
Appendix for “Personalized Benchmarking with the Ludwig Benchmarking Toolkit”

A Appendix

A.1 Metrics

LBT reports a wide-range of performance metrics and training metadata for all experiments run using LBT. All metrics recorded at training time fall into the following categories: model metadata, hardware information, inference efficiency, fiscal cost, training efficiency, performance and carbon footprint. For carbon footprint tracking we use the Python experiment-impact-tracker package [14]. In Table A.1, we provide a more detailed overview of all metrics currently supported by LBT. The performance metrics listed represent a small subset of the performance metrics reported in LBT experiments and we refer readers to the Ludwig source code for a more detailed list: https://github.com/ludwig-ai/ludwig/blob/master/ludwig/utils/metrics_utils.py

Table A.1: **LBT Pre-Computed Metrics.** This table lists the off-the-shelf metrics and metadata reported for any experiments run using LBT.

Model Metadata	Model size (bytes)
Hardware Info	Total CPU cores, Total GPUs, GPU Type, System Memory
Inference Efficiency	Inference time (s)
Cost	Total training cost (\$)
Training Efficiency	Time per training step (s), Total training time (s)
Carbon Footprint	Power consumption (kWh), Carbon impact (Kg)
Performance	Accuracy, Precision, Recall, F1, Jaccard Index, Sensitivity, Specificity, AUC, and many more

Metric Details.

- Hardware information: system information is collected using the psutil [58] and GPUtil python modules [59].
- Inference time (s): computed as the average time (as measured using the Python date-time module) for a model to perform inference on one data sample. The average time is approximated using 25 random samples from the test set.
- Total training cost (\$): measured as the total dollar spend per trained model. This value is computed by using the average hourly public cloud instance prices for the machines in the experiments.
- Time per training step (s): measured as the average time to train one batch in a given epoch.
- Total training time (s): measured as the total time to train the model. The value we report is the `time_total_s` metric reported by Ray Tune.
- Power consumption (kWh), Carbon impact (Kg): data for these metrics is collected using the experiment-impact-tracker python package [14].

Custom Metrics. To add a custom metric, a user needs to register a new LBT metric as shown in Figure A.1 (top).

A.2 Evaluation Tools

LBT integrates with two open-source evaluation tools for measuring subpopulation based performance and robustness: Robustness Gym (RG) and TextAttack.

```

@register_metric("training_speed")
class TrainingSpeed(BaseMetric):
    def run(cls, dataset_path: str, model_dir: str):
        ...
        return training_speed

@register_dataset("new_dataset")
class NewDataset(LBTDataset):
    def __init__(self, cache_dir):
        self.name = "new_dataset"
        self.file_name = "new_dataset.csv"
        self.cache_dir = cache_dir
    def download(self):
        ...
    def process(self):
        ...
    def load(self):
        return pandas.read_csv(os.path.join(self.cache_dir, self.file_name))

```

Figure A.1: **User-defined metric and dataset.** Example of how custom metrics and datasets are added to LBT experiments.

RG API. LBT’s integration with RG lets users inspect model performance on a set of pre-built data subpopulations (e.g., sentence length, positive words, negative words, gender bias pairs, entity, POS, etc.) as well as add more subpopulations for their data and use cases. Defining a new subpopulation requires implementing and registering a new BaseSubpopulation class. Figure A.2 shows how to define a new data subpopulation that contains all samples with naughty and obscene words. Figure A.3 shows how the RG API is used.

```

@register_lbtsubpop("naughty_and_obscene")
class NaughtyObsceneSubpopulation(BaseSubpopulation):
    def __init__(self):
        self.name = "naughty_and_obscene"
    def score_fn(self, batch, columns):
        ...
    def get_naughty_obscene_word_list(self):
        ...
    def get_subpops(self, spacy):
        return [
            HasAnyPhrase(
                phrase_groups=[self.get_naughty_obscene_word_list()],
                identifiers=[Identifier("Naughty and Obscene Words")],
            )
        ]

```

Figure A.2: **User-define RG subpopulation.** Example of how a custom RG subpopulation is defined in LBT. The example identifies data subpopulations that contain naughty and obscene words.

TextAttack API. LBT’s integration with TextAttack helps LBT users evaluate model robustness to input perturbations. The integration can be used to generate adversarial attacks against models trained in LBT. Moreover, users can use the TextAttack interface to augment data files. Figure A.3 demonstrates how LBT’s TextAttack API is used.

A.3 Off-the-shelf Models

The model architectures currently supported in the toolkit include RNNs, CNNs (both stacked and parallel architectures), LSTMs [41], Transformers [60], TabNet [61], ResNet [62], MLP-Mixer [63], Vision Transformer (ViT) [64], most of the pretrained language language models available in HuggingFace such as BERT [36], RoBERTa [39], ELECTRA [38], DistilBERT [37], XLNet [65], T5 [40], XLM [66], GPT [67], GPT-2 [68], ALBERT [69], FlauBERT [70], CamemBERT [71], CTRL [72], XLM-RoBERTa [73] and Longformer [74]. Because of Ludwig’s extensibility, adding an additional model is relatively easy. Instructions for adding new models can be found here:

https://ludwig-ai.github.io/ludwig-docs/developer_guide/add_an_encoder/

```

from lbt.tools.robustnessgym import RG
from lbt.tools.textattack import attack, augment

# Robustness Gym API Usage
RG(dataset_name="AGNews", models=["BERT", "RNN"], path_to_dataset="agnews.csv",
    subpopulations=["length", "entities", "positive_words", "negative_words"]))

# TextAttack API Usage
attack(dataset_name="AGNews", path_to_model="agnews/model/rnn_model",
    path_to_dataset="agnews.csv", attack_recipe=["CharSwapAugmenter", "TextFooler"])

augment(dataset_name="AGNews", transformations_per_example=1
    path_to_dataset="agnews.csv", augmenter=["WordNetAugmenter"])

```

Figure A.3: **Sample API usage of evaluation tools.** Example of how to use the TextAttack API to generate adversarial attacks and augment data. The figure also demonstrates how to use the RG API to evaluate model performance on various data subpopulations.

A.4 Text Classification Case Study: Additional Experimental Details

For the purposes of replication, all experiment configuration files used in the study can be found at the link below. Each configuration file contains the hyperparameter search space, training parameters and model specific parameters used for each model and dataset hyperparameter optimization search. Replicating an experiment given one such configuration files requires setting up LBT, and running the following command: `python experiment_driver.py -reproduce <path to experiment config file>`

Configuration files:

https://drive.google.com/drive/folders/1CdziTuzPHUUnRlcMzJ0EpMspMI_67Era?usp=sharing

Datasets. For the MGB dataset, the results displayed are for the Wizard subset. For the SBF dataset, we report the accuracy for the intent classification task.

Hyperparameters. For each dataset, we set the batch size to be the maximum permissible size that fits in 16GB of GPU memory and hold the batch size constant across all pretrained models (with the exception of SST5 and AGNews where all pre-trained models are fine-tuned with a batch size of 16, and T5-small is fine-tuned with a batch size of 32). We also hold the batch size constant across the RNN and SP-CNN models trained from scratch. For a given dataset, this value may be larger than the batch size used by the pretrained models. We decided to not hold batch size consistent across all architectures as training an RNN with a batch size of 16 would be an inefficient use of resources.

A.5 Additional Case Study: Image Classification Benchmark Experiment

We demonstrate that LBT can be used for benchmarking models in tasks beyond the NLP domain by setting up a small image-classification benchmarking study which compares the efficacy of a stacked convolutional architecture (similar to VGG [75]) and a ResNet-18 [62] model on the CIFAR10 [76] and MNIST [77] datasets. We configure the hyperparameter search space such that we optimize over a limited set of hyperparameters, namely batch size, learning rate, and dimensions/number of fully connected layers. We use the skopt hyperparameter optimization algorithm [45] and sample a total of 10 parameter configurations for each model and dataset pair. All models are trained using the Adam optimization [44] algorithm. Finally, we emphasize that we perform no image augmentation and preprocessing strategies (e.g. random crops) to improve performance.

Table A.2 shows the accuracy of the best performing model for each dataset and model pair.

The experiment configuration files can be found here:

<https://drive.google.com/drive/folders/1jGdbs2en3AJI4onjnH15n4l1Tbo7S1X7?usp=sharing>

A.6 Lessons Learned

Over the course of this work, we gained several insights that we would like to share with maintainers of other benchmark software libraries.

Table A.2: **Image Classification Experiment Results.** The table reports the accuracy of the top performing models for each dataset and model pair in the small, image classification experiment

Model	Dataset	
	CIFAR10	MNIST
ResNet-18	0.804	0.994
Stacked CNN	0.697	0.993

Power of low-code, configurable interfaces. While developing and using LBT, the value of a low-code, configuration file interface became abundantly clear. One of the key value-adds of a configuration-based interface is that it enables users to declare their needs and constraints in plain English, making it simple for a wide range of audiences to set up experiments fairly quickly. Moreover, reproducibility is a natural by-product of a declarative, configuration-driven approach as the main input to the system is a set of files that can be stored and shared. In [A.4](#) we show how sharing and replicating an existing experiment from a set of configuration files is trivial.

Lean on the open-source community. LBT is built using a number of open-source packages including Ludwig [\[24\]](#), Ray Tune [\[78\]](#), Robustness Gym [\[16\]](#), TextAttack [\[15\]](#) and the experiment-impact-tracker [\[14\]](#). Relying on existing open-source packages enabled us to integrate the best-of-breed solutions for various tasks (e.g. Ray Tune for distributed hyperparameter tuning) without having to re-invent the wheel.

Provide abstractions for extensibility. One of the key design choices of LBT was to structure the codebase to be modular and extendable. By implementing proper abstractions, adding multiple tooling integrations became relatively simple. Moreover, adding additional metrics over time (such as carbon footprint tracking) was also made easier through the implementation of simple abstractions like a metric registry.

A.7 Additional Figures

This section contains additional figures referenced in the body of the main paper.

```

hyperopt:
  executor:
    cpu_resources_per_trial: 1
    gpu_resources_per_trial: 1
    kubernetes_namespace: ray
    type: ray
  goal: maximize
  metric: accuracy
  output_feature: label
  parameters:
    content.reduced_output:
      categories:
        - cls_pooled
        - sum
        - avg
      space: choice
      type: category
    label.fc_layers:
      categories:
        -
          - fc_size: 512
          - fc_size: 256
        -
          - fc_size: 512
        -
          - fc_size: 256
      space: choice
      type: category
    training.learning_rate:
      lower: 2.0e-05
      space: loguniform
      type: float
      upper: 0.01
  sampler:
    num_samples: 20
    search_alg:
      max_concurrent: 5
      type: skopt
    type: ray
  input_features:
    - encoder: bert
      level: word
      name: content
      preprocessing:
        word_sequence_length_limit: 128
        word_tokenizer: hf_tokenizer
        pretrained_model_name_or_path: bert-base-uncased
      pretrained_model_name_or_path: bert-base-uncased
      type: text
  output_features:
    - name: label
      type: category
  training:
    batch_size: 16
    early_stop: 3
    epochs: 25
    eval_batch_size: 64
    learning_rate: 0.01
    validation_metric: accuracy

```

Figure A.4: **Sample LBT experiment configuration.** Example of an LBT experiment configuration file generated at runtime. The file records the model architecture, training parameters, and hyperparameter search space of the given experiment