multiple iterations are needed since these methods

make use of random values. A lower score means a better performing model. All of the resulting scores plotted per descriptor (*small* amounts) can be found in the figures list at the end of this report. 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 feature 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. In figure 4 both of these optimum are plotted. When taking into consideration the elbow method, the *SIFT* descriptor performs best with a feature 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 is more general and needs a lot less computation work for training. The Daisy approach has signs of overfitting. The difference of score between unseen test data and training data is considerably larger than was the case with the optimum of SIFT.

To validate that the DAISY descriptor would reach the absolute minimum, an additional loop was made to test larger feature amounts: 500, 750, 1000, 1250 and 1500. All of the resulting scores plotted per descriptor (*large* amounts) can be found in the figures list at the end of this report as well. In figure 5, both of these optimum are plotted once again. It becomes apparent that the *DAISY* descriptor does indeed have the absolute minimum of all descriptors, albeit with only a small lead. It also becomes apparent that overfitting is indeed possible by increasing the feature amount! The optimum was found to be 1500 as can be seen in figure 10 available in the figures list. Since the difference with the SIFT approach is negligible and this approach shows signs of overfitting, it isn't explored further.

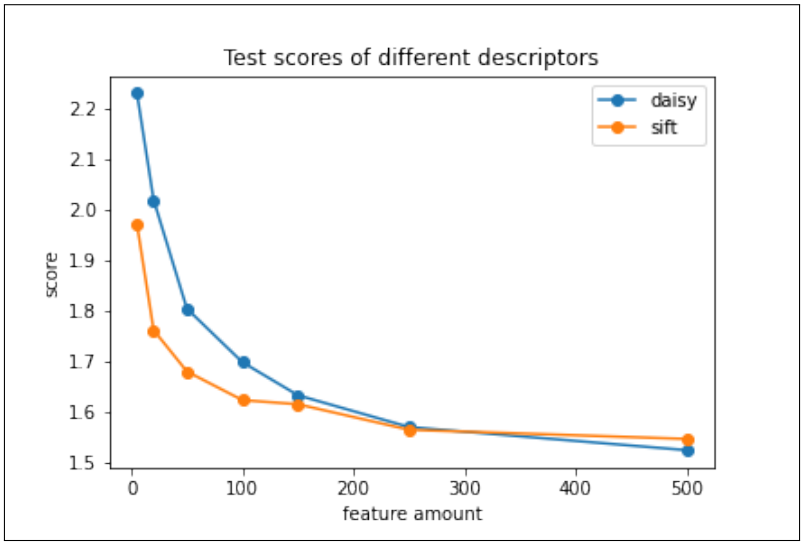


Figure 4: Scores in multi-class Log Loss of the Logistic Regression model using different (small) amounts of features and descriptors.

Figure 5: Scores in multi-class Log Loss of the Logistic Regression model using different (large) amounts of features and descriptors.

3.5 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. Later progress might even change this approach altogether, as discussed in section 4.1. For now, 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. The splitting should thus be done with the unbalance in mind since using pure random splitting could lead to validation or training sets without instances of specific classes. When making the training set to large, overfitting becomes more likely. When making the training set to small, the model might not be fit enough. When making the validation set to small, the scores might not be representative enough. A healthy balance has to be found. 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 were used for testing:

- Descriptors: SIFT
- Feature amounts: 100
- Test sizes: 5%, 10%, 15%, 20%, 25%, 30%, 40%, 50%.

Figure 6 shows the result of this experiment. It is visible that with a test size of higher then 30%, the model seems to perform worse since it doesn't have enough training data. When taking a test size that is too small, it seems that the results aren't representative. A healthy balance seems to be around 15%.

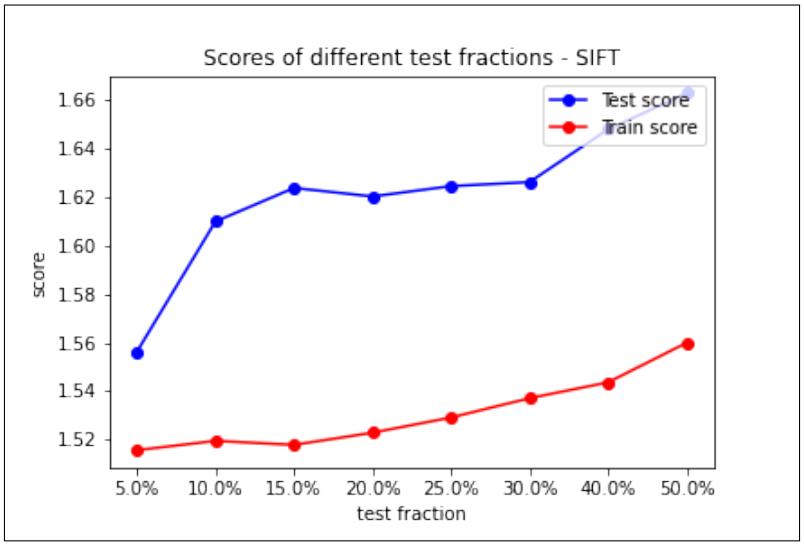


Figure 6: Scores in multi-class Log Loss of the SIFT driven Logistic Regression model using different test sizes.

3.6 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:

- *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.

Before experimenting, it was assumed that changing the class weight parameter to balanced would enhance the performance. This was assumed because the training data is very unbalanced whilst *normal* input for model predictions wouldn't show this unbalance. Neither using the generated validation sets shown in figure 11, available in the figures list, nor on the Kaggle competition page, a better score was achieved. The fact that the performance isn't better for the validation sets isn't unexpected. This is due to the unbalance of the training set also being apparent in the validation set since this is a subset. However, the fact that the score is worse on the Kaggle competition, a 0.06 difference, 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. This is visualised in figure 12 available in the figures list. This parameter will be kept on the default, being true.

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.

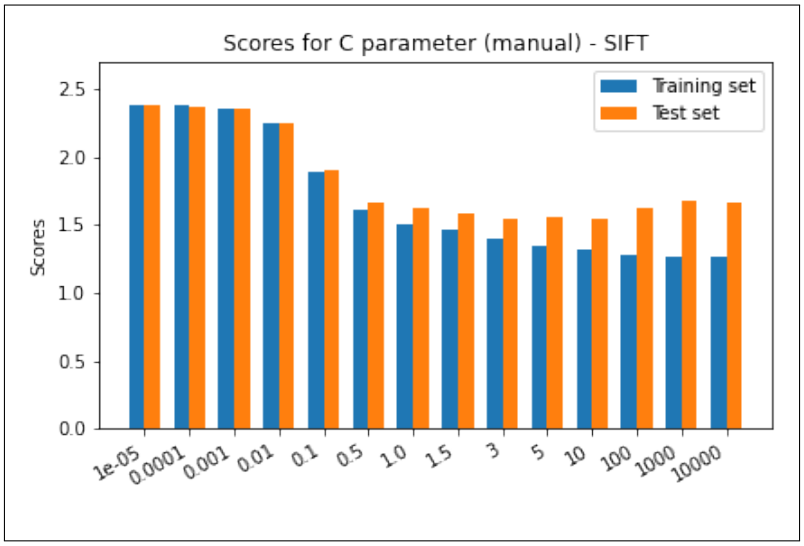


Figure 7: Scores in multi-class Log Loss of the SIFT driven Logistic Regression model using different test sizes.

3.7 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
- Feature amounts: 100
- Sample size: 15%
- Class weight: None
- C: 3
- Max it-er: 250
- Fit intercept: false
- Score received from validation set: 1.48756
- Score received on Kaggle: 1.60289

Part IV

What's next

4.1 Further development

Due to limited time available, this intermediate report and the current state of the project isn't overwhelming in any stretch of the imagination. Because of this, a special thanks is given to the teacher and teaching assistants who've softened the requirements for the intermediate report.

In the time available until the final deadline many new possibilities for creating a better model will be explored, this might include but is not limited to:

- Preprocessing can be explored.
- Optimisation of the point of interest detection can be done.
- Testing other descriptors could be done.
- A better alternative to splitting the data into train and validation sets can be explored, e.g. StratifiedKFold.
- Testing different cluster algorithm can be explored.
- More available models need to be tested and compared to the linear baseline model.
- Nesting multiple good performing models might be an option, using weighted probabilities.
- Making a full pipeline for everything done in the linear baseline model and using grid search for all parameters. Results should be similar but this would be a more elegant way of doing things and it could form a faster template for testing new models.

Recommendation on which things to explore first is asked from the teaching assistants. Some aspects of the created models and pipeline might also be modified further once the open issues discussed in section 4.2 are resolved.

4.2 Open issues

While developing the first models many questions arose. Most are already answered by googling or discussion with colleagues. Open questions that are not yet answered are listed below, for which guidance from the teaching assistants is asked.

- The data is not available as a pickle file for the SURF descriptor. Can this be made available?
- The code written for the linear baseline model hasn't been validated, however, since it will be used as a "template", it shouldn't have to many major errors.
- Whilst experiments show that setting the class_weight parameter to "balanced" results in worse scores on the Kaggle page, this is not what was expected. The reason why this isn't so is also unclear.

- Does the unbalance of the training data cause issues when using a split for validation? E.g. bad classification on classes with few instances doesn't have a huge impact on the score while they will have a huge impact on the eventual Kaggle score. Is taking a fixed number of instances from each class a solution? Does the score have to be normalized over the class size?

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.

Overview of features data

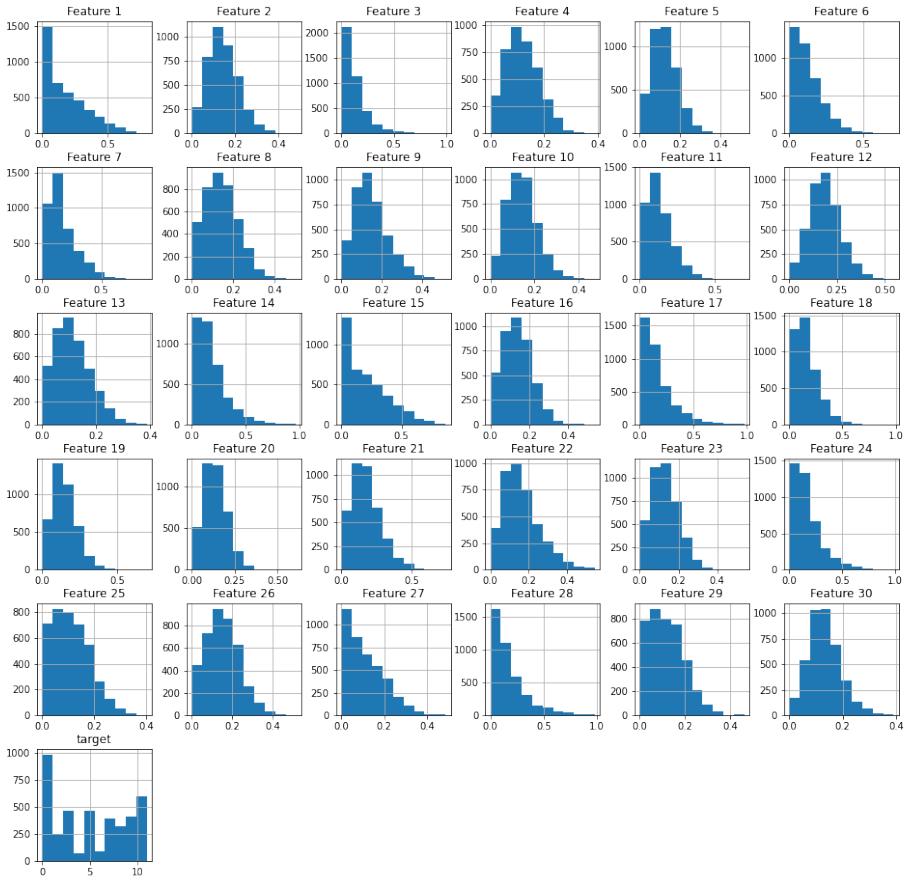
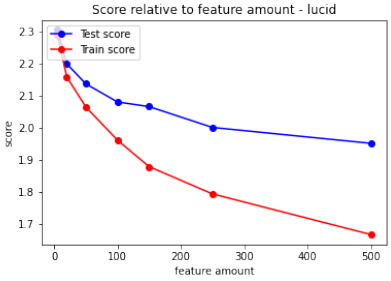
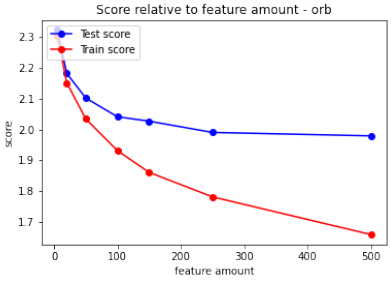
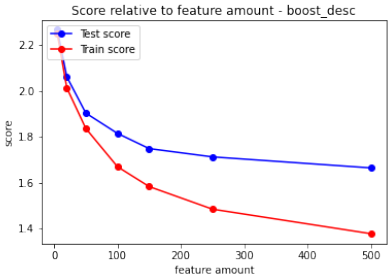


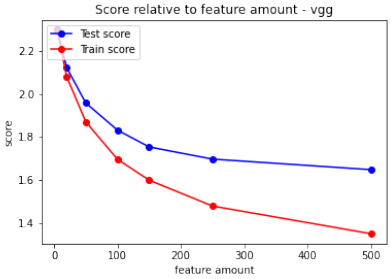
Figure 9: An overview of the first 30 features data from a SIFT descriptor.

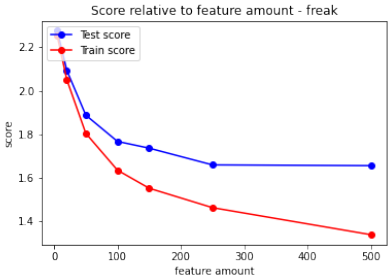
Linear baseline model - input optimisation (small)

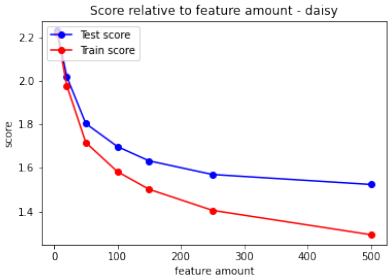


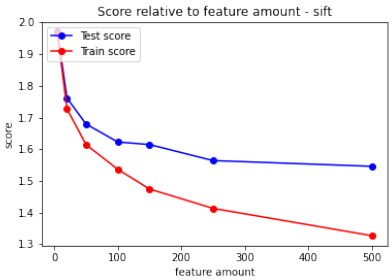




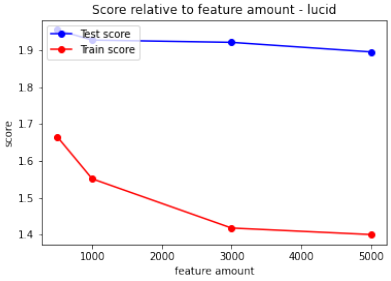


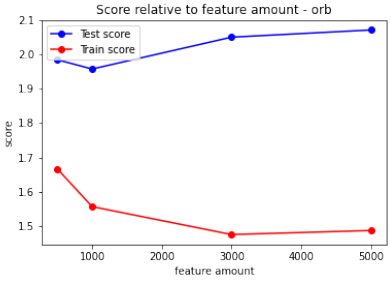


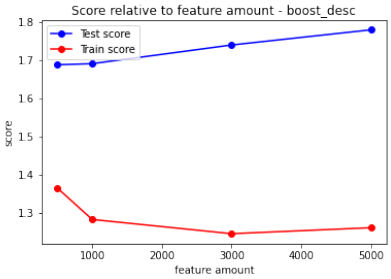


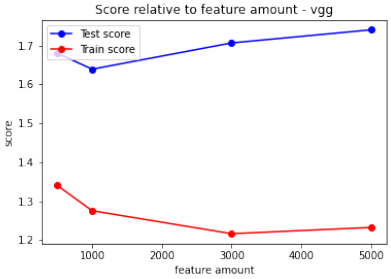


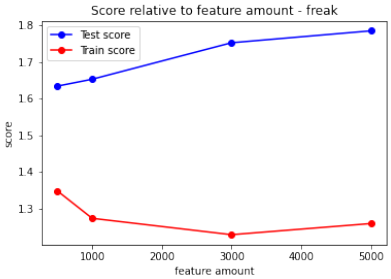
Linear baseline model - input optimisation (large)

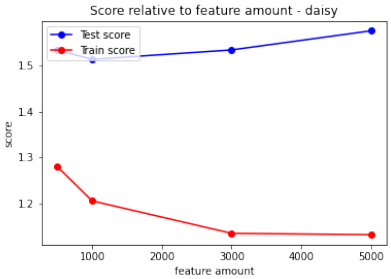


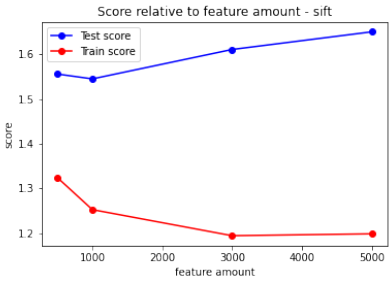












Linear baseline model - optimal DAISY

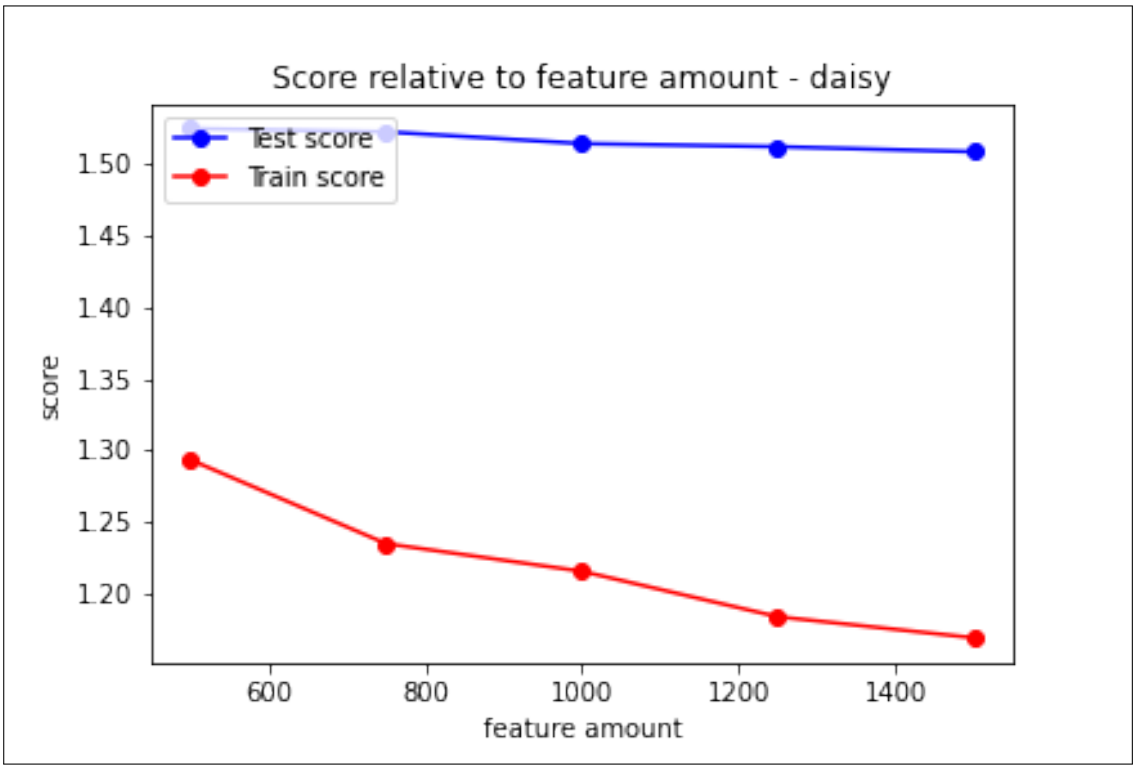


Figure 10: An extra test for finding the optimal feature amounts for the DAISY descriptor.

Linear baseline model - class weight parameter

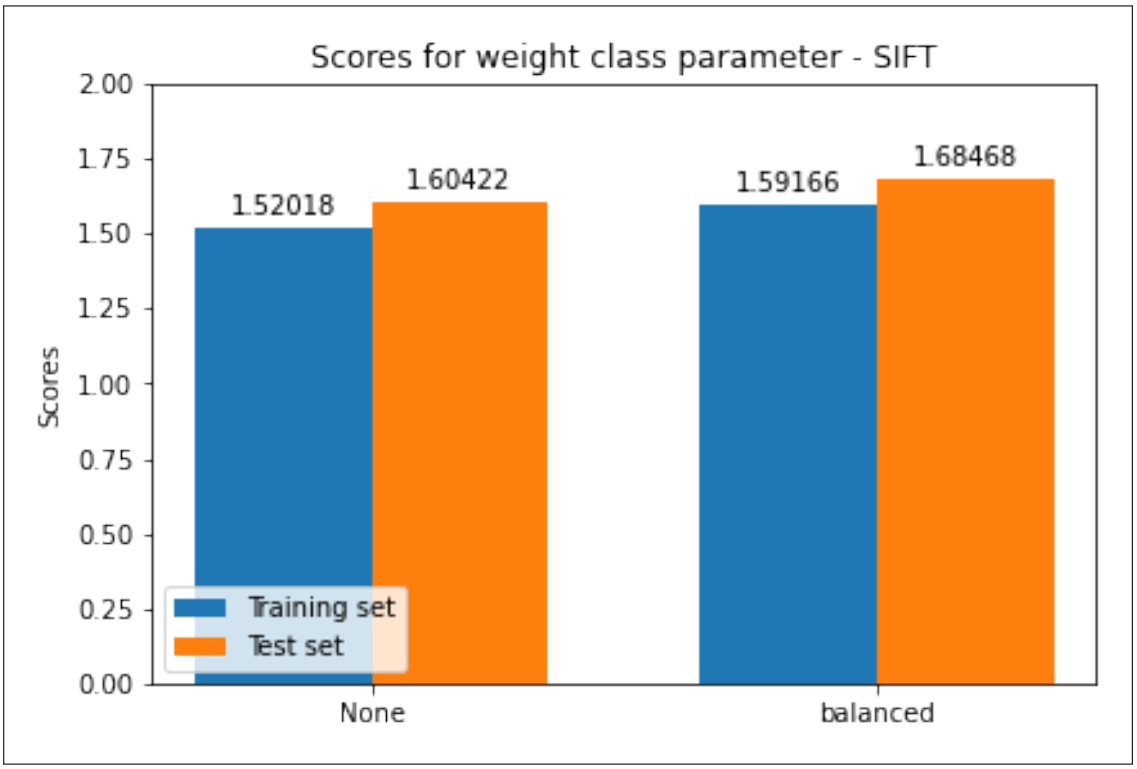


Figure 11: Average multi-class Log Loss score over 10 iterations.

Linear baseline model - fit intercept parameter

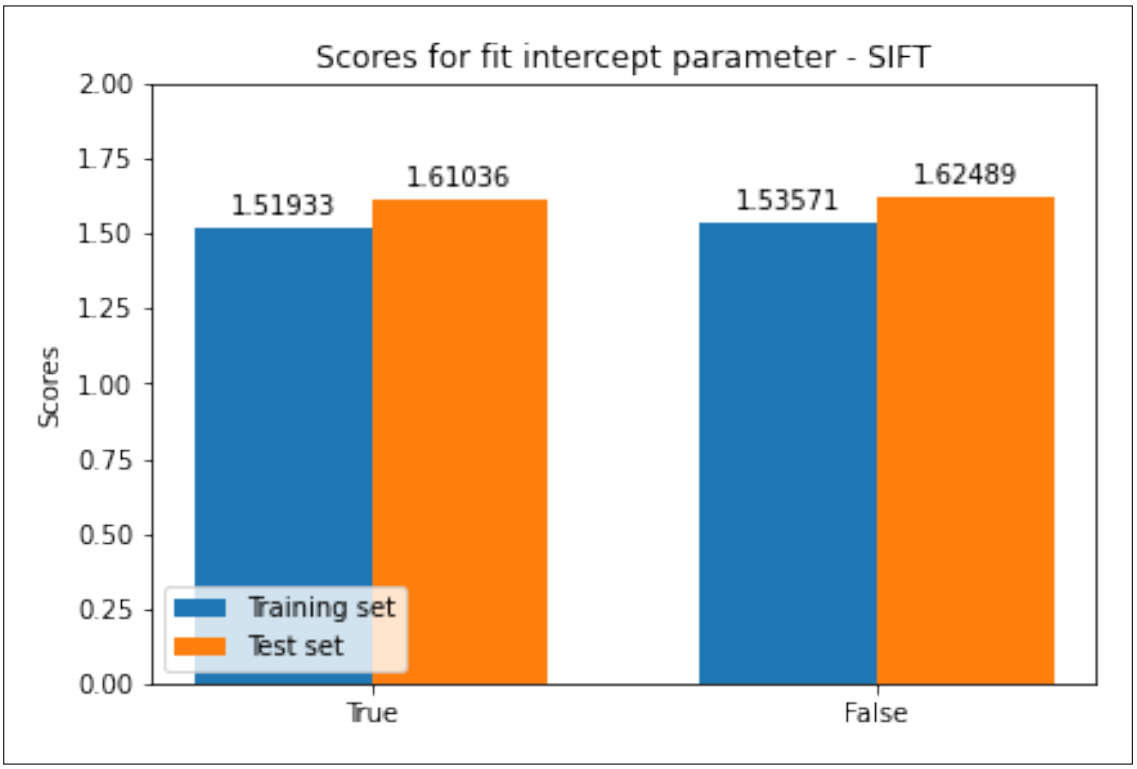


Figure 12: Average multi-class Log Loss score over 5 iterations.

References

Bontinck, L. (2020). *Machine learning project* [GitHub commit: 7220044a6afaa1d327f7964d5b93320422942194]. 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/