ML2P-VAE

Geoffrey Converse

```
library(ML2Pvae)
```

Contents

Introduction

This package allows easy implementation of the ML2P-VAE method for estimating parameters in Item Response Theory (IRT). This method was first proposed by Curi et. al. [2], and was later compared with a similar method which used regular autoencoders, rather than variational autoencoders (VAE) [1]. Other parameter estimation methods rely on numerical integration or MCMC methods, and are infeasible when the latent ability dimension is greater than 8. ML2P-VAE methods learn one function mapping response sets to latent abilities, and another function mapping latent abilities to the probability of answering items correctly. The latter is the Multidimensional Logistic 2-Parameter model.

Installation and Setup

This package requires an installation of Python 3, along with the Python libraries `tensorflow`, `keras`, and `tensorflow-probability`. These can be installed via pip from the system terminal with the command `pip install <pkg>`.

Inside of `R`, `ML2Pvae` requires the CRAN packages `tensorflow` and `tfprobability`, installed via `install.packaegs('<pkg>')`. It also requires the package `keras`, but this must be downloaded from Git, rather than CRAN. This can be done using the `devtools` package by running `devtools::install_github('rstudio/keras')`.

Multidimensional Logistic 2-Parameter (ML2P) Model

In Item Response Theory, the goal is often to quantify the relation between the underlying ability (or abilities) of students and their performance on an assessment. Formally, we assume that every student j posesses some latent ability value Θ_j, which is not directly observable. Often, we assume that Θ_j is distributed normally across the population of N students. In a practical setting, our data for each student may be a binary vector $u_j \in \mathbb{R}^n$ where a 1 (resp. 0) corresponds to a correct (resp. incorrect) answer, to an exam consisting of n questions (items). Given a student j's assessment results, how can we infer the latent value Θ_j?

A naive approach is to simply use the raw exam score, or accuracy. But if two students both score 80% on an exam while missing different items (so $u_1 - u_2 \neq 0_n$), it is unlikely that they have the exact same ability value. Not all exam items are equal - they often vary in difficulty and in the skills required to answer them correctly.

Instead, theory has been developed [6] that states that the probability of student j answering item i correctly is a function of Θ_j, along with parameters Λ_i associated with item i:

$$P(u_{ij} = 1|\Theta_j) = f(\Theta_j; \Lambda_i)$$

In this work, we are concerned with the case where $\Theta_j = (\theta_{j1}, ..., \theta_{jK})^\top \in \mathbb{R}^K$, so that an exam is designed to assess multiple skills. For example, an elementary math exam may test the abilities "add," "subtract," "multiply," and "divide." In order to keep track of which items asses which abilities, we can develop a binary matrix $Q \in \mathbb{R}^{K \times n}$ where

$$Q_{ki} = \begin{cases} 1 & \text{if item } i \text{ requires skill } k \\ 0 & \text{otherwise} \end{cases}$$

It is important to note that multiple items can require multiple skills.

The model that this package focuses on is the Multidimensional Logistic 2-Parameter (ML2P) model. The probability of a correct response is given by

$$P(u_{ij} = 1|\Theta_j) = \frac{1}{1 + \exp[-\vec{a}_i^\top \Theta_j + b_i]} = \frac{1}{1 + \exp[-\sum_{k=1}^K a_{ik}\theta_{jk} + b_i]}$$

where the discrimination parameters $\vec{a}_i^\top = (a_{ik})_{1 \leq k \leq K}$ determine *how much* of skill k is required to answer item i correctly, and the difficulty parameter b_i quantifies the difficulty of item i.

In application, we often only have response vectors for a set of students - so we need to estimate the ability parameters Θ_j for each student, along with the discrimination and difficulty parameters for each item. While there are many parameter estimation methods, including the Expectation-Maximization algorithm and various MCMC methods, many of them become infeasible as the number of latent traits increases beyond $\dim(\Theta_j) = 8$.

ML2P-VAE Description

Certain modifications to the typical VAE architecture [3] are required in order to be used as parameter estimation methods. First, there are no hidden layers in the decoder. Instead, the non-zero weights in the decoder, are determined by a given Q-matrix. These connect the encoded distribution layer directly to the output layer. The output layer must use the sigmoidal activation function,

$$\sigma(z_i) = \frac{1}{1 + e^{-z_i}} \tag{1}$$

where $z_i = \sum_{k=1}^{K} w_{k,i}\alpha_k + b_i$. Here, $w_{i,k}$ is the weight between the k-th and i-th nodes in the second-to-last and output layer, respectively. α_k is the activation of the k-th node in the second-to-last layer, and b_i is the bias value of the i-th node in the output layer. Note that the form of this activation function is identical to that of the ML2P model. This allows for interpretataion of the weights and biases in the final network layer as estimates to the discrimination and difficulty parameters, respectively. The activation values $\alpha_{:}$, which are produced by the encoder of the VAE, can be interpreted as estimates to the student ability parameters Θ.

Description of Exported Functions

There are two functions in this package which are available for the user. Both of them assist in constructing an ML2P-VAE model, but the two make different assumptions about the latent abilities. Each of these two functions return three Keras models: the encoder, decoder, and full VAE. This way, the user can train the VAE, use the ecoder to obtain ability parameter estimates from student's response sets, and use the decoder to obtain item parameter estimates.

Independent Latent Traits

`build_vae_standard_normal()` assumes that each of the latent abilities are independent of one another. Thus, the abilities Θ are distrubited according to $\mathcal{N}(0, I)$. Constructing this model yields a similar architecture to a typical VAE, which are often trained to a normal independent distribution.

The encoder outputs $2 \cdot$ (`num_skills`) nodes, half of which represent the means of each latent trait, and the other half represent the log variance of each ability. There are no hidden layers in the decoder, and the encoded distribution is mapped directly to the output layer. The nonzero weights between these layers are determined by the Q-matrix.

```
build_vae_standard_normal(num_items,
                          num_skills,
                          Q_matrix,
                          model_type = 2,
                          enc_hid_arch=c(ceiling((num_items + num_skills)/2)),
                          hid_enc_activations=rep('sigmoid', length(enc_hid_arch)),
                          output_activation='sigmoid',
                          kl_weight=1)
```

The first three arguments must be provided by the user. `num_items` and `num_skills` describe the assessment length and the number of abilities being evaluated by the assessment. `Q_matrix` is a `num_skills` by `num_items` matrix which specifies the relationship between items and abilities. This matrix specifies the nonzero weights in the decoder of the VAE, which allows interpretation of network parameters as discrimination/difficulty parameter estimates and the hidden encoded layer as ability parameter estimates. Without the`Q_matrix`, estimation of these parameters would not be possible.

The other parameters in `build_vae_standard_normal()` have default settings which can easily be adjusted. `model_type` can be either 1 or 2, specifying the use of a 1-Parameter or 2-Parameter Logistic model. If the

1-PL model is used, then all discrimination parameters (the weights in the decoder) are fixed to be equal to 1, and only the difficulty parameters (the biases in the output layer) are estimated.

`enc_hid_arch` is a vector of integers, who's length determines the number of hidden layers in the encoder, and the entries describe the number of nodes in the hidden layers. The default setting is to have one hidden layer in the encoder, with size halfway between `num_items` (the size of the input layer) and `num_skills` (the dimension of the encoded distribution). `hid_enc_activations` is a vector of strings, which specifies the activation functions to be used in each encoder hidden layer. Note that this must have the same length as `enc_hid_arch`, and each string must be a valid activation function supported by Tensorflow. Valid options include: 'elu', 'exponential', 'hard_sigmoid', 'linear', 'relu', 'selu', 'sigmoid', 'softmax', 'softplus', 'softsign', 'tanh'.

`output_activation` specifies the activation function in the output of the decoder. The default is to use a sigmoidal activation function, and should not be changed when using ML2P-VAE to estimate logistic models. If a different activation function is used, then the decoder cannot be interpreted as a ML2P model. The final option is the hyperparameter `kl_weight`, which changes the value of λ in the VAE loss function

$$\mathcal{L}(w) = \mathcal{L}_0(w) + \lambda \mathcal{L}_{KL}(w),$$

where $\mathcal{L}_0$ is the reconstruction loss of the VAE, and $\mathcal{L}_{KL}$ is the KL-Divergence [4] between $\mathcal{N}(0, I)$ and the distribution learned by the encoder.

Correlated Latent Traits

The second model, `build_vae_normal_full_covariance()`, allows the latent traits to be correlated with one another, i.e. that latent abilities Θ are distrubited acording to $\mathcal{N}(\mu, \Sigma)$. However, these relationships must be known, as the ML2P-VAE method does not estimate correlations between abilities. Thus, the full covariance/correlation matrix Σ must be provided in addition to the student response sets and the Q-matrix.

The architecture of the decoder in `build_vae_normal_full_covariance()` is exactly the same as that of `build_vae_standard_normal()`. But the encoder outputs $K + K(K+1)/2$, where $K =$`num_skills`. The first K represent the means of each latent ability, and the remaining $K(K+1)/2$ nodes form a lower triangular matrix L which represents the matrix logarithm of the Cholesky decomposition of a covariance matrix. Thus for a data sample x_0, we can obtain μ_0 and L_0, which we can use to construct $\Sigma_0 = e^{L_0} \cdot \left(e^{L_0}\right)^\top$. By construction, the matrix Σ_0 will always be symmetric and positive definite, regardless of the input x_0 or values of trainable parameters in the encoder. Thus Σ_0 can be interpreted as a covariance matrix, and the encoder maps each point x_0 to a multivariate Gaussian distribution $\mathcal{N}(\mu_0, \Sigma_0)$.

```
build_vae_normal_full_covariance(num_items,
                                 num_skills,
                                 Q_matrix,
                                 model_type = 2,
                                 mean_vector = rep(0, num_skills),
                                 covariance_matrix = diag(num_skills),
                                 enc_hid_arch = c(ceiling((num_items + num_skills)/2)),
                                 hid_enc_activations = rep('sigmoid', length(enc_hid_arch)),
                                 output_activation = 'sigmoid',
                                 kl_weight = 1)
```

Every argument from `build_vae_standard_normal()` is also included in `build_vae_normal_full_covariance()`. There are two additional parameters, `mean_vector` and `covariance_matrix`, which specify μ and Σ in the assumed multivariate Gaussian distribution of latent traits. `mean_vector` must be of length `num_skills`, and `covariance_matrix` must be a `num_skills` by `num_skills` symmetric positive definite matrix.

The default settings for these two arguments specify a $\mathcal{N}(0, I)$ distribution, but this method should never be used as such. Though it will work, `build_vae_standard_normal()` will build a more efficient and more

accurate model for assuming independent latent abilities. `build_vae_full_covariance()` is most effectively used when these parameters specify a more compilcated distribution, such as $\mathcal{N}(\mu, \Sigma)$ or $\mathcal{N}(0, \Sigma)$.

Model Training

For users unfamiliar with Tensorflow and Keras, this package provides a simple wrapper function which allows for training of an ML2P-VAE model. `train_model()` essentially just calls `keras::fit()` with the same parameters. The output of this function is a history of the Keras fit, which can be plotted to show the training loss after each epoch.

```
train_model(model,
            train_data,
            num_epochs = 10,
            batch_size = 1,
            validation_split = 0.15,
            shuffle = FALSE,
            verbose = 1)
```

The inputs to `train_model()` include a compiled Keras model, which nedds to have autoencoder or VAE structure where the input and output layers are the same dimension. This can be obtained from the third item returned from `build_vae_standard_normal()` or `build_vae_full_covariance()`. The argument `train_data` is a `num_train` by `num_items` binary matrix, where entry $(j, i) = 1$ if student j answered item i correct, and 0 otherwise. The remaining arguments are optional and have default values. The number of loops over the training data is set by `num_epochs`. The default `batch_size` is set to be 1 (pure stochastic gradient descent), but can be any integer less than or equal to `num_train`. `validation_split` determines the percentage of `train_data` to be held out of training and used in a validation set. `shuffle` tells whether to randomly shuffle the data after epoch or not. The options for `verbose` are 0: silent, 1: progress bar, and 2: one line per epoch.

Fetching Item Parameter Estimates

After a model has been trained, the item parameter estimates are found in the trainable variables in the VAE. The function `get_item_parameter_estimates()` looks at all trainable parameters of the decoder part of the VAE and returns the values which serve as estimates to the item paramters.

```
get_item_parameter_estimates(decoder, model_type = 2)
```

Only one argument is required, the `decoder`, which takes in a Keras model. This can be obtained from the second item returned by `build_vae_standard_normal()` or `build_vae_full_covariance()`. The optional argument `model_type` defaults to the 2-parameter logistic model, but can be set to 1 for the 1-parameter logistic model. In the former case, `get_item_parameter_estimates()` returns the output layer biases and the weights connected to the output layer, which serve as estimates to difficulty and discrimination paramters, respectively. If `model_type = 1`, then only the output layer biases (difficulty parameter estimates) are returned.

Converting Response Sets into Ability Parameter Estimates

The encoder of a trained ML2P-VAE model maps student responses to their estimates of their ability parameter. The function `get_ability_parameter_estimates()` acts in a similar way to `keras::predict()`. Because of the output shape of the encoder, we include this function to conveniently reshape and format the output.

```
get_ability_parameter_estimates(encoder, responses)
```

The `encoder` argument takes in a Keras model, and should be obtained from the first item returned by `build_vae_standard_normal()` or `build_vae_full_covariance()`. Note that only the encoder, not the full VAE, should be inputted here. The `responses` field represents that data to get skill estimates for, and takes in a matrix with the same number of columns as `num_items`. In practice, the input to `responses` will be `train_data`, with the number of rows equal to the number of students.

Exported Data

We include a data set in the `ML2Pvae` package for demonstrative use. The data is from a simulated 30 item exam which assesses 3 latent traits. The latent abilities for 5000 students, found in the data frame `theta_true`, were sampled from $\mathcal{N}(0, \Sigma)$. Here, Σ specifies the correlations between the 3 abilities, and is found in the data frame `correlation_matrix`. Discrimination and difficulty paramters were sampled uniformly from $[0.25, 1.75]$ and $[-3, 3]$ respectively, and entries in the Q-matrix were sampled from Bern(0.35). These values can be found in the data frames `disc_true`, `diff_true`, and `q_matrix`. Probabilities for each student answering each question correctly were calculated with the ML2P model [5]. These probabilities were sampled from to generate a response to each item on the assessment for each student. This is the main piece of data used for training, and is found in the data frame `responses`.

Examples

In this section, an example script is given showing how to construct, train, and evaluate an ML2P-VAE model. Three ML2P-VAE models are constructed, and all use the same data described previously. Each model yields estimates to item parameters and student ability parameters.

The first uses `build_vae_standard_normal()`, and assumes that the latent traits are independent of one another. We know that for the data provided, this assumption is false - as the responses were generated dependent on the matrix Σ. But `build_vae_standard_normal()` can be useful if the true correlations between latent traits are not known, i.e. `correlation_matrix` is not given.

The second ML2P-VAE model makes stronger assumptions - that the exact relationship between abilities is known (and given). `build_vae_normal_full_covariance()` encodes the response set to reflect the known correlations. On this data, it is unsurprising that this model yields better results than the first model, since it uses additional information about the distribution of the latent traits.

The last model again uses `build_vae_standard_normal()`, but demonstrates some other uses of the software. We demonstrate that this software can be used create a model with a single latent trait θ, so that single-dimensional IRT models can be estimated. We also specify `model_type = 1` here, which trains a 1-parameter logistic model, fixing the decoder weights to be equal to 1. We know both of these assumptions to be false for the exported dataset, so we don't expect great results. Also note that the assumptions `num_skills = 1` and `model_type = 1` do not both need to be used together, and that a 1-parameter logistic model would also work with `build_vae_full_covariance()`.

A demo on constructing, training, and evaluating ML2P-VAE models

```
# Load data
data <- as.matrix(responses)
Q <- as.matrix(q_matrix)

# Model parameters
num_items <- as.double(dim(Q)[2])
num_skills <- as.double(dim(Q)[1])
```

```
num_students <- dim(data)[1]
means <- rep(0,num_skills)
enc_arch <- c(16L, 8L)
enc_act <- c('relu', 'tanh')
out_act <- 'sigmoid'
kl <- 1
```

Model 1: An ML2P-VAE model assuming latent traits are independent

```
models_ind <- build_vae_standard_normal(num_items,
                                        num_skills,
                                        Q,
                                        model_type = 2,
                                        enc_hid_arch = enc_arch,
                                        hid_enc_activation = enc_act,
                                        output_activation = out_act)
encoder_ind <- models_ind[[1]]
decoder_ind <- models_ind[[2]]
vae_ind <- models_ind[[3]]
encoder_ind
#> Model
#> Model: "functional_1"
#> ____________________________________________________________________________
#> Layer (type)              Output Shape      Param #  Connected to
#> ============================================================================
#> input (InputLayer)        [(None, 30)]      0
#> ____________________________________________________________________________
#> hidden_1 (Dense)          (None, 16)        496      input[0][0]
#> ____________________________________________________________________________
#> hidden_2 (Dense)          (None, 8)         136      hidden_1[0][0]
#> ____________________________________________________________________________
#> z_mean (Dense)            (None, 3)         27       hidden_2[0][0]
#> ____________________________________________________________________________
#> z_log_var (Dense)         (None, 3)         27       hidden_2[0][0]
#> ____________________________________________________________________________
#> z (Concatenate)           (None, 6)         0        z_mean[0][0]
#>                                                      z_log_var[0][0]
#> ____________________________________________________________________________
#> lambda (Lambda)           (None, 3)         0        z[0][0]
#> ============================================================================
#> Total params: 686
#> Trainable params: 686
#> Non-trainable params: 0
#> ____________________________________________________________________________
decoder_ind
#> Model
#> Model: "functional_3"
#> _________________________________________________________________
#> Layer (type)                 Output Shape              Param #
#> =================================================================
#> latent_inputs (InputLayer)   [(None, 3)]               0
#> _________________________________________________________________
```

```
#> vae_out (Dense)                      (None, 30)                      120
#> ================================================================================
#> Total params: 120
#> Trainable params: 120
#> Non-trainable params: 0
#> ________________________________________________________________________________
vae_ind
#> Model
#> Model: "functional_5"
#> ________________________________________________________________________________
#> Layer (type)                         Output Shape                    Param #
#> ================================================================================
#> input (InputLayer)                   [(None, 30)]                    0
#> ________________________________________________________________________________
#> functional_1 (Functional)            [(None, 3), (None, 3), (None, 3 686
#> ________________________________________________________________________________
#> functional_3 (Functional)            (None, 30)                      120
#> ================================================================================
#> Total params: 806
#> Trainable params: 806
#> Non-trainable params: 0
#> ________________________________________________________________________________
```

Model 2: An ML2P-VAE model assuming correlation among traits is known

```
# Need to load in covariance matrix for this model type
cov <- as.matrix(correlation_matrix)
models_cor <- build_vae_normal_full_covariance(num_items,
                                               num_skills,
                                               Q,
                                               model_type = 2,
                                               mean_vector = means,
                                               covariance_matrix = cov,
                                               enc_hid_arch = enc_arch,
                                               hid_enc_activations = enc_act,
                                               output_activation = out_act,
                                               kl_weight = kl)
encoder_cor <- models_cor[[1]]
decoder_cor <- models_cor[[2]]
vae_cor <- models_cor[[3]]
encoder_cor
#> Model
#> Model: "functional_7"
#> ________________________________________________________________________________
#> Layer (type)              Output Shape      Param #  Connected to
#> ================================================================================
#> input (InputLayer)        [(None, 30)]      0
#> ________________________________________________________________________________
#> hidden_1 (Dense)          (None, 16)        496      input[0][0]
#> ________________________________________________________________________________
#> hidden_2 (Dense)          (None, 8)         136      hidden_1[0][0]
#> ________________________________________________________________________________
```

```
#> z_mean (Dense)              (None, 3)         27        hidden_2[0][0]
#> __________________________________________________________________________
#> z_log_cholesky (Dense)      (None, 6)         54        hidden_2[0][0]
#> __________________________________________________________________________
#> z (Concatenate)             (None, 9)         0         z_mean[0][0]
#>                                                         z_log_cholesky[0][0]
#> __________________________________________________________________________
#> lambda_1 (Lambda)           (None, 3)         0         z[0][0]
#> ==========================================================================
#> Total params: 713
#> Trainable params: 713
#> Non-trainable params: 0
#> __________________________________________________________________________
decoder_cor
#> Model
#> Model: "functional_9"
#> _________________________________________________________________
#> Layer (type)                 Output Shape              Param #
#> =================================================================
#> latent_inputs (InputLayer)   [(None, 3)]               0
#> _________________________________________________________________
#> vae_out (Dense)              (None, 30)                120
#> =================================================================
#> Total params: 120
#> Trainable params: 120
#> Non-trainable params: 0
#> _________________________________________________________________
vae_cor
#> Model
#> Model: "functional_11"
#> _________________________________________________________________
#> Layer (type)                 Output Shape              Param #
#> =================================================================
#> input (InputLayer)           [(None, 30)]              0
#> _________________________________________________________________
#> functional_7 (Functional)    [(None, 3), (None, 6), (None, 3 713
#> _________________________________________________________________
#> functional_9 (Functional)    (None, 30)                120
#> =================================================================
#> Total params: 833
#> Trainable params: 833
#> Non-trainable params: 0
#> _________________________________________________________________
```

Model 3: A single dimensional 1-PL model

```
# For single dimensional IRT, connect all items to the one skill
q_ones <- matrix(rep(1, num_items), nrow = 1, ncol = num_items)
models_1pl <- build_vae_standard_normal(num_items,
                                        num_skills = 1,
                                        Q = q_ones,
                                        model_type = 1,
```

```
                                    enc_hid_arch = enc_arch,
                                    hid_enc_activation = enc_act,
                                    output_activation = out_act)
encoder_1pl <- models_1pl[[1]]
decoder_1pl <- models_1pl[[2]]
vae_1pl <- models_1pl[[3]]
encoder_1pl
#> Model
#> Model: "functional_13"
#> ________________________________________________________________________________
#> Layer (type)                Output Shape              Param #   Connected to
#> ================================================================================
#> input (InputLayer)          [(None, 30)]              0
#> ________________________________________________________________________________
#> hidden_1 (Dense)            (None, 16)                496       input[0][0]
#> ________________________________________________________________________________
#> hidden_2 (Dense)            (None, 8)                 136       hidden_1[0][0]
#> ________________________________________________________________________________
#> z_mean (Dense)              (None, 1)                 9         hidden_2[0][0]
#> ________________________________________________________________________________
#> z_log_var (Dense)           (None, 1)                 9         hidden_2[0][0]
#> ________________________________________________________________________________
#> z (Concatenate)             (None, 2)                 0         z_mean[0][0]
#>                                                                 z_log_var[0][0]
#> ________________________________________________________________________________
#> lambda_2 (Lambda)           (None, 1)                 0         z[0][0]
#> ================================================================================
#> Total params: 650
#> Trainable params: 650
#> Non-trainable params: 0
#> ________________________________________________________________________________
decoder_1pl
#> Model
#> Model: "functional_15"
#> _________________________________________________________________
#> Layer (type)                 Output Shape              Param #
#> =================================================================
#> latent_inputs (InputLayer)   [(None, 1)]               0
#> _________________________________________________________________
#> vae_out (Dense)              (None, 30)                60
#> =================================================================
#> Total params: 60
#> Trainable params: 60
#> Non-trainable params: 0
#> _________________________________________________________________
vae_1pl
#> Model
#> Model: "functional_17"
#> _________________________________________________________________
#> Layer (type)                 Output Shape              Param #
#> =================================================================
#> input (InputLayer)           [(None, 30)]              0
#> _________________________________________________________________
```

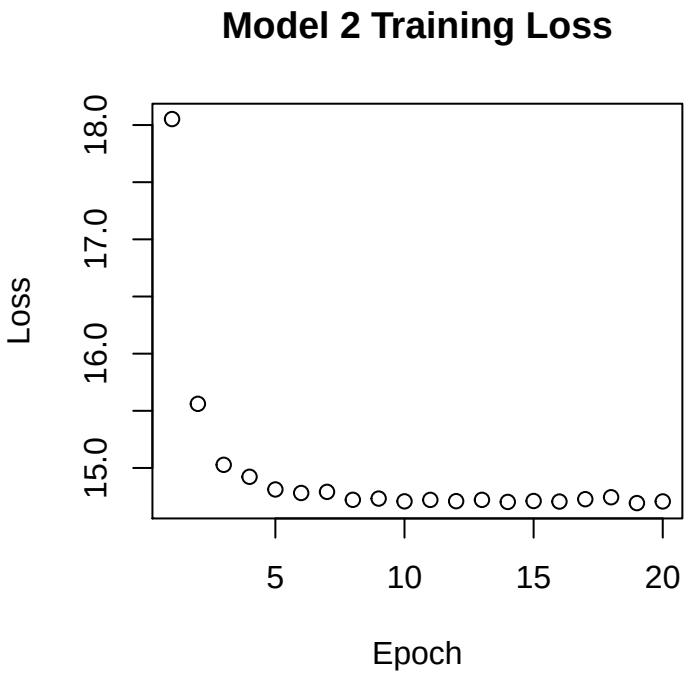
```
#> functional_13 (Functional)          [(None, 1), (None, 1), (None, 1 650
#> ________________________________________________________________________________
#> functional_15 (Functional)          (None, 30)                        60
#> ================================================================================
#> Total params: 710
#> Trainable params: 710
#> Non-trainable params: 0
#> ________________________________________________________________________________
```

Training

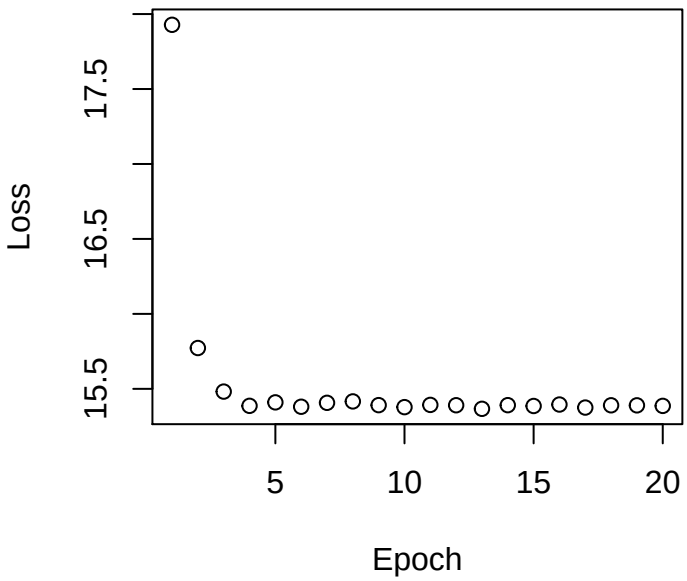
```
# Training parameters
num_train <- floor(0.8 * num_students)
num_test <- num_students - num_train
data_train <- data[1:num_train,]
data_test <- data[(num_train+1):num_students,]
num_epochs <- 20
batch_size <-2
```

```
# Train Model 1
history_ind <- train_model(vae_ind, data_train, num_epochs = num_epochs, batch_size = batch_size, verbos
plot(history_ind,
     main = 'Model 1 Training Loss',
     sub = '',
     xlab = 'Epoch',
     ylab = 'Loss')
```

```r
# Train Model 2
history_cor <- train_model(vae_cor, data_train, num_epochs = 20, batch_size = batch_size, verbose = 0)
plot(history_cor,
     main = 'Model 2 Training Loss',
     sub = '',
     xlab = 'Epoch',
     ylab = 'Loss')
```




```r
# Train Model 3
history_1pl <- train_model(vae_1pl, data_train, num_epochs = num_epochs, batch_size = batch_size, verbos
plot(history_1pl,
     main = 'Model 3 Training Loss',
     sub = ' ',
     xlab = 'Epoch',
     ylab = 'Loss')
```




```
# Get parameter estimates for Model 1
item_param_estimates_ind <- get_item_parameter_estimates(decoder_ind, model_type = 2)
diff_est_ind <- item_param_estimates_ind[[1]]
disc_est_ind <- item_param_estimates_ind[[2]]
test_theta_est_ind <- get_ability_parameter_estimates(encoder_ind, data_test)[[1]]
all_theta_est_ind <- get_ability_parameter_estimates(encoder_ind, data)[[1]]
```

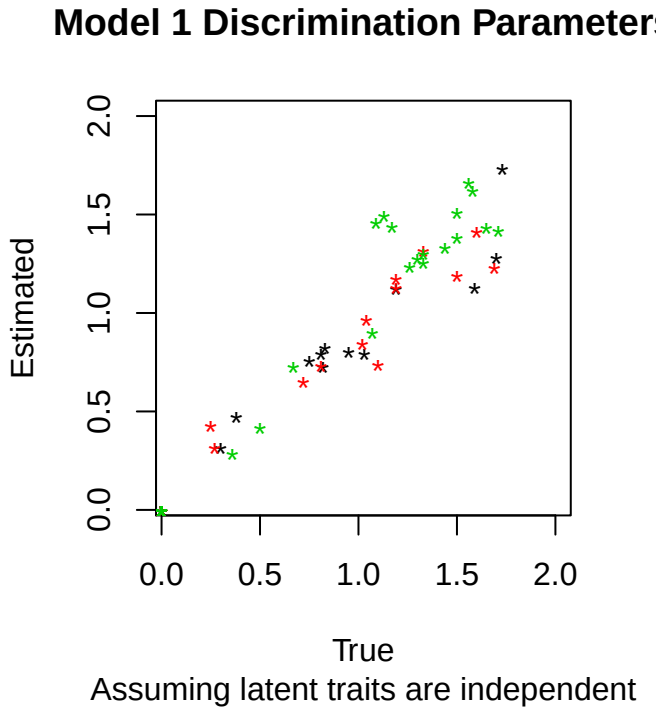
```
# Get parameter estimates for Model 2
item_param_estimates_cor <- get_item_parameter_estimates(decoder_cor, model_type = 2)
diff_est_cor <- item_param_estimates_cor[[1]]
disc_est_cor <- item_param_estimates_cor[[2]]
test_theta_est_cor <- get_ability_parameter_estimates(encoder_cor, data_test)[[1]]
all_theta_est_cor <- get_ability_parameter_estimates(encoder_cor, data)[[1]]
```

```
# Get parameter estimates for Model 3
item_param_estimates_1pl <- get_item_parameter_estimates(decoder_1pl, model_type = 1)
diff_est_1pl <- item_param_estimates_1pl[[1]]
# No discrimination parameters in this model
test_theta_est_1pl <- get_ability_parameter_estimates(encoder_1pl, data_test)[[1]]
all_theta_est_1pl <- get_ability_parameter_estimates(encoder_1pl, data)[[1]]
```

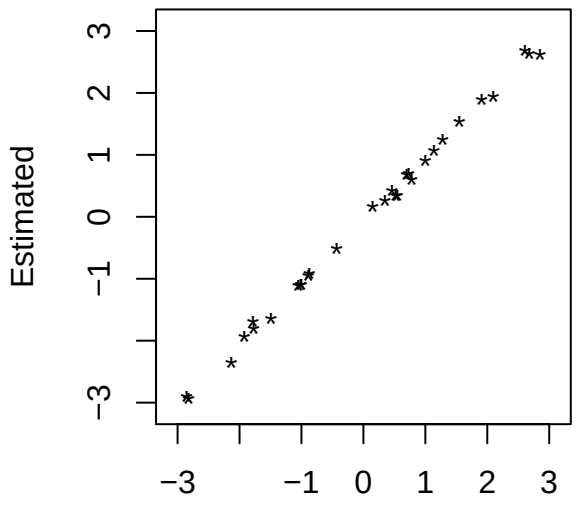
```
# Load in true values (included in this pacakge)
disc_true <- as.matrix(disc_true)
diff_true <- as.matrix(diff_true)
theta_true<- as.matrix(theta_true)
```

Results for each model

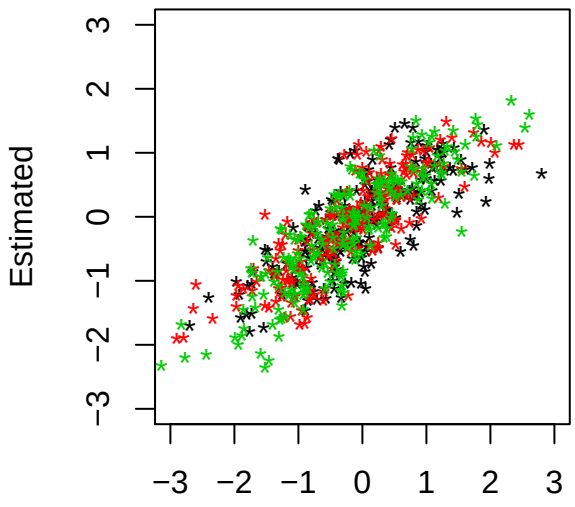
```
# Examine Model 1 estimates
par(pty="s")
matplot(t(disc_true), t(disc_est_ind), pch = '*',
        xlim = c(0.05, 2), ylim = c(0.05, 2),
        main = 'Model 1 Discrimination Parameters',
        sub = 'Assuming latent traits are independent',
        xlab = 'True', ylab = 'Estimated')
```



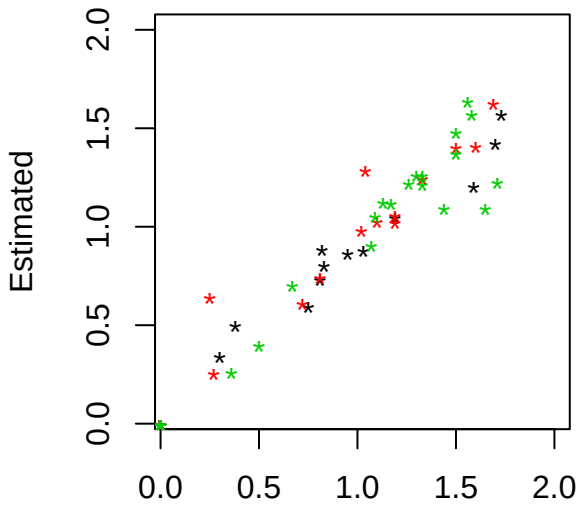

```
par(pty="s")
plot(diff_true, diff_est_ind, pch = '*',
     xlim = c(-3.1,3.1), ylim = c(-3.1,3.1),
     main = 'Model 1 Difficulty Parameters',
     sub = 'Assuming latent traits are independent',
     xlab = 'True', ylab = 'Estimated')
```



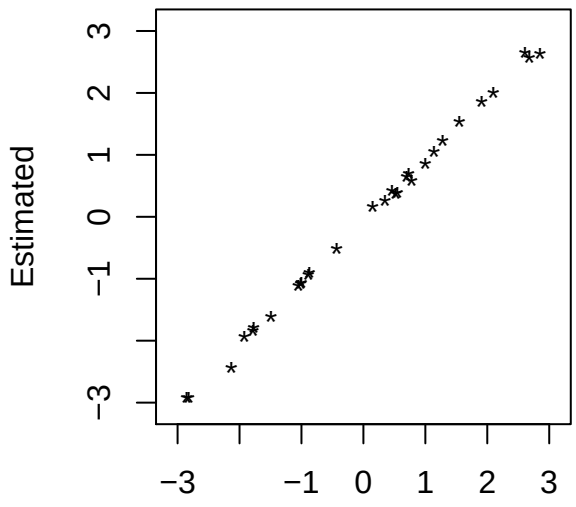

```
par(pty="s")
matplot(theta_true[4200:4400,], all_theta_est_ind[4200:4400,], pch = '*',
        xlim = c(-3,3), ylim = c(-3,3),
        main = 'Model 1 Ability Parameters',
        sub = 'Assuming latent traits are independent',
        xlab = 'True', ylab = 'Estimated')
```



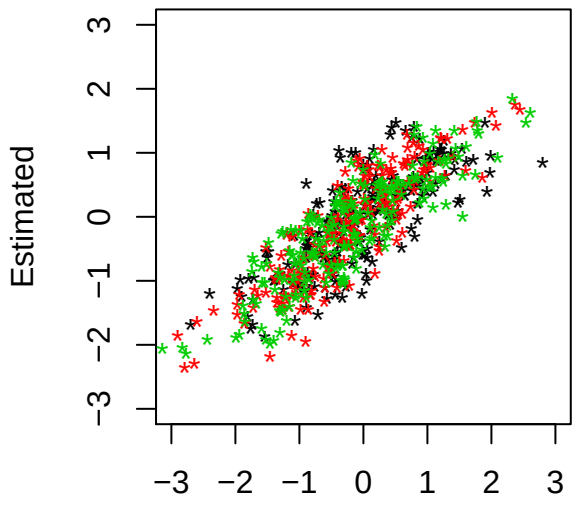

```
# Examine Model 2 estimates
par(pty="s")
matplot(t(disc_true), t(disc_est_cor), pch = '*',
        xlim = c(0.05, 2), ylim = c(0.05, 2),
        main = 'Model 2 Discrimination Parameters',
        sub = 'Assuming known correlation between latent traits',
        xlab = 'True', ylab = 'Estimated')
```



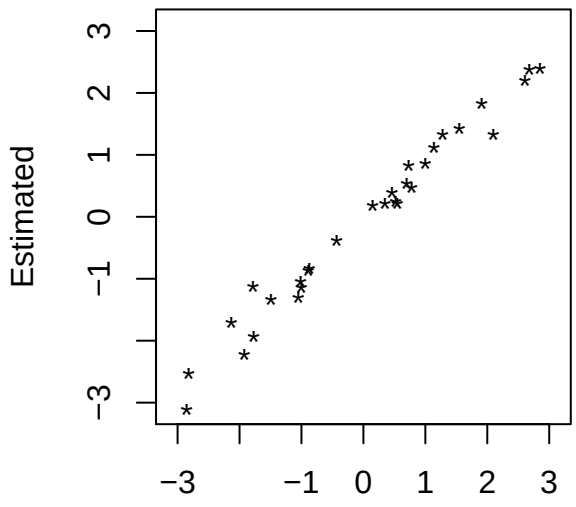

```
par(pty="s")
plot(diff_true, diff_est_cor, pch = '*',
     xlim = c(-3.1,3.1), ylim = c(-3.1,3.1),
     main = 'Model 2 Difficulty Parameters',
     sub = 'Assuming known correlation between latent traits',
     xlab = 'True', ylab = 'Estimated')
```



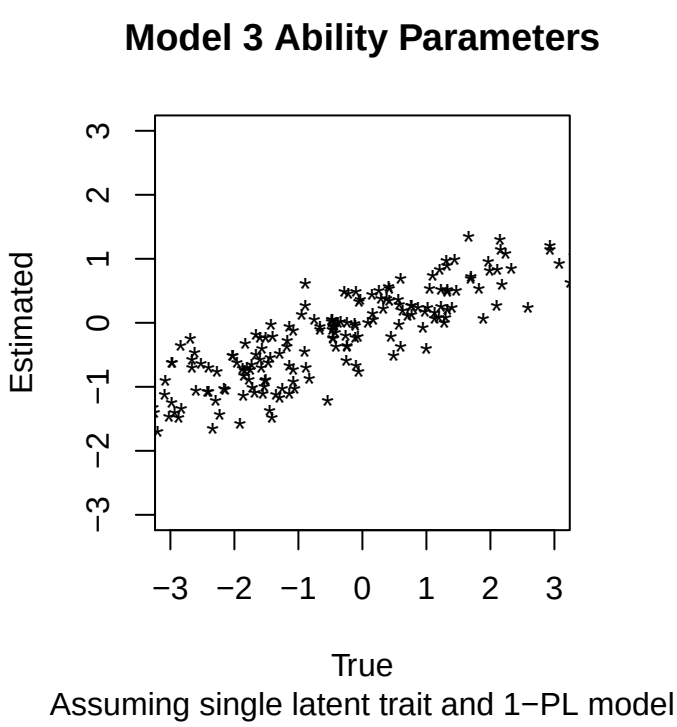

```
par(pty="s")
matplot(theta_true[4200:4400,], all_theta_est_cor[4200:4400,], pch = '*',
        xlim = c(-3,3), ylim = c(-3,3),
        main = 'Model 2 Ability Parameters',
        sub = 'Assuming known correlation between latent traits',
        xlab = 'True', ylab = 'Estimated')
```




```
# Examine Model 3 estimates
theta_summed <- rowSums(theta_true[4200:4400,])
par(pty="s")
plot(diff_true, diff_est_1pl, pch = '*',
     xlim = c(-3.1,3.1), ylim = c(-3.1,3.1),
     main = 'Model 3 Difficulty Parameters',
     sub = 'Assuming single latent trait and 1-PL model',
     xlab = 'True', ylab = 'Estimated')
```




```
par(pty="s")
plot(theta_summed, all_theta_est_1pl[4200:4400,1], pch = '*',
     xlim = c(-3,3), ylim = c(-3,3),
     main = 'Model 3 Ability Parameters',
     sub = 'Assuming single latent trait and 1-PL model',
     xlab = 'True', ylab = 'Estimated')
```



Discussion

In the above examples, we demonstrate how to build, train, and evaluate ML2P-VAE models for IRT parameter estimation. In this case, we knew the exact distribution of the simulated students latent traits, so it shouldn't be a surprise that the item and ability estimates are most accurate for Model 2. The structure of Model 1 assumes that latent traits are independent, and Model 3 assumes there is only one latent ability (uni-dimensional IRT), in addition to using a 1-parameter logistic model instead of a 2-parameter logistic model. Note that the two assumptions made in Model 3 could easily be implemented independent of one another.

In real applications, a known covariance matrix for the latent traits may not be known. In this case, one can either simply assume independence and use `build_vae_standard_normal()`, or estimate a covariance matrix from the data. This can be done by multiplying the (`num_students` × `num_items`) response matrix by the (`num_items` × `num_skills`) Q-matrix, and then taking the Pearson correlation of the columns of the resulting matrix. This will yield a (`num_skills` × `num_skills`) approximate correlation matrix which can be used in `build_vae_full_covariance()`.

References

[1] Converse, Curi, Oliveira. "Autoencoders for Educational Assessment." In proceedings of the Conference for Artificial Intelligence in Education (AIED), 2019.

[2] Curi, Converse, Hajewski, Oliveira. "Interpretable Variational Autoencoders for Cognitive Models." In proceedings of the International Joint Conference on Neural Networks, 2019.

[3] Kingma and Welling. "Auto-Encoding Variational Bayes." In proceedings of The International Conference on Learning Representations, 2014.

[4] Kullback and Leibler. “On information and sufficiency.” Annals of Mathematical Statistics, page 79–86, 1951.

[5] McKinley and Reckase. “The use of the General Rasch Model with Multidimensional Item Response Data.” American College Testing, 1980.

[6] Thissen and Wainer “Test Scoring”. Erlbaum Associates, Publishers, 2001.