COMP 4211 Project Report (Spring 2019)

CHEN, Yifei (20328874), LI, Siqi (20328056)

Part 0: Project Introduction

Our project applies the dataset about Google app store, which include googleplaystore.csv and googleplaystore_user_reviews.csv, to implement this project. Our objectives of the projects are:

1. Predicting sentiments for reviews;
2. Predicting ratings for apps.

The project is under the environment of python3. We will present the details of each part in the following sections

Part 1: Predicting sentiments for reviews

Inspired by what we learned about classification model in class, we decide to use the decision tree model to solve this problem. Furthermore, we also apply some advanced decision tree model for this question to make some comparison.

1. Data Preprocessing

   Firstly, we load the data from googleplaystore_user_reviews.csv via panda:

```
# For Reviews Dataset
# Reading data from CSV
review_data = pd.read_csv('googleplaystore_user_reviews.csv')
print(review_data.shape)
review_data.head(20) # briefly shows the orginal dataset
```

(64295, 5)

	App	Translated_Review	Sentiment	Sentiment_Polarity	Sentiment_Subjectivity
0	10 Best Foods for You	I like eat delicious food. That's I'm cooking ...	Positive	1.000000	0.533333
1	10 Best Foods for You	This help eating healthy exercise regular basis	Positive	0.250000	0.288462
2	10 Best Foods for You	NaN	NaN	NaN	NaN
3	10 Best Foods for You	Works great especially going grocery store	Positive	0.400000	0.875000
4	10 Best Foods for You	Best idea us	Positive	1.000000	0.300000
5	10 Best Foods for You	Best way	Positive	1.000000	0.300000
6	10 Best Foods for You	Amazing	Positive	0.600000	0.900000

Then, what we need are valid rows (with effective Translated_Review and sentiment). And we change the value of sentiment to integers for convenience of future training:

```
# Draw out all the valid reviews and corresponding sentiments
review_data=pd.concat([review_data.Translated_Review,review_data.Sentiment],axis=1)
number_of_reviews = len(review_data.Sentiment)


for i in range(number_of_reviews):
    #print (review_data.Sentiment[i])
    if review_data.Sentiment[i] == 'Positive':
        review_data.Sentiment[i] = 0
    elif review_data.Sentiment[i] == 'Negative':
        review_data.Sentiment[i] = 2
    elif review_data.Sentiment[i] == 'Neutral':   # Neutral sentiment. label with 1  (middle)
        review_data.Sentiment[i] = 1

review_data.dropna(inplace=True)  # For drop nan values. It makes confuse for our model.
review_data.head(20)
```

	Translated_Review	Sentiment
0	I like eat delicious food. That's I'm cooking ...	0
1	This help eating healthy exercise regular basis	0
3	Works great especially going grocery store	0
4	Best idea us	0
5	Best way	0
6	Amazing	0
8	Looking forward app,	1
9	It helpful site ! It help foods get !	1
10	good you.	0

To get a better input feature set, we also need to regularize the translated review, which contains:

a. Delete '!', '.', '"', '...', ':', '#' and so on;

b. Delete the words which are unnecessary, which are absolutely without sentiment (eg. 'a', 'for', 'are', 'the', and so on);

c. Break the sentence in to words (features);

d. Lower cases;

e. Lemmatization (eg. Liked -> like, plays -> play).

```
import re
import nltk
import nltk as nlp
from nltk.corpus import stopwords as stopwords
from nltk.stem.porter import PorterStemmer
from nltk.tokenize import word_tokenize

#length = review_data.shape[0]
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('wordnet')

stop_words = stopwords.words('english')
#generate the processed review list
#prepara for the futher input matrix
review_list=[]
for i in review_data.Translated_Review:
    review=re.sub("[^a-zA-Z]"," ",i)
    review=review.lower()
    review=nltk.word_tokenize(review)
    review = [word for word in review if word not in stop_words]
    lemma=nlp.WordNetLemmatizer()
    review=[lemma.lemmatize(word) for word in review]
    review=" ".join(review)
    review_list.append(review)

print(review_list[:10])
print(len(review_list))
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/yifeichen/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to /Users/yifeichen/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package wordnet to
[nltk_data]     /Users/yifeichen/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
['like eat delicious food cooking food case best food help lot also best shelf life', 'help eating healthy exercise reg
ular basis', 'work great especially going grocery store', 'best idea u', 'best way', 'amazing', 'looking forward app',
'helpful site help food get', 'good', 'useful information amount spelling error question validity information shared fi
xed star given']
37427
```

There are too many features in total (total number of unique words) actually. We don't want a model which are time costing and likely overfit, so we only select 2000 features in input. Also, each review will be represented as a vector of these feature, the value is the number of the occurrence of corresponding word.

```
from sklearn.feature_extraction.text import CountVectorizer
# I found that there are total 17577 unique words, it is not efficient
# and time wasting for constructing the future decision tree
#Therefore, i decide to set the max_features to be 2000
count_vectorizer=CountVectorizer(max_features= 2000)
#Get the matrix about # of apperance of words of each review,
input_map=count_vectorizer.fit_transform(review_list).toarray()
features_list=count_vectorizer.get_feature_names()
print(len(input_map[1,:])) # verify we select 2000 features (word)
#print(features_list)
```

```
2000
```

```
# Then, prepare the correct format of data for a decision tree
# Label vector
labels= review_data['Sentiment'].values[:]
review_labels=[]
for sentiment in review_data.Sentiment:
    review_labels.append(sentiment)

print(labels)
#print(review_labels)
print(len(review_labels)) # verify the number of labels equals the number of rows of input
```

```
[0 0 0 ... 2 0 2]
37427
```

Lastly, with train-test split:

```
# Data split
from sklearn.model_selection import train_test_split
review_train, review_test, labdel_train, label_test = train_test_split(input_map,review_labels,test_size=0.2,
                                                                        random_state=4211)
```

2. <u>Method 1: Single decision tree</u>

For this method, we import the decision tree classifier from scikit-learn. And we construct a class for my model, which includes initialization, train, test, and confusion matrix functions:

```
from sklearn import tree
import matplotlib.pyplot as plt
import time

from sklearn import metrics

class decision_tree_classifier:
    def __init__(self,criterion,depth):
        self.cri = criterion
        self.dep = depth
        self.decision_tree = tree.DecisionTreeClassifier(criterion = criterion,max_depth = depth)
        print ("construct decision tree with criterion: %s   and max_depth: %s " %(criterion,depth))


    def train(self,text,label):
        text_train, text_val, label_train, label_val = train_test_split(text,label,test_size=0.2,random_state=4211)

        start_time_tree = time.time()
        self.decision_tree = self.decision_tree.fit(text_train,label_train)
        val_predict = self.decision_tree.predict(text_val)
        val_correct = 0
        for i in range (len(label_val)):
            if val_predict[i] == label_val[i]:
                val_correct +=1
        val_accuracy = float(val_correct)/len(label_val)

        time_cost = round(time.time()-start_time_tree , 3)
        print("decision tree training time: %s" %time_cost)
        print("decision tree validation accuracy: %s" %val_accuracy)

        self.confusion_matrix(val_predict,label_val,"validation")

        return (val_accuracy,time_cost)


    def test(self,text,label):
        predict = self.decision_tree.predict(text)
        n_corrects = 0
        for i in range (len(label)):
            if predict[i] == label[i]:
                n_corrects +=1
        accuracy = float(n_corrects)/len(label)
        print("decision tree test accuracy: %s" %accuracy)

        self.confusion_matrix(predict,label,"test")
        return accuracy
```

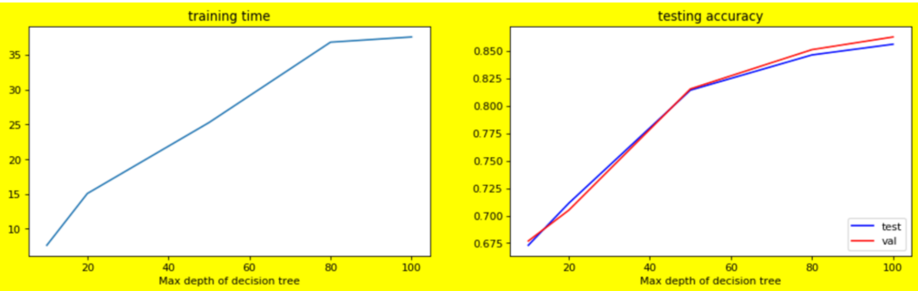
```
    def confusion_matrix(self,predict,true,val_or_test):
        confusion = metrics.confusion_matrix(true, predict)
        plt.figure()
        plt.imshow(confusion, interpolation='nearest', cmap='Pastel1')
        plt.title('Criterion:%s   Max_depth:%s    %s' %(self.cri,self.dep,val_or_test), size = 15)
        plt.colorbar()
        plt.ylabel('Actual label', size = 15)
        plt.xlabel('Predicted label', size = 15)
        tick_marks = np.arange(3)
        plt.xticks(tick_marks, ["Positive", "Nertual","Negative"], size = 10)
        plt.yticks(tick_marks, ["Positive", "Nertual","Negative"], size = 10)
        plt.tight_layout()
        width, height = confusion.shape
        for x in range(width):
            for y in range(height):
                plt.annotate(str(confusion[x][y]), xy=(y, x),
                horizontalalignment='center',
                verticalalignment='center')
```

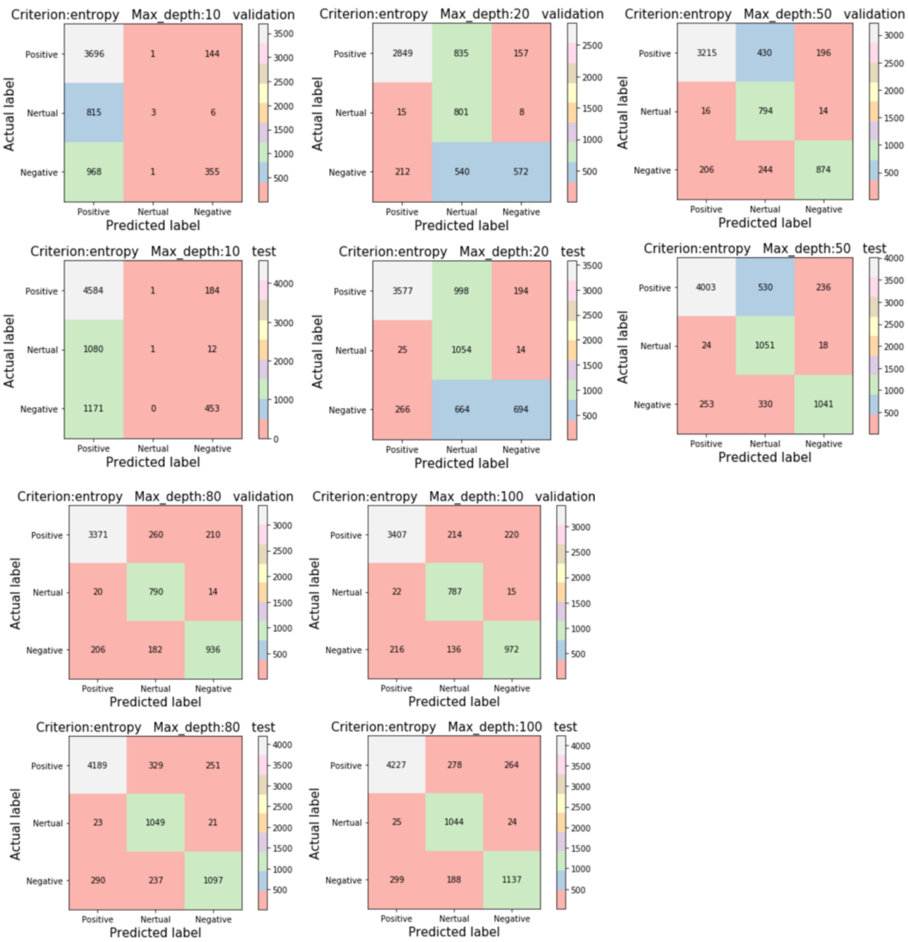
Then what we do is to set different max-depth of tree, see the results (the criterion can also be changed, here we only use "entropy" as the way to calculate information gain):

```
# Now I set different max_depth for the model and compare the result
depth_list = [10,20,50,80,100]
train_time_list = []
test_acc_list = []
val_acc_list = []

for i in range(len(depth_list)):
    decision_tree = decision_tree_classifier('entropy',depth_list[i])
    val_accuracy,time_cost = decision_tree.train(review_train,label_train)
    accuracy = decision_tree.test(review_test,label_test)

    val_acc_list.append(val_accuracy)
    train_time_list.append(time_cost)
    test_acc_list.append(accuracy)
    print()

plt.figure(num=None, figsize=(15, 4), dpi=80, facecolor='yellow')
plt.subplot(1,2,1)
plt.plot(depth_list,train_time_list)
plt.title("training time")
plt.xlabel("Max depth of decision tree")

plt.subplot(1,2,2)
l1, = plt.plot(depth_list,test_acc_list,color = 'b',label='test')
l2, = plt.plot(depth_list,val_acc_list,color = 'r',label='val')
plt.legend(handles=[l1, l2], labels=['test', 'val'],loc='lower right')
plt.title("testing accuracy")
plt.xlabel("Max depth of decision tree")

plt.show()
```

Below are the results:

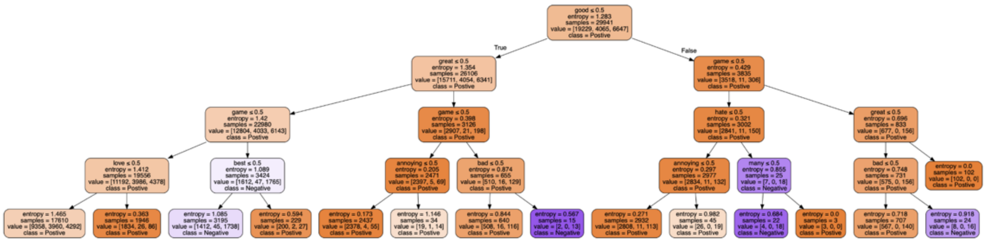
```
construct decision tree with criterion: entropy   and max_depth: 10
decision tree training time: 7.634
decision tree validation accuracy: 0.6769076640507597
decision tree test accuracy: 0.6729895805503606

construct decision tree with criterion: entropy   and max_depth: 20
decision tree training time: 15.073
decision tree validation accuracy: 0.7049590916680581
decision tree test accuracy: 0.7113278119155757

construct decision tree with criterion: entropy   and max_depth: 50
decision tree training time: 25.259
decision tree validation accuracy: 0.8153281015194523
decision tree test accuracy: 0.8141864814320064

construct decision tree with criterion: entropy   and max_depth: 80
decision tree training time: 36.812
decision tree validation accuracy: 0.8510602771748205
decision tree test accuracy: 0.8462463264760887

construct decision tree with criterion: entropy   and max_depth: 100
decision tree training time: 37.559
decision tree validation accuracy: 0.8625813992319252
decision tree test accuracy: 0.8559978626769971
```





We find that the accuracy increases with the increasement of the depth of the tree, while it also takes more time. Finally, when we set the depth to be 100, the accuracy is close to 86% Beyond that, we also find how to visualize the tree, in case that the tree with the depth of 100 is to large. Here we only visualize a small part.

```
# a sample of visualization
# need the library graphviz, which may not be installed in other computers
from IPython.display import Image
import pydotplus

decision_tree_visual = tree.DecisionTreeClassifier(criterion = 'entropy',max_depth = 4)
decision_tree_visual.fit(review_train,label_train)
dot_data = tree.export_graphviz(decision_tree_visual, out_file=None,
                         feature_names=features_list,
                         class_names=["Postive","Nertual","Negative"],
                         filled=True, rounded=True,
                         special_characters=True)
graph = pydotplus.graph_from_dot_data(dot_data)
Image(graph.create_png())
```



3. Method 2: Random Forest

Random forest is an advanced ensemble learning method which use multiple trees and subsets of dataset. Here we import the model from sckit-learn and use GridSearchCV to tune the parameter set.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
#from sklearn import cross_validation, metrics


param_set= [{'n_estimators': [50], 'max_depth':[10]},
            {'n_estimators': [100], 'max_depth':[10]},
            {'n_estimators': [150], 'max_depth':[10]},
            {'n_estimators': [50], 'max_depth':[20]},
            {'n_estimators': [100], 'max_depth':[20]},
            {'n_estimators': [150], 'max_depth':[20]},
            {'n_estimators': [50], 'max_depth':[50]},
            {'n_estimators': [100], 'max_depth':[50]},
            {'n_estimators': [150], 'max_depth':[50]},
            {'n_estimators': [100], 'max_depth':[100]}]

random_forest = RandomForestClassifier(criterion = 'gini',n_jobs = 5)
gsearch1= GridSearchCV(random_forest,param_set,cv = 5)

gsearch1.fit(review_train,label_train)
```

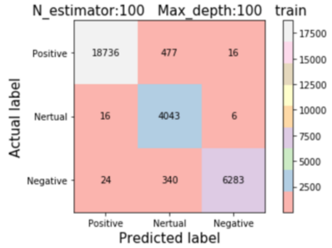
Here is the report:

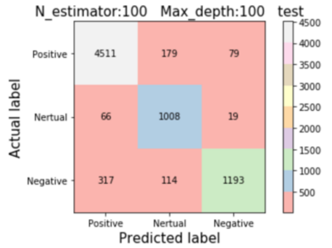
```
print("Grid scores on for parameters set:")
print()
means = gsearch1.cv_results_['mean_test_score']
#average = average + clf3.cv_results_['mean_test_score']
stds = gsearch1.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, gsearch1.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r"
          % (mean, std * 2, params))
print()
print("Best parameters set found:")
print(gsearch1.best_params_)
```

```
Grid scores on for parameters set:

0.648 (+/-0.005) for {'max_depth': 10, 'n_estimators': 50}
0.646 (+/-0.002) for {'max_depth': 10, 'n_estimators': 100}
0.646 (+/-0.002) for {'max_depth': 10, 'n_estimators': 150}
0.691 (+/-0.012) for {'max_depth': 20, 'n_estimators': 50}
0.691 (+/-0.008) for {'max_depth': 20, 'n_estimators': 100}
0.691 (+/-0.007) for {'max_depth': 20, 'n_estimators': 150}
0.775 (+/-0.010) for {'max_depth': 50, 'n_estimators': 50}
0.776 (+/-0.008) for {'max_depth': 50, 'n_estimators': 100}
0.776 (+/-0.005) for {'max_depth': 50, 'n_estimators': 150}
0.887 (+/-0.005) for {'max_depth': 100, 'n_estimators': 100}

Best parameters set found:
{'max_depth': 100, 'n_estimators': 100}
```

Then we choose the best parameter set to train the model as well as testing it:

```
Random forest training time with optimal parameters set found: 81.84
Random forest test accuracy with optimal parameters set found: 0.8966069997328346
```





Here we see that it takes more time than single decision tree with the same max_depth, but it shows higher accuracy.

One interesting we find is that in the cases of small max_depth, the model does not perform better than single decision tree. We guess this is because the model suffers from bagging of attributes especially when constructing shallow trees (less judgements and low accuracy). Even though there is an effect of 'ensemble', the model cannot perform pretty well.

4. <u>Method 3: Gradient Boosting Decision Tree</u>

Besides using the above two method, we also try another advanced decision tree method to implement the task, which focuses on reducing the error to optimize the model.

Unfortunately, GDBT takes lots of time when we set the depth to be large. For this method, we just set depth to be 5 or 10.

```python
# Method 3 GBDT Classification
from sklearn.ensemble import GradientBoostingClassifier

'''
param_set_gbdt= [{'n_estimators': [20], 'max_depth':[20],'learning_rate':[0.1]},
                {'n_estimators': [30], 'max_depth':[20],'learning_rate':[0.1]},
                {'n_estimators': [40], 'max_depth':[20],'learning_rate':[0.1]},
                {'n_estimators': [20], 'max_depth':[50],'learning_rate':[0.1]},
                {'n_estimators': [30], 'max_depth':[50],'learning_rate':[0.1]},
                {'n_estimators': [40], 'max_depth':[50],'learning_rate':[0.1]},
                {'n_estimators': [20], 'max_depth':[20],'learning_rate':[0.05]},
                {'n_estimators': [30], 'max_depth':[20],'learning_rate':[0.05]},
                {'n_estimators': [40], 'max_depth':[20],'learning_rate':[0.05]},
                {'n_estimators': [20], 'max_depth':[50],'learning_rate':[0.05]},
                {'n_estimators': [30], 'max_depth':[50],'learning_rate':[0.05]},
                {'n_estimators': [40], 'max_depth':[50],'learning_rate':[0.05]}]
'''

# since i find that the gbdt cost lots of time, i decide to choose a much easier set
param_set_gbdt= [{'n_estimators': [10], 'max_depth':[5],'learning_rate':[0.1]},
                {'n_estimators': [20], 'max_depth':[5],'learning_rate':[0.1]},
                {'n_estimators': [10], 'max_depth':[10],'learning_rate':[0.1]},
                {'n_estimators': [20], 'max_depth':[10],'learning_rate':[0.1]}]
gbdt = GradientBoostingClassifier(loss='deviance',criterion='friedman_mse',
                                  random_state = 4211,max_features='sqrt',
                                  subsample=0.6 ,  min_samples_leaf=40,min_samples_split = 80)
gsearch2= GridSearchCV(gbdt,param_set_gbdt,cv = 5)

gsearch2.fit(review_train,label_train)

# max_features='sqrt'  subsample=0.8   min_samples_leaf=20
```

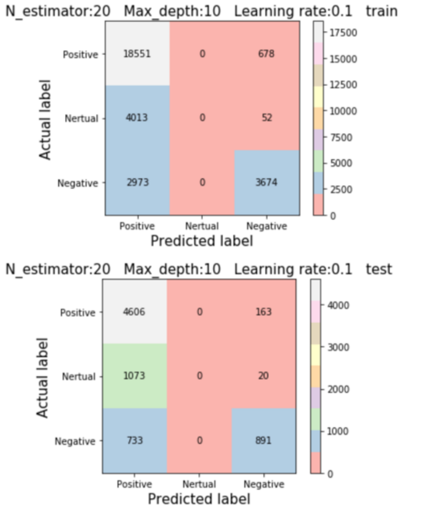
And the result from gridSearchCV

```python
print("Grid scores on for parameters set:")
print()
means = gsearch2.cv_results_['mean_test_score']
#average = average + clf3.cv_results_['mean_test_score']
stds = gsearch2.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, gsearch2.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r"
          % (mean, std * 2, params))
print()
print("Best parameters set found:")
print(gsearch2.best_params_)
```

```
Grid scores on for parameters set:

0.643 (+/-0.002) for {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 10}
0.649 (+/-0.003) for {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 20}
0.654 (+/-0.011) for {'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 10}
0.678 (+/-0.018) for {'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 20}

Best parameters set found:
{'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 20}
```

Then we apply this parameter set:

```
GBDT training time with optimal parameters set found: 687.85
GBDT with optimal parameters set found: 0.734304034197168
```



It is obvious that GDBT takes significantly more time (687.85 as shown in figure) than the former two methods. However, what surprises us is that it performs much better than the former two methods in the same max-depth (67% vs 65% vs 73%).

Part 2: Predicting ratings of the APP

Inspired by some news that some APP companies hire people to rate them higher, we hope to use the other features to predict the rating of the APP and find a better model to help marketing research. In this part, we use two models, namely, linear regression and KNN model.

Here are all the packages that we used:

```
import re
import sys
import time
import datetime
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import preprocessing
from sklearn.neighbors import KNeighborsRegressor
from sklearn.ensemble import RandomForestRegressor
import pandas as pd
import numpy as np
import seaborn as sns
from sklearn import metrics
from sklearn.model_selection import train_test_split
import random
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_validate
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_digits
from sklearn.model_selection import learning_curve
from sklearn.model_selection import ShuffleSplit
from sklearn import datasets, linear_model
from sklearn.metrics import log_loss
sys.setrecursionlimit(100000) #Increase the recursion limit of the OS
%matplotlib inline
from impyute.imputation.cs import fast_knn
```

1. <u>Data Preprocessing</u>

Here are the variables in the file of GoogleAppStore.csv:

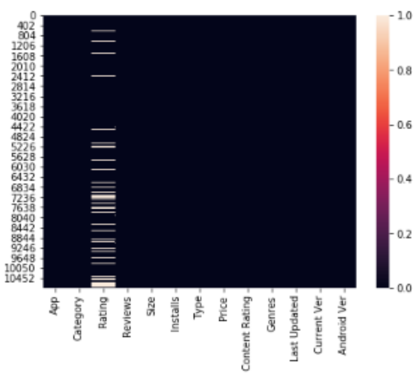
Variables	Type	Explanation
App	string	Application name
Category	string	Category the app belongs to
Rating(Y)	decimal	Overall user rating of the app (as when scraped)
Reviews	integer	Number of user reviews for the app (as when scraped)
Size	string	Size of the app (as when scraped)
Installs	string	Number of user downloads/installs for the app (as when scraped)
Type	string	Paid or Free
Price	string	Price of the app (as when scraped)
Content Rating	string	Age group the app is targeted at - Children / Mature 21+ / Adult
Genres	string	An app can belong to multiple genres (apart from its main category). For eg, a musical family game will belong to Music, Game, Family genres.
Last Updated	string	Date when the app was last updated on Play Store (as when scraped)
Current Ver	string	Current version of the app available on Play Store (as when scraped)
Android Ver	string	Min required Android version (as when scraped)

In this part, we did two steps. First is to deal with the missing data problem and the second is to transfer the type of the variables to numeric ones for KNN model training.

1.1 After loading the data, we find the summary of the missing values. Most of the missing values are Rating, which is our dependent variable. Thus, we tried to two methods to deal with the problem. One is to delete all the missing value (around 1400) and the other is to use median imputation trick to fill the blank. And we build df2 and df data framework respectively.

```
# load the data
df = pd.read_csv('../input/googleplaystore.csv')
df2 = pd.read_csv('../input/googleplaystore.csv')

# Exploring missing data and checking if any has NaN values
plt.figure(figsize=(7, 5))
sns.heatmap(df.isnull())
# df.isnull().any()
# df.isnull().sum()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f438a1fdc18>
```

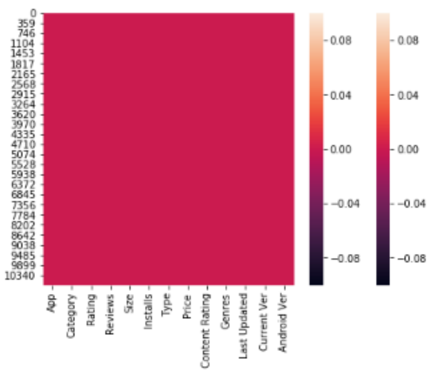


```python
# clean all non numerical values & unicode characters before the cleaning
replaces = [u'\u00AE', u'\u2013', u'\u00C3', u'\u00E3', u'\u00B3', '[', ']', "'"]
# Because there are too many missing values for the variable, we consider several ways to deal with it
# 1.  Filling the rating with the median of all the APP
df['Rating'] = df['Rating'].fillna(df['Rating'].median())
# df['Rating'] = df['Rating'].fillna(0.0)
# 2. Drop the value in the training part as the total data set is large enough

# Missing value of Current Ver
for i in replaces:
    df['Current Ver'] = df['Current Ver'].astype(str).apply(lambda x : x.replace(i, ''))
regex = [r'[-+|/:/;(_)@]', r'\s+', r'[A-Za-z]+']
for j in regex:
    df['Current Ver'] = df['Current Ver'].astype(str).apply(lambda x : re.sub(j, '0', x))

df['Current Ver'] = df['Current Ver'].astype(str).apply(lambda x : x.replace('.', ',',1).replace('.', '').replace(',', '.',1)).astype(float)
df['Current Ver'] = df['Current Ver'].fillna(df['Current Ver'].median())

# Missing value of Current Ver
for i in replaces:
    df2['Current Ver'] = df2['Current Ver'].astype(str).apply(lambda x : x.replace(i, ''))
regex = [r'[-+|/:/;(_)@]', r'\s+', r'[A-Za-z]+']
for j in regex:
    df2['Current Ver'] = df2['Current Ver'].astype(str).apply(lambda x : re.sub(j, '0', x))

df2['Current Ver'] = df2['Current Ver'].astype(str).apply(lambda x : x.replace('.', ',',1).replace('.', '').replace(',', '.',1)).astype(float)
df2['Current Ver'] = df2['Current Ver'].fillna(df2['Current Ver'].median())

# delete the remaining missing value
df.dropna(inplace = True)
df2.dropna(inplace = True)
```

After processing the code, we double checked the missing value and we can see there is no missing value left for both df and df2 data framework.

```python
# Checking missing data
plt.figure(figsize=(7, 5))
sns.heatmap(df.isnull())
sns.heatmap(df2.isnull())
df.isnull().any()
df.isnull().sum()
df2.isnull().any()
df2.isnull().sum()
```

```
App               0
Category          0
Rating            0
Reviews           0
Size              0
Installs          0
Type              0
Price             0
Content Rating    0
Genres            0
Last Updated      0
Current Ver       0
Android Ver       0
dtype: int64
```



1.2 Transfer the type of the variables in to numerical ones

For the following steps, in order to process the data in the (KNN) machine learning algorithms, we need to first convert them from strings to numbers. Here are the brief steps that we use:

a. Delete the characteristics such as M, k, +, ... and so on

b. Keep the unit the same. We convert the size all into the units of M.

c. Transfer the category values into a set of dummies.

d. Transfer all the strings into int/float

```
# Clean Rating
df['Rating'] = df['Rating'].astype(int).fillna(0)

# Categories: string -> integer -> dummies
# Cleaning Categories into integers
CateString = df["Category"]
cateVal = df["Category"].unique()
cateValCount = len(cateVal)
category_dict = {}
for i in range(0,cateValCount):
    category_dict[cateVal[i]] = i
df["CateCount"] = df["Category"].map(category_dict).astype(int)
df['CateCount'] = df['CateCount'].astype(int).fillna(0)

# Making the list for spliting training and test sets
cateList = df['Category'].unique().tolist()
cateList = ['cate_' + word for word in cateList]
df = pd.concat([df, pd.get_dummies(df['Category'], prefix='cate')], axis=1)

# transfer Size to integer
df['Size'].fillna(0, inplace=True)
# Convert kbytes to Mbytes
k_indices = df['Size'].loc[df['Size'].str.contains('k',na=False)].index.tolist()
converter = pd.DataFrame(df.loc[k_indices, 'Size'].apply(lambda x: x.strip('k')).astype(float).apply(lambda x: x / 1024).apply(lambda x: round(x, 3)).astype(str))
df.loc[k_indices,'Size'] = converter
# delete "M" and change the "varies with device" to 0
df['Size'] = df['Size'].apply(lambda x: x.strip('M'))
df[df['Size'] == 'Varies with device'] = '0'
df['Size'] = df['Size'].astype(float).fillna(0.0)
```

```
#Cleaning no of installs classification to the integer
df['Installs'] = df['Installs'].apply(lambda x : x.strip('+').replace(',', ''))
df['Installs'] = df['Installs'].astype(int).fillna(0)

#Converting Type classification into binary
def cateype(types):
    if types == 'Free':
        return 0
    else:
        return 1
df['Type'] = df['Type'].map(cateype)

#Cleaning of content rating classification
RatingL = df['Content Rating'].unique()
RatingDict = {}
for i in range(len(RatingL)):
    RatingDict[RatingL[i]] = i
df['Content Rating'] = df['Content Rating'].map(RatingDict).astype(int)

#dropping of unrelated and unnecessary items
df.drop(labels = ['Last Updated','Current Ver','Android Ver','App'], axis = 1, inplace = True)

#Cleaning of genres
# count the number of the genres as genres_c
GenresL = df.Genres.unique()
GenresDict = {}
for i in range(len(GenresL)):
    GenresDict[GenresL[i]] = i
df['Genrescount'] = df['Genres'].map(GenresDict).astype(int)

#Cleaning prices dealing with $$$
def price_clean(price):
    if price == '0':
        return 0
    else:
        price = price[1:]
        price = float(price)
        return price

df['Price'] = df['Price'].map(price_clean).astype(float)
```

```
# convert reviews to numeric
df['Reviews'] = df['Reviews'].astype(int)

print("df")
df.info()
df.head()
```

Same operations are conducted to df2.

2. Training the model to predict the rating

2.1 Split the data into training data and test data

In our project, we use Ranking as Y and other parameters as X. Here is the name and the order of the variables ($X_1, X_2, X_3 \ldots X_{40}$) and the rest of them are all category dummies.

X:

```
Data columns (total 40 columns):
Reviews                         8669 non-null int32
Size                            8669 non-null float64
Installs                        8669 non-null int32
Type                            8669 non-null int64
Price                           8669 non-null float64
Content Rating                  8669 non-null int32
CateCount                       8669 non-null object
cate_ART_AND_DESIGN             8669 non-null object
cate_AUTO_AND_VEHICLES          8669 non-null object
cate_BEAUTY                     8669 non-null object
cate_BOOKS_AND_REFERENCE        8669 non-null object
cate_BUSINESS                   8669 non-null object
cate_COMICS                     8669 non-null object
cate_COMMUNICATION              8669 non-null object
cate_DATING                     8669 non-null object
cate_EDUCATION                  8669 non-null object
cate_ENTERTAINMENT              8669 non-null object
cate_EVENTS                     8669 non-null object
cate_FAMILY                     8669 non-null object
cate_FINANCE                    8669 non-null object
cate_FOOD_AND_DRINK             8669 non-null object
cate_GAME                       8669 non-null object
cate_HEALTH_AND_FITNESS         8669 non-null object
cate_HOUSE_AND_HOME             8669 non-null object
cate_LIBRARIES_AND_DEMO         8669 non-null object
cate_LIFESTYLE                  8669 non-null object
cate_MAPS_AND_NAVIGATION        8669 non-null object
cate_MEDICAL                    8669 non-null object
cate_NEWS_AND_MAGAZINES         8669 non-null object
cate_PARENTING                  8669 non-null object
cate_PERSONALIZATION            8669 non-null object
cate_PHOTOGRAPHY                8669 non-null object
cate_PRODUCTIVITY               8669 non-null object
cate_SHOPPING                   8669 non-null object
cate_SOCIAL                     8669 non-null object
cate_SPORTS                     8669 non-null object
cate_TOOLS                      8669 non-null object
cate_TRAVEL_AND_LOCAL           8669 non-null object

cate_VIDEO_PLAYERS              8669 non-null object
cate_WEATHER                    8669 non-null object
dtypes: float64(2), int32(3), int64(1), object(34)
```

We split the data into training set and testing set randomly with the ratio of 4:1 for both df and df2. Here is the code to do it:

```
#Integer encoding
X = df.drop(labels = ['Category','Rating','Genres','Genrescount'],axis = 1)
Y = df.Rating
train_X, test_X, train_Y, test_Y = train_test_split(X, Y, test_size=0.20, random_state = 10)
# change the train_Y into int
train_Y = train_Y.astype(int)

X_2 = df2.drop(labels = ['Category','Rating','Genres','Genrescount'],axis = 1)
Y_2 = df2.Rating
train_X2, test_X2, train_Y2, test_Y2 = train_test_split(X_2, Y_2, test_size=0.20, random_state = 10)
train_Y2 = train_Y2.astype(int)
```

2.2 Linear Regression Model with Machine Learning.

In this part, we use package sklearn to simulate the linear regression model of both datasets.

```python
from sklearn import datasets, linear_model
# import time

#Integer encoding
X = df.drop(labels = ['Category','Rating','Genres','Genrescount'],axis = 1)
Y = df.Rating
train_X, test_X, train_Y, test_Y = train_test_split(X, Y, test_size=0.20, random_state = 10)
# define linear regression function
train_Y = train_Y.astype(int)

#start to calculate time
# t1 = time.process_time()

linear_model = linear_model.LinearRegression()
linear_model.fit(train_X,train_Y)
Results_linear = linear_model.predict(test_X)
accuracy_linear = linear_model.score(test_X,test_Y)
print(accuracy_linear)
#end of the counting time
# elapsed = (time.process_time() - t)
```

And here is the summary of the prediction scores for df and df2. As df has more data and there is a possibility of overestimation of the model(as we use median to imuputate), df-score is reasonable to be a bit higher than df2.

	Df	df2
Scores	0.8909545779324055	0.8895438549421302

And the coefficients of the models are

```
Array1([ 1.50095393e-08,  4.04776442e-04, -1.00368831e-10,  5.64456313e-02,
        -8.52792123e-04,  1.73335607e-04,  1.77019458e-01,  3.92565141e+00,
         3.61221886e+00,  3.56379021e+00,  3.39633660e+00,  3.18165777e+00,
         2.91597366e+00,  2.76091902e+00,  2.45396653e+00,  2.59238334e+00,
         2.22738048e+00,  2.34638672e+00,  6.53794864e-01,  1.83008306e+00,
         1.64743068e+00,  8.73540109e-01,  1.55862330e+00,  1.35762081e+00,
         1.28975414e+00,  9.53192024e-01, -2.04679535e+00,  5.10878464e-01,
        -1.65735618e+00, -1.01738305e+00, -6.24581691e-01, -7.50583416e-02,
        -9.11517948e-01,  2.17817112e-01,  3.74920930e-01, -1.95027140e-01,
        -7.15134488e-01, -4.85635995e-01, -1.65703228e+00, -1.26279713e+00])
Array2 ([ 1.32047150e-08,  9.57315902e-04,  1.62036632e-11,  6.49847289e-02,
        -8.22135046e-04,  2.15862994e-03,  1.75212427e-01,  3.93711052e+00,
         3.65375638e+00,  3.62417650e+00,  3.34258626e+00,  3.08019270e+00,
         2.87869358e+00,  2.73823847e+00,  2.33950757e+00,  2.59837083e+00,
         2.26402765e+00,  2.37100539e+00,  6.63118082e-01,  1.84486859e+00,
         1.54567372e+00,  8.78664008e-01,  1.56672156e+00,  1.32626721e+00,
         1.24208863e+00,  9.07304812e-01, -2.04944264e+00,  4.59255172e-01,
        -1.64986969e+00, -9.78979564e-01, -5.94404928e-01, -3.74283075e-02,
        -9.21340102e-01,  2.54165027e-01,  4.12564411e-01, -2.06124415e-01,
        -7.30563216e-01, -4.89426779e-01, -1.64892974e+00, -1.23270231e+00])
```

We can see from the coefficients that the categories play an important role in the rating of the APP. It could because people using specific APP has specific personality (strict or casual); or it could because certain kinds of APP are easier to improve.

2.3 K-Nearest Neighbors (KNN)

We first look at the 17 closest neighbors and compare the accuracy with two model:

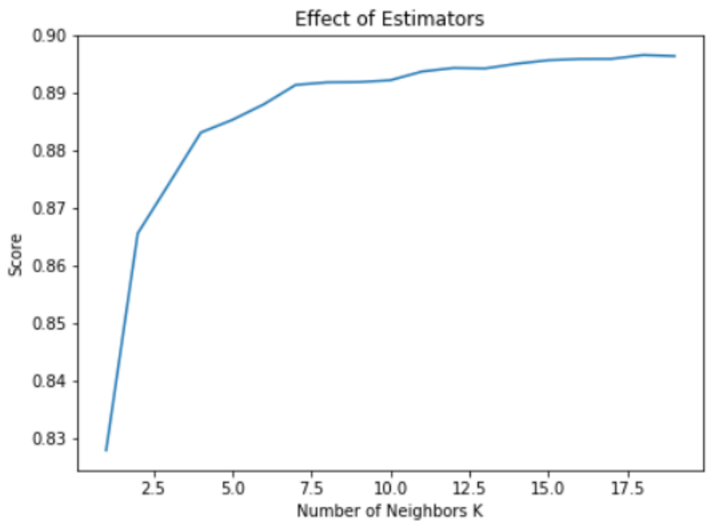
```
# Look at the 17 closest neighbors
KNN_model = KNeighborsRegressor(n_neighbors=17)
# # Find the mean accuracy of knn regression using X_test and y_test
KNN_model.fit(train_X, train_Y)

Results_KNN = KNN_model.predict(test_X)
accuracy_KNN = KNN_model.score(test_X,test_Y)
print('Accuracy: ' + str(np.round(accuracy_KNN*100, 2)) + '%')

# Calculate the mean accuracy of the KNN model

# accuracy_KNN = KNN_model.score(test_X,regression_test_Y)
# print('Accuracy: ' + str(np.round(accuracy*100, 2)) + '%')
# Try different numbers of n_estimators
n_neighbors = np.arange(1, 20, 1)
scores = []
for n in n_neighbors:
    KNN_model.set_params(n_neighbors=n)
    KNN_model.fit(train_X, train_Y)
    scores.append(KNN_model.score(test_X,test_Y))
plt.figure(figsize=(7, 5))
plt.title("Effect of Estimators")
plt.xlabel("Number of Neighbors K")
plt.ylabel("Score")
plt.plot(n_neighbors, scores)
```

	Df	df2
Scores	0.8959	0.8922

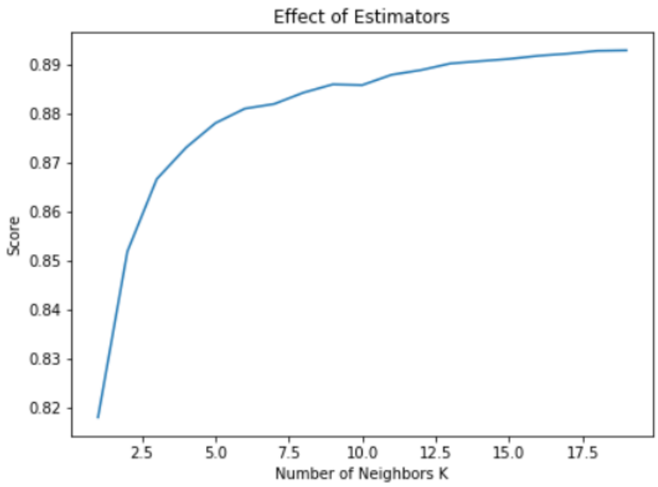
The similar argument could be conducted as the above.

Then, we conducted several other numbers of neighbors to re-run the code. Here is the graph of their scores:

df dataset:



df2 dataset:



2.4 Compare different models

Linear regression is an example of a parametric approach because it assumes a linear functional form for f(X). On the other hand, K-Nearest Neighbors (KNN), which is a non-parametric method.

Linear regression methods

1. Advantages
   1.1 Easy to fit. One needs to estimate a small number of coefficients.
   1.2 Easy to interpret the relationship.
2. Disadvantages
   1.1 They make strong assumptions about the form of f(X).
   1.2 Suppose the true relationship is far from linear, then the resulting model will provide a poor fit to the data, and any conclusions drawn from it will be suspect.

KNN models

1. Advantages
   1.1 They do not assume an explicit form for f(X), providing a more flexible approach.
2. Disadvantages
   2.1 They can be often more complex to understand and interpret
   2.2 If there is a small number of observations per predictor, then parametric methods then to work better.

Also, from the result of our code, we could find that the results are kind of similar. We would recommend to use dataset df and linear regression model to interpret which factors are more important to raise the ranking; we would recommend KNN model for model prediction.

Workload Distribution

Part 1: CHEN, Yifei.
Part 2: LI, Siqi.
Video: CHEN, Yifei; LI, Siqi.
Report: CHEN, Yifei; LI, Siqi.