

Spring 5-18-2018

## Detecting Rip Currents from Images

Corey C. Maryan  
*University of New Orleans*, cmaryan@uno.edu

Follow this and additional works at: <https://scholarworks.uno.edu/td>



Part of the [Artificial Intelligence and Robotics Commons](#)

---

### Recommended Citation

Maryan, Corey C., "Detecting Rip Currents from Images" (2018). *University of New Orleans Theses and Dissertations*. 2473.

<https://scholarworks.uno.edu/td/2473>

This Thesis is protected by copyright and/or related rights. It has been brought to you by ScholarWorks@UNO with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis has been accepted for inclusion in University of New Orleans Theses and Dissertations by an authorized administrator of ScholarWorks@UNO. For more information, please contact [scholarworks@uno.edu](mailto:scholarworks@uno.edu).

# Detecting Rip Currents from Images

A Thesis

Submitted to the Graduate Faculty of the  
University of New Orleans  
in partial fulfillment of the  
requirements for the degree of

Master of Science  
in  
Computer Science

by

Corey Maryan

B.A. Southeastern Louisiana University, 2013

May, 2018

## Acknowledgments

I would like to thank Dr. Mahdi Abdelguerfi for all of the opportunities he has given me during my career at the University of New Orleans. He has given me more than I could ever ask for. I would also like to thank him for recommending the thesis idea.

I would like to thank Dr. Md Tamjidul Hoque for providing me with the assistance I needed for both the machine learning portion and writing portion of this project. He kept a considerate attitude throughout the course of the project and was always eager to help.

I would also like to thank Dr. Elias Ioup and Dr. Christopher Michael for helping lay the groundwork for collaboration with Naval Research Labs and for thesis guidance. I would also like to thank Dr. Elias Ioup for jointly recommending the thesis idea.

Finally, I would like to thank Devin Frey for sparking my initial interest in computer science.

# Table of Contents

Abstract .....	vi
Introduction.....	1
Literature Review.....	5
Machine Learning .....	5
Neural Networks .....	6
Support Vector Machines .....	11
Ada-Boost .....	15
PCA.....	16
Other Machine Learning Algorithms.....	17
Object Detection .....	18
Haar Features .....	24
Oceanography .....	27
Orthorectification.....	32
Methodology .....	36
Rip Current Dataset.....	36
Max distance from the average .....	37
Matlab .....	37
Implementation .....	38
Support Vector Machines .....	38
Features for training.....	38
Implementation .....	39
Convolutional Neural Net.....	40
TensorFlow framework.....	40
Implementation .....	41
Viola-Jones .....	41
OpenCV library.....	41
Implementation .....	42
Meta Learner .....	43
Additional Features .....	43
Implementation .....	45
Results.....	48
Max distance from the average .....	48
Support Vector Machines .....	49
Convolutional Neural Net.....	51
Viola-Jones .....	52
Meta-Learner.....	55
New Features .....	55
Stacking.....	56
Meta-learner .....	57
Discussions .....	61
Max distance from the average .....	61
Support Vector Machines .....	61
Convolutional Neural Net.....	62
Viola-Jones .....	63

Meta-Learner.....	64
New Features .....	64
Classifier Stacking .....	64
Conclusions.....	66
Bibliography .....	67
Appendices.....	69
Ada-Boost in Java .....	69
Integral Image in Java .....	83
New Haar Features in Java .....	86
Max distance from the average in Matlab.....	92
Meta-Learner Feature Vector in Python .....	98
Meta-Learner in Python .....	104
Vita.....	111

## List of Figures

Figure 1: Neural network node .....	7
Figure 2: A Neural Network .....	8
Figure 3: Neural network algorithm .....	9
Figure 4: A hyperplane .....	11
Figure 5: Non-linearly separability .....	12
Figure 6: Maximal Margin.....	13
Figure 7: Ada-Boost algorithm .....	15
Figure 8: Attentional cascading .....	18
Figure 9: The Viola-Jones Algorithm .....	19
Figure 10: Viola-Jones image .....	20
Figure 11: Example Haar Features .....	24
Figure 12: Applying a feature .....	25
Figure 13: Integral Image point .....	25
Figure 14: Integral Image Regions .....	26
Figure 15: A rip current .....	27
Figure 16: Min max rip currents .....	29
Figure 17: Transposed Sandbar locations .....	31
Figure 18: Time averaged Shorline.....	32
Figure 19: Figure 18 transposed .....	33
Figure 20: A picture from an Argus site .....	34
Figure 21: Argus picture of rip currents .....	34
Figure 22: Rain droplets.....	34
Figure 23: Sun blocking a camera .....	34
Figure 24: Rip current samples .....	36
Figure 25: Max distance from average .....	37
Figure 26: TensorFlow Setup.....	40
Figure 27: Viola-Jones Model .....	42
Figure 28: Matrix .....	43
Figure 29: New Features results .....	44
Figure 30: New Features .....	45
Figure 31: Meta-classifier data .....	46
Figure 32: Basic classifier parameters .....	47
Figure 33: Meta-learner .....	47
Figure 34: Max distance from the average .....	48
Figure 35: Detection rate max distance .....	49
Figure 36: False positive rate max distance.....	50
Figure 37: SVM accuracy .....	50
Figure 38: CNN detection rate.....	51
Figure 39: CNN false positive count .....	52
Figure 40: CNN image.....	52
Figure 41: Viola-Jones Detection rate .....	53
Figure 42: Viola-Jones False positive count.....	54
Figure 43: Viola-Jones image .....	54
Figure 44: Accuracy adding new features .....	55

Figure 45: Detection rate after stacking.....	56
Figure 46: False positive rate after stacking .....	57
Figure 47: Viola-Jones vs meta-learner detection rate .....	58
Figure 48: Viola-Jones vs meta-learner false positive count .....	58
Figure 49: Comparing detection rates.....	59
Figure 50: Comparing false positive counts .....	60
Figure 51: Viola-Jones vs meta-learner image .....	60
Figure 52: Java source for Viola-Jones.....	69

## **Abstract**

Rip current images are useful for assisting in climate studies but time consuming to manually annotate by hand over thousands of images. Object detection is a possible solution for automatic annotation because of its success and popularity in identifying regions of interest in images, such as human faces. Similarly to faces, rip currents have distinct features that set them apart from other areas of an image, such as more generic patterns of the surf zone. There are many distinct methods of object detection applied in face detection research. In this thesis, the best fit for a rip current object detector is found by comparing these methods. In addition, the methods are improved with Haar features exclusively created for rip current images. The compared methods include max distance from the average, support vector machines, convolutional neural networks, the Viola-Jones object detector, and a meta-learner. The presented results are compared for accuracy, false positive rate, and detection rate. Viola-Jones has the top base-line performance by achieving a detection rate of 0.88 and identifying only 15 false positives in the test image set of 53 rip currents. The described meta-learner integrates the presented Haar features, which are developed in accordance with the original Viola-Jones algorithm. Ada-Boost, a feature ranking algorithm, shows that the newly presented Haar features extract more meaningful data from rip current images than some of the current features. The meta-classifier improves upon the stand-alone Viola-Jones when applying these features by reducing its false positives by 47% while retaining a similar computational cost and detection rate.



# 1. Introduction

Rip currents are nearshore phenomena caused by the breaking of waves in an along-shore direction. They account for the majority of lifeguard rescues at beaches. For oceanography, rip current images assist in climate studies but must be manually annotated. “Argus site” locations such as Duck, North Carolina hold rip current images in a backlog of beach imagery [1]. Argus sites are groups of cameras that regularly take photos of shorelines. These images go through the process of “orthorectification” [2],[3]. Orthorectification alters the perspective of shoreline photos by combining snapshots taken from cameras at different angles and turning them into one image. The final image shows one, continuous bird’s eye view of the shoreline, which permits easier identification of rip currents. There are many Argus sites at different locations taking photos on an hourly basis. This amounts to several thousand images that may contain multiple rip currents per image. Neither the time nor the manpower exist for manually annotating every image, which makes conducting climate studies involving rip currents difficult. The lack of resources creates a need for an automated method to find rip currents in images. The automation of rip current detection can reduce the risk to swimmers and provide data for nearshore studies in a shorter amount of time. Finding an efficient method to automatically find these rip currents naturally lends itself to computing since computers process data far more quickly and accurately than humans. Specifically, a branch of computer science called “machine learning” provides a possible, effective solution because of its success in identifying regions of interest in images [4-6].

Machine learning involves the development of algorithms that train models, which make predictions on a set of test data [7]. Predicting the class of a data sample is known as “classification”. Binary classification assumes each data sample belongs in 1 of 2 classes. For

example, if a model trains to find rip currents in an image, then a collection of images is the test data. Some images exemplify rip currents and have a class label of 1 while other images have a class label of 0. The model is exposed to the images and trains with an algorithm. Then, the model anticipates the class of unfamiliar images. Training a model with images involves extracting data from each image via “computer vision” techniques. Computer vision is an area of computer science concerned with image processing and “feature extraction” [8]. Feature extraction collects descriptive data from each data sample, which trains a model. Feature extraction, as a tool, reduces the amount of input data for the model. A large amount of data yields more time and processing power for training. The reduction of input data makes feature extraction important as it reduces runtime and conserves memory. A model identifies certain areas in an image after it trains on computer vision features. Identifying a specific region of an image with machine learning algorithms is “object detection”.

Object detection lends itself to a broad category than simply rip currents. Human faces are a popular object for detection research [9],[10]. Models built for face detection train on images of faces and develop from learning algorithms. An algorithm describes how a model updates itself in accordance with the data. Learning algorithms find a pattern between samples and update a model to achieve a higher performance on the dataset based on how many instances a model incorrectly predicts. The most important goal for a detection model is finding an accurate tradeoff between detected objects and misclassified non-object instances.

This thesis project aims to find a model that precisely detects rip currents from nearshore imagery. Different methods for classification are compared, including: max distance from the average [11], Support Vector Machines (SVM) [12], TensorFlow convolutional neural networks (CNNs) [13], the OpenCV implementation of Viola-Jones [10] ,[14], and classifier stacking [15].

Principle component analysis (PCA) [11] reduces the number of dimensions for a data sample. This is important to the research as image data contain a large number of dimensions. This project attempts to apply PCA to detection by finding the maximum distance from the average data sample. A SVM [16] is another type of model that separates 2 classes with the greatest distance by finding the maximal margin of separation. Classifier stacking [15] is a meta-technique that trains a classifier on the output of many different models to increase accuracy.

TensorFlow is a framework developed by Google [13] and is applied to many machine learning obstacles. Convolutional neural networks built with TensorFlow detect objects from images. CNNs generate their own features through “image convolution” [4], [6]. Image convolution applies a “kernel” to generate image effects, such as edge detection and blurs. A kernel is a matrix paired with a region of an image and alters the makeup of the pixel it centers around. These filters extract data from each image and eventually become features for training a model. CNNs do not need their designs fine-tuned for a specific object when built with TensorFlow [13], which makes them promising candidates for developing a rip current detection model.

Another popular algorithm for object detection is “Viola-Jones” [10]. Viola-Jones employs a unique set of features for extracting data, known as “Haar features”. Haar features are sets of adjacent rectangles that correspond to different regions in an image. The value of a Haar feature is extracted from the grey-scale pixel values in the rectangles. The values of Haar features are generated while training the model. Viola-Jones has a technique to choose the best Haar feature for the dataset [10]. The advantage of Viola-Jones is the speed of extracting Haar features. Viola-Jones is trainable on many different types of images, including rip currents.

The OpenCV package for Viola-Jones [14], TensorFlow framework [13] for CNN models, Matlab for max distance from the average, and the Scikit-learn package [17] for basic classifiers are run to generate results. Methods are a collection of machine learning tools, which produces an object detection model in Python. Python is a popular programming language for computer vision and machine learning because of the functionality already included. The built-in implementation of the models are applied to the problem instead of constructing one from scratch. Python is also user friendly and a viable choice for this project. Models make a prediction on a dataset of rip current images and are compared with a predefined benchmark of test images. The dataset is developed from 514 rip current samples manually extracted from the surf zone of larger images. The large images are downloaded from a website of beach imagery that contains many photos from previously mentioned Argus sites [1]. A larger dataset is created by warping the samples from the smaller dataset with OpenCV.

This project expands upon the Haar feature space for the Viola-Jones classifier by finding features better suited for rip currents. This project also contributes a meta-classifier that improves upon the most accurate results of the current tools. The meta-classifier trains not only on the output of many different models, but on the optimal Haar features chosen by the same algorithm Viola-Jones employs.

## 2. Literature Review

The literature review covers various topics relating to machine learning and how it is involved with object detection and oceanography. The machine learning section covers different algorithms applied in research. These algorithms are employed by object detection and can extract data with Haar features. Certain oceanographic research involving rip currents depends on these machine learning algorithms and orthorectification.

### 2.1 Machine Learning

Machine learning concerns fitting data to a model with an algorithm [12], [7]. The models then make a prediction on a data sample with no class label. This prediction is conducted with previous occurrences of similar data. For example, a model trains to predict whether a given image is a rip current. The class of the image defines if it is a rip current. Images labeled 1 are of rip currents or are labeled 0 if not of rip currents. The images in class 1 are positive samples while the images in class 0 are negative samples. The learning algorithm processes training data from each image. An example of training data is the location of dark regions in an image. Machine learning implementations employ either “supervised”, “unsupervised”, semi-supervised or reinforcement learning [7].

Unsupervised learning is the training of a model when the dataset has no class labels, or “ground-truth” data [7]. The ground-truth data describes the correct prediction on a data sample. Updating a model is more difficult when its predictions cannot compare with ground-truth data. Unsupervised learning is applied when class labels are unknown. This can be useful for tasks such as feature selection. These types of tasks are not dependent on the class of each sample.

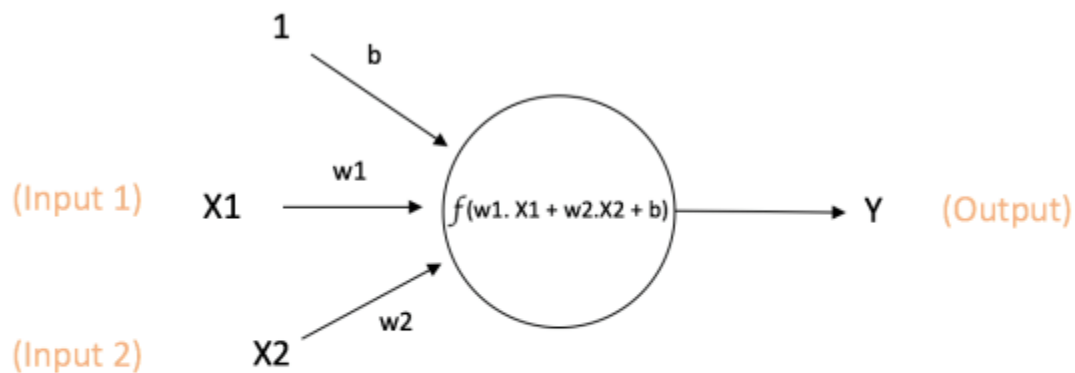
Supervised learning concerns a model trained with data that has class labels [7].

Supervised learning is the focus of this project because the location of rip currents in images is known. The learning is supervised since the model's predictions are compared to a class label. These forms of learning are the training each model goes through.

Training data are a set of features, which describe each sample in a numerical fashion [7]. The evaluation of a model is performed after training. A model compares the predicted class to the actual class. A “false negative” is an incorrect prediction on a positive sample. An incorrect prediction on a negative sample is a “false positive”. A correct prediction on a positive sample defines a “true positive”. A “true negative” identifies a correct prediction on a negative sample. The goal of a model is to maximize the number of true positives while minimizing the number of false positives. Maximizing true positives minimizes false negatives and minimizing false positives maximizes true negatives. Learning algorithms such as neural networks, support vector machines or Ada-boost enhance a model's predictions on data.

### **2.1.1 Neural Networks**

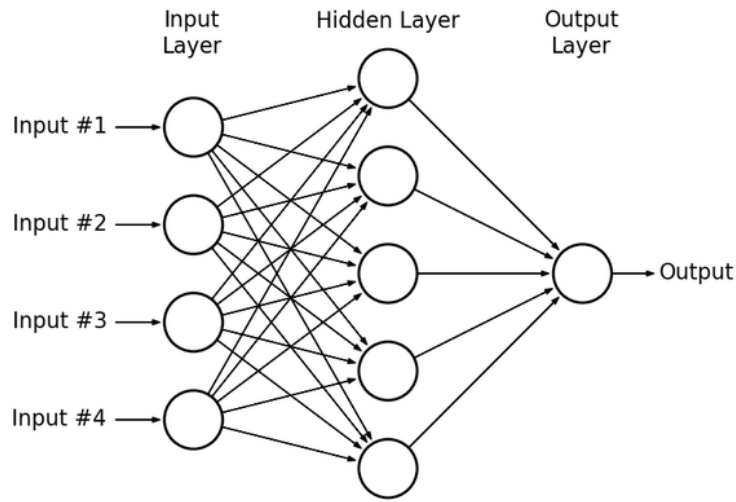
Neural networks form around the concept of the human brain [12]. Basic units in the network are “nodes”. Nodes are similar to a neuron of the human brain. A node is made of three parts, including: the input, the activation function, and the output. An example of a node is seen in Figure 1. In Figure 1, the output of the neuron  $Y$  is equal to a function of its inputs. These nodes connect to other nodes by links. The inputs of a node are outputs from other nodes. The outputs result from multiplying each link weight by an input, then adding them together. Links are input links or output links depending on the direction of the link.



$$\text{Output of neuron} = Y = f(w_1 \cdot X_1 + w_2 \cdot X_2 + b)$$

**Figure 1.** A neural network node. Here the output of the node  $Y$  is based around applying the function  $f$  to the input of the node,  $w_1 \cdot X_1 + w_2 \cdot X_2 + b$  where  $w_1$  and  $w_2$  are the weights of each link.  $b$  = bias,  $w_i = i^{\text{th}}$  weight.

Links have a weight associated with them, which determine how strongly a node's output is considered when the network classifies a sample. The nodes become a network when every node connects to every other node. An example network is in Figure 2. In Figure 2, a neural network is made of three different layers: the input layer, the output layer, and a hidden layer. The input layer takes the initial data. The output layer predicts the class label. The label is based on the output from nodes in the hidden layers. A “single layer perceptron” is composed of an input layer directly connected to an output layer [12]. A “multi-layer feed-forward network” has 1 or more hidden layers. The hidden layers represent the complex relationships between the input and output nodes. A more complex relationship needs additional hidden layers. The neural network trains by propagating through these layers until it reaches the output layer. The network's error is the output found versus the desired output.



**Figure 2.** An example of a multiple layer perceptron with 1 hidden layer.

“Back-propagation” is the network updating the weights in reverse order [12]. The algorithm for training a neural network is seen in Figure 3. In line 1 of Figure 3, the algorithm takes an input dataset  $x$  and an output dataset  $y$ . A data structure is setup to store the error for each node. Everything below line 3 repeats until convergence is determined at line 20. Small, random numbers are assigned to every weight in the network. Each node in the input layer is then assigned an input from the dataset. Line 9 starts forward propagation. The inputs and outputs for every node of every layer after the input layer are calculated through the activation function. In Line 13-14, the error of each node in the output layer is calculated in accordance with the class label. Line 15-17 applies back-propagation. This starts from the output layer and works backward. The error of each node is found until the input layer is reached. The weights update in lines 18 and 19. In line 20 the network checks for convergence. The algorithm exits if the model did not improve or ran a specific number of “epochs”. An epoch is an iteration of training for the network. The algorithm resumes at line 4 if these conditions are not met.



---

**Algorithm 1:** Neural Network Learning

---

```
1  input: input  $x$ , output  $y$ , network with  $N$  layers, weights  $w_{i,j}$  function  $f$ 
2  variable:  $E$  is a an array of errors indexed by  $i$  and  $j$ 
3  do:
4      for each weight in the network:
5          assign a small random number
6      for each example  $(x, y)$  in the dataset:
7          for each node in the input layer  $k$ :
8              assign the node to an input  $x$ 
9          for each layer  $m$  after  $k$ :
10             for each node  $n$  in  $m$ :
11                 set  $\text{input}_n = \sum (w_{m,n} * \text{output}(n_{m-1}))$ 
12                 set  $\text{output}_n = f(\text{input}_n)$ 
13             for each node  $j$  in the output layer  $q$ :
14                 set  $E[j] = f'(\text{input}_j) * (y_j - \text{output}_j)$ 
15             for each layer starting from  $q$ :
16                 for each node  $i$  in the layer  $q-1$ :
17                     set  $E[i] = f'(\text{input}_i) * \sum_j (w_{i,j} * E[j])$ 
18             for each weight  $w_{i,j}$  in the network:
19                 set  $w_{i,j} = w_{i,j} + \alpha + \text{output of node } i * E[j]$ 
20  while (terminating condition does not hold)
```

---

**Figure 3.** An algorithm for training a neural network.

---

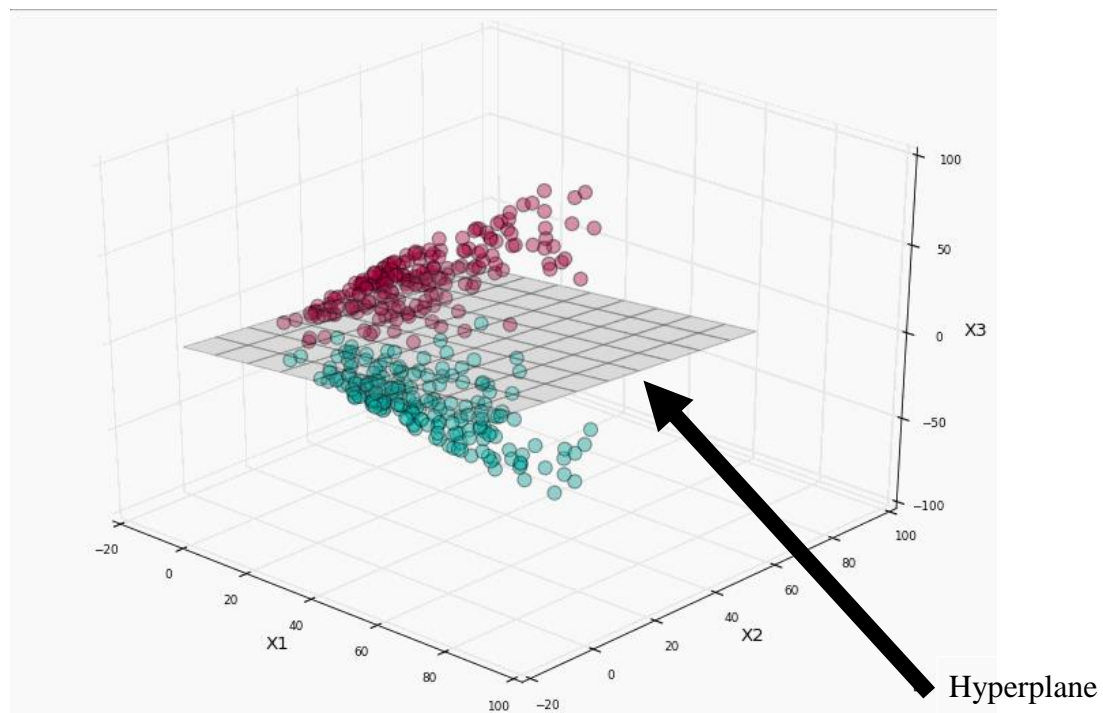
Images contain vast information for a dataset. Building an efficient network is unrealistic if the dataset is too large. For example, each pixel in a 224 by 244 image has 3 dimensions for its red, green, and blue color value. A node is created for every pixel and every dimension amounting to 150,528 nodes in the input layer. Parameters total to 22,658,678,784 after adding the links to each node. The memory needs of parameters yield an inefficient use of resources. A

224 by 224 image is relatively small, which makes the creation of nodes more difficult for larger images. Instead, a “convolutional neural network” is built.

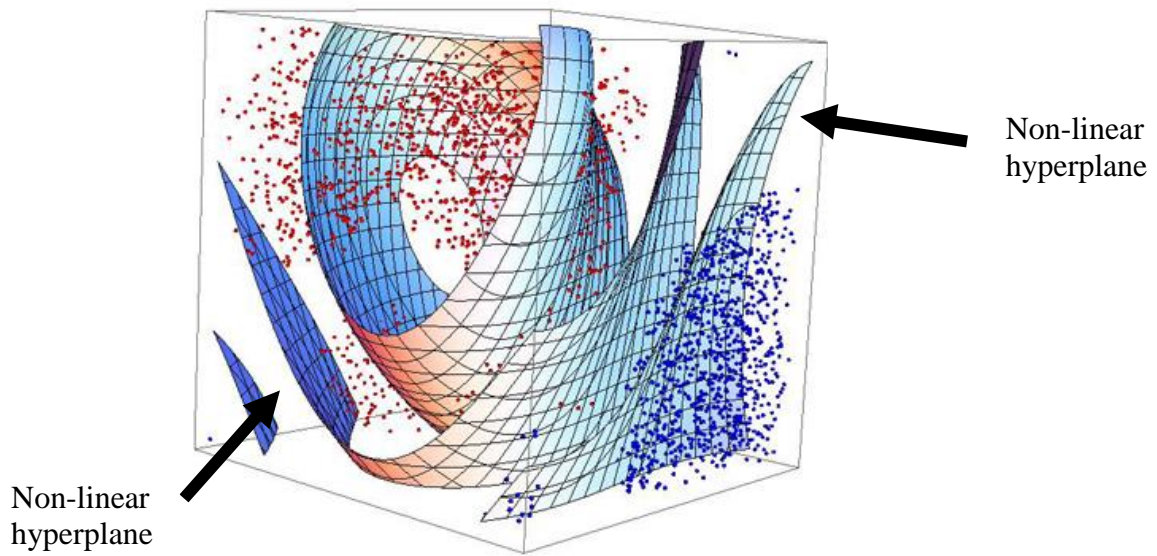
A convolutional neural network uses image processing techniques to extract and learn features [5]. The input from these features is sent to a traditional set of fully connected layers. The extraction of features is performed through “image convolution”. Image convolution is the application of a filter to an image. When performing convolution, a square matrix is placed over some center point in the image  $(x, y)$ . Each number in the matrix corresponds to a weight. An output is generated by setting  $(x, y)$  equal to the result of multiplying the weights in the kernel with the values at the positions around  $(x, y)$ . The results are added together to produce  $(x, y)$ . Convolution transforms pixels, which finds edges or generates blur effects. The effect depends upon the kernel. These operations are the basis for the “convolution layer”. The convolutional layer is the first layer after the input layer in a convolutional neural network. The convolutional layer contains a set hyperparameters, including: the number of filters (kernels), the distance every filter extends over the image (receptive field), the “stride”, and the amount of “zero-padding”. Stride is the number of pixels skipped when centering the kernel at the next pixel in the region. Zero-padding assigns values to regions outside the image, which handles cases when the kernel falls out of bounds. The layer accepts an input region of the image and outputs a corresponding volume of data based on the number of filters. For example, a region of input from an image is 7 by 7 by 3. The width, height, and depth of this volume are 7, 7, and 3 respectively. 2 kernels, 3 by 3 in size, are convoluted with the region. Convolution produces an output of size 3 by 3 by 2 if a stride of 2 is applied. The third dimension (depth) is equal to the number of filters given as input. Local connectivity is an important property of the nodes in a convolutional layer. Nodes are connected locally in terms of width and height, but fully-connected along the depth

dimension. The local connection heavily reduces the number of input parameters, which saves memory. An activation layer applies an activation function to the input volume after a convolutional layer. A recurrence of convolutional layers are each followed by an activation layer, which is needed to transform the input for future layers. The data alters into input for the fully connected layers as it passes through each convolutional layer since the number of input dimensions gradually decreases. The convolutional layers intermingle with pooling layers. The pooling layers reduce the size of the input volume on the width and height dimensions. This reduction in volume assists the computational effort for convoluting at future layers. The data is fed to a traditional feed- forward, fully-connected neural network for prediction on a class label.

### 2.1.2 Support Vector Machines

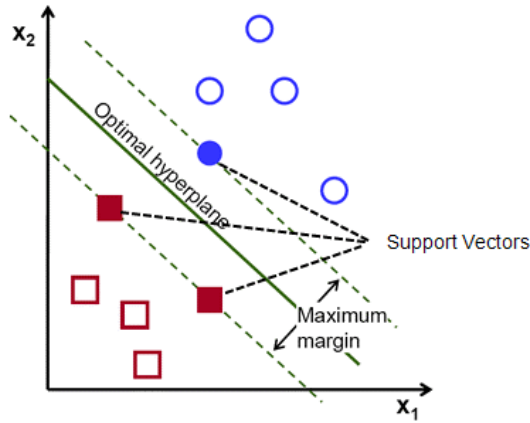


**Figure 4.** An example of a hyperplane in a 3 dimensional space.



**Figure 5.** An example of a non-linearly separability. The hyperplanes needed to separate the data are of complex shapes [16].

Another powerful learner is a support vector machine (SVM). A SVM finds the best formula to separate data into different classes that have the greatest distance between them. This is the “maximal margin” between classes and is represented by support vectors [12]. These outlying points help form the boundary line between their class and a neighboring class, called a “hyperplane”. An example is seen in Figure 4. In Figure 4, the hyperplane is the grid in grey separating the two classes by intersecting the green points and red points. This example is “linearly separable”. The relationships between non-linear input data and output data are not separated into well-defined groups. The output is a set of points similar to Figure 5 if a graph is shaped with non-linearly separable data. Notice the separating plane is not a function of the input since the hyperplanes are curved in many directions. The most accurate hyperplane has the largest margin of separation. “Margin” refers to the distance between the hyperplane and the closest vector on each side of the plane. An example is seen in Figure 6. The solid colored vectors in red and blue represent the support vectors and are closest to the separator.



**Figure 6.** A maximal margin. The margin is the distance of the closest support vectors in solid red and blue to one another. Each belong to a separate class. The optimal hyperplane is the center of the margin [18].

Therefore, the margin is twice the distance from the separator to the closest support vector. Data points are given a class, 1 or -1, in a binary classification problem. The separator is the set of points defined by (1).

$$\{x : w \cdot x + b = 0\} \quad (1)$$

Finding this set of points involves solving the optimization problem defined in (2). New points are classified by (3) after the optimum vector is found.

$$\operatorname{argmax} \alpha \sum \alpha_j - \frac{1}{2} \sum \alpha_j \alpha_k y_j y_k (x_j \cdot x_k) \quad (2)$$

$$h(x) = \operatorname{sign} (\sum_i \alpha_i y_i (x \cdot x_i) - b) \quad (3)$$

This solution works if the classes are linearly separable. “Data projection” is applied when classes non-linearly separable and plots data in a higher or lower dimensional space. For example, additional features are generated with (4), (5) and (6). Graphing in a 3-

$$x_1^2 \tag{4}$$

$$x_2^2 \tag{5}$$

$$x_1^{\frac{1}{2}} \cdot x_2 \tag{6}$$

dimensional space allows for more hyperplanes to be found because the data is now linearly separable [12]. Projecting into higher dimensions is the “kernel trick”. The kernel function is projecting the data and contains any number of dimensions. The dimensionality of the kernel function allows for projection into a potentially infinite number of dimensions. A “hard margin” is found when a margin separates data perfectly into each class by a straight line. Figure 6 is an example of a hard margin. The separator is kept noisy when data is noisier. “Noise” is a property of the data that distorts a data sample. An example is glare on a camera lens. A more realistic classifier is produced if the separator retains the noisy property of the data. The target separator is a “soft-margin” for non-linearly separable data. This margin adjusts itself for outlying points, which increases the accuracy of the final classifier.

There are some disadvantages to SVM’s. SVM’s tend not to scale to a dataset that has more dimensions than samples [7]. The correct kernel must also be chosen. A kernel to start with is a “radial basis function” (RBF) kernel because it projects data into an infinite dimensional space [7]. The input parameters “*gamma*” and “*C*” must also be chosen if the RBF is chosen. *C* is the penalty parameter for incorrect classification and *Gamma* is the coefficient for the kernel. These parameters are found through a process called “grid search”. Grid search tests a range of *gamma* and *C* coefficients through “10-fold cross-validation”. 10-fold cross-validation breaks data into 10 equally sized sets. The model trains on 9 folds and tests on the 1 remaining. The model’s performance is a combination of testing results from each holdout fold after training on the other 9. The best parameters are chosen based on the model’s performance.

## 2.1.3 Ada-Boost

Ada-boost searches through a set of features and ranks them in terms of error [19]. These features build “decision stumps”, known as weak classifiers. Decision stumps are simple classifiers that make a guess at classification, but are combined to form an accurate model. Ada-Boost continuously re-evaluates a model with a test dataset and weighs features accordingly. An example of the Ada-Boost algorithm is seen in Figure 7.

---

**Algorithm 2:** Ada-Boost

---

```
1  input: example images with labels  $\epsilon \in \{1, -1\}$  for positives  $i$  and negatives  $j$  respectively
2  input: number of weak classifiers  $n$ 
3  initialize weights  $w = 1 / (2 * \text{size of the set they reside in } (i \text{ or } j))$ 
4  for  $k = 1$  to  $n$ :
5      for weights  $w_{i,j}$  in  $i$  and  $j$ 
6           $\sum w_{i,j} = 1$ 
7      for each haar feature  $h$  in set  $p$ :
8          for each sample image  $x$  in  $i$  and  $j$ :
9              acquire value  $v_x = h(x)$ 
10             add  $v_x$  to value collection  $y$ 
11             sort  $y$ 
12             for each value  $v_q$  in  $y$ :
13                 error  $e_v = \min(e_v, \sum w_{y(0,q-1)}, \sum w_{y(q+1, \text{size}(y))})$ 
14             add  $\min_e(p)$  to weak classifier collection  $m$ 
15             for each sample image  $x$  in  $i$  and  $j$ :
16                 if  $\min_e(p)(x)$  is classified correctly:
17                      $w_x = w_x * \frac{e}{1-e}$ 
18 return  $m$ 
```

---

**Figure 7.** The Ada-Boost algorithm.

---

The inputs for Ada-Boost are a set of labeled images and the number of weak classifiers to return. At line 3 the weights of each input image are initialized. Lines 5 through 17 repeat for every weak classifier needed, which is based on  $n$ . Lines 5-6 normalize the weights of the images every time a new search starts. Lines 7-13 repeat for every feature in the feature space. A feature generates a collection of feature values after it is applied to the image collection. Lines 8-10 generate the feature values. Line 11 sorts the values and line 13 calculates the error for the feature based upon the weights of the images. The error for the feature is found by comparing 2 different values. Errors are assigned at each feature value in the sorted set. The first error results from setting all image above the current image to 1 and below to 0. The other error is found by doing the opposite operation of the previous error. Error calculations are done in one pass over the dataset. The feature with the minimum error is found at line 14, after every feature has an error. The feature with the lowest error becomes a weak classifier. This weak classifier is added to the current collection. In lines 15 through 17, the weight of the images update in reference to images incorrectly classified by the weak classifier. The algorithm continues until  $k$  equals  $n$ .

An advantage of Ada-Boost is its resistance to “overfitting” the data [20]. Overfitting occurs when the model learns the training dataset too well. This decreases generalization. Generalization affects the prediction on samples not in the dataset. Ada-Boost also does not require many parameters, which lowers memory needs [10].

## **2.1.4 PCA**

Principle component analysis (PCA) is a method of analyzing the “variance” of a dataset. Variance defines the difference between each data sample and the average data sample. PCA reduces the number of dimensions in a feature vector [7]. Any dataset has a corresponding “Eigen vector”. The Eigen vector contains the components that make up the greatest variance in



the dataset. “Singular value decomposition” generates the Eigen vector by factoring the covariance matrix [11]. The Eigen components are ranked from greatest to least variance retained [11]. Dimension reduction employs Eigen components to project the dataset upon. Projection is calculated through multiplication of the feature vector and the components, which results in a new feature vector for the data sample. This vector has a size equal to the number of components projected upon.

### **2.1.5 Other Machine Learning Algorithms**

This section briefly outlines algorithms that create the meta-classifier. These algorithms include: decision trees, K-nearest neighbors, Naïve Bayes and bagging.

A decision tree is a structure built from nodes and branches [21]. The branches extend from every node. Nodes correspond to a test on an input variable while the branches are the results of the test. Leaf nodes represent class labels when training a decision tree since they are the final decision for the tree. Nodes check an input feature value, typically being for inequality to a threshold value.

K-nearest neighbors (KNN) classify a data sample based on the class of a certain number of neighbors,  $k$  [22]. Neighbors are assigned a weight based on how far they are from the sample. Heftier neighbors have a greater impact on classification.

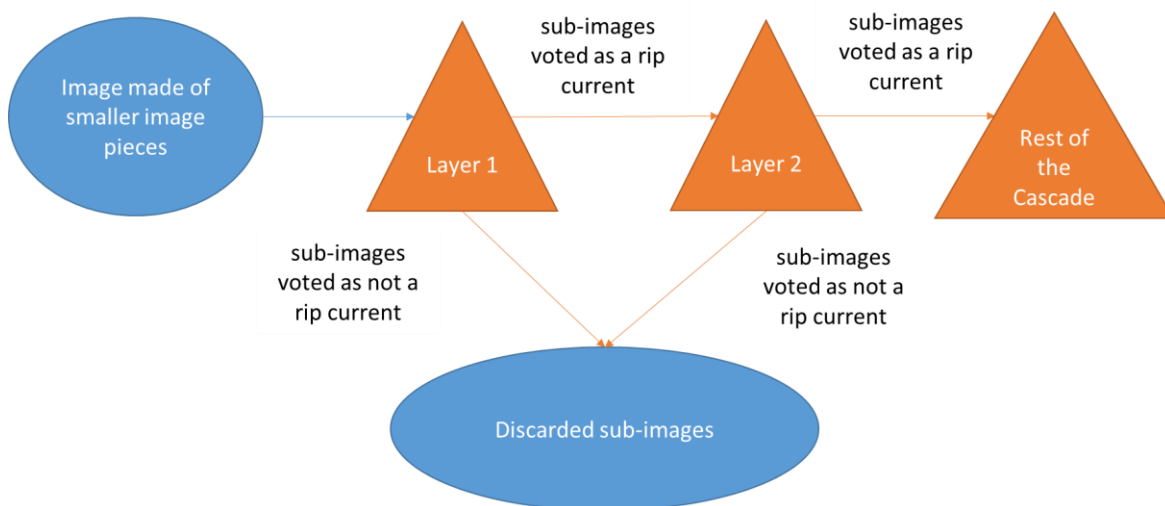
Naïve Bayes classifies a sample with the independent probability of features in the training data [23]. The model assumes the probability of every feature is not dependent on the probability of another feature, which drastically reduces the amount of computation needed to train a classifier.

Bagging involves a variety of models that make a prediction on a data sample. Each model votes on what the class of the data is [7]. The final classification has the most votes. For

example, the outputs of a neural network, KNN, and Naïve Bayes are given a weight. A neural network and Naïve Bayes model vote on the same class. The final output is their class since it is 2 to 1. However, the weight of KNN also affects the vote. If KNN has a larger weight than the other two classifiers combined, then it chooses the class of KNN. The class predictions of these algorithms assist in producing accurate object detection models.

## 2.2 Object Detection

Computer vision is the study of algorithms that alter and manipulate data contained in an image [8]. The algorithms extract features for machine learning. Object detection is a combination of computer vision and machine learning, which identifies certain regions of an image. In contrast, object recognition concerns identifying a particular instance of an identified object. Methods need “positive” and “negative” images. A positive image contains the object being sought after while a negative image does not. Object detection also employs a training algorithm, such as Viola-Jones or a convolutional neural network.



**Figure 8.** A Diagram showing attentional cascading for a rip current detector. Only sub-windows of an image that passes each layer of the cascade (found to be rip currents) are processed by further layers.

One type of object detection algorithm is the Viola-Jones object detector [10]. Viola-Jones uses Ada-Boost as the learning algorithm, Haar features, and “cascading architecture”. An example of cascading architecture is shown in Figure 8. The detector starts with every sub window in an image. Windows are checked at every position and scale in the image. Each sub window is passed to a layer of the cascade for processing. The layers that vote the window contains the object pass the image along to the next layer of the cascade. The layers that vote the image does not contain the object discard it. The model is evaluated by its “detection rates” and “false positive rates”. The detection rate is the number of positives correctly classified divided by the total positive windows while the false positive rate is the number of negatives misclassified divided by the total negative windows. A cascade’s detection rate and false positive rate are equal to the product of every layers’ rate. For example, a cascading object detector has 4 layers. Every layer has a detection rate of 0.99. The total detection rate for the classifier is  $0.99^4$  or 0.96.



**Figure 9.** An image that has been classified based on faces using Viola-Jones. The faces are surrounded by red boxes.

Notice how the detection rate decreases as the number of layers increase. The detection rate decreases because each layer cannot achieve a detection rate of 1.0, which causes them to lose positive images. In contrast, the product of the false positive rates work in favor of the cascade. Layers train to reject false positives of the previous layer. A higher detection rate and lower false positive rate make it difficult to find a collection of suitable weak classifiers.

Only 0.067 seconds are needed to evaluate a 384 by 288 image on a 700 MHz Pentium III processor [10]. The rapid detection speed creates “real-time object detection”. Real-time object detection involves detecting objects at the speed of video frames and is associated with the

---

**Algorithm 3:** Viola-Jones Cascading Classifier

---

```

1  input: target false positive rate  $fp$ 
2  input: false positive rate per layer  $fp_i$ 
3  input: detect rate per layer  $dt_i$ 
4  input : set of positive and negative images  $j$ 
5  variable  $i = 0$  starting detection rate  $d = 1.0$ ; starting false positive rate  $f = 1.0$ 
6  while  $f < fp$ :
7       $i++$ 
8       $n = 0$ 
9      while  $fp_m < f * fp_{i-1}$ 
10          $n++$ 
11         collection of weak classifiers  $m = \text{Ada-Boost}(n, j)$ 
12         evaluate collection on test set of images
13         while  $dt_m < d * dt_{i-1}$ 
14             decrease threshold  $k$  for collection and re-evaluate
15         set  $m$  as a layer  $i$  for cascaded classifier
16         delete any negatives in  $j$  correctly classified

```

---

**Figure 10.** The Viola-Jones Algorithm.

---

rejection rate of negative samples. The rejection rate of negative samples depends on how low the false positive rate per layer is set while the misclassification rate of positive samples depends on how high the detection rate per layer is set. Negative windows are rejected quicker if the false positive rate is lower. Positive images are more slowly discarded if the detection rate is higher. An example of an image classified by the face detector is shown in Figure 9. The detected faces are in red boxes.

An example of the Viola-Jones algorithm is seen in Figure 10. The Viola-Jones object detector is given set of positive and negative examples on which to train and test. Other inputs are an acceptable detection rate per layer, a target false positive rate, and a false positive rate per layer. In line 5,  $i$  is initialized to 0.  $I$  represents the current layer the algorithm is building. The starting false positive rate and detection rate for the cascade are initialized to 1.0. The classifier considers every image a rip current when they are set to 1.0. Lines 6 through 16 repeat until the classifier achieves the target false positive rate. At line 7 a new layer of the cascade starts. Line 8 sets the number of weak classifiers returned from Ada-Boost to 0. Line 9 through line 14 repeat until the layer achieves the false positive rate needed. In line 10 the number of weak classifiers needed increases. This starts at 1 to try the fastest possible computation time for each layer. At line 11, Ada-Boost ranks every feature in the feature-space in terms of weighted error with the algorithm described in section 2.1.3. Ada-Boost returns weak classifiers in a group of size  $n$ . The collection becomes a potential layer of the cascade. On line 12 the algorithm evaluates the performance of the layer by classifying a test set of images. Lines 13 and 14 repeat until the layer has an acceptable detection rate. Decreasing the threshold causes the detection rate and false positive rate to increase. At line 9, the algorithms checks if the layer still has an acceptable false

positive rate. A layer with an acceptable false positive rate becomes a layer of the cascade. Otherwise, the layer is discarded and the process starts over, with  $n$  increased by 1.

Open-source Computer Vision (OpenCV) is a Python package [14] for object detection employing the Viola-Jones algorithm. The advantage the OpenCV implementation has over the original is a more robust feature set [24]. This set includes features rotated at 45 degree angles. OpenCV automatically produces a dataset by superimposing a slightly warped, positive sample on top of negative sample. This introduces a risk of overfitting to an artificial dataset. The object detector becomes adequate for only detecting artificial samples if the samples do not contain meaningful data. The OpenCV package also saves a cascade by writing every layer to a file. The files are accessed at a later time to continue training instead of building a new cascade from scratch.

Recently, Google has implemented an object detection framework containing convolution neural nets [9], called TensorFlow. This framework employs checkpoints of previous models, which accelerates training to just a few hours [13]. Models built with TensorFlow spend weeks training on many different objects. Then, they fine tune the final layer to the detect the object needed [13]. TensorFlow has a variety of configurations for building a customized network. Two examples are the Inception and Mobilenet models.

In [6], the authors address a need for an efficient convolutional neural network. The need is generated from the current method for improving performance of a network, which involves increasing the width and parameters of the network. This makes the network prone to overfitting. The Inception model is introduced to alleviate the issue. 5 by 5 convolutions are expensive to compute, given enough filters are applied at a convolution layer. A 1 by 1 convolution is applied to the output volume from the previous layer before performing larger convolutions, which

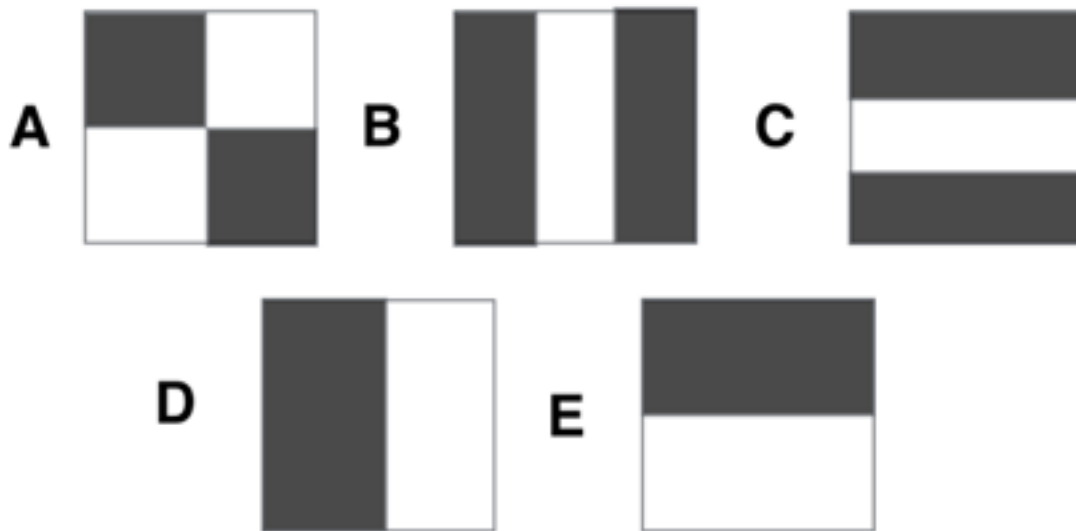
reduces the depth dimension. A 1 by 1 convolution keeps the same width and height of the input, but reduces the depth if the number of filters applied is less than the previous output volume's depth. 1 by 1 convolutions make larger networks without sacrificing memory efficiency and make local machines with less memory able to build larger networks. Inception networks consist of inception modules, which contain a number of 1 by 1 convolution operations followed by a larger size convolution or a pooling operation. The final Inception model has one fully-connected layer at the end of the network. During training, it has a fully connected layer at the end of each module in addition to a connection to the next module. This makes back-propagation start at every module, which speeds up the training process. The fully-connected layers are discarded after the model is complete.

Mobilenets are introduced in [4] as a different approach to reducing the size and computational effort of convolutional neural networks. These networks contain “depth-wise separable convolutions”. Depth-wise separable convolutions break up the convolution step into 2 parts. The first part involves applying a single filter to every input volume depth, called “depth-wise convolution”. The second step applies a 1 by 1 convolution to the input volume. The 2 results are added together, which achieves a similar effect of normal convolution, but at a reduced cost of parameters. The paper also introduces a method of thinning the network with a “width multiplier”. This is a number less than one multiplied to the input depth and output depth. The width multiplier reduces the width of each layer of the network, which further decreases the cost of parameters.

Convolutional neural networks prove beneficial for building an object detector of many different objects. Other approaches, like Viola-Jones, weigh heavily on the features they calculate. The features Viola-Jones calculates are Haar features.

## 2.3 Haar Features

Haar features form the main basis for a Viola-Jones [14] and are rectangular regions of an image. The regions expose unique features extracted for training. Examples of Haar features are seen in Figure 11. The feature (A) is known as “Four”, (B) as “Three Horizontal”, (C) as “Three Vertical”, (D) as “Two Horizontal”, and (E) as “Two Vertical”. The feature types are chosen in accordance with similarities of light and dark values in all faces [10]. The similarities are found by taking a large set of face images, all of the same size, and averaging each pixel. An example of an applied Haar feature is shown in Figure 12. The black regions of the Haar features are covering the image with different orientations. The feature values are not calculated from the original image, but from the “integral image”. An example is seen in Figure 13.

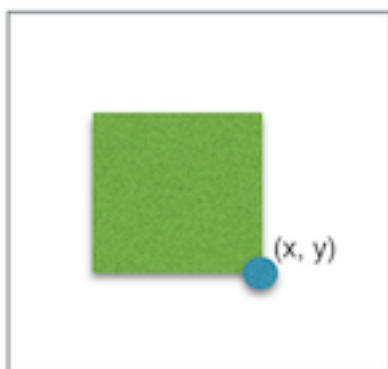


**Figure 11.** Example haar features (A) The “Four” feature, (B) The “Three Horizontal” feature, (C) The “Three Vertical” feature, (D) The “Two Horizontal” feature, (E) The “Two Vertical” feature.

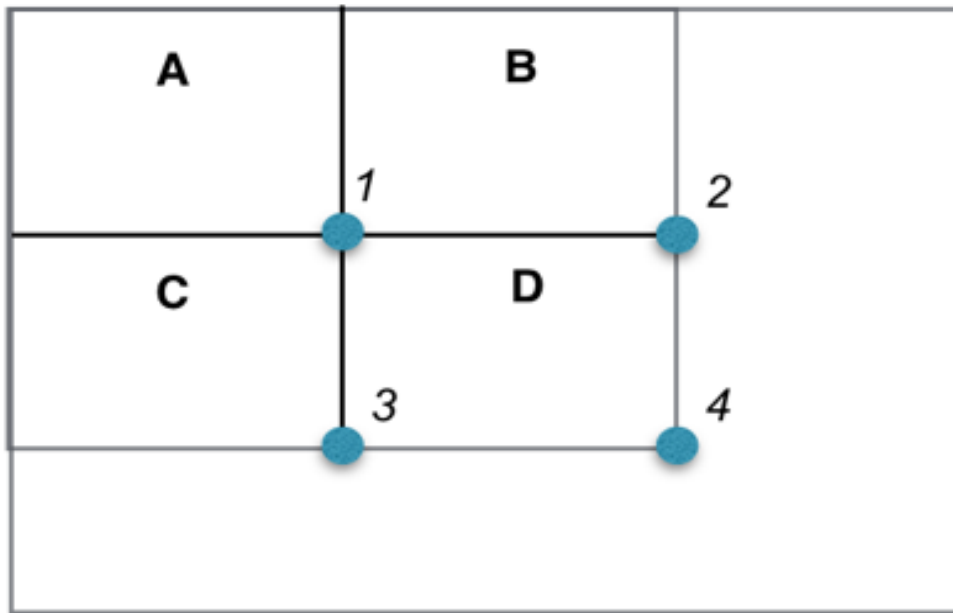




**Figure 12.** Applying a Haar feature to an image. The feature sits on top of an area of the image. The value at  $(x, y)$  is the sum of the gray-scales values above and to the left of the pixel. The difference between the areas under the rectangles is taken after the integral image values for each of the rectangles are calculated. The difference of areas is the sum of integral image pixels in white rectangles subtracted from the sum of pixels in black rectangles. An example of this is shown in Figure 14. Figure 14 is a Haar rectangle with four regions: A, B, C, and D. The region of A is the integral image value at point 1.



**Figure 13.** Integral Image point. The point  $(x, y)$  = sum of all pixels above and to the left of  $(x, y)$ .



**Figure 14.** Integral Image Regions: (A) a region of the image denoted by point 1, (B) a region of the image denoted by  $2-1$ , (C) a region of the image denoted by  $3-1$ , (D) a region of the image denoted by  $4+1-(2+3)$ .

Region B is the integral image value at point 2 subtracted from the value at point 1. Region C is the integral image value at point 1 subtracted from point 3. Region D is the integral image value at 4 added to the value at point 1. The final result is  $(1+4) - (2+3)$ . Applying a Haar feature to an image yields a single value, which is cheap to hold in memory and quickly computed. Integral image values are accessed instantly while normal image values are iterated over. Therefore, normal image values take more time to compute. The access speed results in immediate evaluation time. A Haar feature placed at every width and height in a 24 by 24 image creates over 160,000 features in the feature space. Ada-Boost iterates over the feature space every time a weak classifier is needed. Adding more features slows down the search.

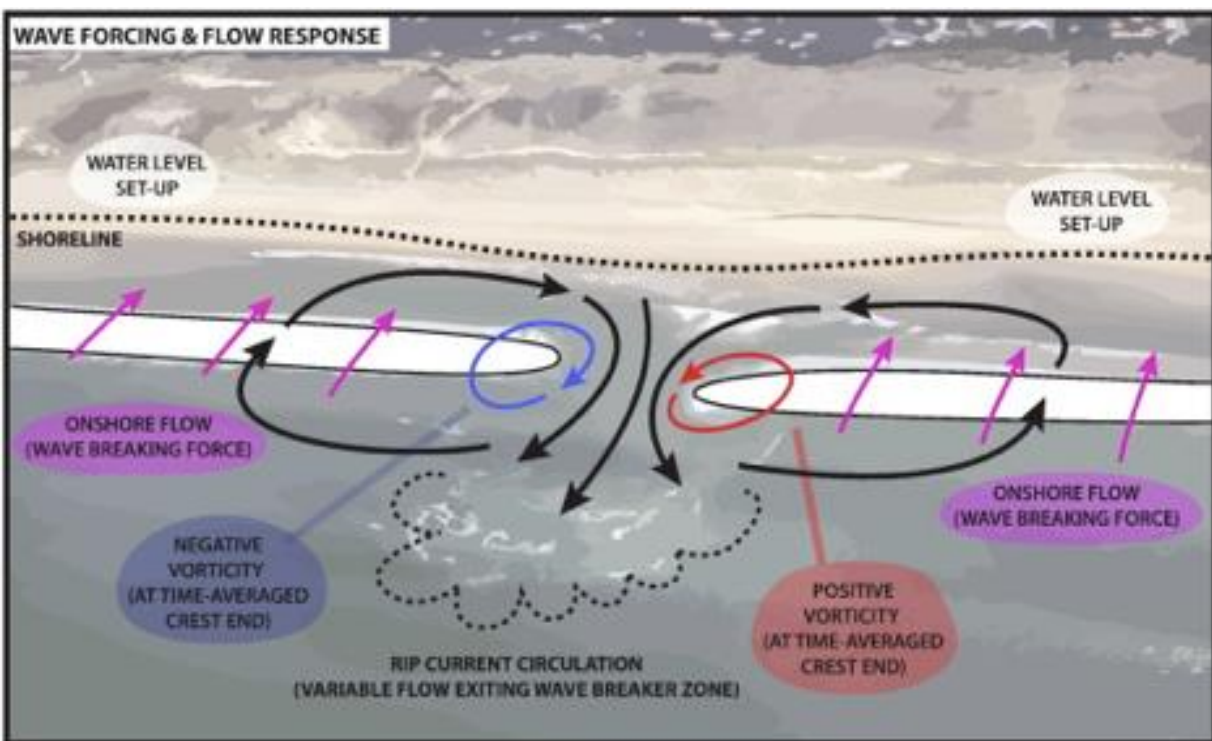
Normal Haar features are limited upright regions in horizontal and vertical directions. The OpenCV implementation has a more robust set with rotated and upright regions [24]. The

original Haar features cannot locate tilted faces, which creates the need for rotated features. These features are rotated by 45 degrees to mimic a rotated face.

There is no alternative to blunt force when searching for the optimal feature [25]. Searching the entire feature collection comprises the inefficiency of these features. Most implementations of Viola-Jones have Haar features for face detection because they are optimized for faces. There has been no further research into optimizing these features for different objects.

## 2.4 Oceanography

Oceanography is an Earth science covering a range of topics, including: ocean currents, waves, plate tectonics and the sea floor [26]. Nearshore research comprises one area of oceanographic research. An example of nearshore research are rip current studies. Rip currents are narrow regions of the surf zone, are perpendicular to the shoreline, and are created by wave

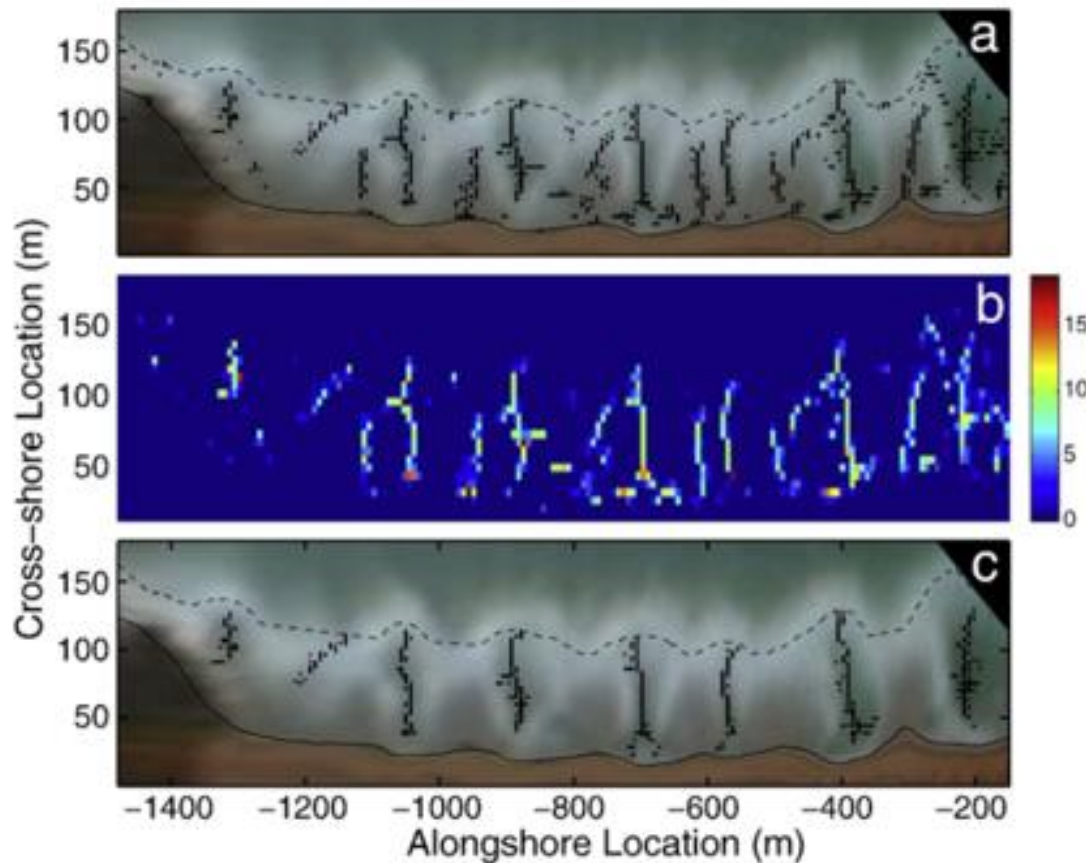


**Figure 15.** A Depiction of a rip current. The wave breaking force toward the shore is in purple. The backward circular motion caused from this force is generating a rip current [27].

breaking [28]. An example is shown in Figure 15 [27]. In Figure 15, the dotted line near the top of the figure is a shoreline. The forces causing waves to move toward the shoreline are in purple. The wave breaking forces are at opposite ends of the rip current and are in red and blue. More than 100 people annually die from rip currents in the United States alone and rip currents account for over 80% of the rescues performed by beach lifeguards [29]. People have a difficult time recognizing rip currents when not trained to spot them [30]. Rip currents also play a large role in beach erosion because of their seaward force from shoreline [31], which causes a large amount of erosion for countries like Korea. Rip currents images assist in climate studies for oceanography, but must be manually annotated by hand [32]. One image can contain many different rip currents, which makes annotating thousands of images unreasonable. This can make climate studies harder to conduct since they need the number of rip currents in each image and the location of each rip current. The previous reasons generate the importance of an accurate method for automatic rip current identification in images. One possible solution is machine learning because it has become a popular, successful approach for locating certain regions of an image [4-6].

Some oceanographic research is conducted with computer algorithms. One study involves rip current behavior and video imagery [28]. In this study, a semi-automated algorithm is introduced and adopts “local minima and maxima” for detecting rip channel patterns. The local minima and maxima identify darkest and lightest areas of the image, respectively. The algorithm is semi-automated because of human correction. Finding rip currents with light and dark values is a contribution as the center of rip currents appear darker. A sample from the study is seen in Figure 16 [28]. The rip currents and shoreline are seen from a bird’s-eye view at the

bottom of the image. In Figure 16, (a) the results of the automated portion of the algorithm are shown.



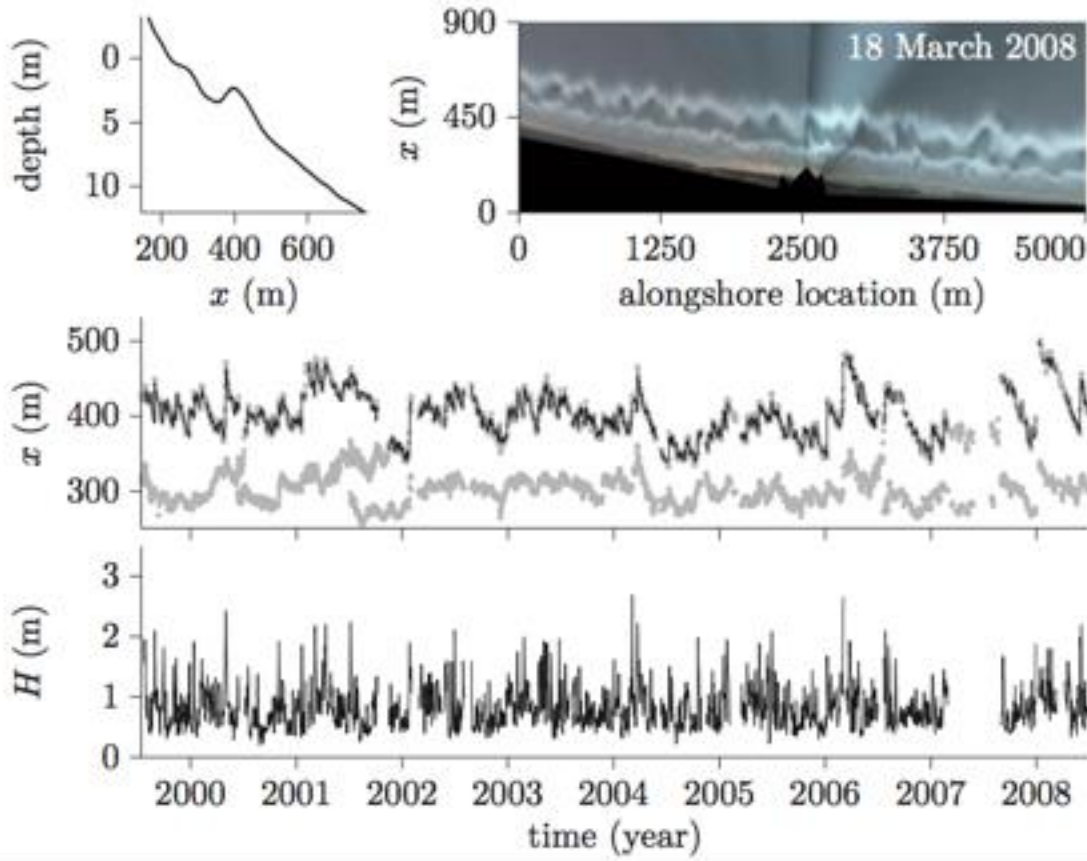
**Figure 16.** A semi-automated process for detecting rip currents: (a) the results of the algorithm guessing at where rip currents are located, (b) the maxima and minima pixels in the image within a 7.5 by 7.5 region, and (c) the final results once a human has corrected the initial results.

Figure 16, (b) shows warmer colors for minima and cooler colors for maxima. Figure 16, (c) shows the same shoreline after human modification. The algorithm finds the local maxima and minima located in a 7.5 by 7.5 area of the image. All pixels outside the dotted area are not considered since they are not near the shore. A few specification rules are introduced to objectively remove pixels not considered part of the rip current. First, a rip current must be a part

of at least 5 connected local minima pixels. Second, segments of rip currents within 40 m of one another are joined. Third, the rip current is split into 2 different rip currents if it has sections connected by a line parallel to the shore. Finally, a rip current is removed if it does not extend from the shoreline to the tip of the bar-line (dotted line). These images all have a similar orientation, which allows the algorithm more generalizability. This makes the algorithm useful for images distinct from the samples in the study.

Another study has already noted disadvantages of rip current imagery [33]. This study discusses the “noise” found in such imagery. Noise makes it difficult to detect rip currents in shoreline imagery because of how it interferes with meaningful data. An example could be rain droplets or sun glare in the images. The solution is to apply a Gaussian blur, which filters out noise. This involves a blending of color values of a particular group of pixels. Next, the image is segmented into different regions based on the red, green, and blue values. Finally, color values are grouped together to obtain a synthetic, noiseless image. A digitized version of rip currents are manually created as a detection benchmark. This makes the process semi-automated. The detection rate is 34% for original images and 41% for synthetic images. Correcting noise helps minimize the variance of the samples, which generalizes the application of the algorithm.

In the following study, neural networks predict sandbar movement [34]. Sandbar locations are output nodes and wave heights are input nodes. Predictions on sandbar locations are more accurate when wave height is high and less accurate when the wave height is low. Previous models cannot capture the data relationships because the data is non-linear. Hidden layers capture non-linearity in data, which makes the study a contribution [7]. Example data from the experiment is seen in Figure 17 [34]. The image in the top right is a bird’s-eye view and is similar to images from the last study.



**Figure 17.** The white foam from the image in to top right is transposed onto  $x$  (m).  $H(m)$  is the wave height at time  $t$  (year) corresponding to sandbar location at time  $t$  (year).

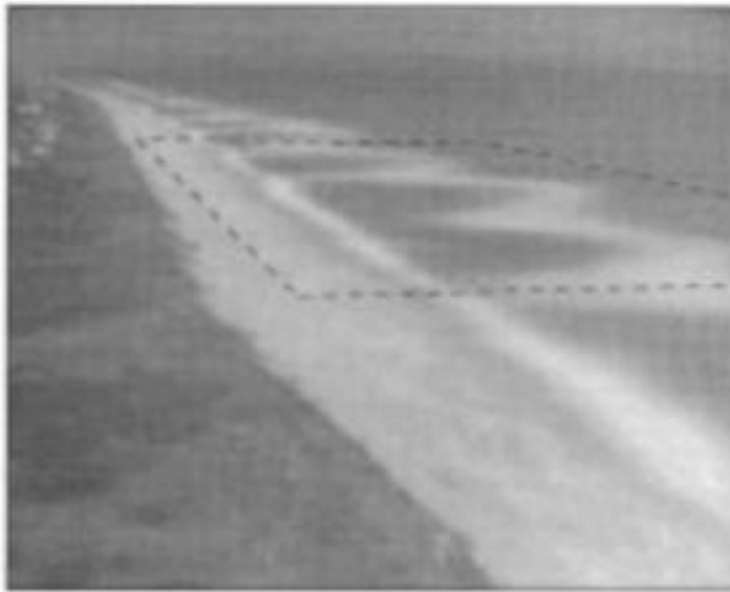
White, foamy regions from the image are placed onto a horizontal plane  $x$  (m).  $H$  (m) is wave height over an 8 year period. Sandbar location is the data's class label. In this situation, the neural networks are "recurrent". Ergo, the output of one time step is the input for the next time step. For example, there is a sandbar location  $x$  (m) at time  $t$ . Wave height, averaged over time  $t-1$  through time  $t$ , predicts sandbar location at time  $t$ . Time step  $t$  becomes  $t-1$  and predicts sandbar location for the next time step  $t$ , previously  $t+1$ . A trained network predicts sandbar locations outside of the training set.

Rip currents are a neglected focus of study for machine learning. The imagery from previous studies has the potential for assisting in rip currents identification since it is

“normalized”. Normalization constrains data between 0 and 1, which highlights similarities in extracted features. For images, this involves aligning each image to the same orientation and size. These studies rely heavily on the bird’s-eye view images of the shoreline. These images are not taken from a bird’s eye view, but created through the process of orthorectification.

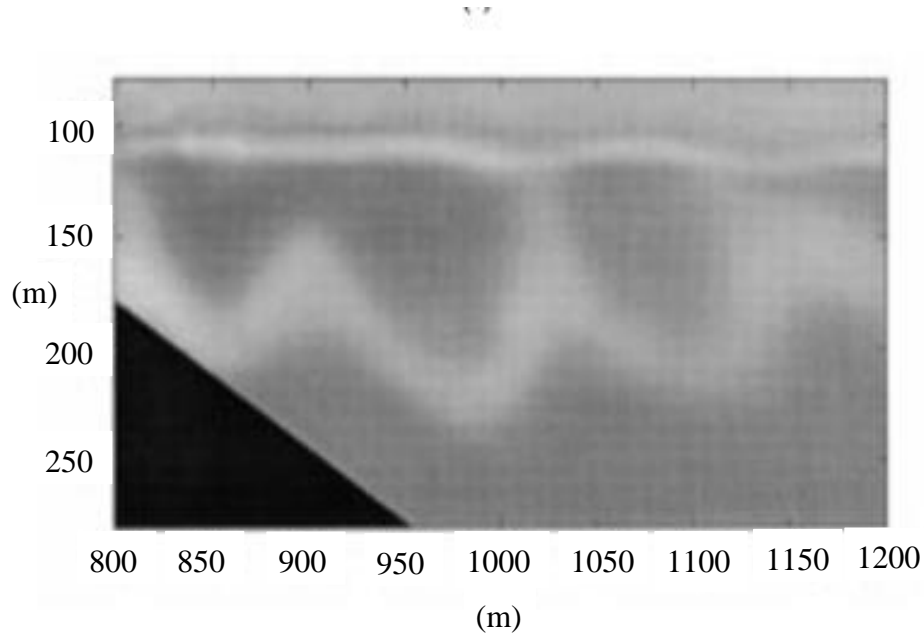
## 2.5 Orthorectification

Previously mentioned studies [28],[34],[33] include imagery of rip currents. This is not simple photography. Multiple cameras capture images of the shoreline at several different angles. These images are “orthorectified”. This process converts the perspective of an image into a different perspective [3],[2]. A 10 minute clip of video footage is averaged at every pixel to enhance the wave and sandbar behavior in a resulting image. The image looks similar to Figure 18 after it is averaged over 5-10 minutes [3].



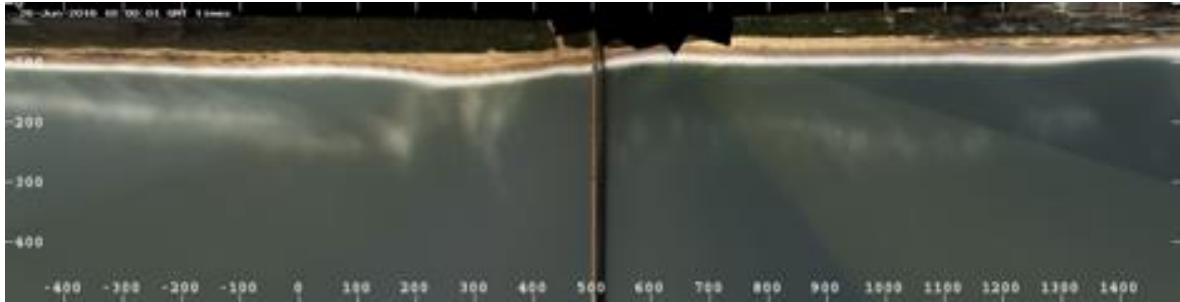
**Figure 18.** A time averaged image of the shoreline from a nearby camera.



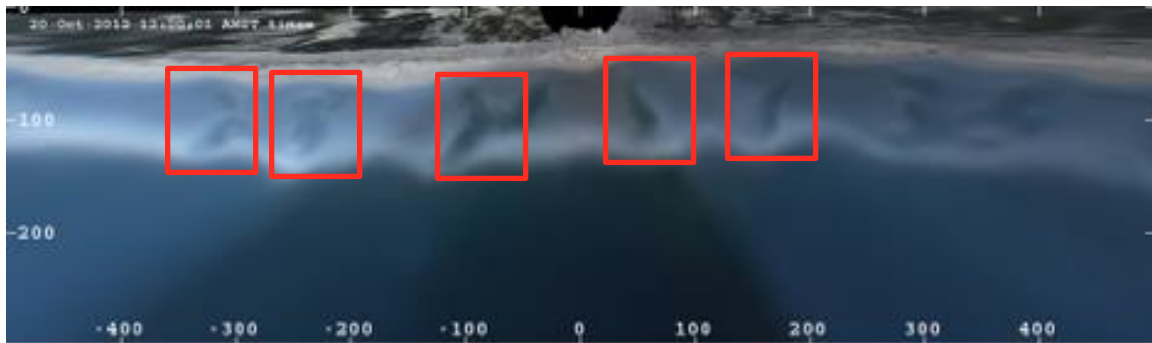


**Figure 19.** The dotted region of Figure 18 transposed onto a 2-D plane.

In Figure 18, the surf zone of a beach front image is smoothed over time. The smoothing allows the motion of the waves and sandbar location to appear well-defined. The center of the image is used as a reference point when transforming coordinates  $(x, y, z)$  into a 2 dimensional plane  $(u, v)$  through matrix multiplication. The camera is calibrated after the multiplication is complete. The camera is corrected through the equation described in [3],[2], hence the dotted region in the image looks like Figure 19 [3]. Figures 18 and 19 are taken from an older Argus model from 1997. Argus sites [1], are set up at many beaches around the world. Multiple cameras are set up for one beach. Combining the images at different locations produces a full view of the shoreline. This is seen in Figure 20. Figure 20 is an example of an image from the Duck, North Carolina camera site. In Figure 21, rip currents are darker blue regions of the surf zone, are surrounded in red boxes, and are taken from Secret Harbour, Australia. Rip currents have less defined shapes and sizes than other objects. There is some uncertainty whether the dark blue region farthest to the right is a true rip current due to the horizontal orientation.



**Figure 20.** An updated picture from an Argus site [1].



**Figure 21.** An Argus picture of rip currents.



**Figure 22.** An Argus picture with rain droplets on the lens of one camera.



**Figure 23.** An Argus picture with the sun blocking a camera.

Typically, a rip current has a perpendicular orientation to the beach.

Images can be distorted or unclear. An image looks like Figure 22 on a rainy day. In Figure 22, rain is distorting one of the cameras. The rain produces a “stretched” look for the area marked in red. In addition, Figure 23 is produced if the sun causes a reflection off of the ocean. In Figure 23, lighter areas covering the shoreline are annotated in red. These distortions make images not as desirable for a dataset. The images need to be as clear as possible to retain the most meaningful data for a model. The orthorectified images create a usable dataset because of their similar orientation for rip currents. A dataset of these images builds a rip current object detector. Rip current regions in the images are pulled out of the larger image. The rip currents are processed to create a list of descriptive features. The descriptive features train different models for classifying images. There have been no studies applying machine learning to rip current detection in orthorectified images.

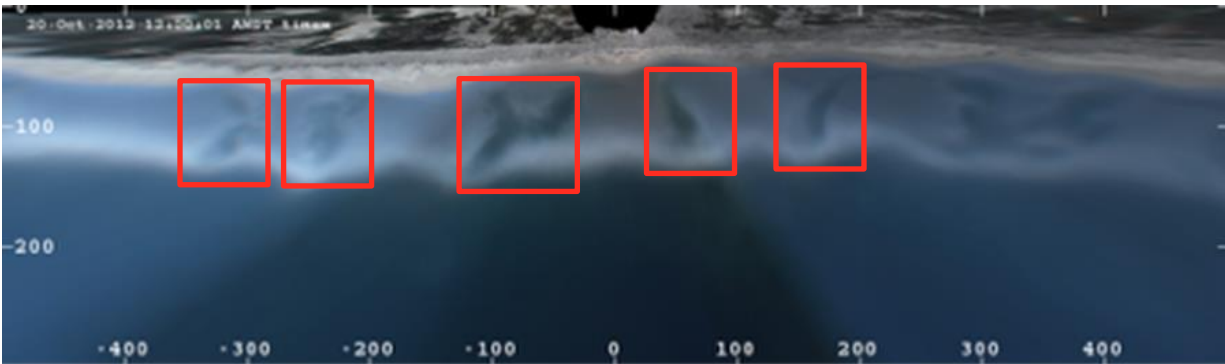
### 3. Methodology

The following topics describe the dataset and how it builds rip current object detectors.

Rip Current Dataset discusses how and where rip current images are generated from. These images create training data and a benchmark for comparing different methods of rip current classification. Next, the implementation and package details of max distance from average, SVMs, convolutional neural network, Viola-Jones, and the meta-learner are shown. The new features are revealed in the meta-learner section. The implementations explain how models conduct classification. Methods generate a collection of values for comparison of detected rip currents, including: accuracy, detection rate, false positive rate, and false positive count.

#### 3.1 Rip Current Dataset

The data is downloaded from the Coastal Imaging lab web site [1]. The website contains a collection of backlogged imagery from different beaches. An example is shown in Figure 24. Figure 24 contains 5 rip currents extracted for training. These images are orthorectified by the



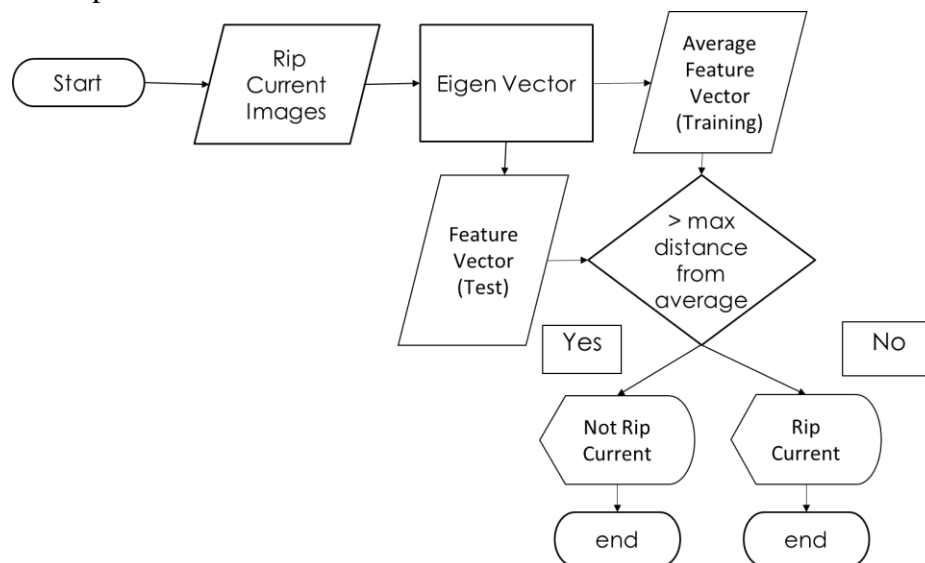
**Figure 24.** A depiction of beach imagery from an Argus website. It contains 5 rip currents annotated in red.

method mentioned in section 2.5. Orthorectification allows rip current images to have similar orientation, which spawns similar features between the samples. All rip currents come specifically from the Duck, North Carolina and Secret Harbour, Australia datasets. 514 examples

of rip currents, 24 by 24 in size, are extracted by hand from every image after shoreline images are downloaded from the site. In addition, 800 24 by 24 negative examples are pulled in a similar fashion from the surf zone. Surf zone negatives are part of the dataset because rip currents only appear in images with a surf zone. Faces and other objects appear in many different places. The false positive rate for the model reaches an acceptable level sooner when there are fewer types of negatives to classify. Models created from negatives in any image are compared against surf zone models.

## 3.2 Max Distance from Average

This section explains how to generate the max distance from the average rip current. Then, the implementation of the classifier is described. This is the baseline classifier. Every other classifier improves upon it.



**Figure 25.** A depiction of the max distance from the average classifying an image.

### 3.2.1 Matlab

PCA runs on the set of rip currents, which finds the max distance from the average. Matlab code generates the Eigen components for the rip currents. This code is found in appendix

D. Only training on positive samples reduces the training data needed to create a classifier. This eliminates computation effort as the size of the dataset has a significant impact on training time.

### **3.2.2 Implementation**

A depiction of the implementation is seen in Figure 25. Every image is a 1 by 576 vector when processed column wise. The 24 columns in the image are appended together to produce an input vector. Singular value decomposition yields the Eigen vector for the collection of rip current vectors. The vectors are projected toward the Eigen vector through matrix multiplication, which creates a feature set for the image. The maximum distance from the average feature vector is produced by finding the average feature vector. The resulting feature vector has size 1 when 1 component is projected toward. The average of the single values creates a midpoint in a range of possible values. The absolute value of the greatest difference between any one sample feature and the average is a threshold. This threshold classifies the images. Every component is projected toward, which identifies the component with the smallest threshold. A projected sample is classified as a rip current if it has a distance from the average less than or equal to the threshold. It is classified as a non-rip current if the distance from the average is greater than the threshold.

## **3.3 Support Vector Machines**

This section reveals the SVMs built for classification. Formulas for processing the SVM features and how the SVMs train on them are discussed.

### **3.3.1 Features for training**

Meaningful features are the most important part in creating an accurate SVM [7]. Haar features from Viola-Jones are created and applied to every training image. Haar features capture

a generous portion of image data. Not every feature is considered because the feature space is too broad. SVMs do not scale well when there are more features than there are samples [7]. Instead, an average of 10 Haar feature types are placed into the feature vector, which limits the training vector size to 10 at the cost of descriptive information. This includes 5 new Haar identified in section 3.7.1. Features in each category are grouped for averaging. The ratio of black to white pixels in the image and “circularity” are also appended to the vector. Circularity represents how close to a perfect circle an object is. An image consisting of either black or white pixels produces a ratio between the two when they are counted and divided. A black and white image is developed from setting every pixel in an image to totally black or totally white, which depends on a threshold value. Circularity is calculated from a segmented image. A segmented image is the result of applying a Matlab routine to a black and white image. Formula (7) produces circularity. In the segmented image,  $bwow$  is the number of black pixels without a white neighbor and  $bww$  is the number of black pixels with a white neighbor.

$$4\pi(bwow/bww^2) \tag{7}$$

Circularity is a significant feature as rip currents have a semi-circular shape to them.

### 3.3.2 Implementation

The Scikit-learn package for Python creates and trains the SVM [17]. The SVM has a RBF kernel because it projects data into an infinite dimensional space, which reveals hyperplanes. 10-fold cross-validation evaluates the classifier. The model randomly chooses 10% of the training data. The model tests on the 10% holdout after it finishes training on the other 90%. Next, the model chooses 10% of the untested data and it repeats the process. This continues until the model tests on 100% of the training data. The SVM runs on a Linux server and takes

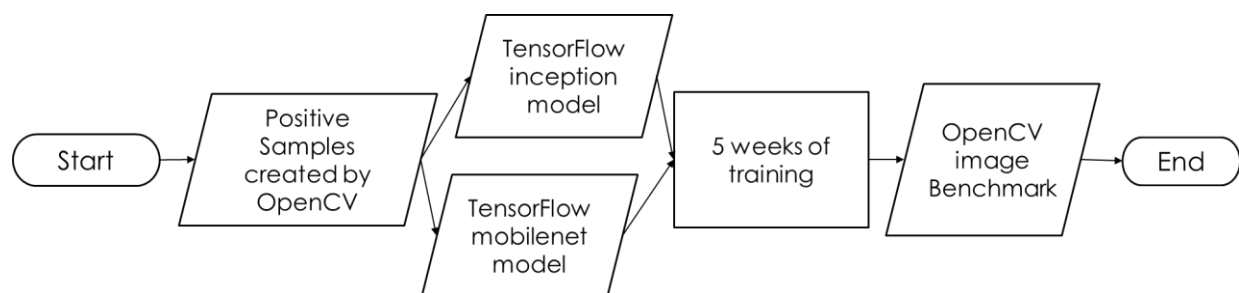
only a few minutes to complete. A robust scaler from the Scikit-learn package scales the data while grid search optimizes the results by finding  $C$  and  $\gamma$ .

## 3.5 Convolutional Neural Networks

This section explains the CNNs compared. First, TensorFlow and the input needed to train the networks are described. Next, how the networks intend to accomplish object detection is revealed.

### 3.5.1 TensorFlow Framework

Tensorflow is a Machine Learning framework developed by Google. It is praised for its fast training and ease of use [13]. TensorFlow supplies convolutional neural network configurations for building models. Images generated from the OpenCV package train the models. The training of convolution neural networks requires a large number of annotated images. There are about 100 large, original images containing rip currents. More training samples are needed. Therefore, they are trained on 4000 positive and 8000 negative images generated by OpenCV. A record of every rip current location is created as input for the framework. The models are evaluated on 10% of the training data.



**Figure 26.** How networks are compared.



### 3.5.2 Implementation

The convolutional neural network setup is seen in Figure 26. There are different model configurations provided by Google [35]. One example is “SDD mobilenet V2 coco”. This model trades training speed for accuracy. Another example is “faster rcnn inception resnet V2”. Inception focuses on precision and accuracy in place of training speed as it has larger layers. Training both models covers a wider range of CNN capabilities. Models are built from scratch since the prebuilt models are not trained on rip current images [13]. The configurations for the models are not changed from default. They run for 5 weeks and are evaluated on the same set of orthorectified images as the other models.

The CNNs learn specific filters applied to different areas of an image. An example of a filter is horizontal line detection. Thousands of samples teach CNNs important filters for a specific object. Applying a certain filter activates a neuron path, which leads to a region classified as an object of interest.

## 3.6 Viola-Jones

This section discusses the package containing an implementation of Viola-Jones. Then, Viola-Jones is explained by showing how it classifies an image and showing the manner of choosing its features.

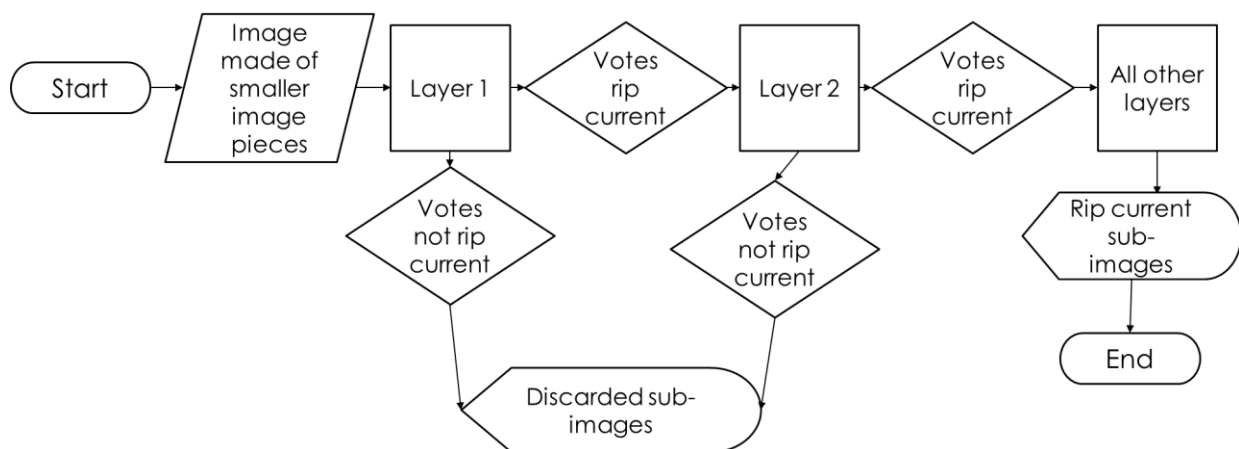
### 3.6.1 OpenCV Library

OpenCV [14] is a popular python library that contains numerous methods to conduct Machine Learning experiments. One built-in function trains a cascading object detector similar to Viola-Jones [24]. OpenCV also has a method for automatically creating positive samples. The 461 rip current samples yield 4000 training samples. Imposing a 24 by 24 sample onto a larger,

negative image of the shoreline creates new samples. The samples are slightly warped to mimic different angles. A bash script produces negative samples in a similar manner. The default warp parameter of 0.1 is applied 10 times to every sample. The 800 surf negative samples are imposed onto a larger image of a shoreline containing no rip currents. The procedure spawns 8000 negatives. Some models train on created negatives of the shoreline while other models train on random, negative images. Different models train with either 4000 positive samples or 461 positive samples. The command for building a cascading object detector runs with default settings. Every model is evaluated on 12 orthorectified images containing about 53 total rip currents. This benchmark consists of 10% of the smaller dataset.

### 3.6.2 Implementation

The Viola-Jones classifier is composed of layers. Figure 27 indicates how the classifier makes a prediction. Small images, part of the original large image, are taken as input. Viola-Jones scans the small images at every scale and location. The layers vote on whether a small image is a rip current. The small image passes to the next layer if a layer decides it is a rip current. The small image passes to the next layer if a layer decides it is a rip current. The small image is discarded if a layer predicts it is not a rip current. Small images



**Figure 27.** The Viola-Jones classifier.

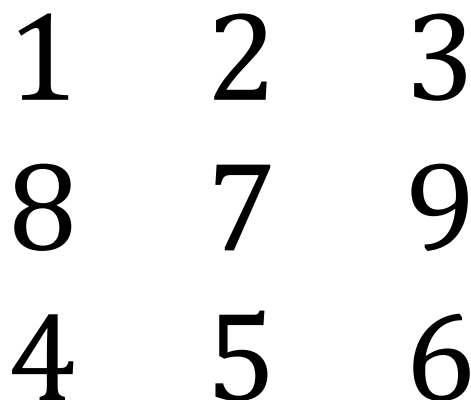
passing every layer in the cascade are considered rip currents. A layer is a series of weak classifiers. A combination of weak classifiers results in a strong classifier, known as a layer of the cascade. Ada-boost picks the best Haar features for the dataset. The chosen features become weak classifiers.

## 3.7 Meta Learner

This section explains the meta-learner. First, the meta-learner's features are introduced. The features develop from the original Haar features in the Viola-Jones algorithm [10]. Ada-boost chooses the optimal features for the dataset. The chosen features are appended to a feature vector, which trains the meta-classifier. The meta-classifier combines many classifiers together to provide a final classification.

### 3.7.1 Additional Haar Features

A more robust feature set is needed to capture the complexities of rip currents because the original set of Haar features are optimized for faces. The average image of all rip current samples assists in creating new features. Consequentially, 19 new features spawn from a 3 by 3



1	2	3
8	7	9
4	5	6

**Figure 28.** The matrix for designing the new features. Every number represents an area of space extracted from the integral image for a haar feature formula.

Feature	Formula	Detection Rate	False Positive Rate
X	$[1 + 3 + 4 + 6] - [7]$	0.996	0.5
T	$[1 + 2 + 3 + 5] - [7]$	0.996	0.5
Inverted T	$[6 + 4 + 5 + 7] - [2]$	0.995	0.6
Three Columns	$[3 + 9 + 6 + 1 + 8 + 4 + 7 + 5] - [2]$	0.996	0.5
Cross	$[2 + 7 + 9 + 8] - [5]$	0.995	0.8
I	$[1 + 3 + 4 + 6] - [2 + 7 + 5]$	1	1
T (b)	$[1 + 3] - [2 + 7 + 5]$	1	1
Short T	$[1 + 3] - [2 + 7]$	1	1
Inverted T (b)	$[6 + 4] - [5 + 2 + 7]$	1	1
V	$[1 + 3] - [7]$	1	1
^	$[4 + 6] - [7]$	1	1
[	$[5 + 7 + 2] - [6 + 3]$	1	1
[ (b)	$[5 + 2 + 6 + 3] - [7]$	1	1
]	$[5 + 2 + 1 + 4] - [7]$	1	1
>	$[1 + 4] - [7]$	1	1
<	$[3 + 6] - [7]$	1	1
] (b)	$[3 + 6 + 9] - [2 + 5]$	1	1
[ (c)	$[8 + 4 + 1] - [2 + 5]$	1	1
L	$[6] - [2 + 7 + 5]$	1	1

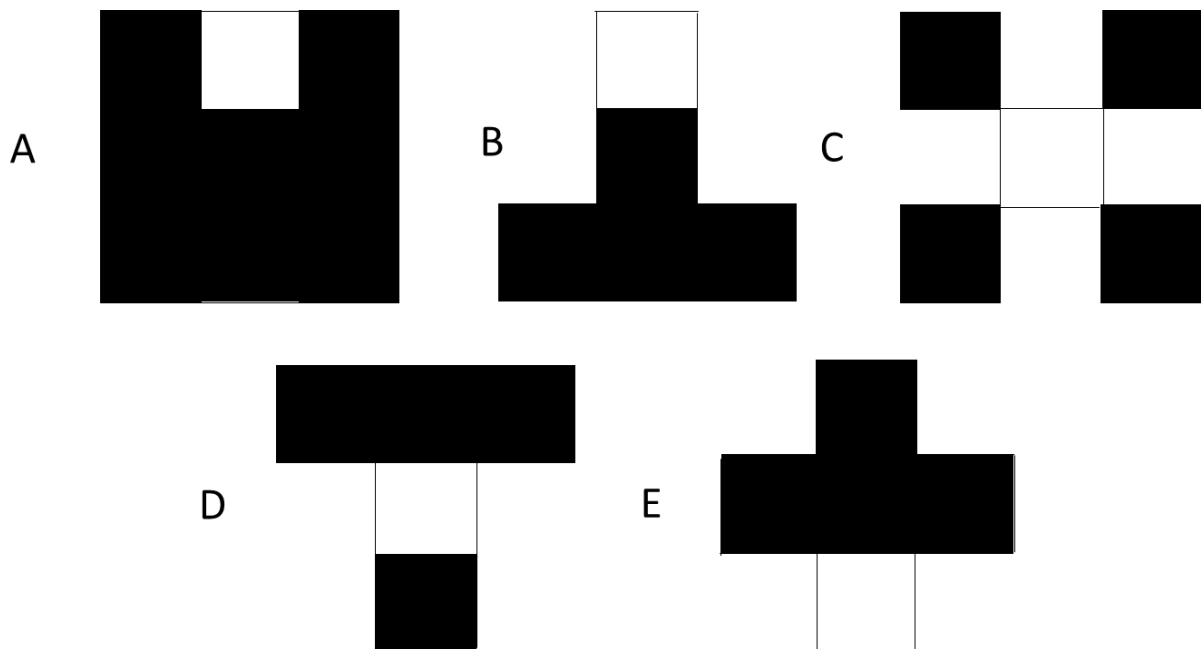
**Figure 29.** The results of using each feature to train a 10 layer cascade of weak classifiers. The Formula column refers to the matrix described in Figure 24.

matrix shown in Figure 28. Numbers 1-9 represent different regions of a rectangle. A new Haar feature is created by changing the difference formula between regions. The algorithms creating and applying Haar features are provided in [25]. The false positive rates and detection rates compare the features. The test results are seen in Figure 29. X, T, Inverted T, Three Columns, and Cross have the top performance of the features. The names reflect how the features appear. An evaluation on 19 features in the 3 by 3 matrix finds the mentioned 5. The code for evaluating the features is in appendix C. The features are tasked with building 10 layers of a cascade 5 times over. The top features have a false positive rate lower than 1. Every other feature has a false positive rate of 1, which means the features fail the test since they cannot separate samples accurately. The formula column in Figure 29 depicts rectangles operated on in the 3 by 3 matrix.

A 3 by 3 matrix has more rectangles than a 2 or 3 by  $x < 3$  matrix, which creates more robust features. They are larger than their previous counterpart, thus fewer of them in a 24 by 24 window. Fewer features produce a smaller feature space, which takes less time for Ada-Boost to search. Examples of the top performing features are seen in Figure 30. Ada-Boost builds 9 layers of a Viola-Jones cascade with 30% new features and 70% original features. The features chosen by Ada-Boost make a new feature vector, which trains the meta-classifier.

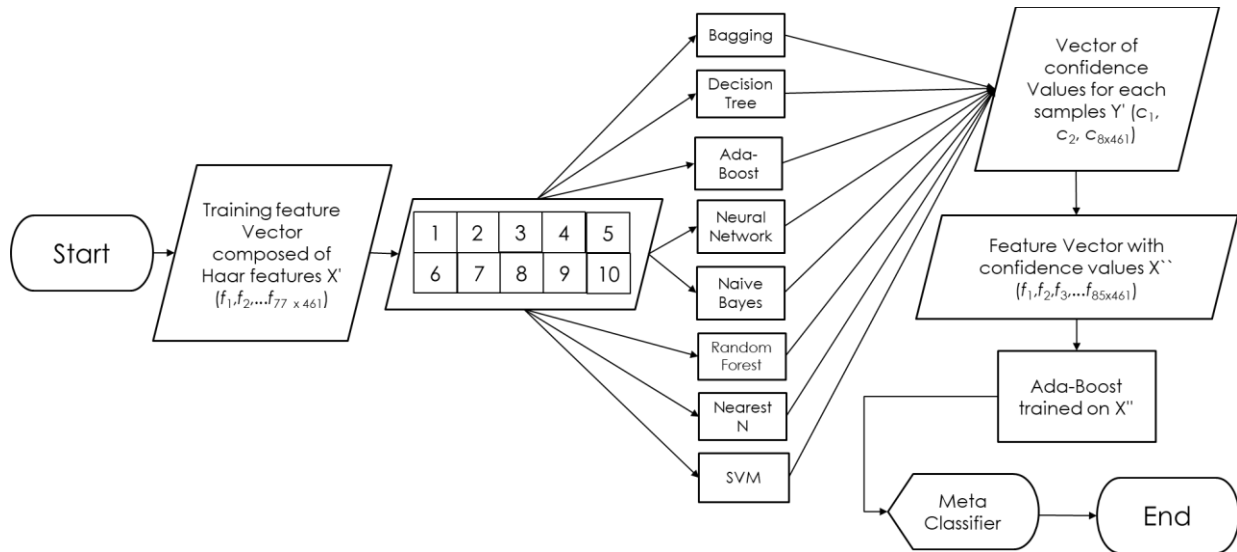
### 3.7.2 Implementation

A collection of classifiers from the Scikit-learn package train on a feature vector comprised of 77 Haar features found by Ada-boost. The meta-learner is seen in Figure 31. A technique known as stacking [15] improves results. A model yields the class probability of every test sample after training. 10-fold cross validation assists in recording a probability for every sample in the dataset.



**Figure 30.** The Newly Added Features: (A) “Three columns”, (B) “Inverted T”, (C) “X”, (D) “T” and (E) “Cross”.

These models include: SVM, neural network, decision tree, bagging, Naïve Bays, Ada-Boost, nearest neighbors, and random forest. They represent most of the Scikit-learn models. Figure 32 identifies the parameters for creating each model. Training every model in the package is a brute force approach. This finds the most appropriate model for the dataset. The models are built with default parameters, except for the SVM. A meta-classifier trains on a feature vector made from 77 Haar features and 8 model confidence values. The meta-classifier trains as every type of classifier and is evaluated as each model. The top performer is chosen for the final model. The method for classification is in Figure 33. In Figure 33, the meta-classifier and basic classifiers train on the data generated from the method in Figure 31. The OpenCV Viola-Jones classifier runs on the rip current benchmark, which acquires the rip current output windows. A feature vector for every output window is then created. First, 77 Haar features applied to the output windows create a feature vector for the basic classifiers. Next, the basic classifiers make predictions on the feature vectors, which generate the confidence values for every model.

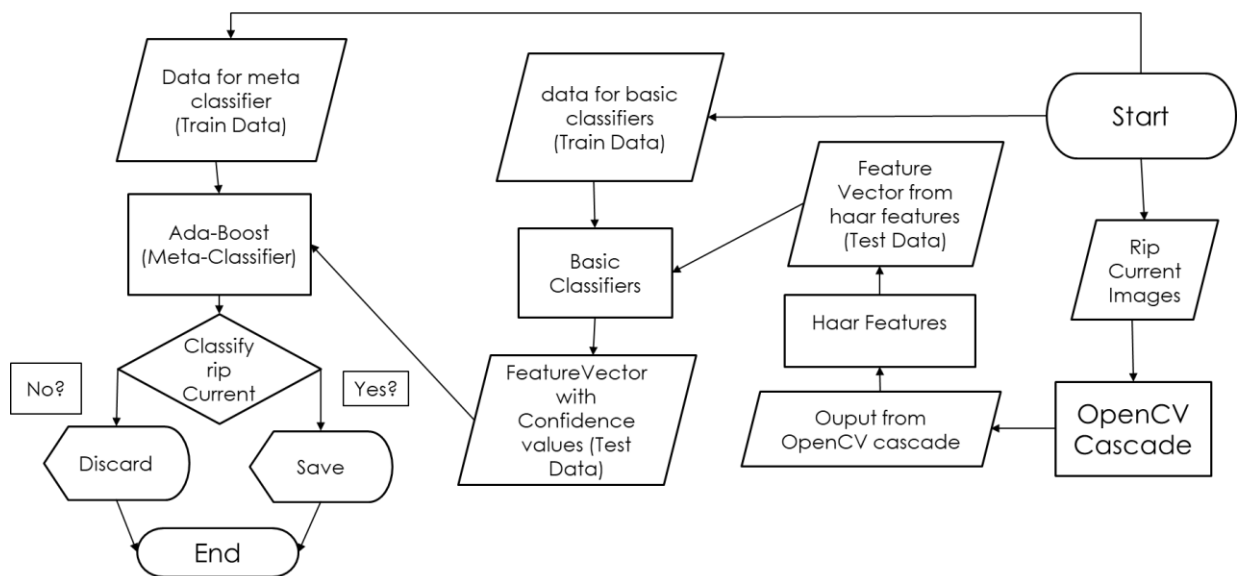


**Figure 31.** A Flow chart representing the meta classifier. The basic classifiers train on a Haar feature vector. The confidence values from every fold are appended to the original feature vector creating  $X''$ .

Model									
Bagging	N_estimators = 10	max_samples = 1.0	Max_features = 1.0	Bootstrap = true	Bootstrap_features = false				
Decision Tree	criterion = gini	splitter = best	no max depth	min_samples_split = 2					
Ada-Boost	base = DecisionTreeClassifier	n_estimators = 50	learning_rate = 1	real-boosting					
Neural Network	hidden_layer_size = 100	activation = relu	adam solver	alpha = 0.0001	batch size = 200	epoch = 200	beta_1 = 0.9	Beta_2 = 0.999	Epsilon = 1e-8
Random Forest	N_estimators = 10	criterion = gini	Max_features = n_features	Max_depth = none	min_samples_split = 2	min_samples_leaf = 1	No weights		
Nearest Neighbor	n_neighbors = 5	Uniform Weights	Auto algorithm	Leaf_size = 30	p = 2				
SVM	kernel = RBF	C = 4.0	gamma = 0.00390625						

**Figure 32.** The parameters for each basic classifier.

Third, the confidence values of the 8 basic classifiers are appended to the original feature vector. Finally, the resulting feature vector is handed, as input, to the meta-learner for final classification. The results of the meta-classifier are compared against the traditional, stand-alone Viola-Jones.



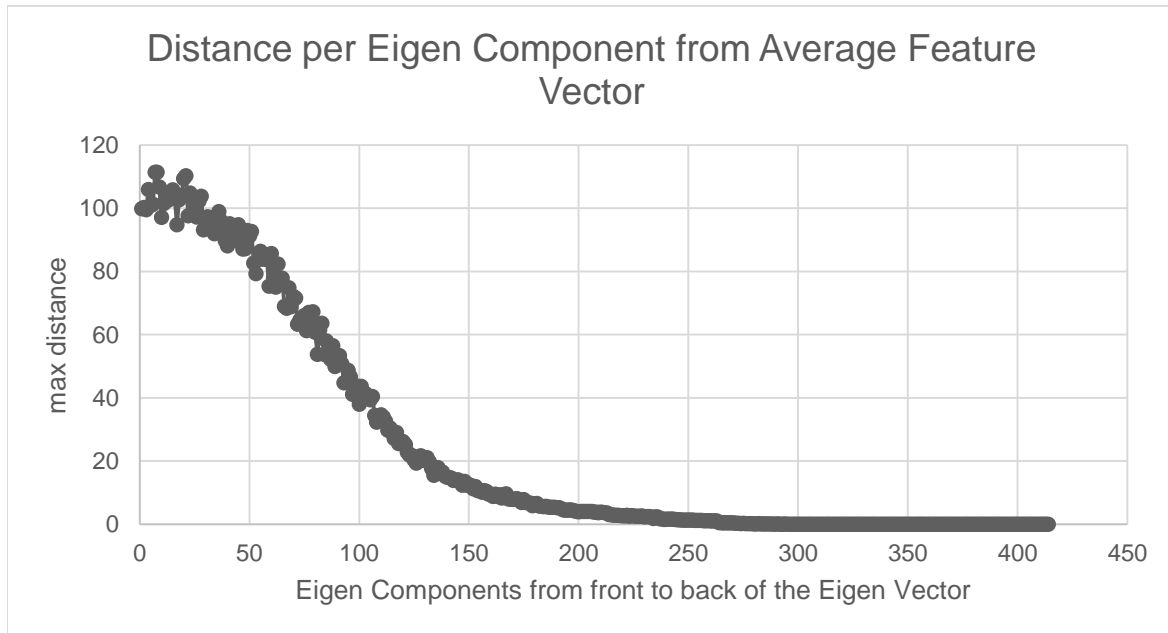
**Figure 33.** A Flow-chart describing the meta-learner source code in Python.

## 4. Results

This section contains the results for max distance from the average, SVMs, convolutional neural networks, Viola-Jones, and the meta-learner. The meta-learner section is broken into the new Haar features, stacking, and the actual meta-learner.

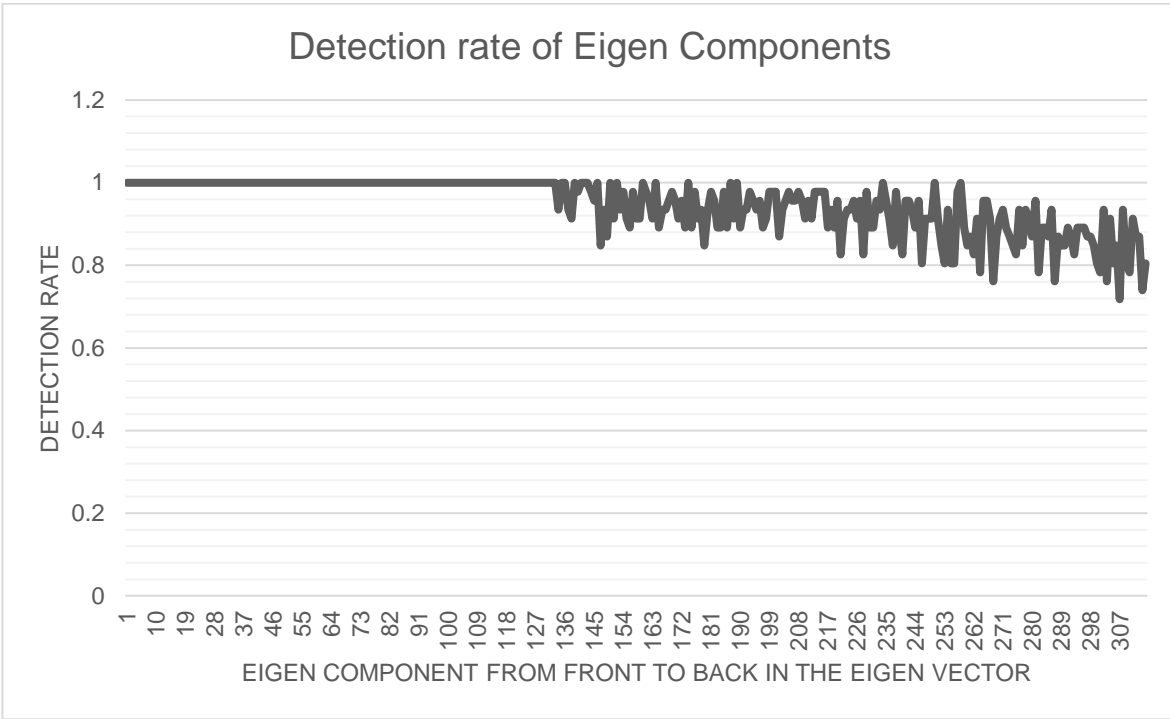
### 4.1 Max Distance from the Average

The results for max distance from the average on the rip current dataset are in Figures 34, 35, and 36. Components in the Eigen vector are on x axis. The graphs mimic the structure of the Eigen vector. The x axis starts from the component in the front of the vector and works backward. In Figure 34, the y axis is the max distance from the average. The distance from the average decreases as rip current samples project toward components at the end of the Eigen vector. Distance decreases because components retain less variance from front to back in the vector. Figure 35 reveals the detection rate for the rip current dataset. The detection rate



**Figure 34.** The max distance from the average feature vector generated for every component projected toward.



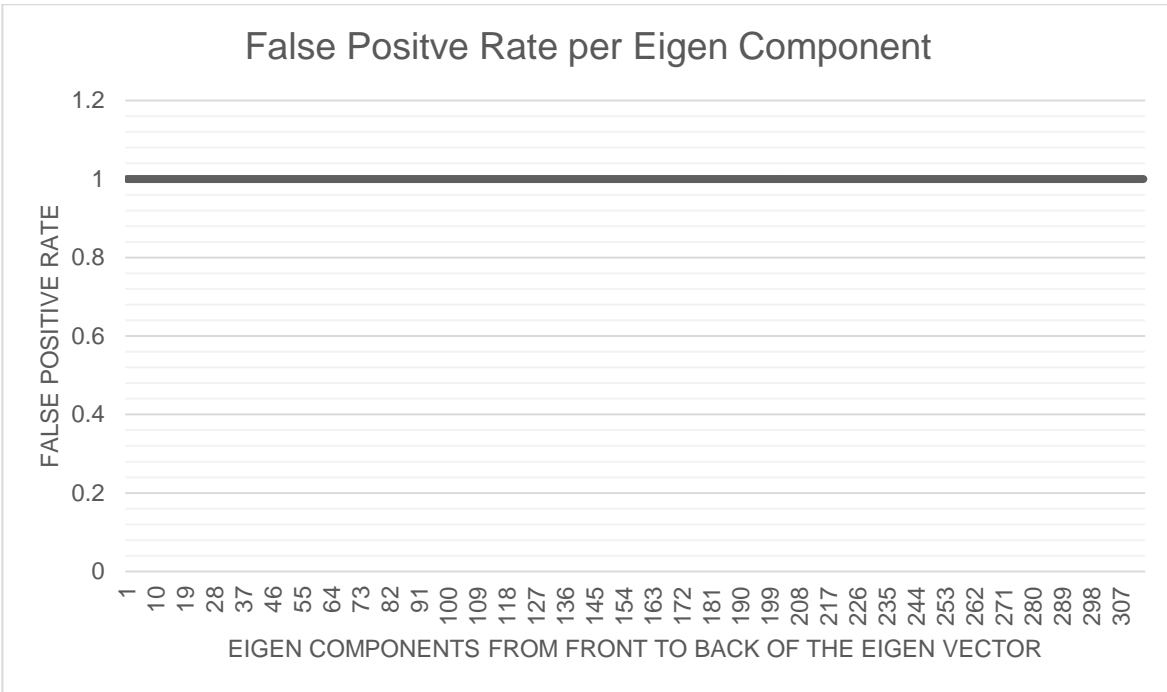


**Figure 35.** The detection rate for the distance yielded from every component.

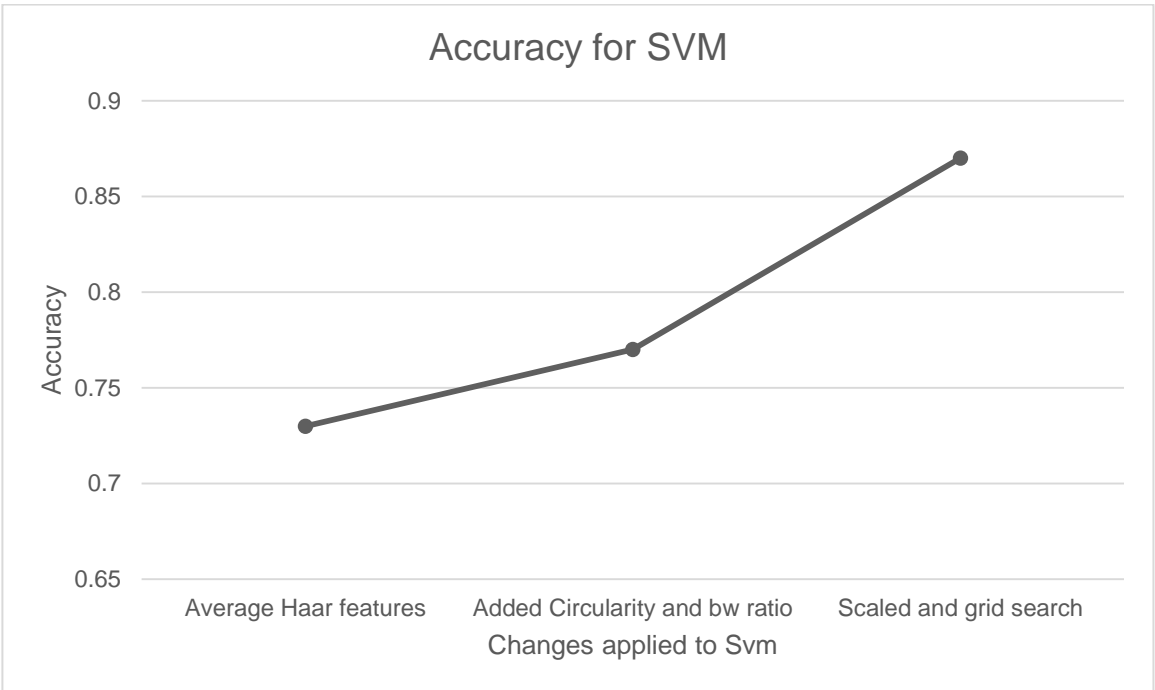
gradually decreases as components near the end of the Eigen vector generate the maximum distance from the average. The detection rate decreases as more outlying rip currents from the dataset are lost. The detection rate reaches a minimum rate of 71% for Eigen component number 307. The maximum distance from the average is greater at the front of the Eigen vector. In consequence, this yields a larger range of rip current values. The large range holds the detection rate at 1.0 because of its inclusive nature. Figure 36 identifies the false positive rate of every component. The component projection has no effect on the false positive rate. Every negative sample is misclassified as a rip current. The range of values is not small enough to divide rip currents from other images, which creates a 100% false positive rate.

## 4.2 Support Vector Machines

The results for the Scikit-learn SVM with a RBF kernel and 10-fold cross-validation is seen in Figure 37. Black-white ratio and circularity produces a 4% higher accuracy.



**Figure 36.** The false positive rate for the distance yielded from every component.

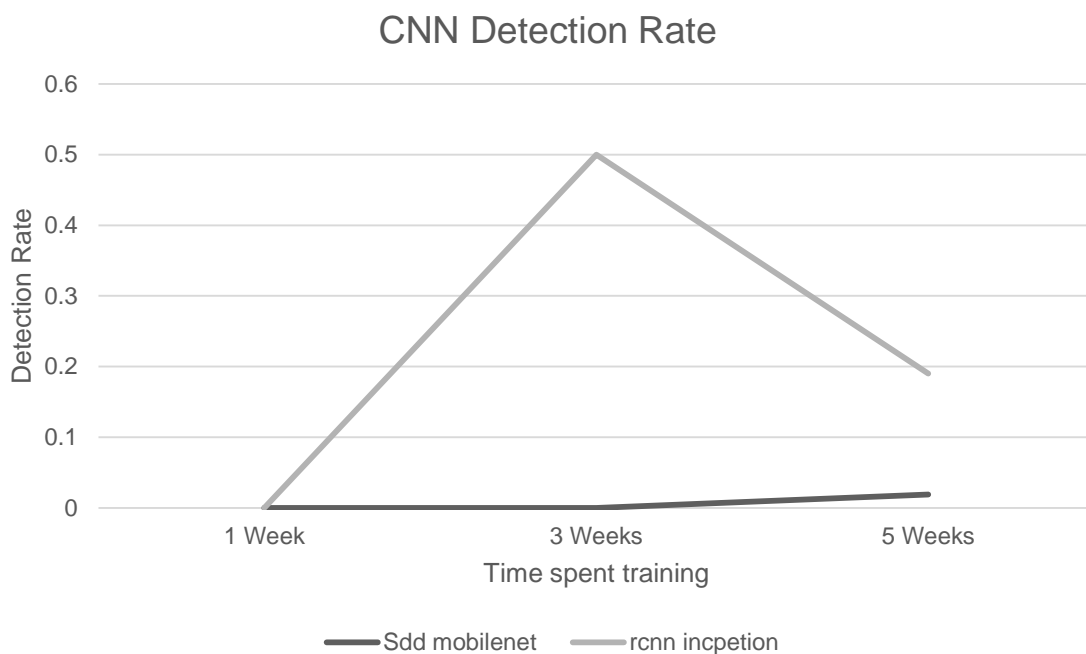


**Figure 37.** The results for Scikit-learn SVM. The average Haar feature vector does not contain the circularity or black-white ratio feature. The last version applies a Robust scaler and parameters found with grid search.

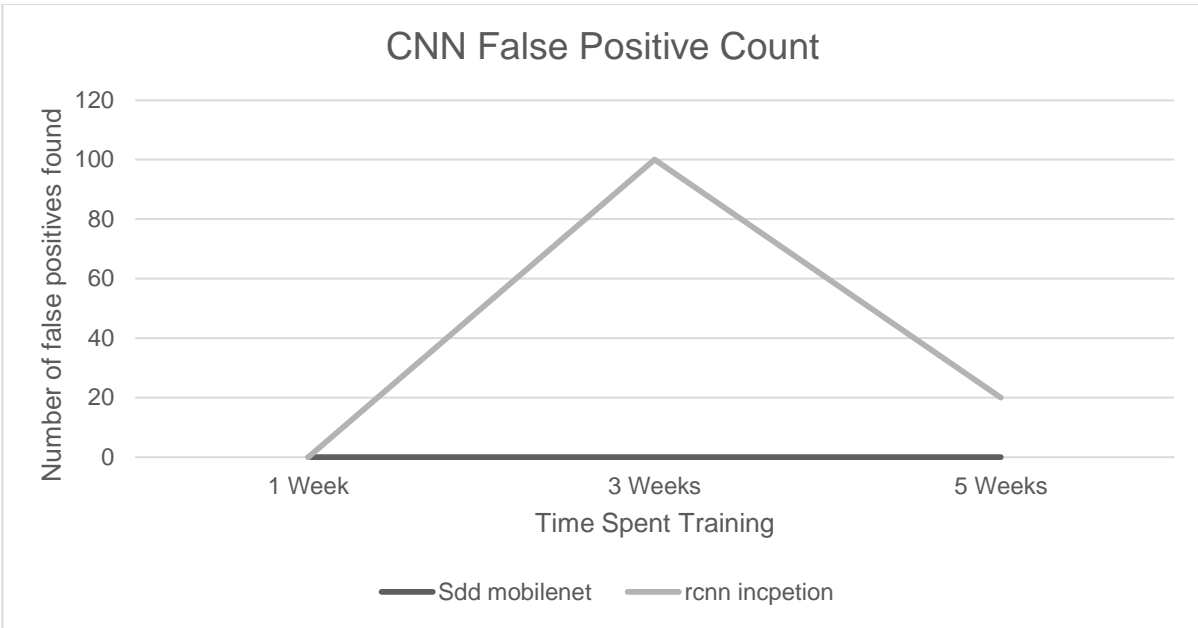
The results also describe the accuracy when applying a robust scaler and optimal parameters to the SVM. The numerical results for grid search are  $C = 4.0$  and  $\gamma = 0.00390625$ . Scaling and optimal parameters increase the accuracy by 10%.

### 4.3 Convolutional Neural Networks

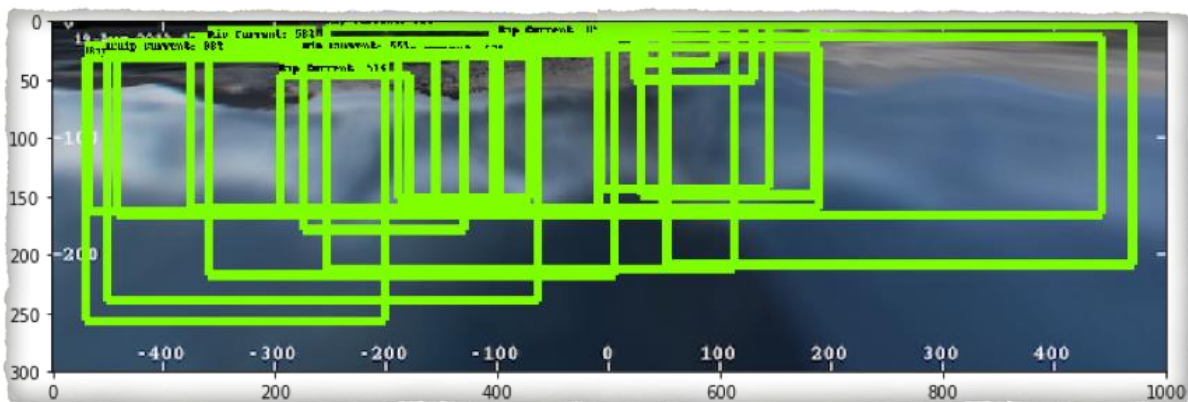
Figure 38 identifies the detection rate of the 2 convolutional neural networks. The models make no predictions after week 1 of training. In week 3, rcnn Inception has the highest detection rate for the networks with 50%. In week 5, the detection rate for the Inception model drops from 50% to 19%. The detection rate for Mobilenet increases slightly after 5 weeks of training to 1.9%. Figure 39 reveals the number of false positives found by the models. The false positive count decreases from 100 to 20 for the Inception model, after week 5 of training. The false positive count for the sdd Mobilenet model remains unchanged at 0 for all 5 weeks of



**Figure 38.** The detection rate results after 1-5 weeks of training for CNNs.



**Figure 39.** The false positive count after 1-5 weeks of training the CNNs.



**Figure 40.** An image classified by the inception model after 3 weeks of training. The green boxes show detected rip currents in the image.

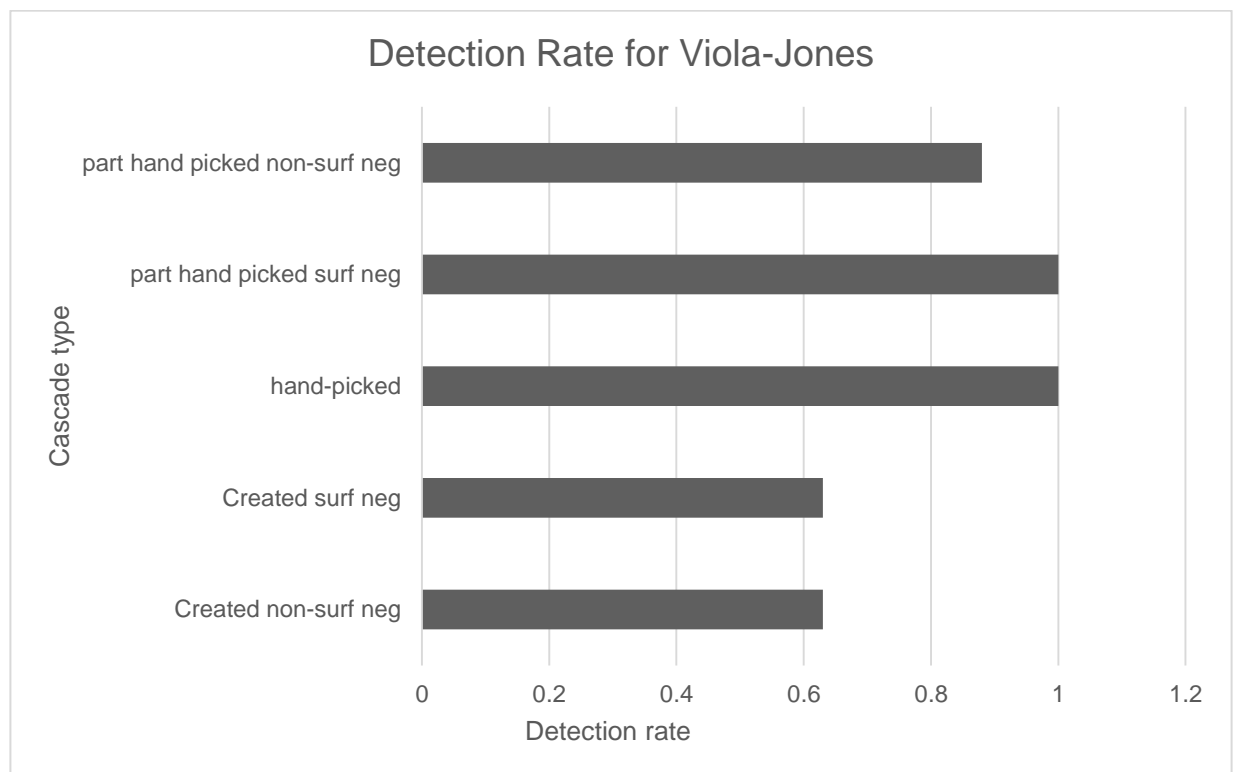
training. Figure 40 is an example image classified by the inception model. It contains a large number of imprecise detections around the surf zone.

## 4.4 Viola-Jones

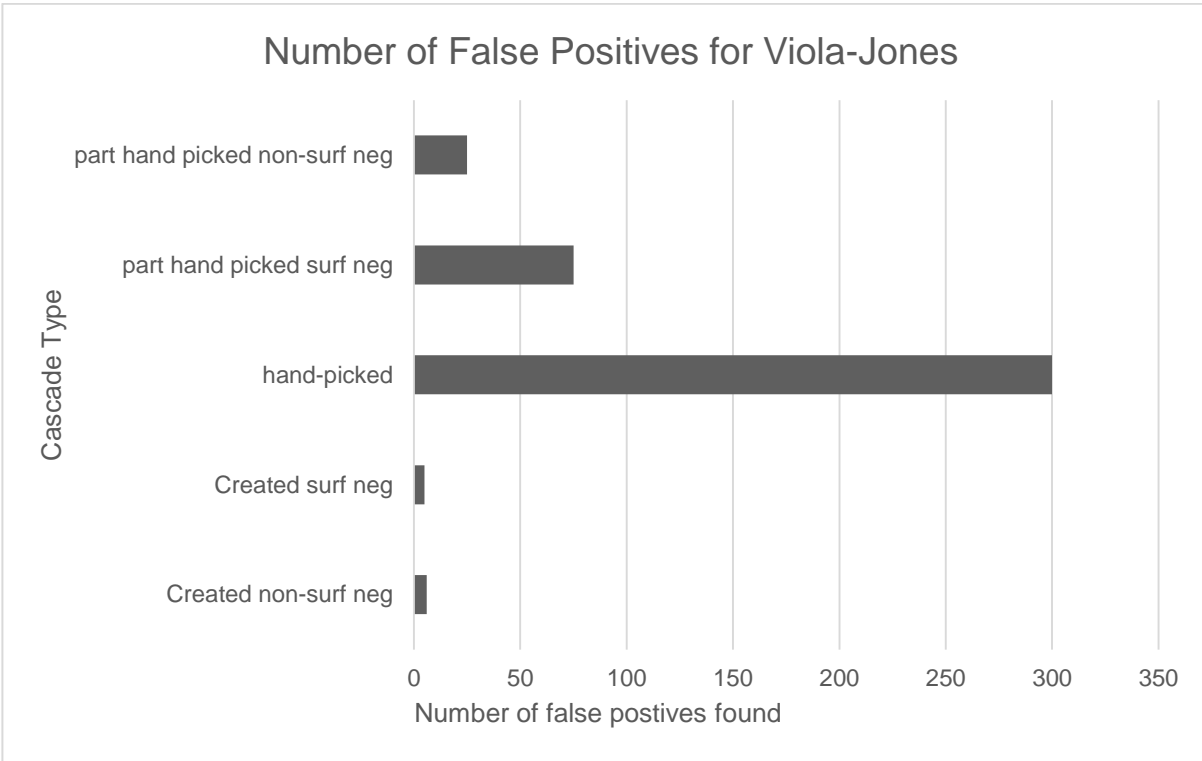
The OpenCV library creates many models for rip current classification from the Viola-Jones algorithm. In Figure 41, the y axis is the type of cascade built. These types correspond to positive and negative samples in every training set. “Created Non-Surf Neg”, “Created Surf

Neg”, “Part Hand-Picked Surf Neg”, and “Part Hand-Picked Non-Surf Neg” train with negative samples created by OpenCV. “Created Surf Neg” and “Created Non-Surf Neg” also train on created, positive samples, which are warped and superimposed onto a large negative image by OpenCV. “Hand-Picked” trains on manually extracted, negative samples from the surf zone.

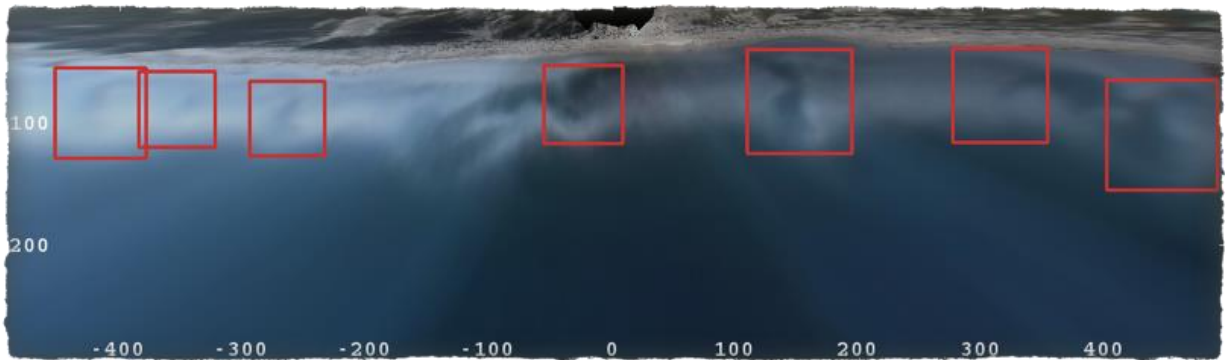
Figure 41 depicts the detection rate on 12 images containing 53 total rip currents. Figure 42 identifies the number of false positives in the same images. Hand-Picked finds 300 false positives the benchmark images. This is the maximum number of false positives for any cascade. Hand-Picked also trains on the smallest dataset. It has a detection rate of 1.0, therefore it finds all rip currents in the 12 images. Created Surf Neg and Created Non-Surf Neg train on the largest dataset, have a detection rate of 0.63, or 63%, and find the lowest number of false positives. Part Hand-Picked Surf Neg has a 1.0 detection rate, but locates 75 false positives in the benchmark



**Figure 41.** The detection rate of every cascade on the rip current test images. The cascade names correspond to the types of samples the cascade trains on.

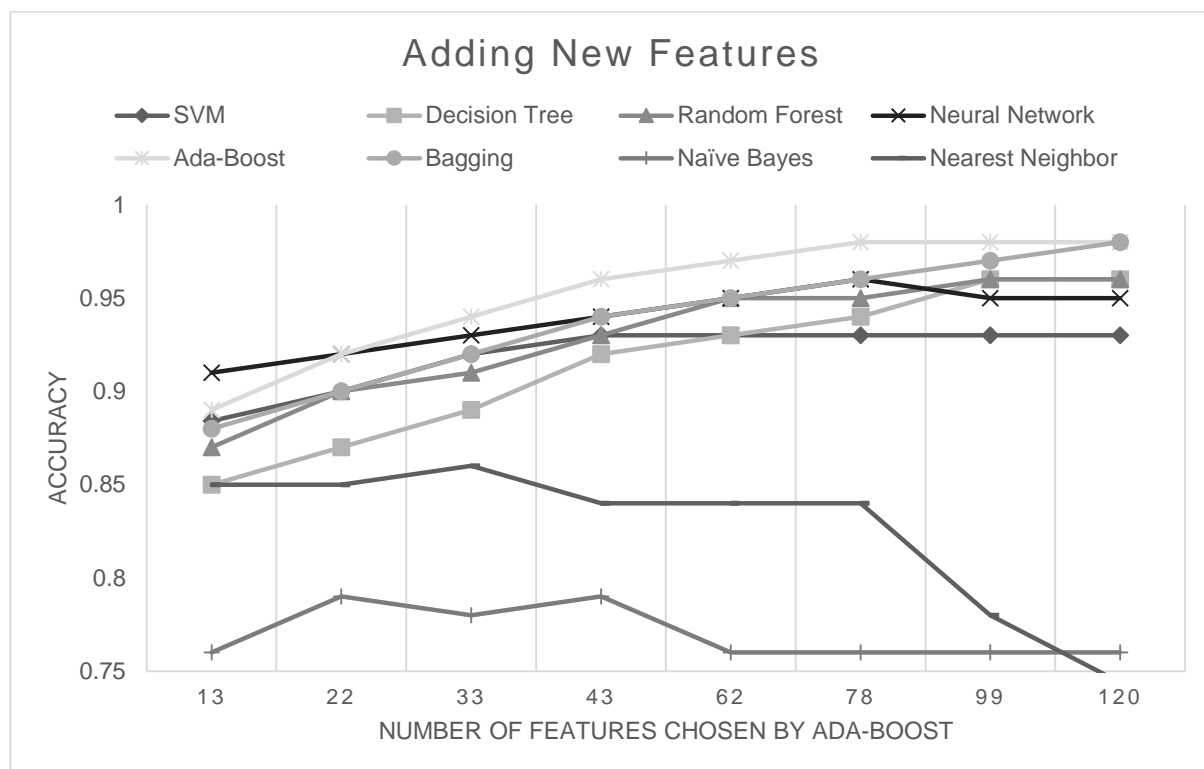


**Figure 42.** The number of false postives found by each cascade.



**Figure 43.** An image of rip currents classified by an OpenCV object detector. The red boxes indicate rip currents in the image.

images. Part Hand-Picked Non-Surf Neg has an 88% detection rate and finds 15 false positives in 12 images. Figure 43 is an example image classified by Part Hand-Picked Non-Surf Neg. It finds every rip current in the image except for the box in the middle. This box is a false positive and should be moved slightly to the left.



**Figure 44.** The accuracy after continuously adding new features to the feature vector.

## 4.6 Meta Learner

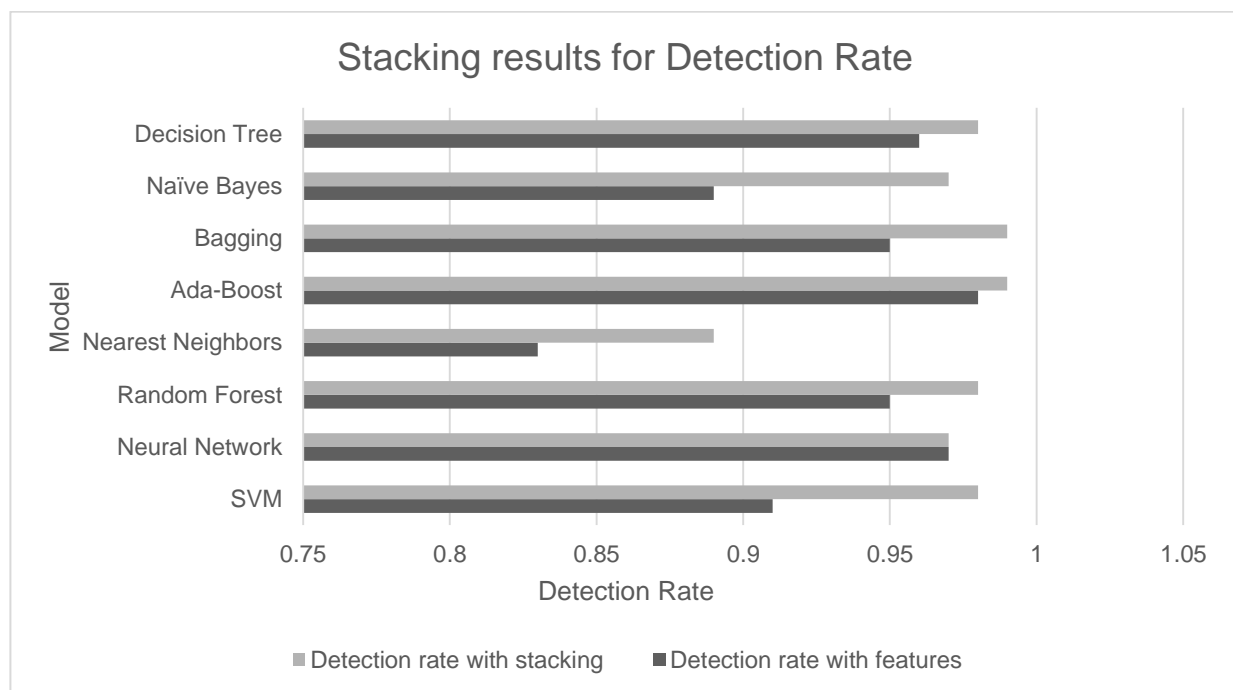
This section describes the results for the new features that are used to build the meta-classifier and the results of the meta-learner. The first subsection reveals accuracies for every basic model with new features. Then, the detection rate and false positive rate after stacking is provided. Finally, the detection rate and false positive count for the meta-classifier is described. This includes a comparison of the meta-classifier and Viola-Jones at different layers and a final comparison of every object detection model with notable performance in the project.

### 4.6.1 Results for New Features

In Figure 44, the accuracies for every model are presented as more features are appended to the training vector. The accuracies combine the true positive and true negative rates. Accuracy describes how many images are correctly classified during 10-fold cross validation. Accuracies

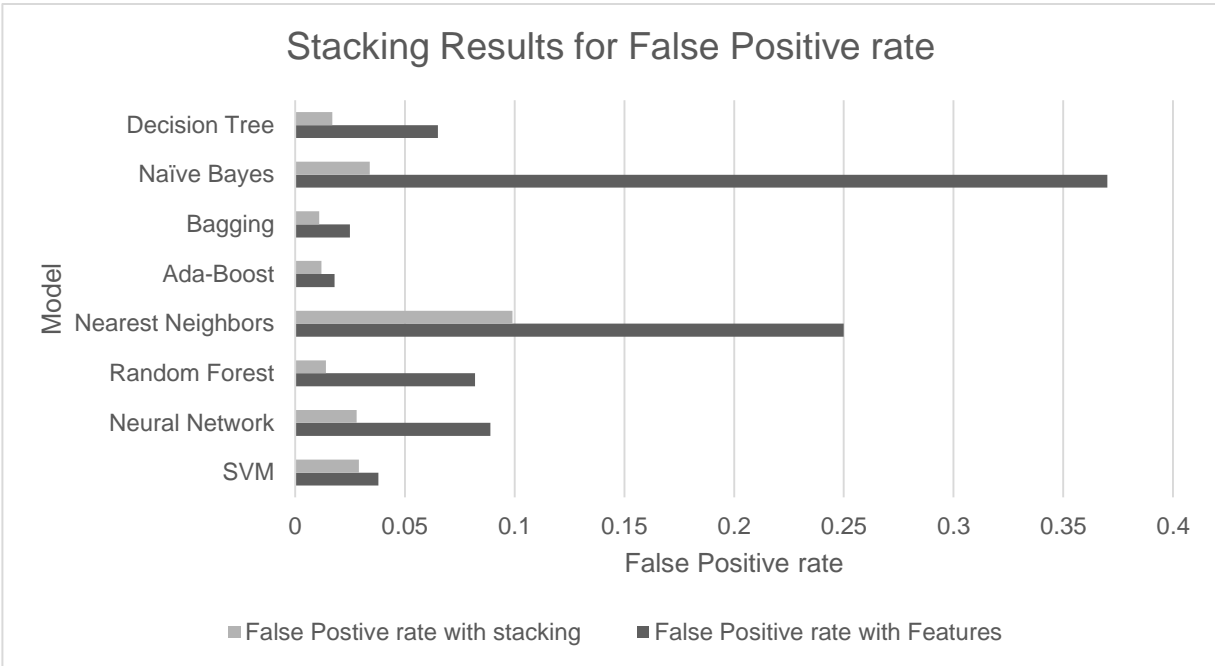
are at their lowest when the vector is composed of 13 Haar features. The accuracies of all models except Naïve Bayes and nearest neighbor increase as more Haar features are added for training. Ada-Boost achieves the highest accuracy of 98% at 77 Haar features. Bagging reaches 98% accuracy at 120 Haar features. Random forest and the decision tree levels off at 96% accuracy with 99 Haar features. Neural networks achieve an accuracy of 96% with 78 Haar features added. SVMs peak at 93% accuracy and level off at 43 Haar features introduced. The accuracy for nearest neighbors continues to decrease as more Haar features are appended. KNN reaches 86% accuracy after 33 Haar features are added. The accuracy Naïve Bayes increases slightly, decreases until 62 Haar features are inside the feature vector, and levels off at 76% accuracy.

## 4.6.2 Results for Stacking



**Figure 45.** The detection rate of every model after stacking the confidence of every model.



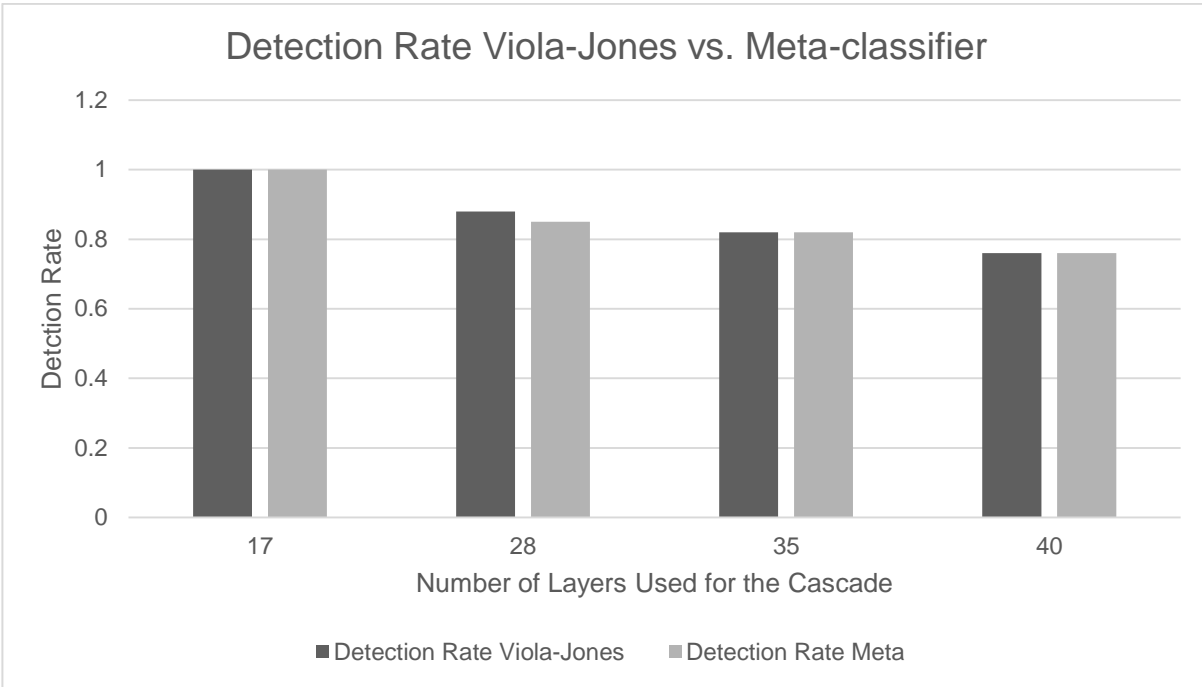


**Figure 46.** The false positive rate after stacking the confidence of every model.

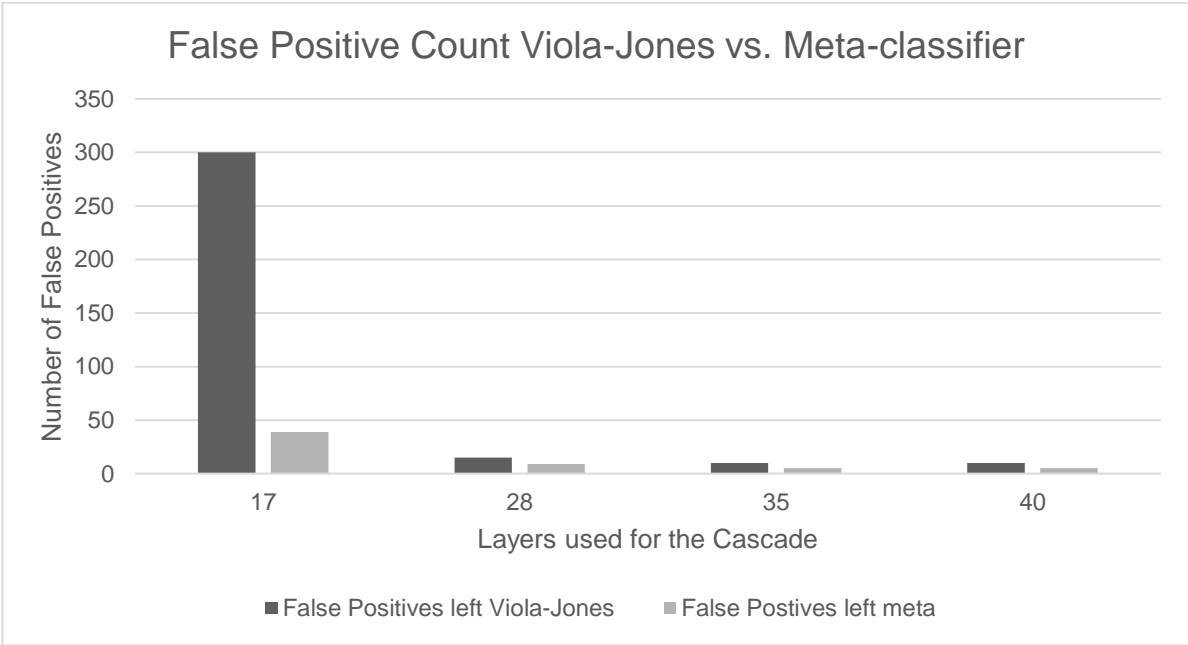
In Figure 45, the detection rate after adding the confidence of each learner to the feature vector is seen. Models' detection rates increase after they are stacked. Detection increases by 3.75% on average after stacking. Figure 46 contains the false positive rates after stacking models. The false positive rate decreases by an average of 10% after stacking confidence values. Bagging and Ada-boost have the highest detection rate at 99% and have the lowest false positive rate at 1%. Nearest neighbors has the lowest detection rate and has the highest false positive rate at 89% and 25%, respectively.

### 4.6.3 Results for Meta-Classifier

In Figure 47, the detection rate of Viola-Jones is compared against the meta-classifier at layer 17, 28, 35, and 40. Figure 48 has a comparison of the false positive count for those same layers. The detection rate decreases from 88% to 85% for layer 28 after adding the meta-classifier, but remains the same at every other layer.

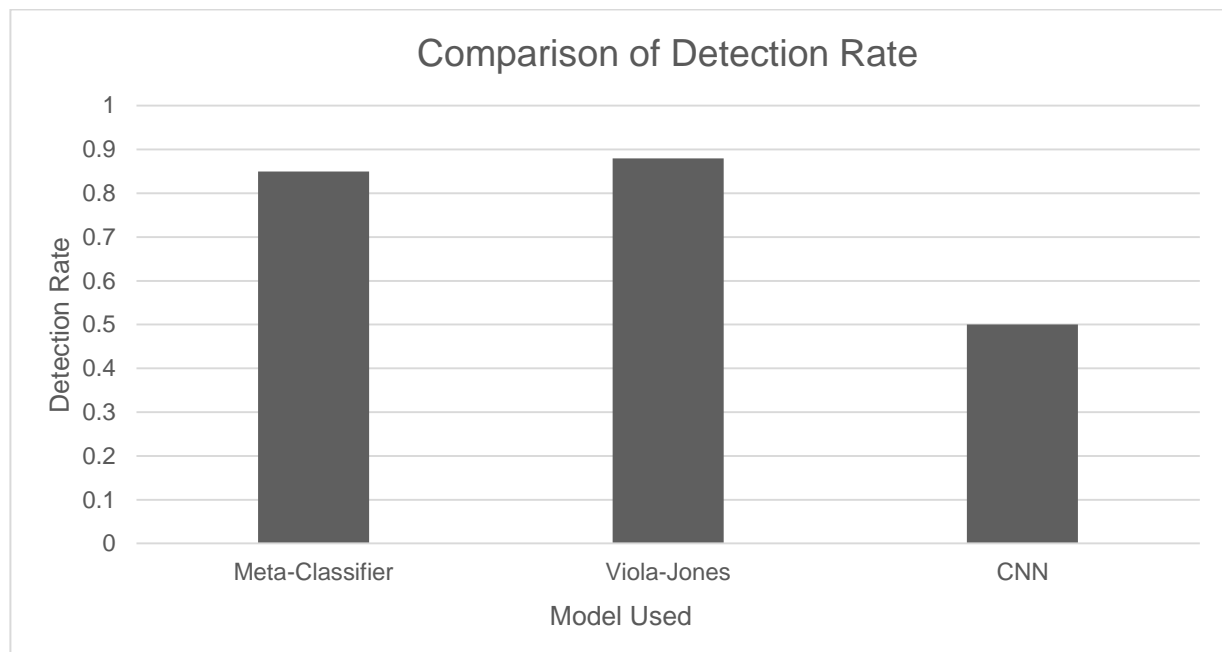


**Figure 47.** A comparison of the Viola-Jones classifier and the meta classifier detection rates at a varying number of layers.



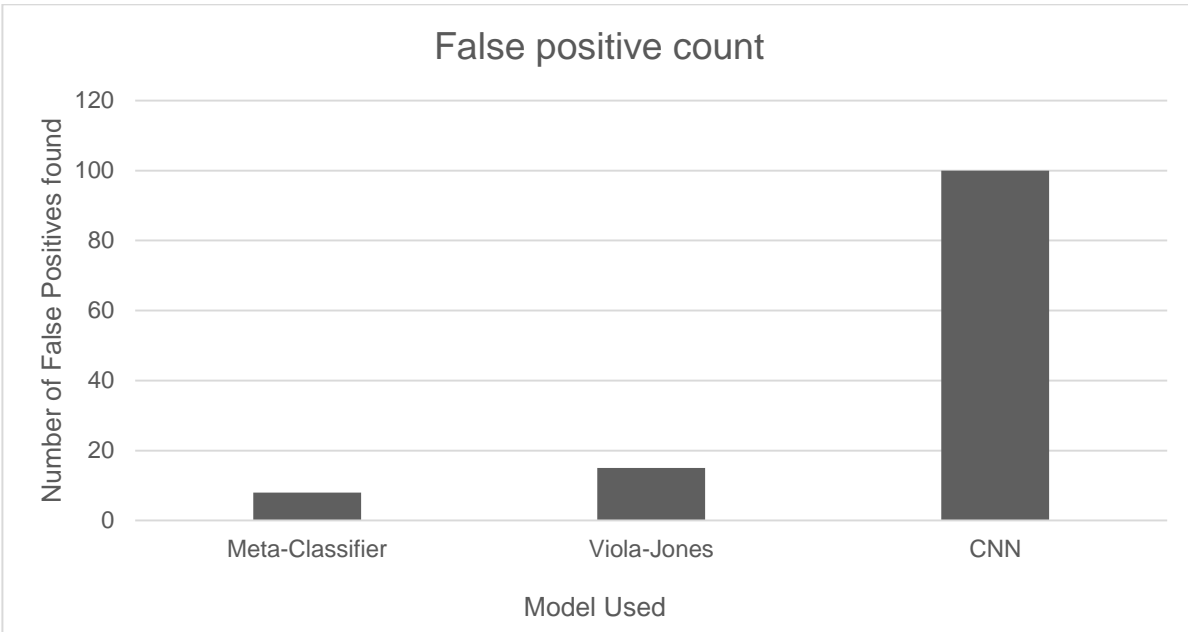
**Figure 48.** A comparison of the false positives found at a varying number of layers for the Viola-Jones classifier and the meta classifier.

The meta-classifier reduces the number of false positives by 47% at layer 28. The meta-classifier achieves a lower false positive rate at this layer than a 40 layer Viola-Jones cascade.

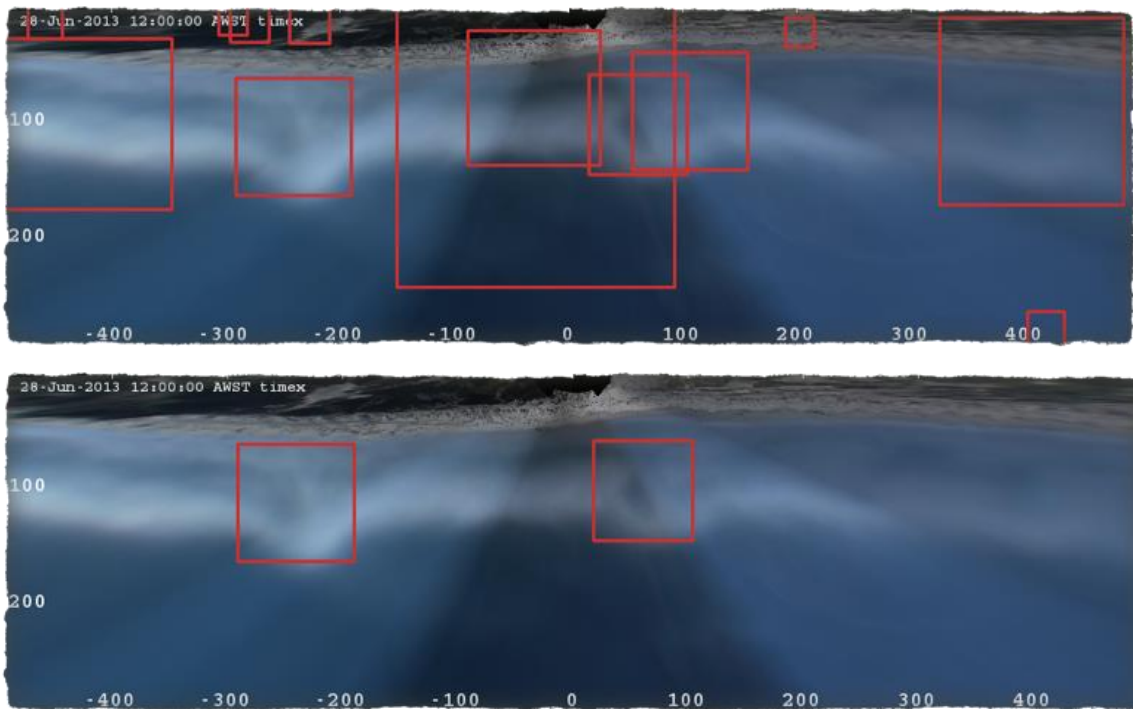


**Figure 49.** A comparison of detection rate for Viola-Jones, CNN, and the meta-classifier.

The false positive rate for a cascade is equal to the product of every layer's false positive rate. Layers have a false positive rate of 0.7, consequently a false positive rate of  $4.6e-5$  for a 28 layer cascade. Adding the meta-learner with a false positive rate of 0.01 to the end of the cascade produces a false positive rate of  $4.6e-7$ . Figure 49 depicts a detection rate comparison of CNN, Viola-Jones, and the meta-classifier at peak performance. The Viola-Jones model has the highest detection rate of 88%. The meta-learner has the next highest detection rate of 85%. In Figure 50, the false positive counts of the classifiers are compared. CNN has the highest false positive count at 100 followed by Viola-Jones at 15. The meta-classifier has the lowest false positive count. Figure 51 compares the same image classified by Viola-Jones and the meta-classifier. There are more windows labeled as rip currents by Viola-Jones. The meta-classifier finds fewer false positives while still locating the 2 rip currents in the image.



**Figure 50.** A comparison of the false positive count for Viola-Jones, CNN, and the meta-classifier.



**Figure 51.** The top image classified by Viola-Jones versus the bottom image classified by the meta classifier.

## 5. Discussion

This section reveals underlying details, which concern results for max distance from the average, SVMs, convolutional neural networks, Viola-Jones, and the meta-learner. The meta-learner section is broken into a discussion of the new Haar features and classifier stacking.

### 5.1 Max Distance from the Average

The max distance from the average decreases as components toward the end of the Eigen vector are projected upon. This is expected as components near the front of the Eigen vector maximize variance while components near the rear minimize variance. The differences between each rip current are retained as more variance is retained. Only commonalities between rip current samples remain as variance decreases. The threshold is less inclusive as the maximum difference from the average decreases. The exclusion decreases the detection rate.

Different components do not change the number of correctly classified negative samples. The features generated from the components cannot capture the difference between a rip current and something not a rip current. Too much information is lost when projecting toward only one dimension, which produces entangled values of negative samples and positive samples. Max distance from the average has the weakest false positive rate, but PCA may be useful for future work with rip current recognition or dimension reduction.

### 5.2 Support Vector Machines

Training support vector machines on circularity and black-white ratio increases performance by 4%. The 4% increase in performance supports circularity and black-white ratio as viable features. The improvement from circularity is due to the semi-circle nature of a rip current sample. The rip current samples also have similar lighting. The lighting and orientation

of the samples supports the black-white ratio improvement. Scaling and grid searching improve performance by an additional 10%. This further backs the need for this optimization.

SVMs prove to be a less effective meta-classifier. Ada-Boost chooses the features for training, which gives SVMs weaker classification performances by comparison. SVMs are not acceptable as a method of classification for rip currents when training on average Haar features since their accuracies max out at 88%. This rate sets a ceiling for accuracy of the detector. The target rate of a classifier during evaluation is 99% because it is matching a layer of Viola-Jones. A 12% loss of samples is too large when given such a small sample size.

## **5.3 Convolutional Neural Networks**

The CNNs need a longer amount of time to train than the Viola-Jones detectors. The training is longer because models are built from scratch. The variables of the convolutional layer are not global. The lack of fully connected nodes may reduce performance from 3 weeks of training to 5 weeks of training as important variables might not be saved. A decrease in performance may also be due to overfitting. There are not an adequate number of rip current samples for creating a classifier with convolutional neural nets. CNNs require a larger number of samples to avoid overfitting [4-6]. The detection boxes located on test images are large and contain many rip currents. The models cannot differentiate between individual rip currents, which may be attributed to the stitching process of convolutional neural nets. This process adjoins neighboring boxes together if they are within a small enough range.

The CNN models perform worse than Viola-Jones due to the number of samples needed for the learners. Ada-Boost performs adequately on fewer samples [20]. A neural network needs many thousands of samples. The CNN face detectors attain a decent model after training with close to 100000 samples.

The model configurations for the CNNs are not optimized for the rip current dataset. Optimization for the configurations or a higher quality annotated dataset of rip current images can deeply improve performance.

## 5.4 Viola-Jones

24 by 24, negative images are highly ineffective for building a Viola-Jones object detector. The Hand-Picked cascade has the largest false positive rate of any cascade type, which bolsters OpenCV's need to down-sample larger negative images. The cascades that train on created, positive samples have a lower detection rate on the test images. This performance is the result of transformations to the data by OpenCV. Warping the data is no substitute for a meaningful sample when there are many more warped samples than there are natural samples. Tilting the original samples causes them to lose meaningful data around the edges of the sample.

The cascades training on few samples have a perfect detection rate, but their false positive rate is also high. Detection rate decreases as of the number of samples increase. A larger number of positive samples yields a harder false positive rate test, which requires more layers to pass. Adding more layers to the cascade reduces the false positive rate, but the detection rate as well.

A balance of the detection and false positive rates is important when choosing a cascade. The Part Hand-Picked Any Neg cascade correctly classifies 88% of rip currents in all 12 images and only finds 15 false positives. Hence, it is the top performer for Viola-Jones. The larger cascades are not as accurate, which have a detection rate of 60%. This is not an efficient tradeoff for attaining less than 5 false positives. Viola-Jones has the best predictions on the dataset for the pre-existing tools. A dataset of 24 by 24 samples is allowed by OpenCV for training. This data

yields the highest quality results. The Viola-Jones detector also locates where rip currents are in the image, which can be counted if needed.

Ada-Boost is decent for generalizing a detector since it only considers a hard margin for each weak classifier it adds [20]. Ada-boost does not maximize accuracy by adjusting the margin for outliers, which leads to less of an over fit.

## **5.5 Meta Learner**

The following subsections explain the impact of adding the new Haar features to the feature vector and stacking models to create the meta-learner.

### **5.5.1 New Features**

The new features improve the accuracy of almost every model in Scikit-learn. Learners such as neural networks and SVM continue to increase as they train on more features since they adequately handle larger dimensional feature spaces. KNN and Naive-Bayes suffer when adding more features because they cannot handle larger dimensional feature spaces.

Ada-Boost, random forest, decision tree, and bagging performances continue to increase as more features are appended to the feature vector. These features are chosen in accordance with splitting the data. A classifier splitting the data has an increased performance if it receives a feature that splits the data effectively, which the Haar features are optimized to do. A mixture of the new and old Haar features improve results. This is supported by Ada-Boost reaching a peak accuracy of 98%.



## 5.5.2 Classifier stacking

The performance of classifiers increase when training on the confidence values of every learner. Therefore, confidence values help compensate for what a classifier is lacking in classification.

The meta-classifier has a better performance than a strong classifier (layer) of a Viola-Jones cascade. A layer has a false positive rate of 50% to 70%. The meta-classifier has a false positive rate of 1%. Thus, the false positive rate of the final classifier is improved. The detection rates of a cascade layer and the meta-classifier, at best, remain the same. The lack of change is due to a reclassification of output windows. The positive windows are lost forever if the Viola-Jones classifier misses them. The meta-classifier does not classify windows that are not output from Viola-Jones since it piggybacks on Viola-Jones. Overall, the meta-learner has the most promising results.

## 6. Conclusions

Max distance from the average is not an adequate classifier for rip currents and should be avoided when only projecting toward 1 dimension. PCA has not been concluded as tool for recognition or feature dimension reduction on rip currents. SVMs also have a peak accuracy inadequate for detecting rip currents when training on average Haar features.

The annotated dataset created by OpenCV is ineffective for training CNNs. This reinforces that superimposing a smaller image onto a larger image does not produce meaningful samples for a small dataset of rip currents. They may be useful if a larger dataset is created. A larger, manually annotated dataset is needed to fully explore classifying rip currents with CNNs.

Viola-Jones is a decent method for creating a rip current classifier. Warped, positive samples decrease the performance of the cascade as it over fits to the samples. Large, negative images and smaller, positive samples are needed for producing a significant cascade.

Classifier stacking is effective for increasing the performance of a model. The new Haar features are beneficial for classifying the rip current dataset. The classifiers have the most improvement from the Haar features when they are built upon splitting the dataset with a threshold value.

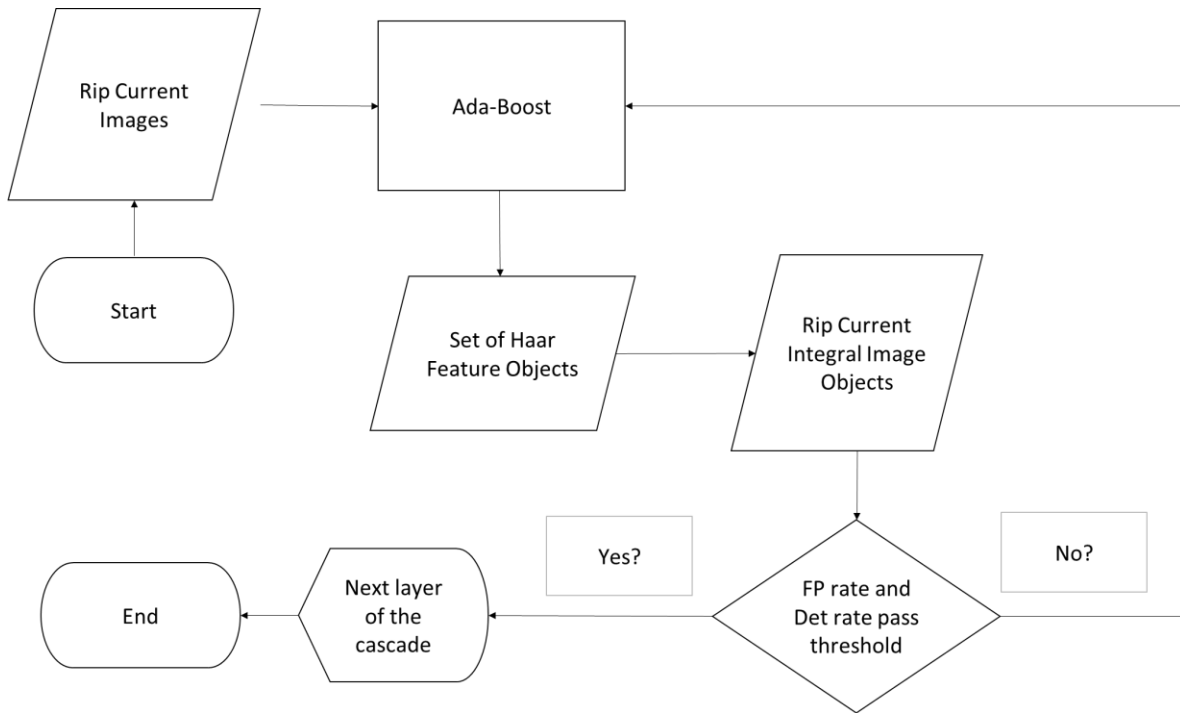
The most effective dataset is the manually picked 461 rip currents along with random, large, and negative images. This dataset in combination with the meta-classifier has the best performance when assuming a lower false positive rate is desired. The Ada-boost and bagging algorithms produce the most accurate meta-classifiers. The meta-classifier has the potential to assist in automated rip current identification for climate studies.

## Bibliography

1. University, O.S. *Coastal Imaging Lab*. 2009 2009/01/05; Available from: [cil-www.coas.oregonstate.edu](http://cil-www.coas.oregonstate.edu).
2. Heikkilä, J. and O. Silvén, *A Four-step Camera Calibration Procedure with Implicit Image Correction*. 1997.
3. Holland, K.T., et al., *Practical Use of Video Imagery in Nearshore Oceanographic Field Studies*. IEEE JOURNAL OF OCEANIC ENGINEERING, 1997. **22**(1).
4. Howard, A.G., et al., *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017.
5. Krizhevsky, A., I. Sutskever, and G.E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*. 2012: p. 9.
6. Szegedy, C., et al., *Going deeper with convolutions*. 2015.
7. Bishop, C., *Pattern Recognition and Machine Learning*. Information Science and Statistics. 2009: Springer-Verlag New York.
8. Szeliski, R., *Computer Vision: Algorithms and Applications*. 2010: Springer Science & Business Media. 812.
9. Stewart, R. and M. Andriluka, *End-to-end people detection in crowded scenes*. CoRR, 2015. **abs/1506.04878**.
10. Viola, P. and M.J. Jones, *Robust Real-Time Face Detection*. International Journal of Computer Vision, 2004. **57**(2): p. 137-154.
11. Jolliffe, I.T., *Principle Component Analysis*. 2013: Springer Science & Business Media.
12. Russell, S. and P. Norvig, *Artificial Intelligence: A Modern Approach*. 2009: Prentice Hall Press. 1152.
13. *TensorFlow*. 2017; Develop]. Available from: [https://www.tensorflow.org/tutorials/image\\_retraining](https://www.tensorflow.org/tutorials/image_retraining).
14. Bradski, G., *The OpenCV Library*. Dr. Dobb's Journal of Software Tools, 2000.
15. Wolpert, D.H., *Stacked Generalization*.
16. Maurya, A. *Support Vector Machines: How does going to higher dimension help data get linearly separable which was non linearly separable in actual dimension?* 2013; Available from: <https://www.quora.com/Support-Vector-Machines-How-does-going-to-higher-dimension-help-data-get-linearly-separable-which-was-non-linearly-separable-in-actual-dimension>.
17. Fabian Pedregosa, G.V., Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, and V.D. Ron Weiss, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, Edouard Duchesnay, *Scikit-learn: Machine Learning in Python*. Journal of Machine Learning Research. **12**: p. 2825-2830.
18. Bhalla, D. *Support Vector Machine Simplified using R*. Available from: <https://www.listendata.com/2017/01/support-vector-machine-in-r-tutorial.html>.
19. Benediktsson, J., J. Kittler, and F. Roli. *Multiple Classifier Systems*. 2009.
20. Schapire, R.E., *Explaining AdaBoost*. 2012.
21. Loh, W.-Y., *Classification and Regression trees*. 2011: p. 10.
22. Hall, P., B.U. Park, and R.J. Samworth, *Choice of Neighbor Order in Nearest-Neighbor Classification*. The Annals of Statistics, 2008. **36**(5): p. 2135-2152.
23. Zhang, H., *The Optimality of Naive Bayes*. 2004: p. 6.

24. Leinhardt, R. and H. Maydt. *An Extended Set of Haar-Like Features for Rapid Object Detection*. 2002.
25. Jensen, O.H., *Implementing the Viola-Jones Face Detection Algorithm*. 2008.
26. Pinet, C.U.P.R., *Invitation to Oceanography*. 2009: Jones & Bartlett Publishers.
27. Castelle, B., et al., *Rip current types, circulation and hazard*. Earth-Science Reviews, 2016. **163**(Supplement C): p. 1-21.
28. Gallop, S.L., et al., *Storm-driven changes in rip channel patterns on an embayed beach*. Geomorphology, 2011. **127**: p. 179-188.
29. Association, U.S.L. *Rip Currents*. 2017; Available from: [www.usla.org/?page=RIPCURRENTS](http://www.usla.org/?page=RIPCURRENTS).
30. Leatherman, S.P. *Rip Currents 101*. 2011; Available from: [www.ripcurrents.com/ripcurrents101.html](http://www.ripcurrents.com/ripcurrents101.html).
31. Kim, K.-H., S. Shin, and A.Y.W. Widayati, *Mitigation Measures for Beach Erosion and Rip Current*. Journal of Coastal Research, 2013: p. 290-295.
32. Holman, R.A., et al., *Rip Spacing and persistence on an embayed beach*. Journal of Geophysical Research, 2006. **111**.
33. Sebastian Pitman, S.L.G., Ivan D. Haigh, Sasan Mahmoodi, Gerd Masselink, Roshanka Ranasinghe, *Synthetic Imagery for the Automated Detection of Rip Currents*. Journal of Coastal Research, 2016. **75**: p. 912-916.
34. Pape, L. and B.G. Ruessink, *Neural-network predictability experiments for nearshore sandbar migration*. Continental Shelf Research, 2011. **31**: p. 1033-1042.
35. *TensorFlow Model Configurations*. 2017; Available from: [https://github.com/tensorflow/models/tree/master/research/object\\_detection/samples/configs](https://github.com/tensorflow/models/tree/master/research/object_detection/samples/configs).

## Appendix



**Figure 52.** A flow-chart describing the source code for Java.

### A. Source Code for Ada-Boost in Java

```
package violajones;
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Map.Entry;
import java.util.TreeSet;
```

```
import haar.Cross;
import haar.Four;
import haar.HaarFeature;
import haar.HaarFeature.TYPE;
import haar.IntegralImage;
import haar.InvertedT;
```

```

import haar.T;
import haar.ThreeColumns;
import haar.X;
import haar.ThreeHorizontal;
import haar.ThreeVertical;
import haar.TwoHorizontal;
import haar.TwoVertical;
/**
 * Represents the AdaBoost learning algorithm used in Viola-Jones.
 * @author Corey Maryan
 *
 */
public class AdaBoost {
    public static HaarFeature[] featureSet;
    public static HashMap<HaarFeature,Boolean> used = new HashMap<>();
    private static HashMap<IntegralImage,Boolean> rips;
    public static int totalMax = Integer.MIN_VALUE;
    public static int totalMin = Integer.MAX_VALUE;

    //returns a strong classifier a.k.a. a collection of weak classifiers
    public static HaarFeature[] learn(ArrayList<IntegralImage>
trainPos,ArrayList<IntegralImage> trainNeg,int layer){

        //if there are no images to train on then failure
        if (trainNeg.size()== 0) {
            return null;
        }

        double negWeight = 1.0 / (2.0*trainNeg.size());
        double posWeight = 1.0 / (2.0*trainPos.size());

        //initialize the weights for all images
        for (IntegralImage img : trainPos)
            img.weight = posWeight;
        for (IntegralImage img : trainNeg)
            img.weight = negWeight;

        //if we are building the first layer create the haar feature space
        if (layer == 1) {
            ArrayList<HaarFeature> features = new ArrayList<>();

            //for each feature type, width, height, and positive in a 24x24 window create a feature
            for (TYPE f : TYPE.values()) {
                for (int x = 0;x < 24;x++) {

```

```

for (int y = 0; y < 24; y++) {
    /* if (f == TYPE.CROSS) {
        for (int width = 1; width < 25; width++) {
            for (int height = 1; height < 25; height++) {
                if (x + 3*height-1 < 24 && y + 3*width-1 < 24) {
                    features.add(new Cross(x,y,width,height));
                }
            }
        }
    }
} else if (f == TYPE.INVERTED_T) {
    for (int width = 1; width < 25; width++) {
        for (int height = 1; height < 25; height++) {
            if (x + 3*height-1 < 24 && y + 3*width-1 < 24) {
                features.add(new InvertedT(x,y,width,height));
            }
        }
    }
} else*/ if (f == TYPE.T) {
    for (int width = 1; width < 25; width++) {
        for (int height = 1; height < 25; height++) {
            if (x + 3*height-1 < 24 && y + 3*width-1 < 24) {
                features.add(new T(x,y,width,height));
            }
        }
    }
} else if (f == TYPE.THREE_COLUMNS) {
    for (int width = 1; width < 25; width++) {
        for (int height = 1; height < 25; height++) {
            if (x + 3* height-1 < 24 && y + 3*width-1 < 24) {
                features.add(new ThreeColumns(x,y,width,height));
            }
        }
    }
} else if (f == TYPE.X) {
    for (int width = 1; width < 25; width++) {
        for (int height = 1; height < 25; height++) {
            if (x + 3* height-1 < 24 && y + width*3-1 < 24) {
                features.add(new X(x,y,width,height));
            }
        }
    }
}
}/* else if (f == TYPE.THREE_VERTICAL) {
    for (int width = 1; width < 25; width++) {
        for (int height = 1; height < 25; height++) {
            if (x + 3* height-1 < 24 && y + width-1 < 24) {
                features.add(new ThreeVertical(x,y,width,height));
            }
        }
    }
}

```

```

    }
    }
}
}*/else if (f == TYPE.THREE_HORIZONTAL) {
    for (int width = 1; width < 24; width++) {
        for (int height = 1; height < 24; height++) {
            if (x + height-1 < 24 && y + width*3-1 < 24) {
                features.add(new ThreeHorizontal(x,y,width,height));
            }
        }
    }
}
}*/else if (f == TYPE.TWO_VERTICAL) {
    for (int width = 1; width < 25; width++) {
        for (int height = 1; height < 25; height++) {
            if (x + 2* height-1 < 24 && y + width-1 < 24) {
                features.add(new TwoVertical(x,y,width,height));
            }
        }
    }
}
}*/else if (f == TYPE.TWO_HORIZONTAL) {
    for (int width = 1; width < 25; width++) {
        for (int height = 1; height < 25; height++) {
            if (x + height-1 < 24 && y + width*2-1 < 24) {
                features.add(new TwoHorizontal(x,y,width,height));
            }
        }
    }
}
}*/else if (f == TYPE.FOUR) {
    for (int width = 1; width < 25; width++) {
        for (int height = 1; height < 25; height++) {
            if (x + 2* height-1 < 24 && y + width*2-1 < 24) {
                features.add(new Four(x,y,width,height));
            }
        }
    }
}
}*/
}
}

featureSet = new HaarFeature[features.size()];

for (int i = 0; i < features.size(); i++) {
    featureSet[i] = features.get(i);
}

```



```

    used = new HashMap<>();
    for (HaarFeature feature : features) used.put(feature,false);
}

rips = new HashMap<>();
HaarFeature[] classifiers = new HaarFeature[1];

double totalWeight = 0.0;

for (IntegralImage im : trainPos) totalWeight += im.weight;
for (IntegralImage im : trainNeg) totalWeight += im.weight;

double norm = 1.0/totalWeight;

//normalize the weights for each image
for (IntegralImage im : trainPos) {
    im.weight = im.weight*norm;
    rips.put(im, true);
}

for (IntegralImage im : trainNeg) {
    im.weight = im.weight*norm;
    rips.put(im, false);
}

double bestErrorFound = Double.MAX_VALUE;

//for each feature in the space
for (HaarFeature feature : featureSet) {
    if (used.get(feature))
        continue;

    Map<IntegralImage,Integer> scores = new HashMap<>();

    for (IntegralImage img : trainPos) {
        int score = feature.getFeatureValue(img.integral);
        scores.put(img, score);
    }

    for (IntegralImage img : trainNeg) {
        int score = feature.getFeatureValue(img.integral);
        scores.put(img, score);
    }

    //create a map that can assign a value to each image

```

```

TreeSet<Map.Entry<IntegralImage,Integer>> set = new
TreeSet<Map.Entry<IntegralImage,Integer>>(
    new Comparator<Map.Entry<IntegralImage,Integer>>() {
        @Override public int compare(Map.Entry<IntegralImage,Integer> e1,
Map.Entry<IntegralImage,Integer> e2) {
            int res = e1.getValue().compareTo(e2.getValue());
            return res != 0 ? res : 1;
        }
    }
);

```

```

set.addAll(scores.entrySet());

```

```

HashMap<Double,Integer> type1s = new HashMap<>();
HashMap<Double,Integer> type2s = new HashMap<>();
double tPos = 0.0;
double tNeg = 0.0;
double sPos = 0.0;
double sNeg = 0.0;

```

```

//total weight for pos and neg images
for (IntegralImage im : trainNeg)
    tNeg += im.weight;
for (IntegralImage im : trainPos)
    tPos += im.weight;

```

```

Iterator<Entry<IntegralImage, Integer>> it = set.iterator();
Entry<IntegralImage, Integer> previous = it.next();
Entry<IntegralImage, Integer> next = null;

```

```

while(it.hasNext()) {
    //iterate over the image list and compare weight totals
    if (previous.equals(set.first()) && next == null) {
        double type1 = sPos + (tNeg-sNeg);
        double type2 = sNeg + (tPos-sPos);

        if (type1 > type2) {
            type2s.put(type2,previous.getValue());
        } else {
            type1s.put(type1,previous.getValue());
        }

        next = it.next();
    } else if (it.hasNext()) {
        if (rips.get(previous.getKey())){
            sPos += previous.getKey().weight;

```

```

    } else {
        sNeg += previous.getKey().weight;
    }

    double type1 = sPos + (tNeg-sNeg);
    double type2 = sNeg + (tPos-sPos);

    if (type1 > type2) {
        type2s.put(type2,previous.getValue());
    } else {
        type1s.put(type1,previous.getValue());
    }

    previous = next;
    next = it.next();
    if (!it.hasNext()) {
        previous = next;

        if (rips.get(previous.getKey())){
            sPos += previous.getKey().weight;
        } else {
            sNeg += previous.getKey().weight;
        }

        double lastType1 = sPos + (tNeg-sNeg);
        double lastType2 = sNeg + (tPos-sPos);

        if (lastType1 > lastType2) {
            type2s.put(lastType2,previous.getValue());
        } else {
            type1s.put(lastType1,previous.getValue());
        }
    }
}

//find the minimum error based on the weights
if (type2s.isEmpty() && type1s.isEmpty()) {
    feature.weightedError = Double.MAX_VALUE;
} else if (type2s.isEmpty()){
    feature.weightedError = Collections.min(type1s.keySet());
    feature.setThreshold(type1s.get(feature.weightedError));
    feature.setPolarity(-1);
} else if (type1s.isEmpty()) {
    feature.weightedError = Collections.min(type2s.keySet());

```

```

        feature.setThreshold(type2s.get(feature.weightedError));
        feature.setPolarity(1);
    } else {
        double type2Min = Double.MAX_VALUE;
        double type1Min = Double.MAX_VALUE;
        type2Min = Collections.min(type2s.keySet());
        type1Min = Collections.min(type1s.keySet());
        feature.weightedError = type1Min;
        feature.setThreshold( type1s.get(type1Min));
        feature.setPolarity(-1);

        if (feature.weightedError > type2Min) {
            feature.weightedError = type2Min;
            feature.setThreshold(type2s.get(type2Min));
            feature.setPolarity(1);
        }
    }

    if (feature.weightedError > 0) {
        feature.weight = Math.log((1.0-feature.weightedError)/feature.weightedError);
    } else {
        feature.weightedError = 0;
        feature.weight = 4;
    }

    if (feature.weightedError < bestErrorFound) {
        bestErrorFound = feature.weightedError;
        classifiers[0] = feature;
    }
}

for (IntegralImage img : trainPos) {
    int score = classifiers[0].getFeatureValue(img.integral);

    if (score > totalMax) {
        totalMax = score;
    }
}

for (IntegralImage img : trainNeg) {
    int score = classifiers[0].getFeatureValue(img.integral);

    if (score < totalMin) {
        totalMin = score;
    }
}

```

```

double correctPos = 0.0;
double correctNeg = 0.0;
double correctPos2 = 0.0;
double correctNeg2 = 0.0;
int previous = classifiers[0].getPolarity();
//try each polarity and pick the one that works the best
classifiers[0].setPolarity(1);

for (IntegralImage im : trainPos) if (im.label == classifiers[0].getClass(im.integral))
    correctPos++;
for (IntegralImage im : trainNeg) if (im.label == classifiers[0].getClass(im.integral))
    correctNeg++;

classifiers[0].setPolarity(-1);

for (IntegralImage im : trainPos) if (im.label == classifiers[0].getClass(im.integral))
    correctPos2++;
for (IntegralImage im : trainNeg) if (im.label == classifiers[0].getClass(im.integral))
    correctNeg2++;

if (previous == classifiers[0].getPolarity()) {
    classifiers[0].error = 1.0-
(correctPos2+correctNeg2*1.0)/(trainPos.size()+trainNeg.size()*1.0);
} else {
    classifiers[0].error = 1.0-
(correctPos+correctNeg*1.0)/(trainPos.size()+trainNeg.size()*1.0);
}
classifiers[0].setPolarity(previous);
bestErrorFound = classifiers[0].weightedError;
return classifiers;
}

//an alternate method for added more classifiers if some amount is not good enough
public static HaarFeature[] extend(ArrayList<IntegralImage>
trainPos,ArrayList<IntegralImage> trainNeg,HaarFeature[] previousClassifiers) {
    if (trainNeg.size() == 0) {
        return null;
    }

    HaarFeature[] classifiers = new HaarFeature[previousClassifiers.length+1];

    for (int i = 0; i < previousClassifiers.length;i++) classifiers[i] = previousClassifiers[i];

    int i = previousClassifiers.length;

```

```

double totalWeight = 0.0;

for (IntegralImage im : trainPos) totalWeight += im.weight;
for (IntegralImage im : trainNeg) totalWeight += im.weight;

double norm = 1.0/totalWeight;

for (IntegralImage im : trainPos) im.weight = im.weight*norm;
for (IntegralImage im : trainNeg) im.weight = im.weight*norm;

double bestErrorFound = Double.MAX_VALUE;

for (HaarFeature feature : featureSet) {
    if (used.get(feature))
        continue;

    Map<IntegralImage,Integer> scores = new HashMap<>();

    for (IntegralImage img : trainPos) {
        int score = feature.getFeatureValue(img.integral);
        scores.put(img, score);
    }

    for (IntegralImage img : trainNeg) {
        int score = feature.getFeatureValue(img.integral);
        scores.put(img, score);
    }

    TreeSet<Map.Entry<IntegralImage,Integer>> set = new
TreeSet<Map.Entry<IntegralImage,Integer>>(
        new Comparator<Map.Entry<IntegralImage,Integer>>() {
            @Override public int compare(Map.Entry<IntegralImage,Integer> e1,
Map.Entry<IntegralImage,Integer> e2) {
                int res = e1.getValue().compareTo(e2.getValue());
                return res != 0 ? res : 1;
            }
        }
    );

    set.addAll(scores.entrySet());

    HashMap<Double,Integer> type1s = new HashMap<>();
    HashMap<Double,Integer> type2s = new HashMap<>();
    double tPos = 0.0;
    double tNeg = 0.0;
    double sPos = 0.0;

```

```

double sNeg = 0.0;

for (IntegralImage im : trainNeg)
    tNeg += im.weight;
for (IntegralImage im : trainPos)
    tPos += im.weight;

Iterator<Entry<IntegralImage, Integer>> it = set.iterator();
Entry<IntegralImage, Integer> previous = it.next();
Entry<IntegralImage, Integer> next = null;

while(it.hasNext()) {
    if (previous.equals(set.first()) && next == null) {
        double type1 = sPos + (tNeg-sNeg);
        double type2 = sNeg + (tPos-sPos);

        if (type1 > type2) {
            type2s.put(type2,previous.getValue());
        } else {
            type1s.put(type1,previous.getValue());
        }

        next = it.next();
    } else if (it.hasNext()) {
        if (rips.get(previous.getKey())){
            sPos += previous.getKey().weight;
        } else {
            sNeg += previous.getKey().weight;
        }

        double type1 = sPos + (tNeg-sNeg);
        double type2 = sNeg + (tPos-sPos);

        if (type1 > type2) {
            type2s.put(type2,previous.getValue());
        } else {
            type1s.put(type1,previous.getValue());
        }

        previous = next;
        next = it.next();

        if (!it.hasNext()) {
            previous = next;
        }
    }
}

```

```

        if (rips.get(previous.getKey())){
            sPos += previous.getKey().weight;
        } else {
            sNeg += previous.getKey().weight;
        }

        double lastType1 = sPos + (tNeg-sNeg);
        double lastType2 = sNeg + (tPos-sPos);
        if (lastType1 > lastType2) {
            type2s.put(lastType2,previous.getValue());
        } else {
            type1s.put(lastType1,previous.getValue());
        }
    }
}

if (type2s.isEmpty() && type1s.isEmpty()) {
    feature.weightedError = Double.MAX_VALUE;
} else if (type2s.isEmpty()){
    feature.weightedError = Collections.min(type1s.keySet());
    feature.setThreshold(type1s.get(feature.weightedError));
    feature.setPolarity(-1);
} else if (type1s.isEmpty()) {
    feature.weightedError = Collections.min(type2s.keySet());
    feature.setThreshold(type2s.get(feature.weightedError));
    feature.setPolarity(1);
} else {
    double type2Min = Double.MAX_VALUE;
    double type1Min = Double.MAX_VALUE;
    type2Min = Collections.min(type2s.keySet());
    type1Min = Collections.min(type1s.keySet());

    feature.weightedError = type1Min;
    feature.setThreshold(type1s.get(type1Min));
    feature.setPolarity(-1);

    if (feature.weightedError > type2Min) {
        feature.weightedError = type2Min;
        feature.setThreshold(type2s.get(type2Min));
        feature.setPolarity(1);
    }
}

```



```

        if (feature.weightedError > 0 && feature.weightedError < 1) {
            feature.weight = Math.log((1.0-feature.weightedError)/feature.weightedError);
        } else {
            feature.weightedError = 0;
            feature.weight = 4;
        }

        if (feature.weightedError < bestErrorFound) {
            bestErrorFound = feature.weightedError;
            classifiers[i] = feature;
        }
    }

    bestErrorFound = classifiers[i].weightedError;

for (IntegralImage img : trainPos) {
    int score = classifiers[i].getFeatureValue(img.integral);
    if (score > totalMax) {
        totalMax = score;
    }
}

for (IntegralImage img : trainNeg) {
    int score = classifiers[i].getFeatureValue(img.integral);

    if (score < totalMin) {
        totalMin = score;
    }
}

double correctPos = 0.0;
double correctNeg = 0.0;
double correctPos2 = 0.0;
double correctNeg2 = 0.0;
int previous = classifiers[i].getPolarity();
classifiers[i].setPolarity(1);

for (IntegralImage im : trainPos) if (im.label == classifiers[i].getClass(im.integral))
    correctPos++;
for (IntegralImage im : trainNeg) if (im.label == classifiers[i].getClass(im.integral))
    correctNeg++;

classifiers[i].setPolarity(-1);

```

```

        for (IntegralImage im : trainPos) if (im.label == classifiers[i].getClass(im.integral))
correctPos2++;
        for (IntegralImage im : trainNeg) if (im.label == classifiers[i].getClass(im.integral))
correctNeg2++;
        if (previous == classifiers[i].getPolarity()) {
            classifiers[i].error = 1.0-
(correctPos2+correctNeg2*1.0)/(trainPos.size()+trainNeg.size()*1.0);
        } else {
            classifiers[i].error = 1.0-
(correctPos+correctNeg*1.0)/(trainPos.size()+trainNeg.size()*1.0);
        }
        classifiers[i].setPolarity(previous);
        used.put(classifiers[i],true);

//update the weights for each image based on what was misclassified
if (bestErrorFound >= 0) {
    for (IntegralImage im : trainPos)
        if (im.label == classifiers[i].getClass(im.integral))
            im.weight = im.weight*(bestErrorFound/(1.0-bestErrorFound));

    for (IntegralImage im : trainNeg)
        if (im.label == classifiers[i].getClass(im.integral))
            im.weight = im.weight*(bestErrorFound/(1.0-bestErrorFound));
    } else {
        System.out.println("Cannot acheive lower error.");
        classifiers = null;
    }
    return classifiers;
}
}

```

## B. Source Code for Integral Image in Java

```
package haar;

import java.awt.Color;
import java.awt.image.BufferedImage;
import java.io.Serializable;

public class IntegralImage implements Serializable{

    /**
     *
     */
    private static final long serialVersionUID = -3700022838878542056L;
    public int[][] integral;
    public int label;
    public double weight;

    //calculate the integral for a 2-d matrix
    public IntegralImage(int[][] img,int label) {
        this.label = label;
        integral = new int[img.length][img[0].length];
        integral[0][0] = img[0][0]; //rbg should be the same for each color now that it is grey scale
        //used to convert the image to grey-scale
        for (int i=0;i<img.length;i++) {
            for (int j=0;j<img[0].length;j++) {

                if (j-1 < 0) {
                    if(i-1 < 0){
                        integral[i][j] = img[i][j];

                    } else {
                        integral[i][j] = img[i][j] + img[i-1][j];
                    }
                } else if (i-1 < 0) {
                    if(j-1 < 0){
                        integral[i][j] = img[i][j];

                    } else {
                        integral[i][j] = img[i][j] + img[i][j-1];
                    }
                } else {
                    integral[i][j] = img[i][j] + img[i][j-1] + img[i-1][j] - img[i-1][j-1];
                }
            }
        }
    }
}
```

```

    }
  }
}

```

```

//calculate the integral for a BufferedImage object
public IntegralImage(BufferedImage img,int label) {
    this.label = label;
    integral = new int[img.getWidth()][img.getHeight()];

```

```

//used to convert the image to grey-scale

```

```

for (int i=0;i<img.getWidth();i++) {
    for (int j=0;j<img.getHeight();j++) {

```

```

        Color c = new Color(img.getRGB(i, j));

```

```

        int red = (int)(c.getRed() * 0.299);

```

```

        int green = (int)(c.getGreen() * 0.587);

```

```

        int blue = (int)(c.getBlue() *0.114);

```

```

        Color newColor = new Color(red+green+blue,red+green+blue,red+green+blue);

```

```

        img.setRGB(i, j, newColor.getRGB());
    }
}

```

integral[0][0] = (img.getRGB(0, 0)>>16)&0xFF;//rbg should be the same for each color  
now that it is grey scale

```

for (int i = 0;i < img.getWidth();i++) {
    for (int j = 0;j < img.getHeight();j++) {
        Color color = new Color(img.getRGB(i, j));
        if (j-1 < 0) {
            if(i-1 < 0){
                integral[i][j] = color.getBlue();

            }else{
                Color shiftColor = new Color(img.getRGB(i-1, j));
                integral[i][j] = color.getBlue() + shiftColor.getBlue();
            }
        }else if (i-1 < 0) {
            if(j-1 < 0){
                integral[i][j] = color.getBlue();

            }else{
                Color shiftColor = new Color(img.getRGB(i, j-1));
                integral[i][j] = color.getBlue() + shiftColor.getBlue();
            }
        } else {

```

```

        Color shiftColor = new Color(img.getRGB(i, j-1));
        Color shiftColor2 = new Color(img.getRGB(i-1, j));
        Color shiftColor3 = new Color(img.getRGB(i-1, j-1));
        integral[i][j] = color.getBlue() + shiftColor.getBlue() + shiftColor2.getBlue() -
shiftColor3.getBlue();
    }
}
}
}

```

```

//this method uses the top left point and bottom right point to calculate the area of a rectangle
public static int getAreaSum(int[][] integral,int topLeftX,int topLeftY,int bottomRightX,int
bottomRightY) {
    int topRightX = topLeftX;
    int topRightY = bottomRightY;
    int bottomLeftX = bottomRightX;
    int bottomLeftY = topLeftY;
    return integral[bottomRightX][bottomRightY] - integral[topRightX][topRightY] -
integral[bottomLeftX][bottomLeftY] + integral[topLeftX][topLeftY];
}
}

```

### C. Source Code for New Haar features in Java

```
package haar;

public class T extends HaarFeature {

    public T(int x, int y,int width, int height) {
        super(x,y,width,height);
        this.featureType = TYPE.T;
        this.feature_orientation[0] = 3;
        this.feature_orientation[1] = 3;
    }

    //each feature that extends HaarFeature must implement this getFeatureValueMethod in a
    unique way. This is how we attain the feature for an image.
    @Override
    public int getFeatureValue(int[][] intImage) {
        int first = IntegralImage.getAreaSum(intImage,this.x,this.y,this.x+this.height-
        1,this.y+this.width-1);

        int second = IntegralImage.getAreaSum(intImage,this.x,this.y+this.width,this.x+this.height-
        1,this.y+this.width*2-1);

        int third = IntegralImage.getAreaSum(intImage, this.x, this.y+2*this.width,
        this.x+this.height-1, this.y+3*this.width-1);

        int fifth =
        IntegralImage.getAreaSum(intImage,this.x+2*this.height,this.y+this.width,this.x+this.height
        *3-1,this.y+this.width*2-1);

        int seventh = IntegralImage.getAreaSum(intImage, this.x+this.height, this.y+this.width,
        this.x+this.height*2-1, this.y+2*this.width-1);

        return first+second+third-seventh+fifth;
    }
}

public class InvertedT extends HaarFeature {

    public InvertedT(int x, int y,int width, int height) {
        super(x,y,width,height);
        this.featureType = TYPE.INVERTED_T;
        this.feature_orientation[0] = 3;
        this.feature_orientation[1] = 3;
    }
}
```

```

    }

    @Override
    public int getFeatureValue(int[][] intImage) {
        int second =
            IntegralImage.getAreaSum(intImage,this.x,this.y+this.width,this.x+this.height-
            1,this.y+this.width*2-1);

        int fourth = IntegralImage.getAreaSum(intImage, this.x+2*this.height, this.y,
            this.x+this.height*3-1, this.y+this.width-1);

        int fifth =
            IntegralImage.getAreaSum(intImage,this.x+2*this.height,this.y+this.width,this.x+this.height*3-1,this.y+this.width*2-1);

        int sixth = IntegralImage.getAreaSum(intImage, this.x+2*this.height, this.y+2*this.width,
            this.x+this.height*3-1, this.y+3*this.width-1);

        int seventh = IntegralImage.getAreaSum(intImage, this.x+this.height, this.y+this.width,
            this.x+this.height*2-1, this.y+2*this.width-1);

        return sixth+fourth+fifth- second+seventh;
    }
}

public class ThreeColumns extends HaarFeature{

    public ThreeColumns(int x, int y,int width, int height) {
        super(x,y,width,height);
        this.featureType = TYPE.THREE_COLUMNS;
        this.feature_orientation[0] = 3;
        this.feature_orientation[1] = 3;
    }

    @Override
    public int getFeatureValue(int[][] intImage) {
        int first = IntegralImage.getAreaSum(intImage,this.x,this.y,this.x+this.height-
        1,this.y+this.width-1);

        int second = IntegralImage.getAreaSum(intImage,this.x,this.y+this.width,this.x+this.height-
        1,this.y+this.width*2-1);

        int third = IntegralImage.getAreaSum(intImage, this.x, this.y+2*this.width,
            this.x+this.height-1, this.y+3*this.width-1);
    }
}

```

```

        int fourth = IntegralImage.getAreaSum(intImage, this.x+2*this.height, this.y,
        this.x+this.height*3-1, this.y+this.width-1);

        int fifth =
        IntegralImage.getAreaSum(intImage,this.x+2*this.height,this.y+this.width,this.x+this.height
        *3-1,this.y+this.width*2-1);

        int sixth = IntegralImage.getAreaSum(intImage, this.x+2*this.height, this.y+2*this.width,
        this.x+this.height*3-1, this.y+3*this.width-1);

        int seventh = IntegralImage.getAreaSum(intImage, this.x+this.height, this.y+this.width,
        this.x+this.height*2-1, this.y+2*this.width-1);

        int eighth = IntegralImage.getAreaSum(intImage, this.x+this.height, this.y,
        this.x+this.height*2-1, this.y+this.width-1);

        int ninth = IntegralImage.getAreaSum(intImage, this.x+this.height*2, this.y+this.width,
        this.x+this.height*2-1, this.y+3*this.width-1);

        return third+ninth+sixth+first+eighth+fourth-second+seventh+fifth;
    }
}

public class X extends HaarFeature{

    public X(int x, int y,int width, int height) {
        super(x,y,width,height);
        this.featureType = TYPE.X;
        this.feature_orientation[0] = 3;
        this.feature_orientation[1] = 3;
    }

    @Override
    public int getFeatureValue(int[][] intImage) {
        int first = IntegralImage.getAreaSum(intImage,this.x,this.y,this.x+this.height-
        1,this.y+this.width-1);

        int third = IntegralImage.getAreaSum(intImage, this.x, this.y+2*this.width,
        this.x+this.height-1, this.y+3*this.width-1);

        int fourth = IntegralImage.getAreaSum(intImage, this.x+2*this.height, this.y,
        this.x+this.height*3-1, this.y+this.width-1);

        int sixth = IntegralImage.getAreaSum(intImage, this.x+2*this.height, this.y+2*this.width,
        this.x+this.height*3-1, this.y+3*this.width-1);

```



```

        int seventh = IntegralImage.getAreaSum(intImage, this.x+this.height, this.y+this.width,
        this.x+this.height*2-1, this.y+2*this.width-1);

        return first+third+fourth+sixth-seventh;
    }
}

public class Cross extends HaarFeature{

    public Cross(int x, int y,int width, int height) {
        super(x,y,width,height);
        this.featureType = TYPE.CROSS;
        this.feature_orientation[0] = 3;
        this.feature_orientation[1] = 3;
    }

    @Override
    public int getFeatureValue(int[][] intImage) {
        int second =
        IntegralImage.getAreaSum(intImage,this.x,this.y+this.width,this.x+this.height-
        1,this.y+this.width*2-1);

        int fifth =
        IntegralImage.getAreaSum(intImage,this.x+2*this.height,this.y+this.width,this.x+this.height
        *3-1,this.y+this.width*2-1);

        int seventh = IntegralImage.getAreaSum(intImage, this.x+this.height, this.y+this.width,
        this.x+this.height*2-1, this.y+2*this.width-1);

        int eighth = IntegralImage.getAreaSum(intImage, this.x+this.height, this.y,
        this.x+this.height*2-1, this.y+this.width-1);

        int ninth = IntegralImage.getAreaSum(intImage, this.x+this.height*2, this.y+this.width,
        this.x+this.height*2-1, this.y+3*this.width-1);

        return second+seventh-fifth+ninth+eighth;
    }
}

import violajones.AdaBoost;

public abstract class HaarFeature {

    public TYPE featureType;

```

```

protected int x;
protected int y;
protected int width;
protected int height;
private int threshold;
private int polarity;
public final int[] feature_orientation = new int[2];
public double error = Double.POSITIVE_INFINITY;
public double weight = 0.0;
public double weightedError = Double.POSITIVE_INFINITY;

public enum TYPE {

    TWO_VERTICAL,TWO_HORIZONTAL,THREE_HORIZONTAL,THREE_VERTICAL,FOUR,X,T,INVERTED_T,THREE_COLUMNS,CROSS;
}

public HaarFeature (int x,int y, int width, int height){
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
}

public int getX() {
    return this.x;
}

public int getY() {
    return this.y;
}

public int getWidth() {
    return this.width;
}

public int getHeight() {
    return this.height;
}

public int getThreshold() {
    return this.threshold;
}

public int getPolarity() {
    return this.polarity;
}

```

```

}

public final void setThreshold(int threshold) {
    this.threshold = threshold;
}

public final void setWeight(int weight) {
    this.weight = weight;
}

public final void setPolarity(int polarity) {
    this.polarity = polarity;
}

public abstract int getFeatureValue(int[][] intImage);

public final int getClass(int[][] img){
    int score = getFeatureValue(img);

    double normalScore = ((score - AdaBoost.totalMin)*1.0/(AdaBoost.totalMax-
AdaBoost.totalMin)*1.0);

    double normalThreshold = ((this.threshold - AdaBoost.totalMin)*1.0/(AdaBoost.totalMax-
AdaBoost.totalMin)*1.0);

    if (normalScore * this.polarity < normalThreshold*this.polarity) return 1;
    else return 0;
}
}

```

#### D. Source Code for PCA in Matlab

```
% @Maryan: Rip Current Detection using PCA
% Load the training and test images
=====
=====
Train_image_vec = uint8(zeros(576,415)); % Dimension of an image is = 112 x 92 = 10304
pixels; I want to load 40 x 8 [index: 1 to 8] = 320 images for training
Test_image_vec = uint8(zeros(576,46)); % I want to load 40 x 2 [index: 9 to 10] = 80 images
for testing
Test_image_neg_vec = uint8(zeros(576,46));

%Load rip
currents=====
%IF USING WINDOWS CHANGE THE FORWARD SLASHES TO BACK SLASHES
trainImageFiles = dir('positive_rips/*.pgm')
testImageFiles = dir('test_rips/*.pgm')
testImageFilesNeg = dir('negatives/*.pgm')

for i=1:415
    img = imread(strcat('positive_rips/',trainImageFiles(i).name));
    Train_image_vec(:,i) = reshape(img, 576,1);
end

for i=1:46
    img = imread(strcat('test_rips/',testImageFiles(i).name));
    Test_image_vec(:,i) = reshape(img, 576,1);
end

for i=1:46
    img = imread(strcat('negatives/',testImageFilesNeg(i).name));
    Test_image_neg_vec(:,i) = reshape(img, 576,1);
end
%=====
=====
% Take the average of the respective pixel-values of all the training images
=====
avg=uint8(mean(Train_image_vec,2)); % 2=> row-wise avg

% Remove the computed average value from the each of the training images and create "Normal
vector"=====
Normal_Train_Vec=[];
for i=1:415
    Normal_Train_Vec(:, i) = Train_image_vec(:, i) - avg;
end
```

```

%Generate covariance matrix M
=====
=====
M = Normal_Train_Vec' * Normal_Train_Vec;
[U, E, V] = svd(M); % U = EigenVector, E = EigenValue

%Let us pick 1st best 15 EigenVectors' corresponing normal training vectors (from 10304x320
to 10304x15) =====
U=Normal_Train_Vec * U; % Project the normal training vectors towards the EigenValues
U30=U(:,1:30); % Take 15 Eigen_normal_training_vec
last30 = U(:,(415-29):415])
%%=====test ALL components for detection of rip currents
Component_Wise_Det_FP_Rate = []

for i = 1:415
    component = U(:,i);%%select individual component
    Training_Feature = [];

    for j = 1:415
        Training_Feature(j,:)= uint8(single(Normal_Train_Vec(:,j))*single(component));
    %%generate features for this component
    end

    featureAvg = int8(mean(Training_Feature,1))%column wise feature avg
    distancesFromAvg = []

    for j = 1:415
        distancesFromAvg(j,:) = Training_Feature(j,:) - featureAvg%calcualte the diff from
average for each sample
    end

    distancesFromAvg = abs(distancesFromAvg) %distance is always pos
    max_distance = max(distancesFromAvg)%max distance from avg
    numCorPos = 0
    numCorNeg = 0

    %%testing pos samples
    for j = 1:46
        feature_distance = abs(int8(uint8(single(uint8(Test_image_vec(:,j)-
avg))*single(component))) - featureAvg);% get the distance vector from average
        %=====this uses max total distance for this component
        if (feature_distance <= max_distance)
            numCorPos++
        end
    end
end
end

```

```

%%testing neg samples
for j = 1:46
    feature_distance = abs(int8(uint8(single(uint8(Test_image_neg_vec(:,j)-
avg))*single(component)))) - featureAvg)

    if (feature_distance > max_distance_lowest_component)
        numCorNeg++
    end
end

Component_Wise_Det_FP_Rate(i,:) =
[single(numCorPos)/single(46),single(numCorNeg)/single(46)]
end

csvwrite('distancePerComponent.csv',Component_Wise_Det_FP_Rate);

% Extract the features from training ... each row in Training_Feature is the pattern of one
training images.=====
Training_Feature = [];
Training_Lowest_feature = [];
Training_Last_30_feature = [];

for i = 1:415
    Training_Feature(i,:) = uint8(single(Normal_Train_Vec(:,i))*single(U30));
end

for i = 1:415
    Training_Last_30_feature(i,:) = uint8(single(Normal_Train_Vec(:,i))*single(last30));
end

featureAvg = int8(mean(Training_Feature,1));%column-wise feature avg
last30FeatureAvg = int8(mean(Training_Last_30_feature,1));%column-wise feature avg
distances = [];
last30Distances = [];

for i = 1:415
    distances(i,:) = Training_Feature(i,:)-featureAvg; % calculate distance from avg vector
end

for i = 1:415
    last30Distances(i,:) = Training_Last_30_feature(i,:)-last30FeatureAvg; % calculate distance
from avg vector
end

entire_eigen_feature_vector = []

```

```

for i = 1:415
    entire_eigen_feature_vector(i,:) = uint8(single(Normal_Train_Vec(:,i))*single(U));
end

entire_distances = []

featureEntireAvg = single(mean(entire_eigen_feature_vector,1))

for i = 1:415
    entire_distances(i,:) = single(entire_eigen_feature_vector(i,:)-featureEntireAvg); % calculate
distance from avg vector
end

last30Distances = abs(last30Distances)
entire_distances = abs(entire_distances)
distances = abs(distances);
total_distance_per_component = [];

%total distance per compoenent
for i = 1:30
    total_distance_per_component(i,:) = sum(distances(:,i))
end

csvwrite('distancePerComponent.csv',total_distance_per_component);

min_distances = [];
mean_entire_distances = []
mean_distances = mean(distances,1); % the average distance for each component
mean_entire_distances = mean(entire_distances,1)
max_distances = [];

csvwrite('avgDistancePerComponent.csv',mean_distances);
csvwrite('avgDistancePerComponentEntire.csv',mean_entire_distances);

lowest_components = U30(:,[11 18 27 30])
lowest_avg = int8(featureAvg(:,[11 18 27 30]))

for i = 1:415
    Training_Lowest_Feature(i,:)=
        uint8(single(Normal_Train_Vec(:,i))*single(lowest_components));
end

lowest_component_distances = []

for i = 1:415

```

```

        lowest_component_distances(i,:) = int8(Training_Lowest_Feature(i,:))-lowest_avg; %
        calculate distance from avg vector
    end

    lowest_component_distances = abs(lowest_component_distances)

    for i = 1:30
        max_distances(i,:) = max(distances(:,i)) %column wise max distance for each component
        from average feature
    end

    for i = 1:30
        min_distances(i,:) = min(distances(i,:)) %column wise min distance for each component from
        average feature
    end

    max_distance = 0;
    max_distance_lowest_component = 0;

    for i = 1:415
        sum_distance = sum(distances(i,:));

        if sum_distance > max_distance
            max_distance = sum_distance
        end
    end

    for i = 1:415
        sum_distance = sum(lowest_component_distances(i,:));

        if sum_distance > max_distance_lowest_component
            max_distance_lowest_component = sum_distance
        end
    end
    numCor = 0;

    for i = 1:46
        feature_distances = abs(int8(uint8(single(uint8(Test_image_vec(:,i)-
        avg))'*single(lowest_components)))) - lowest_avg;% get the distance vector from average
        %=====this uses max total distance over the whole vector

        max_feature_distance = sum(feature_distances)
        if (max_feature_distance <= max_distance_lowest_component)
            numCor++
        end
    end
end

```



```

for i = 1:46
    feature_distances = abs(int8(uint8(single(uint8(Test_image_neg_vec(:,i)-
avg))*single(lowest_components)))) - lowest_avg)
    feature_distance = sum(feature_distances)

    if (feature_distance > max_distance_lowest_component)
        numCor++
    end
end

for i = 1:46
    %=====the commented out code below does the same as the one line...its just explained
    % Test_img_norm = Test_image_vec(:,i)-avg
    % Test_feature = uint8(single(Test_img_norm)*single(U30))
    % feature_distance = sum(abs(int8(Test_feature) - featureAvg))
    feature_distances = abs(int8(uint8(single(uint8(Test_image_neg_vec(:,i)-avg))*single(U30)))) -
featureAvg);% get the distance vector from average
    for i = 1:30
        if feature_distances(:,i) > 127 % this compares individual distances for each component
            numWrong++
        end
    end
    %=====this uses max total distance over the whole vector
    max_feature_distance = sum(feature_distances)
    if (max_feature_distance <= max_distance)
        numCor++
    end
end

for i = 1:46
    feature_distances = abs(int8(uint8(single(uint8(Test_image_neg_vec(:,i)-avg))*single(U30)))) -
featureAvg)
    if (feature_distance > max_distance)
        numCor++
    end
end

result = single(numCor)/single(92)

```

## E. Source Code for generating Meta Learner Feature Vector in Python

```
from numpy import genfromtxt
from sklearn import svm
from sklearn import neural_network
from sklearn import neighbors
from sklearn import tree
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier, BaggingClassifier
from sklearn import naive_bayes
from random import *
from csv import reader
from sklearn.preprocessing import *
import numpy as np
import math

# Load a CSV file
def load_csv(filename):
    dataset = list()
    with open(filename, 'r') as file:
        csv_reader = reader(file)
        for row in csv_reader:
            if not row:
                continue
            dataset.append(row)
    return dataset

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
    dataset_split = list()
    dataset_copy = list(dataset)
    fold_size = int(len(dataset) / n_folds)
    for i in range(n_folds):
        fold = list()

        while len(fold) < fold_size:
            index = randrange(len(dataset_copy))
            fold.append(dataset_copy.pop(index))

        dataset_split.append(fold)
    while len(dataset_copy) > 0:
        index = randrange(len(dataset_split))
        dataset_split[index].append(dataset_copy.pop(0))
    return dataset_split

#data for the meta-learner
data = load_csv('new_raw_data_with_ensemble_small.csv')
```

```

#rect_test_data = np.genfromtxt('testRect.csv',delimiter=',')
#rect_test_data = np.asarray(rect_test_data,np.float64)
conf_fold = [dict(),dict(),dict(),dict(),dict(),dict(),dict(),dict()]

#conf_fold = [dict(),dict(),dict()]
folds = cross_validation_split(data, 10)
det_rates = [[],[],[],[],[],[],[],[]]
fp_rates = [[],[],[],[],[],[],[],[]]
accuracy = [[],[],[],[],[],[],[],[]]

#det_rates = [[],[],[]]
#fp_rates = [[],[],[]]
#accuracy = [[],[],[]]

names = ["Nearest Neighbors",
         "RBF SVM",
         "Decision Tree",
         "Random Forest", "Neural Net",
         "AdaBoost",
         "Bagging",
         "Naive Bayes"]
fold_idx = 0
for fold in folds:
    train_data = list(folds)
    train_data.remove(fold)
    train_data = sum(train_data,[])
    test_data = list()
    classes = []
    test_classes = []
    train_copy = []

    #removing the class label from test and training data
    for row in train_data:
        row_copy = list(row)
        train_copy.append(row_copy)
        classes.append(row_copy[len(row_copy)-1])
        row_copy.remove(row_copy[len(row_copy)-1])

    for row in fold:
        row_copy = list(row)
        test_data.append(row_copy)
        test_classes.append(row_copy[len(row_copy)-1])
        row_copy.remove(row_copy[len(row_copy)-1])

    train_copy = np.asarray(train_copy,np.float64)
    test_data = np.asarray(test_data,np.float64)

```

```

test_classes = np.asarray(test_classes,np.float64)
classes = np.asarray(classes,np.float64)
scaler = RobustScaler(copy=True)
train_copy = scaler.fit_transform(train_copy)
# print str(len(train_copy[0]))+' '+str(len(rect_test_data[0]))
test_data = scaler.transform(test_data)

#creating models
classifiers = [
    neighbors.KNeighborsClassifier(50),
    svm.SVC(kernel='rbf',probability=True,C=4.0,gamma=0.00390625),
    tree.DecisionTreeClassifier(max_depth=10),
    RandomForestClassifier(max_depth=10),
    neural_network.MLPClassifier(max_iter=600),
    AdaBoostClassifier(n_estimators=200),
    BaggingClassifier(),
    naive_bayes.GaussianNB()
]
idx = 0
probs = []
for model in classifiers:
    model.fit(train_copy,classes)
    results = model.predict(test_data)
    prob = model.predict_proba(test_data)

#these lines are used if we are generating data for output from OpenCV
# if fold_idx == 0:
#     scale_rect_data = scaler.transform(rect_test_data)
#     prob_rect = model.predict_proba(scale_rect_data)
#     probs.append(prob_rect)
false_positive = 0
false_negative = 0
true_positive = 0
true_negative = 0
num_pos = 0
num_neg = 0

for i in range(0,results.size):
    if results[i] == 0 and test_classes[i] == 1:
        false_negative += 1
        num_pos += 1
    elif results[i] == 1 and test_classes[i] == 0:
        false_positive += 1
        num_neg += 1
    elif results[i] == 1 and test_classes[i] == 1:
        true_positive += 1

```

```

        num_pos += 1
    elif results[i] == 0 and test_classes[i] == 0:
        true_negative += 1
        num_neg += 1

#map the confidence for each sample classified to its sample
    for index in range(len(prob)):

        conf_fold[idx][fold[index][0]+fold[index][1]+fold[index][33]+fold[index][34]] =
prob[index][1]

        det_rates[idx].append((true_positive)/(num_pos*1.0))
        fp_rates[idx].append(false_positive/(num_neg*1.0))
        accuracy[idx].append((true_positive+true_negative)/(num_pos+num_neg*1.0))
        #print 'accuracy ' + str((true_positive+true_negative)/(num_pos+num_neg*1.0))
        #print "det " + str((true_positive)/(num_pos*1.0)) + " fp " +
str(false_positive/(num_neg*1.0)) + " tn " + str(true_negative/(num_neg*1.0)) + " fn " +
str(false_negative/(num_pos*1.0))+ "\n"
        idx += 1
    if fold_idx == 0:
        for model_conf in probs:
            pos_conf = []
            for conf in model_conf:
                pos_conf.append(conf[1])
            rect_test_data = np.c_[rect_test_data,pos_conf]

        fold_idx+= 1
average_det = 0.0
average_fp = 0.0
average_acc = 0.0
classes = []
idx = 0

for rates in det_rates:
    average_det = 0
    if not rates:
        continue
    for rate in rates:
        average_det += rate
    print 'average det for ' + names[idx] + ' ' + str(average_det/(len(rates)*1.0))
    idx+= 1

idx = 0

for rates in fp_rates:
    average_fp = 0
    if not rates:

```

```

        continue
    for rate in rates:
        average_fp += rate
    print 'average fp for ' + names[idx] + ' ' + str(average_fp/(len(rates)*1.0))
    idx += 1

idx = 0

for rates in accuracy:
    average_acc = 0
    if not rates:
        continue
    for rate in rates:
        average_acc += rate
    print 'average accuracy for ' + names[idx] + ' ' + str(average_acc/(len(rates)*1.0))
    idx += 1
idx = 0

# These lines are used to calculate the Pearson correlation coefficient between each model
# for i in range(len(conf_fold)):
#     for j in range(i+1, len(conf_fold)):
#         xy = 0
#         totalX = 0
#         totalY = 0
#         totalXsq = 0
#         totalYsq = 0
#         for row in data:
#             for key in conf_fold[i]:
#                 if key == row[0]+row[1]+row[10]+row[11]:
#                     totalX += conf_fold[i][key]
#                     totalY += conf_fold[j][key]
#                     totalXsq += (conf_fold[i][key]*conf_fold[i][key])
#                     totalYsq += (conf_fold[j][key]*conf_fold[j][key])
#                     xy += conf_fold[i][key] * conf_fold[j][key]
#         pcc = ((len(data)*xy) - (totalX*totalY))/(math.sqrt((len(data)*totalXsq) -
# math.pow(totalX,2)) * math.sqrt((len(data)*totalYsq) - math.pow(totalY,2)))
#         print 'the pcc between ' + names[i] + ' and ' + names[j] + ' is ' + str(pcc)
# data_np = genfromtxt('best_feature_data_small.csv', delimiter=',')
data_np = np.asarray(data_np, np.float64)
class_col = data_np[:, len(data_np[0])-1]
data_np = np.delete(data_np, len(data_np[0])-1, 1)

for model in conf_fold:
    conf = []
    for row in data:
        for key in model:

```

```

        if row[0]+row[1]+row[33]+row[34] == key:
            conf.append(model[key])
    data_np = np.c_[data_np,conf]

data_np = np.c_[data_np,class_col]
np.savetxt('new_raw_data_with_ensemble_small.csv',data_np,delimiter=',')
#np.savetxt('ready_rect.csv',rect_test_data,delimiter=',')

```

## F. Source Code for Meta learner in Python

```
from numpy import genfromtxt
from sklearn import svm
from sklearn import neural_network
from sklearn import neighbors
from sklearn import tree
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier, BaggingClassifier
from sklearn import naive_bayes
from random import *
from csv import reader
from sklearn.preprocessing import *
from PIL import Image
import cv2
import numpy as np
from os import listdir, walk
from os.path import isfile, join

#load a .csv file into an array
def load_csv(filename):
    dataset = list()
    with open(filename, 'r') as file:
        csv_reader = reader(file)
        for row in csv_reader:
            if not row:
                continue
            dataset.append(row)
    return dataset

#calculate the corresponding integral for a given image
def calculate_integral(im):
    integral = np.zeros((len(im), len(im[0])), dtype = 'int64')
    integral[0][0] = im[0][0]

    for i in range(len(im)):
        for j in range(len(im[0])):
            if j-1 < 0:
                if i-1 < 0:
                    integral[i][j] = im[i][j]
                else:
                    integral[i][j] = im[i][j] + im[i-1][j]
            elif i-1 < 0:
                if j-1 < 0:
                    integral[i][j] = im[i][j]
                else:
                    integral[i][j] = im[i][j] + im[i][j-1]
```



```

        else:
            integral[i][j] = im[i][j] + im[i][j-1] + (im[i-1][j] - im[i-1][j-1])
    return integral

#get the area based on the rectangle between topLeft and bottomRight
def getAreaSum(integral,topLeftX,topLeftY,bottomRightX,bottomRightY):
    topRightX = topLeftX
    topRightY = bottomRightY
    bottomLeftX = bottomRightX
    bottomLeftY = topLeftY

    return integral[bottomRightX][bottomRightY] - integral[topRightX][topRightY] -
    integral[bottomLeftX][bottomLeftY] + integral[topLeftX][topLeftY]

#apply a feature to an integral image based on which type of feature it is
def getFeatureValue(integral,featureType,x,y,w,h):
    score = 0
    if featureType == 'TWO_VERTICAL':
        first = getAreaSum(integral,x,y,x+h-1,y+w-1)
        second = getAreaSum(integral,x+h,y,x+h*2-1,y+w-1)

        score = first - second
    elif featureType == 'THREE_HORIZONTAL':
        first = getAreaSum(integral,x,y,x+h-1,y+w-1)
        second = getAreaSum(integral,x,y+w,x+h-1,y+w*2-1)
        third = getAreaSum(integral,x,y+w*2,x+h-1,y+3*w-1)

        score = first + third - second
    elif featureType == 'THREE_COLUMNS':
        first = getAreaSum(integral,x,y,x+h-1,y+w-1)
        second = getAreaSum(integral,x,y+w,x+h-1,y+w*2-1)
        third = getAreaSum(integral,x,y+2*w,x+h-1,y+3*w-1)
        fourth = getAreaSum(integral,x+2*h,y,x+h*3-1,y+w-1)
        fifth = getAreaSum(integral,x+2*h,y+w,x+h*3-1,y+w*2-1)
        sixth = getAreaSum(integral,x+2*h,y+2*w,x+h*3-1,y+w*3-1)
        seventh = getAreaSum(integral,x+h,y+w,x+2*h-1,y+2*w-1)
        eighth = getAreaSum(integral,x+h,y,x+2*h-1,y+w-1)
        ninth = getAreaSum(integral,x+h*2,y+w,x+h*2-1,y+3*w-1)

        score = third + ninth + sixth + first + eighth + fourth - second + seventh + fifth
    elif featureType == 'X':
        first = getAreaSum(integral,x,y,x+h-1,y+w-1)
        third = getAreaSum(integral,x,y+2*w,x+h-1,y+3*w-1)
        fourth = getAreaSum(integral,x+2*h,y,x+h*3-1,y+w-1)
        sixth = getAreaSum(integral,x+2*h,y+2*w,x+h*3-1,y+w*3-1)
        seventh = getAreaSum(integral,x+h,y+w,x+2*h-1,y+2*w-1)

```

```

        score = first + third + fourth + sixth - seventh
    elif featureType == 'T':
        first = getAreaSum(integral,x,y,x+h-1,y+w-1)
        second = getAreaSum(integral,x,y+w,x+h-1,y+w*2-1)
        third = getAreaSum(integral,x,y+2*w,x+h-1,y+3*w-1)
        fifth = getAreaSum(integral,x+2*h,y+w,x+h*3-1,y+w*2-1)
        seventh = getAreaSum(integral,x+h,y+w,x+2*h-1,y+2*w-1)

        score = first + second + third - seventh + fifth
    else:
        print 'feature type not recognized'

    return score

#load an OpenCV cascade
rip_current_openCV_model =
cv2.CascadeClassifier('part_hand_picked_any_neg/cascade_stage28.xml')

path_to_test_samples = '/Users/Maryan/Documents/Thesis/cascade_data/data/testing_rips/pos'
only_files = [f for f in listdir(path_to_test_samples) if isfile(join(path_to_test_samples,f))]
test_images = np.empty(len(only_files)-1, dtype=object)
integrals_pos = []
integrals_neg = []
rect_test_data =
genfromtxt('/Users/Maryan/Documents/Thesis/cascade_data/data/ready_rect.csv',delimiter=',')

#this code is used if we want to build the meta-classifier data from scratch
'''
for subdir,dirs,files in
walk('/Users/Maryan/Documents/Thesis/cascade_data/data/testing_rips/results/pos_samples_warped/'):
    for _file in files:
        if _file.endswith('.png'):

integrals_pos.append(calculate_integral(np.array(Image.open('/Users/Maryan/Documents/Thesis/cascade_data/data/testing_rips/results/pos_samples_warped/'+str(_file))),dtype='int64'))

for subdir,dirs,files in walk('/Users/Maryan/Documents/Thesis/rip_positives/positive_rips/'):
    for _file in files:
        if _file.endswith('.png'):

integrals_pos.append(calculate_integral(np.array(Image.open('/Users/Maryan/Documents/Thesis/rip_positives/positive_rips/'+str(_file))),dtype='int64'))

for subdir,dirs,files in walk('/Users/Maryan/Documents/Thesis/unaltered_neg/'):

```

```

    for _file in files:
        if _file.endswith('.pgm'):

integrals_neg.append(calculate_integral(np.array(Image.open('/Users/Maryan/Documents/Thesis
/unaltered_neg/'+str(_file))),dtype='int64'))

for subdir,dirs,files in walk('/Users/Maryan/Documents/Thesis/unaltered_images/'):
    for _file in files:
        if _file.endswith('.pgm'):

integrals_neg.append(calculate_integral(np.array(Image.open('/Users/Maryan/Documents/Thesis
/unaltered_images/'+str(_file))),dtype='int64'))

for subdir,dirs,files in
walk('/Users/Maryan/Documents/Thesis/cascade_data/data/testing_rips/results/neg_opencv_sam
ples/'):
    for _file in files:
        if _file.endswith('.png'):

integrals_neg.append(calculate_integral(np.array(Image.open('/Users/Maryan/Documents/Thesis
/cascade_data/data/testing_rips/results/neg_opencv_samples/'+str(_file))),dtype='int64'))

#basic_train_data = []
'''

for n in range(1,len(only_files)):
    test_images[n-1] = cv2.imread(join(path_to_test_samples,only_files[n]))

haar_feature_data = load_csv('/Users/Maryan/Documents/Thesis/cascade_data/data/features.csv')
#meta_train_data =
np.genfromtxt('/Users/Maryan/Documents/Thesis/cascade_data/data/new_raw_data_with_ensem
ble_7_all_pcc_only_haar.csv',delimiter=',')
#basic_train_data =
np.genfromtxt('/Users/Maryan/Documents/Thesis/cascade_data/data/new_data_raw_7_only_haar
.csv',delimiter=',')
meta_train_data =
np.genfromtxt('/Users/Maryan/Documents/Thesis/cascade_data/data/new_raw_data_with_ensem
ble_small.csv',delimiter=',')
basic_train_data =
np.genfromtxt('/Users/Maryan/Documents/Thesis/cascade_data/data/best_feature_data_small.csv
',delimiter=',')

#create data and scalers for data we need one for the meta and one for the basic classifiers
basic_train_data = np.asarray(basic_train_data,np.float64)
basic_class = basic_train_data[:,len(basic_train_data[0])-1]
basic_train_data = np.delete(basic_train_data,len(basic_train_data[0])-1,1)

```

```

scaler_for_basic = RobustScaler(copy=True)
scaler_for_meta = RobustScaler(copy=True)

basic_train_data = scaler_for_basic.fit_transform(basic_train_data)

#for integral in integrals_pos:
#    feature_vector = []
#    for feature in haar_feature_data:
#
feature_vector.append(getFeatureValue(integral,feature[0],int(feature[1]),int(feature[2]),int(feature[3]),int(feature[4])))
#    basic_train_data.append(feature_vector)

#create basic classifiers
basic_classifiers = [
    neighbors.KNeighborsClassifier(50),
    svm.SVC(kernel='rbf',probability=True,C=4.0,gamma=0.00390625),
    tree.DecisionTreeClassifier(max_depth=10),
    RandomForestClassifier(max_depth=10),
    neural_network.MLPClassifier(max_iter=600),
    AdaBoostClassifier(n_estimators=200),
    BaggingClassifier(),
    naive_bayes.GaussianNB()
]

meta_train_data = np.asarray(meta_train_data,np.float64)
meta_class = meta_train_data[:,len(meta_train_data[0])-1]
meta_train_data = np.delete(meta_train_data,len(meta_train_data[0])-1,1)

'''meta_train_data = []
for classifier in basic_classifiers:
    for row in basic_train_data:
        row.append(classifier.pred_proba())
'''

#create the meta classifier and train it
meta_train_data = scaler_for_meta.fit_transform(meta_train_data)
model = AdaBoostClassifier(n_estimators=300)
model.fit(meta_train_data,meta_class)
rect_test_data = scaler_for_meta.transform(rect_test_data)

#train each basic classifier
for classifier in basic_classifiers:
    classifier.fit(basic_train_data,basic_class)

scale_factor = 1.1

```

```

min_neighbors = 1
min_size = (20,20)
#prediction = model.predict(rect_test_data)

i=1
j = 1
k = 0

#for each image run it through the OpenCV detector then take the remaining windows and use
# the meta-classifier to attain the final classification.
for im in test_images:
    im_copy = im.copy()
    rips = rip_current_openCV_model.detectMultiScale(im_copy,scaleFactor =
scale_factor,minNeighbors=min_neighbors,minSize=min_size)
    for (x,y,w,h) in rips:
        feature_vector = []
        cropped_rip = im[y:y+h, x:x+w]
        resized_image = cv2.resize(cropped_rip,(24,24),interpolation= cv2.INTER_CUBIC)
        gray_image = cv2.cvtColor(resized_image,cv2.COLOR_BGR2GRAY)
        cv2.imwrite('testing_rips/results/rect/'+str(j)+'.png',gray_image)
        integral = calculate_integral(np.array(gray_image,dtype='int64'))

        #apply each feature to the window
        for feature in haar_feature_data:
            feature_vector.append(getFeatureValue(integral,feature[0],int(feature[1]),int(feature[2]),i
nt(feature[3]),int(feature[4])))
        feature_vector = np.reshape(feature_vector,(1,-1))
        scaled_vector = scaler_for_basic.transform(feature_vector)

        #attain the probability for each classifier and append it to the feature vector
        for classifier in basic_classifiers:
            result = classifier.predict_proba(scaled_vector)
            feature_vector = np.c_[feature_vector,result[0][1]]

        feature_vector = scaler_for_meta.transform(feature_vector)

        #classify the final vector using the meta classifier
        prediction = model.predict(feature_vector)

        #only keep the windows classified as a rip current
        if prediction[0] == 1:
            im_copy = cv2.rectangle(im_copy,(x,y),(x+w,y+h),(50,50,200),2)

        j += 1
        k += 1
cv2.imwrite('testing_rips/results/meta_part_hand_neg/'+str(i)+'.png',im_copy)

```

$i += 1$

## **Vita**

The author was born in Mandeville, Louisiana. He acquired his bachelor's degree from Southeastern Louisiana University in 2013. He was first employed as a research assistant at UNO while working on his thesis. He was later employed at Naval Research Labs for the remainder of his thesis as a computer science trainee.