



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 (Rousseau, 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, best performing, 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

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	7
3.1	About this part	8
3.2	Scoring used to evaluate the model	8
3.3	Fine-tuning the input	8
3.4	Fine-tuning the validation set	9
3.5	Fine-tuning the model parameters	9
3.6	The optimal settings for this model	11

<i>CONTENTS</i>	iii
IV Support Vector Classifier	12
4.1 About this part	13
4.2 Scoring and methodology for fine-tuning	13
4.3 The optimal settings for this model	14
V Linear SVC	15
5.1 About this part	16
5.2 Scoring and methodology for fine-tuning	16
5.3 Linear is not the way to go	16
VI Gradient Boosting	17
6.1 About this part	18
6.2 Scoring and methodology for fine-tuning	18
6.3 Interesting but not impressive	18
VII Model analysis	19
7.1 About this part	20
7.2 Non-balanced Linear baseline model	20
7.3 Balanced linear baseline model	20
7.4 Support Vector Classifier model	21
7.5 Linear SVC and Gradient Boosting	22
VIII The final model	23
8.1 About this part	24
8.2 Making a better model	24
8.3 Combining powers	24
8.4 Unexplored possibilities	25

<i>CONTENTS</i>	iv
8.5 The final model and received score	25
IX Conclusion	26
9.1 What is learned	27
9.2 The hurdles	27
9.3 Further extensions	27
More figures	28
References	32
Appendix A: LRM parameters	33

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 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 L^AT_EX 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 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 minimize 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.

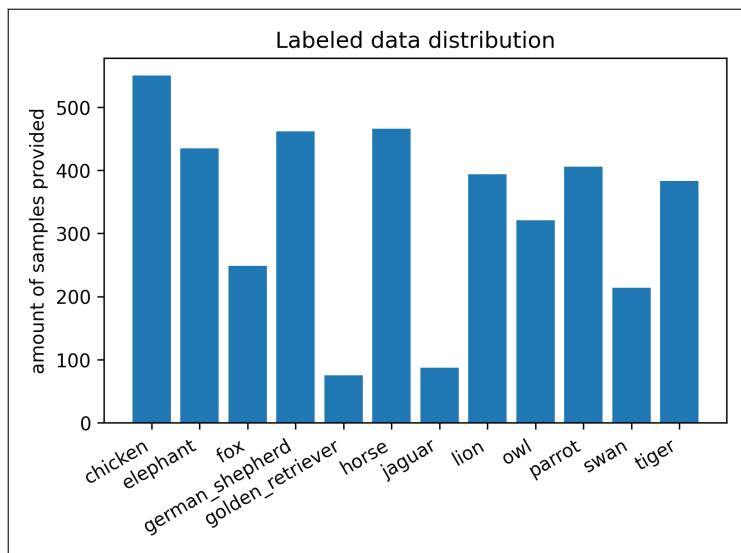


Figure 1: Data distribution training data.

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 14, 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 with by the used descriptor.

2.4 Feature extraction

Some feature extraction has already been provided. In short, images are converted from their 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 2 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 than 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.

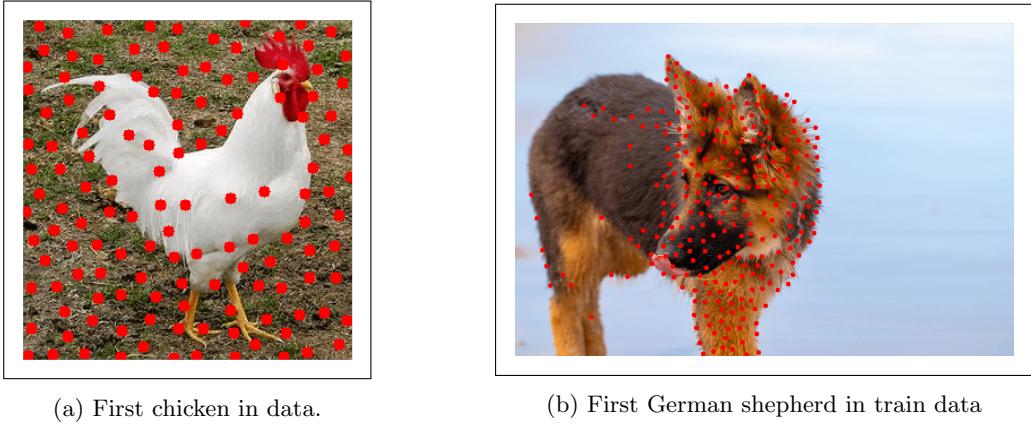


Figure 2: Points of interest found by the Shi-Tomasi corner detector.

2.5 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 minimizes 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 3. 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.

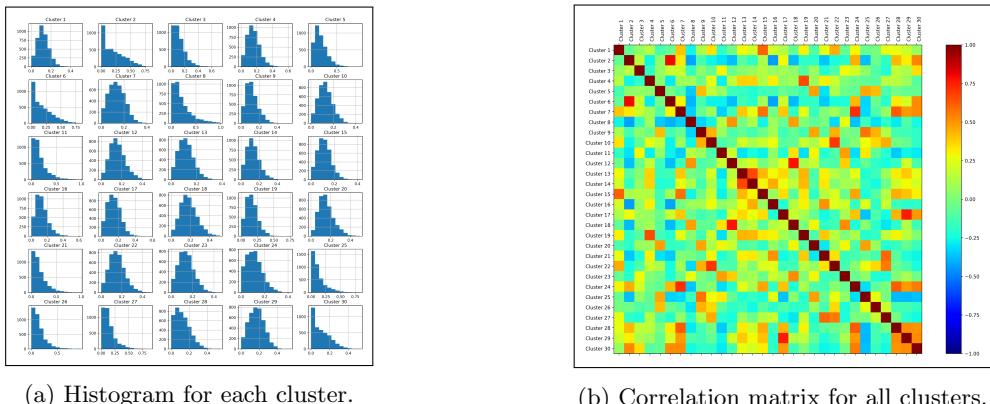


Figure 3: Data analysis of encoded images using SIFT with 30 clusters.

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 than 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 (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). 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 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 (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 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 fewer clusters suggesting it is more general. In contrast, the *Daisy* approach has signs of overfitting. With a lower cluster amount, the correlation between clusters will also be less as discussed in section 2.5, which is favourable. The *SIFT* descriptor with 100 clusters will also be the default for this report.

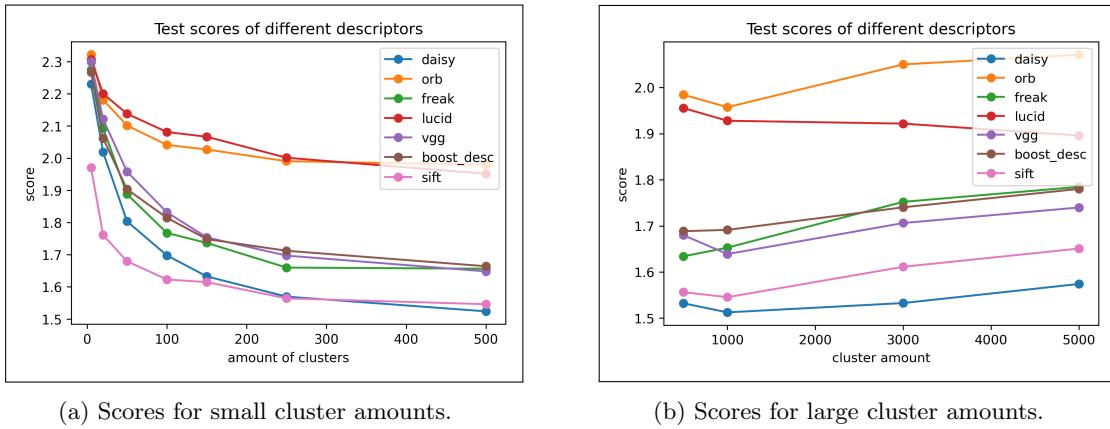


Figure 4: Average MCLL score over 5 trials for different descriptors and cluster amounts. Lower score is better.

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. 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%. This will again be used as default in this report.

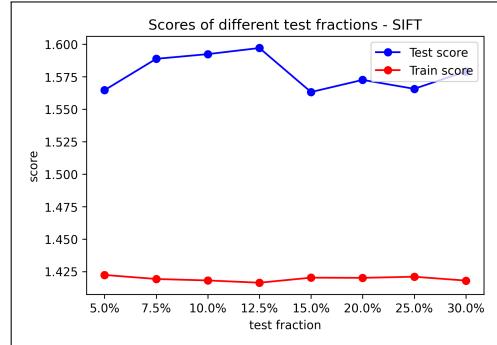


Figure 5: Testing different test sizes.
Lower score is better.

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 A.

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. 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 6.

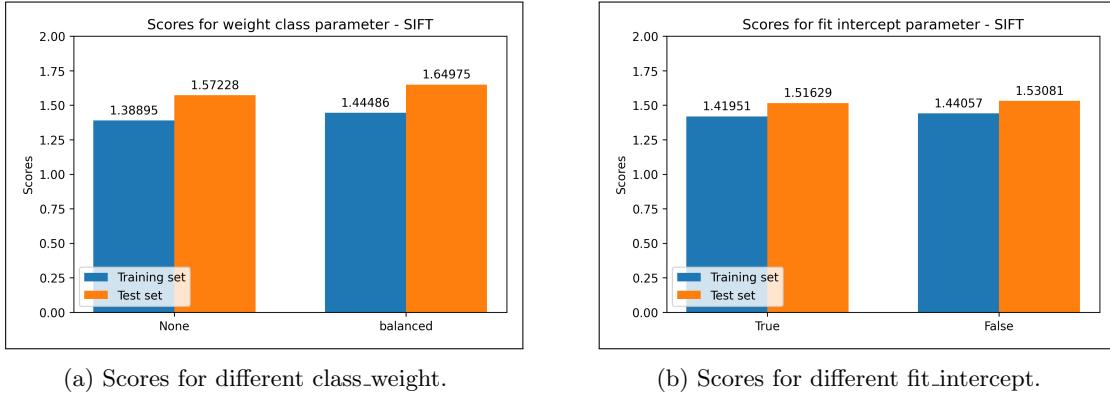


Figure 6: Average MCLL score over 10 trials for different model settings.
Lower score is better.

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 7. Since both approaches yield similar results, finding optimal parameters where no human reasoning is needed will be done using `GridSearchCV` in the future.

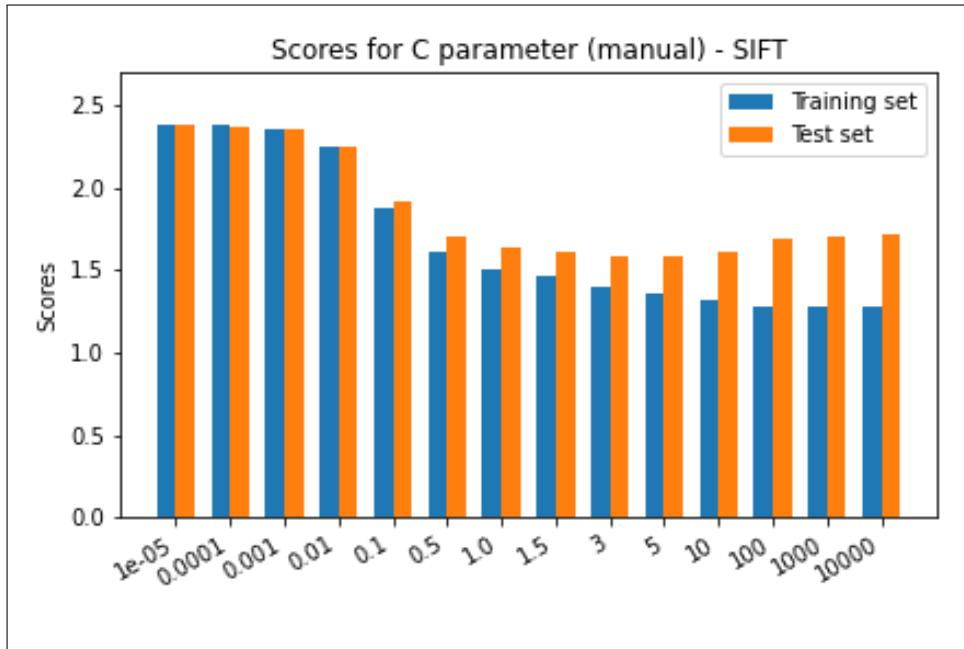


Figure 7: Trying different C values. Lower score is better.

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 given below. Note that the received Kaggle score is not drastically different from the received test scores which suggest no data leakage happened.

- 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

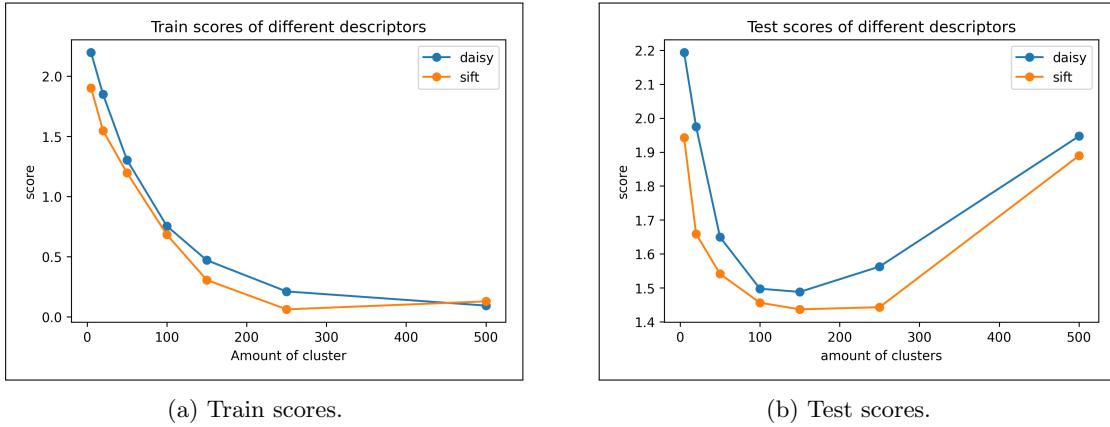
Now that a fine-tuned linear baseline model is established, the search for better models can start. Sci Kit learn has a flow chart for deciding which estimator to use, available here. When following this chart a Support Vector Classifier (SVC) is proposed, thus one is examined here. Since part III already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

4.2 Scoring and methodology for fine-tuning

Again, the multi-class Log Loss (MCLL) score is used to compare models. `GridSearchCV` was used to find optimal parameters were no human reasoning is needed. Figure 8 shows the results of experimenting with different clusters and descriptors. It is noted that overfitting is clearly visible when using over 250 clusters. For the final model SIFT with 100 clusters is used.

Balancing the class weight had a small positive impact, which is great! The non-balanced one is discussed here but found parameters and reasoning are identical for both. 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 .



(a) Train scores.

(b) Test scores.

Figure 8: 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 9. 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.

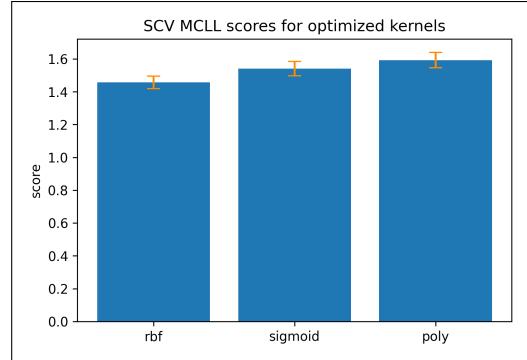


Figure 9: SCV MCLL score per kernel.
Lower score is better.

4.3 The optimal settings for this model

The optimal 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. 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 were 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 determine 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 ± 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. A MCLL score of ± 1.68 on the validation set, a SCV MCLL score of ± 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

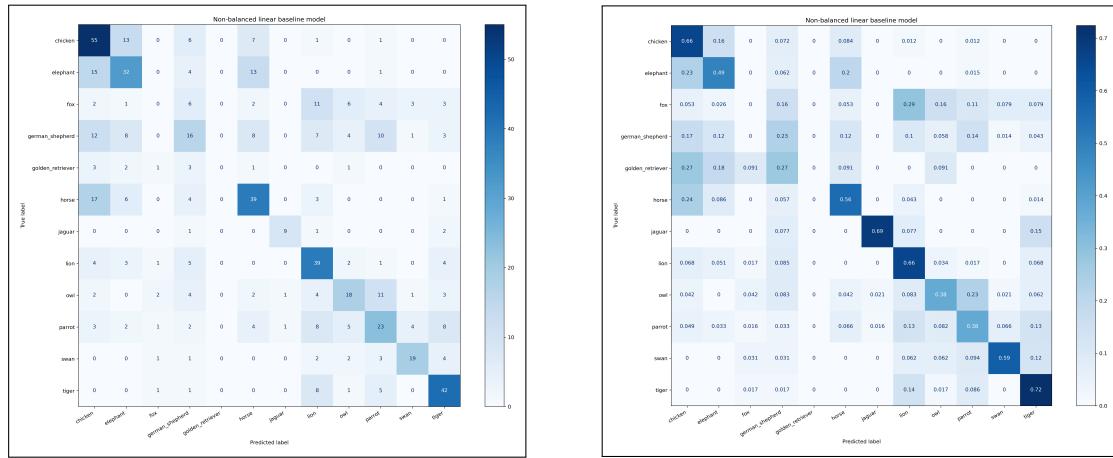
7.1 About this part

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 matrix given in figure 10 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.

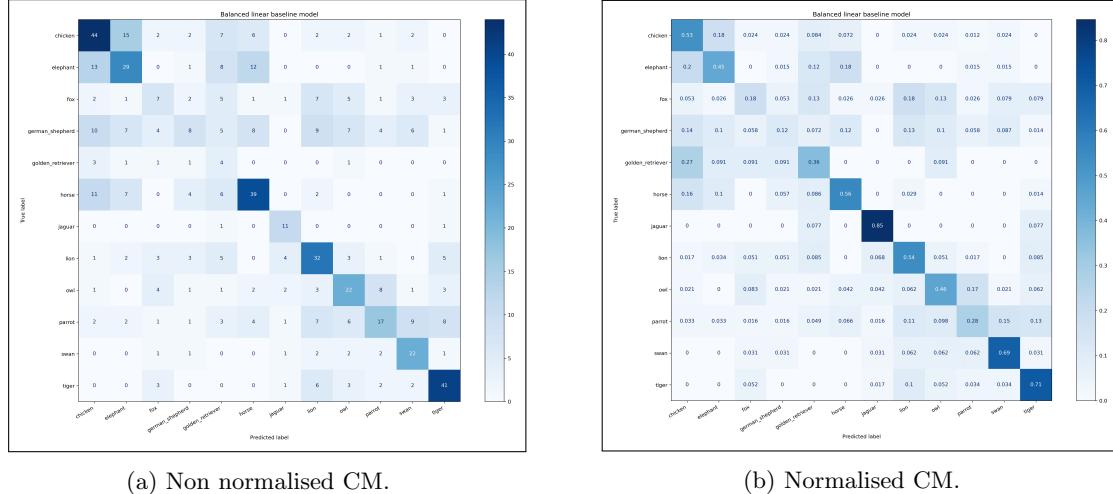
(b) Normalised CM.

Figure 10: 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 matrix given in figure 11 shows a different story and teaches us multiple things, the most important of which are listed here:

- The model now has some success 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 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.

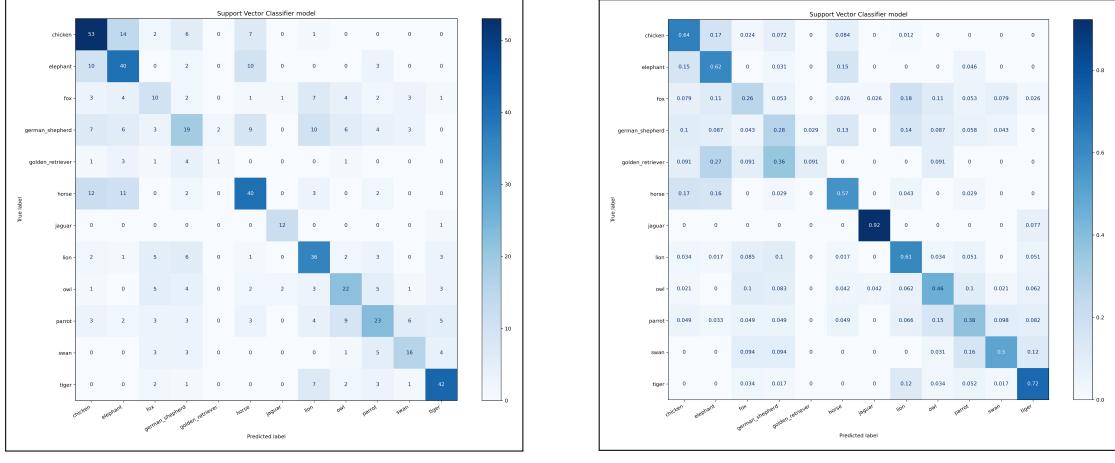
(b) Normalised CM.

Figure 11: 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 12 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 every class 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.

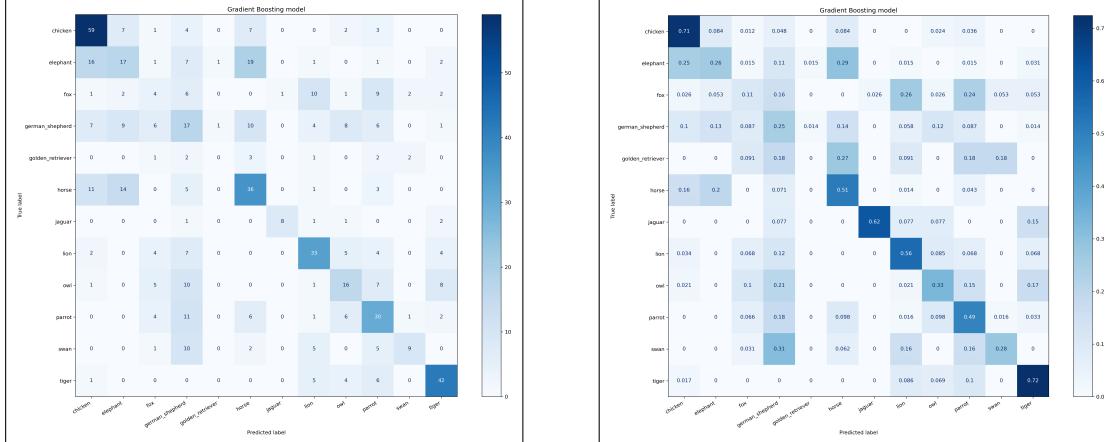
(b) Normalised CM.

Figure 12: 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 19).

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 13, 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 13: 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 the model analysis of part VII. Things that were not tried but which are *assumed* to increase performance are also listed here. This part will only briefly discuss all steps tried since many trials resulted in worse performance. The Notebook corresponding with this part is `final_model.ipynb`.

8.2 Making a better model

From what was learned in the previous parts, a final attempt at making the best linear baseline model (LBM) and support vector classifier (SVC) model was made. The following things were tried and are available in the notebook:

- Different scalers and transformers were tried as preprocessing. The SVC model did not benefit from this but applying the polynomial features transformer on the LBM model increased the performance by 0.01 on the test set and on Kaggle (now 1.57203)!
- The train set was balanced and overfitted, neither of which increased global or individual performance.
- Neither `AdaBoostClassifier` or the `BaggingClassifier` to make an ensemble of models performed well. This can be expected since such ensembles work well when ensembling *weak models*, which SVC and the LBM are not.
- TODO XXX
- For the final model, once all parameters are fine-tuned, the complete training set will be used. This means no scoring can be done except for a Kaggle submission since there is no unseen data left, but it does give the model more data to work with in its final form.

8.3 Combining powers

Multiple models are now finalised, each having some strengths and weaknesses. As a final approach, a model is made which uses the TODO XXX under the hood. It works as follows:

- A class `final_model` is made which has a `fit` and a `predict_proba` function. When fitting this class, it fits all the underlying models.
- When predicting with this class, the `predict_proba` function of the underlying models are called and the results are combined.
- TODO XXX

8.4 Unexplored possibilities

Whilst the achieved model performance is average when looking at the Kaggle leader board, some interesting possible optimisations that came to mind weren't implemented. To show that these were thought of and as possible further extensions some unexplored possibilities are listed here:

- Manipulating the images before using SIFT on them, 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 of orientation or size.
- Fine-tuning of the SIFT descriptor parameters. Experimenting with selections of features instead of just limiting/increasing them.
- 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.
- Using libraries to make a final ensemble of the different good performing models instead of using weighted preferences.
- The decision was made to work with the elbow model to determine an optimal amount of clusters, which turns out to be around 100. Performance could increase when bumping up this cluster amount (e.g. ≥ 1000) and opting for the daisy descriptor as discussed, but all parameters would need re-tuning and the risk of overfitting becomes greater.

8.5 The final model and received score

TODO XXX

Part IX

Conclusion

9.1 What is learned

TODO XXX

9.2 The hurdles

TODO XXX

9.3 Further extensions

TODO XXX

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

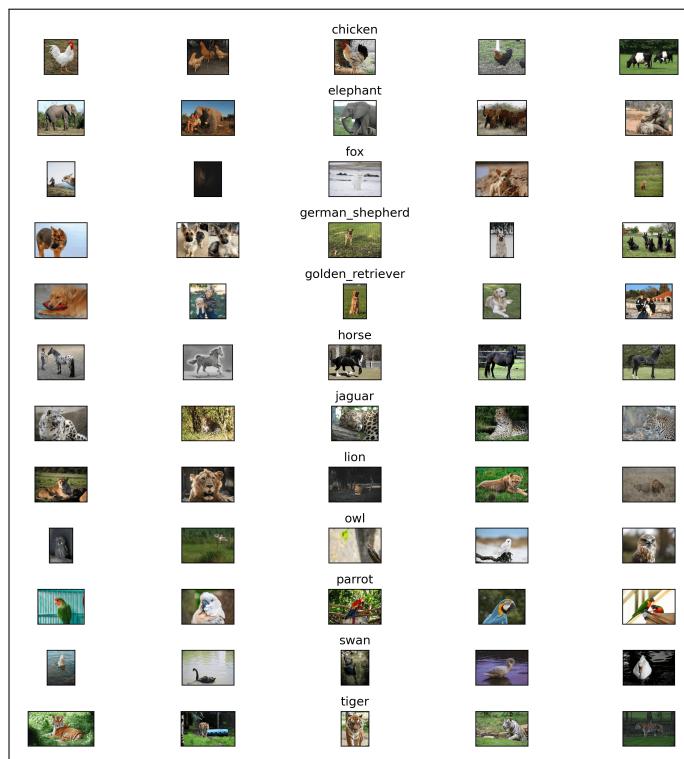


Figure 14: An overview of the supplied data per class.

Learning curves of the models

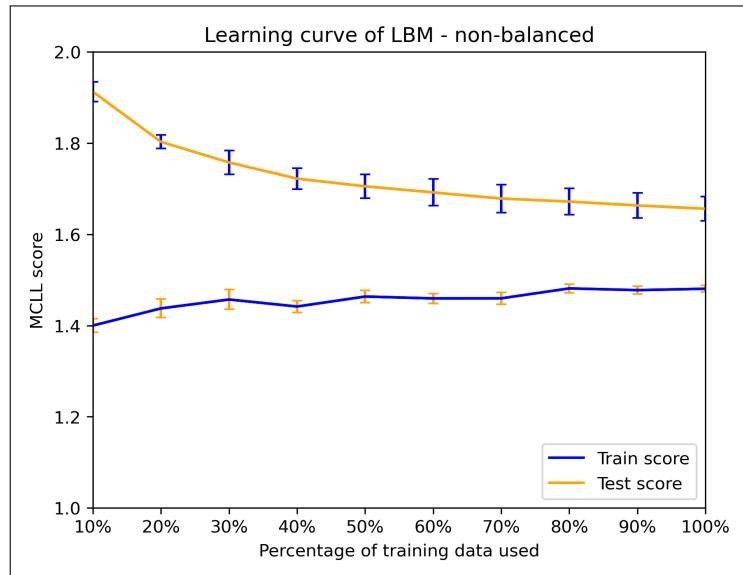


Figure 15: Learning curve of the non-balanced linear baseline model.

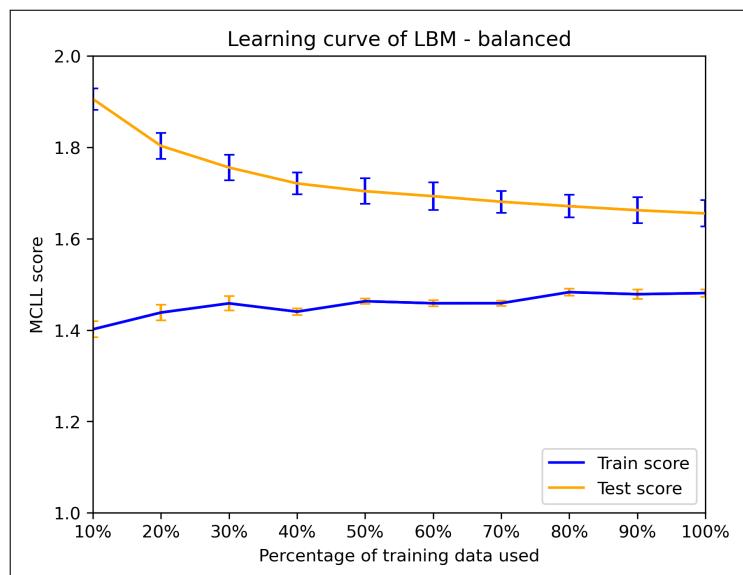


Figure 16: Learning curve of the balanced linear baseline model.

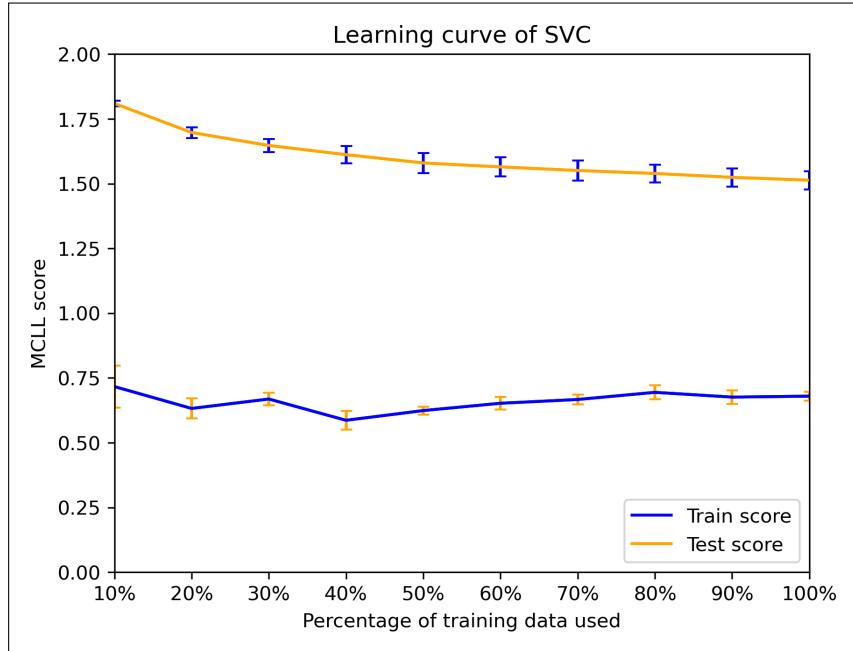


Figure 17: Learning curve of the SVC model.

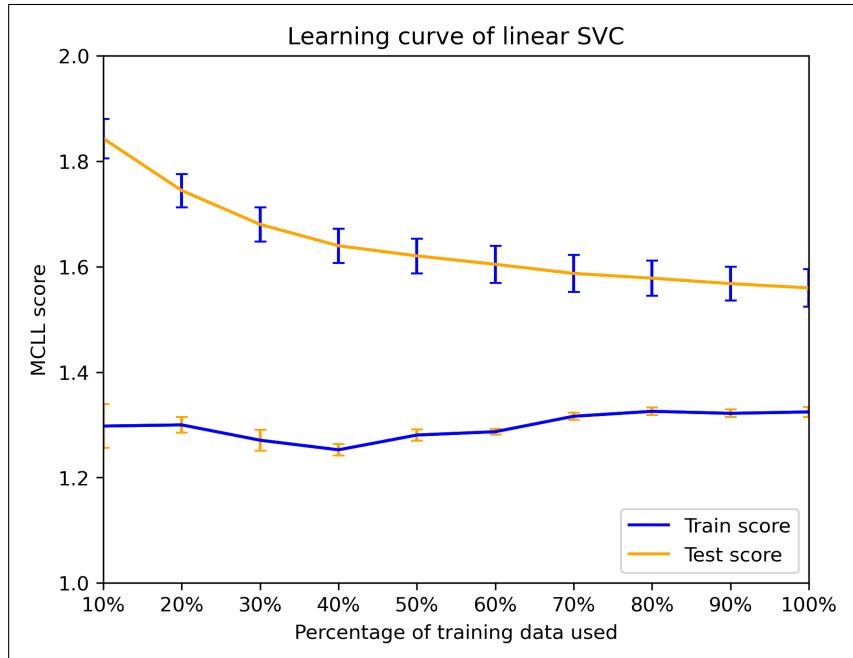
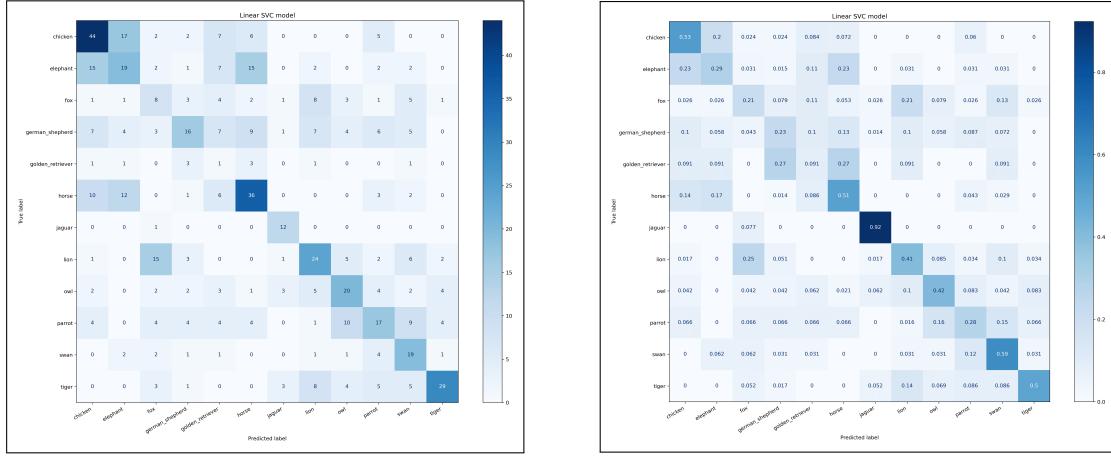


Figure 18: Learning curve of the linear SVC model.

Confusion matrix of linear SVC



(a) Non normalised CM.

(b) Normalised CM.

Figure 19: Confusion matrices for the linear Support Vector Classifier model.

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: 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.