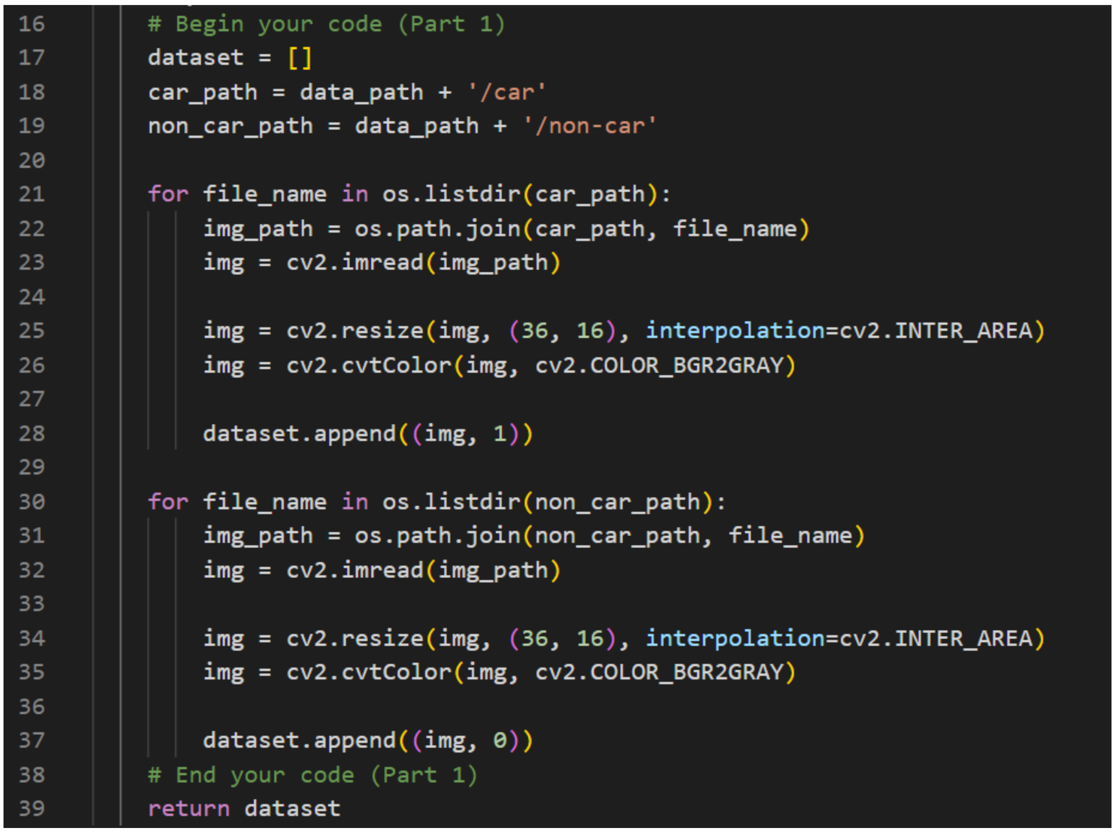
Task A

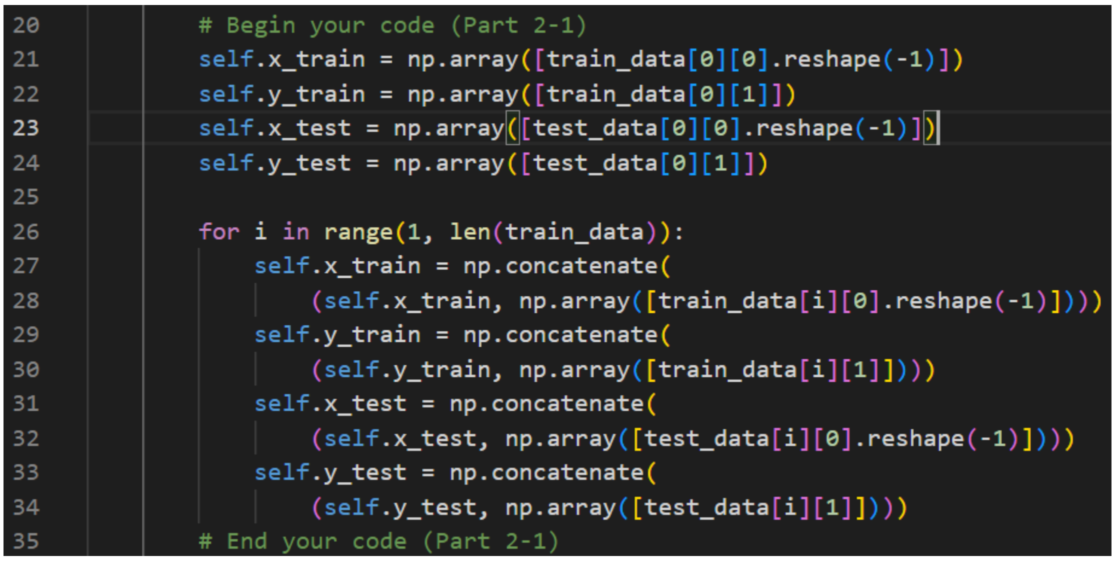
Part 1: Load and prepare your dataset (10%)



```python
    # Begin your code (Part 1)
    dataset = []
    car_path = data_path + '/car'
    non_car_path = data_path + '/non-car'

    for file_name in os.listdir(car_path):
        img_path = os.path.join(car_path, file_name)
        img = cv2.imread(img_path)

        img = cv2.resize(img, (36, 16), interpolation=cv2.INTER_AREA)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        dataset.append((img, 1))

    for file_name in os.listdir(non_car_path):
        img_path = os.path.join(non_car_path, file_name)
        img = cv2.imread(img_path)

        img = cv2.resize(img, (36, 16), interpolation=cv2.INTER_AREA)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        dataset.append((img, 0))
    # End your code (Part 1)
    return dataset
```

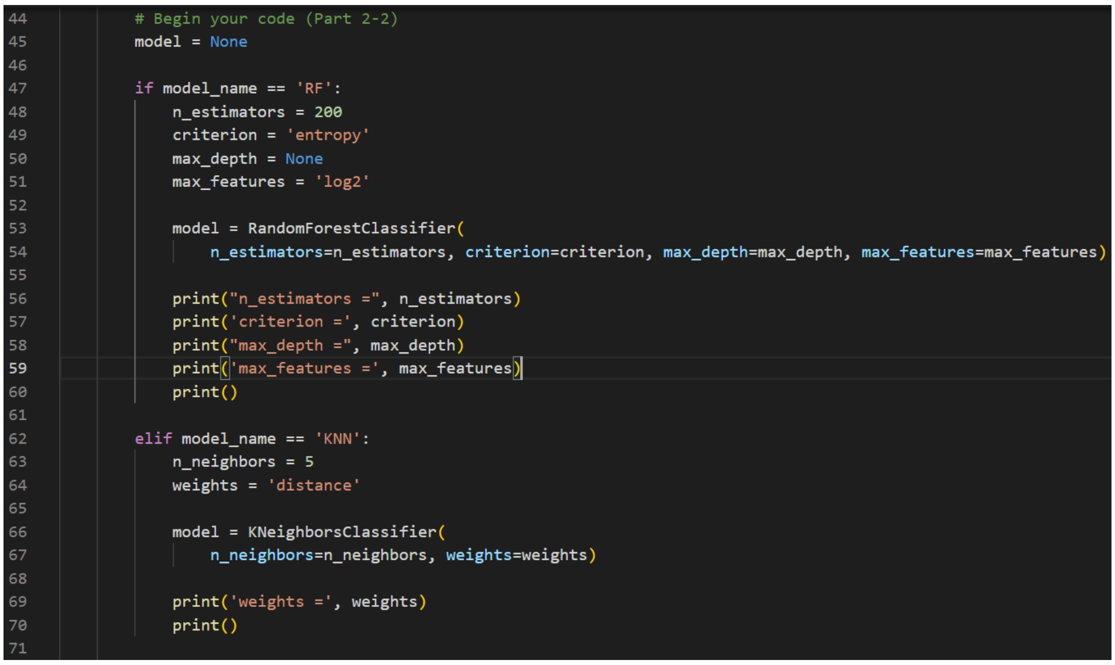
Load images and convert them to 36 x 16 grayscale images. Label those images and finally put the tuples altogether to a list.

Part 2: Build and Train Models (25%)

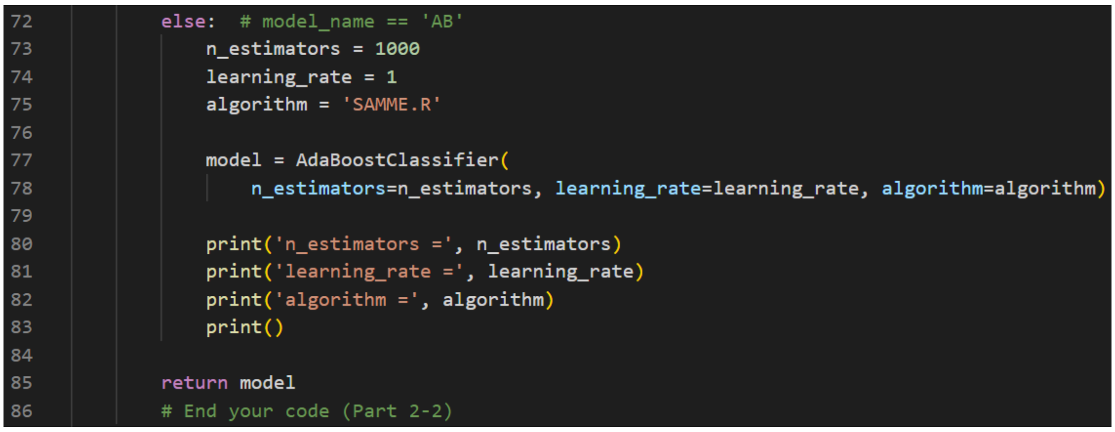


```python
        # Begin your code (Part 2-1)
        self.x_train = np.array([train_data[0][0].reshape(-1)])
        self.y_train = np.array([train_data[0][1]])
        self.x_test = np.array([test_data[0][0].reshape(-1)])
        self.y_test = np.array([test_data[0][1]])

        for i in range(1, len(train_data)):
            self.x_train = np.concatenate(
                (self.x_train, np.array([train_data[i][0].reshape(-1)])))
            self.y_train = np.concatenate(
                (self.y_train, np.array([train_data[i][1]])))
            self.x_test = np.concatenate(
                (self.x_test, np.array([test_data[i][0].reshape(-1)])))
            self.y_test = np.concatenate(
                (self.y_test, np.array([test_data[i][1]])))
        # End your code (Part 2-1)
```

Since an image is in the form of a two-dimensional array, we need to first transpose those images to one-dimensional arrays. Here I apply numpy.reshape() and numpy.concatenate() to do the job.

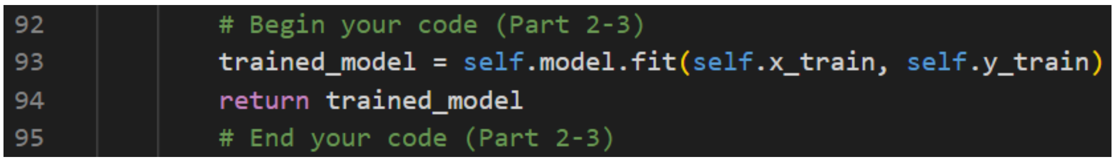


```
        # Begin your code (Part 2-2)
        model = None

        if model_name == 'RF':
            n_estimators = 200
            criterion = 'entropy'
            max_depth = None
            max_features = 'log2'

            model = RandomForestClassifier(
                n_estimators=n_estimators, criterion=criterion, max_depth=max_depth, max_features=max_features)

            print("n_estimators =", n_estimators)
            print('criterion =', criterion)
            print("max_depth =", max_depth)
            print('max_features =', max_features)
            print()

        elif model_name == 'KNN':
            n_neighbors = 5
            weights = 'distance'

            model = KNeighborsClassifier(
                n_neighbors=n_neighbors, weights=weights)

            print('weights =', weights)
            print()

```



```
        else:  # model_name == 'AB'
            n_estimators = 1000
            learning_rate = 1
            algorithm = 'SAMME.R'

            model = AdaBoostClassifier(
                n_estimators=n_estimators, learning_rate=learning_rate, algorithm=algorithm)

            print('n_estimators =', n_estimators)
            print('learning_rate =', learning_rate)
            print('algorithm =', algorithm)
            print()

        return model
        # End your code (Part 2-2)
```

Adjust the hyper-parameters.



```
        # Begin your code (Part 2-3)
        trained_model = self.model.fit(self.x_train, self.y_train)
        return trained_model
        # End your code (Part 2-3)
```

Start training.

1. Explain the difference between parametric and non-parametric models.

Ans: The difference between parametric and non-parametric models is that non-parametric models do not make any assumptions about the underlying distribution of the data. In comparison, parametric models do make assumptions about the probability distribution of the data being modeled, and they typically include one or more parameters that need to be estimated from the data.

2. What is ensemble learning? Please explain the difference between bagging, boosting and stacking.

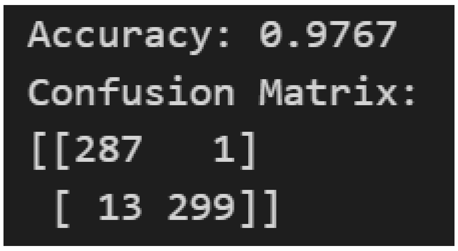
Ans: Ensemble learning is a machine learning technique that combines multiple models to improve the accuracy and robustness of predictions. Bagging and boosting both involve training multiple base models on different subsets of the training data and then combining their predictions to create a diverse set of models. The difference between bagging and boosting is that in boosting, weights of the misclassified instances are increased in order to focus the learning of the next model on the instances that were difficult to classify correctly by the previous model. Stacking involves training multiple base models, which can be completely different and be irrelevant to each other, and then combining their predictions using a meta-model. The meta-model takes the predictions of the base models as input and learns to make the final prediction.

3. Explain the meaning of the “n_neighbors” parameter in KNeighborsClassifier, “n_estimators” in RandomForestClassifier and AdaBoostClassifier.

Ans: “n_neighbors” in KNeighborsClassifier defines a value that the number of nearest neighbors will be considered when making classification predictions. ”n_estimators” in RandomForestClassifier and AdaBoostClassifier specifies the number of decision trees (base estimator) that will be trained on a random subset of the training data.

4. Explain the meaning of four numbers in the confusion matrix.

Ans:



```
Accuracy: 0.9767
Confusion Matrix:
[[287   1]
 [ 13 299]]
```

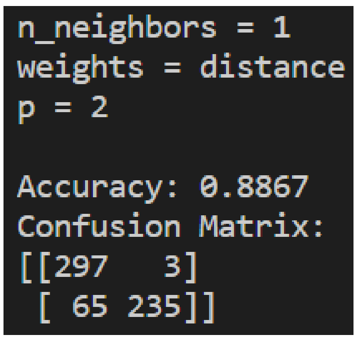
TP = 287, FP = 1, FN = 13, TN = 299.

Upper left: True Positive, which means that the answer should be “yes”, and the model also predicts “yes”.
Upper right: False Positive, which means that the answer should be “no”, but the model predicts “yes” (type I error).
Lower left: False Negative, which means that the answer should be “yes”, but the model predicts “no” (type II error).
Lower right: True Negative, which means that the answer should be “no”, and the model also predicts “no”.

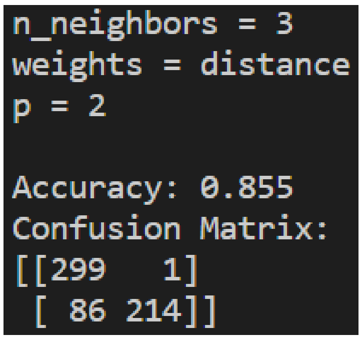
5. In addition to “Accuracy”, “Precision” and “Recall” are two common metrics in classification tasks, how to calculate them, and under what circumstances would you use them instead of “Accuracy”.

Ans: “Precision” equals TP / (TP + FP). “Recall” equals TP / (TP + FN). When the dataset is imbalanced, "Accuracy" can be a misleading metric as it can be high even if the model is only predicting the majority class. Under this situation, we’ll take precision and recall as metrics to evaluate the performance of our model.

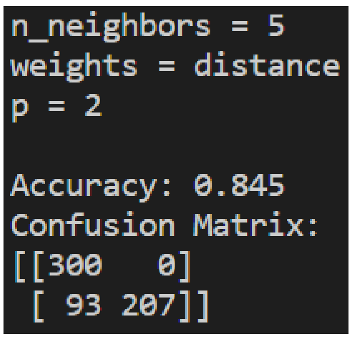
Part 3: Additional experiments (10%)



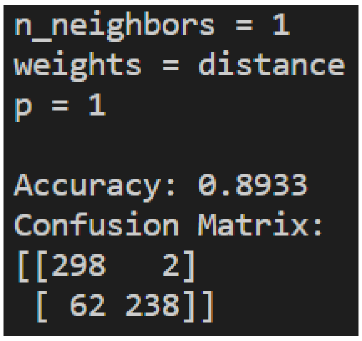
```
n_neighbors = 1
weights = distance
p = 2

Accuracy: 0.8867
Confusion Matrix:
[[297   3]
 [ 65 235]]
```



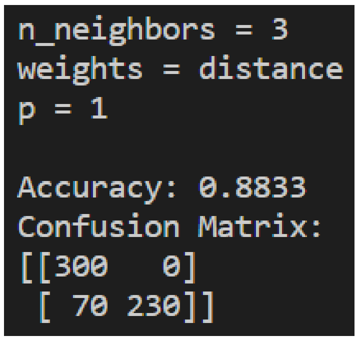
```
n_neighbors = 3
weights = distance
p = 2

Accuracy: 0.855
Confusion Matrix:
[[299   1]
 [ 86 214]]
```



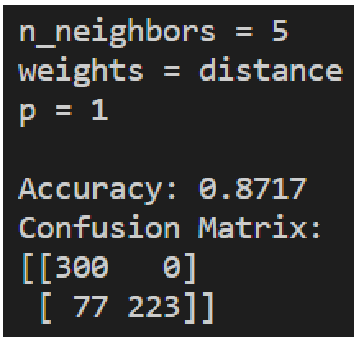
```
n_neighbors = 5
weights = distance
p = 2

Accuracy: 0.845
Confusion Matrix:
[[300   0]
 [ 93 207]]
```



```
n_neighbors = 1
weights = distance
p = 1

Accuracy: 0.8933
Confusion Matrix:
[[298   2]
 [ 62 238]]
```

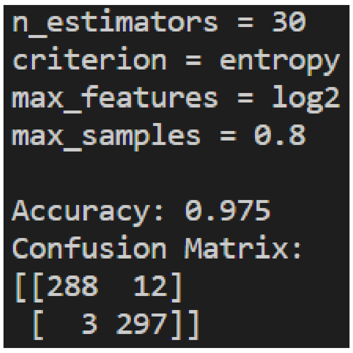


```
n_neighbors = 3
weights = distance
p = 1

Accuracy: 0.8833
Confusion Matrix:
[[300   0]
 [ 70 230]]
```

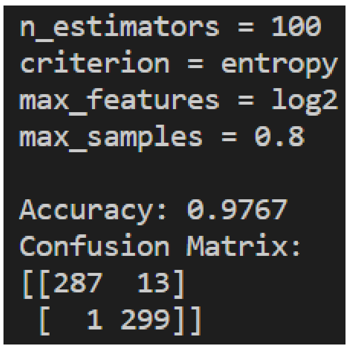


```
n_neighbors = 5
weights = distance
p = 1

Accuracy: 0.8717
Confusion Matrix:
[[300   0]
 [ 77 223]]
```

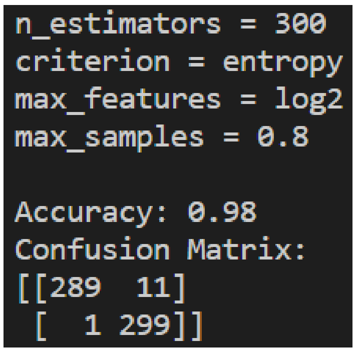
For KNeighborsClassifier, I observe that when “n_neighbors” becomes larger, the accuracy of the model decreases. Also, in this case, Manhattan distance is better than Euclidean distance as the metric, while weights hardly affect the result. Note that precision ≈ 1 in this case.



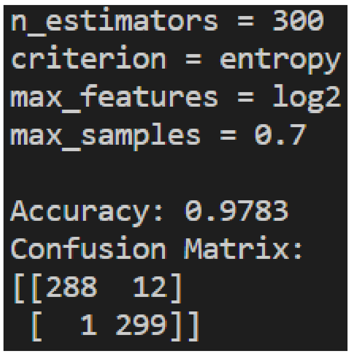
```
n_estimators = 30
criterion = entropy
max_features = log2
max_samples = 0.8

Accuracy: 0.975
Confusion Matrix:
[[288  12]
 [  3 297]]
```



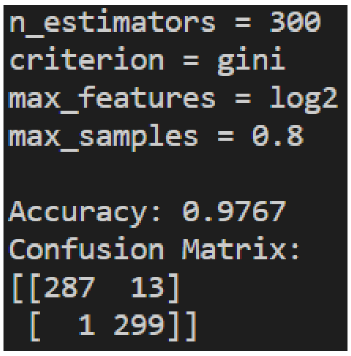
```
n_estimators = 100
criterion = entropy
max_features = log2
max_samples = 0.8

Accuracy: 0.9767
Confusion Matrix:
[[287  13]
 [  1 299]]
```



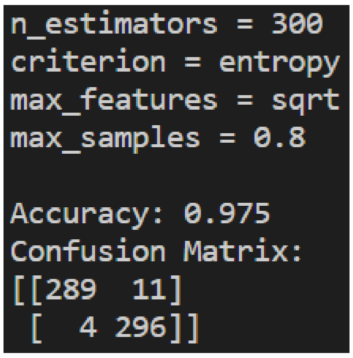
```
n_estimators = 300
criterion = entropy
max_features = log2
max_samples = 0.8

Accuracy: 0.98
Confusion Matrix:
[[289  11]
 [  1 299]]
```



```
n_estimators = 300
criterion = entropy
max_features = log2
max_samples = 0.7

Accuracy: 0.9783
Confusion Matrix:
[[288  12]
 [  1 299]]
```

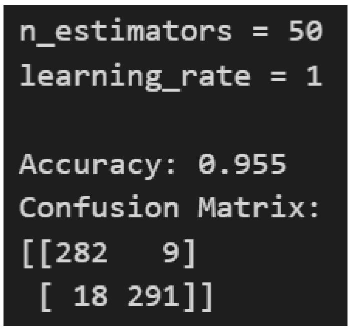


```
n_estimators = 300
criterion = gini
max_features = log2
max_samples = 0.8

Accuracy: 0.9767
Confusion Matrix:
[[287  13]
 [  1 299]]
```

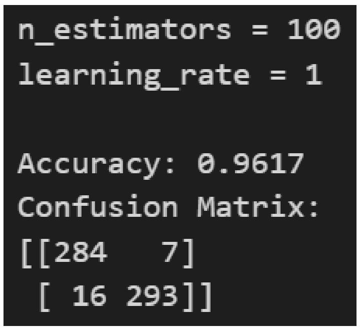


```
n_estimators = 300
criterion = entropy
max_features = sqrt
max_samples = 0.8

Accuracy: 0.975
Confusion Matrix:
[[289  11]
 [  4 296]]
```

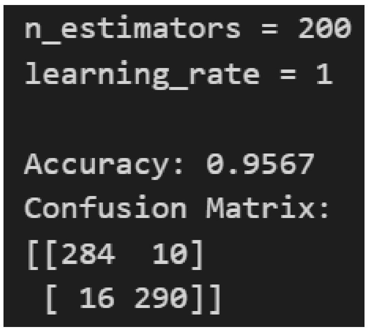
For RandomForestClassifier, from the above results in previous experiments, we can see that criterion = “entropy” and max_features = “log2” are slightly better. We choose n_estimators = 300, and the model will attain about 98% accuracy. I think that the performance is good enough and thus I didn’t apply decision tree pruning. The result shows that FP > FN, in contrast to KNeighborsClassifier (FN > FP)



```
n_estimators = 50
learning_rate = 1

Accuracy: 0.955
Confusion Matrix:
[[282   9]
 [ 18 291]]
```



```
n_estimators = 100
learning_rate = 1

Accuracy: 0.9617
Confusion Matrix:
[[284   7]
 [ 16 293]]
```



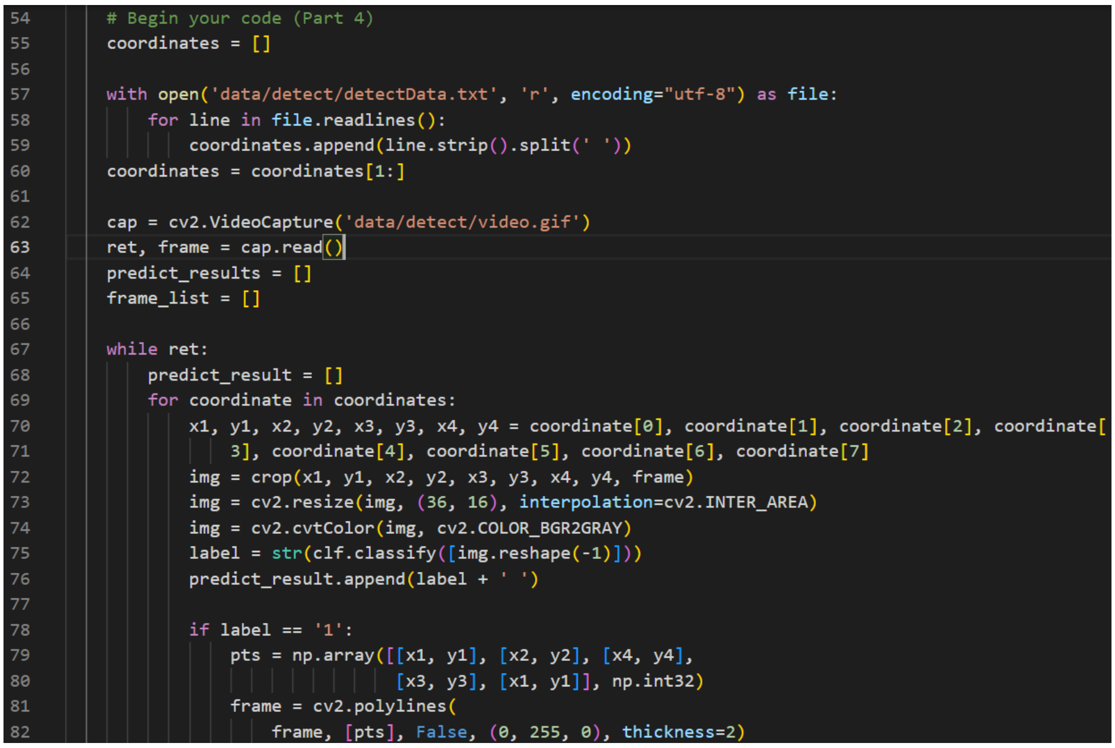
```
n_estimators = 200
learning_rate = 1

Accuracy: 0.9567
Confusion Matrix:
[[284  10]
 [ 16 290]]
```

For AdaBoostClassifier, since its default estimator is DecisionTreeClassifier initialized with max_depth = 1 and we can't import other packages due to the limit of this homework, it can hardly exploit its advantage of learning from previous mistakes. Certainly, we can take RandomForestClassifier as the base estimator for AdaBoostClassifier, but it's time-consuming and the result is not much better. Therefore, I choose RandomForestClassifier to be used in part 4.

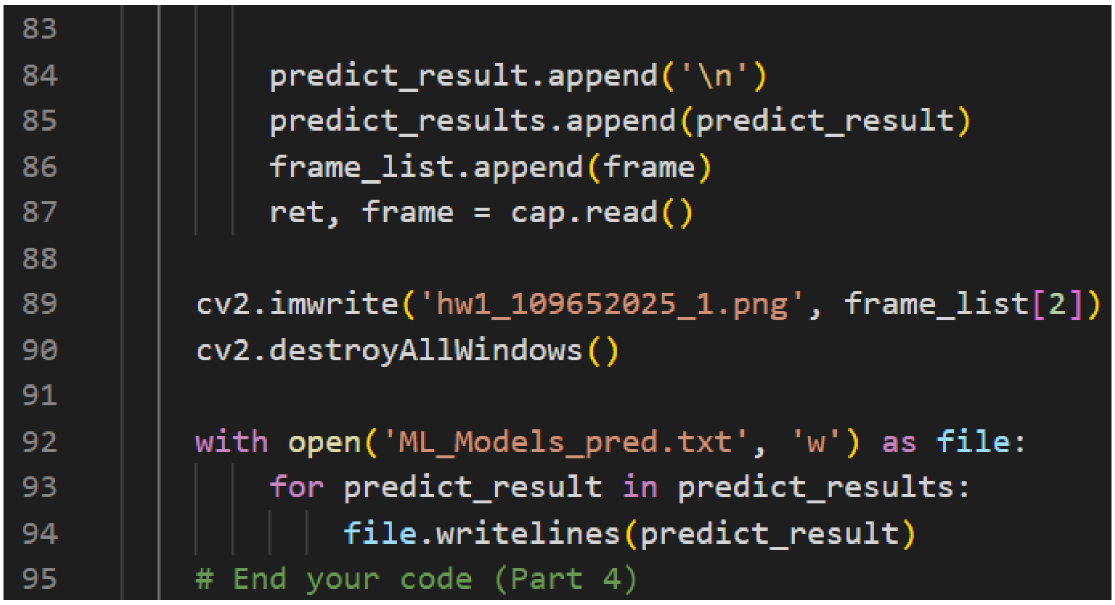
Accuracy: RF(0.98) > AB(0.96) > KNN(0.92)

F1-score: RF(0.98) > AB(0.96) > KNN(0.93)

Part 4: Detect car (15%)

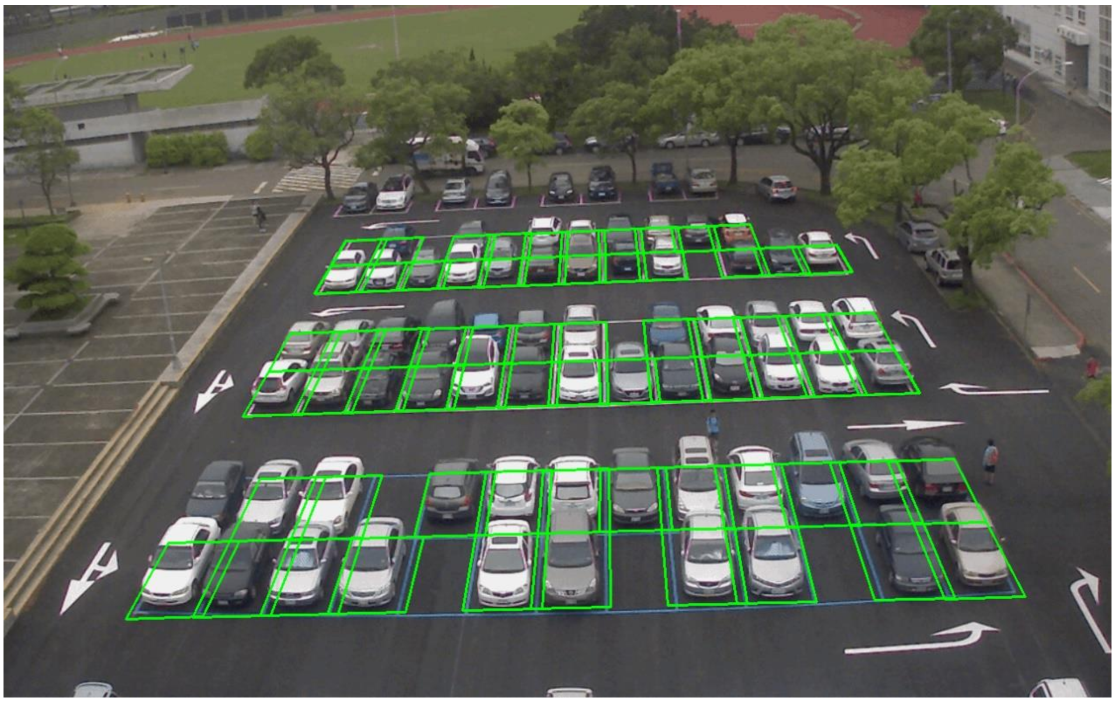


```python
# Begin your code (Part 4)
coordinates = []

with open('data/detect/detectData.txt', 'r', encoding="utf-8") as file:
    for line in file.readlines():
        coordinates.append(line.strip().split(' '))
coordinates = coordinates[1:]

cap = cv2.VideoCapture('data/detect/video.gif')
ret, frame = cap.read()
predict_results = []
frame_list = []

while ret:
    predict_result = []
    for coordinate in coordinates:
        x1, y1, x2, y2, x3, y3, x4, y4 = coordinate[0], coordinate[1], coordinate[2], coordinate[
            3], coordinate[4], coordinate[5], coordinate[6], coordinate[7]
        img = crop(x1, y1, x2, y2, x3, y3, x4, y4, frame)
        img = cv2.resize(img, (36, 16), interpolation=cv2.INTER_AREA)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        label = str(clf.classify([img.reshape(-1)]))
        predict_result.append(label + ' ')

        if label == '1':
            pts = np.array([[x1, y1], [x2, y2], [x4, y4],
                            [x3, y3], [x1, y1]], np.int32)
            frame = cv2.polylines(
                frame, [pts], False, (0, 255, 0), thickness=2)
```



```
            predict_result.append('\n')
            predict_results.append(predict_result)
            frame_list.append(frame)
            ret, frame = cap.read()

        cv2.imwrite('hw1_109652025_1.png', frame_list[2])
        cv2.destroyAllWindows()

        with open('ML_Models_pred.txt', 'w') as file:
            for predict_result in predict_results:
                file.writelines(predict_result)
        # End your code (Part 4)
```

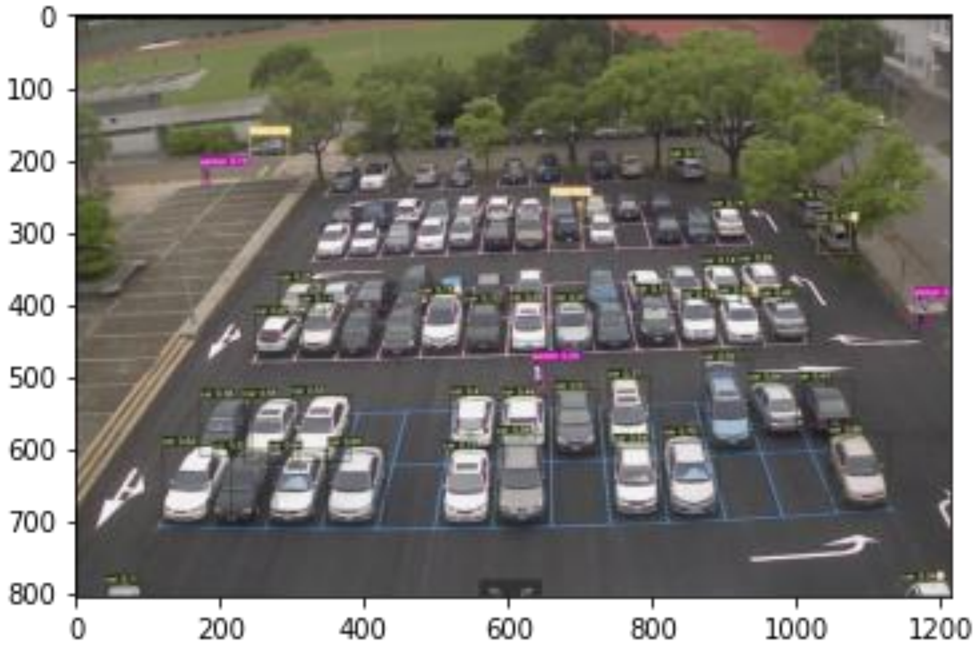
Read and store coordinates of parking areas. Read each frame in the video and use crop() to help extract parking space images. Convert them into 36 x 16 grayscale images and call clf.classify() to make predictions. Store the results to a list in the end.



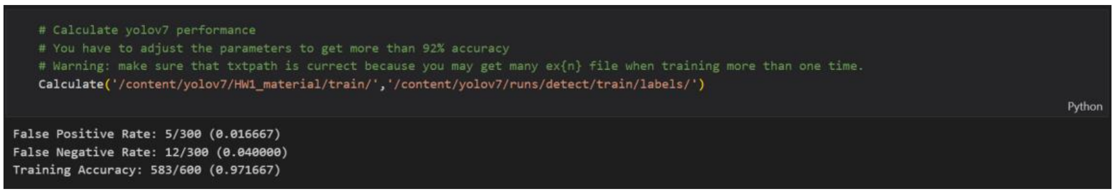
↑ The first frame with the bounding boxes.

Task B

Part 1: Load a pretrained model and directly apply it to the .png to detect the object.(10%)



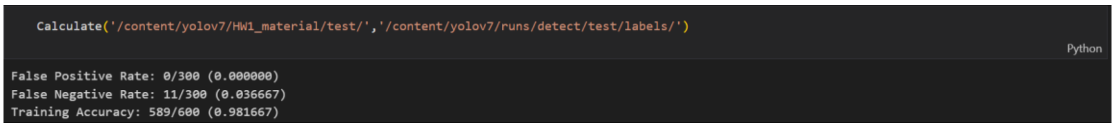
Part 2: Learn to fine tuning yolov7 model (10%)



```
# Calculate yolov7 performance
# You have to adjust the parameters to get more than 92% accuracy
# Warning: make sure that txtpath is currect because you may get many ex{n} file when training more than one time.
Calculate('/content/yolov7/HW1_material/train/','/content/yolov7/runs/detect/train/labels/')
```

Python

```
False Positive Rate: 5/300 (0.016667)
False Negative Rate: 12/300 (0.040000)
Training Accuracy: 583/600 (0.971667)
```



```
Calculate('/content/yolov7/HW1_material/test/','/content/yolov7/runs/detect/test/labels/')
```

Python

```
False Positive Rate: 0/300 (0.000000)
False Negative Rate: 11/300 (0.036667)
Training Accuracy: 589/600 (0.981667)
```

I consider the following parameters when training the model: batch size, epoch and input size. Typically, smaller batch size will perform better than larger batch size (by experience), but it takes more time (larger epoch) to train the model. For batch size = 32 and batch size = 16, our model didn't achieve the goal (accuracy > 0.92). For batch size < 8, we have to set epoch > 100 and thus the model will converge to the final result. When testing the model, we can adjust the confidence threshold (--conf). For example, if we observe that FP >> FN, we can increase the confidence threshold to see whether the performance is better. In the end, I set batch size = 4, epoch = 300, image size = 360 x 160 and confidence threshold = 0.1. The final result is displayed as above.