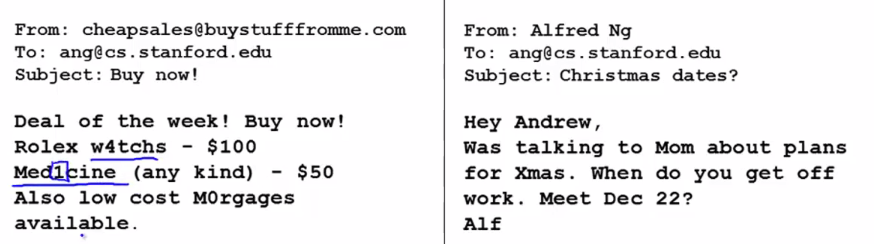
***Building a Spam Classifier***

**I. PRIORITIZING WHAT TO WORK ON**

* Now to talk about ML system design + how to strategize putting one together
* Starting w/ the issue of prioritizing how to spend time on what to work on w/ an example of spam classification.



* Say we have a labeled training set of some of spam emails + some non-spam emails w/ labels y = 1 or 0
* In order to apply classified supervised learning, the 1st decision we must make is *how do we want to represent X* (features of the email)
* Given features X + labels Y in our training set, we can then train a classifier using logistic regression.
* Here's one way to choose a set of features for our emails.
* Come up w/ a list of ~100 words we think are indicative of whether email is spam or non-spam,
* Ex: contains the words 'deal', 'buy' or 'discount' = more likely to be spam
* Ex: contains my name, which may mean the person actually knows who I am, or contains “now” b/c I get a lot of urgent emails, so there are indicative of non-spam
* Given a piece of email, we can then take this it + encode it into a feature vector:
* Take the list of 100 words + check + see whether or not each appears in the email
* Then define a feature vector X with 1/0 for if a word appears in the email
* X(j) = 1 if word j appears in the email
* This gives a feature representation of a piece of email.
* In practice, what's most commonly done is to look through a training set + depict the most frequently occurring n words (n is usually between 10k-50k) + use those as the features, rather than manually picking words
* Now, if building a spam classifier, 1 question you may face *is what the best use of time is in order to make your spam classifier have higher accuracy/low error.*
* 1 natural inclination is going to collect lots of data
* There's this tendency to think that the more data we have the better the algorithm will do.
* In fact, in the email spam domain, there are actually pretty serious projects called **Honeypot Projects** which create fake email addresses + try to get them into the hands of spammers + use that to try to collect tons of spam email to train learning algorithms.
* But we've already seen that getting lots of data will often help, but not all the time.
* But for *most* ML problems, there are a lot of other things you could do to improve performance.
* For spam, 1 thing you might think of is to develop more sophisticated features on the email, maybe based on the email routing info (info contained in the header).
* Very often spammers will try to obscure the origins of the email w/ fake email headers or send email through very unusual sets of CPU service + unusual routes in order to get the spam to you
* Some of this info will be reflected in the header, so one can look at email headers + try to develop more sophisticated features to capture email routing info to ID if something is spam.
* Something else to do is look at the email message body + try to develop more sophisticated features.
* For example, should 'discount' and 'discounts' be treated as the same word? Or should 'deal' and 'dealer' be the same word, even though one is lower case + one in capitalized
* Or do we want more complex features about punctuation b/c maybe spam uses !’s a lot more.
* Could also develop more sophisticated algorithms to detect + maybe correct deliberate misspellings, like m0rtgage, med1cine, w4tches.
* Spammers actually do this, b/c w/“w4tches, a simple spam classification technique might not equate this as the same thing as the word "watches” + it may have a harder time realizing that something is spam w/ these deliberate misspellings.
* While working on a ML problem, very often you can brainstorm lists of different things to try
* What happens far too often is that a research or product group will *randomly* fixate on 1 of these options + sometimes that method turns out not to be the most fruitful way to spend time
* But if you even get to the stage of brainstorming a list of different options to try, you're probably already ahead of the curve.
* Sadly, what most people do is instead of listing out options of things is use “gut feeling,” like "let's just have a huge honeypot project + collect tons more data"
* We can have a more systematic way to choose amongst the options of different things that might work, + therefore be more likely to select a good way to spend time

**II. ERROR ANALYSIS**

* If starting work on a ML problem or building a ML application, it's often considered very good practice to start by building a very *simple* algorithm (quick + dirty) you can *implement quickly + test on CV data*
* Once you've done than, you can plot learning curves of the training + test errors to try to figure out if the learning algorithm is suffering from high bias or high variance, or something else
* Then use that to try to decide if having more data, more features, etc. are likely to help.
* When just starting out on a learning problem, there's really no way to tell in advance whether you need more complex features, or more data, or something else.
* It's just very hard to tell in advance + in the absence of evidence/seeing a learning curve where you should be spending time.
* Often implementing a very quick + dirty implementation + plotting learning curves helps us make these decisions.
* Think of this as a way of avoiding **premature optimization** in CPU programming
* The idea is to instead let evidence guide decisions on where to spend time rather than gut feeling, which is often wrong.
* In addition to plotting learning curves, 1 other thing that's often very useful to do is **error analysis**.
* When building say a spam classifier, I will often look at my CV set + manually look at emails the algorithm is making errors on to see if its misclassifying and/or if there’s any systematic patterns in what type of examples it is misclassifying.
* Often, by doing that, this will inspire you to design *new* features, or will tell you what the current shortcomings of the system are + give you inspiration for improvements to make to it.
* Let's say you've built a spam classifier w/ 500 examples in your CV set, + the algorithm has a very high error rate by misclassifying 100 CV examples.
* Manually examine these 100 errors + manually categorize them based on things like what type of email it is, what cues (features) you think might cause the algorithm misclassify them
* Maybe find that the most common types of spam are emails on from ‘pharmacies’ trying to sell drugs, or to sell fake replicas or other random things, or phishing emails, or other categories.
* Maybe I find 12 are pharma emails, 4 are selling fake replicas, 53 are phishing, + 31 are other types of emails.
* By counting up the number of emails in these different categories, we find the algorithm is doing particularly poorly on filtering out emails trying to steal passwords
* This may suggest that it might be worth effort to look more carefully at that type of email + see if we can come up e/ better features to categorize them correctly.
* Might also do is look at what cues/additional features might help the algorithm classify emails.
* Look back at the hypotheses about features that might help us classify emails better: detect deliberate misspellings (find 5 cases), unusual email routing (16 cases), unusual spamming punctuation (32 cases)
* This tells us that maybe deliberate spellings is a rare phenomenon that’s not worth the time to write algorithms that detect it.
* But we find a lot of spammers using unusual punctuation, then that's a sign it might be worth your while to spend the time to develop more sophisticated features based on punctuation.
* This sort of **error analysi**s *= the process of manually examining mistakes an algorithm makes*, can often help guide you to the most fruitful avenues to pursue.
* What we really want to do w/ a quick + dirty implementation of an algorithm is figure *out what are the most difficult examples for an algorithm to classify*.
* Very often for different learning algorithms, they'll often find similar categories of examples difficult.
* By having a quick and dirty implementation, we have a quick way to ID some examples of hard errors to can focus your efforts on.
* 1 other useful tip for developing learning algorithms is to make sure we have a numerical evaluation of the learning algorithm.
* Often incredibly helpful to get back a single real number (accuracy, or maybe error) that tells you how well the learning algorithm is doing.
* Let's say we're trying to decide whether or not we should treat words like discount, discounts, discounted, discounting as the same word.
* 1 way to do that is to just look at the 1st few characters in the word like, + decide that maybe all these words have roughly similar meanings.
* In NLP, the way this is done is via a type of software called **stemming software** (Porter stemmer)
* Using a stemming software can help but also hurt (mistake “universe” + “university” as being the same thing b/c they start off w/ the same letters).
* So, if trying to decide whether or not to use stemming for a spam classifier, it's not always easy to tell if it’s useful, + error analysis may not actually be helpful for deciding if stemming is a good idea.
* Instead, the best way to figure out if using stemming is good to help your classifier is if you have a way to very quickly just try it + see if it works.
* In order to do this, having a way to numerically evaluate an algorithm is going to be very helpful.
* Concretely, maybe the most natural thing to do is to look at the CV error of the algorithm's performance *with* and *without* stemming.
* So, if you run the algorithm *without* stemming + end up w/ 5% classification error + get 3% *with* stemming, this decrease in error very quickly allows you to decide that stemming is a good idea.
* For this particular problem, there's a very natural, single, real number evaluation metric (CV error)
* Usually, coming up with a single, real number evaluation metric will need a little bit more work.
* 1 more quick example: Trying to decide whether or not to distinguish between upper vs lower case.
* Should that be treated as the same word/feature or as different words/features? Should this be treated as the same feature, or as different features?
* B/c we have a way to evaluate our algorithm (quick + dirty on CV set), then if we don’t distinguish between upper + lower case, maybe we end up w/ 3.2 % error, which is worse than if I use only stemming.
* So when you're developing a learning algorithm, very often you'll be trying out lots of ideas + lots of new versions of the learning algorithm.
* If every time you try out a *new* idea + manually examine a bunch of examples to see if it got better or worse, that's going to make it really hard to make decisions.
* Do you use stemming or not? Do you distinguish upper and lower case or not?
* But by having a single real number evaluation metric, you can just look + see “did the % go up or down?”
* You can use that to much more rapidly try out new ideas + tell if your new idea has improved or worsened the performance of the learning algorithm almost right away
* This will often let you make much faster progress.
* So the strongly recommended way to do error analysis is on the CV sets rather than the test set
* People will do this on the test set, even though it’s definitely a less mathematic appropriate + less recommended way to do error analysis
* So, when starting on a new ML problem, implement a quick + dirty
* Very rare to spend too *little* time on a quick + dirty implementation, but common to spend much too *much* time building it
* Don't worry about it being too quick or too dirty, implement something as quickly as you can.
* Once you have the initial implementation, you have a powerful tool for deciding where to spend time next.
* 1st look at the errors + do error analysis to see what other mistakes it makes + use that to inspire further development.
* 2nd, assuming the quick + dirty implementation incorporated a single real number evaluation metric, use that metric as a vehicle to try out different ideas + quickly see if they are improving performance
* This lets us make decisions about what things to incorporate into the learning algorithm more quickly
* 
* 