Machine Learning Systems Design

Modeling Pipeline

Lecture 10: Model Development and Training

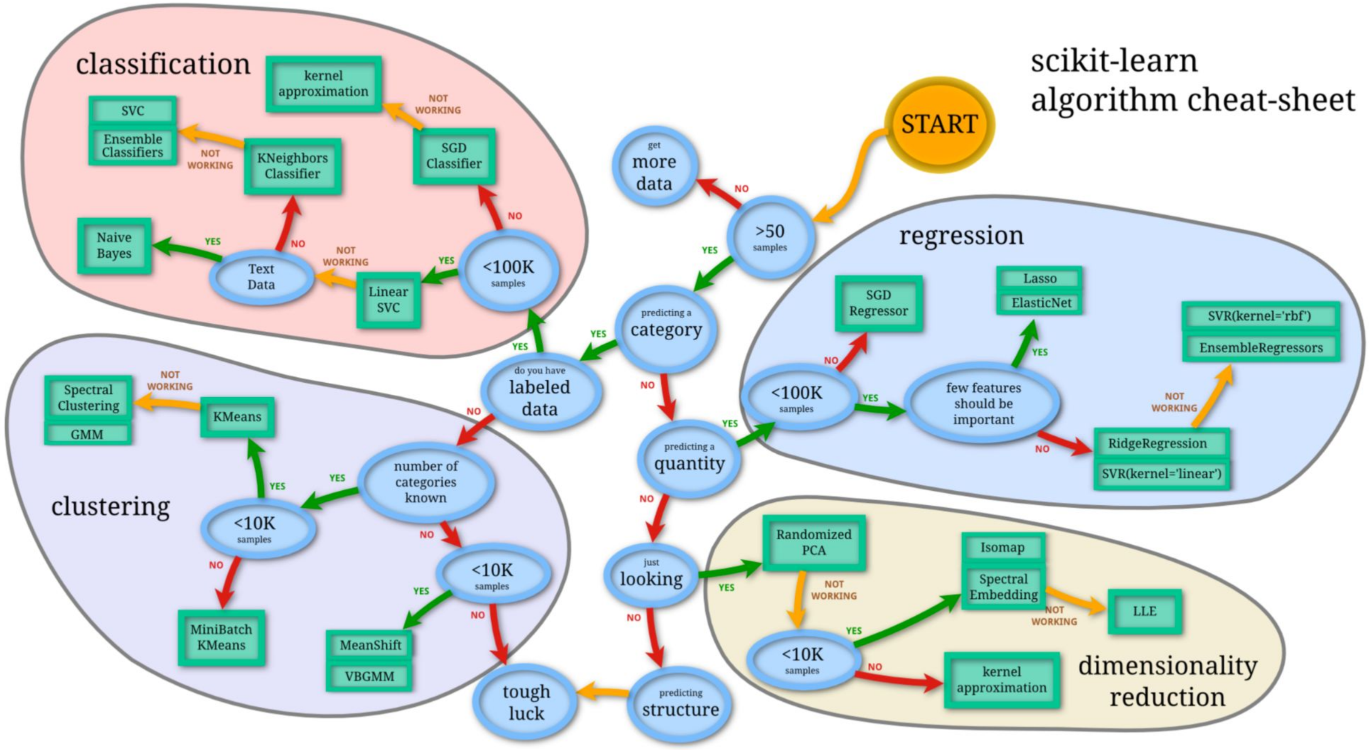
CE 40959 Spring 2023
Ali Zarezade
SharifMLSD.github.io

Agenda

1. Before Start
2. End-to-End vs Pipeline
3. Error Analysis

1. Before Start

Frame the ML problem



Before start

- Frame the ML problem
- Choose a performance metric
  - GMR#2: First, design and implement metrics
  - Establish a single evaluation metric to optimize

Before start

- Frame the ML problem
- Choose a performance metric
- Define an achievable performance level
    - If possible use human performance level (HPL)
    - Use other similar model results

Before start

- Frame the ML problem
- Choose a performance metric
- Define an achievable performance level
- Choose baselines
  - random prediction
  - zero rule

Before start

- Frame the ML problem
- Choose a performance metric
- Define an achievable performance level
- Choose the right baselines
- Split data to train, dev and test sets
    - Dev and test should be from the same distribution
    - Beware of data leakage

Google ML rules: before start

- Rule #1: Don’t be afraid to launch a product without machine learning.
- Rule #2: First, design and implement metrics.
- Rule #3: Choose machine learning over a complex heuristic.

Google ML rules: phase 1, first pipeline

- Rule #4: Keep the first model simple and get the infrastructure right.
    - Your simple model provides you with baseline metrics and a baseline behavior that you can use to test more complex models

Google ML rules: phase 1, first pipeline

- Rule #4: Keep the first model simple and get the infrastructure right.
- Rule #5: Test the infrastructure independently from the machine learning.
    - Test getting data into the algorithm
    - Test getting models out of the training algorithm

2. End-to-End vs Pipeline

The rise of end-to-end learning

Suppose you want to build a system to examine online product reviews and automatically tell you if the writer liked or disliked that product.

+ This is a great mop!
- This mop is low quality--I regret buying it.

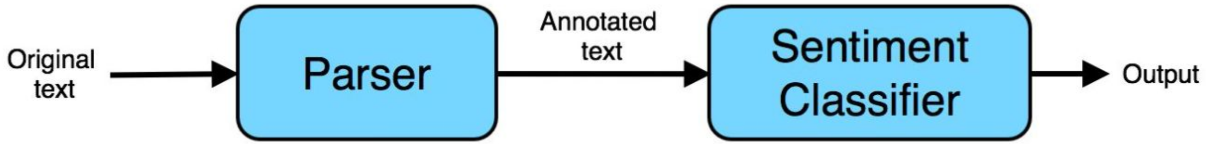
Pipeline or component-based learning

Build a “pipeline” of two components:

- Parser: A system that annotates the text with information identifying the most important words.

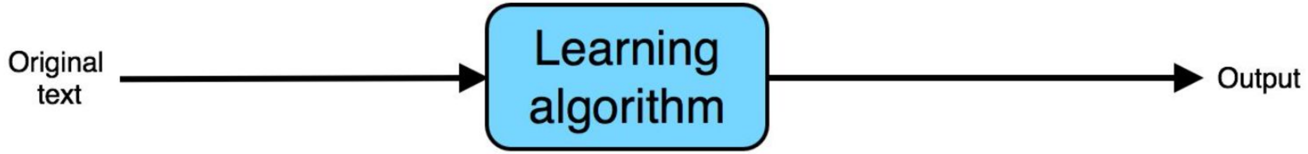
  This is a great$_{\text{Adjective}}$ mop$_{\text{Noun}}$!

- Sentiment classifier: A learning algorithm that takes as input the annotated text and predicts the overall sentiment.



End-to-end learning

An end-to-end learning algorithm for this task would simply take as input the raw, original text “This is a great mop!”, and try to directly recognize the sentiment,



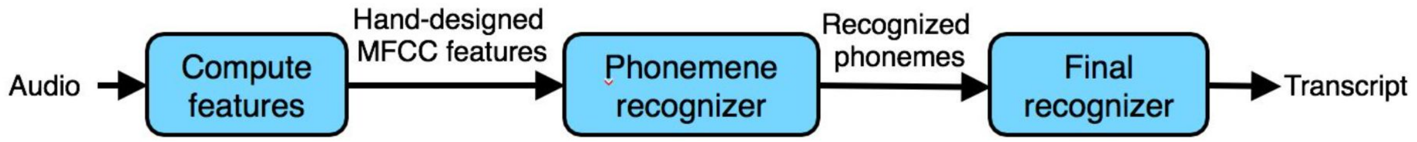
the learning algorithm directly connects the “input end” of the system to the “output end.”

End-to-end or pipeline?

In problems where data is abundant, end-to-end systems have been remarkably successful. But they are not always a good choice. why?

More end-to-end learning examples

Speech recognition system:

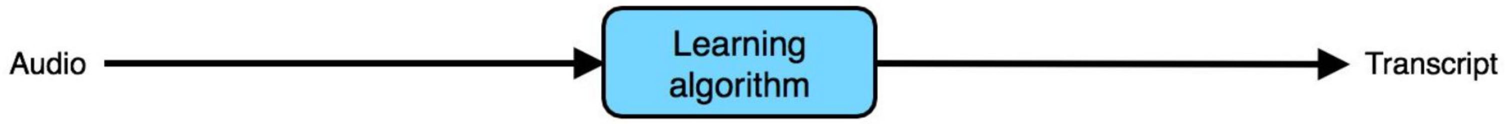


The components work as follows:

1. Compute features
2. Phoneme recognizer
3. Final recognizer

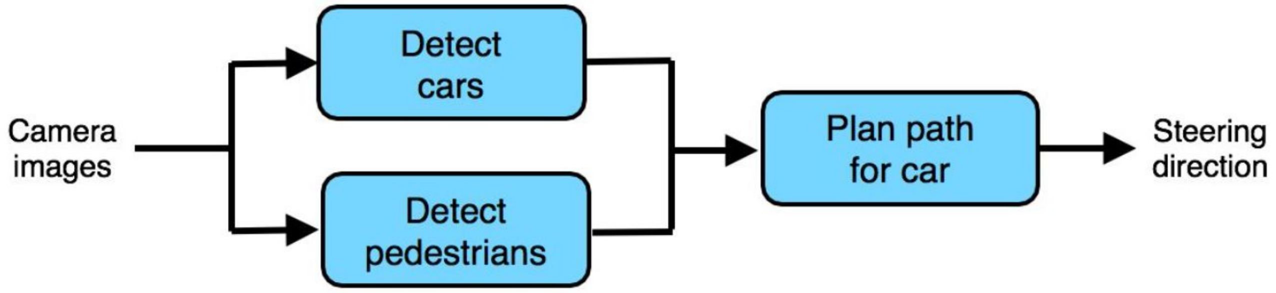
More end-to-end learning examples

In contrast, an end-to-end system might input an audio clip, and try to directly output the transcript:



More end-to-end learning examples

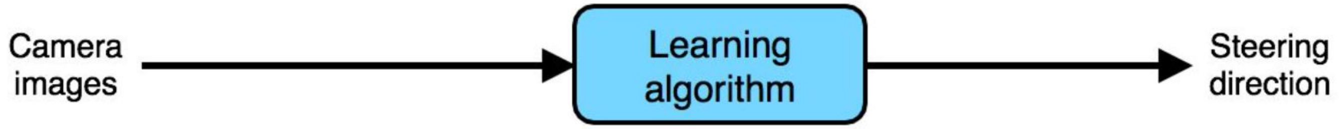
Pipelines can be more complex. For example, here is a simple architecture for an autonomous car:



Not every component in a pipeline has to be learned. For example, the literature on “robot motion planning” has numerous algorithms for the final path planning step for the car. Many of these algorithms do not involve learning

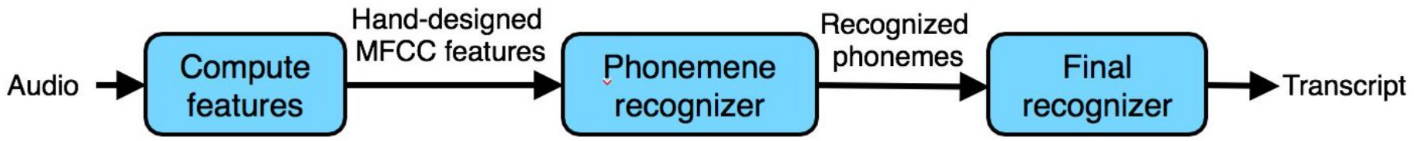
More end-to-end learning examples

In contrast, and end-to-end approach might try to take in the sensor inputs and directly output the steering direction:



Pros and cons of end-to-end learning

Consider the same speech pipeline from our earlier example:



Pros and cons of end-to-end learning

Cons of pipeline:

- **MFCCs** are a set of **hand-designed** audio features. Although they provide a reasonable summary of the audio input, they also simplify the input signal by throwing some information away.
- **Phonemes** are an invention of linguists. They are an **imperfect** representation of speech sounds. To the extent that phonemes are a poor approximation of reality, forcing an algorithm to use a phoneme representation will limit the speech system’s performance.

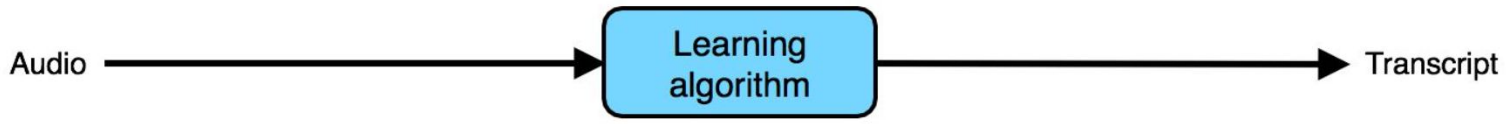
Pros and cons of end-to-end learning

Pros of pipelines:

- The **MFCC** features are **robust** to some properties of speech that do not affect the content, such as speaker pitch. Thus, they help simplify the problem for the learning algorithm.
- To the extent that **phonemes** are a **reasonable** representation of speech, they can also help the learning algorithm understand basic sound components and therefore improve its performance.
- Having more hand-engineered components generally allows a system to **learn with less data**.

Pros and cons of end-to-end learning

Now, consider the end-to-end system:



Pros and cons of end-to-end learning

Cons:

- This system lacks the hand-engineered knowledge. Thus, when the training set is small, it might do worse than the hand-engineered pipeline.

Pros:

- However, when the training set is large, then it is not hampered by the limitations of an MFCC or phoneme-based representation. If the learning algorithm is a large-enough neural network and if it is trained with enough training data, it has the potential to do very well, and perhaps even approach the optimal error rate.

End-to-end or pipeline?

It depends on:

- Data availability
- Task simplicity

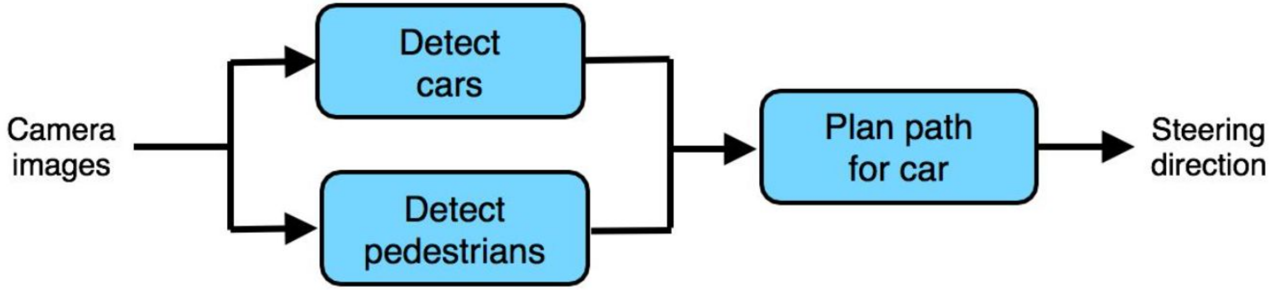
How to choose pipeline components?

When building a components-based system, what are good candidates for the components of the pipeline? How you design the pipeline will greatly impact the overall system’s performance.

Choosing pipeline components: Data availability

One important factor is whether you can easily collect data to train each of the components.

If there is a lot of data available for training “intermediate modules” of a pipeline (such as a car detector or a pedestrian detector), then you might consider using a pipeline with multiple stages.



Choosing pipeline components: Task simplicity

How simple are the tasks solved by the individual components?

You should try to choose pipeline components that are individually easy to build or learn. But what does it mean for a component to be “easy” to learn?

Choosing pipeline components: Task simplicity

Consider these machine learning tasks, listed in order of increasing difficulty:

1. Classifying whether an image is overexposed.
2. Classifying whether an image was taken indoor or outdoor
3. Classifying whether an image contains a cat
4. Classifying whether an image contains a cat with both black and white fur
5. Classifying whether an image contains a Siamese cat (a particular breed of cat)

Choosing pipeline components: Task simplicity

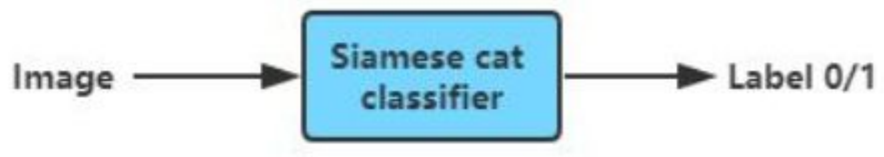
If you are able to take a complex task, and break it down into simpler sub-tasks, then by coding in the steps of the sub-tasks explicitly, you are giving the algorithm prior knowledge that can help it learn a task more efficiently.

Choosing pipeline components: Task simplicity

Suppose you are building a Siamese cat detector.

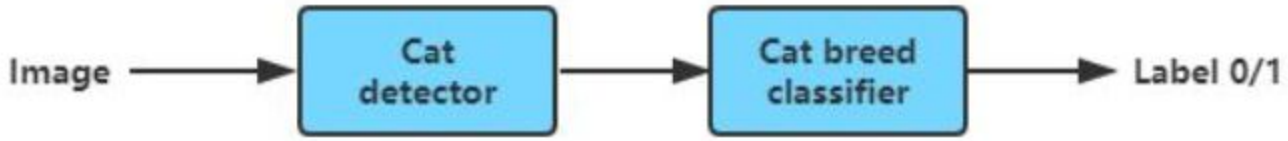
Choosing pipeline components: Task simplicity

This is the pure end-to-end architecture:



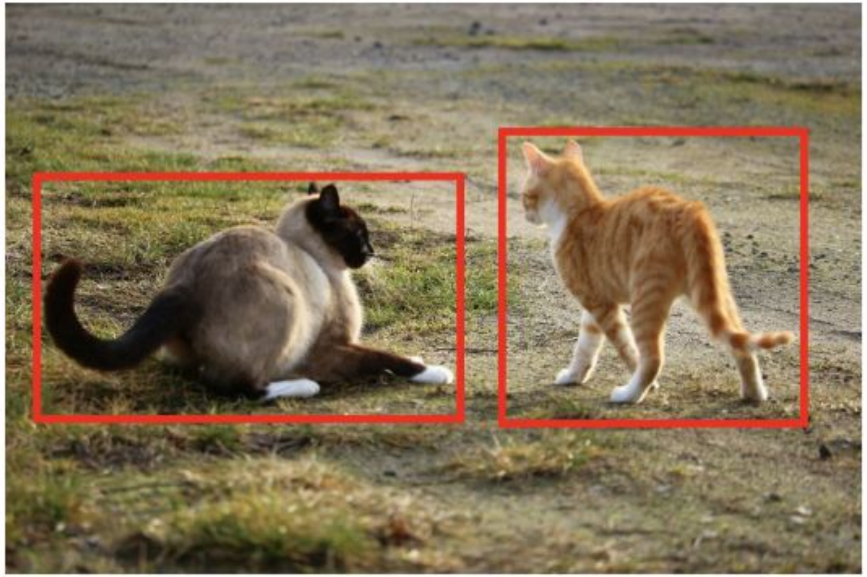
Choosing pipeline components: Task simplicity

In contrast, you can alternatively use a pipeline with two steps:



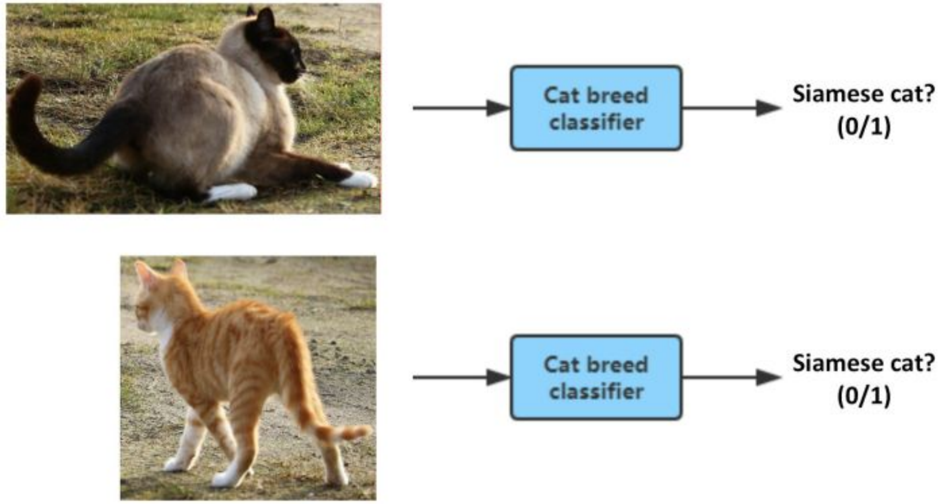
Choosing pipeline components: Task simplicity

The first step (cat detector) detects all the cats in the image.



Choosing pipeline components: Task simplicity

The second step then passes cropped images of each of the detected cats (one at a time) to a cat species classifier, and finally outputs 1 if any of the cats detected is a Siamese cat.



Choosing pipeline components: Task simplicity

Compared to training a purely end-to-end classifier using just labels 0/1, each of the two components in the pipeline--the cat detector and the cat breed classifier--seem much easier to learn and will require significantly less data.

Choosing pipeline components: Task simplicity

In summary, when deciding what should be the components of a pipeline, try to build a pipeline where each component is a relatively “simple” function that can therefore be learned from only a modest amount of data.

3. Error Analysis

Build your first system quickly then iterate

You want to build a new email anti-spam system, there are several ideas:

- Collect a huge training set of spam email.
- Develop features for understanding the text content of the email.
- Develop features for understanding the email envelope/header features
- And more ...

Which one should we follow?

Build your first system quickly then iterate cont.

It is a hard decision, so don’t start off trying to build the perfect system. Instead:

- Build a basic system quickly (in just a few days).
- Then, find clues for the most promising directions to invest your time.

But how find the best direction to follow?

What to do what not to do!

Consider a cat app, after building your first model you may realise that most ot the time it mistakes dogs for cats!

A team member proposes incorporating 3rd party software that will make the system do better on dog images.

Should you ask them to go ahead?

Error analysis

Before investing a month on a 3rd party app, first estimate how much it will actually improve the system’s accuracy, by **error analysis**:

- Gather a sample of 100 dev set examples that your system *misclassified*
- Look at them manually, and count dog images fraction.
- If only 5% of the misclassified images are dogs, and if your overall system is currently 90% accurate, a perfect 3rd party at best result in 90.5% accuracy!

Error analysis

The process of examining dev set examples that your algorithm misclassified, to understand the underlying causes of the errors and then prioritize projects and inspire new directions.

Error analysis

Your team has several ideas for improving the cat detector:

- Fix the problem of your algorithm recognizing *dogs* as cats
- Fix the problem of your algorithm recognizing *great cats*
- Improve the system's performance on *blurry* images

You can efficiently **evaluate all of these ideas in parallel**, by looking through ~100 misclassified dev set images.

Evaluating multiple ideas

Image	Dog	Great cat	Blurry	Comments
1	✔			Unusual pitbull color
2			✔	
3		✔	✔	Lion; picture taken at zoo on rainy day
4		✔		Panther behind tree
% of total	25%	50%	50%	

Evaluating multiple ideas

- After manually categorizing some images, you might think of new categories and re-examine the images in light of the new categories.
- The most helpful error categories will be ones that you have an idea for improving.
- But you don't have to restrict yourself only to error categories you know how to improve; the goal of this process is to build your intuition about the most promising areas to focus on.
- ***Error analysis is an iterative process.***

Evaluating multiple ideas

If you finish the error analysis as follows, then

- Working on the Dog mistakes can eliminate 8% of the errors at most.
- Working on Great Cat or Blurry image errors could help eliminate more errors.

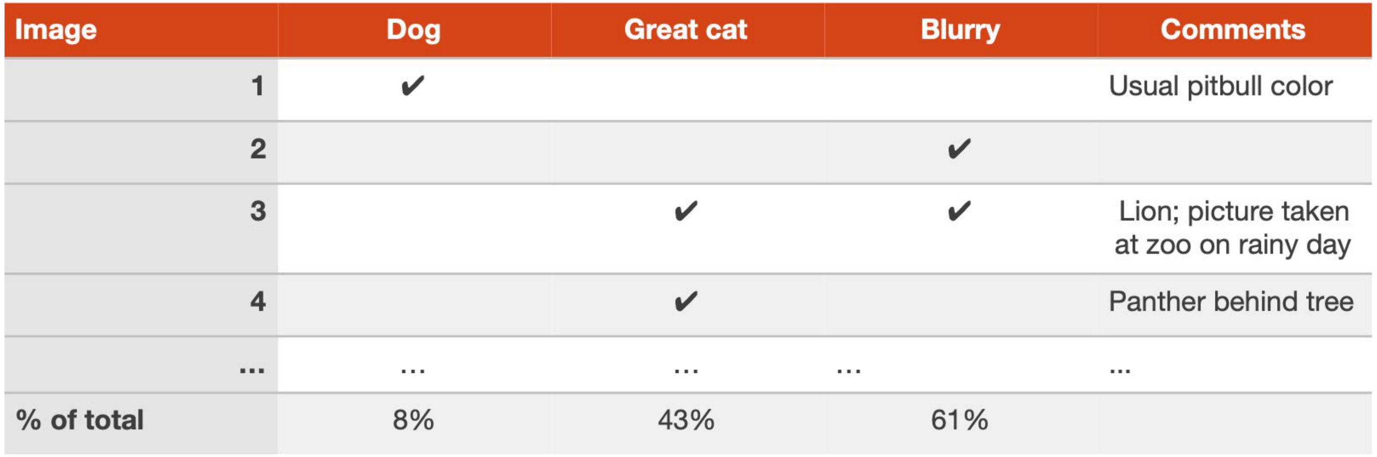


Image	Dog	Great cat	Blurry	Comments
1	✔			Usual pitbull color
2			✔	
3		✔	✔	Lion; picture taken at zoo on rainy day
4		✔		Panther behind tree
...	...	...	...	...
% of total	8%	43%	61%	

Evaluating multiple ideas

But, error analysis does not produce a rigid mathematical formula that tells you what the highest priority task should be.

You also have to take into account other trade-offs like; how much progress you expect to make on different categories, and the amount of work needed to tackle each one, cost, time and so on.

Cleaning up mislabeled dev & test set examples

During error analysis, you might notice some mislabeled examples in your dev set. If it is significant, add a category to keep track of the fraction of mislabeled examples.

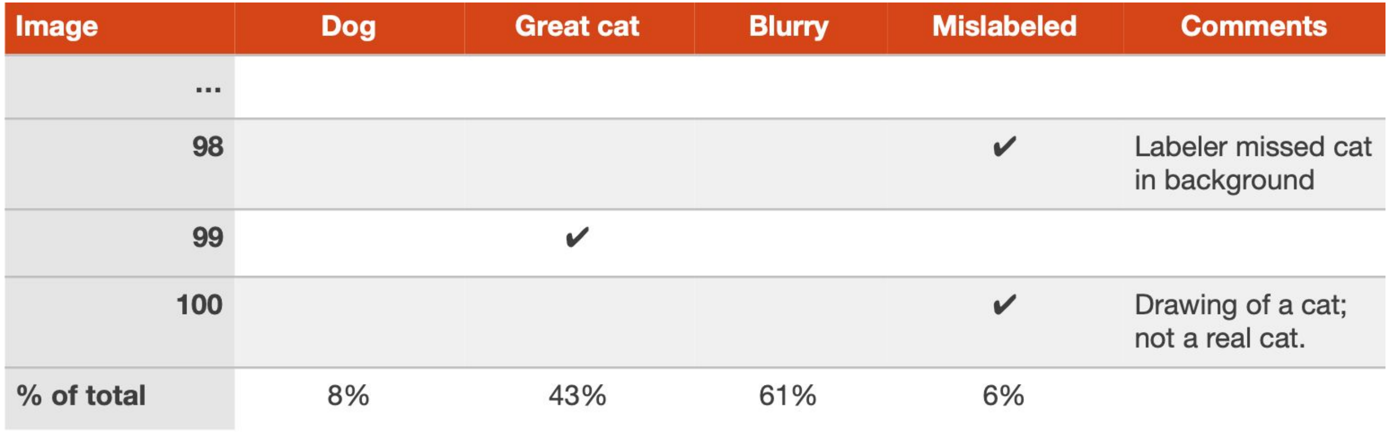


Image	Dog	Great cat	Blurry	Mislabeled	Comments
...					
98				✔	Labeler missed cat in background
99		✔			
100				✔	Drawing of a cat; not a real cat.
% of total	8%	43%	61%	6%	

Cleaning up mislabeled dev & test set cont.

Suppose your classifier’s performance is:

- Overall accuracy on dev set.................... 90% (10% overall error.)
- Errors due to mislabeled examples...... 0.6% (6% of dev set errors.)
- Errors due to other causes...................... 9.4% (94% of dev set errors)

Should you correct the labels in our dev set?

Cleaning up mislabeled dev & test set cont.

Here, the 0.6% inaccuracy due to mislabeling might not be significant enough relative to the 9.4% of errors you could be improving.

There is no harm in manually fixing the mislabeled images in the dev set, but it is not crucial to do so.

Cleaning up mislabeled dev & test set cont.

Suppose you keep improving the cat classifier and reach the following:

- Overall accuracy on dev set.................... 98.0% (2.0% overall error.)
- Errors due to mislabeled examples....... 0.6%. (30% of dev set errors.)
- Errors due to other causes..................... 1.4% (70% of dev set errors)

Should you correct the labels in our dev set now?

Cleaning up mislabeled dev & test set cont.

Now, 30% of your errors are due to the mislabeled dev set images, adding significant error to your estimates of accuracy.

It is now worthwhile to improve the quality of the labels in the dev set.

Cleaning up mislabeled dev & test set cont.

It is not uncommon to start off tolerating some mislabeled dev/test set examples, only later to change your mind as your system improves so that the fraction of mislabeled examples grows relative to the total set of errors.

Whatever process you apply to fixing dev set labels, remember to apply it to the test set labels too so that your dev and test sets continue to be drawn from the **same distribution**.

If you decide to improve the label quality, consider double-checking both the labels of examples that your system misclassified as well as labels of examples it correctly classified! *But*, ***why?***

Cleaning up mislabeled dev & test set cont.

If you have 1,000 dev set examples, and if your classifier has 98.0% accuracy, it is easier to examine the 20 examples it misclassified than to examine all 980 examples classified correctly.

But, it is possible that both the original label and your learning algorithm were wrong on an example. If you fix only the labels of examples that your system had misclassified, you might introduce bias into your evaluation.

If you have a large dev set, split it into two subsets, only one of which you look at

Suppose you have a large dev set of 5,000 examples with a 20% error rate. So, you misclassifying ~1,000 dev images. It takes a long time to manually examine 1,000 images, so we might decide not to use all of them in the error analysis.

In this case, you can split the dev set into two subsets:

- **Eyeball dev set** is one of which you look at.
- **Blackbox dev set** is one of which you don’t look at.

If you have a large dev set cont.

Suppose we want to manually examine about 100 errors for error analysis.

- **Eyeball dev set**: randomly select 10% of the dev set (500 examples, of which we would expect our algorithm to misclassify about 100)
- **Blackbox dev set:** the remaining subset of dev set (4500 examples)

Why do we explicitly separate the dev set into Eyeball and Blackbox dev sets?

If you have a large dev set cont.

Since you will gain intuition about the examples in the Eyeball dev set, you will start to overfit the Eyeball dev set faster.

- You will more rapidly overfit on *Eyeball*.
- You can use *Blackbox* for tuning parameters.
- If you overfit on Eyeball, discard it and fine a new one.

How big should the Eyeball and Blackbox dev sets be?

Your Eyeball dev set should be large enough to give you a sense of your algorithm’s major error categories. On a task that humans do well:

- With 10 mistakes, it’s hard to accurately estimate the impact of different error
- With 20 mistakes, you would start to get a rough sense of the major error sources.
- With ~50 mistakes, you would get a good sense of the major error sources.
- With ~100 mistakes, you would get a very good sense of the major sources of errors.
- More mistakes is good, but it is limited by how much time you have to manually analyze

If your classifier has a 5% error rate. To make sure you have ~100 misclassified examples in the Eyeball dev set, the Eyeball dev set should have ~2,000 examples.

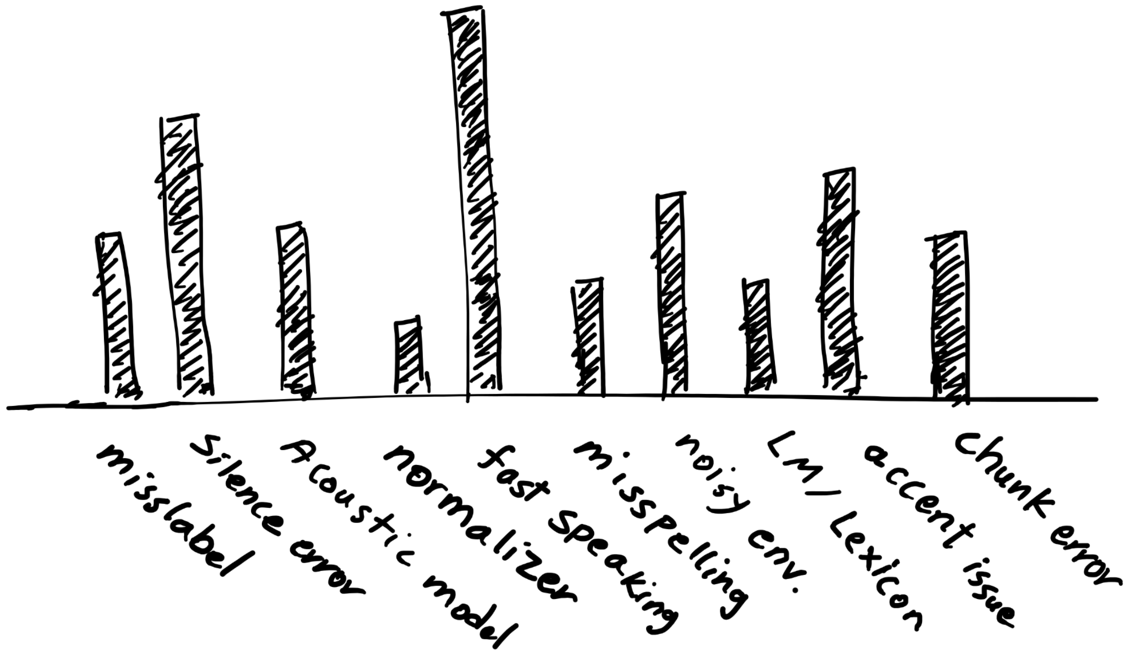
How big should the Eyeball and Blackbox dev sets be? cont.

How about the Blackbox dev set?

- About 1,000-10,000 samples, is enough to tune hyperparameters and select among models.
- About 100 samples, is small but still useful.
- For smaller sizes, your entire dev set might have to be used as the Eyeball dev set.

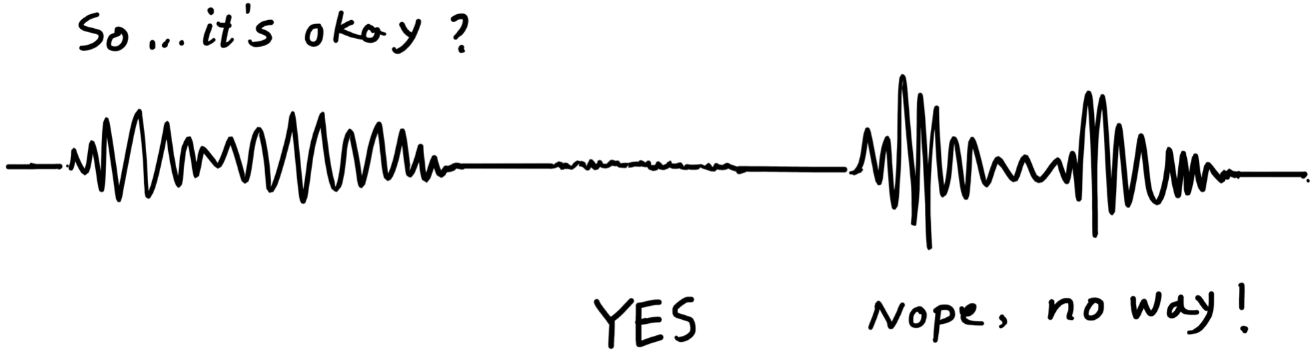
Next step: create some hypothesis

After doing error analysis, hopefully you have some causes for error that your model generate.



Next step: create some hypothesis

The next step is to create some hypothesis, implement them and finally test whether they improve performance of not.



Summary: basic error analysis

- When you start a new project, especially if it is in an area in which you are not an expert, **it is hard to correctly guess the most promising directions**.
- **So don't start off trying to design and build the perfect system**. Instead build and train a basic system as quickly as possible—perhaps in a few days. Then use error analysis to help you identify the most promising directions and iteratively improve your algorithm from there.
- Carry out **error analysis** by manually examining ~100 dev set examples the algorithm misclassifies and counting the major categories of errors. Use this information to **prioritize what types of errors to work on fixing**.

Summary: basic error analysis

- Consider splitting the dev set into an **Eyeball** dev set, which you will manually examine, and a **Blackbox** dev set, which you will not manually examine. If performance on the Eyeball dev set is much better than the Blackbox dev set, you have overfit the Eyeball dev set and should consider acquiring more data for it.
- The Eyeball dev set should be big enough so that your algorithm misclassifies enough examples for you to analyze. A Blackbox dev set of 1,000-10,000 examples is sufficient for many applications.
- If your dev set is not big enough to split this way, just use the entire dev set as an Eyeball dev set for manual error analysis, model selection, and hyperparameter tuning.

Error Analysis on pipelines

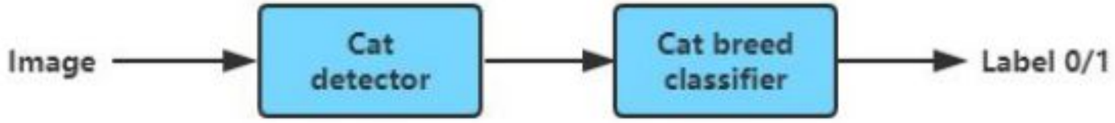
Suppose your system is built using a complex machine learning pipeline, and you would like to improve the system's performance. Which part of the pipeline should you work on improving?

Example: Siamese cat classifier

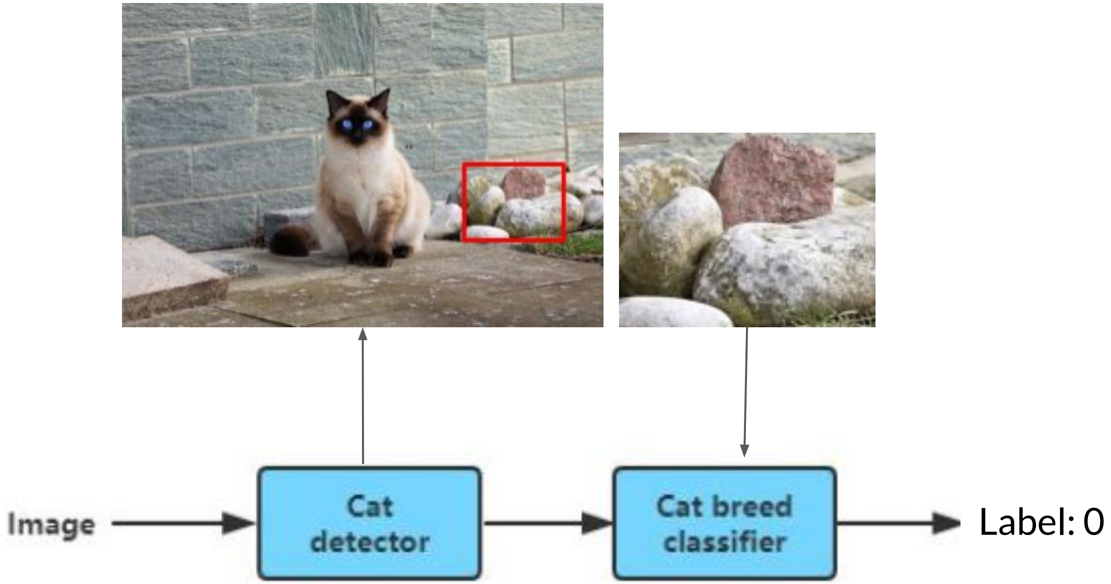
The first part, the cat detector, detects cats and crops them out of the image. The second part, the cat breed classifier, decides if it is a Siamese cat.

It is possible to spend years working on improving either of these two pipeline components.

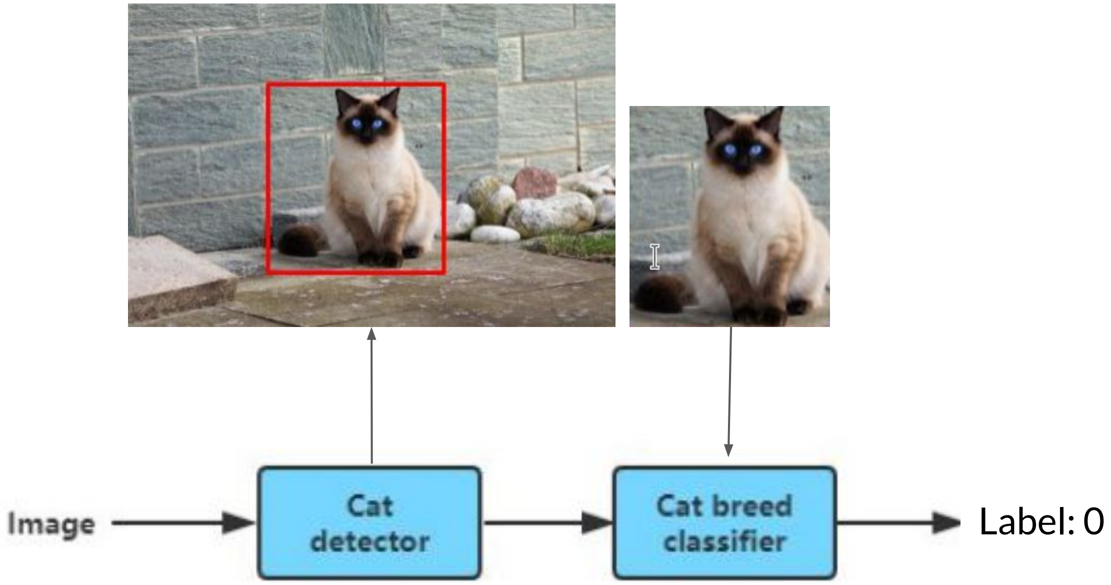
How to decide which component(s) to focus on?



Example: case 1

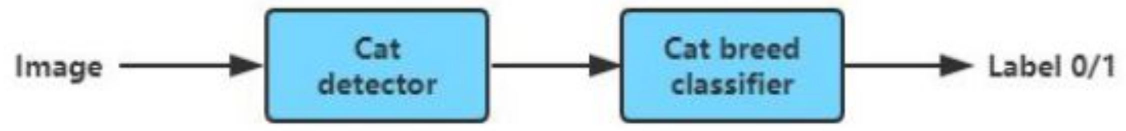


Example: case 2



Solution

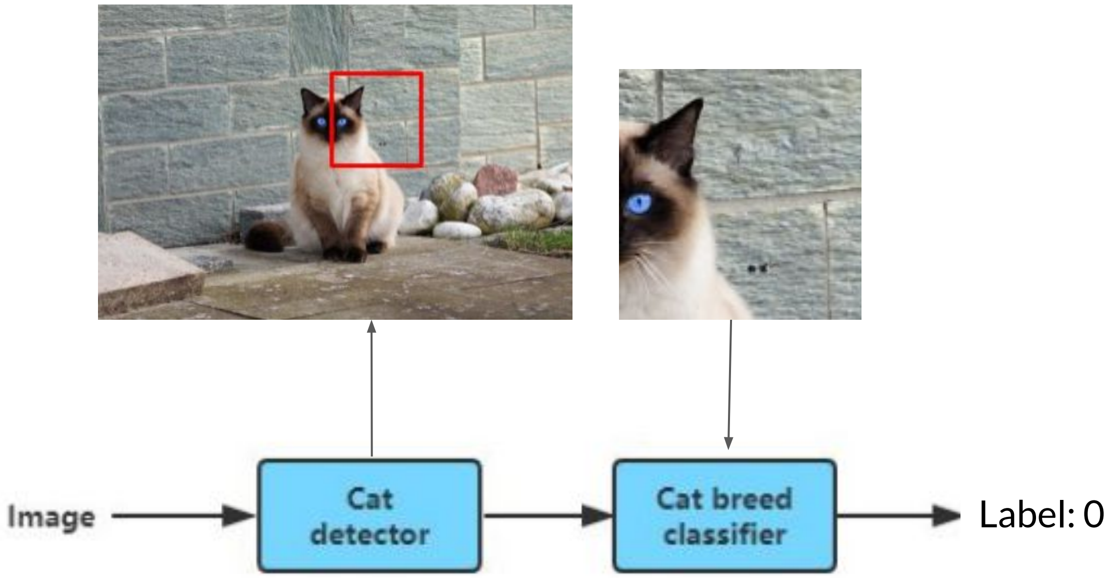
Say you go through 100 misclassified dev set images and find that 90 of the errors are attributable to the cat detector, and only 10 errors are attributable to the cat breed classifier. You can safely conclude that you should focus more attention on improving the cat detector.



90%   10%

Note: You can use these 90 examples to carry out a deeper level of error analysis on the cat detector to see how to improve that.

Example: Attributing error to one part

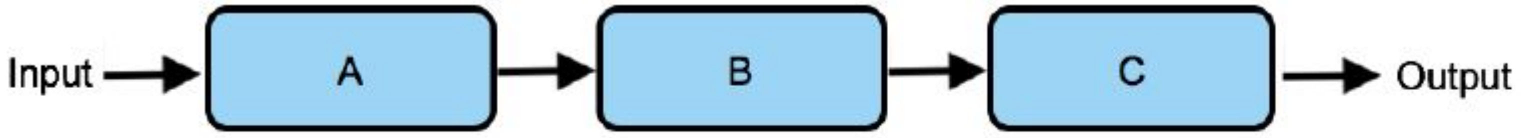


Solution

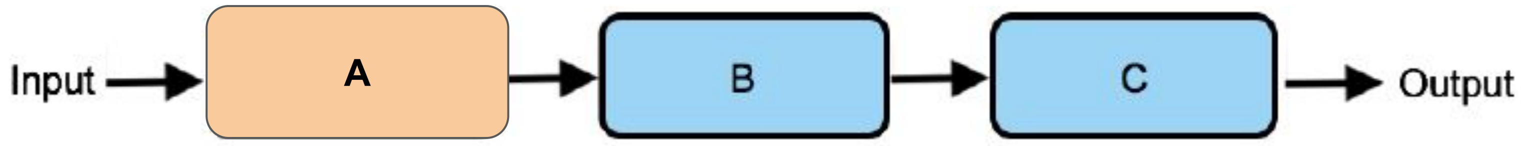
- If the number of ambiguous cases like these is small, you can make whatever decision you want and get a similar result.
- But here is a more formal test that lets you more definitively attribute the error to exactly one part:
  - Replace the cat detector output with a hand-labeled bounding box.
  - Run the corresponding cropped image through the cat breed classifier. If the cat breed classifier still misclassifies it, attribute the error to the cat breed classifier. Otherwise, attribute the error to the cat detector.

General case of error attribution

Here are the general steps for error attribution. Suppose the pipeline has three steps A, B and C, where A feeds directly into B, and B feeds directly into C.

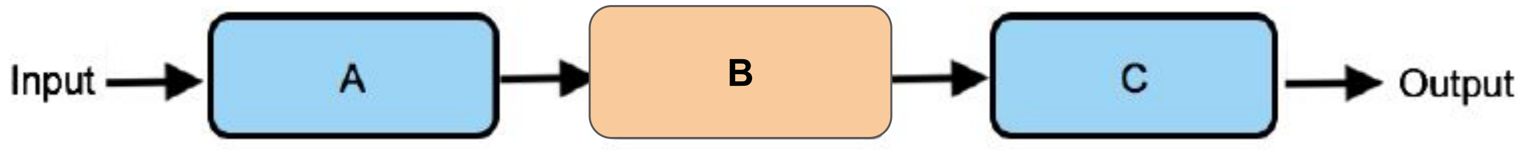


General case of error attribution: Example 1



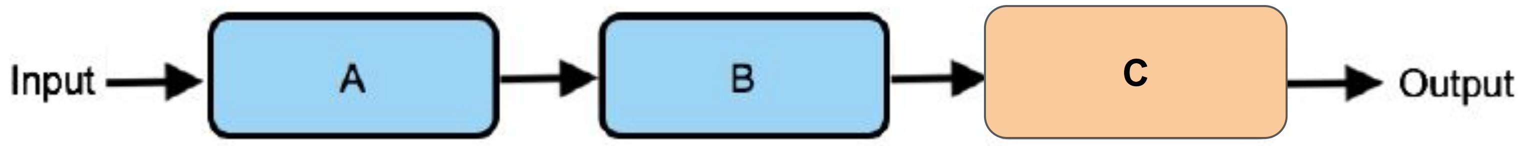
1. Try manually modifying A’s output to be a “perfect” output (e.g., the “perfect” bounding box for the cat), and run the rest of the pipeline B, C on this output. If the algorithm now gives a correct output, then this shows that, if only A had given a better output, the overall algorithm’s output would have been correct; thus, you can attribute this error to component A. Otherwise, go on to Step 2.

General case of error attribution: Example 1



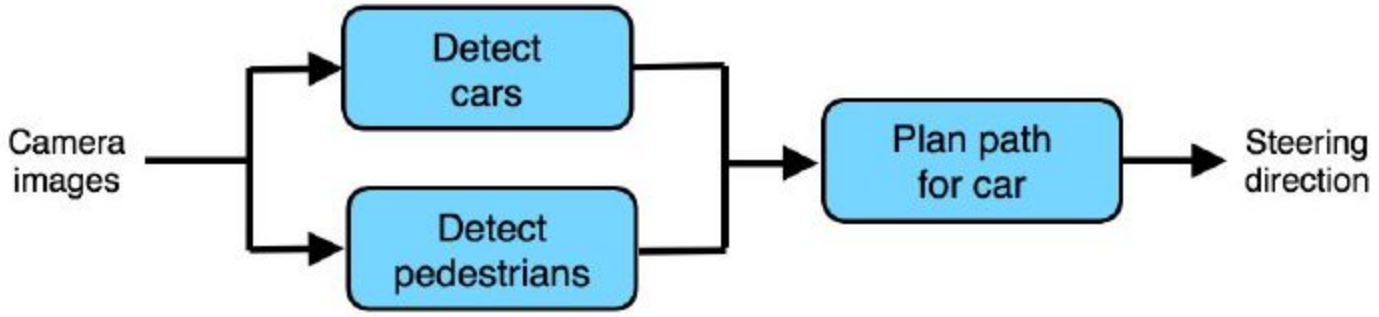
1. X
2. Try manually modifying B's output to be the "perfect" output for B. If the algorithm now gives a correct output, then attribute the error to component B. Otherwise, go on to Step 3

General case of error attribution: Example 1

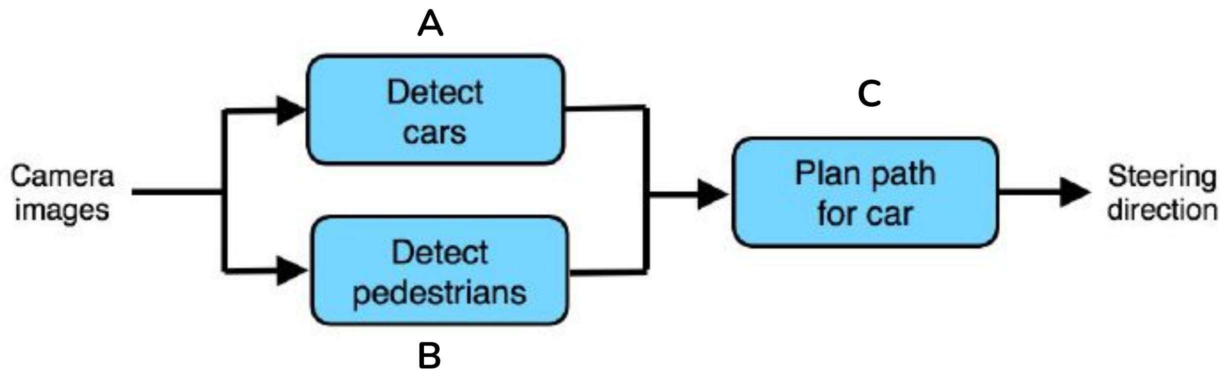


1. X
2. X
3. Attribute the error to component C.

General case of error attribution: Example 2

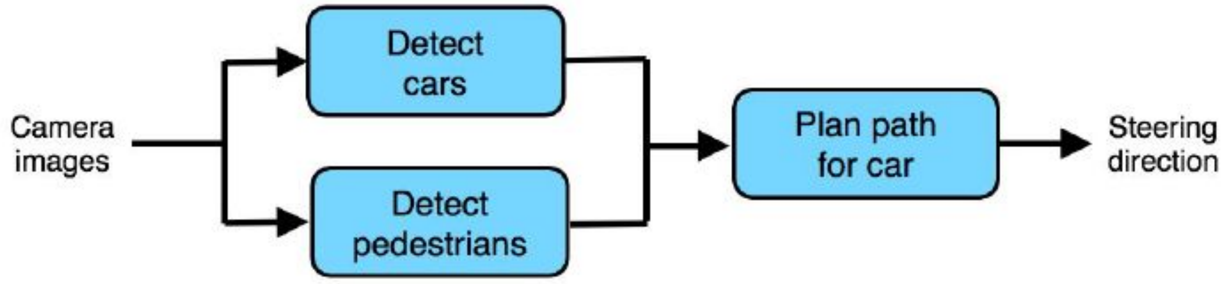


General case of error attribution: Example 2



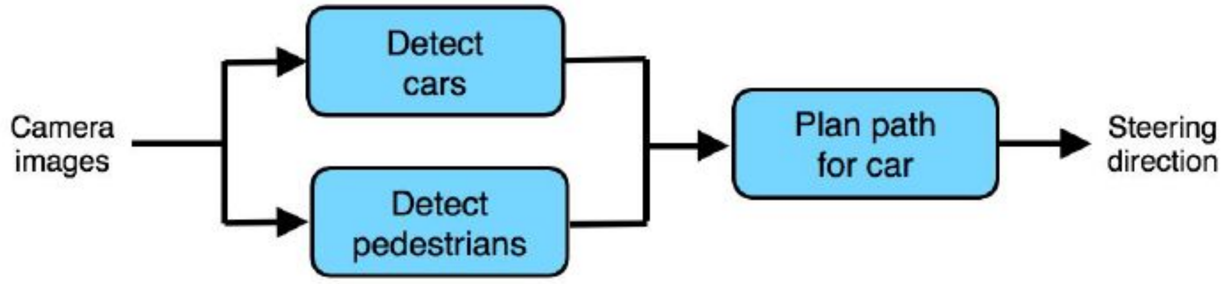
1. Try manually modifying A (detecting cars)’s output to be a “perfect” output (e.g., manually go in and tell it where the other cars are). Run the rest of the pipeline B, C as before, but allow C (plan path) to use A’s now perfect output. If the algorithm now plans a much better path for the car, then this shows that, if only A had given a better output, the overall algorithm’s output would have been better; Thus, you can attribute this error to component A. Otherwise, go on to Step 2.

General case of error attribution: Example 2



1. X
2. Try manually modifying B (detect pedestrian)'s output to be the "perfect" output for B. If the algorithm now gives a correct output, then attribute the error to component B. Otherwise, go on to Step 3.

General case of error attribution: Example 2



1. X
2. X
3. Attribute the error to component C.

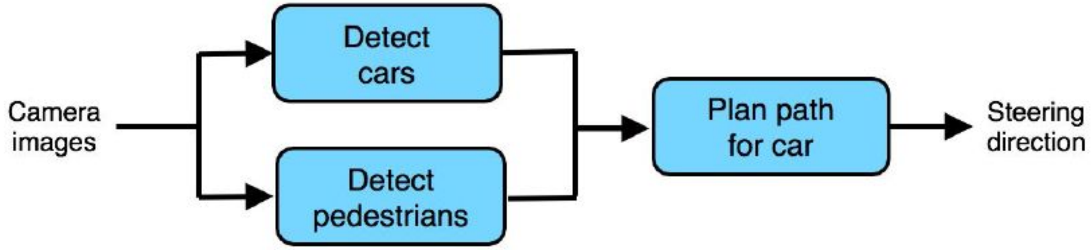
General case of error attribution

The components of an ML pipeline should be ordered according to a Directed Acyclic Graph (DAG), meaning that you should be able to compute them in some fixed left-to-right order, and later components should depend only on earlier components' outputs. So long as the mapping of the components to the A->B->C order follows the DAG ordering, then the error analysis will be fine.

Comparison to human-level performance

- Many error analysis processes work best when we are trying to automate something humans can do and can thus benchmark against human-level performance. Most of our preceding examples had this implicit assumption. If you are building an ML system where the final output or some of the intermediate components are doing things that even humans cannot do well, then some of these procedures will not apply.
- This is another advantage of working on problems that humans can solve--you have more powerful error analysis tools, and thus you can prioritize your team's work more efficiently.

Comparison to human-level performance



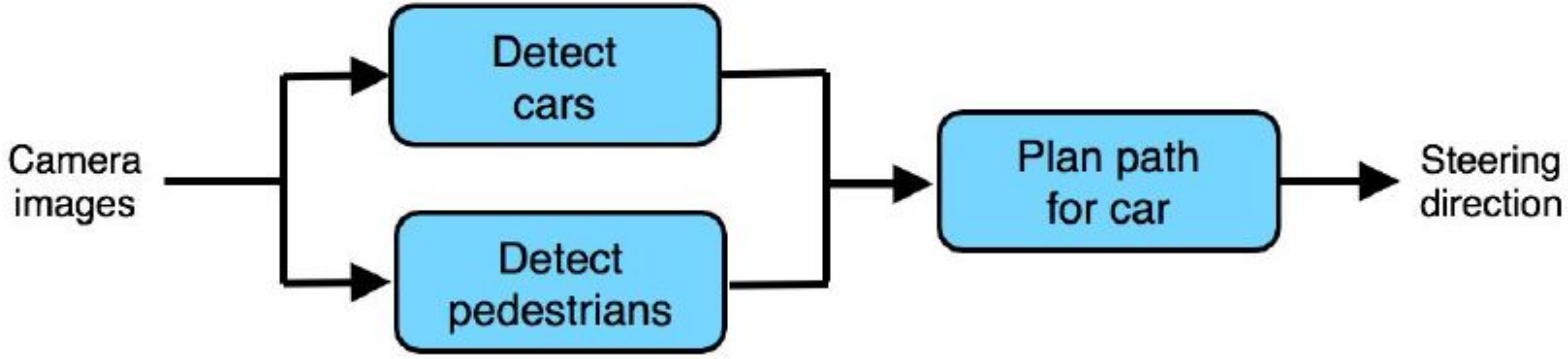
1. How far is the Detect cars component from human-level performance at detecting cars?
2. How far is the Detect pedestrians component from human-level performance?
3. How far is the overall system's performance from human-level performance?

Comparison to human-level performance

- If you find that one of the components is far from human-level performance, you now have a good case to focus on improving the performance of that component.
- There is no one “right” way to analyze a dataset, and there are many possible useful insights one could draw. Similarly, there is no one “right” way to carry out error analysis. So feel free to experiment with other ways of analyzing errors as well.

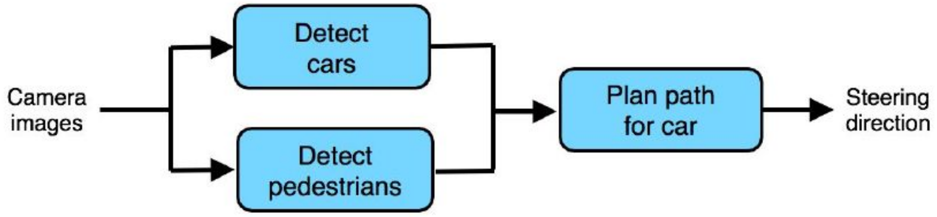
Spotting a flawed ML pipeline

What if each individual component of your ML pipeline is performing at human-level performance or near-human-level performance, but the overall pipeline falls far short of human-level?



Spotting a flawed ML pipeline

This usually means that the pipeline is flawed and needs to be redesigned. Error analysis can also help you understand if you need to redesign your pipeline.



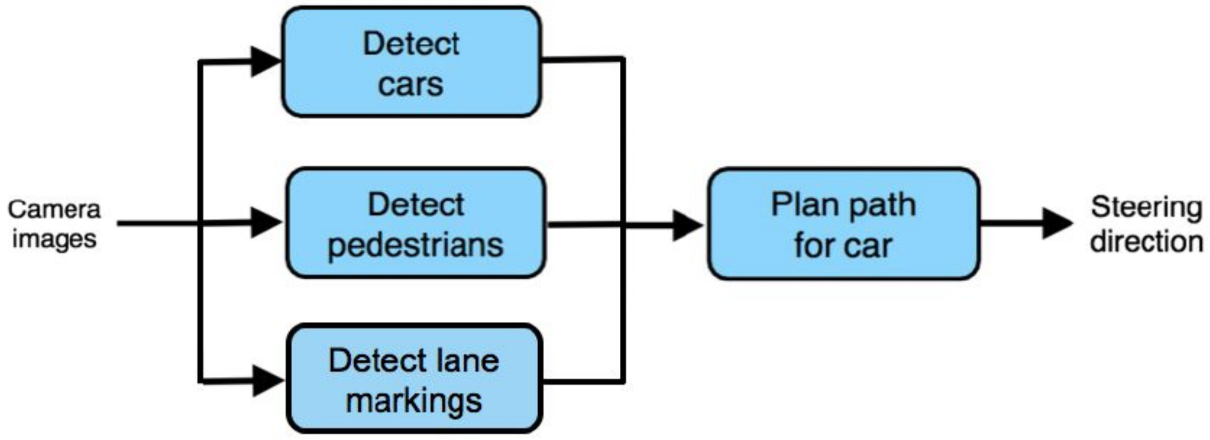
However, your overall self-driving car is performing significantly below human-level performance. I.e., humans given access to the camera images can plan significantly better paths for the car. What conclusion can you draw?

Spotting a flawed ML pipeline

The only possible conclusion is that the **ML pipeline is flawed**. In this case, the Plan path component is doing as well as it can given its inputs, but the inputs do not contain enough information. You should ask yourself what other information, other than the outputs from the two earlier pipeline components, is needed to plan paths very well for a car to drive. In other words, what other information does a skilled human driver need?

Spotting a flawed ML pipeline

For example, suppose you realize that a human driver also needs to know the location of the lane markings. This suggests that you should redesign the pipeline as follows:



Machine Learning Systems Design

Modeling Pipeline

Next Lecture: Model Development and Training (cont.)

CE 40959 Spring 2023
Ali Zarezade
SharifMLSD.github.io