Tianchengfinal package Instructions

Tiancheng Yang

08-01-2023

The `Tianchengfinal` package is built to explore the impact of word embedding dimensions in neural network language models, as proposed in the seminal work of (Bengio, Ducharme, and Vincent 2000).This package offers a set of functions for data preprocessing, model training, simulation, and visualization.

Background

The `Tianchengfinal` package is built upon the foundational work of Bengio et al. (2000), which proposed a novel approach to tackle the curse of dimensionality in Natural Language Processing (NLP) and to learn the similarity between words. This was achieved by embedding words in a continuous vector space where semantically and syntactically similar words are mapped to nearby points.

The authors designed a neural network language model (NNLM) for learning these word feature vectors simultaneously with the probability function parameters of word sequences, essentially modeling the joint distribution of word sequences. The NNLM model consists of an embedding layer, a hidden layer, and an output layer.

The model takes integer-encoded vocabulary as inputs which represent sequences of words extracted from a text corpus. An embedding layer is utilized to convert these inputs into dense vector embeddings, with the output dimension determined by the desired size of word embedding vectors.

Subsequent transformations of the word embedding vectors are performed in the hidden Layer, which employs a hyperbolic tangent (tanh) activation function. The model's final layer, the output layer, deploys a softmax activation function to output a probability distribution over the vocabulary for the subsequent word of the input word sequence.

The primary aim of the model is to maximize the predictive accuracy of the next word in a sequence based on its preceding words. The model did so by learning the word vectors and the parameters of the probability function simultaneously. This unique characteristic set the proposed model apart from its predecessors, making it a significant milestone in language modeling.

Installation

Important: Given the complex dependencies associated with the TensorFlow package for R, it is highly recommended to execute the toy experiment on Google Colab. Colab conveniently has all the necessary dependencies pre-installed for users.

Colab preparation

Here is a **link** to a publicly shared Google Colab notebook where you can easily execute the provided code. Alternatively, you could also create a new notebook in your own Google Colab environment and follow the instructions.

In Colab, the default runtime type is Python, please click on the `Runtime` button on the top left and select `Change Runtime Type` and change that to R.

Then, please install the dependency file for `fftwtools` via linux shell

```
# install dependency file for fftwtools via linux shell
shell_call <- function(command, ...) {
  result <- system(command, intern = TRUE, ...)
  cat(paste0(result, collapse = "\n"))
}

shell_call("apt-get install libfftw3-dev")
```

Package Installation

After that, you could download and install the package `Tianchengfinal` directly from github:

```
require(devtools)
devtools::install_github("TianchengY/Tianchengfinal")
library(Tianchengfinal)
```

Content

Functions

The main functions of `Tianchengfinal` package was classified in to folloing files:

```
# \\\ R
#     \\\ rocstories.R
#     \\\ data_processing.R
#     \\\ model.R
#     \\\ simulation.R
#     \\\ table.R
#     \\\ plot.R
```

- The `rocstories.R` contains the document for the ROCStories dataset (Mostafazadeh et al. 2016).
- The `data_processing.R` contains functions that help to preprocess and clean the data.
- The `model.R` includes functions for model training and prediction.
- The `simulation.R` includes functions to run Monte Carlo simulations.
- The `table.R` provides functions to generate tables for model results and performance metrics.
- The `plot.R` provides various visualization tools to help understand the data and model performance.

Main Function for Simulation

Let's take a closer look on the arguments of the most important function `run_simluations`

Argument	Description	Default Value
`embedding_dim_values`	A numeric vector that contains the values of the embedding dimension to be tested.	-
`param_values`	A numeric vector that contains the values of the target hyperparameter to be tested.	-
`param_name`	A character string that specifies the name of the target hyperparameter. Default is "context_size".	"context_size"
`other_params`	A list that contains the values of the other hyperparameters. See below for details.	See below

The `other_params` list should include the following elements:

Element	Description	Default Value
`data`	A data frame that contains the data to be processed.	-
`n_simulations`	An integer that specifies the number of simulations to run. Default is 50.	50
`random_seed`	An integer that sets the seed for reproducibility. Default is 900.	900
`train_portion`	A numeric value that specifies the portion of the data to be used for training. Default is 0.8.	0.8
`val_portion`	A numeric value that specifies the portion of the data to be used for validation. Default is 0.1.	0.1
`test_portion`	A numeric value that specifies the portion of the data to be used for testing. Default is 0.1.	0.1
`lowest_frequency`	An integer that specifies the lowest frequency of words to be included in the vocabulary. Default is 3.	3
`batch_size`	An integer that specifies the batch size. Default is 256.	256
`epochs`	An integer that specifies the number of epochs. Default is 20.	20
`h`	An integer that specifies the number of units in the hidden layer. Default is 50.	50
`learning_rate`	A numeric value that specifies the learning rate. Default is 5e-3.	5e-3
`early_stop_min_delta`	A numeric value that specifies the minimum change in the monitored quantity to qualify as an improvement. Default is 0.01.	0.01
`early_stop_patience`	An integer that specifies the number of epochs with no improvement after which training will be stopped. Default is 2.	2
`verbose`	An integer that specifies the verbosity mode. Default is 1.	1

The Toy Experiment

Due to the computation limitations and reproducibility requirements, the experiment was divided into two stages: a toy experiment for reproducibility and a full experiment for final results. For the toy example, only two combinations were applied with an embedding size of 3 and 30 and context sizes of 2. Also, to save time and memory (RAM), the toy example executed only three simulations for each parameter combination, and employed a batch size of 64, as opposed to the default 256. The execution time for the toy experiment was approximately 15 minutes when run on Google Colab.

```r
# load package built-in dataset rocstories
data(rocstories)

# set hyparameters and arguments for the toy experiment
embedding_dim_values <- c(3,30)
context_size_values <- c(2)
other_params <- list(data=rocstories,n_simulations = 3, random_seed = 900,
                     train_portion = 0.8, val_portion = 0.1, test_portion = 0.1,
                     lowest_frequency = 3,batch_size=64,epochs=20,h=50,
                     learning_rate=5e-3, early_stop_min_delta=0.01,
                     early_stop_patience=2,verbose=0)

# run simulation, it will take 15ish minutes
results <- run_simulations(embedding_dim_values, context_size_values, "context_size", other_params)
```

```r
# show the results in a table
create_table(results)
```

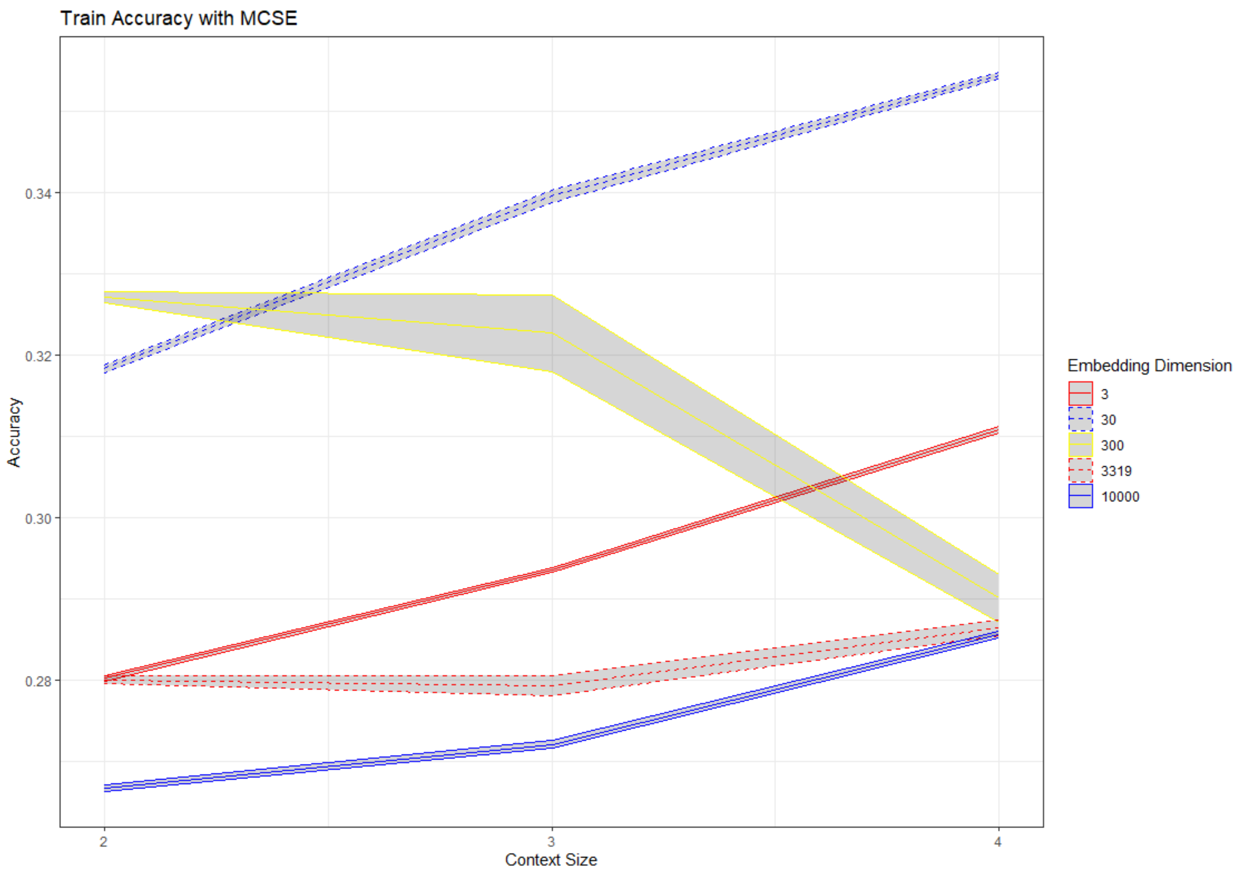
Embedding Dim	Context Size	Train Accuracy	MCSE Train Accuracy	Test Accuracy	MCSE Test Accuracy
3	2	0.2797316	0.0018436449	0.2679446	0.0027887997
30	2	0.3116650	0.0006235941	0.2749734	0.0006922258

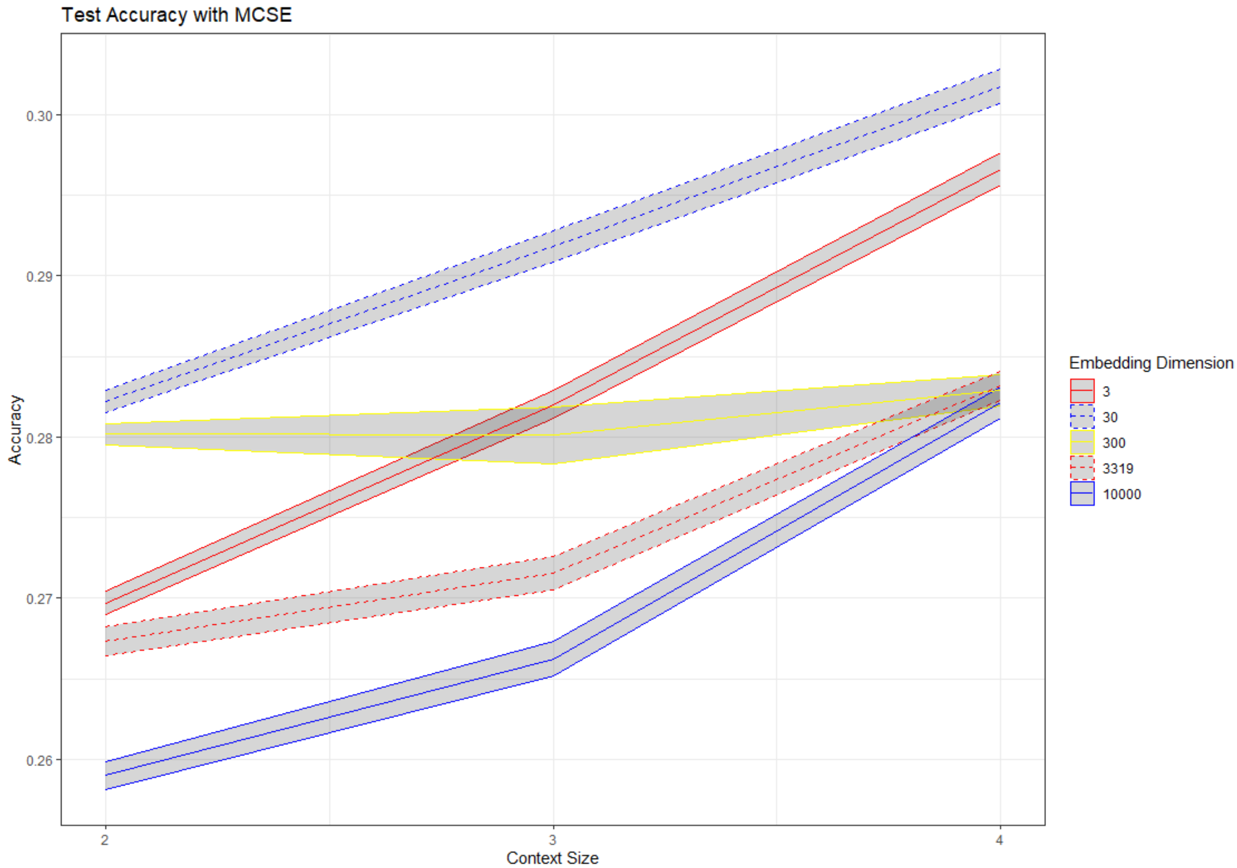
Full Experiment

For the full experiment, the experiment was set up to run 50 simulations for each combination for embedding sizes of (3, 30, 300, 3319, 10000) and context sizes of (2, 3, 4), which provides a compelling Monte Carlo standard error in an acceptable time frame, approximately 3 hours running with a dedicated GPU (30+ hours on CPU).

```r
# load package built-in dataset rocstories
data(rocstories)

# set hyparameters and arguments for the full experiment
embedding_dim_values <- c(3,30,300,3319,10000)
context_size_values <- c(2,3,4)
other_params <- list(data=rocstories,n_simulations = 50, random_seed = 900,
                     train_portion = 0.8, val_portion = 0.1, test_portion = 0.1,
                     lowest_frequency = 3,batch_size=256,epochs=20,h=50,
                     learning_rate=5e-3,
                     early_stop_min_delta=0.01,early_stop_patience=2,verbose=0)

# run simulation
results <- run_simulations(embedding_dim_values, context_size_values, "context_size", other_params)
```

```r
# plot training and test accuracy with MCSE vs. context size for word embeddings
plot_train_accuracy(results, "Context Size", context_size_values)
plot_test_accuracy(results, "Context Size", context_size_values)

# show the results in a table
create_table(results)
```


Train Accuracy with MCSE
0.34
0.32
0.30
0.28
Accuracy
2
3
4
Context Size
Embedding Dimension
3
30
300
3319
10000



embedding_dim	context_size	train_accuracy	mcse_train_accuracy	test_accuracy	mcse_test_accuracy
10000	2	0.2667965	0.0004101	0.2590224	0.0008682
10000	3	0.2721489	0.0005049	0.2662629	0.0010507
10000	4	0.2856559	0.0004161	0.2821448	0.0009761
3319	2	0.2800627	0.0004461	0.2673450	0.0008760
3319	3	0.2793337	0.0012798	0.2715629	0.0010635
3319	4	0.2864773	0.0009960	0.2832104	0.0008959
300	2	0.3270765	0.0007316	0.2801885	0.0006613
300	3	0.3226938	0.0047072	0.2801317	0.0017662
300	4	0.2900868	0.0029968	0.2829235	0.0009604
30	2	0.3183021	0.0004686	0.2822268	0.0007069
30	3	0.3395251	0.0007663	0.2918262	0.0009806
30	4	0.3542792	0.0003975	0.3017532	0.0010521
3	2	0.2802385	0.0003503	0.2697220	0.0007106
3	3	0.2935849	0.0003225	0.2820350	0.0008403
3	4	0.3108114	0.0003704	0.2966302	0.0010157

Assertions and Testings

Function Assertions

All functions within the `Tianchengfinal` package includes strict assertions at the beginning. These assertions ensure that the arguments passed to each function are valid and safe, thereby enhancing the robustness and reliability of the package's operations.

For example, in the function `split_data`, the following assertions will be executed to ensure all arguments are valid:

```
split_data <- function(x_data, y_data, vocab, random_seed=900, train_portion=0.8,
                       val_portion=0.1, test_portion=0.1) {
  # Assertions
  if (!is.character(vocab) || length(vocab) < 1) {
    stop("vocab must be a non-empty character vector")
  }
  if (!is.numeric(random_seed) || length(random_seed) != 1 || round(random_seed) !=
      random_seed || random_seed < 0) {
    stop("random_seed must be a non-negative integer")
  }
  if (!is.numeric(train_portion) || length(train_portion) != 1 || train_portion < 0 ||
      train_portion > 1) {
    stop("train_portion must be a numeric value between 0 and 1")
  }
  if (!is.numeric(val_portion) || length(val_portion) != 1 || val_portion < 0 ||
      val_portion > 1) {
    stop("val_portion must be a numeric value between 0 and 1")
  }
  if (!is.numeric(test_portion) || length(test_portion) != 1 || test_portion < 0 ||
      test_portion > 1) {
    stop("test_portion must be a numeric value between 0 and 1")
  }

  # Rest Code
  ...
}
```

Testthat

Testthat is a convenient and powerful library for unit testing. Here is an example for testing the assertions of the function `split_data`:

```
# Test the function
test_that("split_data function throws an error when arguments are incorrect", {
  # Test that an error is thrown when vocab is not a character vector
  expect_error(split_data(x_data = x_data, y_data = y_data, vocab = 1, random_seed =
                           900, train_portion = 0.8, val_portion = 0.1, test_portion
                          = 0.1))

  # Test that an error is thrown when random_seed is not a non-negative integer
  expect_error(split_data(x_data = x_data, y_data = y_data, vocab = vocab, random_seed
                          = "900", train_portion = 0.8, val_portion = 0.1,
                          test_portion = 0.1))

  # Test that an error is thrown when train_portion is not a numeric value between 0 and 1
  expect_error(split_data(x_data = x_data, y_data = y_data, vocab = vocab, random_seed
                          = 900, train_portion = 1.5, val_portion = 0.1, test_portion
                          = 0.1))

  # Test that an error is thrown when val_portion is not a numeric value between 0 and 1
  expect_error(split_data(x_data = x_data, y_data = y_data, vocab = vocab, random_seed
                          = 900, train_portion = 0.8, val_portion = 1.5, test_portion
```

```
                          = 0.1))

  # Test that an error is thrown when test_portion is not a numeric value between 0 and 1
  expect_error(split_data(x_data = x_data, y_data = y_data, vocab = vocab, random_seed
                          = 900, train_portion = 0.8, val_portion = 0.1, test_portion
                          = 1.5))
})
```

One could simply run `devtools::test()` to test all unit tests.

References

Bengio, Yoshua, Réjean Ducharme, and Pascal Vincent. 2000. "A Neural Probabilistic Language Model." *Advances in Neural Information Processing Systems* 13.

Mostafazadeh, Nasrin, Nathanael Chambers, Xiaodong He, Devi Parikh, Dhruv Batra, Lucy Vanderwende, Pushmeet Kohli, and James Allen. 2016. "A Corpus and Cloze Evaluation for Deeper Understanding of Commonsense Stories." In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 839–49. San Diego, California: Association for Computational Linguistics. https://doi.org/10.18653/v1/N16-1098.