On confidence determination

Problem

Let's define confidence as the chance of a prediction coming true, such that, given an infinite number of exact predictions all with confidence `c` we would expect their overall accuracy to be equal to `c`.

With that out of the way, does confidence have any role to play in machine learning ?

Are confidence determination models useful ? If so, how and why ?

Is confidence a useful tool in our ontology at all or should we eat it up inside some other more useful concept ?

The role of confidence

I think confidence as defined above can be though of as playing several roles.

1. A factor by which we weight our predictions*(without modifying the model)*.

Suppose we have a unfiorm target with 3 values: A, B, C. In the case of B and C the "cost" of a false positive is equal to the cost of a true positive, however the "cost" of classifying something as A incorrectly is so great that it's equal to the benefit of x10 correct classifications of A. There's two ways to go about solving this issue:

- Negatively weighting A during training. For example using a criterion with 10+epsilon greater penalty for miss-classifying something as A then for classifying A as something else.
- Obtaining a confidence value for our prediction, only trust predictions A above a certain confidence.

Once we have a confidence (c), we can rephrase each prediction of A as: `A = Pred_A | c > c_lim , null | c <= c_lim` , where `c_lim=0.9`. Thus for every A we predict we can be assured that the cost function we are maximizing for is > 1.

Even if the confidence doesn't translate into real world probabilities (due to training data scarcity, imperfect models and overfitting), we can still obtain a sample (`C`) of all confidences on the validation dataset and define `c_lim` such that `P(True|Pred_A) > 0.9 | (A | C > c_lim )` .

Even if our validation dataset is too small to trust this approach we can defensively set `c_lim` higher than the value determine on the validation set.

The confidence based approach seems superior since:

- It doesn't bias the model based on some external cost function that might later change. Given a change in the TP/FP cost balance for A we can reflect this in our predictions by just tweaking `c_lim`.
- It allow to quick tweaking if production data doesn't match behavior on validation data. Given that our decided upon `c_lim` yield accuracy < 0.9 on the production data we can try and increase is until we get a desired accuracy without retraining the whole model.
- It allows us to predict unknowns. If we bias the loss function we are still misclassifying A as B or C in order to avoid the large cost of a FP for A. In the confidence case, if our confidence is too small we can instead just say "I think this is likely and A but I'm not confident enough for it to be worth treating as such".

2. Useful in increasing accuracy on tasks where we can make a variable number of predictions

So, quick and dirty example here is obviously stock market. We have a model that predicts `Y` as the change in a stock's price in the next 15 minutes, for any given stock every 15 minutes. We might be making ~8000 predictions with our model, but we only need 2-3 correct predictions coupled with no false predictions to achieve our goal (get silly rich). In this hypothetical we can take only e.g. `Y | c > 0.8` . Thus turning even a bad model into a potentially great model assuming that said model is capable of making a few good predictions and our confidence determination is on point.

Conversely, not many people can do this, so based on this particular example I think it's fair to speculate something like: "The kind of data that yield very imperfect models are likely to also yield very imperfect confidence determination algorithms and/or have no edge cases where the confidence can be rightfully determined as very high". This is a speculation, not a mathematical or otherwise rigorous stipulation.

To formalize this a bit more we can define the following "cost function" for deploying our algorithm into production:

```
cost = prediction_error*inverse_log_func(pct_predictions_made)
```

By `inverse_log_func` I mean any scaling factor that will reduce cost if more predictions are made at less than linear rate. To get back to the stock model, assume that we are predicting in 3 situations:

1. We predict the price change for all stocks and have a `prediction_error=0.5`
2. We predict the price change for 0.4 of all stocks and have a `prediction_error=0.2`
3. We predict the price change for 0.1 of all stocks and have a `prediction_error=0.05`

Let's compute the cost using `inverse_log_func=1/ln(1+100n)` : 1. `cost(1) ~= 0.11` 2. `cost(2) ~= 0.05` 3. `cost(3) ~= 0.02`

In a case like this a confidence determination mechanism and increase the overall performance of a model without needing to improve the model itself, potentially even in a case where we can find "no way" of improving the model.

This is more or less a generalization of case 1, but I think it's useful to keep both of them in mind since the first one i Looking at this generic example this seems fairly promising, since we can basically say "a confidence determination that's better than the overall accuracy score can improve a model past the point where it's overall usefulness scales better with increased accuracy than with the number of predictions it can make".s easier to conceptualize and builds up to this one.

3.Useful as a model training or choosing mechanism

Supposed we have several models that can make predictions as an ensemble and on their one they have about equal accuracy, or suppose that we want to pick between several models that all obtain around the same accuracy when cross validated on the relevant data.

In the first case, we basically have to weight each model in the ensemble by 1/n, in the second case, we have to pick a model at random.

But suppose that we instead have a separate confidence model trained alongside each model.

In the ensemble case, training this confidence model along side our normal models can yield a confidence that a given model in the ensemble is making a correct prediction. Assume a situation with `n` labels and `n` model for which `Mx` is 0.9 good at predicting `x` when it appears and and `1/n` accurate at predicting any other label, all other model are `1/n` good at predicting all other labels.

Normally, this model would be close to random, however, provided that our confidence is 100% correct our formula for picking a prediction for when `Mx` predicts `x` becomes:

```
[0 0 ... 1 ... 0] * 0.9 + 1/n * other_pred_vectors
[1 2 ... x ... n]
```

Which basically means that for `1 < n < 11` we are guaranteed to get 90% accuracy for predicting `x` and the accuracy stays better than random for bigger `n`s (though surprisingly enough the computations for this case aren't very easy)

Obviously this is a made up case, assuming ensemble models predict randomly defeats the point of an ensemble, but going from `P(correct|X)=1/n` to `P(correct|X)=0.9` is a fairly huge leap by just adding a confidence value. I'm fairly sure this still holds if `P((correct|x)|Pred(Mx)=x)=0.9 for any Mx for x in (1..n)`, but the proof is a bit harder here. However, in this case, the behavior would be closer to what we would expect from a "real" ensemble model.

The second case is a bit trickier, but we can assume a sort of "overfitting" when we pick the model out of a multitude. Assume we have a training methodology that draw out a "best candidate" on some validation set that we are constantly evaluation models on during training. Do this enough time and you end up overfitting the validation set.

However, assume that instead of evaluating the accuracy on the validation set we also evaluate the confidence. Given that confidence and accuracy are independently determined, the chance of both overfitting a validation set at the same time is `1/n` where `n` is the number of checks we run against out validation set.

Thus, if instead of our model picking methodology being "best model on the validation set" it becomes "best model with top 80th percentile accuracy and 80th percentile confidence on the validation set". At least on an intuitive level, this seems like it could prevent overfitting on the validation data.

The roles of confidence are not confidence-specific

In other word, a confidence value can help us:

1. Select a subset of predictions with higher accuracies under various scenarios
2. Modify the behavior of our predictive models both during training and during inference (see point 3)

But a lot of things to do (2) and (1) seems like something that could be inherent in our very models and thus needn't require a complex confidence determination mechanism. Some examples I use in practice are: determining categorical certainty by looking at the non-max values in the one hot output vector, using a quantile loss to determine a confidence range instead of an exact value for numerical predictions, predicting linear instead of binary values in order to determine likelihood of the outcome predicted.

Confidence and explainability

I'd argue that a confidence determination mechanism can be interesting based on the inputs we feed into it, fundamentally there's 3 different things that can determine confidence.

First let's define a few terms, we have some inputs (X), some output which for the sake of argument we can just treat as a label for each input sample (Y) and a machine learning model trying to infer a relationship between the two (M). The predictions of the machine learning model will be denoted `Yh`

As such, after a prediction is made, confidence can be determine based on 3 variables `X`, `Yh` and `M`.

For example, we can take just the inputs into our model and say "Oh, the SNR here looks horrible based on <simple heuristic that's hard to built in into our model>, let's assign this a low confidence".

We can also look at the output and say, "Oh, the model is usually accurate, but currently it's predicted that only appears 2 times in the training data and the model was wrong every time it predicted it, so let's assign a low confidence".

We can also look at the model itself and say, "Oh, the model's activations usually match these clusters of patterns, but the current activations look really weird, this is behavior very different from what we've previously seen, let's assign a low confidence".

Granted, both Y and M stem from X, but independently analyzing them might lead to results which are easier to act upon. I.e. "This one pixel on the dog image looks kinda weird" is less useful to say than "The n-th layer of your model has parts x,y,z which activate too strongly due to this random pixel". Both statements are hard to act upon, but at least there's some chance of being able to make meaningful change based on the second (e.g. use some kind of regularization to normalize whatever is happening in the n-th layer).

This is all fine and dandy except that, well, there's no way to know which of the two things are easier to 'act upon' in a given scenario, even worst, outside of contrived examples, `X` is usually by far the easiest variable to act upon.

Thus, even though confidence *might* play a role in exaplinability, I think this only comes up in rather contrived scenario when we imagine very complex confidence models that are able to converge spectacularly well. However (see above), I'd tend to think that in most cases, if a confidence model can converge well on a problem, so can a predictive model and thus the confidence/explainability component lose a lot of their usefulness.

Confidence and training

A more interesting role in confidence determining models would be more them to serve as secondary loss functions to our models.

Does this seem silly, unintuitive or counter-productive ? Well, consider this:

Given `M` and a confidence determining model `C` that takes `Yh` and produces a confidence `c`, we'll train `M` by using some function that incorporates both `M`'s own loss function and the gradients from the input layer of `C` (in which we input `Yh`).

`C` is trained to generate a confidence value, one easy way to do this is to have `C` try to predict a normalized value of the loss of `Yh` based on knowing `X` and `Yh` but not `Y`.

Sounds a bit strange, but let me propose a different scenario:

What if `C` were to just be discriminating between `Yh` being correct or not, just a 0 and 1 loss, maybe being feed some `Y` values instead at the beginning as to not make it's job too easy.

Then, we could backpropagate the cost of this binary decision through `C` and into the output layer of `M`.

What I'm describing is basically just the training methodology for a GAN. Those seem to work stellarly well, so why wouldn't this ?

In addition to that, unlike GANs, we have the option of feeding the activation of any of the intermediary layers of `M`, as well as the original input `X`, into `C`. Granted, there's less evidence that this should work, but at least intuitively it seems like it could be an extra useful datapoint.

Well, I can think of several reasons for that, but I'm fairly sure most of them I could also apply to GANs. However, the advantage we have over GANs is that `C` is not working *against* `M` in this case, it's just trying to predict it's behavior. Minimizing the loss for `C` (having it be better able to predict how far `Yh` will be from `Y`) will not necessarily negatively affect `M`'s performance.

With GANs there is a risk of `G` being actually good, but just facing off against a very well trained `D` that pays close attention to minute features that can't be perfectly replicated by `G` given such a large SNR in the loss function. However, with this approach, there's no incentive or way for `C` behave in ways that increase the loss of `M`.

On the whole, this is rather promising contextual evidence and good directions to start digging further into this topic with some experimenting.

Experiment

In order to ruminate on the idea of confidence determination I will do the following experiment:

Take a model (M) and train it on 4 datasets:

The first dataset (a) is a fully deterministic dataset dictated by some n-th degree polynomial function:

`f(X) = c1 \* X[1]^1 + ... cn * X[n]^n`

Just to keep it simple let's say `ck = k or ck = (n+1-k)`

We might also play with variation of this function where we remove the power or the coefficient (e.g. `f(X) = 1*X[1]...n*X[n]` and `f(X) = X[1]^1 + ... X[n]^n`). The choice is not that important, what we care about is getting a model that converges **almost perfectly** in a somewhat short amount of time.

We might also go with an even simpler "classification" version, where `f(X) = x % 3 if x < 10 else None for x in X`, thus getting an input of only 0, 1 and 2. Whichever of these 3 labels is more common, that's the value of `Y`.

The value to be predicted, `Y = 0 | f(x) < lim1, 1 | lim1 < f(x) < lim2, 2 | lim2 < f(x)`

lim1 and lim2 will be picked such that, for randomly generate values in X between 1 and 9, the split between the potential values of Y in the dataset is 33/33/33.

M will be picked such that it can converge on a solution to this equation with some accuracy that's close to perfect. Let's say M is a fully connected network with 2 hidden layers of sizes: `[SUM(k for k from 1 to n+1), n+1]`, the need for the added slack in the architecture will become apparent in the next 2 datasets.

The second dataset (b) is similar to (a) except for the value of Y, which is:

`Y = 0 | f(x) < lim1, 1 | lim1 < f(x)`

But, given some irregularity in the inputs, say `X[0] == 5 => Y = 2`

lim1 will be picked such that the initial split is 50/50, given that we are using the 1st degree variable to introduce this irregularity in 1/9th of samples, I assume the final split will be around 11/44.5/44.5.

The third dataset (c) is similar to (a) except that the when Y == 2 there's a 1/3 chance that Y will randomly become 0.

These 3 dataset are symbolic, though obviously not fully representative of 3 "cases" in which I'd like to think about the idea of confidence:

a) Represents a case where the datasets follows a straight forward equation which, in theory, the model should be able to approximate with ~100% accuracy.

b) Represents a case where the datasets follows a deterministic equation, but this equation has an irregularity which breaks it's linearity in several places, which seem "random" when looking at the final value but obvious when looking at the input independently.

c) Represents a case where noise is added to the data, such that 100% accuracy can't be reached. However, assuming a perfect model, we know the confidence we should have in our predictions: `P(correct|1) = 1, P(correct|0) = 75, P(correct|2) = 66`.

I want to test some solutions here and evaluate them based on six criteria:

1. Accuracy, overall accuracy achieved on a testing datasets generated by the same process as the original ones.
2. Given that I only pick predictions with high confidences (say 0.8 quantile), how high is the accuracy for this subset of predictios ?
3. Given a random subset of predictions + confidences, is the mean of the confidence values equal to the accuracy for that subset of predictions ?
4. same as (4) but on the whole dataset rather than a random subset.
5. Assuming a situation where a wrong predictions is `-1` and a correct prediction `1`, and the accuracy score is the sum of all these `-1` and `1` values. Would weighting this value by their associated confidence increase the value of this sum ?

Some of these may not seems that intuitive/interesting, but are useful to evaluate during the experiment in order to make small tweaks and find errors with the

models (e.g. criteria 1,2,5 and 6)

The solutions will be:

- Just letting `M` predict, confidence will be == overall acc % on the training set.
- Let `M` predict and have a separate model `C` that predicts the confidence (on a scale from 0 to 1) based on the inputs into `M` and the outputs `M` produces.
- Let a different model `M'` predict, this is a model just like `M` *but* includes an additional cell into the outputs that represents a confidence value (on a scale from 0 to 1) and include additional cells evenly distributed in all 3 layers in order to have an `M'` training in roughly equal time to that of `M` + `C`.
- Let `M` predict and have a separate model `C` that predicts the confidence just like before, however, the loss from the `Yh` component of the last layer of `C` is backpropagated through `M`.