VRIJE
UNIVERSITEIT
BRUSSEL

ANIMAL CLASSIFICATION AI

Machine Learning

Lennert Bontinck

January, 2021

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.

Afterwards *Support vector Classifiers* (SVC) are explored. Part IV discusses non-linear SVC models with different kernels and finds the *rbf kernel* to be the best from three tested kernels. Part V focuses on linear SVC models in a similar fashion and finds them to perform worse. An ensemble approach is discussed in part VI. This approach makes use of Gradient Boosting which, while interesting, didn't perform well.

It is chosen to do the model analysis in a separate part, part VII. Afterwards, part VIII goes over some other possible optimisations considering what is learned from all experiments thus far. Some of these are implemented whilst others are just discussed in a theoretical manner. This results in the final model. Afterwards, in part IX, a short conclusion is given.

It is noted that the page count of this document doesn't represent its actual length due to clear part separation from the used template and large figures. Content that isn't crucial or is a repetition of supplied information is discarded or given as an appendix.

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 and all models make use of the scikit-learn library.

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 for reuse later. 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 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 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 of 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. Data leakage is also an important factor that will have to be taken into account when creating a further train and validation set from the training data. 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 notebook 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 data 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.

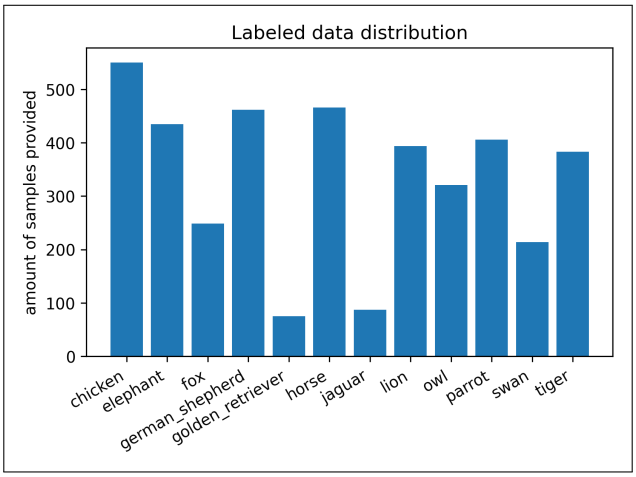


Figure 1: Data distribution training data.

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 13, 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 or are already taken into account by the used descriptor.

2.4 Feature extraction and numerical representation

Due to exclusive use of the supplied SIFT descriptor features for this report, information for this section is given as an appendix (appendix A). The most important finding is the fact that the supplied `createCodebook` helper function uses `MiniBatchKMeans` for a given cluster amount, which can be fine-tuned. More clusters will often result in better performance but gives the risk of overfitting. It's also said the descriptor might benefit from fine-tuning as well, but this will not be done in this report, as will be discussed in part VIII.

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 and will lay some defaults for the report.

3.2 Scoring used to evaluate the model

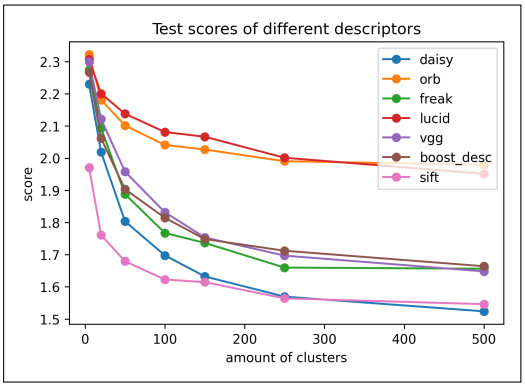
The multi-class Log Loss (abbreviated MCLL) score of a validation set taken from the training set is used to evaluate the model. This validation set kept the unbalance in mind. Stratified K-Folds cross-validation (abbreviated SCV) is also performed using MCLL for the final model (5 folds). The Stratified variant of K-Folds will again aid in keeping the unbalance in mind. This scoring strategy is the same as used in the Kaggle competition and will be the default in this report.

3.3 Fine-tuning the input

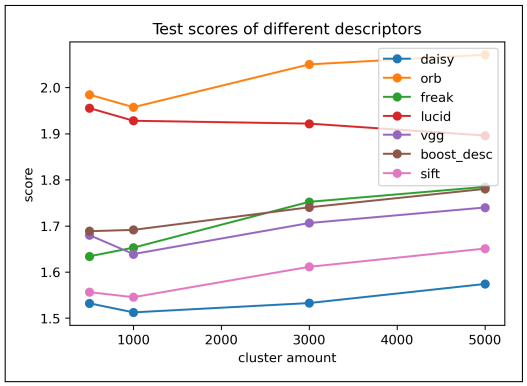
The first step in finding optimal settings for the model is finding optimal settings for the input of the model. In this case, the parameters that can control the input are the number of clusters each image is separated into and the descriptor used as discussed in Appendix A. In general, more clusters often correspond to a better score, however, including many clusters can lead to overfitting. 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 (large): 500, 1000, 3000 and 5000.

A comparison was done by averaging the MCLL 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 2 shows the found results for all used descriptors and cluster amounts. When taking into consideration the elbow method, the *SIFT* descriptor is preferred with a cluster amount around 100. When looking at the minimum, it seems that the *DAISY* descriptor would come out on top. The SIFT approach seems more viable since the difference in score is not that significant and the SIFT approach uses fewer clusters suggesting it is more general. With a lower cluster amount, the correlation between clusters will also be less as discussed in appendix A, which is favourable. The SIFT descriptor with 100 clusters will be used as default for this report, but will be reconsidered in part VIII.



(a) Scores for small cluster amounts.



(b) Scores for large cluster amounts.

Figure 2: Average MCLL score over 5 trials for different descriptors and cluster amounts. Lower score is better.

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. This validation set will often be referred to as test set in this report and accompanied code, since the actual test set is only used for Kaggle submissions. 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 test sizes were tested for the otherwise optimal settings: 5%, 7.5%, 10%, 12.5%, 15%, 20%, 25%, 30%. Figure 3 shows the result of this experiment using average MCLL score over 5 trials. A healthy balance seems to be around 15%. This will again be used as default in this report. A perfectly balanced test set will be considered in part VIII.

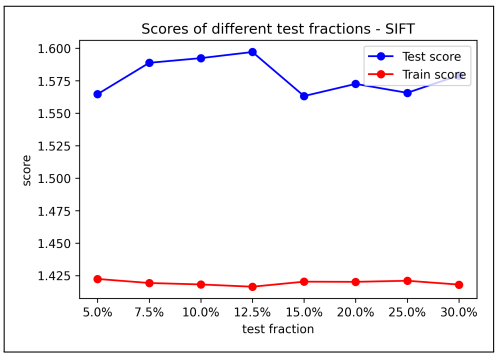
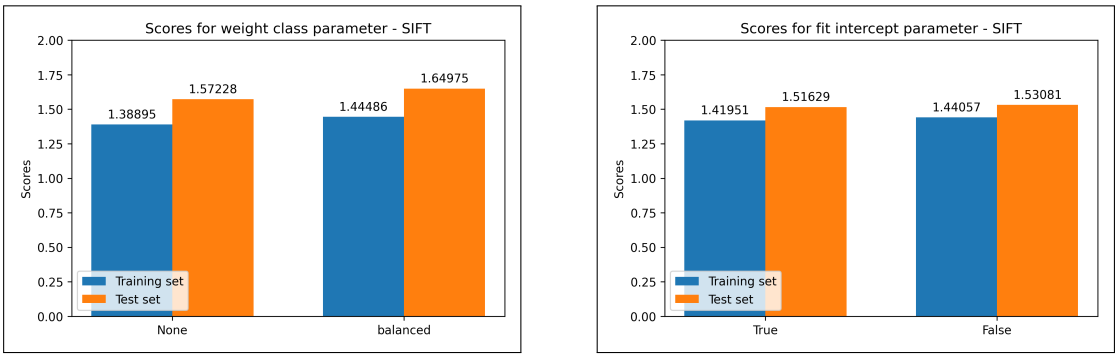


Figure 3: Testing different test sizes. Lower score is better.

3.5 Fine-tuning the model parameters

Now that some of the important 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 evaluation set 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. The model analysis in part VII will reveal these scores don't tell everything. Thus, the balanced variant is still committed and considered since by human reasoning it should be better. Setting the fit intercept parameter to false has a negative impact, albeit minor. These experiments are visualised in figure 4.



(a) Scores for different class_weight.

(b) Scores for different fit_intercept.

Figure 4: Average MCLL score over 10 trials for different model settings. Lower score is better.

The default value for the maximum allowed iterations is 100. After experimenting it was found that changing this to 250 resulted in convergence for most scenario's. Finally the *hyperparameter* C has to be optimized. This was done by using `GridSearchCV` from the Sci Kit Learn library using Stratified K-Fold (5 folds) to keep unbalance in mind. This performs an exhaustive search over specified parameter values for the model. 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 and 10000. According to Grid Search, values between 3 and 5 are optimal for C (keeping standard deviation in mind). This happens to be close to the default of 1, but an improvement is made: SCV MCLL score of 1.57 vs 1.63! A similar result should be reached by performing the more manual methods used for previous parameters, which is indeed the case. This manual method is visualised in figure 5. Since both approaches yield similar results, finding optimal parameters where no human reasoning is needed will be done using `GridSearchCV` in the future.

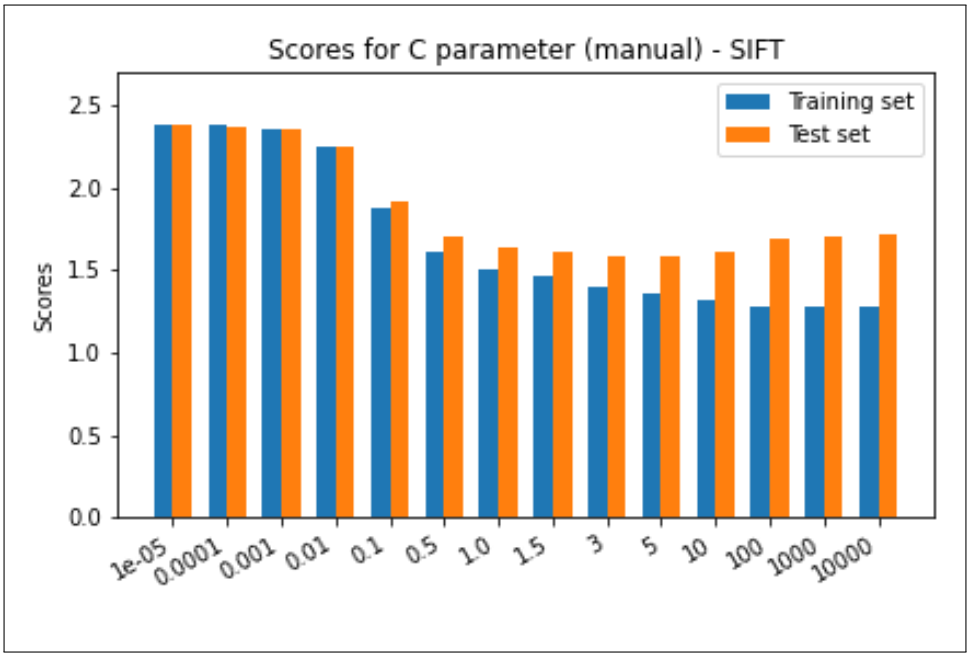


Figure 5: Trying different C values. Lower score is better.

3.6 The final settings for this model

After all the fine-tuning discussed in the previous sections, a final model is formed. This model doesn't use any preprocessing, which will be experimented with in part VIII. The optimal settings and received score for the SIFT descriptor are given below.

- Descriptor used: SIFT with 100 clusters.
- Sample size for validation set: 15%.
- Class weight: None (but balanced also considered).
- C and fit intercept: 3 and False.
- MCLL score for validation set: ± 1.61 (non-balanced) and ± 1.66 (balanced).
- SCV MCLL score (5 K-fold): ± 1.55 (non-balanced) and ± 1.63 (balanced).
- Score received on Kaggle: 1.60289 (non-balanced) and 1.67565 (balanced).

Part IV

Support Vector Classifier

4.1 About this part

Now that a fine-tuned linear baseline model is established, the search for better models can start. Sci Kit learn has a flow chart for deciding which estimator to use, available here. When following this chart a Support Vector Classifier (SVC) is proposed, thus one is examined in this report. Since part III already exhaustively discusses the model testing strategy used, this report will go into less detail from now on to avoid repetition. The Notebook corresponding with this part is `support_vector_classifier.ipynb`.

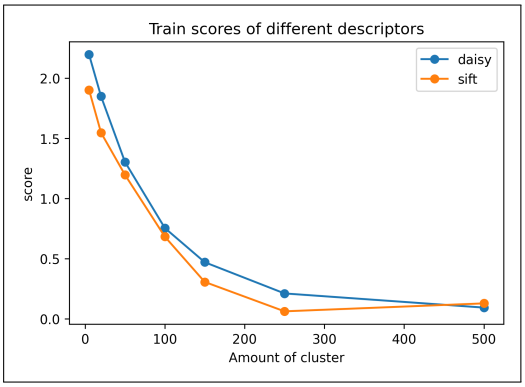
4.2 Scoring and methodology for fine-tuning

Again, the MCLL score is used to compare models. `GridSearchCV` was used to find optimal parameters were no human reasoning is needed. In this part, SIFT with 100 clusters was used to find optima parameters, this is reconsidered in part VIII.

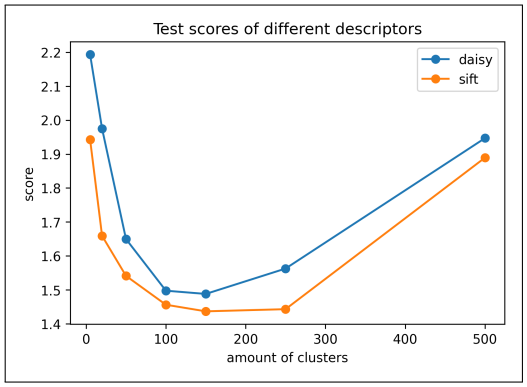
Balancing the class weight had a small positive impact, which is great! Making the toll more precise had a negligible impact (-0.00) on the results thus it is left default. `GridSearchCV` was used to find the optima for C and gamma for the *rbf*, *sigmoid* and *poly* kernel. For the poly kernel, it was also used to find an optimal degree. It was insured `GridSearchCV` uses Stratified K-Folds cross-validation (SCV) with MCLL scoring to keep the unbalance of classes in mind. What follows are tested parameters and the found optimal for each of these experiments:

- Rbf kernel
  - Tested C: 0.001, 0.01 , 0.1, 0.5, 1.0, 1.5, 3, 5, 10 and 100.
  - Tested gamma: "scale", "auto", 0.001, 0.01 , 0.1, 0.5, 1.0, 1.5, 3, 5, 10 and 100.
  - Found optima: C = 1.5 — gamma = 0.01, auto or scale — SCV MCLL score of ± 1.46.
- Sigmoid kernel
  - Tested C: 0.001, 0.01 , 0.1, 0.5, 1.0, 1.5, 3, 5, 10 and 100.
  - Tested gamma: "scale", "auto", 0.001, 0.01 , 0.1, 0.5, 1.0, 1.5, 3, 5, 10 and 100.
  - Found optima: C = 5 or 10 — gamma = 0.001 — SCV MCLL score of ± 1.54.
- Poly kernel:
  - Tested C: 0.1, 0.5, 1.0, 1.5, 3 and 5.
  - Tested gamma: "scale", "auto", 0.001, 0.01 , 0.1, 0.5, 1.0 and 1.5.
  - Tested degree: 2, 3, 4, 5, 7 and 10.
  - Found optima: C between 1 and 5 — degree = 3 — gamma = scale, auto or 0.01 — SCV MCLL score of ± 1.59.

Figure 6 shows the results of experimenting with different clusters and descriptors for the found optima. Over-fitting is visible when using over 250 clusters. This conclusion only holds for the current settings, which uses a fixed gamma. This will again be reconsidered in part VIII.



(a) Train scores.



(b) Test scores.

Figure 6: Averaged MCLL scores for optimal SVC model using different descriptors and clusters amounts over 3 trials. Lower score is better.

A histogram showing the SCV MCLL score for each optimal kernel configuration together with the standard deviation is given in figure 7. It is visible that the rbf kernel performed the best. Thus, it was chosen to optimize this kernel further by doing another grid search that included more values around the found optima C and gamma. The results from this further fine-tuning were negligible, C = 1.75 and gamma = 0.01 is used for the final model.

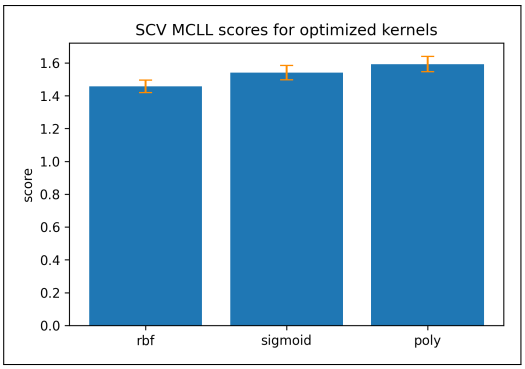


Figure 7: SCV MCLL score per kernel. Lower score is better.

4.3 The final settings for this model

The final settings and received score for the described model are:

- Descriptor used: SIFT with 100 clusters
- Sample size for validation set: 15%
- Class weight: balanced
- C: 1.75 — gamma: 0.01 — kernel: rbf — tol: 0.001
- MCLL score for validation set: ± 1.45 (balanced and unbalanced)
- SCV MCLL score: ± 1.48 (balanced and unbalanced)
- Score received on Kaggle: 1.55355 (non-balanced), 1.53626 (balanced)

Part V

Linear SVC

5.1 About this part

In the previous part, three different kernels for SVC were explored. Another kernel available for SVC is the *linear kernel*. However, there's also a separate model, `LinearSVC`, available as well. The performance of the linear kernel and this separate model are explored in this part. The Notebook corresponding with this part is `linear_SVC.ipynb`.

5.2 Scoring and methodology for fine-tuning

The algorithms are tested and evaluated as the non-linear SVC from part IV. In order for MCLL scoring to work, a `predict_proba` function should be available. For `LinearSVC` this is not the case, it can, however, be implemented by wrapping it with Sci-Kit's `CalibratedClassifierCV`. Arguably, this wrapping can influence the performance, however, the Kaggle competition also requires a `predict_proba` function thus the wrapping is necessary and its performance impact should be taken into account. Both balanced and non-balanced variant were considered, the received scores were comparable and the balanced variant is preferred and discussed. A short discussion of the tried parameters and found results using `GridSearchCV` is listed here:

- `SVC` with linear kernel:
  - Tested C: 0.001, 0.01 , 0.1, 0.5, 1.0, 1.5, 3, 5, 10 and 100.
  - Tested gamma: "scale", "auto", 0.001, 0.01 , 0.1, 0.5, 1.0, 1.5, 3, 5, 10 and 100.
  - Found optima: C: 0.01 — gamma had no influence, 0.001 is chosen — SCV MCLL score of ± 1.56.
- `LinearSVC`:
  - Tested C: 0.001, 0.01 , 0.1, 0.5, 1.0, 1.5, 3, 5, 10 and 100.
  - Tested penalty and loss: l1 and l2 — hinge and squared hinge.
  - Tested dual: True and False.
  - Found optima: C: 0.1 — penalty: l1 — dual: False — loss: squared hinge — SCV MCLL score of ± 1.63.

5.3 Linear is not the way to go

The SVC model with linear kernel performed better than the LinearSVC model. Further fine-tuning around the found optima didn't gain any performance. It is noted that LinearSVC often *fitted quicker*, which perhaps explains it's existence. Since the linear SVC approach performed worse then the optimal SVC found in part IV it was not discussed in great detail and is not explored further. Only SIFT with 100 clusters was tested. During testing, the received MCLL score for the validation set was ± 1.60 and the SCV MCLL score was ± 1.55. The received Kaggle score for the optimal linear SVC model is 1.59520.

Part VI

Gradient Boosting

6.1 About this part

All previous models where standalone classifiers, to make things more interesting an ensemble is tried next. An ensemble consists of cleverly combining multiple *weaker* classifiers to form a final *strong* classifier. Gradient Boosting and random forest are common ensembles. A quick experiment showed that default Gradient Boosting performed better than an optimised random forest (MCLL of 1.68 vs 1.74). Thus, it is chosen to only discuss Gradient Boosting further. The Notebook corresponding with this part is `gradient_boosting.ipynb`.

6.2 Scoring and methodology for fine-tuning

Fine-tuning and evaluation of the model is done in an equal manner as the previous models. This model doesn't have a parameter like class weight which specifies to work in a balanced manner. Due to the many parameters that need fine-tuning and limited computing power, the use of `GridSearchCV` was split over 2 independent searches. Found optima were retested to validate they don't influence previous determined parameters. The tried parameters using `GridSearchCV` are listed here:

- Experiment 1:
  - Tested loss: deviance and exponential.
  - Tested learning rates: 0.001, 0.01 , 0.1 and 0.5.
  - Tested amount of estimators: 50, 100, 250, 500, 600, 750 and 1000.
- Experiment 2:
  - Tested subsample: 0.5, 0.75, 1.0 and 1.5.
  - Tested minimum sample split: 2, 3 and 5.
  - Tested max depth: 3, 5, 7 and 10.

6.3 Interesting but not impressive

Whilst the Gradient Boosting approach is interesting, it didn't score well and there are signs of overfitting with an MCLL score of $\pm$ 0.21 on the training data using some settings. The latter could perhaps be resolved but since the model performed worse than the linear baseline model throughout testing, it was not explored further. Only SIFT with 100 clusters was considered. A MCLL score of $\pm$ 1.68 on the validation set, a SCV MCLL score of $\pm$ 1.62 and a Kaggle score of 1.70494 were achieved with the following parameters:

- Loss and learning rate: deviance and 0.01.
- Amount of estimators and subsample: 750 and 0.75.
- Minimum sample split and max depth: 5.

Part VII

Model analysis

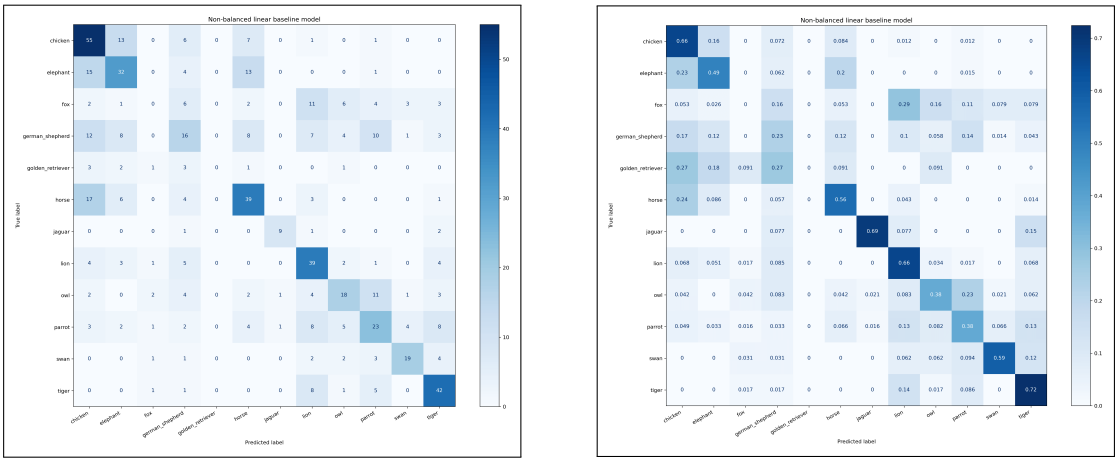
7.1 About this part

In the previous parts, many different models were developed and the most interesting ones were discussed. This part will analyse these models by briefly discussing the confusion matrices. The learning curves don't tell much but they are included in the extra figures list for completeness. The Notebook corresponding with this part is `model_analysis.ipynb`.

7.2 Non-balanced Linear baseline model

The confusion matrices given in figure 8 teaches us multiple things, the most important of which are listed here:

- Even though the jaguar class had very few examples, it classifies considerably well.
- The classifier completely fails in classifying the fox class even though the unbalance of the class wasn't so drastic. It also completely fails in classifying golden retriever, which was heavily unbalanced.
- An owl is more mistaken for a parrot then vice versa.
- The model prefers classes which have a higher frequency when uncertain.



(a) Non normalised CM.

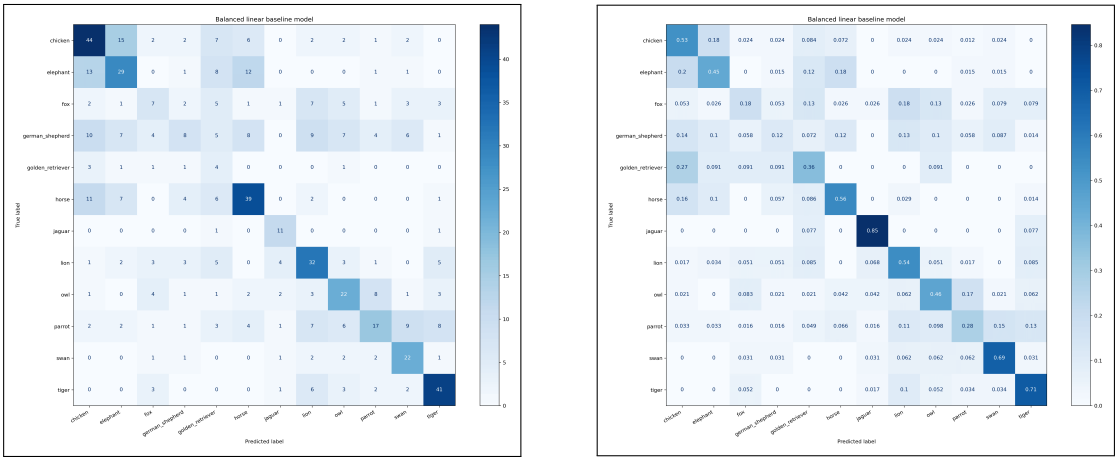
(b) Normalised CM.

Figure 8: Confusion matrices for the non-balanced linear baseline model.

7.3 Balanced linear baseline model

It was said in section 3.5 that it was weird the balanced LBM model received a worse MCLL score. The confusion matrices given in figure 9 shows a different story and teaches us multiple things, the most important of which are listed here:

- The model now has some succession for each class, all be it low. This behaviour is more preferred and whilst it had a worse MCLL score it's confusion matrix shows it had a better understanding of the problem. The worse MCLL score is most likely due to the unbalance in the test set and when using CSV, the worse Kaggle score suggest the test data for the public leaderboard is also unbalanced.
- The correct classification of most classes has risen. These few that have dropped are classes with a high frequency suggesting it's better labelling was based on frequency bias instead of conceptual understanding for the non-balanced model.



(a) Non normalised CM.

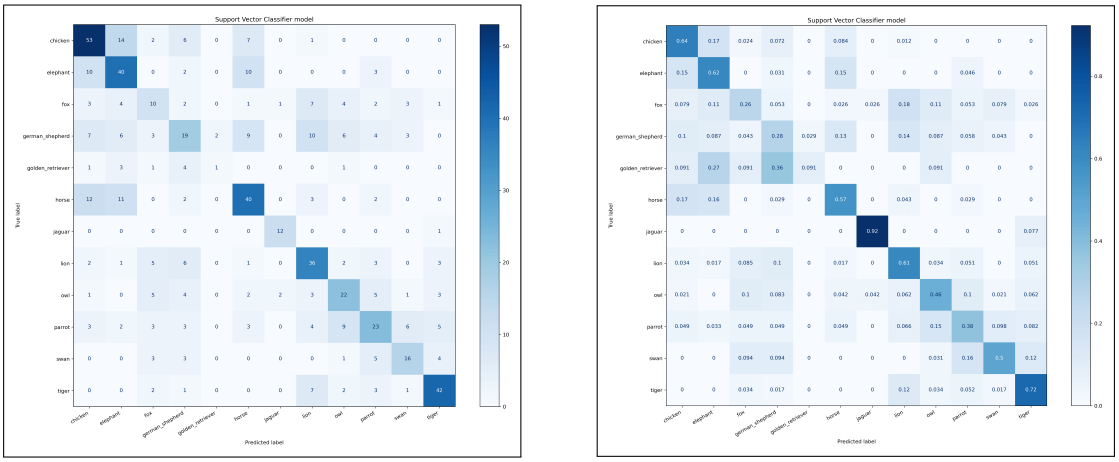
(b) Normalised CM.

Figure 9: Confusion matrices for the balanced linear baseline model.

7.4 Support Vector Classifier model

The Support Vector Classifier model was the best performing model. This is shown in figure 10 where the correct classification is considerably better than with the LBM. There are some important notes to be made:

- The model is considerably worse at identifying golden retrievers and swans compared to the balanced LBM.
- The model outperforms the non-balanced LBM for almost all classes but classifying dogs and swans remains a difficult task.
- When wrongly classifying, the made error is comparable to the one for the balanced LBM, e.g. a fox is often misinterpreted as a lion. This suggests there's a correlation between these classes.



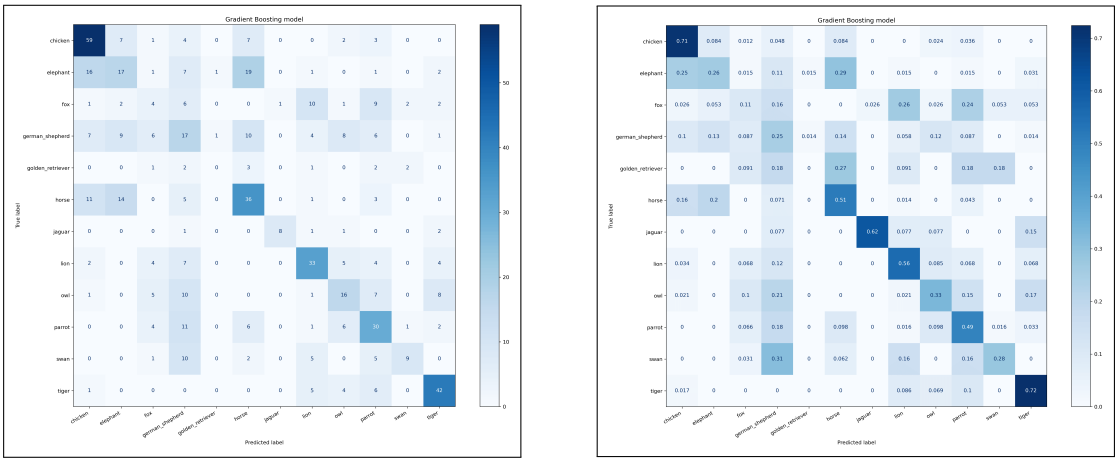
(a) Non normalised CM. (b) Normalised CM.

Figure 10: Confusion matrices for the Support Vector Classifier model.

7.5 Linear SVC and Gradient Boosting

As discussed in part V, the linear SVC variant performed worse than the just discussed SVC variant. The confusion matrices reflect this by just performing worse overall with no noteworthy other differences. For completeness the matrices are given in the extra's figures list (figure 18).

As discussed in part VI, the Gradient Boosting model didn't perform that well. This is also reflected in the confusion matrices, given in figure 11, with worse overall scores than both the SVC model and balanced LBM model. One important thing to note is that the gradient boosting algorithm performed well on parrots compared to the other models, with an accuracy of 50%.



(a) Non normalised CM. (b) Normalised CM.

Figure 11: Confusion matrices for the Gradient Boosting model.

Part VIII

The final model

8.1 About this part

Whilst a great insight on the working of different models and a typical AI pipeline has been achieved with the experiments discussed so far, many things were left unexplored. This is mostly due to limited computational power and time available. A final attempt is made to make a better performing model using the further insight received from experiments up until now. Things that were not tried but which are *assumed* to increase performance are also listed here. This part will only briefly discuss all steps to avoid repetition and an even longer document. The Notebooks corresponding with this part are `final_model_exploration.ipynb`, `final_model.ipynb`, `svc_large.ipynb`, `clustering.ipynb` and `more_helpers.py`. The Notebooks provide inline comments on the found results.

8.2 Making a better individual model

Firstly, some of the made decisions early on, such as the used descriptor and cluster amount, are reconsidered. The following things were tried to make a better performing LBM and SVC model:

- Different scalers and transformers were tried as preprocessing (after clustering). The SVC model did not benefit from this but applying the polynomial features transformer on the LBM model significantly increased the performance by 0.1 on the test set and Kaggle (now 1.57203)!
- The train set was balanced and overfitted, neither of which increased global or individual performance. The balanced train set does give a more representative MCLL score and thus will be used to determine the weighted probabilities for the final model.
- Neither `AdaBoostClassifier` or the `BaggingClassifier` to make an ensemble of the LBM and SCV models performed well. This can be expected since such ensembles work well when combining *weak models*, which SVC and the LBM are not.
- The SIFT descriptor with 100 clusters was used as the default for this report, this is now reconsidered by testing SVC with more clusters. It was found that giving SVC more clusters drastically improved its performance without seeming to lose generality based on the confusion matrix. 1250 clusters seemed optimal.
- The `createCodebook` helper function uses `MiniBatchKMeans` for a given cluster amount, according to Alonso (2013), using this faster variant over regular K-Means results in worse performance. This was explored and it was found opting for regular K-Means does indeed increase model performance, all be it with the cost of a tremendous slower clustering. Some different clustering algorithms, such as `GaussianMixture` were also tried, but this didn't yield an increase in performance.
- Preprocessing after and before clustering was also considered. One of which is using *Root-SIFT* as proposed by Vijayan and KP (2020). Opting to use root-sift combined with `PowerTransformer` yielded in an even bigger increase of model performance. Special attention needs to be paid that the scaler and clustering used for the test data are the same as the one used for the training.

8.3 Combining powers

Multiple models are now finalised, each having some strengths and weaknesses. Some classes are easier to distinguish than others. The ensembles discussed in part VI suggest *cleverly combining* multiple models and *submodels* can increase performance, especially for the unbalanced classes.

As a final approach, a model is made which uses multiple models and submodels under the hood. This final model is represented by the custom class `FinalModel` which has a `fit`, `predict` and a `predict_proba` function. Since this isn't a scikit classifier, another way of generating confusion matrices is considered by editing open-source code by Cipriano (2020). The weight given to certain classifiers is determined from their confusion matrix. All of the confusion matrices for the underlying models are given in the extra figures list. The general flow of the final model is as follows:

- When fitting the model, multiple underlying models are fitted. These models are:
  - The Linear Baseline Model (LBM) and Support Vector Classifier (SVC) for all of the input data.
  - The LBM and SVC model to distinguish between 5 merged classes based on similarity from mislabelling:
    - Chicken.
    - Big: horse and elephant.
    - Catish: fox, lion, tiger and jaguar.
    - Dog: German shepherd and golden retriever.
    - Flying: owl, parrot and swan.
  - The LBM and SVC to distinguish between *dogs or others*.
  - The LBM and SVC sub-model for classifying a subset:
    - Big: horse and elephant.
    - Catish: fox, lion, tiger and jaguar.
    - Dog: German shepherd and golden retriever.
    - Flying: owl, parrot and swan.
- When predicting proabilities, the following steps occur:
  - The `predict_proba` function for all underlying models is called and the results are saved.
  - The final probabilities to return is determined as follows:

* If the SVC or LBM model's `predict_proba` is very certain, 96%+ and 98%+ respectively, it's probabilities are returned.
* If SVC or LBM model for distinguishing the merged classes:
  - Is very certain (92% to 95%+), then
  - The scores received from the related sub-models are added to the scores for all animals, then
  - The resulting probabilities are renormalized, thus
  - The animals from the merged class have *boosted* probabilities (by a factor of 4)
* If SVC or LBM model for distinguishing the merged classes:
  - Is somewhat certain (82% to 86%+), then
  - The scores received from the related sub-models are added to the scores for all animals, then
  - The resulting probabilities are renormalized, thus
  - The animals from the merged class have *boosted* probabilities (by a factor of 0.5)
* If all of the above didn't yield a result, the classifier (SVC or LBM) that has the highest certainty after weighing with accuracy, is chosen.

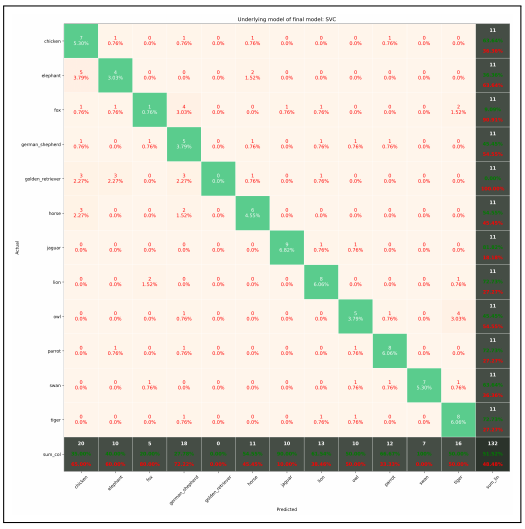
8.4 Unexplored possibilities

Whilst the achieved model performance is above average when looking at the Kaggle leader board, some interesting possible optimisations that came to mind could improve it even more. To show that these were thought of and as possible further extensions some unexplored possibilities are listed here:

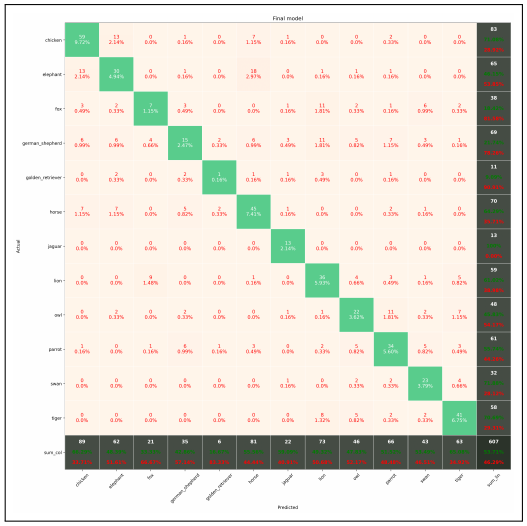
- Fine-tuning and preprocessing the SIFT descriptor, although trivial actions such as resizing are expected to have minimal impact since SIFT determines interesting points in a manner which is not directly dependent on orientation or size.
- Perhaps making a selection of the found features and clusters might further aid performance. For example: Using the correlation matrix to remove disruptive features or clusters.
- Using a different descriptor such as HoG (Histogram of Gradients) which is said to work well with humans and animals.
- Trying more models such as XGBoost, a modification on the tried Gradient Boosting model said to work well with animals.
- Only the LBM model was explored with different descriptors and cluster amounts. The SVC model with rbf kernel was also tested with varying cluster amounts. Ideally, all models should be tested with all descriptors and varying cluster amounts, but this is (very!) time-consuming and increasing cluster amounts increases the risk of overfitting.
- The parameters for the sub-models could be further fine-tuned.

8.5 The final models and received score

The two models that will be submitted for the Kaggle submission are the best performing SVC model and the final model discussed in this part. These models differ quite a lot when looking at the confusion matrices given in figure 12. The SVC model reaches higher peaks when looking at the best performance on a certain class while the final model seems more balanced. The Kaggle score of the best SVC model is 1.35777, the score of the best final model is 1.40394. This again suggests the data for the public leaderboard isn't that balanced.



(a) CM of the final model's underlying SVC model.



(b) CM of the final model.

Figure 12: Confusion matrices of the final model and its underlying SVC model.

Part IX

Conclusion

9.1 What is learned and achieved

This project has given a lot of insight into the working of different models and how model exploration is done. It has also shown that just blindly trusting `GridSearchCV` or received validation scores isn't a good idea. Whilst there is still a lot of room for improvement, the resulting final model seems to be pretty balanced and has an average accuracy of 50% or more for all classes. Taking into account this is using the more *old-school* Visual-Bag-of-Words approach, the results are satisfying and above average when looking at the public Kaggle leaderboard.

9.2 The hurdles

Some of the calculations can take a lot of time, which means making small errors can be very time punishing. Some of the fundamentals of a model, such as used descriptor and cluster amount, are also fixed for the whole model exploration. Changing these fundamental parameters can influence the whole model's performance and thus can not be done *after the fact*. This resulted in much recomputing work, which again, costs a lot of time. Due to limited computation power and time, experimenting with descriptors had to be kept to a minimum. This is most likely the part where most of the remaining gains can be made.

There is one known *data leakage* present, namely the clustering of the features is done using all training data instead of only the *training split*. When this flaw was discovered, K-Means clustering was already in use. Since this clustering algorithm takes 12+ hours to complete, this type of *data leakage* had to be kept in due to time limits. In an ideal world, only the training split would be used when using a validation set for validating the model. For the eventual model, the complete training data would be used and thus the clustering would be done on the complete training data, as is now the case. Thus, the expected impact of this data leakage for the final model is not expected to be drastic, but it was thought of.

9.3 Further extensions

As discussed, the models created in this report are by no means perfect. Most of the interesting further extensions are in experimenting with different descriptors and settings for them. A list of these possible extensions is given in section 8.4.

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

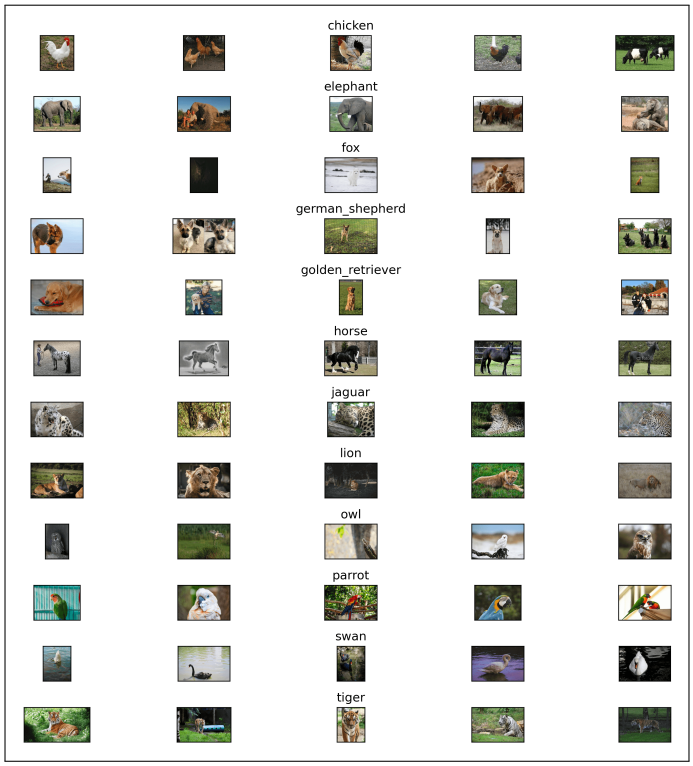


Figure 13: An overview of the supplied data per class.

Learning curves of the models

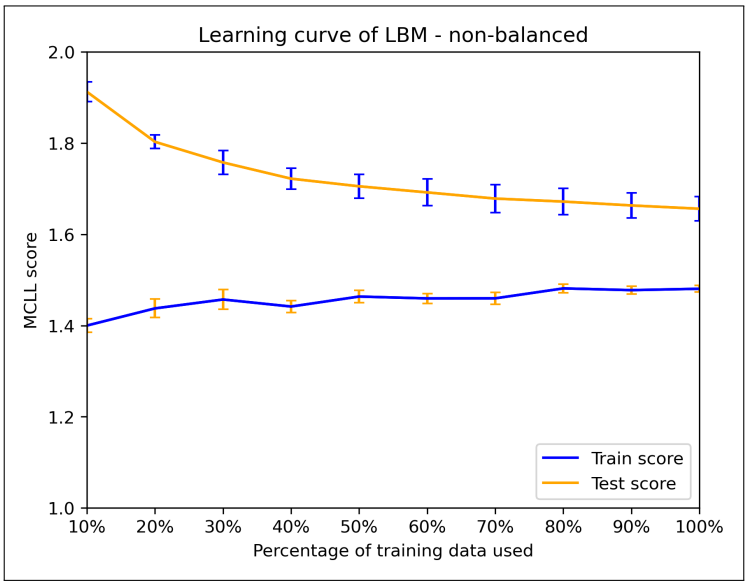


Figure 14: Learning curve of the non-balanced linear baseline model.

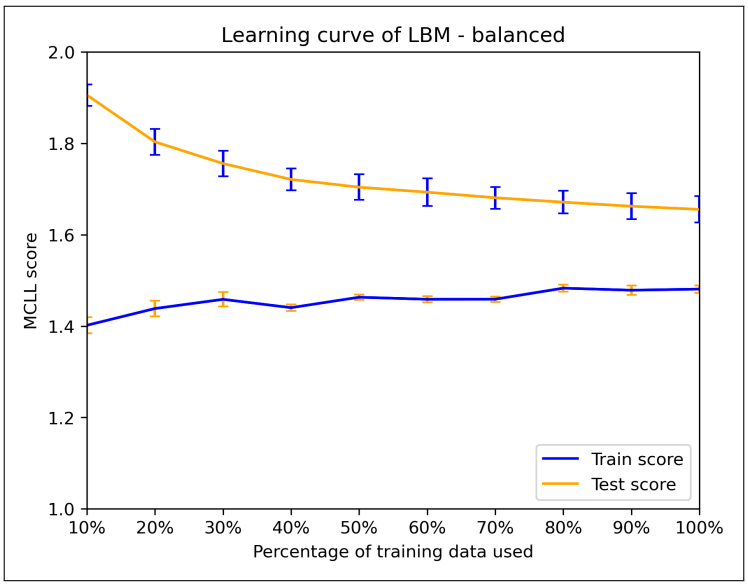


Figure 15: Learning curve of the balanced linear baseline model.

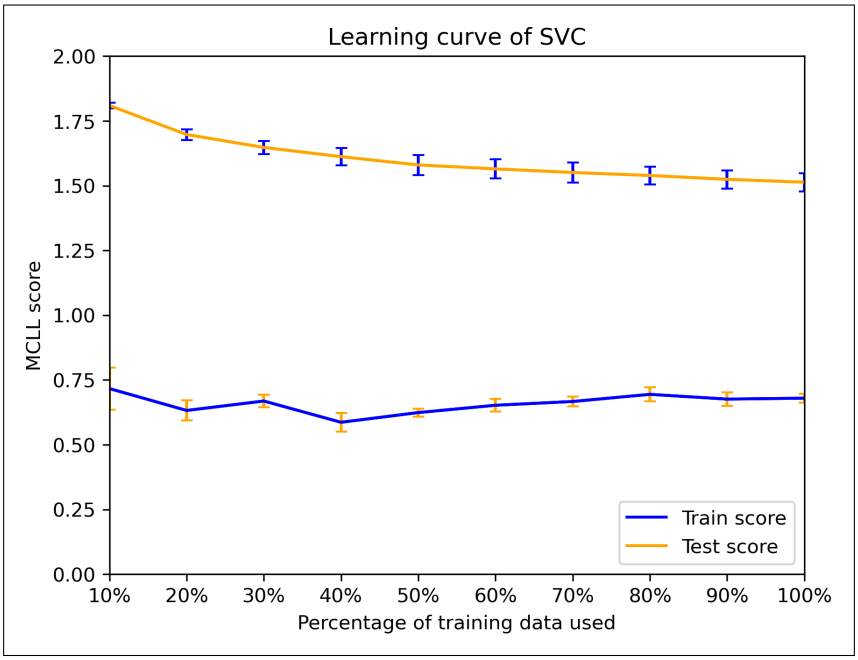


Figure 16: Learning curve of the SVC model.

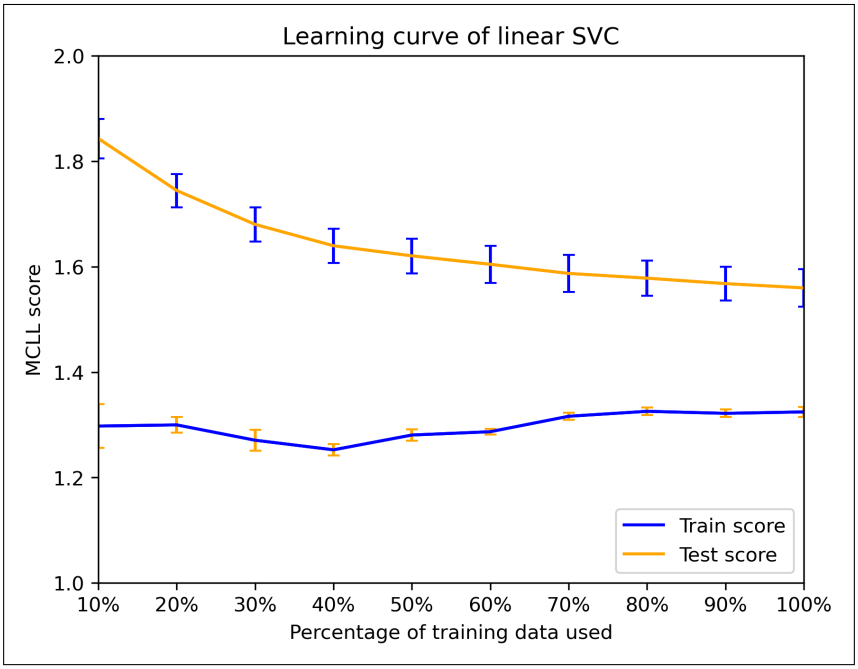
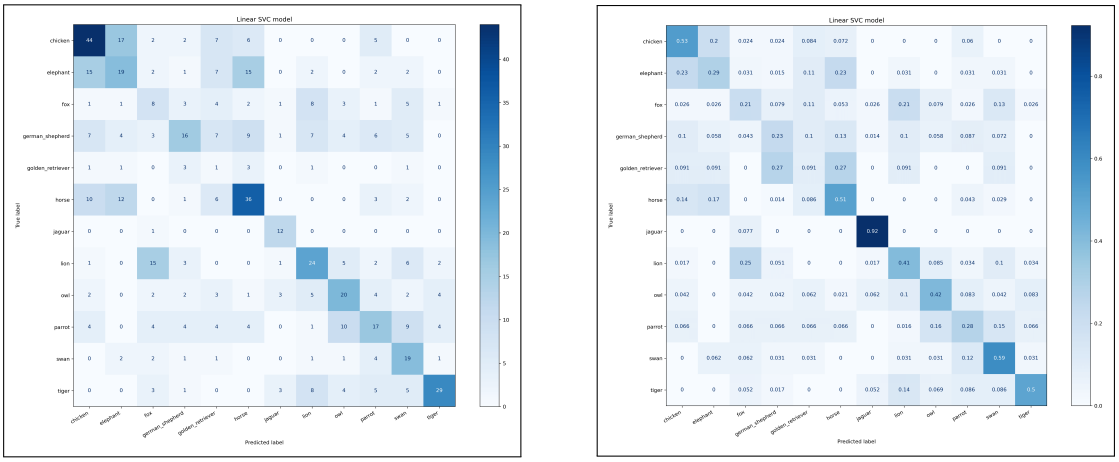


Figure 17: Learning curve of the linear SVC model.

Confusion matrix of linear SVC

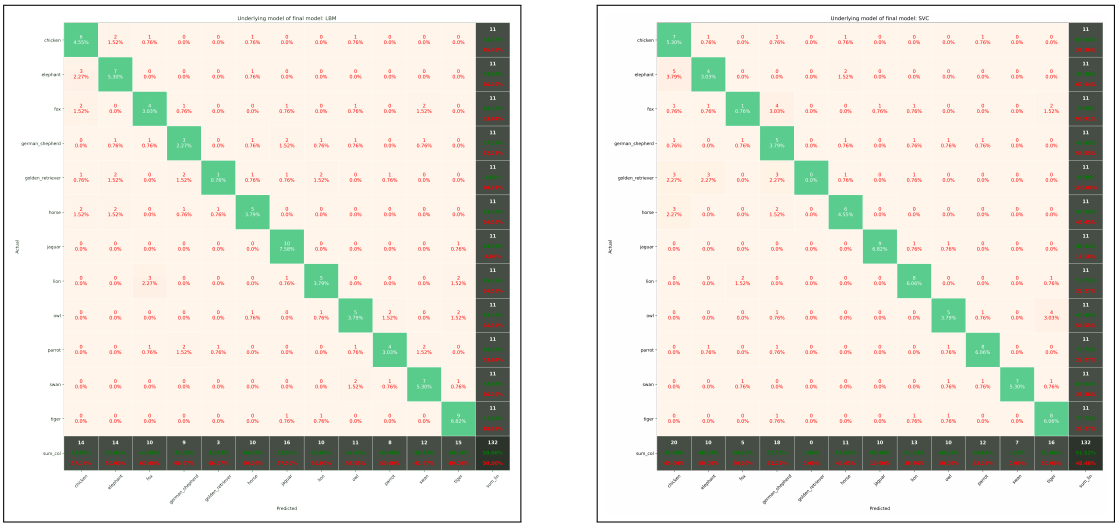


(a) Non normalised CM.

(b) Normalised CM.

Figure 18: Confusion matrices for the linear Support Vector Classifier model.

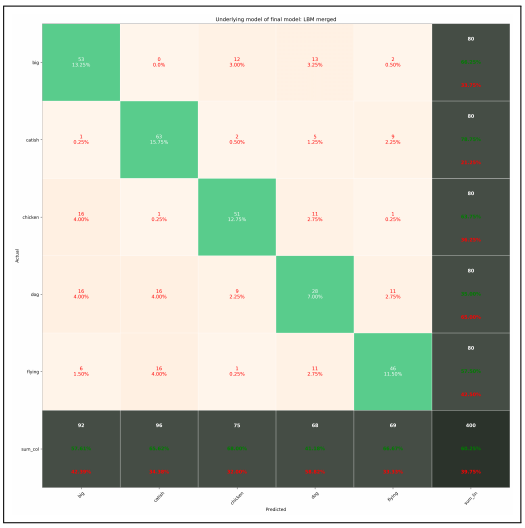
Confusion matrices of the final model's underlying models



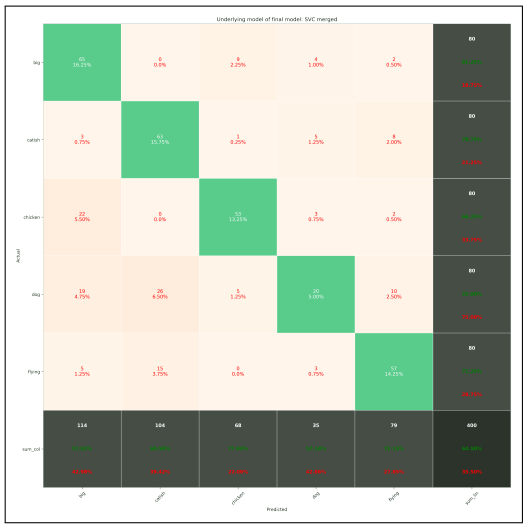
(a) Underlying LBM model.

(b) Underlying SVC model.

Figure 19: Underlying models of the final model using the complete data set.

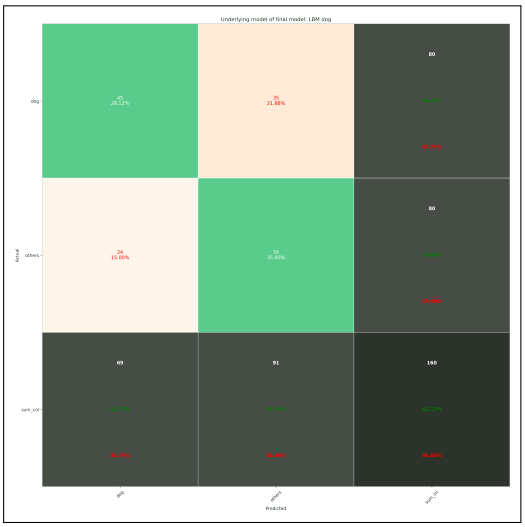


(a) Underlying LBM model.

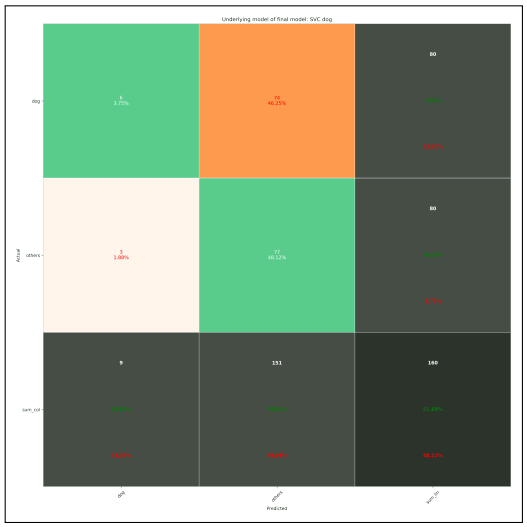


(b) Underlying SVC model.

Figure 20: Underlying models of the final model using the merged classes data set.

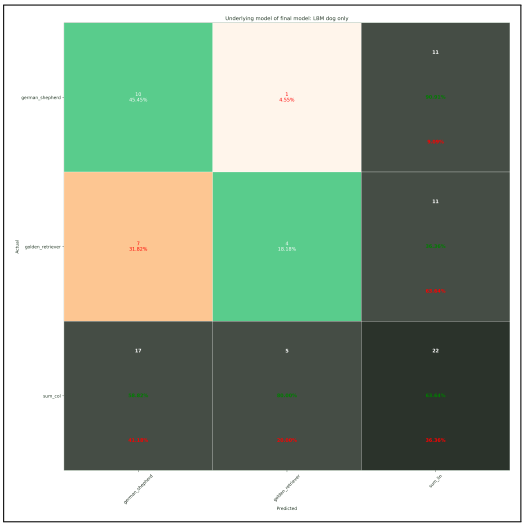


(a) Underlying LBM model.

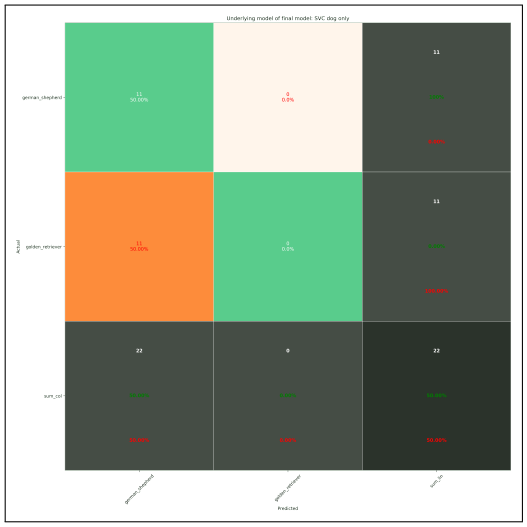


(b) Underlying SVC model.

Figure 21: Underlying models of the final model using the dogs vs others data set.

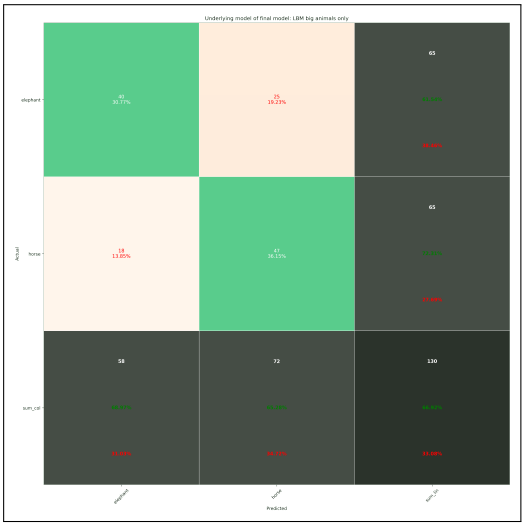


(a) Underlying LBM sub-model.

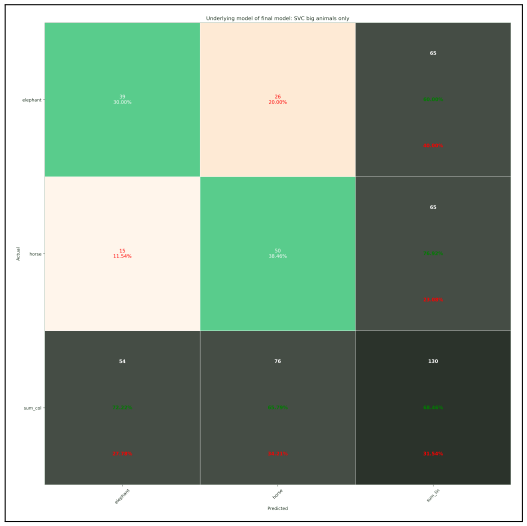


(b) Underlying SVC sub-model.

Figure 22: Underlying sub-models of the final model using a dog only split of the data set.

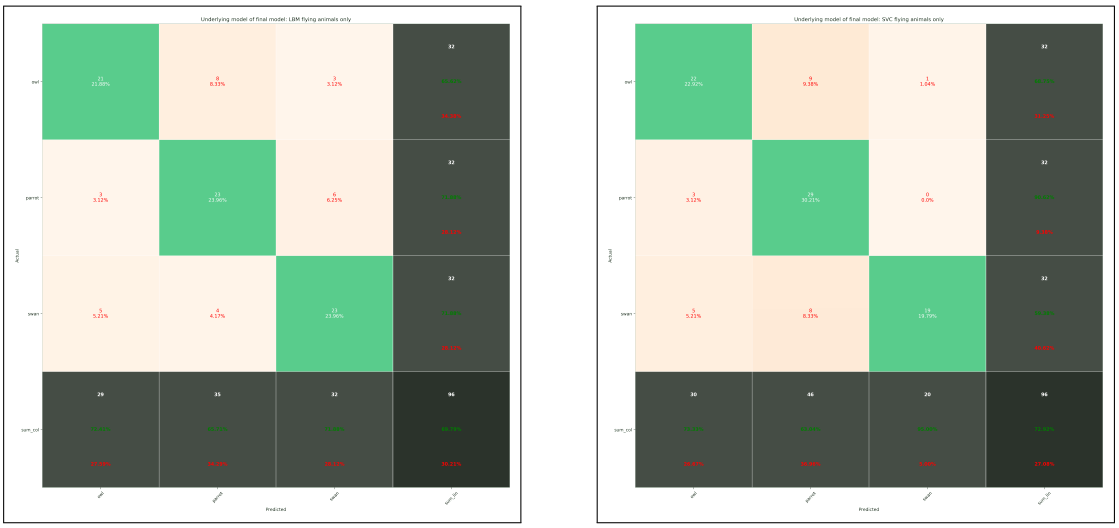


(a) Underlying LBM sub-model.



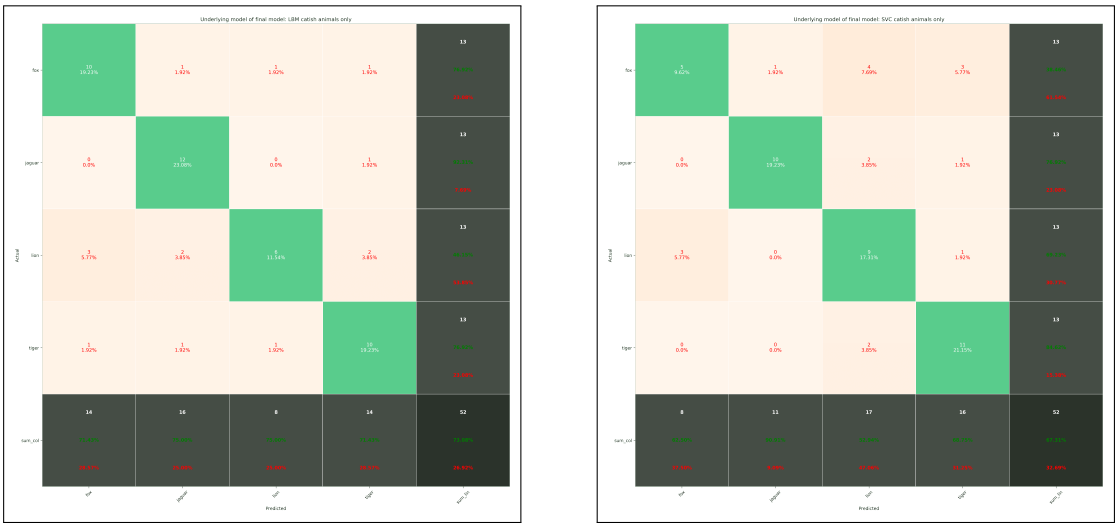
(b) Underlying SVC sub-model.

Figure 23: Underlying sub-models of the final model using a *big animals* split.



(a) Underlying LBM sub-model. (b) Underlying SVC sub-model.

Figure 24: Underlying sub-models of the final model using a *flying animals* split.



(a) Underlying LBM sub-model. (b) Underlying SVC sub-model.

Figure 25: Underlying sub-models of the final model using a *catish animals* split.

References

Alonso, J. (2013). K-means vs mini batch k-means: A comparison.

Bontinck, L. (2020). *Machine learning project* [GitHub commit: 427818389ce19ba304ef7159842910e1f7a958b3]. Retrieved January 14, 2021, from https://github.com/VUB-CGT/ml-project-2020-pikawika

Cipriano, W. (2020). *Confusion matrix in python* [GitHub commit: 8f5688b138b6ab0fa2678cb74fb4e8d602027b08]. Retrieved January 10, 2021, from https://github.com/wcipriano/pretty-print-confusion-matrix

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/

Vijayan, V., & KP, P. (2020). A comparative analysis of rootsift and sift methods for drowsy features extraction [Third International Conference on Computing and Network Communications (CoCoNet'19)]. *Procedia Computer Science*, *171*, 436–445. https://doi.org/https://doi.org/10.1016/j.procs.2020.04.046

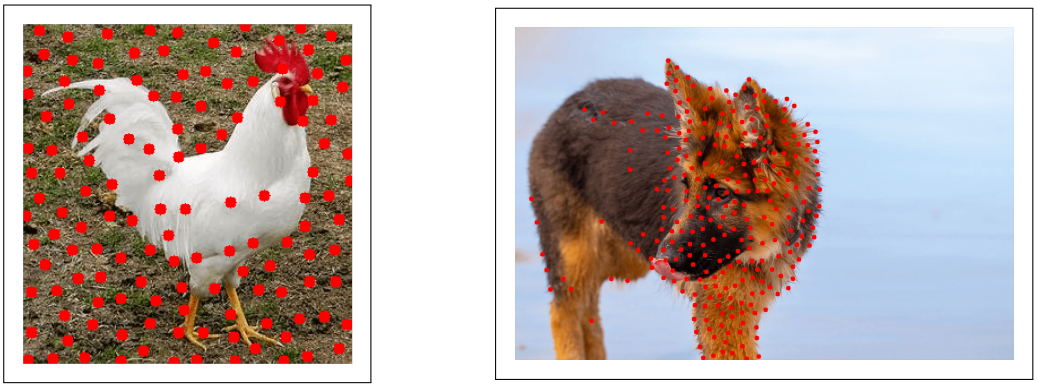
Appendix A: Working and representation of the descriptors

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 since it consists of provided code for the Kaggle competition.

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 often found by using the *Shi-Tomasi corner detector*, but some descriptors have different implementations. As the name suggests, these interesting points are *strong corners on an image*.

Shown in figure 26 is an example output of interesting points found by the Shi-Tomasi corner detector. It's clear that its 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. It's also noted that SIFT is used for the models in this report, which uses a different, patented, technique for interesting point detection.



(a) First chicken in data.

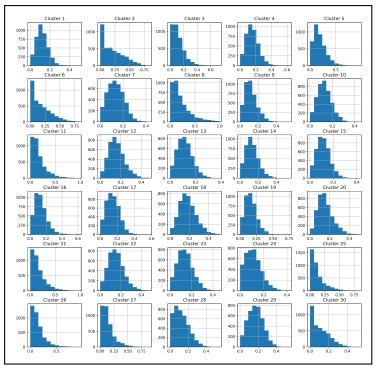
(b) First German shepherd in train data

Figure 26: Points of interest found by the Shi-Tomasi corner detector.

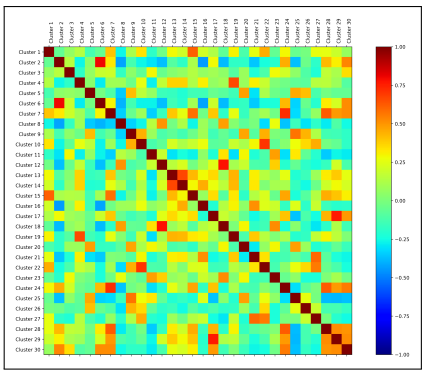
The numerical representation

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 using the `createCodebook` function which uses Mini-Batch K-Means clustering. This could also benefit from fine-tuning.

An overview showing histograms for each word (cluster) and a corresponding correlation matrix when opting for SIFT with 30 clusters is shown in figure 27. It is visible that the values are normalized. 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 in this case ($|\text{correlation value}| \neq 1$). A low correlation between clusters is mostly positive for model building since it suggests 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 27: Data analysis of encoded images using SIFT with 30 clusters.

Appendix B: Important LBM 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.