DESIGN, IMPLEMENTATION AND EVALUATION

The whole code is written as functions. Each function is used for a specific purpose, and its use is further described below.

Data understanding

Firstly, it is important to understand what kind of data we want to feed into the machine learning algorithm and define the problem.

In the given task, we want to predict two different features texture and colour. Each of these features further contains multiple classes. Therefore, the problem was defined as two multiclass classifications.

The data set contains information about images where objects are annotated in terms of bounding boxes. The features in the data set are image and object ID, position and size of the bounding box. Further, it contains histograms of oriented gradients (HOG) that are widely used in computer vision and image processing as a feature descriptor usually used for object detection. Oriented Gabor filters used for texture analysis and the CIELAB colour space express colour as three values.

Data analysis and cleaning

When it is understood what the data represent, they must be thoroughly analysed visually as well as using functions for analysis. Function loadData() loads the given training data set and creates a data analysis csv document. This document contains information such as mean, counts for each feature, minimal and maximal values. When checking for NaN values, it was noticed that row 619 is missing almost all values; thus, loadData() drops this row.

	lightness_0_0	lightness_0_1	lightness_0_2	lightness_1_0
lightness_0_0	1.000000	0.793679	0.641806	0.794227
lightness_0_1	0.793679	1.000000	0.818616	0.747392
lightness_0_2	0.641806	0.818616	1.000000	0.619202
lightness_1_0	0.794227	0.747392	0.619202	1.000000
lightness_1_1	0.624346	0.805875	0.689668	0.773649
lightness_1_2	0.581439	0.730773	0.828271	0.647495
lightness_2_0	0.697669	0.616031	0.557381	0.788735
lightness_2_1	0.631309	0.717198	0.663116	0.736021
lightness_2_2	0.576873	0.651360	0.714766	0.603599
redgreen_0_0	1.000000	0.793679	0.641806	0.794227
redgreen_0_1	0.793679	1.000000	0.818616	0.747392
redgreen_0_2	0.641806	0.818616	1.000000	0.619202

Figure 1: Colour space Pearson correlation coefficient

The data are then further inspected with dataInspection() function. This function inspects correlations between data. It was found that three colour space values are just copies of each other, and its Pearson correlation coefficient is one (figure 1). It prints a

number of different classes that are in the colour and text feature. Other features were also checked for correlation, but nothing significant was found. It was also discovered that the data set is imbalanced, which means that some of the classes are less represented than others.

Function make_hist() creates histograms that describe how the data are distributed. In this case, the data tend to take the shape of normal distribution.

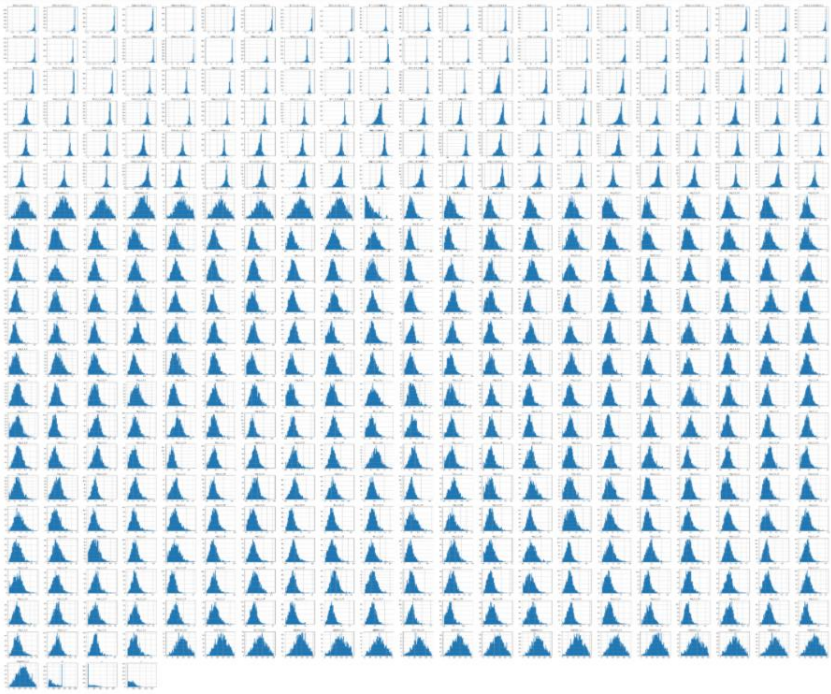

Figure 2: Histograms of the features

In response to the data analysis, there is cleanData() function that takes in panda's data frame and returns a cleaned one. This function calls another function that checks for duplicates in training data, and if there are any, it drops them. Then it drops columns that were deduced that are not relevant for predictions. These columns are the dimensions of the bounding box and IDs of the object and image. Because of the found correlation in colour space, two of the three colour values were dropped since they do not bring new information to predictions. Many zeros were found in the HOG data, it was considered imputing them with mean values, but later during the training, there was no observed improvement in the accuracy.

Dealing with imbalanced data

In the analysis part, it was discovered that the data set is imbalanced. There are few options for how to deal with such data. Here are presented three options:

1. Under-sampling – The main point of under-sampling the data is to have the same number of samples in the majority class as in the minority class; thus, under-sampling randomly deletes samples from the majority class. The program contains a manually created function underSample() for custom under-sampling. While building

the program, it was experimented with this function, but it did not bring any improvements.

2. Over-sampling Using SMOTE technique – It has the same point as under-sampling, but unlike under-sampling, it uses the k-neighbours algorithm to generate more samples and add them to minority class and even the counts of the majority class. It is implemented in the program as overSampleSMOTE() function.

3. Balancing weights in the training model. This assigns different weights to minority and majority class so that they are balanced. In the end, this option was selected since it generated the best results—more about this in the training part.

Pre-processing the data

There is a function called preprocessor() that creates a pipeline for pre-processing the data.

It starts with imputing the missing values. There are no missing data in the training set since the row with missing values were dropped, but if the option with replacing the zeros with NaN in HOG was chosen, it would impute the missing values with mean. Imputation also comes in handy while predicting when we feed some test data with missing values; the algorithm takes the mean from training data and tries its best prediction.

Another component in the pipeline is the standard scaler. It is important to scale the data so that all the features have a similar scale and decrease the bias. When examining the data, it could be observed that it tends to have a normal distribution, so when the data are scaled by standardisation, it takes properties of normal distribution ($\mu = 0$, $\sigma = 1$). Unlike min-max scaling, it does not have specific bound values, but standardisation is much less affected by outliers, which may be an important feature in this case because outliers have not been explicitly excluded. However, a min-max scaler was also tested, and the results on training were not significantly different. The main point here is to have data on a similar scale.

The last component of the pipeline is the feature selector. It uses ExtraTreesClassifier (ensemble learning – in this case it means connecting many decision trees together) that fits randomised decision trees on different sub-samples and uses averaging to improve predictions which produces feature importance which is impurity-based feature importance. This is then fed into the SelectFromModel(), which is used to select only features with high importance weight and discards irrelevant features. This method speeds up the training since only important features are consider, and it can improve accuracy as well. The alone featureSelection() function is also implemented.

In the program, there are two functions for creating a pipeline (pre-processor), one with and the other without feature selection. For training, the model feature selection was used.

Training and model selection

Support Vector Machine was used as the main model, a powerful and versatile Machine Learning model that can perform either linear or non-linear classification (depending on the kernel). It is especially well suited for the classification of complex but small or medium-sized datasets, and it is effective in high dimensional space. The main aim of the SVM is to fit the line between the data so that it maximises the margin between them defined by support vectors (instances located on the margin). This improves the generalisation performance of the classifier. Some data are not directly linearly separable; thus, SVM implements a soft margin that allows margin violations. The C hyperparameter controls this; a smaller C means a wider margin but more violations. SVM works great in combination with the kernel trick that prevents combinatorial explosion of the number of features. This is because kernels implicitly transform the input data into a higher-dimensional space where a linear separator may exist, even if the original data are non-separable. The kernel trick is implemented in scikit-learn's SVC() estimator.

Non-linear classification with Gaussian Radial Basis Function (RBF) as a kernel was used because it performed the best in comparison to other available kernels that were all tested on the basic SVC() model. With the RBF kernel, there is another parameter gamma, that must be considered. This parameter describes how much influence a single training example has; the larger it is, the closer other examples must be to be affected.

As it was mentioned before, the dataset is imbalanced, and there were named three options. The third option was balancing the weights of the classes, which is done through the parameter class_weight.

It was said that this problem is multiclass classification. There are estimators (models) that are inherently multiclass classification, such as MLPClassifier (neural networks), but SVM is not. SVM can implement techniques such as One-vs-One and One-vs-The-Rest methods to deal with such classification. It is internally implemented in SVC() model defined by the parameter decision_funtion_shape. In this case, one-vs-the-rest was chosen. It creates a classifier for each class against the rest of the data; also break_ties parameter is set to true; thus, it will break ties between classes according to the confidence values of the decision function.

For hyperparameter tuning, the grid search technique was used to find the most appropriate parameters (C and gamma) (GridSearchCV). Unlike a random search, this allows picking the exact parameters to try, which tries all the combinations of given parameters. This is implemented in the hyperparameterTun() function, which prints the best parameters when it is done.

The parameters from grid search were then used in the function trainSVMmodel() that is used for training separately for color and texture. When the model is trained, this function saves it into the models folder. After the training is done, it calls another function called TrainEvaluation(). This function then prints out accuracy and balanced accuracy that is used to describe accuracy when the data are imbalanced.

It further prints precision and recalls for each class and displays a confusion matrix. This all is for the training set. It also performs cross-validation to simulate the test set and prints its accuracy and balanced accuracy for five folds. To keep the balanced representation of classes in each fold, the StratifiedKFold was used. Cross-validation is useful to simulate how the model could perform on the test set and check if its predictions are consistent (so they are not just random predictions).

When the models for color and texture are trained, they can predict the test set. For this the function testPredictions() is used. It loads the test data and models and then internally calls predictTest() function that performs predicting and saving the results to the csv format.

Evaluation

All of the results presented below are for the SVM model for color class.

At first, the prediction was made on the basic SVM model without considering imbalanced data. Its performance can be seen in figure 3 on the training set and its confusion matrix in figure 4. It is possible to see that due to imbalanced data, the balanced score is low in comparison to classical accuracy. Some of the classes were not even predicted once.

```
Accuracy on training set: 0.7475655430711611
Balanced accuracy on training set: 0.3548223711910925
              precision    recall  f1-score   support

       beige       0.00      0.00      0.00         6
       black       1.00      0.74      0.85        73
        blue       1.00      0.22      0.36        87
       brown       0.81      0.67      0.73       218
        gray       1.00      0.33      0.49       101
       green       0.63      0.94      0.76       297
      orange       0.00      0.00      0.00        18
        pink       0.00      0.00      0.00        17
      purple       0.00      0.00      0.00         6
         red       1.00      0.25      0.40        16
       white       0.76      0.97      0.85       476
      yellow       1.00      0.15      0.26        20
```

Figure 3: Predicting on the training set (imbalanced data not considered)

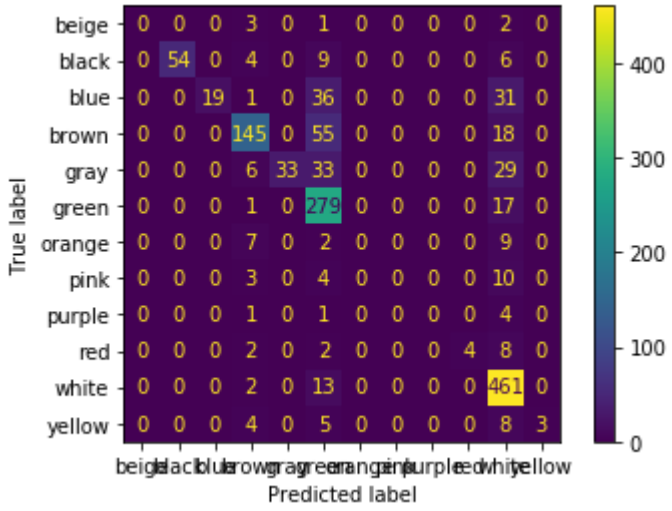

Figure 4: Confusion matrix - Predicting on the training set (imbalanced data not considered)

To simulate scores on testing the cross-validation was used (figure 5). When there is not enough data, it is good to use cross-validation instead of just splitting data into the training set and validation set.

```
Accuracy score on cross validation: [0.50187266 0.50561798 0.47565543 0.51310861 0.49812734]
Balanced accuracy score on cross validation: [0.15314519 0.15216133 0.14356085 0.15639976 0.15081089]
```

Figure 5: Cross-validation results (imbalanced data not considered)

To deal with imbalanced data, the class_weight parameter was set to balanced as explained in the training part. Balancing the weights helped the model improve its performance on the training set and cross-validation—results presented in figure 6, 7 and 8.

```
Accuracy on training set: 0.7325842696629213
Balanced accuracy on training set: 0.8778941696789074
              precision    recall  f1-score   support

       beige       1.00      1.00      1.00         6
       black       0.66      0.95      0.78        73
        blue       0.46      0.83      0.60        87
       brown       0.66      0.57      0.61       218
        gray       0.67      0.79      0.72       101
       green       0.68      0.65      0.66       297
      orange       1.00      1.00      1.00        18
        pink       1.00      1.00      1.00        17
      purple       1.00      1.00      1.00         6
         red       1.00      1.00      1.00        16
       white       0.89      0.75      0.82       476
      yellow       1.00      1.00      1.00        20
```

Figure 6: Predicting on the training set (imbalanced data considered)

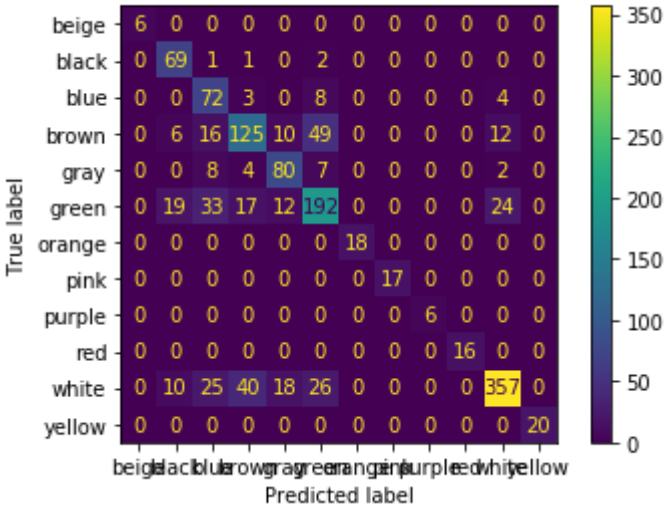

Figure 7: Confusion matrix - Predicting on the training set (imbalanced data considered)

```
Accuracy score on cross validation: [0.48314607 0.44194757 0.42696629 0.45318352 0.43820225]
Balanced accuracy score on cross validation: [0.23477851 0.1837239  0.18386113 0.20315028 0.18273254]
```

Figure 8: Cross-validation results (imbalanced data considered)

It can be concluded that the model is overfitting because it did not perform well on a cross-validation set that should simulate generalising. The fact that the model is overfitting means it basically learnt the model by heart since it is performing quite well on the instances that it already seen in the training set.

After hyperparameter tuning, the model got a maximal score on cross-validation presented in figure 9.

```
Accuracy score on cross validation: [0.49438202 0.46816479 0.48689139 0.47940075 0.48120301]
Balanced accuracy score on cross validation: [0.21244808 0.19666885 0.19480805 0.19919015 0.18021656]
```

Figure 9: Cross-validation results after hyperparameter tunning

The same approach was applied when training for predicting the texture class.

```
Accuracy score on cross validation: [0.39325843 0.39700375 0.3670412  0.34456929 0.41353383]
Balanced accuracy score on cross validation: [0.24588359 0.27495561 0.26421144 0.29566375 0.30399257]
```

Figure 10: Cross-validation results after hyperparameter tunning for texture class

In conclusion, the trained SVM models performed 20.52% on color and 41.324% on texture on the test set. To improve the performance further, it would need to be supplied with more data balanced as much as possible. Other than that, ensemble learning could be tested. It means to train more algorithms and then create something like a voting system to predict the class with the most votes.

Comparison with logistic regression

As mentioned, SVM tries to find the widest possible margin, while Logistic regression optimises the log-likelihood function, with probabilities modelled by the sigmoid function. Logistic regression is here used as a linear model for multiclass classification using the One-vs-The-Rest technique, and unlike SVM, it is not using kernels. The class_weight in the model is set to balanced, so it is balancing the weights for the imbalanced classes. Nevertheless, as can be seen in figure 11 and 12, it performs worse than SVM.

```
Accuracy score on cross validation: [0.33333333 0.37453184 0.34831461 0.35580524 0.33082707]
Balanced accuracy score on cross validation: [0.18076605 0.16297053 0.14184413 0.17548189 0.12381091]
```

Figure 11: Logistic regression - Cross-validation results – color class

```
Accuracy score on cross validation: [0.3670412  0.35580524 0.38202247 0.32209738 0.41353383]
Balanced accuracy score on cross validation: [0.20629794 0.21545867 0.21703378 0.18206823 0.2318608 ]
```

Figure 12: Logistic regression - Cross-validation results – texture class

Random Forest Classifier

Decision trees are SVM universal Machine Learning algorithms that can perform both classification and regression. Decision trees can perform multiclass classification by nature, so they do not need to do anything like one vs the rest technique. They make very few assumptions about the data and can fit even complex datasets; thus, they are prone to overfitting. To avoid that, there must be set proper regularisation such as

reducing the maximal depth of the tree, the minimum number of samples a node must have before it can be split, the minimum number of samples a leaf node must have, the maximum number of leaf nodes and some others.

To achieve more precise predictions, ensemble learning comes in handy. RandomForestClassifier puts together a number of decision tree estimators and predicts according to the most votes for the given class. Random Forests are very powerful and, together with neural networks, are considered black-box models, making great predictions, but it is hard to tell why they did them. The trees inside are each trained on a different random subset of the training data, which is usually done by bagging method, which means when the sample is picked to the subset, it is then returned into the whole subset, that is sampling with replacement.

In figure 13, there is a score of unregularised Random Forest. It can be seen that it overfits the data.

```
Accuracy on training set: 1.0
Balanced accuracy on training set: 1.0
              precision    recall  f1-score   support

       beige       1.00      1.00      1.00         6
       black       1.00      1.00      1.00        73
        blue       1.00      1.00      1.00        87
       brown       1.00      1.00      1.00       218
        gray       1.00      1.00      1.00       101
       green       1.00      1.00      1.00       296
      orange       1.00      1.00      1.00        18
        pink       1.00      1.00      1.00        17
      purple       1.00      1.00      1.00         6
         red       1.00      1.00      1.00        16
       white       1.00      1.00      1.00       476
      yellow       1.00      1.00      1.00        20
```

Figure 13: Random forest without regularisation (with no parameters on the training set)

```
Accuracy score on cross validation: [0.56554307 0.51310861 0.49438202 0.51685393 0.5112782 ]
Balanced accuracy score on cross validation: [0.21020547 0.17392542 0.15458831 0.16444755 0.16598157]
```

Figure 14: Random forest without regularisation Cross-validation results

In the program it is possible to train RandomForestClassifier by calling trainRandForestModel() function. Further, there was done hyperparameter tunning that achieved subsequent results.

```
Accuracy score on cross validation: [0.51685393 0.52059925 0.52059925 0.52059925 0.5       ]
Balanced accuracy score on cross validation: [0.19566477 0.17248475 0.17993589 0.17043089 0.15118069]
```

Figure 15: Random forest with regularisation

To achieve better results, further search for better hyperparameters must be done.