EVA AI-Relational Database System

Georgia Tech Database Group

Apr 17, 2023

OVERVIEW

1 What is EVA? 3

2 Key Features 5

3 Next Steps 7

4 Illustrative EVA Applications 9
- 4.1 Traffic Analysis Application using Object Detection Model . . . 9
- 4.2 MNIST Digit Recognition using Image Classification Model . . . 9
- 4.3 Movie Analysis Application using Face Detection + Emotion Classfication Models . . . 9

5 Community 11

6 Video Database System 13
- 6.1 Usability and Application Maintainability . . . 13
- 6.2 GPU Cost and Human Time . . . 13

7 Getting Started 15
- 7.1 Part 1: Install EVA . . . 15
- 7.2 Launch EVA server . . . 15
- 7.3 Part 2: Start a Jupyter Notebook Client . . . 15
- 7.4 Part 3: Register an user-defined function (UDF) . . . 16
- 7.5 Part 5: Start a Command Line Client . . . 17

8 List of Notebooks 19

9 Start EVA Server 21
- 9.1 Launch EVA server . . . 21

10 MNIST TUTORIAL 23
- 10.1 Start EVA server . . . 23
- 10.2 Downloading the videos . . . 23
- 10.3 Upload the video for analysis . . . 23
- 10.4 Visualize Video . . . 24
- 10.5 Create an user-defined function (UDF) for analyzing the frames . . . 24
- 10.6 Run the Image Classification UDF on video . . . 24
- 10.7 Visualize output of query on the video . . . 25

11 Object Detection Tutorial 27
- 11.1 Start EVA server . . . 27
- 11.2 Download the Videos . . . 27

11.3 Load the surveillance videos for analysis . . . 27
11.4 Visualize Video . . . 28
11.5 Register YOLO Object Detector an an User-Defined Function (UDF) in EVA . . . 28
11.6 Run Object Detector on the video . . . 29
11.7 Visualizing output of the Object Detector on the video . . . 29
11.8 Dropping an User-Defined Function (UDF) . . . 33

12 EMOTION ANALYSIS **35**
12.1 Start EVA Server . . . 35
12.2 Video Files . . . 35
12.3 Adding the video file to EVADB for analysis . . . 36
12.4 Visualize Video . . . 36
12.5 Create an user-defined function(UDF) for analyzing the frames . . . 36
12.6 Run the Face Detection UDF on video . . . 37
12.7 Run the Emotion Detection UDF on the outputs of the Face Detection UDF . . . 38

13 Custom Model Tutorial **43**
13.1 Start EVA server . . . 43
13.2 Download custom user-defined function (UDF), model, and video . . . 43
13.3 Load video for analysis . . . 44
13.4 Visualize Video . . . 44
13.5 Create GenderCNN and FaceDetector UDFs . . . 44
13.6 Run Face Detector on video . . . 45
13.7 Composing UDFs in a query . . . 46
13.8 Visualize Output . . . 47

14 EVA Query Language Reference **55**
14.1 LOAD . . . 55
14.2 SELECT . . . 56
14.3 EXPLAIN . . . 57
14.4 SHOW . . . 57
14.5 CREATE . . . 57
14.6 DROP . . . 58
14.7 INSERT . . . 58
14.8 DELETE . . . 59
14.9 RENAME . . . 59

15 User-Defined Functions **61**
15.1 Part 1: Writing a custom UDF . . . 61
15.2 Part 2: Registering and using the UDF in queries . . . 63

16 HuggingFace Models in EVA **65**
16.1 Creating UDF from HuggingFace . . . 65
16.2 Supported Tasks . . . 65

17 Configure GPU **67**

18 EVA Internals **69**
18.1 Path of a Query . . . 69
18.2 Topics . . . 70

19 Contributing **71**
19.1 Setting up the Development Environment . . . 71
19.2 Testing . . . 71
19.3 Submitting a Contribution . . . 72

19.4 Code Style . . . 72
19.5 Debugging . . . 72
19.6 Architecture Diagram . . . 74
19.7 Troubleshooting . . . 74

20 Debugging **75**
20.1 Setup debugger . . . 75
20.2 Alternative: Manually Setup Debugger for EVA . . . 76

21 Extending EVA **79**
21.1 Command Handler . . . 79
21.2 1. Parser . . . 79
21.3 2. Statement To Plan Convertor . . . 80
21.4 3. Plan Generator . . . 81
21.5 4. Plan Executor . . . 82
21.6 Additional Notes . . . 83

22 EVA Release Guide **85**
22.1 Part 1: Before You Start . . . 85
22.2 Part 2: Release Steps . . . 85

23 Packaging **89**
23.1 Models . . . 89
23.2 Datasets . . . 89

24 Versions **93**

AI-Relational Database System | SQL meets Deep Learning

CHAPTER

ONE

WHAT IS EVA?

EVA is an **open-source AI-relational database with first-class support for deep learning models**. It aims to support AI-powered database applications that operate on both structured (tables) and unstructured data (videos, text, podcasts, PDFs, etc.) with deep learning models.

EVA accelerates AI pipelines using a collection of optimizations inspired by relational database systems including function caching, sampling, and cost-based operator reordering. It comes with a wide range of models for analyzing unstructured data including image classification, object detection, OCR, face detection, etc. It is fully implemented in Python, and licensed under the Apache license.

EVA supports a AI-oriented query language for analysing unstructured data. Here are some illustrative applications:

- Examining the emotion palette of actors in a movie
- Analysing traffic flow at an intersection
- Classifying images based on their content
- Recogizing license plates
- Analysing toxicity of social media memes

If you are wondering why you might need a video database system, start with page on Video Database Systems. It describes how EVA lets users easily make use of deep learning models and how they can reduce money spent on inference on large image or video datasets.

The Getting Started page shows how you can use EVA for different computer vision tasks, and how you can easily extend EVA to support your custom deep learning model in the form of user-defined functions.

The User Guides section contains Jupyter Notebooks that demonstrate how to use various features of EVA. Each notebook includes a link to Google Colab, where you can run the code by yourself.

CHAPTER

TWO

KEY FEATURES

1. With EVA, you can **easily combine SQL and deep learning models to build next-generation database applications**. EVA treats deep learning models as functions similar to traditional SQL functions like SUM().
2. EVA is **extensible by design**. You can write an **user-defined function** (UDF) that wraps arounds your custom deep learning model. In fact, all the built-in models that are included in EVA are written as user-defined functions.
3. EVA comes with a collection of **built-in sampling, caching, and filtering optimizations** inspired by relational database systems. These optimizations help **speed up queries on large datasets and save money spent on model inference**.

CHAPTER

THREE

NEXT STEPS

Getting Started A step-by-step guide to installing EVA and running queries

Query Language List of all the query commands supported by EVA

User Defined Functions A step-by-step tour of registering a user defined function that wraps around a custom deep learning model

CHAPTER

FOUR

ILLUSTRATIVE EVA APPLICATIONS

4.1 Traffic Analysis Application using Object Detection Model

4.2 MNIST Digit Recognition using Image Classification Model

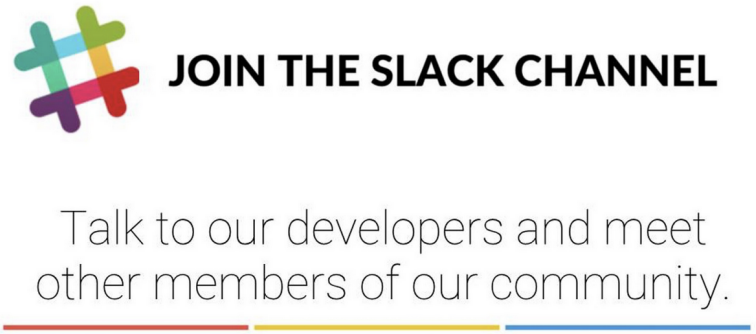
4.3 Movie Analysis Application using Face Detection + Emotion Classfication Models

CHAPTER

FIVE

COMMUNITY

Join the EVA community on Slack to ask questions and to share your ideas for improving EVA.



CHAPTER

SIX

VIDEO DATABASE SYSTEM

Over the last decade, deep learning models have radically changed the world of computer vision. They are accurate on a variety of tasks ranging from image classification to emotion detection. However, there are two challenges that prevent a lot of users from benefiting from these models.

6.1 Usability and Application Maintainability

To use a vision model, the user must do a lot of imperative programming across low-level libraries, like OpenCV and PyTorch. This is a tedious process that often leads to a complex program or Jupyter Notebook that glues together these libraries to accomplish the given task. This programming complexity **prevents a lot of people who are experts in other domains from benefiting from these models**.

Historically, database systems have been successful because the **query language is simple enough** in its basic structure that users without prior experience are able to learn a usable subset of the language on their first sitting. EVA supports a declarative SQL-like query language, called EVAQL, that is designed to make it easier for users to leverage these models. With this query language, the user may **compose multiple models in a single query** to accomplish complicated tasks with **minimal programming**.

Here is a illustrative query that examines the emotions of actors in a movie by leveraging multiple deep learning models that take care of detecting faces and analyzing the emotions of the detected bounding boxes:

```
SELECT id, bbox, EmotionDetector(Crop(data, bbox))
FROM Interstellar
     JOIN LATERAL UNNEST(FaceDetector(data)) AS Face(bbox, conf)
WHERE id < 15;
```

By using a declarative language, the complexity of the program or Jupyter Notebook is significantly reduced. This in turn leads to more **maintainable code** that allows users to build on top of each other's queries.

6.2 GPU Cost and Human Time

From a cost standpoint, it is very expensive to run these deep learning models on large image or video datasets. For example, the state-of-the-art object detection model takes multiple GPU-decades to process just a year's worth of videos from a single traffic monitoring camera. Besides the money spent on hardware, this also increases the time that the user spends waiting for the model inference process to finish.

EVA **automatically** optimizes the queries to **reduce inference cost and query execution time** using its Cascades-style query optimizer. EVA's optimizer is tailored for video analytics. The Cascades-style extensible query optimization framework has worked very well for several decades in SQL database systems. Query optimization is one of the signature components of database systems — the bridge that connects the declarative query language to efficient execution.

CHAPTER

SEVEN

GETTING STARTED

7.1 Part 1: Install EVA

EVA supports Python (versions >= 3.7). To install EVA, we recommend using the pip package manager:

```
pip install evadb
```

7.2 Launch EVA server

EVA is based on a client-server architecture. To launch the EVA server, run the following command on the terminal:

```
eva_server &
```

7.3 Part 2: Start a Jupyter Notebook Client

Here is an illustrative Jupyter notebook focusing on MNIST image classification using EVA. The notebook works on Google Colab.

7.3.1 Connect to the EVA server

To connect to the EVA server in the notebook, use the following Python code:

```
# allow nested asyncio calls for client to connect with server
import nest_asyncio
nest_asyncio.apply()
from eva.server.db_api import connect

# hostname and port of the server where EVA is running
connection = connect(host = '0.0.0.0', port = 5432)

# cursor allows the notebook client to send queries to the server
cursor = connection.cursor()
```

7.3.2 Load video for analysis

Download the MNIST video for analysis.

```
!wget -nc https://www.dropbox.com/s/yxljxz6zxoqu54v/mnist.mp4
```

Use the LOAD statement is used to load a video onto a table in EVA server.

```
cursor.execute('LOAD VIDEO "mnist.mp4" INTO MNISTVideoTable;')
response = cursor.fetch_all()
print(response)
```

7.4 Part 3: Register an user-defined function (UDF)

User-defined functions allow us to combine SQL with deep learning models. These functions wrap around deep learning models.

Download the user-defined function for classifying MNIST images.

```
!wget -nc https://raw.githubusercontent.com/georgia-tech-db/eva/master/tutorials/apps/
↪mnist/eva_mnist_udf.py
```

```
cursor.execute("""CREATE UDF IF NOT EXISTS MnistCNN
                  INPUT  (data NDARRAY (3, 28, 28))
                  OUTPUT (label TEXT(2))
                  TYPE  Classification
                  IMPL  'eva_mnist_udf.py';
                """)
response = cursor.fetch_all()
print(response)
```

7.4.1 Run a query using the newly registered UDF!

```
cursor.execute("""SELECT data, MnistCNN(data).label
                  FROM MNISTVideoTable
                  WHERE id = 30;""")
response = cursor.fetch_all()
```

7.4.2 Visualize the output

The output of the query is visualized in the notebook.

7.5 Part 5: Start a Command Line Client

Besides the notebook interface, EVA also exports a command line interface for querying the server. This interface allows for quick querying from the terminal:

```
>>> eva_client
eva=# LOAD VIDEO "mnist.mp4" INTO MNISTVid;
@status: ResponseStatus.SUCCESS
@batch:

0 Video successfully added at location: mnist.p4
@query_time: 0.045

eva=# SELECT id, data FROM MNISTVid WHERE id < 1000;
@status: ResponseStatus.SUCCESS
@batch:
            mnistvid.id     mnistvid.data
    0          0             [[[ 0 2 0]\n [0 0 0]\n...
    1          1             [[[ 2 2 0]\n [1 1 0]\n...
    2          2             [[[ 2 2 0]\n [1 2 2]\n...
    ..       ...
    997        997           [[[ 0 2 0]\n [0 0 0]\n...
    998        998           [[[ 0 2 0]\n [0 0 0]\n...
    999        999           [[[ 2 2 0]\n [1 1 0]\n...

[1000 rows x 2 columns]
@query_time: 0.216

eva=# exit
```

CHAPTER

EIGHT

LIST OF NOTEBOOKS

This section contains Jupyter Notebooks that demonstrate how to use various features of EVA. Each notebook includes a link to Google Colab, where you can run the code by yourself.

- Starting EVA Server
- MNIST Image Classification
- Object Detection
- Emotion Analysis
- Using a Custom Model

CHAPTER

NINE

START EVA SERVER

9.1 Launch EVA server

We use this notebook for launching the EVA server.

```
## Install EVA package if needed
# %pip install "evadb[dev]" --quiet

import os
import time
from psutil import process_iter
from signal import SIGTERM
import re
import itertools

def shell(command):
    print(command)
    os.system(command)

def stop_eva_server():
    for proc in process_iter():
        if proc.name() == "eva_server":
            proc.send_signal(SIGTERM)

def launch_eva_server():
    # Stop EVA server if it is running
    stop_eva_server()

    os.environ['GPU_DEVICES'] = '0'

    # Start EVA server
    shell("nohup eva_server >> eva.log 2>&1 &")

    last_few_lines_count = 3
    try:
        with open('eva.log', 'r') as f:
            for lines in itertools.zip_longest(*[f]*last_few_lines_count):
                print(lines)
    except FileNotFoundError:
        pass
```

(continues on next page)

(continued from previous page)

```
    # Wait for server to start
    time.sleep(10)

def connect_to_server():
    from eva.server.db_api import connect
    %pip install nest_asyncio --quiet
    import nest_asyncio
    nest_asyncio.apply()

    # Connect client with server
    connection = connect(host = '127.0.0.1', port = 5432)
    cursor = connection.cursor()

    return cursor

# Launch server
launch_eva_server()
```

```
nohup eva_server > eva.log 2>&1 &
```

CHAPTER

TEN

MNIST TUTORIAL

10.1 Start EVA server

We are reusing the start server notebook for launching the EVA server.

```
!wget -nc "https://raw.githubusercontent.com/georgia-tech-db/eva/master/tutorials/00-
↪start-eva-server.ipynb"
%run 00-start-eva-server.ipynb
cursor = connect_to_server()
```

```
File '00-start-eva-server.ipynb' already there; not retrieving.

nohup eva_server > eva.log 2>&1 &
Note: you may need to restart the kernel to use updated packages.
```

10.2 Downloading the videos

```
# Getting MNIST as a video
!wget -nc https://www.dropbox.com/s/yxljxz6zxoqu54v/mnist.mp4
# Getting a udf
!wget -nc https://raw.githubusercontent.com/georgia-tech-db/eva/master/tutorials/apps/
↪mnist/eva_mnist_udf.py
```

```
File 'mnist.mp4' already there; not retrieving.

File 'eva_mnist_udf.py' already there; not retrieving.
```

10.3 Upload the video for analysis

```
cursor.execute('DROP TABLE MNISTVid')
response = cursor.fetch_all()
print(response)
cursor.execute('LOAD VIDEO "mnist.mp4" INTO MNISTVid')
response = cursor.fetch_all()
print(response)
```

```
@status: ResponseStatus.SUCCESS
@batch:
                                        0
0  Table Successfully dropped: MNISTVid
@query_time: 0.26187248099995486
@status: ResponseStatus.SUCCESS
@batch:
                               0
0  Number of loaded VIDEO: 1
@query_time: 1.0106322069996168
```

10.4 Visualize Video

```
from IPython.display import Video
Video("mnist.mp4", embed=True)
```

```
<IPython.core.display.Video object>
```

10.5 Create an user-defined function (UDF) for analyzing the frames

```
cursor.execute("""CREATE UDF IF NOT EXISTS MnistCNN
                  INPUT  (data NDARRAY (3, 28, 28))
                  OUTPUT (label TEXT(2))
                  TYPE  Classification
                  IMPL  'eva_mnist_udf.py'
        """)
response = cursor.fetch_all()
print(response)
```

```
@status: ResponseStatus.SUCCESS
@batch:
                                                  0
0  UDF MnistCNN successfully added to the database.
@query_time: 4.144361197000762
```

10.6 Run the Image Classification UDF on video

```
cursor.execute("""SELECT data, MnistCNN(data).label
                  FROM MNISTVid
                  WHERE id = 30 OR id = 50 OR id = 70 OR id = 0 OR id = 140""")
response = cursor.fetch_all()
print(response.as_df())
```

```
                                    mnistvid.data mnistcnn.label
0  [[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], ...              6
```

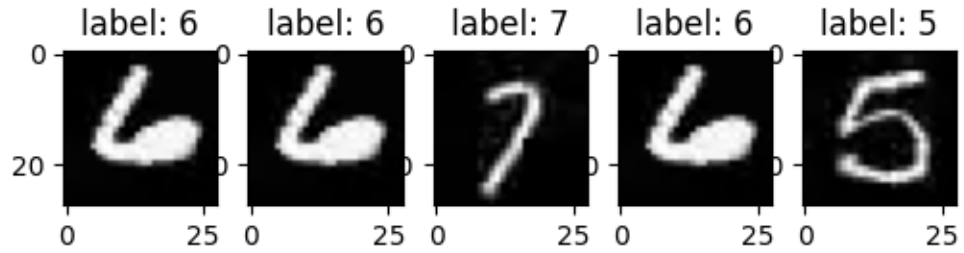
(continues on next page)

(continued from previous page)

```
1  [[[2, 2, 2], [2, 2, 2], [2, 2, 2], [2, 2, 2], ...              2
2  [[[13, 13, 13], [2, 2, 2], [2, 2, 2], [13, 13,...              3
3  [[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], ...              7
4  [[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], ...              5
```

10.7 Visualize output of query on the video

```
# !pip install matplotlib
import matplotlib.pyplot as plt
import numpy as np

# create figure (fig), and array of axes (ax)
fig, ax = plt.subplots(nrows=1, ncols=5, figsize=[6,8])

df = response.batch.frames
for axi in ax.flat:
    idx = np.random.randint(len(df))
    img = df['mnistvid.data'].iloc[idx]
    label = df['mnistcnn.label'].iloc[idx]
    axi.imshow(img)

    axi.set_title(f'label: {label}')

plt.show()
```



CHAPTER

ELEVEN

OBJECT DETECTION TUTORIAL

11.1 Start EVA server

We are reusing the start server notebook for launching the EVA server.

```
!wget -nc "https://raw.githubusercontent.com/georgia-tech-db/eva/master/tutorials/00-
↪start-eva-server.ipynb"
%run 00-start-eva-server.ipynb
cursor = connect_to_server()
```

```
File '00-start-eva-server.ipynb' already there; not retrieving.

nohup eva_server > eva.log 2>&1 &
Note: you may need to restart the kernel to use updated packages.
```

11.2 Download the Videos

```
# Getting the video files
!wget -nc https://www.dropbox.com/s/k00wge9exwkfxz6/ua_detrac.mp4?raw=1 -O ua_detrac.mp4
# Getting the Yolo object detector
!wget -nc https://raw.githubusercontent.com/georgia-tech-db/eva/master/eva/udfs/yolo_
↪object_detector.py
```

```
File ‘ua_detrac.mp4’ already there; not retrieving.
File ‘yolo_object_detector.py’ already there; not retrieving.
```

11.3 Load the surveillance videos for analysis

11.3.1 We use regular expression to load all the videos into the table

```
cursor.execute('DROP TABLE ObjectDetectionVideos')
response = cursor.fetch_all()
print(response)
cursor.execute('LOAD VIDEO "ua_detrac.mp4" INTO ObjectDetectionVideos;')
```

(continues on next page)

(continued from previous page)

```
response = cursor.fetch_all()
print(response)
```

```
@status: ResponseStatus.SUCCESS
@batch:
                                                    0
0  Table Successfully dropped: ObjectDetectionVideos
@query_time: 0.3437608620006358
@status: ResponseStatus.SUCCESS
@batch:
                             0
0  Number of loaded VIDEO: 1
@query_time: 1.8837439379994976
```

11.4 Visualize Video

```
from IPython.display import Video
Video("ua_detrac.mp4", embed=True)
```

```
<IPython.core.display.Video object>
```

11.5 Register YOLO Object Detector an an User-Defined Function (UDF) in EVA

```
cursor.execute("""CREATE UDF IF NOT EXISTS YoloV5
      INPUT  (frame NDARRAY UINT8(3, ANYDIM, ANYDIM))
      OUTPUT (labels NDARRAY STR(ANYDIM), bboxes NDARRAY FLOAT32(ANYDIM, 4),
                scores NDARRAY FLOAT32(ANYDIM))
      TYPE  Classification
      IMPL  'yolo_object_detector.py';
      """)
response = cursor.fetch_all()
print(response)
```

```
@status: ResponseStatus.SUCCESS
@batch:
                                          0
0  UDF YoloV5 already exists, nothing added.
@query_time: 0.012568520000058925
```

11.6 Run Object Detector on the video

```
cursor.execute("""SELECT id, YoloV5(data)
                  FROM ObjectDetectionVideos
                  WHERE id < 20""")
response = cursor.fetch_all()
response.as_df()
```

```
    objectdetectionvideos.id  ...                                    yolov5.scores
0                          0  ...  [0.9019389748573303, 0.8878239393234253, 0.854...
1                          1  ...  [0.898006021976471, 0.8685559630393982, 0.8364...
2                          2  ...  [0.8956703543663025, 0.8518660068511963, 0.848...
3                          3  ...  [0.8803831338882446, 0.8661114573478699, 0.849...
4                          4  ...  [0.8975156545639038, 0.8809235692024231, 0.843...
5                          5  ...  [0.9156808257102966, 0.8994483351707458, 0.846...
6                          6  ...  [0.9120900630950928, 0.9068282842636108, 0.841...
7                          7  ...  [0.9046719670295715, 0.8707321882247925, 0.845...
8                          8  ...  [0.8893280625343323, 0.8552687168121338, 0.853...
9                          9  ...  [0.8876321911811829, 0.855767548084259, 0.8532...
10                        10  ...  [0.8991596102714539, 0.8353574872016907, 0.830...
11                        11  ...  [0.9085389971733093, 0.8469472527503967, 0.838...
12                        12  ...  [0.9089045524597168, 0.8619462847709656, 0.854...
13                        13  ...  [0.8987026214599609, 0.88194340467453, 0.85846...
14                        14  ...  [0.8915325403213501, 0.8549790978431702, 0.854...
15                        15  ...  [0.8875617980957031, 0.8631888031959534, 0.847...
16                        16  ...  [0.8969656825065613, 0.863024890422821, 0.8239...
17                        17  ...  [0.8984966278076172, 0.8612021207809448, 0.834...
18                        18  ...  [0.8983429074287415, 0.8600167036056519, 0.840...
19                        19  ...  [0.8998422026634216, 0.8530007004737854, 0.848...

[20 rows x 4 columns]
```

11.7 Visualizing output of the Object Detector on the video

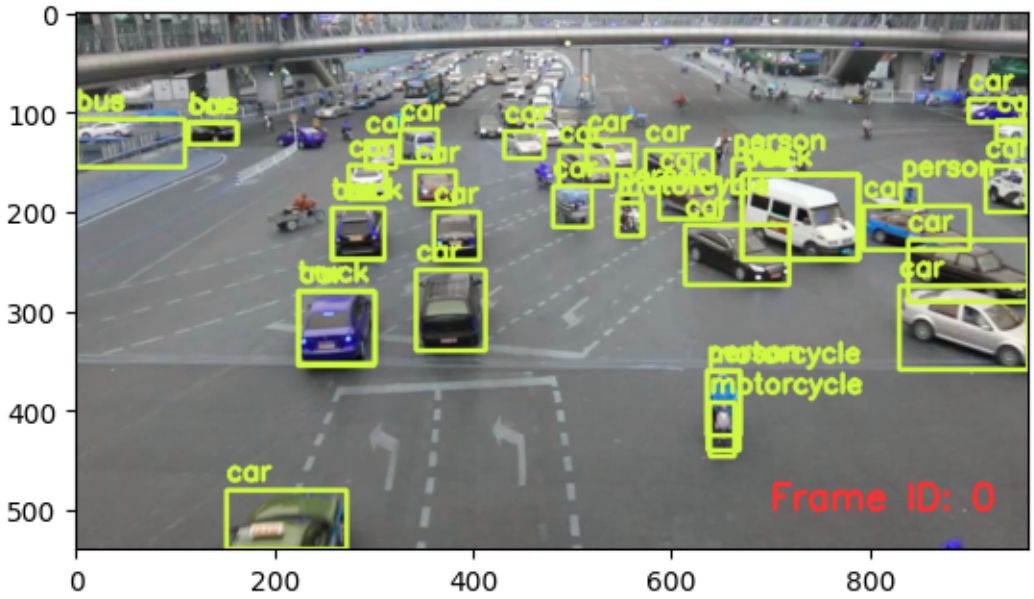
```
import cv2
from pprint import pprint
from matplotlib import pyplot as plt

def annotate_video(detections, input_video_path, output_video_path):
    color1=(207, 248, 64)
    color2=(255, 49, 49)
    thickness=4

    vcap = cv2.VideoCapture(input_video_path)
    width = int(vcap.get(3))
    height = int(vcap.get(4))
    fps = vcap.get(5)
    fourcc = cv2.VideoWriter_fourcc('m', 'p', '4', 'v') #codec
    video=cv2.VideoWriter(output_video_path, fourcc, fps, (width,height))
```

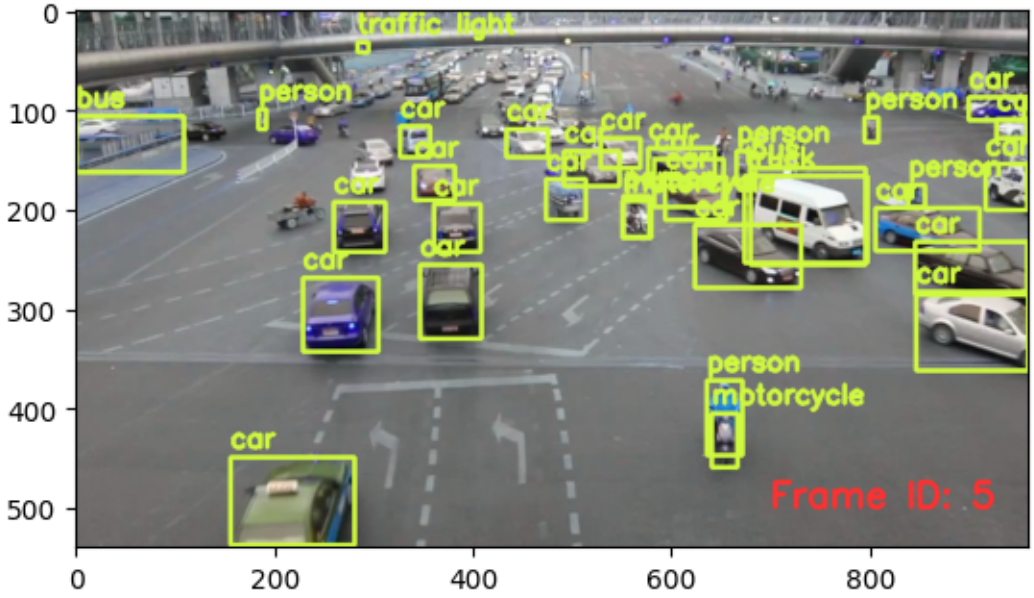
(continues on next page)

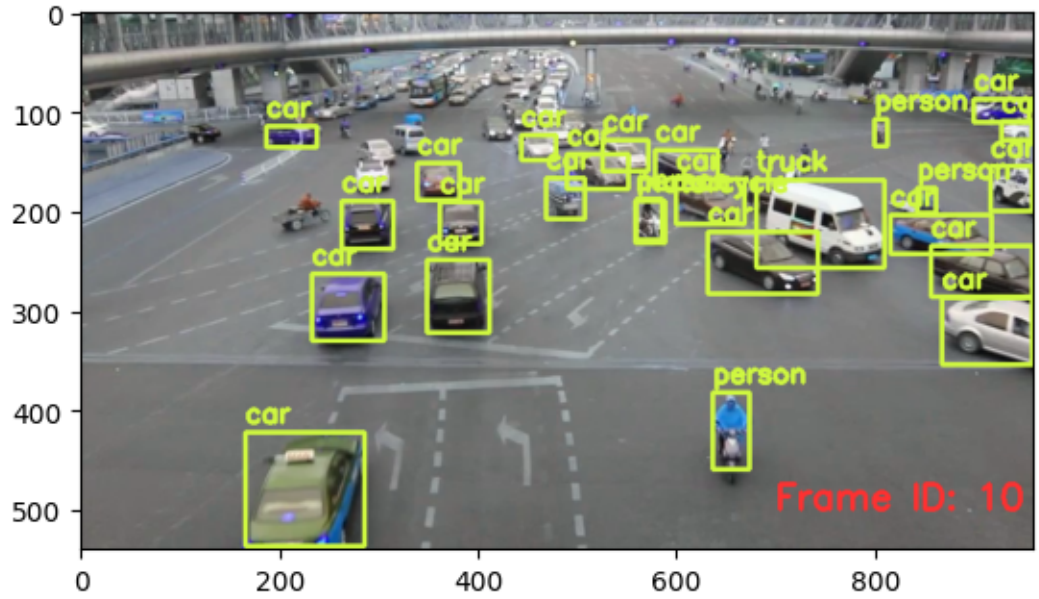
(continued from previous page)

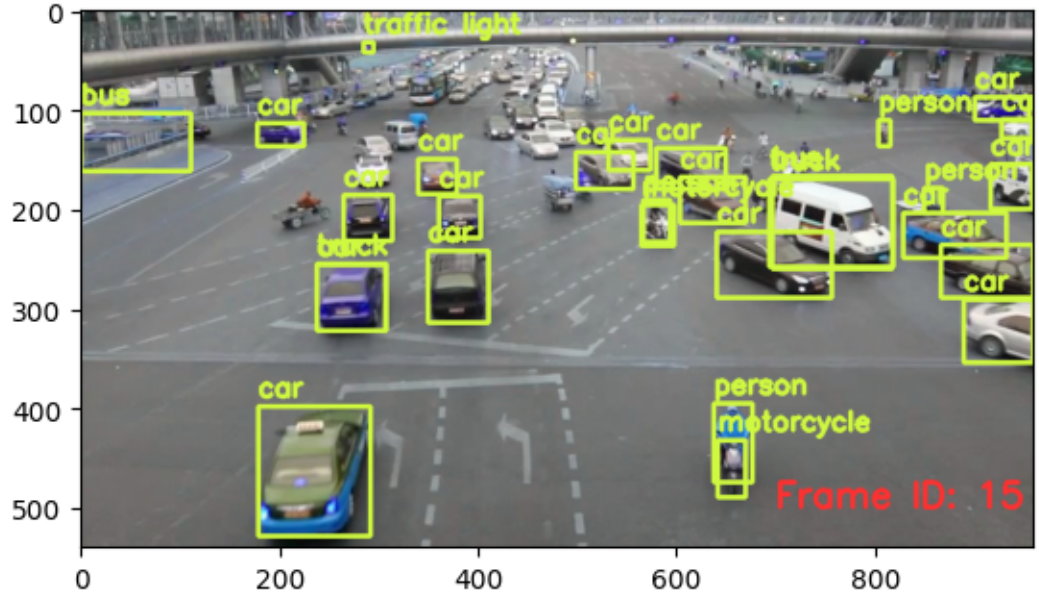
```
    frame_id = 0
    # Capture frame-by-frame
    # ret = 1 if the video is captured; frame is the image
    ret, frame = vcap.read()

    while ret:
        df = detections
        df = df[['yolov5.bboxes', 'yolov5.labels']][df.index == frame_id]
        if df.size:
            dfLst = df.values.tolist()
            for bbox, label in zip(dfLst[0][0], dfLst[0][1]):
                x1, y1, x2, y2 = bbox
                x1, y1, x2, y2 = int(x1), int(y1), int(x2), int(y2)
                # object bbox
                frame=cv2.rectangle(frame, (x1, y1), (x2, y2), color1, thickness)
                # object label
                cv2.putText(frame, label, (x1, y1-10), cv2.FONT_HERSHEY_SIMPLEX, 0.9,␣
↪color1, thickness)
                # frame label
                cv2.putText(frame, 'Frame ID: ' + str(frame_id), (700, 500), cv2.FONT_
↪HERSHEY_SIMPLEX, 1.2, color2, thickness)
            video.write(frame)

            # Stop after twenty frames (id < 20 in previous query)
            if frame_id == 20:
                break

            # Show every fifth frame
            if frame_id % 5 == 0:
                plt.imshow(frame)
                plt.show()


        frame_id+=1
        ret, frame = vcap.read()

    video.release()
    vcap.release()
```

```
from ipywidgets import Video, Image
input_path = 'ua_detrac.mp4'
output_path = 'video.mp4'

dataframe = response.as_df()
annotate_video(dataframe, input_path, output_path)
Video.from_file(output_path)
```


Frame ID: 0


Frame ID: 5






```
Video(value=b'\x00\x00\x00\x1cftypisom\x00\x00\x02\x00isomiso2mp41\x00\x00\x00\x08free\
↪x00\x0c\x01N...')
```

11.8 Dropping an User-Defined Function (UDF)

```
cursor.execute("DROP UDF YoloV5;")
response = cursor.fetch_all()
print(response)
```

```
@status: ResponseStatus.SUCCESS
@batch:
                                    0
0  UDF YoloV5 successfully dropped
@query_time: 0.16062777899996966
```

CHAPTER

TWELVE

EMOTION ANALYSIS

12.1 Start EVA Server

We are reusing the start server notebook for launching the EVA server

```
!wget -nc "https://raw.githubusercontent.com/georgia-tech-db/eva/master/tutorials/00-
↪start-eva-server.ipynb"
%run 00-start-eva-server.ipynb
cursor = connect_to_server()
```

```
File '00-start-eva-server.ipynb' already there; not retrieving.

nohup eva_server > eva.log 2>&1 &
Note: you may need to restart the kernel to use updated packages.
```

12.2 Video Files

getting some video files to test

```
# A video of a happy person
!wget -nc https://www.dropbox.com/s/gzfhwmib7u804zy/defhappy.mp4?raw=1 -O defhappy.mp4

# Adding Emotion detection
!wget -nc https://raw.githubusercontent.com/georgia-tech-db/eva/master/eva/udfs/emotion_
↪detector.py

# Adding Face Detector
!wget -nc https://raw.githubusercontent.com/georgia-tech-db/eva/master/eva/udfs/face_
↪detector.py
```

```
File 'defhappy.mp4' already there; not retrieving.
File 'emotion_detector.py' already there; not retrieving.

File 'face_detector.py' already there; not retrieving.
```

12.3 Adding the video file to EVADB for analysis

```
cursor.execute('DROP TABLE HAPPY')
response = cursor.fetch_all()
print(response)
cursor.execute('LOAD VIDEO "defhappy.mp4" INTO HAPPY')
response = cursor.fetch_all()
print(response)
```

```
@status: ResponseStatus.SUCCESS
@batch:
                                      0
0  Table Successfully dropped: HAPPY
@query_time: 0.3072362639995845
@status: ResponseStatus.SUCCESS
@batch:
                               0
0  Number of loaded VIDEO: 1
@query_time: 2.031892749999315
```

12.4 Visualize Video

```
from IPython.display import Video
Video("defhappy.mp4", height=450, width=800, embed=True)
```

```
<IPython.core.display.Video object>
```

12.5 Create an user-defined function(UDF) for analyzing the frames

```
cursor.execute("""CREATE UDF IF NOT EXISTS EmotionDetector
        INPUT (frame NDARRAY UINT8(3, ANYDIM, ANYDIM))
        OUTPUT (labels NDARRAY STR(ANYDIM), scores NDARRAY FLOAT32(ANYDIM))
        TYPE  Classification IMPL 'emotion_detector.py';
        """)
response = cursor.fetch_all()
print(response)
cursor.execute("""CREATE UDF IF NOT EXISTS FaceDetector
                  INPUT  (frame NDARRAY UINT8(3, ANYDIM, ANYDIM))
                  OUTPUT (bboxes NDARRAY FLOAT32(ANYDIM, 4),
                          scores NDARRAY FLOAT32(ANYDIM))
                  TYPE  FaceDetection
                  IMPL  'face_detector.py';
        """)
response = cursor.fetch_all()
print(response)
```

```
@status: ResponseStatus.SUCCESS
@batch:
                                                                    0
0  UDF EmotionDetector successfully added to the database.
@query_time: 4.813677786999506
@status: ResponseStatus.SUCCESS
@batch:
                                                                0
0  UDF FaceDetector successfully added to the database.
@query_time: 0.46618744500028697
```

12.6 Run the Face Detection UDF on video

```
cursor.execute("""SELECT id, FaceDetector(data)
                  FROM HAPPY WHERE id<10""")
response = cursor.fetch_all()
response.as_df()
```

```
   happy.id                          facedetector.bboxes  \
0         0  [[502, 94, 762, 435], [238, 296, 325, 398]]
1         1                        [[501, 96, 763, 435]]
2         2                        [[504, 97, 766, 437]]
3         3                        [[498, 90, 776, 446]]
4         4                        [[496, 99, 767, 444]]
5         5  [[499, 87, 777, 448], [236, 305, 324, 407]]
6         6                        [[500, 89, 778, 449]]
7         7                        [[501, 89, 781, 452]]
8         8                        [[503, 90, 783, 450]]
9         9                        [[508, 87, 786, 447]]

        facedetector.scores
0  [0.99990165, 0.79820216]
1                [0.999918]
2               [0.9999138]
3              [0.99996686]
4               [0.9999982]
5   [0.9999136, 0.83697325]
6               [0.9999131]
7               [0.9999124]
8              [0.99994683]
9                [0.999949]
```

12.7 Run the Emotion Detection UDF on the outputs of the Face Detection UDF

```
cursor.execute("""SELECT id, bbox, EmotionDetector(Crop(data, bbox))
                  FROM HAPPY JOIN LATERAL  UNNEST(FaceDetector(data)) AS Face(bbox,␣
↪conf)
                  WHERE id < 15;""")
response = cursor.fetch_all()
response.as_df()
```

```
    happy.id             Face.bbox emotiondetector.labels  \
0          0   [502, 94, 762, 435]                   happy
1          0  [238, 296, 325, 398]                 neutral
2          1   [501, 96, 763, 435]                   happy
3          2   [504, 97, 766, 437]                   happy
4          3   [498, 90, 776, 446]                   happy
5          4   [496, 99, 767, 444]                   happy
6          5   [499, 87, 777, 448]                   happy
7          5  [236, 305, 324, 407]                 neutral
8          6   [500, 89, 778, 449]                   happy
9          7   [501, 89, 781, 452]                   happy
10         8   [503, 90, 783, 450]                   happy
11         9   [508, 87, 786, 447]                   happy
12        10   [505, 86, 788, 452]                   happy
13        10  [235, 309, 322, 411]                 neutral
14        11   [514, 85, 790, 454]                   happy
15        12   [514, 86, 790, 454]                   happy
16        13   [515, 87, 790, 454]                   happy
17        14   [516, 86, 792, 455]                   happy

    emotiondetector.scores
0                 0.999642
1                 0.780949
2                 0.999644
3                 0.999668
4                 0.999654
5                 0.999649
6                 0.999710
7                 0.760779
8                 0.999671
9                 0.999671
10                0.999689
11                0.999691
12                0.999729
13                0.407872
14                0.999745
15                0.999729
16                0.999718
17                0.999739
```

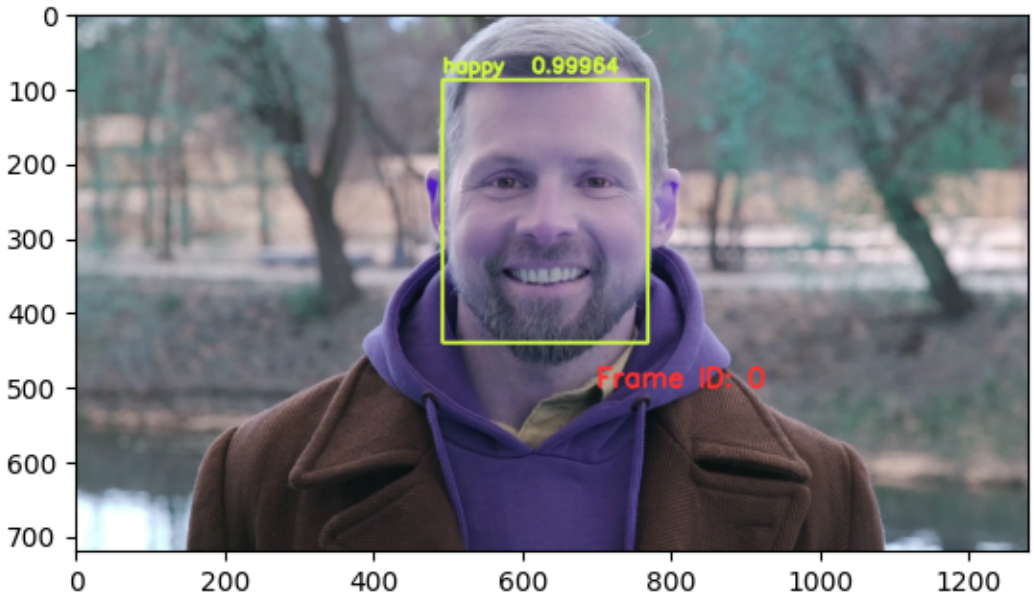
```
import cv2
from pprint import pprint
from matplotlib import pyplot as plt

def annotate_video(detections, input_video_path, output_video_path):
    color1=(207, 248, 64)
    color2=(255, 49, 49)
    thickness=4

    vcap = cv2.VideoCapture(input_video_path)
    width = int(vcap.get(3))
    height = int(vcap.get(4))
    fps = vcap.get(5)
    fourcc = cv2.VideoWriter_fourcc('m', 'p', '4', 'v') #codec
    video=cv2.VideoWriter(output_video_path, fourcc, fps, (width,height))

    frame_id = 0
    # Capture frame-by-frame
    # ret = 1 if the video is captured; frame is the image
    ret, frame = vcap.read()

    while ret:
        df = detections
        df = df[['Face.bbox', 'emotiondetector.labels', 'emotiondetector.scores']][df.
↪index == frame_id]
        if df.size:

            x1, y1, x2, y2 = df['Face.bbox'].values[0]
            label = df['emotiondetector.labels'].values[0]
            score = df['emotiondetector.scores'].values[0]
            x1, y1, x2, y2 = int(x1), int(y1), int(x2), int(y2)
            # object bbox
            frame=cv2.rectangle(frame, (x1, y1), (x2, y2), color1, thickness)
            # object label
            cv2.putText(frame, label, (x1, y1-10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, color1,
↪ thickness)
            # object score
            cv2.putText(frame, str(round(score, 5)), (x1+120, y1-10), cv2.FONT_HERSHEY_
↪SIMPLEX, 0.9, color1, thickness)
            # frame label
            cv2.putText(frame, 'Frame ID: ' + str(frame_id), (700, 500), cv2.FONT_
↪HERSHEY_SIMPLEX, 1.2, color2, thickness)

            video.write(frame)
            # Show every fifth frame
            if frame_id % 5 == 0:
                plt.imshow(frame)
                plt.show()

        frame_id+=1
        ret, frame = vcap.read()

    video.release()
```

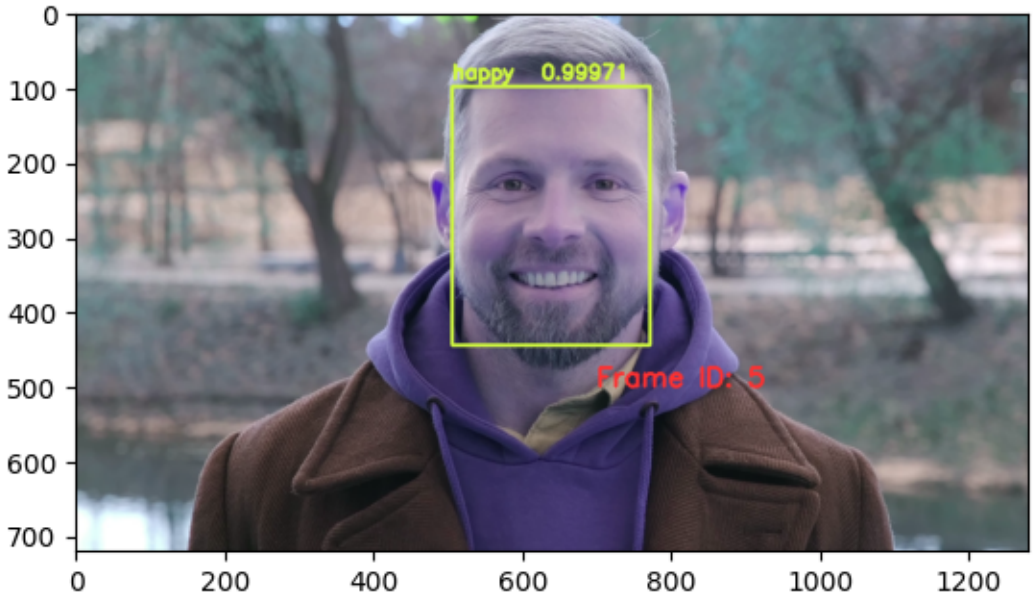
(continues on next page)

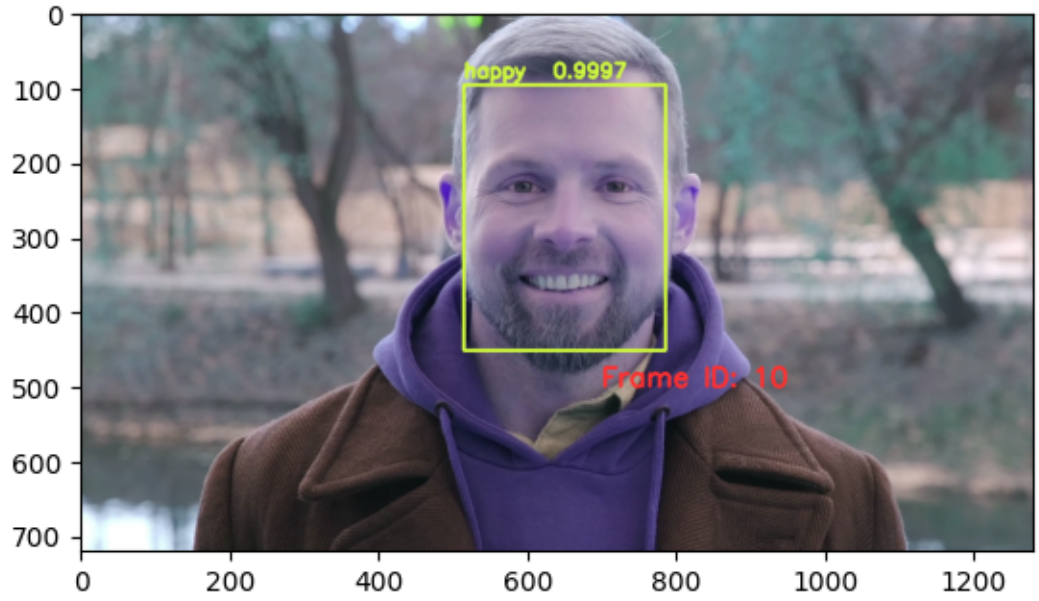
(continued from previous page)

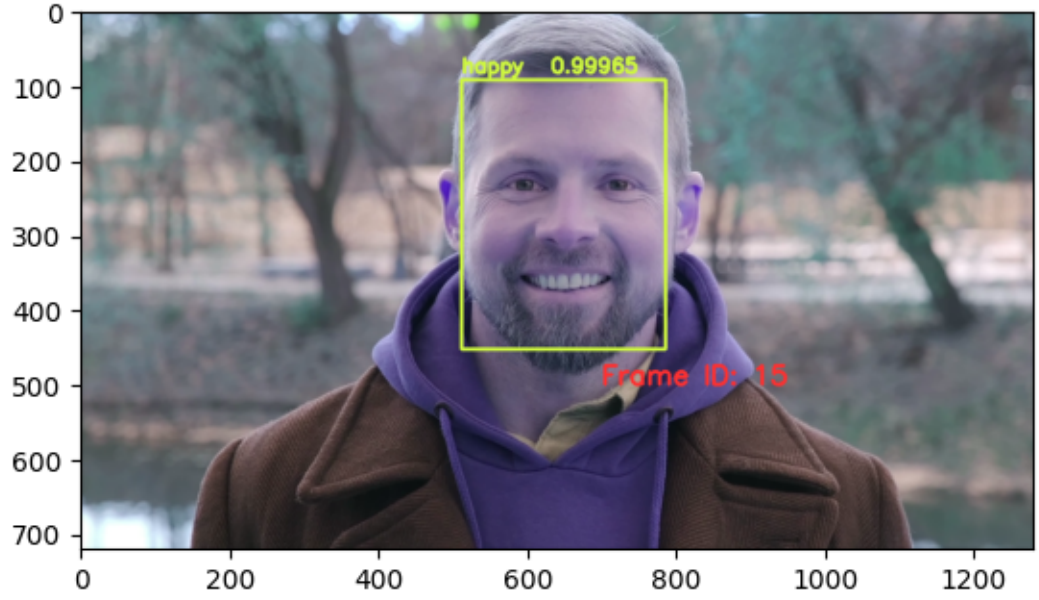
```
    vcap.release()
```

```
from ipywidgets import Video, Image
input_path = 'defhappy.mp4'
output_path = 'video.mp4'

dataframe = response.as_df()
annotate_video(dataframe, input_path, output_path)
```






happy 0.9997
Frame ID: 10
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


happy 0.99965
Frame ID: 15
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

CHAPTER

THIRTEEN

CUSTOM MODEL TUTORIAL

13.1 Start EVA server

We are reusing the start server notebook for launching the EVA server.

```
!wget -nc "https://raw.githubusercontent.com/georgia-tech-db/eva/master/tutorials/00-
↪start-eva-server.ipynb"
%run 00-start-eva-server.ipynb
cursor = connect_to_server()
```

```
File '00-start-eva-server.ipynb' already there; not retrieving.

nohup eva_server > eva.log 2>&1 &
Note: you may need to restart the kernel to use updated packages.
```

13.2 Download custom user-defined function (UDF), model, and video

```
# Download UDF
!wget -nc https://www.dropbox.com/s/nht5lwjemmclqx3/gender.py?raw=1 -O gender.py

# Download built-in Face Detector
!wget -nc https://raw.githubusercontent.com/georgia-tech-db/eva/master/eva/udfs/face_
↪detector.py

# Download models
!wget -nc https://www.dropbox.com/s/yyp7awyczv7esf4/gender.pth?raw=1 -O gender.pth

# Download videos
!wget -nc https://www.dropbox.com/s/f5rorxf0840ajjd/short.mp4?raw=1 -O short.mp4
```

```
File 'gender.py' already there; not retrieving.
File 'face_detector.py' already there; not retrieving.

File 'gender.pth' already there; not retrieving.
File 'short.mp4' already there; not retrieving.
```

13.3 Load video for analysis

```
cursor.execute("DROP TABLE TIKTOK;")
response = cursor.fetch_all()
print(response)
cursor.execute("LOAD VIDEO 'short.mp4' INTO TIKTOK;")
response = cursor.fetch_all()
print(response)
cursor.execute("""SELECT id FROM TIKTOK WHERE id < 5""")
response = cursor.fetch_all()
print(response)
```

```
@status: ResponseStatus.FAIL
@batch:
 None
@error: Table: TIKTOK does not exist
@status: ResponseStatus.SUCCESS
@batch:
                              0
0  Number of loaded VIDEO: 1
@query_time: 2.2460366699997394
@status: ResponseStatus.SUCCESS
@batch:
    tiktok.id
0           0
1           1
2           2
3           3
4           4
@query_time: 0.0720941989993662
```

13.4 Visualize Video

```
from IPython.display import Video
Video("short.mp4", embed=True)
```

```
<IPython.core.display.Video object>
```

13.5 Create GenderCNN and FaceDetector UDFs

```
cursor.execute("""DROP UDF GenderCNN;""")
response = cursor.fetch_all()
print(response)

cursor.execute("""CREATE UDF IF NOT EXISTS
                  GenderCNN
                  INPUT (data NDARRAY UINT8(3, 224, 224))
```

(continues on next page)

(continued from previous page)

```
                  OUTPUT (label TEXT(10))
                  TYPE  Classification
                  IMPL 'gender.py';
        """)
response = cursor.fetch_all()
print(response)

cursor.execute("""CREATE UDF IF NOT EXISTS
                  FaceDetector
                  INPUT  (frame NDARRAY UINT8(3, ANYDIM, ANYDIM))
                  OUTPUT (bboxes NDARRAY FLOAT32(ANYDIM, 4),
                          scores NDARRAY FLOAT32(ANYDIM))
                  TYPE  FaceDetection
                  IMPL  'face_detector.py';
        """)
response = cursor.fetch_all()
print(response)
```

```
@status: ResponseStatus.SUCCESS
@batch:
                                       0
0  UDF GenderCNN successfully dropped
@query_time: 0.13388806600050884
@status: ResponseStatus.SUCCESS
@batch:
                                                         0
0  UDF GenderCNN successfully added to the database.
@query_time: 0.44734299300034763
@status: ResponseStatus.SUCCESS
@batch:
                                                        0
0  UDF FaceDetector already exists, nothing added.
@query_time: 0.006585866000023088
```

13.6 Run Face Detector on video

```
cursor.execute("""SELECT id, FaceDetector(data).bboxes
                  FROM TIKTOK WHERE id < 10""")
response = cursor.fetch_all()
response.as_df()
```

```
   tiktok.id    facedetector.bboxes
0          0  [[81, 194, 282, 470]]
1          1  [[82, 194, 283, 469]]
2          2  [[82, 195, 283, 470]]
3          3  [[82, 194, 284, 472]]
4          4  [[84, 197, 283, 472]]
5          5  [[85, 199, 283, 471]]
6          6  [[84, 199, 284, 472]]
```

(continues on next page)

(continued from previous page)

```
7          7  [[84, 198, 284, 473]]
8          8  [[85, 197, 283, 472]]
9          9  [[86, 198, 282, 476]]
```

13.7 Composing UDFs in a query

Detect gender of the faces detected in the video by composing a set of UDFs (GenderCNN, FaceDetector, and Crop)

```
cursor.execute("""SELECT id, bbox, GenderCNN(Crop(data, bbox))
                  FROM TIKTOK JOIN LATERAL  UNNEST(FaceDetector(data)) AS Face(bbox,␣
↪conf)
                  WHERE id < 50;""")
response = cursor.fetch_all()
response.as_df()
```

```
    tiktok.id            Face.bbox gendercnn.label
0           0  [81, 194, 282, 470]          female
1           1  [82, 194, 283, 469]          female
2           2  [82, 195, 283, 470]          female
3           3  [82, 194, 284, 472]          female
4           4  [84, 197, 283, 472]          female
5           5  [85, 199, 283, 471]          female
6           6  [84, 199, 284, 472]          female
7           7  [84, 198, 284, 473]          female
8           8  [85, 197, 283, 472]          female
9           9  [86, 198, 282, 476]          female
10         10  [85, 199, 283, 477]          female
11         11  [85, 200, 282, 478]          female
12         12  [94, 199, 288, 481]          female
13         13  [93, 199, 285, 480]          female
14         14  [95, 203, 288, 481]          female
15         15  [95, 201, 287, 479]          female
16         16  [95, 205, 287, 476]          female
17         17  [94, 204, 287, 475]          female
18         18  [84, 203, 283, 484]          female
19         19  [86, 196, 283, 474]          female
20         20  [84, 198, 280, 476]          female
21         21  [82, 206, 278, 483]          female
22         22  [81, 207, 277, 481]          female
23         23  [80, 206, 277, 482]          female
24         24  [80, 207, 276, 482]          female
25         25  [82, 209, 276, 481]          female
26         26  [83, 200, 278, 477]          female
27         27  [82, 206, 279, 485]          female
28         28  [82, 204, 280, 485]          female
29         29  [86, 202, 279, 476]          female
30         30  [88, 197, 282, 473]          female
31         31  [87, 205, 284, 482]          female
32         32  [89, 207, 284, 482]          female
33         33  [94, 202, 288, 475]          female
```

(continues on next page)

(continued from previous page)

```
34          34  [94, 205, 287, 481]          female
35          35  [95, 203, 288, 479]          female
36          36  [96, 203, 290, 480]          female
37          37  [96, 202, 288, 477]          female
38          38  [96, 201, 289, 475]          female
39          39  [96, 203, 290, 478]          female
40          40  [94, 199, 289, 475]          female
41          41  [93, 199, 288, 474]          female
42          42  [92, 197, 289, 472]          female
43          43  [95, 199, 291, 471]          female
44          44  [93, 199, 292, 470]          female
45          45  [93, 200, 291, 468]          female
46          46  [94, 209, 285, 480]          female
47          47  [90, 207, 285, 482]          female
48          48  [88, 206, 284, 480]          female
49          49  [88, 207, 283, 480]          female
```

13.8 Visualize Output

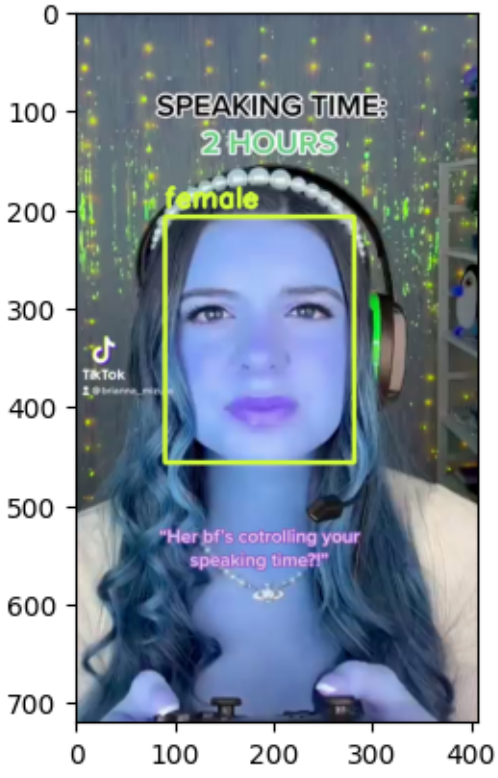
```
import cv2
from matplotlib import pyplot as plt

def annotate_video(detections, input_video_path, output_video_path):
    color=(207, 248, 64)
    thickness=4

    vcap = cv2.VideoCapture(input_video_path)
    width = int(vcap.get(3))
    height = int(vcap.get(4))
    fps = vcap.get(5)
    fourcc = cv2.VideoWriter_fourcc(*'MP4V') #codec
    video=cv2.VideoWriter(output_video_path, fourcc, fps, (width,height))

    frame_id = 0
    # Capture frame-by-frame
    ret, frame = vcap.read()  # ret = 1 if the video is captured; frame is the image

    while ret:
        df = detections
        df = df[['Face.bbox', 'gendercnn.label']][df['tiktok.id'] == frame_id]

        if df.size:
            for bbox, label in df.values:
                x1, y1, x2, y2 = bbox
                x1, y1, x2, y2 = int(x1), int(y1), int(x2), int(y2)
                frame=cv2.rectangle(frame, (x1, y1), (x2, y2), color, thickness) #␣
↪object bbox
                cv2.putText(frame, str(label), (x1, y1-10), cv2.FONT_HERSHEY_SIMPLEX, 0.
↪9, color, thickness-1) # object label
```

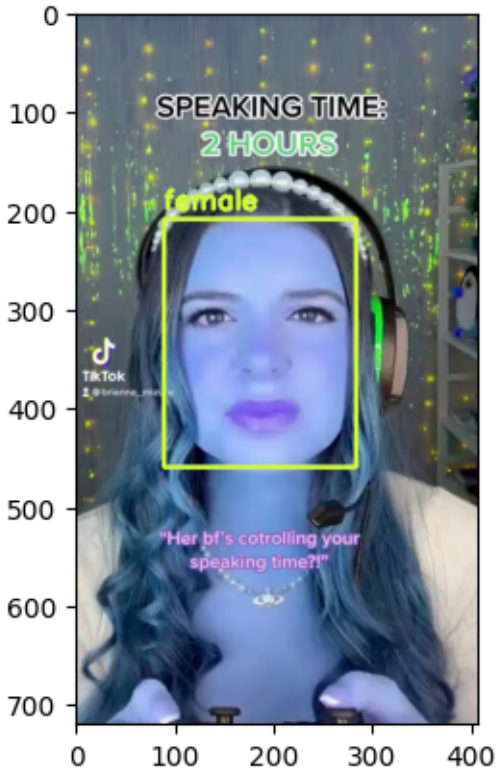
(continues on next page)

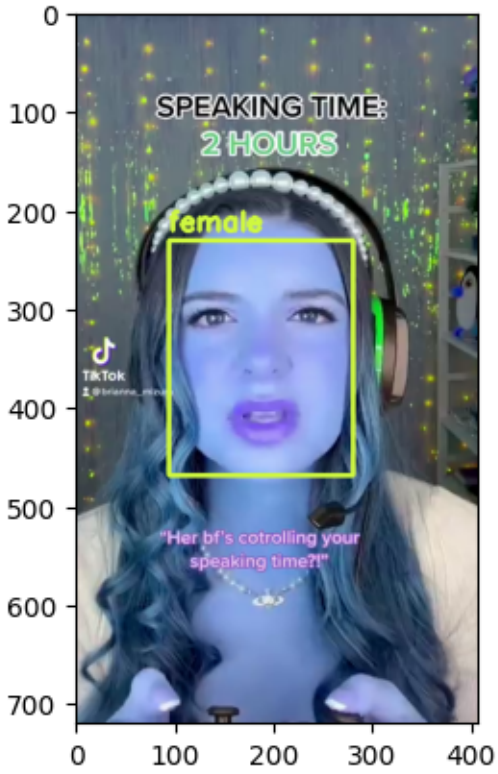
(continued from previous page)

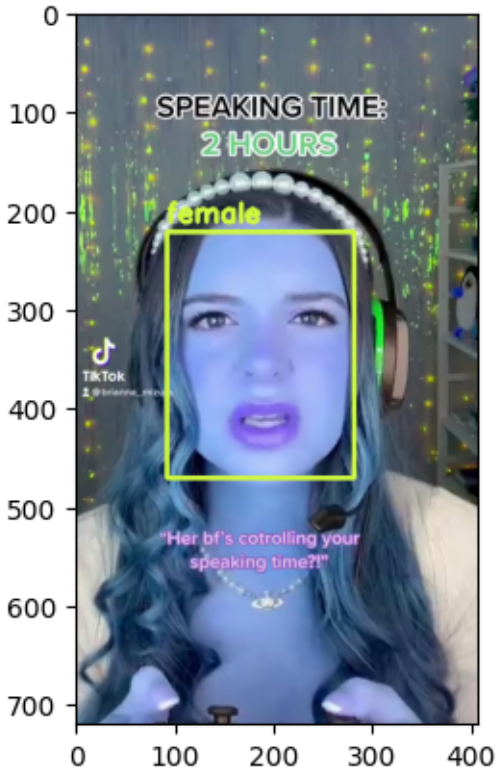
```
            video.write(frame)
            # Show every fifth frame
            if frame_id % 5 == 0:
                plt.imshow(frame)
                plt.show()

        if frame_id == 50:
            return

        frame_id+=1
        ret, frame = vcap.read()

    video.release()
    vcap.release()
```

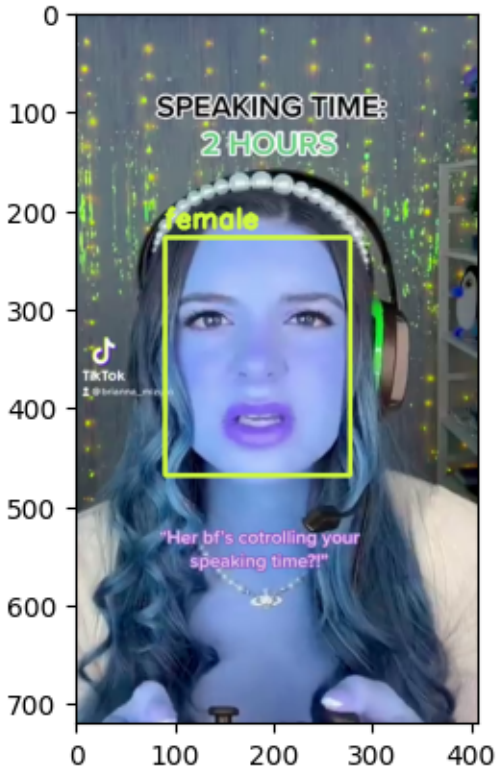
```
#!pip install ipywidgets
from ipywidgets import Video
input_path = 'short.mp4'
output_path = 'annotated_short.mp4'

dataframe = response.as_df()
annotate_video(dataframe, input_path, output_path)
```

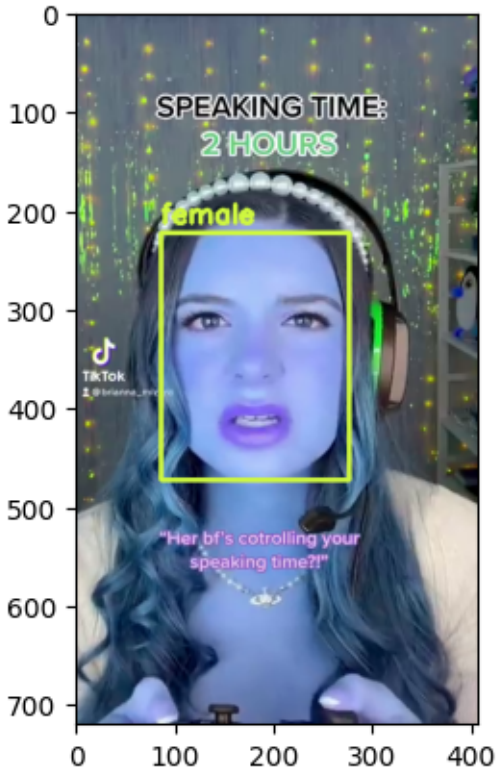


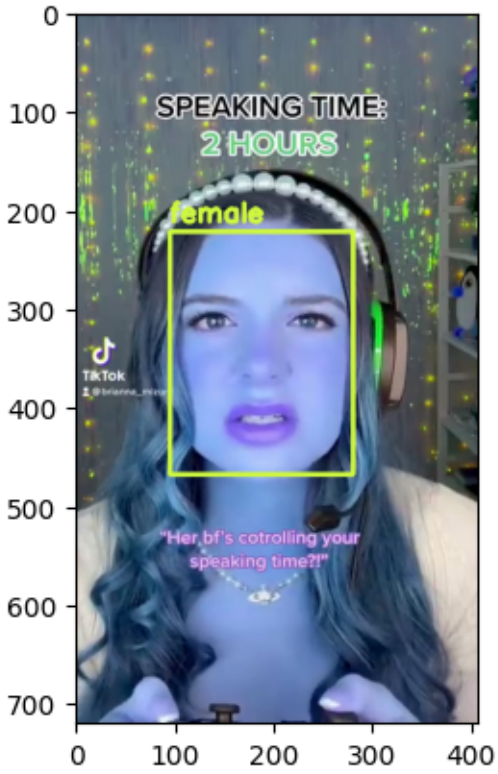

SPEAKING TIME:
2 HOURS
female
"Her bf's cotrolling your speaking time?!"
TikTok

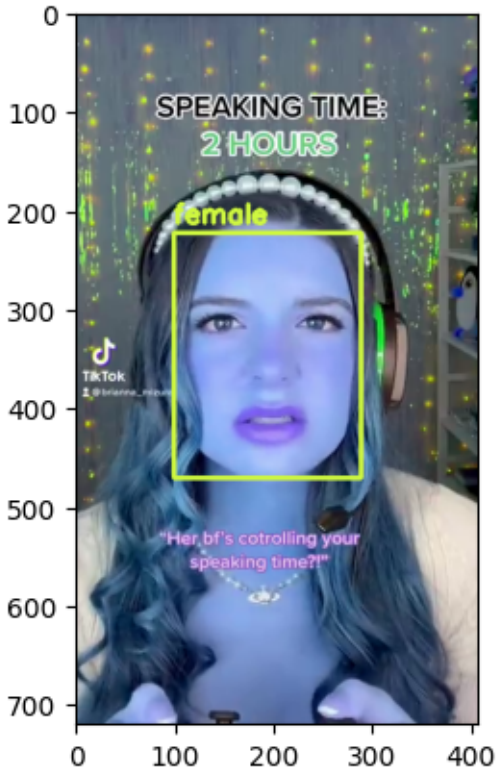

SPEAKING TIME:
2 HOURS
female
"Her bf's cotrolling your speaking time?!"
TikTok

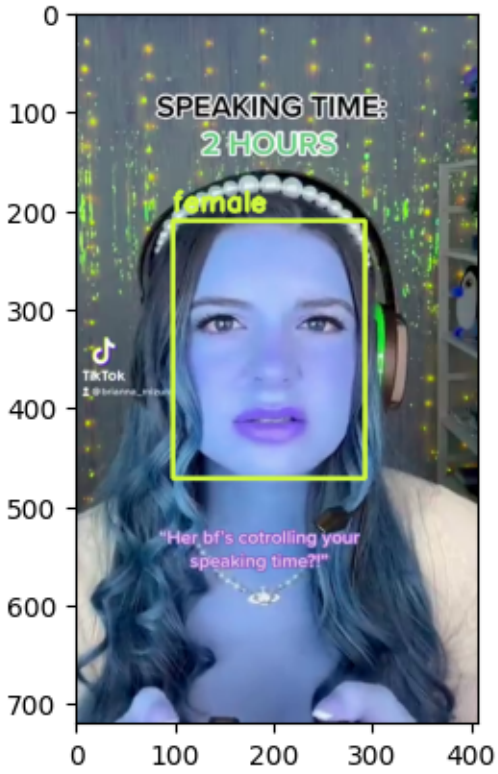

SPEAKING TIME:
2 HOURS
female
TikTok
"Her bf's cotrolling your speaking time?!"

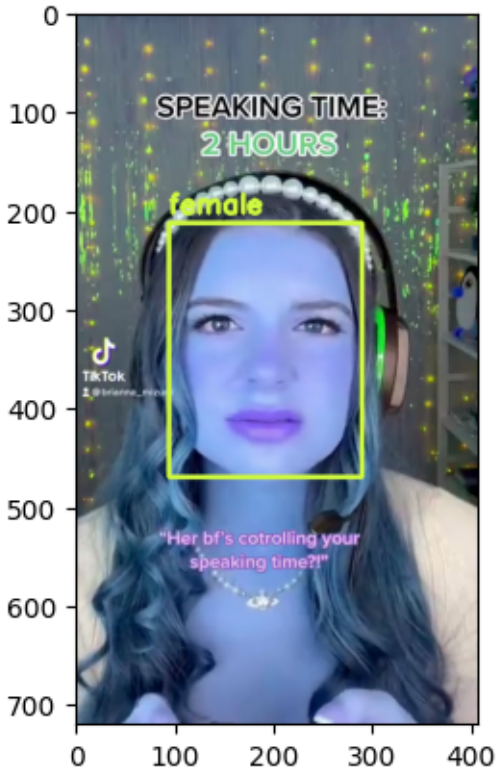

SPEAKING TIME:
2 HOURS
female
TikTok
"Her bf's cotrolling your speaking time?!"


SPEAKING TIME:
2 HOURS
female
TikTok
"Her bf's cotrolling your speaking time?!"


SPEAKING TIME:
2 HOURS
female
TikTok
"Her bf's cotrolling your speaking time?!"


SPEAKING TIME:
2 HOURS
female
TikTok
"Her bf's cotrolling your speaking time?!"


SPEAKING TIME:
2 HOURS
female
TikTok
"Her bf's cotrolling your speaking time?!"


SPEAKING TIME:
2 HOURS
female
TikTok
"Her bf's cotrolling your speaking time?!"
0
100
200
300
400
500
600
700
0
100
200
300
400

CHAPTER

FOURTEEN

EVA QUERY LANGUAGE REFERENCE

EVA Query Language (EVAQL) is derived from SQL. It is tailored for video analytics. EVAQL allows users to invoke deep learning models in the form of user-defined functions (UDFs).

Here is an example where we first define a UDF wrapping around the FastRCNN object detection model. We then issue a query with this function to detect objects.

```
--- Create an user-defined function wrapping around FastRCNN ObjectDetector
CREATE UDF IF NOT EXISTS FastRCNNObjectDetector
INPUT  (frame NDARRAY UINT8(3, ANYDIM, ANYDIM))
OUTPUT (labels NDARRAY STR(ANYDIM), bboxes NDARRAY FLOAT32(ANYDIM, 4),
        scores NDARRAY FLOAT32(ANYDIM))
TYPE  Classification
IMPL  'eva/udfs/fastrcnn_object_detector.py';

--- Use the function to retrieve frames that contain more than 3 cars
SELECT id FROM MyVideo
WHERE ArrayCount(FastRCNNObjectDetector(data).label, 'car') > 3
ORDER BY id;
```

This page presents a list of all the EVAQL statements that you can leverage in your Jupyter Notebooks.

14.1 LOAD

14.1.1 LOAD VIDEO FROM FILESYSTEM

```
LOAD VIDEO 'test_video.mp4' INTO MyVideo;
```

- **test_video.mp4** is the location of the video file in the filesystem on the client.
- **MyVideo** is the name of the table in EVA where this video is loaded. Subsequent queries over the video must refer to this table name.

When a video is loaded, there is no need to specify the schema for the video table. EVA automatically generates the following schema with two columns: `id` and `data`, that correspond to the frame id and frame content (in Numpy format).

14.1.2 LOAD VIDEO FROM S3

```
LOAD VIDEO 's3://bucket/dummy.avi' INTO MyVideo;
LOAD VIDEO 's3://bucket/eva_videos/*.mp4' INTO MyVideos;
```

The videos are downloaded to a directory that can be configured in the EVA configuration file under *storage:s3_download_dir*. The default directory is `~/.eva/s3_downloads`.

14.1.3 LOAD CSV

To **LOAD** a CSV file, we need to first specify the table schema.

```
CREATE TABLE IF NOT EXISTS MyCSV (
                id INTEGER UNIQUE,
                frame_id INTEGER,
                video_id INTEGER,
                dataset_name TEXT(30),
                label TEXT(30),
                bbox NDARRAY FLOAT32(4),
                object_id INTEGER
            );

LOAD CSV 'test_metadata.csv' INTO MyCSV;
```

- **test_metadata.csv** needs to be loaded onto the server using **LOAD** statement.
- The CSV file may contain additional columns. EVA will only load the columns listed in the defined schema.

14.2 SELECT

14.2.1 SELECT FRAMES WITH PREDICATES

Search for frames with a car

```
SELECT id, frame
FROM MyVideo
WHERE ['car'] <@ FastRCNNObjectDetector(frame).labels
ORDER BY id;
```

Search frames with a pedestrian and a car

```
SELECT id, frame
FROM MyVideo
WHERE ['pedestrian', 'car'] <@ FastRCNNObjectDetector(frame).labels;
```

Search for frames containing greater than 3 cars

```
SELECT id FROM MyVideo
WHERE ArrayCount(FastRCNNObjectDetector(data).label, 'car') > 3
ORDER BY id;
```

14.2.2 SELECT WITH MULTIPLE UDFS

Compose multiple user-defined functions in a single query to construct semantically complex queries.

```
SELECT id, bbox, EmotionDetector(Crop(data, bbox))
FROM HAPPY JOIN LATERAL UNNEST(FaceDetector(data)) AS Face(bbox, conf)
WHERE id < 15;
```

14.3 EXPLAIN

14.3.1 EXPLAIN QUERY

List the query plan associated with a EVAQL query

Append `EXPLAIN` in front of the query to retrieve the plan.

```
EXPLAIN SELECT CLASS FROM TAIPAI;
```

14.4 SHOW

14.4.1 SHOW UDFS

List the registered user-defined functions

```
SHOW UDFS;
```

14.5 CREATE

14.5.1 CREATE TABLE

To create a table, specify the schema of the table.

```
CREATE TABLE IF NOT EXISTS MyCSV (
                id INTEGER UNIQUE,
                frame_id INTEGER,
                video_id INTEGER,
                dataset_name TEXT(30),
                label TEXT(30),
                bbox NDARRAY FLOAT32(4),
                object_id INTEGER
 );
```

14.5.2 CREATE UDF

To register an user-defined function, specify the implementation details of the UDF.

```
CREATE UDF IF NOT EXISTS FastRCNNObjectDetector
INPUT  (frame NDARRAY UINT8(3, ANYDIM, ANYDIM))
OUTPUT (labels NDARRAY STR(ANYDIM), bboxes NDARRAY FLOAT32(ANYDIM, 4),
        scores NDARRAY FLOAT32(ANYDIM))
TYPE  Classification
IMPL  'eva/udfs/fastrcnn_object_detector.py';
```

14.5.3 CREATE MATERIALIZED VIEW

To create a view with materialized results – like the outputs of deep learning model, use the following template:

```
CREATE MATERIALIZED VIEW UADETRAC_FastRCNN (id, labels) AS
SELECT id, FastRCNNObjectDetector(frame).labels
FROM UADETRAC
WHERE id<5;
```

14.6 DROP

14.6.1 DROP TABLE

```
DROP TABLE DETRACVideo;
```

14.6.2 DROP UDF

```
DROP UDF FastRCNNObjectDetector;
```

14.7 INSERT

14.7.1 TABLE MyVideo

MyVideo Table schema

```
CREATE TABLE MyVideo
(id INTEGER,
data NDARRAY FLOAT32(ANYDIM));
```

14.7.2 INSERT INTO TABLE

Insert a tuple into a table.

```
INSERT INTO MyVideo (id, data) VALUES
    (1,
        [[[40, 40, 40] , [40, 40, 40]],
         [[40, 40, 40] , [40, 40, 40]]]);
```

14.8 DELETE

14.8.1 DELETE INTO TABLE

Delete a tuple from a table based on a predicate.

```
DELETE FROM MyVideo WHERE id<10;
```

14.9 RENAME

14.9.1 RENAME TABLE

```
RENAME TABLE MyVideo TO MyVideo1;
```

CHAPTER

FIFTEEN

USER-DEFINED FUNCTIONS

This section provides an overview of how you can create and use a custom user-defined function (UDF) in your queries. For example, you could write an UDF that wraps around a PyTorch model.

15.1 Part 1: Writing a custom UDF

Illustrative UDF implementation

During each step, use this UDF implementation as a reference.

1. Create a new file under *udfs/* folder and give it a descriptive name. eg: *fastrcnn_object_detector.py*, *midas_depth_estimator.py*.

Note: UDFs packaged along with EVA are located inside the udfs folder.

2. Create a Python class that inherits from *PytorchClassifierAbstractUDF*.

- The *PytorchClassifierAbstractUDF* is a parent class that defines and implements standard methods for model inference.
- *_get_predictions()* - an abstract method that needs to be implemented in your child class.
- *classify()* - A method that receives the frames and calls the *_get_predictions()* implemented in your child class. Based on GPU batch size, it also decides whether to split frames into chunks and performs the accumulation.
- Additionally, it contains methods that help in:
  - Moving tensors to GPU
  - Converting tensors to numpy arrays.
  - Defining the *forward()* function that gets called when the model is invoked.
  - Basic transformations.

You can however **choose to override** these methods depending on your requirements.

3. A typical UDF class has the following components:

- *__init__()* constructor:
  - Define variables here that will be required across all methods.
  - Model loading happens here. You can choose to load custom models or models from torch.

  * **Example of loading a custom model:**

```
custom_model_path = os.path.join(EVA_DIR, "data", "models", "vehicle_
↪make_predictor", "car_recognition.pt")
self.car_make_model = CarRecognitionModel()
self.car_make_model.load_state_dict(torch.load(custom_model_path))
self.car_make_model.eval()
```

  * **Example of loading a torch model:**

```
self.model = torchvision.models.detection.fasterrcnn_resnet50_
↪fpn(pretrained=True)
self.model.eval()
```

- *labels()* method:
  - This should return a list of strings that your model aims to target.
  - The index of the list is the value predicted by your model.
- *_get_predictions()* method:
  - This is where all your model inference logic goes.
  - While doing the computations, keep in mind that each call of this method is with a batch of frames.
  - **Output from each invoke of the model needs to be appended to a dataframe and returned as follows:**

```
predictions = self.model(frames)
outcome = pd.DataFrame()
for prediction in predictions:

    ## YOUR INFERENCE LOGIC

    # column names depend on your implementation
    outcome = outcome.append(
        {
            "labels": pred_class,
            "scores": pred_score,
            "boxes": pred_boxes
        },
        ignore_index=True)
```

In case you have any other functional requirements (defining custom transformations etc.) you can choose to add more methods. Make sure each method you write is clear, concise and well-documented.

15.2 Part 2: Registering and using the UDF in queries

Now that you have implemented your UDF we need to register it in EVA. You can then use the function in any query.

1. Register the UDF with a query that follows this template:

   `**CREATE UDF [IF NOT EXISTS] <name>** INPUT ([<arg_name> <arg_data_type>] [, …]) OUTPUT ([<result_name> <result_data_type>] [, …]) TYPE <udf_type_name> IMPL '<path_to_implementation>'``

   where,

   - **<name>** - specifies the unique identifier for the UDF.
   - **[<arg_name> <arg_data_type>] [, …]** - specifies the name and data type of the udf input arguments. Name is kept for consistency (ignored by eva right now), arguments data type is required. ANYDIM means the shape is inferred at runtime.
   - **[<result_name> <result_data_type>] [, …]** - specifies the name and data type of the udf output arguments. Users can access a specific output of the UDF similar to access a column of a table. Eg. <name>.<result_name>
   - **<udf_type_name>** - specifies the identifier for the type of the UDF. UDFs of the same type are assumed to be interchangeable. They should all have identical input and output arguments. For example, object classification can be one type.
   - **<path_to_implementation>** - specifies the path to the implementation class for the UDF

   Here, is an example query that registers a UDF that wraps around the 'FastRCNNObjectDetector' model that performs Object Detection.

   ```
   CREATE UDF IF NOT EXISTS FastRCNNObjectDetector
   INPUT  (frame NDARRAY UINT8(3, ANYDIM, ANYDIM))
   OUTPUT (labels NDARRAY STR(ANYDIM), bboxes NDARRAY FLOAT32(ANYDIM,␣
   ↪4),
           scores NDARRAY FLOAT32(ANYDIM))
   TYPE  Classification
   IMPL  'eva/udfs/fastrcnn_object_detector.py';
   ```

   - Input is a frame of type NDARRAY with shape (3, ANYDIM, ANYDIM). 3 channels and any width or height.
   - **We return 3 variables for this UDF:**
     - *labels*: Predicted label
     - *bboxes*: Bounding box of this object (rectangle coordinates)
     - *scores*: Confidence scores for this prediction

   A status of 0 in the response denotes the successful registration of this UDF.

3. Now you can execute your UDF on any video:

```
SELECT id, Unnest(FastRCNNObjectDetector(data)) FROM MyVideo;
```

4. You can drop the UDF when you no longer need it.

```
DROP UDF IF EXISTS FastRCNNObjectDetector;
```

CHAPTER

SIXTEEN

HUGGINGFACE MODELS IN EVA

This section provides an overview of how you can use out-of-the-box HuggingFace models in EVA.

16.1 Creating UDF from HuggingFace

EVA supports UDFS similar to Pipelines in HuggingFace.

```
CREATE UDF IF NOT EXISTS HFObjectDetector
TYPE  HuggingFace
'task' 'object-detection'
'model' 'facebook / detr-resnet-50'
```

EVA supports all arguments supported by HF pipelines. You can pass those using a key value format similar to task and model above.

16.2 Supported Tasks

EVA supports the following tasks from huggingface:

- Audio Classification
- Automatic Speech Recognition
- Text Classification
- Summarization
- Text2Text Generation
- Text Generation
- Image Classification
- Image Segmentation
- Image-to-Text
- Object Detection
- Depth Estimation

CHAPTER

SEVENTEEN

CONFIGURE GPU

1. Queries in EVA use deep learning models that run much faster on a GPU as opposed to a CPU. If your workstation has a GPU, you can configure EVA to use the GPU during query execution. Use the following command to check your hardware capabilities:

```
ubuntu-drivers devices
nvidia-smi
```

A valid output from the command indicates that your GPU is configured and ready to use. If not, you will need to install the appropriate GPU driver. This page provides a step-by-step guide on installing and configuring the GPU driver in the Ubuntu Operating System.

- When installing an NVIDIA driver, ensure that the version of the GPU driver is correct to avoid compatibiility issues.
- When installing cuDNN, you will need to create an account and ensure that you get the correct *deb* files for your operating system and architecture.

2. You can run the following code in a Jupyter notebook to verify that your GPU is detected by PyTorch:

```
import torch
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(device)
```

Output of *cuda:0* indicates the presence of a GPU. 0 indicates the index of the GPU in system. If you have multiple GPUs on your workstation, the index must be updated accordingly.

3. Now configure the executor section in eva.yml as follows:

```
executor:
    gpus: {'127.0.1.1': [0]}
```

Here, *127.0.1.1* is the loopback address on which the EVA server is running. 0 refers to the GPU index to be used.

CHAPTER

EIGHTEEN

EVA INTERNALS

18.1 Path of a Query

The following code represents a sequence of operations that can be used to execute a query in a evaql database. found in eva/server/command_handler.py

> Parse the query using the Parser() function provided by the evaql library. The result of this step will be a parsed representation of the query in the form of an abstract syntax tree (AST).

```
stmt = Parser().parse(query)[0]
```

Bind the parsed AST to a statement context using the StatementBinder() function. This step resolves references to schema objects and performs other semantic checks on the query.

```
StatementBinder(StatementBinderContext()).bind(stmt)
```

Convert the bound AST to a logical plan using the StatementToPlanConvertor() function. This step generates a logical plan that specifies the sequence of operations needed to execute the query.

```
l_plan = StatementToPlanConvertor().visit(stmt)
```

Generate a physical plan from the logical plan using the plan_generator.build() function. This step optimizes the logical plan and generates a physical plan that specifies how the query will be executed.

```
p_plan = plan_generator.build(l_plan)
```

Execute the physical plan using the PlanExecutor() function. This step retrieves the data from the database and produces the final output of the query.

```
output = PlanExecutor(p_plan).execute_plan()
```

Overall, this sequence of operations represents the path of query execution in a evaql database, from parsing the query to producing the final output.

18.2 Topics

18.2.1 Catalog

Catalog Manager

Explanation for developers on how to use the eva catalog_manager.

CatalogManager class that provides a set of services to interact with a database that stores metadata about tables, columns, and user-defined functions (UDFs). Information like what is the data type in a certain column in a table, type of a table, its name, etc.. It contains functions to get, insert and delete catalog entries for Tables, UDFs, UDF IOs, Columns and Indexes.

This data is stored in the eva_catalog.db file which can be found in *~/.eva/<version>/* folder.

Catalog manager currently has 5 services in it:

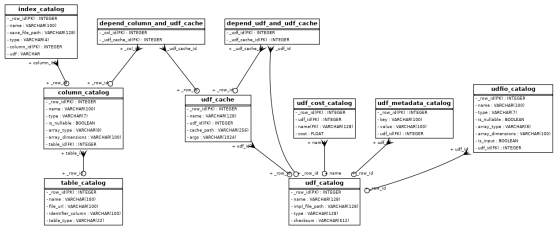
```
TableCatalogService()
ColumnCatalogService()
UdfCatalogService()
UdfIOCatalogService()
IndexCatalogService()
```

Catalog Services

This class provides functionality related to a table catalog, including inserting, getting, deleting, and renaming table entries, as well as retrieving all entries. e.g. the TableCatalogService contains code to get, insert and delete a table.

Catalog Models

These contain the data model that is used by the catalog services. Each model represents a table in the underlying database.



CHAPTER

NINETEEN

CONTRIBUTING

We welcome all kinds of contributions to EVA.

- Code reviews
- Improving documentation
- Tutorials and applications
- New features

19.1 Setting up the Development Environment

First, you will need to checkout the repository from GitHub and build EVA from the source. Follow the following instructions to build EVA locally. We recommend using a virtual environment and the pip package manager.

```
git clone https://github.com/georgia-tech-db/eva.git && cd eva
python3 -m venv test_eva_db       # create a virtual environment
source test_eva_db/bin/activate   # activate the virtual environment
pip install --upgrade pip         # upgrade pip
pip install -e ".[dev]"           # build and install the EVA package
bash script/test/test.sh          # run the eva EVA suite
```

After installing the package locally, you can make changes and run the test cases to check their impact.

```
pip install .         # reinstall EVA package to include local changes
pkill -9 eva_server   # kill running EVA server (if any)
eva_server&           # launch EVA server with newly installed package
```

19.2 Testing

Check if your local changes broke any unit or integration tests by running the following script:

```
bash script/test/test.sh
```

If you want to run a specific test file, use the following command.

```
python -m pytest test/integration_tests/test_select_executor.py
```

Use the following command to run a specific test case within a specific test file.

```
python -m pytest test/integration_tests/test_select_executor.py -k 'test_should_load_and_
↪select_in_table'
```

19.3 Submitting a Contribution

Follow the following steps to contribute to EVA:

- Merge the most recent changes from the master branch

```
git remote add origin git@github.com:georgia-tech-db/eva.git
git pull . origin/master
```

- Run the *test script* to ensure that all the test cases pass.
- If you are adding a new EVAQL command, add an illustrative example usage in the documentation.
- Run the following command to ensure that code is properly formatted.

```
python script/formatting/formatter.py
```

19.4 Code Style

We use the black code style for formatting the Python code. For docstrings and documentation, we use Google Pydoc format.

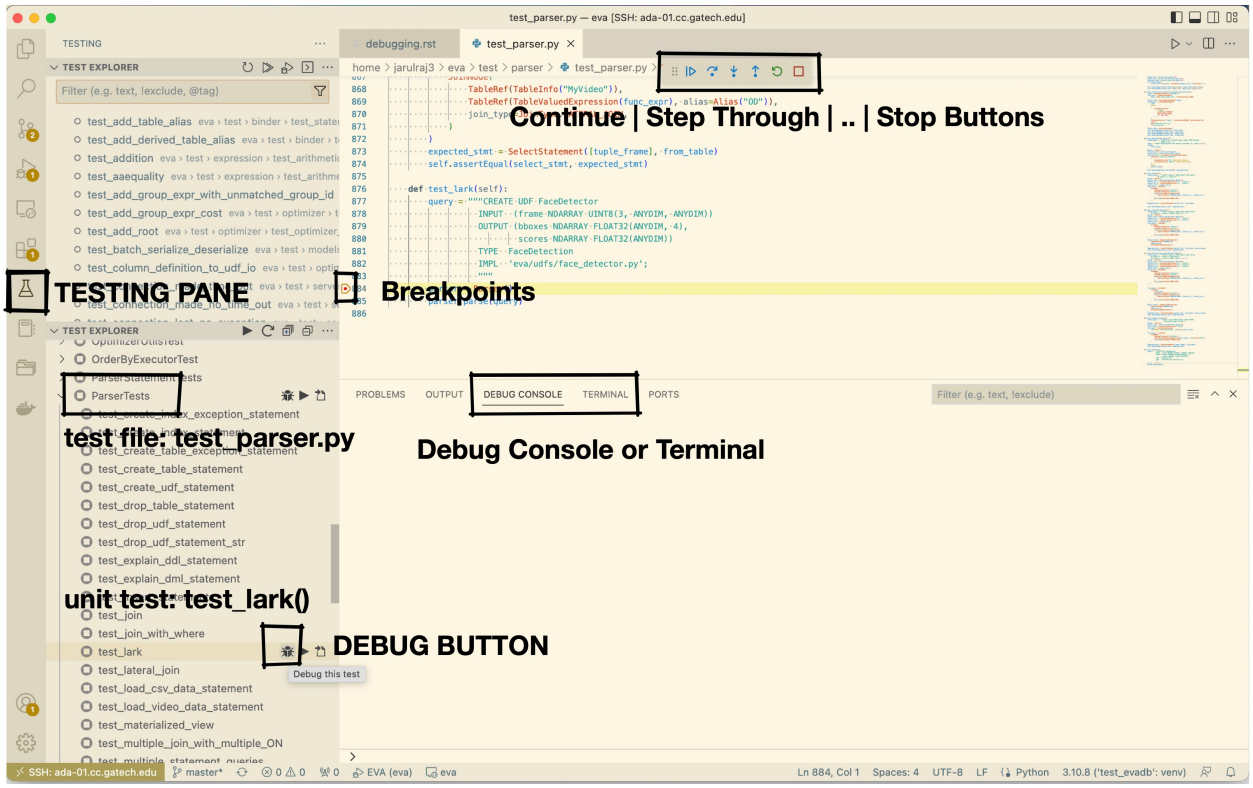
```
def function_with_types_in_docstring(param1, param2) -> bool:
    """Example function with types documented in the docstring.

    Additional explanatory text can be added in paragraphs.

    Args:
        param1 (int): The first parameter.
        param2 (str): The second parameter.

    Returns:
        bool: The return value. True for success, False otherwise.
```

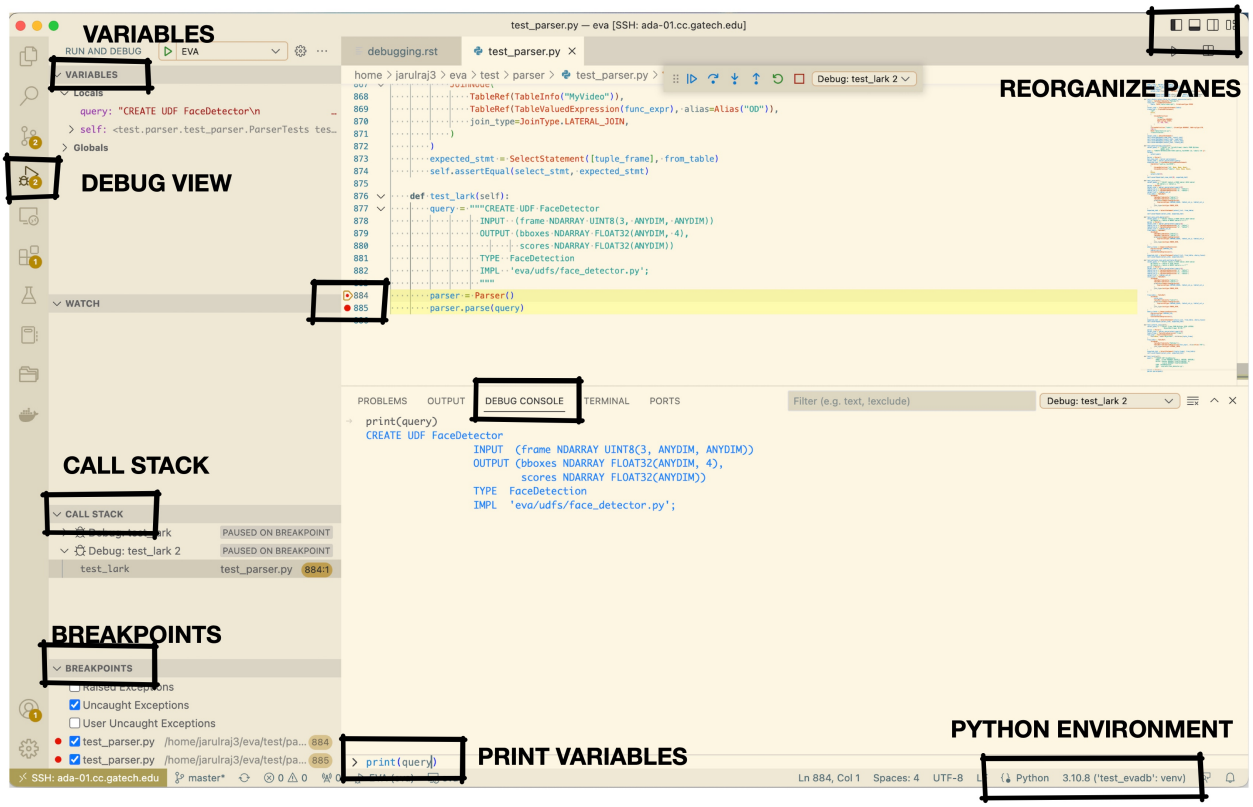
19.5 Debugging

We recommend using Visual Studio Code with a debugger for developing EVA. Here are the steps for setting up the development environment:

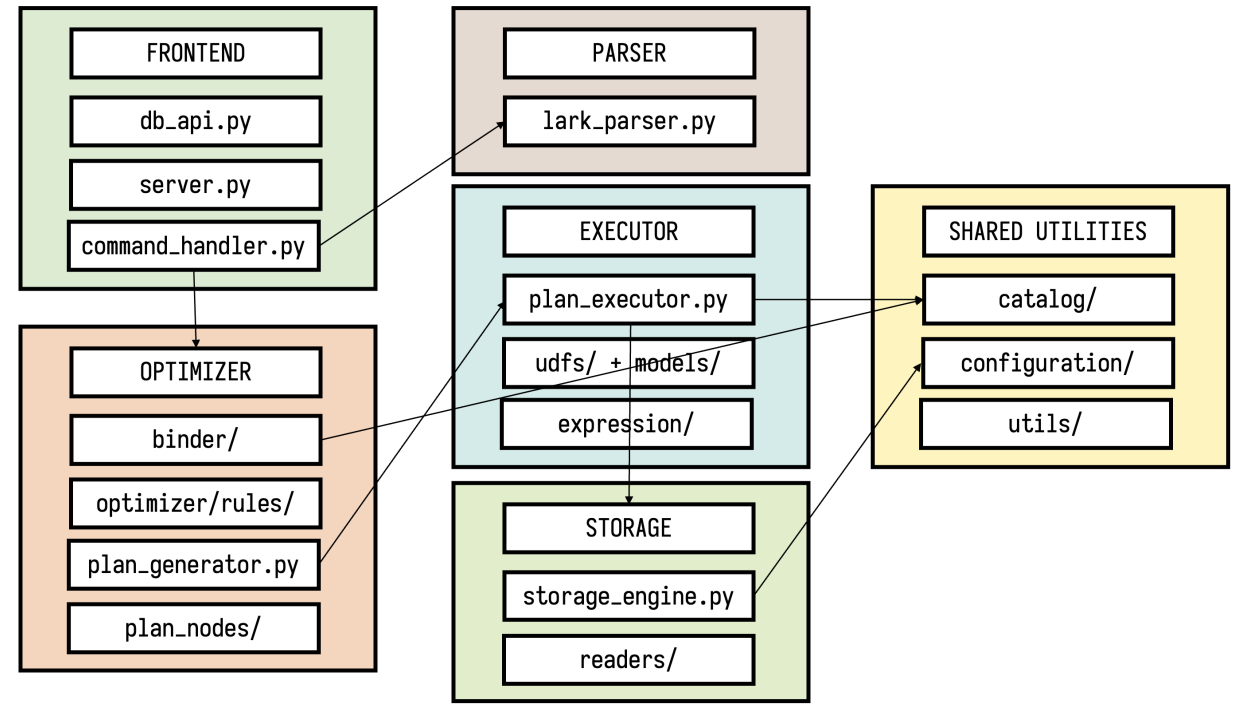
1. Install the Python extension in Visual Studio Code.
2. Install the Python Test Explorer extension.

3. Follow these instructions to run a particular test case from the file: Getting started.


Continue | Step Through | .. | Stop Buttons
TESTING PANE
Breakpoints
test file: test_parser.py
Debug Console or Terminal
unit test: test_lark()
DEBUG BUTTON


VARIABLES
REORGANIZE PANES
DEBUG VIEW
CALL STACK
BREAKPOINTS
PRINT VARIABLES
PYTHON ENVIRONMENT

19.6 Architecture Diagram



19.7 Troubleshooting

If the test suite fails with a *PermissionDenied* exception, update the *path_prefix* attribute under the *storage* section in the EVA configuration file (`~/.eva/eva.yml`) to a directory where you have write privileges.

CHAPTER

TWENTY

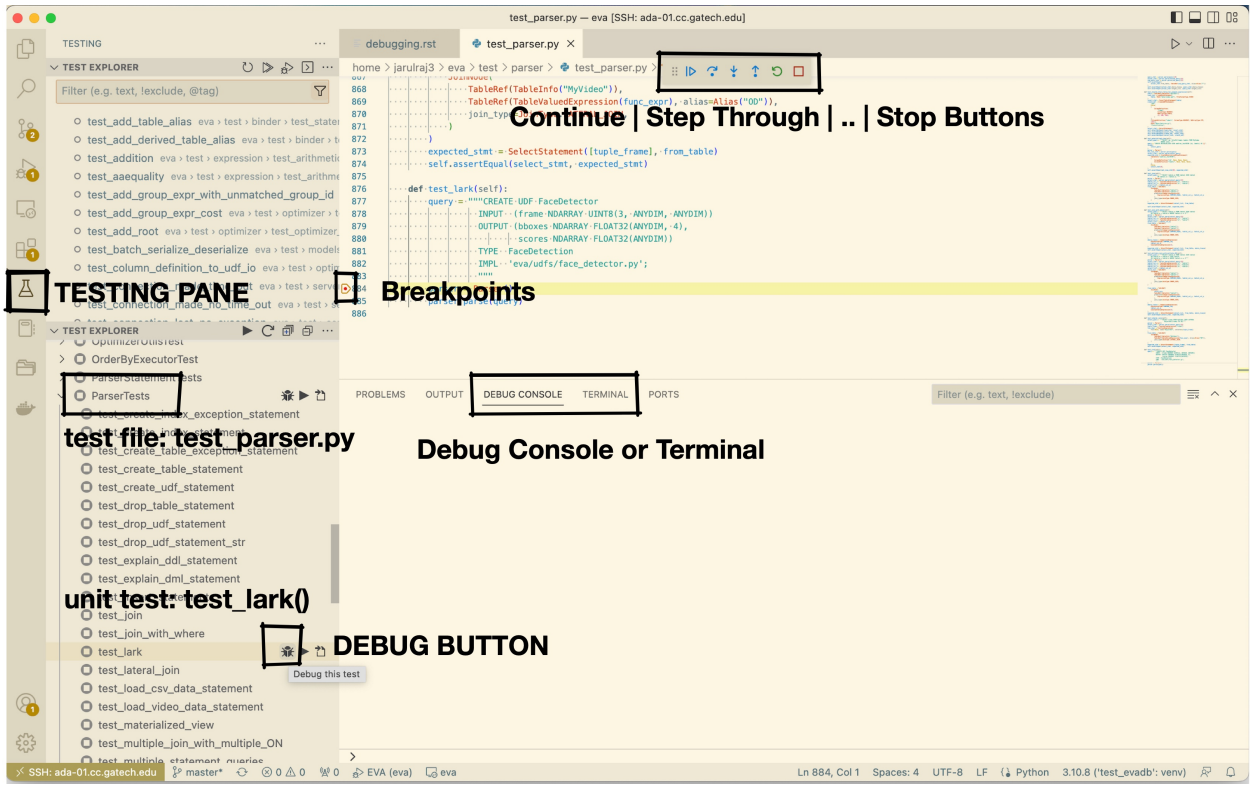
DEBUGGING

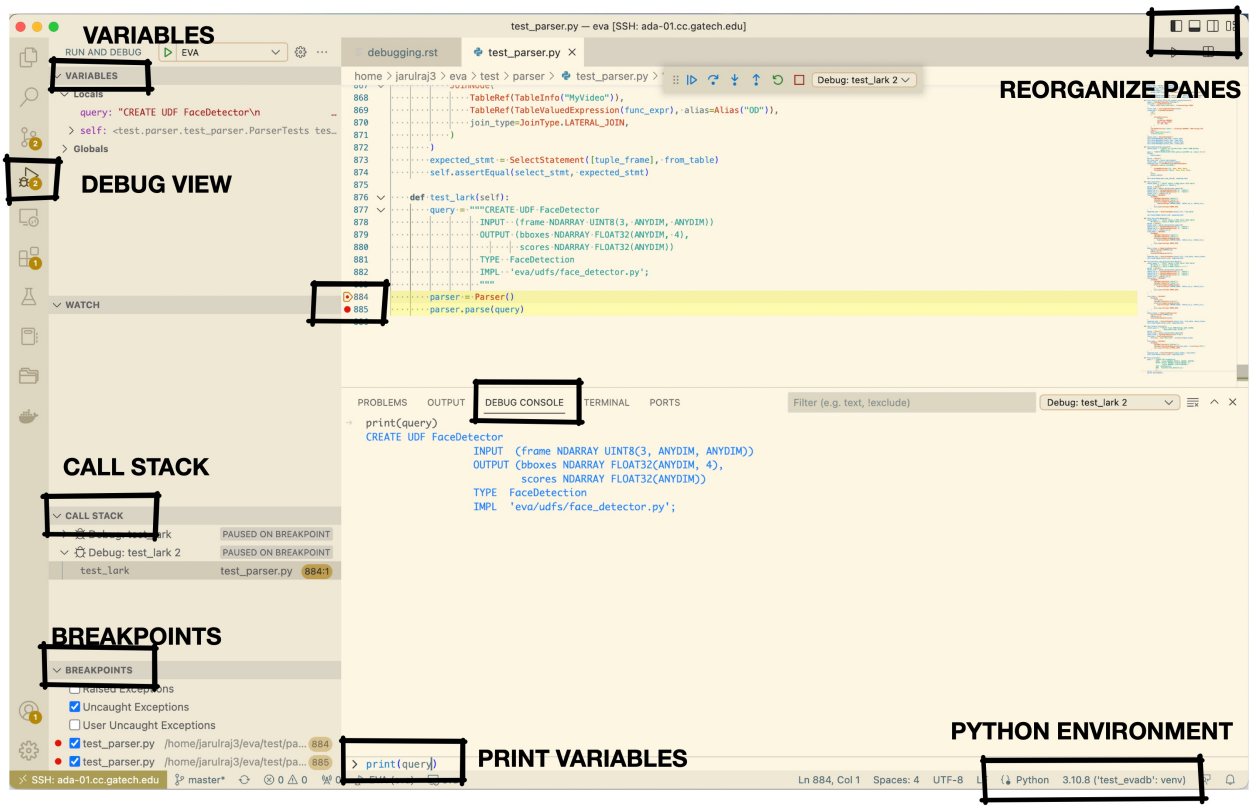
We recommend Visual Studio Code with a debugger for debugging EVA. This tutorial presents a detailed step-by-step process of using the debugger.

20.1 Setup debugger

1. Install the Python extension in Visual Studio Code.
2. Install the Python Test Explorer extension.

3. Follow these instructions to run a particular test case from the file: Getting started.





20.2 Alternative: Manually Setup Debugger for EVA

When you press the debug icon, you will be given an option to create a `launch.json` file.

While creating the JSON file, you will be prompted to select the environment to be used. Select the python environment from the `command palette` at the top. If the Python environment cannot be seen in the drop-down menu, try installing the python extension, and repeat the process.

Once you select the python environment, a `launch.json` file will be created with the default configurations set to debug a simple .py file.

More configurations can further be added to the file, to modify the environment variables or to debug an entire folder or workspace directory. Use the following configuration in the JSON file:

```
{

    "version": "0.2.0",
    "configurations": [

        {
            "name": "Python: test_pytorch.py",
            "type": "python",
            "request": "launch",
            "program": "${workspaceFolder}/test/integration_tests/test_pytorch.py",
            "console": "integratedTerminal",
            "cwd": "${workspaceFolder}",
            "env": {"PYTHONPATH": "${workspaceRoot}"}
```

(continues on next page)

(continued from previous page)

```
        }

    ]
}
```

You can modify the fields of the above JSON file as follows:

`name`: It is the reader-friendly name to appear in the Debug launch configuration dropdown.

`type`: The type of debugger to use for this launch configuration.

`program`: The executable or file to run when launching the debugger. In the above example, test_integration.py will be executed by the debugger.

`env`: Here you specify the environment variables. In the above example, the path for the conda environment for Eva has been specified.

Using these configuration variables, you can run the debugger both locally as well as on a remote server.

CHAPTER

TWENTYONE

EXTENDING EVA

This document details the steps involved in adding support for a new operator (or command) in EVA. We illustrate the process using a DDL command.

21.1 Command Handler

An input query string is handled by **Parser**, **StatementTOPlanConvertor**, **PlanGenerator**, and **PlanExecutor**. We discuss each part separately.

```
def execute_query(query) -> Iterator[Batch]:
    """
    Execute the query and return a result generator.
    """
    #1. parser
    stmt = Parser().parse(query)[0]
    #2. statement to logical plan
    l_plan = StatementToPlanConvertor().visit(stmt)
    #3. logical to physical plan
    p_plan = PlanGenerator().build(l_plan)
    #4. parser
    return PlanExecutor(p_plan).execute_plan()
```

21.2 1. Parser

The parser firstly generate **syntax tree** from the input string, and then tansform syntax tree into **statement**.

The first part of Parser is build from a LARK grammar file.

21.2.1 parser/eva

- `eva.lark` - add keywords(eg. CREATE, TABLE) under **Common Keywords**
  - Add new grammar rule (eg. create_table)
  - Write a new grammar, for example:

```
create_table: CREATE TABLE if_not_exists? table_name create_definitions
```

The second part of parser is implemented as **parser visitor**.

21.2.2 parser/lark_visitor

- `_[cmd]_statement.py` - eg. class CreateTable(evaql_parserVisitor)
  - Write functions to transform each input data from syntax tree to desired type. (eg. transform Column information into a list of ColumnDefinition)
  - Write a function to construct [cmd]Statement and return it.
- `__init__.py` - import `_[cmd]_statement.py` and add its class to `ParserVisitor`'s parent class.

```
from src.parser.parser_visitor._create_statement import CenameTable
class ParserVisitor(CommonClauses, CreateTable, Expressions,
                    Functions, Insert, Select, TableSources,
                    Load, Upload):
```

21.2.3 parser/

- `[cmd]_statement.py` - class [cmd]Statement. Its constructor is called in `_[cmd]_statement.py`
- `types.py` - register new StatementType

21.3 2. Statement To Plan Convertor

The part transforms the statement into corresponding logical plan.

21.3.1 Optimizer

- `operators.py`
  - Define class Logical[cmd], which is the logical node for the specific type of command.

```
class LogicalCreate(Operator):
    def __init__(self, video: TableRef, column_list: List[DataFrameColumn], if_not_
↪exists: bool = False, children=None):
    super().__init__(OperatorType.LOGICALCREATE, children)
    self._video = video
    self._column_list = column_list
    self._if_not_exists = if_not_exists
    # ...
```

  - Register new operator type to **class OperatorType**, Notice that must add it **before LOGICALDELIMITER** !!!
- `statement_to_opr_convertor.py`
  - import resource

```
from src.optimizer.operators import LogicalCreate
from src.parser.rename_statement import CreateTableStatement
```

  - implement **visit_[cmd]**() function, which converts statement to operator

```
# May need to convert the statement into another data type.
# The new data type is usable for excuting command.
# For example, column_list -> column_metadata_list

def visit_create(self, statement: AbstractStatement):
    video_ref = statement.table_ref
    if video_ref is None:
        LoggingManager().log("Missing Table Name In Create Statement",
                             LoggingLevel.ERROR)

    if_not_exists = statement.if_not_exists
    column_metadata_list = create_column_metadata(statement.column_list)

    create_opr = LogicalCreate(
        video_ref, column_metadata_list, if_not_exists)
    self._plan = create_opr
```

- modify visit function to call the right visit_[cmd] funciton

```
def visit(self, statement: AbstractStatement):
    if isinstance(statement, SelectStatement):
        self.visit_select(statement)
    #...
    elif isinstance(statement, CreateTableStatement):
        self.visit_create(statement)
    return self._plan
```

21.4 3. Plan Generator

The part transformed logical plan to physical plan. The modified files are stored under **Optimizer** and **Planner** folders.

21.4.1 plan_nodes/

- `[cmd]_plan.py` - class [cmd]Plan, which stored information required for rename table.

```
class CreatePlan(AbstractPlan):
    def __init__(self, video_ref: TableRef,
                 column_list: List[DataFrameColumn],
                 if_not_exists: bool = False):
        super().__init__(PlanOprType.CREATE)
        self._video_ref = video_ref
        self._column_list = column_list
        self._if_not_exists = if_not_exists
    #...
```

- `types.py` - register new plan operator type to PlanOprType

21.4.2 optimizer/rules

- `rules.py`-
  - Import operators
  - Register new ruletype to **RuleType** and **Promise** (place it **before IMPLEMENTATION_DELIMETER** !!)
  - implement class `Logical[cmd]ToPhysical`, its memeber function apply() will construct a corresbonding `[cmd]Plan` object.

```
class LogicalCreateToPhysical(Rule):
    def __init__(self):
    pattern = Pattern(OperatorType.LOGICALCREATE)
    super().__init__(RuleType.LOGICAL_CREATE_TO_PHYSICAL, pattern)

def promise(self):
    return Promise.LOGICAL_CREATE_TO_PHYSICAL

def check(self, before: Operator, context: OptimizerContext):
    return True

def apply(self, before: LogicalCreate, context: OptimizerContext):
    after = CreatePlan(before.video, before.column_list, before.if_not_exists)
    return after
```

- `rules_base.py`-
  - Register new ruletype to **RuleType** and **Promise** (place it **before IMPLEMENTATION_DELIMETER** !!)
- `rules_manager.py`-
  - Import rules created in `rules.py`
  - Add imported logical to physical rules to `self._implementation_rules`

21.5 4. Plan Executor

`PlanExecutor` uses data stored in physical plan to run the command.

21.5.1 executor/

- `[cmd]_executor.py` - implement an executor that make changes in **catalog**, **metadata**, or **storage engine** to run the command.
  - May need to create helper function in CatalogManager, DatasetService, DataFrameMetadata, etc.

```
class CreateExecutor(AbstractExecutor):
    def exec(self):
        if (self.node.if_not_exists):
            # check catalog if we already have this table
            return
```

(continues on next page)

(continued from previous page)

```
        table_name = self.node.video_ref.table_info.table_name
        file_url = str(generate_file_path(table_name))
        metadata = CatalogManager().create_metadata(table_name, file_url, self.node.
↪column_list)

        StorageEngine.create(table=metadata)
```

21.6 Additional Notes

Key data structures in EVA:

- **Catalog**: Records `DataFrameMetadata` for all tables.
  - data stored in DataFrameMetadata: `name`, `file_url`, `identifier_id`, `schema`
    - `file_url` - used to access the real table in storage engine.
  - For the `RENAME` table command, we use the `old_table_name` to access the corresponing entry in metadata table, and the `modified name` of the table.
- **Storage Engine**:
  - API is defined in `src/storage`, currently only supports create, read, write.

CHAPTER

TWENTYTWO

EVA RELEASE GUIDE

22.1 Part 1: Before You Start

Make sure you have PyPI account with maintainer access to the EVA project. Create a .pypirc in your home directory. It should look like this:

```
[distutils]
index-servers =
pypi
pypitest

[pypi]
username=YOUR_USERNAME
password=YOUR_PASSWORD
```

Then run `chmod 600 ./.pypirc` so that only you can read/write the file.

22.2 Part 2: Release Steps

1. Ensure that you're in the top-level `eva` directory.
2. Ensure that your branch is in sync with the `master` branch:

```
$ git pull origin master
```

3. Add a new entry in the Changelog for the release.

```
##  [0.0.6]
### [Breaking Changes]
### [Added]
### [Changed]
### [Deprecated]
### [Removed]
```

Make sure `CHANGELOG.md` is up to date for the release: compare against PRs merged since the last release.

4. Update version to, e.g. `0.0.6` (remove the `+dev` label) in `eva/version.py`.
5. Commit these changes and create a PR:

```
git checkout -b release-v0.0.6
git add . -u
git commit -m "[RELEASE]: v0.0.6"
git push --set-upstream origin release-v0.0.6
```

6. Once the PR is approved, merge it and pull master locally.
7. Tag the release:

```
git tag -a v0.0.6 -m "v0.0.6 release"
git push origin v0.0.6
```

8. Build the source and wheel distributions:

```
rm -rf dist build  # clean old builds & distributions
python3 setup.py sdist  # create a source distribution
python3 setup.py bdist_wheel  # create a universal wheel
```

9. Check that everything looks correct by installing the wheel locally and checking the version:

```
python3 -m venv test_evadb  # create a virtualenv for testing
source test_evadb/bin/activate  # activate virtualenv
python3 -m pip install dist/evadb-0.9.1-py3-none-any.whl
python3 -c "import eva; print(eva.__version__)"
```

10. Publish to PyPI

```
pip install twine  # if not installed
twine upload dist/* -r pypi
```

11. A PR is automatically submitted (this will take a few hours) on [*conda-forge/eva-feedstock*](https://github.com/conda-forge/eva-feedstock) to update the version. * A maintainer needs to accept and merge those changes.
12. Create a new release on Github. * Input the recently-created Tag Version: `v0.0.6` * Copy the release notes in `CHANGELOG.md` to the GitHub tag. * Attach the resulting binaries in (`dist/evadb-x.x.x.*`) to the release. * Publish the release.
13. Update version to, e.g. `0.9.1+dev` in `eva/version.py`.
14. Add a new changelog entry for the unreleased version in *CHANGELOG.md*:

```
##  [Unreleased]
### [Breaking Changes]
### [Added]
### [Changed]
### [Deprecated]
### [Removed]
```

15. Commit these changes and create a PR:

```
git checkout -b bump-v0.9.1+dev
git add . -u
git commit -m "[BUMP]: v0.9.1+dev"
git push --set-upstream origin bump-v0.9.1+dev
```

16. Add the new tag to the EVA project on ReadTheDocs,

- Trigger a build for main to pull new tags.
- Go to the `Versions` tab, and `Activate` the new tag.
- Go to Admin/Advanced to set this tag as the new default version.
- **In `Overview`, make sure a build is triggered:**
  - For the tag `v0.9.1`
  - For `latest`

Credits: Snorkel

CHAPTER

TWENTYTHREE

PACKAGING

This section describes practices to follow when packaging your own models or datasets to be used along with EVA.

23.1 Models

Please follow the following steps to package models:

- Create a folder with a descriptive name. This folder name will be used by the UDF that is invoking your model.
- **Place all files used by the UDF inside this folder. These are typically:**
  - Model weights (The .pt files that contain the actual weights)
  - Model architectures (The .pt files that contain model architecture information)
  - Label files (Extra files that are used in the process of model inference for outputting labels.)
  - Other config files (Any other config files required for model inference)
- Zip this folder.
- Upload the zipped folder to this link inside the models folder.

23.2 Datasets

Please follow the following steps to package datasets:

- Create a folder for your dataset and give it a descriptive name.
- This dataset folder should contain 2 sub-folders named 'info' and 'videos'. For each video entry in the videos folder, there should be a corresponding CSV file in the info folder with the same name. The structure should look like:

```
anipi@blade:~/codes/eva$ tree data/datasets/bddtest/
data/datasets/bddtest/
├── info
│   ├── 00a04f658c891f94.csv
│   ├── 00a04f65af2ab984.csv
│   ├── 00a0f0083c67908e.csv
│   ├── 00a0f008a315437f.csv
│   ├── 00a2e3ca5c856cde.csv
│   ├── 00a2e3ca62992459.csv
│   ├── 00a2f5b6d4217a96.csv
│   ├── 00a395fed60c0b47.csv
│   ├── 00a820ef2b98dcf5.csv
│   └── 00a9cd6bb39be004.csv
└── videos
    ├── 00a04f658c891f94.mp4
    ├── 00a04f65af2ab984.mp4
    ├── 00a0f0083c67908e.mp4
    ├── 00a0f008a315437f.mp4
    ├── 00a2e3ca5c856cde.mp4
    ├── 00a2e3ca62992459.mp4
    ├── 00a2f5b6d4217a96.mp4
    ├── 00a395fed60c0b47.mp4
    ├── 00a820ef2b98dcf5.mp4
    └── 00a9cd6bb39be004.mp4
```

- The videos folder should contain the raw videos in a standard format like mp4 or mov.
- The info folder should contain the meta information corresponding to each video in CSV format. Each row of this CSV file should correspond to 1 unique object in a given frame. Please make sure the columns in your CSV file exactly match to these names. Here is a snapshot of a sample CSV file:

	A	B	C	D	E	F	G
1	id	frame_id	video_id	dataset_name	label	bbox	object_id
2	5699	0	3	bddtest	car	798.6408024113638, 240.2771362586605, 1052.2802666724026, 400.0394580512265	65349
3	5700	0	3	bddtest	car	739.5381062355658, 261.8937644341801, 845.1270207852194, 360	65350
4	5701	0	3	bddtest	car	707.9445727482679, 251.91685912240186, 807.7136258660508, 339.2147806004619	65351
5	5702	0	3	bddtest	car	478.717964316695, 272.0163071719262, 520.0833211152326, 298.941818301956	65352
6	5703	0	3	bddtest	car	688.5975996216332, 253.2197568376064, 731.6349495979853, 324.2814277287924	65353
7	5704	0	3	bddtest	car	645.4521625163827, 267.5229357798165, 675.64875491481, 297.71952817824376	65354

The columns represent the following:

  - id - (Integer) Auto incrementing index that is unique across all files (Since the CSV files are written to the same meta table, we want it to be unique across all files)
  - frame_id - (Integer) id of the frame this row corresponds to.
  - video_id - (Integer) id of the video this file corresponds to.
  - dataset_name - (String) Name of the dataset (should match the folder name)
  - label - (String) label of the object this row corresponds to.
  - bbox - (String) comma separated float values representing x1, y1, x2, y2 (top left and bottom right) coordinates of the bounding box

    - object_id - (Integer) unique id for the object corresponding to this row.

- Zip this folder.
- Upload the zipped folder to this link inside the datasets folder.

Note: In the future, will provide utility scripts along with EVA to download models and datasets easily and place them in the appropriate locations.

CHAPTER

TWENTYFOUR

VERSIONS

- stable version
- v0.1.3
- v0.1.2
- v0.1.1