Lamalab Tool and Paper Notes

LamaLab

4/11/24

Table of contents

1 Tool and paper minutes

In our group seminars, we have a tradition of dedicating a few minutes to showcase tools/software/tricks/methods that we find useful. This repository is a collection of these tool minutes.

Part I

Tools

2 Hydra

2.1 Getting started

Hydra is an open-source Python framework that simplifies the development of research and other complex applications. The key feature is the ability to dynamically create a hierarchical configuration by composition and override it through config files and the command line. The name Hydra comes from its ability to run multiple similar jobs - much like a Hydra with multiple heads.

2.1.1 Key features:

- Hierarchical configuration composable from multiple sources
- Configuration can be specified or overridden from the command line
- Dynamic command line tab completion
- Run your application locally or launch it to run remotely
- Run multiple jobs with different arguments with a single command

2.1.2 Installation

```
pip install hydra-core --upgrade
```

2.1.3 Basic example

Config, e.g., in `conf/config.yaml`:

```
db:
driver: mysql
user: omry
pass: secret
```

3 IP Rotator

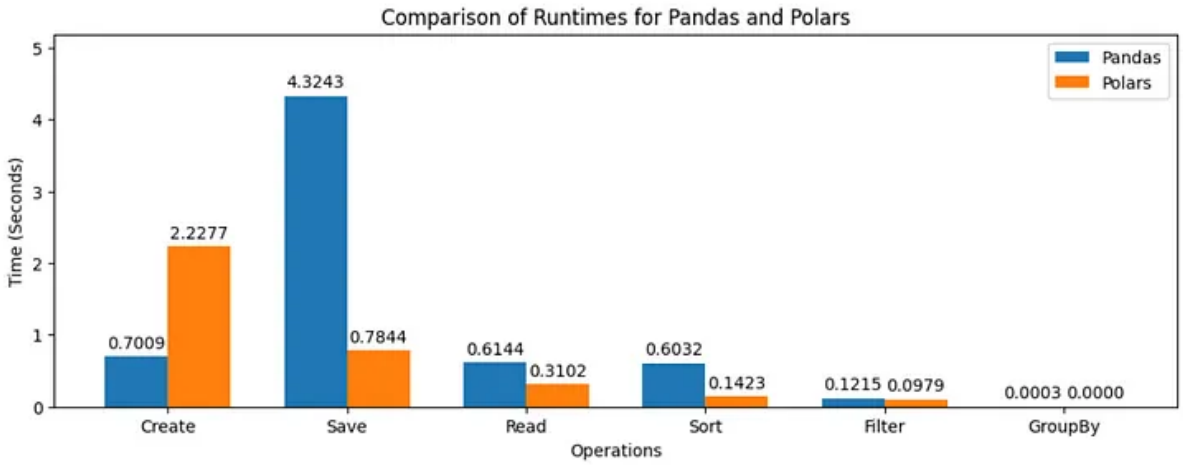
3.1 GitHub repository

iq-requests-rotator

3.1.1 Example usage:

```
import requests
from requests_ip_rotator import ApiGatewaywith ApiGateway("https://site.com") as g:
    session = requests.Session()
    session.mount("https://site.com", g)    response = session.get("https://site.com/index
    print(response.status_code)
```

4 Polars



4.1 An alternative to pandas

The advantages of polars can be directly seen in the image above. It is clear from the graph that Polars perform faster than Pandas for most operations. This is particularly true for the GroupBy operation, where Polars is nearly 20 times faster than Pandas. The Filter operation is also significantly faster in Polars, while Create operations are somewhat faster in Pandas. Overall, Polars seems to be a more performant library for data manipulation, particularly for large datasets.

4.2 Syntax example

```
import polars as pl

q = (
    pl.scan_csv("docs/data/iris.csv")
    .filter(pl.col("sepal_length") > 5)
    .group_by("species")
```

```
    .agg(pl.all().sum())
)

df = q.collect()
```

5 Thunder Client

Thunder Client is a lightweight alternative to Postman that can be used directly from VS-Code.

You can use it to test your API endpoints.

For an example, see this video.

5.1 Installation

Install the Thunder client extension from the marketplace.

6 tmux

`tmux` is a terminal multiplexer. It lets you switch easily between several programs in one terminal, detach them (they keep running in the background) and reattach them to a different terminal. And do a lot more.

6.1 Installation

```
sudo apt install tmux
```

or on Mac

```
brew install tmux
```

6.2 Usage

Let's assume you are via ssh on a remote server and you want to run a long running process. You can use `tmux` to run the process in a session and then detach from it. You can then log out and log back in later to check on the process. Your process will still be running, even if your ssh session is closed.

6.2.1 On the remote server

```
tmux new -s myprocess
```

Then run your process. When you are done, detach from the session by pressing `Ctrl+b` and then `d`.

6.2.2 On the remote server later

```
tmux ls
```

This will list all the sessions. You can then reattach to the session you want by typing:

```
tmux attach -t myprocess
```

6.2.3 Panes

You can split your terminal into panes. This is useful if you want to run multiple processes in the same terminal. You can split the terminal vertically by pressing `Ctrl+b` and then `"` or horizontally by pressing `Ctrl+b` and then `%`.

To move panes around, you can use `Ctrl+b` and then `o` to cycle through the panes.

7 Robust statistics and Trimean

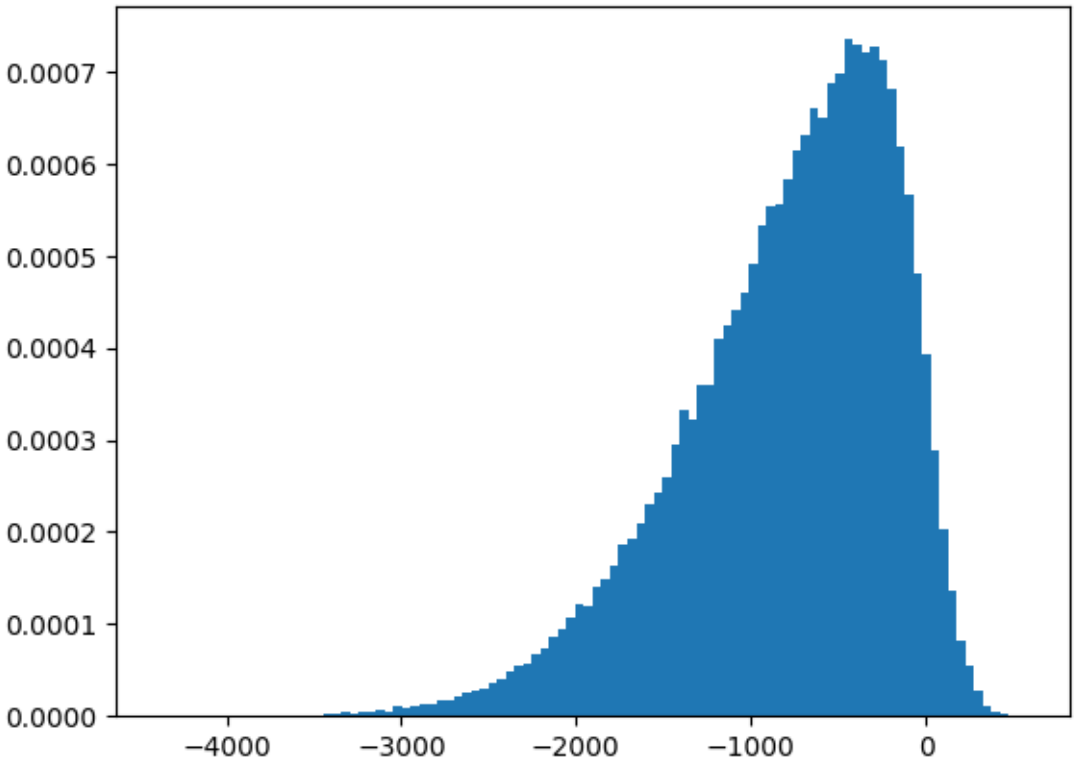
```
from scipy.stats import skewnorm
import numpy as np
import matplotlib.pyplot as plt
```

Let's generate some data that might be something we find in the real world.

```
skew_magnitude = -6
arr = skewnorm.rvs(skew_magnitude, loc=0, scale=1000, size=100000)
```

(The skew is a third-order moment.)

```
plt.hist(arr, bins=100, density=True)
plt.show()
```



Let's get a very common measure of central tendency:

```
np.mean(arr)
```

```
-789.5809069979605
```

The mean overstates the central tendency because of the skew.

The mean is defined as

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

and treats all numbers equally. No matter how big or small.

One can "fix" this by looking at "robust" statistics that are often rank based. Rank based means that we sort the data and then base our statistics on the rank of the data. In this way, they are no longer sensitive to outliers.

```python
def interquartile_range(arr):
    q1 = np.percentile(arr, 25)
    q3 = np.percentile(arr, 75)
    return q3 - q1

print("Median", np.percentile(arr, 50))
print("Interquartile Range", interquartile_range(arr))
print("Mean", arr.mean())
print("Standard Deviation", arr.std())
```

```
Median -679.7024551978025
Interquartile Range 834.2816858677052
Mean -789.5809069979605
Standard Deviation 614.9363837309692
```

A very nice measure of centrality is the so-called trimean.

> "An advantage of the trimean as a measure of the center (of a distribution) is that it combines the median's emphasis on center values with the midhinge's attention to the extremes."
>
> — Herbert F. Weisberg, Central Tendency and Variability

It is defined as

$$\text{trimean} = \frac{Q_1 + 2Q_2 + Q_3}{4}$$

where Q_1 is the first quartile, Q_2 is the median, and Q_3 is the third quartile.

```python
def trimean(arr):
    q1 = np.percentile(arr, 25)
    q3 = np.percentile(arr, 75)
    median = np.percentile(arr, 50)
    return (q1 + 2*median + q3)/4

print("Trimean", trimean(arr))
```

```
Trimean -708.4430042323374
```

8 Easy fast `.apply` for pandas

`apply` in `pandas` is slow. This is the case because it does not take advantage of vectorization. That means, in general, if you have something for which there is a built-in `pandas` (or `numpy`) function, you should use that instead of `apply`, because those functions will be optimized and typically vectorized.

The `pandarallel` package allows you to parallelize `apply` on a `pandas` `DataFrame` or `Series` object. It does this by using `multiprocessing`. However, since it uses multiple processes, it will use more memory than a simple `apply`.

If your data just barley fits in memory, you should not use `pandarallel`. However, if it does fit in memory, and you have a lot of cores, then `pandarallel` can speed up your code significantly with just changing one line of code.

```
from pandarallel import pandarallel

pandarallel.initialize(progress_bar=True)

# df.apply(func)
df.parallel_apply(func)
```

9 BFG Repo-Cleaner

If you did not take with your `.gitignore` or just used `git add .` you might have by accident committed large files. This might lead to an error like

```
remote: error: See https://gh.io/lfs for more information.
remote: error: File reports/gemini-pro/.langchain.db is 123.01 MB; this exceeds GitHub's file
remote: error: GH001: Large files detected. You may want to try Git Large File Storage - http
To github.com:lamalab-org/chem-bench.git
 ! [remote rejected]     kjappelbaum/issue258 -> kjappelbaum/issue258 (pre-receive hook decli
error: failed to push some refs to 'github.com:lamalab-org/chem-bench.git'
```

To fix this, you need to remove the large files. A convenient tool for doing this is BFG.

Once you download the file you can run it using something like

```
java -jar ~/Downloads/bfg-1.14.0.jar --strip-blobs-bigger-than 100M --no-blob-protection
```

to remove large files.

Note that this here uses `--no-blob-protection` as BFG defaults to not touching the last commit.

After the BFG run, it will prompt you to run something like

```
git reflog expire --expire=now --all && git gc --prune=now --aggressive
```

10 showyourwork

showyourwork : https://github.com/showyourwork is a framework for building reproducible papers. The package works on a combination of Tex and Python code, where you can on the fly modify your plots.

The pre-requisites are: 1. define a conda environment with the packages are that necessary for plotting 2. use the `\script{}`, `\variable{}` and other commands to link your figures/tables to a Python script. 3. compile the paper

Part II

Papers

11 Uncertainty-Aware Yield Prediction with Multimodal Molecular Features

11.1 Why discussing this paper?

I chose Chen et al.'s paper (Chen et al. 2024) for our journal club because

- An important and interesting problem in chemistry
- Uses many of the techniques we care about in our group

11.2 Context

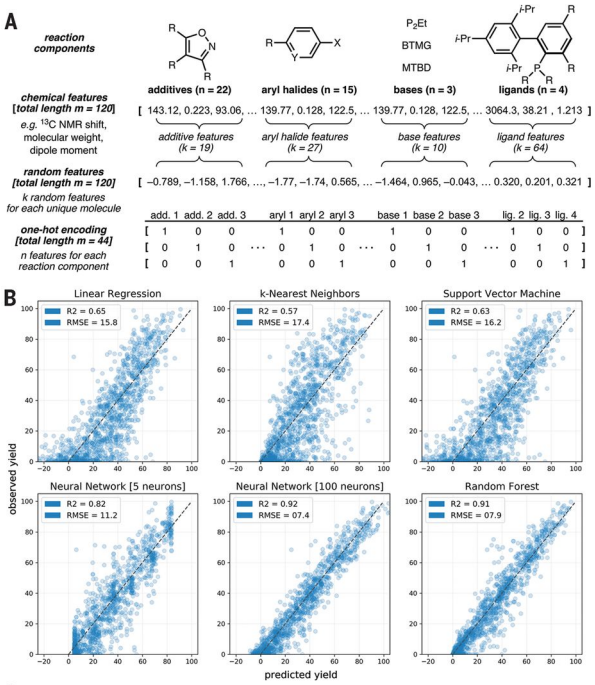
Predicting the yield of chemical reactions is a crucial task in organic chemistry. It can help to optimize the synthesis of new molecules, reduce the number of experiments needed, and save time and resources. However, predicting the yield of a reaction is challenging due to the complexity of chemical reactions and the large number of factors that can influence the outcome.

11.3 Prior work

11.3.1 Ahneman et al. (2018)

Ahneman et al. (Ahneman et al. 2018) reported in *Science* a random forest model that predicts the yield of chemical reactions in a high-throughput dataset (palladium-catalyzed Buchwald-Hartwig cross-coupling reactions). For this, the authors created a set of features using computational techniques.

A very interesting aspect of this work is the subsequent exchange with Chuang and Keiser (Chuang and Keiser 2018) who point out that the chemical features used in the work by Ahneman et al. perform not distinguishably better than non-meaningful features.



C

		Lin. Reg.	*k*-NN	SVM	NN	RF
Ahneman et al.	R^2	0.66	0.64	0.64	0.93	0.92
	RMSE	15.6	16.1	16.1	7.0	7.5
Random Features	R^2	0.65	0.57	0.63	0.92	0.91
	RMSE	15.8	17.4	16.2	7.4	7.9
One-Hot Encoded	R^2	0.65	0.45	0.66	0.93	0.90
	RMSE	15.8	19.8	15.5	7.1	8.6

Figure 11.1: Figure taken from Chuang and Keiser's paper(Chuang and Keiser 2018) illustrating models trained with various featurization approaches.

11.3.2 Schwaller et al. (2020, 2021)

Schwaller et al. (Schwaller et al. 2020, 2021) utilized BERT models with a regression head to predict yields based on reaction SMILES.

They observed multiple interesting effects:

- The performance on high-throughput datasets is good, on USPTO datasets the models are not predictive (R^2 on a random split of 0.117 for the gram scale)
- The yield distribution depends on the scale, which might be due to reaction at larger scale being better optimized

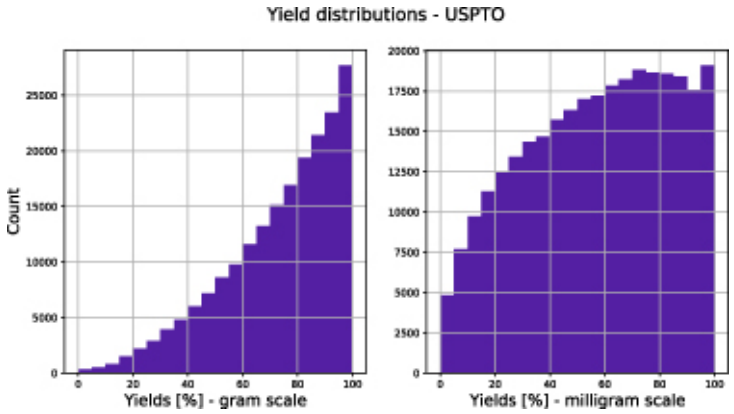


Figure 11.2: Figure taken from Schwaller et al. (Schwaller et al. 2021) illustrating the distribution of yields on different scales.

11.3.3 Kwon et al. (2022)

Kwon et al. (Kwon et al. 2022), in contrast, used graph neural networks to predict yields. They pass reactants and products through a graph neural network and concatenate the embeddings to predict the yield. They train on a similar loss as the work at hand (but use also use dropout Monte-Carlo (Gal and Ghahramani 2016) to estimate the epistemic uncertainty).

11.4 Problem setting

- prior works perform well on high-throughput datasets but not on real-world datasets
- this is partially due to a lot of noise in datasets
- of course, reaction conditions are important, too

Additionally, the authors propose that the previous representations might not be “rich” enough to capture the complexity of chemical reactions.

11.5 Approach

The authors propose to fuse multiple features. In addition, they also use a special loss function and a mixture of experts (MoE) model used to transform human-designed features.

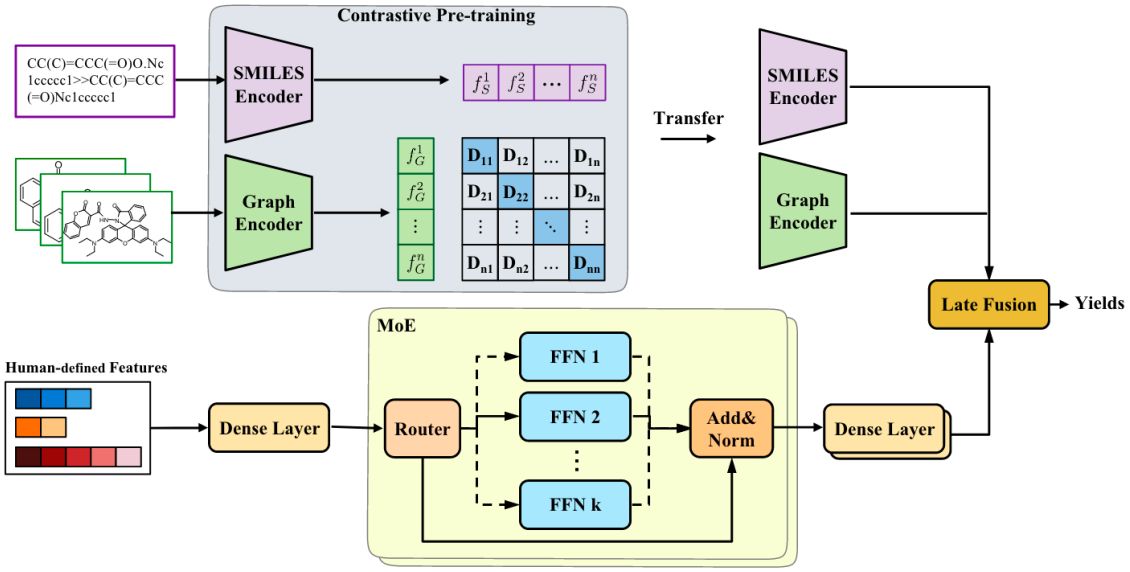


Figure 11.3: Overview of the model architecture. Figure taken from Chem et al. (Chen et al. 2024)

11.5.1 Graph encoder and SMILES encoder

The authors pretrain the graph and SMILES encoders using a contrastive loss. The graph encoder is a GNN, the SMILES encoder is a transformer.

11.5.1.1 Graph convolutional neural network

Their graph encoder is basically a message graph convolutional neural network. The authors use the `DGL` library to implement this.

The forward pass looks like this:

```
for _ in range(self.num_step_message_passing):
    node_feats = self.activation(self.gnn_layer(g, node_feats, edge_feats)).unsqueeze(0)
    node_feats, hidden_feats = self.gru(node_feats, hidden_feats)
```

```
    node_feats = node_feats.squeeze(0)
```

Where the GNN layer performs a simple operation such as

$$\mathbf{x}_i' = \Theta^\top \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \mathbf{x}_j$$

where $\hat{d}_i$ is the degree of node i and Θ is a learnable weight matrix. $\mathcal{N}(i)$ is the set of neighbors of node i. $\mathbf{x}_i$ is the node embedding of node i, $e_{j,i}$ is the edge feature between node i and j.

The node embeddings are then aggregated using Set2Set pooling (Vinyals, Bengio, and Kudlur 2016).

11.5.1.2 SMILES encoder

For encoding SMILES, the use a transformer model. In their code, they seem to pass through only one transformer layer.

The forward pass looks like this:

```
x = self.token_embedding(text)
x = x + self.positional_embedding
x = x.permute(1, 0, 2)  # NLD -> LND
x = self.transformer(x)
x = x.permute(1, 0, 2)  # LND -> NLD
x = self.ln_final(x)
x = self.pooler(x[:,0,:])
```

They take the first token of the sequence and pass it through a linear layer to get the final representation.

11.5.1.3 Contrastive training

The authors use a contrastive loss to train the encoders.

$$\mathcal{L}_c = -\frac{1}{2} \log \frac{e^{\langle f_G^j, f_S^j \rangle / \tau}}{\sum_{k=1}^N e^{\langle f_G^j, f_S^k \rangle / \tau}} - \frac{1}{2} \log \frac{e^{\langle f_G^j, f_S^j \rangle / \tau}}{\sum_{k=1}^N e^{\langle f_G^k, f_S^j \rangle / \tau}},$$

In contrastive training, we try to maximize the similarity between positive pairs and minimize the similarity between negative pairs. In the equation above, f_G^j and f_S^j are the representations of the graph and SMILES of the same reaction, respectively. τ is a temperature parameter.

Such contrastive training allows to pretrain the encoders on a large dataset without labels.

> **Note**
>
> Contrastive learning is one of the most popular methods in self-supervised learning. A good overview can be found in Lilian Weng's amazing blog.

11.5.2 Human-features encoder

The authors also encode additional features with feedforward networks in a mixture of experts (MoE) model. The key idea behind MoE is that we replace "conventional layers" with "MoE layers" which are copies of the same layer. A gating network decides, based on the input, which layer to use. This is powerful if we sparsely select the experts-then only a subset of all weights are used in a given forward pass.

$$\mathrm{MoE}\left(x_H\right) = \sum_{i=1}^{t} \mathcal{G}\left(x_H\right)_i \cdot E_i\left(x_H\right)$$

This is a mixture of experts model. The authors use a gating network $\mathcal{G}$ to decide which expert to use. The experts E_i are simple feedforward networks. The gating network might be a simple softmax layer:

$$G_\sigma(x) = \mathrm{Softmax}\left(x \cdot W_g\right)$$

in practice, one can improve that by adding sparsity (e.g. selecting top-k).

> **Note**
>
> MoE (Shazeer et al. 2017) has become popular recently as a way to scale LLMs. You might have across model names like Mixtral-8x7B (Jiang et al. 2024), which indicates that the model is a mixture of 8 experts, each of which is a 7B parameter model. The total number of parameters is 47B parameters, but the inference cost is similar to the one of a 14B parameter model. (Note however, that memory consumption is still high as all experts need to be loaded into memory.)
> This blog by Cameron Wolfe gives a good overview. You might also find Yannic Kilcher's video about Mixtral of Experts useful.

11.5.3 Fusion

The fusion of the different features is done by concatenating them

The complete forward pass looks like this:

```
r_graph_feats = torch.sum(torch.stack([self.clme.mpnn(mol) for mol in rmols]), 0)
p_graph_feats = self.clme.mpnn(pmols)
feats, a_loss = self.mlp(input_feats)
seq_feats = self.clme.transformer(smiles)
concat_feats = torch.cat([r_graph_feats, p_graph_feats, feats, seq_feats], 1)
out = self.predict(concat_feats)
```

where the `mpnn` method is the graph encoder, the `transformer` method is the SMILES encoder, and the `mlp` method is the human-features encoder.

11.5.4 Uncertainty (quantification)

The authors define the prediction as

$$\hat{y} = \mu(x) + \epsilon * \sigma(x)$$

where $\mu(x)$ is the prediction, $\sigma(x)$ is the uncertainty, and ϵ is a random variable sampled from a normal distribution.

The model is trained with a loss function that includes the uncertainty:

$$\mathcal{L}_u = \frac{1}{N}\sum_{i=1}^{N}\left[\frac{1}{\sigma\left(x_i\right)^2}\left\|y_i - \mu\left(x_i\right)\right\|^2 + \log\sigma\left(x_i\right)^2\right]$$

The σ term is capturing observation noise (aleatoric uncertainty).

> **Note**
>
> This loss comes from the idea of variational inference.
>
> $$\mathcal{L}(\lambda) = -\mathbb{E}_{q(\theta;\lambda)}[\log p(\mathbf{y} \mid \mathbf{x}, \theta)] + \mathrm{KL}(q(\theta;\lambda)\|p(\theta))$$
>
> In this equation, the first term is the negative log-likelihood, and the second term is the KL divergence between the approximate posterior $q(\theta;\lambda)$ and the prior $p(\theta)$. The KL divergence is a measure of how much the approximate posterior diverges from the prior. The idea is to minimize the negative log-likelihood while keeping the approximate

posterior close to the prior. This is a way to quantify the uncertainty in the model. The idea comes from Bayesian inference, where we want to estimate the posterior distribution over the parameters of the model. In practice, this is intractable, so we use variational inference to approximate the posterior with a simpler distribution. The posterior (which quantifies uncertainty) is typically computationally expensive to compute, so we use variational inference to approximate it with a simpler distribution, this is called variational inference. Since during training, we do some sampling, we need to perform a reparametrization trick (Kingma, Salimans, and Welling 2015) to make the gradients flow through the sampling operation.

11.6 Results

As in most ML papers, we have tables with bold numbers, e.g. for a dataset with amide coupling reactions:

Model	MAE ↓	RMSE ↓	R^2 ↑
Mordred	15.99 ± 0.14	21.08 ± 0.16	0.168 ± 0.010
YieldBert	16.52 ± 0.20	21.12 ± 0.13	0.172 ± 0.016
YieldGNN	$\underline{15.27 \pm 0.18}$	$\underline{19.82} \pm 0.08$	$\underline{0.216} \pm 0.013$
MPNN	16.31 ± 0.22	20.86 ± 0.27	0.188 ± 0.021
Ours	$\mathbf{14.76 \pm 0.15}$	$\mathbf{19.33 \pm 0.10}$	$\mathbf{0.262 \pm 0.009}$

Here, their model outperforms the baselines. But it is also interesting to see how well the Mordred baseline performs compared to much more complex models.

The pattern of their model being bold in tables is persistent across datasets.

11.6.1 Ablations

The authors perform ablations to understand the importance of the different components of their model. While there are some differences, the differences are not drastic (partially overlapping errorbars).

Model	MAE ↓	RMSE ↓	R^2 ↑
Ours	14.76 ± 0.15	19.33 ± 0.10	0.262 ± 0.009
w/o UQ	15.08 ± 0.13	19.63 ± 0.09	0.249 ± 0.009
w/o $\mathcal{L}_r$	14.80 ± 0.16	19.51 ± 0.10	0.261 ± 0.010
w/o MoE	15.12 ± 0.18	20.03 ± 0.13	0.230 ± 0.012

Model	MAE ↓	RMSE ↓	R^2 ↑
w/o Seq.	14.97 ± 0.16	19.55 ± 0.11	0.261 ± 0.010
w/o Graph	15.06 ± 0.15	19.59 ± 0.10	0.260 ± 0.009
w/o H.	15.83 ± 0.20	20.46 ± 0.18	0.212 ± 0.016

11.7 Take aways

- A lot of machinery, but not a drastic improvement
- It is the data, stupid! (It is not really clear how this is even supposed to work with information about the conditions)
- Interestingly, they didn't test USPTO or other datasets
- Their approach with frozen encoders is interesting, it would have been interesting to see learning curves to better understand the data efficiency of the approach

11.8 References

Ahneman, Derek T., Jesús G. Estrada, Shishi Lin, Spencer D. Dreher, and Abigail G. Doyle. 2018. "Predicting Reaction Performance in c–n Cross-Coupling Using Machine Learning." *Science* 360 (6385): 186–90. https://doi.org/10.1126/science.aar5169.

Chen, Jiayuan, Kehan Guo, Zhen Liu, Olexandr Isayev, and Xiangliang Zhang. 2024. "Uncertainty-Aware Yield Prediction with Multimodal Molecular Features." *Proceedings of the AAAI Conference on Artificial Intelligence* 38 (8): 8274–82. https://doi.org/10.1609/aaai.v38i8.28668.

Chuang, Kangway V., and Michael J. Keiser. 2018. "Comment on 'Predicting Reaction Performance in c–n Cross-Coupling Using Machine Learning'." *Science* 362 (6416). https://doi.org/10.1126/science.aat8603.

Gal, Yarin, and Zoubin Ghahramani. 2016. "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning." In *International Conference on Machine Learning*, 1050–59. PMLR.

Jiang, Albert Q., Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, et al. 2024. "Mixtral of Experts." https://arxiv.org/abs/2401.04088.

Kingma, Diederik P., Tim Salimans, and Max Welling. 2015. "Variational Dropout and the Local Reparameterization Trick." https://arxiv.org/abs/1506.02557.

Kwon, Youngchun, Dongseon Lee, Youn-Suk Choi, and Seokho Kang. 2022. "Uncertainty-Aware Prediction of Chemical Reaction Yields with Graph Neural Networks." *Journal of Cheminformatics* 14 (1). https://doi.org/10.1186/s13321-021-00579-z.

Schwaller, Philippe, Alain C Vaucher, Teodoro Laino, and Jean-Louis Reymond. 2020. "Data Augmentation Strategies to Improve Reaction Yield Predictions and Estimate Uncertainty."

Chemrxiv Preprint.

———. 2021. "Prediction of Chemical Reaction Yields Using Deep Learning." *Machine Learning: Science and Technology* 2 (1): 015016.

Shazeer, Noam, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer." https://arxiv.org/abs/1701.06538.

Vinyals, Oriol, Samy Bengio, and Manjunath Kudlur. 2016. "Order Matters: Sequence to Sequence for Sets." https://arxiv.org/abs/1511.06391.