Lab 6 - Bayesian Knowledge Tracing (BKT) and Variants

This tutorial is partially based on the pyBKT model tutorial and the Jupyter notebooks available on GitHub at https://github.com/CAHLR/pyBKT.

One notable application of machine learning in education is represented **knowledge inference models**, which aim to understand how well a student is learning concepts or skills. Being able to monitor this knowledge makes it possible to improve and personalize online learning platforms or intelligent tutoring systems, by focusing on areas the student is weak in and accelerating learning of certain concepts.

In this tutorial, we study a range of popular models for modelling students' knowledge based on **Bayesian Knowledge Tracing (BKT)**. BKT was introduced in 1995 as a means to model students' knowledge as a **latent variable** in online learning environments. Specifically, the environment can maintain an estimate of the **probability that the student has learned a set of skills**, which is statistically equivalent to a 2-node dynamic Bayesian network.

For this tutorial, we will rely on a Python implementation of the Bayesian Knowledge Tracing algorithm and more recent variants, estimating student cognitive mastery from problem solving sequences, known under the name of **pyBKT**. This package can be used to define and fit many BKT variants.

These variants are derived from a range of papers published in the educational data mining literature and, in this tutorial, we will provide you with the main notions and implementation details needed to investigate BKT models in practice.

Expected Tasks

- Follow the pyBKT getting started showcase.
- Solve a range of exercises on BKT models.

Learning Objectives

- Instantiate and run a pipeline on BKT models.
- Conduct fine-grained analyses on specific learning skills.
- Understand and experiment with different variants of BKT.
- Compare the performance of BKT setups under different evaluation methods.
- Inspect the influence of a BKT variant on the internal BKT parameters.

More information on the PyBKT is provided in the corresponding Github repository.

```
# Traditional packages
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
```

```
import math

%matplotlib inline
```

Introduction

BKT models operationalize the learning of a student as a **Markov process**, building upon the idea that, while students interact with an educational environment, their skill in a given concept improves. To move the theory behind BKT into practice, variables related to forgetting, learning, guessing, slipping, and so on need to be modelled, controlling for instance how fast and how well learning is happening for the student.

The BKT model assumes that the student's knowledge can be estimated by means of standardized questions, which can be answered correctly or incorrectly, on a concept or combination of concepts. BKT also assumes that initially a student may not know about a concept, but their knowledge gets better with learning and practice related to that concept. The following concepts will be

- P_0 is the initial probability of mastering that concept (skill).
- P_F is the probability that the student forgot something previously learned on the concept (skill).
- P_L is the probability that the student has learned something that was previous not known on the concept (skill).
- P_S is the probability that the student gave a wrong answer even though they had learned the concept (skill).
- P_G is the probability that the student guessed the right answer while not knowing the concept (skill).

In this tutorial, we will use a dataset of the student's responses to questions in a test, along with whether they answered correctly or incorrectly, and we will use a BKT model to find the values of the above probabilities.

The ASSISTments data set

ASSISTments is a free tool for assigning and assessing math problems and homework. Teachers can select and assign problem sets. Once they get an assignment, students can complete it at their own pace and with the help of hints, multiple chances, and immediate feedback. Teachers get instant results broken down by individual student or for the whole class. Please, find more information on the platform here.

In this tutorial, we will play with a simplified version of a dataset collected from the ASSISTments tool, saved on a CSV files with the following columns:

- user_id: The ID of the student doing the problem.
- template_id: The ID of the template in ASSISTment (assistments with the same template ID have similar questions).

- assistment_id: The ID of the ASSISTment (an assistment consists of one or more problems).
- order_id: These IDs are chronological and refer to the id of the original problem log.
- problem_id: The ID of the problem.
- skill_name: Skill name associated with the problem.
- correct: 1 if correct on the first attempt, 0 if incorrect on the first attempt or asked for help.
- ms_first_response: The time in milliseconds for the student's first response.
- attempt_count: Number of student attempts on this problem.
- hint_count: Number of student hints asked by the student on this problem.
- hint_total: Number of possible hints to be asked on this problem.

```
DATA_DIR = "./../../data/"
as_data = pd.read_csv(DATA_DIR + 'as_supersmall.csv',
encoding='latin', low_memory=False)

as_data.head(10)
```

```
   user_id  template_id  assistment_id  order_id  problem_id
skill_name  \
0    70733        30060          33175  35278766       51460  Box and
Whisker
1    70733        30060          33182  35278780       51467  Box and
Whisker
2    70733        30059          33107  35278789       51392  Box and
Whisker
3    70733        30060          33187  35278802       51472  Box and
Whisker
4    70733        30059          33111  35278810       51396  Box and
Whisker
5    70872        30059          33136  32268742       51421  Box and
Whisker
6    70872        30799          33144  32268764       51429  Box and
Whisker
7    72059        30799          33155  33409110       51440  Box and
Whisker
8    72059        30060          33181  33409165       51466  Box and
Whisker
9    72059        30060          33168  33409366       51453  Box and
Whisker

   correct  ms_first_response  attempt_count  hint_count  hint_total
0        0               9575              2           0           4
1        1               6422              1           0           4
2        0              11365              3           0           3
3        1               4412              1           0           4
4        1               6902              1           0           3
5        1               7281              1           0           3
6        1               7234              1           0           3
```

```
7         0              38290              2           0           3
8         0               8366              4           0           4
9         1               9661              1           0           4
```

Before delving into the pyBKT description and showcase, we invite you to spend some time to explore the toy dataset presented in this tutorial, e.g., how many students/problems/skills are included, examine the skills in more detail etc. Here, you could therefore add one or more cells to perform your exploration.

The pyBKT Package

In this tutorial, we use the pyBKT package, a Python implementation of the Bayesian Knowledge Tracing algorithm and variants, estimating student cognitive mastery from problem solving sequences. First, we install the package:

```
%pip install pyBKT
```

```
Defaulting to user installation because normal site-packages is not
writeable
Requirement already satisfied: pyBKT in /usr/local/lib/python3.8/dist-
packages (1.4)
Requirement already satisfied: requests in
/usr/local/lib/python3.8/dist-packages (from pyBKT) (2.28.1)
Requirement already satisfied: sklearn in
/usr/local/lib/python3.8/dist-packages (from pyBKT) (0.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.8/dist-
packages (from pyBKT) (1.22.4)
Requirement already satisfied: pandas in
/usr/local/lib/python3.8/dist-packages (from pyBKT) (1.4.3)
Requirement already satisfied: python-dateutil>=2.8.1 in
/usr/local/lib/python3.8/dist-packages (from pandas->pyBKT) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in
/usr/local/lib/python3.8/dist-packages (from pandas->pyBKT) (2022.2.1)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.8/dist-packages (from requests->pyBKT)
(2022.6.15)
Requirement already satisfied: charset-normalizer<3,>=2 in
/usr/local/lib/python3.8/dist-packages (from requests->pyBKT) (2.1.1)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in
/usr/local/lib/python3.8/dist-packages (from requests->pyBKT)
(1.26.12)
Requirement already satisfied: idna<4,>=2.5 in
/usr/local/lib/python3.8/dist-packages (from requests->pyBKT) (3.3)
Requirement already satisfied: scikit-learn in
/usr/local/lib/python3.8/dist-packages (from sklearn->pyBKT) (1.0.2)
Requirement already satisfied: six>=1.5 in /usr/lib/python3/dist-
packages (from python-dateutil>=2.8.1->pandas->pyBKT) (1.14.0)
Requirement already satisfied: joblib>=0.11 in
/usr/local/lib/python3.8/dist-packages (from scikit-learn->sklearn-
>pyBKT) (1.1.0)
```

```
Requirement already satisfied: scipy>=1.1.0 in
/usr/local/lib/python3.8/dist-packages (from scikit-learn->sklearn-
>pyBKT) (1.9.1)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/usr/local/lib/python3.8/dist-packages (from scikit-learn->sklearn-
>pyBKT) (3.1.0)
Note: you may need to restart the kernel to use updated packages.
```

Then, we can import the core class provided by the package, that is Model.

```
from pyBKT.models import Model
```

The first step is to construct a BKT model. To be instantiated, a BKT model requires a series of parameters, whose default value and meaning is provided below (e.g., num_fits, seed, defaults, and any model variant(s) that may be used). Each parameter can be modified during fit/crossvalidation time too.

- **Defaults generic parameters**:
  - num_fits (5) is the number of initialization fits used for the BKT model.
  - defaults (None) is a dictionary that can be used to pass values different than the default ones during initialization.
  - parallel (True) indicates whether the computation will use multi-threading.
  - skills ('.*') is a regular expression used to indicate the skills the BKT model will be run on.
  - seed (random.randint(0, 1e8)) is a seed that can be setup to enable reproducible experiments.
  - folds (5) is the number of folds used in case of cross-validation.
  - forgets (False) indicates whether the model will consider that the student may give a wrong answer even though they had learned the concept.
- **Defaults additional parameters**:
  - order_id ('order_id') is the name of the CSV column for the chronological IDs that refer to the original problem log.
  - skill_name ('skill_name') is the name of the CSV column for the skill name associated with the problem.
  - correct ('correct') is the name of the CSV column for the correct / incorrect label on the first attempt.
  - user_id ('user_id') is the name of the CSV column for the ID of the student doing the problem.
  - multilearn ('template_id') is the name of the column for checking whether there is a multi-skill object.
  - multiprior ('correct') is the name of the CSV column for mapping multi-prior knowledge.
  - multigs ('template_id') is the name of the CSV column corresponding to the desired guess/slip classes.

- **Initializers for learnable parameters**:
  - 'prior' (None, no inizialization) is the initial probability of answering the question correct.
  - 'learns' (None, no inizialization) is the probability that the student has learned something that was previous not known.
  - 'guesses' (None, no inizialization) is the probability that the student guessed the right answer while not knowing the concept.
  - 'slips' (None, no inizialization) is the probability that the student gave a wrong answer even though they had learned the concept.
  - 'forgets' (None, no inizialization) is the probability that the student forgot something previously learned.

If you have doubts on the meaning of certain parameters, please ask to TAs or move on the next examples (they will help you understand).

```
model = Model(seed=0)
model
```

```
Model(parallel=True, num_fits=5, seed=0, defaults=None)
```

The Model class is inspired by scikit-learn and, therefore, provides a range of methods a model can be called with:

- The **fit** method fits a BKT model given model and data information. Takes arguments skills, number of initialization fits, default column names (i.e. correct, skill_name), parallelization, and model types.
- The **predict** method predicts using the trained BKT model and test data information. Takes test data path or DataFrame as arguments. Returns a dictionary mapping skills to predicted values for those skills. Note that the predicted values are a tuple of (correct_predictions, state_predictions).
- The **evaluate** method evaluates a BKT model given model and data information. Takes a metric and data path or DataFrame as arguments. Returns the value of the metric for the given trained model tested on the given data.
- The **crossvalidate** method crossvalidates (trains and evaluates) the BKT model. Takes the data, metric, and any arguments that would be passed to the fit function (skills, number of initialization fits, default column names, parallelization, and model types) as arguments.

We will show a range of examples for each of the above methods.

Fitting and evaluating a model

```
model = Model(seed=0)
%time model.fit(data=as_data, skills='Box and Whisker')
%time model.evaluate(data=as_data, metric='auc')
```

```
CPU times: user 682 ms, sys: 12 ms, total: 694 ms
Wall time: 311 ms
```

```
CPU times: user 152 ms, sys: 0 ns, total: 152 ms
Wall time: 94.6 ms
```

```
0.5674965857758167
```

First, we have fitted a BKT model on the 'Box and Whisker' skill and, then, evaluate the corresponding **training AUC** (0.64). Note that we have run the BKT fitting process on the full dataset, to understand how well the BKT model can fit the data. Evaluation methods like cross-validation will be presented later in this notebook. Furthermore, the default metric displayed is RMSE, but pyBKT supports AUC ('auc'), RMSE ('rmse'), and accuracy ('accuracy') as metrics. We will also see how to add other metrics.

For each skill, you can get the learned parameters for 'prior', 'learns', 'guesses', 'slips', and 'forgets'. Specifically:

- **prior** (P_0¿ the prior probability of "knowing".
- **forgets** (P_F¿: the probability of transitioning to the "not knowing" state given "known".
- **learns** (P_L): the probability of transitioning to the "knowing" state given "not known".
- **slips** (P_S): the probability of picking incorrect answer, given "knowing" state.
- **guesses** (P_G): the probability of guessing correctly, given "not knowing" state.

```
model.coef_
```

```
{'Box and Whisker': {'prior': 0.8533977868556804,
  'learns': array([0.19638124]),
  'guesses': array([0.24682242]),
  'slips': array([0.21955686]),
  'forgets': array([0.])}}
```

```
model.params()
```

			value
skill	param	class	
Box and Whisker	prior	default	0.85340
	learns	default	0.19638
	guesses	default	0.24682
	slips	default	0.21956
	forgets	default	0.00000

We could initialize the prior knowledge to $1e-40$ for Box and Whisker, before fitting the model.

```
model = Model(seed=0)

model.coef_ = {'Box and Whisker': {'prior': 1e-40}}
model.coef_
```

```
{'Box and Whisker': {'prior': 1e-40}}
```

Then, we can fit the model and observe the resulting AUC score. How does it compares to the AUC score of the previous model.

```
%time model.fit(data=as_data, skills='Box and Whisker')
%time model.evaluate(data=as_data, metric='auc')
```

```
CPU times: user 261 ms, sys: 0 ns, total: 261 ms
Wall time: 130 ms
CPU times: user 112 ms, sys: 0 ns, total: 112 ms
Wall time: 48.8 ms
```

```
0.5535245298875933
```

```
model.params()
```

```
                                  value
skill           param   class
Box and Whisker prior   default 0.00000
                learns  default 0.56145
                guesses default 0.62351
                slips   default 0.22136
                forgets default 0.00000
```

You can also train simple BKT models on different skills in the data set.

```
model = Model(seed=0)
%time model.fit(data=as_data, skills=['Box and Whisker', 'Scatter
Plot'])
%time model.evaluate(data=as_data, metric='auc')
```

```
CPU times: user 248 ms, sys: 0 ns, total: 248 ms
Wall time: 149 ms
CPU times: user 86.6 ms, sys: 0 ns, total: 86.6 ms
Wall time: 54.3 ms
```

```
0.6625740160222918
```

And, then, observed the learned parameters for each skill. Note that, when multiple skills are passed to fit, the method will run a fitting procedure for each skill, separately (in this case, we will have two BKT models).

```
model.params()
```

```
                                  value
skill           param   class
Box and Whisker prior   default 0.96146
                learns  default 0.01810
                guesses default 0.00036
                slips   default 0.23364
                forgets default 0.00000
Scatter Plot    prior   default 0.48923
                learns  default 0.56128
```

```
                guesses default 0.71140
                slips   default 0.03956
                forgets default 0.00000
```

You can also enable forgetting, by setting the corresponding parameter in the fit method.

```
model = Model(seed=0)
%time model.fit(data=as_data, skills='Box and Whisker', forgets=True)
%time model.evaluate(data=as_data, metric='auc')
```

```
CPU times: user 73.4 ms, sys: 0 ns, total: 73.4 ms
Wall time: 49.4 ms
CPU times: user 125 ms, sys: 0 ns, total: 125 ms
Wall time: 39.8 ms
```

```
0.5721189200546275
```

```
model.params()
```

```
                                  value
skill           param   class
Box and Whisker prior   default 0.72657
                learns  default 0.25751
                guesses default 0.36213
                slips   default 0.19171
                forgets default 0.01321
```

Or train a multiguess and slip BKT model on the same skills in the data set. The **multigs** model fits a different guess/slip rate for each class. Note that, with *multigs=True*, the guess and slip classes will be specified by the *template_id*. You can specify a custom column mapping by doing *multigs='column_name'*.

```
model = Model(seed=0)
%time model.fit(data=as_data, skills=['Box and Whisker'],
multigs=True)
%time model.evaluate(data=as_data, metric='auc')
```

```
CPU times: user 173 ms, sys: 0 ns, total: 173 ms
Wall time: 62.1 ms
CPU times: user 128 ms, sys: 0 ns, total: 128 ms
Wall time: 116 ms
```

```
0.7049059775186469
```

And finally, we show the BKT paramaters. By enabling *multigs*, the guess and slip classes will be specified by the template_id and, by setting *multigs=True*, the guess and slip classes will be specified by default by the template_id classes. Note that assistments with the same template ID have similar questions. What could you observe by looking at the different learned guesses and slips values below?

```
model.params()
```

```
                                value
skill           param   class
Box and Whisker prior   default 0.09283
                learns  default 0.07509
                guesses 30059   0.75371
                        30060   0.59704
                        30799   0.70751
                        63446   0.00000
                        63447   0.16724
                        63448   1.00000
                slips   30059   0.01865
                        30060   0.16663
                        30799   0.04409
                        63446   1.00000
                        63447   0.99707
                        63448   0.00000
                forgets default 0.00000
```

The **multilearn** model fits a different learn rate (and forget rate if enabled) rate for each class specified. Note that, with multilearn=True, the learn classes are specified by the *template_id*. You can specify a custom column mapping by doing *multilearn='column_name'*.

```
model = Model(seed=0)
%time model.fit(data=as_data, skills=['Box and Whisker'],
multilearn=True)
%time model.evaluate(data=as_data, metric='auc')
```

```
CPU times: user 108 ms, sys: 0 ns, total: 108 ms
Wall time: 55.3 ms
CPU times: user 119 ms, sys: 3.37 ms, total: 122 ms
Wall time: 40.4 ms
```

```
0.5694925937598487
```

Looking at the parameters, we will observe a `leanrs` score for each template_id (the class column in the paras dataframe). In this case, what could you observe by looking at the different `leanrs` values below?

```
model.params()
```

```
                                value
skill           param   class
Box and Whisker prior   default 0.94327
                learns  30059   0.02049
                        30060   0.42985
                        30799   0.25272
                        63446   0.84344
                        63447   0.66021
                        63448   0.25677
                guesses default 0.05763
                slips   default 0.23095
```

```
                forgets 30059   0.00000
                        30060   0.00000
                        30799   0.00000
                        63446   0.00000
                        63447   0.00000
                        63448   0.00000
```

You can also combine multiple variants, and use a different column to specify the different learn and forget classes. In this case, we use user_id, assuming that we are interested in learning the parameters for each student, and we also enable forgetting.

```
model = Model(seed=0)
%time model.fit(data=as_data, skills=['Box and Whisker'],
forgets=True, multilearn='user_id')
%time model.evaluate(data=as_data, metric='auc')
```

```
CPU times: user 777 ms, sys: 1.33 ms, total: 778 ms
Wall time: 368 ms
CPU times: user 106 ms, sys: 362 µs, total: 106 ms
Wall time: 121 ms
```

```
0.6405609832965647
```

Once we run a BKT model with `forgets=True` and `multilearn='user_id'`, we will observe individual scores for each student, as shown below.

```
model.params()
```

```
                                  value
skill           param   class
Box and Whisker prior   default 0.05051
                learns  70733   0.08695
                        70872   0.11881
                        72059   0.07787
                        79748   0.11881
                        79750   0.13948
                        79769   0.10011
                        81641   0.02619
                        82482   0.12312
                        82533   0.16158
                        83169   0.11881
                        84316   0.07571
                        85725   0.13006
                        86489   0.09430
                        91260   0.11881
                        91301   0.11881
                        91406   0.16158
                        91409   0.11659
                        91436   0.01063
                        91459   0.10580
                        96217   0.10079
```

```
          96242   0.07787
          96248   0.06080
          96249   0.06670
          96266   0.11881
          96268   0.08741
          96292   0.07408
          96294   0.11825
          96296   0.11028
guesses default 0.70234
slips   default 0.05860
forgets 70733   0.31890
          70872   0.23699
          72059   0.35471
          79748   0.23699
          79750   0.22281
          79769   0.26551
          81641   0.66799
          82482   0.26814
          82533   0.19007
          83169   0.23699
          84316   0.35968
          85725   0.23422
          86489   0.34166
          91260   0.23699
          91301   0.23699
          91406   0.19007
          91409   0.28950
          91436   0.78903
          91459   0.32115
          96217   0.27475
          96242   0.35471
          96248   0.40840
          96249   0.42812
          96266   0.23699
          96268   0.26268
          96292   0.36373
          96294   0.27470
          96296   0.26723
```

The best performing models are typically those that combine several useful variants, such as the multilearn and multiguess/slip class variants. After this lab session, you might be interested in testing with other skills and see whether this observations is true for other skills as well.

Make predictions

As we said, the predict method can be executed on the trained BKT model, obtaining a dictionary mapping skills to predicted values for those skills, namely correct_predictions (each score is between 0 and 1 that measures the extent to which the model thinks that the student will answer correctly to that question) and state_predictions (each score between 0

and 1 that measures the extent to which the student has mastered that skill, after that question).

Note that, in the example below, we have run the BKT fitting process on the full dataset, to understand how well the BKT model can fit the data. Evaluation methods like cross-validation will be presented slightly after in this notebook.

```
model = Model(seed=0)
%time model.fit(data=as_data, skills='Box and Whisker')
%time model.evaluate(data=as_data, metric='auc')
%time preds = model.predict(data=as_data)
```

```
CPU times: user 83.1 ms, sys: 4.64 ms, total: 87.7 ms
Wall time: 48.6 ms
CPU times: user 137 ms, sys: 0 ns, total: 137 ms
Wall time: 57.2 ms
CPU times: user 195 ms, sys: 0 ns, total: 195 ms
Wall time: 98 ms
```

```
preds[preds['skill_name']=='Box and Whisker'][['user_id', 'correct',
'correct_predictions', 'state_predictions']]
```

	user_id	correct	correct_predictions	state_predictions
0	70733	0	0.66780	0.81542
1	70733	1	0.59418	0.69635
2	70733	0	0.74813	0.94535
3	70733	1	0.70860	0.88142
4	70733	1	0.77085	0.98210
..	...	...	...	...
219	96296	1	0.78191	1.00000
220	96296	1	0.78191	1.00000
221	96296	1	0.78191	1.00000
222	96296	1	0.78191	1.00000
223	96296	1	0.78191	1.00000

```
[224 rows x 4 columns]
```

Note that, if the BKT model is asked to predict on skills not included in the training set, the output predictions for that skills will be a best effort guess of 0.5 for both the correct and state predictions.

Extend the evaluation

The pyBKT package makes also possible to extend the range of metrics you can compute while evaluating a BKT model. To this end, you need to define a custom function that, given true_vals (true values for the correct target) and pred_vals (the predicted values for the correct target), computes and returns the score corresponding to the desired metric.

```
def mae(true_vals, pred_vals):
    return np.mean(np.abs(true_vals - pred_vals))
```

```
%time model.evaluate(data=as_data, metric=mae)
```

```
CPU times: user 200 ms, sys: 1.41 ms, total: 201 ms
Wall time: 60.9 ms
```

```
0.3699436448390422
```

Perform cross validation

Finally, the pyBKT package offers also a cross-validation method. You can specify the number of folds, a seed, and a metric (one of the three default ones, namely 'rmse', 'auc' or 'accuracy', or a custom Python function as we have see above). Furthermore, simialrly to the fit method, arguments for cross-validation a BKT variant and for defining the data path/data and skill names are accepted.

```
model = Model(seed=0)
%time model.crossvalidate(data=as_data, skills='Box and Whisker',
folds=5, metric='auc')
```

```
CPU times: user 3.06 s, sys: 3.31 ms, total: 3.06 s
Wall time: 1.5 s
```

```
                    auc
skill
Box and Whisker 0.54551
```

In this showcase, we just opted to five folds due to the time constraints. In the other cases, you need to select an appropriate number of folds based on the data you are dealing with, as discussed in the lectures.

Exercises

That's your turn! We ask you to complete the following exercises. In case you do not finish them during the lab session, please feel free to complete later, at your earliest convenience. TAs are happy to address any question or doubts you might have.

Kindly note that the following exercises have the goal of supporting you in getting familiar with the library functions, and may not fully represent the sequences of steps and the design choices made in a real-world or homework scenario. Elements concerning the latter scenarios will be discussed during the session. Furthermore, due to running time constraints, the following exercises will be run in a train-test split or full data set mode, while we leave the opportunity to run them under a cross-validation setting after this lab session.

In all your models, we ask you to set the *seed* to 0, to let you reproduce the same results across different runs.

Note that the expected running time may vary according to the device or environment.

Question 1 [expected total time for BKT fitting: 2 mins]

- Fit a BKT model with default parameters on the full data set, only for the skill 'Addition and Subtraction Integers'.
- Compute the correct predictions from the BKT model, by using the predict method of the Model class.
- Manually calculate the RMSE between the true correct value and the predicted correct value (refer to Slide 51 of Lecture 4 to get the RMSE formula).
- Compare with the RMSE returned by the evaluate method of the BKT model.

```
### EXERCISE CELL ###
def rmse(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred) ** 2))

model = Model(seed=0)
%time model.fit(data=as_data, skills='Addition and Subtraction
Integers')
preds = model.predict(data=as_data)

preds_filtered = preds[preds['skill_name'].str.contains('Addition and
Subtraction Integers')]
manual_training_rmse = rmse(preds_filtered['correct'],
preds_filtered['correct_predictions'])

print('Manual RMSE:', manual_training_rmse)
print('pyBKT RMSE:', model.evaluate(data=as_data, metric='rmse'))
```

```
CPU times: user 2.53 s, sys: 15.3 ms, total: 2.55 s
Wall time: 1.28 s
Manual RMSE: 0.46390773499967014
pyBKT RMSE: 0.46390773499967064
```

Question 2 [expected total time for BKT fitting: 7 mins]

- Perform a user-based train-test split of the data, with 20% of the users in the test set.
- Fit the two BKT model variants on the training set, only for the skill 'Addition and Subtraction Integers'.
  - default;
  - forgets=True;
- Which model variant listed below has the highest test AUC for 'Addition and Subtraction Integers' in the test set?

```
### EXERCISE CELL ###
users = as_data['user_id'].unique()
users_train = list(np.random.choice(users, int(len(users) * 0.8),
replace=False))
users_test = list(set(users) - set(users_train))

X_train, X_test = as_data[as_data['user_id'].isin(users_train)],
as_data[as_data['user_id'].isin(users_test)]
```

```
models = {}

model = Model(seed=0, num_fits=1)
%time model.fit(data=X_train, skills='Addition and Subtraction
Integers')
models['simple'] = model.evaluate(data=X_test, metric='auc')

model = Model(seed=0, num_fits=1)
%time model.fit(data=X_train, skills='Addition and Subtraction
Integers', forgets=True)
models['forgets'] = model.evaluate(data=X_test, metric='auc')

df = pd.DataFrame(models.items())
df.columns = ['Model Type', 'AUC']
df.set_index('Model Type')
```

```
CPU times: user 447 ms, sys: 4.42 ms, total: 451 ms
Wall time: 182 ms
CPU times: user 733 ms, sys: 0 ns, total: 733 ms
Wall time: 388 ms
```

Model Type	AUC
simple	0.55297
forgets	0.53747

Question 3 [expected total time for BKT fitting: 3 mins]

- Bin values in the ms_first_response column in *as_data* to categories ('less than 10s', 'less than 20s', 'less than 30s','less than 40s', 'less than 50s', 'other').
- Fit BKT models with different learn rates, according to the ms_first_response categories above, on the full data set, only for the skill 'Addition and Subtraction Integers'. You need to play with the multilearn parameter of the BKT fit method.
- Create a bar plot to show the P_L (learns) value for each ms_first_response category above. You basically need to play with the dataframe returned by model.params(), to prepare the data to be shown in the plot.
- Does binned response time influence the P_L parameter for the skill 'Addition and Subtraction Integers'? Which bin result in the highest P_L scores?

```
### EXERCISE CELL ###

skill = 'Addition and Subtraction Integers'

# Binning
learn_maps = {0: 'less than 10s', 1: 'less than 20s', 2: 'less than
30s', 3: 'less than 40s', 4: 'less than 50s'}
as_data['bin_s_first_response'] = (as_data['ms_first_response'] // (10
* 1000)).map(learn_maps).fillna('other')
```

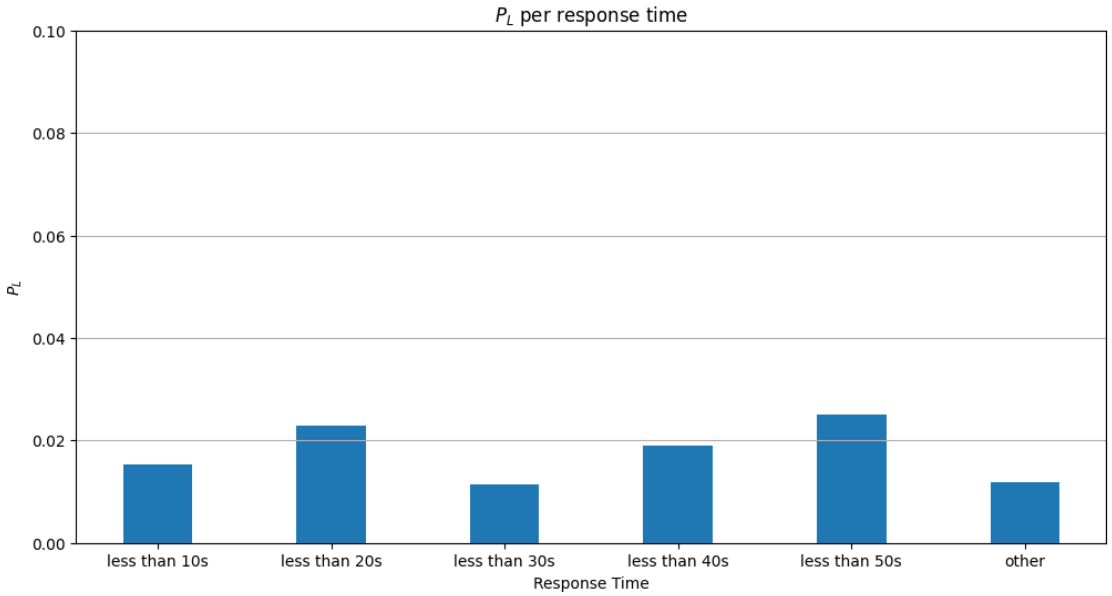
```
# Modelling
model = Model(seed=0, num_fits=1)
%time model.fit(data=as_data, skills=skill,
multilearn='bin_s_first_response')
params = model.params().reset_index()

params
```

```
CPU times: user 496 ms, sys: 0 ns, total: 496 ms
Wall time: 272 ms
```

	skill	param	class	value
0	Addition and Subtraction Integers	prior	default	0.74974
1	Addition and Subtraction Integers	learns	less than 10s	0.01536
2	Addition and Subtraction Integers	learns	less than 20s	0.02294
3	Addition and Subtraction Integers	learns	less than 30s	0.01146
4	Addition and Subtraction Integers	learns	less than 40s	0.01898
5	Addition and Subtraction Integers	learns	less than 50s	0.02503
6	Addition and Subtraction Integers	learns	other	0.01188
7	Addition and Subtraction Integers	guesses	default	0.37005
8	Addition and Subtraction Integers	slips	default	0.20730
9	Addition and Subtraction Integers	forgets	less than 10s	0.00000
10	Addition and Subtraction Integers	forgets	less than 20s	0.00000
11	Addition and Subtraction Integers	forgets	less than 30s	0.00000
12	Addition and Subtraction Integers	forgets	less than 40s	0.00000
13	Addition and Subtraction Integers	forgets	less than 50s	0.00000
14	Addition and Subtraction Integers	forgets	other	0.00000

```
plt.figure(figsize = (12, 6))
plt.title(r'$P_{L}$ per response time')
params_learns = params[params['param'] ==
'learns'].sort_values(by='class').copy()
labels = params_learns['class']
values = params_learns['value']
plt.bar(labels, values, width=0.4)
plt.ylabel(r'$P_{L}$')
plt.xlabel('Response Time')
plt.ylim([0, .1])
plt.grid(axis='y')
plt.show()
```



Question 4 [expected total time for BKT fitting: 8 mins]

- Use the same bins ms_first_response to categories ('less than 10s', 'less than 20s', 'less than 30s','less than 40s', 'less than 50s', 'other').
- Fit a BKT model with template-id multilearn (default), on the full data set, only for the skill 'Addition and Subtraction Integers'.
- Fit a BKT model with binned-response-time-based multilearn, on the full data set, only for the skill 'Addition and Subtraction Integers'.
- Does the binned-response-time-based multilearn improve the AUC of the model compared to the default template_id-based multilearn?

```
### EXERCISE CELL ###
skill = 'Addition and Subtraction Integers'

model = Model(seed=0, num_fits=1)
%time model.fit(data=as_data, skills=skill, multilearn=True)
default_multilearn_auc = model.evaluate(data=as_data, metric='auc')

model = Model(seed=0, num_fits=1)
%time model.fit(data=as_data, skills=skill,
multilearn='bin_s_first_response')
time_multilearn_auc = model.evaluate(data=as_data, metric='auc')

'AUC Improvement using Response Time:', time_multilearn_auc -
default_multilearn_auc
```

```
CPU times: user 455 ms, sys: 17.5 ms, total: 473 ms
Wall time: 162 ms
CPU times: user 420 ms, sys: 8.13 ms, total: 428 ms
Wall time: 205 ms

('AUC Improvement using Response Time:', -0.0031074282332050895)
```

Summary

In this tutorial, we have seen several important aspects of Bayesian Knowledge Tracing (BKT). We have shown how a typical data set for knowledge tracing should look like. We have illustrated how BKT models can be trained on different skills. We have shown how different variants of BKT can help you improve the goodness of your model. Many of the ideas described in this tutorial can be adapted to other data sets and projects. Finally, we have shown some examples of predictions and evaluations, covering also cross-validation. If you are interested in the implementation details of the different variants, we invite you to explore the codebase stored in the pyBKT Github repository.