Image Classification Report

1. Dataset analysis

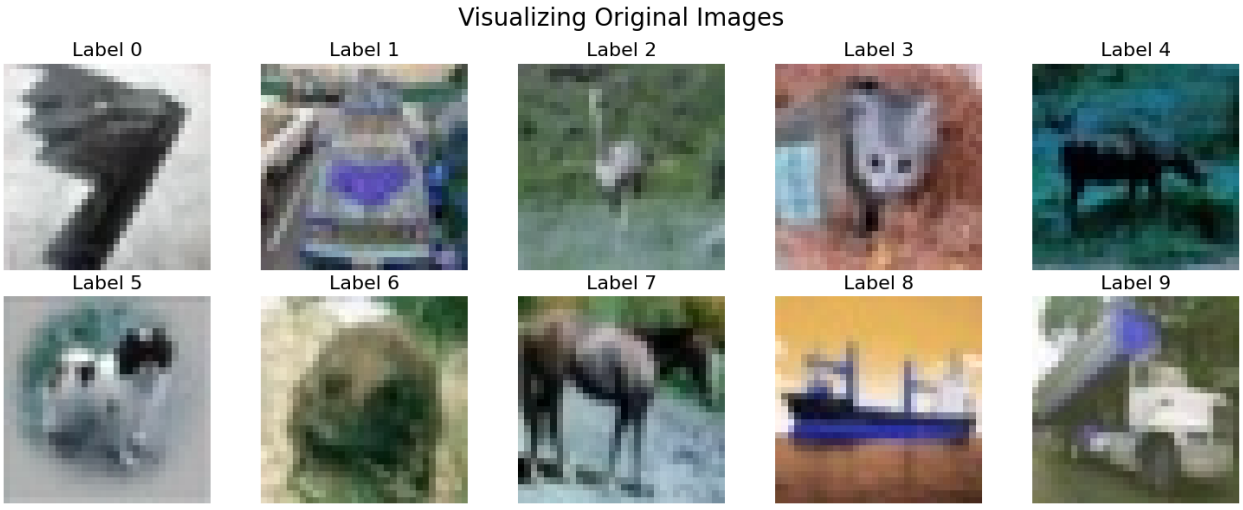
1.1 Statistics on the number of categories

The training dataset consists of 10 labels, each containing approximately 5,000 images. As the dataset is balanced, I do not need to apply any techniques to balance the dataset across the labels.

The number of images and category for each label is shown in the table below:

Label	Number of Images	Category
0	5010	Plane
1	5012	Car
2	5038	Bird
3	5007	Cat
4	4995	Deer
5	4993	Dog
6	4955	Frog
7	5000	Horse
8	5020	Ship
9	4970	Truck

1.2 Visualizations of one example for each category



2. Classifier Exploration

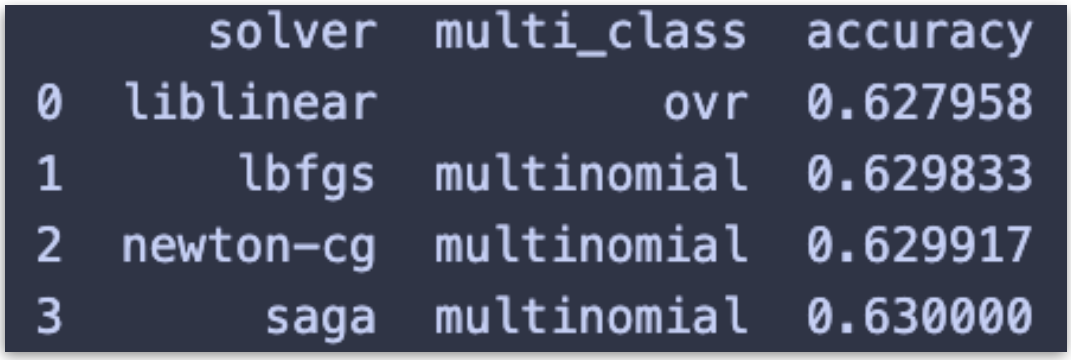
I employed three classifiers for this experiment: **K-Nearest Neighbours (KNN)**, **Logistic Regression**, and **Support Vector Machine (SVM)**. To ensure efficiency and consistency, *20% of the training dataset* was allocated for training and validation across all models. The preprocessing steps were standardised and uniformly applied to all classifiers, ensuring a level playing field with no model benefiting from discrepancies in data preparation. Furthermore, hyperparameter tuning was conducted for each classifier to fine-tune their respective parameters, maximising their performance.

2.1 Linear Regression:

I began by tuning the hyperparameter "C" to determine the optimal regularisation strength, testing values within the range [0.001, 0.01, 0.1, 1, 10, 100]. After thorough evaluation, the best value of "C" was identified as 0.001.

Building on this, I optimised additional hyperparameters, such as the solver, by systematically comparing liblinear, lbfgs, newton-cg, and saga using GridSearchCV for comprehensive exploration. For the multi-class classification strategy, I adopted the "ovr" (One-vs-Rest) approach with the liblinear solver, while the "multinomial" strategy was applied for the remaining solvers, ensuring tailored optimisation for each configuration.

- ***C: [0.001, 0.01, 0.1, 1, 10, 100] -> 0.001 is the best***
- ***Solver: liblinear, lbfgs, newton-cg, saga***
- ***Multi_class: ovr, liblinear***



	solver	multi_class	accuracy
0	liblinear	ovr	0.627958
1	lbfgs	multinomial	0.629833
2	newton-cg	multinomial	0.629917
3	saga	multinomial	0.630000

The comparison of various parameter combinations

We can observe that using **saga** as the solver and **multinomial** as the multi-class strategy yields the highest accuracy, achieving a value of 0.63.

2.2 KNN:

I carried out a hyperparameter tuning process using **GridSearchCV** to identify the optimal configuration for maximising accuracy. The parameter grid included variations in the number of neighbours (n_neighbors) ranging from 2 to 30 in increments of 3, weighting methods (uniform and distance), and distance metrics (p=1 for Manhattan distance and p=2 for Euclidean distance).

GridSearchCV was implemented with 5-fold cross-validation to evaluate each parameter combination, with the mean cross-validation accuracy serving as the primary evaluation metric.

- **N-neighbours: range(2, 30 ,3)**
- **P: [1 (Manhattan distance), 2 (Euclidean distance)]**
- **Weights: ['uniform', 'distance']**

	param_n_neighbors	param_weights	param_p	mean_test_score	std_test_score
21	17	distance	1	0.554125	0.006387
17	14	distance	1	0.553833	0.008278
33	26	distance	1	0.553833	0.006085
25	20	distance	1	0.553167	0.005306
29	23	distance	1	0.552875	0.004847
28	23	uniform	1	0.550750	0.006696
16	14	uniform	1	0.550417	0.005988
13	11	distance	1	0.550208	0.007828
32	26	uniform	1	0.550083	0.007263
24	20	uniform	1	0.549958	0.005124
37	29	distance	1	0.549500	0.008291
20	17	uniform	1	0.549125	0.005974
9	8	distance	1	0.548292	0.007824
36	29	uniform	1	0.547375	0.008235
12	11	uniform	1	0.546750	0.008683
8	8	uniform	1	0.543833	0.008042
5	5	distance	1	0.533792	0.008954
4	5	uniform	1	0.531542	0.012622
15	11	distance	2	0.530292	0.006262
19	14	distance	2	0.528167	0.009303
27	20	distance	2	0.527125	0.007935
23	17	distance	2	0.527042	0.009750
14	11	uniform	2	0.526917	0.008081
22	17	uniform	2	0.525500	0.010185
18	14	uniform	2	0.525417	0.010427
31	23	distance	2	0.524417	0.007801
11	8	distance	2	0.524333	0.006882
26	20	uniform	2	0.524083	0.008716
30	23	uniform	2	0.521458	0.007220
35	26	distance	2	0.521417	0.005425
10	8	uniform	2	0.521000	0.007357
39	29	distance	2	0.519792	0.005554
34	26	uniform	2	0.519083	0.006737
38	29	uniform	2	0.518250	0.004477
1	2	distance	1	0.513333	0.006153
7	5	distance	2	0.510917	0.009278
6	5	uniform	2	0.508250	0.009628
0	2	uniform	1	0.503250	0.008332

Comparison of various parameter combinations (Sorted in Accuracy)

```
Best parameters: {'n_neighbors': 17, 'p': 1, 'weights': 'distance'}
Best cross-validation accuracy: 0.554125
```

2.3 SVM:

I utilized GridSearchCV to optimize hyperparameters for different kernels. For the linear kernel, the regularization parameter C was tuned across the range [0.001, 0.01, 0.1, 1, 10]. For the RBF kernel, I explored various combinations of C and gamma. Each configuration was thoroughly evaluated using 3-fold cross-validation to ensure reliable and robust performance assessment.

- **C: [0.001, 0.01, 0.1, 1, 10]**
- **Kernal: [linear, rbf]**
- **Gamma: ['scale', 'auto', 0.01, 0.1, 1]**

Comparisons for linear kernel:

	param_C	mean_test_score	std_test_score
0	0.001	0.624708	0.003241
1	0.010	0.613500	0.003457
2	0.100	0.592583	0.007938
3	1.000	0.582000	0.003309
4	10.000	0.580583	0.004456

Results of SVM model with linear kernel sorted in accuracy

Comparisons for rbf kernel:

	param_C	param_gamma	mean_test_score	std_test_score
20	10.000	scale	0.680417	0.000880
21	10.000	auto	0.680375	0.000919
15	1.000	scale	0.676458	0.005789
16	1.000	auto	0.676417	0.005847
10	0.100	scale	0.573542	0.010978
11	0.100	auto	0.573458	0.011116
6	0.010	auto	0.283958	0.004757
5	0.010	scale	0.283708	0.004556
22	10.000	0.01	0.110792	0.001192
17	1.000	0.01	0.110250	0.001159
23	10.000	0.1	0.104917	0.000156

Results of SVM model with rbf kernel sorted in accuracy

After analysing the results of the SVM model using both the linear and RBF kernels, it appears that the model performs better with C = 1/10. Based on this observation, I will narrow the focus to the range [1, 10] for C when optimising the SVM model with the RBF kernel.

I observed that C = 2, 3, and 4 deliver better performance compared to the other values tested.

Comparisons for rbf kernel:

	param_C	param_gamma	mean_test_score	std_test_score
7	4	auto	0.683375	0.001472
6	4	scale	0.683333	0.001332
4	3	scale	0.682708	0.001032
5	3	auto	0.682625	0.001150
2	2	scale	0.682542	0.001532
3	2	auto	0.682542	0.001378
9	5	auto	0.682542	0.000915
8	5	scale	0.682500	0.000890
10	6	scale	0.682292	0.000819
11	6	auto	0.682250	0.000736
15	8	auto	0.682250	0.000835
14	8	scale	0.682208	0.000926
12	7	scale	0.682208	0.000868
17	9	auto	0.682208	0.000825
19	10	auto	0.682208	0.000825
13	7	auto	0.682167	0.000831
16	9	scale	0.682167	0.000920
18	10	scale	0.682167	0.000920
0	1	scale	0.677625	0.003488
1	1	auto	0.677542	0.003422

Results of SVM model with rbf kernel sorted in accuracy (c = range(1,10))

Comparisons for rbf kernel:

	param_C	param_gamma	mean_test_score	std_test_score
5	4	auto	0.708521	0.004343
4	4	scale	0.708500	0.004352
3	3	auto	0.707813	0.003512
2	3	scale	0.707792	0.003577
1	2	auto	0.706125	0.003114
0	2	scale	0.706083	0.003103

Results of SVM model with rbf kernel sorted in accuracy (tested with 40% of training datasets)

The results of the SVM models with C = 2/ 3 /4 and gamma = scale/auto were similar, making it reasonable to select any of these configurations. The minor differences in accuracy are likely attributable to random variations or errors.

Ultimately, I chose to use the SVM model with the RBF kernel, C = 2.5, and gamma = scale for the final implementation.

2.4 Comparison

In the comparison of classifiers, the **SVM model** demonstrated the highest accuracy among the three models evaluated. Based on this result, the SVM was selected for further training using the full training dataset, with an 80%-20% train-test split. After training and testing on this split, the SVM achieved an **accuracy of 0.7837**, reaffirming its superior performance over the other classifiers. Consequently, I have chosen to use the **SVM model with the RBF kernel ,C = 2.5 & gamma = scale** as the final model for the Kaggle competition.

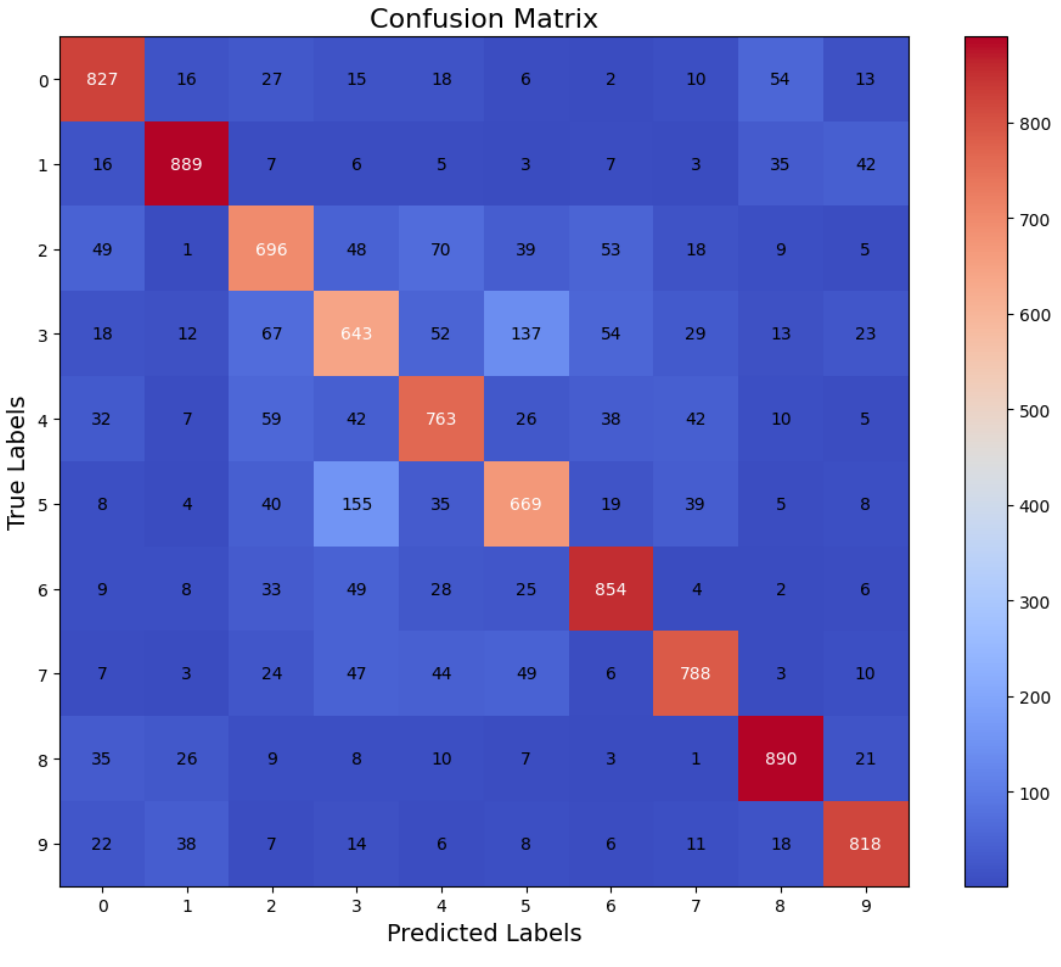
```
Test Accuracy: 0.7837
Train Accuracy: 0.989825
```

Testing Accuracy Result of Final model

```
Classification Report (Test Set):
              precision    recall  f1-score   support

     Class 0       0.81      0.84      0.82       988
     Class 1       0.89      0.88      0.88      1013
     Class 2       0.72      0.70      0.71       988
     Class 3       0.63      0.61      0.62      1048
     Class 4       0.74      0.75      0.74      1024
     Class 5       0.69      0.68      0.69       982
     Class 6       0.82      0.84      0.83      1018
     Class 7       0.83      0.80      0.82       981
     Class 8       0.86      0.88      0.87      1010
     Class 9       0.86      0.86      0.86       948

    accuracy                           0.78     10000
   macro avg       0.78      0.78      0.78     10000
weighted avg       0.78      0.78      0.78     10000
```



3 Final solution description

As previously mentioned, I implemented a series of preprocessing steps to prepare the image data before fitting it into the model. These processing steps are outlined below.

3.1 Apply Augmentation to the data

I apply data augmentation technique to expand the training dataset and introduce variations, thereby improving the robustness of the image classification model. The augmentation function includes transformations such as horizontal flipping, vertical flipping, combined flipping, random rotations, noise addition, and brightness adjustment.

For the final implementation, I prioritised **brightness adjustment** and **horizontal flipping**, as this combination achieved better accuracy. This approach tripled the dataset size, increasing it to 150,000 samples.

```
if "flip_horizontal" in actions:
    new_imgs.append(cv2.flip(img, 1))
    new_y.append(y[idx])
```

Code for the horizontal flipping

```
if "brightness" in actions:
    factor = np.random.uniform(0.8, 1.2)  # Smaller range for more natural changes
    offset = np.random.randint(-50, 50)  # Random offset for brightness adjustment
    brightness_img = cv2.convertScaleAbs(img, alpha=factor, beta=offset)
    new_imgs.append(brightness_img)
    new_y.append(y[idx])
```

Code for the brightness adjustment

3.2 Feature Extraction

After applying augmentation to the data, I implemented multiple feature extraction techniques for image classification tasks. Including Histogram of Oriented Gradients (HOG), Daisy descriptors, Edge Orientation Histogram (EOH), and average pooling.

- **HOG - extract texture and edge direction features from color images.**
- **Daisy - compute gradient information from grayscale images, capturing the texture in a dense and spatially distributed manner.**
- **EOH - calculates edge orientation histograms using gradient magnitudes and angles, dividing the image into cells and aggregating edge information across cells.**
- **Average pooling - reduces the image resolution while preserving essential spatial details, summarizing local pixel information.**

Among the four feature extractors, **HOG** demonstrated the most significant improvement in accuracy. However, the other feature extractors also contributed to enhancing the overall model performance. As a result, all four techniques were included in the feature extraction process.

To ensure consistency across the different feature types, the extracted feature vectors were normalized using **StandardScaler**.

3.3 Training and Results

Using the same parameter settings outlined in Section 2.4 for the SVM model, I generated the output file (test.csv) and submitted it to Kaggle for evaluation. The accuracy in Kaggle with 20% of the test data is 0.794.