Lenskit for Python 0.8.4

Modello dei dati

lkpy utilizza per rappresentare **items** e **users** soltanto i loro ID, questo non si evince direttamente dalle dichiarazione dei metodi, perché in python le variabili non sono tipizzate e non sono stati usati dei type hint, ma in alcuni unit test vengono usati date frame simili a quello in figura

```
simple_ratings = pd.DataFrame.from_records([
    (1, 6, 4.0),
    (2, 6, 2.0),
    (1, 7, 3.0),
    (2, 7, 2.0),
    (3, 7, 5.0),
    (4, 7, 2.0),
    (1, 8, 3.0),
    (2, 8, 4.0),
    (3, 8, 3.0),
    (4, 8, 2.0),
    (5, 8, 3.0),
    (6, 8, 2.0),
    (1, 9, 3.0),
    (3, 9, 4.0)
], columns=['user', 'item', 'rating'])
```

Le preferenze degli users sono memorizzate in oggetti ratings, strutturati in **data frame** della libreria **pandas**, quindi per ogni user e item viene espresso, se presente, un rating numerico.

I rating impliciti sono trattati da appositi algoritmi basati su matrix factorization implementati nella libreria *Implicit*, cioè *Alternating Least Squares* (ALS) e *Bayesian Personalized Ranking* (BPR); in questo caso qualsiasi valore di rating non nullo è interpretato come una preferenza implicita.

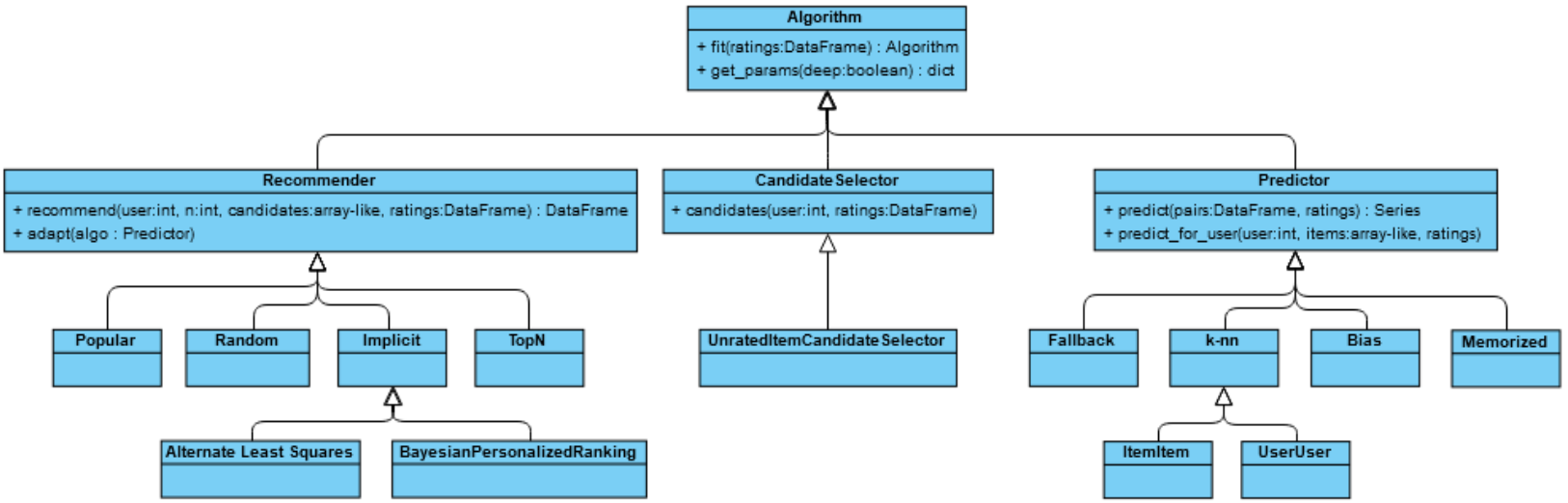
Sorgenti dati

I dati su cui il recommender potrà basare le proprie predizioni si ottengono attraverso il metodo *read_df_detect* del modulo *data.py* (data utilities); questo metodo rileva in automatico il formato del file (uno tra .csv e .parquet) e lo converte in un data frame pandas.

Il file può essere fornito tramite path del filesystem o un file URL (http o ftp), in generale va bene un qualunque **file-like object**, cioè un oggetto dotato di un metodo read() che copre il ruolo di StringIO

Architettura

Nel diagramma in figura sono descritte le principali interfacce e classi del framework, l'architettura risiede fondamentalmente nelle interfacce Algorithm, Recommender, CandidateSelector e Predictor; i livelli inferiori della gerarchia sono implementazioni, o categorie di implementazioni, che dipendono dagli algoritmi che si intende realizzare perciò non sono caratterizzanti dell'architettura del framework.



I blocchi in questo diagramma non corrispondono strettamente a classi nel codice sorgente, per esempio non esiste una classe k-nn, tuttavia è stata introdotta nel diagramma per esplicitare che ItemItem e UserUser sono entrambi appartenenti a questa categoria.

L'architettura dell'intero framework è **definita esclusivamente da ciò che concerne gli algoritmi** dato che non sono presenti né classi dedicate alla rappresentazione di users, items e ratings; né classi dedicate alla valutazione dato che per quest'ultima fase sono utilizzate semplici funzioni.

Le quattro interfacce fondamentali sono dichiarate nel modulo ***__init__.py*** e realizzate attraverso le ABC (abstract base class) python, segue descrizione:

- **Algorithm** – *lenskit.algorithms.Algorithm*
  segue il paradigma fit-predict utilizzato nell'implementazione dei classificatori supervisionati in SciKit, quindi sono dichiarati i metodi:
  - fit: esegue il training del model utilizzando i rating noti

    ```
    abstract fit(ratings, **kwargs)
    ```

  - get_params: restituisce tutti i parametri, ossia tutti gli attributi avvalorati nel costruttore, dell'istanza su cui viene chiamato

    ```
    get_params(deep=True)
    ```

- **Recommender** – *lenskit.algorithms.Recommender*
  estensione dell'interfaccia Algorithm, aggiunge il metodo *recommend*

  ```
  abstract recommend(user, n=None, candidates=None, ratings=None)
  ```

  le cui implementazioni hanno l'obiettivo di invocare i metodi di predizione degli score e fornire un data frame di suggerimenti, alcuni algoritmi restituiscono un data frame senza score.
  Questo metodo non è dichiarato in Algorithm perchè è ammessa l'esistenza di algoritmi per cui non è implementata la modalità con cui ottenere i suggerimenti, ma solo la funzione di predizione, su questo tipo di algoritmi può essere invocato il metodo adapt dell'interfaccia recommend

  ```
  classmethod adapt(algo)
  ```

  Quest'ultimo metodo adatta tale particolare tipologia di algoritmi assegnando come implementazione del metodo recommend la tecnica default, ossia TopN; adapt pone come **restrizione** che l'algoritmo su cui è chiamato implementi i metodi dichiarati in Predictor

- **CandidateSelector** – *lenskit.algorithms.CandidateSelector*
  Estensione dell'interfaccia Algorithm, aggiunge il metodo *candidates*

  ```
  abstract candidates(user, ratings=None)
  ```

  Lo scopo di questo metodo è identificare gli item candidati al suggerimento per un dato user
- **Predictor** – *lenskit.algorithms.Predictor*
  Estensione dell'interfaccia Algorithm, aggiunge i metodi:
  - predict: calcola lo score predetto per coppie user – item

    ```
    predict(pairs, ratings=None)
    ```

  - predict_for_user: calcola lo score predetto per un user su diversi items

    ```
    abstract predict_for_user(user, items, ratings=None)
    ```

Algoritmi implementati

Per alcuni importanti algoritmi, suddivisi nelle tipologie basic, k-nn e matrix factorization, sono fornite le implementazioni nei file del package *algorithms.* Si ha una classe, che implementa una delle interfacce precedentemente descritte, per ogni algoritmo.

Gli algoritmi k-nn memorizzano il modello appreso, dato in input il data frame dei ratings, tramite matrici ottimizzate dalla libreria Math Kernel Library (MKL).

Per item_knn e user_knn, i parametri configurabili sono:

- **`nnbrs`**`(int)`: massimo numero di neighbor
- **`min_nbrs`**`(int)`: massimo numero di neighbor
- **`min_sim`**`(double)`: soglia minima di similarità
- **`center`**`(bool)`: true indica che i rating sono normalizzati con il mean-centering
- **`aggregate`**: tipo di aggregazione, *'sum'* o *'weighted average'*

La funzione di similarità è hardcoded nell'algoritmo

Per la tipologia basic sono forniti, oltre al recommender default TopN, altri utili algoritmi:

- **`MostPopular`** e **`Random`** utili per la fase iniziale di utilizzo del recommender in cui la sparsità dei dati è elevata
- **`UnratedItemCandidateSelector`**: seleziona come candidati al suggerimento gli item non ancora votati
- **`FallbackPredictor`**: richiede in input uno o più algoritmi, esegue la predizione con il primo di essi e tenta di riparare le mancanze con i successivi

Valutazione

Crossfold

Le tecniche di crossfold sono messe a disposizione come **funzioni** del modulo crossfold.py; ognuna di esse restituisce un generator (costrutto python simile ad una collezione iterabile) di coppie train – test; gli oggetti train - test della coppia sono anch'essi data frame pandas.

Esempio: `def` **`partition_rows`**`(data, partitions, *, rng_spec=None)`

Calcolo metriche

Le implementazioni del calcolo delle metriche sulle predizioni sono fornite come **funzioni** dei moduli *predict.py* e topn.py (metriche che considerano anche il ranking) nel package *metrics.* Tali funzioni seguono una struttura comune, richiedono in input due pandas ***series***: predictions and truths, rispettivamente i rating predetti e i rating noti dal test set.

Esempio: `def` **`rmse`**`(predictions, truth, missing='error')`

Esempio di utilizzo

Esempio di loading di un dataset, dichiarazione di un recommender, esecuzione delle predizioni, crossfold e calcolo di una metrica.

```
import pandas as pd
from lenskit import batch, topn, datasets, util
from lenskit import crossfold as xf
from lenskit.algorithms import knn, Recommender

ml = datasets.ML100K('ml-100k')
ratings = ml.ratings
algo = knn.ItemItem(30)

def eval(train, test):
        # ensure the recommender is a top-N recommender
        rec = Recommender.adapt(algo)
        # cloning the model ensures forward compatibility
        # with things like parallelism
        rec = util.clone(rec)
        rec.fit(train)
        users = test.user.unique()
        recs = batch.recommend(algo, model, users, 100)
        return recs

# compute evaluation
splits = list(xf.partition_users(ratings, 5, xf.SampleFrac(0.2))
recs = pd.concat((eval(p.train, p.test) for p in splits), ignore_index=True)

# integrate test data; since splits are disjoint users for a single data set, this is sufficient
all_test = pd.concat(p.test for p in splits)

# compute metric results
rla = topn.RecListAnalysis()
rla.add_metric(topn.ndcg)
metrics = rla.compute(recs, all_test)
```