Sam Davies s1220039 & Michael Inglis s1232123
24 February 2015

IVR Practical 1

Shape Recognition

Introduction

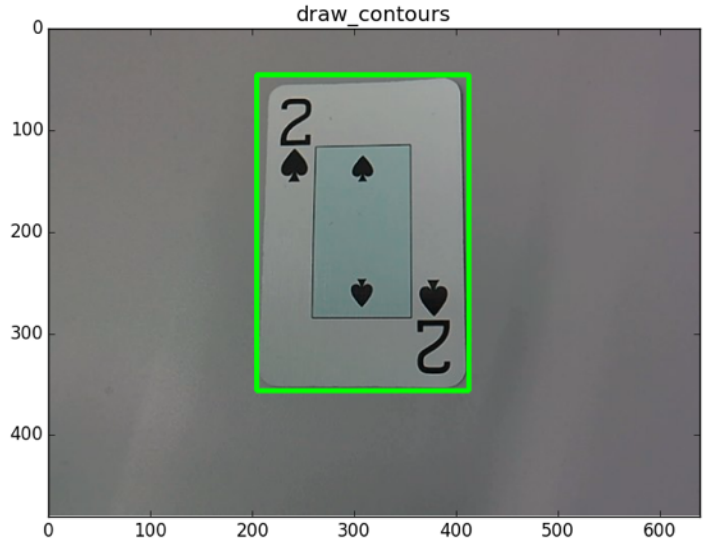
We opted to use Python as our programming language as both of us lacked experience with Matlab and were confident with Python. After trying out simpleCV and realising it didn't have the extensive feature set that we required for vision we used openCV which is a vision library and the scikit-learn library for the machine learning. Work was split evenly on both the code and report.

Methods

The first thing in any classification problem is to create valid training data. To do this with the training image files we had to work with, we first used thresholding. On getting the most out of our training data (and thinking ahead with the new data in the live demo), we decided to use two different types of object finder on the images described in detail below.

Simple Threshold Object Finder:

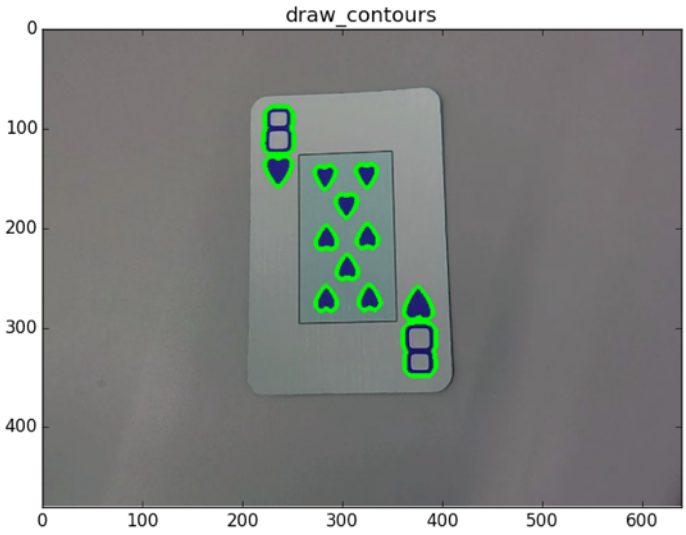
We first applied a gaussian blur. This helps us remove noise as contrast below a certain threshold gets blurred. This object finder then converts to a grayscale image. Converting it to grayscale allows us to threshold in the next step. We then apply a fixed threshold of hard coded values which best suited our dataset. Thresholding the image gives us a binary pixels (black or white) which allows us to easily draw contours along the edge of any black/white pixels. Our simple object finder ignores the card's centre box and the objects inside as we set it up originally to only look at the outer number and suit, something we decided against later on. The largest four contours are then taken by removing any non child contours of the card itself which removes the centre rectangle and all the symbols inside. These are assumed to be our numbers and suit. We then move onto creating feature vectors for each respective object.



Adaptive Threshold Object Finder:

The adaptive threshold gives us better results for images with varying illumination. We again used a filter, but instead used a median filter, as it gave us a decent result without too much tweaking. The adaptive threshold was then applied after making the image grayscale and and the contours were drawn. When this is applied, the contours in the centre box are indeed recognised. This function was also used later on for counting the symbols in the box. This version of the thresholding essentially collects more information than the simple object finder as it contains more contours and probably gives a more accurate representation of the features despite not achieving a higher classification percentage. We then remove unwanted contours. This is done in two steps. The second thing we do is remove irregular contours in which the aspect ratio of a symbol must be between a certain threshold. This was done to help remove any erroneous contours. With this done, we can then move ahead with the creation of feature vectors.

Once we have a set of contours with either of the above methods, we are able to construct a feature vector for each one. What we decided on was to use compactness, extent and the mean red value within that object. The compactness and the extent were derived from the perimeter and area respectively. The mean red value was taken by taking the mean red of all the pixels in the object. This allows us to easily identify between red and black cards.



Other features we tested in our feature vector were moment and hull. This didn't however produce better results, so we decided to exclude them. Other features such as area and perimeter were excluded to avoid overfitting with the machine learning as they appeared in compactness and extent.

Given that we had our set of training feature vectors, it is now possible to start classifying our test set. For each individual card, we go through the same process as above. Gaining feature vectors from either the simple or adaptive thresholding. Once we have valid feature vectors, we then used a Gaussian Bayes classifier to predict the suit. This process was done for every valid feature vector. We then took the mode of all the feature vectors to come up with the most commonly occurring suit. This produced far better results than the original K-Nearest Neighbour classifier that we tried. This could be due to a limited amount of training data and the fact a Bayes classifier assumes independence on each feature.

In coming up with the numbers, we originally used a similar method to the suit classification. This yielded less than satisfactory results, mainly because there was limited training data (each suit had 9 to train from, while the numbers only worked from 4 training examples) and we'd have to solve the fact that 6 and 9 are physically the same when they are rotation invariant.

Given that our original method of number classification wasn't particularly good, we opted to use adaptive thresholding again on the card and count the number of objects. Our thresholding and blurring appeared to do a decent job of ignoring the square box in the centre, so it was just a case of counting the number of objects above a certain size and subtracting four (the two numbers and two large symbols) to calculate the number that appears on the card. From this we were able to compare our results to the actual card and construct our results and the confusion matrices seen below.

Results

Given that we tested the two different threshold settings and got different results, we considered it worth including both sets of results. Shown below are the two confusion matrices for the suits when using adaptive and simple thresholding. We see for the adaptive thresholding, that not all cards had objects recognised. The cards [18,26,27], (5 of Hearts, 3 of Hearts and 3 of Clubs respectively) failed to have any objects and were thus excluded from the matrix, thus giving us our object detector (at finding the least amount of objects required for classification) a score of 91% accuracy. One thing worth noticing with these results which is that the misclassified cards are all found to be a suit of similar colour.

Adaptive Thresholding Suits

	Spades	Hearts	Clubs	Diamonds
Spades	7			
Hearts		8		
Clubs	1		7	
Diamonds		3		3

The second suit confusion matrix below which featured the simple thresholding was rather interesting in that it managed to classify at least every card. It however misclassified two diamonds as spades. Somewhat bizarre given that the colour of the card is taken into account on the feature vector.

Simple Thresholding Suits

	Spades	Hearts	Clubs	Diamonds
Spades	8			
Hearts		8		
Clubs			8	
Diamonds	2			6

The result of the number classification were the same on both threshold variants were the same given that the number counting is done using the adaptive threshold independently from the thresholding for the suit classification. This then gives us the same result of the cards [18,26,27] not having any symbols and thus being excluded from the matrix. What we see here are two numbers, 3 and 5 failing to properly be counted.

Adaptive Thresholding Numbers

	2	3	4	5	6	7	8	9
2	4							
3		2						
4			4					
5				2	1			
6					4			
7						4		
8							4	
9								4

Comparing the whole set of cards with what they actually are, we get the following results:

Total Percentage Classified Correctly for Adaptive Thresholding - 75.0%

Total Percentage Classified Correctly for Simple Thresholding - 84.375%

What we see here is that generally the simple thresholding tends to preform better. However both still has room for improvement and suggestions are listed below.

Discussion

As with any classifying problem, having more training data which includes variants of condition such as focus and illumination or even different types of cards, will always benefit the problem. This would allow us to compare different classifiers which would preform better such as KNN. We also could have tried a support vector machine and see what different results could've been achieved with that.

We attempted to use a median blur instead of the gaussian blur but got a worse result in the simple object finder. This initially gave us a worse result but not much tweaking of the parameters was done, so it's difficult to tell what the effect could've been. The same can be said for the Adaptive filtering but with the opposite filtering. The gaussian didn't give us a good classification but no parameter tweaking was done. The median filter with the parameter we have seemed to work quite well however. This would be worth looking into in significant depth as our adaptive thresholding managed to split cards into at least the correct colour of suit (possibly due to blurred clubs looking like blurred spades) while the simple thresholding failed to get 100% on this but managed to at least classify every card. Perhaps adapting different parameters to suit the non-classified cards could be beneficial as long as it does not affect the rest of the correctly classified cards.

Code Appendix

```
from count_symbols import CountSymbols
from featuriser import Featuriser

__author__ = 'Sam Davies and Mingles'
import cv2
import numpy as np
from scipy import stats

from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix


class CardClassifier(object):

    def __init__(self):
        self.train = []
        self.train_labels = []
```

```
def get_objects_with_label(self, img, label):
    """
    Extract all the objects from an image and add each
    object's feature vector to the training data
    :param img: the image to extract from
    :param label: the label for all objects in this image
    """

    feature_vectors = Featuriser(img).feature_vectors

    for feature_vector in feature_vectors:
        self.train.append(feature_vector)
        self.train_labels.append(label)
        # print "{0} -- label {1}".format(feature_vector, label)
        # print "------------------"

def classify_card(self, img):
    """
    Classify the image by matching each feature of the card to features
    from the test set, then voting on the most occurring card.
    :param img: the image to classify
    """
    feature_vectors = Featuriser(img).feature_vectors
    to_classify = np.array(feature_vectors).astype(np.float32)

    train = np.array(self.train).astype(np.float32)
    train_labels = np.array(self.train_labels).astype(np.float32)


    if to_classify.shape != (0, ):
        gnb = GaussianNB()
        gnb.fit(train, train_labels)
        results = gnb.predict(to_classify)

        suit = []

        for result in results:
            digit = self.convert_class_to_digit(result)
            print "class: {0} -- suit: {1}".format(result, result % 4)
            suit.append(result % 4)


        single_suit = stats.mode(suit, axis=None)[0]
        single_card_number = CountSymbols(img).symbol_count - 4.0

        single_class = self.convert_suit_and_num_to_card(single_card_number, single_suit)
        print "AVERAGE class: {0} --  suit: {1} --  digit: {2}"\
            .format(single_class[0], single_suit[0], single_card_number)
        return single_class, single_suit[0], single_card_number
    else:
        return -1, -1, -1

@staticmethod
def convert_class_to_digit(card_num):
    """
    Given a card number, find the corresponding digit
    :param card_num: card number
```

```
        :return: digit
        """
        return ((int(card_num) - 1)/4) + 2

    @staticmethod
    def convert_suit_and_num_to_card(digit, suit):
        """
        Given a suit and a digit, find the card number
        :param digit: digit
        :param suit: suit
        :return: the card number
        """
        card_num_bit = (digit - 1) * 4
        mod_bit = ((suit - 1) % 4) + 1
        return card_num_bit - (4 - mod_bit)

    def add_training_images(self, labels):
        """
        Add all the cards in training set to the training data
        :param labels: a list of labels for the training cards
        """
        for x in range(1, len(labels)):
            img = cv2.imread('Images/ivr1415pract1data1/train{0}.jpg'.format(x))
            self.get_objects_with_label(img, labels[x-1])

    def classify_all_test_cards(self, labels):
        """
        Counts the number of cards in the test set which are classified correctly
        :param labels: the labels of the test set cards
        :return: the percent correct and the individual classes and numbers
        """
        count = 0
        single_suits = []
        single_nums = []

        for x in range(1, len(labels)+1):
            img = cv2.imread('Images/ivr1415pract1data2/test{0}.jpg'.format(x))
            label = labels[x-1]
            print("-----")
            print "Classifying card " + str(label)
            classification, single_suit, single_num = self.classify_card(img)
            single_suits.append(single_suit)
            single_nums.append(single_num)
            if classification != -1 and classification == label:
                count += 1
        return (100.0 * count / len(labels)), single_suits, single_nums

    @staticmethod
    def get_test_label(num):
        """
        given the card number in the test set, find the corresponding label in the training set
        :param num: the number of the card in test set
        :return: the label from the training set
        """
        return (29 - (4 * (int(num)/4))) + (int(num) % 4)

    def get_confusion_matrices(self, test_labels, suits_pred, nums_pred):
        """
```

```
        display the confusion matrices for the suits and the digits
        :param test_labels: the real labels for the images
        :param suits_pred: the predicted suits
        :param nums_pred: the predicted digits
        """
        nums_test = []
        suits_test = []

        for i in range(0, 32):
            nums_test.append(c.convert_class_to_digit(test_labels[i]))
            suits_test.append((test_labels[i]) % 4)

        print "----------"
        print "SUIT CONFUSION MATRIX"
        suits_test, suits_pred, bad_suits = self.removed_bad_classes(suits_test, suits_pred)
        print "Removed cards - {0}".format(bad_suits)
        conf_suit = confusion_matrix(suits_test, suits_pred)
        print conf_suit

        print "----------"
        print "DIGIT CONFUSION MATRIX"
        nums_test, nums_pred, bad_num = self.removed_bad_classes(nums_test, nums_pred)
        print "Removed cards - {0}".format(bad_num)
        conf_num = confusion_matrix(nums_test, nums_pred)
        print conf_num

    def removed_bad_classes(self, test, pred):
        """
        Remove the cards from for which now feature vectors were found
        :param test: the real labels
        :param pred: the predicted labels
        :return: the edited test adn pred along with the bad cards
        """
        bad = []
        new_test = []
        new_pred = []
        for p in range(0, len(pred)):
            if pred[p] < 0:
                bad.append(p)
            else:
                new_pred.append(int(pred[p]))
                new_test.append(test[p])
        return new_test, new_pred, bad


if __name__ == "__main__":
    c = CardClassifier()
    training_labels = []
    testing_labels = []
    for i in range(0, 32):
        training_labels.append(i+1)
        testing_label = c.get_test_label(i)
        testing_labels.append(testing_label)
        print("training_label: " + str(i+1) + " testing_label: " + str(testing_label))

    c.add_training_images(training_labels)
    correctly_classified, suits_pred, nums_pred = c.classify_all_test_cards(testing_labels)
    print "--------------------------------"
```

```
    print "{0}% Correctly Classified".format(correctly_classified)
    print "--------------------------------"

    c.get_confusion_matrices(testing_labels, suits_pred, nums_pred)


__author__ = 'Sam Davies and Mingles'
import cv2
import numpy as np
from countour_finder import ContourFinder


class FeaturiserSimple(object):

    def __init__(self, img):
        gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        contours_sorted, _, _ = self.img_to_contours(gray_img)

        min_area = 110
        max_area = cv2.contourArea(contours_sorted[0])/25
        relevant_contours = self.find_relevant_contours(contours_sorted, min_area, max_area)

        self.feature_vectors = [self.get_feature_vector(cnt, img, gray_img) for cnt in relevant_contours]

    @staticmethod
    def get_feature_vector(cnt, img, gray_img):
        """
        Extract the feature vector of the given contour
        :param cnt: the contour to extract from
        :return: the feature vector extracted
        """
        moments = cv2.moments(cnt)

        area = cv2.contourArea(cnt)
        perimeter = cv2.arcLength(cnt, True)

        hull = cv2.convexHull(cnt)
        hull_area = cv2.contourArea(hull)
        solidity = float(area)/hull_area

        x, y, w, h = cv2.boundingRect(cnt)
        rect_area = w*h
        extent = float(area)/rect_area

        mask = np.zeros(gray_img.shape, np.uint8)
        cv2.drawContours(mask, [cnt], 0, 255, -1)
        mean_val = cv2.mean(img, mask=mask)

        hu_moment = cv2.HuMoments(moments)
        #print hu_moment

        compactness = perimeter * perimeter / (4 * np.pi * area)

        feature_vector = [compactness, extent, mean_val[2]]
        return feature_vector

    def img_to_contours(self, gray_img):
```

```
        """
        Get a list of all contours in this image sorted by area descending
        :param gray_img: the image to get contours from
        :return: contours sorted by area descending
        """
        # turn the image into binary (black and white, no grey)
        blur = cv2.GaussianBlur(gray_img, (1, 1), 1000)
        ret, thresh = cv2.threshold(blur, 129, 255, cv2.THRESH_BINARY)
        # find all the contours in the image, all areas of joint white/black
        contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

        card_cnt_index, card_cnt = self.max_contour_area_index(contours)

        # removed all non childs of the card
        good_cnts = [card_cnt]
        for n in range(0, len(contours)):
            # make sure that the contours parent is the card
            if hierarchy[0][n][3] == card_cnt_index:
                good_cnts.append(contours[n])

        # figure out the largest contour areas
        return sorted(good_cnts, key=cv2.contourArea, reverse=True), contours, hierarchy

    @staticmethod
    def max_contour_area_index(contours, excluding=[]):
        max_area = 0
        max_area_index = 0
        for b in range(0, len(contours)):
            if cv2.contourArea(contours[b]) > max_area and b not in excluding:
                max_area = cv2.contourArea(contours[b])
                max_area_index = b
        return max_area_index, contours[max_area_index]

    @staticmethod
    def find_relevant_contours(contours_sorted, min_area, max_area):
        """
        Using a heuristic, find the meaningful contours from a list of contours
        :param contours_sorted: the full list of contours
        :return: only the meaningful contours
        """
        if contours_sorted:
            # draw all the contours who's area is between 2 thresholds

            relevant_contours = []
            # print "max area {0}".format(max_area)
            for cnt in contours_sorted[1:]:
                area = cv2.contourArea(cnt)
                if min_area < area < max_area:
                    relevant_contours.append(cnt)
                else:
                    if min_area > area:
                        break
            return relevant_contours
        else:
            return []


class FeaturiserAdaptive(ContourFinder):
```

```
    def __init__(self, img):
        super(FeaturiserAdaptive, self).__init__(img)
        self.feature_vectors = [self.get_feature_vector(cnt, img, self.grey_image) for cnt in
                                            self.symbol_contours]

    @staticmethod
    def get_feature_vector(cnt, img, gray_img):
        return FeaturiserSimple.get_feature_vector(cnt, img, gray_img)


# class Featuriser(FeaturiserSimple):
class Featuriser(FeaturiserAdaptive):

    def __init__(self, img):
        super(Featuriser, self).__init__(img)


import cv2
from matplotlib import pyplot as plt

__author__ = 'Sam Davies and Mingles'


class ContourFinder(object):

    def __init__(self, img):
        gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        gray_img = cv2.medianBlur(gray_img, 5)
        self.grey_image = gray_img

        contours, hierarchy = self.image_to_contours()

        card_index, card_contour = self.find_card_contour(contours)

        card_area = cv2.contourArea(card_contour)
        self.min_area = 0.07 * cv2.contourArea(card_contour)
        self.max_area = 0.6 * cv2.contourArea(card_contour)

        good_contours = self.remove_non_child([0], contours, hierarchy)

        # print "card area " + str(cv2.contourArea(card_contour))
        # print "min " + str(self.min_area)
        # print "max " + str(self.max_area)

        good_contours = self.remove_bad_contours(good_contours)
        self.good_contours = sorted(good_contours, key=cv2.contourArea, reverse=True)

        self.symbol_contours = self.find_symbol_contours(self.good_contours)

    def find_card_contour(self, contours):
        image_index, image_contour = self.max_contour_area_index(contours)
        return self.max_contour_area_index(contours, excluding=[image_index])

    def find_inner_card_contour(self, contours):
        image_index, image_contour = self.max_contour_area_index(contours)
        card_index, card_contour = self.max_contour_area_index(contours, excluding=[image_index])
        return self.max_contour_area_index(contours, excluding=[image_index, card_index])
```

```
    def image_to_contours(self):
        # turn the image into binary (black and white, no grey)
        thresh = cv2.adaptiveThreshold(self.grey_image, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
cv2.THRESH_BINARY, 11, 3)
        return cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

    @staticmethod
    def remove_non_child(parent_indices, contours, hierarchy):
        # removed all non childs of the card
        good_cnts = []
        for n in range(0, len(contours)):
            # make sure that the contours parent is the card
            if hierarchy[0][n][3] in parent_indices:
                good_cnts.append(contours[n])
        return good_cnts

    def find_symbol_contours(self, contours):
        symbol_contours = []

        for cnt in contours:
            if self.max_area > cv2.contourArea(cnt):
                if cv2.contourArea(cnt) > self.min_area:
                    symbol_contours.append(cnt)
                else:
                    break
        return symbol_contours

    @staticmethod
    def max_contour_area_index(contours, excluding=[]):
        max_area = 0
        max_area_index = 0
        for b in range(0, len(contours)):
            if cv2.contourArea(contours[b]) > max_area and b not in excluding:
                max_area = cv2.contourArea(contours[b])
                max_area_index = b
        return max_area_index, contours[max_area_index]

    @staticmethod
    def draw_contours(card, contours):
        for cnt in contours:
            cv2.drawContours(card, [cnt], 0, (0, 255, 0), 3)

        plt.imshow(card)
        plt.title("draw_contours")
        plt.show()

    @staticmethod
    def remove_bad_contours(contours):

        good_contours = []

        # remove stretched shapes
        for cnt in contours:
            rect = cv2.minAreaRect(cnt)
            width = rect[1][0]
            height = rect[1][1]
            if height != 0:
                if 3 > (width/height) > (1.0/3):
```

```
                good_contours.append(cnt)
        return good_contours


from countour_finder import ContourFinder

__author__ = 'Sam Davies and Mingles'


class CountSymbols(ContourFinder):

    def __init__(self, img):
        super(CountSymbols, self).__init__(img)

        self.symbol_count = len(self.symbol_contours)
        print "------"
```