Lab 07 Solution - Knowledge Tracing

Introduction

During the last lectures and lab session, you have dealt into one notable application of machine learning in education, namely knowledge tracing. Machine-learning models optimized for this task aim to understand how well a student is learning a portfolio of skills. Monitoring this knowledge by means of automated models allows to personalize online learning platforms, focusing the assessment on skills the student is weak in and accelerating learning of certain skills.

You are asked to work on the ASSISTment data set presented last week and to complete the following tasks:

- Compare three knowledge tracing models (BKT, AFM, PFA) in terms of AUC and RMSE.
- Generate and discuss the learning curves for a BKT model on a specific set of skills.

You can use pyBKT and pyAFM throughout this tutorial.

```
# Principal package imports
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
import scipy as sc

# Scikit-learn package imports
from sklearn import feature_extraction, model_selection, metrics

### YOUR ADDITIONAL IMPORT STATEMENTS BELOW (please, do not make any
imports elsewhere in the notebook) ###

# PyBKT package imports
from pyBKT.models import Model

# PyAFM package imports
from pyafm.custom_logistic import CustomLogistic
```

The 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. More information on the platform can be found here.

We will play with a simplified version of a dataset collected from the ASSISTments tool, saved in a CSV file with the following columns:

Name	Description
user_id	The ID of the student who is solving the problem.
order_id	The temporal ID (timestamp) associated with the student's answer to the problem.
problem_id	The ID of the problem.
skill_name	The name of the skill associated with the problem.
correct	The student's performance on the problem: 1 if the problem's answer is correct at the first attempt, 0 otherwise.
prior_success	The number of prior problems on that skill the student correctly answered at the first attempt.
prior_failure	The number of prior problems on that skill the student wrongly answered at the first attempt.

Load the data set.

```
DATA_DIR = "./../../data/"
data = pd.read_csv(DATA_DIR + 'as_hw_cmp.csv')
```

As a first step, we compute the total number of interactions, the number of unique students, and the number of unique skills.

```
len(data.index), len(data['user_id'].unique()),
len(data['skill_name'].unique())
```

```
(26409, 1014, 3)
```

We then also take a look at the skills included in the data set.

```
data['skill_name'].unique()
```

```
array(['Circle Graph', 'Venn Diagram', 'Mode'], dtype=object)
```

Finally, we show the first ten rows of the data dataframe, to have an idea of how the data looks like.

```
data.head(20)
```

```
    user_id  order_id  problem_id    skill_name  correct
prior_success  \
0        14  21617623       93383  Circle Graph        0
0
1        14  21617632       93407  Circle Graph        1
0
2        14  21617641       93400  Circle Graph        0
1
3        14  21617650       93419  Circle Graph        0
```

1
4 14 21617659 93420 Circle Graph 0
1
5 14 21617667 93415 Circle Graph 0
1
6 14 21617675 93423 Circle Graph 0
1
7 14 21617692 57695 Circle Graph 0
1
8 14 21617731 58596 Circle Graph 1
1
9 14 21617749 57647 Circle Graph 0
2
10 14 21617805 58566 Circle Graph 1
2
11 14 21617825 58551 Circle Graph 0
3
12 64525 28186893 92320 Circle Graph 1
0
13 64525 28187093 92335 Circle Graph 1
1
14 64525 32413158 92327 Circle Graph 1
2
15 64525 33022751 93432 Circle Graph 0
3
16 64525 33023039 93447 Circle Graph 1
3
17 64525 33023131 93448 Circle Graph 1
4
18 64525 33023183 93429 Circle Graph 1
5
19 64525 33023245 57689 Circle Graph 0
6

	prior_failure
0	0
1	1
2	1
3	2
4	3
5	4
6	5
7	6
8	7
9	7
10	8
11	8
12	0
13	0
14	0

```
15              0
16              1
17              1
18              1
19              1
```

1 Knowledge Tracing: Model Performance Comparison

In this section, we ask you to evaluate (i) a Bayesian Knowledge Tracing (BKT) model, (ii) an Additive Factor Model (AFM), and (iii) a Performance Factor Analysis (PFA) model on the skills 'Circle Graph', 'Venn Diagram', and 'Mode', by performing a user-stratified 10-fold cross validation and monitoring the Root Mean Squared Error (RMSE) and the Area Under the ROC Curve (AUC) as performance metrics. Then, we ask you to visually report the RMSE and AUC scores achieved by the three student's models in the user-stratified 10-fold cross validation, in such a way that the models' performance can be easily and appropriately compared against each other.

For your convenience, you will be guided in completing this section through seven main tasks:

- Task 1.1: Group k-fold initialization.
- Task 1.2: BKT evaluation.
- Task 1.3: AFM evaluation.
- Task 1.4: PFA evaluation.
- Task 1.5: Performance metrics plotting.
- Task 1.6: Performance metrics discussion.

Task 1.1

Given that the main objective of this homework section is to evaluate three student's knowledge tracing models under a `user-stratified 10-fold cross validation`, in this task, we ask you to complete the body of a function named `create_iterator`. This function should create an iterator object able to split student's interactions included in `data` in `10 folds` such that the same student does not appear in two different folds. To do so, you can appropriately initialize a scikit-learn's GroupKFold iterator with non-overlapping groups and returning the iterator, i.e., `model_selection.GroupKFold(...).split(...)`.

For convenience, we present you an illustrative example assuming that (i) you have four data samples and that (ii) the first two data samples belong to group 0 and the last two data samples belong to group 2. The data samples associated with a group should not appear in multiple folds or, in other words, the data samples associated with a group should appear all in the same fold. Please, find below a way to use the scikit-learn's GroupKFold object to

create folds that meet this property (here, we simulate this scenario by considering only a 2-fold creation strategy):

```
X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]]) y = np.array([1, 2, 3, 4]) groups = np.array([0, 0, 2, 2]) group_kfold = model_selection.GroupKFold(n_splits=2).split(X, y, groups)
```

Finally, we provide an illustrative example not related with the task on how this iterator can be then used to generate training and test folds:

```
for train_index, test_index in group_kfold:     print('TRAIN:', train_index, 'TEST:', test_index)     X_train, X_test = X[train_index], X[test_index]     y_train, y_test = y[train_index], y[test_index]     print(X_train, '-', X_test, '-', y_train, '-', y_test)
```

The above for loop generates the following output. It can be observed that the data samples belonging to a group all appear in the same fold, as expected.

```
TRAIN: [0 1] TEST: [2 3] [[1 2] [3 4]] - [[5 6] [7 8]] - [1 2] - [3 4]

TRAIN: [2 3] TEST: [0 1] [[5 6] [7 8]] - [[1 2] [3 4]] - [3 4] - [1 2]
```

Please, find more information about the GroupKFold iterator in the scikit-learn documentation.

```
def create_iterator(data):
    '''
    Create an iterator to split interactions in data in 10 folds, with the same student not appearing in two diverse folds.
    :param data:        Dataframe with student's interactions.
    :return:            An iterator.
    '''
    ### YOUR CODE HERE ###

    # Both passing a matrix with the raw data or just an array of indexes works
    X = np.arange(len(data.index))
    # Groups of interactions are identified by the user id (we do not want the same user appearing in two folds)
    groups = data['user_id'].values
    return model_selection.GroupKFold(n_splits=10).split(X, groups=groups)
```

Let's check the output of this function and a few properties of the iterator.

```
tested_user_ids = set()
for iteration, (train_index, test_index) in enumerate(create_iterator(data)):
    user_ids = data['user_id'].unique()
    train_user_ids = data.iloc[train_index]['user_id'].unique()
    test_user_ids = data.iloc[test_index]['user_id'].unique()
```

```
    print('Iteration:', iteration)
    print('Intersection between train and test user ids:',
set(train_user_ids) & set(test_user_ids))
    print('All user ids in train and test user union:',
len(set(train_user_ids).union(set(test_user_ids))) == len(user_ids))
    print('User ids tested more than once:', set(tested_user_ids) &
set(test_user_ids))
    tested_user_ids = tested_user_ids.union(set(test_user_ids))
    print()
```

```
Iteration: 0
Intersection between train and test user ids: set()
All user ids in train and test user union: True
User ids tested more than once: set()

Iteration: 1
Intersection between train and test user ids: set()
All user ids in train and test user union: True
User ids tested more than once: set()

Iteration: 2
Intersection between train and test user ids: set()
All user ids in train and test user union: True
User ids tested more than once: set()

Iteration: 3
Intersection between train and test user ids: set()
All user ids in train and test user union: True
User ids tested more than once: set()

Iteration: 4
Intersection between train and test user ids: set()
All user ids in train and test user union: True
User ids tested more than once: set()

Iteration: 5
Intersection between train and test user ids: set()
All user ids in train and test user union: True
User ids tested more than once: set()

Iteration: 6
Intersection between train and test user ids: set()
All user ids in train and test user union: True
User ids tested more than once: set()

Iteration: 7
Intersection between train and test user ids: set()
All user ids in train and test user union: True
User ids tested more than once: set()
```

```
Iteration: 8
Intersection between train and test user ids: set()
All user ids in train and test user union: True
User ids tested more than once: set()

Iteration: 9
Intersection between train and test user ids: set()
All user ids in train and test user union: True
User ids tested more than once: set()
```

```
len(tested_user_ids) == len(user_ids)
```

```
True
```

On a given iteration, no user appears in both training and test sets. The union of the users in both training and test sets given us the full set of user ids in the dataset. Each user appears in the test set exactly once.

Task 1.2

In this task, we ask you to evaluate a `BKT model` with all default parameters, namely `Model(seed=0)` in pyBKT, through a `10-fold user-stratified cross-validation`, computing the following performance metrics: `RMSE` and `AUC`. To do so, you should use the `create_iterator` function, defined in Task 1.2, to create the training and test set for each fold, starting from the interactions in `data`.

No plotting is needed, it is enough to print the scores for each metric in the cell.

Please, note that this task may require a long running time (e.g., about 40 to 90 minutes), depending on your implementation. Just as an indication, on a Dell XPS 13, one fold lasts around 7 minutes.

```
### YOUR CODE HERE (please, feel free to add extra cells to solve this
task, after this first one) ###
rmse_bkt, auc_bkt = [], []
for iteration, (train_index, test_index) in
enumerate(create_iterator(data)):
    # Split data in training and test sets
    X_train, X_test = data.iloc[train_index], data.iloc[test_index]
    # Initialize and fit the model
    model = Model(seed=0)
    %time model.fit(data=X_train)
    # Compute RMSE
    train_rmse = model.evaluate(data=X_train, metric='rmse')
    test_rmse = model.evaluate(data=X_test, metric='rmse')
    rmse_bkt.append(test_rmse)
    # Compute AUC
    train_auc = model.evaluate(data=X_train, metric='auc')
```

```
    test_auc = model.evaluate(data=X_test, metric='auc')
    auc_bkt.append(test_auc)
    # Print progress
    print('Iteration:', iteration, 'RMSE', (train_rmse, test_rmse),
'AUC', (train_auc, test_auc))
```

```
Wall time: 6min 30s
Iteration: 0 RMSE (0.3619326883451476, 0.3627377393397434) AUC
(0.8768635724803968, 0.8804119587687254)
Wall time: 8min 1s
Iteration: 1 RMSE (0.3623758521598453, 0.35652947004237456) AUC
(0.8777360935826011, 0.871462831416175)
Wall time: 7min 37s
Iteration: 2 RMSE (0.3614677583161969 3, 0.3685555348264736) AUC
(0.8785462996317557, 0.8645564407924567)
Wall time: 6min 32s
Iteration: 3 RMSE (0.36123289139100484, 0.3765145626874925) AUC
(0.8785240813856332, 0.8692037292316623)
Wall time: 7min 24s
Iteration: 4 RMSE (0.3612297575354389, 0.37934386367709344) AUC
(0.8772757259188736, 0.8785882024159103)
Wall time: 6min 37s
Iteration: 5 RMSE (0.36384158637003744, 0.3501882134966003) AUC
(0.8766541779809577, 0.885324470100699)
Wall time: 7min 37s
Iteration: 6 RMSE (0.36361727299904206, 0.3499285472259435) AUC
(0.8749834945119513, 0.8995316915740698)
Wall time: 5min 59s
Iteration: 7 RMSE (0.3614092428844653, 0.3662077388390171) AUC
(0.8796324144055722, 0.8541902758970246)
Wall time: 8min 12s
Iteration: 8 RMSE (0.36266788598266114, 0.36405419887577556) AUC
(0.8765203013778682, 0.884975143324418)
Wall time: 8min 17s
Iteration: 9 RMSE (0.3633233826910702, 0.35314683335159985) AUC
(0.8788799587872089, 0.8733444928021428)
```

Finally, we show the mean and the standard deviation of the RMSE and AUC across folds.

```
'RMSE', np.mean(rmse_bkt), np.std(rmse_bkt)
```

```
('RMSE', 0.3627206702362114, 0.009824846598894346)
```

```
'AUC', np.mean(auc_bkt), np.std(auc_bkt)
```

```
('AUC', 0.8761589236323284, 0.011948209404677711)
```

Task 1.3

In this task, we ask you to evaluate an `AFM model` with all default parameters (e.g., no custom bounds, default l2 regularization, and fit_intercept=True) through a `10-fold user-stratified cross-validation`, computing the following performance metrics: RMSE and AUC. To do so, exactly as you should have done for the BKT model in Task 1.3, you should use the `create_iterator` function, defined in Task 1.2, to create the training and test set for each fold, starting from the interactions in `data`.

No plotting is needed, it is enough to print the scores for each metric in the cell.

The following cells include some utility functions that are needed to generate the `X` and `y` data in a format that is accepted by pyAFM model objects. To complete this task, you can build on top of the `X` and `y` created for you with the following cells. Please, refer to Tutorial 6 for further information on pyAFM.

```
def read_as_student_step(data):
    skills, opportunities, corrects, user_ids = [], [], [], []

    for row_id, (_, row) in enumerate(data.iterrows()):

        # Get attributes for the current interaction
        user_id = row['user_id']
        skill_name = row['skill_name']
        correct = row['correct']
        prior_success = row['prior_success']
        prior_failure = row['prior_failure']

        # Update the number of opportunities this student had with
this skill
        opportunities.append({skill_name: prior_success +
prior_failure})

        # Update information in the current
        skills.append({skill_name: 1})

        # Answer info
        corrects.append(correct)

        # Student info
        user_ids.append({user_id: 1})

    return (skills, opportunities, corrects, user_ids)

def prepare_data_afm(skills, opportunities, corrects, user_ids):

    sv = feature_extraction.DictVectorizer()
    qv = feature_extraction.DictVectorizer()
    ov = feature_extraction.DictVectorizer()
    S = sv.fit_transform(user_ids)
```

```
    Q = qv.fit_transform(skills)
    O = ov.fit_transform(opportunities)
    X = sc.sparse.hstack((S, Q, O))
    y = np.array(corrects)

    return (X.toarray(), y)
```

Prepare the X and y arrays to be used to evaluate the AFM model.

```
%time skills, opportunities, corrects, user_ids =
read_as_student_step(data)
%time X, y = prepare_data_afm(skills, opportunities, corrects,
user_ids)
```

```
Wall time: 3.25 s
Wall time: 161 ms
```

```
### YOUR CODE HERE (please, feel free to add extra cells to solve this
task, after this first one) ###
rmse_afm, auc_afm = [], []
for iteration, (train_index, test_index) in
enumerate(create_iterator(data)):
    # Split data in training and test sets
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    # Initialize and fit the model
    afm = CustomLogistic()
    %time afm.fit(X_train, y_train)
    # Make predictions
    y_train_pred = afm.predict_proba(X_train)
    y_test_pred = afm.predict_proba(X_test)
    # Compute RMSE
    train_rmse = metrics.mean_squared_error(y_train, y_train_pred,
squared=False)
    test_rmse = metrics.mean_squared_error(y_test, y_test_pred,
squared=False)
    rmse_afm.append(test_rmse)
    # Compute AUC
    train_auc = metrics.roc_auc_score(y_train, y_train_pred)
    test_auc = metrics.roc_auc_score(y_test, y_test_pred)
    auc_afm.append(test_auc)
    # Print progress
    print('Iteration:', iteration, 'RMSE', (train_rmse, test_rmse),
'AUC', (train_auc, test_auc))
```

```
Wall time: 12.4 s
Iteration: 0 RMSE (0.3588795022813869, 0.45202131504519033) AUC
(0.8573893360851267, 0.6130214085464276)
Wall time: 9.38 s
Iteration: 1 RMSE (0.358820570463362, 0.4140457202925771) AUC
(0.8599132460943775, 0.6829414951498932)
```

```
Wall time: 11.4 s
Iteration: 2 RMSE (0.36066012598738856, 0.41602149060121024) AUC
(0.8557095628637055, 0.6852909560866977)
Wall time: 11.5 s
Iteration: 3 RMSE (0.35944514606534345, 0.44356724973709066) AUC
(0.8562528592307947, 0.6138551113690778)
Wall time: 12.3 s
Iteration: 4 RMSE (0.3575889560213242, 0.4341065247494938) AUC
(0.8578727813677345, 0.7078581737186332)
Wall time: 11.3 s
Iteration: 5 RMSE (0.36185077745423777, 0.3972852615444283) AUC
(0.8538340075189449, 0.7296433207291607)
Wall time: 15.1 s
Iteration: 6 RMSE (0.3592706434559734, 0.42327898547747866) AUC
(0.8566766924289391, 0.713221927205779)
Wall time: 12.7 s
Iteration: 7 RMSE (0.3579997163243513, 0.4150620859259244) AUC
(0.8608480096454952, 0.7035631340449515)
Wall time: 12.4 s
Iteration: 8 RMSE (0.35934539076985417, 0.412558505056829) AUC
(0.8565544841259949, 0.7380982429105336)
Wall time: 11.8 s
Iteration: 9 RMSE (0.36080917079239705, 0.39844331773368125) AUC
(0.8564244224956705, 0.6960964178105123)
```

Finally, we show the mean and the standard deviation of the RMSE and AUC across folds.

```
'RMSE', np.mean(rmse_afm), np.std(rmse_afm)
```

```
('RMSE', 0.4206390456163904, 0.0170239025795021)
```

```
'AUC', np.mean(auc_afm), np.std(auc_afm)
```

```
('AUC', 0.6883590187571667, 0.040906170725414366)
```

Task 1.4

In this task, we ask you to evaluate a `PFA model` with all default parameters (e.g., no custom bounds, default l2 regularization, and fit_intercept=True) through a `10-fold user-stratified cross-validation`, computing the following performance metrics: `RMSE` and `AUC`. To do so, exactly as you should have done for the BKT and AFM models in Task 1.3 and 1.4, you should use the `create_iterator` function, defined in Task 1.2, to create the training and test set for each fold, starting from the interactions in `data`.

No plotting is needed, it is enough to print the scores for each metric in the cell.

The following cells include some utility functions that are needed to generate the `X` and `y` data in a format that is accepted by pyAFM model objects. To complete this task, you can

build on top of the X and y created for you with the following cells. Please, refer to Tutorial 6 for further information on pyAFM.

```
def read_as_success_failure(data):
    n_succ, n_fail = [], []

    # Create the n_succ and n_fail variables required by pyAFM
    for i, row in data.iterrows():
        n_succ.append({row['skill_name']: int(row['prior_success'])})
        n_fail.append({row['skill_name']: int(row['prior_failure'])})

    return n_succ, n_fail

def prepare_data_pfa(skills, corrects, user_ids, n_succ, n_fail):

    s = feature_extraction.DictVectorizer()
    q = feature_extraction.DictVectorizer()
    succ = feature_extraction.DictVectorizer()
    fail = feature_extraction.DictVectorizer()
    S = s.fit_transform(user_ids)
    Q = q.fit_transform(skills)
    succ = succ.fit_transform(n_succ)
    fail = fail.fit_transform(n_fail)
    X = sc.sparse.hstack((S, Q, succ, fail))
    y = np.array(corrects)

    return (X.toarray(), y)
```

Prepare the X and y arrays to be used to evaluate the PFA model.

```
%time n_succ, n_fail = read_as_success_failure(data)
%time X, y = prepare_data_pfa(skills, corrects, user_ids, n_succ,
n_fail)
```

```
Wall time: 2.45 s
Wall time: 202 ms
```

```
### YOUR CODE HERE (please, feel free to add extra cells to solve this
task, after this first one) ###
rmse_pfa, auc_pfa = [], []
for iteration, (train_index, test_index) in
enumerate(create_iterator(data)):
    # Split data in training and test sets
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    # Initialize and fit the model
    pfa = CustomLogistic()
    %time pfa.fit(X_train, y_train)
    # Make predictions
    y_train_pred = pfa.predict_proba(X_train)
    y_test_pred = pfa.predict_proba(X_test)
```

```
    # Compute RMSE
    train_rmse = metrics.mean_squared_error(y_train, y_train_pred,
squared=False)
    test_rmse = metrics.mean_squared_error(y_test, y_test_pred,
squared=False)
    rmse_pfa.append(test_rmse)
    # Compute AUC
    train_auc = metrics.roc_auc_score(y_train, y_train_pred)
    test_auc = metrics.roc_auc_score(y_test, y_test_pred)
    auc_pfa.append(test_auc)
    # Print progress
    print('Iteration:', iteration, 'RMSE', (train_rmse, test_rmse),
'AUC', (train_auc, test_auc))
```

```
Wall time: 14.8 s
Iteration: 0 RMSE (0.3592612212713895, 0.41275331953657246) AUC
(0.8559670398622193, 0.7395491745249625)
Wall time: 18 s
Iteration: 1 RMSE (0.35885738244034426, 0.40379001344655385) AUC
(0.8588426867798364, 0.7158009486936392)
Wall time: 16.1 s
Iteration: 2 RMSE (0.3607598183660183, 0.40249078543884526) AUC
(0.8543376406280749, 0.7588350189847844)
Wall time: 15.4 s
Iteration: 3 RMSE (0.3597105765032166, 0.4269537850499162) AUC
(0.8548760790495966, 0.6866375244866865)
Wall time: 22.8 s
Iteration: 4 RMSE (0.35764536305385825, 0.4241454765292951) AUC
(0.856519188631309, 0.7512259604689446)
Wall time: 16.6 s
Iteration: 5 RMSE (0.362098709421017, 0.38602406435072795) AUC
(0.8523760148856947, 0.778039312118549)
Wall time: 19 s
Iteration: 6 RMSE (0.35953901533416016, 0.4052646257412933) AUC
(0.8551210748717755, 0.7672193517267152)
Wall time: 14.7 s
Iteration: 7 RMSE (0.3581739548411531, 0.4019915603509969) AUC
(0.8595009440354477, 0.7540208189912775)
Wall time: 17.3 s
Iteration: 8 RMSE (0.3595624198884521, 0.4019961800046421) AUC
(0.855078464451322, 0.7843115114157678)
Wall time: 16.1 s
Iteration: 9 RMSE (0.36104152124203504, 0.3871205028813061) AUC
(0.8552466760104702, 0.7484679444258453)
```

Finally, we show the mean and the standard deviation of the RMSE and AUC across folds.

```
'RMSE', np.mean(rmse_pfa), np.std(rmse_pfa)
```

```
('RMSE', 0.40525303133301493, 0.012702984340139625)
```

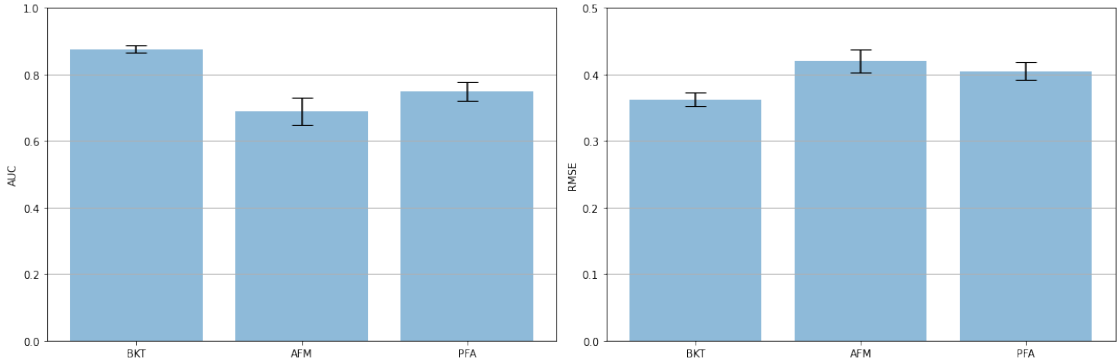
```
'AUC', np.mean(auc_pfa), np.std(auc_pfa)
```

```
('AUC', 0.7484107565837173, 0.027615946214022805)
```

Task 1.5

In this task, we ask you to visually report the RMSE and AUC scores achieved by the three student's models in the user-stratified 10-fold cross validation performed in Task 1.2, 1.3, and 1.4 respectively, in such a way that the models' performances can be easily and appropriately compared against each other.

```
### YOUR CODE HERE (please, feel free to add extra cells to solve this
task, after this first one) ###

m = {'AUC': {'BKT': auc_bkt, 'AFM': auc_afm, 'PFA': auc_pfa}, 'RMSE':
{'BKT': rmse_bkt, 'AFM': rmse_afm, 'PFA': rmse_pfa}}
limits = {'AUC': 1, 'RMSE': 0.5}

plt.figure(figsize=(15, 5))

for metric_idx, metric_key in enumerate(m.keys()):

    # Create the subplot for the current metric
    plt.subplot(1, len(m), metric_idx + 1)

    # Compute means, standard deviations, and labels
    means, errors, labels = [], [], []
    for model_key, model_scores in m[metric_key].items():
        means.append(np.mean(model_scores))
        errors.append(np.std(model_scores))
        labels.append(model_key)

    # Plot values
    x_pos = np.arange(len(labels))
    plt.bar(x_pos, means, yerr=errors, align='center', alpha=0.5,
ecolor='black', capsize=10)

    # Make decorations
    plt.grid(axis='y')
    plt.xticks(x_pos, labels)
    plt.ylabel(metric_key)
    plt.ylim(0, limits[metric_key])

# Show the plot
plt.tight_layout()
plt.show()
```



Task 1.6 Please compare and discuss the performance metric scores achieved by the student's models.

From the left plot, it can be observed that the BKT model exhibited a higher AUC score (over 0.87) and a lower standard deviation of AUC score across folds (0.01) with respect to AFM and PFA models, indicating that the predictive power of the BKT model is higher and more stable across folds than the one of the other two models, when AUC is considered. Specifically, the average BKT AUC is 0.12-point higher (0.87 vs 0.75) the average PFA AUC and 0.18-point higher (0.87 vs 0.69) than the average AFM AUC. Comparing the latter two models between each other, PFA outperforms AFM in terms of average AUC (0.75 vs 0.69, 0.06-point higher), though the difference in AUC between PFA and AFM is smaller than the difference of both with BKT. Furthermore, PFA exhibited a smaller standard deviation across folds (0.027) with respect to AFM (0.04). It can be hence observed that including information on both prior success and failure (PFA) allows the model to perform better and have a more stable performance across folds, with respect to the case where only general information on the opportunities is considered (AFM).

Similarly, the right plot highlights that the BKT model performs better, on average, with respect to AFM and PFA in terms of RMSE, i.e., the RMSE score is lower for the BKT model (0.36). AFM and PFA were able to achieve an average RMSE score of 0.42 (0.06-point higher than BKT) and 0.405 (0.045-point higher than BKT) respectively, showing again how using more fine-grained information on the failure and success of a student can lead to better predictions, but not as much as BKT. The standard deviation in performance across folds is quite small for all three models. Specifically, while being already the top performer in terms of average RMSE, the BKT model exhibited also the lowest standard deviation (0.009). AFM and PFA showed slightly higher standard deviations (0.018 and 0.12), showing that their performance were less stable across folds for the former. In general, the lower the RMSE is, the lower the standard deviation of the model performance is too. This strengthen the good performance shown by BKT, especially.

Based on our results, we can generally observe that the characteristics and properties of the BKT model allow us to make better estimations of student's skill mastery, compared to the two other considered models. One reason behind this finding might be that the skills at hand seem to meet well BKT assumptions, specifically that (i) knowledge can be divided into different skills, (ii) the definition of skills is accurate enough, (iii) each task corresponds to a single skill, and (iv) there is no connection between the skills (we should

mention that there is no "extensive" connection, though for certain skills some interrelationships exist). For AFM and PFA ones, leveraging prior failure and success opportunities (PFA) and fed them into the model leads to good performance with respect to AFM, but not as much we can with BKT. One reason behind the performance differences across models (and a step for further improvement of all of them) might lie also on the fact that we considered only a basic BKT (no forgetting or other advanced modelling) and we have not tuned certain aspects (e.g., bounds and regularizations in PFA and AFM).

2 Knowledge Tracing: Learning Curves Comparison

In this section, you should fit a Bayesian Knowledge Tracing (BKT) model on the three skills included in the `data` data set, and compute the corresponding predictions. Then, for each skill included in the data dataframe, you should visually report and discuss (i) the learning curve and (ii) the bar plot representing the number of students who reached a given number of opportunities for that skill, obtained with the BKT model fitted on the above-mentioned skills, in such a way that they can be easily and appropriately compared. No comparison with other baseline model is required.

For your convenience, you will be guided in completing this section through three main tasks:

- Task 2.1: BKT fit and prediction.
- Task 2.2: Learning curves and bar plots generation.
- Task 2.3: Learning curves and bar plots discussion.

Task 2.1

In this task, we ask you to fit a BKT model with all default parameters, i.e., `Model(seed=0)` in pyBKT, on the full `data` data set (no split into train and test set needed as we are not assessing predictive performance of the model here). Once you BKT model is fitted, we ask you to create a dataframe named `predictions` with four columns `user_id`, `skill_name`, `y_true`, `y_pred_bkt`. This dataframe should include one row per interaction in `data`, where user_id is the id of the student associated with that interaction, skill_name is the name of the skill involved in that interaction, y_true is the student's performance on that interaction (1 if correct at the first attempt, 0 otherwise), and y_pred_bkt is the prediction made by the pre-trained BKT model for that interaction.

Please, note that this task may require a long running time (e.g., about 10 to 20 minutes), depending on your implementation. Just as an indication, on a Dell XPS 13, the fit process lasts around 7 minutes.

```
### YOUR CODE HERE (please, feel free to add extra cells to solve this
task, after this first one) ###

# Initialize the model
```

```
model = Model(seed=0)

# Fit the model on the entire dataset
%time model.fit(data=data)
```

```
Wall time: 6min 40s
```

```
# Make predictions
predictions = model.predict(data=data)[['user_id', 'skill_name',
'correct', 'correct_predictions']]

# Rename the dataframe columns as per instructions
predictions.columns = ['user_id', 'skill_name', 'y_true',
'y_pred_bkt']
```

As a double check, we show the first ten rows of the predictions dataframe.

```
predictions.head()
```

	user_id	skill_name	y_true	y_pred_bkt
0	14	Circle Graph	0	0.46431
1	14	Circle Graph	1	0.33660
2	14	Circle Graph	0	0.56816
3	14	Circle Graph	0	0.44325
4	14	Circle Graph	0	0.31982

Task 2.2

In this task, for each skill, we ask you to visually report and discuss (i) the `learning curve` and (ii) the `bar plot` representing the number of students who reached a given number of opportunities (similar to the visualizations done in Tutorial 6), obtained by the BKT model fitted on that skill, in such a way that they can be easily and appropriately compared. To do so, we ask you to use the predictions you stored in the dataframe `predictions`.

No comparison with other baseline model is required.

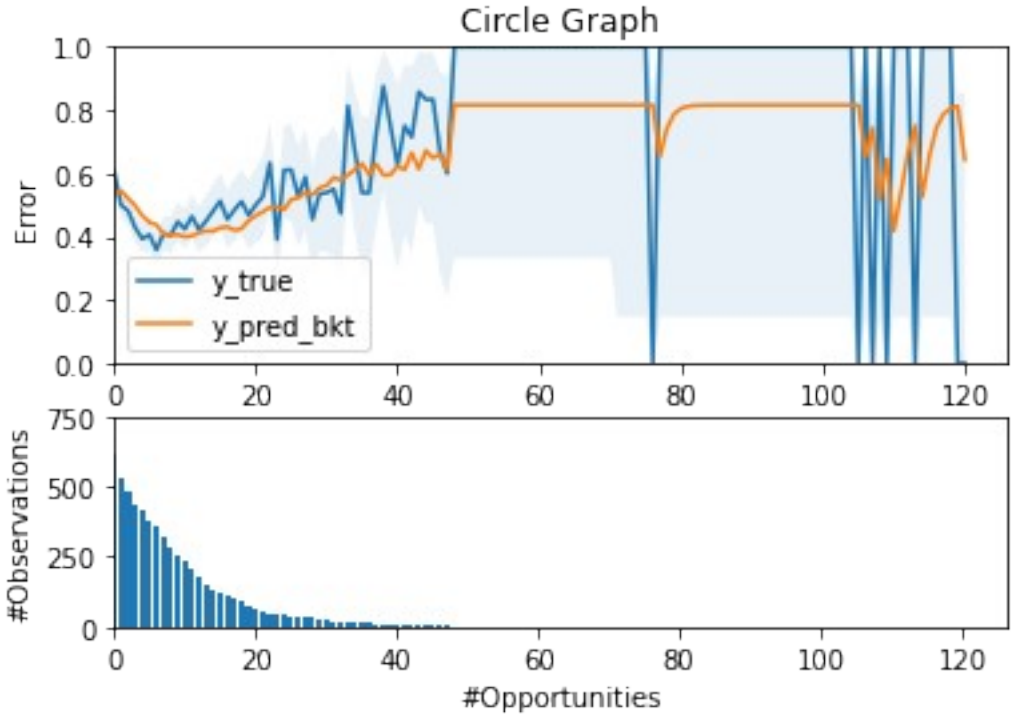
Please, refer to Tutorial 6 for further information on learning curve and bar plotting for student's knowledge tracing models.

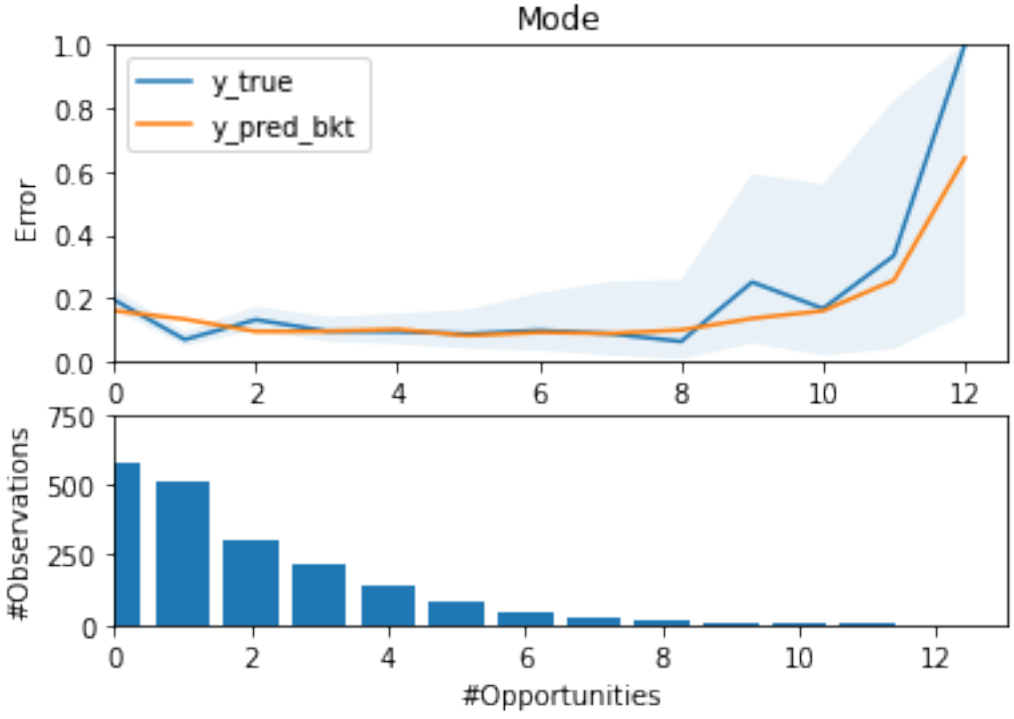
```
### YOUR CODE HERE (please, feel free to add extra cells to solve this
task, after this first one) ###
def avg_y_by_x(x, y):
    # Transform lists into arrays
    x = np.array(x)
    y = np.array(y)

    # Sort the integer id representing the number of opportunities in
increasing order
```

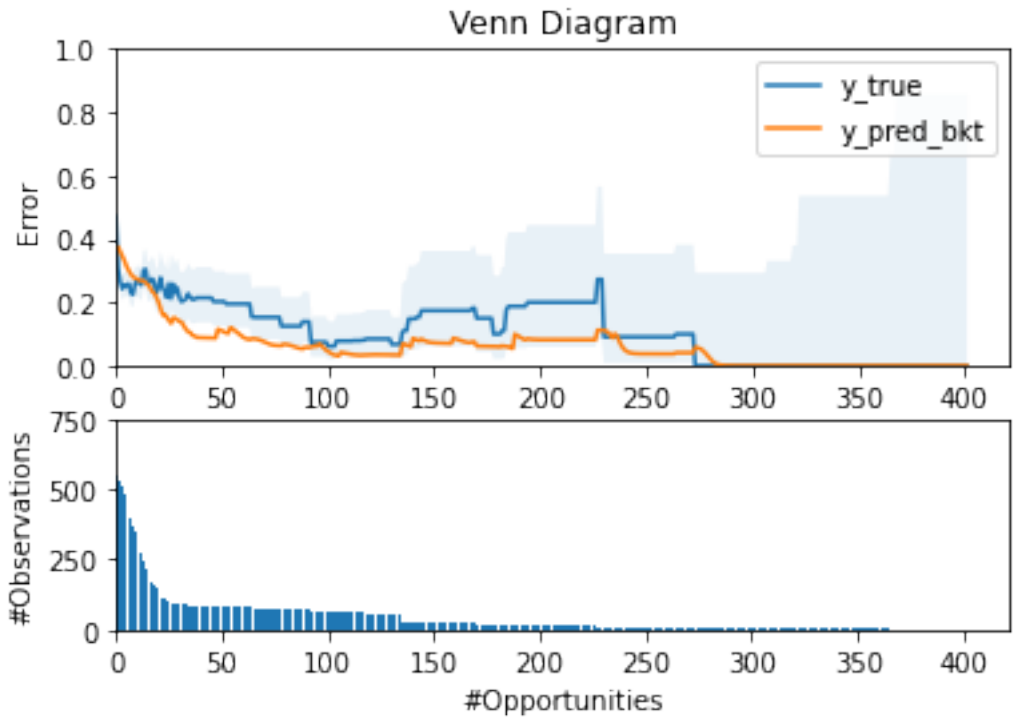
```
    xs = sorted(list(set(x)))

    # Supporting lists to store the:
    # - xv: integer identifier of the number of opportunities
    # - yv: average value across students at that number of
opportunities
    # - lcb and ucb: lower and upper confidence bound
    # - n_obs: number of observartions present at that number of
opportunities (on per-skill plots, it is the #students)
    xv, yv, lcb, ucb, n_obs = [], [], [], [], []

    # For each integer identifier of the number of opportunities
0, ...
    for v in xs:
        ys = [y[i] for i, e in enumerate(x) if e == v] # We retrieve
the values for that integer identifier
        if len(ys) > 0:
            xv.append(v) # Append the integer identifier of the number
of opportunities
            yv.append(sum(ys) / len(ys)) # Append the average value
across students at that number of opportunities
            n_obs.append(len(ys)) # Append the number of observartions
present at that number of opportunities


            # Prepare data for confidence interval computation
            unique, counts = np.unique(ys, return_counts=True)
            counts = dict(zip(unique, counts))

            if 0 not in counts:
                counts[0] = 0
            if 1 not in counts:
                counts[1] = 0

            # Calculate the 95% confidence intervals
            ci = sc.stats.beta.interval(0.95, 0.5 + counts[0], 0.5 +
counts[1])
            lcb.append(ci[0])
            ucb.append(ci[1])

    return xv, yv, lcb, ucb, n_obs

### YOUR CODE HERE (please, feel free to add extra cells to solve this
task, after this first one) ###

for plot_id, skill_name in enumerate(data['skill_name'].unique()): #
For each skill under consideration

    preds = predictions[predictions['skill_name'] == skill_name] #
Retrieve predictions for the current skill
```

```
    xp = []
    yp = {}
    for col in preds.columns: # For y_true and and y_pred_bkt columns,
initialize an empty list for curve values
        if 'y_' in col:
            yp[col] = []

    for user_id in preds['user_id'].unique(): # For each user
        user_preds = preds[preds['user_id'] == user_id] # Retrieve the
predictions on the current skill for this user
        xp += list(np.arange(len(user_preds))) # The x-axis values go
from 0 to |n_opportunities|-1
        for col in preds.columns:
            if 'y_' in col: # For y_true and and y_pred_bkt columns
                yp[col] += user_preds[col].tolist() # The y-axis value
is the success rate for this user at that opportunity

    fig, axs = plt.subplots(2, 1, gridspec_kw={'height_ratios': [3,
2]}) # Initialize the plotting figure

    lines = []
    for col in preds.columns:
        if 'y_' in col: # For y_true and and y_pred_bkt columns
            x, y, lcb, ucb, n_obs = avg_y_by_x(xp, yp[col]) #
Calculate mean and 95% confidence intervals for success rate
            y = [1-v for v in y] # Transform success rate in error
rate
            if col == 'y_true': # In case of ground-truth data, we
also show the confidence intervals
                axs[0].fill_between(x, lcb, ucb, alpha=.1)
            model_line, = axs[0].plot(x, y, label=col) # Plot the
curve
            lines.append(model_line) # Store the line to then set the
legend

    # Make decorations for the learning curve plot
    axs[0].set_title(skill_name)
    axs[0].legend(handles=lines)
    axs[0].set_ylabel('Error')
    axs[0].set_ylim(0, 1)
    axs[0].set_xlim(0, None)

    # Plot the number of observations per number of opportunities bars
and make decorations
    axs[1].set_xlabel('#Opportunities')
    axs[1].bar([i for i in range(len(n_obs))], n_obs)
    axs[1].set_ylabel('#Observations')
    axs[1].set_ylim(0, 750)
    axs[1].set_xlim(0, None)
```

```
# Plot the learning curve and the bar plot
plt.show()
```







Task 2.3 Please discuss all visualizations (learning curves and bar plots) obtained with the BKT model.

First, we discuss each skill separately. Then, given that we are showing the same aspects for the three skills, we provide few more observations aimed at comparing the findings we obtained across the skills under consideration.

- "**Circle Graph**". From the error rate in the ground-truth data (y_true, blue), it can be observed that this skill appears quite hard for students, with an initial error rate of around 0.60 in the first opportunity. The error rate goes slightly down, as we would usually expect, as much as the students play with the skill, reaching the lowest error rate of 0.38 at around 7 opportunities. However, starting from #opportunities around equal to 7, the error rate starts going up till 0.80 after 50 opportunities. It seems that a large part of the students managed to master a bit this skill in the first opportunities, but then they start facing difficulties in mastering it. This behavior might be due to the fact that the skill is ill-defined or that the problems become too difficult or are not aligned well with the skill. After 50 opportunities, the error rate starts jumping between 0 and a 1 just because there are only few students. In terms of confidence interval (blue area), it can be observed that the error rate is quite stable at the earlier stage. Starting from around 7 opportunities, the error rate is less stable probably due to the same reasons we provided above. The confidence interval finally becomes large when only few students keep playing with this skill for a higher number of opportunities. When it comes to consider the error rate obtained by the BKT model estimations (y_pred_bkt, orange), it can be observed that the model tends to overestimate a bit the error rate during the first opportunities (the orange line is above the blue one, the model tends to predict more errors than

expected), while the opposite pattern is observed for higher numbers of opportunities (the model tends to predict less errors than expected). However, in general, the model fits well the ground-truth data. Looking at the bar plot at the bottom, students appear reasonably engaged with this skill only for few opportunities (particularly, starting from 35 opportunities, the number of involved students is low, and the error rate starts jumping more).

- "**Mode**". Based on the patterns of the ground-truth data (y_true, blue), the error rate observed for this skill at the earlier stages is of around 0.20. The students thus are already good in problems involving this skill, somehow. While this might depend on the intrinsic knowledge/background of the students about this skill, this behavior might also appear due to the fact that the skill is easier and easier to master in general or that the problems presented to the students on that skill are too easy, for instance. While the error rate goes slightly down when the number of opportunities increases, the difference between the initial error rate and the error rate experienced (for instance) after 5 or 6 opportunities is relatively small (around 0.10 of error rate in the latter case). This observation might be justified by the fact that students played with the skill only for few opportunities (so there is no so much room for improving more) and by the already low initial error rate (students might use those opportunities just to refine their knowledge, with few errors still happening). The going-up behavior at the end appears mainly due to the very low number of students involved at that point. The blue area shows us that the confidence interval is very small for the first three or four opportunities (so behavior is more consistent among students), while it starts increasing later probably also due to the peculiar patterns behind students who interact more with this skill. The BKT model (y_pred_bkt, orange) is able to fit the ground-truth data very well across the number of opportunities, especially till around 8 opportunities. One important observation comes from the bar plot, which shows that the number of students involved in this skill drastically goes down especially after three/four opportunities.

- "**Venn Diagram**". This third skill appears like another somehow easy skill since the beginning (y_true, blue), but a bit more harder than "Mode". Except for a few fluctuations in the very first opportunities (where the curve slightly increases, maybe because they are just the first opportunities or due to the type/difficulty of problems proposed at that point), the error rate goes down till around 0.08, reached at around 100 opportunities. Probably, once only those students struggling more with this skill tend to stay for a higher number of opportunities, the error rate goes up again. The confidence interval for the ground-truth data (blue area) is relatively high (if compared with the one observed for the other two skills during the first opportunities). Naturally, it increases even more when the number of students reaching that number of opportunities is very low (#opportunities > 130). Comparing the ground-truth estimations with the BKT estimations (y_pred_bkt, orange), except for the first opportunities, it can be observed that BKT tends to consistently underestimate the error rate and the BKT error rate is often outside the confidence interval of actual data. The number of students per opportunity naturally

goes down as observed for the other skills, but interestingly it can be observed that a set of students seems to still keep playing with this skill from around 30 opportunites till 100 opportunities stably (the bar plot values appear somehow similar in that range).

Overall, comparing the considered three skills, "Circle Graph" seems to be really going up (e.g., ill-defined skill or due to the type/difficulty of problems), while, for "Mode" and "Venn Diagram", the going up later seems to come only from the very low number of students. Especially but not only for the "Venn Diagram" skill, the BKT generally tends to under-estimate the error rate. One reason behind this observation could be that the BKT model we adopted did not consider forgetting, for instance. In terms of number of opportunities, students experienced a lower maximum number of opportunities on the "Mode" skill, compared to the others. Compared to "Circle Graph", the skill "Venn Diagram" seems to involve lots of students even for higher number of opportunities (e.g., #opportunities > 30).