Recommendation Systems with TensorFlow

Introduction

In this lab, we will create a movie recommendation system based on the MovieLens dataset available here. The data consists of movies ratings (on a scale of 1 to 5). Specifically, we'll be using matrix factorization to learn user and movie embeddings. Concepts highlighted here are also available in the course on Recommendation Systems.

Objectives

1. Explore the MovieLens Data
2. Train a matrix factorization model
3. Inspect the Embeddings
4. Perform Softmax model training

In [1]:
```
# Ensure the right version of Tensorflow is installed.
!pip freeze | grep tensorflow==2.6
```

In [3]:
```
from __future__ import print_function

import numpy as np
import pandas as pd
import collections
from mpl_toolkits.mplot3d import Axes3D
from IPython import display
from matplotlib import pyplot as plt
import sklearn
import sklearn.manifold
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

# Add some convenience functions to Pandas DataFrame.
pd.options.display.max_rows = 10
pd.options.display.float_format = '{:.3f}'.format
def mask(df, key, function):
  """Returns a filtered dataframe, by applying function to key"""
  return df[function(df[key])]

def flatten_cols(df):
  df.columns = [' '.join(col).strip() for col in df.columns.values]
  return df

pd.DataFrame.mask = mask
pd.DataFrame.flatten_cols = flatten_cols
```

```
WARNING:tensorflow:From /opt/conda/lib/python3.7/site-packages/tensorflow/pytho
n/compat/v2_compat.py:101: disable_resource_variables (from tensorflow.python.op
s.variable_scope) is deprecated and will be removed in a future version.
```

```
Instructions for updating:
non-resource variables are not supported in the long term
```

In [3]:

```
#Let's install Altair for interactive visualizations

!pip install git+git://github.com/altair-viz/altair.git
```

In [11]:

```
import altair as alt
alt.data_transformers.enable('default', max_rows=None)
#alt.renderers.enable('colab')
```

Out[11]:

```
DataTransformerRegistry.enable('default')
```

We then download the MovieLens Data, and create DataFrames containing movies, users, and ratings.

In [1]:

```
# Download MovieLens data.
print("Downloading movielens data...")
from urllib.request import urlretrieve
import zipfile

urlretrieve("http://files.grouplens.org/datasets/movielens/ml-100k.zip", "moviel
zip_ref = zipfile.ZipFile('movielens.zip', "r")
zip_ref.extractall()
print("Done. Dataset contains:")
print(zip_ref.read('ml-100k/u.info'))
```

```
Downloading movielens data...
Done. Dataset contains:
b'943 users\n1682 items\n100000 ratings\n'
```

In [4]:

```
# Load each data set (users, ratings, and movies).
users_cols = ['user_id', 'age', 'sex', 'occupation', 'zip_code']
users = pd.read_csv(
    'ml-100k/u.user', sep='|', names=users_cols, encoding='latin-1')

ratings_cols = ['user_id', 'movie_id', 'rating', 'unix_timestamp']
ratings = pd.read_csv(
    'ml-100k/u.data', sep='\t', names=ratings_cols, encoding='latin-1')

# The movies file contains a binary feature for each genre.
genre_cols = [
    "genre_unknown", "Action", "Adventure", "Animation", "Children", "Comedy",
    "Crime", "Documentary", "Drama", "Fantasy", "Film-Noir", "Horror",
    "Musical", "Mystery", "Romance", "Sci-Fi", "Thriller", "War", "Western"
]
movies_cols = [
    'movie_id', 'title', 'release_date', "video_release_date", "imdb_url"
] + genre_cols
movies = pd.read_csv(
    'ml-100k/u.item', sep='|', names=movies_cols, encoding='latin-1')

# Since the ids start at 1, we shift them to start at 0. This will make handling
# indices easier later
users["user_id"] = users["user_id"].apply(lambda x: str(x-1))
movies["movie_id"] = movies["movie_id"].apply(lambda x: str(x-1))
movies["year"] = movies['release_date'].apply(lambda x: str(x).split('-')[-1])
```

```
ratings["movie_id"] = ratings["movie_id"].apply(lambda x: str(x-1))
ratings["user_id"] = ratings["user_id"].apply(lambda x: str(x-1))
ratings["rating"] = ratings["rating"].apply(lambda x: float(x))
```

In [5]:

```
# Compute the number of movies to which a genre is assigned.
genre_occurences = movies[genre_cols].sum().to_dict()

# Since some movies can belong to more than one genre, we create different
# 'genre' columns as follows:
# - all_genres: all the active genres of the movie.
# - genre: randomly sampled from the active genres.
def mark_genres(movies, genres):
  def get_random_genre(gs):
    active = [genre for genre, g in zip(genres, gs) if g==1]
    if len(active) == 0:
      return 'Other'
    return np.random.choice(active)
  def get_all_genres(gs):
    active = [genre for genre, g in zip(genres, gs) if g==1]
    if len(active) == 0:
      return 'Other'
    return '-'.join(active)
  movies['genre'] = [
      get_random_genre(gs) for gs in zip(*[movies[genre] for genre in genres])]
  movies['all_genres'] = [
      get_all_genres(gs) for gs in zip(*[movies[genre] for genre in genres])]

mark_genres(movies, genre_cols)

# Create one merged DataFrame containing all the movielens data.
movielens = ratings.merge(movies, on='movie_id').merge(users, on='user_id')
```

In [6]:

```
# Utility to split the data into training and test sets.
def split_dataframe(df, holdout_fraction=0.1):
  """Splits a DataFrame into training and test sets.
  Args:
    df: a dataframe.
    holdout_fraction: fraction of dataframe rows to use in the test set.
  Returns:
    train: dataframe for training
    test: dataframe for testing
  """
  test = df.sample(frac=holdout_fraction, replace=False)
  train = df[~df.index.isin(test.index)]
  return train, test
```

Exploring the Movielens Data

Before we dive into model building, let's inspect our MovieLens dataset. It is usually helpful to understand the statistics of the dataset.

Users

We start by printing some basic statistics describing the numeric user features.

In [7]:

```
users.describe()
```

Out[7]:

	age
count	943.000
mean	34.052
std	12.193
min	7.000
25%	25.000
50%	31.000
75%	43.000
max	73.000

We can also print some basic statistics describing the categorical user features

In [8]:

```
users.describe(include=[np.object])
```

Out[8]:

	user_id	sex	occupation	zip_code
count	943	943	943	943
unique	943	2	21	795
top	0	M	student	55414
freq	1	670	196	9

We can also create histograms to further understand the distribution of the users. We use Altair to create an interactive chart.

In [12]:

```
# The following functions are used to generate interactive Altair charts.
# We will display histograms of the data, sliced by a given attribute.

# Create filters to be used to slice the data.
occupation_filter = alt.selection_multi(fields=["occupation"])
occupation_chart = alt.Chart().mark_bar().encode(
    x="count()",
    y=alt.Y("occupation:N"),
    color=alt.condition(
        occupation_filter,
        alt.Color("occupation:N", scale=alt.Scale(scheme='category20')),
        alt.value("lightgray")),
).properties(width=300, height=300, selection=occupation_filter)

# A function that generates a histogram of filtered data.
def filtered_hist(field, label, filter):
  """Creates a layered chart of histograms.
  The first layer (light gray) contains the histogram of the full data, and the
  second contains the histogram of the filtered data.
  Args:
    field: the field for which to generate the histogram.
    label: String label of the histogram.
```

```
      filter: an alt.Selection object to be used to filter the data.
    """
    base = alt.Chart().mark_bar().encode(
        x=alt.X(field, bin=alt.Bin(maxbins=10), title=label),
        y="count()",
    ).properties(
        width=300,
    )
    return alt.layer(
        base.transform_filter(filter),
        base.encode(color=alt.value('lightgray'), opacity=alt.value(.7)),
    ).resolve_scale(y='independent')
```

Next, we look at the distribution of ratings per user. Clicking on an occupation in the right chart will filter the data by that occupation. The corresponding histogram is shown in blue, and superimposed with the histogram for the whole data (in light gray). You can use SHIFT+click to select multiple subsets.

What do you observe, and how might this affect the recommendations?

In [13]:

```
users_ratings = (
    ratings
    .groupby('user_id', as_index=False)
    .agg({'rating': ['count', 'mean']})
    .flatten_cols()
    .merge(users, on='user_id')
)

# Create a chart for the count, and one for the mean.
alt.hconcat(
    filtered_hist('rating count', '# ratings / user', occupation_filter),
    filtered_hist('rating mean', 'mean user rating', occupation_filter),
    occupation_chart,
    data=users_ratings)
```

Out[13]:

Movies

It is also useful to look at information about the movies and their ratings.

In [14]:

```
movies_ratings = movies.merge(
    ratings
    .groupby('movie_id', as_index=False)
    .agg({'rating': ['count', 'mean']})
    .flatten_cols(),
    on='movie_id')

genre_filter = alt.selection_multi(fields=['genre'])
genre_chart = alt.Chart().mark_bar().encode(
    x="count()",
    y=alt.Y('genre'),
    color=alt.condition(
        genre_filter,
        alt.Color("genre:N"),
```

```
        alt.value('lightgray'))
).properties(height=300, selection=genre_filter)
```

In [15]:

```
(movies_ratings[['title', 'rating count', 'rating mean']]
 .sort_values('rating count', ascending=False)
 .head(10))
```

Out[15]:

	title	rating count	rating mean
49	Star Wars (1977)	583	4.358
257	Contact (1997)	509	3.804
99	Fargo (1996)	508	4.156
180	Return of the Jedi (1983)	507	4.008
293	Liar Liar (1997)	485	3.157
285	English Patient, The (1996)	481	3.657
287	Scream (1996)	478	3.441
0	Toy Story (1995)	452	3.878
299	Air Force One (1997)	431	3.631
120	Independence Day (ID4) (1996)	429	3.438

In [16]:

```
(movies_ratings[['title', 'rating count', 'rating mean']]
 .mask('rating count', lambda x: x > 20)
 .sort_values('rating mean', ascending=False)
 .head(10))
```

Out[16]:

	title	rating count	rating mean
407	Close Shave, A (1995)	112	4.491
317	Schindler's List (1993)	298	4.466
168	Wrong Trousers, The (1993)	118	4.466
482	Casablanca (1942)	243	4.457
113	Wallace & Gromit: The Best of Aardman Animatio...	67	4.448
63	Shawshank Redemption, The (1994)	283	4.445
602	Rear Window (1954)	209	4.388
11	Usual Suspects, The (1995)	267	4.386
49	Star Wars (1977)	583	4.358
177	12 Angry Men (1957)	125	4.344

Finally, the last chart shows the distribution of the number of ratings and average rating.

In [17]:

```
# Display the number of ratings and average rating per movie.
alt.hconcat(
    filtered_hist('rating count', '# ratings / movie', genre_filter),
    filtered_hist('rating mean', 'mean movie rating', genre_filter),
```

```
    genre_chart,
    data=movies_ratings)
```

Out[17]:

Preliminaries

Our goal is to factorize the ratings matrix A into the product of a user embedding matrix U and movie embedding matrix V, such that $A \approx UV^\top$ with $U = \begin{bmatrix} u_{1} \\ \hline \vdots \\ \hline u_{N} \end{bmatrix}$ and $V = \begin{bmatrix} v_{1} \\ \hline \vdots \\ \hline v_{M} \end{bmatrix}$.

Here

- N is the number of users,
- M is the number of movies,
- A_{ij} is the rating of the jth movies by the ith user,
- each row U_i is a d-dimensional vector (embedding) representing user i,
- each rwo V_j is a d-dimensional vector (embedding) representing movie j,
- the prediction of the model for the (i, j) pair is the dot product $\langle U_i, V_j \rangle$.

Sparse Representation of the Rating Matrix

The rating matrix could be very large and, in general, most of the entries are unobserved, since a given user will only rate a small subset of movies. For effcient representation, we will use a tf.SparseTensor. A `SparseTensor` uses three tensors to represent the matrix: `tf.SparseTensor(indices, values, dense_shape)` represents a tensor, where a value $A_{ij} = a$ is encoded by setting `indices[k] = [i, j]` and `values[k] = a`. The last tensor `dense_shape` is used to specify the shape of the full underlying matrix.

Toy example

Assume we have 2 users and 4 movies. Our toy ratings dataframe has three ratings,

user_id	movie_id	rating
0	0	5.0
0	1	3.0
1	3	1.0

The corresponding rating matrix is

$$ A = \begin{bmatrix} 5.0 & 3.0 & 0 & 0 \\ 0 & 0 & 0 & 1.0 \end{bmatrix} $$
And the SparseTensor representation is,

```
SparseTensor(
  indices=[[0, 0], [0, 1], [1,3]],
```

```
    values=[5.0, 3.0, 1.0],
    dense_shape=[2, 4])
```

Exercise 1: Build a tf.SparseTensor representation of the Rating Matrix.

In this exercise, we'll write a function that maps from our `ratings` DataFrame to a `tf.SparseTensor`.

Hint: you can select the values of a given column of a Dataframe `df` using `df['column_name'].values`.

In [26]:

```
def build_rating_sparse_tensor(ratings_df):
  """
  Args:
    ratings_df: a pd.DataFrame with `user_id`, `movie_id` and `rating` columns.
  Returns:
    A tf.SparseTensor representing the ratings matrix.
  """
  # ======================== Complete this section ============================
  indices = ratings_df[['user_id', 'movie_id']].values
  values = ratings_df['rating'].values
  # ============================================================================

  return tf.SparseTensor(
      indices=indices,
      values=values,
      dense_shape=[users.shape[0], movies.shape[0]])
```

Calculating the error

The model approximates the ratings matrix A by a low-rank product $UV^\top$. We need a way to measure the approximation error. We'll start by using the Mean Squared Error of observed entries only (we will revisit this later). It is defined as

$$ \begin{align*} \text{MSE}(A, UV^\top) &= \frac{1}{|\Omega|}\sum_{(i, j) \in\Omega}{(A_{ij} - (UV^\top)_{ij})^2} \\ &= \frac{1}{|\Omega|}\sum_{(i, j) \in\Omega}{(A_{ij} - \langle U_i, V_j\rangle)^2} \end{align*} $$
where Ω is the set of observed ratings, and $|\Omega|$ is the cardinality of Ω.

Exercise 2: Mean Squared Error

Write a TensorFlow function that takes a sparse rating matrix A and the two embedding matrices U, V and returns the mean squared error $\text{MSE}(A, UV^\top)$.

Hints:

- in this section, we only consider observed entries when calculating the loss.
- a `SparseTensor` `sp_x` is a tuple of three Tensors: `sp_x.indices`, `sp_x.values` and `sp_x.dense_shape`.
- you may find `tf.gather_nd` and `tf.losses.mean_squared_error` helpful.

In [27]:

```
def sparse_mean_square_error(sparse_ratings, user_embeddings, movie_embeddings):
  """
  Args:
    sparse_ratings: A SparseTensor rating matrix, of dense_shape [N, M]
    user_embeddings: A dense Tensor U of shape [N, k] where k is the embedding
      dimension, such that U_i is the embedding of user i.
    movie_embeddings: A dense Tensor V of shape [M, k] where k is the embedding
      dimension, such that V_j is the embedding of movie j.
  Returns:
    A scalar Tensor representing the MSE between the true ratings and the
      model's predictions.
  """
  # ======================== Complete this section ============================
  predictions = tf.gather_nd(tf.matmul(user_embeddings, movie_embeddings, transp
                            sparse_ratings.indices)
  loss = tf.losses.mean_squared_error(sparse_ratings.values,predictions)
  # ===========================================================================
  return loss
```

Note: One approach is to compute the full prediction matrix $UV^\top$, then gather the entries corresponding to the observed pairs. The memory cost of this approach is $O(NM)$. For the MovieLens dataset, this is fine, as the dense $N \times M$ matrix is small enough to fit in memory ($N = 943$, $M = 1682$).

Another approach (given in the alternate solution below) is to only gather the embeddings of the observed pairs, then compute their dot products. The memory cost is $O(|\Omega| d)$ where d is the embedding dimension. In our case, $|\Omega| = 10^5$, and the embedding dimension is on the order of 10, so the memory cost of both methods is comparable. But when the number of users or movies is much larger, the first approach becomes infeasible.

In [28]:

```
#Alternate Solution
def sparse_mean_square_error(sparse_ratings, user_embeddings, movie_embeddings):
  """
  Args:
    sparse_ratings: A SparseTensor rating matrix, of dense_shape [N, M]
    user_embeddings: A dense Tensor U of shape [N, k] where k is the embedding
      dimension, such that U_i is the embedding of user i.
    movie_embeddings: A dense Tensor V of shape [M, k] where k is the embedding
      dimension, such that V_j is the embedding of movie j.
  Returns:
    A scalar Tensor representing the MSE between the true ratings and the
      model's predictions.
  """
  predictions = tf.reduce_sum(
      tf.gather(user_embeddings, sparse_ratings.indices[:, 0]) *
      tf.gather(movie_embeddings, sparse_ratings.indices[:, 1]),
      axis=1)
  loss = tf.losses.mean_squared_error(sparse_ratings.values, predictions)
  return loss
```

Training a Matrix Factorization model

CFModel (Collaborative Filtering Model) helper class

This is a simple class to train a matrix factorization model using stochastic gradient descent.

The class constructor takes

- the user embeddings U (a `tf.Variable`).
- the movie embeddings V, (a `tf.Variable`).
- a loss to optimize (a `tf.Tensor`).
- an optional list of metrics dictionaries, each mapping a string (the name of the metric) to a tensor. These are evaluated and plotted during training (e.g. training error and test error).

After training, one can access the trained embeddings using the `model.embeddings` dictionary.

Example usage:

```
U_var = ...
V_var = ...
loss = ...
model = CFModel(U_var, V_var, loss)
model.train(iterations=100, learning_rate=1.0)
user_embeddings = model.embeddings['user_id']
movie_embeddings = model.embeddings['movie_id']
```

In [29]:

```
class CFModel(object):
  """Simple class that represents a collaborative filtering model"""
  def __init__(self, embedding_vars, loss, metrics=None):
    """Initializes a CFModel.
    Args:
      embedding_vars: A dictionary of tf.Variables.
      loss: A float Tensor. The loss to optimize.
      metrics: optional list of dictionaries of Tensors. The metrics in each
        dictionary will be plotted in a separate figure during training.
    """
    self._embedding_vars = embedding_vars
    self._loss = loss
    self._metrics = metrics
    self._embeddings = {k: None for k in embedding_vars}
    self._session = None

  @property
  def embeddings(self):
    """The embeddings dictionary."""
    return self._embeddings

  def train(self, num_iterations=100, learning_rate=1.0, plot_results=True,
            optimizer=tf.train.GradientDescentOptimizer):
    """Trains the model.
    Args:
      iterations: number of iterations to run.
      learning_rate: optimizer learning rate.
      plot_results: whether to plot the results at the end of training.
      optimizer: the optimizer to use. Default to GradientDescentOptimizer.
    Returns:
```

```
      The metrics dictionary evaluated at the last iteration.
    """

    with self._loss.graph.as_default():
      opt = optimizer(learning_rate)
      train_op = opt.minimize(self._loss)
      local_init_op = tf.group(
          tf.variables_initializer(opt.variables()),
          tf.local_variables_initializer())
      if self._session is None:
        self._session = tf.Session()
        with self._session.as_default():
          self._session.run(tf.global_variables_initializer())
          self._session.run(tf.tables_initializer())
          #tf.train.start_queue_runners()

    with self._session.as_default():
      local_init_op.run()
      iterations = []
      metrics = self._metrics or ({},)
      metrics_vals = [collections.defaultdict(list) for _ in self._metrics]

      # Train and append results.
      for i in range(num_iterations + 1):
        _, results = self._session.run((train_op, metrics))
        if (i % 10 == 0) or i == num_iterations:
          print("\r iteration %d: " % i + ", ".join(
                ["%s=%f" % (k, v) for r in results for k, v in r.items()]),
                end='')
          iterations.append(i)
          for metric_val, result in zip(metrics_vals, results):
            for k, v in result.items():
              metric_val[k].append(v)

      for k, v in self._embedding_vars.items():
        self._embeddings[k] = v.eval()

      if plot_results:
        # Plot the metrics.
        num_subplots = len(metrics)+1
        fig = plt.figure()
        fig.set_size_inches(num_subplots*10, 8)
        for i, metric_vals in enumerate(metrics_vals):
          ax = fig.add_subplot(1, num_subplots, i+1)
          for k, v in metric_vals.items():
            ax.plot(iterations, v, label=k)
          ax.set_xlim([1, num_iterations])
          ax.legend()
      return results
```

Exercise 3: Build a Matrix Factorization model and train it

Using your `sparse_mean_square_error` function, write a function that builds a `CFModel` by creating the embedding variables and the train and test losses.

In [30]:

```
def build_model(ratings, embedding_dim=3, init_stddev=1.):
  """
  Args:
```

```
    ratings: a DataFrame of the ratings
    embedding_dim: the dimension of the embedding vectors.
    init_stddev: float, the standard deviation of the random initial embeddings.
  Returns:
    model: a CFModel.
  """
  # Split the ratings DataFrame into train and test.
  train_ratings, test_ratings = split_dataframe(ratings)
  # SparseTensor representation of the train and test datasets.
  # ========================= Complete this section ============================
  A_train = build_rating_sparse_tensor(train_ratings)
  A_test = build_rating_sparse_tensor(test_ratings)
  # ============================================================================
  # Initialize the embeddings using a normal distribution.
  U = tf.Variable(tf.random_normal(
      [A_train.dense_shape[0], embedding_dim], stddev=init_stddev))
  V = tf.Variable(tf.random_normal(
      [A_train.dense_shape[1], embedding_dim], stddev=init_stddev))
  # ========================= Complete this section ============================
  train_loss = sparse_mean_square_error(A_train, U, V)
  test_loss = sparse_mean_square_error(A_test, U, V)
  # ============================================================================
  metrics = {
      'train_error': train_loss,
      'test_error': test_loss
  }
  embeddings = {
      "user_id": U,
      "movie_id": V
  }
  return CFModel(embeddings, train_loss, [metrics])
```

Great, now it's time to train the model!

Go ahead and run the next cell, trying different parameters (embedding dimension, learning rate, iterations). The training and test errors are plotted at the end of training. You can inspect these values to validate the hyper-parameters.

Note: by calling `model.train` again, the model will continue training starting from the current values of the embeddings.

In [31]:

```
# Build the CF model and train it.
model = build_model(ratings, embedding_dim=30, init_stddev=0.5)
```

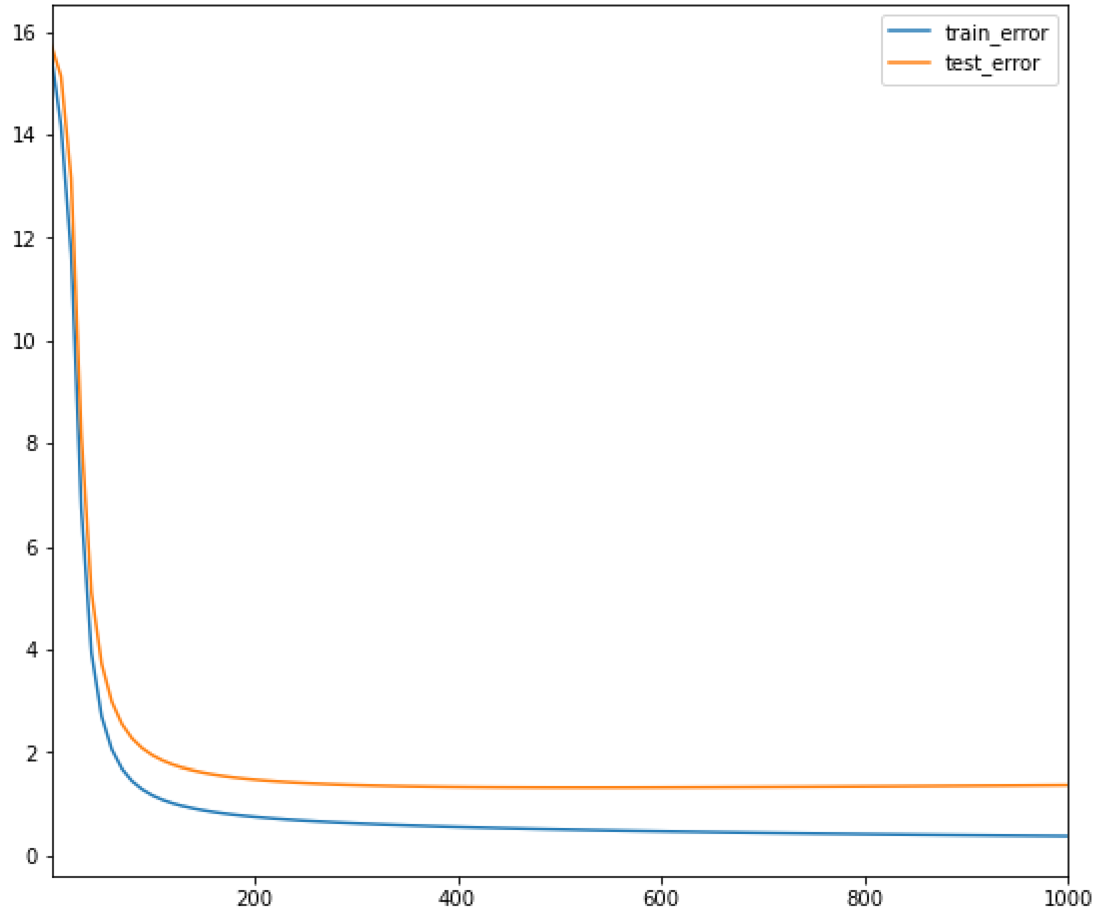
In [32]:

```
model.train(num_iterations=1000, learning_rate=10.)
```

```
2021-11-09 13:26:48.948936: I tensorflow/core/common_runtime/process_util.cc:14
6] Creating new thread pool with default inter op setting: 2. Tune using inter_o
p_parallelism_threads for best performance.
 iteration 1000: train_error=0.374785, test_error=1.366195
```

Out[32]:

```
[{'train_error': 0.3747848, 'test_error': 1.3661946}]
```



The movie and user embeddings are also displayed in the right figure. When the embedding dimension is greater than 3, the embeddings are projected on the first 3 dimensions. The next section will have a more detailed look at the embeddings.

Inspecting the Embeddings

In this section, we take a closer look at the learned embeddings, by

- computing your recommendations
- looking at the nearest neighbors of some movies,
- looking at the norms of the movie embeddings,
- visualizing the embedding in a projected embedding space.

Exercise 4: Write a function that computes the scores of the candidates

We start by writing a function that, given a query embedding $u \in \mathbb R^d$ and item embeddings $V \in \mathbb R^{N \times d}$, computes the item scores.

As discussed in the lecture, there are different similarity measures we can use, and these can yield different results. We will compare the following:

- dot product: the score of item j is $\langle u, V_j \rangle$.
- cosine: the score of item j is $\frac{\langle u, V_j \rangle}{\|u\|\|V_j\|}$.

Hints:

- you can use `np.dot` to compute the product of two np.Arrays.
- you can use `np.linalg.norm` to compute the norm of a np.Array.

In [33]:

```
DOT = 'dot'
COSINE = 'cosine'
def compute_scores(query_embedding, item_embeddings, measure=DOT):
  """Computes the scores of the candidates given a query.
  Args:
    query_embedding: a vector of shape [k], representing the query embedding.
    item_embeddings: a matrix of shape [N, k], such that row i is the embedding
      of item i.
    measure: a string specifying the similarity measure to be used. Can be
      either DOT or COSINE.
  Returns:
    scores: a vector of shape [N], such that scores[i] is the score of item i.
  """
  # ========================= Complete this section ============================
  u=query_embedding
  V= item_embeddings
  if  measure == COSINE:
    V = V / np.linalg.norm(V, axis=1, keepdims=True)
    u = u / np.linalg.norm(u)
  scores = u.dot(V.T)
  # ============================================================================
  return scores
```

Equipped with this function, we can compute recommendations, where the query embedding can be either a user embedding or a movie embedding.

In [34]:

```
def user_recommendations(model, measure=DOT, exclude_rated=False, k=6):
  if USER_RATINGS:
    scores = compute_scores(
        model.embeddings["user_id"][943], model.embeddings["movie_id"], measure)
    score_key = measure + ' score'
    df = pd.DataFrame({
        score_key: list(scores),
        'movie_id': movies['movie_id'],
        'titles': movies['title'],
        'genres': movies['all_genres'],
    })
    if exclude_rated:
      # remove movies that are already rated
      rated_movies = ratings[ratings.user_id == "943"]["movie_id"].values
      df = df[df.movie_id.apply(lambda movie_id: movie_id not in rated_movies)]
    display.display(df.sort_values([score_key], ascending=False).head(k))

def movie_neighbors(model, title_substring, measure=DOT, k=6):
  # Search for movie ids that match the given substring.
  ids =  movies[movies['title'].str.contains(title_substring)].index.values
  titles = movies.iloc[ids]['title'].values
  if len(titles) == 0:
    raise ValueError("Found no movies with title %s" % title_substring)
  print("Nearest neighbors of : %s." % titles[0])
  if len(titles) > 1:
    print("[Found more than one matching movie. Other candidates: {}]".format(
        ", ".join(titles[1:])))
```

```
    movie_id = ids[0]
    scores = compute_scores(
        model.embeddings["movie_id"][movie_id], model.embeddings["movie_id"],
        measure)
    score_key = measure + ' score'
    df = pd.DataFrame({
        score_key: list(scores),
        'titles': movies['title'],
        'genres': movies['all_genres']
    })
    display.display(df.sort_values([score_key], ascending=False).head(k))
```

Movie Nearest neighbors

Let's look at the neareast neighbors for some of the movies.

In [35]:

```
movie_neighbors(model, "Aladdin", DOT)
movie_neighbors(model, "Aladdin", COSINE)
```

```
Nearest neighbors of : Aladdin (1992).
[Found more than one matching movie. Other candidates: Aladdin and the King of T
hieves (1996)]
```

	dot score	titles	genres
94	6.135	Aladdin (1992)	Animation-Children-Comedy-Musical
142	5.973	Sound of Music, The (1965)	Musical
908	5.403	Dangerous Beauty (1998)	Drama
81	5.303	Jurassic Park (1993)	Action-Adventure-Sci-Fi
619	5.227	Chamber, The (1996)	Drama
567	5.167	Speed (1994)	Action-Romance-Thriller

```
Nearest neighbors of : Aladdin (1992).
[Found more than one matching movie. Other candidates: Aladdin and the King of T
hieves (1996)]
```

	cosine score	titles	genres
94	1.000	Aladdin (1992)	Animation-Children-Comedy-Musical
27	0.810	Apollo 13 (1995)	Action-Drama-Thriller
70	0.810	Lion King, The (1994)	Animation-Children-Musical
0	0.799	Toy Story (1995)	Animation-Children-Comedy
619	0.796	Chamber, The (1996)	Drama
81	0.790	Jurassic Park (1993)	Action-Adventure-Sci-Fi

It seems that the quality of learned embeddings may not be very good. Can you think of potential techniques that could be used to improve them? We can start by inspecting the embeddings.

Movie Embedding Norm

We can also observe that the recommendations with dot-product and cosine are different: with dot-product, the model tends to recommend popular movies. This can be explained by the fact that in matrix factorization models, the norm of the embedding is often correlated with popularity (popular movies have a larger norm), which makes it more likely to recommend more popular items. We can confirm this hypothesis by sorting the movies by their embedding norm, as done in the next cell.

In [36]:

```
def movie_embedding_norm(models):
  """Visualizes the norm and number of ratings of the movie embeddings.
  Args:
    model: A MFModel object.
  """
  if not isinstance(models, list):
    models = [models]
  df = pd.DataFrame({
      'title': movies['title'],
      'genre': movies['genre'],
      'num_ratings': movies_ratings['rating count'],
  })
  charts = []
  brush = alt.selection_interval()
  for i, model in enumerate(models):
    norm_key = 'norm'+str(i)
    df[norm_key] = np.linalg.norm(model.embeddings["movie_id"], axis=1)
    nearest = alt.selection(
        type='single', encodings=['x', 'y'], on='mouseover', nearest=True,
        empty='none')
    base = alt.Chart().mark_circle().encode(
        x='num_ratings',
        y=norm_key,
        color=alt.condition(brush, alt.value('#4c78a8'), alt.value('lightgray'))
    ).properties(
        selection=nearest).add_selection(brush)
    text = alt.Chart().mark_text(align='center', dx=5, dy=-5).encode(
        x='num_ratings', y=norm_key,
        text=alt.condition(nearest, 'title', alt.value('')))
    charts.append(alt.layer(base, text))
  return alt.hconcat(*charts, data=df)

def visualize_movie_embeddings(data, x, y):
  nearest = alt.selection(
      type='single', encodings=['x', 'y'], on='mouseover', nearest=True,
      empty='none')
  base = alt.Chart().mark_circle().encode(
      x=x,
      y=y,
      color=alt.condition(genre_filter, "genre", alt.value("whitesmoke")),
  ).properties(
      width=600,
      height=600,
      selection=nearest)
  text = alt.Chart().mark_text(align='left', dx=5, dy=-5).encode(
      x=x,
      y=y,
      text=alt.condition(nearest, 'title', alt.value('')))
  return alt.hconcat(alt.layer(base, text), genre_chart, data=data)

def tsne_movie_embeddings(model):
```

```
    """Visualizes the movie embeddings, projected using t-SNE with Cosine measure.
    Args:
      model: A MFModel object.
    """
    tsne = sklearn.manifold.TSNE(
        n_components=2, perplexity=40, metric='cosine', early_exaggeration=10.0,
        init='pca', verbose=True, n_iter=400)

    print('Running t-SNE...')
    V_proj = tsne.fit_transform(model.embeddings["movie_id"])
    movies.loc[:,'x'] = V_proj[:, 0]
    movies.loc[:,'y'] = V_proj[:, 1]
    return visualize_movie_embeddings(movies, 'x', 'y')
```

In [37]:

```
movie_embedding_norm(model)
```

Out[37]:

Note: Depending on how the model is initialized, you may observe that some niche movies (ones with few ratings) have a high norm, leading to spurious recommendations. This can happen if the embedding of that movie happens to be initialized with a high norm. Then, because the movie has few ratings, it is infrequently updated, and can keep its high norm. This can be alleviated by using regularization.

Try changing the value of the hyperparameter `init_stddev`. One quantity that can be helpful is that the expected norm of a d-dimensional vector with entries $\sim \mathcal N(0, \sigma^2)$ is approximatley $\sigma \sqrt d$.

How does this affect the embedding norm distribution, and the ranking of the top-norm movies?

In [41]:

```
#@title Solution
model_lowinit = build_model(ratings, embedding_dim=30, init_stddev=0.06)
model_lowinit.train(num_iterations=1000, learning_rate=10.)
movie_neighbors(model_lowinit, "Aladdin", DOT)
movie_neighbors(model_lowinit, "Aladdin", COSINE)
movie_embedding_norm([model, model_lowinit])
```

```
 iteration 1000: train_error=0.351465, test_error=0.976743Nearest neighbors of :
Aladdin (1992).
[Found more than one matching movie. Other candidates: Aladdin and the King of T
hieves (1996)]
```

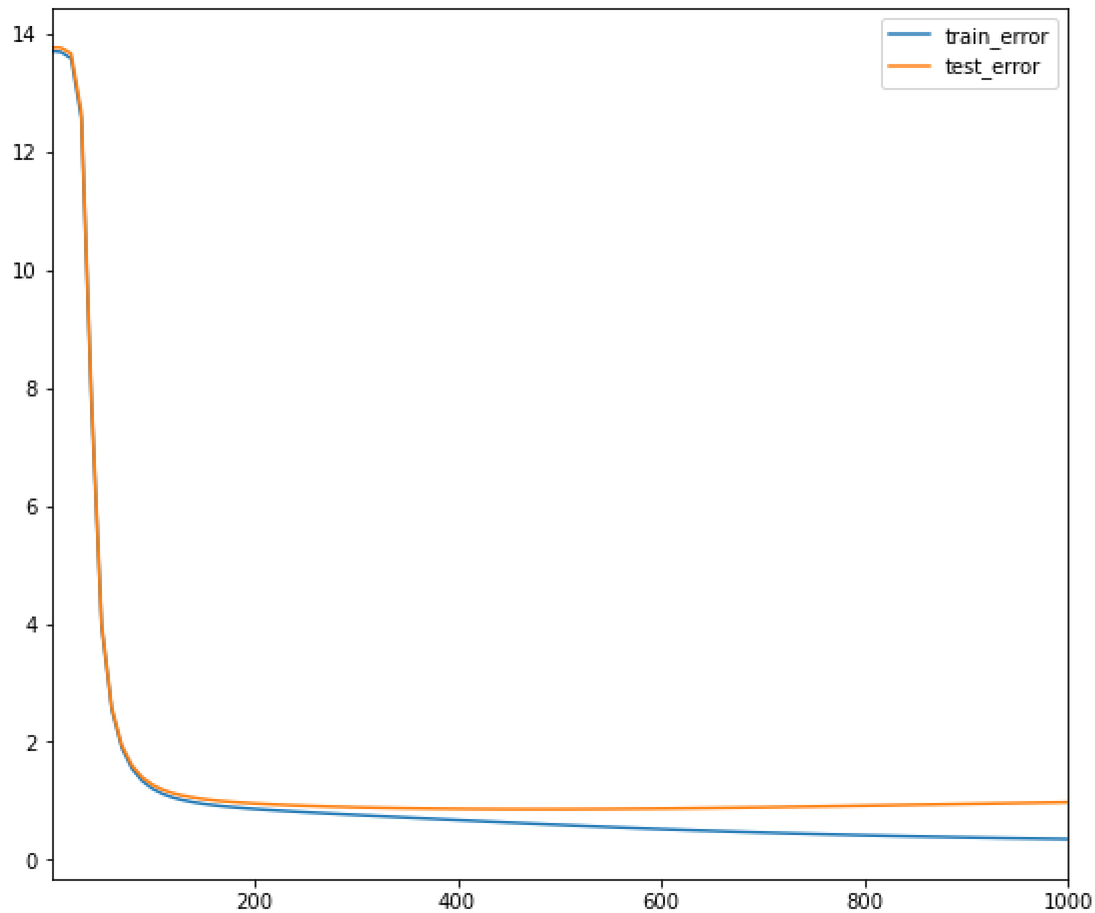
	dot score	titles	genres
94	5.429	Aladdin (1992)	Animation-Children-Comedy-Musical
173	4.979	Raiders of the Lost Ark (1981)	Action-Adventure
257	4.909	Contact (1997)	Drama-Sci-Fi
21	4.636	Braveheart (1995)	Action-Drama-War
49	4.617	Star Wars (1977)	Action-Adventure-Romance-Sci-Fi-War
87	4.599	Sleepless in Seattle (1993)	Comedy-Romance

```
Nearest neighbors of : Aladdin (1992).
[Found more than one matching movie. Other candidates: Aladdin and the King of T
```

```
hieves (1996)]
```

	cosine score	titles	genres
94	1.000	Aladdin (1992)	Animation-Children-Comedy-Musical
1517	0.873	Losing Isaiah (1995)	Drama
1132	0.851	Escape to Witch Mountain (1975)	Adventure-Children-Fantasy
1671	0.851	Kika (1993)	Drama
1145	0.834	Calendar Girl (1993)	Drama
138	0.832	Love Bug, The (1969)	Children-Comedy

Out[41]:



Embedding visualization

Since it is hard to visualize embeddings in a higher-dimensional space (when the embedding dimension $k > 3$), one approach is to project the embeddings to a lower dimensional space. T-SNE (T-distributed Stochastic Neighbor Embedding) is an algorithm that projects the embeddings while attempting to preserve their pariwise distances. It can be useful for visualization, but one should use it with care. For more information on using t-SNE, see How to Use t-SNE Effectively.

In [42]:

```
tsne_movie_embeddings(model_lowinit)
```

```
Running t-SNE...
[t-SNE] Computing 121 nearest neighbors...
[t-SNE] Indexed 1682 samples in 0.001s...
[t-SNE] Computed neighbors for 1682 samples in 0.127s...
[t-SNE] Computed conditional probabilities for sample 1000 / 1682
/opt/conda/lib/python3.7/site-packages/sklearn/manifold/_t_sne.py:793: FutureWar
ning: The default learning rate in TSNE will change from 200.0 to 'auto' in 1.2.
  FutureWarning,
/opt/conda/lib/python3.7/site-packages/sklearn/manifold/_t_sne.py:827: FutureWar
ning: 'square_distances' has been introduced in 0.24 to help phase out legacy sq
uaring behavior. The 'legacy' setting will be removed in 1.1 (renaming of 0.26),
and the default setting will be changed to True. In 1.3, 'square_distances' will
be removed altogether, and distances will be squared by default. Set 'square_dis
tances'=True to silence this warning.
  FutureWarning,
[t-SNE] Computed conditional probabilities for sample 1682 / 1682
[t-SNE] Mean sigma: 0.124110
/opt/conda/lib/python3.7/site-packages/sklearn/manifold/_t_sne.py:986: FutureWar
ning: The PCA initialization in TSNE will change to have the standard deviation
of PC1 equal to 1e-4 in 1.2. This will ensure better convergence.
  FutureWarning,
[t-SNE] KL divergence after 250 iterations with early exaggeration: 58.031914
[t-SNE] KL divergence after 400 iterations: 2.243138
```

Out[42]:

You can highlight the embeddings of a given genre by clicking on the genres panel (SHIFT+click to select multiple genres).

We can observe that the embeddings do not seem to have any notable structure, and the embeddings of a given genre are located all over the embedding space. This confirms the poor quality of the learned embeddings. One of the main reasons is that we only trained the model on observed pairs, and without regularization.

Softmax model

In this section, we will train a simple softmax model that predicts whether a given user has rated a movie.

The model will take as input a feature vector x representing the list of movies the user has rated. We start from the ratings DataFrame, which we group by user_id.

In [43]:

```
rated_movies = (ratings[["user_id", "movie_id"]]
                .groupby("user_id", as_index=False)
                .aggregate(lambda x: list(x)))
rated_movies.head()
```

Out[43]:

	user_id	movie_id
0	0	[60, 188, 32, 159, 19, 201, 170, 264, 154, 116...
1	1	[291, 250, 49, 313, 296, 289, 311, 280, 12, 27...
2	10	[110, 557, 731, 226, 424, 739, 722, 37, 724, 1...

	user_id	movie_id
3	100	[828, 303, 595, 221, 470, 404, 280, 251, 281, ...
4	101	[767, 822, 69, 514, 523, 321, 624, 160, 447, 4...

We then create a function that generates an example batch, such that each example contains the following features:

- movie_id: A tensor of strings of the movie ids that the user rated.
- genre: A tensor of strings of the genres of those movies
- year: A tensor of strings of the release year.

In [44]:

```
years_dict = {
    movie: year for movie, year in zip(movies["movie_id"], movies["year"])
}
genres_dict = {
    movie: genres.split('-')
    for movie, genres in zip(movies["movie_id"], movies["all_genres"])
}

def make_batch(ratings, batch_size):
  """Creates a batch of examples.
  Args:
    ratings: A DataFrame of ratings such that examples["movie_id"] is a list of
      movies rated by a user.
    batch_size: The batch size.
  """
  def pad(x, fill):
    return pd.DataFrame.from_dict(x).fillna(fill).values

  movie = []
  year = []
  genre = []
  label = []
  for movie_ids in ratings["movie_id"].values:
    movie.append(movie_ids)
    genre.append([x for movie_id in movie_ids for x in genres_dict[movie_id]])
    year.append([years_dict[movie_id] for movie_id in movie_ids])
    label.append([int(movie_id) for movie_id in movie_ids])
  features = {
      "movie_id": pad(movie, ""),
      "year": pad(year, ""),
      "genre": pad(genre, ""),
      "label": pad(label, -1)
  }
  batch = (
      tf.data.Dataset.from_tensor_slices(features)
      .shuffle(1000)
      .repeat()
      .batch(batch_size)
      .make_one_shot_iterator()
      .get_next())
  return batch

def select_random(x):
  """Selectes a random elements from each row of x."""
  def to_float(x):
```

```
    return tf.cast(x, tf.float32)
  def to_int(x):
    return tf.cast(x, tf.int64)
  batch_size = tf.shape(x)[0]
  rn = tf.range(batch_size)
  nnz = to_float(tf.count_nonzero(x >= 0, axis=1))
  rnd = tf.random_uniform([batch_size])
  ids = tf.stack([to_int(rn), to_int(nnz * rnd)], axis=1)
  return to_int(tf.gather_nd(x, ids))
```

Loss function

Recall that the softmax model maps the input features x to a user embedding $\psi(x) \in \mathbb R^d$, where d is the embedding dimension. This vector is then multiplied by a movie embedding matrix $V \in \mathbb R^{m \times d}$ (where m is the number of movies), and the final output of the model is the softmax of the product $$ \hat p(x) = \text{softmax}(\psi(x) V^\top). $$ Given a target label y, if we denote by $p = 1_y$ a one-hot encoding of this target label, then the loss is the cross-entropy between $\hat p(x)$ and p.

Exercise 5: Write a loss function for the softmax model.

In this exercise, we will write a function that takes tensors representing the user embeddings $\psi(x)$, movie embeddings V, target label y, and return the cross-entropy loss.

Hint: You can use the function `tf.nn.sparse_softmax_cross_entropy_with_logits` , which takes `logits` as input, where `logits` refers to the product $\psi(x) V^\top$.

In [49]:

```
def softmax_loss(user_embeddings, movie_embeddings, labels):
  """Returns the cross-entropy loss of the softmax model.
  Args:
    user_embeddings: A tensor of shape [batch_size, embedding_dim].
    movie_embeddings: A tensor of shape [num_movies, embedding_dim].
    labels: A sparse tensor of dense_shape [batch_size, 1], such that
      labels[i] is the target label for example i.
  Returns:
    The mean cross-entropy loss.
  """
  # ========================= Complete this section ============================
  logits = tf.matmul(user_embeddings, movie_embeddings, transpose_b=True)
  loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
      logits=logits, labels=labels))
  # ============================================================================
  return loss
```

Exercise 6: Build a softmax model, train it, and inspect its embeddings.

We are now ready to build a softmax CFModel. Complete the `build_softmax_model` function in the next cell. The architecture of the model is defined in the function `create_user_embeddings` and illustrated in the figure below. The input embeddings (movie_id, genre and year) are concatenated to form the input layer, then we have hidden layers with dimensions specified by the `hidden_dims` argument. Finally, the last hidden layer is multiplied by the movie embeddings to obtain the logits layer. For the target label, we will use a randomly-sampled movie_id from the list of movies the user rated.

Softmax model

Complete the function below by creating the feature columns and embedding columns, then creating the loss tensors both for the train and test sets (using the `softmax_loss` function of the previous exercise).

In [50]:

```
def build_softmax_model(rated_movies, embedding_cols, hidden_dims):
  """Builds a Softmax model for MovieLens.
  Args:
    rated_movies: DataFrame of traing examples.
    embedding_cols: A dictionary mapping feature names (string) to embedding
      column objects. This will be used in tf.feature_column.input_layer() to
      create the input layer.
    hidden_dims: int list of the dimensions of the hidden layers.
  Returns:
    A CFModel object.
  """
  def create_network(features):
    """Maps input features dictionary to user embeddings.
    Args:
      features: A dictionary of input string tensors.
    Returns:
      outputs: A tensor of shape [batch_size, embedding_dim].
    """
    # Create a bag-of-words embedding for each sparse feature.
    inputs = tf.feature_column.input_layer(features, embedding_cols)
    # Hidden layers.
    input_dim = inputs.shape[1].value
    for i, output_dim in enumerate(hidden_dims):
      w = tf.get_variable(
          "hidden%d_w_" % i, shape=[input_dim, output_dim],
          initializer=tf.truncated_normal_initializer(
              stddev=1./np.sqrt(output_dim))) / 10.
      outputs = tf.matmul(inputs, w)
      input_dim = output_dim
      inputs = outputs
    return outputs

  train_rated_movies, test_rated_movies = split_dataframe(rated_movies)
  train_batch = make_batch(train_rated_movies, 200)
  test_batch = make_batch(test_rated_movies, 100)

  with tf.variable_scope("model", reuse=False):
    # Train
    train_user_embeddings = create_network(train_batch)
    train_labels = select_random(train_batch["label"])
  with tf.variable_scope("model", reuse=True):
    # Test
    test_user_embeddings = create_network(test_batch)
    test_labels = select_random(test_batch["label"])
    movie_embeddings = tf.get_variable(
        "input_layer/movie_id_embedding/embedding_weights")

  # ========================= Complete this section ============================
  test_loss = softmax_loss(
```

```
      test_user_embeddings, movie_embeddings, test_labels)
  train_loss = softmax_loss(
      train_user_embeddings, movie_embeddings, train_labels)
  _, test_precision_at_10 = tf.metrics.precision_at_k(
      labels=test_labels,
      predictions=tf.matmul(test_user_embeddings, movie_embeddings, transpose_b=
      k=10)
  # ============================================================================

  metrics = (
      {"train_loss": train_loss, "test_loss": test_loss},
      {"test_precision_at_10": test_precision_at_10}
  )
  embeddings = {"movie_id": movie_embeddings}
  return CFModel(embeddings, train_loss, metrics)
```

Train the Softmax model

We are now ready to train the softmax model. You can set the following hyperparameters:

- learning rate
- number of iterations. Note: you can run `softmax_model.train()` again to continue training the model from its current state.
- input embedding dimensions (the `input_dims` argument)
- number of hidden layers and size of each layer (the `hidden_dims` argument)

Note: since our input features are string-valued (movie_id, genre, and year), we need to map them to integer ids. This is done using `tf.feature_column.categorical_column_with_vocabulary_list`, which takes a vocabulary list specifying all the values the feature can take. Then each id is mapped to an embedding vector using `tf.feature_column.embedding_column`.

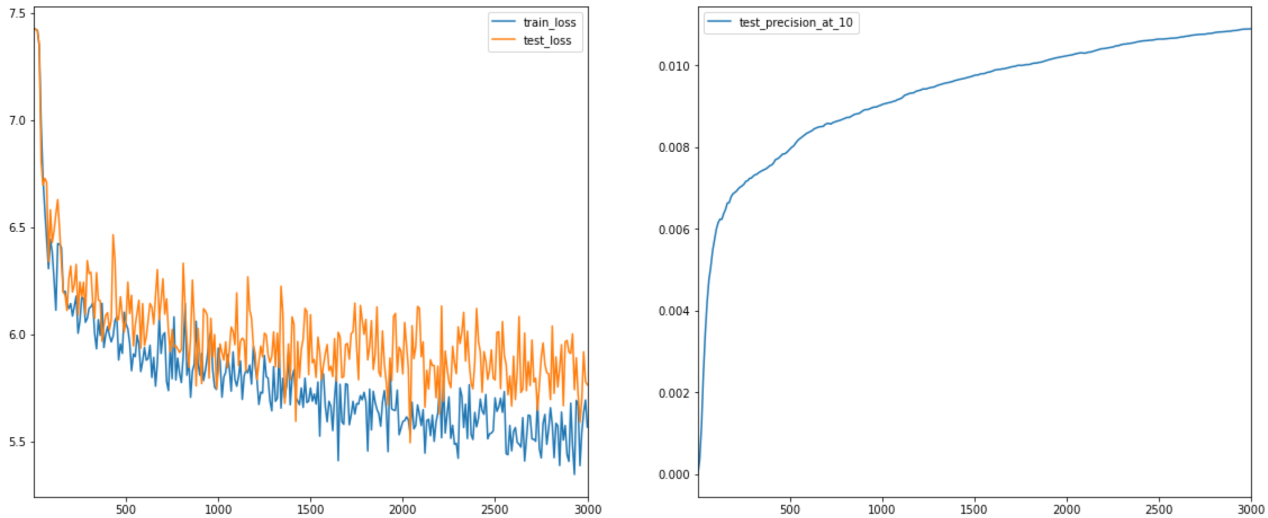
In [51]:

```
# Create feature embedding columns
def make_embedding_col(key, embedding_dim):
  categorical_col = tf.feature_column.categorical_column_with_vocabulary_list(
      key=key, vocabulary_list=list(set(movies[key].values)), num_oov_buckets=0)
  return tf.feature_column.embedding_column(
      categorical_column=categorical_col, dimension=embedding_dim,
      # default initializer: trancated normal with stddev=1/sqrt(dimension)
      combiner='mean')

with tf.Graph().as_default():
  softmax_model = build_softmax_model(
      rated_movies,
      embedding_cols=[
          make_embedding_col("movie_id", 35),
          make_embedding_col("genre", 3),
          make_embedding_col("year", 2),
      ],
      hidden_dims=[35])

softmax_model.train(
    learning_rate=8., num_iterations=3000, optimizer=tf.train.AdagradOptimizer)
```

```
WARNING:tensorflow:From /opt/conda/lib/python3.7/site-packages/tensorflow/pytho
```

```
n/training/adagrad.py:77: calling Constant.__init__ (from tensorflow.python.ops.
init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the c
onstructor
 iteration 3000: train_loss=5.567315, test_loss=5.764771, test_precision_at_10=
0.010898
```

Out[51]:
```
({'train_loss': 5.5673146, 'test_loss': 5.764771},
 {'test_precision_at_10': 0.010898033988670444})
```



Inspect the embeddings

We can inspect the movie embeddings as we did for the previous models. Note that in this case, the movie embeddings are used at the same time as input embeddings (for the bag of words representation of the user history), and as softmax weights.

In [52]:
```
movie_neighbors(softmax_model, "Aladdin", DOT)
movie_neighbors(softmax_model, "Aladdin", COSINE)
```

```
Nearest neighbors of : Aladdin (1992).
[Found more than one matching movie. Other candidates: Aladdin and the King of T
hieves (1996)]
```

	dot score	titles	genres
94	24.390	Aladdin (1992)	Animation-Children-Comedy-Musical
49	21.160	Star Wars (1977)	Action-Adventure-Romance-Sci-Fi-War
172	20.512	Princess Bride, The (1987)	Action-Adventure-Comedy-Romance
143	19.767	Die Hard (1988)	Action-Thriller
120	19.607	Independence Day (ID4) (1996)	Action-Sci-Fi-War
0	19.134	Toy Story (1995)	Animation-Children-Comedy

```
Nearest neighbors of : Aladdin (1992).
[Found more than one matching movie. Other candidates: Aladdin and the King of T
hieves (1996)]
```

	cosine score	titles	genres
94	1.000	Aladdin (1992)	Animation-Children-Comedy-Musical

	cosine score	titles	genres
70	0.788	Lion King, The (1994)	Animation-Children-Musical
81	0.751	Jurassic Park (1993)	Action-Adventure-Sci-Fi
172	0.733	Princess Bride, The (1987)	Action-Adventure-Comedy-Romance
7	0.730	Babe (1995)	Children-Comedy-Drama
167	0.722	Monty Python and the Holy Grail (1974)	Comedy

In [53]:

```
movie_embedding_norm(softmax_model)
```

Out[53]:

In [54]:

```
tsne_movie_embeddings(softmax_model)
```

```
Running t-SNE...
[t-SNE] Computing 121 nearest neighbors...
[t-SNE] Indexed 1682 samples in 0.001s...
[t-SNE] Computed neighbors for 1682 samples in 0.104s...
[t-SNE] Computed conditional probabilities for sample 1000 / 1682
/opt/conda/lib/python3.7/site-packages/sklearn/manifold/_t_sne.py:793: FutureWar
ning: The default learning rate in TSNE will change from 200.0 to 'auto' in 1.2.
  FutureWarning,
/opt/conda/lib/python3.7/site-packages/sklearn/manifold/_t_sne.py:827: FutureWar
ning: 'square_distances' has been introduced in 0.24 to help phase out legacy sq
uaring behavior. The 'legacy' setting will be removed in 1.1 (renaming of 0.26),
and the default setting will be changed to True. In 1.3, 'square_distances' will
be removed altogether, and distances will be squared by default. Set 'square_dis
tances'=True to silence this warning.
  FutureWarning,
[t-SNE] Computed conditional probabilities for sample 1682 / 1682
[t-SNE] Mean sigma: 0.189549
/opt/conda/lib/python3.7/site-packages/sklearn/manifold/_t_sne.py:986: FutureWar
ning: The PCA initialization in TSNE will change to have the standard deviation
of PC1 equal to 1e-4 in 1.2. This will ensure better convergence.
  FutureWarning,
[t-SNE] KL divergence after 250 iterations with early exaggeration: 53.194855
[t-SNE] KL divergence after 400 iterations: 1.289873
```

Out[54]:

Congratulations!

You have completed this lab.

If you would like to further explore these models, we encourage you to try different hyperparameters and observe how this affects the quality of the model and the structure of the embedding space. Here are some suggestions:

- Change the embedding dimension.
- In the softmax model: change the number of hidden layers, and the input features. For example, you can try a model with no hidden layers, and only the movie ids as inputs.

- Using other similarity measures: In this notebook, we used dot product $d(u, V_j) = \langle u, V_j \rangle$ and cosine $d(u, V_j) = \frac{\langle u, V_j \rangle}{\|u\|\|V_j\|}$, and discussed how the norms of the embeddings affect the recommendations. You can also try other variants which apply a transformation to the norm, for example $d(u, V_j) = \frac{\langle u, V_j \rangle}{\|V_j\|^\alpha}$.

Challenge

With everything you learned during the Advanced Machine Learning on Google Cloud, can you try and push the model to the AI Platform for predictions?