Are visual bag-of-word models still relevant for image classification with the rise of open source deep learning frameworks?

P. KOVACS, UvA ID: 11389311
Y. TAN, UvA ID: 10185658
T. AKKERMANS, UvA ID: 11323671
D. OOSTERMAN, UvA ID: 10306048
T. BRITS, UvA ID: 10086358

---

1 INTRODUCTION

Is it more efficient these days to let a machine learn about a domain than to have domain expert program a machine? Most machine learning models use domain knowledge for feature engineering to reach high accuracies. In deep nets these features are learned automatically. Deep learning recently became available for mere mortals with the rise of user friendly frameworks like TensorFlow and the availability of pre-trained models. In this paper we compare the performance of feature engineered classification models on a 1000 image training sample with pre-trained deep learning nets.

2 METHODS

2.1 PCA dimension reduction presentation

In this method the flattened array of all RGB values is reduced in dimensionality using Principal Component Analysis with randomized SVD. The coefficients of the top n eigenvectors are used as features. (scikit-learn – User Guide - 2.5.1.3)

2.2 Visual Bag of Words representation

Visual bag of words is a method analogous to the bag of words technique used in text analysis: each image (document) is represented by extracted features (words) and mapped to a codebook (vocabulary).

For images the features can be random patches of pixels using the RGB color values, SIFT descriptors (Lowe, 2004) or others extractions. The codebook is defined as the centroids of the clusters of the features to which features are mapped by Euclidean nearest distance. The resulting normalized histogram of counts of mapped clusters has a fixed size for each image. This histogram is then the feature representation for that image. Clustering was done with k-Means for the RGB patches and with the faster MiniBatchKMeans (Sculley, 2010) for the SIFT features.

2.3 Deep Nets

Convolutional neural networks (CNN) are able to learn robust feature representations. However, learning them is quite expensive in terms of required computing power and data. Trying to learn discriminative features with a number of images as low as 1.000 would not be fruitful, therefore we will exploit the idea that low level filters in CNNs are similar for most image collections: in the network's first layers, CNN filters learn how to activate on edges and basic shapes (such as circles and rectangles) or how to detect different textures. These features are then combined into more complex detectors (that can recognize for example faces or eyes). To solve our problem of image classification with a low number of images per class, we have decided to use pre-trained network weights (convolutional filters) from a publicly available network called AlexNet (Krizhevsky, 2012). See the appendix for the AlexNet architecture.

3 EXPERIMENTS

3.1 PCA dimension reduction on full image RGB values

We varied the number of eigenvectors to be used and have found that keeping only 5 eigenvectors yielded the best accuracy (0.30) on the validation set using a KNN classifier. This dimensional number is a surprisingly low number given that we start out with 196.608 features. The 5 eigenvectors explained 49.5% of the variation in the images.

Other classifiers were applied. The average accuracies on the validation set are found below:

Classifier	**Accuracy (average of 5 runs)**
K-Nearest Neighbors (n=9)	0.30
Support Vector Machine	0.14
Decision Tree	0.21
Neural network (with backpropagation)	0.23
Gradient Boosting	0.28
AdaBoost (with Decision Tree)	**0.31**

The highest accuracy attained was 0.31 using AdaBoost with a Decision Tree Classifier. Just below that, an accuracy of 0.30 is achieved using K-Nearest Neighbors with n=9. We expect the random patch BoW approach to improve upon the PCA approach as BoW is translation invariant even though spatial information is lost.

3.2 Random patches with BoW representation

For patch sampling, we first construct a codebook by taking 200 random patches for each image with a shape of either 3x3 or 5x5 pixels and use the flattened array of RGB values as patch descriptors. A codebook of centroids is constructed by clustering all descriptors in 32 clusters.

Normalized histograms of an image are made by mapping 500 patch descriptors to their closest (by Euclidean distance) cluster. Classification is done using k-Nearest Neighbors and Weighted Nearest Neighbors. The accuracies on the valuation set for different patch sizes and different values for *k* can be found in the table below:

Method	**Accuracy (3x3 patches)**	**Accuracy (5x5 patches)**
K-Nearest Neighbors (k=1, 3, **10**, 15, 30)	0.38, 0.23, **0.395**, 0.215, 0.18	0.39, 0.245, 0.24, 0.17
Weighted Nearest Neighbors	0.385	0.37

The best accuracy of 0.395 was attained for patches of shape 3x3 using k-Nearest Neighbors with k=10.

3.2.1 Cosine distance.

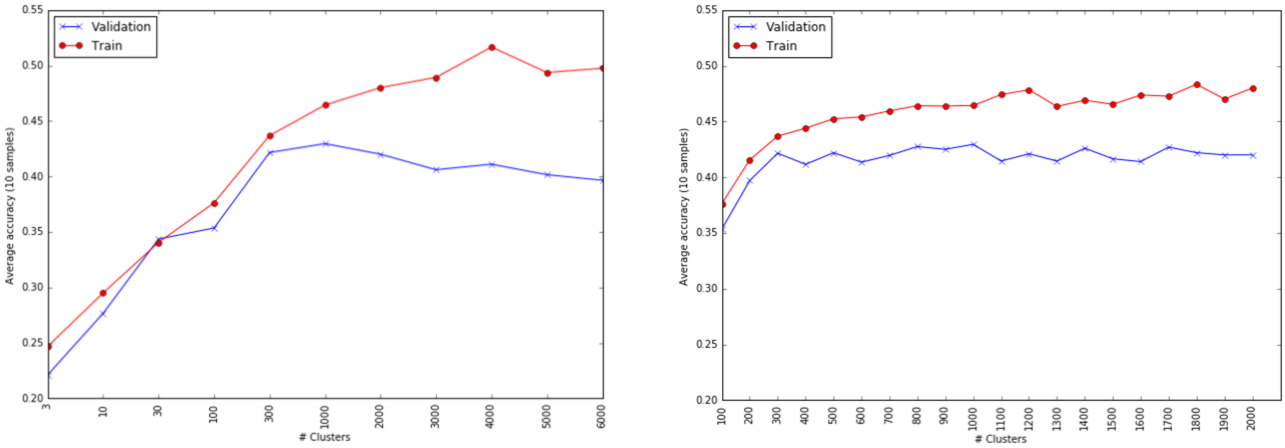
We experimented using the cosine similarity instead of Euclidean distance for classification. Since the cosine similarity measures the similarity of vectors by examining the direction, we expected that its performance might prove a good alternative to Euclidean distance, which measures the distance between the points along the vectors but is susceptible to be affected by the magnitude of vectors. This resulted in an accuracy of 0.305 for 3x3 patches making use of k-Nearest Neighbors with k=10. This causes the accuracy to be considerably lower that we decide to stay with Euclidean distance.

3.3 SIFT features

3.3.1 Baseline model. The SIFT parameters of the python course notebook were used. The baseline hyper parameters were set to a codebook cluster size of 100 and using 100 nearest neighbors in the classification task. The resulting accuracy on the validation set was 0.42. The model trained on both the train and validation set combined got an accuracy of 0.36 on the test set. Not a bad way to start compared to RGB patches.

To optimize this model, we investigated the codebook cluster size and multiple classifiers.

3.3.2 Codebook cluster size optimization.



The figure above on the left shows our initial run with in the first 6 increasing in a logarithmic scale (until 1000) and after that by the thousand. The accuracy scores are an average of 10 runs.

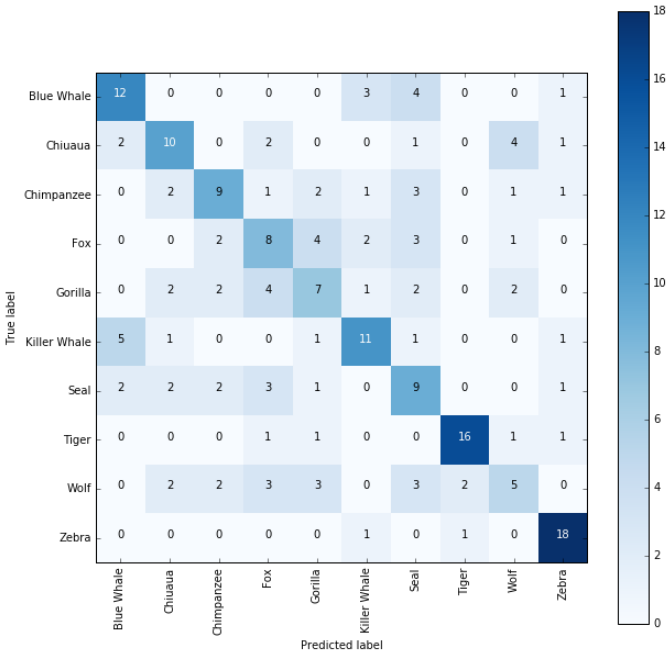
The classifier used was a Support Vector Machine (Cortes, Vapnik, 1995) with C=1 and scikit-learn default settings as this proved to give the best results in experiments.

We further investigated the region from 200 clusters to 2000 as this is where the train and validation accuracy is reasonably close and also still reasonably high. Higher number of of clusters gave a wider separation between train and test accuracies and thus probably overfitting or the clustering did not converge well enough to describe the data in a meaningful way. Lower number of clusters were performing less than optimal as higher accuracies with not much difference in train and test set are easily obtained with an increase in clusters. A number of 1000 clusters was chosen as the optimal size.

3.3.3 *Testing multiple classifiers*.
We chose 13 classifiers (see appendix) available in the scikit-learn library and tested which performed best. The (neural network) Multi-layer Perceptron classifier scored the highest accuracy of 0.51 (see appendix) on the validation set. We saw that 6 other classifiers performed decent as well with accuracies above 0.4. Therefore, it might be interesting to see what the result is of using multiple classifiers in the model. A fusion model was implemented that combined predictions of all classifiers but that performed worse than the MLP alone.

3.3.4 Confusion matrix analysis.

To get more insight in the classification results, we plotted the predictions into a confusion matrix format that shows which species are hard to classify and which species are often confused. In the figure below you can see the confusion matrix for the MLP classifier:



Here you can see that a killer whale is often mistaken for a blue whale. The same holds for the Chihuahua and the wolf and for the blue whale and the seal. You can also see that some species are hard to define in general. The wolf for example is incorrectly classified as a Chihuahua, chimpanzee, fox, gorilla, seal and tiger. All more than once. There was a high variety of predictions between the classifiers. It was therefore hard to discover a general classification issue throughout all classifiers.

3.3.5 Combining SIFT with RGB features

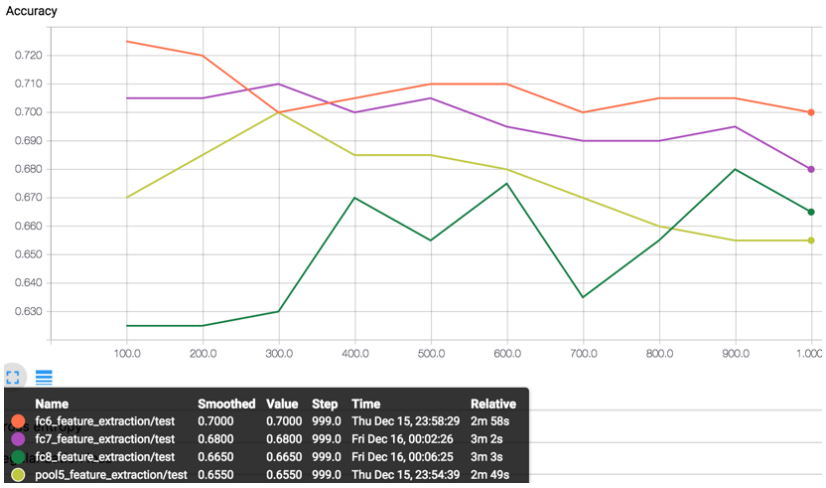
We attempted to achieve a higher accuracy by combining the RGB and SIFT features by appending each SIFT key point descriptor with the corresponding 3x3 RGB patch. Since RGB values do not occur in SIFT features, we expected to improve detection performance.

We applied the baseline BoW model: K-Means clustering (100 clusters) with a kNN (k=10) classifier. The addition of the RGB information increased the accuracy on the validation set slightly from 0.375 to 0.395.

3. 4 Using Deep Nets

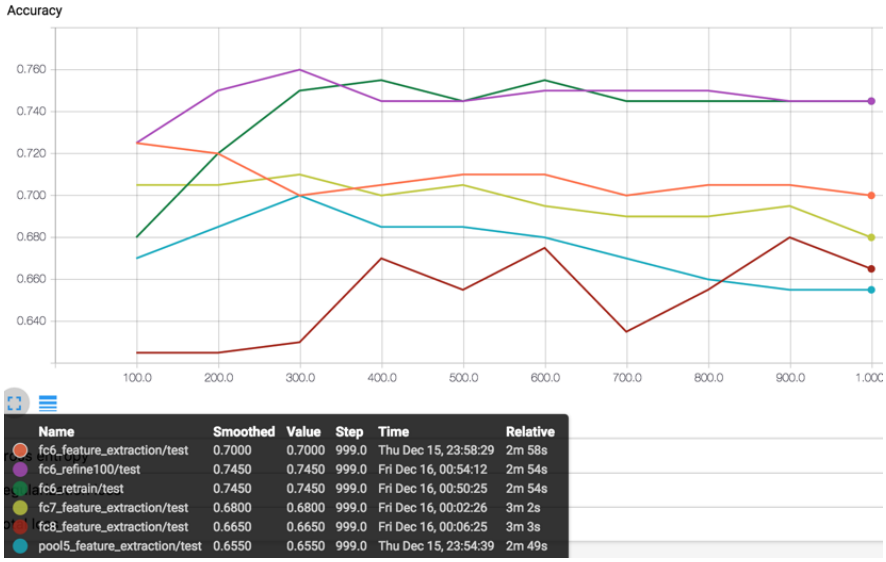
We belief that the low accuracy of BoW classifiers is caused by problems on multiple levels: starting with SIFT not being a really good feature extractor, losing information when constructing our visual dictionary and losing spatial information when converting extracted visual words into histograms. Convolutional networks mitigate those shortcomings.

AlexNet
The first question that arises is 'how many layers would should we take in order to have best feature representation of image?'. The first thing we tried was to take all the convolutional layers and then train a Softmax classifier on top of them. This approach returned high results (0.655), but we were interested in what would happens when we add more complex feature extractors by adding fully connected pre-trained layers?



As we see in the figure above, using the pool5 layer (the last layer before the two fully connected layers) yields non-optimal results. Therefore, we decided to stick with best performing solution: the first fully connected layer after pooling layer.

Feature extraction vs. retraining vs refining
After multiple experiments with AlexNet, we started wondering whether the pre-trained weights were good enough for our own dataset. After all, they have been trained on a different dataset. Until now we have been using it only for feature extraction, but what if we could make these feature detectors more specific for our classification task? We retrained our network (weights will be initialized with pre-trained ones) and also refined it: for the first k steps we used our network as feature extractor and after k iterations we trained all weights.

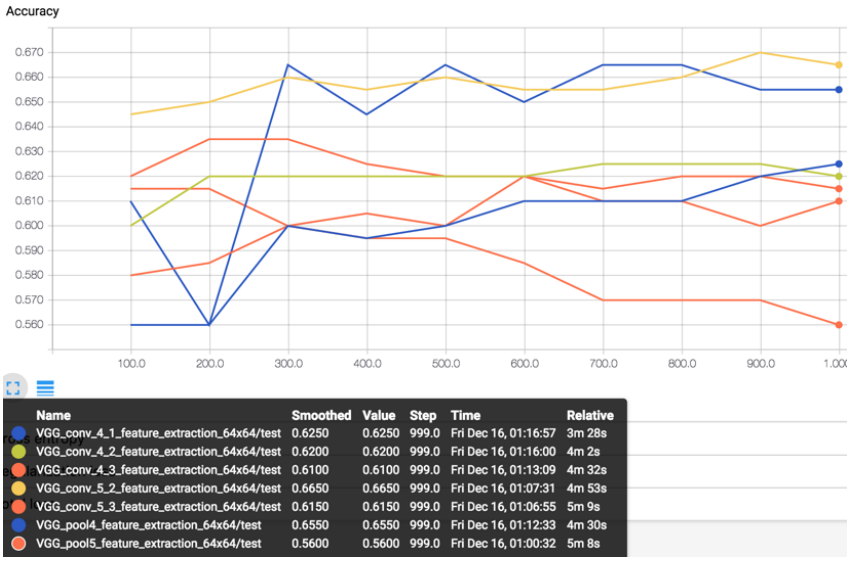


As we can see from the figure above, both retraining and refining helped improve the accuracy to 0.745.

VGG network

Noticing that AlexNet is a decent feature extractor for our task, we decided to try the deeper VGG architecture. Training deeper architecture is computationally very expensive. This lead us to experiment on reducing computational time by using smaller image sizes (64x64 pixels) but unfortunately image shrinking had a huge negative impact on our accuracy. For this reason, we decided to stick with AlexNet size images of 227x277.

As pool5 seems to be the best performing layer we tried to retrain it. Earlier, this tactic was successful yet this time it resulted in drastic drop in performance of 8%.



After training our best VGG model with an additional 200 images from the validation dataset, we achieved accuracy of 0.88 on the Kaggle test samples.

REFERENCES

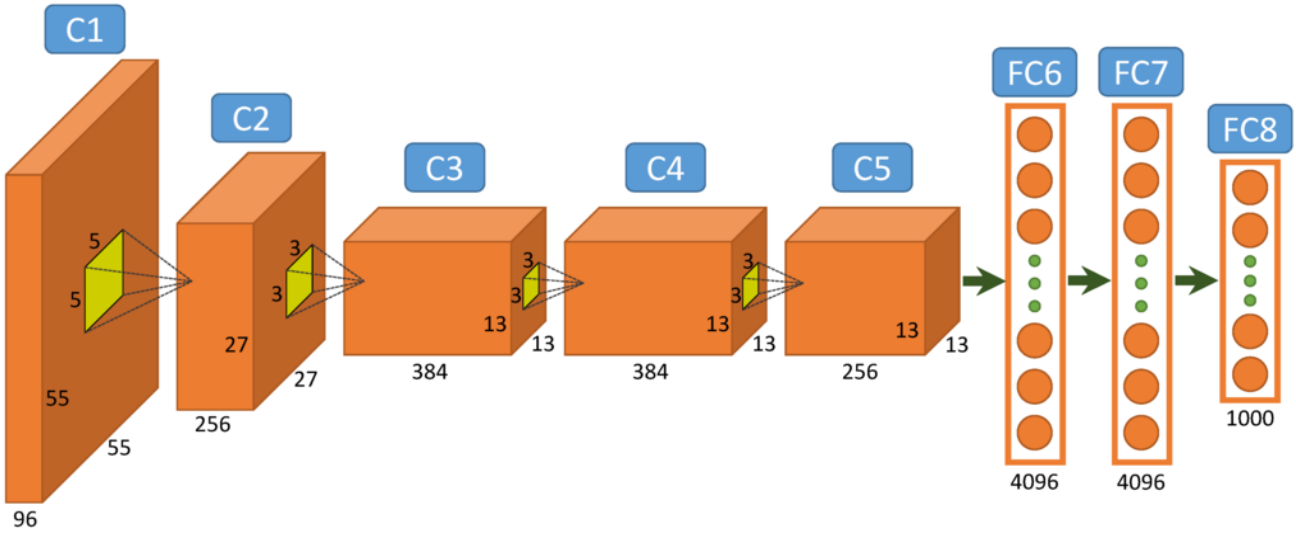
[1] David G. Lowe, 2004. Distinctive image features from scale-invariant keypoints International Journal of Computer Vision, 60, 2, pp.91-110
[2] D. Lowe, 2010. Web-Scale K Means Clustering Google, Inc. Pittsburgh. PA USA
[3] Alex Krizhevsky, 2012, ImageNet Classification with Deep Convolutional Neural Networks
[4] Cortes, C. & Vapnik, V. 1995, Support-vector networks. Machine learning, 20(3), 273-297

TEAM REFLECTION

All team members were happy with the workload distribution and contributed equally. All team members did research, implemented machine learning models and analyzed different aspects leading to the optimization of the results.

APPENDIX

AlexNet Architecture



C layers are convolutional layers; FC layers are fully connected layers.
The pool5 layer is the pooling layer after convolutional layer 5.

SIFT classifier settings

Accuracy	Note
0.36400	• SIFT, • KMeans k=100, • KNeighborsClassifier k=98
0.41200	• SIFT, • Kmeans k=100 • KNeighborsClassifier(n_neighbors=74) • RandomForestClassifier(n_estimators=300) • GradientBoostingClassifier(n_estimators=300,max_depth=3) Fusion with most common
0.47200	• SIFT • KMeans k=1000 • Fusion (RandomForestClassifier(n_estimators=1000) • GradientBoostingClassifier, SVC(kernel="linear", C=1) • Gridsearch for GradientBoostingClassifier: GradientBoostingClassifier_params = {'learning_rate':0.01, 'max_features':'sqrt', 'n_estimators':1000, 'max_depth':9, 'min_samples_split':162, 'min_samples_leaf':30, 'max_features':30,'subsample':0.8}
0.47200	• SIFT • Kmeans 5000 clusters • fusion of classifiers above 0.5 accuracy
0.42800	• reduced SIFT-points 50 random • Fusion Kmeans k=100 • Fusion of 10 classifiers
0.45600	• SIFT reduced to 50 random points per image • KMeans k=200 max_iter=300, voting-fusion of (KNeighborsClassifier(21) • SVC(kernel="linear", C=0.00015) • GradientBoostingClassifier(n_estimators=300,max_depth=3) • SVC(kernel="sigmoid", C=430), GaussianProcessClassifier() • RandomForestClassifier(max_depth=3,n_estimators=300) • MLPClassifier(max_iter=200,hidden_layer_sizes=200,activation='logistic',solver='adam') • GaussianNB() • NearestCentroid())

Best performing classifiers per species based on F1 score.

Species	Classifier index	Classifier
Blue Whale	12	NearestCentroid(metric='euclidean', shrink_threshold=None)
Chiuaua	9	MLPClassifier(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9, beta_2=0.999, early_stopping=False, epsilon=1e-08,hidden_layer_sizes=(100,), learning_rate='constant', learning_rate_init=0.001, max_iter=200, momentum=0.9, nesterovs_momentum=True, power_t=0.5, random_state=None, shuffle=True, solver='adam', tol=0.0001, validation_fraction=0.1, verbose=False, warm_start=False)
Chimpanzee	5	SGDClassifier(alpha=0.0001, average=False, class_weight=None, epsilon=0.1,eta0=0.0, fit_intercept=True, l1_ratio=0.15,learning_rate='optimal', loss='hinge', n_iter=5, n_jobs=-1,penalty='l2', power_t=0.5, random_state=None, shuffle=True,verbose=0, warm_start=False)
Fox	4	LinearDiscriminantAnalysis(n_components=None, priors=None, shrinkage=None, solver='svd', store_covariance=False, tol=0.0001)
Gorilla	3	NuSVC(cache_size=200, class_weight=None, coef0=0.0, decision_function_shape=None, degree=3, gamma='auto', kernel='rbf', max_iter=-1, nu=0.5, probability=False, random_state=None, shrinking=True, tol=0.001, verbose=False)
Killer Whale	9	MLPClassifier(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9,beta_2=0.999, early_stopping=False, epsilon=1e-08,hidden_layer_sizes=(100,), learning_rate='constant', learning_rate_init=0.001, max_iter=200, momentum=0.9,nesterovs_momentum=True, power_t=0.5, random_state=None,shuffle=True, solver='adam', tol=0.0001, validation_fraction=0.1,verbose=False, warm_start=False)
Seal	8	RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini', max_depth=None, max_features='auto', max_leaf_nodes=None, min_impurity_split=1e-07, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, n_estimators=1000, n_jobs=-1, oob_score=False, random_state=None, verbose=0, warm_start=False)
Tiger	9	MLPClassifier(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9, beta_2=0.999, early_stopping=False, epsilon=1e-08, hidden_layer_sizes=(100,), learning_rate='constant', learning_rate_init=0.001, max_iter=200, momentum=0.9, nesterovs_momentum=True, power_t=0.5, random_state=None, shuffle=True, solver='adam', tol=0.0001, validation_fraction=0.1, verbose=False, warm_start=False)
Wolf	1	SVC(C=1, cache_size=200, class_weight=None, coef0=0.0, decision_function_shape=None, degree=3, gamma='auto', kernel='rbf', max_iter=-1, probability=False, random_state=None, shrinking=True, tol=0.001, verbose=False)
Zebra	9	MLPClassifier(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9, beta_2=0.999, early_stopping=False, epsilon=1e-08, hidden_layer_sizes=(100,), learning_rate='constant', learning_rate_init=0.001, max_iter=200, momentum=0.9, nesterovs_momentum=True, power_t=0.5, random_state=None, shuffle=True, solver='adam', tol=0.0001, validation_fraction=0.1, verbose=False, warm_start=False)

Initial accuracies using several scikit-learn classifiers

Classifier	Parameters	Initial accuracy
KNeighborsClassifier()	algorithm='auto', leaf_size=30, metric='minkowski', metric_params=None, n_jobs=-1, n_neighbors=250, p=2,weights='uniform'	0.355
SVC()	C=1, cache_size=200, class_weight=None, coef0=0.0, decision_function_shape=None, degree=3, gamma='auto', kernel='rbf', max_iter=-1, probability=False, random_state=None, shrinking=True, tol=0.001, verbose=False	0.405
GradientBoostingClassifier()	criterion='friedman_mse', init=None, learning_rate=0.1, loss='deviance', max_depth=3, max_features=None, max_leaf_nodes=None, min_impurity_split=1e-07, in_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, n_estimators=100, presort='auto', random_state=None, subsample=1.0, verbose=0, warm_start=False	0.395
NuSVC()	cache_size=200, class_weight=None, coef0=0.0, decision_function_shape=None, degree=3, gamma='auto', kernel='rbf', max_iter=-1, nu=0.5, probability=False, random_state=None, shrinking=True, tol=0.001, verbose=False	0.445
LinearDiscriminantAnalysis()	n_components=None, priors=None, shrinkage=None, solver='svd', store_covariance=False, tol=0.0001	0.265
SGDClassifier()	alpha=0.0001, average=False, class_weight=None, epsilon=0.1, eta0=0.0, fit_intercept=True, l1_ratio=0.15, learning_rate='optimal', loss='hinge', n_iter=5, n_jobs=-1, penalty='l2', power_t=0.5, random_state=None, shuffle=True, verbose=0, warm_start=False	0.37
GaussianProcessClassifier()	copy_X_train=True, kernel=None, max_iter_predict=100, multi_class='one_vs_rest', n_jobs=-1, n_restarts_optimizer=0, optimizer='fmin_l_bfgs_b', random_state=None, warm_start=False	0.44
DecisionTreeClassifier()	class_weight=None, criterion='gini', max_depth=6, max_features=None, max_leaf_nodes=None, min_impurity_split=1e-07, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, presort=False, random_state=None, splitter='best'	0.205
RandomForestClassifier()	bootstrap=True, class_weight=None, criterion='gini', max_depth=None, max_features='auto', max_leaf_nodes=None, min_impurity_split=1e-07, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, n_estimators=1000, n_jobs=-1, oob_score=False,random_state=None, verbose=0, warm_start=False	0.495
MLPClassifier()	activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9, beta_2=0.999, early_stopping=False, epsilon=1e-08,hidden_layer_sizes=(100,), learning_rate='constant', learning_rate_init=0.001, max_iter=200, momentum=0.9, nesterovs_momentum=True, power_t=0.5, random_state=None, shuffle=True, solver='adam', tol=0.0001, validation_fraction=0.1, verbose=False, warm_start=False	0.51
AdaBoostClassifier()	algorithm='SAMME.R', base_estimator=None, learning_rate=1.0, n_estimators=50, random_state=None	0.23
GaussianNB()	priors=None	0.25
NearestCentroid()	metric='euclidean', shrink_threshold=None	0.41

Voting models

<table>
<tr><th>Voting model</th><th>Accuracy</th><th>F1-score per species</th></tr>
<tr><td>All 13, equal weight</td><td>0.45</td><td>
<table>
<tr><th>Species</th><th>f1-score</th></tr>
<tr><td>Blue Whale</td><td>0.61</td></tr>
<tr><td>Chihuahua</td><td>0.38</td></tr>
<tr><td>Chimpanzee</td><td>0.48</td></tr>
<tr><td>Fox</td><td>0.33</td></tr>
<tr><td>Gorilla</td><td>0.23</td></tr>
<tr><td>Killer Whale</td><td>0.31</td></tr>
<tr><td>Seal</td><td>0.29</td></tr>
<tr><td>Tiger</td><td>0.65</td></tr>
<tr><td>Wolf</td><td>0.36</td></tr>
<tr><td>Zebra</td><td>0.74</td></tr>
<tr><td>Average</td><td>0.44</td></tr>
</table>
</td></tr>
<tr><td>0.4 accuracy threshold (6 remaining classifiers), equal weight</td><td>0.46</td><td>
<table>
<tr><th>Species</th><th>f1-score</th></tr>
<tr><td>Blue Whale</td><td>0.63</td></tr>
<tr><td>Chiuaua</td><td>0.40</td></tr>
<tr><td>Chimpanzee</td><td>0.55</td></tr>
<tr><td>Fox</td><td>0.33</td></tr>
<tr><td>Gorilla</td><td>0.40</td></tr>
<tr><td>Killer Whale</td><td>0.33</td></tr>
<tr><td>Seal</td><td>0.29</td></tr>
<tr><td>Tiger</td><td>0.63</td></tr>
<tr><td>Wolf</td><td>0.29</td></tr>
<tr><td>Zebra</td><td>0.72</td></tr>
<tr><td>Average</td><td>0.46</td></tr>
</table>
</td></tr>
<tr><td>All 13, extra iteration adding opinion of best classifier per species (based on f1-score), with the original all 13, equal weight voting model as starting point to further improve.</td><td>0.55</td><td>
<table>
<tr><th>Species</th><th>f1-score</th></tr>
<tr><td>Blue Whale</td><td>0.67</td></tr>
<tr><td>Chiuaua</td><td>0.47</td></tr>
<tr><td>Chimpanzee</td><td>0.64</td></tr>
<tr><td>Fox</td><td>0.38</td></tr>
<tr><td>Gorilla</td><td>0.39</td></tr>
<tr><td>Killer Whale</td><td>0.53</td></tr>
<tr><td>Seal</td><td>0.39</td></tr>
<tr><td>Tiger</td><td>0.77</td></tr>
<tr><td>Wolf</td><td>0.37</td></tr>
<tr><td>Zebra</td><td>0.81</td></tr>
<tr><td>Average</td><td>0.54</td></tr>
</table>
</td></tr>
<tr><td>0.4 accuracy threshold (6 remaining classifiers), extra iteration adding opinion of best classifier per species (based on f1-score), 0.4 threshold equal weight voting model as starting point to further improve.</td><td>0.535</td><td>
<table>
<tr><th>Species</th><th>f1-score</th></tr>
<tr><td>Blue Whale</td><td>0.65</td></tr>
<tr><td>Chiuaua</td><td>0.43</td></tr>
<tr><td>Chimpanzee</td><td>0.62</td></tr>
<tr><td>Fox</td><td>0.39</td></tr>
<tr><td>Gorilla</td><td>0.41</td></tr>
<tr><td>Killer Whale</td><td>0.36</td></tr>
<tr><td>Seal</td><td>0.44</td></tr>
<tr><td>Tiger</td><td>0.78</td></tr>
<tr><td>Wolf</td><td>0.41</td></tr>
<tr><td>Zebra</td><td>0.78</td></tr>
<tr><td>Average</td><td>0.53</td></tr>
</table>
</td></tr>
</table>

SIFT - KNN hyper parameter tuning

Baseline hyper parameter values

These parameters are used unless deviations are mentioned

```
# select features to combine
USE_SIFT = True
USE_RGB = False

# sift point types
SIFT_USE_DENSE_POINTS = True
SIFT_USE_HARRIS_POINTS = True
SIFT_USE_HESSIAN_POINTS = True

# sift point parameters
SIFT_DENSE_POINT_STRIDE = 25
SIFT_HARRIS_POINT_SIGMA = 1.0
SIFT_HARRIS_POINT_MAG_THRESHOLD = 15
SIFT_HARRIS_POINT_HES_THRESHOLD = 10
SIFT_HARRIS_POINT_NSM_NEIGHBORHOOD = 10
SIFT_HESSIAN_POINT_SIGMA = 1.0
SIFT_HESSIAN_POINT_MAG_THRESHOLD = 5
SIFT_HESSIAN_POINT_NSM_NEIGHBORHOOD = 10

# rgb features
RGB_PATCH_SIZE = (3, 3)

# code book generation
CODE_BOOK_KMEANS_CLUSTERS = 100
CODE_BOOK_KMEANS_USE_MINI_BATCH = True

# histogram
HISTOGRAM_IS_BINARY = False

# k nearest neighbor classifier
CLASSIFIER_KNN_NEIGHBORS = 10
CLASSIFIER_KNN_WEIGHTS = 'uniform'
```

Codebook: using standard k-means vs. k-means mini batch

Question: How fast are the different clustering techniques?

No. Clusters	K-Means	K-Means Mini Batch (batch size = 100)
100	7.69	0.76
200	11.60	1.32

Does the clustering method have a noticeable impact on the accuracy?

K-Means	K-Means Mini Batch (100)
0.355	0.380

CODE_BOOK_KMEANS_CLUSTERS = 100

Conclusion: the k-means mini batch is an order of magnitude faster and delivers a better result.

Contribution of SIFT point type to accuracy

Do certain point types have a much higher contribution to the accuracy?

SIFT point type	Dense	Harris	Hessian
Accuracy	0.295	0.345	0.310

CODE_BOOK_KMEANS_CLUSTERS = 100
SIFT_DENSE_POINT_STRIDE = 25

Conclusion: all three SIFT point types have similar contributions to the overall accuracy.

Effect of dense point stride on accuracy

A smaller stride means more dense points features but do they contribute to a higher accuracy?

Dense Point Stride	10	15	20	25	30
Accuracy	0.400	0.340	0.295	0.305	0.250

CODE_BOOK_KMEANS_CLUSTERS = 100
SIFT_USE_HARRIS_POINTS = False
SIFT_USE_HESSIAN_POINTS = False

Conclusion: Although smaller strides yield better accuracy, they are CPU/memory heavy. The stride of 15 seems like a good compromise.

Note: a stride of 5 results in a training features pickles file of 16.25GB! Stride 15 in 1.82GB.

Effect of codebook random initialization of clusters on accuracy

K-means is sensitive to the initialization, what is the effect on accuracy?

Try	#1	#2	#3	#4	#5
Accuracy	0.360	0.365	0.380	0.350	0.350

CODE_BOOK_KMEANS_CLUSTERS = 100

Conclusion: there is variability due to initialization. Accuracy standard deviation is 0.011.

Effect of no. of neighbors in kNN classifier

No. neighbors	1	2	5	10	20	50
Accuracy	0.295	0.31	0.375	0.375	0.35	0.375

SIFT_DENSE_POINT_STRIDE = 15
CODE_BOOK_KMEANS_CLUSTERS = 70

Conclusion: 5-10 neighbors seems sufficient

Effect of weight function in kNN classifier

Weight function	uniform	distances
Accuracy	0.375	0.35

SIFT_DENSE_POINT_STRIDE = 15
CODE_BOOK_KMEANS_CLUSTERS = 70
CLASSIFIER_KNN_NEIGHBORS = 5

Conclusion: to use the inverted distance as weight does not increase model accuracy

RGB Features

We can combine SIFT descriptors with RGB values for each SIFT keypoint with different patch sizes. Due to the extra information we might need larger k-means cluster size so this is varied as well.

cluster/patch size	1x1	3x3	5x5
100	0.395	0.375	0.375
1000	0.340	0.340	0.310

SIFT_DENSE_POINT_STRIDE = 25
CLASSIFIER_KNN_NEIGHBORS = 10

Conclusion: the addition of the RGB information increased the accuracy slightly from 0.375 to 0.395.

Binary Histogram

Literature suggests that a binary histogram may outperform a normalized histogram. In a binary histogram each assigned cluster will have the value of one and clusters without assignments have value zero.

Is binary / cluster size	100	500	1000
FALSE	0.37	0.37	0.37
TRUE	0.35	0.35	0.35

# select features to combine
USE_SIFT = True
USE_RGB = True

# rgb features
RGB_PATCH_SIZE = (3, 3)

# k nearest neighbor classifier
CLASSIFIER_KNN_NEIGHBORS = 1

Conclusion: Binary histograms result in a lower accuracy and will not be used.