arXiv:cs.LG/2004.10703 (2020)

ktrain: A Low-Code Library for Augmented Machine Learning

Arun S. Maiya AMAIYA@IDA.ORG
Institute for Defense Analyses
Alexandria, VA, USA

Abstract

We present **ktrain**, a low-code Python library that makes machine learning more accessible and easier to apply. As a wrapper to TensorFlow and many other libraries (*e.g.,* `transformers`, `scikit-learn`, `stellargraph`), it is designed to make sophisticated, state-of-the-art machine learning models simple to build, train, inspect, and apply by both beginners and experienced practitioners. Featuring modules that support `text` data (*e.g.,* text classification, sequence tagging, open-domain question-answering), `vision` data (*e.g.,* image classification), `graph` data (*e.g.,* node classification, link prediction), and `tabular` data, **ktrain** presents a simple unified interface enabling one to quickly solve a wide range of tasks in as little as three or four "commands" or lines of code.

1. Introduction

Machine learning workflows can be quite involved and challenging for newcomers to master. Consider the following steps.

1) Model-Building. The training data may reside in a number of different formats from files in folders to CSVs or *pandas* dataframes. If the data is large, it must be wrapped in a generator. Data must be preprocessed in specific ways depending on different factors such as the language of training texts (*e.g.,* English vs. Chinese) and whether or not transfer learning is being employed. Learning rates, learning rate schedules, number of epochs, weight decay, and many other hyperparameters and settings must be selected or implemented.
2) Model-Inspection. Once trained, a model is inspected in terms of both its successes and failures. This may include classification reports on validation performance, easily identifying examples that the model is getting the most wrong, and Explainable AI methods to understand why mistakes were made.
3) Model-Application. Both the model and the potentially complex set of steps required to preprocess raw data into the format expected by the model must be easily saved, transferred to, and executed on new data in a production environment.

ktrain is a Python library for machine learning with the goal of presenting a simple, unified interface to easily perform the above steps regardless of the type of data (*i.e.,* text vs. images vs. graphs). Moreover, each of the three steps above can be accomplished in as little as three or four lines of code, which we refer to as "low-code" machine learning. **ktrain** can be used with any machine learning model implemented in TensorFlow Keras (`tf.keras`). In addition, **ktrain** currently includes out-of-the-box support for the following data types and tasks:

Text **Data:**

- **Text Classification**: auto-categorize documents across different dimensions
- **Text Regression**: predict numerical values (*e.g.,* prices) from textual descriptions
- **Sequence Tagging**: extract sequences of words that represent some concept of interest (*e.g.,* Named Entity Recognition or NER)
- **Unsupervised Topic Modeling**: discover latent themes buried in large document sets
- **Document Similarity with One-Class Learning**: find and score new documents based on thematic similarity to a set of seed documents
- **Document Recommendation**: recommend or return documents that are semantically-related to given text (*i.e.,* semantic search)
- **Text Summarization**: generate short summaries of long documents
- **Open-Domain Question-Answering**: submit questions to a large text corpus and receive exact answers

Vision **Data:**

- **Image Classification**: auto-categorize images across various dimensions
- **Image Regression**: predict numerical values (*e.g.,* age of person) from photos

Graph **Data:**

- **Node Classification**: auto-categorize nodes in a graph (*e.g.,* social media accounts)
- **Link Prediction**: predict missing links in social networks (*e.g.,* friend suggestions)

Tabular **Data:** classification and regression on data stored in tables

Many of the tasks above allow users to either choose from a menu of state-of-the-art models or employ a custom model. With respect to text classification, for example, available models include cutting-edge Transformer models like BERT (Devlin et al., 2018; Wolf et al., 2019) in addition to fast models such as fastText (Joulin et al., 2016) and NBSVM (Wang and Manning, 2012) that are amenable to being trained on a standard laptop CPU. Other features include a learning-rate-finder to estimate an optimal learning rate (Smith, 2018), easy-to-access learning rate schedules like the **1cycle policy** (Smith, 2018) and Stochastic Gradient Descent with Restarts (SGDR) (Loshchilov and Hutter, 2016), state-of-the-art optimizers like AdamW (Loshchilov and Hutter, 2017), ability to easily inspect classifications through Explainable AI and other methods, and a simple prediction API for use in deployment scenarios. **ktrain** is also bundled with pretrained, ready-to-use NER models for English, Chinese, and Russian. **ktrain** is open-source, free to use under a permissive Apache license, and available on GitHub at: `https://github.com/amaiya/ktrain`. In the next section, we compare and contrast our work with AutoML approaches.

2. Augmented ML

Automatic machine learning (AutoML) solutions typically place a strong emphasis on automating subsets of the model-building process such as architecture search and model selection (He et al., 2019). By contrast, **ktrain** places less emphasis on this aspect of automation and instead focuses on either partially or fully automating other aspects of the machine learning (ML) workflow. For these reasons, **ktrain** is less of a traditional AutoML platform

and more of what might be called a "low-code" ML platform. Through automation or semi-automation, **ktrain** facilitates the full machine learning workflow from curating and preprocessing inputs (*i.e.,* ground-truth-labeled training data) to training, tuning, troubleshooting, and applying models. In this way, **ktrain** is well-suited for domain experts who may have less experience with machine learning and software coding. Where possible, **ktrain** automates (either algorithmically or through setting well-performing defaults), but also allows users to make choices that best fit their unique application requirements. In this way, **ktrain** uses automation to augment and complement human engineers rather than attempting to entirely replace them. In doing so, the strengths of both are better exploited. Following inspiration from a blog post[1] by Rachel Thomas of `fast.ai` (Howard and Gugger, 2020), we refer to this as *Augmented Machine Learning* or AugML. For the remainder of this short paper, we will provide code examples to demonstrate ease-of-use.

3. Building Models

Supervised learning tasks in **ktrain** follow a standard, easy-to-use template, which we now describe.

STEP 1: Load and Preprocess Data. This step involves loading data from different sources and preprocessing it in a way that is expected by the model. In the case of text, this may involve language-specific preprocessing (*e.g.,* tokenization). In the case of images, this may involve auto-normalizing pixel values in a way that a chosen model expects. In the case of graphs, this may involve compiling attributes of nodes and links in the network (Data61, 2018). All preprocessing methods in **ktrain** return a `Preprocessor` instance that encapsulates all the preprocessing steps for a particular task, which can be employed when using the model to make predictions on new, unseen data.

STEP 2: Create Model. Users can create and customize their own model using `tf.keras` or select a pre-canned model with well-chosen defaults (*e.g.,* pretrained BERT text classifier (Devlin et al., 2018), models for sequence tagging (Lample et al., 2016), pretrained Residual Networks (He et al., 2015) for image classification). In the latter case, the model is automatically configured by inspecting the data (*e.g.,* number of classes, multilabel vs. multi-classification). At this stage, both the model and the datasets are wrapped in a `ktrain.Learner` instance, which is an abstraction to facilitate training.

STEP 3: Estimate Learning Rate. Users can employ the use of a learning rate range test (Smith, 2018) to estimate the optimal learning rate given the model and data. Some models like BERT have default learning rates that work well, so this step is optional.

STEP 4: Train Model. The **ktrain** package allows one to easily try different learning rate schedules. For instance, the `fit_onecycle` method employs a 1cycle policy (Smith, 2018). The `autofit` method employs a triangular learning rate schedule (Smith, 2018) with automatic early stopping and reduction of maximal learning rate upon plateau. Thus, specifying the number of epochs is optional in `autofit`. The `fit` method, when

1. `https://www.fast.ai/2018/07/16/auto-ml2/`

supplied with the `cycle_len` parameter, decays the learning rate each cycle using cosine annealing. Users can easily experiment with what works best for a particular problem.

To illustrate ease of use, we provide fully-complete examples for two different tasks.

3.1 Example: Text Classification

The first example is Chinese text classification. More specifically, we train a Chinese-language sentiment-analyzer on a dataset of hotel reviews.[2]

Fine-Tuning a BERT Text Classifier for Chinese:

```
import ktrain
from ktrain text as txt
# STEP 1: load and preprocess data
trn, val, preproc = txt.texts_from_folder('ChnSentiCorp', maxlen=75,
                                          preprocess_mode='bert')
# STEP 2: load model and wrap in Learner
model = txt.text_classifier('bert', trn, preproc=preproc)
learner = ktrain.get_learner(model,train_data=trn, val_data=val)
# STEP 3: estimate learning rate
learner.lr_find(show_plot=True)
# STEP 4: train model
learner.fit_onecycle(2e-5, 4)
```

Notice here that there is nothing special we need to do to support Chinese versus other languages like English. The language and character encoding are auto-detected and processing proceeds accordingly. Moreover, models are configured automatically through data inspection. For instance, the data is automatically analyzed to determine the number of categories, whether or not categories are mutually-exclusive or not, and if targets are numerical or categorical. The model is then auto-configured appropriately.

3.2 Example: Image Classification

In the next example, we build an image classifier on the *Dogs vs. Cats* dataset[3] with a standard ResNet50 model pretrained on ImageNet. As you can see in the code example below, the steps are very similar to the previous text classification example despite the task being completely different.

Fine-Tuning a Pretrained ResNet50 Image Classifier:

```
import ktrain
from ktrain import vision as vis
# STEP 1: load and preprocess data
data_aug = vis.get_data_aug(horizontal_flip=True)
(trn,val,preproc)=vis.images_from_folder(datadir='data/dogscats',data_aug=data_aug,
                                         train_test_names=['train', 'valid'])
# STEP 2: load model and wrap in Learner
model = vis.image_classifier('pretrained_resnet50', trn, val, freeze_layers=15)
learner = ktrain.get_learner(model=model,train_data=trn,val_data=val,batch_size=64)
# STEP 3: find good learning rate
learner.lr_find(show_plot=True)
# STEP 4: train
learner.autofit(1e-4)
```

2. https://github.com/Tony607/Chinese_sentiment_analysis
3. https://www.kaggle.com/c/dogs-vs-cats

A unified interface to different and disparate machine learning tasks reduces cognitive load and allows users to focus on more important tasks that may require domain expertise or are less amenable to automation.

4. Evaluating and Applying Models

Once a model is trained, we would like to evaluate how well it learned what it was supposed to learn. **ktrain** provides a simple interface to perform various analyses to this end. To compute detailed validation (or test) metrics, the `evaluate` method can be invoked. We can also easily identify the examples that the model got the most wrong by viewing examples with the highest validation (or test) loss using `view_top_losses`:

Evaluating and Inspecting Models:

```
# compute validation metrics, confusion matrices, and show report
learner.evaluate()
# example output of evaluate (uses validation set by default)
#                        precision    recall  f1-score
#
#           alt.atheism       0.92      0.93      0.93
#         comp.graphics       0.97      0.97      0.97
#               sci.med       0.97      0.95      0.96
#soc.religion.christian       0.96      0.96      0.96
#
#              accuracy                           0.96
#             macro avg       0.95      0.96      0.95
#          weighted avg       0.96      0.96      0.96

# view validation examples with highest loss
learner.view_top_losses()
```

ktrain also features a simple and easy-to-use prediction API to make predictions on new and unseen examples. A `Predictor` instance encapsulates both the model (*i.e.,* the underlying `tf.keras` model) and the preprocessing steps (*i.e.,* a `Preprocessor` instance) required to transform raw data into the format expected by the model. The `Predictor` instance can easily be saved and reloaded for deployments to production environments.

Making Predictions on New Data:

```
predictor = ktrain.get_predictor(learner.model, preproc) # create predictor
predictor.predict(raw_data)                               # make predictions
predictor.save('/tmp/mypredictor')                        # save predictor
predictor = ktrain.load_predictor('/tmp/mypredictor')     # reload predictor
```

For a subset of tasks like text classification and image classification, `Predictor` instances expose an `explain` method that will attempt to *explain* how a model arrived at a decision for a particular example: `predictor.explain(raw_data)`. This can shed light on why certain decisions were successfully or unsuccessfully made by the model. Explainable AI in **ktrain** is powered by libraries such as **shap** (Lundberg and Lee, 2017) and **eli5** with **lime** (Ribeiro et al., 2016).

5. Non-Supervised ML Tasks

All the examples covered thus far involve *supervised* machine learning. Other tasks such as training *unsupervised* topic models to discover latent themes in document sets or using *pre-*

trained NER models follow slightly different steps than those described previously. Despite involving a different pipeline, these non-supervised tasks also employ a low-code API and can be implemented in as little as three lines of code. To illustrate this, we provide a code example for a fully-functional, end-to-end, **open-domain question-answering system** using the well-studied *20 Newsgroups* dataset.[4] We will first load the dataset into a Python list called `docs` using `scikit-learn` (Pedregosa et al., 2011) (see Appendix A). The basic idea here is to use the document set as a knowledge base that can be issued natural language questions to receive exact answers. In this case, we would like to issue questions about the subject matter buried in the *20 Newsgroups* dataset and receive exact answers. To accomplish this, the following steps are performed:

1. Index documents to a search engine.
2. Use the search index to locate documents that contain words in the question.
3. Extract paragraphs from these documents for use as contexts and use a BERT model pretrained on the SQuAD dataset to parse out candidate answers.
4. Sort and prune candidate answers by confidence scores and returns results.

This entire workflow to build an end-to-end, open-domain question-answering (QA) system can be implemented with a surprisingly minimal amount of code with **ktrain**:

Building an End-to-End Open-Domain QA System in ktrain

```
# build open-domain QA system
from ktrain import text
text.SimpleQA.initialize_index('/tmp/myindex')
text.SimpleQA.index_from_list(docs, '/tmp/myindex', commit_every=len(docs))
qa = text.SimpleQA('/tmp/myindex')

# ask a question
qa.ask('When did the Cassini probe launch?') # returns "October of 1997"
```

As shown above, upon building the QA system in **only 3 lines of code**, we can submit natural language questions and receive exact answers. In the example shown, we use the `ask` method to submit the question, "**When did the Cassini probe launch?**". The candidate answer with the highest confidence score returned by the `ask` method is the correct answer of "**October of 1997**" (see Appendix B). Note that, for document sets that are too large to fit into a Python list, one can index documents using `index_from_folder` instead of `index_from_list`. See Appendix C for some additional low-code ML examples.

6. Conclusion

This work presented **ktrain**, a low-code platform for machine learning. **ktrain** currently includes out-of-the-box support for training models on `text`, `vision`, and `graph` data. As a simple wrapper to TensorFlow Keras, it is also sufficiently flexible for use with custom models and data formats, as well. Inspired by other low-code (and no-code) open-source ML libraries such as `fastai` (Howard and Gugger, 2020) and `ludwig` (Molino et al., 2019), **ktrain** is intended to help further democratize machine learning by enabling beginners and domain experts with minimal programming or data science experience to build sophisticated

4. `http://archive.ics.uci.edu/ml/datasets/Twenty+Newsgroups`

machine learning models with minimal coding. It is also a useful toolbox for experienced practitioners needing to rapidly prototype deep learning solutions.

References

CSIRO's Data61. Stellargraph machine learning library. https://github.com/stellargraph/stellargraph, 2018.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *arXiv preprint arXiv:1908.00709*, 2019.

Jeremy Howard and Sylvain Gugger. Fastai: A layered api for deep learning. *Information*, 11(2): 108, Feb 2020. ISSN 2078-2489. doi: 10.3390/info11020108. URL http://dx.doi.org/10.3390/info11020108.

Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural architectures for named entity recognition. *arXiv preprint arXiv:1603.01360*, 2016.

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017.

Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. Ludwig: a type-based declarative deep learning toolbox. *arXiv preprint arXiv:1909.07930*, 2019.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier. *arXiv preprint arXiv:1602.04938*, 2016.

Leslie N. Smith. A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*, 2018.

Sida Wang and Christopher D. Manning. Baselines and bigrams: Simple, good sentiment and topic classification. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Short Papers - Volume 2*, ACL 12, page 9094, USA, 2012. Association for Computational Linguistics.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rmi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

Appendix A. Loading the 20 Newsgroups Dataset

For the open-domain question-answering code example, we load the *20 Newsgroups Dataset* using `scikit-learn`.

```
# load 20newsgroups dataset into a Python list
from sklearn.datasets import fetch_20newsgroups
remove = ('headers', 'footers', 'quotes')
newsgroups_train = fetch_20newsgroups(subset='train', remove=remove)
newsgroups_test = fetch_20newsgroups(subset='test', remove=remove)
docs = newsgroups_train.data +  newsgroups_test.data
```

Appendix B. Open-Domain Question-Answering Example

We include a screenshot of a Jupyter notebook showing results from the question-answering API in `ktrain`.

```
answers = qa.ask('When did the Cassini probe launch?')
qa.display_answers(answers[:5])
```

	Candidate Answer	Context	Confidence	Document Reference
0	in october of 1997	cassini is scheduled for launch aboard a titan iv / centaur in october of 1997 .	0.348673	59
1	on january 26,1962	ranger 3, launched on january 26,1962 , was intended to land an instrument capsule on the surface of the moon, but problems during the launch caused the probe to miss the moon and head into solar orbit.	0.195162	8525
2	on november 5,1964	mariner 3, launched on november 5,1964 , was lost when its protective shroud failed to eject as the craft was placed into interplanetary space.	0.162835	8525
3	launched october 18,1962	ranger 5, launched october 18,1962 and similar to ranger 3 and 4, lost all solar panel and battery power enroute and eventually missed the moon and drifted off into solar orbit.	0.077810	8525
4	2001	possible launch dates : 1996 for imaging orbiter, 2001 for rover.	0.069741	59

We use the `qa.display` method to format and display answers within a Jupyter notebook. The top candidate answer indicates that the Cassini space probe was launched in **October of 1997**, which is correct. The specific answer within its context is highlighted in red under the column **Context**. Since we used `index_from_list` to index documents, the last column (populated from the `reference` field in the returned `answers` dictionaries) shows the list index associated with the newsgroup posting containing the answer. This `reference` field can be used to peruse the entire document containing the answer with `print(docs[59])`. If using `index_from_folder` to index documents, then the `reference` field will be populated with the relative file path of the document instead.

Appendix C. Additional Low-Code ML Examples

In this final section, we include some additional low-code examples of both supervised and non-supervised machine learning tasks in **ktrain** to further illustrate ease-of-use.

Named Entity Recognition with BioBERT embeddings:

```
import ktrain
from ktrain import text as txt
x_train= [['IL-2', 'responsiveness', 'requires', ...], ...]
y_train=[['B-protein', 'O', 'O', ...], ...]
(trn, val, preproc) = txt.entities_from_array(x_train, y_train)
model = txt.sequence_tagger('bilstm-bert', preproc,
                            bert_model='monologg/biobert_v1.1_pubmed')
learner = ktrain.get_learner(model, train_data=trn, val_data=val,batch_size=128)
learner.fit(0.01, 1, cycle_len=5) # decays lr with cosine annealing
```

Node Classification with Graph Neural Networks:

```
import ktrain
from ktrain import graph as gr
# STEP 1: load and preprocess data
(trn, val, preproc) = gr.graph_nodes_from_csv('cora.content','cora.cites',sep='\t')
# STEP 2: load model and wrap in Learner
model = gr.graph_node_classifier('graphsage', trn)
learner = ktrain.get_learner(model, train_data=trn, val_data=val batch_size=64)
# STEP 3: estimate learning rate
learner.lr_find(max_epochs=50, show_plot=True)
# STEP 4: train model
learner.autofit(0.01) # triangular lr schedule with early stopping
```

Document Recommendation Engine: (no labeled training examples required)

```
import ktrain
tm = ktrain.text.get_topic_model(docs, n_features=10000)
tm.build(docs, threshold=0.25) # documents to semantically meaningful vectors
tm.train_recommender()         # trains Nearest Neighbors model
rawtext = "Elon Musk leads Space Exploration Technologies (SpaceX), where he..."
tm.recommend(text=rawtext, n=5) # top 5 suggestions based on thematic similarity
```

Zero-Shot Topic Classification: (no labeled training examples required)

```
from ktrain import text
zsl = text.ZeroShotClassifier()
topic_strings=['politics', 'elections', 'sports', 'films', 'television']
doc = "I am unhappy with decisions of government and will vote in 2020."
zsl.predict(doc, topic_strings=topic_strings, include_labels=True)
# output:
# [('politics', 0.9829), 'elections', 0.9881),
# ('sports', 0.0003), ('films', 0.0008), ('television', 0.0004)]
```