**Model Development and Offline Evaluation**

**Model Development and Training**

**Evaluating ML Models**

Because time and compute power are limited resources, you have to be strategic about what models you select.

Select the model depends on different resons the first is what your task about? And the secund is the data size and time to deploy, training time, prediction explanation (ml more explanation than NN models)

Six tips for model selection (Without getting into specifics of different algorithms)

1. Avoid the state-of-the-art trap: Don’t be fast to choose cutting-edge models without evaluating whether they are truly necessary for your problem. Simpler solutions may be more effective, faster, and cheaper than state-of-the-art models, which may not always outperform simpler models on your data.
2. Start with the simplest models: Begin with simple models that are easier to deploy and debug. These models serve as a baseline for comparison with more complex models, which may require more effort to improve.
3. Avoid human biases in selecting models: if you like a model, we will stick with it to try on it 100 times but for the other model that you make just one or two test on it this is not fair to go with the other model
4. Evaluate good performance now versus good performance later: The best model now does not always mean the best model two months from now. think how model performance might change with more data over time. A model that performs well initially might not be the best as more training data becomes available. Use learning curves to estimate potential future improvements.
5. Evaluate trade-offs: One classic example of trade-off is the false positives and false negatives trade-off. Reducing the number of false positives might increase the number of false negatives, and vice versa. Or A more complex model can give a better performance, but its results are less interpretable
6. Understand your model’s assumptions: Every model is based on assumptions, such as the independence of data points or the smoothness of input-output relationships. Understanding these assumptions helps in choosing the best model for your specific problem.

**Ensembles**

This method mean that you can use more than one model to make predication and each model of them called base learner, the prediction depends on voted for the best choice, and you can combine different models.

It is complex task but it can boost the performance.

There are three ways to create an ensemble: bagging, boosting, and stacking. But in some paper ensembles mean (bagging and boosting together).

Bagging:

instead of training one classifier on the entire dataset, you sample with replacement to create different datasets, called bootstraps, and train a model on them, for classification (majority vote) for regression (averaged). For example, random forest model, bagging can reduce the variance and prevent overfitting

Boosting:

It converts weak learner to strong one by training them iteratively. In each iteration (misclassified data will go to next learner (weighted data)), each model trained on weighted examples. For example gradient boosting machine (GBM), it focuses on difficult samples and improves overall accuracy.

Stacking:

Combines predictions from multiple models using another model (a meta-learner) to make the final prediction.

**Experiment Tracking and Versioning**

The process of tracking the progress and results of an experiment is called **experiment tracking**, Tools like MLflow, Weights & Biases, and DVC simplify tracking and offer dashboards for insights.

**Versioning** The process of logging all the details of an experiment for the purpose of possibly recreating it later or comparing it with other experiments, tools like DVC handle data versioning but with limitations.

Experiment tracking

* *loss curve*
* *model performance*
* log of *corresponding sample, prediction, and ground truth label*.
* *speed* of your model
* *System performance metrics*
* *parameter and hyperparameter*

By observing how a certain change in a component affects the model’s performance, you gain some understanding into what that component does.

Versioning

It is important to ensure that Hyperparameter changes weren’t properly documented and the Code modifications were not versioned ant the Data differences or updates went untracked.

It is important for Reproducibility and Standard Practice for Code using git.

But it has some challenges like Data Size: is that because data is often much larger than code, and Defining Differences (Diffs): Unlike code, where line-by-line comparisons work, defining meaningful diffs in data is unclear. Merge Conflicts Combining two different data versions doesn’t make logical sense without an associated model or context. Regulatory Compliance Regulations like GDPR may require deleting user data, complicating data versioning if older data must be removed.

**Debugging ML Models**

Challenges in debugging ML

First, ML models might appear to work but produce wrong results so it’s (silence flier).

Second, Fixes may require retraining the model, which is time-consuming.

Third, Bugs could stem from data, features, code, or infrastructure, making pinpointing the issue difficult. And it is Cross-Functional Complexity application

**Common Causes of Failure**

* Theoretical Constraints: Model assumptions don’t match the data for example using a linear model for nonlinear decision boundaries).
* Poor Implementation: Errors in coding the model for example, forgetting to stop gradient updates during evaluation).
* Hyperparameter Choices: Incorrect settings such as poor learning rate can prevent convergence.
* Data Problems: Issues like noisy labels, outdated normalization statistics, or incorrect data-label pairs.
* Feature Issues: Too few features can reduce predictive power, while too many can cause overfitting or data leakage.

Techniques:

*Start simple*

Build the simplest version of the model and add components gradually to isolate performance issues then test the model on a small subset of data. Important note: (Currently, many people start out by cloning an open-source implementation of a state-of-the-art model and plugging in their own data. On the off-chance that it works, it’s great. But if it doesn’t, it’s very hard to debug the system because the problem could have been caused by any of the many components in the model.)

*Overfit a single batch*

If it can’t overfit a small amount of data, there might be something wrong with your implementation.

*Set a Random Seed*

Ensures consistency across runs by eliminating randomness from weight initialization, dropout, or data shuffling.

**Distributed Training**

Data Too Large for Memory such as medical scans or language models often exceeds memory capacity.

For batch size small batch sizes lead to unstable gradient descent. And very large batch sizes require careful adjustment of learning rates, which becomes less effective beyond a certain point.

Data parallelism

Technique for scaling you split your data on multiple machines, train your model on all of them, and accumulate gradients.

It has some problems like Synchronous Gradient Descent (SGD): Slows down due to the straggler problem, where one slow machine holds up the entire process. Asynchronous SGD: Risk of gradient staleness, where weights are updated inconsistently across machines.

But you can make some Optimization by Sparse gradient updates reduce conflicts and improve efficiency.

Model parallelism

Split the model itself across multiple machines. For example: Machine 1 computes layer 1, Machine 2 computes layer 2, and so on. But Some components depend on the output of others, causing delays

Model parallelism can be misleading because in some cases parallelism doesn’t mean that different parts of the model in different machines are executed in parallel. For example, if your model is a massive matrix and the matrix is split into two halves on two machines, then these two halves might be executed in parallel.

Pipeline ParallelismProcesses micro-batches in stages to maximize parallel execution.

**AutoML**

**Soft AutoML: Hyperparameter tuning**

AutoML refers to automating the process of finding ML algorithms to solve real world problems.

With different sets of hyperparameters, the same model can give drastically different performances on the same dataset.

The goal of hyperparameter tuning is to find the optimal set of hyperparameters for a given model within a search space—the performance of each set evaluated on a validation set.

Popular methods for hyperparameter tuning include random search,26 grid search, and Bayesian optimization. Note that: sensitive hyperparameters should be more carefully tuned and use your validation data to tune them than using test split to prevent the overfitting on test data.

**Hard AutoML: Architecture search and learned optimizer**

Architectural search or neural architecture search (NAS) for neural networks, it searches for the optimal model architecture.

It is contained three components:

* *search space:* Defines possible architectures
* *performance estimation strategy: To evaluate the performance of a candidate architecture without having to train each candidate architecture from scratch until convergence.*
* *search strategy:* To explore the search space. A simple approach is random search—randomly (which is unpopular because its prohibitively expensive) like Reinforcement Learning: Rewards architectures with better performance and Evolutionary Algorithms: Mutates architectures, selects the best, and iterates.

Benefits of Hard AutoML: Auto-generated architectures like EfficientNet achieve state-of-the-art accuracy with up to 10x better efficiency, NAS and learned optimizers can generalize across multiple tasks, saving time and cost during production and it can Unlocking New Possibilities by Enables solving complex real-world problems that existing architectures and optimizers cannot handle.

But the challenges that it need a Training NAS or learned optimizers is computationally expensive, limiting it to a few companies (Google, OpenAI). And the Complexity Optimizers require careful tuning, and learned optimizers add another layer of complexity.

**Four Phases of ML Model Development**

there are four phases of adopting ML:

Phase 1. Before machine learning: start with non-ML solutions simplest solution with lowest accuracy like how Facebook started. (If you think that machine learning will give you a 100% boost, then a heuristic will get you 50% of the way there. You might even find that non-ML solutions work fine and you don’t need ML yet.)

Phase 2. Simplest machine learning models: For your first ML model, you want to start with a simple algorithm, something that gives you visibility into its working to allow you to validate the usefulness of your problem framing and your data.

Phase 3. Optimizing simple models: Once you have your ML framework in place, you can focus on optimizing the simple ML models with different objective functions, hyperparameter search, feature engineering, more data, and ensembles.

Phase 4. Complex models: Once you’ve reached the limit of your simple models and your use case demands significant model improvement, experiment with more complex models.

**Model Offline Evaluation**

Without clear evaluation metrics, it becomes challenging to compare models, find the best solution, and justify using ML to stakeholders.

Metrics like F1 scores or accuracy are meaningless without comparing them to relevant baselines.

**Baseline evaluation**

Continuously compare against baselines and refine based on both performance and practicality.

* Random Baseline: Predicts randomly.
* Simple Heuristic: Uses basic logic instead of ML.
* Zero Rule Baseline: Always predicts the most frequent class.
* Human Baseline: Compares model performance against human experts.
* Existing Solutions: Compares new ML models to current business logic or third-party solutions.

A “good” system may not always be “useful” (better than prior ML systems but worse than human benchmarks).

A “bad” system may still be “useful” if it offers tangible benefits like improved efficiency or lower costs.

**Evaluation Methods**

Perturbation tests

Test how models handle noisy inputs. Because a good model performs well even with noisy, real-world data. Sensitive models may degrade quickly with minor user changes.

Invariance tests

Ensure that certain input changes don’t affect outputs. So sensitive features (like gender) should often be excluded during training.

Directional expectation tests

Certain changes to the inputs should, cause predictable changes in outputs. (like houses price with area), If the outputs change in the opposite expected direction, your model might not be learning the right thing

Model Calibration

Ensure predicted probabilities match real-world outcomes. You can use tools to Measure calibration curves (scikit-learn) and Use Platt scaling or similar techniques to improve calibration.

To measure a model’s calibration, a simple method is counting: you count the number of times your model outputs the probability X and the frequency Y of that prediction coming true, and plot X against Y.

Confidence measurement

It is to show the predictions that your model is certain about. Avoid showing low-confidence predictions to users. Set thresholds for when to discard or request more input.

Slice-based evaluation

Evaluate the model performance across subsets of data. (slicing your dataset)

Imbalanced Slices: A model performs well on the majority group but poorly on the minority.

Critical Slices: Prioritize subsets like paid vs. free users in churn prediction.

Simpson’s Paradox: A model may perform better overall but worse in every individual slice.

Slicing is, unfortunately, still more of an art than a science, requiring intensive data

exploration and analysis.

Here are the three main approaches:

* Heuristics-Based: Use domain knowledge (e.g., device type or geography).
* Error Analysis: Review misclassified samples for patterns.
* Automated Tools: Use algorithms like beam search or clustering to find slices.