Neural Network Recipe Recommendation System with Embeddings for Dietary Restriction

Business Problem:

It can be hard to continuously come up with new and interesting recipes to cook, especially if you have certain dietary restrictions. Many people use websites such as food.com to find, try, and rate recipes. From user and recipe data from Food.com, can we provide users with recommendations for the next recipes that users should try, taking into account their dietary specifications?

Pre-Processing and Data Exploration:

Loading the data and necessary packages:

```
# TF's recommender imports
!pip install -q tensorflow-recommenders
!pip install -q tensorflow_ranking
!pip install -q --upgrade tensorflow-datasets
!pip install -q scann
```

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 475.2/475.2 MB 2.9 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5.5/5.5 MB 102.7 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 442.0/442.0 kB 40.9 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.7/1.7 MB 78.2 MB/s eta 0:00:00
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is
scann 1.2.10 requires tensorflow~=2.13.0, but you have tensorflow 2.15.0.post1 which is incompatible.
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is
tensorflow-serving-api 2.14.1 requires tensorflow<3,>=2.14.1, but you have tensorflow 2.13.1 which is incompatible.
```

```
import tensorflow as tf
print("TensorFlow version:", tf.__version__)
```

```
TensorFlow version: 2.13.1
```

```
import pandas as pd
import nltk
nltk.download('punkt')
from nltk.corpus import stopwords
import re
import numpy as np
import pickle
from nltk.tokenize import RegexpTokenizer, word_tokenize
import io
from collections import defaultdict
import os
import pprint
import tempfile
from typing import Dict, Text
import numpy as np
import tensorflow_datasets as tfds
import tensorflow_recommenders as tfrs
import tensorflow_ranking as tfr
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=Tr
```

Getting the data from kaggle:

```
 #upload the kaggle.json file to load kaggle data
```

```
from google.colab import files
files.upload()
```

```
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
```

```
!pip install -q kaggle

# This permissions change avoids a warning on Kaggle tool startup.
!chmod 600 ~/.kaggle/kaggle.json
```

```
#download the data from kaggle
!kaggle datasets download -d shuyangli94/food-com-recipes-and-user-interactions
```

```
#unzip data from kaggle
import zipfile

# Define the path to your zip file
file_path = '/content/drive/MyDrive/Capstone/capstone_data/food-com-recipes-and-user-interactions.zip'
# Unzip the file to a specific destination
with zipfile.ZipFile(file_path, 'r') as zip_ref:
    zip_ref.extractall('/content/drive/MyDrive/Capstone/capstone_data')

```

Loading the user and item data:

```
#read in the specific datasets to be used:
user_data = pd.read_csv('/content/drive/MyDrive/Capstone/capstone_data/RAW_interactions.csv')
recipe_data = pd.read_csv('/content/drive/MyDrive/Capstone/capstone_data/RAW_recipes.csv')
```

Data Exploration

Viewing the data to gain understanding of its contents:

```
user_data.head()
```

	user_id	recipe_id	date	rating	review
0	38094	40893	2003-02-17	4	Great with a salad. Cooked on top of stove for...
1	1293707	40893	2011-12-21	5	So simple, so delicious! Great for chilly fall...
2	8937	44394	2002-12-01	4	This worked very well and is EASY. I used not...
3	126440	85009	2010-02-27	5	I made the Mexican topping and took it to bunk...
4	57222	85009	2011-10-01	5	Made the cheddar bacon topping, adding a sprin...

```
user_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1132367 entries, 0 to 1132366
Data columns (total 5 columns):
 #   Column     Non-Null Count    Dtype
---  ------     --------------    -----
 0   user_id    1132367 non-null  int64
 1   recipe_id  1132367 non-null  int64
 2   date       1132367 non-null  object
 3   rating     1132367 non-null  int64
 4   review     1132198 non-null  object
dtypes: int64(3), object(2)
memory usage: 43.2+ MB
```

```
recipe_data.head()
```

	name	id	minutes	contributor_id	submitted	tags	nutrition	n_steps	steps	description	ingredients	n_i
0	arriba baked winter squash mexican style	137739	55	47892	2005-09-16	['60-minutes-or-less', 'time-to-make', 'course...	[51.5, 0.0, 13.0, 0.0, 2.0, 0.0, 4.0]	11	['make a choice and proceed with recipe', 'dep...	autumn is my favorite time of year to cook! th...	['winter squash', 'mexican seasoning', 'mixed ...	
1	a bit different breakfast pizza	31490	30	26278	2002-06-17	['30-minutes-or-less', 'time-to-make', 'course...	[173.4, 18.0, 0.0, 17.0, 22.0, 35.0, 1.0]	9	['preheat oven to 425 degrees f', 'press dough...	this recipe calls for the crust to be prebaked...	['prepared pizza crust', 'sausage patty', 'egg...	
2	all in the kitchen chili	112140	130	196586	2005-02-25	['time-to-make', 'course', 'preparation', 'mai...	[269.8, 22.0, 32.0, 48.0, 39.0, 27.0, 5.0]	6	['brown ground beef in large pot', 'add choppe...	this modified version of 'mom's' chili was a h...	['ground beef', 'yellow onions', 'diced tomato...	
3	alouette potatoes	59389	45	68585	2003-04-14	['60-minutes-or-less', 'time-to-make', 'course...	[368.1, 17.0, 10.0, 2.0, 14.0, 8.0, 20.0]	11	['place potatoes in a large pot of lightly sal...	this is a super easy, great tasting, make ahea...	['spreadable cheese with garlic and herbs', 'n...	
	amish					['weeknight',	[352.9, 1.0		['mix all	my dh's amish	['tomato juice'	

```
recipe_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 231637 entries, 0 to 231636
Data columns (total 12 columns):
 #   Column          Non-Null Count   Dtype 
---  ------          --------------   ----- 
 0   name            231636 non-null  object
 1   id              231637 non-null  int64 
 2   minutes         231637 non-null  int64 
 3   contributor_id  231637 non-null  int64 
 4   submitted       231637 non-null  object
 5   tags            231637 non-null  object
 6   nutrition       231637 non-null  object
 7   n_steps         231637 non-null  int64 
 8   steps           231637 non-null  object
 9   description     226658 non-null  object
 10  ingredients     231637 non-null  object
 11  n_ingredients   231637 non-null  int64 
dtypes: int64(5), object(7)
memory usage: 21.2+ MB
```

Renaming the 'id' column in the recipe dataframe to match the recipe_id column in the user datatframe:

```
recipe_data = recipe_data.rename(columns={"id": "recipe_id"})
```

Investigating the total number of unique users and recipes in the data:

```
print(len(user_data['user_id'].unique()))
print(len(recipe_data['recipe_id'].unique()))
```

```
226570
231637
```

With 226,570 users and 231,637 recipes, there are less users than there are recipes.

Data Preparation

Dropping the columns I know I won't be working with:

```
user_recipe_ratings = user_data.drop(columns=['review', 'date'])
```

```
recipe_data = recipe_data.drop(columns=['contributor_id', 'submitted', 'nutrition', 'steps', 'minutes', 'n_steps', 'n_ingredi

```

```
# Making sure text features are strings so that they can be cleaned properly

recipe_data['tags'] = recipe_data['tags'].astype(str)
```

```
# Creating a function to perform cleaning steps at once (Removes numbers and unnecessary characters, makes all letters lowerc
nltk.download('stopwords')
stopwords_list = stopwords.words('english')

no_bad_chars = re.compile('[!\"#$%&()*+-./:;<=>?@[\]^_`{|}~\n - ]')
no_nums = re.compile('[\d-]')

def clean_text(text):
    text = no_nums.sub('', text)
    text = no_bad_chars.sub(' ', text)
    text = text.lower()
    text = ' '.join(word for word in text.split() if word not in stopwords_list)
    return text
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```
 #Applying text cleaning function to text columns
 recipe_data_cleaned = recipe_data.copy()
 recipe_data['name'] = recipe_data['name'].astype(str)
 recipe_data_cleaned['name'] = (recipe_data['name']).apply(clean_text)
 recipe_data['tags'] = recipe_data['tags'].astype(str)
 recipe_data_cleaned['tags'] = (recipe_data['tags']).apply(clean_text)
 recipe_data_cleaned.head()
```

	name	recipe_id	tags	description	ingredients
0	arriba baked winter squash mexican style	137739	'minutesorless' 'timetomake' 'course' 'maining...	autumn is my favorite time of year to cook! th...	['winter squash', 'mexican seasoning', 'mixed ...
1	a bit different breakfast pizza	31490	'minutesorless' 'timetomake' 'course' 'maining...	this recipe calls for the crust to be prebaked...	['prepared pizza crust', 'sausage patty', 'egg...
2	all in the kitchen chili	112140	'timetomake' 'course' 'preparation' 'maindish'...	this modified version of 'mom's' chili was a h...	['ground beef', 'yellow onions', 'diced tomato...
3	alouette potatoes	59389	'minutesorless' 'timetomake' 'course' 'maining...	this is a super easy, great tasting, make ahea...	['spreadable cheese with garlic and herbs', 'n...
4	amish tomato ketchup for canning	44061	'weeknight' 'timetomake' 'course' 'mainingredi...	my dh's amish mother raised him on this recipe...	['tomato juice', 'apple cider vinegar', 'sugar...

Feature Engineering to categorize each recipe as different diet types (Vegetarian, Vegan, and/or Gluten-Free):

```
# creating a new column to classify recipes as gluten-free or not
GF = []
#tags column contains the most information on gluten-free recipes
#(see miscellaneous notebook)
for row in recipe_data_cleaned['tags']:
    if "gluten-free" in row : GF.append("Gluten-Free")
    elif "gluten free" in row : GF.append("Gluten-Free")
    else: GF.append("None")
```

```
recipe_data_cleaned['GF'] = GF
```

```
#Ingredient lists for diet filtering:
vegan = ['ham', 'beef', 'meat', 'chicken', 'pork', 'bacon', 'sausage', 'lamb', 'veal', 'turkey', 'steak', 'rib', 'frankfurter

vegetarian = ['ham', 'beef', 'meat', 'chicken', 'pork', 'bacon', 'sausage', 'lamb', 'veal', 'turkey', 'steak', 'rib', 'frankf
```

```
# creating two new columns to classify recipes as vegetarian or not
# and as vegan or not:

recipe_data_cleaned['vegetarian'] = None
recipe_data_cleaned['vegan'] = None
```

```
# Filtering through the 'ingredients' column for ingredients that
# aren't vegetarian or vegan
vege_pattern = '|'.join(vegetarian)
vegan_pattern = '|'.join(vegan)


recipe_data_cleaned.vegetarian = recipe_data_cleaned.ingredients.str.contains(vege_pattern)
recipe_data_cleaned.vegan = recipe_data_cleaned.ingredients.str.contains(vegan_pattern)
```

```
# Changing Boolean values to words to indicate the diet-type
recipe_data_cleaned['vegetarian'] = recipe_data_cleaned['vegetarian'].astype(str)
recipe_data_cleaned['vegetarian'] = recipe_data_cleaned['vegetarian'].replace({'False': 'Vegetarian', 'True': 'None'})
```

```
recipe_data_cleaned['vegan'] = recipe_data_cleaned['vegan'].astype(str)
recipe_data_cleaned['vegan'] = recipe_data_cleaned['vegan'].replace({'False': 'Vegan', 'True': 'None'})
```

```
#making one column of the diet types of each recipe combined and dropping the individual columns
recipe_data_cleaned['diets_combined'] = recipe_data_cleaned[['vegetarian', 'vegan', 'GF']].values.tolist()
recipe_data_cleaned = recipe_data_cleaned.drop(columns=['GF', 'vegetarian', 'vegan'])
recipe_data_cleaned.head()
```

	name	recipe_id	tags	description	ingredients	diets_combined
0	arriba baked winter squash mexican style	137739	'minutesorless' 'timetomake' 'course' 'maining...	autumn is my favorite time of year to cook! th...	['winter squash', 'mexican seasoning', 'mixed ...	[Vegetarian, None, None]
1	a bit different breakfast pizza	31490	'minutesorless' 'timetomake' 'course' 'maining...	this recipe calls for the crust to be prebaked...	['prepared pizza crust', 'sausage patty', 'egg...	[None, None, None]
2	all in the kitchen chili	112140	'timetomake' 'course' 'preparation' 'maindish'...	this modified version of 'mom's' chili was a h...	['ground beef', 'yellow onions', 'diced tomato...	[None, None, None]
3	alouette potatoes	59389	'minutesorless' 'timetomake' 'course' 'maining...	this is a super easy, great tasting, make ahea...	['spreadable cheese with garlic and herbs', 'n...	[Vegetarian, None, None]
4	amish tomato ketchup for canning	44061	'weeknight' 'timetomake' 'course' 'mainingredi...	my dh's amish mother raised him on this recipe...	['tomato juice', 'apple cider vinegar', 'sugar...	[Vegetarian, Vegan, None]

Modeling

Before I can use the data with TensorFlow, I need to ensure that all features (except for rating) are strings:

```
recipe_data_cleaned['diets_combined'] = recipe_data_cleaned['diets_combined'].astype(str)
recipe_data_cleaned['recipe_id'] = recipe_data_cleaned['recipe_id'].astype(str)
recipe_data_cleaned['name'] = recipe_data_cleaned['name'].astype(str)
user_recipe_ratings['user_id'] = user_recipe_ratings['user_id'].astype(str)
user_recipe_ratings['recipe_id'] = user_recipe_ratings['recipe_id'].astype(str)

```

```
#making sure this worked as intended:
user_recipe_ratings.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1132367 entries, 0 to 1132366
Data columns (total 3 columns):
 #   Column     Non-Null Count    Dtype
---  ------     --------------    -----
 0   user_id    1132367 non-null  object
 1   recipe_id  1132367 non-null  object
 2   rating     1132367 non-null  int64
dtypes: int64(1), object(2)
memory usage: 25.9+ MB
```

Next, I create one dataframe with all the necessary features to make things simpler:

```
merged_df = user_recipe_ratings.merge(recipe_data_cleaned, on="recipe_id", how="left")
merged_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1132367 entries, 0 to 1132366
Data columns (total 8 columns):
 #   Column          Non-Null Count    Dtype
```

```
 ---  ------          --------------    -----
  0   user_id         1132367 non-null  object
  1   recipe_id       1132367 non-null  object
  2   rating          1132367 non-null  int64
  3   name            1132367 non-null  object
  4   tags            1132367 non-null  object
  5   description     1108857 non-null  object
  6   ingredients     1132367 non-null  object
  7   diets_combined  1132367 non-null  object
 dtypes: int64(1), object(7)
 memory usage: 77.8+ MB
```

```
merged_df = merged_df.drop(columns=['tags', 'description', 'ingredients'])
merged_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1132367 entries, 0 to 1132366
Data columns (total 5 columns):
 #   Column          Non-Null Count    Dtype
---  ------          --------------    -----
 0   user_id         1132367 non-null  object
 1   recipe_id       1132367 non-null  object
 2   rating          1132367 non-null  int64
 3   name            1132367 non-null  object
 4   diets_combined  1132367 non-null  object
dtypes: int64(1), object(4)
memory usage: 51.8+ MB
```

This merged dataframe needs to be turned into a TensorFlow dataset:

```
merged_ds = tf.data.Dataset.from_tensor_slices(dict(merged_df))

recipes_ds = tf.data.Dataset.prefetch(merged_ds, buffer_size=tf.data.AUTOTUNE)
```

```
print(recipes_ds)
```

```
<_PrefetchDataset element_spec={'user_id': TensorSpec(shape=(), dtype=tf.string, name=None), 'recipe_id': TensorSpec(shape=(
```

Preparing the data for modeling:

```
#Selecting the necessary features from the dataset:
ratings = (recipes_ds.map(lambda x: {
    "user_id": x["user_id"],
    "rating": x["rating"],
    "name": x["name"],
    "diets_combined": x["diets_combined"],

    }))
recipes = (recipes_ds.map(lambda x:x["name"]))
```

```
#mapping all values in each column
#recipe_ids = ratings.map(lambda x: x["recipe_id"])
user_ids = ratings.map(lambda x: x["user_id"])
names = ratings.map(lambda x: x["name"])
diets = ratings.map(lambda x: x["diets_combined"])
```

```
unique_user_ids =  merged_df["user_id"].unique().astype(str)
unique_names =  merged_df["name"].unique().astype(str)
unique_diets =  merged_df["diets_combined"].unique().astype(str)
```

```
#Then shuffle, batch, and cache the training and evaluation data:
tf.random.set_seed(42)
shuffled = recipes_ds.shuffle(100_000, seed=42, reshuffle_each_iteration=False)

train = shuffled.take(100_000)
test = shuffled.skip(100_000).take(30_000)

cached_train = train.shuffle(100_000).batch(8192)
cached_test = test.batch(4096).cache()
```

Multi-task model:

```
# This is multitask recommender model adapted from TensorFlow's website (https://www.tensorflow.org/recommenders/examples/mu
# It conducts both two-tower retrieval and ranking tasks depending on which weight you assign each task.
# This model contains no extra feature embeddings (only looks at user_id, recipe_id, and ratings for creating recommendation

class UserRecipesModel(tfrs.models.Model):

  def __init__(self, rating_weight: float, retrieval_weight: float) -> None:
    # We take the loss weights in the constructor: this allows us to instantiate
    # several model objects with different loss weights.

    super().__init__()

    embedding_dimension = 32

    # User and recipe models.
    self.recipe_model: tf.keras.layers.Layer = tf.keras.Sequential([
      tf.keras.layers.StringLookup(
        vocabulary=unique_names, mask_token=None),
      tf.keras.layers.Embedding(len(unique_names) + 1, embedding_dimension)
    ])
    self.user_model: tf.keras.layers.Layer = tf.keras.Sequential([
      tf.keras.layers.StringLookup(
        vocabulary=unique_user_ids, mask_token=None),
      tf.keras.layers.Embedding(len(unique_user_ids) + 1, embedding_dimension)
    ])

    # A small model to take in user and recipe embeddings and predict ratings.
    # We can make this as complicated as we want as long as we output a scalar
    # as our prediction.
    self.rating_model = tf.keras.Sequential([
        tf.keras.layers.Dense(256, activation="relu"),
        tf.keras.layers.Dense(128, activation="relu"),
        tf.keras.layers.Dense(1),
    ])

    # The tasks
    self.rating_task: tf.keras.layers.Layer = tfrs.tasks.Ranking(
        loss=tf.keras.losses.MeanSquaredError(),
        metrics=[tf.keras.metrics.RootMeanSquaredError()],
    )
    self.retrieval_task: tf.keras.layers.Layer = tfrs.tasks.Retrieval(
        metrics=tfrs.metrics.FactorizedTopK(
            candidates=recipes.batch(128).map(self.recipe_model)
        )
    )


    # "since we have two tasks and two losses - we need to decide on how important each loss is.
    # We can do this by giving each of the losses a weight, and treating these weights as hyperparameters"

    # The loss weights.
    self.rating_weight = rating_weight
    self.retrieval_weight = retrieval_weight

  def call(self, features: Dict[Text, tf.Tensor]) -> tf.Tensor:
    # We pick out the user features and pass them into the user model.
    user_embeddings = self.user_model(features["user_id"])
    # And pick out the recipe features and pass them into the recipe model.
    recipe_embeddings = self.recipe_model(features["name"])

    return (
        user_embeddings,
        recipe_embeddings,
        # We apply the multi-layered rating model to a concatentation of
```

```
        # We apply the multi-layered rating model to a concatentation of
        # user and recipe embeddings.
        self.rating_model(
            tf.concat([user_embeddings, recipe_embeddings], axis=1)
        ),
    )

  def compute_loss(self, features: Dict[Text, tf.Tensor], training=False) -> tf.Tensor:

    ratings = features.pop("rating")

    user_embeddings, recipe_embeddings, rating_predictions = self(features)

    # We compute the loss for each task.
    rating_loss = self.rating_task(
        labels=ratings,
        predictions=rating_predictions,
    )
    retrieval_loss = self.retrieval_task(user_embeddings, recipe_embeddings)

    # And combine them using the loss weights.
    return (self.rating_weight * rating_loss
            + self.retrieval_weight * retrieval_loss)
```

```
#Ranking specialized model (only the ranking task has weight)
#Adam optimizer
model_1a = UserRecipesModel(rating_weight=1.0, retrieval_weight=0.0)
model_1a.compile(optimizer=tf.keras.optimizers.Adam(0.05))

```

```
#For these models I only have them set to fit for 1 epoch currently, due to computation expense and run time, but I have trie
model_1a.fit(cached_train, epochs=1)

```

```
13/13 [==============================] - 3350s 256s/step - root_mean_squared_error: 4.9599 - factorized_top_k/top_1_categori
<keras.src.callbacks.History at 0x7df566d2e2f0>
```

```
metrics = model_1a.evaluate(cached_test, return_dict=True)


```

```
8/8 [==============================] - 1022s 126s/step - root_mean_squared_error: 2.2481 - factorized_top_k/top_1_categorica
Ranking RMSE: 2.248.
```

```
print(f"Retrieval top-100 accuracy: {metrics['factorized_top_k/top_100_categorical_accuracy']:.3f}.")
print(f"Ranking RMSE: {metrics['root_mean_squared_error']:.3f}.")
```

- Top-100 accuracy: 3.0e-4
- RMSE: 2.248

```
#Retrieval specialized model (only the retrieval task has weight)

model_1b = UserRecipesModel(rating_weight=0.0, retrieval_weight=1.0)
model_1b.compile(optimizer=tf.keras.optimizers.Adam(0.05))
```

```
model_1b.fit(cached_train, epochs=1)
metrics = model_1b.evaluate(cached_test, return_dict=True)


```

```
13/13 [==============================] - 3250s 248s/step - root_mean_squared_error: 4.6520 - factorized_top_k/top_1_categori
8/8 [==============================] - 1007s 124s/step - root_mean_squared_error: 4.6844 - factorized_top_k/top_1_categorica
Ranking RMSE: 4.684.
```

```
print(f"Retrieval top-100 accuracy: {metrics['factorized_top_k/top_100_categorical_accuracy']:.3f}.")
print(f"Ranking RMSE: {metrics['root_mean_squared_error']:.3f}.")
```

```
Retrieval top-100 accuracy: 0.000.
Ranking RMSE: 4.684.
```

- Top-100 accuracy: 0.000

- RMSE: 4.684

```
#joint model (both tasks have weight)

model_1 = UserRecipesModel(rating_weight=1.0, retrieval_weight=1.0)
model_1.compile(optimizer=tf.keras.optimizers.Adam(0.05))
```

```
model_1.fit(cached_train, epochs=1)
metrics = model_1.evaluate(cached_test, return_dict=True)
print(metrics)
print(f"Retrieval top-100 accuracy: {metrics['factorized_top_k/top_100_categorical_accuracy']:.3f}.")
print(f"Ranking RMSE: {metrics['root_mean_squared_error']:.3f}.")
```

```
13/13 [==============================] - 3210s 245s/step - root_mean_squared_error: 4.8690 - factorized_top_k/top_1_categori
8/8 [==============================] - 988s 122s/step - root_mean_squared_error: 2.1959 - factorized_top_k/top_1_categorical
{'root_mean_squared_error': 2.195901870727539, 'factorized_top_k/top_1_categorical_accuracy': 9.999999747378752e-05, 'factor
Retrieval top-100 accuracy: 0.001.
Ranking RMSE: 2.196.
```

- Top-100 accuracy: 0.001
- RMSE: 2.196

Model with extra embeddings:

```
#the user model with no additional embeddings:
class UserModel2(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.user_embeddings = tf.keras.Sequential(
[tf.keras.layers.StringLookup(vocabulary=unique_user_ids,    mask_token=None),tf.keras.layers.Embedding(len(unique_user_ids)+
    def call(self, inputs):
          return self.user_embeddings(inputs["user_id"])
```

```
# recipe model with only the diet embedding added
class RecipeModel2(tf.keras.Model):

 def __init__(self):
   super().__init__()

   max_tokens = 10_000

   # self.recipe_embedding = tf.keras.Sequential([
   #   tf.keras.Sequential(
   #       [tf.keras.layers.StringLookup(vocabulary=unique_recipe_names, mask_token=None),
   #        tf.keras.layers.Embedding(len(unique_recipe_ids)+1, 32)
   #        ])
   #   ])
   self.recipe_embedding = tf.keras.Sequential([
       tf.keras.layers.StringLookup(vocabulary=unique_names, mask_token=None),
       tf.keras.layers.Embedding(len(unique_names)+1, 32)])

   self.diet_embedding = tf.keras.Sequential([
       tf.keras.layers.StringLookup(vocabulary=unique_diets, mask_token=None),
       tf.keras.layers.Embedding(len(unique_diets)+1, 32)])

   self.text_vectorizer =  tf.keras.layers.TextVectorization(max_tokens=max_tokens)

   self.text_vectorizer.adapt(diets)
   # self.text_vectorizer.adapt(names)

 def call(self, inputs):
   return tf.concat( [self.recipe_embedding(inputs), self.diet_embedding(inputs)],axis=1)

```

```
#combined model


#https://blog.searce.com/recommendation-systems-using-tensorflow-recommenders-d7d12167b0b7
class RecipeRecommendModel2(tfrs.models.Model):

    def __init__(self, rating_weight, retrieval_weight):
        super().__init__()
        embedding_dimension = 32
        self.query_model = tf.keras.Sequential([UserModel2(), tf.keras.layers.Dense(embedding_dimension)])
        self.candidate_model = tf.keras.Sequential([RecipeModel2(), tf.keras.layers.Dense(embedding_dimension)])
        self.rating_model = tf.keras.Sequential(
            [tf.keras.layers.Dense(256, activation="relu"),
            tf.keras.layers.Dense(128, activation="relu"),
            tf.keras.layers.Dense(1)]
            )
        self.retrieval_task = tfrs.tasks.Retrieval(
            metrics=tfrs.metrics.FactorizedTopK(candidates=recipes.batch(128).map(self.candidate_model))
            )
        self.rating_task = tfrs.tasks.Ranking(
            loss=tf.keras.losses.MeanSquaredError(), metrics=[tf.keras.metrics.RootMeanSquaredError()])
       # The loss weights.
        self.rating_weight = rating_weight
        self.retrieval_weight = retrieval_weight

    def call(self, features: Dict[Text, tf.Tensor]) -> tf.Tensor:
        user_embeddings = self.query_model({"user_id": features["user_id"]})
        recipe_embeddings = self.candidate_model({"name":features["name"]})
        return (user_embeddings, recipe_embeddings, self.rating_model(tf.concat([user_embeddings, recipe_embeddings],axis=1))

    def compute_loss(self, features: Dict[Text, tf.Tensor], training=False) -> tf.Tensor:

        ratings = features.pop("rating")
        user_embeddings, recipe_embeddings, rating_predictions = self(features)
        # We compute the loss for each task.
        rating_loss = self.rating_task(labels=ratings, predictions=rating_predictions)
        retrieval_loss = self.retrieval_task(user_embeddings, recipe_embeddings)
        # And combine them using the loss weights.
        return (self.rating_weight * rating_loss + self.retrieval_weight * retrieval_loss)

```

```
model_2 = RecipeRecommendModel2(1, 1)
model_2.compile(optimizer=tf.keras.optimizers.Adam(0.05))
```

```
from keras.callbacks import EarlyStopping
es = EarlyStopping(monitor='root_mean_squared_error', patience =2)
```

```
history = model_2.fit(cached_train, epochs=3, callbacks=es )
```

```
Epoch 1/3
13/13 [==============================] - 3101s 236s/step - factorized_top_k/top_1_categorical_accuracy: 2.4000e-04 - factori
Epoch 2/3
13/13 [==============================] - 3145s 241s/step - factorized_top_k/top_1_categorical_accuracy: 0.0020 - factorized_
Epoch 3/3
13/13 [==============================] - 3126s 239s/step - factorized_top_k/top_1_categorical_accuracy: 0.0191 - factorized_
```

```
history.history
```


```
{'factorized_top_k/top_1_categorical_accuracy': [0.00023999999393709004,
  0.0020099999383091927,
  0.019139999523758888],
 'factorized_top_k/top_5_categorical_accuracy': [0.00033000000985339284,
  0.0024300001095980406,
  0.029020000249147415],
 'factorized_top_k/top_10_categorical_accuracy': [0.00044999999227002263,
  0.003120000008493662,
  0.039570000022649765],
 'factorized_top_k/top_50_categorical_accuracy': [0.001339999958872795,
  0.007689999882131815,
  0.08913999795913696],
 'factorized_top_k/top_100_categorical_accuracy': [0.0022700000554323196,
  0.011950000189244747,
  0.12381000071763992],
 'root_mean_squared_error': [15.090824127197266,
  1.998063564300537,
  1.5534210205078125],
 'loss': [12518.3125, 11603.45703125, 9435.779296875],
```

```
 'regularization_loss': [0, 0, 0],
 'total_loss': [12518.3125, 11603.45703125, 9435.779296875]}
```

```
model_2.evaluate(cached_test)
```

```
8/8 [==============================] - 981s 121s/step - factorized_top_k/top_1_categorical_accuracy: 3.3333e-05 - factorized
[3.333333370392211e-05,
 6.666666740784422e-05,
 6.666666740784422e-05,
 0.00019999999494757503,
 0.0006000000284984708,
 1.3955219984054565,
 10845.0078125,
 0,
 10845.0078125]
```

- RMSE: 1.3955
- Top-100 accuracy: 6.0000e-04 (0.0006)

Efficient Serving (ScaNN)

First, creating a Brute Force retrieval method to compare the retrieval efficiency

```
#trying out baseline method for serving:

# Override the existing streaming candidate source.
brute_force = tfrs.layers.factorized_top_k.BruteForce(model_2.query_model)

brute_force.index_from_dataset(
    recipes.batch(128).map(lambda name: (name, model_2.candidate_model(name)))
)

```

```
<tensorflow_recommenders.layers.factorized_top_k.BruteForce at 0x7e7ef53112d0>
```

```
#Looking for user_id to use for example recommendations:
user_data.head()
```

	user_id	recipe_id	date	rating	review
0	38094	40893	2003-02-17	4	Great with a salad. Cooked on top of stove for...
1	1293707	40893	2011-12-21	5	So simple, so delicious! Great for chilly fall...
2	8937	44394	2002-12-01	4	This worked very well and is EASY. I used not...
3	126440	85009	2010-02-27	5	I made the Mexican topping and took it to bunk...
4	57222	85009	2011-10-01	5	Made the cheddar bacon topping, adding a sprin...

```
%timeit _, names = brute_force({"user_id":tf.constant(["38094"])}, k=3)
```

```
24.6 ms ± 165 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Using ScaNN (scalable nearest neighbors) to improve retrieval efficiency:

```

 scann = tfrs.layers.factorized_top_k.ScaNN(
     model_2.query_model,
     num_leaves=100,
     num_leaves_to_search=400,

 )

 scann.index_from_dataset(tf.data.Dataset.zip((recipes.batch(128).map(lambda recipe: (recipe, model_2.candidate_model(recipe)
```

```
<tensorflow_recommenders.layers.factorized_top_k.ScaNN at 0x7e7ef5471a80>
```

```
 %timeit _, names = scann({"user_id":tf.constant(["38094"])}, k=3)
```

```
3.96 ms ± 27.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Evaluating the two serving methods:

```
# Override the existing streaming candidate source.
model_2.retrieval_task.factorized_metrics = tfrs.metrics.FactorizedTopK(
    candidates=brute_force
)
# Need to recompile the model for the changes to take effect.
model_2.compile()


%time bf_result = model_2.evaluate(cached_test, return_dict=True, verbose=False)
```

```
CPU times: user 1.83 s, sys: 1.14 s, total: 2.98 s
Wall time: 654 ms
```

```
# Override the existing streaming candidate source.
model_2.retrieval_task.factorized_metrics = tfrs.metrics.FactorizedTopK(
    candidates=scann
)
# Need to recompile the model for the changes to take effect.
model_2.compile()


%time scann_result = model_2.evaluate(cached_test, return_dict=True, verbose=False)
```

```
CPU times: user 1.75 s, sys: 1.13 s, total: 2.87 s
Wall time: 737 ms
```

The ScaNN retrieval method takes much less time than the Brute Force method.

Generating an example set of recommendations for user 38094:

```
_, recs = scann({"user_id":tf.constant(["38094"])})
print(f"Top recommendations: {np.unique(recs)[:3]}")
```

```
Top recommendations: [b'cherry and blueberry trifle' b'chicken and basil meatballs'
 b'chicken and vegetable salad']
```

For user #38094, the top 3 recommended recipes are:

- 'cherry and blueberry trifle'
- 'chicken and basil meatballs'
- 'chicken and vegetable salad'

Limitations and Next Steps

Next Steps:

- Deploy model
- Update model with new data
- Add more diet types
- Try to improve model metrics:
  - experiment with different depths
  - try using a feature cross
  - further tune model parameters

Limitations:

- Model takes a long time to run and is computationally expensive