Advanced Lane Finding Project

The goals / steps of this project are the following:

- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- Detect lane pixels and fit to find the lane boundary.
- Determine the curvature of the lane and vehicle position with respect to center.
- Warp the detected lane boundaries back onto the original image.
- Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

Writeup / README

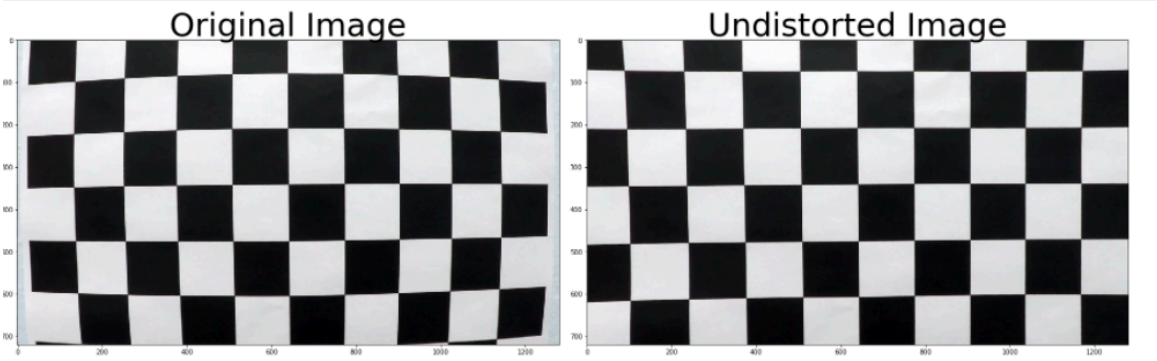
Camera Calibration

1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this step is contained in the "**Step1: Camera Calibration**" section of the IPython notebook located in "CarND-Advanced-Lane-Lines-Rober_v06.ipynb".

I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array of coordinates, and `objpoints` will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. `imgpoints` will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:



Pipeline (single images)

1. Provide an example of a distortion-corrected image.

The code for this step is contained in the "**Step2: Distortion correction**" section of the IPython notebook located in "CarND-Advanced-Lane-Lines-Rober_v06.ipynb".

To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:

2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

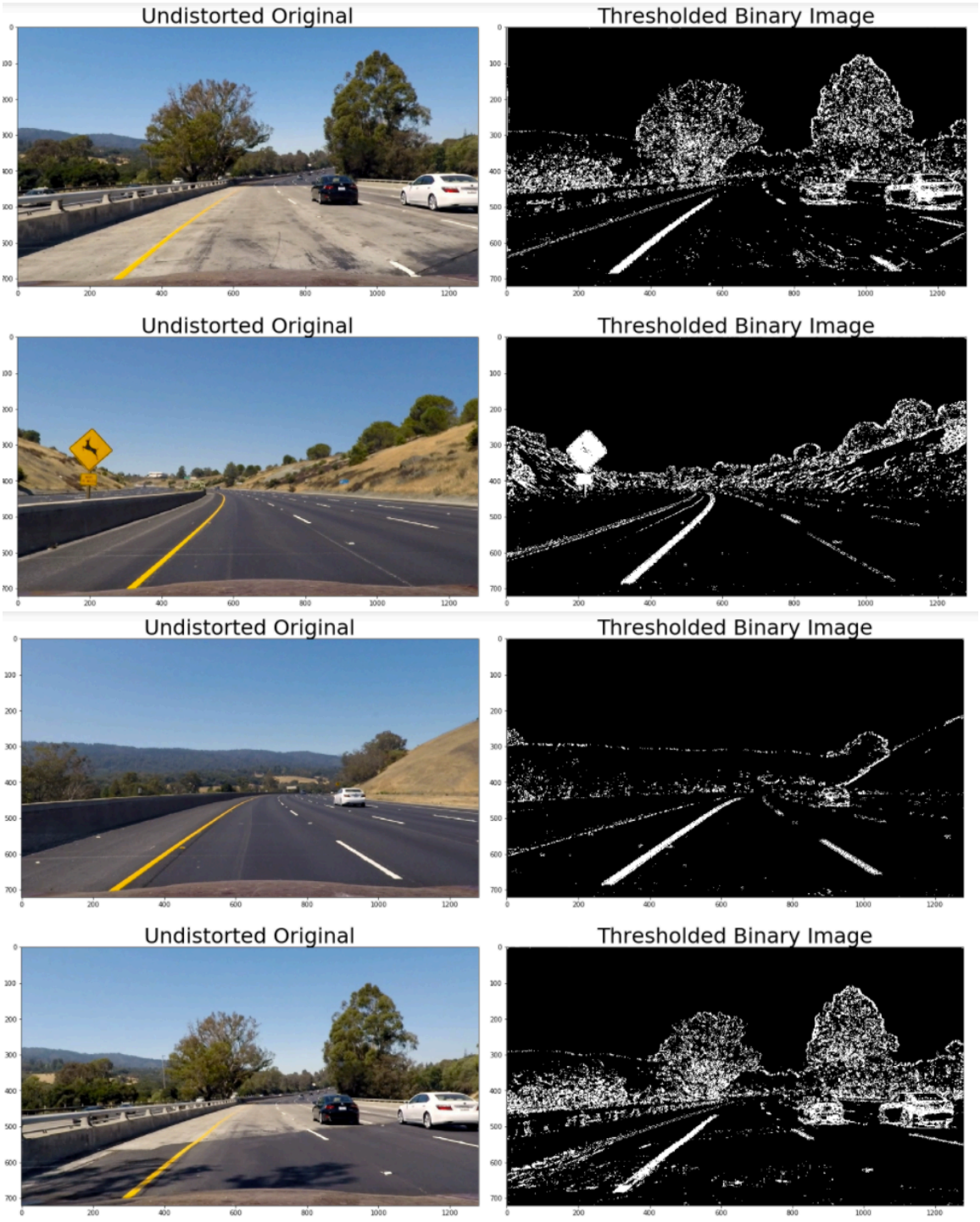
The code for this step is contained in the "**3. Color & Gradient Threshold**" section of the IPython notebook located in "CarND-Advanced-Lane-Lines-Rober_v06.ipynb" and the thresholding steps & functions in file `thresholding.py`.

I tested several combinations of color and gradient thresholds to generate a binary image.

Different channels of RGB and HLS spaces have been tested.
+ Gradient thresholds using HLS Space
+ Color thresholds using RGB & HLS Space

Finally, I select R channel from RGB and S channel from HLS to generate a combined image.

Here's some examples of my output for this step.



3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

The code for this step is contained in the "**4. Perspective Transform**" section of the IPython notebook located in "CarND-Advanced-Lane-Lines-Rober_v06.ipynb". The code for my perspective transform includes a function called `warper()`, which

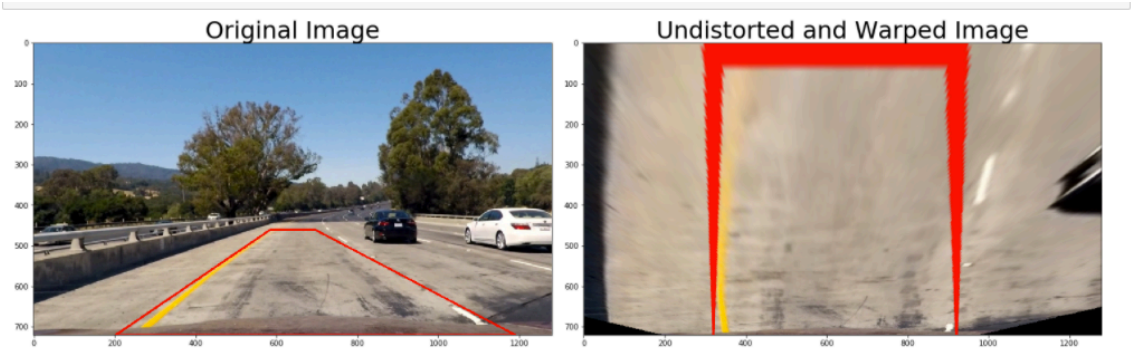
appears in the file `lines.py` .The `warper()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. I chose the hardcode the source and destination points in the following manner:

```python
src = np.float32(
   [[(img_size[0] / 2) - 55, img_size[1] / 2 + 100],
   [((img_size[0] / 6) - 10), img_size[1]],
   [(img_size[0] * 5 / 6) + 60, img_size[1]],
   [(img_size[0] / 2 + 55), img_size[1] / 2 + 100]])
dst = np.float32(
   [[(img_size[0] / 4), 0],
   [(img_size[0] / 4), img_size[1]],
   [(img_size[0] * 3 / 4), img_size[1]],
   [(img_size[0] * 3 / 4), 0]])
```

This resulted in the following source and destination points:

Source	Destination
585, 460	320, 0
203, 720	320, 720
1127, 720	960, 720
695, 460	960, 0

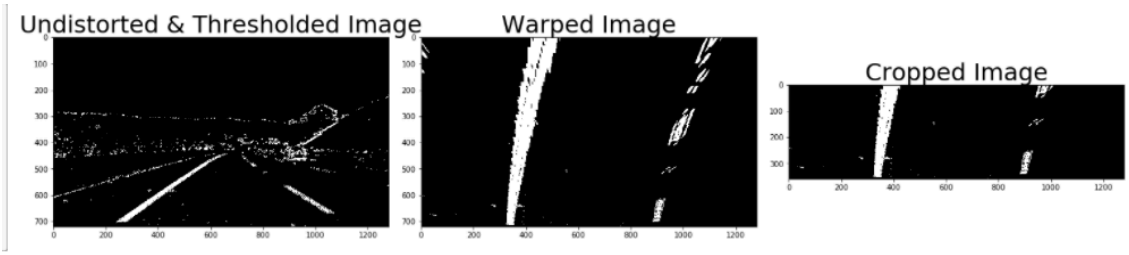
I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.

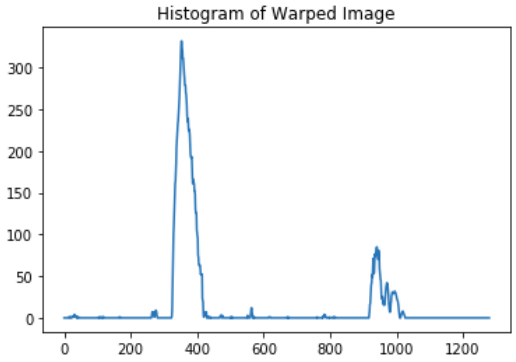


4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

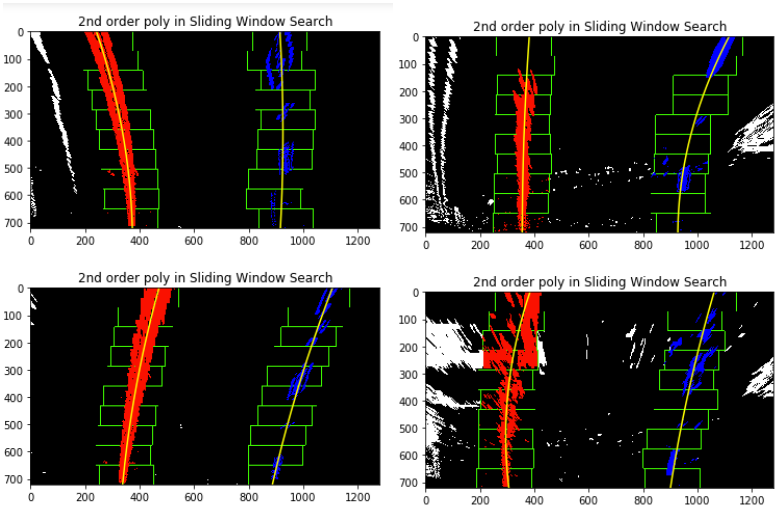
The code for this step is contained in the "**5. Detect lane lines**" section of the IPython notebook located in "CarND-Advanced-Lane-Lines-Rober_v06.ipynb".

I first take a **histogram** along all the columns in the *lower half* of the image like this:





Then I use 'fit_poly()' function,, which appears in the file `lines.py`, to fit my lane lines with a 2nd order polynomial kinda like this.

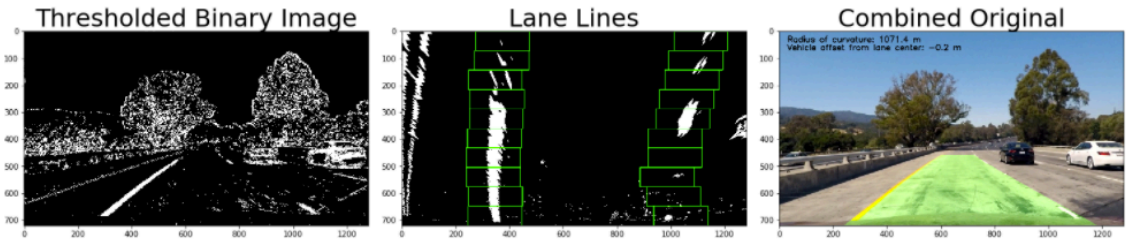


5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

The code for this step is contained in the “**6. Determine the Lane Curvature**” section of the IPython notebook located in "CarND-Advanced-Lane-Lines-Rober_v06.ipynb".
I use ‘find_curvature ()’ function, which is defined in lines 178-203, in the file `lines.py`.
I use ‘find_vehicle_offset ()’ function, which is defined in lines 208-223, in the file `lines.py

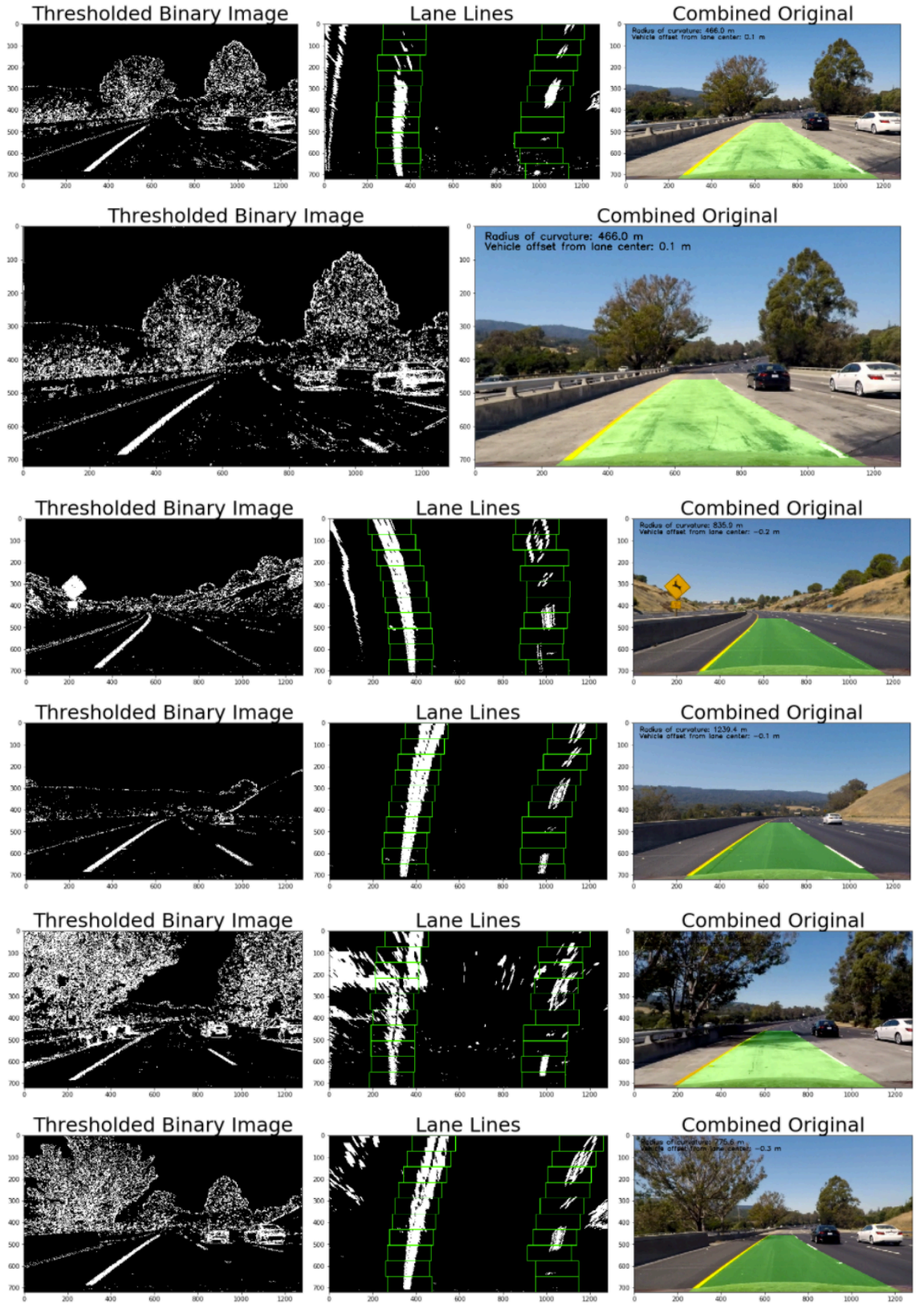
Here's some examples of my output for this step.

```
Radius of curvature 1172.37680842 meters right_curverad 970.387903886 meters
Vehicle offset:  -0.169017711576 meters
Radius of curvature 659.712384324 meters right_curverad 1012.16446408 meters
Vehicle offset:  -0.241873519301 meters
Radius of curvature 1269.00520592 meters right_curverad 1209.86535763 meters
Vehicle offset:  -0.0777966175549 meters
Radius of curvature 3272.79521854 meters right_curverad 119.050390505 meters
Vehicle offset:  -0.208670254844 meters
Radius of curvature 430.827996003 meters right_curverad 1712.74165231 meters
Vehicle offset:  -0.0175863772592 meters
Radius of curvature 1139.82624125 meters right_curverad 411.295316108 meters
Vehicle offset:  -0.272162638563 meters
```



6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

The code for this step is contained in the “**7. Drawing**” section of the IPython notebook located in "CarND-Advanced-Lane-Lines-Rober_v06.ipynb”. Here is an example of my result on a test image:


Thresholded Binary Image
Lane Lines
Combined Original
Radius of curvature: 466.0 m
Vehicle offset from lane center: 0.1 m
Thresholded Binary Image
Combined Original
Radius of curvature: 466.0 m
Vehicle offset from lane center: 0.1 m
Thresholded Binary Image
Lane Lines
Combined Original
Radius of curvature: 835.9 m
Vehicle offset from lane center: −0.2 m
Thresholded Binary Image
Lane Lines
Combined Original
Radius of curvature: 1239.4 m
Vehicle offset from lane center: −0.1 m
Thresholded Binary Image
Lane Lines
Combined Original
Thresholded Binary Image
Lane Lines
Combined Original
Radius of curvature: 276.6 m
Vehicle offset from lane center: −0.3 m

Pipeline (video)

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

Here's a [link to my video result](./project_video_output_v06.mp4)

Discussion

1. Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

My pipeline is very fragile. Manual feature engineering is a hard task. You need much time to do a good job. When you think that your system is ok, any a little change in the input ecosystem and your system is broken. After of hard coding hours, your system will get back to work. But this is not the way to follow.

IMHO, deep learning is the tool for all these tasks.

In his last QA video, "Q&A with Sebastian Thrun", Thrun talks about use DL for this tasks. https://youtu.be/a8qva_gpwBU

Francois Chollet (Keras), in his last book 'Deep Learning with Python', gives several properties of DL that justify its use for theses tasks:

‘’’’

These important properties can be broadly sorted into 3 categories:

***Simplicity**. Deep learning removes the need for feature engineering, replacing complex, brittle and engineering-heavy pipelines with simple end-to-end trainable models typically built using only 5 or 6 different tensor operations.*

***Scalability**. Deep learning is highly amenable to parallelization on GPUs or TPUs, making it capable of taking full advantage of Moore's law. Besides, deep learning models are trained by iterating over small batches of data, allowing them to be trained*

on datasets of arbitrary size (the only bottleneck being the amount of parallel computational power available, which thanks to Moore's law is a fast-moving barrier).

***Versatility and reusability**. Contrarily to many prior machine learning approaches, deep learning models can be trained on additional data without restarting from scratch, making them viable for continuous online learning, an important property for very large production models. Furthermore, trained deep learning models are repurposeable and thus resuable: for instance it is possible to take a deep learning model trained for image classification and drop it into a video processing pipeline. This allows us to reinvest previous work into increasingly complex and powerful models.*

"""