Decoder based Regression with non-conventional numerical representations

Animesh Madaan
Indian Institute of Technology Kanpur
manimesh22@iitk.ac.in

Aujasvit Datta
Indian Institute of Technology Kanpur
aujasvitd22@iitk.ac.in

Sankalp Mittal
Indian Institute of Technology Kanpur
sankalpm22@iitk.ac.in

Viral Chitlangia
Indian Institute of Technology Kanpur
viralc22@iitk.ac.in

1 Problem Motivation

Decoding-based regression is a paradigm that treats numerical prediction as a sequence decoding task instead of directly generating a continuous value. In this, the numerical target is seen as a sequence of discrete tokens (eg: bits or digits) that represent the numeric target, much like how language models generate text. For example, a number like 1.75 might be generated as the token sequence <1><.><7><5> in digits or <1><.><1><1> in binary bits. This stands in direct contrast to traditional regression heads which output a single numeric tensor in a single go [8].

The motivation for using decoding-based regression is that it offers greater flexibility in modeling arbitrary outcome distributions without assuming a specific parametric form. Because the output is generated token by token, the model can in principle approximate any continuous distributions using a finite number of discrete bins, similar to how adding more digits yields higher precision. Another motivation comes from the recent observations that large language models can perform surprisingly well on regression tasks when numbers are represented as text. Prior work demonstrated that LLM services like ChatGPT and Gemini can achieve regression accuracy on par with classical methods (e.g. random forests) by reasoning over in-context examples and outputting answers in numeric token form. [8] formalized this idea of decoding-based regression, providing theoretical support and experimental evidence for its effectiveness over a variety of tasks.

Despite these advantages, decoding-based regression introduces new challenges and questions. One key issue is the choice of numerical representation or tokenization schemes (deciding how number should be represented as tokens) and their impact on model performance. For example, representing numbers in a higher base (eg: decimal digits) would produce shorter token sequences for the same precision (hence saving computational power) but increase the token alphabet size, making it harder for the model to learn each symbol. The original work primarily modeled numerical outputs as streams of binary bits. The motivation of our project is to investigate these alternative representations of numbers (beyond binary and decimal representations) such as p-adic expansions or the Stern-Brocot tree in the decoding-based regression framework to see if they offer performance benefits or new insights.

2 Past work

2.1 Regression

In classical machine learning, regression is almost always done with numeric outputs: models produce a real number (or a set of numbers) directly, and training optimizes a regression loss like mean squared error. Common approaches include linear regression, decision trees, and neural networks with a

dedicated output neuron (or head) for the target. These traditional regressors typically assume a single-point production or rely on predefined distributional forms and try to learns its parameters (for example, it may assume the output distribution to be a gaussian and try to learn its mean or variance) Such pointwise or parametric approaches can struggle to capture complex or arbitrary outcome distributions.

2.2 Text-to-text regression

While large language models are trained to generate text, they have been inadvertently applied to regression tasks by encoding numerical inputs and outputs as text and essentially treating regression as a special case of language generation. [10] showed that prompting service LLMs like ChatGPT and Gemini with a few in-context examples enbles them to perform regression reasonably well. In their work, models like ChatGPT were given textual descriptions of input features and asked to output a number in textual form – effectively leveraging the language model's capabilities to produce a number. An example of the prompt they used is given below. Similarly, other researchers explored fine-tuning or adapting LLMs for tasks that involve numbers: for instance, training language models on mathematical problem sets scientific data where numeric values appear as tokens[7]. These studies demonstrated that using tokens to represent numbers is feasible; however, they often treat the problem as text generation with no handling for numerical accuracy.

```
The task is to provide your best estimate for "Output". Please
provide that and only that, without any additional text.

Feature 0: -2.06
Output: -81.93

Feature 0: -0.64
Output: -25.33

Feature 0: 1.62
Output:
```

Figure 1: Example of prompt used to create textual descriptions of input features for regression problems in [10]

One limitation noted is that standard cross entropy does not penalize near-miss numeric errors appropriately. For example, if the target value was 1.200 and the model's output was 1.199, the penalty would be high because the last three digits are different (and thus cross entropy loss would be high) even though the output is very close to the target. This highlights that while language models can output numbers, they lack an inherent notion of numeric distance under the usual training, which is an important consideration for decoding-based regression models as well. [11] addressed this by proposing a regression-like loss on number tokens to encourage numerical closeness.

2.3 Hybrid approaches (text-to-anything)

Some prior efforts combined text and traditional regression in hybrid architectures. [9] attached the learned text embeddings of inputs from an LLM to a classic numeric head to perform regression, effectively using an LLM for feature extraction and a standard regressor for output. Similarly, [6] used a vector-to-text regression hybrid through an in-context neural process. However, there had been no comprehensive study of the pure "anything-to-text" case until [8], which is the basis of our work.

2.4 Decoding-based Regression

[8] proposed modeling regression targets as sequences of discrete tokens, such as binary digits, predicted using a causal Transformer decoder. The model is trained with cross-entropy loss at each step and defines a probability distribution over real-valued outputs through these token sequences.

They proved this method is universal can approximate any continous distribution and demonstrated competitive performance on standard regression tasks, especially when output distributions are complex or multimodal.

3 Proposed Method

Building on the work of [8], we implemented a few other tokenizers other than the already implemented **Normalized** and **Unnormalized** Tokenizers. We added the following tokenizers -

- **p-adic Tokenizers[3]:** Using the really interesting representation of decimal numbers from Number Theory we implemented the p-adic tokenizer, where p is any prime number. This representation instead of representing numbers as an infinite string of numbers towards the right, represents them as an infinite string towards the left. Unfortunately, this representation works only for numbers in the range $(-1, 1)$, so we chose a hybrid approach by tokenizing the integral and fractional parts of the number separately.
  For the integral part we just took the representation of the number in base p.
  For the fractional part we converted the number into its p-adic format using the standard conversion. Let $x \in \mathbb{Q} \cap (0, 1)$. Its p-adic expansion can be written as:
  $$x = \sum_{n=N}^{\infty} a_n p^n, \quad \text{with } a_n \in \{0, 1, \ldots, p-1\}, \quad N < 0$$
  where the coefficients a_n are computed recursively using a p-adic division algorithm.
  One thing that is different in this approach than the approach used in the original paper is that in the paper [8] they used fixed length representation using the **IEEE Format** representation for numbers, we have actually used a variable-length encoding for the numbers which allows the model to be flexible depending on the length of the numbers involved this makes it incredibly efficient for small numbers while for large numbers the number of parameters might become too large.
- **Stern-Brocot Tree Tokenizer[5]:** Stern-brocot trees are an incredibly useful structure used in number theory and CS to generate all positive rational numbers, they give us a way to enumerate all rationals as a sequence of left or right branches of the tree.

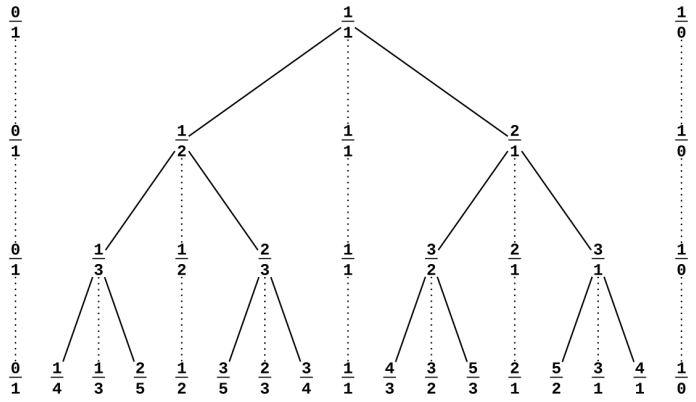

Figure 2: Stern Brocot Tree example

Since the tree can only enumerate all the positive rationals we used an extra token to show that sign of our output. Notice that the number of numbers that can be represented increases exponentially with the depth so we can use pretty shallow depth trees to represent a lot of inputs. It doesn't require normalised inputs and can work with both normalized and unnormalized inputs. also, it has only 2 possible tokens, which makes learning easy for the model.

Both of these methods can be used to efficiently approximate numbers depending on what we keep as their length as this also gives us an additional advantage of using these tokenizers.

4 Experiments and Evaluation

The code for this project can be found on GitHub.

4.1 Regression

All of the experiments used the same configuration for the encoder and decoder. We used the same configuration across all files for consistency. The following hyperparameters were used:

- **Encoder:**
  - 2 dense layers with 512 variables and *relu* activation
  - 1 dense output layer with 512 variables
- **Decoder:**
  - 2 transformer layers
  - 4 attention heads
  - 256 hidden variables for each attention layer
  - 0.1 dropout rate
- **Training:** We used the following parameters for training out model
  - 50 epochs of training for 4 iterations.
- **Data:** We got the data from UCI Datasets[2] - *Challenger*, *Gas*, *Housing*, *Autos*, *Fertility* and *Yacht*.

We had to limit our training time and model complexity due to time limitation, but still got a significant improvement over the Unnormalized Tokenization by using p-adic Tokenization. Here, **UD** refers to Unnormalized Tokenization.

Table 1: Negative log-likelihood (NLL) on UCI regression datasets (mean ± std over runs).

Dataset	NLL(UD)	NLL(p-adic)	NLL from the paper(UD)
Challenger	0.5705 ± 0.4374	0.3678 ± 0.3991	0.14 ± 0.14
Gas	0.1927 ± 0.1434	0.1831 ± 0.1369	0.02 ± 0.01
Housing	1.1531 ± 0.1981	0.75001 ± 0.2360	0.41 ± 0.03
Autos	1.6842 ± 0.2361	0.9657 ± 0.2464	0.48 ± 0.05
Fertility	0.7394 ± 0.2942	0.4677 ± 0.1857	0.31 ± 0.09
Yacht	1.4087 ± 0.0981	1.0622 ± 0.0631	0.39 ± 0.02

4.2 Density Estimation

All of the experiments used the same configuration for the encoder and decoder. We used the same configuration across all files for consistency. The following hyperparameters were used:

- **Encoder:**
  - 1 dense layer with 256 variables and *relu* activation
  - 1 dense output layer with 256 variables
- **Decoder:**
  - 2 transformer layers
  - 4 attention heads
  - 256 hidden variables for each attention layer
  - 0.1 dropout rate
- **Training:** We used the following parameters for training out model
  - 100 epochs of training
  - 5000 training points generated using a custom function for each shape.

- **Generation:** For density estimation we generated 3000 points to check how much the density overlapped. We used x as a latent variable which is generated randomly uniformly from a fixed range on the basis of which we see how well our model has learnt the distribution $p(y \mid x)$.

We chose these configurations taking into consideration training time constraints as we did not have a lot of computing power available to use. The model complexity is not as high as used in the seed paper [8].
The following shapes and patterns were used these have been listed in increasing order of complexity to estimate.

Zig Zag lines

For the **Unnormalized tokenizer** implemented in the seed paper [8] using our training configuration the following plot was obtained as in Fig 3

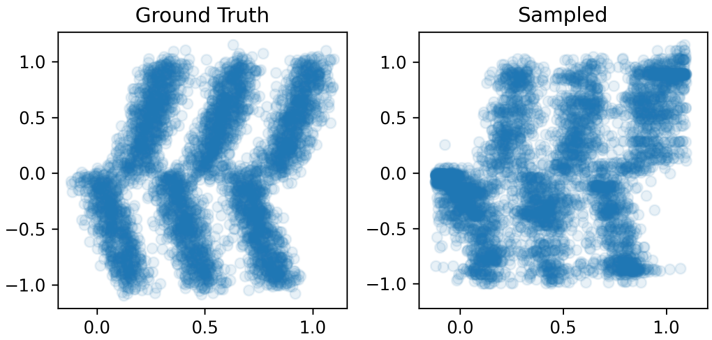

Figure 3: Zig Zag lines density estimation with Unnormalized tokenizer

The plot for the p**-adic tokenizer** is as in Fig 4

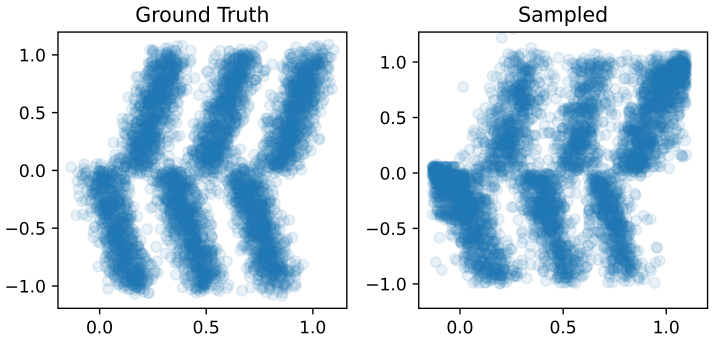

Figure 4: Zig Zag lines density estimation with p-adic tokenizer

The plot for the **SternBrocot Tokenizer** is as in Fig 5
Notice that in the *Stern Brocot Tokenizer* density estimation 5, there are bands of y that have no point in them. This is because the depth of the tree was not set high enough, so certain ranges of numbers could not be represented at all.
This was the simplest shape used for our experiments, so as expected, all the models gave decent results for this.

Hollow Square

The following plot for the p**-adic tokenizer** to estimate the hollow square shape was obtained as in Fig 6

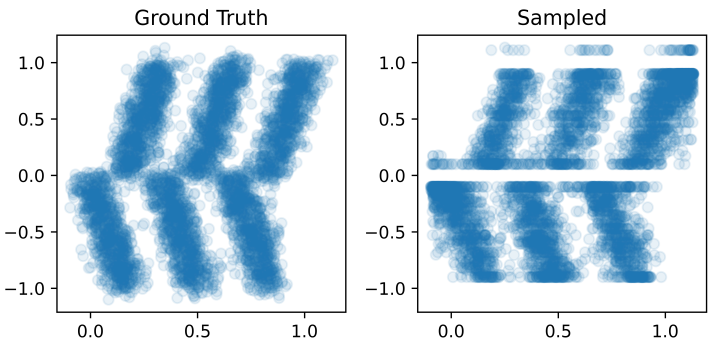

Figure 5: Zig Zag lines density estimation with Stern Brocot Tree tokenizer

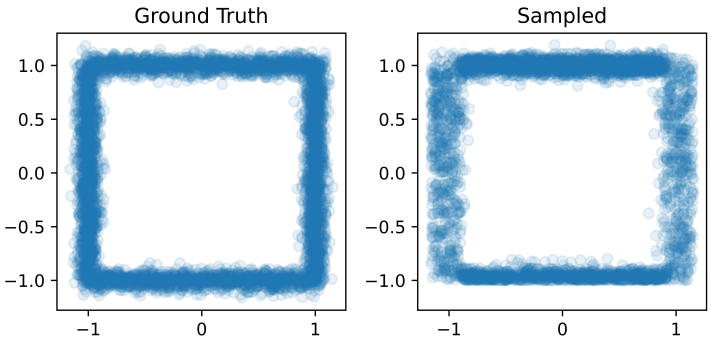

Figure 6: Hollow Square density estimation with p-adic tokenizer

The plot for the **SternBrocot Tokenizer** is as in Fig 7

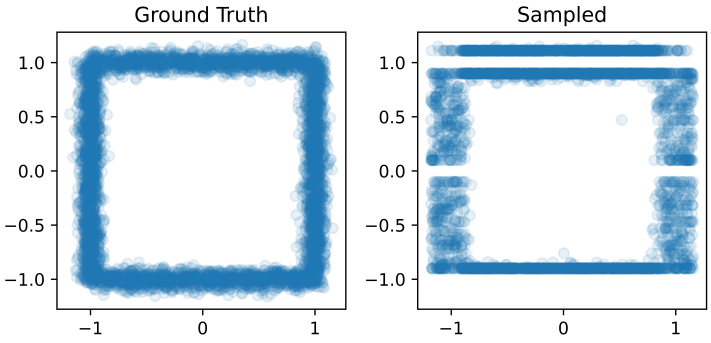

Figure 7: Hollow Square density estimation with Stern Brocot Tree tokenizer

Again we can see that the *Stern Brocot Tree tokenizer* gives empty bands. As expected, since this shape is not very complicated, the estimates are decent.

Half Moons

This is the first actually complicated shape and uses curves and not just straight lines, so it might be hard for our tokenizers to learn; let's see how It performs.
For the p**-adic Tokenizer** the following plot is obtained in Fig 8
For the **SternBrocot Tokenizer** the following plot is obtained in Fig 9
Again we can see that the *Stern Brocot Tree tokenizer* gives empty bands.

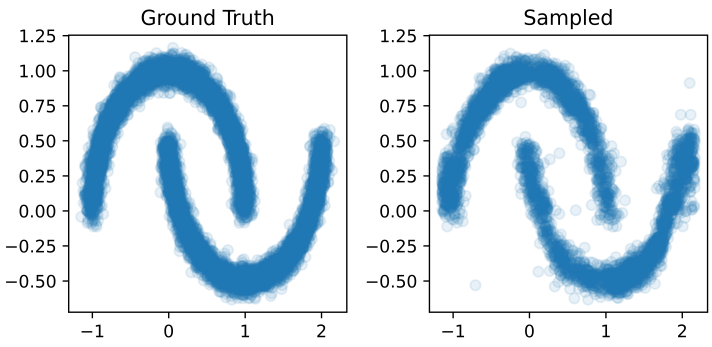

Figure 8: Half Moons density estimation with p-adic tokenizer

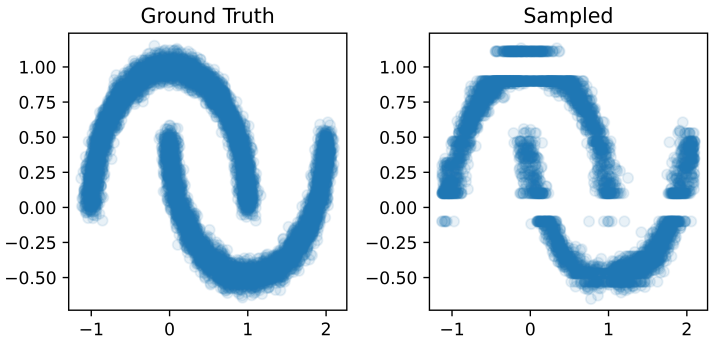

Figure 9: Half Moons density estimation with Stern Brocot Tree tokenizer

Since this shape is a little complicated, the estimates are decent but there is some irregularities in the estimation of the *p-adic Tokenizer* and a lot of outliers can be seen, these are not present in the *Stern Brocot Tokenizer* suggesting good estimates if a higher depth is used.

Spiral

This is the most complicated shape of all we have used let us first look how the **Unnormalized Tokenizer** does on this using our configuration so that we have a benchmark for seeing how our model does in complex settings. This is obtained in Fig 10

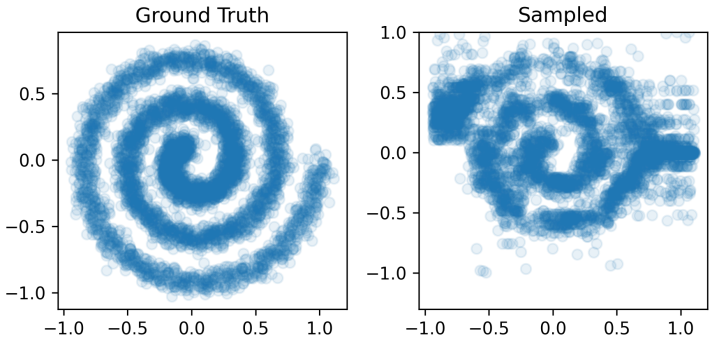

Figure 10: Spiral density estimation with Unnormalized tokenizer

The plot for the p**-adic tokenizer** is in Fig 11
The plot for the **SternBrocot Tokenizer** is in Fig 12 As we can clearly see the *Unnormalized*

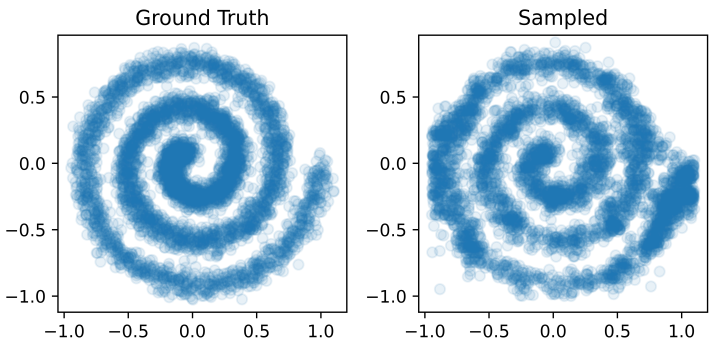

Figure 11: Spiral density estimation with p-adic tokenizer

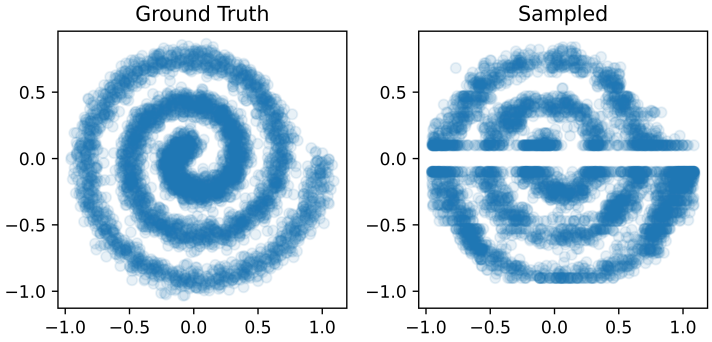

Figure 12: Spiral density estimation with Stern Brocot Tree tokenizer

Decoder performs terribly 10 in this case and just cannot access the last arm of the spiral, meanwhile our tokenizers have no such issue and give really good results, especially for the *p-adic Tokenizer* suggesting that it is really good for such task. Even the *Stern Brocot Tokenizer* only has the issue that it is giving the empty bands but it is efficiently able to learn the shape of the figure.

As we can clearly see, these density estimation experiments show that our new tokenization schemes are worthy contenders to be actually used in practice, as they are giving comparable results to the ones found by the authors [8] with much fewer training resources and time. It remains to be explored how the efficiency of these tokenizers is affected if we allow for increased model complexity and harder evaluation tasks.

5 Description of Tools/Software used

The code for this project was written in Python, and the initial codebase was available on GitHub. Tensorflow[1] was used for building the Attention based Transformer model. We used libraries like JaxTyping[4], fractions and typing to manipulate the numbers and string data for tokenization. We used those libraries to create tokenization functions for p-adic Tokenization and Stern Brocot Tokenization.

6 Learnings from the Project

Throughout the project, we gained a deeper understanding of how regression tasks can be framed as sequence modeling problems using tokenized representations of numbers. We learned about non-standard numerical representations such as p-adic expansions and the Stern-Brocot tree, and how such theoretical constructs can be used to improve machine learning models.

From an implementation perspective, we became familiar with setting up Transformer-based encoder-decoder architectures using TensorFlow. We also developed practical experience in dataset preparation, model training, and experiment tracking and logging.

7 Future Work

We can work on integrating self-length adjusting tokenization schemes depending on the output to this model, along with using IEEE type tokenization schemes for p-adic numbers to actually make the sequences fixed length. This is in addition to the future work already mentioned in the seed paper [8]. Numerous ways to extend decoding-based regression include improved tokenization schemes or other basis distributions besides piecewise constants. Further applications exist in other domains such as computer vision, where the encoder may be a convolutional network, or for multi-target regression, where the regressor needs to predict multiple different y-value targets

8 Member Contribution

The following were the member contributions

- **Animesh Madaan**
  - Wrote the initial density estimation code from scratch
  - Performed the regression experiments, aggregated the results and did a comparative analysis
  - Contributed to the regression experiments and result interpretation parts of the report
- **Aujasvit Datta**
  - Helped in setting up initial codebase and bringing it to working condition
  - Performed literature review to understand the past work done in the area of decoding regression
  - Wrote the initial draft of the report
- **Sankalp Mittal**
  - Created the Stern Brocot Tokenizer
  - Ran the experiments for Density Estimation and writing code to run the model on these experiments
  - Plotting the figures from the generated data
- **Viral Chitlangia**
  - Created the p-adic tokenizer
  - Ran a few experiments for Regression
  - Contributed to the Regression Analysis part of the Report

References

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). Tensorflow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[2] Evans, T. (2021). Uci datasets. `https://github.com/treforevans/uci_datasets`.

[3] Gouvêa, F. Q. (1997). *p-adic Numbers: An Introduction*. Universitext. Springer, 2nd edition.

[4] Kidger, P. (2023). Jaxtyping: Numpy-style static type checking for jax, pytorch, and numpy. `https://github.com/patrick-kidger/jaxtyping`. Version 0.2.23.

[5] Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 3rd edition. See section on mediants and the Stern–Brocot tree.

[6] Nguyen, T., Zhang, Q., Yang, B., Lee, C., Bornschein, J., Miao, Y., Perel, S., Chen, Y., and Song, X. (2024). Predicting from strings: Language model embeddings for bayesian optimization.

[7] Nogueira, R., Jiang, Z., and Lin, J. (2021). Investigating the limitations of transformers with simple arithmetic tasks.

[8] Song, X. and Bahri, D. (2025). Decoding-based regression.

[9] Tang, E., Yang, B., and Song, X. (2025). Understanding llm embeddings for regression.

[10] Vacareanu, R., Negru, V.-A., Suciu, V., and Surdeanu, M. (2024). From words to numbers: Your large language model is secretly a capable regressor when given in-context examples.

[11] Zausinger, J., Pennig, L., Chlodny, K., Limbach, V., Ketteler, A., Prein, T., Singh, V. M., Danziger, M. M., and Born, J. (2024). Regress, don't guess – a regression-like loss on number tokens for language models.