ARTEFACT RESEARCH CENTER

CentraleSupélec

INSEAD

London Business School

Choice-Learn

A deep learning lens on choice modeling

Vincent Auriau, Ali Aouad, Antoine Désir, Emmanuel Malherbe

Choice-Learn: choice modeling in Python

Choice

Motivations: why is it important?
>>> Faire lien avec business use case Artefact

>>> Missing tools because of scale: orders of magnitude

> slide summary: contribution agenda
> Existing work
> what's new ? - choice learn presentation

Agenda

Choice-Learn is a choice modeling Python library that specializes in handling large datasets and models

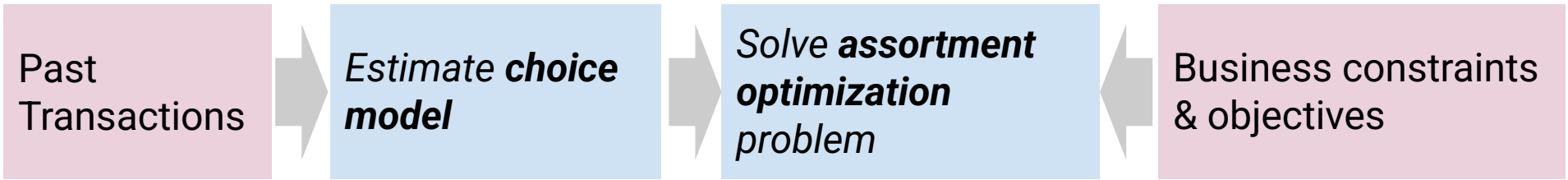
Agenda

Choice-Learn is a choice modeling Python library that specializes in handling large datasets and models

- What needs does our new library address?
- How does the library work?
- A few examples and computational experiments

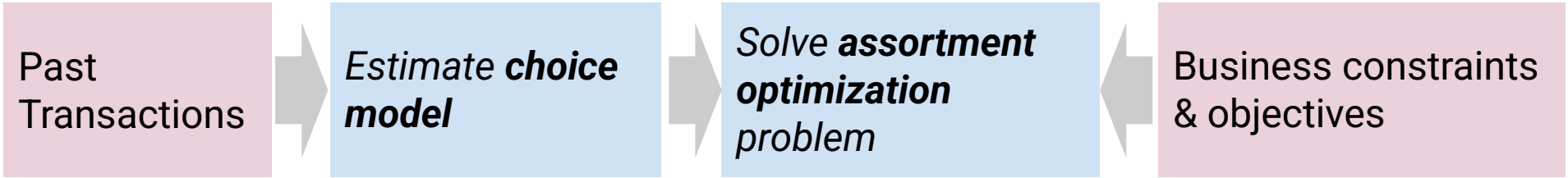
Some context: Optimizing a supermarket's assortment

A classical assortment optimization pipeline:



Some context: Optimizing a supermarket's assortment

A classical assortment optimization pipeline:



Retail constraints

- Mandatory products
- Fixed number of products by store
- Fixed number of products in a single stores cluster

Retail Objectives (depending on the category):

- Maximize margin
- Maximize number of sales

Some context: Optimizing a supermarket's assortment

A classical assortment optimization pipeline:

Past Transactions → *Estimate **choice model*** → *Solve **assortment optimization** problem* ← Business constraints & objectives

---

Retail Dataset (Fortenova)
- 589 supermarkets
- 129 categories
- 25.000 different items
- 5 billions transactions

→ ***Which choice model formulation should be used ?***
How to implement it ?

Main packages for choice modeling in Python

BioGeme

+ Free form of utility specification
+ Well established

- No “Deep Learning” models
- Data batching unavailable

PyLogit

+ Well established
+ Efficient estimation of defined models

- Limited list of models
- Data batching unavailable

Torch-Choice

+ Built for large datasets
+ Data batching available
+ Backed efficiently by PyTorch

- Limited list of models
- Difficult to customize



Python: the go-to programming language for companies : user-friendly & high capabilities



Disseminated open-sourced implementations of published models

Main packages for choice modeling in Python

BioGeme

+ Free form of utility specification
+ Well established

- No Machine Learning models
- Data batching unavailable

PyLogit

+ Well established
+ Efficient estimation of defined models

- Limited list of models
- Data batching unavailable

Torch-Choice

+ Built for large datasets
+ Data batching available
+ Backed efficiently by PyTorch

- Limited list of models
- Difficult to customize



Disseminated open-sourced implementations of published models

- *Disparate data format*
-

ARTEFACT

Choice-Learn ambitions

Scalable

- *Handles large datasets*
- *Handles large models*

Models Library

- *Literature models are implemented and “ready-to-use”*
- *Classical as well as non linear models integrated*

Flexible

- *Custom models can be defined to fit any use case*
- *Code organized in independent bricks*

Operations

- *Integrates operationalized pipelines with well known formulations of assortment optimization or pricing*

Choice-Learn development axes

Testing different models

Writing an out-of scope model

Adaptation to the disparate data format

Observing numerical differences

Using different code signatures & structures

Sometimes data pre-processing is enough

Or it is needed to write everything from scratch

Choice-Learn Ambitions:

Scalable	Model Library	Flexible	Operations
Handles large datasets and large models	*Handles different models, classical as well as non linear*	*Custom models can be defined to fit any use case*	*Integrates and is thought to be integrated in operationalized pipelines*

Choice-Learn development axes

Main encountered difficulties:

- Differences in data formats needed
- Numerical differences
- Optimization differences
- Model structure different than classical ML models

Examples:

- Long, wide, etc…
- log(P + \eps)
- different optimizers can lead to slightly different results)
- from scikit-learn, DL libraries

Scalable	Model Library	Flexible	Operations
Handles large datasets and large models	*Handles different models, classical as well as non linear*	*Custom models can be defined to fit any use case*	*Integrates and is thought to be integrated in operationalized pipelines*

Data Scalability

Avoid features repetition: “long format” repeats over context features for each alternative

Data Scalability

Avoid features repetition: “long format” repeats over context features for each alternative

	case	client_id	alt	purpose	income	price
0	0	0	train	1	2	112
1	0	0	car	1	2	117
2	0	0	SM	1	2	63
3	1	0	train	1	2	103
4	1	0	car	1	2	117
5	1	0	SM	1	2	60
6	2	0	train	1	2	130
7	2	0	car	1	2	117
8	2	0	SM	1	2	67
9	3	1	train	1	1	170
10	3	1	car	1	1	0
11	3	1	SM	1	1	70
12	4	1	train	1	1	213
13	4	1	car	1	1	0
14	4	1	SM	1	1	84

Extract of the SwissMetro dataset

Separation into two indexes: ‘shared’ and ‘items’ features

case	alt	price
0	train	112
0	car	117
0	SM	63
1	train	103
1	car	117
1	SM	60
2	train	130
2	car	117
2	SM	67
3	train	170
3	car	0
3	SM	70
4	train	213
4	car	0
4	SM	84

case	client_id	purpose	income
0	0	1	2
1	0	1	2
2	0	1	2
3	1	1	1
4	1	1	1

Data Scalability

Avoid features repetition: “long format” repeats over context features for each alternative

	case	client_id	alt	purpose	income	price
0	0	0	train	1	2	112
1	0	0	car	1	2	117
2	0	0	SM	1	2	63
3	1	0	train	1	2	103
4	1	0	car	1	2	117
5	1	0	SM	1	2	60
6	2	0	train	1	2	130
7	2	0	car	1	2	117
8	2	0	SM	1	2	67
9	3	1	train	1	1	170
10	3	1	car	1	1	0
11	3	1	SM	1	1	70
12	4	1	train	1	1	213
13	4	1	car	1	1	0
14	4	1	SM	1	1	84

Extract of the SwissMetro dataset

Separation into two indexes: ‘shared’ and ‘items’ features

case	alt	price
0	train	112
0	car	117
0	SM	63
1	train	103
1	car	117
1	SM	60
2	train	130
2	car	117
2	SM	67
3	train	170
3	car	0
3	SM	70
4	train	213
4	car	0
4	SM	84

case	client_id	purpose	income
0	0	1	2
1	0	1	2
2	0	1	2
3	1	1	1
4	1	1	1

Data Scalability

Avoid features repetition: “long format” repeats over context features for each alternative

Features Storage: repeated features (e.g. a one-hot) can be stored on the side and reference only by an ID

case	client_id	purpose	income
0	0	1	2
1	0	1	2
2	0	1	2
3	1	1	1
4	1	1	1



case	client_id
0	0
1	0
2	0
3	1
4	1

client_id	purpose	income
0	1	2
1	1	1

Data Scalability

Avoid features repetition: “long format” repeats over context features for each alternative

Features Storage: repeated features (e.g. a one-hot) can be stored on the side and reference only by an ID

case	client_id	purpose	income
0	0	1	2
1	0	1	2
2	0	1	2
3	1	1	1
4	1	1	1



case	client_id
0	0
1	0
2	0
3	1
4	1

client_id	purpose	income
0	1	2
1	1	1

Data Scalability

Avoid features repetition: “long format” repeats over context features for each alternative

Features Storage: repeated features (e.g. a one-hot) can be stored on the side and reference only by an ID

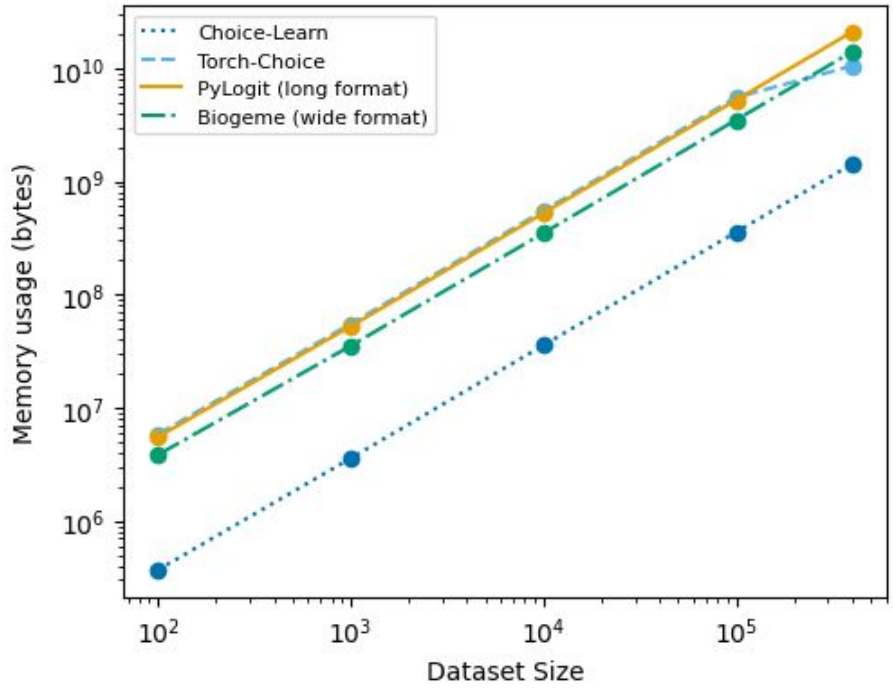
Data structure reconstructed when needed in **batches**: batch optimizers integrated

Data Scalability

Avoid features repetition: “long format” repeats over context features for each alternative

Features Storage: repeated features (e.g. a one-hot) can be stored on the side and reference only by an ID

Data structure reconstructed when needed in **batches**: batch optimizers integrated



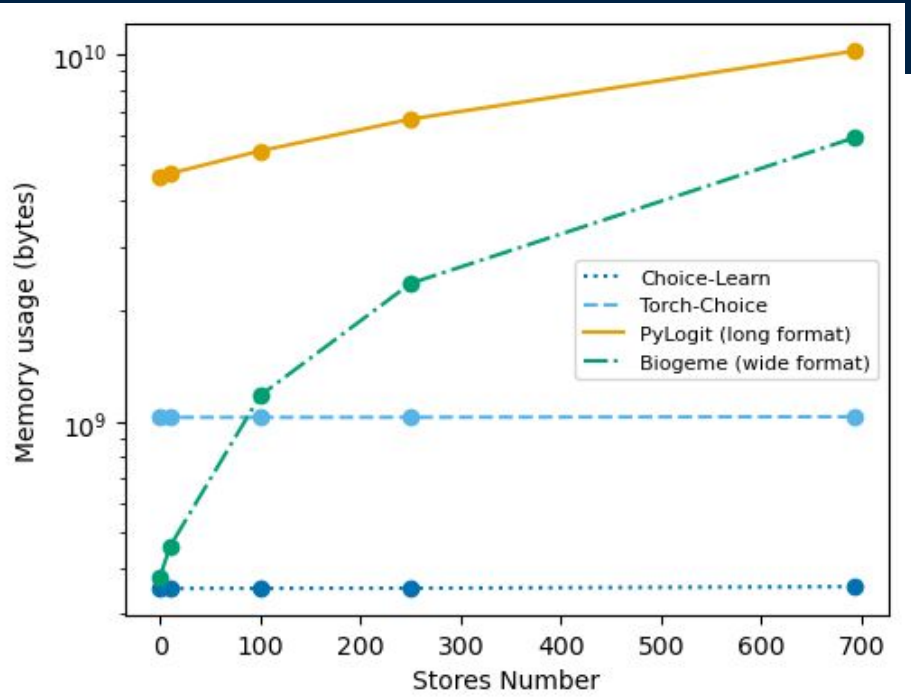
Experiment: Comparing RAM usage for different size of the Expedia dataset

Data Scalability

Avoid features repetition: “long format” repeats over context features for each alternative

Features Storage: repeated features (e.g. a one-hot) can be stored on the side and reference only by an ID

Data structure reconstructed when needed in **batches**: batch optimizers integrated



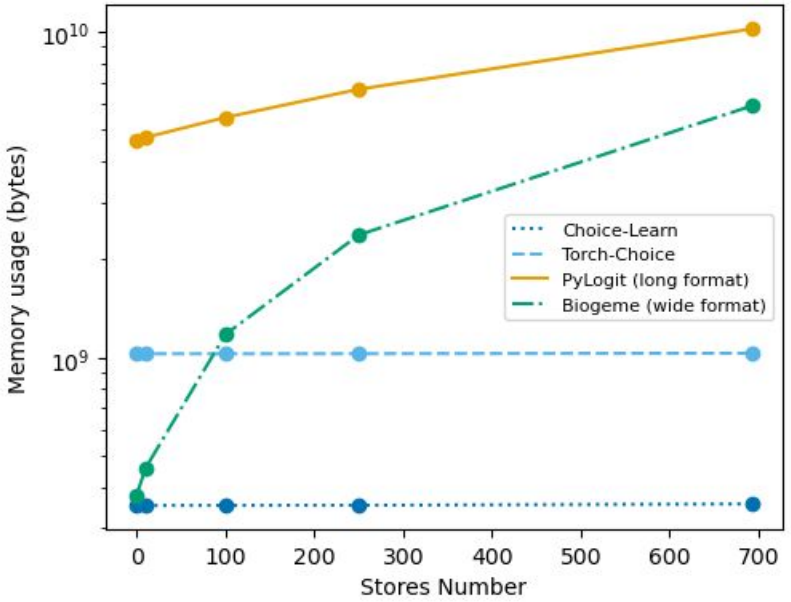
Experiment: Comparing RAM usage for different number of stores on our Retail dataset

Data scalability

Avoid features repetition: “long format” repeats over context features for each alternative

Features Storage: repeated features (e.g. a one-hot) can be stored on the side and reference only by an ID (e.g. the index of the 1)

Data structure reconstructed when needed in **batches**: batch optimizers integrated



(d) Memory usage of our retail dataset for different number of stores

A collection of ready-to-use models

Currently available models:

Logit based:
- MNL
- Conditional Logit
- Nested Logit
- etc…

Neural-Network based:
- ResLogit - 2019 [1]
- TasteNet - 2020 [2]
- RUMnet - 2022 [3]
- etc…

& more to come…

> Reaching equivalent results as other packages / official implementations

[1] Wong M, Farooq B. ResLogit: A residual neural network logit model for data-driven choice modelling.
[2] Han Y, Pereira FC, Ben-Akiva M, Zegras C. A neural-embedded choice model: Tastenet-mnl modeling taste heterogeneity with flexibility and interpretability
[3] Aouad A, Désir A. Representing random utility choice models with neural networks.

Models estimation method



- Efficient implementation of **optimizers** (L-BFGS, SGD, Adam, etc…)
- Efficient implementation of mathematical **operations** - Graph Mode
- Optimization **tools** (regularization, LR scheduling, etc…) available
- **Production** ready

be LaTeXified

$$U(0) = beta\ .\ price(0) + gamma_0\ .\ time(0)$$
$$U(1) = ASC_1 + beta\ .\ price(1) + gamma_1\ .\ time(1)$$
$$U(2) = ASC_2 + beta + price(2) + gamma_2\ .\ time(2)$$

To be code re

```
from choice_learn.models import ConditionalLogit

model = ConditionalLogit()
model.add_coefficient(feature_name="intercept", indexes = [1, 2])
model.add_coefficient(feature_name="time",
coefficient_name="gamma",  indexes = [0, 1, 2])
model.add_shared_coefficient(feature_name="price",
coefficient_name="beta",  indexes = [0, 1, 2])

model.fit(dataset)
```

Models estimation



- Efficient implementation of **optimizers** (L-BFGS, SGD, Adam, etc…)
- Efficient implementation of mathematical **operations** - Graph Mode
- Optimization **tools** (regularization, LR scheduling, etc…) available
- **Production** ready

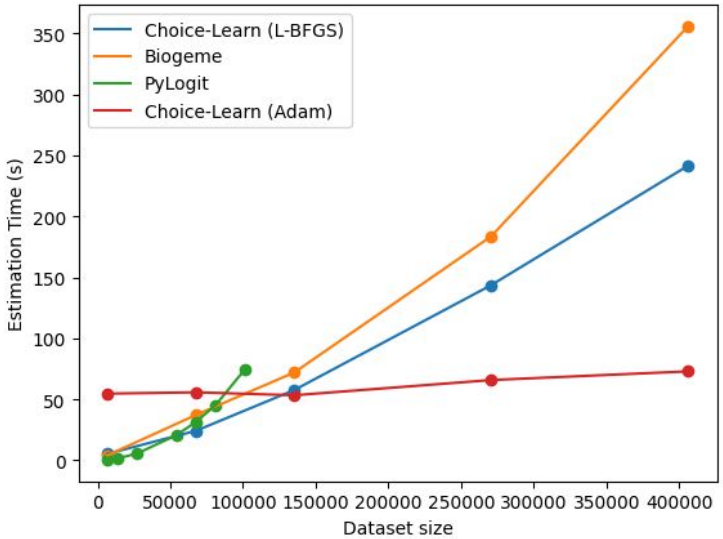
Models estimation

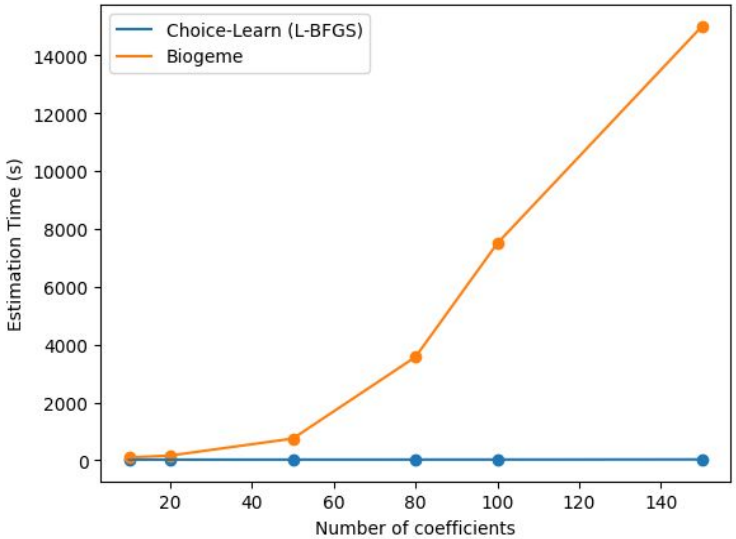
TensorFlow

- Efficient implementation of **optimizers** (L-BFGS, SGD, Adam, etc…)
- Efficient implementation of mathematical **operations** - Graph Mode
- Optimization **tools** (regularization, LR scheduling, etc…) available
- **Production** ready

Estimation time for the **Conditional Logit**:

- w.r.t **dataset size** (4 alternatives - 13 coefficients)
- w.r.t. **number of coefficients to estimate** (10 alternatives - 10.000 samples)





Model example: the Conditional Logit

Utility formulation

$$U(0) = \beta \cdot price(0) + \gamma_0 \cdot time(0)$$
$$U(1) = ASC_1 + \beta \cdot price(1) + \gamma_1 \cdot time(1)$$
$$U(2) = ASC_2 + \beta \cdot price(2) + \gamma_2 \cdot time(2)$$

Report example

	Coefficient Name	Coefficient Estimation	Std. Err	z_value	P(.>z)
0	beta_cost	-0.033339	0.007095	-4.699046	2.622604e-06
1	beta_frequence	0.092529	0.005097	18.151926	0.000000e+00
2	beta_ovt	-0.043004	0.003225	-13.335685	0.000000e+00
3	beta_ivt_0	0.059509	0.010073	5.908032	0.000000e+00
4	beta_ivt_1	-0.006784	0.004433	-1.530138	1.259825e-01
5	beta_ivt_2	-0.006460	0.001898	-3.403067	6.663799e-04
6	beta_ivt_3	-0.001450	0.001187	-1.221395	2.219366e-01

```python
from choice_learn.models import ConditionalLogit

model = ConditionalLogit()
model.add_coefficient(feature_name="intercept",
                      indexes = [1, 2])

model.add_coefficient(feature_name="time",
                      coefficient_name="gamma",
                      indexes = [0, 1, 2])

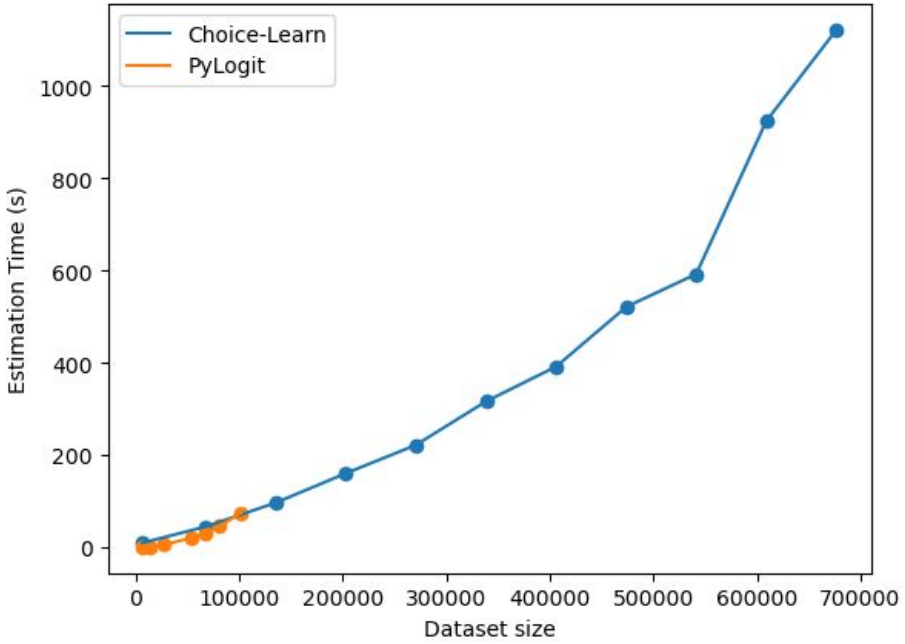
model.add_shared_coefficient(feature_name="price",
                             coefficient_name="beta",
                             indexes = [0, 1, 2])
model.fit(dataset)

```

Code example

ARTEFACT

Estimation Performances



ient Name	Coefficient Estimation	Std. Err	z_value	P(.>z)
beta_cost	-0.033339	0.007095	-4.699046	2.622604e-06
_frequence	0.092529	0.005097	18.151926	0.000000e+00
beta_ovt	-0.043004	0.003225	-13.335685	0.000000e+00
beta_ivt_0	0.059509	0.010073	5.908032	0.000000e+00
beta_ivt_1	-0.006784	0.004433	-1.530138	1.259825e-01
beta_ivt_2	-0.006460	0.001898	-3.403067	6.663799e-04
beta_ivt_3	-0.001450	0.001187	-1.221395	2.219366e-01
ntercept_0	0.698366	1.280191	0.545517	5.853977e-01
intercept_1	1.844099	0.708427	2.603088	9.238839e-03
ntercept_2	3.274184	0.624342	5.244213	1.192093e-07
_income_0	-0.089087	0.018347	-4.855648	1.192093e-06
a_income_1	-0.027993	0.003872	-7.228693	0.000000e+00
_income_2	-0.038147	0.004083	-9.342735	0.000000e+00

Conditional Logit formulation and its corresponding choice-learn code

ARTEFACT

Flexible Code: allow for efficient customization

Why allow for model customization ?

- Models wrapped in a user interface can have some limitations
- Manually coded models can be more efficient
- Being able to quickly design new models with minimal code writing

“Idea”

- No need to rewrite common structure of choice models
- Inheritance scheme useful to share code structure and methods: *softmax, NLL, etc…*
- A model is usually defined by its utility function - and so should be its code

Example of a customized model

Rewriting the Conditional Logit model as a custom model: $U(x_i, z) = \beta^1 \cdot x_i + \beta_i^2 \cdot z + ASC_i$

```python
from choice_learn.models import ChoiceModel

Class ExampleModel(ChoiceModel):

    def __init__(self, n_item_features, n_customer_features, n_items):

        self.beta_1 = tf.Variable(shape=(1, n_items_features))
        self.beta_2 = tf.Variable(shape=(n_items, n_customer_features))

        self.asc = tf.Variable(shape=(1, n_items))

    def compute_utility(self, X, Z):
        utilities = [ ]
        for item in range(self.n_items):
            utility_z = tf.dot(self.beta_2[ i ] , Z)
            utility_i = tf.dot( self.beta_1 , X[ i ] )
            utilities.append(utility_z + utility_i + self.asc[ i ] )
        return utilities

```

Example of a customized model

Rewriting the Conditional Logit model as a custom model: $U(x_i, z) = \beta^1 \cdot x_i + \beta_i^2 \cdot z + ASC_i$

```python
from choice_learn.models import ChoiceModel

Class ExampleModel(ChoiceModel):

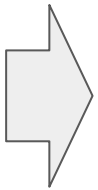
    def __init__(self, n_item_features, n_customer_features, n_items):

        self.beta_1 = tf.Variable(shape=(1, n_items_features))
        self.beta_2 = tf.Variable(shape=(n_items, n_customer_features))

        self.asc = tf.Variable(shape=(1, n_items))

    def compute_utility(self, X, Z):
        utilities = [ ]
        for item in range(self.n_items):
            utility_z = tf.dot(self.beta_2[ i ] , Z)
            utility_i = tf.dot( self.beta_1 , X[ i ] )
            utilities.append(utility_z + utility_i + self.asc[ i ] )
        return utilities

```



```python
model = ExampleModel(
  n_items=4,
  n_items_features=2,
 n_customer_features=5,
)

model.fit(dataset)
model.predict(dataset)

```

Example of a customized model

```python
from choice_learn.models import ChoiceModel

class ExampleModel(ChoiceModel):

    def __init__(self, n_item_features, n_customer_features, n_items):

        self.beta_1 = tf.Variable(shape=(1, n_items_features))
        self.beta_2 = tf.Variable(shape=(n_items_features, n_customer_features))
        self.asc = tf.Variable(shape=(1, n_items))

    def compute_utility(self, X, Z):
        utilities = [ ]
        for item in range(self.n_items):
            utility_z = tf.dot(self.beta_2[ i ] , Z)
            utility_i = tf.dot( self.beta_1 , X[ i ] )
            utilities.append(utility_z + utility_i + self.asc[ i ] )
        return utilities

```

Rewriting the MultiNomialLogi
as a custom model

$$U(x_i, z) = \beta^1 \cdot x_i + \beta^2_i \cdot z + asc_i$$

LaTeXified

Title

```python
from choice_learn.models import ChoiceModel

Class ExampleModel(ChoiceModel):

    def __init__(self, n_item_features, n_customer_features, n_items):

        self.beta_1 = tf.Variable(shape=(1, n_items_features))
        self.beta_2 = tf.Variable(shape=(n_items,
n_customer_features))
        self.asc = tf.Variable(shape=(1, n_items))

    def compute_utility(self, X, Z):
        utilities = [ ]
        for item in range(self.n_items):
            utility_z = tf.dot(self.beta_2[ i ] , Z)
            utility_i = tf.dot( self.beta_1 , X[ i ] )
            utilities.append(utility_z + utility_i + self.asc[ i ] )
        return utilities

```

oiceModel

):

tures, n_customer_features, n_items):

self.beta_1 = tf.Variable(shape=(1, n_items_features))
self.beta_2 = tf.Variable(shape=(n_items_features, n_customer_features))
self.asc = tf.Variable(shape=(1, n_items))

def compute_utility(self, X, Z):
utilities = [ ]
for item in range(self.n_items):
utility_z = tf.dot(self.beta_2[i] , Z)
utility_i = tf.dot(self.beta_1 , X[i])
utilities.append(utility_z + utility_i + self.asc[i])
return utilities

To be code re

ARTEFACT

Operations

Integration of Mixed Integer / Linear Programming for :

- **Assortment Optimization (vanilla MIP formulation)**
- **Pricing (discrete price ladder)**
- & more to come…





How to get started ?



Github:
- Tutorials
- Documentation
- “Community hub”

Discussions, raise issue, suggest improvement, new models & more !



Ready to use and open-Sourced on PyPI:

“ pip install choice-learn ”



Paper published in the Journal of Open-Source Software

Explore our GitHub and Star us if you like the package !

github.com/artefactory/choice-learn



Thank you for your listening !

A few examples

Application to our dataset

> lister les difficulties to overcome
>

> example d'un report

> slide future work ? (benchmarking, project, etc..)