Freie Universität Berlin

Prof. R. Rojas

Mustererkennung, WS17/18

Übungsblatt 10

Boyan Hristov, Nedeltscho Petrov

27. Dezember 2017

Link zum Git Repository: https://github.com/BoyanH/FU-MachineLearning-17-18/tree/master/Solutions/Homework10

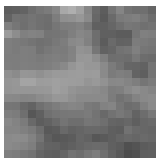
Ada Boost und Face Detection

Dataset

```
\item MIT-CBCL-database
\item Link: \url{https://github.com/HyTruongSon/Pattern-Classification/tree/master/MIT-CBCL
    -database}
\item Training set - 2 429 Bilder mit Gesichter, 4 548 ohne
\item Test set - 472 Bilder mit Gesichter, 23 573 ohne
```

Der Datensatz hat leider wenigere Bilder mit Gesichter. Das ist im Testfall ganz gut, beim Trainieren aber eher unausreichend. Man könnte theoretisch beim Trainieren noch weitere Daten, z.B. von dem Gesichtsdatensatz aus der vorherigen Hausaufgaben, benutzen um bessere Ergebnisse zu bekommen. Hier ist es aber sehr schön, dass die Bilder ohne Gesichter ganz gut gewählt sind, dass sie ähnliche Farben und einige ähnliche Striche haben, damit der Lernalgorithmus besser wird.

Hier ist ein gutes Beispiel für ein Bild, dass kein Gesicht enthält, aber man auch als Mensch nicht ganz klar damit kommt. (pgm Bild aus Datensatz zu finden in ./Dataset/train/non-face/B1_00005.pgm). Auf dem Bildschirm sah es einigermaßen ähnlicher, oder vielleicht haben wir zu viel Phantasie.



Gesichtserkennung und allgemeine Vorgehensweise

Wir haben uns entschieden, Face Detection zu implementieren, da uns das Paper gut gefallen hat. Dabei haben wir wie im Paper vorgeschlagen erstmal das Integralbild für je Beispiel berechnet und davon viele schwache Klassifizierer erstellt, wobei je Klassifizier eine Form von den im Paper beschriebene haben kann und dazu noch beliebige Höhe, Breite und Position. Dabei könnten wir noch mehr Positionen und Größen generieren, waren aber dabei ziemlich aufmerksam, da das Performance bei mehreren Features drastisch senkt. Im Paper wurde vorgeschlagen, dazu noch mit beliebige Tresholds auszuprobieren und das ganze als separate Klassifizierer zu betrachten. Wir haben aber lineare Regression genommen, um von allen Werten, berechnet von allen Integralbilder und Features, möglichst schnell (beides Implementieraufwand und Performance, einigermaßen) eine möglichst gute lineare Trennung zwischen den beiden Klassen anhand eines Werts zu finden. Also unsere einzelne Klassifizierer sind lineare Regressionen, alle anhand von nur 1 Feature (z.B Differenz zwischen die Helligkeit im linken und rechten Bereich des Bildes).

Wir haben auch ausprobiert, bessere Ergebnisse zu bekommen, in dem wir eine Klasse mehr tolerieren.

Ergebnisse

Wir haben mit unterschiedlichen Parameter das ganze ausprobiert. Dabei haben wir bemerkt, dass je mehr Features und Klassifizierer man am Anfang generiert, desto bessere Ergebnisse man bekommt. Bei mehrere ausgewählte Klassifizierer (nach AdaBoost) oder bei Tolerierung einer Klasse kann man schlecht was deutlich besseres bekommen. Diesmal haben wir auch die Konfussionsmatrix benutzt, da wir oft ganz gute Fehlerrate bekommen können, ohne echt gute Ergebnisse zu haben. Das ist so, da es deutlich mehrere Bilder ohne Gesichter im Testdatensatz gibt, was dazu führt, dass man 30 Prozent Fehlerrate bekommen kann, wenn man immer die negative Klasse vorhersagt.

Hier sind einige gute Beispiele (entweder relativ gute Ergebnisse oder ganz schlechte). Alle Beispiele sind mit 2420 initiale Features bzw. Klassifizierer.

fp_tolerate=0.3 (Wert zwischen 0 und 1 wie viel man die positive Klasse bevorzugt, später weiter erklärt) 60 Klassifizierer. Hier muss man aufpassen, dass wir prozentuell ganz gute Fehlerrate haben, aber kaum welche Gesichter erkennen.

```
2420 weak classifiers were successfully trained and their predictions saved!
Score train: 0.7556256270603411
Score test: 0.9167394468704513
Confusion matrix train:
[[4325 1482]
 [ 223  947]]
Confusion matrix test:
[[22032   461]
 [ 1541    11]]
```

fp_tolerate=0.7, 300 Klassifizier

```
2420 weak classifiers were successfully trained and their predictions saved!
Score train: 0.8251397448760213
Score test: 0.8468288625493866
Confusion matrix train:
[[4154  826]
 [ 394 1603]]
Confusion matrix test:
```

```
[[20234   344]
 [ 3339   128]]
```

Hier ist die Fehlerrate größer, die Ergebnisse sind aber deutlich besser, da man circa 84% der Bilder ohne Gesichter erkennt und dazu noch 37% der Bilder mit Gesichter erkennt.

Es gibt noch 4 Beispiele in GitHub, die sind aber nicht zu interessant.

Evaluation

Gesichtserkennung ist ein relativ schwieriges Entscheidungsproblem, deswegen sind unsere Ergebnisse auch nicht perfekt. Wir haben aber bemerkt, dass AdaBoost eine ganz effektive und schnelle Methode ist. Die ist relativ leicht einzustellen, man kann relativ leicht Klassen bevorzugen. Wenn man auch eine gute Menge von schwache Klassifizierer hat, die anhand von total unteschiedliche Features funktionieren, kann man auch sehr gute Ergebnisse bekommen. In unserem Fall, wenn man ganz simple Features benutzt, da man nicht sicher ist wie man mit so ein Problem vorgehen kann, hat es auch super funktioniert. Das gute ist auch, dass wir mit dem Ansatz skalierbare Klassifizerer benutzt haben und so alle Arten von Bilder und alle Größen von Gesichter erkennen können, ohne zu höhe Laufzeitskosten.

Erklärung des Codes

Diesmal werden wir nicht alles erklären, da das Code etwas mehr ist, man kann aber alles in GitHub finden.

Berechnung des Integralbilds

Wir haben das Integralbild iterativ und nicht wie im Paper vorgeschlagen rekursiv berechnet, da es uns auch in dem Fall intuitiver war. Hier ist row_sum die Summe aller Pixeln über Position x. Damit kann man die in den Integralimage gleich nach unten propagieren und mit dem Integralimage unten links aufsummieren. So bearbeitet man das Bild von oben links nach unten rechts und pusht, so zu sagen, die Ergebnisse immer nach unten.

```
def get_integral_image(img):
    row_sum = np.zeros(img.shape)
    integral_image = np.zeros((img.shape[0] + 1, img.shape[1] + 1), dtype=np.int64)

    for y, row in enumerate(img):
        for x, col in enumerate(row):
            row_sum[y, x] = row_sum[y - 1, x] + img[y, x]
            integral_image[y + 1, x + 1] = integral_image[y + 1, x] + row_sum[y, x]

    return integral_image
```

Berechnung der Summe innerhalb eines Rechtecks im Integralbild

Die Vorgehensweise wurde sogar schon gegeben im Paper, nicht viel zu erklären hier. (Seite 140, Figure 3)

```
def get_integral_img_sub_sum(integral_image, top_left_position, bottom_right_position):
    top_left = top_left_position[1], top_left_position[0]
    bottom_right = bottom_right_position[1], bottom_right_position[0]

    if top_left == bottom_right:
        return integral_image[top_left]

    top_right = bottom_right[0], top_left[1]
    bottom_left = top_left[0], bottom_right[1]

    return np.int64(np.int64(integral_image[bottom_right]) - np.int64(integral_image[
    top_right]) -
                    np.int64(integral_image[bottom_left]) + np.int64(integral_image[
    top_left]))
```

Berechnung der Features

Hier ist nur die Berechnung eines Features zu sehen, die Methode ist aber zeimlich lang und langweilig. Hier ist nur interessant, dass wir Prozente für die Größe und Position für je Feature benutzt haben, um das ganze skalierbar zu haben. Nachdem muss man nur alle notwendige Positionen auf dem Integralbild berechnen und damit auch die Summen, nachher muss man nur das Endprodukt berechnen.

```
img_width = integral_image.shape[1] - 1
        img_height = integral_image.shape[0] - 1

        pos_x = int(self.xp * img_width)
        pos_y = int(self.yp * img_height)
        top_left = pos_x, pos_y

        br_x = int(pos_x + self.wp * img_width)
        br_y = int(pos_y + self.hp * img_height)
        bottom_right = br_x, br_y

        if self.type == FDFType.TWO_RECTANGLE_HORIZONTAL:
            middle_left = top_left[0], int(top_left[1] + self.hp * img_width / 2)
            middle_right = bottom_right[0], middle_left[1]
            a = get_integral_img_sub_sum(integral_image, top_left, middle_right)
            b = get_integral_img_sub_sum(integral_image, middle_left, bottom_right)

            return a - b
```

Generierung der Features

Wie man hier sieht, könnten wir deutlich mehrere Positionen nehmen, war aber zu ineffizient.

```
        percents = np.arange(.4, .65, .05)
        # for a number of various feature positions, sizes and types,
        # create the pool of classifiers which will later be sieved with AdaBoost
        # Total amount of classifiers (features for face detection) is 405
        for wp in percents:
            for hp in percents:
                for xp in np.arange(0.1, 1 - wp, .1):
                    for yp in np.arange(0.1, 1 - hp, .1):
                        for f_type in types:
                            feature = FDFeature(wp, hp, f_type, xp, yp)
                            classifier = WeakClassifier(feature)
                            classifier_predictions.append(classifier.fit_predict(X_, y))
                            possible_classifiers.append(classifier)
```

Predict nach AdaBoost und damit Gesichtserkennung

Wir haben bei der Berechnung von je Gewicht eines Klassifizierers die Teilung durch 2 weggelassen, deswegen auch hier bei dem Vergleich mit der Summe aller Gewichten. Sonst ist alles wie im Paper.

```
    def predict(self, X):
        X_ = self.transform(X)
        predictions = np.zeros(len(X))

        for i, classifier in enumerate(self.classifiers):
            predictions += classifier.predict(X_) * self.cw[i]

        weights_sum = self.cw.sum()
        return np.vectorize(lambda p: 1 if p >= weights_sum else 0)(predictions)
```

AdaBoost

Hier haben wir eher die im Tutorium vorgeschlagene Implementierung geguckt, aber das Paper war auch hilfreich. Im Großen und Ganzen ist alles wie im Tutorium besprochen. Wir haben aber hier fp_tolerate genommen, dass bei der Initialisierung der Gewichte der einzelnen Beispiele helfen soll. Damit sollen erstmall die False Negatives minimiert werden, wie man auch oben bei den Ergebnissen sieht. Die ganze Idee ist, dass man versucht eine Summe von Gewicht 1 gleichmäßig erstmal zwischen den beiden Klassen und dann auch innerhalb einer Klasse zu verteilen. Mit dem Parameter können wir aber definieren, dass eine Klasse, wie in dem Fall, 70% aller Gewichte bekommen soll und damit bevorzugt werden soll. Wie man in den Ergebnissen sieht, hat das funktioniert, aber eigentlich besser (zu schlechtere deutlich Ergebnisse geführt) wenn man das falsch anwendet (fp_tolerate = 0.3).

```
class AdaBoost:
    @staticmethod
    def boost_classifiers(classifier_pool, predictions, labels, k, fp_tolerate=0.7):
        data_size = len(labels)
        classifiers_count = len(classifier_pool)
        pos_size = len(labels[np.where(labels == 1)])
        neg_size = data_size - pos_size
        cw = []  # weights of chosen classifiers
        # initialize data set weights

        # here we take a slightly different approach from the general ada boost one
        # we have an fp_tolerate argument which can be set between 0 and 1
        # 0 fully ignores false positives and only tries to correct false negatives, 1 is
    opposite case
        # using this argument we can decide whether the error is equally separated between
    fp and fn
        data_weight = np.vectorize(lambda x: fp_tolerate / pos_size if x == 1 else (1 -
    fp_tolerate) / neg_size)(labels)
        classifiers = []

        # calculate error vector for each classifier
        # e.g. if a classifiers mistakes only 2. and 4. sample, its vector would be
    [0,1,0,1,0...,0]
        # that way, we can easily multiply the data-set weight vector by the error vector
    later on
        # to get e_i
        classifiers_e_vectors = np.zeros((classifiers_count, data_size), dtype=np.float64)
        for i in range(classifiers_count):
            wrong_idx = predictions[i] != labels
            # expected_pos = np.array(labels) == np.array([1] * len(labels))
            # false_negatives = np.logical_and(wrong_idx, expected_pos)
            # false_positives = np.invert(false_negatives)
            # wrongs = np.where(wrong_idx)
```

```
            classifiers_e_vectors[i][wrong_idx] = 1

        for j in range(k):
            e = classifiers_e_vectors.dot(data_weight) / data_weight.sum()
            idx_best = e.argmin()
            classifiers.append(classifier_pool[idx_best])

            e_i = e[idx_best]
            new_cw = np.log((1 - e_i) / (e_i + np.nextafter(0, 1)))
            cw.append(new_cw)

            signs = (classifiers_e_vectors[idx_best] - 0.5) * 2
            dw_update = np.vectorize(lambda s: np.exp(s * new_cw))(signs)
            data_weight = data_weight * dw_update

            classifier_pool = classifier_pool[:idx_best] + classifier_pool[idx_best + 1:]
            predictions = predictions[:idx_best] + predictions[idx_best + 1:]
            classifiers_e_vectors = np.delete(classifiers_e_vectors, idx_best, 0)

        return classifiers, np.array(cw)
```

Vollständiges Code

AdaBoost.py

```
import numpy as np


class AdaBoost:
    @staticmethod
    def boost_classifiers(classifier_pool, predictions, labels, k, fp_tolerate=0.7):
        data_size = len(labels)
        classifiers_count = len(classifier_pool)
        pos_size = len(labels[np.where(labels == 1)])
        neg_size = data_size - pos_size
        cw = []  # weights of chosen classifiers
        # initialize data set weights

        # here we take a slightly different approach from the general ada boost one
        # we have an fp_tolerate argument which can be set between 0 and 1
        # 0 fully ignores false positives and only tries to correct false negatives, 1 is opposite case
        # using this argument we can decide whether the error is equally separated between fp and fn
        data_weight = np.vectorize(lambda x: fp_tolerate / pos_size if x == 1 else (1 - fp_tolerate) / neg_size)(labels)
        classifiers = []

        # calculate error vector for each classifier
        # e.g. if a classifiers mistakes only 2. and 4. sample, its vector would be [0,1,0,1,0...,0]
        # that way, we can easily multiply the data-set weight vector by the error vector later on
        # to get e_i
        classifiers_e_vectors = np.zeros((classifiers_count, data_size), dtype=np.float64)
        for i in range(classifiers_count):
            wrong_idx = predictions[i] != labels
            # expected_pos = np.array(labels) == np.array([1] * len(labels))
            # false_negatives = np.logical_and(wrong_idx, expected_pos)
            # false_positives = np.invert(false_negatives)
            # wrongs = np.where(wrong_idx)

            classifiers_e_vectors[i][wrong_idx] = 1
```

```
        for j in range(k):
            e = classifiers_e_vectors.dot(data_weight) / data_weight.sum()
            idx_best = e.argmin()
            classifiers.append(classifier_pool[idx_best])

            e_i = e[idx_best]
            new_cw = np.log((1 - e_i) / (e_i + np.nextafter(0, 1)))
            cw.append(new_cw)

            signs = (classifiers_e_vectors[idx_best] - 0.5) * 2
            dw_update = np.vectorize(lambda s: np.exp(s * new_cw))(signs)
            data_weight = data_weight * dw_update

            classifier_pool = classifier_pool[:idx_best] + classifier_pool[idx_best + 1:]
            predictions = predictions[:idx_best] + predictions[idx_best + 1:]
            classifiers_e_vectors = np.delete(classifiers_e_vectors, idx_best, 0)

        return classifiers, np.array(cw)
```

FaceDetection.py

```
import numpy as np
from Classifier import Classifier
from Parser import get_test_set, get_train_set
from Helpers import get_integral_image
from FDFeature import FDFeature
from FDFType import FDFType
from WeakClassifier import WeakClassifier
from AdaBoost import AdaBoost

types = [FDFType.TWO_RECTANGLE_HORIZONTAL, FDFType.TWO_RECTANGLE_VERTICAL,
         FDFType.THREE_RECTANGLE_HORIZONTAL, FDFType.THREE_RECTANGLE_VERTICAL,
         FDFType.FOUR_RECTANGLE]


class FaceDetection(Classifier):
    def __init__(self, k=300):
        self.classifiers = []
        self.cw = []  # classifier weights
        self.k_classifiers = k

    def fit(self, X, y):
        X_ = self.transform(X)

        # initialize all possible classifiers
        possible_classifiers = []
        classifier_predictions = []
        percents = np.arange(.4, .65, .05)

        # for a number of various feature positions, sizes and types,
        # create the pool of classifiers which will later be sieved with AdaBoost
        # Total amount of classifiers (features for face detection) is 405
        for wp in percents:
            for hp in percents:
                for xp in np.arange(0.1, 1 - wp, .1):
                    for yp in np.arange(0.1, 1 - hp, .1):
                        for f_type in types:
                            feature = FDFeature(wp, hp, f_type, xp, yp)
                            classifier = WeakClassifier(feature)
                            classifier_predictions.append(classifier.fit_predict(X_, y))
                            possible_classifiers.append(classifier)
```

```
        print('{} weak classifiers were successfully trained and their predictions saved!'.format(
            len(possible_classifiers)))

        self.classifiers, self.cw = AdaBoost.boost_classifiers(
            possible_classifiers, classifier_predictions, y, self.k_classifiers)

    def transform(self, X):
        return np.vectorize(lambda x: get_integral_image(x),
                            signature='(m,n)->(z,c)')(X)

    def predict(self, X):
        X_ = self.transform(X)
        predictions = np.zeros(len(X))

        for i, classifier in enumerate(self.classifiers):
            predictions += classifier.predict(X_) * self.cw[i]

        weights_sum = self.cw.sum()
        return np.vectorize(lambda p: 1 if p >= weights_sum else 0)(predictions)


X_test, y_test = get_test_set()
X_train, y_train = get_train_set()

ds_size = None
fd = FaceDetection()
fd.fit(X_train[:ds_size], y_train[:ds_size])
predictions_train = fd.predict(X_train[:ds_size])
predictions_test = fd.predict(X_test[:ds_size])
print('Score train: {}'.format(fd.score(predictions_train, y_train[:ds_size])))
print('Score test: {}'.format(fd.score(predictions_test, y_test[:ds_size])))
print('Confusion matrix train: \n{}'.format(fd.confusion_matrix(predictions_train, y_train[:ds_size])))
print('Confusion matrix test: \n{}'.format(fd.confusion_matrix(predictions_test, y_test[:ds_size])))

# print(X_test[0].shape)
#
# feature = FDFeature(.7, .3, FDFType.TWO_RECTANGLE_HORIZONTAL, .2, .2)
# wc = WeakClassifier(feature)
# X_train_ = np.vectorize(lambda x: get_integral_image(x),
#                         signature='(m,n)->(z,c)')(X_train)
# wc.fit(X_train_, y_train)
# print(wc.score(X_train_, y_train))
```

FDFeature.py

```
from FDFType import FDFType
from Helpers import get_integral_img_sub_sum


class FDFeature:
    def __init__(self, width_percent, height_percent, type, pos_x_percent, pos_y_percent):
        self.wp = width_percent
        self.hp = height_percent
        self.type = type
        self.xp = pos_x_percent
        self.yp = pos_y_percent

    def get_value(self, integral_image):
        img_width = integral_image.shape[1] - 1
        img_height = integral_image.shape[0] - 1
```

```
        pos_x = int(self.xp * img_width)
        pos_y = int(self.yp * img_height)
        top_left = pos_x, pos_y

        br_x = int(pos_x + self.wp * img_width)
        br_y = int(pos_y + self.hp * img_height)
        bottom_right = br_x, br_y

        if self.type == FDFType.TWO_RECTANGLE_HORIZONTAL:
            middle_left = top_left[0], int(top_left[1] + self.hp * img_width / 2)
            middle_right = bottom_right[0], middle_left[1]
            a = get_integral_img_sub_sum(integral_image, top_left, middle_right)
            b = get_integral_img_sub_sum(integral_image, middle_left, bottom_right)

            return a - b
        elif self.type == FDFType.TWO_RECTANGLE_VERTICAL:
            middle_top = int(top_left[0] + img_width / 2 * self.wp), top_left[1]
            middle_bottom = middle_top[0], bottom_right[1]

            a = get_integral_img_sub_sum(integral_image, top_left, middle_bottom)
            b = get_integral_img_sub_sum(integral_image, middle_top, bottom_right)

            return a - b
        elif self.type == FDFType.THREE_RECTANGLE_HORIZONTAL:
            one_third_left = top_left[0], int(top_left[1] + 1 / 3 * img_height * self.hp)
            one_third_right = bottom_right[0], one_third_left[1]
            two_thirds_left = top_left[0], int(top_left[1] + 2 / 3 * img_height * self.hp)
            two_thirds_right = bottom_right[0], two_thirds_left[1]

            a = get_integral_img_sub_sum(integral_image, top_left, one_third_right)
            b = get_integral_img_sub_sum(integral_image, one_third_left, two_thirds_right)
            c = get_integral_img_sub_sum(integral_image, two_thirds_left, bottom_right)

            return a - b + c
        elif self.type == FDFType.THREE_RECTANGLE_VERTICAL:
            one_third_top = int(top_left[0] + 1 / 3 * self.wp * img_width), top_left[1]
            one_third_bottom = one_third_top[0], bottom_right[1]
            two_thirds_top = int(top_left[0] + 2 / 3 * self.wp * img_width), top_left[1]
            two_thirds_bottom = two_thirds_top[0], bottom_right[1]

            a = get_integral_img_sub_sum(integral_image, top_left, one_third_bottom)
            b = get_integral_img_sub_sum(integral_image, one_third_top, two_thirds_bottom)
            c = get_integral_img_sub_sum(integral_image, two_thirds_top, bottom_right)

            return a - b + c
        elif self.type == FDFType.FOUR_RECTANGLE:
            middle_left = top_left[0], int(top_left[1] + self.hp * img_width / 2)
            middle_right = bottom_right[0], middle_left[1]
            middle_top = int(top_left[0] + img_width / 2 * self.wp), top_left[1]
            middle_bottom = middle_top[0], bottom_right[1]
            middle = middle_top[0], middle_left[1]

            a = get_integral_img_sub_sum(integral_image, top_left, middle)
            b = get_integral_img_sub_sum(integral_image, middle_top, middle_right)
            c = get_integral_img_sub_sum(integral_image, middle_left, middle_bottom)
            d = get_integral_img_sub_sum(integral_image, middle, bottom_right)

            return (a + d) - (b + c)

        return 0
```

FDFType.py

```
from enum import Enum
```

```
class FDFType(Enum):
    TWO_RECTANGLE_HORIZONTAL = 1
    TWO_RECTANGLE_VERTICAL = 2
    THREE_RECTANGLE_HORIZONTAL = 3
    THREE_RECTANGLE_VERTICAL = 4

    # it makes no sense to differ between horizontal and vertical here
    # as it is the difference between the diagonals
    # positive or negative, it is the same number, we don't really care
    FOUR_RECTANGLE = 5
```

WeakClassifier.py

```
from sklearn.linear_model import LinearRegression
import numpy as np
from Classifier import Classifier

class WeakClassifier(Classifier):
    def __init__(self, feature):
        self.feature = feature
        self.lr = LinearRegression()

    def fit(self, X, y):
        X_ = self.transform(X)
        y_ = np.vectorize(lambda x: 1 if x == 1 else -1)(y)
        self.lr.fit(X_, y_)

    def fit_(self, X, y):
        self.lr.fit(X, y)

    def fit_predict(self, X, y):
        X_ = self.transform(X)
        y_ = np.vectorize(lambda x: 1 if x == 1 else -1)(y)
        self.fit_(X_, y_)
        return self.predict_(X_)

    def predict(self, X):
        X_ = self.transform(X)
        return self.predict_(X_)

    def predict_(self, X):
        predictions = self.lr.predict(X)
        return np.vectorize(lambda x: 1 if x > 0 else 0)(predictions)

    def transform(self, X):
        return np.vectorize(lambda x: self.feature.get_value(x),
                            signature='(m,n)->()')(X).reshape(-1,1)
```

Helpers.py

```
import numpy as np


def get_integral_image(img):
    row_sum = np.zeros(img.shape)
    integral_image = np.zeros((img.shape[0] + 1, img.shape[1] + 1), dtype=np.int64)

    for y, row in enumerate(img):
```

```
        for x, col in enumerate(row):
            row_sum[y, x] = row_sum[y - 1, x] + img[y, x]
            integral_image[y + 1, x + 1] = integral_image[y + 1, x - 1 + 1] + row_sum[y, x]

    return integral_image


def get_integral_img_sub_sum(integral_image, top_left_position, bottom_right_position):
    # as x and y coordinates in an image are flipped, therefore they are flipped within the
     integral image as well
    # as you can see in the upper method, we take the rows as y and cols as x, but in the
    final result we leave
    # them as they are

    top_left = top_left_position[1], top_left_position[0]
    bottom_right = bottom_right_position[1], bottom_right_position[0]

    # sum of a single cell, as coords of top left and bottom right corner of sub-image are
    identical
    if top_left == bottom_right:
        return integral_image[top_left]

    top_right = bottom_right[0], top_left[1]
    bottom_left = top_left[0], bottom_right[1]

    return np.int64(np.int64(integral_image[bottom_right]) - np.int64(integral_image[
    top_right]) -
                    np.int64(integral_image[bottom_left]) + np.int64(integral_image[
    top_left]))
```

Parser.py

```
import numpy as np
import glob
import os
import cv2

def read_pgm(file_name):
    return cv2.imread(file_name, -1)


def get_data_set(data_set_type='test'):
    data_folder = os.path.abspath(os.path.join(os.path.dirname(__file__), './Dataset'))
    faces = glob.glob(data_folder + '/{}/face/*.pgm'.format(data_set_type))
    non_faces = glob.glob(data_folder + '/{}/non-face/*.pgm'.format(data_set_type))
    data_set = []
    labels = []

    for file in faces:
        image = read_pgm(file)
        data_set.append(image)

    for file in non_faces:
        image = read_pgm(file)
        data_set.append(image)

    labels = labels + ([1]*len(faces))
    labels = labels + ([0]*len(non_faces))
    labels = np.array(labels)

    data_set = np.array(data_set)
    np.random.seed(1)
    idx = np.random.permutation(len(data_set))
```

```
    # return a random permutation so positive and negative samples are mixed
    return data_set[idx], labels[idx]


def get_test_set():
    return get_data_set('test')


def get_train_set():
    return get_data_set('train')
```