AI/ML for Software Engineers - Prediction & Neural networks

Codesmith

Will Sentance

CEO & Founder at Codesmith

Frontend Masters

Academic studies

Harvard University

Oxford University

ML/AI & Neural Networks

Prediction in software engineering changes what we can build with code.

In this workshop we'll build out full mental models of prediction and model development so you can be part of it

From predicting fraud to predicting image content to predicting responses to questions

Enables emergent phenomena and maybe even intelligence if we think that data distributions are 'all there is'

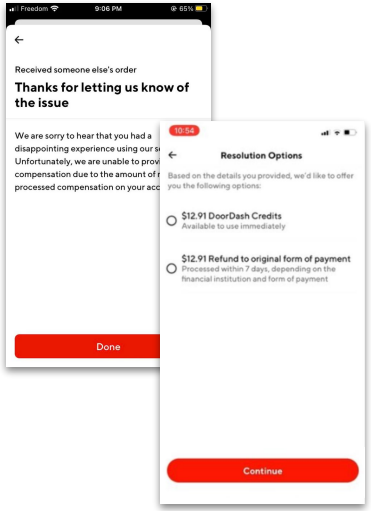
At the core we develop rules to match patterns in known data (a sample) & generalize to a population with unknown data

Depends on enormous 'compute' power, deep understanding the data & creative scientific research to find the best fit rules (models) to capture those patterns.

Software Engineers are at the heart of turning these ML/AI models into products in collab with ML Engs & Data Scientists.

But it all starts with predicting whether doordash refund request is legit

- To automate that decision rather than back and forth chat - How?
- By using a sample of ‘historic’ refund requests - with 2x sets of data:
    - 1. Bunch of info on each request & 2. Decision the customer service rep made (label) for the request: Yes/No (or e.g. 1 or -1)
- If we can work out a way (rules/instructions) to convert the info on each req in the sample into the Yes/No (1 or -1) that we have - we can use that ‘convertor’ on future refund requests - to PREDICT the right decision 1 or -1.
    - What info on each request will matter?



Data exploration - know your data and how to ‘convert’ it

1. Select sample of historic refund requests - each has data on

   a. Background on the request (Account history, request size, IP patterns)

   b. Whether the request was refunded or not Yes/No (labeled 1 or -1)

2. Pick background info that matters (select your ‘features’)

3. Identify rules (boundaries - max/min values for the background data) that convert as many of sample correctly as possible to their labels 1 or -1 - a ‘best fit’ for sample

4. Double-check the the converter on some more sample data (validation) to prevent overfitting

5. Then apply the converter to new refund reqs without label and make a prediction!

~But how did we identify the boundaries?~

ML/AI - the concept and the discipline

In ‘machine learning’ the machine (computer) tries different values (e.g. here combinations of ‘boundaries’) to assess how many of sample convert to the label given by the customer service rep

- There are many combinations of (parameter) values to try before the machine has ‘learned’ the best set (that maximizes the number of correct conversions for the sample)

The discipline of ML & AI:

- **Techniques/tools for finding best convertor** (the ‘model’ - here a ‘decision tree’ but there’s many: SVMs, regression, NNs)
- **Ways to minimize time/effort** it takes to the convertor & its details (model parameters)
- **Ensuring the sample reflects the population** and is aligned w our goals/values (preventing overfitting, drift, bias & misalignment)

Getting our fraud detection model to ‘production’ (live online)

Data exploration & model **prototype model** (often Data Scientist-led)

- Smaller sample (100k refunds - 10-20% of final dataset)
- Understand problem & data - e.g. pre-processing, feature selection/pruning/ engineering, fixing imbalanced data (synthetic minority oversampling), interpretability
- Experiment w/ different models/approaches - e.g. ensemble learning (here: random forests) - evaluation metrics (accuracy, precision/recall) + tracking (MLFlow)

Produce **production model** (often ML Engineer-led)

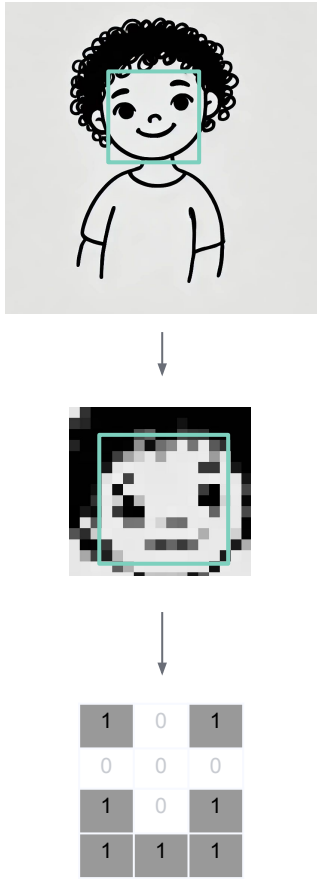
- Performance of training/inference at scale w/ full dataset (e.g. 1m refunds) - e.g. XGBoost
- End-to-end deployment (preprocessing, training, server/API design)
- Ongoing model performance - optimize retraining & prevent ‘drift’

What else could we predict?

If we have **labeled data** - i.e. some info represented as numbers + labels of what the info is

- And **create a convertor** for that sample (converting the input values into the label values)
- Then, assuming the sample reflects population
- We can use that convertor on the rest of population (that isn't labeled) to predict those labels

What other info represented as 'numbers' + 'labels of what that info is' can you think of?



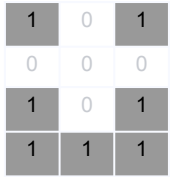
Neural networks

For images (and much other labeled data) - game was changed by neural networks (introduced 1950s, took off in 2000s) - can interpret images better than any prior technique

Convert pixels (#s) into output label (here smile: 1 or smile: -1) by:

- **Multiplying each pixel** by a different number & add up the total
- Comparing the total to the actual output label value (smile 1 or -1)
- **Adjusting the #s we multiply each pixel by** until the pixels labeled ‘smile:1’ add to 1 & ‘smile: -1’ add to -1 - gives us a grid of multipliers

Ignore pixels (multiply by 0) that are same between differently labeled images - have no effect



smile: 1

smile: -1

Neural networks

Then assuming the sample reflects the population, **we can use this converter on new images in population that don't have a label 1 or -1**

- And run the convertor on them to **predict their label**

Association with brain structure is more metaphorical

- Nodes (here pixels) as neurons & connections (multipliers/weights) as synapses
- Brain doesn't mirror the math used to find best fit multipliers

So do these multipliers have any meaning?

smile: 1

smile: -1

The different numbers multiplying each pixel then have some potential meaning

If pixels are 1 when dark (for features of face e.g. eyes, mouth)

And ‘smile: 1’ when pic has smile, and ‘smile: -1’ when not

- Then the **multipliers will be positive (+ve)** for those **pixels that are most distinctive to smile** images (smile:1)
- The multipliers will be **negative (-ve)** for **pixels most distinctive to not-smile** labeled images (smile: -1)

In this case the multipliers are **like correlations** - you’ll see high +ve multipliers for pixels strongly associated/correlated with smile:1 (ie smile)

Let’s **validate our model** with some more images

- Hm, more pixels matter than we thought so accuracy has dropped, we need to expand sample and add multipliers for more (all) pixels

Figuring out (learning) multipliers for all the pixels with a larger sample (1)

We need a process to **figure out the multipliers** that convert as many of the sample's pixels to match their labels (smile 1 or -1) as possible

- Once the sample's too big to 'see'

We can't 'brute force' all multiplier (weight) combos to get best fit - **trillions of combos**

So start with 0s for all 12 multipliers (weights) & **one-by-one** 'compute' for each weight: is it better to tweak it ↑/↓ to convert more (of the 4 images) accurately to their target labels?

Then make **all the changes at once** (one iteration of training using gradient descent) and start again with new round of independent weight checks

smile: 1

smile: -1

smile: -1

smile: 1

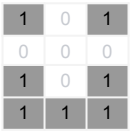
Figuring out (learning) multipliers for all the pixels with a larger sample (2)

But surely we can't check the impact of each weight change independently like that?

If we keep our tweaks small enough we can - because we:

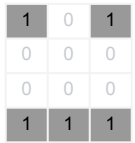
- (a) avoid interaction effects (small change won't change whether to ↑/↓ other weights)
- (b) prevent overshooting (local minima vs global minimum error)'

Machine learning = 'steady computer tweaking'



smile: 1

smile: -1



smile: -1

smile: 1

After 1 round of changes to the multipliers our accuracy gets better

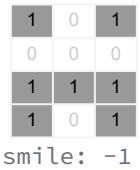
Keep tweaking until we have a fit?

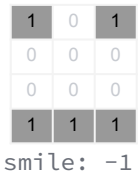
- But there's actually no set of multipliers (weights) that get perfect 1 & -1 output for even our tiny sample
- The 4th image's converted output is changing too fast

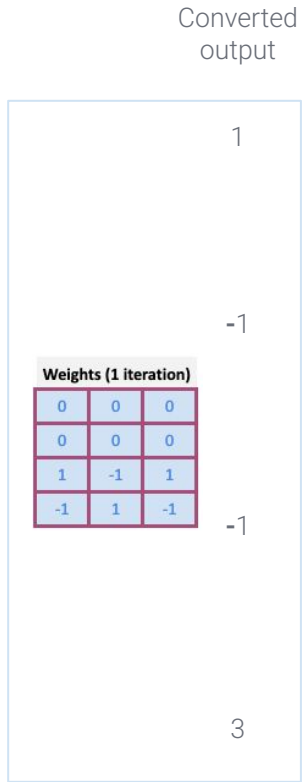
Could look for how positive (+ve) and negative (-ve) the converted output is

- More +ve, more confident it's a smile
- More -ve, more confident it's not a smile

But how +ve/-ve is confident? Introducing sigmoid function







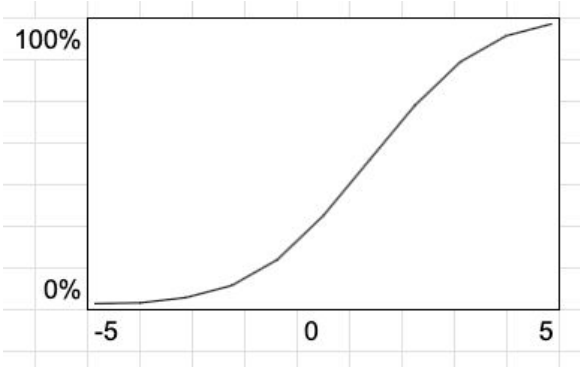
Sigmoid (σ) enables a set of weights that actually work for this sample

Sigmoid function converts all positive (+ve) numbers to >50% and all negative (-ve) numbers to <50%

- greater than 5 = ~99%
- less than -5 = ~1%

In the sample, switch label to smile:100% or smile:0% then run the sigmoid function on output of the convertor to get percentages (%s)

- Conversion now doesn't need to be exact - difference between output of image 1 and 3 dampened by squashing with sigmoid
- Convertor only has to convert the pixels to whatever labels we gave them (then look for those labels - **now percentages** - when we run it on the population)

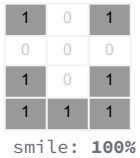


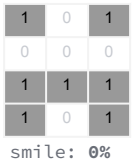
Expanding our sample and applying gradient descent - the machine is learning

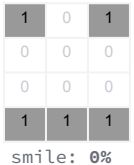
Small adjustments to weights to improve conversion accuracy (ie reduce avg error)

- After **1 iteration** already have reduced error
- After **100 iterations**, weights get close enough for our model to be run on a validation image:
- [1,0,1,0,0,0,1,1,0,1,1,1], smile: 0%

How's it do on this validation data?







After 1 round	Converted output	Sigmoid-adjusted
	σ(1)	73%
	σ(-1)	27%
	σ(-1)	27%
	σ(3)	95%

Weights (1 iteration)

0	0	0
0	0	0
1	-1	1
-1	1	-1

Avg Error: 21%

After 100 rounds	Converted output	Sigmoid-adjusted
	σ(2)	89%
	σ(-2.4)	8%
	σ(-2.3)	9%
	σ(5.2)	99%

(100x iterations)

-0.5	0.0	-0.5
0.0	0.0	0.0
2.2	-2.5	2.2
-1.6	1.9	-1.6

Avg Error: 7%

Target output
100%
0%
0%
100%

Getting our neural network to production (live online)

Data exploration & develop **prototype model**

- Sample of 100k labeled color images (1000x1000 RGB)
- Understand data & pre-process - Dimensionality reduction (with e.g. CNNs, Autoencoders), normalize/augment data
- Neural network design - add layers (output of weights input to more weights), feedforward vs recurrent, CNNs, transformers
- Refine via evaluation metrics + hyperparameters (learning rate, regularization techniques)

Develop & deploy **production model**

- Scaled-up model - distribute training workload across machines (GPUs/TPUs)
- Production ready data/model pipeline - API design & deployment
- Monitoring and periodic retraining + interpretability (e.g. SHAP)

Combining models - Predicting pixels (image and video)

- Image generation via GANs (and diffusion, CVAE models)
  - Model generating random noise passed to Smile recognition model
  - Performance monitored and random noise model refines generation - until passing smile recognition
- Text-conditional image generation (e.g. Dall-E 2 from OpenAI)
  - CLIP (Contrastive Language-Image Pretraining) - maps images to words/phrases
  - GLIDE - diffusion model for image generation

You've seen the core principles of Machine Learning & Neural Networks

- **Probability** - we can make predictions using a **model** of the population by generalizing from a sample
- **Layers** of multipliers (**weights**) - model **parameters** that capture the association between **features** (input values) and **label/class**
- **Training** to iteratively tweak the weights and get better matching (**gradient descent**)
- Model refinement - **Hyperparameters** (e.g. learning rate) **Activation** functions (e.g. sigmoid)
- Model pipeline from **preprocessing** of the data before **training** & **inference**, to **API design**

Extending these principles to unfathomable scale

- All AI/ML models we develop & deploy as engineers build on these principles & techniques
- Much of their power come from the vast scale of training data, features & model complexity
- Next up we'll see this **play out in LLMs** (large language models) - where models may have billions of parameters, samples with trillions of elements and potential 'emergent phenomena'
    - But throughout, remember that it all depends on our ability to **generalize from a sample to the population** by **building a model** that **captures the patterns of the sample** - and (assuming the sample reflects the population) we make predictions about our population

It just happens the 'population' here is something akin to all possible use of human language - which has some pretty profound consequences

Where to go next - get building with code (and now models)

- Prediction (ML/AI models) in software engineering has **changed what we can tackle with code** - whole new domains open up to us (law, health, climate)
- Software Engineers (w Data Scientists, & ML Engineers) are at the heart of turning these ML/AI models into **products that solve problems**
- Recognize what makes you great at building with code - flexibility and capacities
    - Integrate these new mental models and tools but center your problem solving & technical communication (as the tools will change)
- Next steps:
    - Investigate ML models development **from these principles** of sample & generalization (e.g. data exploration & model prototyping)
    - Experiment w deploying models in production and the tools to scale these and integrate with **your application builds**