myFM

Release 0.2.1

Tomoki Ohtsuki

Apr 19, 2023

BASIC USAGE

myFM is an unofficial implementation of Bayesian Factorization Machines in Python/C++. Notable features include:

- Implementation of all corresponding functionalities in libFM MCMC engine (including grouping & relation block)
- A simpler and faster implementation using Pybind11 and Eigen
- Gibbs sampling for **ordinal regression** with probit link function. See *the tutorial* for its usage.
- Support variational inference, which converges faster and requires lower memory (but usually less accurate than the Gibbs sampling).

In most cases, you can install the library from PyPI:

```
pip install myfm
```

It has an interface similar to sklearn, and you can use them for wide variety of prediction tasks. For example,

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn import metrics

from myfm import MyFMClassifier

dataset = load_breast_cancer()
X = StandardScaler().fit_transform(dataset['data'])
y = dataset['target']

X_train, X_test, y_train, y_test = train_test_split(
   X, y, random_state=42
)
fm = MyFMClassifier(rank=2).fit(X_train, y_train)

print(metrics.roc_auc_score(y_test, fm.predict_proba(X_test)))
# 0.9954
```

Try out the following *examples* to see how Bayesian approaches to explicit collaborative filtering are still very competitive (almost unbeaten)!

CHAPTER

ONE

QUICK START

1.1 Installation

On MacOS/Linux First try:

```
pip install myfm
```

If it works, you can now try the examples.

If there is something nasty, then read the detailed installation guide and figure out what went wrong. Of course, feel free to create an issue on GitHub!

1.2 A toy example

Let us first look at how *myfm.MyFMClassifier* works for a toy example provided in pyFM.

```
import myfm
from sklearn.feature_extraction import DictVectorizer
import numpy as np
train = [
    {"user": "1", "item": "5", "age": 19},
    {"user": "2", "item": "43", "age": 33},
    {"user": "3", "item": "20", "age": 55},
    {"user": "4", "item": "10", "age": 20},
]
v = DictVectorizer()

X = v.fit_transform(train)

# Note that X is a sparse matrix
print(X.toarray())

# The target variable to be classified.
y = np.asarray([0, 1, 1, 0])
fm = myfm.MyFMClassifier(rank=4)
fm.fit(X,y)

# It also supports prediction for new unseen items.
fm.predict_proba(v.transform([{"user": "1", "item": "10", "age": 24}]))
```

As the example suggests, *myfm.MyFMClassifier* takes sparse matrices of scipy.sparse as its input. In the above example, sklearn's DictVectorizer transforms the categorical variables (user id and movie id) into a one-hot encoded vectors.

As you can see, :py:class:MyFMClassifier: can make predictions against new (unseen) items despite the fact that it is an MCMC solver. This is possible because it simply retains all the intermediate (noisy) samples.

For more practical example with larger data, move on to *Movielens examples* .

CHAPTER

TWO

A BASIC TUTORIAL WITH MOVIELENS 100K

FMs perform remarkably well on datasets with huge and sparse feature matrices, and the most common examples are (explicit) collaborative filtering tasks.

Let us examine the power of the Bayesian Factorization Machines by testing a series of APIs in myFM using the well-known Movielens 100k dataset.

2.1 Pure Matrix Factorization

First let us consider the probabilistic Matrix Factorization. That is, we model the user u's rating response to movie i, which we write r_{ui}, as

$$r_{ui} \sim w_0 + b_u + d_i + \vec{u}_u \cdot \vec{v}_j$$

This formulation is equivalent to Factorization Machines with

1. User IDs treated as a categorical feature with one-hot encoding
2. Movie IDs treated as a categorical feature with one-hot encoding

So you can efficiently use encoder like sklearn's OneHotEncoder to prepare the input matrix.

```
import numpy as np
from sklearn.preprocessing import MultiLabelBinarizer, OneHotEncoder
from sklearn import metrics

import myfm
from myfm.utils.benchmark_data import MovieLens100kDataManager

FM_RANK = 10

data_manager = MovieLens100kDataManager()
df_train, df_test = data_manager.load_rating_predefined_split(fold=3)

FEATURE_COLUMNS = ['user_id', 'movie_id']
ohe = OneHotEncoder(handle_unknown='ignore')

X_train = ohe.fit_transform(df_train[FEATURE_COLUMNS])
X_test = ohe.transform(df_test[FEATURE_COLUMNS])
y_train = df_train.rating.values
y_test = df_test.rating.values
```

(continues on next page)

(continued from previous page)

```
fm = myfm.MyFMRegressor(rank=FM_RANK, random_seed=42)
fm.fit(X_train, y_train, n_iter=200, n_kept_samples=200)

prediction = fm.predict(X_test)
rmse = ((y_test - prediction) ** 2).mean() ** .5
mae = np.abs(y_test - prediction).mean()
print(f'rmse={rmse}, mae={mae}')
```

The above script should give you RMSE=0.8944, MAE=0.7031 which is already impressive compared with other recent methods.

2.2 Assuming Separate Variance for movie & user

In Probabilistic Matrix Factorization, we usually assume user vectors and item vectors are drawn from separate normal priors:

$$u_i \sim \mathcal{N}(\mu_U, \Sigma_U)$$
$$v_i \sim \mathcal{N}(\mu_I, \Sigma_I)$$

However, we haven't provided any information about which columns are users' and items'.

You can tell *myfm.MyFMRegressor* these information (i.e., which parameters share a common mean and variance) by `group_shapes` option:

```
fm_grouped = myfm.MyFMRegressor(
    rank=FM_RANK, random_seed=42,
)
fm_grouped.fit(
    X_train, y_train, n_iter=200, n_kept_samples=200,
    group_shapes=[len(group) for group in ohe.categories_]
)

prediction_grouped = fm_grouped.predict(X_test)
rmse = ((y_test - prediction_grouped) ** 2).mean() ** .5
mae = np.abs(y_test - prediction_grouped).mean()
print(f'rmse={rmse}, mae={mae}')
```

This will slightly improve the performance to RMSE=0.8925, MAE=0.7001.

2.3 Adding Side information

It is straightforward to include user/item side information.

First we retrieve the side information from `Movielens100kDataManager`:

```
user_info = data_manager.load_user_info().set_index('user_id')
user_info["age"] = user_info.age // 5 * 5
user_info["zipcode"] = user_info.zipcode.str[0]
user_info_ohe = OneHotEncoder(handle_unknown='ignore').fit(user_info)
```

(continues on next page)

(continued from previous page)

```
movie_info = data_manager.load_movie_info().set_index('movie_id')
movie_info['release_year'] = [
    str(x) for x in movie_info['release_date'].dt.year.fillna('NaN')
]
movie_info = movie_info[['release_year', 'genres']]
movie_info_ohe = OneHotEncoder(handle_unknown='ignore').fit(movie_info[['release_year']])
movie_genre_mle = MultiLabelBinarizer(sparse_output=True).fit(
    movie_info.genres.apply(lambda x: x.split('|'))
)
```

Note that the way movie genre information is represented in `movie_info` DataFrame is a bit tricky (it is already binary encoded).

We can then augment `X_train` / `X_test` with auxiliary information. The hstack function of `scipy.sparse` is very convenient for this purpose:

```
import scipy.sparse as sps
X_train_extended = sps.hstack([
    X_train,
    user_info_ohe.transform(
        user_info.reindex(df_train.user_id)
    ),
    movie_info_ohe.transform(
        movie_info.reindex(df_train.movie_id).drop(columns=['genres'])
    ),
    movie_genre_mle.transform(
        movie_info.genres.reindex(df_train.movie_id).apply(lambda x: x.split('|'))
    )
])

X_test_extended = sps.hstack([
    X_test,
    user_info_ohe.transform(
        user_info.reindex(df_test.user_id)
    ),
    movie_info_ohe.transform(
        movie_info.reindex(df_test.movie_id).drop(columns=['genres'])
    ),
    movie_genre_mle.transform(
        movie_info.genres.reindex(df_test.movie_id).apply(lambda x: x.split('|'))
    )
])
```

Then we can regress `X_train_extended` against `y_train`

```
group_shapes_extended = (
    [len(group) for group in ohe.categories_] +
    [len(group) for group in user_info_ohe.categories_] +
    [len(group) for group in movie_info_ohe.categories_] +
    [ len(movie_genre_mle.classes_)]
)

fm_side_info = myfm.MyFMRegressor(
```

(continues on next page)

(continued from previous page)

```
    rank=FM_RANK, random_seed=42,
)
fm_side_info.fit(
    X_train_extended, y_train, n_iter=200, n_kept_samples=200,
    group_shapes=group_shapes_extended
)

prediction_side_info = fm_side_info.predict(X_test_extended)
rmse = ((y_test - prediction_side_info) ** 2).mean() ** .5
mae = np.abs(y_test - prediction_side_info).mean()
print(f'rmse={rmse}, mae={mae}')
```

The result should improve further with RMSE = 0.8855, MAE = 0.6944.

Unfortunately, the running time is somewhat (~ 4 times) slower compared to the pure matrix-factorization described above. This is as it should be: the complexity of Bayesian FMs is proportional to $O(\mathrm{NNZ})$ (i.e., non-zero elements of input sparse matrix), and we have incorporated various non-zero elements (user/item features) for each row.

Surprisingly, we can still train the equivalent model in a running time close to pure MF if represent the data in Relational Data Format. See *next section* for how Relational Data Format works.

CHAPTER

THREE

TIMESVD++ FLIPPED WITH RELATION BLOCKS

As mentioned in the *Movielens example*, the complexity of Bayesian FMs is proportional to $O(\mathrm{NNZ})$. This is especially troublesome when we include SVD++-like features in the feature matrix. In such a case, for each user, we include all of the item IDs that the user had interacted with, and the complexity grows further by a factor of $O(\mathrm{NNZ}/N_U)$.

However, we can get away with this catastrophic complexity if we notice the repeated pattern in the input matrix. Interested readers can refer to [Rendle, '13] and libFM's Manual for details.

Below let us see how we can incorporate SVD++-like features efficiently using the relational data again using Movielens 100K dataset.

3.1 Building SVD++ Features

In [Rendle, et al., '19], in addition to the user/movie id, they have made use of the following features to improve the accuracy considerably:

1. User Implicit Features: All the movies the user had watched
2. Movie Implicit Features: All the users who have watched the movie
3. Time Variable: The day of watch event (regarded as a categorical variable)

Let us construct these features.

```
from collections import defaultdict
import numpy as np
from sklearn.preprocessing import OneHotEncoder
from sklearn import metrics
import myfm
from myfm import RelationBlock
from scipy import sparse as sps

from myfm.utils.benchmark_data import MovieLens100kDataManager

data_manager = MovieLens100kDataManager()

# fold 1 is the toughest one
df_train, df_test = data_manager.load_rating_predefined_split(fold=1)

date_ohe = OneHotEncoder(handle_unknown='ignore').fit(
    df_train.timestamp.dt.date.values.reshape(-1, 1)
)
def categorize_date(df):
```

(continues on next page)

(continued from previous page)

```
    return date_ohe.transform(df.timestamp.dt.date.values[:, np.newaxis])

# index "0" is reserved for unknown ids.
user_to_index = defaultdict(lambda : 0, { uid: i+1 for i,uid in enumerate(np.unique(df_
↪train.user_id)) })
movie_to_index = defaultdict(lambda: 0, { mid: i+1 for i,mid in enumerate(np.unique(df_
↪train.movie_id))})
USER_ID_SIZE = len(user_to_index) + 1
MOVIE_ID_SIZE = len(movie_to_index) + 1
```

Above we constructed dictionaries which map user/movie id to the corresponding indices. We have preserved the index ‘’0” for ‘’Unknown” user/movies, respectively.

To do the feature-engineering stated above, we have to memoize which users/movies had interactions with which movies/users.

```
# The flags to control the included features.
use_date = True # use date info or not
use_iu = True # use implicit user feature
use_ii = True # use implicit item feature

movie_vs_watched = dict()
user_vs_watched = dict()
for row in df_train.itertuples():
    user_id = row.user_id
    movie_id = row.movie_id
    movie_vs_watched.setdefault(movie_id, list()).append(user_id)
    user_vs_watched.setdefault(user_id, list()).append(movie_id)

if use_date:
    X_date_train = categorize_date(df_train)
    X_date_test  = categorize_date(df_test)
else:
    X_date_train, X_date_test = (None, None)
```

We can then define functions which maps a list of user/movie ids to the features represented in sparse matrix format:

```
# given user/movie ids, add additional infos and return it as sparse
def augment_user_id(user_ids):
    Xs = []
    X_uid = sps.lil_matrix((len(user_ids), USER_ID_SIZE))
    for index, user_id in enumerate(user_ids):
        X_uid[index, user_to_index[user_id]] = 1
    Xs.append(X_uid)
    if use_iu:
        X_iu = sps.lil_matrix((len(user_ids), MOVIE_ID_SIZE))
        for index, user_id in enumerate(user_ids):
            watched_movies = user_vs_watched.get(user_id, [])
            normalizer = 1 / max(len(watched_movies), 1) ** 0.5
            for uid in watched_movies:
                X_iu[index, movie_to_index[uid]] = normalizer
        Xs.append(X_iu)
    return sps.hstack(Xs, format='csr')
```

(continues on next page)

(continued from previous page)

```
def augment_movie_id(movie_ids):
    Xs = []
    X_movie = sps.lil_matrix((len(movie_ids), MOVIE_ID_SIZE))
    for index, movie_id in enumerate(movie_ids):
        X_movie[index, movie_to_index[movie_id]] = 1
    Xs.append(X_movie)

    if use_ii:
        X_ii = sps.lil_matrix((len(movie_ids), USER_ID_SIZE))
        for index, movie_id in enumerate(movie_ids):
            watched_users = movie_vs_watched.get(movie_id, [])
            normalizer = 1 / max(len(watched_users), 1) ** 0.5
            for uid in watched_users:
                X_ii[index, user_to_index[uid]] = normalizer
        Xs.append(X_ii)


    return sps.hstack(Xs, format='csr')
```

3.2 A naive way

We now setup the problem in a non-relational way:

```
train_uid_unique, train_uid_index = np.unique(df_train.user_id, return_inverse=True)
train_mid_unique, train_mid_index = np.unique(df_train.movie_id, return_inverse=True)
user_data_train = augment_user_id(train_uid_unique)
movie_data_train = augment_movie_id(train_mid_unique)

test_uid_unique, test_uid_index = np.unique(df_test.user_id, return_inverse=True)
test_mid_unique, test_mid_index = np.unique(df_test.movie_id, return_inverse=True)
user_data_test = augment_user_id(test_uid_unique)
movie_data_test = augment_movie_id(test_mid_unique)

X_train_naive = sps.hstack([
    X_date_train,
    user_data_train[train_uid_index],
    movie_data_train[train_mid_index]
])

X_test_naive = sps.hstack([
    X_date_test,
    user_data_test[test_uid_index],
    movie_data_test[test_mid_index]
])

fm_naive = myfm.MyFMRegressor(rank=10).fit(X_train_naive, df_train.rating, n_iter=3, n_
↪kept_samples=3)
```

In my environment, it takes ~ 2s per iteration, which is much slower than pure MF example.

3.3 The problem formulation with RelationBlock.

In the above code, we have already seen a hint to optimize the performance. The line

```
user_data_train[train_uid_index],
```

says that each row of the sparse matrix `user_data_train` appears many times, and we will compute the same combination of factors repeatedly.

The role of *myfm.RelationBlock* is to tell such a repeated pattern explicitly so that we can drastically reduce the complexity.

```
block_user_train = RelationBlock(train_uid_index, user_data_train)
block_movie_train = RelationBlock(train_mid_index, movie_data_train)
block_user_test = RelationBlock(test_uid_index, user_data_test)
block_movie_test = RelationBlock(test_mid_index, movie_data_test)
```

We can now feed these blocks into `myfm.MyFMRegressor.fit()` by

```
fm_rb = myfm.MyFMRegressor(rank=10).fit(
    X_date_train, df_train.rating,
    X_rel=[block_user_train, block_movie_train],
    n_iter=300, n_kept_samples=300
)
```

Note that we cannot express `X_date_train` as a relation block and we have supplied such a non-repeated data for the first argument. This time, the speed is 20 iters / s, almost 40x speed up compared to the naive version. This is also much faster than e.g., Surprise's implementation of SVD++.

What the relation format does is to reorganize the computation, but the result should be the same up to floating point artifacts:

```
for i in range(3):
    sample_naive = fm_naive.w_samples[i]
    sample_rb = fm_rb.w_samples[i]
    assert(np.max(np.abs(sample_naive - sample_rb)) < 1e-5)
    # should print tiny numbers
```

The resulting performance measures are RMSE=0.889, MAE=0.7000 :

```
test_prediction = fm_rb.predict(
    X_date_test,
    X_rel=[block_user_test, block_user_test]
)
rmse = ((df_test.rating.values - test_prediction) ** 2).mean() ** 0.5
mae = np.abs(df_test.rating.values - test_prediction).mean()
print(f'rmse={rmse}, mae={mae}')
```

Note that we still haven't exploited all the available ingredients such as user/item side-information and *grouping of the input variables*. See also examples notebooks & scripts for further improved results.

CHAPTER

FOUR

ORDINAL REGRESSION TUTORIAL

4.1 UCLA Dataset

Let us first explain the API of *myfm.MyFMOrderedProbit* using UCLA dataset.

The data description says

> This hypothetical data set has a three level variable called apply, with levels "unlikely", "somewhat likely", and "very likely", coded 1, 2, and 3, respectively, that we will use as our outcome variable. We also have three variables that we will use as predictors: pared, which is a 0/1 variable indicating whether at least one parent has a graduate degree; public, which is a 0/1 variable where 1 indicates that the undergraduate institution is public and 0 private, and gpa, which is the student's grade point average.

We can read the data (in Stata format) using pandas:

```
import pandas as pd
df = pd.read_stata("https://stats.idre.ucla.edu/stat/data/ologit.dta")
df.head()
```

It should print

	apply	pared	public	gpa
0	very likely	0	0	3.26
1	somewhat likely	1	0	3.21
2	unlikely	1	1	3.94
3	somewhat likely	0	0	2.81
4	somewhat likely	0	0	2.53

We regard the target label `apply` as a ordinal categorical variable,

$$(\text{unlikely} = 0) < (\text{somewhat likely} = 1) < (\text{very likely} = 2)$$

so we map `apply` as

```
y = df['apply'].map({'unlikely': 0, 'somewhat likely': 1, 'very likely': 2}).values
```

Prepare other features as usual.

```
from sklearn.model_selection import train_test_split
from sklearn import metrics

X = df[['pared', 'public', 'gpa']].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Now we can feed the data into *myfm.MyFMOrderedProbit*.

```
from myfm import MyFMOrderedProbit
clf = MyFMOrderedProbit(rank=0).fit(X_train, y_train, n_iter=200)

p = clf.predict_proba(X_test)

print(f'rmse={metrics.log_loss(y_test, p)}')
# ~ 0.84, slightly better than constant model baseline.
```

Note that unlike binary probit regression, *MyFMOrderedProbit.predict_proba()* returns 2D (N_item x N_class) array of class probability.

4.2 Movielens ratings as ordinal outcome

Let us now turn back to *Movielens 100K tutorial*.

Although we have treated movie ratings as a real target variable and used *MyFMRegressor*, it is more natural to regard them as ordinal outcomes, as there are no guarantee that the difference between rating 4 vs 5 is equivalent to the one with rating 2 vs 3.

So let us see what happens if we instead use *MyFMOrderedProbit* to predict the rating. If you have followed the steps through *the previous ''grouping" section*, you can train our ordered probit regressor by

```
import numpy as np
from sklearn.preprocessing import MultiLabelBinarizer, OneHotEncoder
from sklearn import metrics

import myfm
from myfm.utils.benchmark_data import MovieLens100kDataManager

FM_RANK = 10

data_manager = MovieLens100kDataManager()
df_train, df_test = data_manager.load_rating_predefined_split(fold=3)

FEATURE_COLUMNS = ['user_id', 'movie_id']
ohe = OneHotEncoder(handle_unknown='ignore')

X_train = ohe.fit_transform(df_train[FEATURE_COLUMNS])
X_test = ohe.transform(df_test[FEATURE_COLUMNS])
y_train = df_train.rating.values
y_test = df_test.rating.values

fm = myfm.MyFMOrderedProbit(
    rank=FM_RANK, random_seed=42,
)
fm.fit(
    X_train, y_train - 1, n_iter=300, n_kept_samples=300,
    group_shapes=[len(group) for group in ohe.categories_]
)
```

Note that we have used `y_train - 1` instead of `y_train`, because rating `r` should be regarded as class `r-1`.

We can predict the class probability given `X_test` as

```
p_ordinal = fm.predict_proba(X_test)
```

and the expected rating as

```
expected_rating = p_ordinal.dot(np.arange(1, 6))
rmse = ((y_test - expected_rating) ** 2).mean() ** .5
mae = np.abs(y_test - expected_rating).mean()
print(f'rmse={rmse}, mae={mae}')
```

which gives us RMSE=0.8906 and MAE=0.6985, a slight improvement over the regression case.

To see why it had an advantage over regression, let us check the posterior samples for the cutpoint parameters.

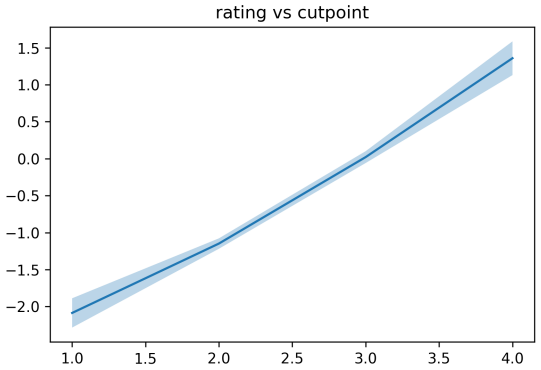
```
cutpoints = fm.cutpoint_samples - fm.w0_samples[:, None]
```

You can see how rating boundaries vs cutpoints looks like.

```
from matplotlib import pyplot as plt
cp_mean = cutpoints.mean(axis=0)
cp_std = cutpoints.std(axis=0)

plt.plot(np.arange(1, 5), cp_mean);
plt.fill_between(
    np.arange(1, 5), cp_mean + 2*cp_std, cp_mean - 2 * cp_std,
    alpha=0.3
)
plt.title('rating boundary vs cutpoint')
```

This will give you the following figure. The line is slightly non-linear, which may explain the advantage of the formulation in ordinal regression.



You can also improve the performance for Movielens 1M & 10M dataset. See our examples directory.

CHAPTER

FIVE

API REFERENCES

5.1 Training API

RelationBlock	The RelationBlock Class.
MyFMRegressor	alias of *myfm.gibbs.MyFMGibbsRegressor*
MyFMClassifier	alias of *myfm.gibbs.MyFMGibbsClassifier*
MyFMGibbsRegressor(rank[, init_stdev, ...])	
MyFMGibbsClassifier(rank[, init_stdev, ...])	Bayesian Factorization Machines for binary classification tasks.
MyFMOrderedProbit(rank[, init_stdev, ...])	Bayesian Factorization Machines for Ordinal Regression Tasks.
VariationalFMRegressor(rank[, init_stdev, ...])	Variational Inference for Regression Task.
VariationalFMClassifier(rank[, init_stdev, ...])	Variational Inference for Classification Task.

5.1.1 myfm.RelationBlock

class myfm.**RelationBlock**

Bases: pybind11_builtins.pybind11_object

The RelationBlock Class.

__init__(*self:* myfm._myfm.RelationBlock, *original_to_block: List[int]*, *data: scipy.sparse.csr_matrix[numpy.float64]*) → None

Initializes relation block.

Parameters

- **original_to_block** (*List[int]*) – describes which entry points to to which row of the data (second argument).
- **data** (*scipy.sparse.csr_matrix[float64]*) – describes repeated pattern.

Note: The entries of *original_to_block* must be in the [0, data.shape[0]-1].

Methods

`__init__`(self, original_to_block, data)	Initializes relation block.

Attributes

block_size
data
feature_size
mapper_size
original_to_block

5.1.2 myfm.MyFMRegressor

myfm.**MyFMRegressor**
alias of *myfm.gibbs.MyFMGibbsRegressor*

5.1.3 myfm.MyFMClassifier

myfm.**MyFMClassifier**
alias of *myfm.gibbs.MyFMGibbsClassifier*

5.1.4 myfm.MyFMGibbsRegressor

class myfm.**MyFMGibbsRegressor**(*rank: int*, *init_stdev: float = 0.1*, *random_seed: int = 42*, *alpha_0: float = 1.0*, *beta_0: float = 1.0*, *gamma_0: float = 1.0*, *mu_0: float = 0.0*, *reg_0: float = 1.0*, *fit_w0: bool = True*, *fit_linear: bool = True*)

Bases: myfm.base.RegressorMixin[myfm._myfm.FM, myfm._myfm.FMHyperParameters], myfm.gibbs.MyFMGibbsBase

__init__(*rank: int*, *init_stdev: float = 0.1*, *random_seed: int = 42*, *alpha_0: float = 1.0*, *beta_0: float = 1.0*, *gamma_0: float = 1.0*, *mu_0: float = 0.0*, *reg_0: float = 1.0*, *fit_w0: bool = True*, *fit_linear: bool = True*)

Setup the configuration.

Parameters

- **rank** (*int*) – The number of factors.
- **init_stdev** (*float, optional (defalult = 0.1)*) – The standard deviation for initialization. The factorization machine weights are randomely sampled from *Normal(0, init_stdev ** 2)*.
- **random_seed** (*integer, optional (default = 0.1)*) – The random seed used inside the whole learning process.

- **alpha_0** (*float, optional (default = 1.0)*) – The half of alpha parameter for the gamma-distribution prior for alpha, lambda_w and lambda_V. Together with beta_0, the priors for these parameters are alpha, lambda_w, lambda_v ~ Gamma(alpha_0 / 2, beta_0 / 2)
- **beta_0** (*float, optional (default = 1.0)*) – See the explanation for alpha_0 .
- **gamma_0** (*float, optional (default = 1.0)*) – Inverse variance of the prior for mu_w, mu_v. Together with mu_0, the priors for these parameters are mu_w, mu_v ~ Normal(mu_0, 1 / gamma_0)
- **mu_0** (*float, optional (default = 0.0)*) – See the explanation for gamma_0.
- **reg_0** (*float, optional (default = 0.0)*) – Inverse variance of the prior for w0. w0 ~ Normal(0, 1 / reg_0)
- **fit_w0** (*bool, optional (default = True)*) – whether to fit w0, by default True.
- **fit_linear** (*bool, optional (default = True)*) – whether to fit linear coefficients, by default True.

Methods

__init__(rank[, init_stdev, random_seed, ...])	Setup the configuration.
fit(X, y[, X_rel, X_test, y_test, ...])	Performs Gibbs sampling to fit the data.
get_hyper_trace()	
predict(X[, X_rel, n_workers])	Make a prediction by compute the posterior predictive mean.

Attributes

V_samples	Obtain the Gibbs samples for factorized quadratic coefficient *V*.
w0_samples	Obtain samples for global bias *w0*.
w_samples	Obtain the Gibbs samples for linear coefficients *w*.

property V_samples: Optional[object]

Obtain the Gibbs samples for factorized quadratic coefficient *V*. Returns *None* if the model is not fit yet.

Returns: Samples for lienar coefficients. The first dimension is for the sample index, the second for the feature index, and the third for the factorized dimension.

fit(*X: Union[numpy.ndarray, scipy.sparse.csr.csr_matrix], y: numpy.ndarray, X_rel: List[myfm._myfm.RelationBlock] = [], X_test: Optional[Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]] = None, y_test: Optional[numpy.ndarray] = None, X_rel_test: List[myfm._myfm.RelationBlock] = [], n_iter: int = 100, n_kept_samples: Optional[int] = None, grouping: Optional[List[int]] = None, group_shapes: Optional[List[int]] = None, callback: Optional[Callable[[int, myfm._myfm.FM, myfm._myfm.FMHyperParameters, myfm._myfm.LearningHistory], Tuple[bool, Optional[str]]]] = None, config_builder: Optional[myfm._myfm.ConfigBuilder] = None*) → *myfm.gibbs.MyFMGibbsRegressor*

Performs Gibbs sampling to fit the data.

Parameters

- **X** (*2D array-like.*) – Input variable.
- **y** (*1D array-like.*) – Target variable.
- **X_rel** (*list of RelationBlock, optional (default=[])*) – Relation blocks which supplements X.
- **n_iter** (*int, optional (default = 100)*) – Iterations to perform.
- **n_kept_samples** (*int, optional (default = None)*) – The number of samples to store. If *None*, the value is set to *n_iter* - 5.
- **grouping** (*Integer List, optional (default = None)*) – If not *None*, this specifies which column of X belongs to which group. That is, if grouping[i] is g, then, w_i and $V_{i,r}$ will be distributed according to $\mathcal{N}(\mu_w[g], \lambda_w[g])$ and $\mathcal{N}(\mu_V[g,r], \lambda_V[g,r])$, respectively. If *None*, all the columns of X are assumed to belong to a single group, 0.
- **group_shapes** (*Integer array, optional (default = None)*) – If not *None*, this specifies each variable group's size. Ignored if grouping is not None. For example, if `group_shapes = [n_1, n_2]`, this is equivalent to `grouping = [0] * n_1 + [1] * n_2`
- **callback** (*function(int, fm, hyper, history) -> (bool, str), optional(default = None)*) – Called at the every end of each Gibbs iteration.

predict(*X: Optional[Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]]*, *X_rel: List[*myfm._myfm.RelationBlock*] = []*, *n_workers: Optional[int] = None*) → object

Make a prediction by compute the posterior predictive mean.

Parameters

- **X** (*Optional[ArrayLike]*) – Main table. When None, treated as a matrix with no column.
- **X_rel** (*List[*RelationBlock*]*) – Relations.
- **n_workers** (*Optional[int], optional*) – The number of threads to compute the posterior predictive mean, by default None

Return type One-dimensional array of predictions.

property w0_samples: Optional[object]

Obtain samples for global bias *w0*. If the model is not fit yet, return *None*.

Returns: Samples for lienar coefficients.

property w_samples: Optional[object]

Obtain the Gibbs samples for linear coefficients *w*. Returns *None* if the model is not fit yet.

Returns: Samples for lienar coefficients. The first dimension is for the sample index, and the second for the feature index.

5.1.5 myfm.MyFMGibbsClassifier

class myfm.**MyFMGibbsClassifier**(*rank: int*, *init_stdev: float = 0.1*, *random_seed: int = 42*, *alpha_0: float = 1.0*, *beta_0: float = 1.0*, *gamma_0: float = 1.0*, *mu_0: float = 0.0*, *reg_0: float = 1.0*, *fit_w0: bool = True*, *fit_linear: bool = True*)

Bases: `myfm.base.ClassifierMixin`[`myfm._myfm.FM`, `myfm._myfm.FMHyperParameters`], `myfm.gibbs.MyFMGibbsBase`

Bayesian Factorization Machines for binary classification tasks.

__init__(*rank: int*, *init_stdev: float = 0.1*, *random_seed: int = 42*, *alpha_0: float = 1.0*, *beta_0: float = 1.0*, *gamma_0: float = 1.0*, *mu_0: float = 0.0*, *reg_0: float = 1.0*, *fit_w0: bool = True*, *fit_linear: bool = True*)

Setup the configuration.

Parameters

- **rank** (*int*) – The number of factors.
- **init_stdev** (*float, optional (defalult = 0.1)*) – The standard deviation for initialization. The factorization machine weights are randomely sampled from *Normal(0, init_stdev ** 2)*.
- **random_seed** (*integer, optional (default = 0.1)*) – The random seed used inside the whole learning process.
- **alpha_0** (*float, optional (default = 1.0)*) – The half of alpha parameter for the gamma-distribution prior for alpha, lambda_w and lambda_V. Together with beta_0, the priors for these parameters are alpha, lambda_w, lambda_v ~ Gamma(alpha_0 / 2, beta_0 / 2)
- **beta_0** (*float, optional (default = 1.0)*) – See the explanation for alpha_0 .
- **gamma_0** (*float, optional (default = 1.0)*) – Inverse variance of the prior for mu_w, mu_v. Together with mu_0, the priors for these parameters are mu_w, mu_v ~ Normal(mu_0, 1 / gamma_0)
- **mu_0** (*float, optional (default = 0.0)*) – See the explanation for gamma_0.
- **reg_0** (*float, optional (default = 0.0)*) – Inverse variance of the prior for w0. w0 ~ Normal(0, 1 / reg_0)
- **fit_w0** (*bool, optional (default = True)*) – whether to fit w0, by default True.
- **fit_linear** (*bool, optional (default = True)*) – whether to fit linear coefficients, by default True.

Methods

__init__(rank[, init_stdev, random_seed, ...])	Setup the configuration.
fit(X, y[, X_rel, X_test, y_test, ...])	Performs Gibbs sampling to fit the data.
get_hyper_trace()	
predict(X[, X_rel, n_workers])	Based on the class probability, return binary classified outcome based on threshold = 0.5.
predict_proba(X[, X_rel, n_workers])	Compute the probability that the outcome will be 1 based on posterior predictive mean.

Attributes

V_samples	Obtain the Gibbs samples for factorized quadratic coefficient *V*.
w0_samples	Obtain samples for global bias *w0*.
w_samples	Obtain the Gibbs samples for linear coefficients *w*.

property V_samples: Optional[object]

Obtain the Gibbs samples for factorized quadratic coefficient *V*. Returns *None* if the model is not fit yet.

Returns: Samples for lienar coefficients. The first dimension is for the sample index, the second for the feature index, and the third for the factorized dimension.

fit(*X: Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]*, *y: object*, *X_rel: List[myfm._myfm.RelationBlock] = []*, *X_test: Optional[Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]] = None*, *y_test: Optional[numpy.ndarray] = None*, *X_rel_test: List[myfm._myfm.RelationBlock] = []*, *n_iter: int = 100*, *n_kept_samples: Optional[int] = None*, *grouping: Optional[List[int]] = None*, *group_shapes: Optional[List[int]] = None*, *callback: Optional[Callable[[int, myfm._myfm.FM, myfm._myfm.FMHyperParameters, myfm._myfm.LearningHistory], Tuple[bool, Optional[str]]]] = None*, *config_builder: Optional[myfm._myfm.ConfigBuilder] = None*) → *myfm.gibbs.MyFMGibbsClassifier*

Performs Gibbs sampling to fit the data.

Parameters

- **X** (*2D array-like.*) – Input variable.
- **y** (*1D array-like.*) – Target variable.
- **X_rel** (*list of RelationBlock, optional (default=[])*) – Relation blocks which supplements X.
- **n_iter** (*int, optional (default = 100)*) – Iterations to perform.
- **n_kept_samples** (*int, optional (default = None)*) – The number of samples to store. If *None*, the value is set to *n_iter* - 5.
- **grouping** (*Integer List, optional (default = None)*) – If not *None*, this specifies which column of X belongs to which group. That is, if grouping[i] is g, then, w_i and $V_{i,r}$ will be distributed according to $\mathcal{N}(\mu_w[g], \lambda_w[g])$ and $\mathcal{N}(\mu_V[g,r], \lambda_V[g,r])$, respectively. If *None*, all the columns of X are assumed to belong to a single group, 0.
- **group_shapes** (*Integer array, optional (default = None)*) – If not *None*, this specifies each variable group's size. Ignored if grouping is not None. For example, if `group_shapes = [n_1, n_2]`, this is equivalent to `grouping = [0] * n_1 + [1] * n_2`
- **callback** (*function(int, fm, hyper, history) -> (bool, str), optional(default = None)*) – Called at the every end of each Gibbs iteration.

predict(*X: Optional[Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]]*, *X_rel: List[myfm._myfm.RelationBlock] = []*, *n_workers: Optional[int] = None*) → object

Based on the class probability, return binary classified outcome based on threshold = 0.5. If you want class probability instead, use *predict_proba* method.

Parameters

- **X** (*Optional[ArrayLike]*) – When None, treated as a matrix with no column.
- **X_rel** (*List[RelationBlock]*) – Relations.

- **n_workers** (*Optional[int], optional*) – The number of threads to compute the posterior predictive mean, by default None

Returns One-dimensional array of predicted outcomes.

Return type np.ndarray

predict_proba(*X: Optional[Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]]*, *X_rel: List[myfm._myfm.RelationBlock] = []*, *n_workers: Optional[int] = None*) → object

Compute the probability that the outcome will be 1 based on posterior predictive mean.

Parameters

- **X** (*Optional[ArrayLike]*) – When None, treated as a matrix with no column.
- **X_rel** (*List[RelationBlock]*) – Relations.
- **n_workers** (*Optional[int], optional*) – The number of threads to compute the posterior predictive mean, by default None

Returns One-dimensional array of probabilities.

Return type np.ndarray

property w0_samples: Optional[object]

Obtain samples for global bias *w0*. If the model is not fit yet, return *None*.

Returns: Samples for lienar coefficients.

property w_samples: Optional[object]

Obtain the Gibbs samples for linear coefficients *w*. Returns *None* if the model is not fit yet.

Returns: Samples for lienar coefficients. The first dimension is for the sample index, and the second for the feature index.

5.1.6 myfm.MyFMOrderedProbit

class myfm.**MyFMOrderedProbit**(*rank: int*, *init_stdev: float = 0.1*, *random_seed: int = 42*, *alpha_0: float = 1*, *beta_0: float = 1*, *gamma_0: float = 1*, *mu_0: float = 0*, *reg_0: float = 1*, *fit_w0: bool = True*, *fit_linear: bool = True*)

Bases: myfm.gibbs.MyFMGibbsBase

Bayesian Factorization Machines for Ordinal Regression Tasks.

__init__(*rank: int*, *init_stdev: float = 0.1*, *random_seed: int = 42*, *alpha_0: float = 1*, *beta_0: float = 1*, *gamma_0: float = 1*, *mu_0: float = 0*, *reg_0: float = 1*, *fit_w0: bool = True*, *fit_linear: bool = True*)

Setup the configuration.

Parameters

- **rank** (*int*) – The number of factors.
- **init_stdev** (*float, optional (defalult = 0.1)*) – The standard deviation for initialization. The factorization machine weights are randomely sampled from *Normal(0, init_stdev ** 2)*.
- **random_seed** (*integer, optional (default = 0.1)*) – The random seed used inside the whole learning process.
- **alpha_0** (*float, optional (default = 1.0)*) – The half of alpha parameter for the gamma-distribution prior for alpha, lambda_w and lambda_V. Together with beta_0, the

priors for these parameters are alpha, lambda_w, lambda_v ~ Gamma(alpha_0 / 2, beta_0 / 2)

- **beta_0** (*float, optional (default = 1.0)*) – See the explanation for alpha_0 .
- **gamma_0** (*float, optional (default = 1.0)*) – Inverse variance of the prior for mu_w, mu_v. Together with mu_0, the priors for these parameters are mu_w, mu_v ~ Normal(mu_0, 1 / gamma_0)
- **mu_0** (*float, optional (default = 0.0)*) – See the explanation for gamma_0.
- **reg_0** (*float, optional (default = 0.0)*) – Inverse variance of the prior for w0. w0 ~ Normal(0, 1 / reg_0)
- **fit_w0** (*bool, optional (default = True)*) – whether to fit w0, by default True.
- **fit_linear** (*bool, optional (default = True)*) – whether to fit linear coefficients, by default True.

Methods

__init__(rank[, init_stdev, random_seed, ...])	Setup the configuration.
fit(X, y[, X_rel, X_test, y_test, ...])	
get_hyper_trace()	
predict(X[, X_rel])	Predict the class outcome according to the class probability.
predict_proba(X[, X_rel, n_workers])	Compute the ordinal class probability.

Attributes

V_samples	Obtain the Gibbs samples for factorized quadratic coefficient *V*.
cutpoint_samples	Obtain samples for the cutpoints.
w0_samples	Obtain samples for global bias *w0*.
w_samples	Obtain the Gibbs samples for linear coefficients *w*.

property V_samples: Optional[object]

Obtain the Gibbs samples for factorized quadratic coefficient *V*. Returns *None* if the model is not fit yet.

Returns: Samples for lienar coefficients. The first dimension is for the sample index, the second for the feature index, and the third for the factorized dimension.

property cutpoint_samples: Optional[object]

Obtain samples for the cutpoints. If the model is not fit yet, return *None*.

Returns: Samples for cutpoints.

predict(*X: Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]*, *X_rel: List[*myfm._myfm.RelationBlock*] = []*) → object

Predict the class outcome according to the class probability.

Parameters

- **X** (*array_like*) – The input data.

- **X_rel** (*List[RelationBlock], optional*) – Relational Block part of the data., by default []

Returns The class prediction

Return type np.int64

predict_proba(*X: Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]*, *X_rel: List[myfm._myfm.RelationBlock] = []*, *n_workers: Optional[int] = None*) → numpy.ndarray

Compute the ordinal class probability.

Parameters

- **X** (*array_like*) – The input data.
- **X_rel** (*List[RelationBlock], optional*) – Relational Block part of the data., by default []

Returns The class probability

Return type np.float

property w0_samples: Optional[object]

Obtain samples for global bias *w0*. If the model is not fit yet, return *None*.

Returns: Samples for lienar coefficients.

property w_samples: Optional[object]

Obtain the Gibbs samples for linear coefficients *w*. Returns *None* if the model is not fit yet.

Returns: Samples for lienar coefficients. The first dimension is for the sample index, and the second for the feature index.

5.1.7 myfm.VariationalFMRegressor

class myfm.**VariationalFMRegressor**(*rank: int*, *init_stdev: float = 0.1*, *random_seed: int = 42*, *alpha_0: float = 1.0*, *beta_0: float = 1.0*, *gamma_0: float = 1.0*, *mu_0: float = 0.0*, *reg_0: float = 1.0*, *fit_w0: bool = True*, *fit_linear: bool = True*)

Bases: myfm.base.RegressorMixin[myfm._myfm.VariationalFM, myfm._myfm.VariationalFMHyperParameters], myfm.variational.MyFMVariationalBase

Variational Inference for Regression Task.

__init__(*rank: int*, *init_stdev: float = 0.1*, *random_seed: int = 42*, *alpha_0: float = 1.0*, *beta_0: float = 1.0*, *gamma_0: float = 1.0*, *mu_0: float = 0.0*, *reg_0: float = 1.0*, *fit_w0: bool = True*, *fit_linear: bool = True*)

Setup the configuration.

Parameters

- **rank** (*int*) – The number of factors.
- **init_stdev** (*float, optional (defalult = 0.1)*) – The standard deviation for initialization. The factorization machine weights are randomely sampled from *Normal(0, init_stdev ** 2)*.
- **random_seed** (*integer, optional (default = 0.1)*) – The random seed used inside the whole learning process.
- **alpha_0** (*float, optional (default = 1.0)*) – The half of alpha parameter for the gamma-distribution prior for alpha, lambda_w and lambda_V. Together with beta_0, the

priors for these parameters are alpha, lambda_w, lambda_v ~ Gamma(alpha_0 / 2, beta_0 / 2)

- **beta_0** (*float, optional (default = 1.0)*) – See the explanation for alpha_0 .
- **gamma_0** (*float, optional (default = 1.0)*) – Inverse variance of the prior for mu_w, mu_v. Together with mu_0, the priors for these parameters are mu_w, mu_v ~ Normal(mu_0, 1 / gamma_0)
- **mu_0** (*float, optional (default = 0.0)*) – See the explanation for gamma_0.
- **reg_0** (*float, optional (default = 0.0)*) – Inverse variance of the prior for w0. w0 ~ Normal(0, 1 / reg_0)
- **fit_w0** (*bool, optional (default = True)*) – whether to fit w0, by default True.
- **fit_linear** (*bool, optional (default = True)*) – whether to fit linear coefficients, by default True.

Methods

__init__(rank[, init_stdev, random_seed, ...])	Setup the configuration.
fit(X, y[, X_rel, X_test, y_test, ...])	Performs batch variational inference fit the data.
predict(X[, X_rel])	Make a prediction based on variational mean.

Attributes

V_mean	Mean of variational posterior distribution of factorized quadratic coefficnent *V*.
V_var	Variance of variational posterior distribution of factorized quadratic coefficnent *V*.
w0_mean	Mean of variational posterior distribution of global bias *w0*.
w0_var	Variance of variational posterior distribution of global bias *w0*.
w_mean	Mean of variational posterior distribution of linear coefficnent *w*.
w_var	Variance of variational posterior distribution of linear coefficnent *w*.

property V_mean: Optional[numpy.ndarray]

Mean of variational posterior distribution of factorized quadratic coefficnent *V*. If the model is not fit yet, returns *None*.

Returns: Mean of variational posterior distribution of factorized quadratic coefficient *V*.

property V_var: Optional[numpy.ndarray]

Variance of variational posterior distribution of factorized quadratic coefficnent *V*. If the model is not fit yet, returns *None*.

Returns: Variance of variational posterior distribution of factorized quadratic coefficient *V*.

fit(*X: Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]*, *y: numpy.ndarray*, *X_rel: List[myfm._myfm.RelationBlock] = []*, *X_test: Optional[Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]] = None*, *y_test: Optional[numpy.ndarray] = None*, *X_rel_test: List[myfm._myfm.RelationBlock] = []*, *n_iter: int = 100*, *grouping: Optional[List[int]] = None*, *group_shapes: Optional[List[int]] = None*, *callback: Optional[Callable[[int, myfm._myfm.VariationalFM, myfm._myfm.VariationalFMHyperParameters, myfm._myfm.VariationalLearningHistory], Tuple[bool, Optional[str]]]] = None*, *config_builder: Optional[myfm._myfm.ConfigBuilder] = None*) → *myfm.variational.VariationalFMRegressor*

Performs batch variational inference fit the data.

Parameters

- **X** (*2D array-like.*) – Input variable.
- **y** (*1D array-like.*) – Target variable.
- **X_rel** (*list of RelationBlock, optional (default=[])*) – Relation blocks which supplements X.
- **n_iter** (*int, optional (default = 100)*) – Iterations to perform.
- **grouping** (*Integer List, optional (default = None)*) – If not *None*, this specifies which column of X belongs to which group. That is, if grouping[i] is g, then, w_i and $V_{i,r}$ will be distributed according to $\mathcal{N}(\mu_w[g], \lambda_w[g])$ and $\mathcal{N}(\mu_V[g,r], \lambda_V[g,r])$, respectively. If *None*, all the columns of X are assumed to belong to a single group, 0.
- **group_shapes** (*Integer array, optional (default = None)*) – If not *None*, this specifies each variable group's size. Ignored if grouping is not None. For example, if `group_shapes = [n_1, n_2]`, this is equivalent to `grouping = [0] * n_1 + [1] * n_2`
- **callback** (*function(int, fm, hyper, history) -> bool, optional(default = None)*) – Called at the every end of each Gibbs iteration.

predict(*X: Optional[Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]]*, *X_rel: List[myfm._myfm.RelationBlock] = []*) → numpy.ndarray

Make a prediction based on variational mean.

Parameters

- **X** (*Optional[ArrayLike]*) – Main Table. When None, treated as a matrix without columns.
- **X_rel** (*List[RelationBlock], optional*) – Relations, by default []

Returns [description]

Return type np.ndarray

property w0_mean: Optional[float]

Mean of variational posterior distribution of global bias *w0*. If the model is not fit yet, returns *None*.

Returns: Mean of variational posterior distribution of global bias *w0*.

property w0_var: Optional[float]

Variance of variational posterior distribution of global bias *w0*. If the model is not fit yet, returns *None*.

Returns: Variance of variational posterior distribution of global bias *w0*.

property w_mean: Optional[numpy.ndarray]

Mean of variational posterior distribution of linear coefficnent *w*. If the model is not fit yet, returns *None*.

Returns: Mean of variational posterior distribution of linear coefficnent *w*.

property w_var: Optional[numpy.ndarray]

Variance of variational posterior distribution of linear coefficnent *w*. If the model is not fit yet, returns *None*.

Returns: Variance of variational posterior distribution of linear coefficnent *w*.

5.1.8 myfm.VariationalFMClassifier

class myfm.**VariationalFMClassifier**(*rank: int*, *init_stdev: float = 0.1*, *random_seed: int = 42*, *alpha_0: float = 1.0*, *beta_0: float = 1.0*, *gamma_0: float = 1.0*, *mu_0: float = 0.0*, *reg_0: float = 1.0*, *fit_w0: bool = True*, *fit_linear: bool = True*)

Bases: myfm.base.ClassifierMixin[myfm._myfm.VariationalFM, myfm._myfm.VariationalFMHyperParameters], myfm.variational.MyFMVariationalBase

Variational Inference for Classification Task.

__init__(*rank: int*, *init_stdev: float = 0.1*, *random_seed: int = 42*, *alpha_0: float = 1.0*, *beta_0: float = 1.0*, *gamma_0: float = 1.0*, *mu_0: float = 0.0*, *reg_0: float = 1.0*, *fit_w0: bool = True*, *fit_linear: bool = True*)

Setup the configuration.

Parameters

- **rank** (*int*) – The number of factors.
- **init_stdev** (*float, optional (defalult = 0.1)*) – The standard deviation for initialization. The factorization machine weights are randomely sampled from *Normal(0, init_stdev ** 2)*.
- **random_seed** (*integer, optional (default = 0.1)*) – The random seed used inside the whole learning process.
- **alpha_0** (*float, optional (default = 1.0)*) – The half of alpha parameter for the gamma-distribution prior for alpha, lambda_w and lambda_V. Together with beta_0, the priors for these parameters are alpha, lambda_w, lambda_v ~ Gamma(alpha_0 / 2, beta_0 / 2)
- **beta_0** (*float, optional (default = 1.0)*) – See the explanation for alpha_0 .
- **gamma_0** (*float, optional (default = 1.0)*) – Inverse variance of the prior for mu_w, mu_v. Together with mu_0, the priors for these parameters are mu_w, mu_v ~ Normal(mu_0, 1 / gamma_0)
- **mu_0** (*float, optional (default = 0.0)*) – See the explanation for gamma_0.
- **reg_0** (*float, optional (default = 0.0)*) – Inverse variance of the prior for w0. w0 ~ Normal(0, 1 / reg_0)
- **fit_w0** (*bool, optional (default = True)*) – whether to fit w0, by default True.
- **fit_linear** (*bool, optional (default = True)*) – whether to fit linear coefficients, by default True.

Methods

__init__(rank[, init_stdev, random_seed, ...])	Setup the configuration.
fit(X, y[, X_rel, X_test, y_test, ...])	Performs batch variational inference fit the data.
predict(X[, X_rel])	Based on the class probability, return binary classified outcome based on threshold = 0.5.
predict_proba(X[, X_rel])	Compute the probability that the outcome will be 1 based on variational mean.

Attributes

V_mean	Mean of variational posterior distribution of factorized quadratic coefficnent *V*.
V_var	Variance of variational posterior distribution of factorized quadratic coefficnent *V*.
w0_mean	Mean of variational posterior distribution of global bias *w0*.
w0_var	Variance of variational posterior distribution of global bias *w0*.
w_mean	Mean of variational posterior distribution of linear coefficnent *w*.
w_var	Variance of variational posterior distribution of linear coefficnent *w*.

property V_mean: Optional[numpy.ndarray]

Mean of variational posterior distribution of factorized quadratic coefficnent *V*. If the model is not fit yet, returns *None*.

Returns: Mean of variational posterior distribution of factorized quadratic coefficient *V*.

property V_var: Optional[numpy.ndarray]

Variance of variational posterior distribution of factorized quadratic coefficnent *V*. If the model is not fit yet, returns *None*.

Returns: Variance of variational posterior distribution of factorized quadratic coefficient *V*.

fit(*X: Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]*, *y: numpy.ndarray*, *X_rel: List[myfm._myfm.RelationBlock] = []*, *X_test: Optional[Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]] = None*, *y_test: Optional[numpy.ndarray] = None*, *X_rel_test: List[myfm._myfm.RelationBlock] = []*, *n_iter: int = 100*, *grouping: Optional[List[int]] = None*, *group_shapes: Optional[List[int]] = None*, *callback: Optional[Callable[[int, myfm._myfm.VariationalFM, myfm._myfm.VariationalFMHyperParameters, myfm._myfm.VariationalLearningHistory], Tuple[bool, Optional[str]]]] = None*, *config_builder: Optional[myfm._myfm.ConfigBuilder] = None*) → *myfm.variational.VariationalFMClassifier*

Performs batch variational inference fit the data.

Parameters

- **X** (*Optional[ArrayLike].*) – Main table. When None, treated as a matrix without columns.
- **y** (*1D array-like.*) – Target variable.
- **X_rel** (*list of RelationBlock, optional (default=[])*) – Relation blocks which supplements X.

- **n_iter** (*int, optional (default = 100)*) – Iterations to perform.
- **grouping** (*Integer List, optional (default = None)*) – If not *None*, this specifies which column of X belongs to which group. That is, if grouping[i] is g, then, w_i and $V_{i,r}$ will be distributed according to $\mathcal{N}(\mu_w[g], \lambda_w[g])$ and $\mathcal{N}(\mu_V[g,r], \lambda_V[g,r])$, respectively. If *None*, all the columns of X are assumed to belong to a single group, 0.
- **group_shapes** (*Integer array, optional (default = None)*) – If not *None*, this specifies each variable group's size. Ignored if grouping is not None. For example, if `group_shapes = [n_1, n_2]`, this is equivalent to `grouping = [0] * n_1 + [1] * n_2`
- **callback** (*function(int, fm, hyper) -> bool, optional(default = None)*) – Called at the every end of each Gibbs iteration.

predict(*X: Optional[Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]]*, *X_rel: List[myfm._myfm.RelationBlock] = []*) → numpy.ndarray

Based on the class probability, return binary classified outcome based on threshold = 0.5. If you want class probability instead, use *predict_proba* method.

Parameters

- **X** (*Optional[ArrayLike]*) – Main Table. When None, treated as a matrix without columns.
- **X_rel** (*List[RelationBlock], optional*) – Relations, by default []

Returns 0/1 predictions based on the probability.

Return type np.ndarray

predict_proba(*X: Optional[Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]]*, *X_rel: List[myfm._myfm.RelationBlock] = []*) → numpy.ndarray

Compute the probability that the outcome will be 1 based on variational mean.

Parameters

- **X** (*Optional[ArrayLike]*) – Main Table. When None, treated as a matrix without columns.
- **X_rel** (*List[RelationBlock], optional*) – Relations, by default []

Returns the probability.

Return type np.ndarray

property w0_mean: Optional[float]

Mean of variational posterior distribution of global bias *w0*. If the model is not fit yet, returns *None*.

Returns: Mean of variational posterior distribution of global bias *w0*.

property w0_var: Optional[float]

Variance of variational posterior distribution of global bias *w0*. If the model is not fit yet, returns *None*.

Returns: Variance of variational posterior distribution of global bias *w0*.

property w_mean: Optional[numpy.ndarray]

Mean of variational posterior distribution of linear coefficnent *w*. If the model is not fit yet, returns *None*.

Returns: Mean of variational posterior distribution of linear coefficnent *w*.

property w_var: Optional[numpy.ndarray]

Variance of variational posterior distribution of linear coefficnent *w*. If the model is not fit yet, returns *None*.

Returns: Variance of variational posterior distribution of linear coefficnent w.

5.2 Benchmark Dataset

utils.benchmark_data.MovieLens100kDataManager([...])	The Data manager for MovieLens 100k dataset.
utils.benchmark_data.MovieLens1MDataManager([...])	
utils.benchmark_data.MovieLens10MDataManager([...])	

5.2.1 myfm.utils.benchmark_data.MovieLens100kDataManager

class myfm.utils.benchmark_data.**MovieLens100kDataManager**(*zippath: Optional[pathlib.Path] = None*)

Bases: myfm.utils.benchmark_data.loader_base.MovieLensBase

The Data manager for MovieLens 100k dataset.

__init__(*zippath: Optional[pathlib.Path] = None*)

Methods

__init__([zippath])	
genres()	
load_movie_info()	load movie meta information.
load_rating_all()	Load the entire rating dataset.
load_rating_kfold_split(K, fold[, random_state])	Load the entire dataset and split it into train/test set.
load_rating_predefined_split(fold)	Read the pre-defined train/test split.
load_user_info()	load user meta information.

Attributes

DEFAULT_PATH
DOWNLOAD_URL

load_movie_info() → pandas.core.frame.DataFrame

load movie meta information.

Returns A dataframe containing meta-information (id, title, release_date, url, genres) about the movies. Multiple genres per movie will be concatenated by "|".

Return type pd.DataFrame

load_rating_all() → pandas.core.frame.DataFrame
Load the entire rating dataset.

Returns all the available ratings.

Return type pd.DataFrame

load_rating_kfold_split(*K: int*, *fold: int*, *random_state: Optional[int] = 0*) → Tuple[pandas.core.frame.DataFrame, pandas.core.frame.DataFrame]
Load the entire dataset and split it into train/test set. K-fold

Parameters

- **K** (*int*) – K in the K-fold splitting scheme.
- **fold** (*int*) – fold index.
- **random_state** (*Union[np.RandomState, int, None], optional*) – Controlls random state of the split.

Returns train and test dataframes.

Return type Tuple[pd.DataFrame, pd.DataFrame]

Raises **ValueError** – When 0 <= fold < K is not met.

load_rating_predefined_split(*fold: int*) → Tuple[pandas.core.frame.DataFrame, pandas.core.frame.DataFrame]
Read the pre-defined train/test split. Fold index ranges from 1 to 5.

Parameters **fold** (*int*) – specifies the fold index.

Returns train and test dataframes.

Return type Tuple[pd.DataFrame, pd.DataFrame]

load_user_info() → pandas.core.frame.DataFrame
load user meta information.

Returns user infomation

Return type pd.DataFrame

5.2.2 myfm.utils.benchmark_data.MovieLens1MDataManager

class myfm.utils.benchmark_data.**MovieLens1MDataManager**(*zippath: Optional[pathlib.Path] = None*)
Bases: myfm.utils.benchmark_data.loader_base.MovieLensBase

__init__(*zippath: Optional[pathlib.Path] = None*)

Methods

__init__([zippath])	

load_rating_all()	Read all (1M) interactions.
load_rating_kfold_split(K, fold[, random_state])	Load the entire dataset and split it into train/test set.

Attributes

DEFAULT_PATH
DOWNLOAD_URL

load_rating_all() → pandas.core.frame.DataFrame
Read all (1M) interactions.

Returns Movielens 1M rating dataframe.

Return type pd.DataFrame

load_rating_kfold_split(*K: int*, *fold: int*, *random_state: Optional[int] = 0*) → Tuple[pandas.core.frame.DataFrame, pandas.core.frame.DataFrame]
Load the entire dataset and split it into train/test set. K-fold

Parameters

- **K** (*int*) – K in the K-fold splitting scheme.
- **fold** (*int*) – fold index.
- **random_state** (*Union[np.RandomState, int, None], optional*) – Controlls random state of the split.

Returns train and test dataframes.

Return type Tuple[pd.DataFrame, pd.DataFrame]

Raises **ValueError** – When 0 <= fold < K is not met.

5.2.3 myfm.utils.benchmark_data.MovieLens10MDataManager

class myfm.utils.benchmark_data.**MovieLens10MDataManager**(*zippath: Optional[pathlib.Path] = None*)
Bases: myfm.utils.benchmark_data.loader_base.MovieLensBase

__init__(*zippath: Optional[pathlib.Path] = None*)

Methods

__init__([zippath])	
load_rating_all()	
load_rating_kfold_split(K, fold[, random_state])	Load the entire dataset and split it into train/test set.

Attributes

DEFAULT_PATH
DOWNLOAD_URL

load_rating_kfold_split(*K: int*, *fold: int*, *random_state: Optional[int] = 0*) → Tuple[pandas.core.frame.DataFrame, pandas.core.frame.DataFrame]

Load the entire dataset and split it into train/test set. K-fold

Parameters

- **K** (*int*) – K in the K-fold splitting scheme.
- **fold** (*int*) – fold index.
- **random_state** (*Union[np.RandomState, int, None], optional*) – Controlls random state of the split.

Returns train and test dataframes.

Return type Tuple[pd.DataFrame, pd.DataFrame]

Raises **ValueError** – When 0 <= fold < K is not met.

5.3 Utilities for Sparse Matrix Construction

utils.encoders.DataFrameEncoder()	Encode pandas.DataFrame into concatenated sparse matrices.
utils.encoders.CategoryValueToSparseEncoder(...)	The class to one-hot encode a List of items into a sparse matrix representation.
utils.encoders.MultipleValuesToSparseEncoder(...)	The class to N-hot encode a List of items into a sparse matrix representation.
utils.encoders.BinningEncoder(x[, n_percentiles])	The class to one-hot encode a List of numerical values into a sparse matrix representation by binning.

5.3.1 myfm.utils.encoders.DataFrameEncoder

class myfm.utils.encoders.**DataFrameEncoder**

Bases: object

Encode pandas.DataFrame into concatenated sparse matrices.

__init__() → None

Construct the encoders starting from empty one.

Methods

__init__()	Construct the encoders starting from empty one.
add_column(colname, encoder)	Add a column name to be encoded / encoder pair.
all_names()	
encode_df(df)	Encode the dataframe into a concatenated CSR matrix.

Attributes

encoder_shapes	Show how the columns for an encoded CSR matrix are organized.

add_column(*colname: str*, *encoder: myfm.utils.encoders.base.SparseEncoderBase*) → *myfm.utils.encoders.base.DataFrameEncoder*

Add a column name to be encoded / encoder pair.

Parameters

- **colname** (*str*) – The column name to be encoded.
- **encoder** (*SparseEncoderBase*) – The corresponding encoder.

encode_df(*df: pandas.core.frame.DataFrame*) → scipy.sparse.csr.csr_matrix

Encode the dataframe into a concatenated CSR matrix.

Parameters **df** (*pd.DataFrame*) – The source.

Returns The result.

Return type sps.csr_matrix

property encoder_shapes: List[int]

Show how the columns for an encoded CSR matrix are organized.

Returns list of length of internal encoders.

Return type List[int]

5.3.2 myfm.utils.encoders.CategoryValueToSparseEncoder

class myfm.utils.encoders.**CategoryValueToSparseEncoder**(*items: typing.Iterable[myfm.utils.encoders.categorical.T]*, *min_freq: int = 1*, *handle_unknown: typing_extensions.Literal[create, ignore, raise] = 'create'*)

Bases: Generic[myfm.utils.encoders.categorical.T], myfm.utils.encoders.base.SparseEncoderBase

The class to one-hot encode a List of items into a sparse matrix representation.

__init__(*items: typing.Iterable[myfm.utils.encoders.categorical.T]*, *min_freq: int = 1*, *handle_unknown: typing_extensions.Literal[create, ignore, raise] = 'create'*)

Construct the encoder by providing a list of items.

Parameters

- **items** (*Iterable[T]*) – The items list.
- **min_freq** (*int, optional*) – The minimal frequency for an item to be retained in the known items list, by default 1
- **handle_unknown** (*Literal["create", "ignore", "raise"], optional*) – How to handle previously unseen values during encoding. If "create", then there is a single category named "__UNK__" for unknown values, ant it is treated as 0th category. If "ignore", such an item will be ignored. If "raise", a *KeyError* is raised. Defaults to "create".

Methods

__init__(items[, min_freq, handle_unknown])	Construct the encoder by providing a list of items.
names()	Description of each non-zero entry.
to_sparse(items)	

names() → List[str]
Description of each non-zero entry.

5.3.3 myfm.utils.encoders.MultipleValuesToSparseEncoder

class myfm.utils.encoders.**MultipleValuesToSparseEncoder**(*items: typing.Iterable[str], min_freq: int = 1, sep: str = ',', normalize: bool = True, handle_unknown: typing_extensions.Literal[create, ignore, raise] = 'create'*)

Bases: *myfm.utils.encoders.categorical.CategoryValueToSparseEncoder*[str]

The class to N-hot encode a List of items into a sparse matrix representation.

__init__(*items: typing.Iterable[str], min_freq: int = 1, sep: str = ',', normalize: bool = True, handle_unknown: typing_extensions.Literal[create, ignore, raise] = 'create'*)
Construct the encoder by providing a list of strings, each of which is a list of strings concatenated by *sep*.

Parameters

- **items** (*Iterable[str]*) – Iterable of strings, each of which is a concatenated list of possibly multiple items.
- **min_freq** (*int, optional*) – The minimal frequency for an item to be retained in the known items list, by default 1.
- **sep** (*str, optional*) – Tells how to separate string back into a list. Defaults to *','*.
- **normalize** (*bool, optional*) – If *True*, non-zero entry in the encoded matrix will have *1 / N ** 0.5*, where *N* is the number of non-zero entries in that row. Defaults to *True*.
- **handle_unknown** (*Literal["create", "ignore", "raise"], optional*) – How to handle previously unseen values during encoding. If "create", then there is a single category named "__UNK__" for unknown values, ant it is treated as 0th category. If "ignore", such an item will be ignored. If "raise", a *KeyError* is raised. Defaults to "create".

Methods

__init__(items[, min_freq, sep, normalize, ...])	Construct the encoder by providing a list of strings, each of which is a list of strings concatenated by *sep*.
names()	Description of each non-zero entry.
to_sparse(items)	

names() → List[str]
Description of each non-zero entry.

5.3.4 myfm.utils.encoders.BinningEncoder

class myfm.utils.encoders.**BinningEncoder**(*x: object*, *n_percentiles: int = 10*)
Bases: myfm.utils.encoders.base.SparseEncoderBase

The class to one-hot encode a List of numerical values into a sparse matrix representation by binning.

__init__(*x: object*, *n_percentiles: int = 10*) → None
Initializes the encoder by compting the percentile values of input.

Parameters

- **x** – list of numerical values.
- **n_percentiles** – number of percentiles computed against x, by default 10.

Methods

__init__(x[, n_percentiles])	Initializes the encoder by compting the percentile values of input.
names()	Description of each non-zero entry.
to_sparse(x)	

names() → List[str]
Description of each non-zero entry.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- search

INDEX

Symbols

A

B

C

D

E

F

L

M

N

P

R

V

W