MACHINE LEARNING FOR SOCIAL MEDIA SITE FOR VISUAL PROGRAMMING

Project Progress

Information Technology Capstone Project

COMP5703/5707/5708

Group Members

1. Wenjing Deng (470349281)
2. Hefei Chen (470488188)
3. Kaijia Chen (480218353)
4. Wentao Wu (480416845)
5. Zaiwo Liu (490191684)
6. Zhou Sha (480322650

TABLE OF CONTENTS

1. PROGRESS & ACHIEVEMENTS

We have three sub-teams worked on three papers. Until now, we have finished three detailed instruction documentations of the code in hierarchy way on Google Colab. We firstly reviewed the paper and tried to have a fully understanding of it. Secondly, we tried to implement the relative code on our local computer. Thirdly, we deployed the code to Google Colab and made our code hierarchical way and comment the code in the corresponding area. Finally, the code ran smoothly and got the results.

1.1 RaCT

1.1.1 Paper introduction

The recommendation system is an important way when it comes to improving the user's network experience. Collaborative filtering, which uses similar users and items patterns to predict user preferences, is a widely used technology in recommendation systems. Because it conforms to the learning paradigm of latent variable models (LVMs), its effectiveness and simplicity make it the dominant method of recommendation system. However, traditional LVMs use linear mapping with limited modeling capability, which may produce suboptimal performance for large datasets. Therefore, the use of deep neural networks (DNN) is a better alternative. There are three deep learning models for collaborative filtering which are variational autoencoders (VAEs), collaborative denoising autoencoder (CDAE) and neural collaborative filtering (NCF). Among them, VAEs alleviates the model size grows linearly problem via amortized inference and integrates the flexible representation from deep neural network into the latent variable model, thus mitigating the limitations of traditional LVMs. Usually, VAEs are trained through maximizing the likelihood (MLE) of users interacting with ground-truth items, but this does not directly maximize recommended quality metrics that people care about, such as top-N projects' ranking. This paper presents a novel actor-critic algorithm for ranking-based training which calls Ranking-Critical Training (RaCT). Here is the schematic diagram:

Compared with the traditional learning-to-ranking method, which needs to re-run the optimization procedure for new list, RaCT amortizes the scoring process through the neural network, and can directly provide the ranking score for the new lists.

1.1.2 Colab Screenshots

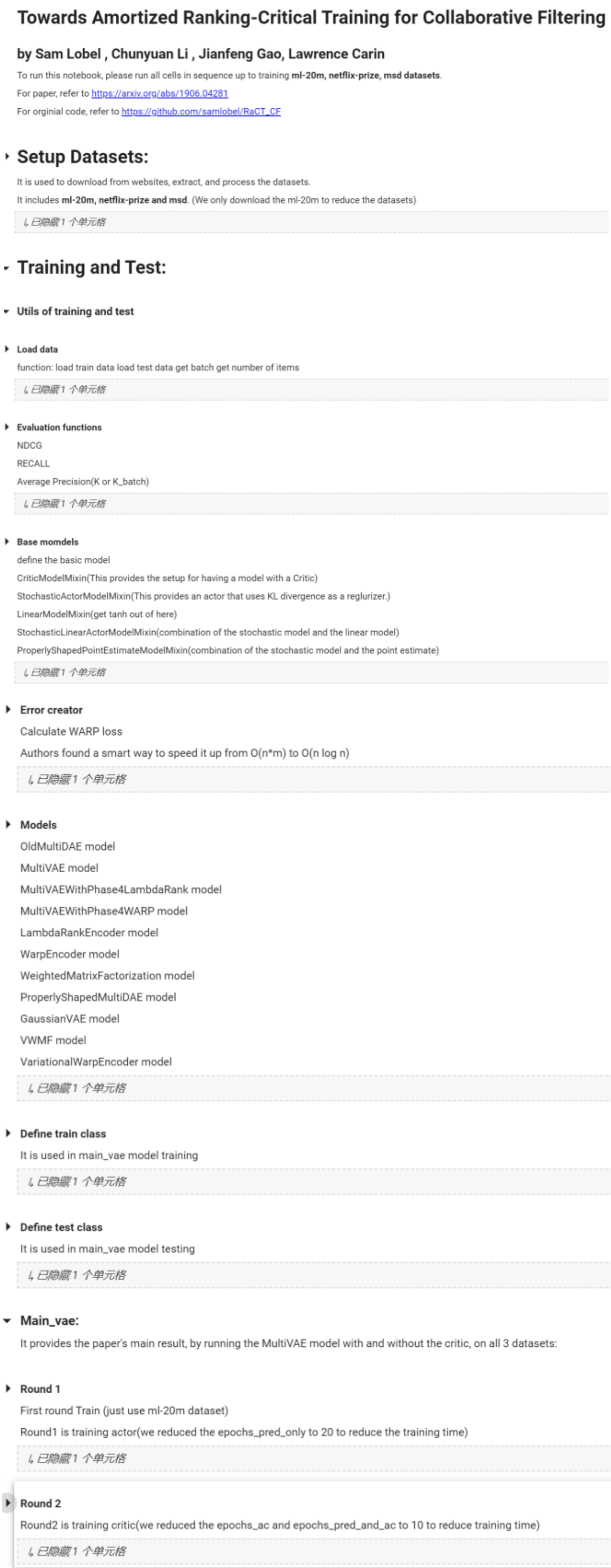


Figure 1: Overview of Colab

```
class StochasticActorModelMixin:
    """This provides an actor that uses KL divergence as a reglurizer. You should have to pass it a
    kl_loss_scaler I think. We can call it regularization_scaler or something."""
    biases_initializer = tf.truncated_normal_initializer(stddev=0.001)
    def forward_pass_actor(self):
        with tf.variable_scope("NormalizeNetworkInput"):
            self._create_normalized_network_input()
        with tf.variable_scope("InferenceGraph"):
            self._q_graph()
        with tf.variable_scope("SampledLatentVector"):
            self._create_sampled_latent_vector()
        with tf.variable_scope("DecoderGraph"):
            self._p_graph()

    def _create_normalized_network_input(self):
        normalized_network_input = tf.nn.l2_normalize(self.batch_of_users, 1)
        normalized_network_input = (normalized_network_input * self.mask) / self.keep_prob_ph  #to prevent the overfitting issue
        self.normalized_network_input = normalized_network_input
    def _q_graph(self):

        h = self.normalized_network_input
        h = slim.fully_connected(
            h, 600, activation_fn=tf.nn.tanh, biases_initializer=self.biases_initializer)

        mean_latent = slim.fully_connected(
            h, 200, activation_fn=None, biases_initializer=self.biases_initializer)
        log_variance_out = slim.fully_connected(
            h, 200, activation_fn=None, biases_initializer=self.biases_initializer)
        std_latent = tf.exp(0.5 * log_variance_out)

        self.log_variance_latent = log_variance_out
        self.mean_latent = mean_latent
        self.std_latent = std_latent

    def _create_sampled_latent_vector(self):
        """Samples from a distribution with mean/std calculated by the network."""
        epsilon = tf.random_normal(tf.shape(self.std_latent))

        sampled_latent_vector = self.mean_latent + self.stddev_effect_on_latent_dim_scaler *\
            epsilon * self.std_latent

        self.sampled_latent_vector = sampled_latent_vector

    def _p_graph(self):
        h = self.sampled_latent_vector
        h = slim.fully_connected(
            h, 600, activation_fn=tf.nn.tanh, biases_initializer=self.biases_initializer)
        h = slim.fully_connected(
            h, self.input_dim, activation_fn=None, biases_initializer=self.biases_initializer)
        self.prediction = h

    def _build_actor_reg(self):
        self.actor_regularization_loss = self.KL = tf.reduce_mean(
            tf.reduce_sum(
                0.5 * (-self.log_variance_latent + tf.exp(self.log_variance_latent) +
                       self.mean_latent**2 - 1),
                axis=1))
```

Figure 2: class of Stochastic Actor Model

(This class is the actor model in RaCT algorithm which can predict the given user's interaction history as the state.)

```
class CriticModelMixin:
    """This provides the setup for having a model with a Critic
    What's the signature? Pretty much, I want to replace phase 3 with minimizing WARP.
    There are some of these that no longer make sense, but that's okay. For example,
    logging expected-ndcg or whatever is no longer a sensible metric.
    ac_train_op
    critic_train_op
    critic_error (maybe)
    I think that's roughly it. So, critic_train_op should be a no-op. There is a tf.no_op apparently.
    And ac_train_op should just be minimizing WARP. """
    def construct_placeholders(self):
        """Any time I need a placeholder, it'll be here. And everything will inherit from this guy.
        That's probably the cleanest way of doing this.
        Sets fields:
            keep_prob_ph:
                The placeholder that determines the dropout fraction. During training, it should
                be something like 0.5. During validation, it should always be 1.
            self.epoch:
                This is useful for doing things that need to scale based on epoch. For example, the
                kl_scaler function will go here.
            self.stddev_effect_on_latent_dim_scaler:
                It's unfortunate that I need this one, but when we're doing testing, we want to
                sample the mean, not anything else.
            self.train_batch_norm:
                I'm not really sure about this one.. But I think it determines if you're training the
                batch-norm or not..."""
        self.keep_prob_ph = tf.placeholder_with_default(1.0, shape=[])
        self.kl_loss_scaler = tf.placeholder_with_default(0.0, shape=None)
        self.stddev_effect_on_latent_dim_scaler = tf.placeholder_with_default(0., shape=[])
        self.train_batch_norm = tf.placeholder_with_default(False, shape=[])
        self.epoch = tf.placeholder_with_default(0.0, shape=[])

    def forward_pass_critic(self):
        with tf.variable_scope("CriticInputVector"):
            self._create_critic_input_vector()

        self.batch_norm_update_ops = tf.get_collection(
            tf.GraphKeys.UPDATE_OPS, scope=".*CriticInputVector.*")

        with tf.variable_scope("RestOfCritic"):
            self._create_critic_network()
```

Figure 3: class of Critic Model

(This class is the critic model which predicts the value of each prediction in actor.)

Round 1

First round Train (just use ml-20m dataset)

Round1 is training actor(we reduced the epochs_pred_only to 20 to reduce the training time)

```
[ ]  import sys
     import os
     UTILS_DIR = os.path.join(os.path.abspath(os.path.dirname('__file__')), '..', 'utils')
     sys.path.insert(1, UTILS_DIR)

     if __name__ == '__main__':

         BREAK_EARLY = False
         BATCH_SIZE = 500

         for data_subdir in ['ml-20m']:# ', netflix-prize', 'msd']:
             actor_path = "VAE_ACTOR_TRAIN_{}".format(data_subdir)
             train(
                 model_class='multi_vae',
                 data_subdir=data_subdir,
                 n_epochs_pred_only=20,
                 n_epochs_ac_only=0,
                 n_epochs_pred_and_ac=0,
                 max_kl=0.2,
                 ac_reg_loss_scaler=0.0,
                 actor_reg_loss_scaler=0.01,
                 evaluation_metric="NDCG",
                 logging_frequency=50,
                 batch_size=BATCH_SIZE,
                 break_early=BREAK_EARLY,
                 verbose=False,
                 version_tag="FULL_RUN_ON_OTHER_DATASETS",
                 path_to_save_actor=actor_path,
                 log_critic_training_error=False,
             )

             print("Now, hopefully on to testing...")

             test(
                 model_class='multi_vae',
                 data_subdir=data_subdir,
                 n_epochs_pred_only=20,
                 n_epochs_ac_only=0,
                 n_epochs_pred_and_ac=0,
                 max_kl=0.2,
                 ac_reg_loss_scaler=0.0,
                 actor_reg_loss_scaler=0.01,
                 evaluation_metric="NDCG",
                 batch_size=BATCH_SIZE,
                 break_early=BREAK_EARLY,
                 verbose=False,
                 version_tag="FULL_RUN_ON_OTHER_DATASETS",
             )

         print("Round 1 done")
```

Figure 4: Round 1 of Main Training

Round 2

Round2 is training critic(we reduced the epochs_ac and epochs_pred_and_ac to 10 to reduce training time)

```
[ ]  print("On to round 2! Now we'll do the critic.")

     train(
                 model_class='multi_vae',
                 data_subdir=data_subdir,
                 n_epochs_pred_only=0,
                 n_epochs_ac_only=10,
                 n_epochs_pred_and_ac=10,
                 max_kl=0.2,
                 ac_reg_loss_scaler=0.0,
                 actor_reg_loss_scaler=0.01,
                 evaluation_metric="NDCG",
                 logging_frequency=50,
                 batch_size=BATCH_SIZE,
                 break_early=BREAK_EARLY,
                 verbose=False,
                 version_tag="FULL_RUN_ON_OTHER_DATASETS",
                 restore_trained_actor_path=actor_path,
             )

     print("Now, hopefully on to testing...")

     test(
                 model_class='multi_vae',
                 data_subdir=data_subdir,
                 n_epochs_pred_only=0,
                 n_epochs_ac_only=10,
                 n_epochs_pred_and_ac=10,
                 max_kl=0.2,
                 ac_reg_loss_scaler=0.0,
                 actor_reg_loss_scaler=0.01,
                 evaluation_metric="NDCG",
                 batch_size=BATCH_SIZE,
                 break_early=BREAK_EARLY,
                 verbose=False,
                 version_tag="FULL_RUN_ON_OTHER_DATASETS",
                 restore_trained_actor_path=actor_path,
             )


     print("Bye bye")
     exit()
```

Figure 5: Round 2 of Main Training

(These two screenshots shows the parameters in the training and testing process.)

```
test batch.
test batch.
Testing done. That broke it out of the loop.
Test UNNORMALIZED DCG@100=2.10775 (0.40559)
Test NDCG@100=0.41396 (0.00209)
Test Recall@50=0.52786 (0.06414)
Test Recall@020=0.38478 (0.06022)
Test NDCG@0200=0.45126 (0.04576)
Test NDCG@5=0.29934 (0.06077)
Test NDCG@3=0.30984 (0.07039)
Test NDCG@1=0.33680 (0.10568)
Round 1 done
```

```
test batch.
test batch.
test batch.
Testing done. That broke it out of the loop.
Test UNNORMALIZED DCG@100=2.14956 (0.41909)
Test NDCG@100=0.41918 (0.00210)
Test Recall@50=0.53122 (0.06403)
Test Recall@020=0.38937 (0.06028)
Test NDCG@0200=0.45657 (0.04618)
Test NDCG@5=0.31066 (0.06230)
Test NDCG@3=0.32310 (0.07194)
Test NDCG@1=0.35380 (0.10692)
Bye bye
```

Figure 6: Main results of Training of Round 1 & 2

(This shows the results of two Round. We can see there is a little improved after Round 2.)

1.1.3 Colab Link

https://colab.research.google.com/drive/1bxyrGw4HW-8Sxp68rYPfRtAECqsC6l5g

1.2 KernelNet

1.2.1 Paper introduction

In the essay *Kernelized Synaptic Weight Matrices*, the author used advanced deep learning for model-based collaborative filtering. Unlike traditional deep learning collaborative filtering, this article begins with a new term, kernelNet, and introduces optimization methods that reduce test errors and loss functions. Ultimately, the entire model consists of an RBF kernel, a supporting RBF kernel, and a fully connected neural network. Its working principle is to create a fully connected neural network with RBF kernel as the core. In this neural network, the weight *w* changes with the movement when scanning dataset. Finally, use L-BFGS to optimize the weight *w* to reduce the value of the loss function.

1.2.2 Colab Screenshots

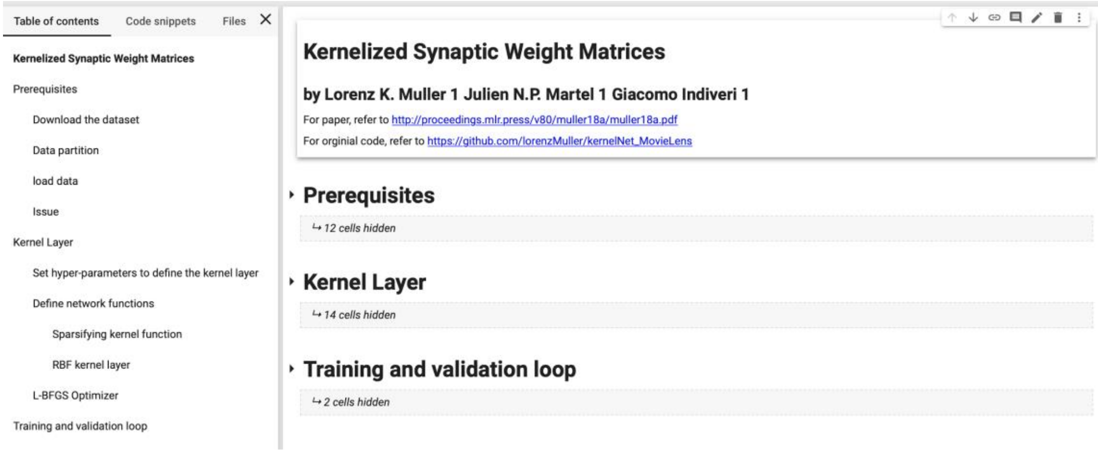
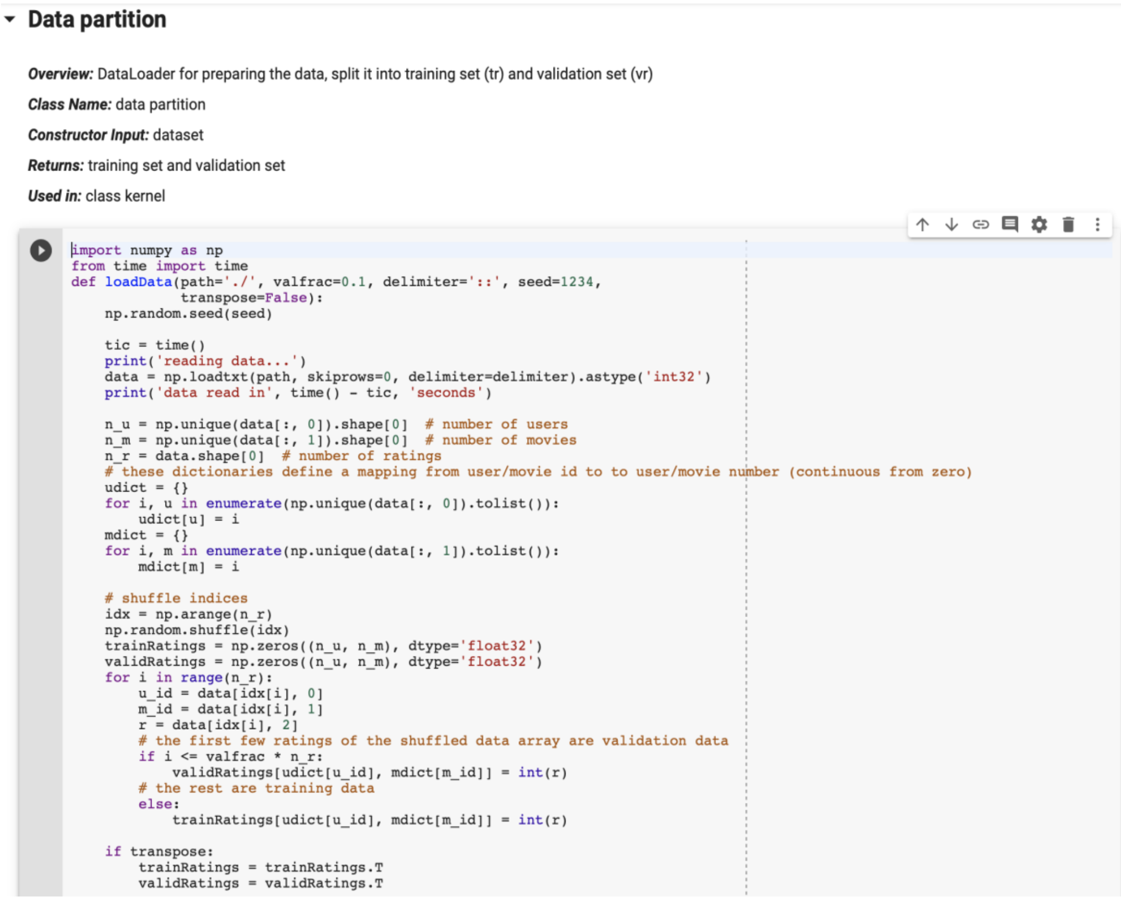


Figure 7: Content and Overview of the code



Data partition

Overview: DataLoader for preparing the data, split it into training set (tr) and validation set (vr)

Class Name: data partition

Constructor Input: dataset

Returns: training set and validation set

Used in: class kernel

```
import numpy as np
from time import time
def loadData(path='./', valfrac=0.1, delimiter='::', seed=1234,
             transpose=False):
    np.random.seed(seed)

    tic = time()
    print('reading data...')
    data = np.loadtxt(path, skiprows=0, delimiter=delimiter).astype('int32')
    print('data read in', time() - tic, 'seconds')

    n_u = np.unique(data[:, 0]).shape[0]  # number of users
    n_m = np.unique(data[:, 1]).shape[0]  # number of movies
    n_r = data.shape[0]  # number of ratings
    # these dictionaries define a mapping from user/movie id to to user/movie number (continuous from zero)
    udict = {}
    for i, u in enumerate(np.unique(data[:, 0]).tolist()):
        udict[u] = i
    mdict = {}
    for i, m in enumerate(np.unique(data[:, 1]).tolist()):
        mdict[m] = i

    # shuffle indices
    idx = np.arange(n_r)
    np.random.shuffle(idx)
    trainRatings = np.zeros((n_u, n_m), dtype='float32')
    validRatings = np.zeros((n_u, n_m), dtype='float32')
    for i in range(n_r):
        u_id = data[idx[i], 0]
        m_id = data[idx[i], 1]
        r = data[idx[i], 2]
        # the first few ratings of the shuffled data array are validation data
        if i <= valfrac * n_r:
            validRatings[udict[u_id], mdict[m_id]] = int(r)
        # the rest are training data
        else:
            trainRatings[udict[u_id], mdict[m_id]] = int(r)

    if transpose:
        trainRatings = trainRatings.T
        validRatings = validRatings.T
```

Figure 8: Data partition function, splitting the data into a training set(*tr*) and a validation set(*vr*)

Overview:

Once the algorithm trains the input with support RBF Kernels, it use full connected neural network to comupute the weght and then use the guaussion kernel to activate the output.

Class Name: kernel_layer

Constructor Input: batch,channels,activation,n_dim,n_hid

Returns: layer output,regularization term

Used in: training data

```
def kernel_layer(x, n_hid=500, n_dim=5, activation=tf.nn.sigmoid, lambda_s=lambda_s,
                 lambda_2=lambda_2, name=''):
    """
    a kernel sparsified layer

    :param x: input [batch, channels]
    :param n_hid: number of hidden units
    :param n_dim: number of dimensions to embed for kernelization
    :param activation: output activation
    :param name: layer name for scoping
    :return: layer output, regularization term
    """

    # define variables
    with tf.variable_scope(name):
        W = tf.get_variable('W', [x.shape[1], n_hid])
        n_in = x.get_shape().as_list()[1]
        u = tf.get_variable('u', initializer=tf.random.truncated_normal([n_in, 1, n_dim], 0., 1e-3))
        v = tf.get_variable('v', initializer=tf.random.truncated_normal([1, n_hid, n_dim], 0., 1e-3))
        b = tf.get_variable('b', [n_hid])

    # compute sparsifying kernel
    # as u and v move further from each other for some given pair of neurons, their connection
    # decreases in strength and eventually goes to zero.
    w_hat = kernel(u, v)

    # compute regularization terms
    #tensorflow can devide the l2 regulation loss value in order to make the result more accuracy.
    sparse_reg = tf.contrib.layers.l2_regularizer(lambda_s)
    sparse_reg_term = tf.contrib.layers.apply_regularization(sparse_reg, [w_hat])

    l2_reg = tf.contrib.layers.l2_regularizer(lambda_2)
    l2_reg_term = tf.contrib.layers.apply_regularization(l2_reg, [W])

    # compute output
    W_eff = W * w_hat
    y = tf.matmul(x, W_eff) + b
    y = activation(y)
    return y, sparse_reg_term + l2_reg_term
```

Figure 9: Kernel layer function, creating the fully-connected neural network to compute the weight by using finite support RBF kernel

Training and validation loop

Overview:

Use rmse to define the efficiency of the model:

$Ermse = \sqrt{\sum}(pi - ri)^2 / N,$

where pi is the predicted rating, ri is the true rating. N is the number of validation samples.

Constructor Input: dataset

Returns: Ermse,validation error,train error

Used in: evaluate the efficiency of the model

```
[ ] init = tf.global_variables_initializer()
    with tf.Session() as sess:
        sess.run(init)
        for i in range(int(n_epoch / output_every)):
            optimizer.minimize(sess, feed_dict={R: tr}) #do maxiter optimization steps
            pre = sess.run(prediction, feed_dict={R: tr}) #predict ratings

            error = (vm * (np.clip(pre, 1., 5.) - vr) ** 2).sum() / vm.sum() #compute validation error
            error_train = (tm * (np.clip(pre, 1., 5.) - tr) ** 2).sum() / tm.sum() #compute train error

            print('.-^-._' * 12)
            print('epoch:', i, 'validation rmse:', np.sqrt(error), 'train rmse:', np.sqrt(error_train))
            print('.-^-._' * 12)

        with open('summary_ml1m.txt', 'a') as file:
            for a in sys.argv[1:]:
                file.write(a + ' ')
            file.write(str(np.sqrt(error)) + ' ' + str(np.sqrt(error_train))
                       + ' ' + str(seed) + '\n')
            file.close()
```

Figure 10: Training and validation loop, using *RMSE* to define the efficiency of the model

```
.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._
INFO:tensorflow:Optimization terminated with:
  Message: b'STOP: TOTAL NO. of ITERATIONS REACHED LIMIT'
  Objective function value: 309810.437500
  Number of iterations: 50
  Number of functions evaluations: 52
.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._
epoch: 16 validation rmse: 0.82989717 train rmse: 0.6868867
.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._
INFO:tensorflow:Optimization terminated with:
  Message: b'STOP: TOTAL NO. of ITERATIONS REACHED LIMIT'
  Objective function value: 308470.625000
  Number of iterations: 50
  Number of functions evaluations: 54
.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._
epoch: 17 validation rmse: 0.82906413 train rmse: 0.6847239
.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._
INFO:tensorflow:Optimization terminated with:
  Message: b'STOP: TOTAL NO. of ITERATIONS REACHED LIMIT'
  Objective function value: 307373.875000
  Number of iterations: 50
  Number of functions evaluations: 54
.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._
epoch: 18 validation rmse: 0.8295139 train rmse: 0.6834792
.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._
INFO:tensorflow:Optimization terminated with:
  Message: b'STOP: TOTAL NO. of ITERATIONS REACHED LIMIT'
  Objective function value: 306335.718750
  Number of iterations: 50
  Number of functions evaluations: 53
.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._.-^-._
epoch: 19 validation rmse: 0.8294705 train rmse: 0.68156505
  ^     ^     ^     ^     ^     ^     ^     ^     ^     ^     ^     ^
```

Figure 11: Result for the final epoch

The model is evaluated by RMSE and sets 20 epoches to fetch the accuracy. The result is shown below and we can figure out that in the 20th epochs the RMSE reaches the lowest number 0.8294705.

1.2.3 Colab Link

https://colab.research.google.com/drive/1l5EP5dMe0XPEQ1H85iuIQE8cQus0O1qm#scrollTo=5-bKMqe0IuGl

1.3 GCMC

1.3.1 Paper Introduction

This paper introduces graph convolutional matrix completion (GC-MC): a graph auto-encoder model which view matrix completion to link prediction problem in the bipartite graph. The rating matrix which represents user-item interaction is converted to the bipartite structure. The table below shows the relationship between matrix completion and link prediction on bipartite graph. After that, applying the end-to-end trainable graph auto-encoder to model and produce latent features of user and item nodes which can be used to reconstruct and predict links by the bilinear decoder model.

Matrix completion	**Link prediction on graph**
Interaction data	Bipartite graph (between user and item nodes)
Observed ratings/purchases	Links
Content information	Node features
Predict ratings	Predict labeled links

Table 1: the relationship between matrix completion and link prediction

1.3.2 Colab Screenshots

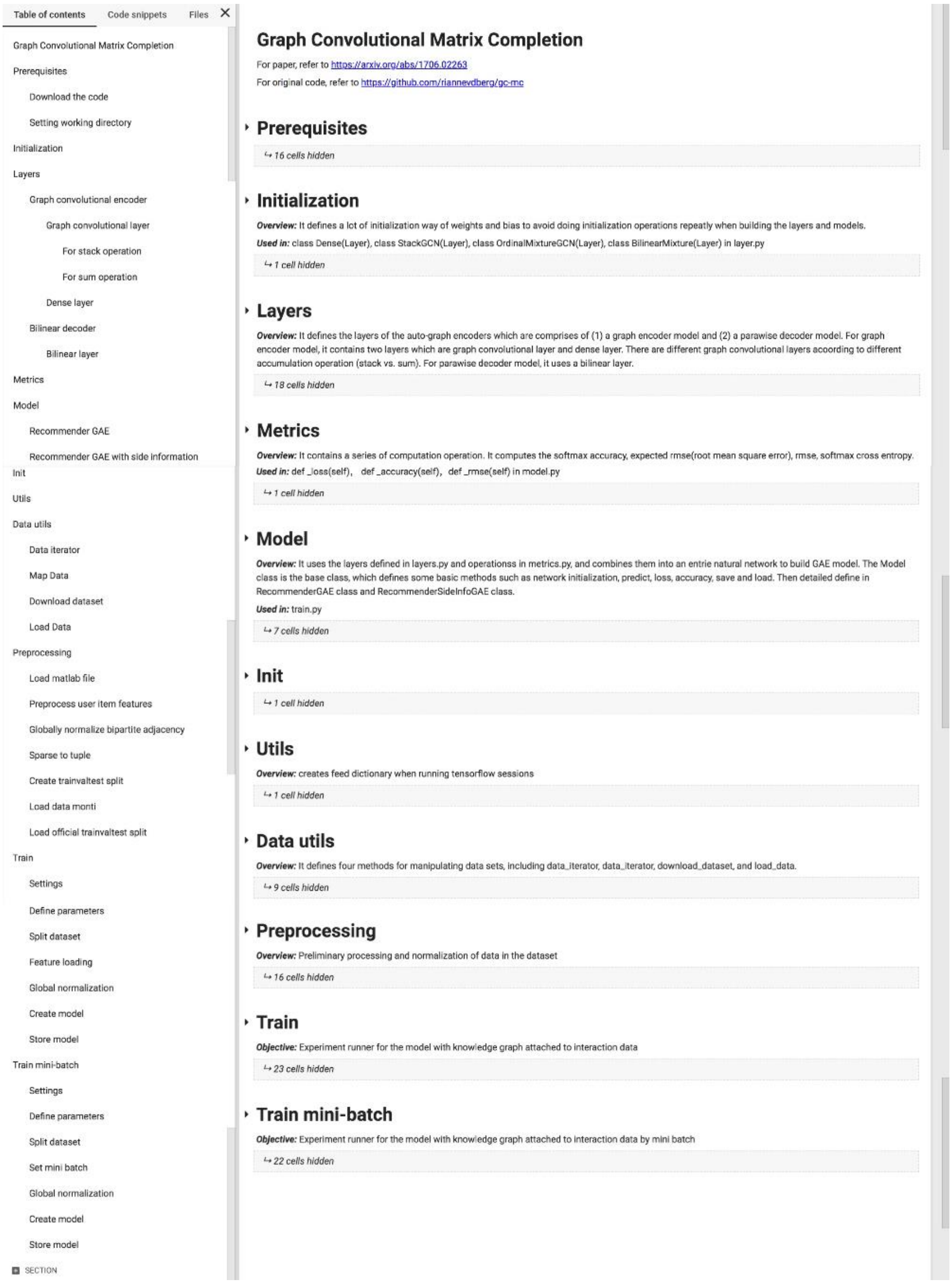


Figure 12: Content and Overview of the code

As the graph above shown, there are a lot of content of GCMC code. However, the graph auto-encoder model is very import which is shown in Layers. It has two parts which can be seen in the graph below: graph convolutional encoder which passes and transform message between users and items and bilinear decoder which reconstructs the links to predict ratings.

- Layers
  - Graph convolutional encoder
    - Graph convolutional layer
      - For stack operation
      - For sum operation
    - Dense layer
  - Bilinear decoder
    - Bilinear layer

Figure 13: The hierarchy of graph auto-encoder model

For graph convolutional encoder, the graph convolutional layer accumulates incoming messages at every node into a single vector by using different accumulation operation. Based on the type of accumulation operation, the difference can be seen in the code. Stack operation is a concatenation of vectors. Sum operation is the summation of all messages.

For stack operation

```
[ ] class StackGCN(Layer):
        """Graph convolution layer for bipartite graphs and sparse inputs."""

        def __init__(self, input_dim, output_dim, support, support_t, num_support, u_features_nonzero=None,
                     v_features_nonzero=None, sparse_inputs=False, dropout=0.,
                     act=tf.nn.relu, share_user_item_weights=True, **kwargs):
            super(StackGCN, self).__init__(**kwargs)

            assert output_dim % num_support == 0, 'output_dim must be multiple of num_support for stackGC layer'

            with tf.variable_scope(self.name + '_vars'):
                self.vars['weights_u'] = weight_variable_random_uniform(input_dim, output_dim, name='weights_u')

                if not share_user_item_weights:
                    self.vars['weights_v'] = weight_variable_random_uniform(input_dim, output_dim, name='weights_v')

                else:
                    self.vars['weights_v'] = self.vars['weights_u']

            self.weights_u = tf.split(value=self.vars['weights_u'], axis=1, num_or_size_splits=num_support)
            self.weights_v = tf.split(value=self.vars['weights_v'], axis=1, num_or_size_splits=num_support)

            self.dropout = dropout

            self.sparse_inputs = sparse_inputs
            self.u_features_nonzero = u_features_nonzero
            self.v_features_nonzero = v_features_nonzero
            if sparse_inputs:
                assert u_features_nonzero is not None and v_features_nonzero is not None, \
                    'u_features_nonzero and v_features_nonzero can not be None when sparse_inputs is True'

            self.support = tf.sparse_split(axis=1, num_split=num_support, sp_input=support)
            self.support_transpose = tf.sparse_split(axis=1, num_split=num_support, sp_input=support_t)

            self.act = act

            if self.logging:
```

Figure 14: Graph convolutional layer with stack operation

For sum operation

```
[ ] class OrdinalMixtureGCN(Layer):

        """Graph convolution layer for bipartite graphs and sparse inputs."""

        def __init__(self, input_dim, output_dim, support, support_t, num_support, u_features_nonzero=None,
                     v_features_nonzero=None, sparse_inputs=False, dropout=0.,
                     act=tf.nn.relu, bias=False, share_user_item_weights=False, self_connections=False, **kwargs):
            super(OrdinalMixtureGCN, self).__init__(**kwargs)

            with tf.variable_scope(self.name + '_vars'):

                self.vars['weights_u'] = tf.stack([weight_variable_random_uniform(input_dim, output_dim,
                                                                                  name='weights_u_%d' % i)
                                                   for i in range(num_support)], axis=0)

                if bias:
                    self.vars['bias_u'] = bias_variable_const([output_dim], 0.01, name="bias_u")

                if not share_user_item_weights:
                    self.vars['weights_v'] = tf.stack([weight_variable_random_uniform(input_dim, output_dim,
                                                                                      name='weights_v_%d' % i)
                                                       for i in range(num_support)], axis=0)

                    if bias:
                        self.vars['bias_v'] = bias_variable_const([output_dim], 0.01, name="bias_v")

                else:
                    self.vars['weights_v'] = self.vars['weights_u']
                    if bias:
                        self.vars['bias_v'] = self.vars['bias_u']

            self.weights_u = self.vars['weights_u']
            self.weights_v = self.vars['weights_v']

            self.dropout = dropout

            self.sparse_inputs = sparse_inputs
```

Figure 15: Graph convolutional layer with sum operation

A dense layer is after which support the final embedding of nodes. The corresponding code is like:

Dense layer

Class Name: Dense

Constructor Input:

dimention of user features (input_dim)

the number hidden units in 1st layer (output_dim)

dropout fraction (dropout)

activation function (act)

weight variable (share_user_item_weights)

Returns: matrix after activation function

Used in: def _build(self) in model.py

```
[ ] class Dense(Layer):
        """Dense layer for two types of nodes in a bipartite graph. """

        def __init__(self, input_dim, output_dim, dropout=0., act=tf.nn.relu, share_user_item_weights=False,
                     bias=False, **kwargs):
            super(Dense, self).__init__(**kwargs)

            with tf.variable_scope(self.name + '_vars'):
                if not share_user_item_weights:

                    self.vars['weights_u'] = weight_variable_random_uniform(input_dim, output_dim, name="weights_u")
                    self.vars['weights_v'] = weight_variable_random_uniform(input_dim, output_dim, name="weights_v")

                    if bias:
                        self.vars['user_bias'] = bias_variable_truncated_normal([output_dim], name="bias_u")
                        self.vars['item_bias'] = bias_variable_truncated_normal([output_dim], name="bias_v")
```

Figure 16: Dense layer

For bilinear decoder, it uses a softmax function through a bilinear operation to get the probability distribution of rating level. This part is represented as bilinear layer in the code as:

Bilinear decoder

Bilinear layer

```
[ ] class BilinearMixture(Layer):
        """
        Decoder model layer for link-prediction with ratings
        To use in combination with bipartite layers.
        """

        def __init__(self, num_classes, u_indices, v_indices, input_dim, num_users, num_items, user_item_bias=False,
                     dropout=0., act=tf.nn.softmax, num_weights=3,
                     diagonal=True, **kwargs):
            super(BilinearMixture, self).__init__(**kwargs)
            with tf.variable_scope(self.name + '_vars'):

                for i in range(num_weights):
                    if diagonal:
                        #  Diagonal weight matrices for each class stored as vectors
                        self.vars['weights_%d' % i] = weight_variable_random_uniform(1, input_dim, name='weights_%d' % i)

                    else:
                        self.vars['weights_%d' % i] = orthogonal([input_dim, input_dim], name='weights_%d' % i)

                self.vars['weights_scalars'] = weight_variable_random_uniform(num_weights, num_classes,
                                                                              name='weights_u_scalars')

                if user_item_bias:
                    self.vars['user_bias'] = bias_variable_zero([num_users, num_classes], name='user_bias')
                    self.vars['item_bias'] = bias_variable_zero([num_items, num_classes], name='item_bias')

            self.user_item_bias = user_item_bias

            if diagonal:
                self._multiply_inputs_weights = tf.multiply
            else:
```

Figure 17: Bilinear layer

1.3.3 Colab Link

https://colab.research.google.com/drive/1WY6c6isb4ahwz11lnCniFDwmUL7KRafu

2. OBSTACLES

2.1 RaCT

We firstly ran setup_data.py to get 3 datsets smoothly, then we ran main_vae.py to check if the results are consistent with the paper. However, other issues occurred, the path error. Because at first time, we used two methods to deploy the code into Colab. One is the mount way, make Colab access our Drive file system. So we can read and run the code directly. But in this way, the path is error. In the code, it uses '../' to access the file path. We made a test to check what is the right way to access the Drive file system. Finally, we figured out this issue by correcting '../' to './'. And another way is getting the shared ID and download them in the Colab. This way can run smoothly with the original code. However, the final version is different from these two ways. We replicated some parts of code to Colab, separated them into block, and made some annotations. When we deployed the code to Google Colab, there were some problems. Firstly, when we ran the code for setup data, it occurred an error which is name '__file__' is not defined. We solved this problem through add single quotation marks to __file__. Secondly, when we ran the code for main training, we met an error called 'ipykernel_launcher.py: error: unrecognized arguments'. We found that the problem code was parser.parse_args() and modified it to parser.parse_known_args()[0] to solve the problem. While due to our large dataset, there is crashing error when running the code. We spent a lot of time with tutors solving this problem. Fortunately, we figured it out by running the block one by one. For the time management, because our large dataset, the time of training took us about 30 hours. At first time, we forget changing the running type to GPU. While after we change it, it also took us about 1 hours due to the large epoch. So according to the code, we modify the epoch during these two rounds. Round 1 is 20 epoch and Round 2 is 20 as well. At this circumstance, it just only took about 37 minutes. And the NDCG evaluation value is about 0.42 which is close to the value in the paper.

2.2 KernelNet

The environment setup session on the local machine is quite straightforward, we simply download the code from the github and put the data under the same folder, then it can be run through Python Console using Python3. However, when we ran the code on MacOS, one little problem happened, because of the format of character encoding is different between different systems like Mac and Windows and Linux, the address of the data file cannot be read in the original code. To solve it, we delete the path of the data and typed the actual path again to make it work. Also, when we try to transfer everything into Google Colab, another issue occured. The settings of hyperparameters lambda_s and lambda_2 in the kernel layer failed, because the path of the code has changed on the Colab server, so when we try to use the sys.argv[0] and sys.argv[1], we cannot get the address and content we truly want, so we have to set the parameters to some default value manually instead of allowing it change upon request. The last problem we found is that under both environments, the running time of the code is too long, it is around 5-10 minutes per epochs. For 20 epochs, we normally have to wait around 2 hours, no matter on the local machine or on Google Colab server. We haven't fixed this latent issue because it might be very common, but we will dig more into this in the following weeks.

2.3 GC-MC

The biggest problem we encountered in running and testing code related to GC-MC was the compatibility of Python 2.x with Python 3.x. The GC-MC code was built in the Python 2.7 environment, and we are using Python 3.6. Therefore, some compatibility-related errors occurred during code execution. For example, in Python 2.7, StingIO can handle not only character streams, but also binary data. However, in Python 3.x, StingIO can only handle character stream data, not binary data. Another major error is TypeError: a bytes-like object is required, not 'str'. The main reason for this error is that Python 3.x and Python 2.x normalize the dataset after downloading the dataset. The decoding of the return value is different. This problem was resolved after we modified the format of the socket return value decoding.

In terms of time performance of the code, we compared the runtime of each epoch with GPU acceleration and without GPU acceleration. Take the Movie Lens 100k as an example. Running the code in colab without using a GPU, each epoch takes about 0.2 seconds. If you use GPU acceleration, each epoch takes about 0.05 seconds. Therefore, GPU-accelerated code runs about four times faster than without the GPU.

3. DEVIATION TO TIMELINE

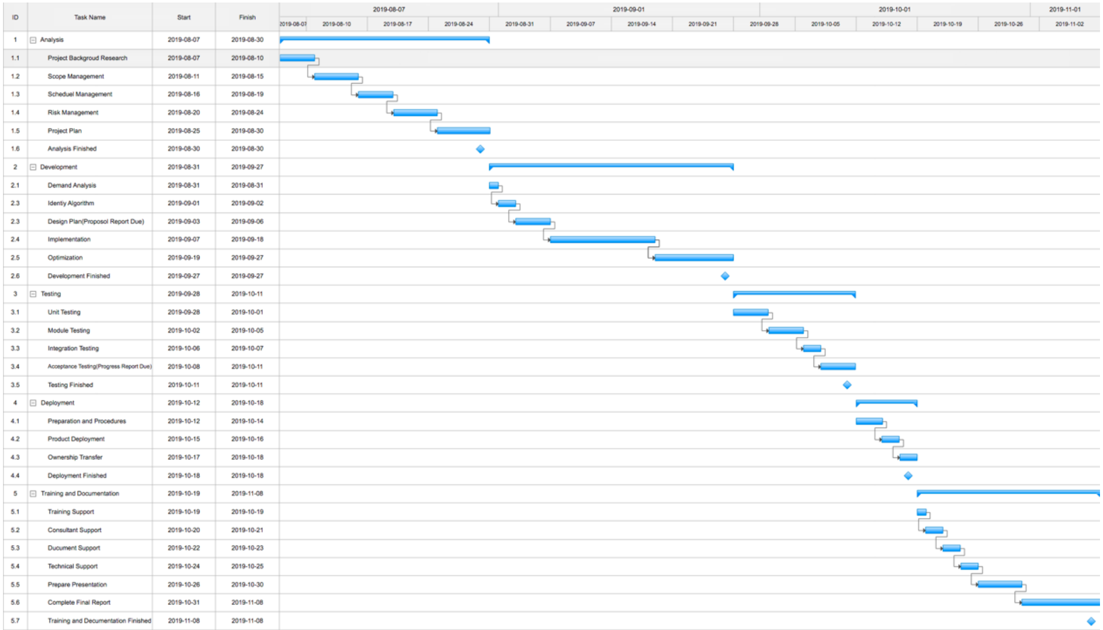


Figure 18: Gantt Chart

W1-W3: background research

W4-W8: replication and analysis of the current state of the article

W9-W12: 1.interface 2.modify current code

Because the training time is too long and there are some errors of the code, we spent a lot of time to solve these problems and got the final results. This led to the changes in timeline.

4. MILESTONES & REPORTING

Milestone	Tasks	Reporting	Date
Week-1	Understand the project & research background knowledge	None	2019-08-09
Week-2	Meet with our client and define project plan	Client meeting to review the project	2019-08-16
Week-3	Identify the requirements of the project clearly & Analyze two websites	Client meeting to identify the requirement about the project	2019-08-23

Week-4	Focus on the recommendation part & Do literature review	Client meeting to compare the difference of two websites	2019-08-30
Week-5	Proposal Report Due & Define recommendation system algorithm	Client meeting to understand the details of papers	2019-09-06
Week-6	Complete design plan	Client meeting to review the design plan	2019-09-13
Week-7	Implement recommendation system	Client meeting to review Implementation	2019-09-20
Week-8	Optimization	Client meeting to review final performance	2019-09-27
Week-9	Testing & Progress Report Due	Client meeting to review test outcome	2019-10-11
Week-10	~~Deployment~~ Deployment & Generalization I	Client meeting to deploy the system	2019-10-18
Week-11	~~Training&Documentation~~ Deployment & Generalization II	Client meeting to review documents	2019-10-25
Week-12	Final Presentation	None	2019-11-01
Week-13	Final Report (thesis)	None	2019-11-08

Table 2: The milestones of the project weekly