DeepMatch Documentation

Release 0.1.3

Weichen Shen

May 17, 2020

Home:

DeepMatch is a deep matching model library for recommendations, advertising, and search. It's easy to **train models** and to **export representation vectors** for user and item which can be used for **ANN search**.You can use any complex model with `model.fit()` and `model.predict()` .

Let's Get Started! or Run examples!

You can read the latest code at https://github.com/shenweichen/DeepMatch

CHAPTER 1

News

05/17/2020 : Add `SDM` model. Changelog

04/10/2020 : Support saving and loading model . Changelog

04/06/2020 : DeepMatch first version .

CHAPTER 2

DisscussionGroup

wechat ID: **deepctrbot**

2.1 Quick-Start

2.1.1 Installation Guide

Now `deepmatch` is available for python `2.7`and `3.5, 3.6, 3.7`.`deepmatch` depends on tensorflow, you can specify to install the cpu version or gpu version through `pip`.

CPU version

```
$ pip install deepmatch[cpu]
```

GPU version

```
$ pip install deepmatch[gpu]
```

2.1.2 Run examples !!

- Run YoutubeDNN on MovieLen1M on Google colab
- YoutubeDNN/MIND with sampled softmax
- SDM with sampled softmax
- DSSM with negative sampling

2.2 Features

2.2.1 Feature Columns

SparseFeat

`SparseFeat` is a namedtuple with signature `SparseFeat(name, vocabulary_size, embedding_dim, use_hash, dtype,embedding_name, group_name)`

- name : feature name
- vocabulary_size : number of unique feature values for sprase feature or hashing space when `use_hash=True`
- embedding_dim : embedding dimension
- use_hash : defualt `False`.If `True` the input will be hashed to space of size `vocabulary_size`.
- dtype : default `float32`.dtype of input tensor.
- embedding_name : default `None`. If None, the embedding_name will be same as `name`.
- group_name : feature group of this feature.

DenseFeat

`DenseFeat` is a namedtuple with signature `DenseFeat(name, dimension, dtype)`

- name : feature name
- dimension : dimension of dense feature vector.
- dtype : default `float32`.dtype of input tensor.

VarLenSparseFeat

`VarLenSparseFeat` is a namedtuple with signature `VarLenSparseFeat(sparsefeat, maxlen, combiner, length_name, weight_name,weight_norm)`

- sparsefeat : a instance of `SparseFeat`
- maxlen : maximum length of this feature for all samples
- combiner : pooling method,can be `sum`,`mean` or `max`

- length_name : feature length name,if `None`, value 0 in feature is for padding.
- weight_name : default `None`. If not None, the sequence feature will be multiplyed by the feature whose name is `weight_name`.
- weight_norm : default `True`. Whether normalize the weight score or not.

2.2.2 Models

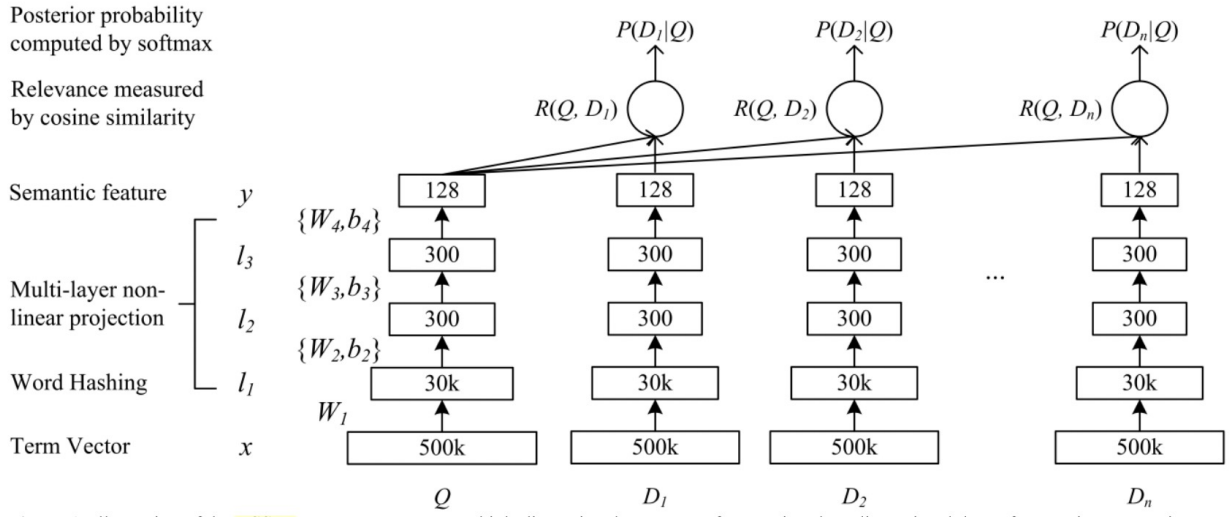
FM (Convolutional Click Prediction Model)

FM Model API

Factorization Machines

DSSM (Deep Structured Semantic Model)

DSSM Model API

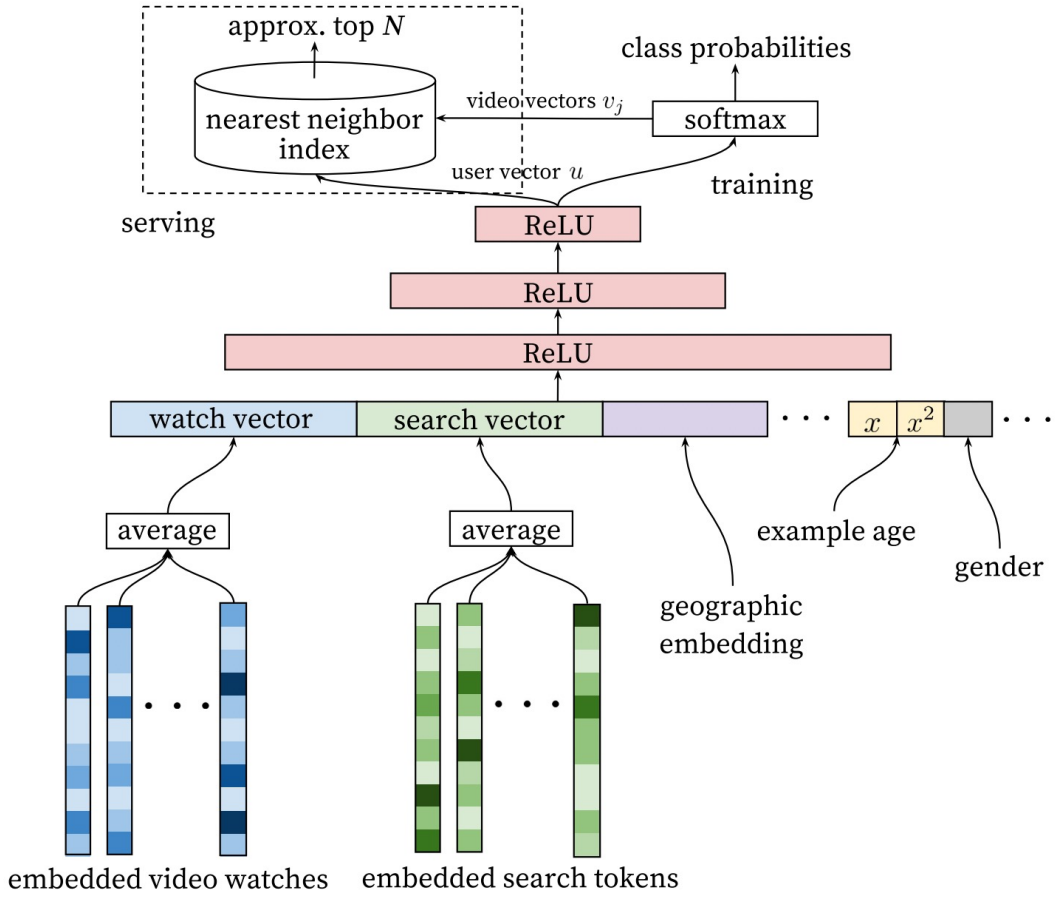


DSSM

Deep Structured Semantic Models for Web Search using Clickthrough Data

YoutubeDNN

YoutubeDNN Model API

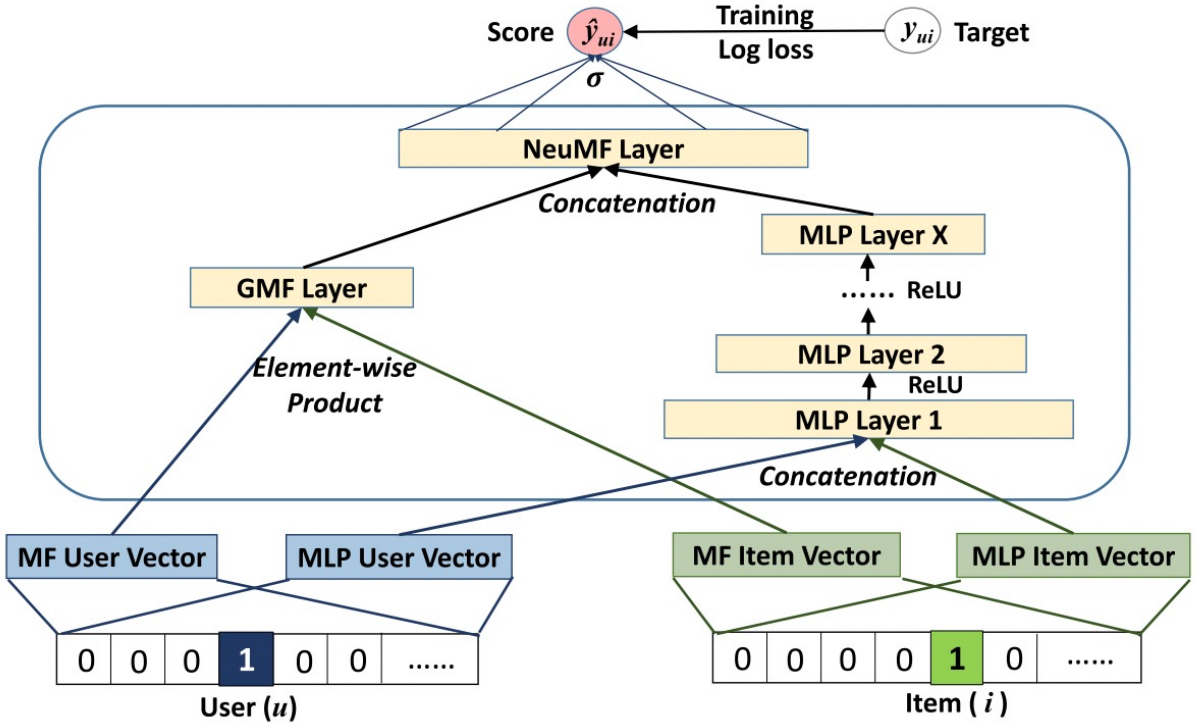


YoutubeDNN

Deep Neural Networks for YouTube Recommendations

NCF (Neural Collaborative Filtering)

NCF Model API



NCF

Neural Collaborative Filtering

SDM (Sequential Deep Matching Model)

SDM Model API

../pics/sdm.jpg

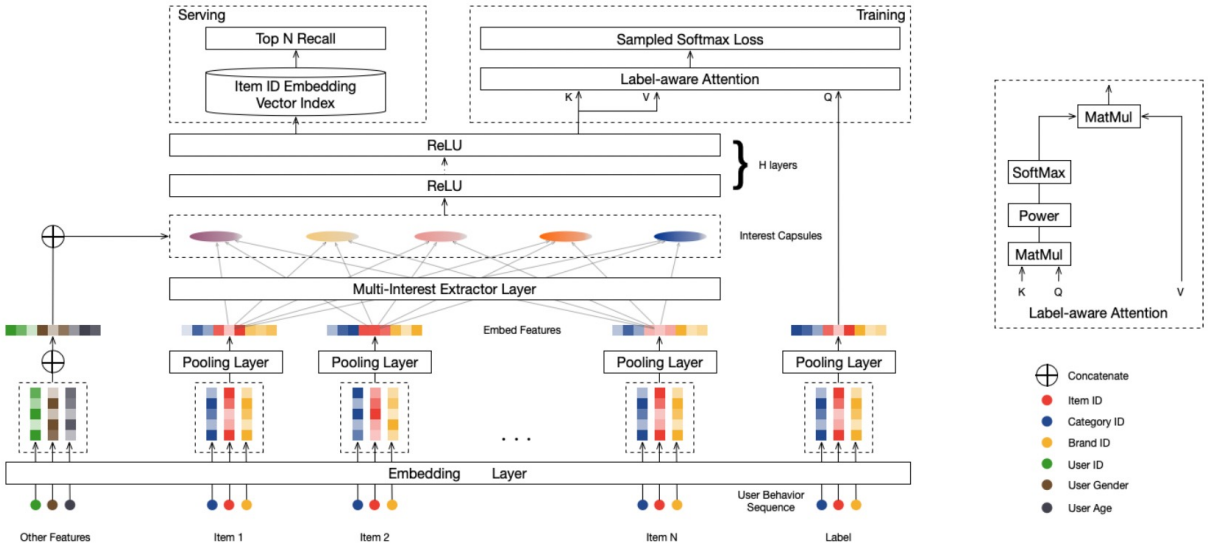
SDM

SDM example

SDM: Sequential Deep Matching Model for Online Large-scale Recommender System

MIND (Multi-Interest Network with Dynamic routing)

MIND Model API



MIND

Multi-interest network with dynamic routing for recommendation at Tmall

2.3 Examples

2.3.1 Run YoutubeDNN on MovieLen1M on Google colab

Open In Colab

2.3.2 YoutubeDNN/MIND with sampled softmax

The MovieLens data has been used for personalized tag recommendation,which contains 668, 953 tag applications of users on movies. Here is a small fraction of data include only sparse field.

	movie_id	user_id	gender	age	occupation	zip	rating
254181	2944	1545	M	25	20	20009	4
481546	2208	2962	M	35	3	94109	3
166949	3629	1062	M	50	19	59457	5
536371	569	3308	F	18	20	15701-1348	2
117094	2763	754	M	35	7	38024	4

This example shows how to use `YoutubeDNN` to solve a matching task. You can get the demo data movielens_sample.txt and run the following codes.

```
import pandas as pd
from deepctr.inputs import SparseFeat, VarLenSparseFeat
from preprocess import gen_data_set, gen_model_input
from sklearn.preprocessing import LabelEncoder
```

(continues on next page)

(continued from previous page)

```
from tensorflow.python.keras import backend as K
from tensorflow.python.keras.models import Model

from deepmatch.models import *
from deepmatch.utils import sampledsoftmaxloss

if __name__ == "__main__":

    data = pd.read_csvdata = pd.read_csv("./movielens_sample.txt")
    sparse_features = ["movie_id", "user_id",
                       "gender", "age", "occupation", "zip", ]
    SEQ_LEN = 50
    negsample = 0

    # 1.Label Encoding for sparse features,and process sequence features with `gen_
↪date_set` and `gen_model_input`

    features = ['user_id', 'movie_id', 'gender', 'age', 'occupation', 'zip']
    feature_max_idx = {}
    for feature in features:
        lbe = LabelEncoder()
        data[feature] = lbe.fit_transform(data[feature]) + 1
        feature_max_idx[feature] = data[feature].max() + 1

    user_profile = data[["user_id", "gender", "age", "occupation", "zip"]].drop_
↪duplicates('user_id')

    item_profile = data[["movie_id"]].drop_duplicates('movie_id')

    user_profile.set_index("user_id", inplace=True)

    user_item_list = data.groupby("user_id")['movie_id'].apply(list)

    train_set, test_set = gen_data_set(data, negsample)

    train_model_input, train_label = gen_model_input(train_set, user_profile, SEQ_LEN)
    test_model_input, test_label = gen_model_input(test_set, user_profile, SEQ_LEN)

    # 2.count #unique features for each sparse field and generate feature config for␣
↪sequence feature

    embedding_dim = 16

    user_feature_columns = [SparseFeat('user_id', feature_max_idx['user_id'],␣
↪embedding_dim),
                            SparseFeat("gender", feature_max_idx['gender'], embedding_
↪dim),
                            SparseFeat("age", feature_max_idx['age'], embedding_dim),
                            SparseFeat("occupation", feature_max_idx['occupation'],␣
↪embedding_dim),
                            SparseFeat("zip", feature_max_idx['zip'], embedding_dim),
                            VarLenSparseFeat(SparseFeat('hist_movie_id', feature_max_
↪idx['movie_id'], embedding_dim,
                                                        embedding_name="movie_id"),␣
↪SEQ_LEN, 'mean', 'hist_len'),
                            ]
```

(continues on next page)

(continued from previous page)

```
    item_feature_columns = [SparseFeat('movie_id', feature_max_idx['movie_id'],␣
↪embedding_dim)]

    # 3.Define Model and train

    K.set_learning_phase(True)

    model = YoutubeDNN(user_feature_columns, item_feature_columns, num_sampled=5,␣
↪user_dnn_hidden_units=(64, embedding_dim))
    # model = MIND(user_feature_columns,item_feature_columns,dynamic_k=False,p=1,k_
↪max=2,num_sampled=5,user_dnn_hidden_units=(64,embedding_dim),init_std=0.001)

    model.compile(optimizer="adam", loss=sampledsoftmaxloss)  # "binary_crossentropy")

    history = model.fit(train_model_input, train_label,  # train_label,
                        batch_size=256, epochs=1, verbose=1, validation_split=0.0, )

    # 4. Generate user features for testing and full item features for retrieval
    test_user_model_input = test_model_input
    all_item_model_input = {"movie_id": item_profile['movie_id'].values, "movie_idx":␣
↪item_profile['movie_id'].values}

    user_embedding_model = Model(inputs=model.user_input, outputs=model.user_
↪embedding)
    item_embedding_model = Model(inputs=model.item_input, outputs=model.item_
↪embedding)

    user_embs = user_embedding_model.predict(test_user_model_input, batch_size=2 **␣
↪12)
    # user_embs = user_embs[:, i, :]  # i in [0,k_max) if MIND
    item_embs = item_embedding_model.predict(all_item_model_input, batch_size=2 ** 12)

    print(user_embs.shape)
    print(item_embs.shape)

    # 5. [Optional] ANN search by faiss  and evaluate the result

    # test_true_label = {line[0]:[line[2]] for line in test_set}
    #
    # import numpy as np
    # import faiss
    # from tqdm import tqdm
    # from deepmatch.utils import recall_N
    #
    # index = faiss.IndexFlatIP(embedding_dim)
    # # faiss.normalize_L2(item_embs)
    # index.add(item_embs)
    # # faiss.normalize_L2(user_embs)
    # D, I = index.search(np.ascontiguousarray(user_embs), 50)
    # s = []
    # hit = 0
    # for i, uid in tqdm(enumerate(test_user_model_input['user_id'])):
    #     try:
    #         pred = [item_profile['movie_id'].values[x] for x in I[i]]
    #         filter_item = None
    #         recall_score = recall_N(test_true_label[uid], pred, N=50)
    #         s.append(recall_score)
```

(continues on next page)

(continued from previous page)

```
    #         if test_true_label[uid] in pred:
    #             hit += 1
    #     except:
    #         print(i)
    # print("recall", np.mean(s))
    # print("hr", hit / len(test_user_model_input['user_id']))
```

2.3.3 SDM with sampled softmax

The MovieLens data has been used for personalized tag recommendation,which contains 668, 953 tag applications of users on movies. Here is a small fraction of data include only sparse field.

	movie_id	user_id	gender	age	occupation	zip	rating
254181	2944	1545	M	25	20	20009	4
481546	2208	2962	M	35	3	94109	3
166949	3629	1062	M	50	19	59457	5
536371	569	3308	F	18	20	15701-1348	2
117094	2763	754	M	35	7	38024	4

This example shows how to use `SDM` to solve a matching task. You can get the demo data movielens_sample.txt and run the following codes.

```
import pandas as pd
from deepctr.inputs import SparseFeat, VarLenSparseFeat
from preprocess import gen_data_set_sdm, gen_model_input_sdm
from sklearn.preprocessing import LabelEncoder
from tensorflow.python.keras import backend as K
from tensorflow.python.keras import optimizers
from tensorflow.python.keras.models import Model

from deepmatch.models import SDM
from deepmatch.utils import sampledsoftmaxloss

if __name__ == "__main__":
    data = pd.read_csvdata = pd.read_csv("./movielens_sample.txt")
    sparse_features = ["movie_id", "user_id",
                       "gender", "age", "occupation", "zip", "genres"]
    SEQ_LEN_short = 5
    SEQ_LEN_prefer = 50

    # 1.Label Encoding for sparse features,and process sequence features with `gen_
↪date_set` and `gen_model_input`

    features = ['user_id', 'movie_id', 'gender', 'age', 'occupation', 'zip', 'genres']
    feature_max_idx = {}
    for feature in features:
        lbe = LabelEncoder()
        data[feature] = lbe.fit_transform(data[feature]) + 1
        feature_max_idx[feature] = data[feature].max() + 1
```

(continues on next page)

(continued from previous page)

```
    user_profile = data[["user_id", "gender", "age", "occupation", "zip", "genres"]].
↪drop_duplicates('user_id')

    item_profile = data[["movie_id"]].drop_duplicates('movie_id')

    user_profile.set_index("user_id", inplace=True)
    #
    # user_item_list = data.groupby("user_id")['movie_id'].apply(list)

    train_set, test_set = gen_data_set_sdm(data, seq_short_len=SEQ_LEN_short, seq_
↪prefer_len=SEQ_LEN_prefer)

    train_model_input, train_label = gen_model_input_sdm(train_set, user_profile, SEQ_
↪LEN_short, SEQ_LEN_prefer)
    test_model_input, test_label = gen_model_input_sdm(test_set, user_profile, SEQ_
↪LEN_short, SEQ_LEN_prefer)

    # 2.count #unique features for each sparse field and generate feature config for␣
↪sequence feature

    embedding_dim = 32
    # for sdm,we must provide `VarLenSparseFeat` with name "prefer_xxx" and "short_xxx
↪" and their length
    user_feature_columns = [SparseFeat('user_id', feature_max_idx['user_id'], 16),
                            SparseFeat("gender", feature_max_idx['gender'], 16),
                            SparseFeat("age", feature_max_idx['age'], 16),
                            SparseFeat("occupation", feature_max_idx['occupation'],␣
↪16),
                            SparseFeat("zip", feature_max_idx['zip'], 16),
                            VarLenSparseFeat(SparseFeat('short_movie_id', feature_max_
↪idx['movie_id'], embedding_dim,
                                                        embedding_name="movie_id"),␣
↪SEQ_LEN_short, 'mean',
                                             'short_sess_length'),
                            VarLenSparseFeat(SparseFeat('prefer_movie_id', feature_
↪max_idx['movie_id'], embedding_dim,
                                                        embedding_name="movie_id"),␣
↪SEQ_LEN_prefer, 'mean',
                                             'prefer_sess_length'),
                            VarLenSparseFeat(SparseFeat('short_genres', feature_max_
↪idx['genres'], embedding_dim,
                                                        embedding_name="genres"), SEQ_
↪LEN_short, 'mean',
                                             'short_sess_length'),
                            VarLenSparseFeat(SparseFeat('prefer_genres', feature_max_
↪idx['genres'], embedding_dim,
                                                        embedding_name="genres"), SEQ_
↪LEN_prefer, 'mean',
                                             'prefer_sess_length'),
                            ]

    item_feature_columns = [SparseFeat('movie_id', feature_max_idx['movie_id'],␣
↪embedding_dim)]

    K.set_learning_phase(True)
```

(continues on next page)

(continued from previous page)

```
    import tensorflow as tf

    if tf.__version__ >= '2.0.0':
        tf.compat.v1.disable_eager_execution()

    # units must be equal to item embedding dim!
    model = SDM(user_feature_columns, item_feature_columns, history_feature_list=[
↪'movie_id', 'genres'],
                units=embedding_dim, num_sampled=100, )

    optimizer = optimizers.Adam(lr=0.001, clipnorm=5.0)

    model.compile(optimizer=optimizer, loss=sampledsoftmaxloss)  # "binary_
↪crossentropy")

    history = model.fit(train_model_input, train_label,  # train_label,
                        batch_size=512, epochs=1, verbose=1, validation_split=0.0, )
    # model.save_weights('SDM_weights.h5')

    K.set_learning_phase(False)
    # 4. Generate user features for testing and full item features for retrieval
    test_user_model_input = test_model_input
    all_item_model_input = {"movie_id": item_profile['movie_id'].values, }

    user_embedding_model = Model(inputs=model.user_input, outputs=model.user_
↪embedding)
    item_embedding_model = Model(inputs=model.item_input, outputs=model.item_
↪embedding)

    user_embs = user_embedding_model.predict(test_user_model_input, batch_size=2 **␣
↪12)
    # user_embs = user_embs[:, i, :]  # i in [0,k_max) if MIND
    item_embs = item_embedding_model.predict(all_item_model_input, batch_size=2 ** 12)

    print(user_embs.shape)
    print(item_embs.shape)

    # 5. [Optional] ANN search by faiss  and evaluate the result

    # test_true_label = {line[0]: [line[3]] for line in test_set}
    #
    # import numpy as np
    # import faiss
    # from tqdm import tqdm
    # from deepmatch.utils import recall_N
    #
    # index = faiss.IndexFlatIP(embedding_dim)
    # # faiss.normalize_L2(item_embs)
    # index.add(item_embs)
    # # faiss.normalize_L2(user_embs)
    # D, I = index.search(np.ascontiguousarray(user_embs), 50)
    # s = []
    # hit = 0
    # for i, uid in tqdm(enumerate(test_user_model_input['user_id'])):
    #     try:
    #         pred = [item_profile['movie_id'].values[x] for x in I[i]]
    #         filter_item = None
```

(continues on next page)

(continued from previous page)

```
    #         recall_score = recall_N(test_true_label[uid], pred, N=50)
    #         s.append(recall_score)
    #         if test_true_label[uid] in pred:
    #             hit += 1
    #     except:
    #         print(i)
    # print("")
    # print("recall", np.mean(s))
    # print("hit rate", hit / len(test_user_model_input['user_id']))
```

2.3.4 DSSM with negative sampling

The MovieLens data has been used for personalized tag recommendation,which contains 668, 953 tag applications of users on movies. Here is a small fraction of data include only sparse field.

	movie_id	user_id	gender	age	occupation	zip	rating
254181	2944	1545	M	25	20	20009	4
481546	2208	2962	M	35	3	94109	3
166949	3629	1062	M	50	19	59457	5
536371	569	3308	F	18	20	15701-1348	2
117094	2763	754	M	35	7	38024	4

This example shows how to use `DSSM` to solve a matching task. You can get the demo data movielens_sample.txt and run the following codes.

```
import pandas as pd
from deepctr.inputs import SparseFeat, VarLenSparseFeat
from preprocess import gen_data_set, gen_model_input
from sklearn.preprocessing import LabelEncoder
from tensorflow.python.keras.models import Model

from deepmatch.models import *

if __name__ == "__main__":

    data = pd.read_csvdata = pd.read_csv("./movielens_sample.txt")
    sparse_features = ["movie_id", "user_id",
                       "gender", "age", "occupation", "zip", ]
    SEQ_LEN = 50
    negsample = 3

    # 1.Label Encoding for sparse features,and process sequence features with `gen_
↪date_set` and `gen_model_input`

    features = ['user_id', 'movie_id', 'gender', 'age', 'occupation', 'zip']
    feature_max_idx = {}
    for feature in features:
        lbe = LabelEncoder()
        data[feature] = lbe.fit_transform(data[feature]) + 1
```

(continues on next page)

(continued from previous page)

```
        feature_max_idx[feature] = data[feature].max() + 1

    user_profile = data[["user_id", "gender", "age", "occupation", "zip"]].drop_
↪duplicates('user_id')

    item_profile = data[["movie_id"]].drop_duplicates('movie_id')

    user_profile.set_index("user_id", inplace=True)

    user_item_list = data.groupby("user_id")['movie_id'].apply(list)

    train_set, test_set = gen_data_set(data, negsample)

    train_model_input, train_label = gen_model_input(train_set, user_profile, SEQ_LEN)
    test_model_input, test_label = gen_model_input(test_set, user_profile, SEQ_LEN)

    # 2.count #unique features for each sparse field and generate feature config for␣
↪sequence feature

    embedding_dim = 16

    user_feature_columns = [SparseFeat('user_id', feature_max_idx['user_id'],␣
↪embedding_dim),
                            SparseFeat("gender", feature_max_idx['gender'], embedding_
↪dim),
                            SparseFeat("age", feature_max_idx['age'], embedding_dim),
                            SparseFeat("occupation", feature_max_idx['occupation'],␣
↪embedding_dim),
                            SparseFeat("zip", feature_max_idx['zip'], embedding_dim),
                            VarLenSparseFeat(SparseFeat('hist_movie_id', feature_max_
↪idx['movie_id'], embedding_dim,
                                                        embedding_name="movie_id"),␣
↪SEQ_LEN, 'mean', 'hist_len'),
                            ]

    item_feature_columns = [SparseFeat('movie_id', feature_max_idx['movie_id'],␣
↪embedding_dim)]

    # 3.Define Model and train

    model = DSSM(user_feature_columns, item_feature_columns)  # FM(user_feature_
↪columns,item_feature_columns)

    model.compile(optimizer='adagrad', loss="binary_crossentropy")

    history = model.fit(train_model_input, train_label,  # train_label,
                        batch_size=256, epochs=1, verbose=1, validation_split=0.0, )

    # 4. Generate user features for testing and full item features for retrieval
    test_user_model_input = test_model_input
    all_item_model_input = {"movie_id": item_profile['movie_id'].values,}

    user_embedding_model = Model(inputs=model.user_input, outputs=model.user_
↪embedding)
    item_embedding_model = Model(inputs=model.item_input, outputs=model.item_
↪embedding)
```

(continues on next page)

(continued from previous page)

```
    user_embs = user_embedding_model.predict(test_user_model_input, batch_size=2 **␣
↪12)
    item_embs = item_embedding_model.predict(all_item_model_input, batch_size=2 ** 12)

    print(user_embs.shape)
    print(item_embs.shape)

    # 5. [Optional] ANN search by faiss  and evaluate the result

    # test_true_label = {line[0]:[line[2]] for line in test_set}
    #
    # import numpy as np
    # import faiss
    # from tqdm import tqdm
    # from deepmatch.utils import recall_N
    #
    # index = faiss.IndexFlatIP(embedding_dim)
    # # faiss.normalize_L2(item_embs)
    # index.add(item_embs)
    # # faiss.normalize_L2(user_embs)
    # D, I = index.search(user_embs, 50)
    # s = []
    # hit = 0
    # for i, uid in tqdm(enumerate(test_user_model_input['user_id'])):
    #     try:
    #         pred = [item_profile['movie_id'].values[x] for x in I[i]]
    #         filter_item = None
    #         recall_score = recall_N(test_true_label[uid], pred, N=50)
    #         s.append(recall_score)
    #         if test_true_label[uid] in pred:
    #             hit += 1
    #     except:
    #         print(i)
    # print("recall", np.mean(s))
    # print("hr", hit / len(test_user_model_input['user_id']))
```

2.4 FAQ

2.4.1 1. Save or load weights/models

To save/load weights,you can write codes just like any other keras models.

```
model = YoutubeDNN()
model.save_weights('YoutubeDNN_w.h5')
model.load_weights('YoutubeDNN_w.h5')
```

To save/load models,just a little different.

```
from tensorflow.python.keras.models import  save_model,load_model
model = DeepFM()
save_model(model, 'YoutubeDNN.h5')# save_model, same as before
```

(continues on next page)

(continued from previous page)

```
from deepmatch.layers import custom_objects
model = load_model('YoutubeDNN.h5',custom_objects)# load_model,just add a parameter
```

2.4.2 2. Set learning rate and use earlystopping

You can use any models in DeepCTR like a keras model object. Here is a example of how to set learning rate and earlystopping:

```
import deepmatch
from tensorflow.python.keras.optimizers import Adam,Adagrad
from tensorflow.python.keras.callbacks import EarlyStopping

model = deepmatch.models.FM(user_feature_columns,item_feature_columns)
model.compile(Adagrad(0.01),'binary_crossentropy',metrics=['binary_crossentropy'])

es = EarlyStopping(monitor='val_binary_crossentropy')
history = model.fit(model_input, data[target].values,batch_size=256, epochs=10,␣
↪verbose=2, validation_split=0.2,callbacks=[es] )
```

2.5 History

- 05/17/2020 : v0.1.3 released.Add `SDM` model .
- 04/10/2020 : v0.1.2 released.Support saving and loading model.
- 04/06/2020 : DeepMatch first version is released on PyPi

2.6 DeepMatch Models API

2.6.1 Methods

compile

```
compile(optimizer, loss=None, metrics=None, loss_weights=None, sample_weight_
↪mode=None, weighted_metrics=None, target_tensors=None)
```

Configures the model for training.

Arguments

- **optimizer**: String (name of optimizer) or optimizer instance. See optimizers.
- **loss**: String (name of objective function) or objective function. See losses. If the model has multiple outputs, you can use a different loss on each output by passing a dictionary or a list of losses. The loss value that will be minimized by the model will then be the sum of all individual losses.
- **metrics**: List of metrics to be evaluated by the model during training and testing. Typically you will use `metrics=['accuracy']`. To specify different metrics for different outputs of a multi-output model, you could also pass a dictionary, such as `metrics={'output_a': 'accuracy'}`.

- **loss_weights**: Optional list or dictionary specifying scalar coefficients (Python floats) to weight the loss contributions of different model outputs. The loss value that will be minimized by the model will then be the weighted sum of all individual losses, weighted by the `loss_weights` coefficients. If a list, it is expected to have a 1:1 mapping to the model's outputs. If a tensor, it is expected to map output names (strings) to scalar coefficients.
- **sample_weight_mode**: If you need to do timestep-wise sample weighting (2D weights), set this to `"temporal"`. `None` defaults to sample-wise weights (1D). If the model has multiple outputs, you can use a different `sample_weight_mode` on each output by passing a dictionary or a list of modes.
- **weighted_metrics**: List of metrics to be evaluated and weighted by sample_weight or class_weight during training and testing.
- **target_tensors**: By default, Keras will create placeholders for the model's target, which will be fed with the target data during training. If instead you would like to use your own target tensors (in turn, Keras will not expect external Numpy data for these targets at training time), you can specify them via the `target_tensors` argument. It can be a single tensor (for a single-output model), a list of tensors, or a dict mapping output names to target tensors.

Raises

- **ValueError**: In case of invalid arguments for `optimizer`, `loss`, `metrics` or `sample_weight_mode`.

fit

```
fit(x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None, validation_
↪split=0.0, validation_data=None, shuffle=True, class_weight=None, sample_
↪weight=None, initial_epoch=0, steps_per_epoch=None, validation_steps=None,␣
↪validation_freq=1)
```

Trains the model for a given number of epochs (iterations on a dataset).

Arguments

- **x**: Numpy array of training data (if the model has a single input), or list of Numpy arrays (if the model has multiple inputs). If input layers in the model are named, you can also pass a dictionary mapping input names to Numpy arrays. `x` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **y**: Numpy array of target (label) data (if the model has a single output), or list of Numpy arrays (if the model has multiple outputs). If output layers in the model are named, you can also pass a dictionary mapping output names to Numpy arrays. `y` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **batch_size**: Integer or `None`. Number of samples per gradient update. If unspecified, `batch_size` will default to 32.
- **epochs**: Integer. Number of epochs to train the model. An epoch is an iteration over the entire `x` and `y` data provided. Note that in conjunction with `initial_epoch`, `epochs` is to be understood as "final epoch". The model is not trained for a number of iterations given by `epochs`, but merely until the epoch of index `epochs` is reached.
- **verbose**: Integer. 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch.
- **callbacks**: List of `tf.keras.callbacks.Callback` instances. List of callbacks to apply during training and validation (if). See callbacks.
- **validation_split**: Float between 0 and 1. Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch. The validation data is selected from the last samples in the `x` and `y` data provided, before shuffling.

- **validation_data**: tuple `(x_val, y_val)` or tuple `(x_val, y_val, val_sample_weights)` on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data. `validation_data` will override `validation_split`.
- **shuffle**: Boolean (whether to shuffle the training data before each epoch) or str (for 'batch'). 'batch' is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks. Has no effect when `steps_per_epoch` is not `None`.
- **class_weight**: Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to "pay more attention" to samples from an under-represented class.
- **sample_weight**: Optional Numpy array of weights for the training samples, used for weighting the loss function (during training only). You can either pass a flat (1D) Numpy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape `(samples, sequence_length)`, to apply a different weight to every timestep of every sample. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`.
- **initial_epoch**: Integer. Epoch at which to start training (useful for resuming a previous training run).
- **steps_per_epoch**: Integer or `None`. Total number of steps (batches of samples) before declaring one epoch finished and starting the next epoch. When training with input tensors such as TensorFlow data tensors, the default `None` is equal to the number of samples in your dataset divided by the batch size, or 1 if that cannot be determined. validation_steps: Only relevant if `steps_per_epoch` is specified. Total number of steps (batches of samples) to validate before stopping.
- **validation_freq**: Only relevant if validation data is provided. Integer or list/tuple/set. If an integer, specifies how many training epochs to run before a new validation run is performed, e.g. `validation_freq=2` runs validation every 2 epochs. If a list, tuple, or set, specifies the epochs on which to run validation, e.g. `validation_freq=[1, 2, 10]` runs validation at the end of the 1st, 2nd, and 10th epochs.

Returns

- A `History` object. Its `History.history` attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

Raises

- **RuntimeError**: If the model was never compiled. ValueError: In case of mismatch between the provided input data and what the model expects.

evaluate

```
evaluate(x=None, y=None, batch_size=None, verbose=1, sample_weight=None, steps=None,␣
↪callbacks=None)
```

Returns the loss value & metrics values for the model in test mode. Computation is done in batches.

Arguments

- **x**: Numpy array of test data (if the model has a single input), or list of Numpy arrays (if the model has multiple inputs). If input layers in the model are named, you can also pass a dictionary mapping input names to Numpy arrays. `x` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **y**: Numpy array of target (label) data (if the model has a single output), or list of Numpy arrays (if the model has multiple outputs). If output layers in the model are named, you can also pass a dictionary mapping output names to Numpy arrays. `y` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).

- **batch_size**: Integer or `None`. Number of samples per evaluation step. If unspecified, `batch_size` will default to 32.
- **verbose**: 0 or 1. Verbosity mode. 0 = silent, 1 = progress bar.
- **sample_weight**: Optional Numpy array of weights for the test samples, used for weighting the loss function. You can either pass a flat (1D) Numpy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape `(samples, sequence_length)`, to apply a different weight to every timestep of every sample. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`.
- **steps**: Integer or `None`. Total number of steps (batches of samples) before declaring the evaluation round finished. Ignored with the default value of `None`.
- **callbacks**: List of `tf.keras.callbacks.Callback` instances. List of callbacks to apply during evaluation. See callbacks.

Returns

- Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

predict

```
predict(x, batch_size=None, verbose=0, steps=None, callbacks=None)
```

Generates output predictions for the input samples.

Computation is done in batches.

Arguments

- **x**: The input data, as a Numpy array (or list of Numpy arrays if the model has multiple inputs). batch_size: Integer. If unspecified, it will default to 32.
- **verbose**: Verbosity mode, 0 or 1.
- **steps**: Total number of steps (batches of samples) before declaring the prediction round finished. Ignored with the default value of None.
- **callbacks**: List of `tf.keras.callbacks.Callback` instances. List of callbacks to apply during prediction. See callbacks.

Returns

- Numpy array(s) of predictions.

Raises

- **ValueError**: In case of mismatch between the provided input data and the model's expectations, or in case a stateful model receives a number of samples that is not a multiple of the batch size.

train_on_batch

```
train_on_batch(x, y, sample_weight=None, class_weight=None)
```

Runs a single gradient update on a single batch of data.

Arguments

- **x**: Numpy array of training data, or list of Numpy arrays if the model has multiple inputs. If all inputs in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.
- **y**: Numpy array of target data, or list of Numpy arrays if the model has multiple outputs. If all outputs in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.
- **sample_weight**: Optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample. In this case you should make sure to specify sample_weight_mode="temporal" in compile().
- **class_weight**: Optional dictionary mapping class indices (integers) to a weight (float) to apply to the model's loss for the samples from this class during training. This can be useful to tell the model to "pay more attention" to samples from an under-represented class.

Returns

- Scalar training loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

test_on_batch

```
test_on_batch(x, y, sample_weight=None)
```

Test the model on a single batch of samples.

Arguments

- **x**: Numpy array of test data, or list of Numpy arrays if the model has multiple inputs. If all inputs in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.
- **y**: Numpy array of target data, or list of Numpy arrays if the model has multiple outputs. If all outputs in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.
- **sample_weight**: Optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`.

Returns

- Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

predict_on_batch

```
predict_on_batch(x)
```

Returns predictions for a single batch of samples.

Arguments

- **x**: Input samples, as a Numpy array.

Returns

- Numpy array(s) of predictions.

fit_generator

```
fit_generator(generator, steps_per_epoch=None, epochs=1, verbose=1, callbacks=None,␣
↪validation_data=None, validation_steps=None, validation_freq=1, class_weight=None,␣
↪max_queue_size=10, workers=1, use_multiprocessing=False, shuffle=True, initial_
↪epoch=0)
```

Trains the model on data generated batch-by-batch by a Python generator (or an instance of `Sequence`). The generator is run in parallel to the model, for efficiency. For instance, this allows you to do real-time data augmentation on images on CPU in parallel to training your model on GPU. The use of `tf.keras.utils.Sequence` guarantees the ordering and guarantees the single use of every input per epoch when using `use_multiprocessing=True`.

Arguments

- **generator**: A generator or an instance of `Sequence` (`tf.keras.utils.Sequence`) object in order to avoid duplicate data when using multiprocessing. The output of the generator must be either

  a tuple `(inputs, targets)` or a tuple `(inputs, targets, sample_weights)`.This tuple (a single output of the generator) makes a single batch. Therefore, all arrays in this tuple must have the same length (equal to the size of this batch). Different batches may have different sizes. For example, the last batch of the epoch is commonly smaller than the others, if the size of the dataset is not divisible by the batch size. The generator is expected to loop over its data indefinitely. An epoch finishes when `steps_per_epoch` batches have been seen by the model.

- **steps_per_epoch**: Integer. Total number of steps (batches of samples) to yield from `generator` before declaring one epoch finished and starting the next epoch. It should typically be equal to `ceil(num_samples / batch_size)` Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.

- **epochs**: Integer. Number of epochs to train the model. An epoch is an iteration over the entire data provided, as defined by `steps_per_epoch`. Note that in conjunction with `initial_epoch`, `epochs` is to be understood as "final epoch". The model is not trained for a number of iterations given by `epochs`, but merely until the epoch of index `epochs` is reached.

- **verbose**: Integer. 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch.

- **callbacks**: List of `tf.keras.callbacks.Callback` instances. List of callbacks to apply during training. See callbacks.

- **validation_data**: This can be either a generator or a `Sequence` object for the validation data tuple `(x_val, y_val)` tuple `(x_val, y_val, val_sample_weights)` on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data.

- **validation_steps**: Only relevant if `validation_data` is a generator. Total number of steps (batches of samples) to yield from `validation_data` generator before stopping at the end of every epoch. It should typically be equal to the number of samples of your validation dataset divided by the batch size. Optional for `Sequence`: if unspecified, will use the `len(validation_data)` as a number of steps.

- **validation_freq**: Only relevant if validation data is provided. Integer or `collections.Container` instance (e.g. list, tuple, etc.). If an integer, specifies how many training epochs to run before a new validation run is performed, e.g. `validation_freq=2` runs validation every 2 epochs. If a Container, specifies the epochs on which to run validation, e.g. `validation_freq=[1, 2, 10]` runs validation at the end of the 1st, 2nd, and 10th epochs.

- **class_weight**: Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to "pay more attention" to samples from an under-represented class.

- **max_queue_size**: Integer. Maximum size for the generator queue. If unspecified, `max_queue_size` will default to 10.

- **workers**: Integer. Maximum number of processes to spin up when using process-based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: Boolean. If `True`, use process-based threading. If unspecified, `use_multiprocessing` will default to `False`. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can't be passed easily to children processes.
- **shuffle**: Boolean. Whether to shuffle the order of the batches at the beginning of each epoch. Only used with instances of `Sequence` (`tf.keras.utils.Sequence`). Has no effect when `steps_per_epoch` is not `None`. initial_epoch: Integer. Epoch at which to start training (useful for resuming a previous training run).

Returns

- A `History` object. Its `History.history` attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

Raises

- **ValueError**: In case the generator yields data in an invalid format.

Example

```
def generate_arrays_from_file(path):
    while True:
        with open(path) as f:
            for line in f:
                # create numpy arrays of input data
                # and labels, from each line in the file
                x1, x2, y = process_line(line)
                yield ({'input_1': x1, 'input_2': x2}, {'output': y})

model.fit_generator(generate_arrays_from_file('/my_file.txt'),
                    steps_per_epoch=10000, epochs=10)
```

evaluate_generator

```
evaluate_generator(generator, steps=None, callbacks=None, max_queue_size=10,␣
↪workers=1, use_multiprocessing=False, verbose=0)
```

Evaluates the model on a data generator. The generator should return the same kind of data as accepted by `test_on_batch`.

Arguments

- **generator**: Generator yielding tuples (inputs, targets) or (inputs, targets, sample_weights) or an instance of Sequence (tf.keras.utils.Sequence) object in order to avoid duplicate data when using multiprocessing.
- **steps**: Total number of steps (batches of samples) to yield from `generator` before stopping. Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.
- **callbacks**: List of `tf.keras.callbacks.Callback` instances. List of callbacks to apply during training. See callbacks.
- **max_queue_size**: maximum size for the generator queue
- **workers**: Integer. Maximum number of processes to spin up when using process based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.

- **use_multiprocessing**: if True, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass non picklable arguments to the generator as they can't be passed easily to children processes.
- **verbose**: verbosity mode, 0 or 1.

Returns

- Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

Raises

- **ValueError**: In case the generator yields data in an invalid format.

predict_generator

```
predict_generator(generator, steps=None, callbacks=None, max_queue_size=10, workers=1,
↪ use_multiprocessing=False, verbose=0)
```

Generates predictions for the input samples from a data generator. The generator should return the same kind of data as accepted by `predict_on_batch`.

Arguments

- **generator**: Generator yielding batches of input samples or an instance of Sequence (`tf.keras.utils.Sequence`) object in order to avoid duplicate data when using multiprocessing.
- **steps**: Total number of steps (batches of samples) to yield from `generator` before stopping. Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.
- **callbacks**: List of `tf.keras.callbacks.Callback` instances. List of callbacks to apply during training. See callbacks.
- **max_queue_size**: Maximum size for the generator queue.
- **workers**: Integer. Maximum number of processes to spin up when using process based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: If `True`, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass non picklable arguments to the generator as they can't be passed easily to children processes.
- **verbose**: verbosity mode, 0 or 1.

Returns

- Numpy array(s) of predictions.

Raises

- **ValueError**: In case the generator yields data in an invalid format.

get_layer

```
get_layer(name=None, index=None)
```

Retrieves a layer based on either its name (unique) or index. If `name` and `index` are both provided, `index` will take precedence. Indices are based on order of horizontal graph traversal (bottom-up).

Arguments

- **name**: String, name of layer.
- **index**: Integer, index of layer.

Returns

- A layer instance.

Raises

- **ValueError**: In case of invalid layer name or index.

2.6.2 deepmatch.models.fm module

2.6.3 deepmatch.models.dssm module

2.6.4 deepmatch.models.youtubednn module

2.6.5 deepmatch.models.ncf module

2.6.6 deepmatch.models.sdm module

2.6.7 deepmatch.models.mind module

CHAPTER 3

Indices and tables

- genindex
- modindex
- search