Thinking back to the goals of this course, we're trying to build systems that

(1) learn knowledge and skills from language,

(2) learn to understand language, and

(3) learn to generate language.

How do we know how well our systems work? And what do we want to use them for? In the ∼2025 era, one answer is, well, just play around with some language models like *ChatGPT* and see if they follow your instructions! This is a really useful strategy for users. However, model developers are constantly making decisions as to whether one system is better than another, whether one method works better than another, etc. For this, we need replicability of experiments, we need thorough and careful evaluations.

Strong evaluations drive improvements in artificial intelligence. This has been true for decades, and is only becoming increasingly relevant.

On evaluations

What is a task?

At its most general, a task is a (possibly infinite) set of problems you want a system to solve. Problems in a task contain inputs, and a system provides outputs. A great example is **machine translation**. The task is to take in text a language, say, *Tamil*, and output the translation of that text into another language, say, *Mandarin*.

Often, a task is approximated with a (set of) dataset(s). In machine translation, for example, there is a popular challenge in which one is expected to translate newswire text. Notice how the specification of a domain (*news*) makes the task more specific, and probably easier (if I know I don't have to translate social media, or mathematics papers, my system can make more assumptions.)

So far, we've looked at language modeling — estimating a distribution $p(x)$ over text. Is this a task? Well, sure, but it's pretty underspecified. What distribution over text are we attempting to estimate? Data from the internet? Language modeling can be a task, but language modeling of web documents could be a task.

What is an evaluation?

I hope I've convinced you that you should hold the term "task" very loosely in your hands. Often what matters about a task is specifying expected behaviors for a system, and figuring out a way of evaluating if the outputs of the model meet those expectations. Here's some useful terminology:

Input distribution. What distribution over inputs do I have an expected behavior over my model for? Say a system takes any string over our vocabulary, so $x \in \mathcal{V}^*$. The system will never see most of these strings. So, we can say we care about some input distribution $p(x)$. This is sometimes referred to as the **domain** of the data, e.g., when we're discussing text, we might say "the *mathematics domain*" or "the *news domain*." When we say these things, we mean that a text is drawn (maybe implicitly) from some distribution related to math or news.

(Optional) Expected Output. For many tasks, we have an expected output. When I ask "2+2?" I expect the answer "4". When I ask "in what season of Avatar the Last Airbender did Zuko join the Gaang?" I expect the answer "3". For some tasks, I maybe

have little idea a priori of what the answer should be. "Write a proof in lean for the following lemma: ..."[1] — maybe I don't know that proof yet, and that's why I want the system to help me! Still, it is often helpful to think of there being a true distribution $p(y \mid x)$, a conditional distribution of outputs conditioned on inputs, that we would like our system to implement. Without thinking about it probabilistically, you could just think we're trying to replicate a function, $f(x) \mapsto y$.

Goodness measure. We might have an expected output y, and we might not, but either way, we want to be able to tell how good our model's output is.

Let g be a function that takes in an input x, a predicted output from our system $\hat{y}$, and optionally a "true/expected" output y, and outputs some score. If there's no expected y, consider it some null value $\varnothing$.

$$g(x, \hat{y}, y) \mapsto c \tag{1}$$

If we have an expected answer y, one simple goodness measure is "exact match," or just the identity:

$$g(x, \hat{y}, y) = \mathbf{1}\{y == \hat{y}\} \tag{2}$$

Don't worry too much about this notation; the right-hand-side just means "1 if y is equal to $\hat{y}$ else 0". For many tasks, this is good enough! But there is a huge space of methods for measuring a fuzzy, or approximate, similarity between outputs.

Train/dev/test splits

Examples of tasks and their expectations

Here are a few tasks and what we expect out of models that perform the tasks. The evaluation styles and considerations here are not unique to the tasks.

Machine Translation. The goal of machine translation is to take a source text in language S and give its translation in language T. So, x is a string like *Zuko made his uncle tea* and $\hat{y}$ is like *Zuko a préparé du thé pour son oncle.*

How do we measure the goodness of the translation? One family of methods is **human evaluation**: ask a human to score the quality of the translation according to some rubric. This could be as simple as a likert-style numerical rating:

$$g(x, \hat{y}) \mapsto c \in \{1, 2, 3, 4, 5\} \tag{3}$$

or a set of minor or major errors:

$$g(x, \hat{y}) \mapsto \{(\text{minor: There's a small error}\ldots), (\text{major: The verb is missing}\ldots)\} \tag{4}$$

It can be hard to evaluate the absolute quality of a translation; especially without a lot of training, what does it mean for the quality to be 4 versus 5? Instead, we might ask people to judge whether they prefer the output of one model over the output of another sometimes called **pairwise preference evaluation.** In our notation, you compare your model's output $\hat{y}$ and the other model's output y:

$$g(x, \hat{y}, y) = \mathbf{1}\{\hat{y} \text{ is preferred over } y\} \tag{5}$$

[1] Lean is a language for writing formal mathematical proofs that are programmatically verifiable.

Human evaluation makes sense because at the end of the day, we want our system to make translations that humans think are correct and fluent. However, human evaluation is expensive and time-consuming, and the development of systems, we also want faster signals of model quality, even if they're not as accurate in measurement as human evaluation. Preferably, these signals should be computable within seconds.

We've already discussed exact match—if we have a dataset of texts in English x, and human-made translations of those texts in Tamil y, we can just compare $y == \hat{y}$ and compute the fraction we got right. The problem here is that (1) if your model isn't that good yet, this provides no signal—you'll probably get 0 translations exactly right! Not so useful for model development. But also (2) even for a good model, there are potentially multiple relatively good translations for that sentence x, so even if your $\hat{y}$ is really good might not be exactly equal to y. We need a coarser metric.

BLEU score. A weird, messy, and exceptionally useful metric historically in the development of machine translation systems is the **BLEU** score [**?**]. Here are the intuitions of BLEU:[2]

- First, a simple metric of quality is precision. What fraction of the words in the prediction $\hat{y}$ are contained in the reference y?
- Second, that's not good enough, since a randomized-order sentence is not a good translation. So, what fraction of pairs of adjacent words in y does my $\hat{y}$ contain? How about 3-word spans? 4-word spans? Compute all such fractions up to 4-word spans.
- Third, uh oh, note that I've only computed precision metrics—whether the spans that show up in $\hat{y}$ also show up in y. If my $\hat{y}$ is just really short, we can make sure it includes just a few key spans and not actually cover everything it needs to translate. So, penalize $\hat{y}$ for being shorter than y (a "brevity penalty".)

The BLEU score formalizes the intuition that you get partial credit for your translation including increasingly long spans from y. And it's *fast* to compute.

There are a ton of details to real BLEU implementations—recalling our lecture on tokenization for example , what are we considering words again?—leading to, e.g., standardized implementations [**?**].

Code generation and test cases. Code generation is a broad term for tasks that involve taking a specification of some function or codebase (often in natural language) and generating code (in, e.g., python) which implements the desired functionality. A nice aspect of code generation as a problem is that we can write **test cases**, much like we do for ourselves when designing and writing code, to see how well the generated code behaves. So, I might have a dataset of $(x_i, (t_{i,1}, \ldots, t_{i,k}))\}_{i=1}^{M}$ of task specifications x, and test cases $(t_1, \ldots, t_k)$. I could then compute average test case pass rate:

$$\frac{1}{M}\sum_{i=1}^{M}\frac{1}{k}\sum_{j=1}^{k} t_{i,j}(\hat{y}_i), \tag{6}$$

where for convenience we're saying $t_{i,j}(\hat{y}_i) = 1$ if the predicted code $\hat{y}_i$ passes test case $t_{i,j}$, and 0 otherwise. You might be right to complain that passing some—but not all—

[2] There are many details of BLEU that we won't go over here, including, e.g., (1) the brevity penalty is computed at the corpus level, (2) the precision metrics cap the number of times a word in the prediction (e.g., *the* can count as occurring in the reference, specifically at the number of times that word actually occurred in the reference).

test cases for a given problem means the code doesn't work. So, we might instead report the fraction of problems for which the code passes *all* test cases:

$$\frac{1}{M}\sum_{i=1}^{M}\mathbf{1}\{\{\forall_{j=1}^{k} \quad t_{i,j}=1\}\} \tag{7}$$

Evaluating annotations: Named entity recognition, coreference, parsing. Often we have annotated structures on top of text that are useful for the categorization of knowledge. Intuitively, these structures are approximations to how we reason about the world. One example is *named entities*. Consider the following sentence with the annotations provided:

[Uncle Iroh]$_{\text{PERSON}}$ and [Zuko]$_{\text{PERSON}}$ must stay incognito in [Ba Sing Se]$_{\text{LOCATION}}$.

Intuitively, named entities refer to specific people, places, or things that we may want to understand and aggregate information about. In this setting, our true output is a list of span indices in the tokens (since, e.g., the same entity may show up multiple times and we want to recognize it each time):

$$\begin{aligned} y = \{ & \\ & \text{span } (0,2) \textit{ Uncle Iroh}, \\ & \text{span } (4,5) \textit{ Zuko}, \\ & \text{span } (9,12) \textit{ Ba Sing Se}, \\ & \} \end{aligned}$$

Our predicted output $\hat{y}$ is likewise a list of spans. What metrics might we report? One is **recall**, which is the fraction of spans in y that are also in $\hat{y}$:

$$\text{recall}(y,\hat{y}) = \frac{1}{|y|}\sum_{s\in y}\mathbf{1}\{s\in\hat{y}\} \tag{8}$$

Note that this gives no partial credit to *almost* getting a span; for example, just returning *Iroh* would give no credit. The issue with reporting just recall is that we can set $\hat{y}$ to be the list of all possible spans $\hat{y} = \{(0,1),(0,2),\ldots,(1,2),\ldots,\}$, and guarantee perfect recall. Not great! We can also report **precision**, which is the fraction of spans in $\hat{y}$ that are also in y:

$$\text{precision}(y,\hat{y}) = \frac{1}{|\hat{y}|}\sum_{s\in\hat{y}}\mathbf{1}\{s\in y\} \tag{9}$$

The potential issue with precision is that we can just set $\hat{y}$ to the span(s) we're most confident in, thus achieving high precision but missing almost all of the true named entities. A common metric for balancing the concerns of recall and precision is the **F1 score**, which is the harmonic mean of precision and recall:

$$\text{F1}(y,\hat{y}) = 2\cdot\frac{\text{precision}(y,\hat{y})*\text{recall}(y,\hat{y})}{\text{precision}(y,\hat{y})+\text{recall}(y,\hat{y})} \tag{10}$$

One might ask, why not just average precision and recall? You could do this as well. The harmonic mean severely punishes you for doing really badly on one of the two constituent metrics. For example, if I have 0.05 precision and 0.95 recall, my arithmetic average is 0.5, but my F1 is 0.095. If precision and recall are the same, then the arithmetic and harmonic means are equal.

There are many other types of annotations we might want to eval. One is **coreference**, the notion of which spans in a text refer to the same entities in a narrative:

[Uncle Iroh]$_1$ went inside the [tea shop]$_2$. Then [he]$_1$ closed [it]$_2$.

Here, *he* and *it* refer back to entities mentioned already in the narrative. Knowing which spans of text refer to the same entity is important, especially when someone is mentioned by name once, and then they or their actions are referred to many times by pronouns or other text later. This notion of understanding *who did what to whom* is often fundamental to the annotations developed for language. We won't go deeply into these in this lecture, but it is helpful to think through how we'd evaluate complex structures.

LLM-As-a-Judge

As LLMs develop increasingly capable instruction following abilities and broad skills across domains, they're increasingly used to provide judgments in lieu of simple metrics or human judgments. The benefits here are in the generality and simplicity of implementing the method. The drawbacks are in the uncertainty of what you're measuring—you don't really know what the model

Rubric-based judgments

Often, we'll task a model with judging across a range of rubric items, in order to better guide its judgments.

Iteration, cost

How do you know if your LLM-as-judge pipeline works? The first thing to do is look at your data. Look at the judgments. You can often tell quite quickly if the judgment function is definitely wrong—it takes longer to tell if it's subtlely wrong. But really you should have a held-out dataset of

You have a few axes to play with when it comes to LLM-as-a-judge pipelines; the two most meaningful are model size and model thinking budgets. We'll talk about this more later, but model size determines how much work goes into computing one token; thinking budget roughly determines the amount of deliberation the model goes through before maling an answer.

Goodhart, misspecification, and the perils of overoptimization

Engineering is often about defining the right metrics, figuring out how to optimize them, and then doing so safely. In machine learning, through gradient descent and various tricks for optimizing non-differentiable functions, we have the capacity to optimize for truly grandiose goals.

However, it remains hard to actually specify what we want. Indeed, the better we get at optimization, the less it's okay to have metrics that only roughly correlate with what we want. Let's go through two case studies.

Optimizing for BLEU. Consider a system that is left to its own devices to maximize BLEU without concern for anything else. What might it learn? How might that differ from what we want out of a translation system?

Optimizing for working code in a sandbox. Consider a system that is left to its own devices to maximize passing test cases without concern for anything else. What

might it learn? How might that differ from what we want out of a software engineering tool?