Individual Project (CS 598: Practical Statistical Learning)
Netid: pushpit2

Project Description and Summary

Dataset:

This project is based on the Kaggle's wine review dataset. The data was scraped from WineEnthusiast. As part of this project we are working with 'winemag-data-130k-v2.csv' data that contain 14 columns.

Goal:

There are following goals, in general, of this project:

1. Data Exploration: Understand the dataset and seek opportunity to apply machine learning modelling especially 'regression'
2. Data Visualization: Generate some visualizations (plots) to better understand the dataset
3. Regression Modelling: Build some (two) regression models to predict the "points" for the wines based on points rated by wine enthusiasts on a scale of 1-100.
4. Recommendation: Build a system to suggest users top-n wineries based on their explicit preferences like their wine description, price etc. As well as their implicit characteristics like their province etc,

Approach:

1. On first look at the data, I can see missing values for some of the columns (e.g. price, region_1, region2). So, the first step, is to clean the data (explained in detailed later).
2. After the data is cleaned, I created some plots to better understand the data and its distribution (details on Page 3)
3. Encode categorical variables for one of the models.
4. Done some basic NLP on description data, basically to vectorize the string (tf-idf used).
5. Build a couple of regression models (Catboost and Ridge) and perform basic hyperparameter tuning and validations done on hold out set and results are summarized appropriately.
6. Approaches used in recommendation systems are explained later.

Results: Regression results:

Regression Model	Test-RMSE (hold out set) - with optimal parameter	Test-RMSE (hold out set) - with default settings
Catboostregressor	1.5594	1.5595
RidgeCV	1.6905	1.6906
RidgeCV – with features selected based on Lasso regression	No tuning was done as this was giving worse results	2.2623

Please note: Recommendation results are on the last page

Data Processing:

Following data cleaning and processing is performed on the raw dataset:

1) Duplicate records are removed. Two rows with same description and title are removed, as this redundant information.

```
data=data.drop_duplicates(['description','title'])
data=data.reset_index(drop=True)
```

2) Dropped records with no price, as I don't want to include those rows in predictions. (One other way might be to substitute the missing value with the median wine price for wine-variety of that record, I have done that when generating recommendations, as I don't want to drop any records with proper description, but for regression models, dropping the records doesn't decrease the performance i.e. doesn't have much significant effect on test-RMSE, so in order to improve training time, I have decided to drop the rows with no price for the purpose of Regression models).

```
data=data.dropna(subset=['price'])
data=data.reset_index(drop=True)
```

3) Description column cleanup and tokenization (as the description words and lengths seems to be a good indicator of points, can also be seen through the plots later)

```
data['description']= data['description'].str.lower()
tokenizer = RegexpTokenizer(r'\w+')
words_descriptions = data['description'].apply(tokenizer.tokenize)
stopword_list = stopwords.words('english')
ps = PorterStemmer()
words_descriptions = words_descriptions.apply(lambda elem: [ps.stem(word) for word in elem
if not word in stopword_list])
data['description_cleaned'] = words_descriptions.apply(lambda elem: ' '.join(elem))
```

4) For one of the regression model (Catboostregressor), there is no need to convert categorical features, as library internally handles categorical predictors. But for other regression model (RidgeCV), I have encoded the categorical features using custom threshold based one hot encoding. For e.g, I have set a custom threshold say 1%, to control the levels for high cardinality predictor like winery, variety etc., if the value doesn't appear in at-least 1% of the records, it is grouped under 'Other_<predictor-name>'. (Can be seen in **Project1_pushpit2_model2** notebook)

```
def myOneHotEncoder(s, thresh, name):
    d = pd.get_dummies(s)
    f = pd.value_counts(s, sort=False, normalize=True) < thresh
    if f.sum() == 0:
        return d
    else:
        return d.loc[:, ~f].join(d.loc[:, f].sum(1).rename('other_' + name))
```

5) Impute missing values in Price by using the median price value for that wine variety, as generally the wines of similar variety are of similar price. One other way of imputing the price is to take median value of price for given province. (I have code for this also in notebook, but used the imputation based on variety for my calculations)

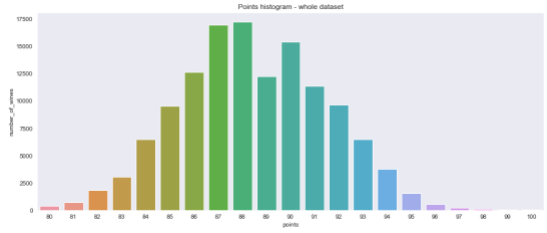
```
groupby_var_df = data.groupby('variety')
median_price_per_variety = groupby_var_df['price'].median()

def set_median_price_by_variety(variety, data):
    data.loc[(data.variety == variety) & (data.price.isna()), 'price'] = median_price_per_variety.get
(variety)

for variety in median_price_per_variety.keys():
    set_median_price_by_variety(variety, data=data)
```

There is some data processing done for recommendation system, it is explained later.

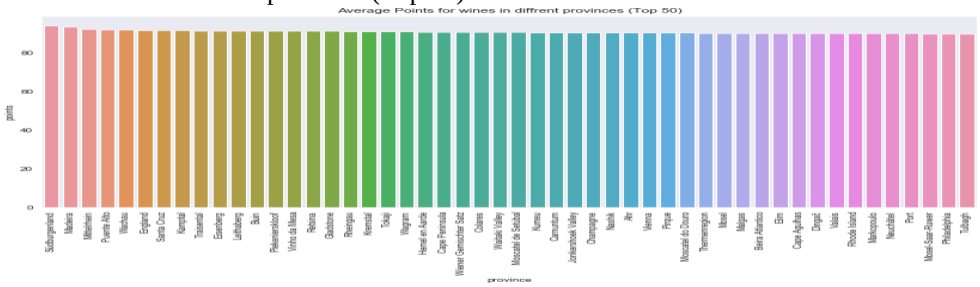
Data Visualization (*copied over only some of the plots due page limitation, other details can be seen in 'Project1_pushpit2_dataviz' notebook)*

I have created following visualization to understand the data:

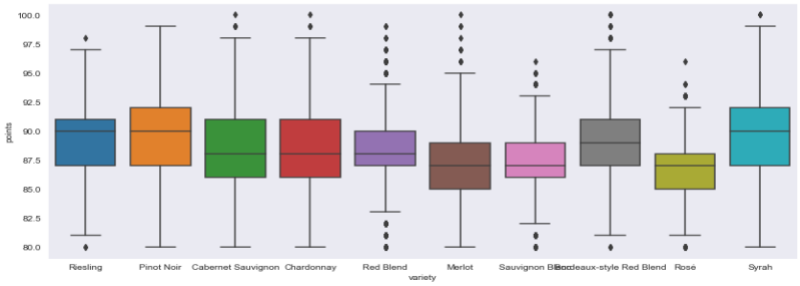
1) Number of wines for each rating point. As we can see lowest rating is 80 and highest number of wines have rating 88



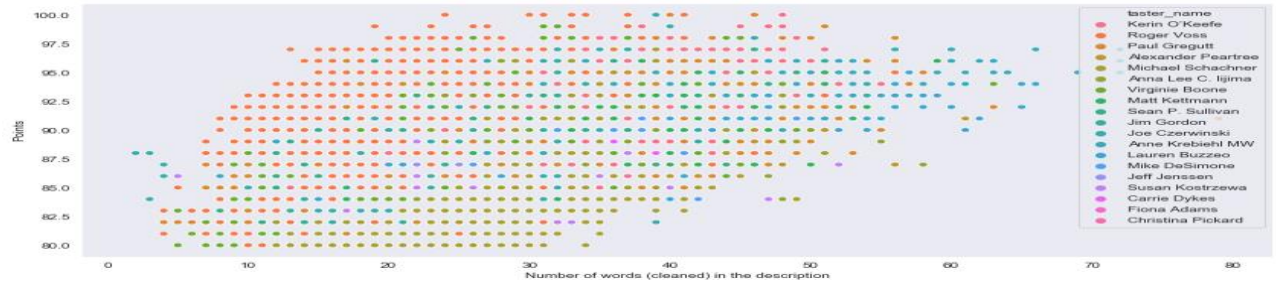
2) Average points for wines in different provinces(Top 50)



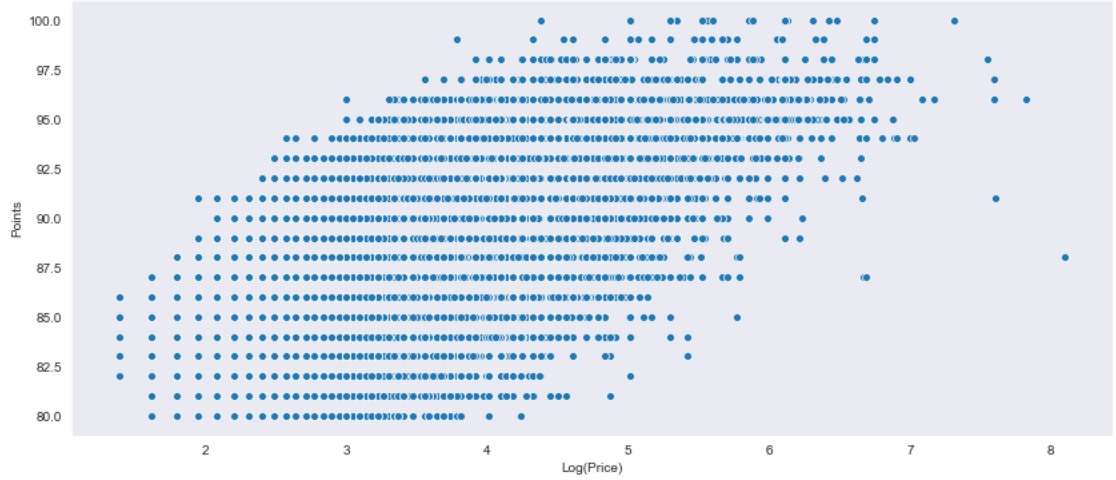
3) Box-plot for variety-point distribution (Top 10):



4) Number of cleaned (stop-words removed) to the points. It can be clearly seen as the number of word increased in the description ratings go up (so description seems to be a good indicator of points).



5) Scatter plot for price-point distribution, similar to above plot here also we can see that price is a good indicator of points. I have converted the price to log scale to reduce the effect of some outliers



Regression Models (*the source code for two regression models are in 'Project1_pushpit2_model1' and 'Project1_pushpit2_model2'*)

Model 1: Catboostregressor

CatBoost is an algorithm for **gradient boosting on decision trees**. Developed by Yandex researchers and engineers, it is the successor of the **MatrixNet algorithm** that is widely used within the company for ranking tasks, forecasting and making recommendations. It is universal and can be applied across a wide range of areas and to a variety of problems. Catboost introduces two critical algorithmic advances - the implementation of **ordered boosting**, a permutation-driven alternative to the classic algorithm, and an innovative algorithm for **processing categorical features**. (See this)

I have chosen CatBoost here, as we can see from the plots (shown on previous page) as well as looking at the dataset in general, majority of seemingly good predictors (province, variety, winery, taster_name etc.) are categorical and Catboost regression can handle the categorical variables out of the box (We don't need to provide the encoding). Also, in my experiment between XGBoost vs Catboost vs RandomForest (*I haven't included the other two in the code file for cleanness sake but training/tuning of other two is almost similar to Catboost*), Catboost outperforms the other two both in terms of the lower generalization error (test-RMSE) as well as training/tuning time.

Model setup

```
1.  model = CatBoostRegressor(random_seed = 100, loss_function = 'RMSE',  iterations=800, depth=depth, learning_rate=learning_
    rate,  l2_leaf_reg = l2_leaf_reg)
2.  model.fit(X_train, y_train, cat_features = categorical_features_indices, verbose=False, eval_set=(X_valid, y_valid))
```

Tuning – Hyperparameter (`depth`: Depth of the tree. `learning_rate`: The learning rate, used for reducing the gradient step. `l2_leaf_reg`: Coefficient at the L2 regularization term of the cost function.)

```
1.  def hyperparamtuning(X_train, Y_train, categorical_features_indices):
2.      model = CatBoostRegressor(cat_features=categorical_features_indices)
3.      grid = {'learning_rate': [0.03, 0.02, 0.01], 'depth': [4, 6, 10], 'l2_leaf_reg': [1, 3, 5,]}
4.      search_results = model.grid_search(grid, X=X_train, y=Y_train, plot=True)
5.      return search_results
6.  X, y , categorical_features_indices = prepare_dataframe(vect, data)
7.  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.10)
8.  hyperparamtuning_res = hyperparamtuning(X_train, y_train, categorical_features_indices)
```

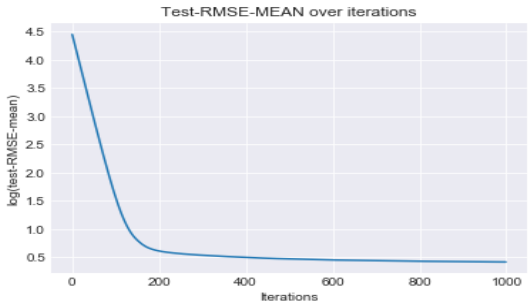
Please note: There are a lot of other parameters which can be tuned to get better model. These three seems to be similar to what I have done for XGBoost and also easy to interpret, so tuned only these for consistency shake and better comparison.

Optimal parameters for the hyperparameter tuning grid search I have done:

```
{'depth': 10, 'l2_leaf_reg': 1, 'learning_rate': 0.03}
```

Test-RMSE-Mean vs iteration during Catboost hyperparameter search (*I tried with GridSearchCV but the search over these params was taking too long (not completed in 12 hrs on my macbook) for the model we have, so I decided to rely on the in-built grid_search in Catboost library. I have seen a little improvement in test-RMSE (0.006%) when using these optimal parameters. I think to improve the model further, I have to start dropping some low importance feature and then run this hyperparameter search. I couldn't complete this experiment due to lack of time and resources.*

```
9.  plt.plot(hyperparamtuning_res['cv_results']['iterations'], hyperparamtuning_res['cv_results']['test-RMSE-mean'])
```



Validation & Result: Test-RMSE (on 10% holdout set) with these optimal params: `1.5594` **and with default settings:** `1.5595`

```
pred = tfidf_model.predict(x_test)
print(math.sqrt(mean_squared_error(pred, y_test)))
```

Interpretation of results and most significant features (in Catboost): (*Please note that the features with names ending with '_nlp' are features added by using the tf-idf vectorization of the cleaned-description string)*

```
tfidf_model.get_feature_importance(prettified=True)[0:11]
```

Predictor	CatboostImportanceFactor
price	22.713024
description_lengths	19.624077
winery	7.864938
taster_name	6.812793
taster_twitter_handle	2.923642
rich_nlp	1.529884
region_1	1.295969
beauti_nlp	1.208322
complex_nlp	1.036663
lack_nlp	0.996292
province	0.952118

The top features here are pretty much in line with my expectations as per the initial data analysis and plots (price, description_lengths, winery and taster_name are good predictors). The great part of this regression model is that it has identified some words (with positive/negative sentiments like rich/beauti/lack etc.) which indicates that description words are significant predictors, especially the ones with positive/negative sentiments. Sentiment analysis of these words can be provide better information which can improve the model further.

Model 2: RidgeCV

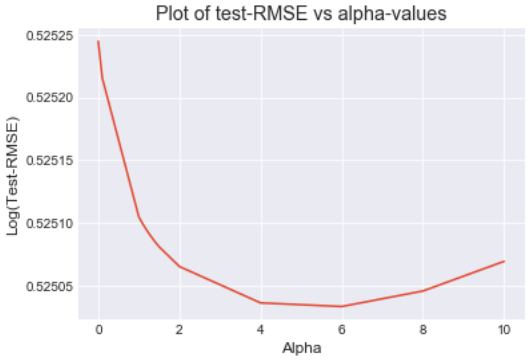
I have chosen Ridge regression as my second model here. Both Ridge and Lasso were giving me similar performance with full set of features that we used for Catboost earlier, but Ridge was slightly ahead. So, including data and information about the ridge regression here in the report (but both Ridge and Lasso regression models are generated and can be seen in the code). The major advantage of ridge regression is coefficient shrinkage and reducing model complexity. Sklearn library provides an easy way to implement both Ridge and Lasso regression. I have used 'RidgeCV' and 'LassoCV' from Sklearn library, these two already have built in Cross-validation to choose the best model.

Model setup:

```
reg = RidgeCV(cv=5, alphas=[alpha]).fit(X_train, y_train)
pred = reg.predict(X_test)
```

Tuning – Hyparameter (*alpha:* ***Regularization strength. Larger values specify stronger regularization. Alpha corresponds to C^-1 in other linear models)*

```
alphas = [0.001, 0.01, 0.1, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 2, 4, 6, 8, 10]
rmse = [0] * len(alphas)
coefs = []
for idx, alpha in enumerate(alphas):
    reg = RidgeCV(cv=5, alphas=[alpha]).fit(X_train, y_train)
    pred = reg.predict(X_test)
    rmse[idx] = math.sqrt(mean_squared_error(y_test, pred))
    coefs.append(reg.coef_)
```



Optimal parameters : *Alpha = 6.0*

Validation & Result: Test-RMSE (on 10% holdout set) with these optimal params: `1.6905` **and with default settings :** `1.69063`

Interpretation of result:
Following are the coefficient of some of the features used for the different alphas tried as part of the tuning

price	description_lengths	Argentina	Armenia	Australia	alphas
0.011012	0.063593	0.057757	-0.898784	1.330893	0.001
0.011012	0.063594	0.059208	-0.890109	1.331544	0.01
0.011013	0.063597	0.072368	-0.811747	1.337051	0.1
0.011024	0.063627	0.130527	-0.429317	1.343762	1
0.011025	0.06363	0.133065	-0.40771	1.342002	1.1
0.011026	0.063633	0.13522	-0.388128	1.34001	1.2
0.011027	0.063636	0.137044	-0.370299	1.337822	1.3
0.011028	0.06364	0.138579	-0.353997	1.335466	1.4
0.011029	0.063643	0.139864	-0.339034	1.332968	1.5
0.011035	0.063659	0.14344	-0.279519	1.318953	2
0.011056	0.063725	0.136609	-0.162115	1.254752	4
0.011076	0.063791	0.120364	-0.112541	1.192423	6
0.011095	0.063858	0.103137	-0.085278	1.135827	8
0.011114	0.063924	0.086764	-0.068091	1.084853	10

Again, here we can see that the coefficient of the some of the important features like ‘price’ ‘description_lengths’ etc are sufficiently large, also, Ridge regression model gave high values for coefficient for Australia (country) and Armenia (country) proving the point that country is one of the good predictors for point. Please note that some provinces (like California) also has larger coefficients (couldn’t show here due to lack of space).

I have also run Lasso regression (haven’t tuned it, as RidgeCV was giving the better performance), but for interpretation sake, the non-zero coefficient predictors in Lasso regression are:
['price', 'description_lengths', 'balanc_nlp', 'black_nlp', 'complex_nlp', 'concentr_nlp', 'dark_nlp', 'drink_nlp', 'eleg_nlp', 'full_nlp', 'miner_nlp', 'rich_nlp', 'ripe_nlp', 'spice_nlp', 'structur_nlp']

But when I ran the RidgeCV with just these predictors, the test-RMSE went up, hence I kept all the original predictors in the Ridge model. Even though Lasso regression wouldn’t help much in feature selection here, but it reinforce our hypothesis that descriptions, price and positive/negative sentiment words in description have big impact on points. This information will definitely come in handy when more advanced modelling techniques like NeuralNetworks etc. will used on this dataset.

Conclusion:
These simple regression models are predicting the points reasonably well with little bit of tuning (although out of the box library functions with default parameters also worked well). I feel that Catboost or similar models are much better here than Ridge/Lasso/Linear regression, because Boosting/Bagging models are easy to interpret hence easier to tune to our needs. Also, better vectorization (may be word2vec) and sentiment analysis of description might further improve the performance.

Recommendation system (*The code can be seen in 'Project1_pushpit2_Q2_recomendation' notebook)*

The system we are trying to build here, basically suggest wineries to user based on their given wine-variety/price and query-string ("Fruity") etc. My understanding is this system is a combination of data filtering based on deterministic inputs like wine-variety, price point and then use similarity metrics to find the wineries producing wines similar to the one user requested. There can be a lot of ways to achieve this, building collaborative filtering model, content based filtering model , using advanced recommendation system libraries like LightFM , but in the interest of time and keeping the things simple yet effective, I have created a simpler model, explained below:

Approach:

1. Filtering to get target dataset:
    a. Filtered the dataset to only have records with wine-variety like '*pinot*noir*' and price less than 20$.
2. Created a tf-idf matrix of the cleaned-descriptions (tokenized, stop-words removed and stemmed).
3. Cosine-similarity based query matching:
    a. For the given query string like "Frutity", created the tf-idf vector using the same vectorizer which is used for the dataset earlier.
    b. Calculated the cosine similarity of the query string vector with dataset vector matrix.
    c. Aggregate the cosine-similarity score for each of the winery.
    d. Also, calculate the aggregate similarity (weighted by points/price ratio = quality index) – This can generate order within the local pinot-noir dataset, based on the quality index, with wineries having better points/price ratio are generally better than other and because we are multiplying it with the similarity score, hence higher value signifies the winery has better quality index and also produces more wines matching the query, showing general capability of that winery to produce such wines.
    e. Save the list of the wineries in the descending order of the aggregated cosine-similarity score (I have taken aggregate score here, because if for same winery we get multiple hits, this winery should come up in our recommendations as this winery generally produce many different wine matchings to the user query and hence can be more useful for the user).
4. I have also collected the aggregated information for each of the winery (from complete dataset).
    a. **'points/price':** Average points/price –this can be used to pick between two wineries, as the winery with this metric higher can show that they get consistently better points then other, which might show overall quality of the winery.
    b. **'desc_length_cleaned_mean' :** As established while building regression models the description length is correlated to points, hence wineries with high description length on average are better rated compared to others.
    c. **'numberofTasters':** Higher the number of taster, better vetted is the winery (I haven't used this much in this here in project, but I think with some more complex model, this might be pretty useful).
    d. **'province':** Above three are basically collected to alter the sorting order of the recommendation. But this is purely collected to do filtering after we got the top recommendations. For e.g a user is from California, and generally like non-local wines, we can use this to filter our sorted recommendation list.
    e. **"below_20":** Count of wine with price below 20$
    f. **"below_20_prop":** Proportion of wines produced by winery priced below 20$ to the total number of wines produced by the winery. When we use this (in conjunction with other factors) to sort the recommendations, we can present this as supporting fact that we are recommending him with the best wineries (based on similarity as well as other criteria) which produce higher proportion of their wines in user's price range.
5. Once above steps are done, the system generates recommendations, it can be configured to generate according to following criteria:
    a. Simple recommendation based on aggregate cosine-similarity metric:
        i. ['Vignerons de Buxy', 'Manuel Olivier', 'Jean-Luc and Paul Aegerter', 'Nuiton-Beaunoy', 'Joseph Drouhin']
    b. Recommendation based on Quality(points/price) weighted aggregate similarity:
        i. ['Vignerons de Buxy', 'Manuel Olivier', 'McManis', 'Barton & Guestier', 'Jean-Luc and Paul Aegerter']
    c. Using the aggregated winery dataset collected in step 4, sorting the top 25 recommendations generated by weighted aggregated similarity (see b):
        i. Based on global quality index (points/price) rating:
            1. ['McManis', 'Gabriel Meffre', 'Domaine Fribourg', 'Barton & Guestier', 'Bougrier']
        ii. Based on global description lengths:
            1. ['Pali', 'Kendall-Jackson', 'Hahn', 'Acrobat', 'Willm']
        iii. Based on global proportion of wines priced below 20:
            1. ['Domaine Fribourg', 'McManis', 'Bougrier', 'Gabriel Meffre', 'RouteStock']

There are numerous ways to generate recommendations and as the question left it pretty open ended, I have presented multiple different recommendations. I found that **weighted-aggregated-similarity** based recommendations are most straightforward. They are easier to explain in order to convince the user as the user query hits maximum number of wines produced by the wineries in the list which are then weighted by wineries' quality index, in simpler words we are recommending the wineries to the user which produces more wines/exactly-similar-wines as per user queries and at the same time have higher quality ratio compared to their peers. As I have collected the province information in the aggregated wine data, we can filter out some recommendations based on any province criterion, if needed. It can be easily done in the code. Excluded from the report due to lack of space.

Conclusion:

All the methods used here are capable of providing recommendation with sufficient and in-depth support data. Also, as mentioned in the question, all these methods first of all use similarity score (which will play an even bigger role when query string is larger, may not be so much in one example given here) and then take variety of metrics to sort the matched wineries (not just average of points! as restricted in question).

Recommendation systems are essentially data-filtering and sorting systems. This is a very simple search-based recommendation system, utilizing the strength of user query matches to filter out and sort wineries for recommendations. I would definitely like to explore more advanced recommendation systems where the data matrix we have collected for wineries can be utilized to do collaborative filtering or other advance modelling. Also, we should be collecting user preference data (to build user profiles) and various queries, if we want to provide more targeted and custom recommendations. One quick improvement that can be done here is to use glove-vectors (word embeddings) to vectorize the descriptions for cosine-similarity step as that might alleviate issue with having queries with similar meaning but different words. Other than this we might need more information around the expectation of this recommendation system to improve it further.