Recommendation in AWS Sagemaker using custom Scikit-learn (KNN) code

In this post I wanted to demonstrate a simple framework for leveraging custom Scikit-learn code within the AWS Sagemaker platform. As a PoC I created a simple music artist recommendation system using an unsupervised implementation of K-Nearest Neighbors (KNN). This blog post serves as a framework for implementing other custom Scikit-learn code as well as a prerequisite for leveraging more advanced Deep Learning libraries (i.e. Pytorch). The complete source code for this project can be accessed through my personal Github repo.

AWS Sagemaker

AWS Sagemaker and similar MLaaS (Machine Learning as a Service) platforms are gaining more adoption by streamlining much of the technical overhead when deploying machine learning models to production. One nice feature of Sagemaker is the contimuum of customization offered when building a model. On the one end, users can leverage out of the box Sagemaker models using a very simple high level API. However, for more specific use cases you can customize all aspects of the modeling process including training, hyperparameter tuning, inference, model building, and input/output functionality. This post will focus on customizing training and inference code and references the Sagemaker Python SDK (software development kit).

Scikit-learn

Scikit-learn is a popular machine learning library and for good reason. It offers a broad set of ML functionality within an intuitive API making it a common first choice for prototyping models. The good news is that Sagemaker offers a framework to host Scikit-learn models. This framework requires the developer create a custom training script and optionally custom inference code. Custom inference is required when leveraging a method other than a model's default *predict()* function. The recommendation PoC outlined below implements both of these modifications.

Use Case PoC - KNN Recommendation

Content recommendation is a very popular machine learning application with a wide array of approaches. A very simple implementation is to use an unsupervised algorithm such as K Nearest Neighbors to quantify similarity between items to recommend. This method is a basic form of "item based" collaborative filtering. We will implement such a system for music artist recommendation.

The Data

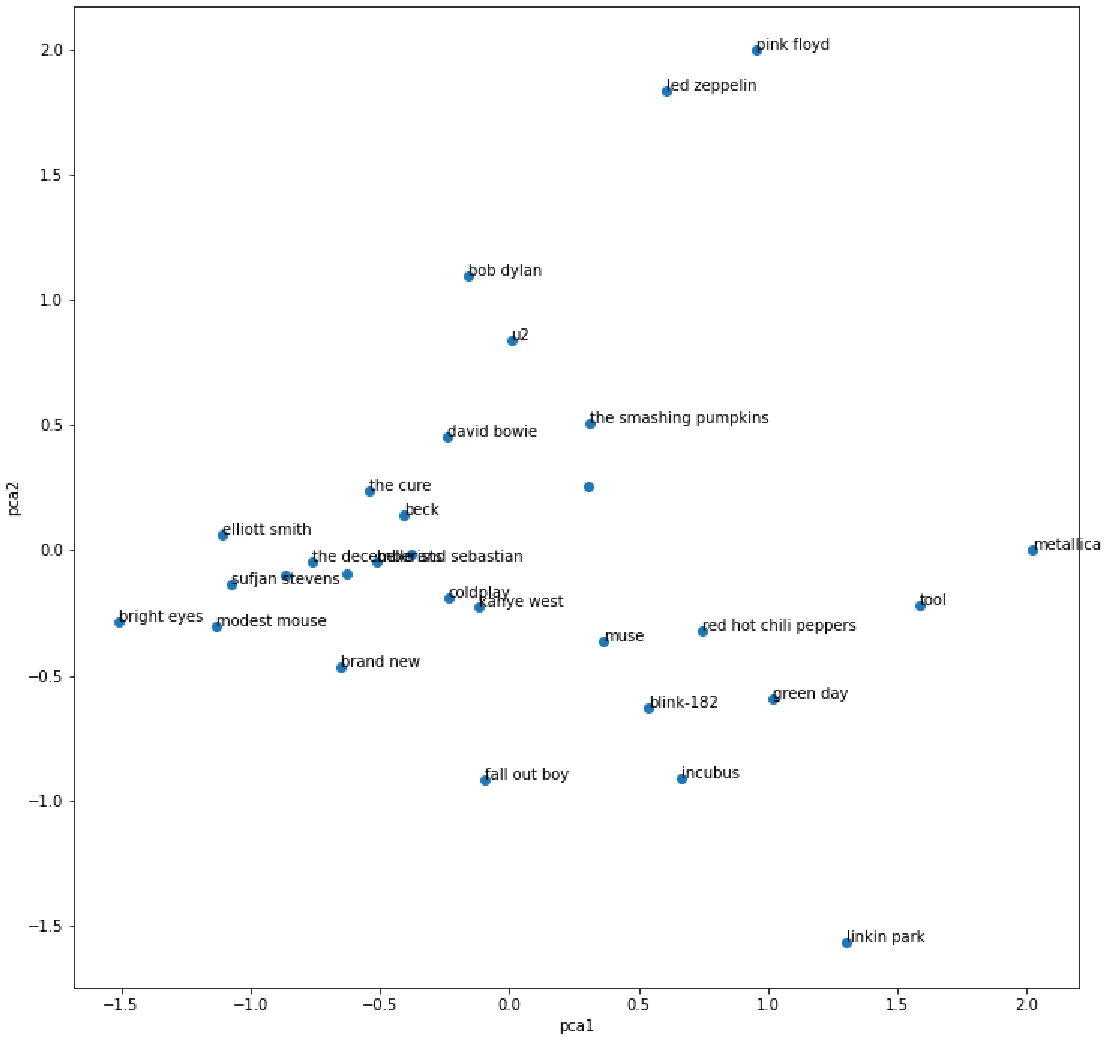
For this demonstration, we will use a dataset sourced from the Last.fm API by Oscar Celma of Pompeu Fabra University. The data is available for non-commercial public use via the university website:

- https://www.upf.edu/web/mtg/lastfm360k

The data contains ~17M records representing total play counts for distinct user + artist combinations. One caveat is that the data was collected in 2008, so our recommendations will reflect this bias.

Approach

For the purposes of this PoC we are only interested in the User ID, Artist, and Play count fields. To make the dataset more manageable we are limiting to users within the United States and filtering to the top 10,000 users and top 1,000 artists by play count. Using this subset of data we will create a sparse matrix (1,000 artists x 10,000 users). For each artist, the play counts for all 10,000 users will represent a feature vector from which can compute similarity between artists. To illustrate this concept, let's look at the the first two principle components (reduced user vectors) for some of the top artists by play volume:



Complete code to ingest and transform the raw data can be found in the below notebooks:

1. ***DataIngest.ipynb***
2. ***DataPrep.ipynb***

The output of these notebooks creates the below files and directories:

Training Data:

/data/artist_user_mtrx.npz

Mapping Dictionaries

/datasources/artist_to_idx.pkl

/datasources/idx_to_artist.pkl

/datasources/user_to_idx.pkl

/datasources/idx_to_user.pkl

Using the described data structure as input (artist_user_mtrx.npz), we fit a KNN model and use the KNN.neighbors method to generate the top K recommendations for any given input artist. The details of this implementation within Sagemaker are outlined in the next section and code detail is found in the ***TrainKNN_topNartists.ipynb*** notebook.

Sagemaker Workfow

At the highest level, a typical Sagemaker workflow using a built-in algorithm consists of the following steps.

1. Create a Sagemaker session and get execution role
2. Upload training data to s3
3. Instantiate an estimator object
4. Train estimator
5. Deploy estimator
6. Generate/evaluate predictions

To leverage custom Scikit-learn KNN code we need to add a few additional steps.

1. Create a Sagemaker session and get execution role
2. Upload training data to s3
3. Instantiate an estimator object
4. ***Create custom train.py script***
5. Train estimator

6. ***Create custom predict.py script***
7. ***Define new model object pointing to custom inference code***
8. Deploy estimator
9. Generate/evaluate predictions

Let's do it!

Create Sagemaker Session and Get Execution Role

The Sagemaker session is a class containing some convenient methods to interact with Sagemaker resources (in our case uploading training data to s3). We also need to get our execution role, which specifies which AWS resources we are able to access.

```
import sagemaker

sagemaker_session = sagemaker.Session()
role = sagemaker.get_execution_role()
```

Upload Training Data to s3 Bucket

Using the Sagemaker session object, we upload the training set within the local 'data' directory we created in data prep. We specify a prefix, which will be the directory name for the training data created within the default s3 bucket.

```
data_dir = 'data'
prefix = 'rcmdKNN'
input_data = sagemaker_session.upload_data(data_dir, key_prefix=prefix
```

Instantiate Estimator Object

Now we are ready to instantiate an estimator. For our purposes we are using a special sklearn estimator object. Within the estimator we define various hyperparameters. Of note are 'entry_point' which points to our custom training script file and 'hyperparameters' which are model hyperparameters that will be parsed within the custom training script.

```
from sagemaker.sklearn.estimator import SKLearn

estimator = SKLearn(entry_point='train.py',
            framework_version='0.23-1',
            role=role,
            train_instance_count=1,
            train_instance_type='ml.c4.xlarge',
            py_version='py3',
            source_dir='source',
            image_uri=None,
            hyperparameters = {'n_neighbors':101,
                        'metric':'cosine',
                        'algorithm':'brute'})
```

Create custom training script (train.py)

Next we create the custom training script. The script has 4 components:

- ***model_fn*** - A function to load an existing model from the default model directory
- ***argument parsing*** - To extract environment variables and model hyperparameters
- ***model fitting*** - Load in the data and fit a KNN model with the parsed hyperparameters
- ***model saving*** - Save the fitted model to the default model directory

```
## train.py ##
from __future__ import print_function
from sklearn.neighbors import NearestNeighbors
import argparse
import os
import pandas as pd
import numpy as np
import scipy
import joblib

def model_fn(model_dir):

    # load using joblib
    model = joblib.load(os.path.join(model_dir, "model.joblib"))

    return model

if __name__ == '__main__':
    parser = argparse.ArgumentParser()

    parser.add_argument('--output-data-dir', type=str, default=os.environ['SM_OUTPUT_DATA_DIR'])
    parser.add_argument('--model-dir', type=str, default=os.environ['SM_MODEL_DIR'])
    parser.add_argument('--data-dir', type=str, default=os.environ['SM_CHANNEL_TRAIN'])

    parser.add_argument('--n_neighbors', type=int, default=10)
    parser.add_argument('--metric', type=str, default='cosine')
    parser.add_argument('--algorithm', type=str, default='brute')

    args = parser.parse_args()

    training_dir = args.data_dir
    train_data = scipy.sparse.load_npz(os.path.join(training_dir, "artist_user_mtrx.npz"))
    train_data = np.array(train_data.todense())

    model = NearestNeighbors(n_neighbors=args.n_neighbors,
                        metric=args.metric,
                        algorithm=args.algorithm)

    model.fit(train_data)

    joblib.dump(model, os.path.join(args.model_dir, "model.joblib"))
```

Train the estimator

Once we have the custom training script ready, its time to fit the model. To do this we simply call .fit on our estimator object and point to the s3 training data location.

```
estimator.fit({'train': input_data})
```

Create a custom inference script (*predict.py*)

Now that our model is fitted, we need to create some custom inference code. Within Sagemaker, the default inference logic calls the *.predict()* method on the created model. However, in our case we are using the unsupervised implemented of KNN. Therefore there is no *.predict()* function available. Instead we need to write a custom predict function that calls *.neighbors()* and returns the indices of the K nearest neighbors. We also need to include the same model loading function (*model_fn*) from *train.py*.

```
## predict.py ##
import joblib
import os

def model_fn(model_dir):
    """Load model from the model_dir. This is the same model that is saved
    in the main if statement.
    """
    print("Loading model.")

    # load using joblib
    model = joblib.load(os.path.join(model_dir, "model.joblib"))
    print("Done loading model.")

    return model

def predict_fn(input_data, model):
    distances, indices=model.kneighbors(input_data)
    return indices[0]
```

Define new model object pointing to custom inference code

Next we will create a new model object pointing to our custom inference code. To do this we instantiate a new model object, passing in the model parameters we previously fit and pointing the entry point to *predict.py*.

```
from sagemaker.sklearn import SKLearnModel

sklearn_model = SKLearnModel(model_data=estimator.model_data,
                             role=role,
                             entry_point="predict.py",
                             source_dir='source',
                             framework_version='0.23-1')
```

Deploy estimator

From here we can call *.deploy*() on the secondary model to generate a "predictor" object that return the indices of K nearest neighbors for a given artist's feature array.

```
predictor = sklearn_model.deploy(instance_type="ml.m4.xlarge", initial_instance_count=1)
```

Generate Predictions

Now we are ready to see the recommender in action. With the help of a few mapping functions we can test out some examples. Let's choose 5 artists from different eras and generate 15 recommendations for each.

```
def process_input(artists, input_data):
    artist_ids = [artist_to_idx[i] for i in artists]
    artist_ids = input_data[artist_ids].reshape(1,-1)
    return artist_ids

def process_output(result, n):
    artists = [idx_to_artist[i] for i in result]
    print ('Top Recommended:', artists[1:n+1])

for artist in ['the beatles', 'eagles', 'genesis', 'nirvana', 'the strokes']:
    print ('\nInput artist: {} \n'.format(artist))
    input_data = process_input([artist], train_data)
    response = predictor.predict(input_data)
    process_output(response, 15)
```

```
Output:
Input artist: the beatles

Top Recommended: ['bob dylan', 'led zeppelin', 'the rolling stones', 'pink floyd', 'radiohead', 'the who', 'john lennon',
'david bowie', 'simon & garfunkel', 'beck', 'the white stripes', 'the beach boys', 'paul mccartney', 'modest mouse', 'the
shins']

Input artist: eagles

Top Recommended: ['elton john', 'billy joel', 'lynyrd skynyrd', 'chicago', 'eric clapton', 'boston', 'creedence clearwater
revival', 'jimmy buffett', 'fleetwood mac', 'tom petty and the heartbreakers', 'the rolling stones', 'led zeppelin', 'james
taylor', 'journey', 'pink floyd']

Input artist: genesis

Top Recommended: ['yes', 'peter gabriel', 'jethro tull', 'rush', 'king crimson', 'styx', 'pink floyd', 'porcupine tree', 'the
police', 'the moody blues', 'queensrÿche', 'dream theater', 'journey', 'ayreon', 'black sabbath']

Input artist: nirvana

Top Recommended: ['alice in chains', 'pearl jam', 'soundgarden', 'the smashing pumpkins', 'stone temple pilots', 'nine
inch nails', 'red hot chili peppers', 'foo fighters', 'led zeppelin', 'the white stripes', 'metallica', 'the beatles', 'pixies',
'green day', 'radiohead']
```

```
Input artist: the strokes

Top Recommended: ['arctic monkeys', 'the killers', 'interpol', 'franz ferdinand', 'the shins', 'kings of leon', 'radiohead',
'the libertines', 'coldplay', 'the white stripes', 'death cab for cutie', 'modest mouse', 'red hot chili peppers', 'arcade fire',
'bloc party']
```

Evaluation

Anecdotally, the system looks to be generating some quality recommendations. However, this type of item based collaborative filtering can be biased toward recommending popular artists. One option would be to increase the K value of the neighbors being generated and focus on recommendations further down the list (they are sorted most to least similar). Another possibly is to increase the original sample size of artists in the training data (>1,000). This would like introduce some more obscure recommendations into the system.

In terms of a more robust evaluation, standard regression and classification metrics do not apply to this specific unsupervised approach. Additionally recommendation quality is inherently subjective and difficult to evaluate compared to more standard machine learning problems. In a production context, some sort of a/b testing framework would be useful in monitoring recommendation quality through real-time user feedback. Alternatively, more advanced approaches such as matrix factorization could be used to structure the problem in classification/regression context.

That being said, let's generate a "nice to know" metric. We can calculate a "hit rate" for recommendations based on a user's favorite artist. The approach is as follows:

- For a random sample of users, identify each user's favorite artist (most played)
- Generate K number of recommendations from the user's favorite artist
- Determine the number of recommended artists that were actually played by the user
- For all users in sample, calculate the hit rate:
    - Hit Rate = # of played artists / # recommended artists

```
def rcmnd_from_fav(user, data, num_preds = 100):
    play_history = data[:,user]
    artist_idx = (-play_history).argsort()[:1]
    predictions = predictor.predict(data[artist_idx].reshape(1,-1))
    return predictions[1:num_preds+1]

def hit_rate(user, data, predictions):
    hits = train_data[:, user][predictions]
    return hits

hits = []
num_users = 100
preds_per_user = 10

for user in np.random.randint(0, train_data.shape[1], size=num_users):
    predictions = rcmnd_from_fav(user, train_data)
    hits = hit_rate(user, train_data, predictions)
    hits+=hits

rate = (np.count_nonzero(hits) / len(hits))

print ('Artist Hit Rate for {} Users ({} Recommendations per User): {:.0%}'\
       .format(num_users, preds_per_user, rate))
```

```
Output:
Artist Hit Rate for 100 Users (10 Recommendations per User): 28%
```

For this random sample of 100 users with 10 recommendations per user, our system has a hit rate of 28%.

Recap

As we demonstrated, it is feasible to implement custom Scikit-learn code into a Sagemaker workflow. Given that Scikit-learn has been in wide use for many years, this approach might have value in productionalizing existing ML workflows prototyped in a local environment. Additionally, this custom approach is required for more advanced deep learning applications using Pytorch or Tensorflow. A natural extension of this project could be to construct a deep learning based recommendation system to improve performance and enable more robust evaluation.