Yelp – NLP, Clustering, PCA, and Recommender

Data Preprocessing:

1. Read and load data into pandas data frame.
2. **Filter** out business that are restaurants.
3. Use business_id as **index** to **join** business information dataset and user review dataset.
4. Keep only relevant columns ('business_id', 'reivew_id', 'text', 'stars', 'date').
5. Filter data frame by date, keep only recent (2017 and after) data.
6. Reset the index and save the data frame, the processed dataset looks like:

business_id	name	categories	avg_stars	cool	date	funny	review_id	stars	text	useful	user_id
-6MefnULPED_I942VcFNA	John's Chinese BBQ Restaurant	[Chinese, Restaurants]	3.0	0	2017-08-17	0	hTGD7YEb43cFCswkpZ9ORA	4	This is one of my top 3 places to get BBQ pork...	2	FEg8v92qx3kK4Hu4TF28Fg
-6MefnULPED_I942VcFNA	John's Chinese BBQ Restaurant	[Chinese, Restaurants]	3.0	0	2017-05-31	0	GAaVSO7hu3peq3cFtno1FQ	3	This restaurant is famous for their BBQ dishes...	0	HPtjvIrhzAUkKsiVkeT4MA

Natural Language Processing (NLP):

1. Define **feature (documents)** X variables ('text' for the review).
2. Define **target** y variables (whether average rating > 4 stars).
3. Split data into **training** and **testing**.

```
# documents is your X, target is your y
X = documents
y = target

# Now split the data to training set and test set
train_X, test_X, train_y, test_y = train_test_split(X, y, test_size = 0.8, random_state = 666)
# Note: test size here is very large to prevent crushing on model training, but in real life should be ~0.3
```

4. Fit **TF-IDF vectorization** on train X.

   Then transform train X, test X, and all documents (train test combined).

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
# Create TfidfVectorizer, and name it vectorizer, choose a reasonable max_features, e.g. 1000
vectorizer = TfidfVectorizer(stop_words = 'english', max_features = 1000)
```

```
# Train the model with your training data
vectorized_train = vectorizer.fit_transform(train_X).toarray()
```

Vectorized output looks like:

```
array([[0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.1772754, 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ]])
```

5. Could look at the vocabs and corresponding idf values:

```
# Get the vocab of your tfidf
vocab = vectorizer.get_feature_names()

for feature, idf in zip (vectorizer.get_feature_names(),vectorizer.idf_):
    print (feature,":", idf)
```

```
able : 4.0333149847030345
absolutely : 4.449475381927947
access : 5.686238009076873
actually : 3.9131706728609705
add : 4.705408756065147
added : 5.244405256797834
additional : 5.175412385310882
advertised : 5.804021044733257
afternoon : 5.398555936625092
ago : 4.417726683613366
ahead : 5.318513228951556
alcohol : 5.580877493419047
amazing : 2.899855964704756
```

Clustering with K Means:

1. **Train** (fit) K-Means on vectorized train X.

```
# Fit k-means on training:
k = 7
kmeans = KMeans(n_clusters = k)
kmeans.fit(vectorized_train_X)
```

2. **Predict** trained model on test X.

```
# Predict on testing:
prediction = kmeans.predict(vectorized_test_X)
prediction[:10]
```

```
array([5, 6, 6, 2, 0, 2, 4, 0, 1, 0], dtype=int32)
```

3. Look at index of the top **cluster centers** of the K-Means model.

```
# Find top 10 features in each centriod:
top_centriods_index = kmeans.cluster_centers_.argsort()[:,-1:989:-1]
top_centriods_index
```

```
array([[322, 362, 765,  46, 628, 117, 988, 292, 689, 756],
       [117, 355, 655, 685, 118,  93, 248, 322, 667, 756],
       [199, 460, 439, 117, 322, 788, 355, 753, 628, 362],
       [887, 117, 322, 355, 946, 433, 765, 356, 242, 114],
       [ 92, 117, 937, 322, 118, 373, 667, 451, 935, 756],
       [469, 946, 322, 480, 887, 988, 117, 950, 355, 401],
       [117, 322, 745, 355, 229, 816, 464, 753, 910, 230]])
```

4. To **make sense** of the cluster centers, see the top words associated with each cluster.

```
# Find out top words each centroid include:
print ('Top words in each cluster:')
vocab = vectorizer.get_feature_names()

for num, centroid in enumerate (top_centriods_index):
    print ("%d: %s" % (num, ",".join(vocab[i] for i in centroid)))
```

Clustering all documents:

```
Top words in each cluster:
0: time,service,did,just,work,car,customer,told,like,don
1: place,love,best,amazing,friendly,staff,ve,nice,food,delicious
2: pizza,crust,good,place,great,cheese,order,sauce,service,ordered
3: food,good,chicken,ordered,service,place,like,just,restaurant,really
4: great,food,service,place,friendly,staff,good,amazing,definitely,recommend
```

Clustering reviews from top rated restaurant:

```
Top words in each cluster:
0: food,great,service,amazing,place,buffet,worth,experience,recommend,selection
1: buffet,good,price,really,buffets,better,dishes,food,quality,selection
2: crab,legs,king,buffet,food,snow,good,section,place,great
3: time,buffet,food,good,wait,just,service,got,dinner,brunch
4: best,buffet,vegas,food,buffets,hands,quality,las,ve,selection
5: line,wait,food,long,time,worth,buffet,waiting,good,hour
6: buffet,food,seafood,good,dessert,station,like,section,try,desserts
```

5. Print out a few **sample reviews** from each cluster to see if the clustering performed well.

```
import random
n_samples = 3
for i in range(kmeans.n_clusters):
    cluster = df_top_restaurant[df_top_restaurant['cluster']==i]
    index = cluster.index.values
    sample = np.random.choice(index, n_samples)
    print('cluster', i, ':')

    for j in range(n_samples):
        print('\n', cluster['stars'][sample[j]], '----')
        print(cluster['text'][sample[j]])
    print ('=' * 100)
pd.options.display.max_colwidth = 1000
```

Dimensionality Reduction with PCA

1. **Standardize** both train X and test X.
2. **Fit** PCA on train X and **transform** both train X and test X.

```
from sklearn.decomposition import PCA

# Let's pick a n_components
n_components = 50
pca = PCA(n_components = n_components)
```

```
# Fit on training data and transform:
reduced_train_X = pca.fit_transform(standardized_train_X)
reduced_train_X.shape
```

```
(975, 50)
```

```
# Transform testing data:
reduced_test_X = pca.transform(standardized_test_X)
reduced_test_X.shape
```

```
(419, 50)
```

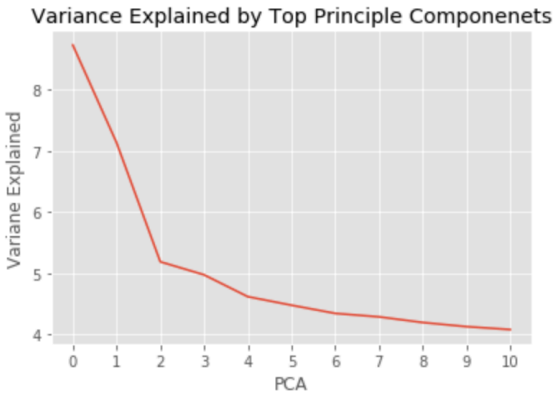
3. Look at **variance explained** by each PC and visualize to identify top PCs.

```
# The amount of variance explained by each of the selected components.
pca.explained_variance_
```

```
array([8.73036814, 7.13486184, 5.18612544, 4.97282323, 4.61521847,
       4.4752566 , 4.34140667, 4.28502882, 4.1925971 , 4.12506303,
       4.07746455, 3.99467888, 3.93698877, 3.8729586 , 3.86507533,
       3.81108962, 3.78766975, 3.75762986, 3.70728274, 3.69058867,
       3.66218064, 3.60410343, 3.57845788, 3.56414891, 3.54551493,
       3.51970428, 3.48660041, 3.48215427, 3.44140503, 3.41624945,
       3.39948145, 3.38165143, 3.35483377, 3.33209768, 3.32366198,
       3.30973666, 3.2921801 , 3.27711635, 3.25848306, 3.20530429,
       3.19423494, 3.15465126, 3.14967994, 3.14229915, 3.12395382,
       3.1029131 , 3.06179338, 3.04567791, 3.02264327, 3.01266673])
```

```
# Line plot
pca_range = np.arange(11)
pca_components = ['PCA %s' % i for i in pca_range]
plt.figure()
plt.plot(pca_range, pca.explained_variance_[:11])
xticks = plt.xticks(pca_range)
plt.xlabel('PCA')
plt.ylabel('Variane Explained')
plt.title('Variance Explained by Top Principle Componenets')
```

```
Text(0.5,1,'Variance Explained by Top Principle Componenets')
```



Similar Review Search Engine:

1. Draw a random sample (**search query**) from the review texts.
2. Use **TF-IDF** to vectorize the sample review.
3. Calculate similarity (cosine similarity) between all vectorized X and the vectorized sample.
4. Get the top similar reviews from vectorized matrix and **map them back** to the review text.

Positive/Negative Review Classifier (Logistic Regression):

1. **Standardize** vectorized train X and test X.
2. **Fit** on train and **predict** on test

```
# Build a Logistic Regression Classifier, train with standardized tf-idf vectors
from sklearn.linear_model import LogisticRegression
logistic = LogisticRegression(C=0.01)
logistic.fit(standardized_train_X, train_y)
```

3. Get **score** (**performance**) for train and test.

Not PCA transformed:

```
# Get score for training set
logistic.score(standardized_train_X, train_y)
```

```
0.9897435897435898
```

```
# Get score for test set
logistic.score(standardized_test_X, test_y)
```

```
0.7231503579952268
```

PCA transformed:

```
# Get score for training set
logistic.score(reduced_train_X, train_y)
```

```
0.8041025641025641
```

```
# Get score for test set, REMEMBER to use PCA-transformed X!
logistic.score(reduced_test_X, test_y)
```

```
0.7494033412887828
```

- No overfitting problem with PCA, and test performance better than before PCA
- Enhanced model performance by reducing overfitting

- What do you see from the training and testing scores with/without PCA?

  Not PCA preprocessed data resulted in **overfitting**. Even after tuning logistic regression regularization parameter C, the overfitting problem still persists. However, after PCA, the accuracy on training and testing are similar, and even though training performance dropped, testing performance actually improved

4. Find **top features (words)** for prediction in logistic regression model.

Positive/Negative Review Classifier (Random Forest):

1. **Standardize** vectorized train X and test X.
2. **Train** a random forest classifier with **grid search** on **cross validation** to tune for the best parameters.

```
# Build a Random Forest Classifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer, roc_auc_score, accuracy_score

# # Choose some parameter combinations to try and define scroing used for grid search
param_grid = {'n_estimators': [20,50,100,200],
              'max_features': ['auto'],
              'criterion': ['gini','entropy'],
              'max_depth': [15,20,25],
              'min_samples_leaf': [2,3,5,10,20],
              'n_jobs':[-1]
             }
forest = RandomForestClassifier()
score_type = make_scorer(accuracy_score)

# Run grid search
grid = GridSearchCV(forest, param_grid, cv=5, scoring = score_type)
grid = grid.fit(standardized_train_X, train_y)

# Re-train random forest with best parameters
forest = grid.best_estimator_

# Fit on training data
forest.fit(standardized_train_X, train_y)
```

3. Get **score** (**performance**) for train and test.

Not PCA transformed:

```
# Get score for training set
forest.score(standardized_train_X, train_y)
```

```
0.9005128205128206
```

```
# Get score for test set
forest.score(standardized_test_X, test_y)
```

```
0.7231503579952268
```

PCA transformed:

```
# Get score for training set
forest.score(reduced_train_X, train_y)
```

```
0.9969230769230769
```

```
# Get score for test set, REMEMBER to use PCA-transformed X!
forest.score(reduced_test_X, test_y)
```

```
0.7350835322195705
```

- Overfitting is still present but less than that for logistic regression (random forest more resistant to multicollinearity?). Reducing PCA did not significantly improve overfitting.

Restaurant Recommender:

1. Get relevant columns only: business_id (**item**), user_id (**user**), **rating**. The cleaned data frame:

	business_id	user_id	stars
0	--6MefnULPED_I942VcFNA	FEg8v92qx3kK4Hu4TF28Fg	4.0
1	--6MefnULPED_I942VcFNA	HPtjvIrhzAUkKsiVkeT4MA	3.0
2	--6MefnULPED_I942VcFNA	x-Gbs8sVid3yhJIoHD6Gfw	2.0
3	--6MefnULPED_I942VcFNA	7Dykd1HolQx8mKPYhYDYSg	2.0
4	--6MefnULPED_I942VcFNA	Z4VF-tv5ibkhv_kzVny0NA	1.0

2. Some users did not rate many items (<20), filter them out.

```
# Remove users without many ratings (<=20):
count = df_relavent['user_id'].value_counts()
df_filtered = df_relavent[df_relavent.groupby('user_id')['user_id'].transform('count') > 20]
df_filtered.shape
```

```
(159765, 3)
```

3. Create **utility matrix** (user by item matrix).

```
df_utility = pd.pivot_table(data=df_filtered,
                            values='stars',
                            index='user_id',
                            columns='business_id',
                            fill_value=0)
```

```
# Convert df_utility into sparse matrix
import scipy.sparse as sps
utility_mat = sps.csr_matrix(df_utility)
```

```
np.unique(utility_mat)
```

```
array([<4127x48162 sparse matrix of type '<class 'numpy.int64'>'
        with 159765 stored elements in Compressed Sparse Row format>],
      dtype=object)
```

4. **Item-item similarity recommender**:

4.1. Calculate item-item **similarity matrix** using **cosine similarity**.

```
# Similarity matrix:
item_similarity = cosine_similarity(utility_mat)
item_similarity
```

```
array([[1., 0., 0., ..., 0., 0., 0.],
       [0., 1., 0., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 1., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 0., 0., 1.]])
```

4.2. Pick a user and identify the items **already rated** by this user.

```
# Prediction on 100th user
user = 99
utility_mat[[user]]
n_items = utility_mat.shape[1] # number of items
```

```
# Items j already rated by user 100
item_rated = utility_mat[[user]].nonzero()[1]
item_rated
```

```
array([47951, 41327, 39234, 37874, 36270, 35102, 34293, 32104, 29988,
       29662, 26809, 26292, 26254, 22978, 18879, 16118, 15984, 15113,
       14194, 10880,  9123,  5212,  1509,  1265], dtype=int32)
```

4.3. Calculate this user's **predicted rating** on **unrated items**.

```
# Predict rating on unrated items:
pred = np.zeros(n_items)
for i in range(n_items):
#   relevant_items = np.intersect1d(neighborhoods[i], item_rated, assume_unique = True)
    pred[i] = (utility_mat[user, item_rated]*item_similarity[i, item_rated])/item_similarity[i, item_rated].sum()
prediction = np.nan_to_num(pred)
print(prediction)
```

```
/Users/annieyang/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:5: RuntimeWarning: invalid value encount
ered in true_divide
  """
```

```
[0.         0.         4.30679929 ... 0.         0.         0.        ]
```

4.4. Make **recommendation** for this user.

```
# Recommend top 10 restaurants:
n = 10

# Get item indexes sorted by predicted rating:
item_sorted = np.argsort(prediction)[::-1]

# Remove items that are already rated:
item_sorted_unrated = [i for i in item_sorted if i not in item_rated]

# Top 10 recommendations:
item_sorted_unrated[:n]
```

```
[31522, 38579, 34970, 16743, 13970, 16256, 29543, 31288, 47223, 29218]
```

5. **Matrix Factorization Recommender – UVD**:

5.1. **Fit** truncatedSVD (**UVD**) on the **original utility matrix**.

```
from sklearn.decomposition import TruncatedSVD
```

```
# Fit UVD on utility matrix:
n_components = 200
UVD = TruncatedSVD(n_components = n_components, n_iter = 5, random_state = 666)
UVD.fit(utility_mat)
```

```
TruncatedSVD(algorithm='randomized', n_components=200, n_iter=5,
       random_state=666, tol=0.0)
```

5.2. Get the components **U** and **V** from the decomposition. Then calculate **predicted utility matrix** from obtained U and V.

```
# Get componenets U and V:
U = UVD.transform(utility_mat) # U = M.dot(V.T)
V = UVD.components_
print(U.shape)
print(V.shape)
```

```
(4127, 200)
(200, 48162)
```

```
# Get transformed utility matrix: M = U*V
    # recall: U = M.dot(V.T), then this is M.dot(V.T).dot(V)
    # original M is transformed to new space, then transformed back
    # this is another way to understand it!
utility_mat_fitted = U.dot(V)
```

5.3. Make **prediction** on a chosen user.

```
# Get recommendation for user 100:
user = 99
n = 10

# Prediction on all restaurants:
prediction = utility_mat_fitted[100,:]

# Sorted rating:
item_sorted = np.argsort(prediction)[::-1]

# Remove items that are already rated:
item_sorted_unrated = [i for i in item_sorted if i not in item_rated]

# Top 10 recommendations:
item_sorted_unrated[:n]
```

```
[43744, 25536, 26667, 35805, 5835, 34862, 7978, 26399, 14300, 46448]
```

6. **Matrix Factorization Recommender – NMF**:

6.1. **Fit NMF** on the **original utility matrix**.

```
from sklearn.decomposition import NMF
```

```
r = 200
nmf = NMF(n_components = r)
nmf.fit(utility_mat)
```

```
NMF(alpha=0.0, beta_loss='frobenius', init=None, l1_ratio=0.0, max_iter=200,
  n_components=200, random_state=None, shuffle=False, solver='cd',
  tol=0.0001, verbose=0)
```

6.2. Get the components **W** and **H** from the decomposition. Then calculate **predicted utility matrix** from obtained W and H.

```
# Get components W and H:
W = nmf.transform(utility_mat)
H = nmf.components_
print(W.shape)
print(H.shape)
```

```
(4127, 200)
(200, 48162)
```

```
# Get transformed utility matrix:
utility_mat_fitted = W.dot(H)
```

6.3. Make **prediction** on a chosen user.

```
# Get recommendation for user 100:
user = 99
n = 10

# Prediction on all restaurants:
prediction = utility_mat_fitted[100,:]

# Sorted rating:
item_sorted = np.argsort(prediction)[::-1]

# Remove items that are already rated:
item_sorted_unrated = [i for i in item_sorted if i not in item_rated]

# Top 10 recommendations:
item_sorted_unrated[:n]
```

```
[28679, 26667, 34862, 43744, 25536, 46448, 16949, 1981, 27057, 31964]
```

Discussion:

1. What is the use case for clustering the reviews, and why would you want to classify +ve/-ve reviews?
   - Can help business to identify the top reasons (top features/words in classification) of why people give/not give 5 stars so that the restaurants can keep up with the positive aspects and improve on negative aspects.
2. What does the clustering results tell us, look for answers from the centroid and examples. How does this work help business?
   - Here K-Means did not always cluster similar opinions (e.g. positive and negative reviews are not separated by clusters), but more on contents on the review (e.g. reviews about buffet in a cluster, waiting time in another, service in another etc.)
   - If restaurants want to know about a particular aspect of their business (e.g. service), they can just read reviews from the 'service' cluster and make suggestions and improvements.

3. How do we recommend to users who haven't had many ratings in the beginning?
    - We can recommend popular restaurants to them, or when they sign up make a survey asking them about what type of restaurant they like.
4. Will PCA help improve performance? What is the disadvantage of PCA?
    - PCA helps to reduce overfitting and improve performance on testing data. However, it is often hard to interpret PCA because each PC is a linear combination of features, so there's not an easy way to transform the results back to business-related answer.