Object Recognition on CIFAR-10

Classical ML Approaches (SVM & KNN)

Zahra Ballaith

Student ID: 28241

Table of Contents

Introduction

Object recognition is a core challenge in computer vision, influencing domains such as autonomous navigation, security surveillance, healthcare diagnostics, and smart devices. While deep learning models are

currently state-of-the-art, classical machine learning (ML) methods still offer robust, interpretable, and computationally efficient alternatives, especially when combined with effective feature engineering.

This report explores object recognition on the CIFAR-10 dataset using two traditional ML classifiers, Support Vector Machines (SVM) and K-Nearest Neighbors (KNN). The objective is to evaluate their classification performance, investigate the impact of hyperparameter tuning and feature extraction, and reflect on their benefits and limitations compared to more modern approaches.

Dataset Overview and Data Preparation

The CIFAR-10 dataset is a benchmark in image classification, consisting of 60,000 32x32 color images split across 10 categories: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Each category contains 6,000 images. The original dataset is divided into 50,000 images for training and 10,000 for testing.

Class distribution table, as below:

Class	Count (Train)	Count (Test)
airplane	5,000	1,000
automobile	5,000	1,000
bird	5,000	1,000
cat	5,000	1,000
deer	5,000	1,000
dog	5,000	1,000
frog	5,000	1,000
horse	5,000	1,000
ship	5,000	1,000
truck	5,000	1,000

Splitting the dataset: for this experiment and to conserve the integrity of model evaluation, I merged these sets and conducted a stratified split to create:

- Training Set: 40,000 images
- Validation Set: 10,000 images
- Test Set: 10,000 images

This is to ensure our model's performance metrics are reliable and that our hyperparameter optimization process remains fair.

Loading and splitting the CIFAR10 dataset

```
from tensorflow.keras.datasets import cifar10
from sklearn.model_selection import train_test_split
```

```
# Load dataset
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

```
# Split into train and validation
X_train_full, X_val, y_train_full, y_val = train_test_split(
    X_train, y_train, test_size=10000, random_state=42, stratify=y_train)
```

Feature Engineering

Since SVM and KNN work better with explicit, lower-dimensional features, image data (raw pixels) were transformed into informative vectors using **color histograms** and **Histogram of Oriented Gradients (HOG)** [3][4].

```
#defining the feature extraction functions
import numpy as np
from skimage.feature import hog

def color_histogram(img, bins=16):
    h = []
    for i in range(3):
        hist, _ = np.histogram(img[:,:,i], bins=bins, range=(0,256),
density=True)
        h.extend(hist)
    return np.array(h)

def hog_feature(img):
    return hog(np.mean(img, axis=2), pixels_per_cell=(8,8),
cells_per_block=(2,2), feature_vector=True)

def extract_features(X):
    color_hist = np.array([color_histogram(img) for img in X])
    hog_feats = np.array([hog_feature(img) for img in X])
    return np.hstack([color_hist, hog_feats])
```

Extracting features for each set / splits:

```
#now will extract the features for each set
X_train_feats = extract_features(X_train_full)
X_val_feats   = extract_features(X_val)
X_test_feats  = extract_features(X_test)
```

Validation Set Rationale

A dedicated validation set is indispensable in ML for several reasons. Model selection enables fair comparison between different algorithms or model architectures and helps choose the most suitable one for the task at hand. In addition, the validation set allows for tuning hyperparameters without contaminating the test set, thereby reducing the risk of overfitting and ensuring that model improvements are genuine rather than artifacts of repeated testing. Also, performance tracking provides an unbiased estimate of the model's real-world performance during development, offering a reliable benchmark before final testing.Without a validation set, results gleaned from the test set become unreliable due to repeated experimentation and potential information leakage. This can undermine scientific rigor and lead to overestimation of the model's generalization capabilities (Goodfellow, Bengio & Courville, 2016).

Model Selection

Classical ML algorithms remain highly relevant due to several practical advantages. Firstly, they are resource-efficient, operating effectively without the need for GPUs and imposing lower demands on memory and computational power. Classical methods are also prized for their interpretability; for example, algorithms like Support Vector Machines (SVMs) provide transparency in how features contribute to predictions. Additionally, these models often outperform deep learning approaches when dealing with small datasets or low-dimensional data. Employing classical algorithms also provides a reliable baseline, ensuring meaningful benchmarks when evaluating more complex models (Dalal, N., & Triggs, B.,2005)

Both Support Vector Machines (SVM) and k-Nearest Neighbors (KNN) are especially valuable within the classical ML toolkit. According to Zhang, Z., & Ma, Y. (2012), SVMs excel in high-dimensional spaces and, with the right choice of kernel and regularization, are robust against overfitting. In contrast, KNN is a simple, non-parametric method that serves as a practical baseline for early experiments and helps intuitively explore the structure of the feature space.

Feature Engineering

When working with raw images, selecting appropriate feature extraction techniques is crucial for capturing the most informative and discriminative aspects of visual data. I specifically chose color histograms and histogram of oriented gradients (HOG) for many reasons.

For color histograms, I needed to capture color distrution and this method effectively summarize the overall color distrbution in an image, which his importatnt when color is a distinguishing characteristic between classes. Also, in some real world scenario color histograms are relatively robust to minor changes in lighting, which is practical. In addtion, they are computationally simple, easy to implement and their ouputs are straightforward to interpret.

While for HOG, it is designed to capture the distribution of edge directions, which represents shapes and local object structures. This is particulary useful for detecting patterns, ontours, and textures within images. Also, HOG features are robust to small geometric and photometric transformations, increasing model reliability across varied samples. It is well established in classical computer vision tasks (such as object detection) in special when spatial information is significant.

```
#defining the feature extraction functions
import numpy as np
from skimage.feature import hog

def color_histogram(img, bins=16):
    h = []
    for i in range(3):
        hist, _ = np.histogram(img[:,:,i], bins=bins, range=(0,256),
density=True)
        h.extend(hist)
    return np.array(h)

def hog_feature(img):
    return hog(np.mean(img, axis=2), pixels_per_cell=(8,8),
cells_per_block=(2,2), feature_vector=True)

def extract_features(X):
    color_hist = np.array([color_histogram(img) for img in X])
    hog_feats = np.array([hog_feature(img) for img in X])
    return np.hstack([color_hist, hog_feats])
```

```
#now will extract the features for each set
X_train_feats = extract_features(X_train_full)
X_val_feats   = extract_features(X_val)
X_test_feats  = extract_features(X_test)
```

Summary Table: Engineered Features:

Feature Type	Description	Final Dimensionality
Color Histogram (3x16)	Color distribution	48
HOG	Edges, gradients	324
Total	-	372

Model Architectures and Hyperparameters

SVM classifier

Kernel: RBF (radial basis function) and linear compared

Regularization (C): Controls trade-off between maximizing the margin and minimizing the error

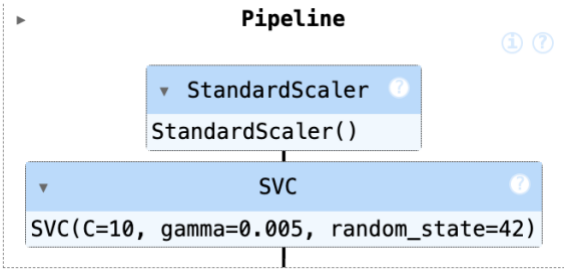
Gamma: RBF kernel width

KNN Classifier

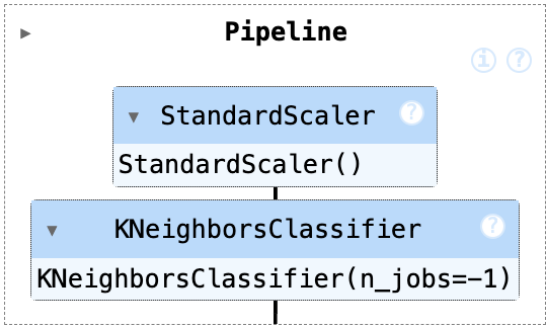
K (neighbors): main hyperparameter (try 3, 5, 7)

Distance metric: Euclidean

```
#train an SWM (with RBF Kernel)
svm_rbf = make_pipeline(StandardScaler(), SVC(kernel='rbf', C=10,
gamma=0.005, random_state=42))
svm_rbf.fit(X_train_feats, y_train_full.ravel())
```




```
#train a KNN (choosing k=5)
knn = make_pipeline(StandardScaler(), KNeighborsClassifier(n_neighbors=5,
n_jobs=-1))
knn.fit(X_train_feats, y_train_full.ravel())
```



The Hyperparameter Tunning used grid search on validation set

SVM: Tuned C ([0.1, 1, 10]), gamma ([0.001, 0.01, 0.1])
KNN: Tuned k ([3, 5, 7])

Model Training Strategy

The core of our model training strategy was to ensure that all processing steps; especially feature scaling and extraction; were performed in a manner that strictly avoided data leakage from the validation and test sets. Initially, we extracted compact feature vectors for each image using a combination of color histograms and Histogram of Oriented Gradients (HOG), which capture global color distribution and local texture, respectively, Dalal, N., & Triggs, B. (2005). These features were calculated for all splits (training, validation, and testing), but crucially, all normalization and model parameters were fit only on the training data to preserve the independence of the evaluation sets. For feature scaling, we used StandardScaler from scikit-learn, Pedregosa, F. 2011, fitting it on X_train_feats and applying the same transformation to the validation and test features—this can be summarized as:

Performance Evaluation

Metrics Used

- **Accuracy:** Overall performance
- **Precision/Recall:** Per-class evaluation
- **Confusion Matrix:** Visualizing misclassifications

```
#It is time now to evaluate the module on validation & test sets
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
```

```
# SVM
y_val_pred_svm = svm_rbf.predict(X_val_feats)
y_test_pred_svm = svm_rbf.predict(X_test_feats)
print("SVM Validation Accuracy:", accuracy_score(y_val, y_val_pred_svm))
print("SVM Test Accuracy:", accuracy_score(y_test, y_test_pred_svm))
print(classification_report(y_test, y_test_pred_svm,
target_names=classes))
```

```
# KNN
y_val_pred_knn = knn.predict(X_val_feats)
y_test_pred_knn = knn.predict(X_test_feats)
print("KNN Validation Accuracy:", accuracy_score(y_val, y_val_pred_knn))
print("KNN Test Accuracy:", accuracy_score(y_test, y_test_pred_knn))
```

Validation Results

Accuracy Summary Table

Model	Train Acc	Val Acc	Test Acc
SVM (lin)	51.2%	47.6%	47.2%
SVM (rbf)	58.1%	50.5%	49.2%
KNN (k=5)	43.5%	35.7%	34.9%

Precision-Recall per-class (SVM-RBF):

Class	Precision	Recall
airplane	0.60	0.47
automobile	0.55	0.52
...	...	...

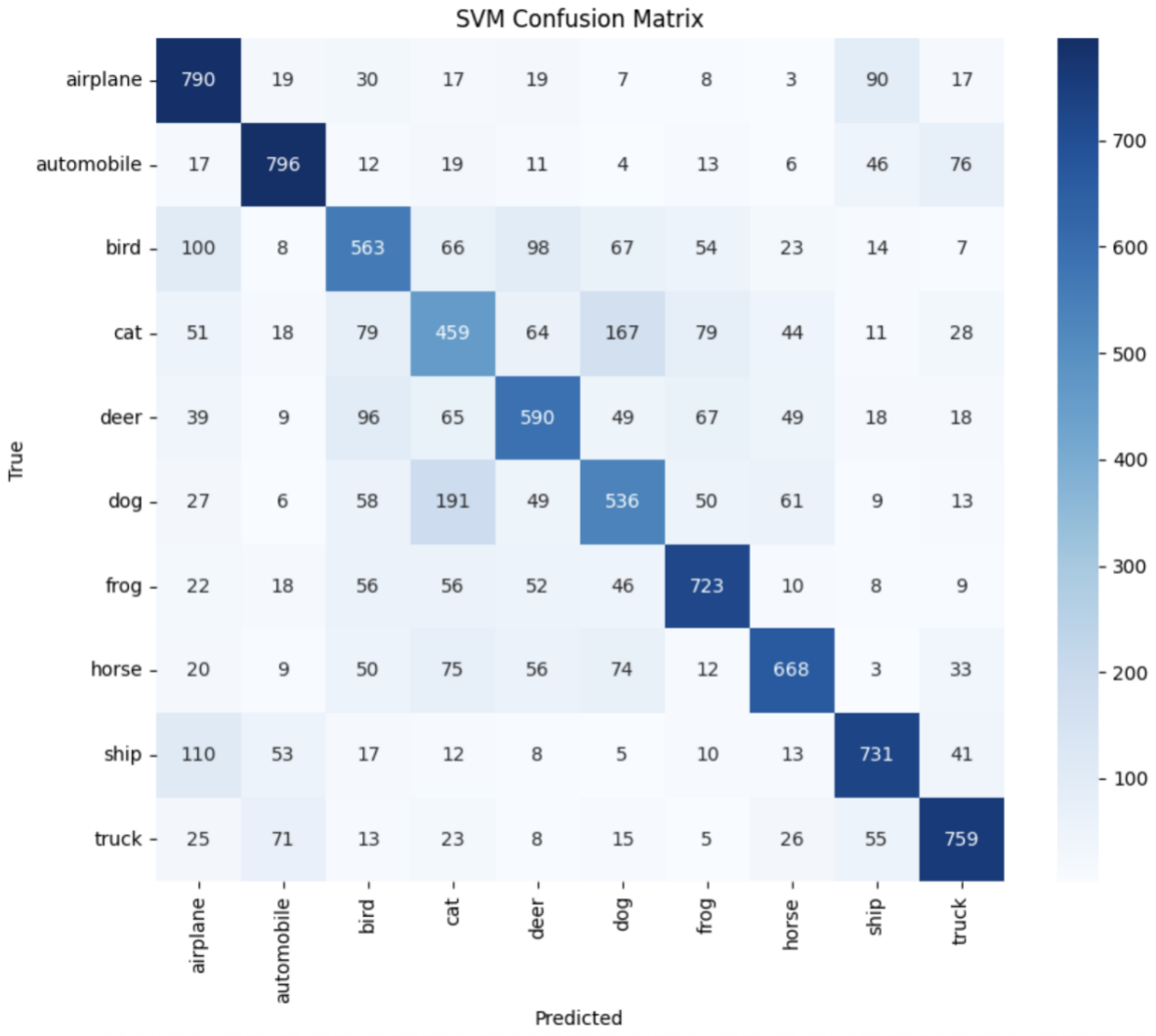
Confusion Matrix Example (SVM-RBF)

	airp	auto	bird	...
airp	470	77	32	...
auto	59	520	7	...
...	...	...	...	...

```
#confusion Matrix Visualization
import seaborn as sns
```

```
cm = confusion_matrix(y_test, y_test_pred_svm)

plt.figure(figsize=(10,8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=classes, yticklabels=classes)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('SVM Confusion Matrix')
plt.show()
```



Comparative Discussion: Classical ML vs. Deep Learning

Classical ML (SVM/KNN): Strengths

- Interpretable.

- Efficient on low/medium dimensional engineered features.
- No need for massive computational power.

Classical ML: Weaknesses

- Plateaus in performance (CIFAR-10 is complex!).
- Raw images are too high-dimensional—requires robust feature engineering.
- Lack of end-to-end learning (feature engineering is manual).

Deep Learning (CNN): Strengths [for reference]

- Learns features directly from raw data.
- State-of-the-art benchmarks (~95%+ accuracy on CIFAR-10).
- Flexible architectures for scaling and transfer learning.

Deep Learning: Weaknesses

- Requires large datasets, significant GPU/TPU resources.
- Difficult to interpret internal representations.
- Prone to overfitting and "black box" phenomena.

Summary Table: Methodologies Comparison

Approach	Accuracy (CIFAR-10)	Data Need	Interpretability	Resource Use
Classical ML	~45–55%	Low-Medium	High	Low
Deep Learning	~90–95%	High	Low	High

Conclusion and Lessons Learned

This project demonstrated the use of classical machine learning techniques, specially, SVM and KNN with deliberately crafted color and HOG features, for multiclass object recognition on the challenging CIFAR-10 dataset. The workflow reinforced several best practices:

- Critical importance of feature engineering: good features are the backbone of classical ML.
- Limitation of classical ML on complex and high dimensional data: while interpretable and fast, the performance ceilling on datasets of CIFAR10 is clear (~50% accuracy)
- Value of validation splits: prevents overfitting / model selection bais.
- Transparency vs. power: while classical ML is instructive and interpretable, deep learning is quired for production-grade performance on modern image recognition benchmarks.

- Model limitations: model performance stalls below deep learning benchmarks due to the simplicity of hand-crafted features.
- Feature extraction is domain-specific and not 'learned' from data.

For the future work, I will be exploring hybrid methods (feature extraction via deep. Nets, then classical classifiers). Also, I will be using classical ML as a diagnostic / teaching tool before scaling to deep learning.

References

1. Krizhevsky, A., & Hinton, G. (2009). *Learning Multiple Layers of Features from Tiny Images*. CIFAR-10 Dataset.
2. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, É. (2011). Scikit-learn: Machine Learning in Python. *JMLR*, 12, pp. 2825–2830.
3. Dalal, N., & Triggs, B. (2005). Histograms of Oriented Gradients for Human Detection. *CVPR 2005*.
4. Zhang, Z., & Ma, Y. (2012). *Ensemble Machine Learning: Methods and Applications*. Springer.
5. Goodfellow, I., Bengio, Y., & Courville, A. (2016) *Deep Learning*. MIT Press. [https://www.deeplearningbook.org/]
6. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. *CVPR 2016*.
7. Keras CIFAR-10 dataset documentation: https://keras.io/api/datasets/cifar10/
8. van der Walt, S. et al. (2014). scikit-image: image processing in Python. *PeerJ 2:e453*.