Synthetic-to-Real Object Detection Using HOG+SVM

Justin Wenzel

May 12, 2025

1. Introduction

Object detection is a core task of computer vision and is used in applications such as robotics, autonomous driving, and surveillance. Object detection is now being done using more advanced deep learning techniques, but it requires a vast amount of data to be trained on. It may be difficult, or even costly, to obtain this data in many cases since it can be time-consuming, costly, or impossible due to privacy issues or unusual subjects. To help solve this issue, the idea of using synthetically generated data created from a simulator has become a potential alternative. But challenges still remain as models trained entirely on synthetic data can suffer in performance when applied to the real world, this is due to a domain gap between synthetic-to-real.

The idea behind this project is to learn more about this domain gap, not try and make the next state-of-the-art optimized deep learning model for helping generalize synthetic to real. Instead, a non-deep learning, more traditional approach is used in this project, specifically known as HOG+SVM, introduced for pedestrian detection by Dalal and Triggs [1]. In this method, Histogram of Gradients (HOG) is used for feature extraction, while Support Vector Machines (SVM) perform classification. The reason behind using this more traditional approach is that it is easier to interpret the model's decision factors and to analyze the effects of the domain gap. The original HOG+SVM uses a linear method which remains as a simple baseline, but we further will also expand by testing against HOG+SVM models that use an additive kernel method introduced by Maji and Malik [2], this will allow for testing on whether non-linear decision boundaries can make a difference in interpreting the domain-shift as not all data is interpretable linearly.

To perform this investigation, synthetic data is generated using Microsoft AirSim [3], which is a simulation environment used in computer vision research involving drones and autonomous vehicles. In this project, it will be used to create a simulated dataset of labeled images of a wolf model that will be created from different angles in multiple scenarios that change in background and weather conditions. The models are then trained on this synthetic dataset and tested on real-world images of wolves to analyze detection performance. Using this setup allows for the synthetic-to-real domain gap to be explored within a controlled environment, along with using a traditional, more transparent model for interpreting how model decisions are impacted by the domain shift.

2. Literature Review

2.1 Histogram of Oriented Gradients (HOG)

Histogram of Oriented Gradients is a feature extraction technique that was initially suggested by Dalal and Triggs in the year 2005 [1], used for pedestrian detection. The main idea of HOG is to represent the appearance of objects and their shapes by describing the distribution of gradient and or intensity and their orientations within the shape of the object. HOG is founded on the observation that the shape of an object can be described well with respect to the local distribution of gradients, regardless of color and texture variations, as it focuses more on shape.

During the calculation of HOG features, the image is first converted into a grayscale image and divided into localized, small regions in the image space called cells, typically of 8x8 pixel size. For every cell, the direction and magnitude of the gradient are computed for every pixel using a derivative mask. The gradients are then pooled into different orientation bins, which form a histogram that represents the direction of edges in that cell. To improve HOG against contrast and illumination changes, the histograms are normalized over a larger spatial region known as blocks, which are typically 2x2 cells in size. These normalized histograms from each block of the image are then finally stacked on top of one another to form a resulting HOG description vector.

This method was first widely used in pedestrian detection and outperformed other feature-extracting techniques available at its time. Also, when it was introduced, it was originally paired with a linear classifier known as Support Vector Machines, creating a full object detection pipeline that was expanded to numerous other objects for detection purposes over time.

2.2 Support Vector Machines (SVM)

Support Vector Machines (SVM) are supervised machine learning algorithms utilized for classification and regression tasks. Initially created and introduced by Vladimir Vapnik and Corinna Cortes [4]. SVMs are based on the principle of selecting the optimal hyperplane that maximally distinguishes data points of various classes. This process minimizes generalization error, which makes SVMs particularly valuable in situations involving many features to try and maximize splitting the data.

For linear cases, SVMs attempt to generate a decision boundary in the input feature space that tries to maximize data separation. The decision function is constructed from a subset of the training data, known as the support vectors, which are the points nearest to the margin and carry the maximum information in defining the optimal decision boundary. Linear SVMs can easily fail to converge when they try to classify nonlinear data.

SVMs also possess an additional ability since they can also use the ”kernel trick” and project the input feature data into a higher-dimensional feature space without ever calculating the transformation. This allows nonlinear classification when combined with kernel functions such as polynomial, radial basis function (RBF), or other kernels as covered in the next section.

2.3 Non-Linear Extension Using Kernels

While Dalal and Triggs initially introduced the idea of using only a linear SVM in the HOG+SVM pedestrian detection pipeline as a baseline. Linear SVMs come with a potential limitation as it assumes that it can linearly separate data in the input feature space. To address this issue, kernel methods can be used to extend the simplicity of basic linear SVMs by mapping the input features into a higher-dimensional space, which allows the models to learn more nonlinear decision boundaries.

This idea of using kernel SVMs to expand the potential capabilities for HOG+SVM was performed by Maji and Malik [2] as they were able to introduce additive kernel SVMs paired with HOG to allow for non-linear classification while still maintaining computational efficiency. They used histogram-based kernels, such as the chi-squared kernel, which is well-suited for comparing feature distributions, such as the features that HOG extracts.

The chi-squared kernel discussed in Maji and Malik's paper is defined below:

$$K_{\chi^2}(x, z) = \sum_{i=1}^{n} \frac{2x_i z_i}{x_i + z_i}$$

where x and z are nonnegative vectors such as normalized histograms. This kernel checks corresponding elements and provides a higher weight to proportional similarity, making it highly appropriate for comparing histogram features such as HOG. While most of Maji and Malik's experiments appear to be performed using the histogram intersection kernel (IKSVM), they briefly describe in section 5.1 that ”Both the histogram intersection kernel and the χ^2 kernels are known to be positive definite for histogram-based features and hence can be used with the standard SVM machinery.” Marking them both as good potential kernels to be used on SVMs paired with HOG feature extraction.

By proving that additive kernels such as the chi-squared enable classical models such as HOG+SVM to create decision boundaries for nonlinear relationships, allows for the interpretation of more complex data. In this project, the chi-squared kernel is implemented through the AdditiveChi2Sampler from scikit-learn to create a kernel-enhanced SVM model using this paper to expand the project work.

3. Methodology

3.1 Data Collection

The synthetic datasets used in training the models were constructed by using Microsoft Air-Sim, which is built on Unreal Engine, and is used in computer vision research which making it a good platform for collecting images to train a computer vision task. A wolf model created by PROTOFACTOR INC [5] that was placed in multiple outdoor-like environments and performed different animations to simulate wolf behavior. The synthetic images were captured using a simulated RGB camera that varied in different positions and distances from the wolf. Along with this, the environment was altered by varying different weather conditions, including fog, snow, rain, and sky color, to create variability in the dataset/ Positive samples were created by cropping around the wolf in the images, while negative samples were cropped from areas without any occurrence of the wolf in view. This environment allowed for easy extraction and creating of a fully labeled, synthetically created training dataset in a reproducible and controlled environment.

In addition to having the synthetic images, real datasets were also used from a publicly available website called Roboflow [6, 7], which is known for being a dataset sharing platform, where users can share labeled images or completed datasets. In this project, two datasets were used from Roboflow: a small and a large dataset. They consisted of annotated images of wolves in different environments, positions, and lightning, which created a diverse set of real-world images between both datasets. While positive samples of the wolves were cropped from the original images by using their annotated bounding boxes, the negative samples were extracted from real natural background scenes that did not include the object. These datasets created a real-world set of examples that were used for training and testing.

In the end three datasets were prepared for the project: (1) a synthetic-only set, (2) a real-only set, (3) a combined dataset of real and synthetic images with 80% and 20% synthetic. Then, finally, the small dataset from Roboflow was used for testing purposes. All training sets had the same number of positive and negative sample images, with 1,824 images in each category (example training images are shown in Figure 1). The final testing set had 68 images with one wolf in each image. This final data setup allowed for an analysis to be performed on how well the models trained on synthetic, real, or mixed datasets would generalize to unseen real-world images.

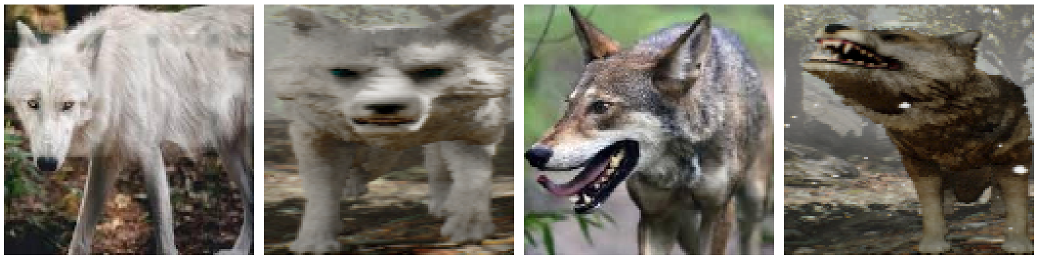


Figure 1: Example training images from the dataset.

3.2 Feature Extraction and Model Training

Feature extraction and model training depended on a two-stage process where the HOG descriptors were extracted from the input image, and then the features were passed to an SVM classifier for training. All images, upon being read in for the first time, were converted to grayscale and resized to the same size of 128x128 pixels; This size was chosen to accommodate the wolf since it can change in size depending on whether it is viewed from the front or side. Everything was implemented in Python using public libraries like OpenCV, NumPy, scikit-image, and sklearn. Guidance for implementing the HOG+SVM pipeline in Python was provided through referencing VanderPlas's Python Data Science Handbook [8] because it provided good examples to begin implementation.

For the linear SVM, HOG features were extracted with a configuration similar to the original Dalal and Triggs setup using 9 orientation bins, 8x8 pixel cell size, 2x2 cell block size, and L2-Hys normalization. This architecture setup was chosen as it was presented as one of their better-performing setups. These HOG features were first extracted using the hog() function from the skimage. A feature module to help calculate and extract the HOG features into vectors. The feature vectors are then passed to a linear SVM (LinearSVC) to train an SVM using scikit-learn, with a regularization parameter of C=0.01 to encourage a simple decision boundary by allowing some training error to limit potential overfitting. Finally, a maximum number of 10000 iterations was allowed to ensure convergence was reached instead of an infinite cycle during training.

The kernelized SVM was also trained using a two-step pipeline, but the model designed originated from Maji and Malik, matching one of their better-performing model setups that used an additive kernel approach. The kernel used in the case of this project was implemented via the AdditiveChi2Sampler from scikit-learn, allowing for the nonlinear data to be projected to a higher space and be linearly separated. The HOG features in this model were extracted using a multi-scale technique, where features were calculated at three different cell sizes, 18x18, 6x6, and 3x3 pixels, and then concatenated into a single descriptor vector. The rest of the implementation followed the same idea as before by training a linear SVM with the same hyperparameters as the previous model, except this time it was passed the concatenated descriptor vectors projected into a higher space.

In the end, both models were trained on a balanced dataset created using reproducible sampling by using a fixed seed for randomization. This finally resulted in a total of six models, which included one linear SVM and a kernel-enhanced SVM trained under each of the three training conditions, including synthetic-only, real-only, and a combined dataset.

3.3 Detection Pipeline

After the models were trained, a sliding window approach was used in the detection pipeline to localize and detect the wolfs in full-resolution test images. The pipeline would take the inputted test image and scale it to different sizes, then the 128x128 patch-sized window would slide across the image, and it would be determined if the window contained a wolf in it at each position.

Firstly, every image was inputted and converted to a grayscale and resized to a resolution of 640x480 pixels to keep computation time low for testing and to match training conditions. Then a multi-scale image pyramid is constructed by scaling the image up and down by a fixed scale factor of 1.25 to multiple levels, creating zoomed-in and out images of the original to allow for the detection of varying-sized wolfs. Then the sliding window of 128x128 was slid across each scale of the image pyramid with a step size of 32. This window size was used to match the patch size during training, and the step size is used to determine how far the window should move on the image horizontally and vertically between predictions. At each window location, HOG features were extracted either using the default extraction technique used on the linear SVM or the multi-scale feature concatenation technique as used on the kernel SVM. These techniques were used with their respective models to ensure HOG features were extracted the same during detection as during training. The extracted feature vectors were then passed into the trained SVM and returned a confidence score for detection.

The windows that had a score above a certain threshold such as 0.5, 1, 1.5, or 2 were kept and saved as potential candidate detections. After all candidate detections were found at each image level, non-maximum suppression (NMS) was used to remove duplicate and overlapping prediction boxes that had an intersection over union of 0.5 and was implemented using the imutils library. All final predictions were then saved and applied over the image for visualization to compute further evaluation metrics.

By using a fixed step size of 32 and different thresholds for the confidence score of the SVM allowed for the sensitivity of detections was tested at different configurations. Using this consistent detection pipeline allowed for both the linear and kernel-based HOG+SVM models to be tested against the same framework implementation, allowing for an equal comparison between the six models that were trained on the different datasets.

3.4 Evaluation Procedure

Evaluating the model's performance was done using the small withheld real-world test set of 68 images from Roboflow, where each image had one instance of a wolf object. This test set was not visible to any training instance and only served as a test set to assess the generalization ability of all the models in the same real-world setting of images. Each image had its own bounding box annotated truth label, which was used for comparison against the predicted detections.

The models' resulting predictions were evaluated using three standard object detection metrics, which include precision, recall, and F1 score. The equations for these metrics are provided below. A prediction was considered to be correct if its bounding box overlapped with an Intersection over Union (IoU) of 0.5 or 50% with the ground-truth box. A prediction that had no overlap or did not reach the 0.5 IoU threshold to the ground-truth box was considered to be a false positive prediction. Any ground-truth box that never had a match to any prediction boxes is considered a false negative because the object was not detected.

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

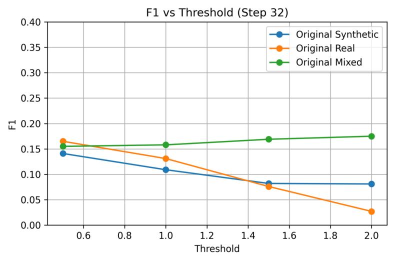
$$\text{F1 Score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

To further evaluate the detector's behavior under different settings, tests were conducted using a variety of detection score thresholds ranging from 0.5 to 2.0 in increments of 0.5. A fixed step size of 32 pixels was used for the sliding of the window. Each model's resulting metrics were computed and saved on each image, which was then used to compare the different trained models.

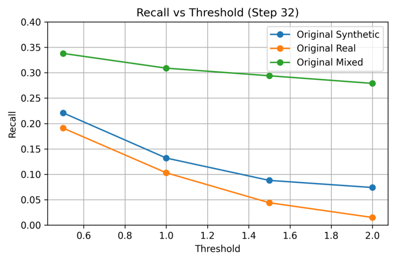
4. Results

The results section presents the quantitative results of evaluating both the linear and kernel HOG+SVM object detector models on the real-world test set. For each model, the performance measures of precision, recall, and F1 score were computed against four decision thresholds from 0.5 to 2.0 using the fixed sliding window step size of 32 pixels. The final results are displayed in a line graph, and a supplemental table is provided with the raw values, which the table also contains true positive (TP), false positive (FP), and false negative (FN) counts at each threshold. The results are then organized by model type of either linear or kernel SVM, and then split by the dataset they were trained on such as synthetic-only, real-only, or mixed data.

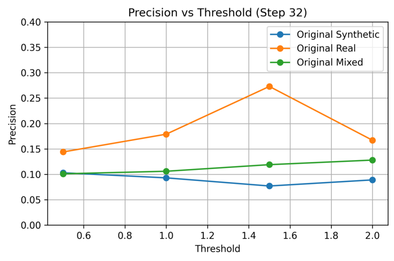
4.1 Linear SVM Results



(a) F1 Score



(b) Recall



(c) Precision

Figure 2: Performance metrics vs. threshold for linear SVM models (step size = 32).

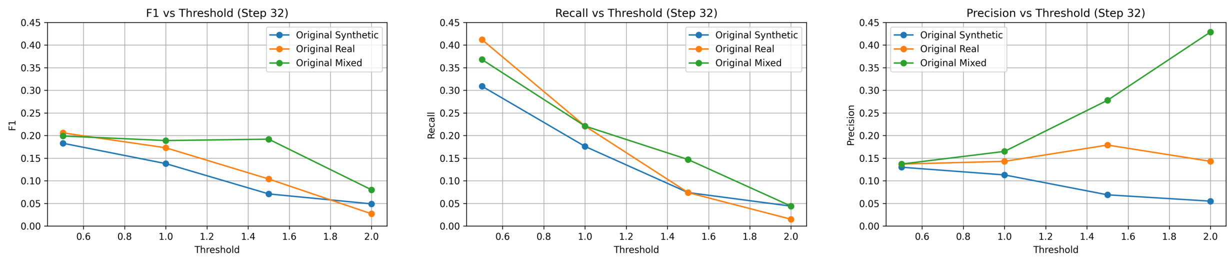
Table 1: Linear SVM performance across different thresholds (step size = 32).

Model	**Threshold**	**Precision**	**Recall**	**F1 Score**	**TP**	**FP**	**FN**
Synthetic-Only	0.5	0.103	0.221	0.141	15	130	53
	1.0	0.093	0.132	0.109	9	88	59
	1.5	0.077	0.088	0.082	6	72	62
	2.0	0.089	0.074	0.081	5	51	63
Real-Only	0.5	0.144	0.191	0.165	13	77	55
	1.0	0.179	0.103	0.131	7	32	61
	1.5	0.273	0.044	0.076	3	8	65
	2.0	0.167	0.015	0.027	1	5	67
Mixed	0.5	0.101	0.338	0.155	23	205	45
	1.0	0.106	0.309	0.158	21	177	47
	1.5	0.119	0.294	0.169	20	148	48
	2.0	0.128	0.279	0.175	19	130	49

The linear SVM outputs revealed a more complex and sometimes conflicting pattern of behavior across the models at each threshold. Precision did have a consistent pattern for the synthetic-only model, as it had a consistent decrease as the threshold increased. But for the real-only model, precision rose and then decreased, most likely caused by its decreasing TP and FP detections, with FP dropping a lot. The mixed model, however, had a consistent increase in precision as the threshold increased.

Recall was behaving more consistent across all models as it dropped when the threshold increased, which is expected behavior as the decision of detection becomes more strict. But the slope at which the decline in TP occurred for each model did vary. The real-only and synthetic-only models had a sharper fall in recall, with real-only having the biggest decrease in TP detections. While the mixed model was not as steep of a loss in its TP detections compared to the others but it did continue to have some of the highest FP detections.

Then, for F1 scores, it consistently remained that the mixed model had the highest results, showing the model's ability to balance its precision and recall more effectively than the other models, but it did have the most detections overall across its TP and FP detections. While real-only did show promise in improving its F1 score, as precision did sometimes increase, it was always offset by the drastic drop in recall. Finally, the synthetic-only model was consistent but was almost always the worst-performing model on all metrics, reflecting poor generalization to real data.



(a) F1 Score (b) Recall (c) Precision

Figure 3: Performance metrics vs. threshold for kernel SVM models (step size = 32).

4.2 Kernel SVM Results

Table 2: Kernel SVM performance across different thresholds (step size = 32).

Model	Threshold	Precision	Recall	F1 Score	TP	FP	FN
Synthetic-Only	0.5	0.130	0.309	0.183	21	141	47
	1.0	0.113	0.176	0.138	12	94	56
	1.5	0.069	0.074	0.071	5	67	63
	2.0	0.055	0.044	0.049	3	52	65
Real-Only	0.5	0.137	0.412	0.206	28	176	40
	1.0	0.143	0.221	0.173	15	90	53
	1.5	0.179	0.074	0.104	5	23	63
	2.0	0.143	0.015	0.027	1	6	67
Mixed	0.5	0.137	0.368	0.199	25	158	43
	1.0	0.165	0.221	0.189	15	76	53
	1.5	0.278	0.147	0.192	10	26	58
	2.0	0.429	0.044	0.080	3	4	65

The kernel SVM models generally had a better F1 score and recall compared to their linear counterpart models, especially when the threshold was lower. A big change this time is that the real-only model actually had the highest recall score of 0.412 at a threshold of 0.5, with the most TP results compared to any other model, but it came at the cost of nearly 176 false positive detections throughout the 68 images, which kept the precision lower.

When it came to precision scores, most of the kernel-based models showed a decreased or minimal improvement as the threshold increased. But the mixed trained model had the most improvement with a precision jump from 0.137 to 0.429, the highest precision compared to any other model, but it still suffered due to its low recall capabilities at a threshold of 2.

This time, the F1 scores followed a more consistent behavior compared to the linear models. In this instance, all the models as thresholds increased, the F1 scores decreased; while the decreasing F1 scores are not ideal, it is a more consistent behavior than seen before. While the real-only model had the highest F1 score, 0.206, at a threshold of 0.5, most likely caused by its high recall results.

Overall, the kernel-based models appeared to perform slightly better as it was able to adapt better to the nonlinear structures in the HOG features. But it should be noted that while the results were still not great, they were more consistent in the kernel SVM models compared to the Linear SVM models.

5. Discussion

The goal of this project was once again to examine whether synthetic images could be used as an effective alternative to train an object detector to generalize to real-world images, using HOG+SVM, a classical object detection pipeline. The focus on using the classical model was to examine how well a synthetically trained system would generalize to the real world, not to achieve state-of-the-art performance. These HOG+SVM models, each trained on the synthetic-only, real-only, and mixed datasets, gave a foundation for understanding how the object detection models behaved under the domain transfer from synthetic to real.

The results clearly show evidence that a domain gap exists between the synthetic and real data. The synthetic-only trained models performed poorly on nearly all metrics. Confirming that synthetic data, regardless of trying to create similarity in shape and environmental setting, lacks something that is in the real-world images. But it's also noticeable that the models trained on real-only data struggled as well, especially as the threshold increased, the recall would drop significantly. This shows that while the synthetic data is lacking in a key feature, the real data is also falling short in performance, which could be due to the data not being diverse enough, or lacking in quantity of training data, or even due to complexity.

Some of the best performance in F1 scores came from the mixed dataset trained models, as they often had higher scores in terms of precision and recall. While the metrics were poor for the mixed models, they were still slightly better and appeared to follow a more gradual and consistent behavior compared to the models trained on the other datasets. This could highlight that the mixed dataset was able to capture more diverse features in its set between the synthetic and real images, creating a decision boundary that is more aligned with the real-world test images. While it's said that the mixed dataset performed the best, it is also noticeable that the linear SVM models on the mixed dataset had a consistently high FP count across all thresholds. This appears to be a more "trigger-happy" kind of behavior because the linear model was not able to create a clean decision boundary to split the data for classification, but the kernel SVM showed a clearer ability to separate the data and did not have this consistently high FP count across all thresholds. This shows the benefits of using non-linear kernels to define more complex decision boundaries.

Nonetheless, it should be noted that all of the models did not really perform well. They all had low precision and recall scores, along with many false positives. This might suggest that the current object of interest for this scenario might have been too complex for the classical HOG+SVM object detection pipeline. Due to the changing shape, angles, positions, and background clutter of the wolfs, more than likely overcomplicated the hand-sculpted features from HOG, and the decision boundaries for the SVM were just too simple. A more uniform or simple object might have provided a more promising outcome and a better starting point for investigating the synthetic-to-real domain adoption through a traditional method. As in the original papers introducing HOG+SVM, the authors showed good results for detecting pedestrians, but pedestrians are often confined to a simpler shape compared to a wolf and this was also noted in the paper as the authors discussed the models poor ability to perform well on a dataset with people in more complex positions besides standing up straight.

A final note is also that the evaluation process itself here was limited. Due to runtime constraints, only a step size of 32 pixels was used at detection time during testing. This helped reduce computational cost as lower step sizes, such as 8 or 16 pixels, took hours to gather results. While lower step sizes might have provided more localization accuracy in each detection, the results were obviously influenced by other factors. The idea of other step sizes, or using a more adaptive step size approach, might be useful for other objects or feature exploration of this topic.

6. Conclusion

While this project's idea of using synthetic data to train models for generalizing on real-world data did not produce great results across all the models trained on the different datasets. It still leaves the idea of using synthetic data to assist in training open for discussion, as the mixed dataset of synthetic and real images appeared to provide some of the best results. But it also highlights the limitations of these traditional object detection pipelines when being used on complex data. A form of future work could be to see if deep learning-based methods can improve or more easily bridge the domain gap between synthetic data for training and generalization to real-world data more easily especially for complex tasks such as the one presented here.

References

[1] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, 2005, pp. 886–893 vol. 1.

[2] S. Maji, A. C. Berg, and J. Malik, "Efficient classification for additive kernel svms," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, 2013.

[3] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," 2017. [Online]. Available: https://arxiv.org/abs/1705.05065

[4] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995. [Online]. Available: https://doi.org/10.1007/BF00994018

[5] PROTOFACTOR INC., "Realistic wolf model - unreal engine asset," https://www.fab.com/listings/2dd7964c-a601-4264-a53d-465dcae1644c, 2025, accessed May 2025.

[6] M. Dhelia, "Wolf detection dataset," https://universe.roboflow.com/test-xgtag/wolf-g0p07-5ocjr/dataset/2, 2025, accessed May 2025.

[7] I. Informatique, "Informatique wolf dataset," https://universe.roboflow.com/test-xgtag/informatique-cs1cx-krmmv/dataset/2, 2025, accessed May 2025.

[8] J. VanderPlas, *Python Data Science Handbook: Essential Tools for Working with Data.* O'Reilly Media, 2016. [Online]. Available: https://jakevdp.github.io/PythonDataScienceHandbook/

Appendix: Source Code

HOG_SVM_Test.py

```

import os
import cv2
import numpy as np
from skimage.feature import hog
from imutils.object_detection import non_max_suppression
from sklearn.pipeline import make_pipeline
import joblib

# Model paths Linear SVM models
#models = {
#    "Original_Synthetic": "hog_svm_original_synthetic_model.pkl",
#    "Original_Real": "hog_svm_original_real_model.pkl",
#    "Original_Mixed": "hog_svm_original_mixed_model.pkl"
#}

# Model paths kernel SVM models
models = {
    "Original_Synthetic": "hog_svm_kernel_synthetic.pkl",
    "Original_Real": "hog_svm_kernel_real.pkl",
    "Original_Mixed": "hog_svm_kernel_mixed.pkl"
}

input_dir = "../HOGSVM_Test/images"
label_dir = "../HOGSVM_Test/labels"
output_dir = "../HOGSVM_Test/Full_Outputs_Final"
os.makedirs(output_dir, exist_ok=True)

hog_params = {
    'pixels_per_cell': (8, 8),
    'cells_per_block': (2, 2),
    'orientations': 9,
    'block_norm': 'L2-Hys'
}

window_size = (128, 128)
scale_factor = 1.25
```

```python
iou_threshold = 0.5
step_sizes = [8, 16, 32] # Smaller step size take longer to run
score_thresholds = [0.5, 1.0, 1.5, 2.0]

# Max image input
MAX_WIDTH = 640
MAX_HEIGHT = 480

# Extract multiscale hog features for models trained using
    multiscale hog feature extraction
def multiscale_hog(image, cell_sizes=[(18, 18), (6, 6), (3, 3)]):
    # use training config
    features = []
    for cell in cell_sizes:
        h = hog(image,
                orientations=9,
                pixels_per_cell=cell,
                cells_per_block=(1, 1),
                block_norm='L2-Hys',
                transform_sqrt=True,
                feature_vector=True)
        features.append(h)
    return np.concatenate(features)

# Creates image pyramid for different size wolf detection
def pyramid(image, scale=1.25, min_size=(128, 128)):
    yield image
    while True:
        w = int(image.shape[1] / scale)
        h = int(image.shape[0] / scale)
        image = cv2.resize(image, (w, h))
        if image.shape[0] < min_size[1] or image.shape[1] <
            min_size[0]:
            break
        yield image

# Apply sliding window over input image returning the window
    coordinates on the image
def sliding_window(image, step_size, window_size):
    for y in range(0, image.shape[0] - window_size[1], step_size):
        for x in range(0, image.shape[1] - window_size[0],
            step_size):
            yield (x, y, image[y:y + window_size[1], x:x +
                window_size[0]])

# Convert labels from Roboflow datasets from YOLO ground truth
    annotation to box coordinates (XC, YC, W, H) to (X1, Y1, X2, Y2)
```

```python
def yolo_to_bbox(yolo_line, img_w, img_h):
    parts = yolo_line.strip().split()
    if len(parts) < 5:
        return None
    x_center, y_center, width, height = map(float, parts[1:5])
    x_center *= img_w
    y_center *= img_h
    width *= img_w
    height *= img_h
    x1 = int(x_center - width/2)
    y1 = int(y_center - height/2)
    x2 = int(x_center + width/2)
    y2 = int(y_center + height/2)
    return [x1, y1, x2, y2]

# Computes intersection over union
def compute_iou(boxA, boxB):
    xA = max(boxA[0], boxB[0])
    yA = max(boxA[1], boxB[1])
    xB = min(boxA[2], boxB[2])
    yB = min(boxA[3], boxB[3])
    interArea = max(0, xB - xA) * max(0, yB - yA)
    boxAArea = (boxA[2]-boxA[0]) * (boxA[3]-boxA[1])
    boxBArea = (boxB[2]-boxB[0]) * (boxB[3]-boxB[1])
    return interArea / float(boxAArea + boxBArea - interArea)

# Evaluate each model
for model_name, model_path in models.items():
    print(f"Evaluating {model_name}...")
    model = joblib.load(model_path)

    result_file = os.path.join(output_dir,
       f"{model_name}_results.txt")
    with open(result_file, 'w') as f_out:

        # Iterate over different step sizes
        for step_size in step_sizes:
            for score_threshold in score_thresholds:

                total_TP, total_FP, total_FN = 0, 0, 0
                image_files = [f for f in os.listdir(input_dir) if
                   f.lower().endswith((".jpg", ".jpeg", ".png"))]

                for img_name in image_files:
                    img_path = os.path.join(input_dir, img_name)
                    label_path = os.path.join(label_dir,
                       os.path.splitext(img_name)[0] + ".txt")

                    image = cv2.imread(img_path)
                    if image is None:
                        continue

                    # Resize images to ensure they are 640x480
                    h, w = image.shape[:2]
                    if w > MAX_WIDTH or h > MAX_HEIGHT:
                        scale = min(MAX_WIDTH / w, MAX_HEIGHT / h)
                        image = cv2.resize(image, (int(w * scale), int(h * scale)))

                    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
                    detections = []

                    # Perform detections on the image pyramid with the sliding window
                    for resized in pyramid(gray, scale=scale_factor, min_size=window_size):
                        scale = gray.shape[1] / float(resized.shape[1])
                        for (x, y, window) in sliding_window(resized, step_size, window_size):
                            if window.shape[0] != window_size[1] or window.shape[1] != window_size[0]:
                                continue
                            #features = hog(window, **hog_params) # For linear SVM trained models, does not use multiscale_hog
                            features = multiscale_hog(window) # For kernel SVM trained models, trained on multiscale_hog descriptors
                            score = model.decision_function([features])[0]
                            if score > score_threshold:
                                x1 = int(x * scale)
                                y1 = int(y * scale)
                                x2 = int((x + window_size[0]) * scale)
                                y2 = int((y + window_size[1]) * scale)
                                detections.append([x1, y1, x2, y2, score])

                    # Apply non max suppresion for overlapping or redundant predictions
                    pred_boxes = non_max_suppression(np.array([d[:4] for d in detections]), probs=[d[4] for d in detections])

                    # Read in ground truth bounding boxes
                    gt_boxes = []
                    if os.path.exists(label_path):
                        with open(label_path, "r") as f:
                            for line in f:
                                box = yolo_to_bbox(line, image.shape[1], image.shape[0])
                                if box:
                                    gt_boxes.append(box)

                    # Match predictions with ground truth boxes
                    matched = set()
                    TP, FP = 0, 0
                    for pred in pred_boxes:
                        match = False
                        for i, gt in enumerate(gt_boxes):
                            if i in matched:
                                continue
                            if compute_iou(pred, gt) >= iou_threshold:
                                TP += 1
                                matched.add(i)
                                match = True
                                break
                        if not match:
                            FP += 1
                    FN = len(gt_boxes) - len(matched)
                    total_TP += TP
                    total_FP += FP
                    total_FN += FN

                # Compute evaluation metrics
                precision = total_TP / (total_TP + total_FP) if (total_TP + total_FP) else 0
                recall = total_TP / (total_TP + total_FN) if (total_TP + total_FN) else 0
                f1 = 2 * (precision * recall) / (precision + recall) if (precision + recall) else 0

                # Log results
                log = f"[Step={step_size} | Thresh={score_threshold}]
```

```
                        Precision={precision:.3f}, Recall={recall:.3f},
                        F1={f1:.3f}, TP={total_TP}, FP={total_FP},
                        FN={total_FN}\n"
                    print(log.strip())
                    f_out.write(log)

print("\n Test complete")
```

Listing 1: HOG+SVM Test Detection Pipeline

HOG_SVM_Linear_Train.py

```

import os
import cv2
import numpy as np
import random
from skimage.feature import hog
from sklearn.svm import LinearSVC
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
import joblib
from tqdm import tqdm

# Set fixed seed for reproducibility
random.seed(42)

# Define dataset paths
pos_dir = r"../HOGSVM_mixed_Dataset/pos"
neg_dir = r"../HOGSVM_mixed_Dataset/neg"

# Target sample size
n_samples = 1824

# HOG parameters (Dalal & Triggs setup)
hog_params = {
    'pixels_per_cell': (8, 8),
    'cells_per_block': (2, 2),
    'orientations': 9,
    'block_norm': 'L2-Hys'
}

X = []
y = []

def extract_hog_features(folder, label, limit=None):
    files = os.listdir(folder)
    if limit:
```

```python
        files = random.sample(files, min(limit, len(files)))
    for filename in tqdm(files, desc=f"Processing label={label}"):
        filepath = os.path.join(folder, filename)
        img = cv2.imread(filepath, cv2.IMREAD_GRAYSCALE)
        if img is None:
            continue
        features = hog(img, **hog_params)
        X.append(features)
        y.append(label)

# Extract fixed number of positive and negative samples
extract_hog_features(pos_dir, label=1, limit=n_samples)
extract_hog_features(neg_dir, label=0, limit=n_samples)

# Convert to NumPy arrays
X = np.array(X)
y = np.array(y)

# Create and train model
clf = make_pipeline(StandardScaler(), LinearSVC(C=0.01,
   max_iter=10000))
print("Training Linear SVM with fixed sample size...")
clf.fit(X, y)

# Save model
joblib.dump(clf, "hog_svm_original_mixed_model.pkl")
print("Model saved as hog_svm_mixed_balanced_model.pkl")
```

Listing 2: Linear HOG+SVM Train Script

HOG_SVM_Kernel_Train.py

```python

import os
import cv2
import numpy as np
import random
from skimage.feature import hog
from sklearn.svm import LinearSVC
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.kernel_approximation import AdditiveChi2Sampler
from tqdm import tqdm
import joblib

# Set fixed seed for reproducibility
random.seed(42)

```

```
# Dataset directories
pos_dir = r"../HOGSVM_Mixed_Dataset/pos"
neg_dir = r"../HOGSVM_Mixed_Dataset/neg"

# Target number of samples
n_samples = 1824
image_size = (128, 128)
cell_sizes = [(18, 18), (6, 6), (3, 3)]
orientations = 9
block_size = (1, 1)

X, y = [], []

def extract_maji_style_hog(img, levels):
    features = []
    for cell in levels:
        h = hog(img,
                orientations=orientations,
                pixels_per_cell=cell,
                cells_per_block=block_size,
                block_norm='L2-Hys',
                transform_sqrt=True,
                feature_vector=True)
        features.append(h)
    return np.concatenate(features)

def load_fixed_samples(folder, label, limit, shuffle=True):
    files = os.listdir(folder)
    if shuffle:
        files = random.sample(files, len(files))
    files = files[:limit]
    for file in tqdm(files, desc=f"Label {label}"):
        path = os.path.join(folder, file)
        img = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
        if img is None:
            continue
        img = cv2.resize(img, image_size)
        feat = extract_maji_style_hog(img, cell_sizes)
        X.append(feat)
        y.append(label)

# Load a fixed number of samples from each class
load_fixed_samples(pos_dir, label=1, limit=n_samples)
load_fixed_samples(neg_dir, label=0, limit=n_samples)

# Convert to arrays
X = np.array(X)
```

```
y = np.array(y)

# Build and train model
clf = make_pipeline(
    AdditiveChi2Sampler(sample_steps=2),
    StandardScaler(),
    LinearSVC(C=0.01, max_iter=10000)
)

print("Training balanced model with Maji et al. settings...")
clf.fit(X, y)
joblib.dump(clf, "hog_svm_kernel_mixed_.pkl")
print("Model saved as hog_svm_maji_balanced.pkl")
```

Listing 3: Kernel HOG+SVM Train Script