Computer Science

Investigating Histograms of Oriented Gradients in Pedestrian Detection

How does the sliding window size, block density, and the derivative mask of a Histogram of Oriented Gradients descriptor impact the performance of a linear Support Vector Machine pedestrian classifier?

Word Count: 2000

May 2025

Contents

1 Introduction

Pedestrian detection is a critical area of research in computer vision and artificial intelligence, with it being one of the extensively studied fields in the past decade [11]. The applications of automatic pedestrian detection span autonomous vehicles, surveillance systems, and robotics [11]. Most notably, automatically detecting pedestrians from moving vehicles could considerable impact economic and social welfare by substantially reducing pedestrian injuries and fatalities, which, in the European Union, make up 20% of all road accidents [28]

Pedestrian detection involves identifying and locating human figures in images or video frames, which presents unique challenges due to the variability in occlusions, diverse backgrounds and changing environmental conditions [11]. Alongside the many variables present in natural pedestrian environments, noise can arise during the image acquisition process, mainly due to imperfect instruments [14]. Therefore cleanly discriminating human appearance and the wide range of poses they can adopt calls for the use of a feature set which would be able to characterise object shape and orientation locally, so that changes in "noisy" regions (like an image's background) do not significantly impact the feature detection in other regions that still provide useful information (like a pedestrian's silhouette).

Histograms of Oriented Gradients (HOG) [7] is well-known [11] image processing algorithms because it solves the problem of variability and noise in pedestrian images by detecting one of the most essential features of images - edges [23] [7]. Despite the suggested superiority of HOG [7] and widespread adoption in modern pedestrian classifiers [11], the parameters used for the algorithm have remained essentially unchanged since the introduction of the method in 2005 [7]

Given the importance of the HOG descriptor in real life applications and significant improvements in the variety, difficulty and scale of pedestrian datasets since 2005 [11], this investigation seeks to maximize the accuracy and performance of a linear Support Vector Machine (SVM) in pedestrian classification by varying the various properties of HOG.

2 Background Information

2.1 Histograms of Oriented Gradients

The most prominent discriminative feature of pedestrians is their shape: limbs, head, and any features with prominent edges [7]. In that regard, HOG features are excellent at pedestrian detection precisely because they prioritize orientation/shape information, unlike other feature descriptors like HaaR wavelets which are colloquially described as "texture features" [41].

2.1.1 One Fundamental Property of Images

At their core, images are matrices that represent pixel intensity values. Elements in grayscale image matrices contain a single intensity value, while elements in colored image matrices contain three (one for each color channel). With this definition of an image, it becomes increasingly simple to understand the meaning of "edge".

An edge is a region in which there is a change of intensity. Figure 1 illustrates the changes in pixel intensity by mapping a row's pixel intensity values to a function's output. Observe that an edge is characterized by the gradient of the pixel intensity function. The function's gradient values are greater at the edges/corners of an object, like a pedestrian's limb, rather than homogeneous areas, like background regions. In this way, gradients may highlight the contours of objects and discard noise/texture information, precisely what is needed in pedestrian detection.

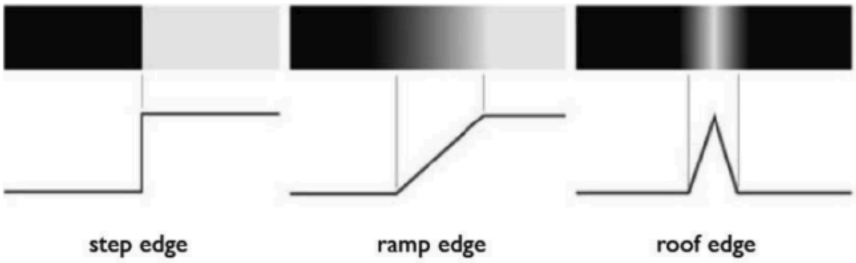

Figure 1: Representation of the three types of edge that can be found in image analysis. Source: [23]

2.1.2 Gradient Computation

In HOG, a derivative mask (also known as a filter or kernel) is used to compute gradient information from an image [7] by performing convolution, the process of adding each element of the image to its local neighbors, weighted by the mask [23], as shown in equation 2.1, where I is the image matrix, K the mask's matrix and k - the "radius" of K (the distance from the center element to an edge element).

$$F(x, y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} I(x+i, y+j) \cdot K(i, j) \tag{2.1}$$

The authors of HOG found that a simple 1D derivative mask of form $[-1, 0, 1]$, formally called a central discrete derivative [23], while being much less computationally expensive than 3x3 Sobel or 2x2 diagonal masks, also performed the best [7].

Convolution on an image I with the aforementioned 1D mask yields a new image F_y defined in 2.2 and convolution with the transposed, or, in other words, "flipped" over its main diagonal, 1D mask yields an image F_x as defined in 2.3.

$$F_y(x_m, y_n) = \frac{\partial I(x_m, y_n)}{\partial x} \approx \mid I(x_m - 1, y_n) - I(x_m + 1, y_n) \mid \tag{2.2}$$

$$F_x(x_m, y_n) = \frac{\partial I(x_m, y_n)}{\partial x} \approx \mid I(x_m, y_n + 1) - I(x_m, y_n - 1) \mid \tag{2.3}$$

Notice however, that $x_m \pm 1$ and $y_n \pm 1$ fall outside $I[0, w-1] \times [0, h-1]$ when $x_m = w-1$ and $y_n = h-1$ respectively. This means that gradient information at image boundaries is lost when using central finite differences for convolution [27]. The information loss is evident in the _hog_channel_gradient where the convolution output at boundary pixels defaults to zero. The nullified boundary pixels may disproportionately impact SVM performance when using smaller detection windows or block sizes, as these zeroed values constitute a larger fraction of the resulting histogram. To address this limitation, this investigation proposes a novel approach that combines central, forward, and backward finite differences [23].

Figure 2 displays the kernels of each of the finite differences. Because both forward and backward differences are not anchored around the central pixel, they can be used to yeild the convoluted intensity values of pixels at the top 2.4 and left 2.5, and bottom 2.6 and right 2.7 edges, respectively.

Forward			Backward			Central		
0	-1	1	-1	1	0	-1	0	1

Figure 2: Three types of finite differences and their corresponding derivative masks. Source: Image by me

$$F_x[x_m, 0] = |I(x_m, 1) - I(x_m, 0)| \tag{2.4}$$

$$F_y[0, y_n] = |I(1, y_n) - I(0, y_n)| \tag{2.5}$$

$$F_x[x_m, h] = |I(x_m, h) - I(x_m, h-1) \quad (2.6)$$

$$F_y[w, y_n] = |I(w, y_n) - I(w-1, y_n) \quad (2.7)$$

With the convoluted pixel values, or, in a sense, the changes in pixel intensity encoded into both F_y and F_x images, combining them into a single feature map G of gradients, or vectors with an angle θ, is as simple as applying the Pythagorean theorem [27], as illustrated in figure 3, where magnitude $= |G(x_m, y_n)| = \sqrt{F_y(x_m, y_n)^2 + F_y(x_m, y_n)^2}$ and $\theta = \arctan\left(\frac{F_y(x_m, y_n)}{F_x(x_m, y_n)}\right)$

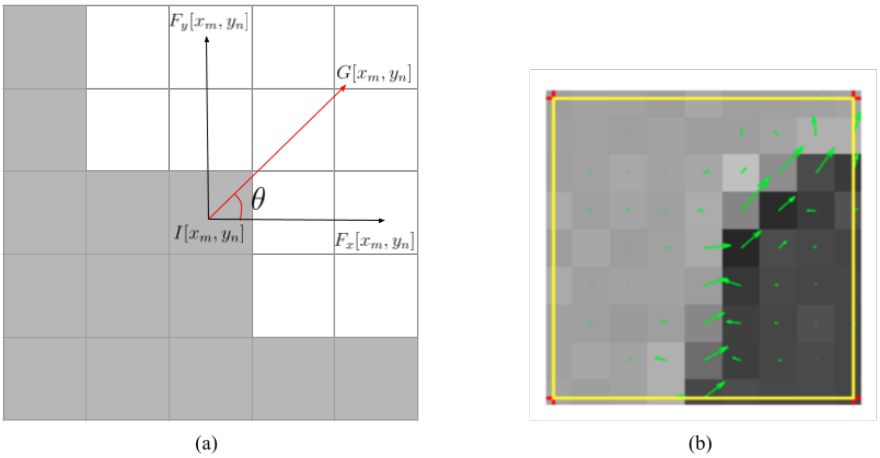

Figure 3: (a) Calculation of gradient vector. Source: Image by me (b) Visualisation of gradient vectors. Source: [27]

2.1.3 Orientation Binning

Orientation Binning hopes to achieve an encoding that is both sensitive to variations in local image content while remaining resistant to miniature changes in pose or appearance. This approach pools gradient orientation information locally, in a similar way that the SIFT feature detector does [17].

The process of orientation binning begins with dividing the constructed feature map of gradients into local spatial regions that the authors of the HOG algorithm called cells, as illustrated in figure 4.

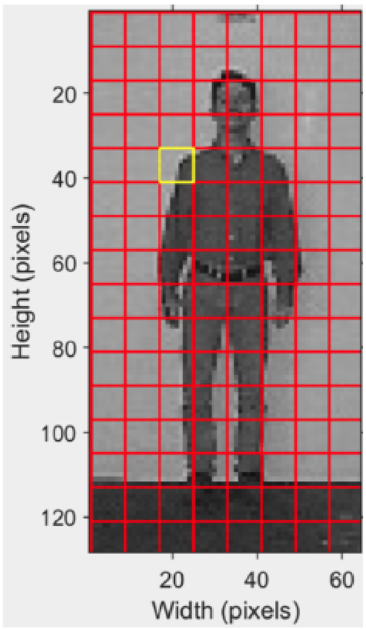

Figure 4: A 128x64 image divided into a grid of 8x8 pixel sized cells. Source: [27]

Each pixel in a cell contributes to the cell's local feature vector of size ω, or, as the author put it, a histogram with ω orientation bins, where the bins are evenly spaced over a 0°-180° "unsigned" gradient, , as shown in figure 5 where θ of each pixel's computed gradient determines which oriented bin, j (from equation 2.8), will receive the computed gradient's magnitude or vote.

$$j = \left\lfloor \left(\frac{\theta\omega}{180} \right) - \frac{1}{2} \right\rfloor \tag{2.8}$$

Vote Value									
Bin Index, j	0	1	2	3	4	5	6	7	8
Bin Boundaries	[0,20]	[20,40]	[40,60]	[60,80]	[80,100]	[100,120]	[120,140]	[140,160]	[160,180]

Figure 5: A histogram with 9 equally distributed bins. Source: Image by me

While it is also viable to use a "signed" gradient with a range of 0° -360°, it is generally unnecessary to know the sign of a gradient orientation since, as mentioned before, object classification is mainly based on edge detection. Both gradients of orientations 90° and 270° convey the same general trend of changing pixel intensity [27]. The original HOG authors show that "signed" gradients, while being uninformative, also decrease performance specifically in pedestrian detection [7], presumably because the wide range of clothing and background colour intensities obfuscate the general shape.

2.1.4 Block Normalisation

The magnitude of gradients can vary widely depending on local variations in illumination and foreground-background contrast. The authors of HOG thus found that local contrast normalisation significantly contributes to classifier performance [7], likely because it allows the classifier to focus on the structure of objects (like edges and gradients) rather than brightness changes. It also ensures contrast invariance, balancing the influence of gradients in both high and low-contrast areas, preventing overemphasis on certain regions. Furthermore, normalization smooths the feature representation, reducing noise and making the extracted features more consistent across the image. By locally adapting to different image regions, normalization helps the classifier identify meaningful patterns and essential details.

Local contrast normalisation is done by grouping the histograms of cells into a single unnormalised descriptor vector, $\vec{f_b} = \{b_i \mid i = 1, 2, \ldots, c_w \cdot c_h\}$ (where c_w represents the number of pixels in a cell's row and c_h represents the number of pixels in a cell's column). Afterwards, one of the popular block normalisation schemas [7], namely L1, $\text{L1} - \text{sqrt}$, L2 and $\text{L2} - \text{hys}$ is applied to $\vec{f_b}$, as illustrated in figure 6

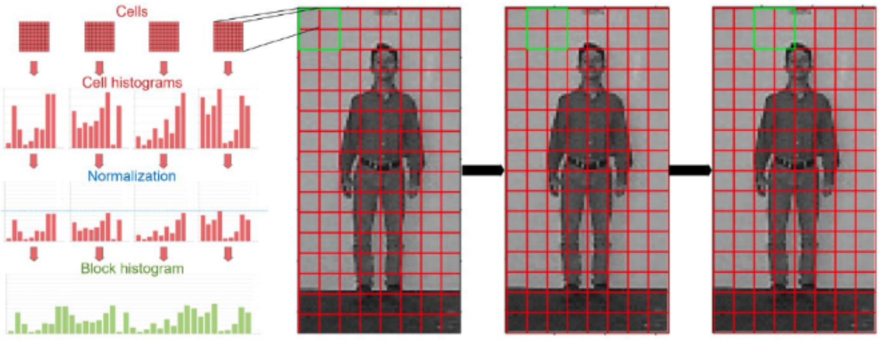

Figure 6: Construction of histogram blocks of size (2,2). Source: [27]

One essential feature of grouping cell histograms into blocks is that the blocks themselves may overlap. Depending on the stride with which the block window moves, the horizontal and vertical overlaps will be $(1 - \frac{\text{block width}}{\text{horizontal block stride}})\%$ and $(1 - \frac{\text{block height}}{\text{vertical block stride}})\%$ respectively. While normalising the same histograms in different block contexts may seem redundant, the authors of HOG found that the increased number of descriptor vectors $\vec{f_b}$ significantly improved performance [7].

2.1.5 Feature Vector Dimensionality

A sliding detection window is essential for object detection tasks like pedestrian classification because it allows the classifier to systematically examine all parts of the image at various positions and scales. Objects of interest, such as pedestrians, can appear at different locations, sizes, and orientations within an image, making it crucial to have a method that can effectively search across the entire image space. The sliding detection window of dimensions W_h and W_w scans the image in a grid-like fashion, shifting over both horizontal and vertical axes. At each location, the window encompasses a region of interest containing a dense grid of overlapping blocks.

As the window moves across the image, the feature descriptors $\vec{f_b}$ within each block's region are computed, normalized, and combined into a larger feature vector, $\vec{L}$, as illustrated in figure 7. The vector $\vec{L}$, representing the entire sliding window at that position, is used as input to the linear Support Vector Machine classifier to decide whether the window contains a pedestrian or not.

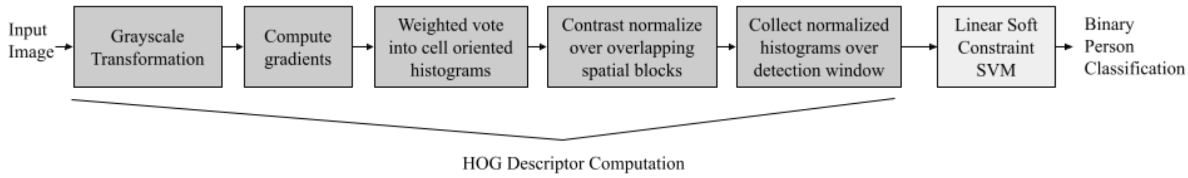

Figure 7: An overview of the HOG feature extraction chain. Source: Adapted by me from [7]

The dimensionality, d, of the vector $\vec{L}$ in essence describes the the total number of individual features, where each feature represents the direction of gradients in a specific region of the image. Formally, it is said that the vector $\vec{L}$ belongs in a feature space of d dimensions ($\vec{L} \in \mathcal{R}^d$). The higher the dimensions of this space, the more information a model has to distinguish between a pedestrian and the background or a humanoid silhouette.

If we were to restrict the possible spatial block region's horizontal, b_w, and vertical, b_h, dimensions to even numbers, it could be easily expressed that the center coordinates, x and y, of any block are bounded within $\left[\frac{b_w}{2}; \frac{W_w}{c_w} - \frac{b_w}{2}\right]$ and $\left[\frac{b_h}{2}; \frac{W_h}{c_h} - \frac{b_h}{2}\right]$ sets of cell values, respectively, as illustrated in figure 8

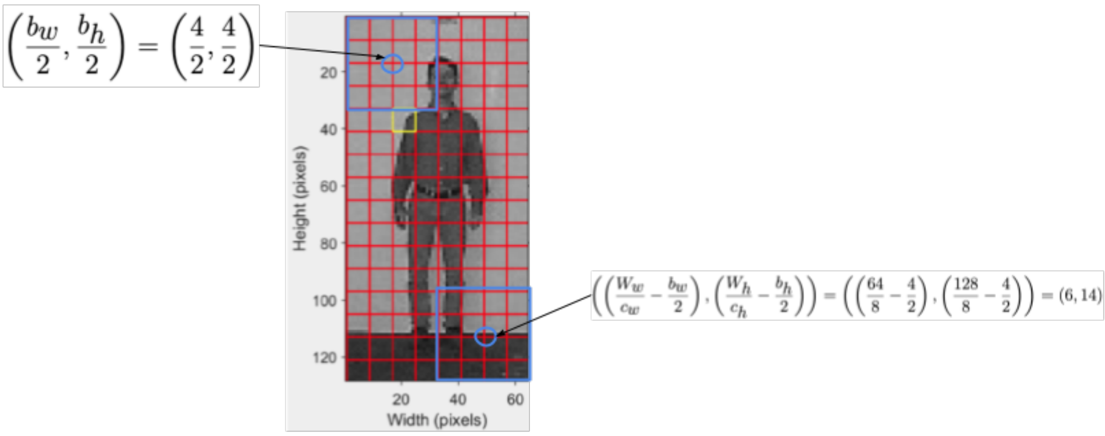

Figure 8: A 128x64 sized image with cells that contain 8x8 pixels and blocks that contain 4x4 cells. The top left-most and bottom-right most block coordinates are each expressed using the aforementioned bounds. Source: Adapted by me from [27]

Since the dimensions of each feature descriptor $\vec{f_b}$ are defined by the number of cells that comprise that descriptor ($c_w \cdot c_h$) and the number of orientation bins (ω) that each cell's histogram contains, and since the total number of descriptors combined to $\vec{L}$ is equal to the number of blocks (with horizontal and vertical strides of s_w and s_h) in the window, it follows that the dimensionality d of the resultant feature vector $\vec{L}$ is a combination of cell size, the number of orientation bins, block size, block stride values, and the size of the window itself, as shown in equation 2.9

$$\begin{aligned} d &= \left\lfloor \frac{\frac{W_w}{c_w} - 2 \cdot \frac{b_w}{2} + 1}{s_w} \right\rfloor \left\lfloor \frac{\frac{W_h}{c_h} - 2 \cdot \frac{b_h}{2} + 1}{s_h} \right\rfloor \cdot b_w b_h \omega \\ &= \left\lfloor \frac{W_w - c_w(b_w - 1)}{s_w c_w} \right\rfloor \left\lfloor \frac{W_h - c_h(b_h + -1)}{s_h c_h} \right\rfloor b_w b_h \omega \end{aligned} \tag{2.9}$$

2.2 Supervised Machine Learning

Machine Learning (ML), on a surface level, is the study of algorithms that are designed to produce outputs without an explicit instruction set generated by a person but rather with reference to the patterns or correlations found in data [21].

In that respect, Supervised ML algorithms are a subset of ML algorithms which attempt to make predictions from data [31]. Such algorithms rely on labeled training datasets, or data sets which provide the correct outputs that an algorithm should produce for each input data point [31].

Supervised ML applications include classifiers, such as a pedestrian detection program, which learn from previously annotated data in the hope of predicting the "class" to which future input data will belong. [24]. For example, a good pedestrian classifier should be able to predict whether an image's window belongs to the class of windows that contain a pedestrian or to the class of windows that do not contain a pedestrian.

Formally, classifier training data is defined as $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, where each x which belongs to a d-dimensional feature space [31] ($x \in \mathcal{R}^d$) and each y belongs to a label space [31] ($y \in \mathcal{C}$). A label space is simply the set of the possible labels or classes to which a data point might belong. Given that the goal of this investigation is to construct such a descriptor which optimises the detection of a pedestrian, the label space contains two labels $+1$ and -1, as required for binary classification [32]. Also notice that x, a data point in D, matches the definition of a fully constructed HOG feature vector $\vec{L}$, meaning that whenever the dimensionality of $\vec{L}$ changes, as defined in section 2.1.5, a new classifier model will have to be trained on a data set which contains points that belong in the appropriate dimension space.

2.3 Support Vector Machines

Support Vector Machines (SVM) is one of the most popular supervised machine learning (ML) algorithms [4] [24]. While there are many types of ML algorithms that can perform classification, such as decision trees [34], naïve bayes [36] and deep learning networks [38], SVMs have become widely adopted because of how effectively they handle high dimensional feature spaces [22]. In classification SVMs are highly regarded for their versatility that extends across multiple data science scenarios [24], like brain disorders research [24], neuroimaging [15] and, of course, pedestrian detection [7].

An SVM decision function can be precisely described as the optimal boundary, or hyperplane (defined through an optimised weight, w, and bias, b, as a set of points such that $\mathcal{H} = \{x | w^\top x + b = 0\}$ [39]), that serves to separate, or classify, data points belonging to one class from another based on the data points' features [24]. The SVM model differs from other approaches that seek to find such a seperating hyperplane (for example, the Perceptron algorithm [35]) in that the SVM attempts to find a hyperplane with the maximum margin between data points closest to the plane (which are called support vectors) [22], as illustrated in figure 9 where the hyperplane is a straight line with a weight that is orthogonal to the line and a bias that is the y intercept of the line.

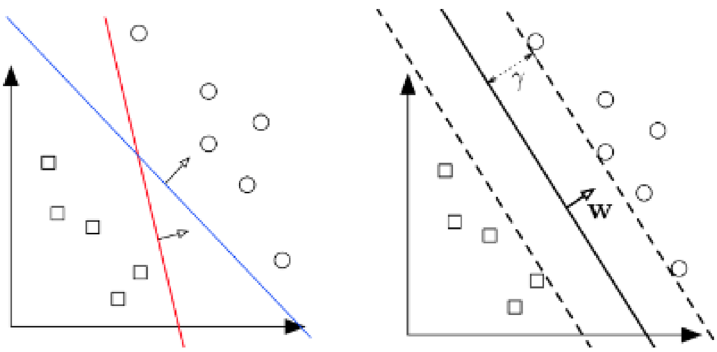

Figure 9: A 2 dimensional space, where each data point has 2 features (one abscissa and one ordinate component) (Left:) Two different separating hyperplanes for the same data set (the multiple possibles hyperplanes of, for example, the perceptron algorithm). (Right:) The maximum margin hyperplane (the only possible hyperplane of the SVM algorithm). Source: [39]

A hyperplane with the maximum possible margin between its support vectors is incredibly useful as it increases the likelihood of producing a generalized classifier, which can accurately seperate unseen data points [32]. By expressing the distance between any point and that point's projection in the hyperplane, as illustrated in figure 10, with the two variables that define the hyperplane itself (the weight and the bias), we get a definition of the margin, γ in 2.10 [39]

$$\gamma(w, b) = \min_{x \in D} \frac{|w^\top x + b|}{||w||_2} \tag{2.10}$$

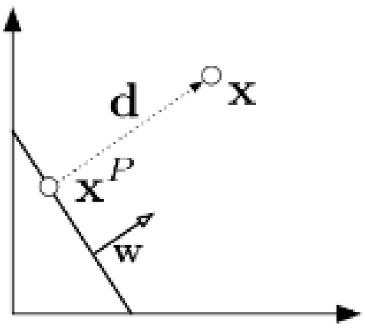

Figure 10: The projection of a data point onto the hyperplane. Source: [39]

With the expression in 2.10, the abstract goal of finding the hyperplane "of best fit" becomes a very concrete optimization problem which seeks to obtain such a weight w and bias b that the margin γ is maximised while maintaining the constraint that the data points of each class must lie on the correct sides of the hyperplane. Mathematically, this constraint is the inequality in equation 2.11 [22], since plugging in any data point x_i into the the equation $w^\top x + b$ will yield an output that is either ≥ 0 or ≤ 0. For positive outputs, the data point (or the input into the equation) will be above the hyperplane, $\mathcal{H}$, so we should expect that data point's label y_i to also be positive, and for negative outputs it, where x_i is below $\mathcal{H}$, a negative label should be expected.

$$y_i(w^\top x_i + b) \geq 0 \tag{2.11}$$

2.3.1 Soft SVM Constraints

Traditionally obtaining the largest possible margin γ would be a quadratic programming problem [2] (as the goal of maximising γ is primarily anchored around minimizing $||w||_2$ from the equation in 2.10 with the linear constraint in 2.11). While an SVM model's hyperplane with the hard constraint in 2.11 could, in theory, be found found using either QCQP [39] or SMO [4] algorithms, in practice pedestrian datasets are incredibly noisy, as mentioned in section 1, while also containing humanoid figures which closely approximate the features of a pedestrian [29], as visualized in figure 12. Because of noise and obfuscation in real world pedestrian data, an SVM with a hard linear constraint would fail to compute the optimal hyperplane as there would be a significant number of outliers or data points which share features common to both classes, as illustrated in figure 11. Instead, in the hope of finding a hyperplane that achieves the best realistically possible classification accuracy, an SVM with a soft constraint, which does allow for some degree of error while maximising γ, ought to be used [7] [33].

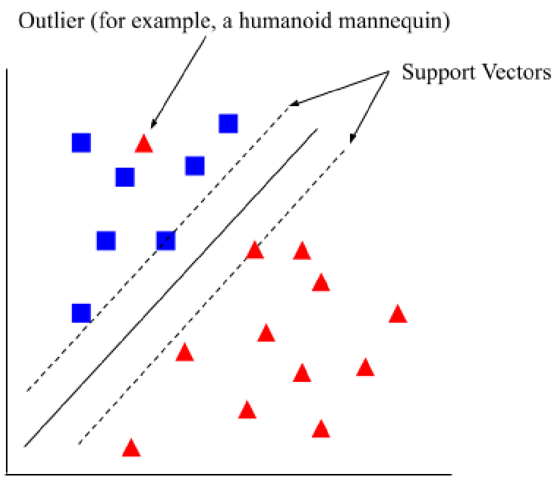

Figure 11: A Data set with two classes and an outlier. Source: Image by me

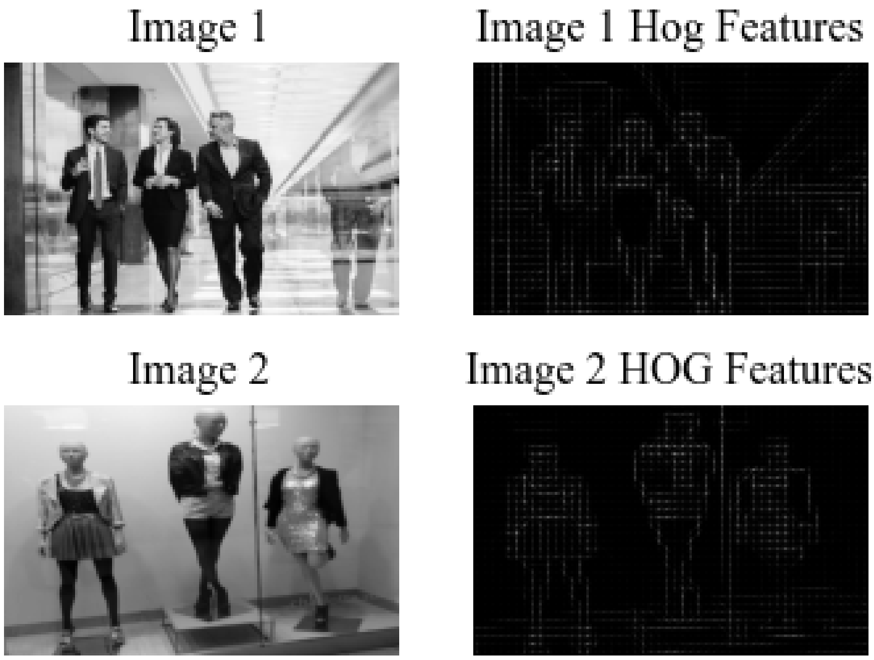

Figure 12: (Image 1:) An Image containing three people/pedestrians in a building. Source: istockphoto.com (Image 2:) An Image containing three mannequins in a store window. Source: theshopcompany.com (Image 1 and 2 Hog Features): Computed HOG Features of Image 1 and Image 2. Source: Image by me

3 Methodology

3.1 Dependant Variables

As mentioned in section 2.2, whenever any of the components of a feature vector's dimensionality, as defined in 2.9, changes, a new model has to be trained. The values for which various sets of HOG parameters will be tested in this investigation are listed in table 1. Notice that the use of a "holistic" derivative mask, as introduced in section 2.1.2 and implemented in appendix A.1.3, is also listed as a dependent variable. While the derivative mask which is used does not change a vector's dimensions it does change the vector's shape and, given the novel approach, it is nonetheless important to test how an SVM reacts to a differently shaped HOG descriptor.

Parameter	*Values*
Window Dimension Pairs (W_h,W_w)	(100, 50), (128, 96), (128, 64), (112, 48)
Cell Histogram Bin Counts (ω)	9, 13, 18
Cell Dimension Pairs (c_w,c_h)	(4,4), (6,6), (8,8), (10,10)
Block Dimension Pairs (b_w,b_h)	(1,1), (2,2), (3,3), (4,4)
Block Stride Dimension Pairs (s_w,s_h)	(1,1), (2,2), (3,3)
Holistic Derivative Mask (appendix A.1.3)	True, False

Table 1: Dependent variables for the experiment

The only restriction on the values in table 1 that can be combined to a set of HOG parameters is $b_w \geq s_w$ and $b_h \geq s_h$, since the use of blocks with stride values greater than block dimensions would result in certain cells being simply ignored for in the resultant feture vector $\vec{L}$. With the restriction, the number of different sets of values is given by N in equation 3.1.

$$\begin{aligned} N &= |\{(W_h, W_w)\}| \times |\{\omega\}| \times |\{(c_w, c_h)\}| \times 2 \times \sum_{\substack{b_w \geq s_w \\ b_h \geq s_h}} |\{(b_w, b_h)\}| \times |\{(s_w, s_h)\}| \\ &= 4 \times 3 \times 4 \times 2 \times 9 = 864 \end{aligned} \tag{3.1}$$

3.2 Data Sets

3.2.1 Labeled Pedestrian Data Set Sources

Many past studies which have evaluated the HOG approach to feature detection have heavily [12] or, in some cases [40], solely relied on the INRIA pedestrian dataset [1], as it has been the most popular data set for pedestrian detection algorithm evaluation [11] since HOG features were first introduced [7]. Nevertheless, there are flaws with the data set, mainly in the limited annotation: many people which appear in test images are not labelled, estimates of each person's visibility are lacking, and there are no class labels for the regions of the images that contain ambiguous objects [29]. Matteo Taiana et al introduced an improved iteration of INRIA with labelling that addresses the aforementioned issues and, as such, their improved INRIA data set [2] will be used in this essay's experiment.

Aside from the shortcomings of labelling in INRIA, the dataset is biased toward large, mostly unoccluded pedestrians [10]. The majority of people found in the dataset's images are at a scale such that their limbs are 6 to 8 pixels wide [7], which can undoubtedly introduce confirmation bias when attempting to evaluate the most performant cell size. As the goal of this investigation is to find a HOG descriptor that performs the best in real world environments, a greater variety of scales and occlusions will be introduced with the use of the more challenging and larger Caltech Pedestrian Dataset [10], which contains richly annotated, low-resolution images of frequently occluded people. Images in real world applications may also include objects, like mannequins or statues, which closely resemble humanoid features, as previously shown in figure 12. Neither INRIA nor the Caltech datasets contain such objects and thus a different dataset which addresses the

[1] URL for the INRIA dataset (the original web page which provided the data set is, as of 2024 October 23rd, not accessible, thus a copy from kaggle is used): https://www.kaggle.com/datasets/jcoral02/inriaperson.

[2] URL for the improved inria labels: http://users.isr.ist.utl.pt/~mtaiana/data.html.

range of false positive in pedestrian detection by providing labelled images with "person-like" objects [16] is also used in the investigation.

3.2.2 Caltech Data Set Transformation

The PASCAL VOC challenges [13] introduced numerous standards in image classification, including the Pascal VOC labelling format, which has become the preferred scheme in many object classification applications, including pedestrian detection [11]. Both INRIA and the PnPLO (person-like) datasets abide this format, however the Caltech data set, since it's comprised of annotated videos rather than images, uses video bounding box labels [20], which are especially useful for applications which involve tracking. This investigation, however, is only concerned with the detection of a pedestrian in an image, and because of that, the video (seq) files and video bounding box annotation (vbb) files are converted to images and Pascal VOC format xml files (appendix A.1.5).

Besides differences in annotation, the Caltech data set videos contain $\sim 250,000$ frames [10], which vastly outnumbers the 1085 images in INRIA [7] and 1339 images in PnPLO [16]. Given both the great quantity of data in Caltech frames and the large amount of models (864 from equation 3.1) that would need to be trained on that data, it becomes apparent that to obtain a training time that is feasible for the computational resources that can be utilised in this investigation, the amount of frames needs to be reduced.

The total running time of the Caltech videos is $\sim$ 10h [10], this gives a frame per second rate of $\sim$ 7 frames/s. Since a person is present in a video for $\sim$ 5s [10], we can approximate that each identifiable individual will, on average, be present in 34 frames and thus retaining only the 30th frame of each video, as done in appendix A.1.6, should not incur a greatly significant cost on the amount of unique training data. By also removing frames that include the label "person?" (line 193 of appendix A.1.5), which denotes ambigious pedestrian figures, the sum of Caltech frames is significantly reduced to 8538.

3.2.3 Window Size Samples

Dalal and Triggs proposed evaluating a detector by classifying cropped windows centered on pedestrians and comparing them to windows sampled at a fixed density from non-pedestrian images [7], thereby eliminating the need to merge nearby detections, using methods like non maximal suppression (NMS), or other post-processing steps. Figure 13 shows a high level overview of per-window data set preparation.

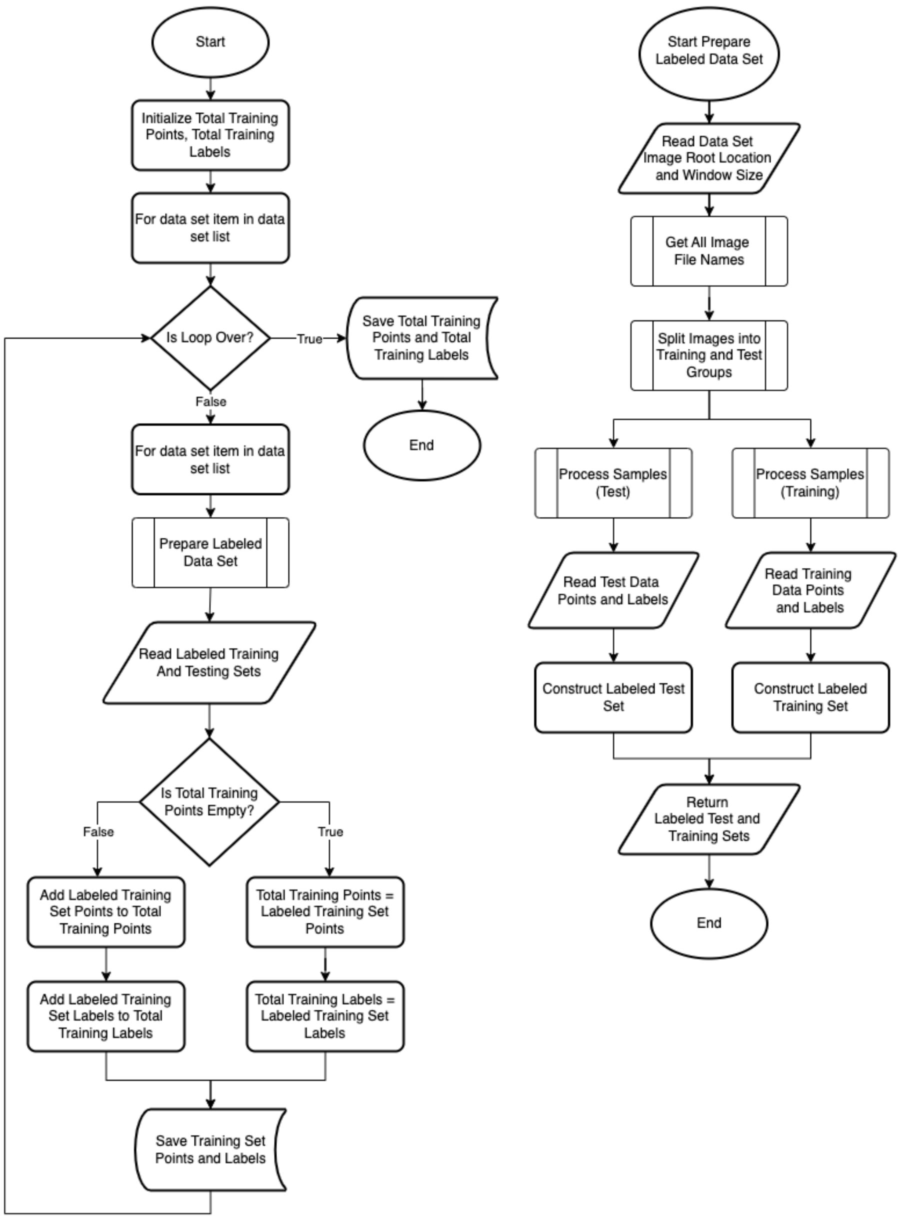

Figure 13: A high level overview flowchart of the process of initializing and saving the total training points and labels alongside each data set's testing points and labels

2 major concerns, however, have been raised with per-window evaluation:

1. NMS may reduce the number of false positives at varying rates for different detection methods [10]

2. The per-window scheme usually relies on the use of cropped positives (windows where a pedestrian is neatly bounded) and uncropped negatives (windows that are not specifically cropped to contain random objects or bacgkround scenery). Classifiers may exploit this window boundary effect as discriminative features leading to good per-window performance but poor performance in real life applications [10]

While concern nr. 1 should not impede this investigation's goal of finding the optimal HOG parameters, as each instance of HOG interacts in a similar fashion with NMS [7], concern nr. 2 is addressed to some degree in line 162 of appendix A.1.7 by applying random value paddings to the bounding boxes that comprise positive samples. This process is further explained in figure 14.

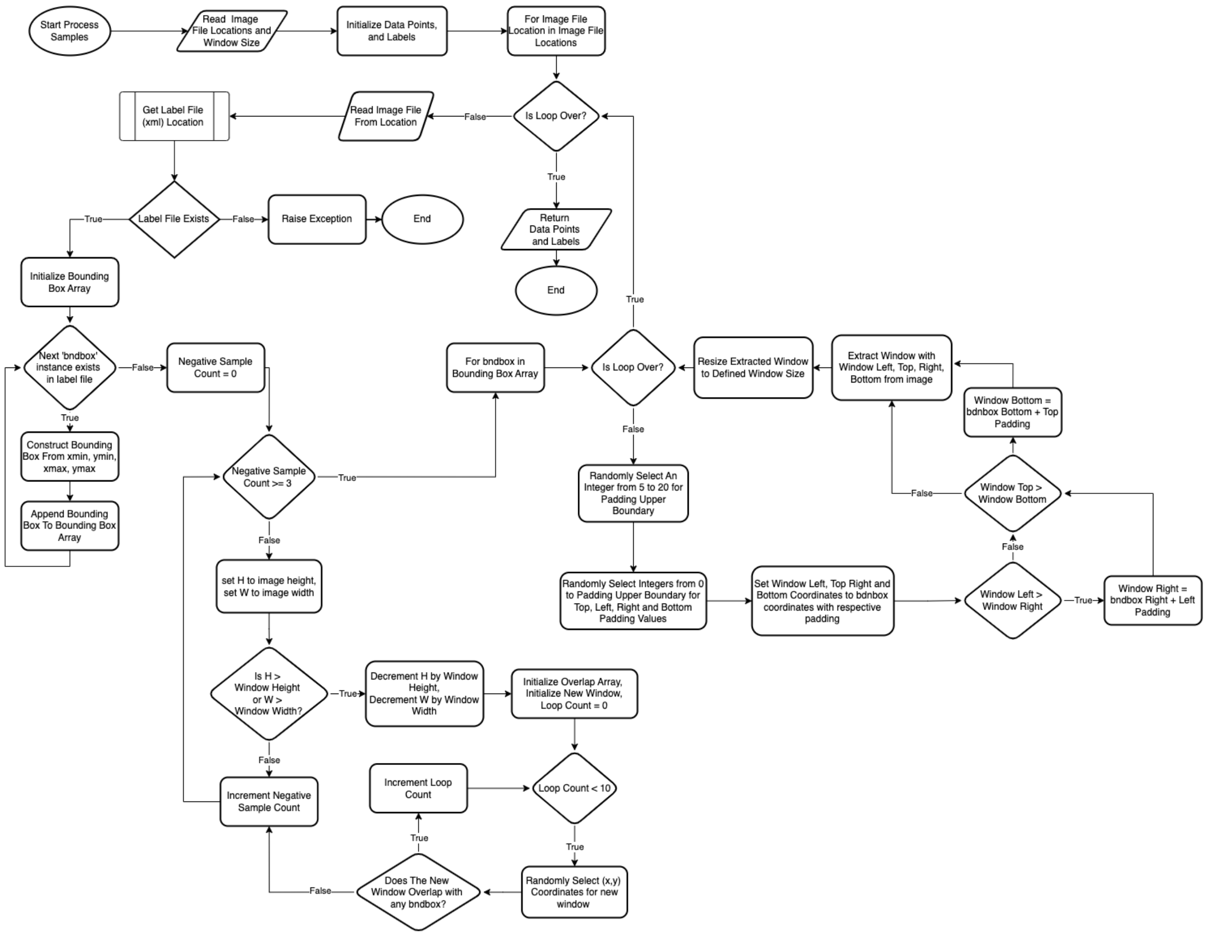

Figure 14: A flowchart of the process of extracting the positive data samples (with some degree of random padding to avoid cropped positive bias) and the process of constructing negative samples

By using an 80/20% training-testing data split, the number of images from section 3.2.2 yields the numbers of different window size samples, as specified in table 2

3.3 Evaluation Metrics

3.3.1 The Basic Confusion Matrix Rates

In essence, all evaluation metrics of binary classification rely on the values of the confusion matrix, a 2×2 contingency table where the positive elements correctly classified as

Window Set	*Positive*	*Negative*
INRIA Testing	361	543
Caltech Testing	2195	2558
PnPLO Testing	596	578
Total Training	12794	14760

(a) Window Size (100, 50)

Window Set	*Positive*	*Negative*
INRIA Testing	361	533
Caltech Testing	2195	2548
PnPLO Testing	596	475
Total Training	12794	14185

(b) Window Size (128, 96)

Window Set	*Positive*	*Negative*
INRIA Testing	361	540
Caltech Testing	2195	2554
PnPLO Testing	596	535
Total Training	12794	14511

(c) Window Size (128, 64)

Window Set	*Positive*	*Negative*
INRIA Testing	361	543
Caltech Testing	2195	2558
PnPLO Testing	596	574
Total Training	12794	14731

(d) Window Size (112, 48)

Table 2: Positive And Negative Window Samples For Each Data Set at Each Window Size.

positives are called true positives (TP), the negative elements wrongly classified as positive are called false positives (FP), the negative elements correctly classified as negatives are called true negatives (TN), and the positive elements wrongly classified as negatives are called false negatives (FN), as shown in figure 15. [6].

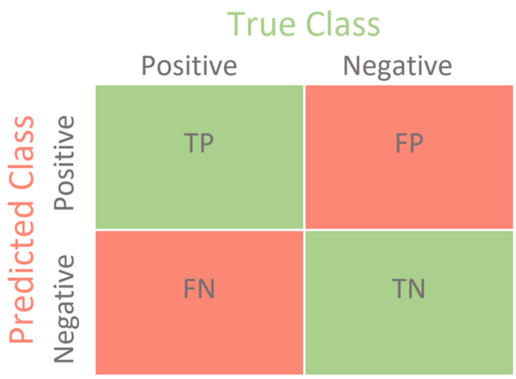

Figure 15: An example of a confusion matrix for binary classification. Source: [1]

The four basic rates for confusion matrices are as follows [6]:

1. Sensitivity, or True Positive Rate, TPR $= \frac{\text{TP}}{\text{TP+FN}}$
2. Specificity, or True Negative Rate, TNR $= \frac{\text{TN}}{\text{TN+FP}}$
3. Precision, or Positive Predictive Value, PPV $= \frac{\text{TP}}{\text{TP+FP}}$
4. Negative Predictive Value, PPV $= \frac{\text{TN}}{\text{TN+FN}}$

3.3.2 Confidence Threshold Curves

Many scoring classifiers produce a real-valued prediction score for each data point and, by assigning a particular threshold value τ a confusion matrix is generated for such a classifier [5]. To summarize the confusion matrix, it is common to to plot one of the aforementioned four basic rates on a cartesian plane at varying τ values, like plotting a ROC curve (where TPR is plotted against the false positive rate FPR $= \frac{\text{TN}}{\text{TN+FP}}$) or the DET curve (FN rate agains FP rate) which is more widespread in pedestrian detection literature [7] [11].

However, unlike methods such as logistic regression [37], which classify a window into one of two classes by estimating the probability that the window belongs to each class, an SVM is not a "probabilistic" model as it simply plots the window's feature vector in a space separated by a hyperplane, and thus there's no probabilistc/scoring confidence τ involved. While it's possible to compute the probabilities of an SVMs prediction using cross-validation in Platt Scaling [25], the operation is known to be very expensive for large datasets [8] alongside being inconsistent with the actual predictions of the SVM [8]. Nevertheless, plots with varying τ values can be extremely informative [19] [8] and thus instead of "probabilities", the distances from each data point to the hyperplane are used a sort of "confidence" value.

3.3.3 Matthew's Correlation Coefficient

While ROC curves (or they DET counterpars) alongside the scalar value of area under the ROC curve (AUC-ROC) are very widespread, they are also fundamentally flawed in that they ignore precision since, fundamentally, AUC-ROC only identifies how well a classifier separates the positive class from the negative class, not how accurate the separation is (a metric which is ever more important in a field like pedestrian detection). Historically, precision recall curves were used to account for the drawbacks of ROC [5]. Quite recently, however, the Matthew's Correlation Coefficient has been proposed as a standard metric for validating biomedical image analysis by an international group of researchers in the field [18], primarily because it is the only rate that maximizes all four of the aforementioned basic rates [18] [6] [5] and is claimed to be the most informative single score to establish the quality of a binary classifier prediction [6]. Because of its discriminatory power, the MCC and a corresponding MCC-F1 curve (explained in more detail in figure 16) will be the primary evaluation metrics used in this investigation.

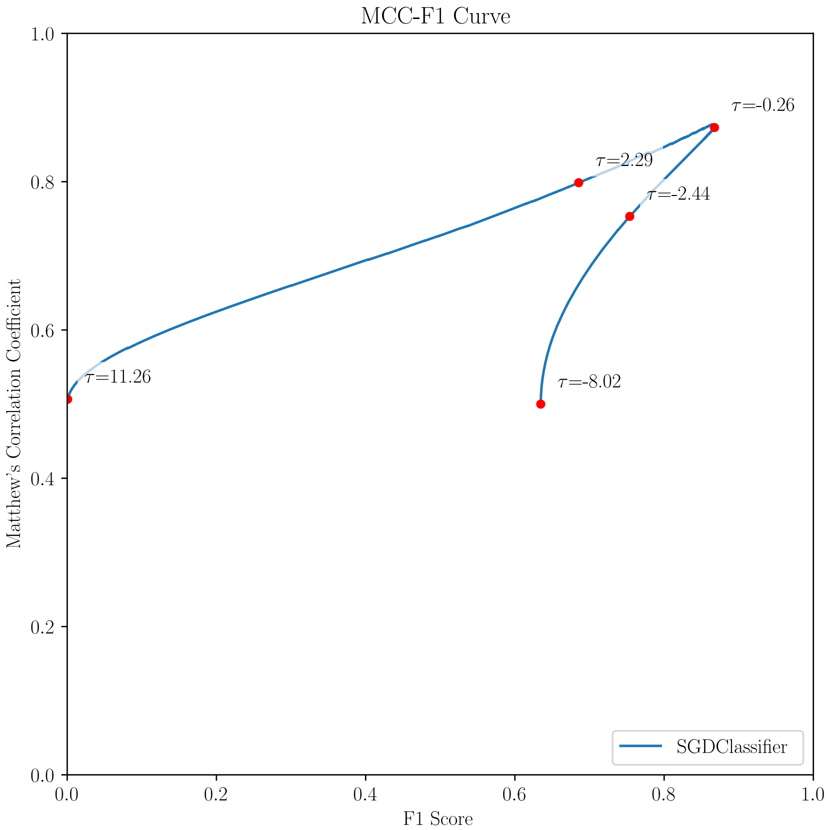

Figure 16: An example of an MCC-F1 curve. Unit-normalized Matthews correlation coefficient (MCC) plotted against the F1 score (the harmonic mean between precision and recall). The random line indicates that a random classifier can achieve a unit-normalized MCC of 0.5. The point of perfect performance is (1,1), representing an ideal classifier that correctly classifies every instance. Conversely, the point of worst performance is (0,0), attained by a classifier that misclassifies all instances. The best threshold point is the location on the curve that is nearest to (1,1). 5 various threshold τ values are scattered along the curve. Source: Image by Me, generated with code in appendix A.1.9

$$\mathrm{MCC} = \frac{\mathrm{TP} \cdot \mathrm{TN} - \mathrm{FP} \cdot \mathrm{FN}}{\sqrt{(\mathrm{TP} + \mathrm{FP}) \cdot (\mathrm{TP} + \mathrm{FN}) \cdot (\mathrm{TN} + \mathrm{FP}) \cdot (\mathrm{TN}) + \mathrm{FN}}}$$

Figure 17: The equation for Matthew's Correlation Coefficient. The values of MCC are bounded within the range $[-1; 1]$, where 1 represents a perfect prediction, 0 represents random prediction and -1 total disagreement between prediction and observation. Refer to [5] regarding the necessary normalization to make the MCC values bounded within $[0; 1]$ so that they can be plotted against F1 scores (which themselves are bounded in $[0; 1]$)

Nevertheless, since much of the literature on pedestrian detection and classification has historically relied on the aforementioned metrics of AUC-ROC, Average Precision and simple Accuracy [7] [11], they are retained to facilitate direct and simple comparison with previous studies.

3.3.4 McNemar's Test for Pairwise Classifier Comparison

There are many ways to perform pairwise classifier comparison, such as conducting 5×2 Cross Validation (CV), which has historically been the preferred scheme in object classification [9]. However, 5×2 CV, as the name implies, needs to be executed 10 times, while a test like McNemar's requires only a single execution. McNemar's test is also a more attractive choice as it performs increasingly better with larger datasets [26]. Additionally, it utilizes a version of the familiar confusion matrix, illustrated in Figure 18.

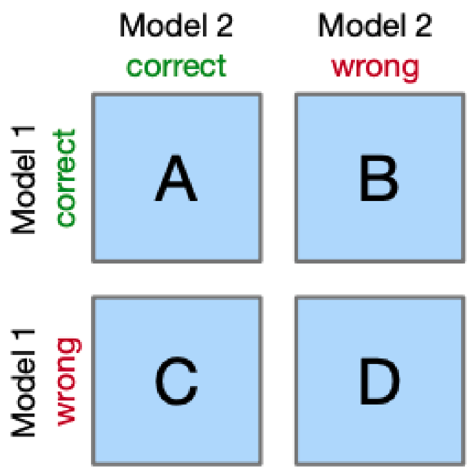

Figure 18: Confusion matrix layout in the context of McNemar's test. Source: [26] Code for the construction of such a matrix can be found in appendix A.1.11

McNemar's test checks if two classifiers have significantly different performance by comparing their disagreement on predictions in the confusion matrix. It calculates a p-value, the probability that the observed difference in performance is due to chance, based on a chi-square statistic [9]. Typically, all p-values ≥ 0.05 indicate that the difference between performance is not significant [26] [9].

3.4 Model Preparation

As mentioned in sections 2.2 and 3.1, a data set has to be uniquely prepared for each of the different 864 SVM models. This is done in two steps: by first preprocessing each window sample and then computing the HOG features (data points) on which a model will be trained and tested.

3.4.1 Preprocessing: Grayscale Image Transformation

The only preprocessing step used in the original HOG paper was gamma/color normalization [7]. While the paper did show that there are modest variations in classifier accuracy depending on whether an RGB, LAB or grayscale colour space is used, it was also shown that the difference in illuminance became even more negligible once block normalization was applied [7]. Thus, for the sake computational simplicity, 3-channeled data points are first transformed to grayscale color spaces.

Given the rather ambiguous nature of assessing which specific method of RGB to grayscale conversion produces universally desirable outputs for all involved input images [3], a simple and widely adopted colour mapping defined in equation 3.2 is used in appendix A.1.1

$$Y \leftarrow 0.2125 \cdot R + 0.7154 \cdot G + 0.0721 \cdot B \tag{3.2}$$

3.4.2 Computing HOG Features

While an in depth explanation of how HOG features are computed was presented in section 2.1, there are a few notes to be made regarding the implementation of HOG in this investigation.

Since neither the scikit-image nor OpenCV libraries provide an implementation of HOG which would allow changing the block stride values, a custom implementation of the algorithm can be found in appendix A.1.4, with figure 19 providing a technical overview. The two parts of the hog pipeline (from figure 7) that have still been reused from scikit-image are the distribution of votes to histogram bins and block normalisation, as shown in figure 19. This is primarily because both parts are highly optimised using Cython. Even while the votes are not distributed using equation 2.8, giving a time complexity of $\mathcal{O}(\omega \cdot c_h \cdot c_w)$ instead of $\mathcal{O}(c_h \cdot c_w)$, the speed of the library's Cython implementation outperforms anything that would be possible using regular python.

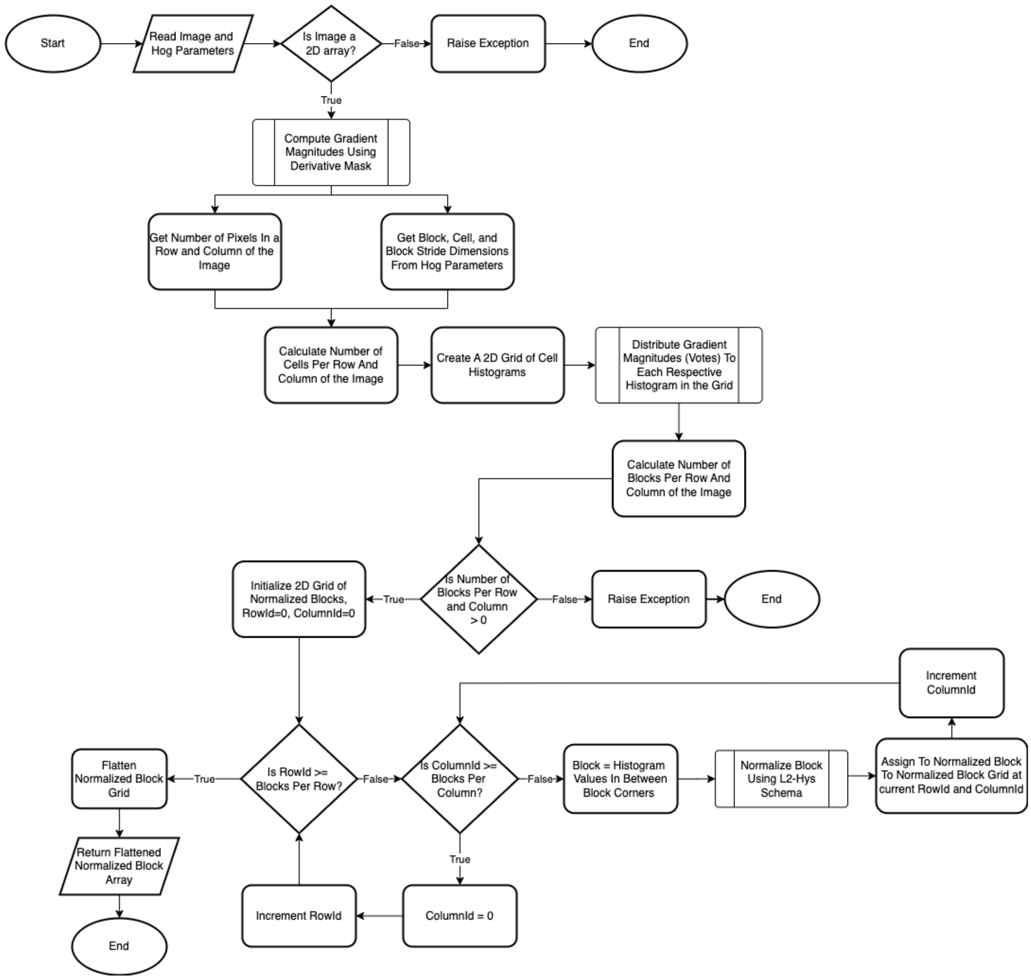

Figure 19: A flowchart of the process of computing HOG features with custom block stride values

3.4.3 Choosing an SVM

The primary factor driving the choice of SVM implementation is time of computation. Given the relatively large number of models that have to be trained on $\sim$ 27,000 samples, an implementation which is able to maximize the hyperplane's margin in the least amount of time while still maintaining relatively decent classification performance is a necessity.

The standard SVM implementation is LibSVM [4] [8], however, it's training times scale quadratically with the number of samples [8] (in practice it took $\sim$ 10 hours to train a single LibSVM model on $\sim$ 27,000 samples). The maintainers scikit-learn recommend using either LibLinear or their own implementation of a linear SVM with stochastic gradient descent (SGD). SGD only uses a subset of samples when determining the cost function's, which, in this case, has inputs of $||w||_2$ and b from section 2.3.1, gradient and the subsequent direction towards the global minima [30]. This is in contrast to regular gradient descent (GD) which uses all samples for gradient calculation. As such, while training a model with SGD would be faster, we should also expect the SGD model to have worse performance guarantees than GD [30].

In practice however, both LibLinear [3] and an SVM with GDC [4] exhibit essentially identical pedestrian classification performance, as evidenced by a McNemar's test p-value of ~ 0.121, further comparisons are made in table 3 and figure 20.

Detector on Dataset	MCC	Accuracy	F1 Score	FPPW	AUC-ROC	AP
LibLinear on INRIA	0.756	0.877	0.858	0.093	0.961	0.948
SGD on INRIA	0.748	0.871	0.853	0.101	0.959	0.944
LibLinear on caltech_30	0.784	0.893	0.882	0.046	0.958	0.958
SGD on caltech_30	0.781	0.891	0.882	0.053	0.955	0.954
LibLinear on PnPLO	0.649	0.825	0.832	0.082	0.910	0.916
SGD on PnPLO	0.627	0.814	0.824	0.094	0.898	0.903

Table 3: The evaluation metrics of a LinearSVC and SGDClassifier SVM implementations, trained on the standard HOG feature parameters [7]: 128×64 windows with 8×8 pixels per cell, 2×2 cells per block, 1×1 block strides. Source: Image by Me, generated with code in appendix A.1.10

.

[3] LibLinear SVM docs: https://scikit-learn.org/1.5/modules/generated/sklearn.svm.LinearSVC.html

[4] SVM with SGD docs: https://scikit-learn.org/1.5/modules/generated/sklearn.linear_model.SGDClassifier.html

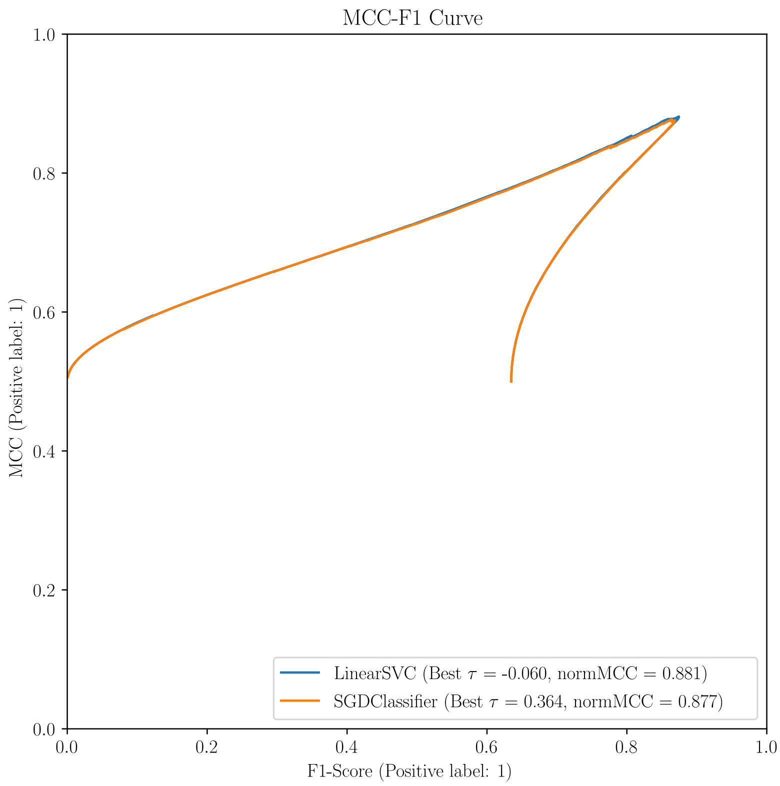

Figure 20: An MCC-F1 curve of both LinearSVC and SGDClassifier trained on the standard HOG feature parameters [7]. Notice that the best performing τ value for SGDClassifier is negative, as τ identifies the distance which allows a point to be classified as a positive. This relates to Soft Constraint SVMs mentioned in section 2.3.1.

References

[1] Arda Aras. *Explaining what learned models predict: In which cases can we trust machine learning models and when is caution required ?* Mar. 2020.

[2] D. Bertsekas. *Lecture 19: Linear and Quadratic Programming Duality.* Massachusetts Institute of Technology. URL: https://see.stanford.edu/materials/aimlcs229/cs229-notes3.pdf (visited on 10/17/2024).

[3] M. Ĉadík. "Perceptual Evaluation of Color-to-Grayscale Image Conversions". In: *Computer Graphics Forum* 27 (Oct. 2008), pp. 1745–1754. DOI: 10.1111/j.1467-8659.2008.01319.x.

[4] Chih-Chung Chang and Chih-Jen Lin. "LIBSVM". In: *ACM Transactions on Intelligent Systems and Technology* 2.3 (Apr. 2011), pp. 1–27. DOI: https://doi.org/10.1145/1961189.1961199.

[5] Davide Chicco and Giuseppe Jurman. "The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation". In: *BMC Genomics* 21.1 (Jan. 2020). DOI: https://doi.org/10.1186/s12864-019-6413-7. URL: https://link.springer.com/article/10.1186/s12864-019-6413-7.

[6] Davide Chicco and Giuseppe Jurman. "The Matthews correlation coefficient (MCC) should replace the ROC AUC as the standard metric for assessing binary classification". In: *BioData Mining* 16.1 (Feb. 2023). DOI: https://doi.org/10.1186/s13040-023-00322-4.

[7] 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)* 1 (2005), pp. 886–893. DOI: 10.1109/cvpr.2005.177.

[8] Scikit-learn developers. *Support Vector Machines — Scikit-learn User Guide.* Accessed: 2024-10-23. 2023. URL: https://scikit-learn.org/stable/modules/svm.html.

[9] Thomas G. Dietterich. "Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms". In: *Neural Computation* 10.7 (Oct. 1998), pp. 1895–1923. ISSN: 0899-7667. DOI: 10.1162/089976698300017197. eprint: https://direct.mit.edu/neco/article-pdf/10/7/1895/814002/089976698300017197.pdf. URL: https://doi.org/10.1162/089976698300017197.

[10] Piotr Dollar et al. "Pedestrian detection: A benchmark". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition* (June 2009). DOI: 10.1109/cvpr.2009.5206631.

[11] Piotr Dollar et al. "Pedestrian Detection: An Evaluation of the State of the Art". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34 (Apr. 2012), pp. 743–761. DOI: 10.1109/TPAMI.2011.155. URL: https://ieeexplore.ieee.org/document/5975165 (visited on 09/18/2024).

[12] Piotr Dollár, Ron Appel, and Wolf Kienzle. "Crosstalk Cascades for Frame-Rate Pedestrian Detection". In: *Lecture notes in computer science* (Jan. 2012), pp. 645–659. DOI: 10.1007/978-3-642-33709-3_46. (Visited on 09/28/2024).

[13] Mark Everingham et al. "The Pascal Visual Object Classes (VOC) Challenge". In: *International Journal of Computer Vision* 88 (Sept. 2009), pp. 303–338. DOI: 10.1007/s11263-009-0275-4.

[14] H. Faraji and W.J. MacLean. "CCD noise removal in digital images". In: *IEEE Transactions on Image Processing* 15 (Sept. 2006), pp. 2676–2685. DOI: 10.1109/tip.2006.877363. (Visited on 04/09/2019).

[15] Michael Hanke et al. "PyMVPA: a Python Toolbox for Multivariate Pattern Analysis of fMRI Data". In: *Neuroinformatics* 7.1 (Jan. 2009), pp. 37–53. DOI: https://doi.org/10.1007/s12021-008-9041-y.

[16] N J Karthika and Saravanan Chandran. "Addressing the False Positives in Pedestrian Detection". In: *Lecture notes in electrical engineering* (Jan. 2020), pp. 1083–1092. DOI: 10.1007/978-981-15-7031-5_103. (Visited on 09/28/2024).

[17] David G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision* 60 (Nov. 2004), pp. 91–110. DOI: 10.1023/b:visi.0000029664.99615.94.

[18] Lena Maier-Hein et al. "Metrics reloaded: recommendations for image analysis validation". In: *Nature Methods* 21.2 (Feb. 2024), pp. 195–212. ISSN: 1548-7105. DOI: 10.1038/s41592-023-02151-z. URL: http://dx.doi.org/10.1038/s41592-023-02151-z.

[19] A. F. Martin et al. "The DET curve in assessment of detection task performance". In: *Eurospeech*. Vol. 4. Sept. 1997, pp. 1895–1898.

[20] MathWorks. *vbbLabeler.m*. https://es.mathworks.com/matlabcentral/mlc-downloads/downloads/submissions/57221/versions/1/contents/Seq_2_AVI/code3.2.1/vbbLabeler.m. Version 1. (Visited on 10/18/2024).

[21] Andrew Ng. *Lecture 1: Machine Learning*. https://www.youtube.com/watch?v=jGwO_UgTS7I. Video Lecture. Apr. 2020.

[22] Andrew Ng. *Support Vector Mchines*. Stanford University. URL: https://see.stanford.edu/materials/aimlcs229/cs229-notes3.pdf (visited on 09/24/2024).

[23] Juan Carlos Niebles and Ranjay Krishna. "Edge Detection". In: *Stanford Vision and Learning Lab*. Lecture 5 - 1. Stanford University, Oct. 2012.

[24] Derek A. Pisner and David M. Schnyer. "Support vector machine". In: Department of Psychology, University of Texas at Austin, Austin, TX, United States, 2020. Chap. 6.

[25] John Platt. "Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods". In: *Advances in large margin classifiers*. Vol. 10. 3. Microsoft Research, 1999, pp. 61–74.

[26] Sebastian Raschka. "Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning". In: *arXiv (Cornell University)* (Nov. 2018). DOI: https://doi.org/10.48550/arxiv.1811.12808.

[27] S V Shidlovskiy et al. "Reducing dimensions of the histogram of oriented gradients (HOG) feature vector". In: *Journal of Physics: Conference Series* 1611 (Aug. 2020), p. 012072. DOI: 10.1088/1742-6596/1611/1/012072. (Visited on 04/22/2022).

[28] Freya Slootmans. *European Road Safety Observatory*. Oct. 2021. URL: https://road-safety.transport.ec.europa.eu/document/download/aaeb811d-f455-4fb0-8a79-7a373626952d_en?filename=FF_pedestrians_20220209.pdf (visited on 09/18/2024).

[29] Matteo Taiana, Jacinto C Nascimento, and Alexandre Bernardino. "An Improved Labelling for the INRIA Person Data Set for Pedestrian Detection". In: *Lecture notes in computer science* (Jan. 2013), pp. 286–295. DOI: 10.1007/978-3-642-38628-2_34. (Visited on 09/18/2024).

[30] Ryan Tibshirani. *Lecture 24: Convex Optimization*. University of California, Berkeley. URL: https://www.stat.cmu.edu/~ryantibs/convexopt-F18/scribes/Lecture_24.pdf (visited on 10/24/2024).

[31] Kilian Weinberger. *Lecture 1: Supervised Learning*. https://www.youtube.com/watch?v=MrLPzBxG95I. Video Lecture. July 2018.

[32] Kilian Weinberger. *Lecture 14: (Linear) Support Vector Machines*. https://www.youtube.com/watch?v=xpHQ6UhMlx4. Video Lecture. July 2018.

[33] Kilian Weinberger. *Lecture 15: (Linear) Support Vector Machines continued*. https://www.youtube.com/watch?v=FwYNPomeBBg. Video Lecture. July 2018.

[34] Kilian Weinberger. *Lecture 28: Ball Trees / Decision Trees*. https://www.youtube.com/watch?v=E1_WCdUAtyE. Video Lecture. July 2018.

[35] Kilian Weinberger. *Lecture 5: Perceptron*. https://www.youtube.com/watch?v=wl7gVvI-HuY&list=PLl8OlHZGYOQ7bkVbuRthEsaLr7bONzbXS&index=6. Video Lecture. July 2018.

[36] Kilian Weinberger. *Lecture 8: Estimating Probabilities from Data: Naive Bayes*. https://www.youtube.com/watch?v=pDHEX2usCS0. Video Lecture. July 2018.

[37] Kilian Weinberger. *Logistic Regression*. Cornell University. URL: https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote06.html (visited on 10/22/2024).

[38] Kilian Weinberger. *Neural Network*. Cornell University. URL: https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote20.html (visited on 10/17/2024).

[39] Kilian Weinberger. *SVM*. Cornell University. URL: https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote09.html (visited on 10/17/2024).

[40] Hongzhi Zhou and Gan Yu. "Research on pedestrian detection technology based on the SVM classifier trained by HOG and LTP features". In: *Future Generation Computer Systems* 125 (Dec. 2021), pp. 604–615. DOI: 10.1016/j.future.2021.06.016. (Visited on 12/06/2021).

[41] Zeeshan Zia. *Why are HOG features more accurate than Haar features in pedestrian detection?* 2015. URL: https://qr.ae/p24ltz (visited on 09/23/2024).

A Appendices

A.1 Python Code Implementations

A.1.1 Grayscale Transformation

```
from skimage.color import rgb2gray
import numpy as np
from tqdm import tqdm
def grayscale_transform(X):
    '''
    Convert a collection of RGB images to grayscale.

    Parameters:
    -----------
    X : list or np.ndarray
        A collection of RGB images, where each image is represented as a 3D array (height x width x channels).

    Returns:
    --------
    np.ndarray
        A 3D numpy array containing the grayscale versions of the input images,
        where each grayscale image is represented as a 2D array (height x width).
    '''
    laba
    return np.array([rgb2gray(img) for img in tqdm(X)])
```

A.1.2 Central Differences Derivative Mask

```
from skimage.feature._hog _hog_channel_gradient
def _central_hog_channel_gradient(channel):
    return _hog_channel_gradient(channel)
```

A.1.3 Holistic Derivative Mask

```python
import numpy as np

def _holistic_hog_channel_gradient(channel):
    '''
    Compute the gradients of a single channel using forward, backward, and central
    ↪ difference methods.

    Parameters:
    -----------
    channel : np.ndarray
        A 2D numpy array representing a single channel of an image.

    Returns:
    --------
    g_row : np.ndarray
        A 2D numpy array containing the gradient along the rows.

    g_col : np.ndarray
        A 2D numpy array containing the gradient along the columns.
    '''
    g_row = np.zeros(channel.shape, dtype=channel.dtype)
    g_col = np.zeros(channel.shape, dtype=channel.dtype)
    # forward difference
    g_row[0, :] = channel[1, :] - channel[0, :]
    g_col[:, 0] = channel[:, 1] - channel[:, 0]
    # backward difference
    g_row[-1, :] = channel[-1, :] - channel[-2, :]
    g_col[:, -1] = channel[:, -1] - channel[:, -2]
    # central difference
    g_row[1:-1, :] = (channel[2:, :] - channel[:-2, :])
    g_col[:, 1:-1] = (channel[:, 2:] - channel[:, :-2])

    return g_row, g_col
```

A.1.4 Modified HOG Computation

```python
def hog(
        image,
        hog_parameters: HOG_Parameters
):
```

```
    '''
    Compute the Histogram of Oriented Gradients (HOG) for the input image.

    Parameters:
    -----------
    image : np.ndarray
        A 2D numpy array representing the input image.

    hog_parameters : HOG_Parameters
        An object containing parameters for the HOG computation, including:
        - pixels_per_cell: Tuple specifying the size of the cells.
        - cells_per_block: Tuple specifying the number of cells per block.
        - block_stride: Tuple specifying the stride between blocks.
        - orientations: Number of orientation bins.
        - holistic_derivative_mask: Boolean to determine the gradient calculation method.

    Returns:
    --------
    np.ndarray
        A 1D numpy array containing the normalized HOG features for the input image.

    Raises:
    -------
    ValueError
        If the input image does not have two spatial dimensions or is too small
        given the specified parameters.
    '''

    image = np.atleast_2d(image)
    float_dtype = utils._supported_float_type(image.dtype)
    image = image.astype(float_dtype, copy=False)

    if image.ndim != 2:
        raise ValueError(
            'Only images with two spatial dimensions are supported.'
        )

    g_row, g_col = _holistic_hog_channel_gradient(
        image) if hog_parameters.holistic_derivative_mask else _central_hog_channel_gradient(
        image)

    s_row, s_col = image.shape[:2]
    c_row, c_col = hog_parameters.pixels_per_cell
    b_row, b_col = hog_parameters.cells_per_block
    b_row_stride, b_col_stride = hog_parameters.block_stride

```

```python
    n_cells_row = int(s_row // c_row)
    n_cells_col = int(s_col // c_col)

    orientation_histogram = np.zeros(
        (n_cells_row, n_cells_col, hog_parameters.orientations), dtype=float
    )
    g_row = g_row.astype(float, copy=False)
    g_col = g_col.astype(float, copy=False)

    _hoghistogram.hog_histograms(
        g_col,
        g_row,
        c_col,
        c_row,
        s_col,
        s_row,
        n_cells_col,
        n_cells_row,
        hog_parameters.orientations,
        orientation_histogram,
    )

    n_blocks_row = (s_row - (b_row + 1) * c_row) // (b_row_stride * c_row)
    n_blocks_col = (s_col - (b_col + 1) * c_col) // (b_col_stride * c_col)
    if n_blocks_col <= 0 or n_blocks_row <= 0:
        min_row = b_row * c_row
        min_col = b_col * c_col
        raise ValueError(
            'The input image is too small given the values of '
            'pixels_per_cell and cells_per_block. '
            'It should have at least: '
            f'{min_row} rows and {min_col} cols.'
        )
    normalized_blocks = np.zeros(
        (n_blocks_row, n_blocks_col, b_row, b_col, hog_parameters.orientations),
        ↪ dtype=float_dtype
    )

    for r in range(0, n_blocks_row):
        for c in range(0, n_blocks_col):
            block = orientation_histogram[
                    r * b_row_stride: r * b_row_stride + b_row,
                    c * b_col_stride: c * b_col_stride + b_col,
                    :
                    ]
            normalized_blocks[r, c, :] = _hog_normalize_block(block,
            ↪ method=hog_parameters.block_norm)
    normalized_blocks = normalized_blocks.ravel()
```

```python

    return normalized_blocks

def hog_transform(X, hog_parameters: HOG_Parameters):
    '''
    Apply the Histogram of Oriented Gradients (HOG) transformation to a collection of
    ↪  images.

    Parameters:
    -----------
    X : list or np.ndarray
        A collection of images, where each image is represented as a 2D numpy array.

    hog_parameters : HOG_Parameters
        An object containing parameters for the HOG computation.

    Returns:
    --------
    np.ndarray
        A 2D numpy array containing the HOG features for each input image,
        with each row representing the features of an individual image.
    '''
    return np.array([hog(img,hog_parameters) for img in tqdm(X)])

```

A.1.5 Caltech Data Set Transformation

```python
import os, glob
import cv2
from scipy.io import loadmat
from collections import defaultdict
import numpy as np
from lxml import etree, objectify

def vbb_anno2dict(vbb_file, cam_id, person_types=None):
    """
    Parse caltech vbb annotation file to dict
    Args:
        vbb_file: input vbb file path
        cam_id: camera id
        person_types: list of person type that will be used (total 4 types: person,
        ↪  person-fa, person?, people).
            If None, all will be used:
    Return:
```

```
        Annotation info dict with filename as key and anno info as value
    """
    filename = os.path.splitext(os.path.basename(vbb_file))[0]
    annos = defaultdict(dict)
    vbb = loadmat(vbb_file)
    # object info in each frame: id, pos, occlusion, lock, posv
    objLists = vbb['A'][0][0][1][0]
    objLbl = [str(v[0]) for v in vbb['A'][0][0][4][0]]
    # person index
    if not person_types:
        person_types = ["person", "person-fa", "person?", "people"]
    person_index_list = [x for x in range(len(objLbl)) if objLbl[x] in person_types]
    for frame_id, obj in enumerate(objLists):
        if len(obj) > 0:
            frame_name = str(cam_id) + "_" + str(filename) + "_" + str(frame_id+1) + ".jpg"
            annos[frame_name] = defaultdict(list)
            annos[frame_name]["id"] = frame_name
            for fid, pos, occl in zip(obj['id'][0], obj['pos'][0], obj['occl'][0]):
                fid = int(fid[0][0]) - 1  # for matlab start from 1 not 0
                if not fid in person_index_list:  # only use bbox whose label is given person type
                    continue
                annos[frame_name]["label"] = objLbl[fid]
                pos = pos[0].tolist()
                occl = int(occl[0][0])
                annos[frame_name]["occlusion"].append(occl)
                annos[frame_name]["bbox"].append(pos)
            if not annos[frame_name]["bbox"]:
                del annos[frame_name]
    return annos


def seq2img(annos, seq_file, outdir, cam_id):
    """
    Extract frames in seq files to given output directories
    Args:
         annos: annos dict returned from parsed vbb file
         seq_file: seq file path
         outdir: frame save dir
         cam_id: camera id
    Returns:
        camera captured image size
    """
    cap = cv2.VideoCapture(seq_file)
    index = 1
    # captured frame list
    v_id = os.path.splitext(os.path.basename(seq_file))[0]
```

```python
    cap_frames_index = np.sort([int(os.path.splitext(id)[0].split("_")[2]) for id in annos.keys()])
    while True:
        ret, frame = cap.read()
        if ret:
            if not index in cap_frames_index:
                index += 1
                continue
            if not os.path.exists(outdir):
                os.makedirs(outdir)
            outname = os.path.join(outdir, str(cam_id)+"_"+v_id+"_"+str(index)+".jpg")
            print("Current frame: ", v_id, str(index))
            cv2.imwrite(outname, frame)
            height, width, _ = frame.shape
        else:
            break
        index += 1
    img_size = (width, height)
    return img_size


def instance2xml_base(anno, img_size, bbox_type='xyxy'):
    """
    Parse annotation data to VOC XML format
    Args:
        anno: annotation info returned by vbb_anno2dict function
        img_size: camera captured image size
        bbox_type: bbox coordinate record format: xyxy (xmin, ymin, xmax, ymax); xywh (xmin, ymin, width, height)
    Returns:
        Annotation xml info tree
    """
    assert bbox_type in ['xyxy', 'xywh']
    E = objectify.ElementMaker(annotate=False)
    anno_tree = E.annotation(
        E.folder('VOC2014_instance/person'),
        E.filename(anno['id']),
        E.source(
            E.database('Caltech pedestrian'),
            E.annotation('Caltech pedestrian'),
            E.image('Caltech pedestrian'),
            E.url('None')
        ),
        E.size(
            E.width(img_size[0]),
            E.height(img_size[1]),
            E.depth(3)
        ),
```

```python
            E.segmented(0),
        )
    for index, bbox in enumerate(anno['bbox']):
        bbox = [float(x) for x in bbox]
        if bbox_type == 'xyxy':
            xmin, ymin, w, h = bbox
            xmax = xmin+w
            ymax = ymin+h
        else:
            xmin, ymin, xmax, ymax = bbox
        xmin = int(xmin)
        ymin = int(ymin)
        xmax = int(xmax)
        ymax = int(ymax)
        if xmin < 0:
            xmin = 0
        if xmax > img_size[0] - 1:
            xmax = img_size[0] - 1
        if ymin < 0:
            ymin = 0
        if ymax > img_size[1] - 1:
            ymax = img_size[1] - 1
        if ymax <= ymin or xmax <= xmin:
            continue
        E = objectify.ElementMaker(annotate=False)
        anno_tree.append(
            E.object(
            E.name(anno['label']),
            E.bndbox(
                E.xmin(xmin),
                E.ymin(ymin),
                E.xmax(xmax),
                E.ymax(ymax)
            ),
            E.difficult(0),
            E.occlusion(anno["occlusion"][index])
            )
        )
    return anno_tree


def parse_anno_file(vbb_inputdir, seq_inputdir, vbb_outputdir, seq_outputdir,
 ↪  person_types=None):
    """
    Parse Caltech data stored in seq and vbb files to VOC xml format
    Args:
        vbb_inputdir: vbb file saved pth
        seq_inputdir: seq file saved path
```

```
        vbb_outputdir: vbb data converted xml file saved path
        seq_outputdir: seq data converted frame image file saved path
        person_types: list of person type that will be used (total 4 types: person, person-fa, person?, people).
            If None, all will be used:
    """
    # annotation sub-directories in hda annotation input directory
    assert os.path.exists(vbb_inputdir)
    sub_dirs = os.listdir(vbb_inputdir)
    for sub_dir in sub_dirs:
        print("Parsing annotations of camera: ", sub_dir)
        cam_id = sub_dir
        vbb_files = glob.glob(os.path.join(vbb_inputdir, sub_dir, "*.vbb"))
        for vbb_file in vbb_files:
            annos = vbb_anno2dict(vbb_file, cam_id, person_types=person_types)
            if annos:
                vbb_outdir = os.path.join(vbb_outputdir, "annotations", sub_dir, "bbox")
                # extract frames from seq
                seq_file = os.path.join(seq_inputdir, sub_dir, os.path.splitext(os.path.basename(vbb_file))[0]+".seq")
                seq_outdir = os.path.join(seq_outputdir, sub_dir, "frame")
                if not os.path.exists(vbb_outdir):
                    os.makedirs(vbb_outdir)
                if not os.path.exists(seq_outdir):
                    os.makedirs(seq_outdir)
                img_size = seq2img(annos, seq_file, seq_outdir, cam_id)
                for filename, anno in sorted(annos.items(), key=lambda x: x[0]):
                    if "bbox" in anno:
                        anno_tree = instance2xml_base(anno, img_size)
                        outfile = os.path.join(vbb_outdir, os.path.splitext(filename)[0]+".xml")
                        print("Generating annotation xml file of picture: ", filename)
                        etree.ElementTree(anno_tree).write(outfile, pretty_print=True)

def main():
    seq_dir = "../Pedestrian-Detection/datasets/caltech_raw/Test"
    vbb_dir = "../Pedestrian-Detection/datasets/caltech_raw/annotations/Test"
    out_dir = "../Pedestrian-Detection/datasets/caltech_parsed/Test"
    frame_out = os.path.join(out_dir, "frame")
    anno_out = os.path.join(out_dir, "annotation")
    person_type = ["person", "people"]
    parse_anno_file(vbb_dir, seq_dir, frame_out, anno_out, person_type)
```

A.1.6 Retain 30th Caltech Data Set Frame

```python
import os
from tqdm import tqdm
def retain_30th_frame():
    root_dir = r'/Users/adamsam/repos/ee/Pedestrian-Detection/datasets/caltech_30/Test'
    annotation_dir = os.path.join(root_dir, 'annotations')
    frame_dir = os.path.join(root_dir, 'frame')
    frame_instance = 0
    for frame_subdir in tqdm(os.listdir(frame_dir)):
        frame_subdir_path = os.path.join(frame_dir, frame_subdir)
        if(os.path.isdir(frame_subdir_path)):
            frame_files = os.listdir(os.path.join(frame_subdir_path, 'frame'))
            for frame_file in frame_files:
                file_location = os.path.join(frame_subdir_path, 'frame', frame_file)

                if not os.path.isfile(file_location):
                    continue

                if frame_instance % 30 != 0:
                    os.remove(file_location)
                    annotation_file_location = os.path.join(annotation_dir, frame_subdir, 'bbox', frame_file.split('.')[0] + '.xml')
                    if os.path.isfile(annotation_file_location):
                        os.remove(annotation_file_location)
                frame_instance += 1
```

A.1.7 Pedestrian Data Set Construction

```python
import cv2
import numpy as np
import os
from sklearn.model_selection import train_test_split
from tqdm import tqdm
import random

window_sizes = [(128, 64), (112, 48), (100, 50), (128, 96)]

class SampleCount:
    def __init__(self, pos_count, neg_count):
        '''
```

```
        Initialize the SampleCount object.

        Parameters:
        -----------
        pos_count : int
            The number of positive samples.

        neg_count : int
            The number of negative samples.
        '''
        self.pos = pos_count
        self.neg = neg_count

class LabeledDataSet:
    def __init__(self, points, labels, sample_count: SampleCount):
        '''
        Initialize the LabeledDataSet object.

        Parameters:
        -----------
        points : np.ndarray
            The data points (images) in the dataset.

        labels : np.ndarray
            The corresponding labels for the data points.

        sample_count : SampleCount
            An object containing the counts of positive and negative samples.
        '''
        self.points = points
        self.labels = labels
        self.sample_count = sample_count

def parse_pascal_voc_annotations(file_name):
    '''
    Parse Pascal VOC annotations from an XML file.

    Parameters:
    -----------
    file_name : str
        The path to the annotation XML file.

    Returns:
    --------
    list
        A list of bounding boxes, each represented as a list of integers [xmin, ymin,
        ↪  xmax, ymax].
    '''
```

```python
    import xml.etree.ElementTree as ET
    tree = ET.parse(file_name)
    root = tree.getroot()
    bbox = []

    for obj in root.findall('object'):
        bndbox = obj.find('bndbox')
        bbox.append([
            int(bndbox.find('xmin').text),
            int(bndbox.find('ymin').text),
            int(bndbox.find('xmax').text),
            int(bndbox.find('ymax').text)
        ])
    return bbox


def prepare_labeled_datasets(image_folder, window_size, test_size=0.2, random_state=42):
    '''
    Prepare labeled datasets for training and testing.

    Parameters:
    -----------
    image_folder : str
        The path to the folder containing images and annotations.

    window_size : tuple
        The size of the sliding window for sample extraction.

    test_size : float
        The proportion of the dataset to include in the test split (default is 0.2).

    random_state : int
        Random seed for reproducibility (default is 42).

    Returns:
    --------
    LabeledDataSet, LabeledDataSet
        The training and testing labeled datasets.
    '''
    image_dir = os.path.join(image_folder, "frame")
    annotation_dir = os.path.join(image_folder, "annotations")

    image_subdirs = [
        os.path.join(image_dir, subdir)
        for subdir in os.listdir(image_dir)
        if os.path.isdir(os.path.join(image_dir, subdir))
    ]
```

```python
images = [os.path.join(subdir, file) for subdir in image_subdirs for file in os.listdir(subdir) if
          os.path.isfile(os.path.join(subdir, file))]


train_images, test_images = train_test_split(images, test_size=test_size, random_state=random_state)

def process_images(image_list):
    data_points = []
    labels = []
    num_pos = 0
    num_neg = 0

    for num, image_file_location in enumerate(tqdm(image_list)):
        image = cv2.imread(image_file_location)

        partial_location = image_file_location.split(os.sep)[-2:]
        annotation_file_location = os.path.join(
            annotation_dir,
            "/".join(map(str, partial_location))
        )[:-4]

        if os.path.exists(annotation_file_location + ".xml"):
            bbox_arr = parse_pascal_voc_annotations(annotation_file_location + ".xml")
        else:
            raise Exception(f"Annotation file {annotation_file_location} not found")

        for _ in range(3):
            h, w = image.shape[:2]

            if h > window_size[0] or w > window_size[1]:
                h = h - window_size[0]
                w = w - window_size[1]
                max_loop = 0
                overlap = []
                new_window = []
                for _ in range(10):
                    x = random.randint(0, w)
                    y = random.randint(0, h)
                    overlap = [True for i in bbox_arr]
                    new_window = [x, y, x + window_size[1], y + window_size[0]]

                    for index, bbox in enumerate(bbox_arr):
                        dx = min(bbox[2], new_window[2]) - max(bbox[0], new_window[0])
```

```
                            dy = min(bbox[3], new_window[3]) - max(bbox[1], new_window[1])
                            if dx <= 0 or dy <= 0:
                                overlap[index] = False
                        if not np.any(overlap):
                            break
                    if not np.any(overlap):
                        img = image[window[1]:window[3], window[0]:window[2]]
                        data_points.append(img)
                        labels.append(0)
                        num_neg += 1

            # Process positive samples (bounding boxes)
            for box in bbox_arr:
                upper_random_boundary = random.randint(5,20)
                pad_left = random.randint(0, upper_random_boundary)
                pad_right = random.randint(0, upper_random_boundary)
                pad_top = random.randint(0, upper_random_boundary)
                pad_bottom = random.randint(0, upper_random_boundary)
                x1 = box[0] + pad_left
                y1 = box[1] + pad_top
                x2 = box[2] - pad_right
                y2 = box[3] - pad_bottom
                if x1 > x2:
                    x2 = min(image.shape[1], box[2] + pad_left)
                if y1 > y2:
                    y2 = min(image.shape[0],box[3]+pad_top)
                img = image[y1:y2, x1:x2]
                img_resized = cv2.resize(img, (window_size[1], window_size[0]))
                data_points.append(img_resized)
                labels.append(1)
                num_pos += 1


        return data_points, labels, num_pos, num_neg

    train_data, train_labels, train_pos, train_neg = process_images(train_images)
    test_data, test_labels, test_pos, test_neg = process_images(test_images)


    labeled_training_set = LabeledDataSet(np.array(train_data), np.array(train_labels), SampleCount(train_pos, train_neg))
    labeled_testing_set = LabeledDataSet(np.array(test_data), np.array(test_labels), SampleCount(test_pos, test_neg))

    return labeled_training_set, labeled_testing_set
```

```python
def get_dataset_path(window_size, category, data_type, dataset=None):
    '''
    Get the file path for the dataset based on the window size, category, and data
    ↪ type.

    Parameters:
    -----------
    window_size : tuple
        The size of the sliding window as (height, width).

    category : str
        The category of the dataset, either 'train' or 'test'.

    data_type : str
        The type of data, either 'point' or 'label'.

    dataset : str, optional
        The name of the dataset (required if category is 'test').

    Returns:
    --------
    str
        The file path for the specified dataset.
    '''

    file_path = ''

    if category not in ['train', 'test']:
        raise ValueError('category must be either "train" or "test"')
    if data_type not in ['point', 'label']:
        raise ValueError('data_type must be either "point" or "label"')

    category_dir = f'../datasets/npy_{category}'

    file_name = f'{data_type}_{window_size[1]}-{window_size[0]}.npy'

    if category == 'train':
        file_path = os.path.join(category_dir, file_name)
    elif category == 'test' and dataset is not None:
        file_path = os.path.join(category_dir, dataset, file_name)

    if not os.path.exists(os.path.dirname(file_path)):
        os.makedirs(os.path.dirname(file_path))

    return file_path


def init_datasets(datasets_path):
```

```python
    '''
    Initialize datasets for different window sizes and save the training and testing sets.

    Parameters:
    -----------
    datasets_path : str
        The path to the datasets directory.
    '''
    for window_size in window_sizes:
        total_training_points = np.array([])
        total_training_labels = np.array([])
        for dataset in ['INRIA', 'caltech_30', 'PnPLO']:
            print(f'\n\nInitializing dataset {dataset} with window size {window_size}\n')
            training_set, testing_set = prepare_labeled_datasets(os.path.join(datasets_path, dataset), window_size)

            print("Training Positives: ", training_set.sample_count.pos)
            print("Training Negatives: ", training_set.sample_count.neg)
            print("Testing Positives: ", testing_set.sample_count.pos)
            print("Testing Negatives: ", testing_set.sample_count.neg)

            # np.concatenate requires identical array dimensions
            if total_training_points.shape[0] == 0:
                total_training_points = training_set.points
                total_training_labels = training_set.labels
            else:
                total_training_points = np.concatenate((total_training_points, training_set.points))
                total_training_labels = np.concatenate((total_training_labels, training_set.labels))

            np.save(get_dataset_path(window_size, 'test', 'point', dataset), testing_set.points)
            np.save(get_dataset_path(window_size, 'test', 'label', dataset), testing_set.labels)

            print("\nInitialized")

        print("\n\nSaving total training sets\n")
        np.save(get_dataset_path(window_size, 'train', 'point'), total_training_points)
        np.save(get_dataset_path(window_size, 'train', 'label'), total_training_labels)

```

A.1.8 Training a Soft Constraint SVM

```python
import os
import joblib
import numpy as np
from hog import HOG_Parameters, hog
from transform import grayscale_transform, hog_transform
from sklearn.svm import SVC

class SVM_Parameters:
    '''
    Class to hold SVM parameters, including HOG parameters and window size.

    Attributes:
    -----------
    hog_parameters : HOG_Parameters
        Parameters for HOG feature extraction.

    window_size : tuple
        The size of the sliding window as (height, width).
    '''
    def __init__(self, hog_parameters: HOG_Parameters, window_size):
        self.hog_parameters = hog_parameters
        self.window_size = window_size
    def get_svm_name(self):
        '''
        Get the name of the SVM model based on HOG parameters and window size.

        Returns:
        --------
        str
            The name of the SVM model.
        '''
        return "svm_" + self.hog_parameters.get_hog_name() + "_window_" +
        ↪  str(self.window_size)

def load_svm(svm_parameters: SVM_Parameters, model_dir, custom_name=None):
    '''
    Load an SVM model from the specified directory.

    Parameters:
    -----------
    svm_parameters : SVM_Parameters
        Parameters associated with the SVM model.

    model_dir : str
```

```
        The directory where the model is stored.

    custom_name : str, optional
        A custom name for the model file.

    Returns:
    --------
    object
        The loaded SVM model.

    Raises:
    -------
    Exception
        If the model file is not found.
    '''
    model_name = custom_name if custom_name is not None else
    ↪  svm_parameters.get_svm_name()
    model_file_name = os.path.join(model_dir, model_name + ".pkl")
    print(model_file_name)
    if os.path.exists(model_file_name):
        return joblib.load(model_file_name)
    raise Exception("Model not found")

def train_svm(svm_parameters: SVM_Parameters, data_points_location, labels_location,
↪  overwrite=False, custom_name=None):
    '''
    Train an SVM model with the given parameters and save it to a file.

    Parameters:
    -----------
    svm_parameters : SVM_Parameters
        Parameters associated with the SVM model.

    data_points_location : str
        Path to the file containing training data points.

    labels_location : str
        Path to the file containing training labels.

    overwrite : bool, optional
        If True, overwrite the existing model.

    custom_name : str, optional
        A custom name for the saved model file.

    kernel_type : str, optional
        The type of kernel to use for the SVM.
    '''
```

```
    from sklearn.linear_model import SGDClassifier
    model_name = custom_name if custom_name is not None else
    ↪  svm_parameters.get_svm_name()

    model_file_path = os.path.join('../saved_models', model_name + ".pkl")

    if os.path.exists(model_file_path):
      if overwrite:
        print("Removing existing model")
        os.remove(model_file_path)
      else:
        print("Model already exists")
        return

    if os.path.exists(data_points_location) and os.path.exists(labels_location):
        training_data_points = np.load(data_points_location)
        training_labels = np.load(labels_location)
    else:
        raise Exception(
            "No data points or labels found",
            data_points_location,
            labels_location
        )

    x_train = np.load(data_points_location)
    y_train = np.load(labels_location)

    x_train_gray = grayscale_transform(x_train)
    x_train_hog = hog_transform(x_train_gray, svm_parameters.hog_parameters)

    sgd_clf = SGDClassifier(random_state=42, max_iter=1000, tol=1e-3)
    sgd_clf.fit(x_train_hog, y_train)

    joblib.dump(sgd_clf, model_file_path)


```

A.1.9 Plotting MCC-F1 Curves

```
from mcc_f1 import mcc_f1_curve
from mcc_f1._plot.base import _get_response

class MCCF1CurveDisplay:
    """MCC-F1 Curve visualization with threshold values."""
```

```python

    def __init__(self, *, f1, mcc, thresholds,
                 mcc_f1=None, estimator_name=None, pos_label=None):
        self.estimator_name = estimator_name
        self.f1 = f1
        self.mcc = mcc
        self.thresholds = thresholds
        self.mcc_f1 = mcc_f1
        self.pos_label = pos_label

    def plot(self, ax=None, *, name=None, n_thresholds=0, **kwargs):
        """Plot visualization with threshold values

        Parameters
        ----------
        ax : matplotlib axes, default=None
            Axes object to plot on. If `None`, a new figure and axes is created.

        name : str, default=None
            Name of ROC Curve for labeling. If `None`, use the name of the estimator.

        n_thresholds : int, default=5
            Number of threshold values to display on the curve.

        Returns
        -------
        display : MCCF1CurveDisplay
            Object that stores computed values.
        """
        name = self.estimator_name if name is None else name

        line_kwargs = {}
        if self.mcc_f1 is not None and name is not None:
            line_kwargs["label"] = f"{name} (MCC-F1 = {self.mcc_f1:.2f})"
        elif self.mcc_f1 is not None:
            line_kwargs["label"] = f"MCC-F1 = {self.mcc_f1:.2f}"
        elif name is not None:
            line_kwargs["label"] = name

        line_kwargs.update(**kwargs)

        import matplotlib.pyplot as plt
        from matplotlib.figure import figaspect
        import numpy as np

        if ax is None:
            fig, ax = plt.subplots(figsize=figaspect(1.))

```

```python
        # Plot the MCC-F1 curve
        self.line_, = ax.plot(self.f1, self.mcc, **line_kwargs)

        # Add threshold values
        if n_thresholds > 0:
            # Get indices for evenly spaced points along the curve
            n_points = len(self.thresholds)
            indices = np.linspace(0, n_points - 1, n_thresholds, dtype=int)

            # Plot threshold points and values
            ax.scatter(self.f1[indices], self.mcc[indices],
                       color='red', zorder=2, s=20)

            for idx in indices:
                # Add annotation with threshold value
                ax.annotate(f'$\\tau$={self.thresholds[idx]:.2f}',
                            (self.f1[idx], self.mcc[idx]),
                            xytext=(10, 10), textcoords='offset points',
                            bbox=dict(facecolor='white', edgecolor='none', alpha=0.7))

        info_pos_label = (f" (Positive label: {self.pos_label})"
                          if self.pos_label is not None else "")

        xlabel = "F1-Score" + info_pos_label
        ylabel = "MCC" + info_pos_label
        ax.set(xlabel=xlabel, ylabel=ylabel, xlim=(0, 1), ylim=(0, 1))

        if "label" in line_kwargs:
            ax.legend(loc="lower right")

        self.ax_ = ax
        self.figure_ = ax.figure
        return self

def plot_mcc_f1_curve(estimator, X, y, *, sample_weight=None,
                      response_method="auto", name=None, ax=None,
                      pos_label=None, n_thresholds=0, **kwargs):
    """Plot MCC-F1 curve with threshold values.

    Parameters
    ----------
    Parameters
    ----------
    estimator : estimator instance
        Fitted classifier or a fitted :class:`~sklearn.pipeline.Pipeline`
        in which the last estimator is a classifier.

    X : {array-like, sparse matrix} of shape (n_samples, n_features)
```

```
            Input values.

        y : array-like of shape (n_samples,)
            Target values.

        sample_weight : array-like of shape (n_samples,), default=None
            Sample weights.

        response_method : {'predict_proba', 'decision_function', 'auto'} \
        default='auto'
            Specifies whether to use :term:`predict_proba` or
            :term:`decision_function` as the target response. If set to 'auto',
            :term:`predict_proba` is tried first and if it does not exist
            :term:`decision_function` is tried next.

        name : str, default=None
            Name of MCC-F1 Curve for labeling. If `None`, use the name of the
            estimator.

        ax : matplotlib axes, default=None
            Axes object to plot on. If `None`, a new figure and axes is created.

        pos_label : str or int, default=None
            The class considered as the positive class when computing the metrics.
            By default, `estimators.classes_[1]` is considered as the positive
            class.

        n_thresholds : int, default=5
            Number of threshold values to display on the curve.
        """
        y_pred, pos_label = _get_response(
            X, estimator, response_method, pos_label=pos_label)

        mcc, f1, thresholds = mcc_f1_curve(y, y_pred, pos_label=pos_label,
                                           sample_weight=sample_weight)
        mcc_f1 = None

        name = estimator.__class__.__name__ if name is None else name

        viz = MCCF1CurveDisplay(
            f1=f1,
            mcc=mcc,
            thresholds=thresholds,
            mcc_f1=mcc_f1,
            estimator_name=name,
            pos_label=pos_label
        )

```

```
    return viz.plot(ax=ax, name=name, n_thresholds=n_thresholds, **kwargs)
```

A.1.10 Evaluate Pedestrian Classifier

```
import os
import numpy as np
from sklearn.metrics import average_precision_score, roc_curve, auc, recall_score,
↪ precision_score, f1_score, \
    precision_recall_curve, confusion_matrix, matthews_corrcoef

from dataset import get_dataset_path, datasets
from parameters import HOG_Parameters, SVM_Parameters
from svm import load_svm
from transform import hog_transform, grayscale_transform
from variables import iterate_model_parameters, get_model_count

score_keys = ['mcc', 'accuracy', 'f1', 'fppw', 'auc_roc', 'average_precision']
score_index_map = {key: i for i, key in enumerate(score_keys)}

def evaluate_pedestrian_classifier(model, X_test, y_test):
    """
    Evaluate a binary classifier for pedestrian detection using multiple metrics.

    Parameters:
    -----------
    model : trained classifier object
        Must implement predict() and predict_proba() or decision_function()
    X_test : array-like
        Test features
    y_test : array-like
        True labels (0 for non-pedestrian, 1 for pedestrian)

    Returns:
    --------
    dict : Dictionary containing evaluation metrics
    """
    metrics = {}

    # If probabilities not available, use decision function
    y_scores = model.decision_function(X_test)
    # Normalize scores to [0,1] range for better interpretability
    y_scores = (y_scores - y_scores.min()) / (y_scores.max() - y_scores.min())

    y_pred = model.predict(X_test)
```

```python

    # Basic classification metrics
    metrics['accuracy'] = np.mean(y_pred == y_test)

    # Confusion matrix and derived metrics
    cm = confusion_matrix(y_test, y_pred)
    metrics['confusion_matrix'] = cm
    metrics['true_negatives'] = cm[0, 0]
    metrics['false_positives'] = cm[0, 1]
    metrics['false_negatives'] = cm[1, 0]
    metrics['true_positives'] = cm[1, 1]

    # Precision, Recall, F1
    metrics['precision'] = precision_score(y_test, y_pred)
    metrics['recall'] = recall_score(y_test, y_pred)
    metrics['f1'] = f1_score(y_test, y_pred)

    # Matthews Correlation Coefficient
    metrics['mcc'] = matthews_corrcoef(y_test, y_pred)
    # Class-wise metrics
    metrics['specificity'] = cm[0, 0] / (cm[0, 0] + cm[0, 1])  # True Negative Rate
    metrics['fall_out'] = cm[0, 1] / (cm[0, 0] + cm[0, 1])  # False Positive Rate
    metrics['miss_rate'] = cm[1, 0] / (cm[1, 0] + cm[1, 1])  # False Negative Rate

    if y_scores is not None:
        # Precision-Recall curve
        precision, recall, pr_thresholds = precision_recall_curve(y_test, y_scores)
        metrics['pr_curve'] = {
            'precision': precision,
            'recall': recall,
            'thresholds': pr_thresholds
        }
        metrics['average_precision'] = average_precision_score(y_test, y_scores)

        # ROC curve
        fpr, tpr, roc_thresholds = roc_curve(y_test, y_scores)
        metrics['roc_curve'] = {
            'fpr': fpr,
            'tpr': tpr,
            'thresholds': roc_thresholds
        }
        metrics['auc_roc'] = auc(fpr, tpr)

    # Add some practical metrics
    total_windows = len(y_test)
    metrics['fppw'] = metrics['false_positives'] / total_windows

    return metrics
```

A.1.11 Construct McNemar's Confusion Matrix

```python
def construct_mcnemar_table(
        y_true,
        model_1_pred,
        model_2_pred
):
    '''
    Constructs a 2x2 contingency table for McNemar's test based on the predictions of two models.

    Parameters:
    -----------
    y_true : list or array-like
        The true class labels for the test set.

    model_1_pred : list or array-like
        The predicted class labels from the first model.

    model_2_pred : list or array-like
        The predicted class labels from the second model.

    Returns:
    --------
    contingency_table : np.ndarray
        A 2x2 numpy array that represents the contingency table:
            [[a, b], [c, d]]
        where:
        - a = Both models correctly classify the instance.
        - b = Model 1 is correct, but Model 2 is incorrect.
        - c = Model 1 is incorrect, but Model 2 is correct.
        - d = Both models incorrectly classify the instance.
    '''
    a = b = c = d = 0

    for i in range(len(y_true)):
        model_1_correct = (model_1_pred[i] == y_true[i])
        model_2_correct = (model_2_pred[i] == y_true[i])

        if model_1_correct and model_2_correct:
            a += 1
        elif model_1_correct and not model_2_correct:
```

```
                b += 1
            elif not model_1_correct and model_2_correct:
                c += 1
            else:
                d += 1
        contingency_table = np.array([[a, b], [c, d]])
        return contingency_table
```

A.2 Tables of Data

A.2.1 INRIA Evaluation

A.2.2 Caltech Evaluation

A.2.3 PnPLO Evaluation