JOKE RECOMMENDER

MID PROJECT REVIEW

Northwestern

Anjali Verma

HIGHLIGHTS

- ***Data Ingestion*** ~ To meet the model objective of **recommending personalized jokes**, the '***Jester'*** dataset which comprises of two tables, one with user ratings corresponding to each joke and the other with complete textual jokes mapped to each joke id was acquired from the source and stored into a publicly accessible **S3 bucket**.

- ***Exploratory data Analysis ~*** The ingested data does not have missing records or inconsistencies. An important aspect was to determine if one user had rated the same joke multiple times and found that there were no duplicate records. Transformed the data into usable format by creating a user-item matrix to calculate similarity between users and between items.

    - ***Interesting insights ~*** Sparsity of an interaction matrix is the number of empty cells(when a user does not rate an item) divided by the number of cells. The sparsity for the interaction matrix for jester data was 0.59 which would be suitable to build a collaborative filtering model. The user ratings would need to be centered since different users use the rating scale differently. (i.e some have a tendency to rate higher, some lower)

PROGRESS REVIEW

Stories that have been completed ~

- ***Story 1.1.1 Defined model objective~*** Model is intended to find similarity between user preferences as well as between types of jokes to be able to recommend customized jokes.
- ***Story 1.1.2 Acquired and stored data~*** Transformed data into format desired by creating interaction matrix
- ***Story 1.1.3 Data Cleaning~*** Checked for duplicates and missing records in data to ensure consistency
- ***Story 1.1.5*** Split the data into training and test sets with 80% of the records going into training and the remaining 20% into test set
- ***Story 1.1.5.2*** Trained a Collaborative Filtering model using Lightfm package in python
- ***Story 1.2.1*** Decided on AUC as the loss function because we're interested in whether a user likes the recommended joke and determining customers' preference in general is more practical than predicting the actual rating they would give a joke.
- ***Story 1.2.3*** Evaluated the initial model that gave a test auc of 0.13
- ***Story 1.3.3 Versioning~*** Created a GitHub repository for collaboration
- ***Story 2.2.1*** Created a database schema in AWS RDS

DEMO/ANALYSIS

```
df = df1.loc[df1['JokeID'].isin(sample_indices)]
df.head()
```

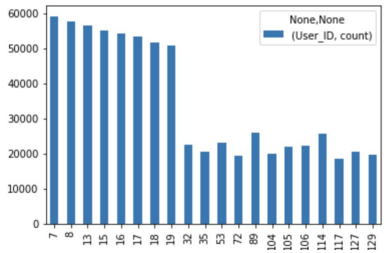
	User_ID	JokeID	Rating
1	1	7	-9.281
2	1	8	-9.281
3	1	13	-6.781
4	1	15	0.875
5	1	16	-9.656

Plotting number of user ratings for each joke id

```
jokeFreq =(df[['User_ID','JokeID']].groupby(['JokeID']).agg(['count']))
jokeFreq
userFreq = (df[['User_ID','JokeID']].groupby(['User_ID']).agg(['count']))
userFreq

jokeFreq.plot.bar()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x106289128>
```




```
df.nunique()
```

```
User_ID    59123
JokeID        20
Rating       641
dtype: int64
```

```
df.isnull().sum()
```

```
User_ID    0
JokeID     0
Rating     0
dtype: int64
```

```
df.duplicated(subset= ['User_ID', 'JokeID'], keep='first').sum()
```

```
0
```

```
print('Duplicated rows: ' + str(df.loc[:,['User_ID','JokeID']].duplicated().sum()))
```

```
Duplicated rows: 0
```

```
len(df)-len(df.drop_duplicates(['User_ID', 'JokeID']))
```

```
0
```

Conclude that no joke is rated more than once by the same user

Model Evaluation

```
# Import the evaluation routines
from lightfm.evaluation import auc_score

# Compute and print the AUC score
train_auc = auc_score(model, train, num_threads=NUM_THREADS).mean()
print('Collaborative filtering train AUC: %s' % train_auc)

# We pass in the train interactions to exclude them from predictions.
# This is to simulate a recommender system where we do not
# re-recommend things the user has already interacted with in the train
# set.


test_auc = auc_score(model, test,num_threads=NUM_THREADS).mean()
print('Collaborative filtering test AUC: %s' % test_auc)

from lightfm.evaluation import precision_at_k,recall_at_k

print("Train precision: %.2f" % precision_at_k(model, train, k=5).mean())#0.65,#0.59
print("Test precision: %.2f" % precision_at_k(model, test,train_interactions=train, k=5).mean())#0.21,#0.2

print("Train precision: %.2f" % recall_at_k(model, train, k=5).mean())#0.34
print("Test precision: %.2f" % recall_at_k(model, test,train_interactions=train, k=5).mean())#0.18
```

```
Collaborative filtering train AUC: nan
Collaborative filtering test AUC: 0.13623527
Train precision: 0.94
Test precision: 0.29
Train precision: 0.60
Test precision: 0.47
```

Model Fitting

```
from lightfm import LightFM

# Set the number of threads; you can increase this
# if you have more physical cores available.
NUM_THREADS = 2
NUM_COMPONENTS = 30
NUM_EPOCHS = 30
ITEM_ALPHA = 1e-6
learning_rate=0.05

# Let's fit a WARP model: these generally have the best performance.
model = LightFM(loss='warp',random_state=123,
                item_alpha=ITEM_ALPHA,
               no_components=NUM_COMPONENTS,
               learning_rate=learning_rate)

# Run 3 epochs and time it.
%time model = model.fit(train,epochs=NUM_EPOCHS,num_threads=NUM_THREADS)
```

```
CPU times: user 17.4 s, sys: 75.5 ms, total: 17.5 s
Wall time: 17 s
```

LESSONS LEARNED

ARCHITECTURAL CONSIDERATIONS

Making software reproducible, robust and extensible

- The project would rely on reusable functionality which would need to be broken up into individual components to compose a modularized system. Learned how to decouple individual pieces of functionality to enable testing and facilitate maintenance
- Building a consistent, common repository structure to facilitate collaboration and communication of the project's components
- Making design decisions in both individual scripts as well as the code base as a whole

TECHNIQUES

- Working with and deploying databases on AWS RDS instances
- Storing data in S3
- Argument parsing

RECOMMENDATIONS

Stories planned to be completed in the following sprint~

- ***Story 1.1.4*** Create metadata about jokes by clustering them into genres or categories
- ***Story 1.2.2*** Tune/optimise model hyperparameters to get a better AUC
- ***Story 1.2.3*** Evaluate each model and select the model with the highest AUC for the test set.
- ***Story 1.3.1 Logging~*** To be able to debug later and communicate code to potential developers or collaborators
- ***Story 1.3.2*** Write ***Unit Tests*** and ***Configure reproducibility tests*** that can be run to test each stage of model development
- ***Story 1.3.4 Documentation~*** Make code readable and reproducible by documenting code

    - ***Story 2.1.1*** Develop a GUI to display results and take user input
    - ***Story 2.2.2*** Link the RDS to the front end app by deploying on a AWS EC2 server running a Flask app