Contents

1 adaboost.py

```python
"""
===================================================
     Introduction to Machine Learning (67577)
===================================================

Skeleton for the AdaBoost classifier.

Author: Gad Zalcberg
Date: February, 2019

"""
import numpy as np
import garcon as gc


class AdaBoost(object):

    def __init__(self, WL, T):
        """
        Parameters
        ----------
        WL : the class of the base weak learner
        T : the number of base learners to learn
        """
        self.WL = WL
        self.T = T
        self.h = [None] * T  # list of base learners
        self.w = np.zeros(T)  # weights

    def train(self, X, y):
        """
        Parameters
        ----------
        X : samples, shape=(num_samples, num_features)
        y : labels, shape=(num_samples, )
        Train this classifier over the sample (X,y)
        After finish the training return the weights of the samples in the last iteration.
        """
        n_samples = X.shape[0]
        D = np.array([1.0 / n_samples] * n_samples)
        # D = np.array([[1.0 / m] * m] * self.T)
        for t in range(self.T):
            gc.log(f'At T = {t}')
            self.h[t] = self.WL(D, X, y)
            y_hat = self.h[t].predict(X)
            mask = y != y_hat
            epsilon = np.matmul(D, mask)
            self.w[t] = 1 / 2 * np.log((1 - epsilon) / epsilon)
            D *= np.exp(-(y * y_hat * self.w[t]))
            D /= np.sum(D)
        # TODO complete this function

    def predict(self, X, max_t):
        """
        Parameters
        ----------
        X : samples, shape=(num_samples, num_features)
        :param max_t: integer < self.T: the number of classifiers to use for the classification
        :return: y_hat : a prediction vector for X. shape=(num_samples)
```

```python
        Predict only with max_t weak learners,
        """
        predictions = np.array([self.h[t].predict(X) for t in range(max_t)])
        multed = np.matmul(self.w[:max_t], predictions)
        signed = np.sign(multed)
        return signed
        # TODO complete this function

    def error(self, X, y, max_t):
        """
        Parameters
        ----------
        X : samples, shape=(num_samples, num_features)
        y : labels, shape=(num_samples)
        :param max_t: integer < self.T: the number of classifiers to use for the classification
        :return: error : the ratio of the correct predictions when predict only with max_t weak learners (float)
        """
        y_hat = self.predict(X, max_t)
        n_wrong = np.sum(y_hat != y)
        return n_wrong / X.shape[0]

        # TODO complete this function
```

2 comparer.py

```python
import os

import garcon as gc
import numpy as np
import pandas as pd
import sklearn.svm as svm
import perceptron as pc
import matplotlib.pyplot as plt

DIM = 2

FIG_DIR1 = './'
FIG_DIR2 = './'


class Comparer:
    def __init__(self):
        gc.log("Creating comparer")
        self._perc = pc.Perceptron()
        self._svm = svm.SVC(C=1e10, kernel='linear')
        self._mu = np.zeros([DIM])
        self._sig = np.eye(DIM)

    def draw_m_points(self, m):
        return np.random.multivariate_normal(mean=self._mu, cov=self._sig,
                                             size=m).T

    def plot_to_file(self, fn, dirnum=1):
        plt.savefig((FIG_DIR1 if dirnum == 1 else FIG_DIR2) + fn)

    def true_label(self, X):
        true_w = np.array([[0.3], [-0.5]])
        true_b = 0.1
        real_val = np.matmul(X, true_w) + true_b
        return np.sign(real_val)

    def draw_svm_hyp(self, X, y):
        classifier = self._svm.fit(X.T, np.ravel(y))
        w = classifier.coef_[0]
        a = -w[0] / w[1]
        xx = np.linspace(-2.5, 2.5)
        yy = a * xx - (classifier.intercept_[0] / w[1])
        plt.plot(xx, yy, label='SVM', color='red')

    def get_svm_accu(self, svm, X, y):
        w = svm.coef_[0]
        val = np.matmul(X, w) + svm.intercept_[0]
        labeled = np.sign(val)
        return np.sum(labeled == np.ravel(y.T)) / X.shape[0]

    def get_perc_accu(self, perc_w, X, y):
        X_1 = np.c_[X, np.ones(X.shape[0])]
        val = np.matmul(X_1, perc_w)
        labeled = np.sign(val)
        return np.sum(labeled == np.ravel(y.T)) / X.shape[0]

    def draw_perc_hyp(self, X, y):
        w = self._perc.fit(X.T, y)
        a = -w[0] / w[1]
```

```
        xx = np.linspace(-2.5, 2.5)
        yy = a * xx - (w[-1] / w[1])
        plt.plot(xx, yy, label='Perceptron', color='green')

    def draw_true_hyp(self):
        w = np.array([[0.3], [-0.5], [0.1]])
        a = -w[0] / w[1]
        xx = np.linspace(-2.5, 2.5)
        yy = a * xx - (w[-1] / w[1])
        plt.plot(xx, yy, label='Real', color='black')

    def init_plot(self, m):
        fig = plt.figure()
        fig.suptitle("SVM vs Perceptron, " + str(m) + " Samples")
        plt.xlabel('x Coordinate')
        plt.ylabel('y Coordinate')

    def compare_one(self, m):
        self.init_plot(m)
        points = self.draw_m_points(m)
        raw_labels = self.true_label(points.T)
        labels = np.hstack((points.T, raw_labels))
        good_points = labels[labels[:, 2] == 1]
        bad_points = labels[labels[:, 2] != 1]
        plt.scatter(good_points[:, 0], good_points[:, 1], label='True',
                    marker='x')
        plt.scatter(bad_points[:, 0], bad_points[:, 1], label='False',
                    marker='x')
        self.draw_svm_hyp(points, raw_labels)
        self.draw_perc_hyp(points, raw_labels)
        self.draw_true_hyp()
        plt.legend()
        self.plot_to_file('svm_vs_perc_' + str(m))

    def compare_many(self):
        gc.log("Comparing many")
        for m in [5, 10, 15, 25, 70]:
            self.compare_one(m)

    def big_test(self):
        gc.log("Big test")
        fig = plt.figure()
        fig.suptitle("SVM vs Perceptron, Accuracy Test")
        plt.xlabel('Train Set Size')
        plt.ylabel('Accuracy (%)')

        k, n_iter = 10000, 500
        M, accurs = [5, 10, 15, 25, 70], [[],[]]
        for m in M:
            svm_accu_sum = 0
            perc_accu_sum = 0
            for i in range(n_iter):
                while True:
                    train_X = self.draw_m_points(m).T
                    train_y = self.true_label(train_X)
                    if np.unique(train_y).shape[0] == 2:
                        break
                while True:
                    test_X = self.draw_m_points(k).T
                    test_y = self.true_label(test_X)
                    if np.unique(test_y).shape[0] == 2:
                        break
                svm = self._svm.fit(train_X, np.ravel(train_y.T))
                perc_w = self._perc.fit(train_X, train_y)
                svm_accu = self.get_svm_accu(svm, test_X, test_y)
                perc_accu = self.get_perc_accu(perc_w, test_X, test_y)
                svm_accu_sum += svm_accu
                perc_accu_sum += perc_accu
```

```
            svm_accu_avg = svm_accu_sum / n_iter
            perc_accu_avg = perc_accu_sum / n_iter
            accurs[0].append(svm_accu_avg)
            accurs[1].append(perc_accu_avg)
        plt.plot(M, accurs[0], label = 'SVM')
        plt.plot(M, accurs[1], label='Perceptron')
        plt.legend()
        self.plot_to_file('q5',2)
```

3 ex4 runme.py

```python
"""
===================================================
     Introduction to Machine Learning (67577)
===================================================

Running script for Ex4.

Author: Gad Zalcberg
Date: February, 2019

"""
FIG_DIR3 = './'

import garcon as gc
import time

import numpy as np
from ex4_tools import DecisionStump, decision_boundaries, generate_data, \
    load_images
import matplotlib.pyplot as plt
from adaboost import AdaBoost
import comparer as cmp
from face_detection import integral_image, WeakImageClassifier


def Q4():
    comp = cmp.Comparer()
    comp.compare_many()
    'TODO complete this function'


def Q5():
    comp = cmp.Comparer()
    comp.big_test()
    'TODO complete this function'


def Q8(noise=0.0):
    n_samples_train, n_samples_test, T = 5000, 200, 500
    train_X, train_y = generate_data(n_samples_train, noise)
    test_X, test_y = generate_data(n_samples_test, noise)
    WL = DecisionStump
    ada = AdaBoost(WL, T)
    ada.train(train_X, train_y)
    T_range = np.arange(1, T)
    train_errs = [ada.error(train_X, train_y, t) for t in T_range]
    test_errs = [ada.error(test_X, test_y, t) for t in T_range]

    fig = plt.figure()
    fig.suptitle("Train vs Test error, Adaboost")
    plt.xlabel('# of Hypotheses (T)')
    plt.ylabel('Error rate (%)')
    plt.plot(T_range, train_errs, label='Train Error')
    plt.plot(T_range, test_errs, label='Test Error')
    # plt.ylim(top=0.06)
    plt.legend()
    plt.savefig(FIG_DIR3 + 'q8' + ('' if noise == 0 else '_' + str(
            noise).replace('.', '_')))

```

```python
    return ada, test_X, test_y, train_X, train_y
    'TODO complete this function'


def Q9(ada, test_X, test_y, noise=0.0):
    # f, axs = plt.subplots(3,2)
    n_classifiers = [5, 10, 50, 100, 200, 500]
    fig = plt.figure()
    fig.suptitle('Decision of the Learned Classifiers')
    for i in range(6):
        plt.subplot(3, 2, i + 1)
        decision_boundaries(ada, test_X, test_y, n_classifiers[i])
    plt.savefig(FIG_DIR3 + 'q9' + ('' if noise == 0 else '_' + str(
            noise).replace('.', '_')))

    'TODO complete this function'


def Q10(ada, train_X, train_y, T_hat=500, noise=0.0):
    fig = plt.figure()
    fig.suptitle('Decision of T-hat')
    decision_boundaries(ada, train_X, train_y, T_hat)
    plt.savefig(FIG_DIR3 + 'q10' + ('' if noise == 0 else '_' + str(
            noise).replace('.', '_')))
    'TODO complete this function'


def Q11():
    'TODO complete this function'


def Q12():
    for noise in [0.01, 0.4]:
        T_hat = 110 if noise==0.01 else 210
        ada, test_X, test_y, train_X, train_y= Q8(noise)
        Q9(ada, test_X, test_y, noise)
        Q10(ada, train_X, train_y, T_hat,noise)
    'TODO complete this function'


def Q17():
    train_images, test_images, train_labels, test_labels = load_images(
            '../Docs/')
    train_images = integral_image(train_images)
    test_images = integral_image(test_images)
    WL, T = WeakImageClassifier, 50
    ada = AdaBoost(WL, T)
    ada.train(train_images, train_labels)
    T_range = np.arange(1, T)
    train_errs = [ada.error(train_images, train_labels, t) for t in T_range]
    test_errs = [ada.error(test_images, test_labels, t) for t in T_range]

    fig = plt.figure()
    fig.suptitle("Train vs Test error, Face Classifier")
    plt.xlabel('# of Hypotheses (T)')
    plt.ylabel('Error rate (%)')
    plt.plot(T_range, train_errs, label='Train Error')
    plt.plot(T_range, test_errs, label='Test Error')
    # plt.ylim(top=0.06)
    plt.legend()
    plt.savefig(FIG_DIR3 + 'q17')
    'TODO complete this function'


def Q18():
    'TODO complete this function'
```

```python
if __name__ == '__main__':
    start_time = time.time()
    Q4()
    Q5()
    learner, test_X, test_y, train_X, train_y = Q8()
    Q9(learner, test_X, test_y)
    Q10(learner, train_X, train_y)
    Q12()
    Q17()
    gc.log('Execution took %s seconds' % (time.time() - start_time))
    'TODO complete this function'
```

4 ex4 tools.py

```python
"""
===================================================
    Introduction to Machine Learning (67577)
===================================================

This module provides some useful tools for Ex4.

Author: Gad Zalcberg
Date: February, 2019

"""
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from itertools import product
from matplotlib.pyplot import imread
import os
from sklearn.model_selection import train_test_split


def find_threshold(D, X, y, sign, j):
    """
    Finds the best threshold.
    D =  distribution
    S = (X, y) the data
    """
    # sort the data so that x1 <= x2 <= ... <= xm
    sort_idx = np.argsort(X[:, j])
    X, y, D = X[sort_idx], y[sort_idx], D[sort_idx]

    thetas = np.concatenate([[-np.inf], (X[1:, j] + X[:-1, j]) / 2, [np.inf]])
    minimal_theta_loss = np.sum(D[y == sign]) #loss of the smallest possible
    # theta
    losses = np.append(minimal_theta_loss, minimal_theta_loss - np.cumsum(D * (y * sign)))
    min_loss_idx = np.argmin(losses)
    return losses[min_loss_idx], thetas[min_loss_idx]


class DecisionStump(object):
    """
    Decision stump classifier for 2D samples
    """
    def __init__(self, D, X, y):
        self.theta = 0
        self.j = 0
        self.sign = 0
        self.train(D, X, y)

    def train(self, D, X, y):
        """
        Train the classifier over the sample (X,y) w.r.t. the weights D over X
        Parameters
        ----------
        D : weights over the sample
        X : samples, shape=(num_samples, num_features)
        y : labels, shape=(num_samples)
        """
        loss_star, theta_star = np.inf, np.inf
        for sign, j in product([-1, 1], range(X.shape[1])):
```

```
                loss, theta = find_threshold(D, X, y, sign, j)
                if loss < loss_star:
                    self.sign, self.theta, self.j = sign, theta, j
                    loss_star = loss

    def predict(self, X):
        """
        Parameters
        ----------
        X : shape=(num_samples, num_features)
        Returns
        -------
        y_hat : a prediction vector for X shape=(num_samples)
        """

        y_hat = self.sign * ((X[:, self.j] <= self.theta) * 2 - 1)
        return y_hat


def decision_boundaries(classifier, X, y, num_classifiers=1, weights=None):
    """
    Plot the decision boundaries of a binary classfiers over X \subseteq R^2

    Parameters
    ----------
    classifier : a binary classifier, implements classifier.predict(X)
    X : samples, shape=(num_samples, 2)
    y : labels, shape=(num_samples)
    title_str : optional title
    weights : weights for plotting X
    """
    cm = ListedColormap(['#AAAAFF','#FFAAAA'])
    cm_bright = ListedColormap(['#0000FF','#FF0000'])
    h = .003  # step size in the mesh
    # Plot the decision boundary.
    x_min, x_max = X[:, 0].min() - .2, X[:, 0].max() + .2
    y_min, y_max = X[:, 1].min() - .2, X[:, 1].max() + .2
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = classifier.predict(np.c_[xx.ravel(), yy.ravel()], num_classifiers)
    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.pcolormesh(xx, yy, Z, cmap=cm)
    # Plot also the training points
    if weights is not None:
        plt.scatter(X[:, 0], X[:, 1], c=y, s=weights, cmap=cm_bright)
    else:
        plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cm_bright)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.xticks([])
    plt.yticks([])
    plt.title(f'num classifiers = {num_classifiers}')
    plt.draw()


def generate_data(num_samples, noise_ratio):
    '''
    generate samples X with shape: (num_samples, 2) and labels y with shape (num_samples).
    num_samples: the number of samples to generate
    noise_ratio: invert the label for this ratio of the samples
    '''
    X = np.random.rand(num_samples, 2) * 2 - 1
    radius = 0.5 ** 2
    in_circle = np.sum(X ** 2, axis=1) < radius
    y = np.ones(num_samples)
    y[in_circle] = -1
    y[np.random.choice(num_samples, int(noise_ratio * num_samples))] *= -1

```

```
    return X, y


def load_images(path_to_data):

    images = []
    labels = []
    for filename in os.listdir(path_to_data + 'faces/FACES'):
        images.append(imread(os.path.join(path_to_data + 'faces/FACES', filename)))
        labels.append(1)
    for filename in os.listdir(path_to_data + 'faces/NFACES'):
        images.append(imread(os.path.join(path_to_data + 'faces/NFACES', filename)))
        labels.append(-1)
    return train_test_split(np.array(images, dtype=np.float64), np.array(labels, dtype=np.float64), test_size=0.33)
```

5 face detection.py

```python
"""
=====================================================
      Introduction to Machine Learning (67577)
=====================================================

Skeleton the weak image classifier.

Author: Gad Zalcberg
Date: February, 2019

"""
import numpy as np
import garcon as gc
import matplotlib.pyplot as plt
from ex4_tools import find_threshold


def S(integrals, a, b):
    '''
    Compute the integral value of the (a,b) cell in images.
    :param integrals: integrals of some image, shape=(num_samples,
    image_height, image_width)
    :param a: row idx to calculate, shape=(0)
    :param b: col idx to calculate, shape=(0)
    :return: A vector of the integral value of (a,b), shape=(num_samples,0)
    '''
    if a >= 0 and b >= 0:
        if a < integrals.shape[1] and b < integrals.shape[2]:
            return integrals[:, a, b]
    return 0.0


def integral_image(images):
    '''
    compute the integral of the images
    :param images: numpy array of images, shape=(num_samples, image_height, image_width)
    :return: numpy array of the integrals of the input, the same shape
    '''
    integral = np.zeros(images.shape)
    for a in range(integral.shape[1]):
        for b in range(integral.shape[2]):
            sum = images[:, a, b]
            sum += S(integral, a - 1, b)
            sum += S(integral, a, b - 1)
            sum -= S(integral, a - 1, b - 1)
            integral[:, a, b] = sum

    return integral
    # TODO complete this function


def sum_square(integrals, up, left, height, width):
    '''
    compute the sum of the pixels in the square between the upper left pixel (up, left)
    and down right pixel (up + height - 1, left + width - 1). include the corners in the square.
    :param integrals: the integrals of the images, shape=(num_samples, image_height, image_width)
    :param up: the up limit of the square
    :param left: the left limit of the square
    :param height: the height of the square
```

```python
    :param width: the width of the square
    :return: the sum of the pixels in the square (int)
    '''
    dr_a, dr_b = up + height - 1, left + width - 1
    dl_a, dl_b = up + height - 1, left
    ur_a, ur_b = up, left + width - 1
    ul_a, ul_b = up, left

    sum = S(integrals, dr_a, dr_b)
    sum -= S(integrals, ur_a - 1, ur_b)
    sum -= S(integrals, dl_a, dl_b - 1)
    sum += S(integrals, ul_a - 1, ul_b - 1)

    return sum
    # TODO complete this function


class WeakImageClassifier:

    def __init__(self, sample_weight, integrals, labels):
        """
        Train the classifier over the sample (integrals,labels) w.r.t. the weights sample_weight over integrals

        Parameters
        ----------
        sample_weight : weights over the sample numpy array, shape=(num_samples)
        integrals: numpy array shape=(num_samples, image_height, image_width), the samples
        labels: numpy array shape=(num_samples)
        """
        _, self.rows, self.cols = integrals.shape
        self.up = 0
        self.height = 0
        self.left = 0
        self.width = 0
        self.theta = np.inf
        self.loss = np.inf
        self.sign = 0
        self.kernel = None
        self.train(integrals, labels, sample_weight)

    def kernel_a(self, integrals, up, left, height, width):
        '''
        calculate the value of Haar feature of type A. the white part is located between the upper left pixel (up, left)
        and down right pixel (up + height - 1, left + width - 1). the black part located in its right side.
        :param integrals: the integrals of the images, shape=(num_samples, image_height, image_width)
        :param up: the up limit of the square
        :param left: the left limit of the square
        :param height: the height of the square
        :param width: the width of the square
        :return: the values of the Haar feature for each pixel- numpy array shape=(num_samples)
        '''
        sum_left_square = sum_square(integrals, up, left, height, width)
        sum_right_square = sum_square(integrals, up, left + width, height,
                                      width)
        feature_values = sum_left_square - sum_right_square
        return feature_values

    def kernel_b(self, integrals, up, left, height, width):
        '''
        calculate the value of Haar feature of type B. the white part is located between the upper left pixel (up, left)
        and down right pixel (up + height - 1, left + width - 1). the black part located right below.
        :param integrals: the integrals of the images, shape=(num_samples, image_height, image_width)
        :param up: the up limit of the square
        :param left: the left limit of the square
        :param height: the height of the square
        :param width: the width of the square
        :return: the values of the Haar feature for each pixel- numpy array shape=(num_samples)
        '''
```

```python
        sum_upper_rec = sum_square(integrals, up, left, height, width)
        sum_lower_rec = sum_square(integrals, up + height, left, height, width)
        feature_values = sum_upper_rec - sum_lower_rec
        return feature_values
        # TODO complete this function

    def kernel_c(self, integrals, up, left, height, width):
        '''
        calculate the value of Haar feature of type C. the first white part is located between the upper left pixel
        (up, left) and down right pixel (up + height - 1, left + width - 1). the black part located in its right side
        and the second right part located in the blacks part side.
        :param integrals: the integrals of the images, shape=(num_samples, image_height, image_width)
        :param up: the up limit of the square
        :param left: the left limit of the square
        :param height: the height of the square
        :param width: the width of the square
        :return: the values of the Haar feature for each pixel- numpy array shape=(num_samples)
        '''
        sum_left_rec = sum_square(integrals, up, left, height, width)
        sum_mid_rec = sum_square(integrals, up, left + width, height, width)
        sum_right_rec = sum_square(integrals, up, left + 2 * width, height,
                                   width)

        feature_values = sum_left_rec - sum_mid_rec + sum_right_rec
        return feature_values
        # TODO complete this function

    def kernel_d(self, integrals, up, left, height, width):
        '''
        calculate the value of Haar feature of type C. the first white part is located between the upper left pixel
        (up, left) and down right pixel (up + height - 1, left + width - 1). the first black parts located in its right
        side, and in it's bottom. the second white part located in its bottom right
        :param integrals: the integrals of the images, shape=(num_samples, image_height, image_width)
        :param up: the up limit of the square
        :param left: the left limit of the square
        :param height: the height of the square
        :param width: the width of the square
        :return: the values of the Haar feature for each pixel- numpy array shape=(num_samples)
        '''
        sum_topleft_square = sum_square(integrals, up, left, height, width)
        sum_topright_square = sum_square(integrals, up, left + width, height,
                                         width)
        sum_bottleft_square = sum_square(integrals, up + height, left, height,
                                         width)
        sum_bottright_square = sum_square(integrals, up + height, left + width,
                                          height, width)
        feature_values = sum_topleft_square - sum_topright_square \
                         - sum_bottleft_square + sum_bottright_square
        return feature_values
        # TODO complete this function

    def evaluate_kernel(self, integrals, labels, kernel, up, left, height,
                        width, weights):
        '''
        Get the feature values according to the following parameters. Try the hypothesis of threshold classifier over
        the feature values. the threshold can be either of type > or of type <.
        :param integrals: the integrals of the images, shape=(num_samples, image_height, image_width)
        :param labels: the labels of the samples shape=(num_samples)
        :param kernel: An Haar feature function of the type {a,b,c,d}.
        :param up: the up limit of the square
        :param left: the left limit of the square
        :param height: the height of the square
        :param width: the width of the square
        :param weights: the current weight of the samples shape=(num_samples)
        '''
        feature_values = kernel(integrals, up, left, height, width)
        self.evaluate_feature_performance(kernel, feature_values, weights,
                                          labels, up, height, left, width, 1)
```

```python
        self.evaluate_feature_performance(kernel, feature_values, weights,
                                          labels, up, height, left, width, -1)

        # TODO complete this function

    def evaluate_all_kernel_types(self, integrals, weights, labels, up, left,
                                  height, width):
        '''
        For each of the {a,b,c,d} kernel functions, if the following parameters are legal, Try the hypothesis of
        threshold classifier over the feature values.
        :param integrals: the integrals of the images, shape=(num_samples, image_height, image_width)
        :param weights: the current weight of the samples, shape=(num_samples)
        :param labels: the labels of the samples, shape=(num_samples)
        :param up: the up limit of the square
        :param left: the left limit of the square
        :param height: the height of the square
        :param width: the width of the square
        '''
        if up + height <= self.rows and left + 2 * width <= self.cols:
            self.evaluate_kernel(integrals, labels, self.kernel_a, up, left,
                                 height, width, weights)

        if up + 2 * height <= self.rows and left + width <= self.cols:
            self.evaluate_kernel(integrals, labels, self.kernel_b, up, left,
                                 height, width, weights)

        if up + height <= self.rows and left + 3 * width <= self.cols:
            self.evaluate_kernel(integrals, labels, self.kernel_c, up, left,
                                 height, width, weights)

        if up + 2 * height <= self.rows and left + 2 * width <= self.cols:
            self.evaluate_kernel(integrals, labels, self.kernel_d, up, left,
                                 height, width, weights)

    def evaluate_feature_performance(self, kernel, feature_values, weights,
                                     labels, up, height, left, width, sign):
        '''
        find the best decision stump hypothesis for given feature value, and update parameters accordingly.
        For given feature values and labels find the ERM for the threshold problem, if the loss value according some
        theta is lower than self.loss update the parameters of self to the parameters of the function and update theta
        to the best theta.
        :param kernel: function- 'kernel_k' (where k in {a,b,c,d})
        :param feature_values: the feature value of the image according to the kernel configured by the following parameters
        :param weights: the of of the samples, shape=(num_samples)
        :param labels: the labels of the data, shape=(num_samples)
        :param up: the up limit of the square
        :param height: the height of the square
        :param left: the left limit of the square
        :param width: the width of the square
        :param sign: whether the upper-left square of the kernel is white or black (equivalent to multiply the feature
        by 1 or -1.
        '''
        # TODO: Possibly reshape
        loss, theta = find_threshold(weights, feature_values.reshape((-1, 1)),
                                     labels,
                                     sign, 0)
        if loss < self.loss:
            self.up = up
            self.height = height
            self.left = left
            self.width = width
            self.theta = theta
            self.loss = loss
            self.sign = sign
            self.kernel = kernel

        # TODO complete this function
```

```
    def train(self, integrals, labels, sample_weight):
        '''
        This function iterate over all possible Haar features (of the 4 types we defined) and find the best hypothesis
        for the current distribution (ERM)
        :param integrals: the integrals of the images in the dataset, shape=(num_samples, image_height, image_width)
        :param labels: the labels of the samples in the dataset, shape=(num_samples)
        :param sample_weight: the current weights of the samples. shape=(num_samples)
        '''
        num_samples, self.rows, self.cols = integrals.shape
        for up in range(self.rows):
            for height in range(1, self.rows + 1):
                for left in range(self.cols):
                    for width in range(1, self.cols + 1):
                        self.evaluate_all_kernel_types(integrals, sample_weight,
                                                       labels, up, left, height,
                                                       width)
        plt.imshow(self.visualize_kernel())
        plt.show()

    def predict(self, integrals):
        '''
        predict labels (whether the image contain face or not) for the images according to their integrals.
        :param integrals: the integrals of the images we want to predict.
        :return: labels of the images
        '''
        feature_values = self.kernel(integrals, self.up, self.left, self.height,
                                     self.width)
        y_hat = self.sign * ((feature_values <= self.theta) * 2 - 1)
        return y_hat
        # TODO complete this function

    def visualize_kernel(self):
        '''
        This function visualize the kernel.
        :return: image of the kernel
        '''
        image = np.zeros((self.rows, self.cols))
        if self.kernel == self.kernel_a:
            image[self.up: self.up + self.height,
            self.left: self.left + self.width] = 1
            image[self.up: self.up + self.height,
            self.left + self.width: self.left + self.width * 2] = -1
        if self.kernel == self.kernel_b:
            image[self.up: self.up + self.height,
            self.left: self.left + self.width] = 1
            image[self.up + self.height: self.up + self.height * 2,
            self.left: self.left + self.width] = -1
        if self.kernel == self.kernel_c:
            image[self.up: self.up + self.height,
            self.left: self.left + self.width] = 1
            image[self.up: self.up + self.height,
            self.left + self.width: self.left + self.width * 2] = -1
            image[self.up: self.up + self.height,
            self.left + self.width * 2: self.left + self.width * 3] = 1
        if self.kernel == self.kernel_d:
            image[self.up: self.up + self.height,
            self.left: self.left + self.width] = 1
            image[self.up: self.up + self.height,
            self.left + self.width: self.left + self.width * 2] = -1
            image[self.up + self.height: self.up + self.height * 2,
            self.left: self.left + self.width] = -1
            image[self.up + self.height: self.up + self.height * 2,
            self.left + self.width: self.left + self.width * 2] = 1
        return image * self.sign
```

6 garcon.py

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

def log(*args):
    print('Log: ', end='')
    for arg in args:
        print(arg, end='')
    print()
```

7 perceptron.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import garcon as gc


class Perceptron:
    def __init__(self):
        self._X_train = None
        self._y_train = None
        self._curr_w = None
        self._inner_vec = None
        self._signs = None

    def init_weights(self, size):
        self._curr_w = np.zeros([size, 1])

    def get_inner(self):
        self._inner_vec = np.matmul(self._X_train, self._curr_w)

    def get_signs(self):
        self._signs = np.sign(self._inner_vec).T
        # self._signs =  self._y_train.T* self._inner_vec

    def check_and_update(self):

        bad_idxs = np.where(self._signs[0] != self._y_train[0])[0]
        if bad_idxs.shape[0] == 0:
            return True
        else:
            # If there are bad indices, we should update and return false.
            some_idx = bad_idxs[0]
            self._curr_w += self._y_train[0][some_idx] * np.array([
                self._X_train[
                    some_idx]]).T
            return False

    def fit(self, X, y):
        '''
        :param X: shape: (n_samples, n_features)
        :param y: shape: (n_samples,1)
        :return:
        '''
        X_1 = np.c_[X, np.ones(X.shape[0])]
        w = np.zeros(X_1.shape[1])
        while True:
            signs = np.sign(np.matmul(X_1, w))
            comp_idxs = np.where(signs != np.ravel(y.T))[0]
            if comp_idxs.shape[0] == 0:
                return w
            w += y[comp_idxs[0], 0] * X_1[comp_idxs[0]]



    def predict(self, x):
        # The real result
        real_res = np.inner(x, self._curr_w)
        return np.sign(real_res)
```

IML (67577) - Exercise 4 - Boosting and SVM

Alon Emanuel - 205894058

May 24, 2019

SVM - Formulation

Q1

- We claim that the following QP problem's objective is equivalent to the Hard-SVM objective:

$$\underset{\mathbf{v}\in\mathbb{R}^n}{\operatorname{argmin}} \left[\frac{1}{2} \begin{bmatrix} | \\ \mathbf{w} \\ | \\ b \end{bmatrix}^T \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & \ddots & & \\ & & & 1 & \\ & & & & 0 \end{bmatrix} \begin{bmatrix} | \\ \mathbf{w} \\ | \\ b \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}^T \begin{bmatrix} | \\ \mathbf{w} \\ | \\ b \end{bmatrix} \right]$$

$$\text{s.t} \begin{bmatrix} - & -\mathbf{x}_1 & - & -1 \\ - & -\mathbf{x}_2 & - & -1 \\ & \vdots & & \\ - & -\mathbf{x}_m & - & -1 \end{bmatrix} \begin{bmatrix} | \\ \mathbf{w} \\ | \\ b \end{bmatrix} \le \begin{bmatrix} -1 \\ -1 \\ \vdots \\ -1 \end{bmatrix}$$

  - $Q = \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & \ddots & & \\ & & & 1 & \\ & & & & 0 \end{bmatrix}$, $\mathbf{v} = \begin{bmatrix} | \\ \mathbf{w} \\ | \\ b \end{bmatrix}$, $\mathbf{a} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$, $A = \begin{bmatrix} - & -\mathbf{x}_1 & - & -1 \\ - & -\mathbf{x}_2 & - & -1 \\ & \vdots & & \\ - & -\mathbf{x}_m & - & -1 \end{bmatrix}$, $\mathbf{d} = \begin{bmatrix} -1 \\ -1 \\ \vdots \\ -1 \end{bmatrix}$.

- **Proof:**
  - Let $\mathbf{w}^*$ and b^* be some optimal solutions for the original Hard-SVM problem.
  - Lets plug it into our new QP objective:

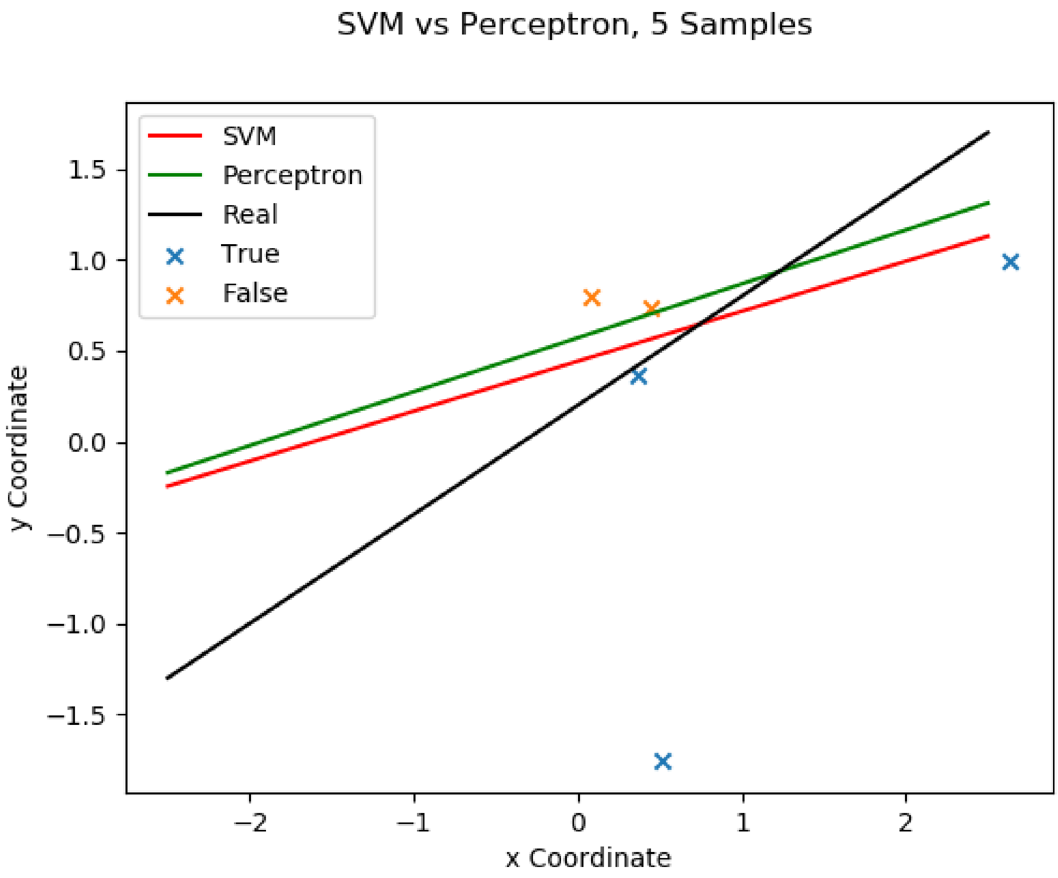
$$\frac{1}{2} \begin{bmatrix} | \\ \mathbf{w}^* \\ | \\ b^* \end{bmatrix}^T \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & \ddots & & \\ & & & 1 & \\ & & & & 0 \end{bmatrix} \begin{bmatrix} | \\ \mathbf{w}^* \\ | \\ b^* \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}^T \begin{bmatrix} | \\ \mathbf{w}^* \\ | \\ b^* \end{bmatrix} = \frac{1}{2} \begin{bmatrix} | \\ \mathbf{w}^* \\ | \\ b^* \end{bmatrix}^T \begin{bmatrix} | \\ \mathbf{w}^* \\ | \\ 0 \end{bmatrix} + 0$$
$$= \frac{1}{2} \|\mathbf{w}^*\|$$

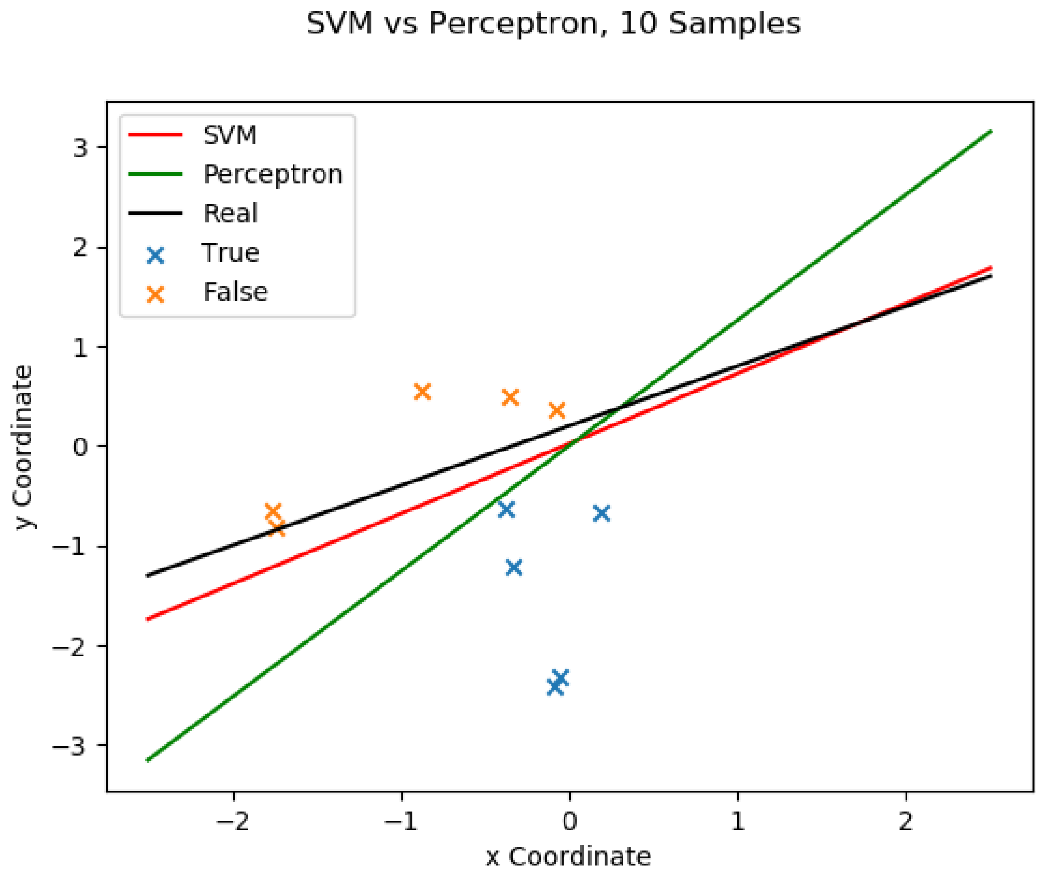
  - Since $\mathbf{w}^*$ optimizes $\|\mathbf{v}\|$, it also optimizes $\frac{1}{2}\|\mathbf{w}^*\|$.
  - Moreover, the restriction from the original objective can be rewritten linearly as we've done: $\begin{bmatrix} - & -\mathbf{x}_1 & - & -1 \\ - & -\mathbf{x}_2 & - & -1 \\ & \vdots & & \\ - & -\mathbf{x}_m & - & -1 \end{bmatrix} \begin{bmatrix} | \\ \mathbf{w} \\ | \\ b \end{bmatrix}$ $\begin{bmatrix} -1 \\ -1 \\ \vdots \\ -1 \end{bmatrix}$.

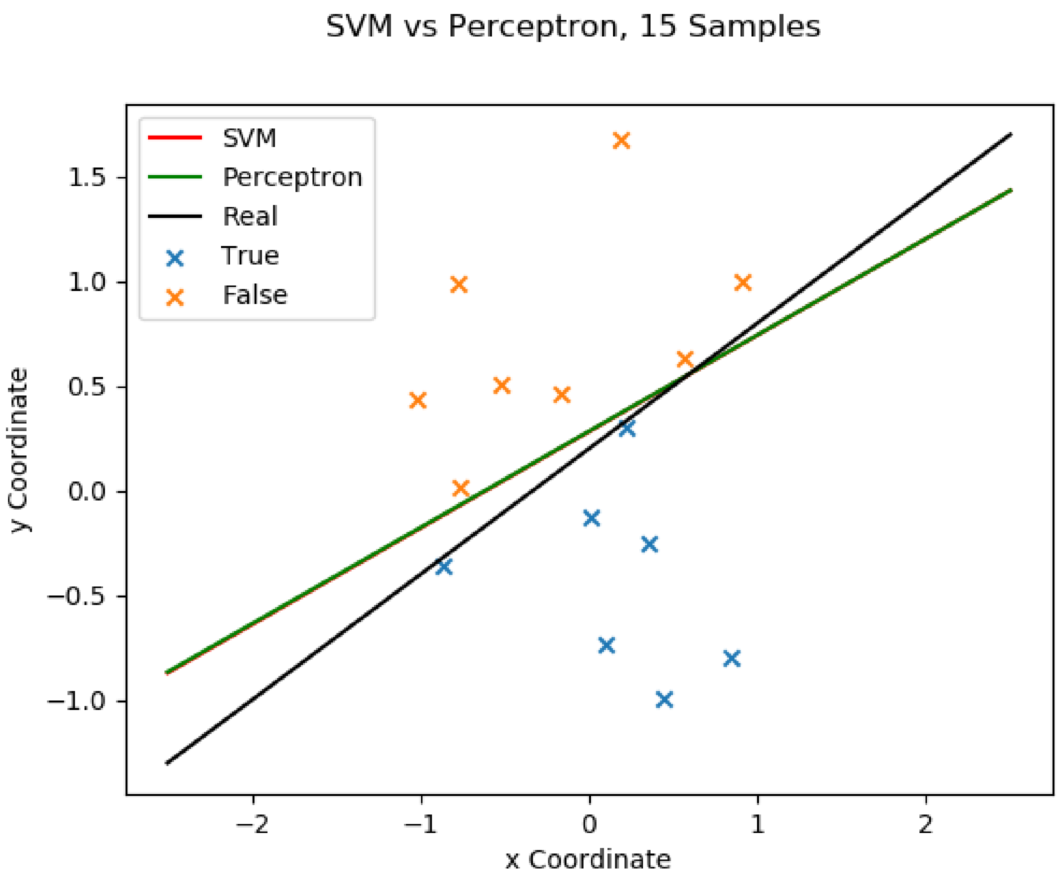
■

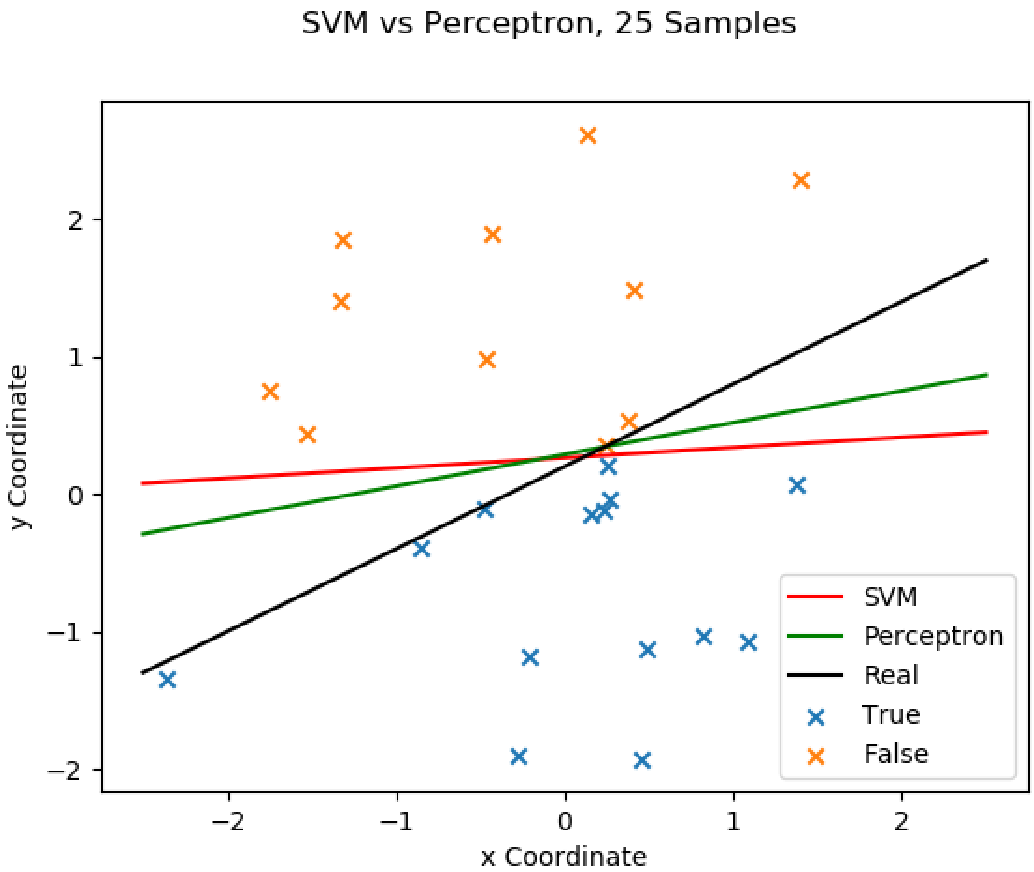
SVM and Generalization

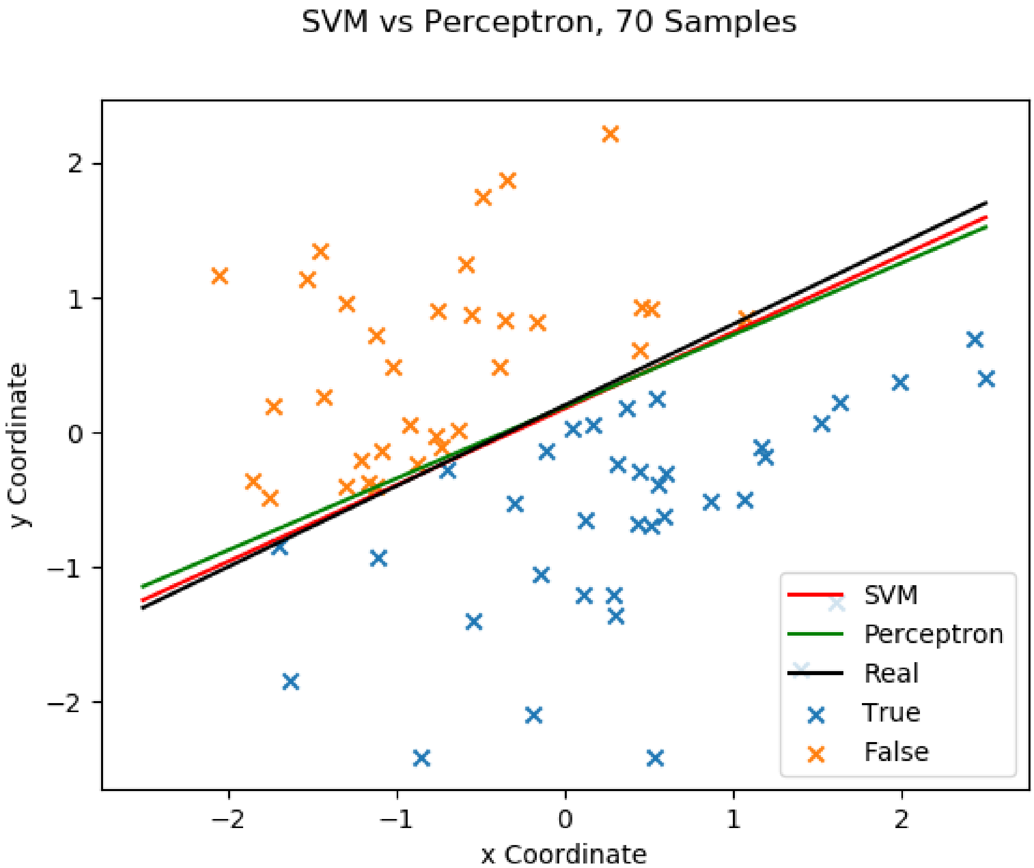
Q4




SVM vs Perceptron, 10 Samples
SVM
Perceptron
Real
True
False
y Coordinate
x Coordinate
3
2
1
0
−1
−2
−3
−2
−1
0
1
2


SVM vs Perceptron, 15 Samples
SVM
Perceptron
Real
True
False
y Coordinate
x Coordinate
1.5
1.0
0.5
0.0
−0.5
−1.0
−2
−1
0
1
2


SVM vs Perceptron, 25 Samples
SVM
Perceptron
Real
True
False
x Coordinate
y Coordinate


SVM vs Perceptron, 70 Samples
SVM
Perceptron
Real
True
False
x Coordinate
y Coordinate

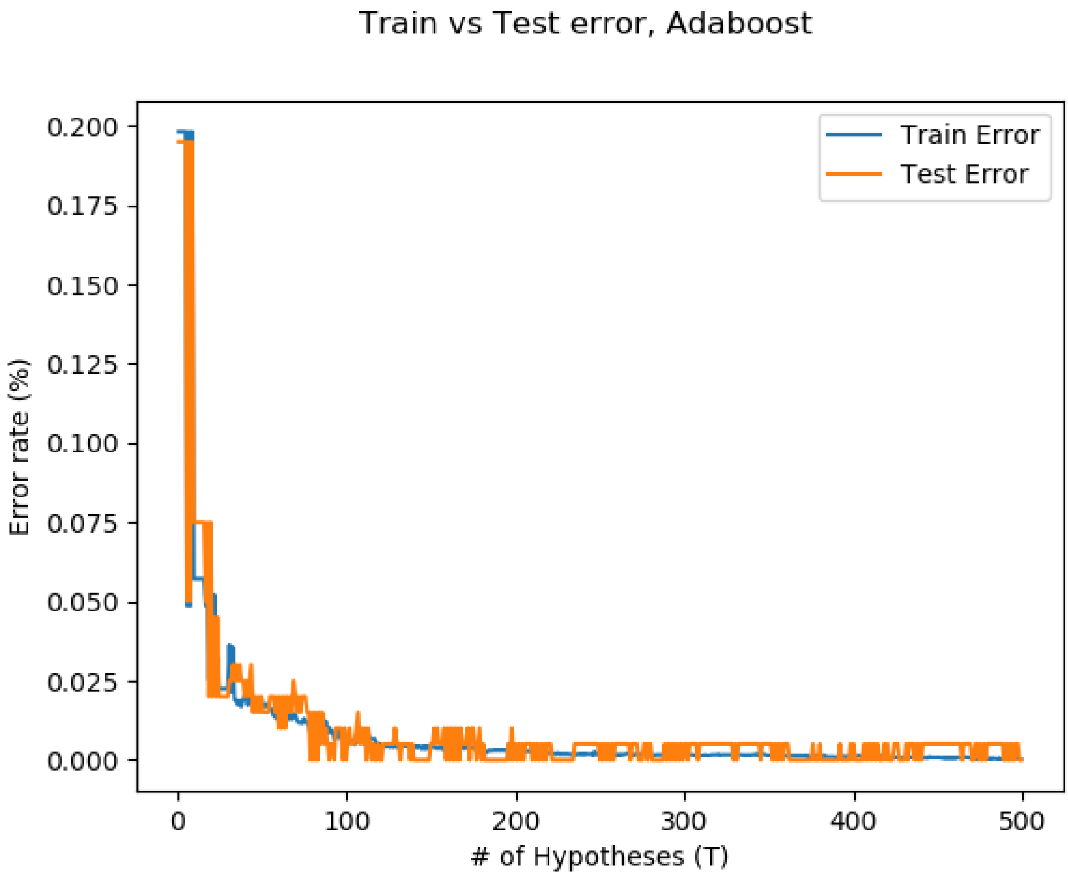
Q5+6

- In the following graph we can see that the SVM did better than the Perceptron.
- This is mostly due to the fact that the SVM finds the separating line which has the largest margin, while the perceptron finds any separating line (the first one it finds).
- A bigger margin translates into a better generalizing line, hence the results.

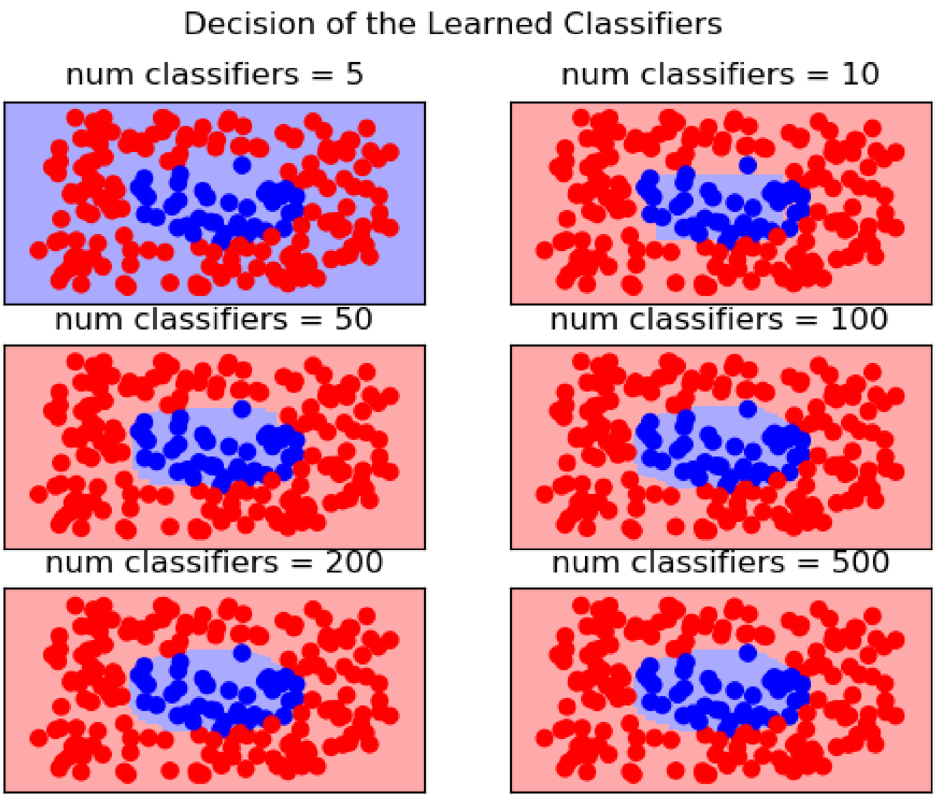
Separate the Inseparable - Adaboost

Q8

- The following is the graph of the error as a function of T, for both the train set and the test set.

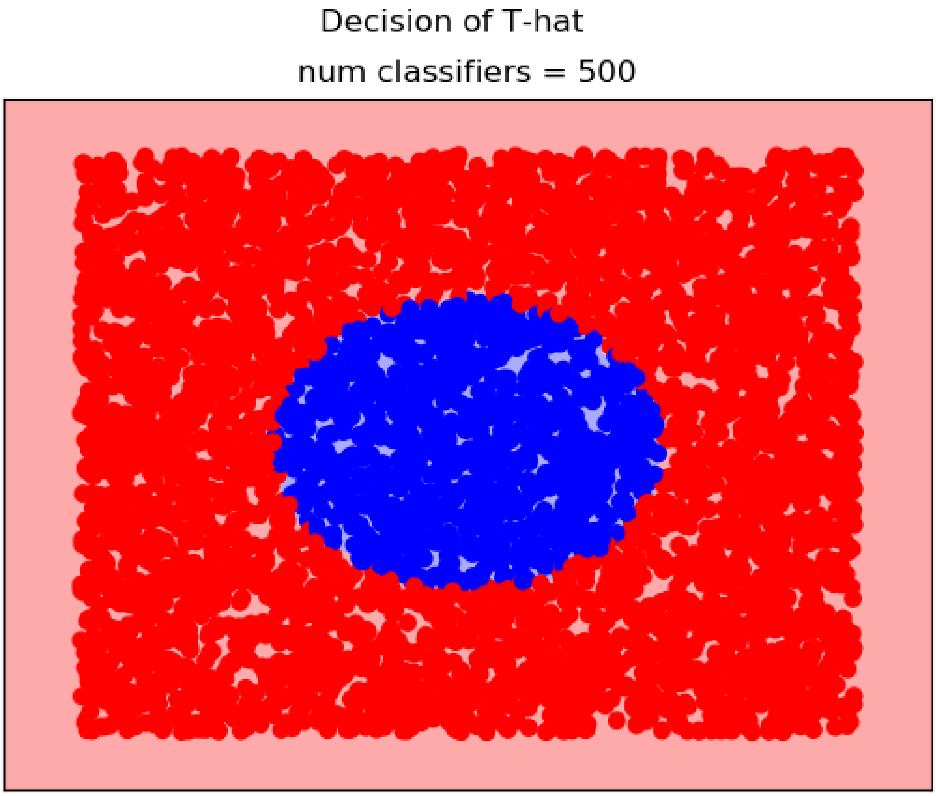

Train vs Test error, Adaboost
Train Error
Test Error
Error rate (%)
# of Hypotheses (T)
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0
100
200
300
400
500

Q9


Decision of the Learned Classifiers
num classifiers = 5
num classifiers = 10
num classifiers = 50
num classifiers = 100
num classifiers = 200
num classifiers = 500

Q10

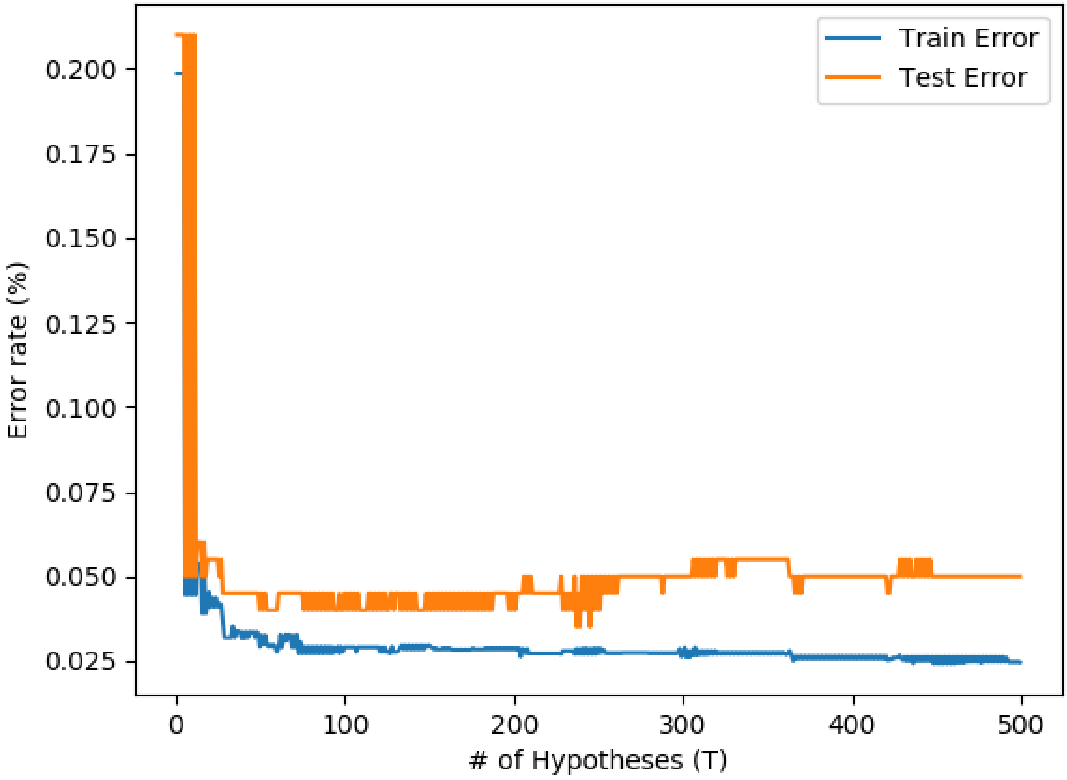
- As we can deduce from the graph in Q8, we see that $\hat{T}$ is equal to the largest T we took, which is 500.
- Its test error stabilizes at ~ 0.017.



Q12

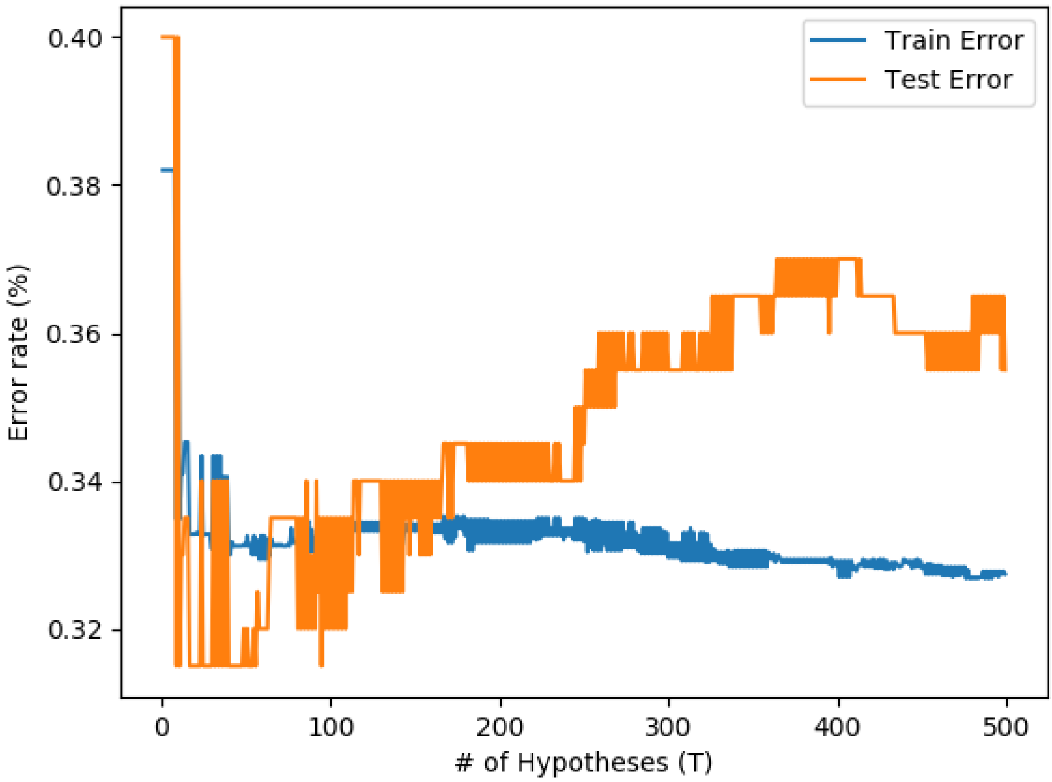
- $Q8$ with noise:
  - We can see that when noise was introduced, the test error curve changed from a monotonically decreasing curve, to a parabola-like curve with a minimum point.
  - This is due to what's called overfitting - when the hypotheses class becomes more complex, it starts to adjust to the bias (~noise), thus generalizing not as good.
  - In the case of Adaboost, the comlexity is controlled by the number of classifiers used in predicting new data - the X axis in these plots.
  - Noise = 0.01:

Train vs Test error, Adaboost



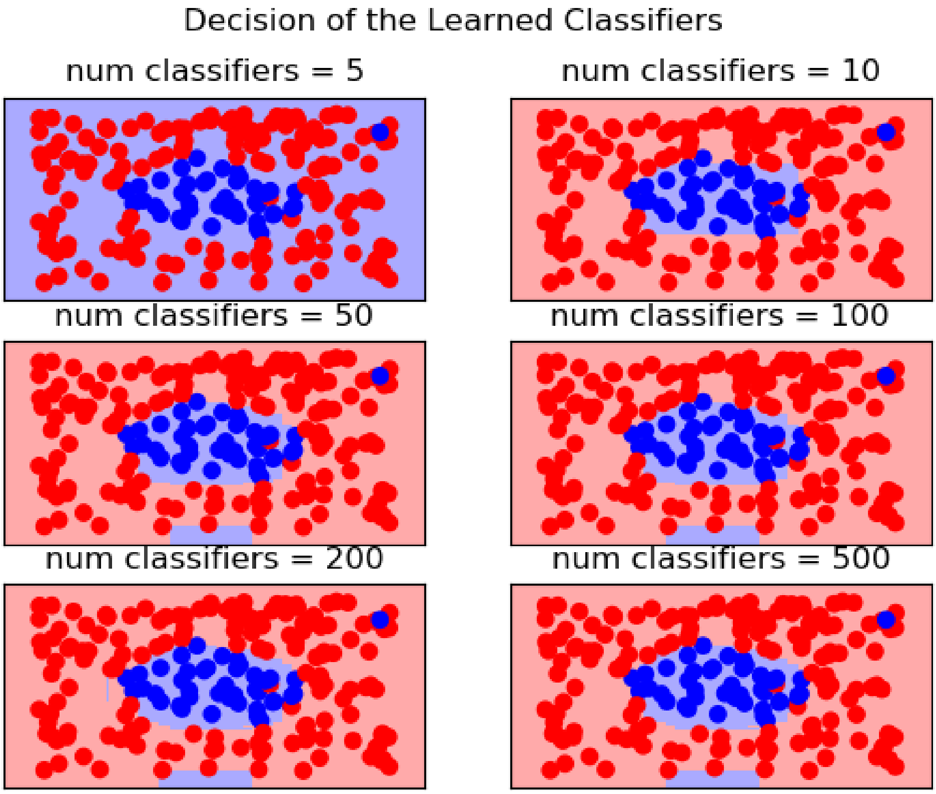
– Noise = 0.4:

Train vs Test error, Adaboost

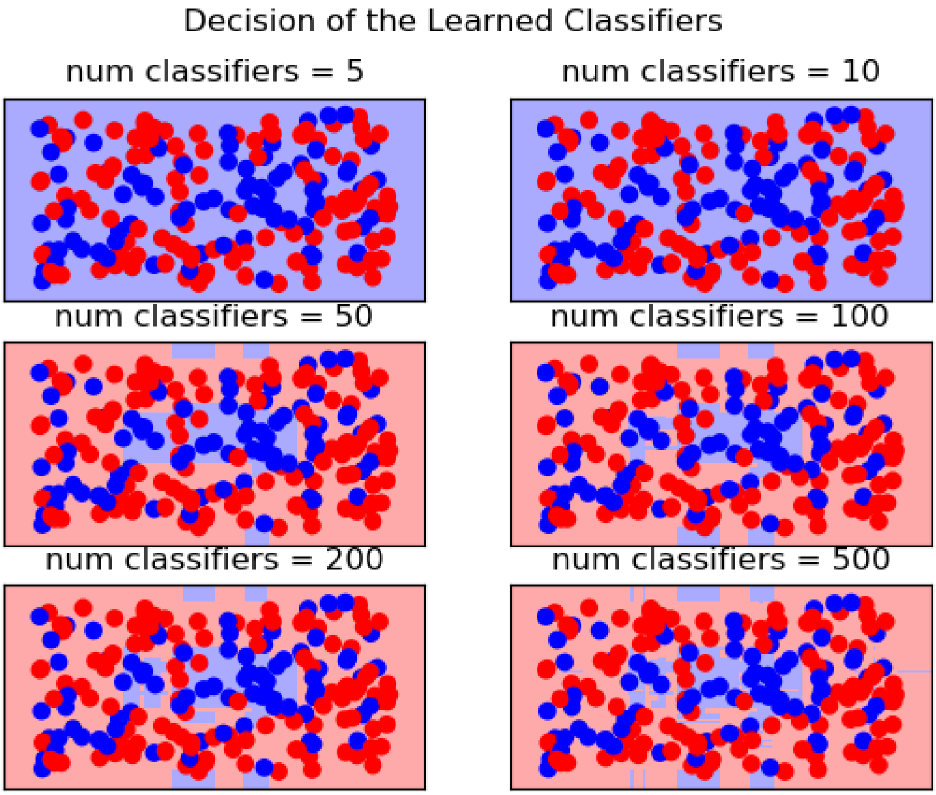


- $Q9$ with noise:

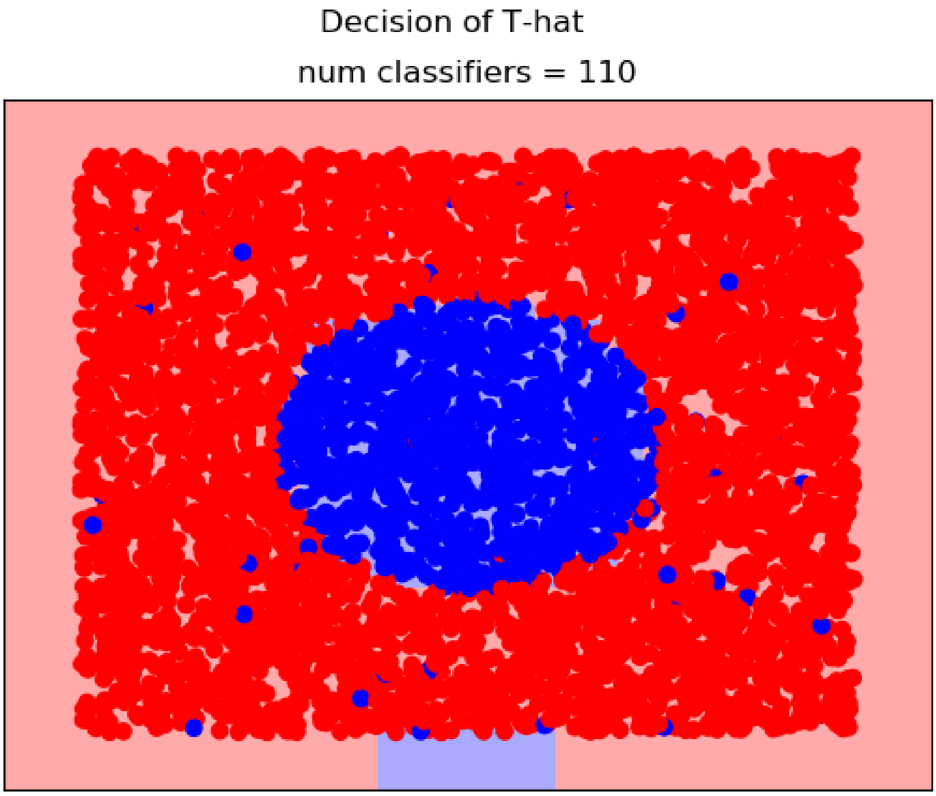
– Noise = 0.01:



– Noise = 0.4:



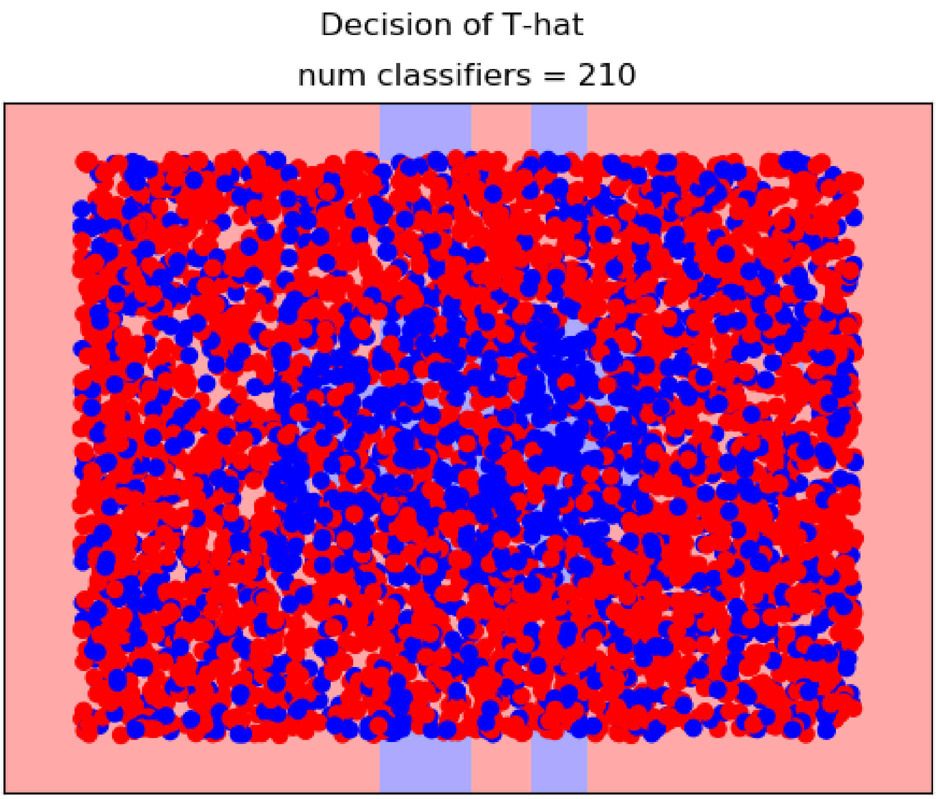
- $Q10$ with noise:
  - Here, we see that $\hat{T}$ is the one that hits the bias-variance tradeoff on spot.
  - Noise = 0.01:



  - Noise = 0.4:

Algorithm 1 Calculating $S(a,b)$ in $O(n)$

- 1. Initialize $sum \leftarrow I(a,b)$.
  2. Update $sum \leftarrow sum + S(a-1,b)$.
  3. Update $sum \leftarrow sum + S(a,b-1)$.
  4. Update $sum \leftarrow sum - S(a-1,b-1)$.



Face Classification

Q13

- First of all, w.l.o.g assume (a,b) isn't an edge pixel (the other case can be handled with minor adjustments).
- Runtime analysis:
  - Stage 1: $O(1)$,
  - Stage 2: $O(1)$,
  - Stage 3: $O(1)$,
  - Stage 4: $O(1)$.
  - Overall, $O(1)$.
  - Since we do this for every entry in the image, we get $O(n)$, as required.

Q14

- **Algorithm:**
- **Rational:**
  - Similar to the Inclusion-exclusion principle, we add and subtract areas of the integral image that we're not added or added twice.

Algorithm 2 Finding Sum of Square in $O(1)$

Input:

- Integral image denoted by S,
- A square represented by four points, $P_\Delta = (a_\Delta, b_\Delta)$ for $\Delta \in \{LU, LD, RU, RD\}$ corresponding to their orientation (LU = Left-Up etc.).

Algorithm:

1. Initialize $sum \leftarrow S(P_{RD})$.
2. Update $sum \leftarrow sum - S(a_{LD}, b_{LD} - 1)$.
3. Update $sum \leftarrow sum - S(a_{RU} - 1, b_{RU})$.
4. Update $sum \leftarrow sum + S(a_{LU} - 1, b_{LU} - 1)$.
5. *return sum*.

Algorithm 3 Calculating a Haar Feature in $O(1)$

1. Calculate sum of all squares.
2. Add up the white squares, subtract the black ones.
3. *return* the result.

Q15

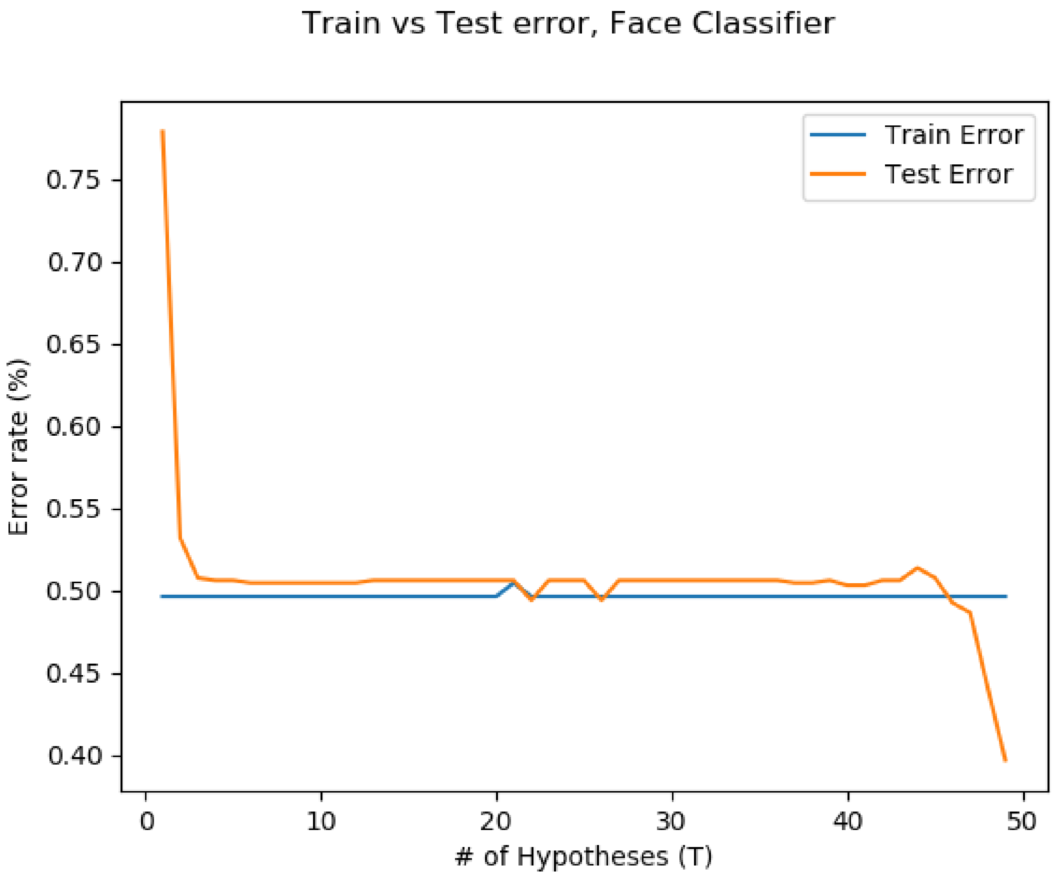
- **Algorithm:**
- **Runtime analysis:**
  - Stage 1, 2 and 3 all take $O(1)$ time, thus the overall procedure takes $O(1)$. ■

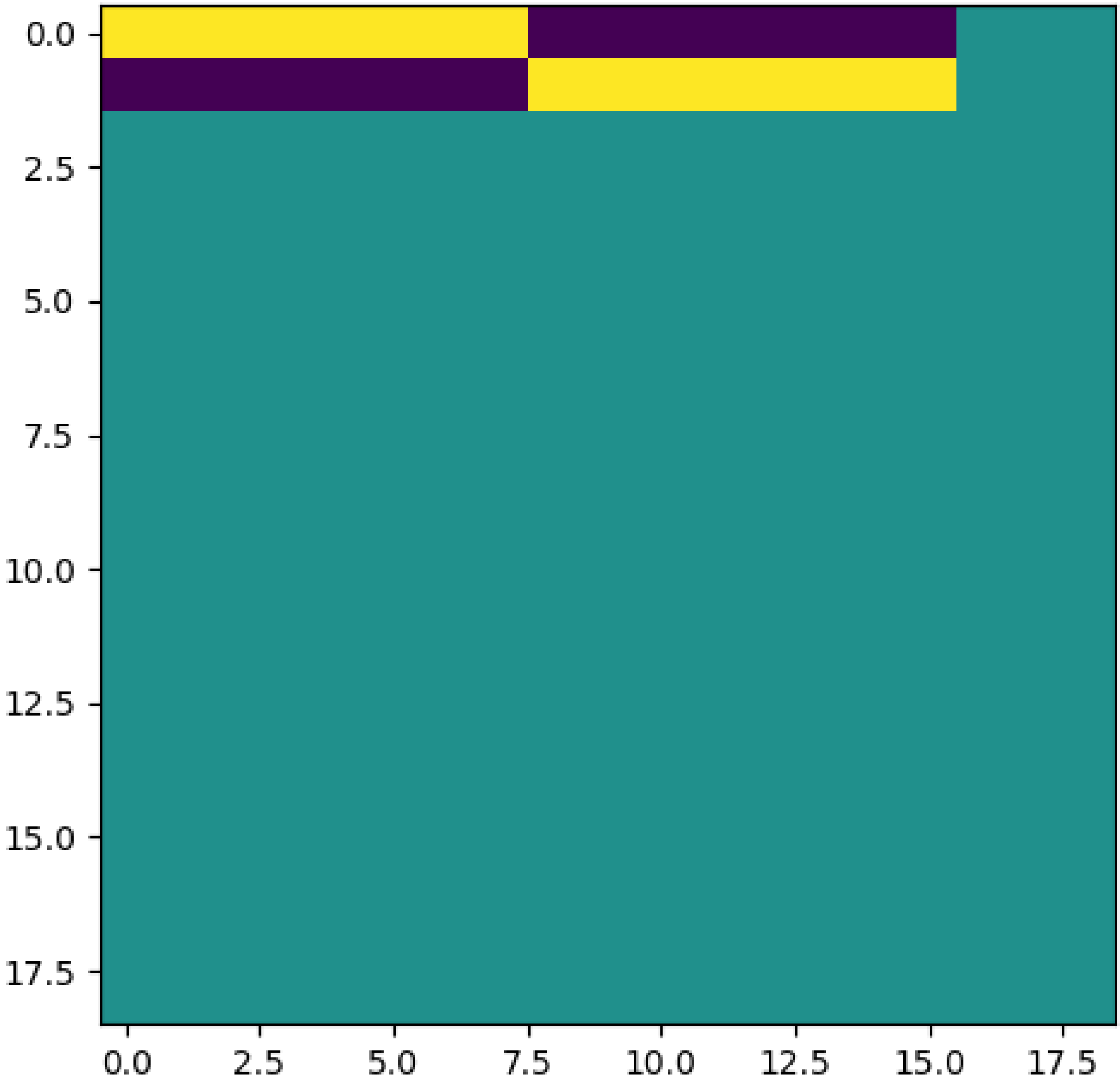
Q17

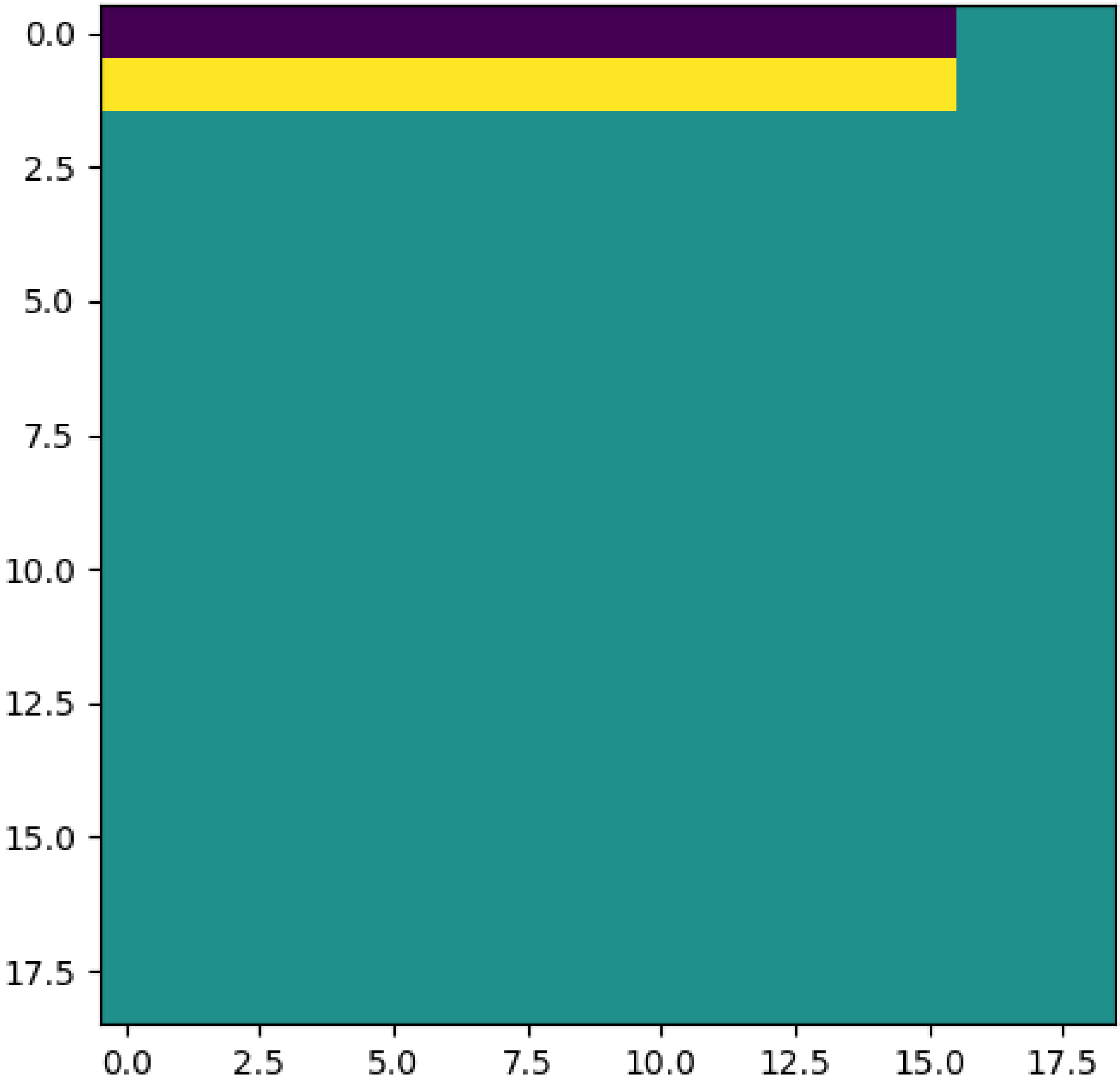
Implemented and ran it, but the result aren't good...

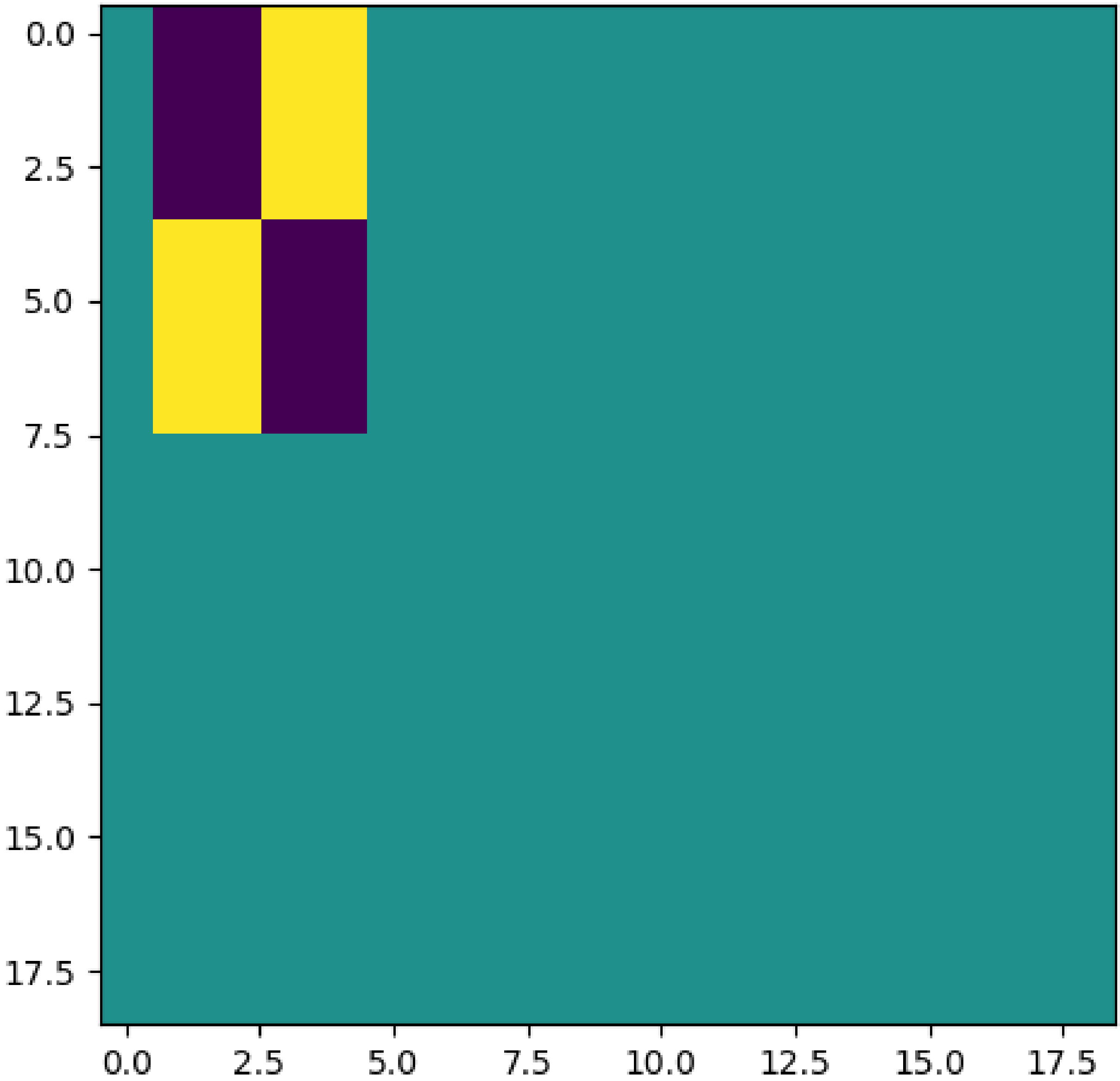
During the training, I plotted the optimal Haar features, and they all seem to have an 'up' value of 0 (= they're all locked to the top border of the image).

Couldn't find the bit of code that caused it though.


Train vs Test error, Face Classifier
Train Error
Test Error
Error rate (%)
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0
10
20
30
40
50
# of Hypotheses (T)


0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5