The Last Few Bits..

CS771: Introduction to Machine Learning

Piyush Rai

Debugging ML Algorithms

What is going wrong?

- What to do when our model (say logistic regression) isn't doing well on test data
  - Use more training examples?
  - Use a smaller number of features?
  - Introduce new features (can be combinations of existing features)?
  - Try tuning the regularization parameter?
  - Run (the iterative) optimizer longer, i.e., for more iterations
  - Change the optimization algorithm (e.g., GD to SGD or Newton..) or the learning rate?
  - Give up and switch to a different model (e.g., SVM or deep neural net)?

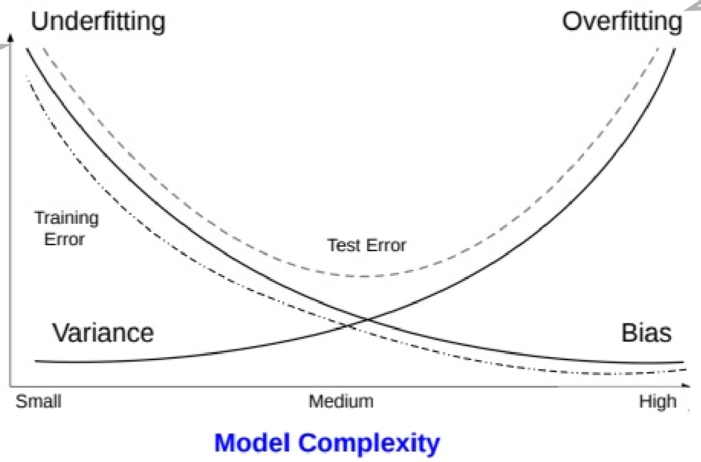
High-Bias or High-Variance?

- The bad performance (low accuracy on test data) of a model could be due either
  - High Bias: Too simple model; doesn't even do well on training data
  - High Variance: Even small changes in training data lead to high fluctuation in model's performance
- High bias means underfitting, high variance means overfitting
- Looking at the training and test error can tell which of the two is the case

Both training and test errors are large

Adding more training examples won't help as the model itself is too simple

Use a more complex model or increase the number of features



Small training error and large test error

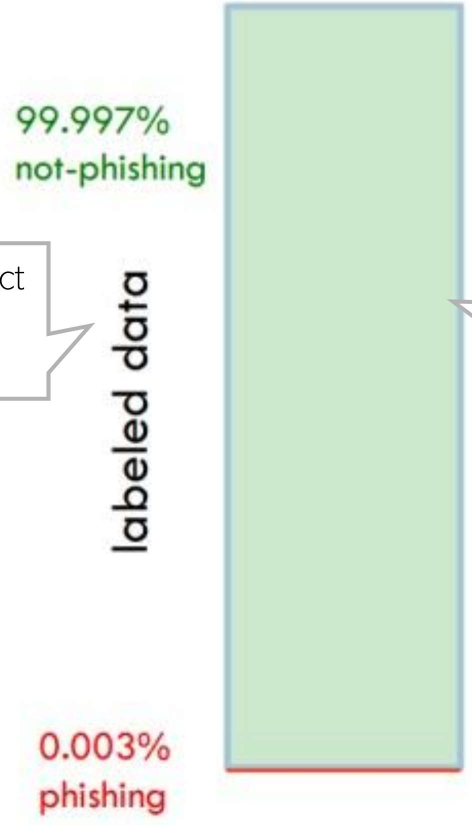
Adding more training examples can possibly help here because we possibly have a complex model but not enough training data to learn it well

Or regularize the model better to prevent overfitting or use fewer features, or switch to a simpler model

Learning from Imbalanced Data

Learning when classes are imbalanced

- When classes are imbalanced, even a “stupid” classifier can give high accuracy but looking at accuracy alone may be misleading



A stupid classifier: Simply predict everything as non-phishing. Gives close to 100% accuracy

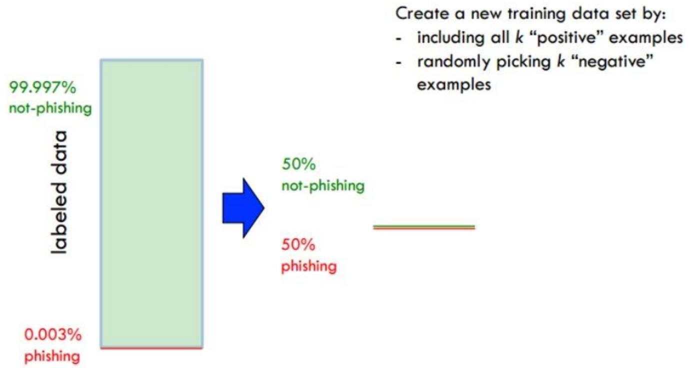
But likely to do badly on test data, particularly phishing mails (all will be predicted as non-phishing)

If we use classification loss/accuracy as our loss function, the learned model may indeed prefer simply predicting each input to be from the majority class

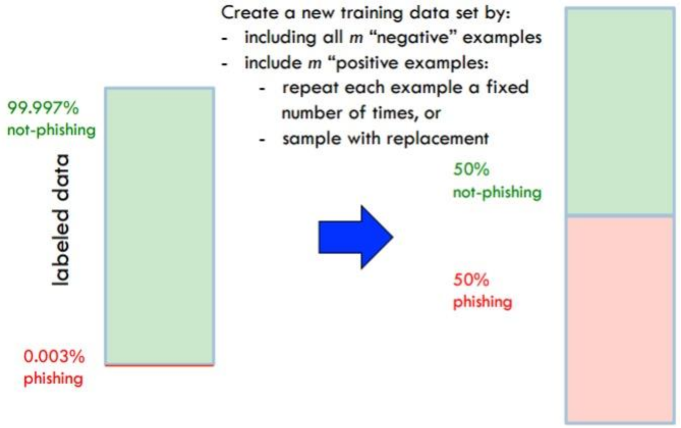
For such data with imbalanced classes, we need ways so that the majority class is not given undue importance

Solution 1: Balancing the training data

- Can balanced the training data by
  - Under-sampling the majority class examples



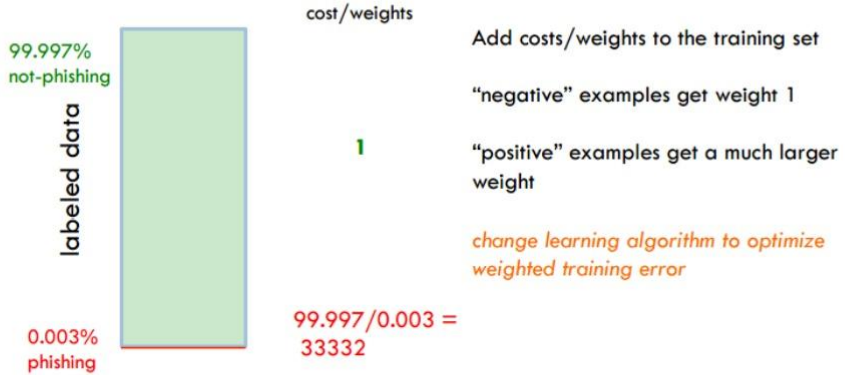
  - Over-sampling the minority class examples



Equivalent to

Weighted loss function with much larger importance given to loss function terms of positive examples than negative examples

$\mathcal{L}(\boldsymbol{w}) = \sum_{i=1}^{N} \beta_{y_i} \ell(\boldsymbol{x}_i, y_i, \boldsymbol{w})$
where $\beta_{+1} \gg \beta_{-1}$



Solution 2: Changing the loss function

- Don't use loss functions that define loss or accuracy on per-example basis

This loss function is a simple sum of losses on individual training examples. Not ideal for imbalanced classes

$$L(\boldsymbol{w}) = \sum_{i=1}^{N} \ell(x_i, y_i, \boldsymbol{w})$$

- Instead, use loss function that use example pairs (one positive and one negative)
- Assuming our model to be defined by some function $f(\boldsymbol{x})$ (e.g., $\boldsymbol{w}^\top \boldsymbol{x}$), define a loss

An input with positive label

An input with negative label

$$\ell(f(\mathbf{x}_n^+), f(\mathbf{x}_m^-)) = \begin{cases} 0, & \text{if } f(\mathbf{x}_n^+) > f(\mathbf{x}_m^-) \\ 1, & \text{otherwise} \end{cases}$$

Now we don't care about per-example accuracy but care about whether the positive examples get a higher score than the negative examples (i.e., we are only preserving their relative rank)

Usual regularaizer on f

$$\sum_{n=1}^{N_+} \sum_{m=1}^{N_-} \ell(f(\mathbf{x}_n^+), f(\mathbf{x}_m^-)) + \lambda R(f)$$

Such loss functions can known as "pairwise loss functions"

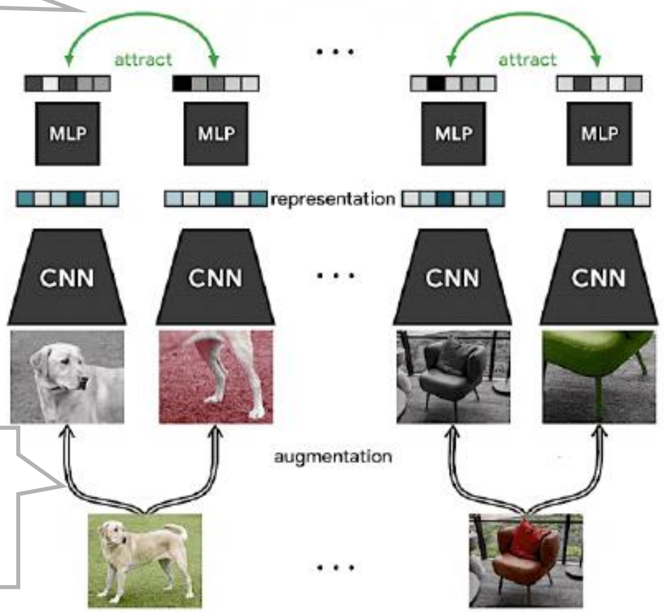
Contrastive Learning

Learning Good Features by Comparison

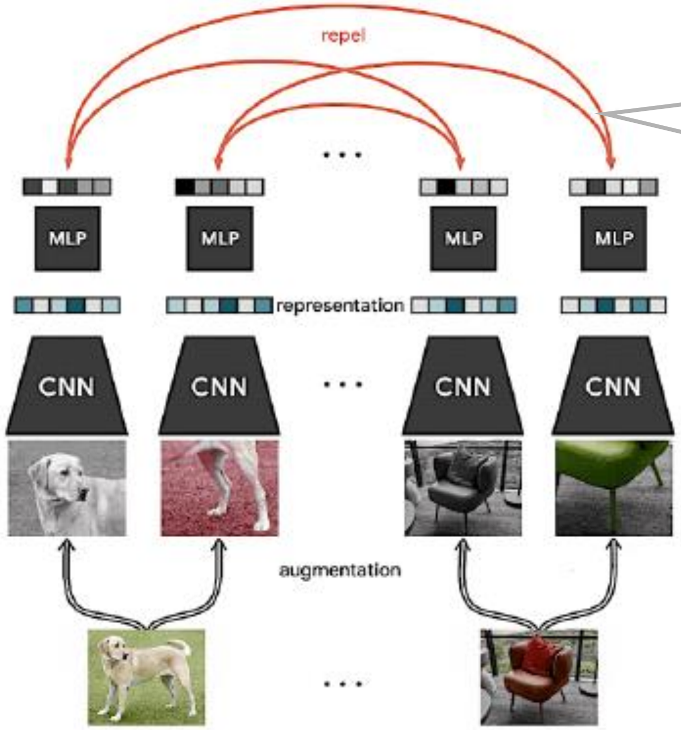
Or "triplets" (e.g., "cat" is more similar to "dog" than to a "table")

- Can learn good features by comparing/"contrasting" similar and dissimilar object pairs
- Such pairs can be provided by to the algorithm (as supervision), or the algorithm can generate such pairs by itself using "data augmentation" (as shown in example below)

The corresponding embeddings for such pairs must be close ("attract") to each other



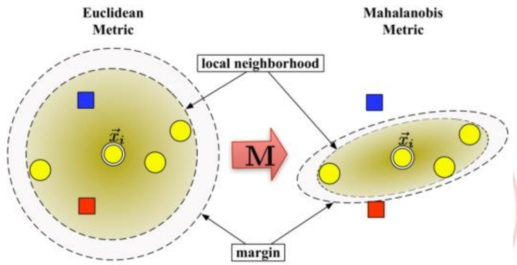
Augmentation by cropping and resizing. The class of the image remains unchanged in this augmentation



The embeddings for "non-match" pairs must be far away ("repel") from each other

Distance metric learning by learning $\mathbf{M}$ given similar/dissimilar pairs

$$d(\boldsymbol{x}_i, \boldsymbol{x}_j) = \sqrt{(\boldsymbol{x}_i - \boldsymbol{x}_j)^\top \mathbf{M} (\boldsymbol{x}_i - \boldsymbol{x}_j)}$$

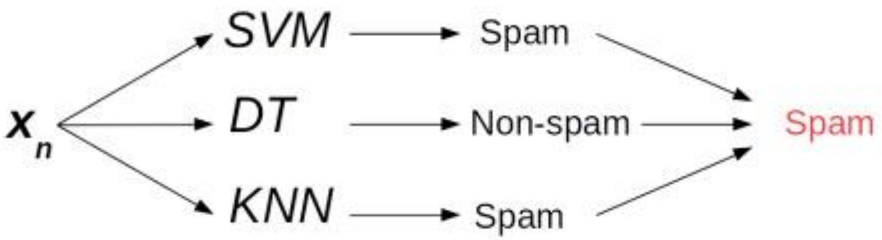


- Such "contrastive learning" of features is also related to "distance metric learning" algos

Ensemble Methods

Some Simple Ensembles

- Voting or Averaging of predictions of multiple models trained on the same data

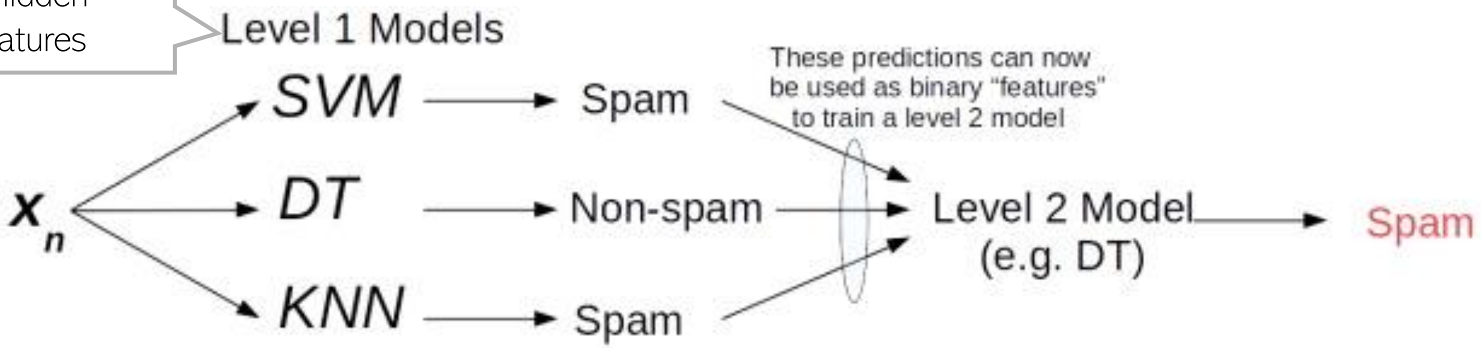


- "Stacking": Use predictions of multiple already trained models as "features" to train a new model and use the new model to make predictions on test data

Stacking sort of has a flavor of deep learning where hidden layers extract good features

Here, that role is being played by these other already trained models

In stacking, level 1 and level 2 models are trained independently (first level 1 and then level 2)

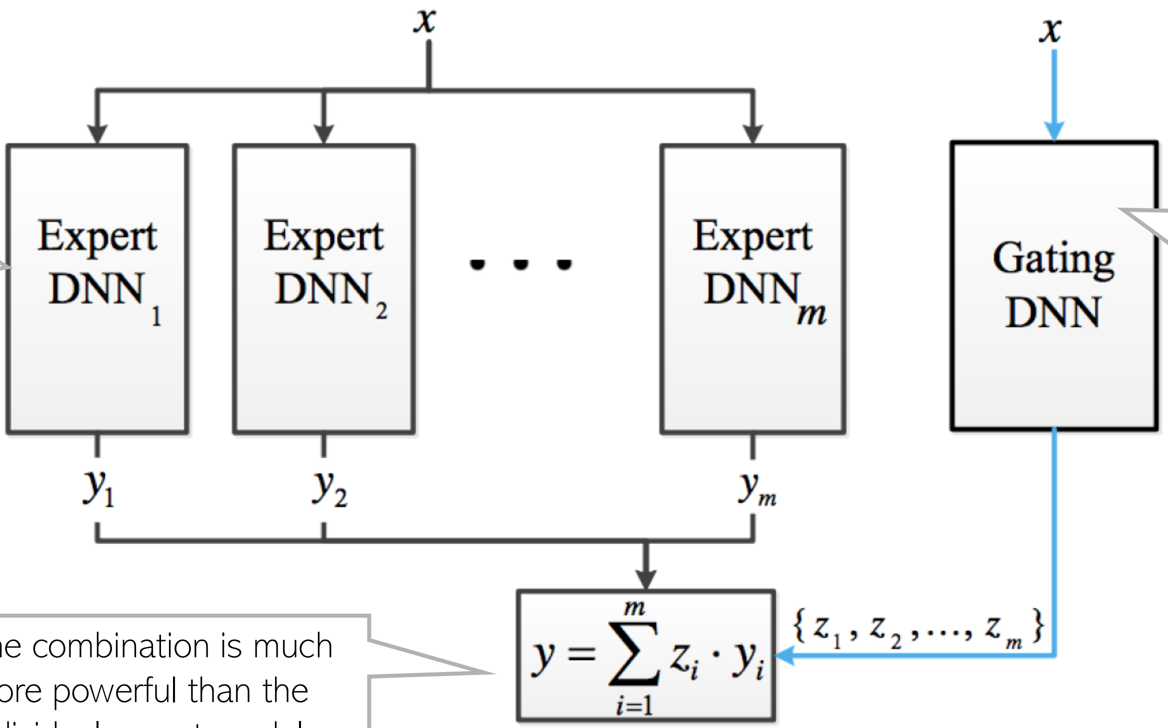


Mixture of Experts (MoE) based Ensemble

- Mixture of Experts (MoE) is a very general idea
- We assume $\boldsymbol{m}$ "simple" models, usually of the same type, e.g., $\boldsymbol{m}$ linear SVMs or $\boldsymbol{m}$ logistic regression models, or $\boldsymbol{m}$ deep neural nets (usually all with same architecture)

Experts models and the gating/routing function are learned jointly using all the training data

MoE can also be formulated like a latent variable model and learned using ALT-OPT or EM



Gating (also called "routing") function which outputs an $\boldsymbol{m}$ dimensional vector $\boldsymbol{z} = [z_1, z_2, \ldots, z_m]$ which indicates us how much importance should be given to each expert for this input $\boldsymbol{x}$

Some variants of MoE pick only one expert or a small subset of expert

In that case $\boldsymbol{z}$ will be a one-hot or sparse binary vector

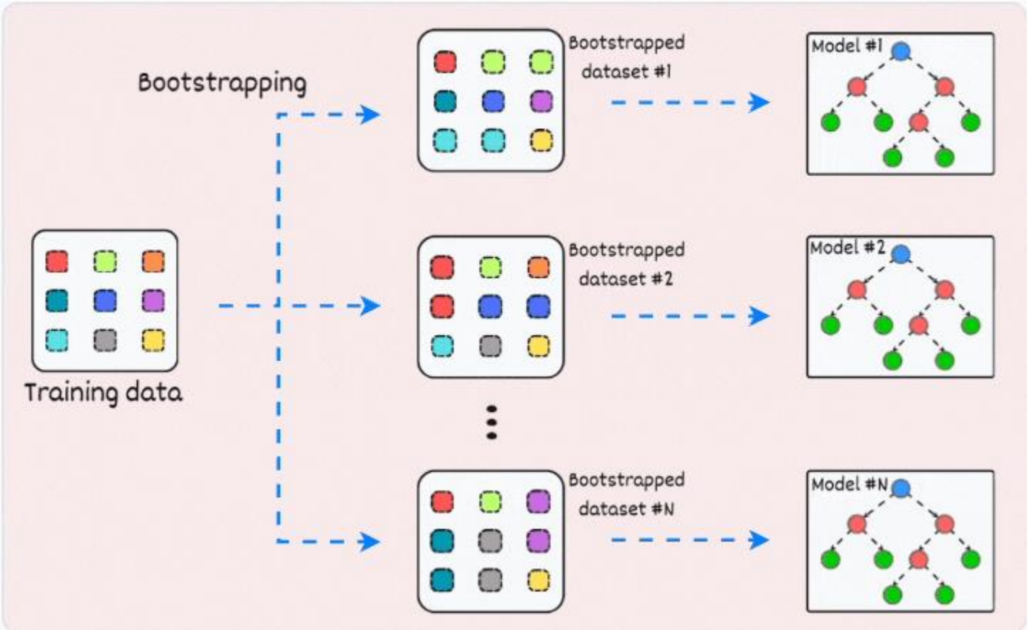
The combination is much more powerful than the individual expert models

- MoE is very popular in classical ML as well as "modern" deep learning

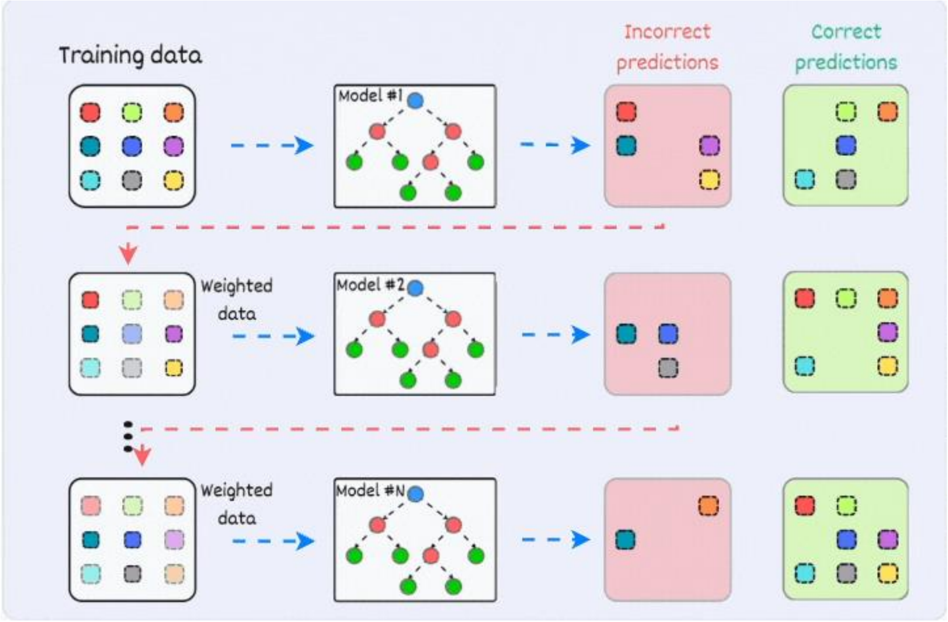
Ensembles using Bagging and Boosting

- Both use a single training set $\mathcal{D}$ to learn an ensemble consisting of several models
- Both construct N datasets from the original training set $\mathcal{D}$ and learn N models

Bagging



Boosting



- Bagging can do this in <u>parallel</u> for all the N models
- Boosting requires a <u>sequential</u> approach for N rounds

Figure credit: https://www.blog.dailydoseofds.com/p/an-animated-guide-to-bagging-and

Bagging (Bootstrap Aggregation)

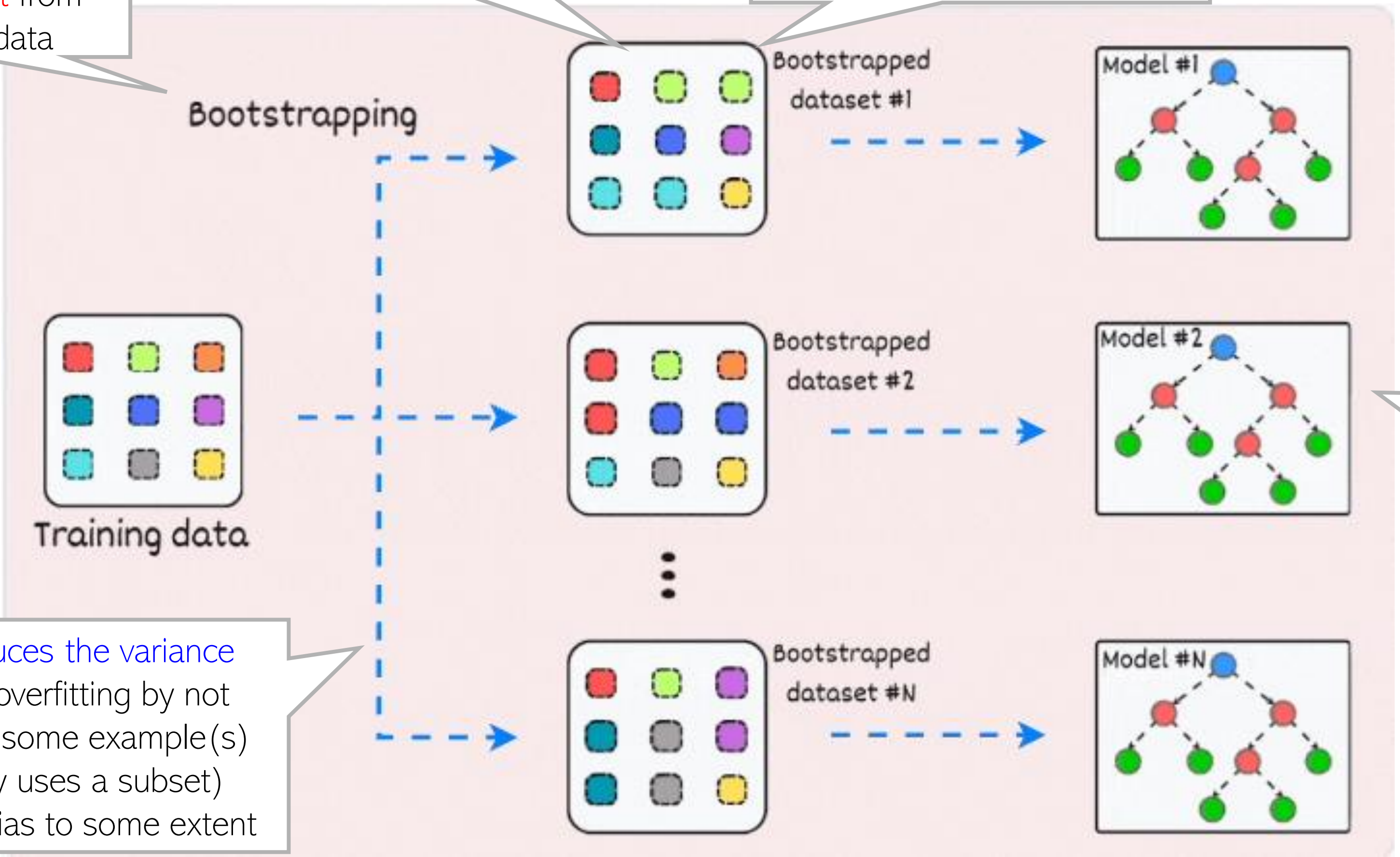
Generated bootstrapped datasets using sampling with replacement from original training data

Each bootstrapped dataset consists of a subset of all the training examples

Sampling with replacement will give about 63% unique training examples (if both datasets are of equal size)

Bootstrapping can also be done at feature level (each model using a randomly chosen subset of all the features)

Random forests use both bootstrapping at examples level and at features level



To predict the output for a new test input, we apply all these models and average their predictions

Bagging primarily reduces the variance (reduces chances of overfitting by not relying completely on some example(s) since each model only uses a subset) but can also reduce bias to some extent

Figure adapted from: https://www.blog.dailydoseofds.com/p/an-animated-guide-to-bagging-and

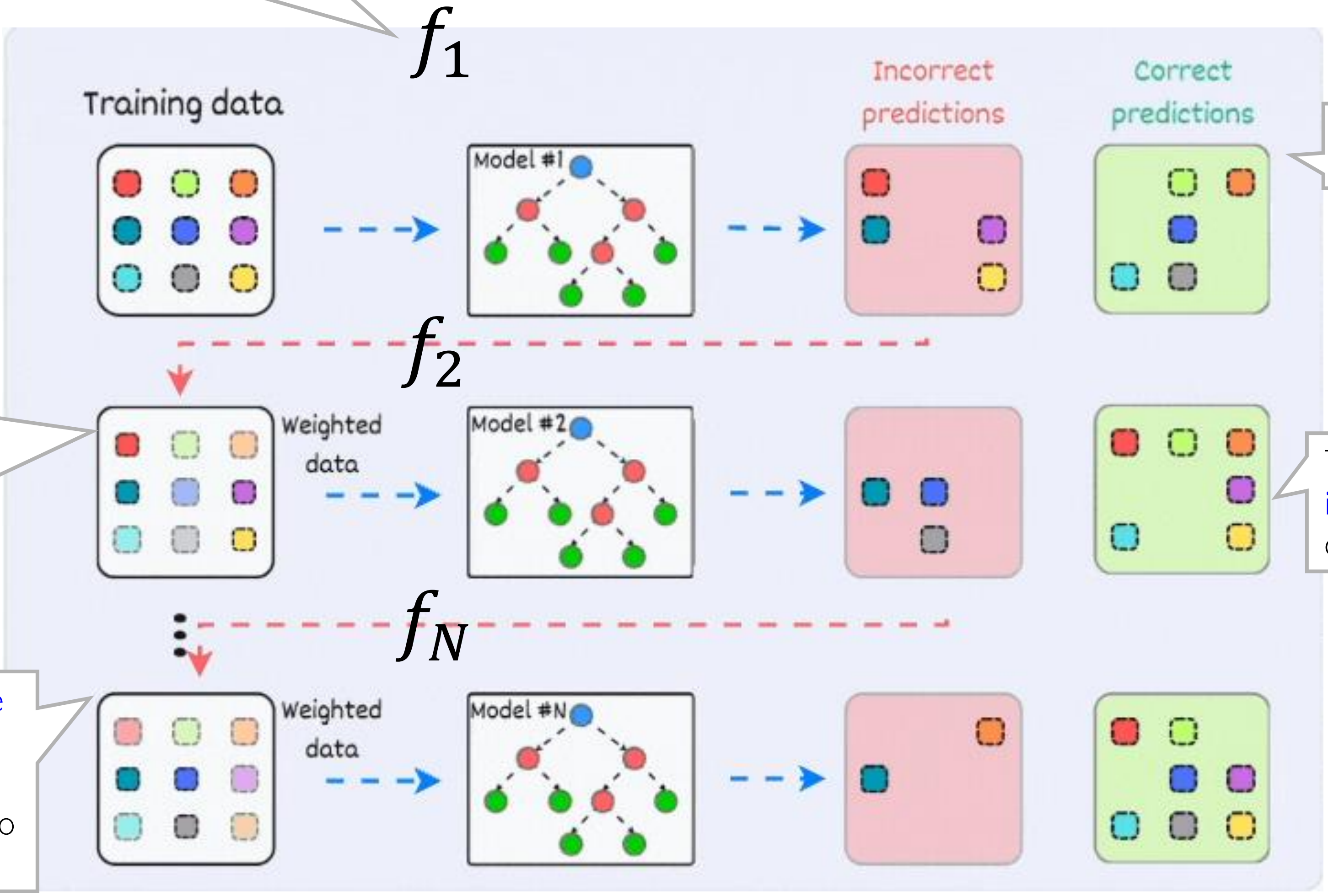
Boosting

Boosting assumes that the individual models are simple/weak models that can be easily learned

Boosting trains them sequentially and combines them to get a "boosted" powerful model

Note that here we have two types of importances: importance of each training example and importance of each model

"Weighted data" means that we are increasing the importance of examples that were mis-predicted in the previous round and decrease it for examples that were correctly predicted



Importance of each model is its accuracy

The final model will be an important-weighted combination of all these models

Boosting primarily reduces the bias by making the weak (underfitted) models stronger but can also reduce variance to some extent

Figure adapted from: https://www.blog.dailydoseofds.com/p/an-animated-guide-to-bagging-and

A Boosting Algo: AdaBoost (Adaptive Boosting)

Importance of the training example $(\boldsymbol{x}_i, y_i)$

We might know this beforehand or estimate it during training

- In many ML problems, we can assign importance weight to each example, e.g., by weighing each term in the loss functions, i.e., $\mathcal{L}(\boldsymbol{w}) = \sum_{i=1}^{N} \beta_i \ell(\boldsymbol{x}_i, y_i, \boldsymbol{w})$

Initially assume equal importance for all training examples

- AdaBoost is based on optimizing such a loss function
  - Initialize the ensemble as $\mathcal{E} = \{\}$ and $\boldsymbol{\beta}$ as $\boldsymbol{\beta}^{(0)} = [\frac{1}{N}, \frac{1}{N}, \dots, \frac{1}{N}]$
  - For round $t = 1,2,\dots,T$
    - $\boldsymbol{w}^{(t)} = \operatorname{argmin}_{\boldsymbol{w}} \sum_{i=1}^{N} \beta_i^{(t-1)} \ell(\boldsymbol{x}_i, y_i, \boldsymbol{w})$ and add it to ensemble $\mathcal{E} = \{\mathcal{E} \cup \boldsymbol{w}^{(t)}\}$
    - Define the total loss of $\boldsymbol{w}^{(t)}$ as $L(\boldsymbol{w}^{(t)}) = \sum_{i=1}^{N} \beta_i^{(t-1)} \ell(\boldsymbol{x}_i, y_i, \boldsymbol{w}^{(t)})$

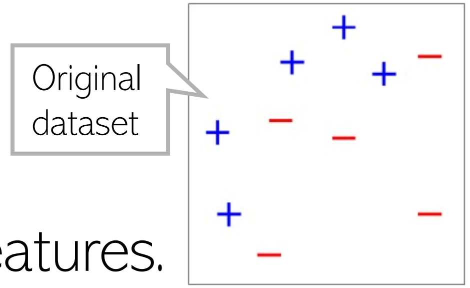
      Or the importance weighted total error
    - Compute the "importance" of $\boldsymbol{w}^{(t)}$ for the $\mathcal{E}$ as $\alpha_t = f(L(\boldsymbol{w}^t))$

      f is some function such that α_t is high if total loss $L(\boldsymbol{w}^t)$ is low, and vice-versa
    - Increase/decrease importance β_i of each training instance $(\boldsymbol{x}_i, y_i)$ for next round as

$$\beta_i^{(t)} \propto \begin{cases} \beta_i^{(t-1)} \times \exp\left(\alpha_t \ell\left(\boldsymbol{x}_i, y_i, \boldsymbol{w}^{(t)}\right)\right) & \text{(Increase if } \boldsymbol{w}^{(t)} \text{ mispredicted } (\boldsymbol{x}_i, y_i)) \\ \beta_i^{(t-1)} \times \exp(-\alpha_t \ell(\boldsymbol{x}_i, y_i, \boldsymbol{w}^{(t)})) & \text{(Decrease if } \boldsymbol{w}^{(t)} \text{ correctly predicted on } (\boldsymbol{x}_i, y_i)) \end{cases}$$

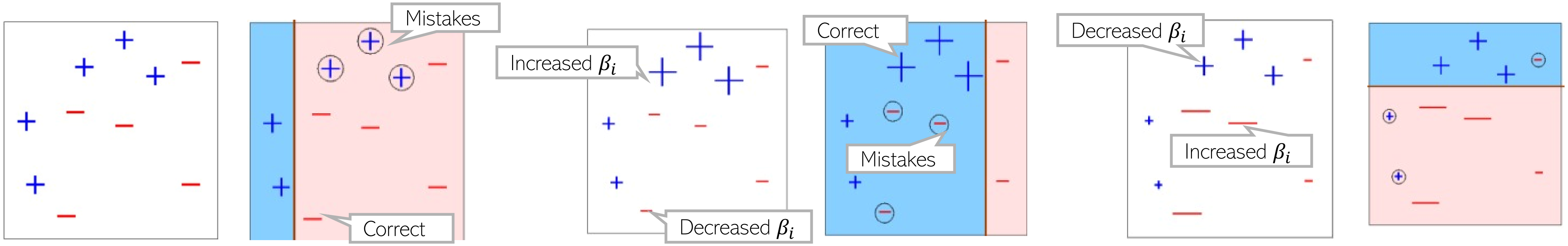
  - Final model is $\hat{\boldsymbol{w}} = \sum_{t=1}^{T} \alpha_t \boldsymbol{w}^{(t)}$ importance-weighted average of all $\boldsymbol{w}^{(t)}$'s

AdaBoost: An Illustration

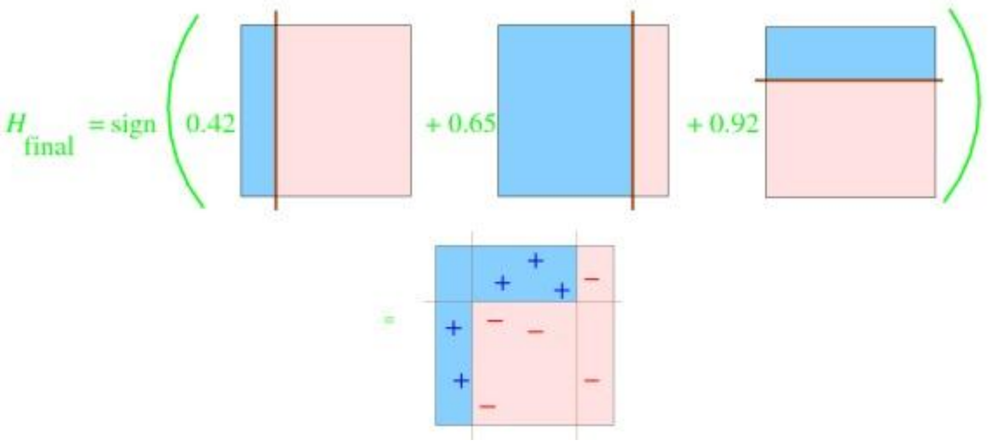


- Suppose we have a binary classification problems with each input having 2 features.
- Suppose we have a weak model like a simple DT (decision stump)
- Illustration of AdaBoost using a decision stump if run for 3 rounds

Round 1 **Round 2** **Round 3**



- The ensemble represents the overall model
- We got a nonlinear model from 3 simple linear models
- Note that the ensemble was constructed sequentially



Gradient Boosting

- Consider learning a function $f(x)$ by minimizing a squared loss $\frac{1}{2}\left(y - f(x)\right)^2$
- Gradient boosting is a sequential way to construct such $f(x)$
- For simplicity, assume we start with $f_0(x) = \frac{1}{N}\sum_{i=1}^{N} y_i$
- Given previously learned model $f_m(x)$, let's assume the following "improvement" to it

$$f_{m+1}(x) = f_m(x) + h(x)$$

"Residual" which, if added to $f_m(x)$, will make the new prediction $f_{m+1}(x)$ closer to y

- Thus the goal for the next round is to learn the "residual" $h(x) = y - f_m(x)$
- Residual is negative gradient of the loss w.r.t. $f(x)$ - thus called "gradient boosting"
- The final model $f_M(x)$, once the residual is sufficiently small, is what we will use
- The idea of gradient boosting is applicable to classification too
- XGBoost (eXtreme Gradient Boosting) is a very popular grad boosting algo

Based on sequentially constructing a DT

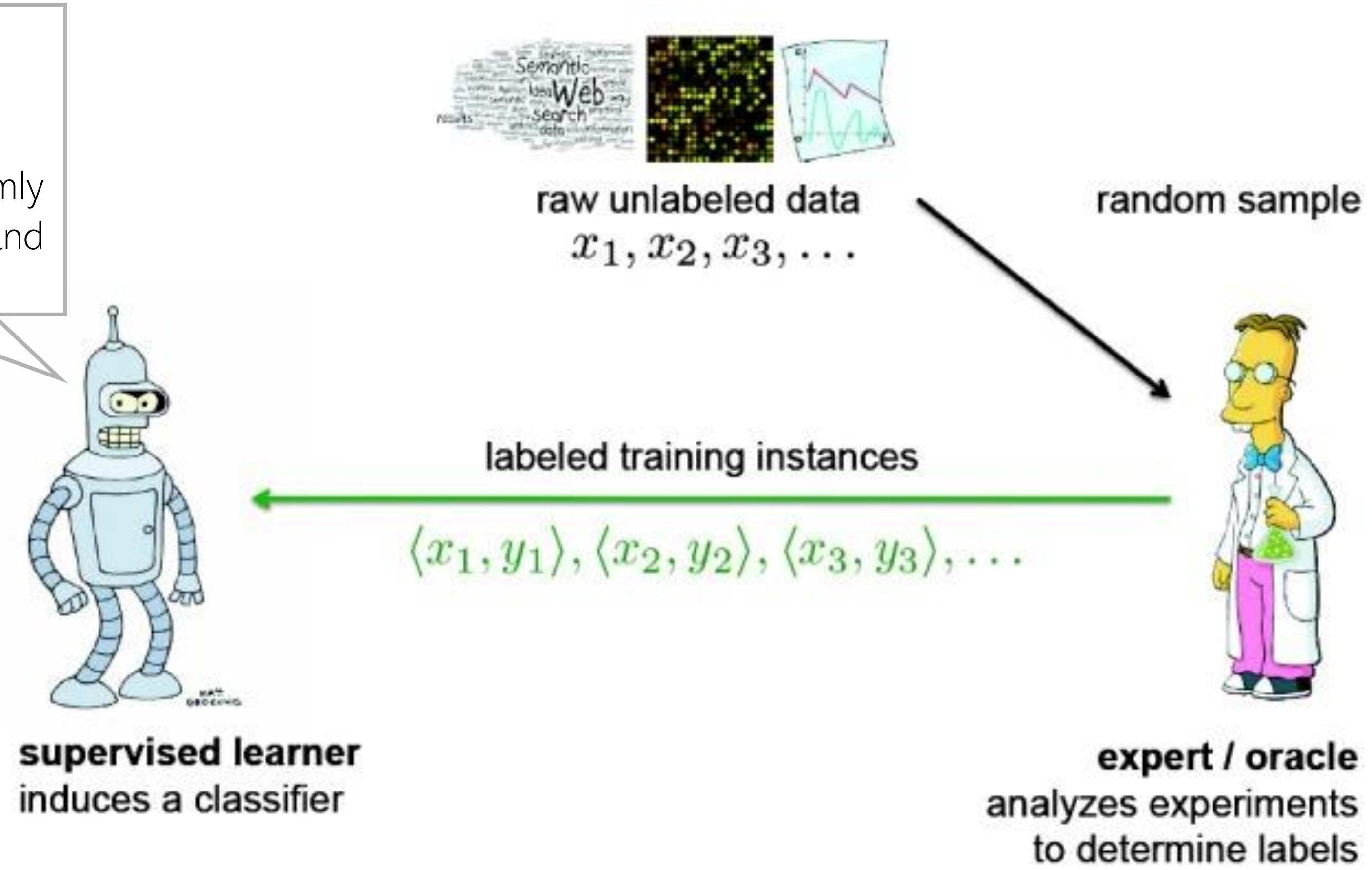
Active Learning

(an example of learning with human-in-the-loop)

Active Learning

- Standard supervised learning is "passive" (learner has no control; we just give it data)
- We take a random sample of inputs, get them labelled by an expert, and train a model

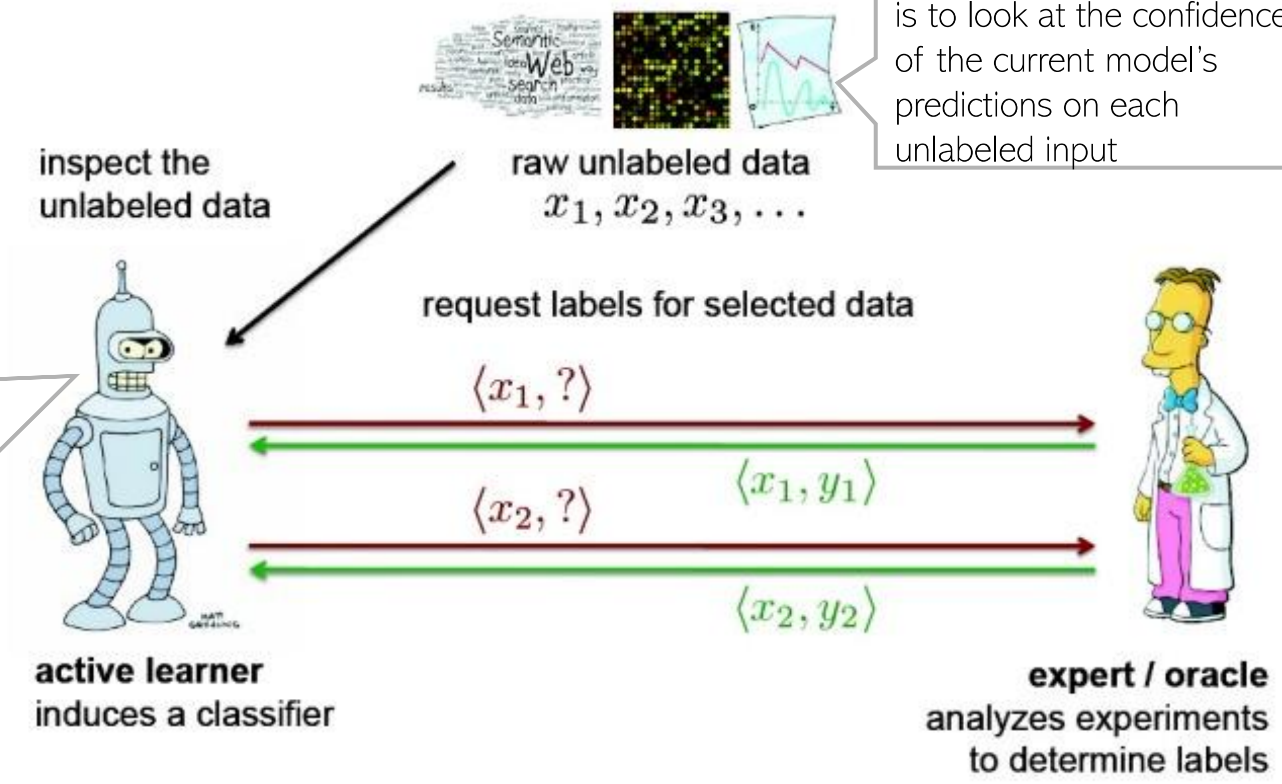
I had no say in what data I want to learn from ☹ I was just provided some randomly chosen labeled data and was asked to learn



Active Learning

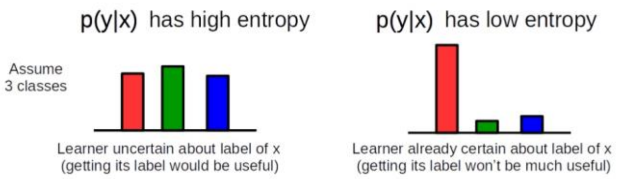
- In active learning, learner can request what training examples it wants to learn from

My current model in round t is f_t. Based on f_t, I have identified some unlabeled inputs that are "hard" for me to predict correctly (with high confidence). Please provide me their true labels. I will add them to my current training set and retrain to update f_t

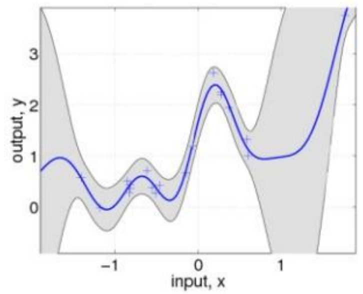


Many ways to identify what "hard" inputs are. One way is to look at the confidence of the current model's predictions on each unlabeled input

Can use entropy as confidence in classification problems



Can use variance as confidence in regression problems

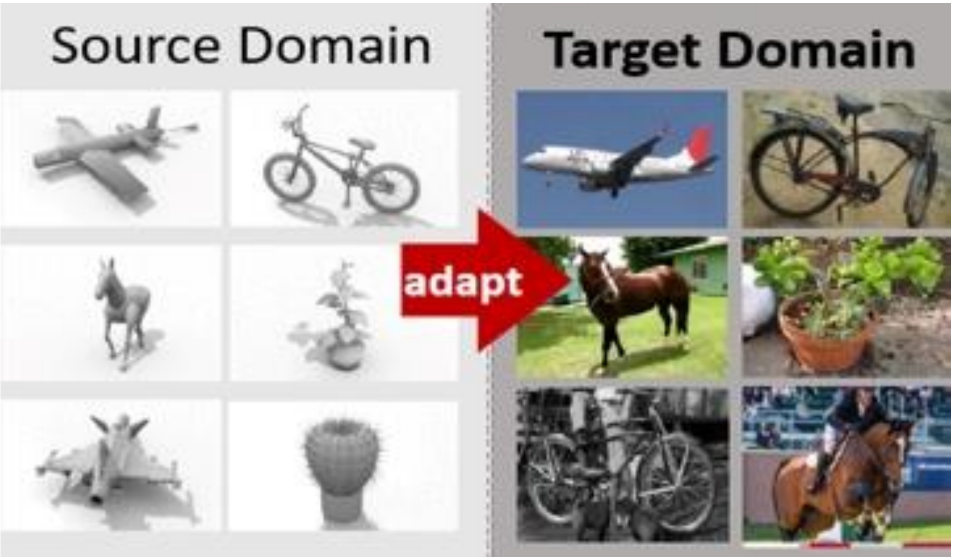


Learning in the wild

Domain Adaptation

- We may have a "source" model trained on data from some domain
- We might want to deploy it in a new domain
- Performance of the source model will suffer
- To prevent this, we usually perform "domain adaptation" or "transfer learning"
- These are broad terms covering a variety of techniques that "finetune" the source model using labelled/unlabeled data from the new domain

We do expect some "commonality" (e.g., some common set of features) between the two domains otherwise we can't hope to have any adaptation/transfer



Zero-Shot Learning

Recall Homework 1 programming problem where we used some simple methods to solve it using attribute vector (an embedding) of each class

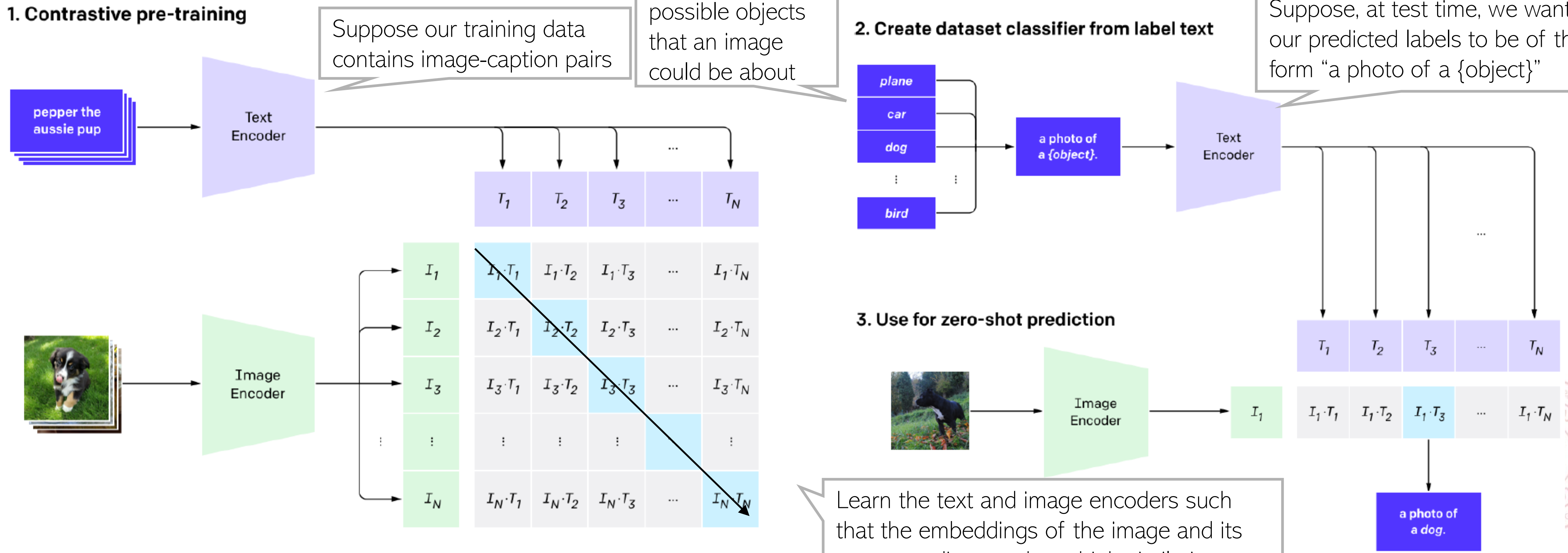
- What if our training data doesn't have the test data classes?
- Several methods to solve ZSL using deep learning. CLIP is a recent approach

CLIP: "Contrastive Language-Image Pre-training" (Radford et al, 2021)

Suppose our training data contains image-caption pairs

List of all possible objects that an image could be about

Suppose, at test time, we want our predicted labels to be of the form "a photo of a {object}"



Learn the text and image encoders such that the embeddings of the image and its corresponding text have high similarity

The ending note..

- Good features are important for learning well
- The "classical" ML methods we studied in this course still continue to have high relevance
- Success of deep learning is largely attributed to (automatically learned) good features
- Deep learning is not a panacea – often simple classical models can do comparably/better
- First understand your data (plot/visualize/look at some statistics of the data, etc)
- Always start with a simple model that you understand well
  - Try to first understand if your data really needs a complex model
- Think carefully about your features, how you compute similarities, etc.
- Helps to learn to first diagnose a learning algorithm rather than trying new ones
  - Understanding of optimization algos, loss function, bias-variance trade-offs, etc is important
- No free lunch. No learning algorithm is "universally" good