Text Mining with sklearn — A case study of Amazon Unblocked Mobile Phones dataset

Haocheng Li (A53240824), Minghao Han(A53246183),
Sha Liu (A53239717), Yaxu Dai (A53237815)

I. Abstract

Rating analysis and prediction have drawn a lot of research interests in previous studies, and recent studies have shown that machine learning techniques achieved better performance than traditional statistical ones. This report applies linear regression as baseline approach, and several text mining models to an *Unlocked Mobile Phones* dataset. Experimental results show that the text mining models outperform the baseline method.

II. Introduction

Ratings prediction is highly related to recommendation systems. A lot of research into this problem has already been done. Three different approaches have emerged, including content-based methods, collaborative methods, and hybrid methods [1]. Content-based methods predict user ratings using the item's features and the user's affinities, while collaborative methods predict ratings using the ratings of other, similar users. Hybrid methods combine the two approaches mentioned before [2]. More recently, Artificial Neural Networks have been utilized to predict user ratings. In addition to the increasing methods, more and more datasets are provided for research. Marovic used *IMDb* database and found that the probabilistic latent semantic analysis achieved the best predictions [2]. Tushar Joshi applied random forest classifier to *Amazon Find Food Reviews* for sentiment classification in 5 classes (5 ratings with value 1 to 5) [3].

In this assignment, we use a dataset of *Unlocked Mobile Phones* for rating predictions, provided by PromptCloud. Preetish Panda from PromptCloud made use of the same dataset, created word clouds for both positive and negative reviews, which were determined by ratings, and did sentiment analysis based on the reviews [4]. Kamath analyzed word frequencies under different ratings and brands [5]. RohitM also created word clouds for several brands and predicted ratings using reviews with some methods including *Decision Tree Classifier, SGD Classifier* [6]. The highest accuracy he got was 0.6723 with Logistic Regression. Eliot BarrilText applied Multilayer Perceptron (MLP), Long Short Term Memory networks (LSTM) and Convolutional Neural Networks (CNN) to the tf-idf matrix of reviews, where CNN resulted in the highest accuracy of 0.6782 [7].

We apply linear regression with some features excluding reviews as baseline method. After that, we use some text mining models to predict ratings by classifications. Without using neural networks, we find it is possible to reach a similar accuracy as Eliot BarrilText got. In this report, we begin with a brief overview of the dataset, and how we do the preprocessing. Baseline method is then described, followed by a detailed description of our text mining models. Finally, we present our results and conclusions.

III. Dataset

A. Data Description

The data set we make use of is from Kaggle, which is called *Amazon Reviews: Unlocked Mobile Phones*. 400,000 reviews and ratings of unlocked mobile phones sold on *Amazon* were extracted. The fields provided in the data set include *Product Name, Brand Name, Price, Rating, Reviews* and *Review Votes*, which are very useful for rating-prediction tasks.

Figure 1 and Figure 2 demonstrate the statistics of *Rating* and *Price*. A linear regression model

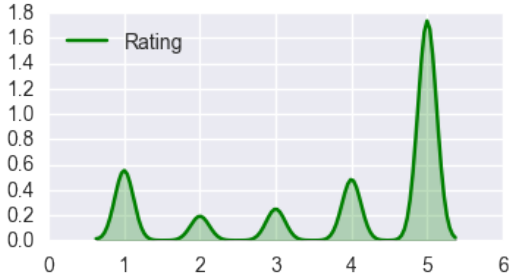

Fig. 1: Kernel density estimate of Ratings

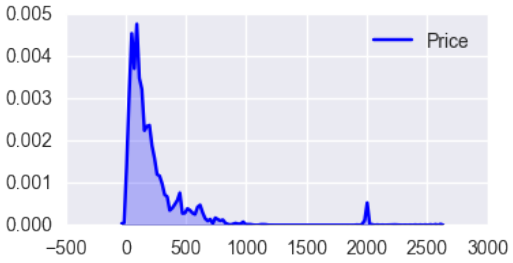

Fig. 2: Kernel density estimate of Price

fitted with price and ratings is shown in Figure 3, from which we can find that the ratings are higher with the price. The basic statistics and properties of the data set are shown in Table 1.

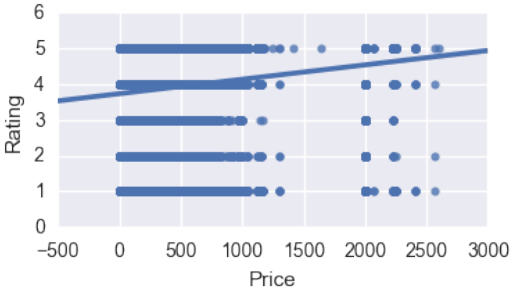

Fig. 3: Price-Ratings relationship (linear regression)

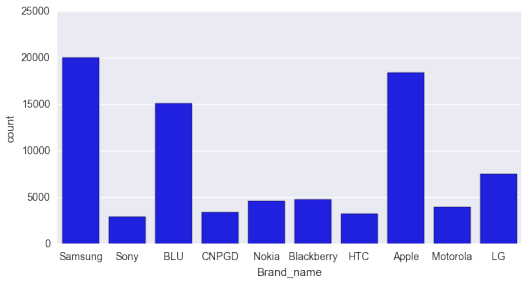

Fig. 4: Top 10 Brands

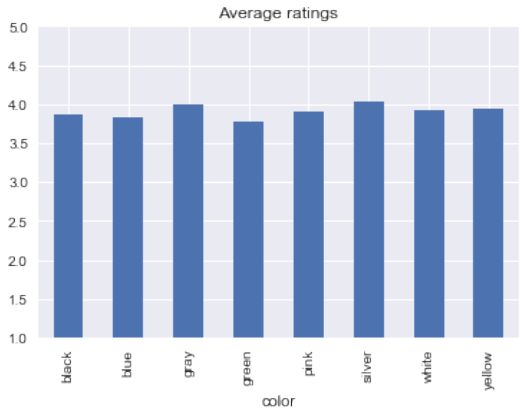

Fig. 5: Color-Ratings relationship (average)

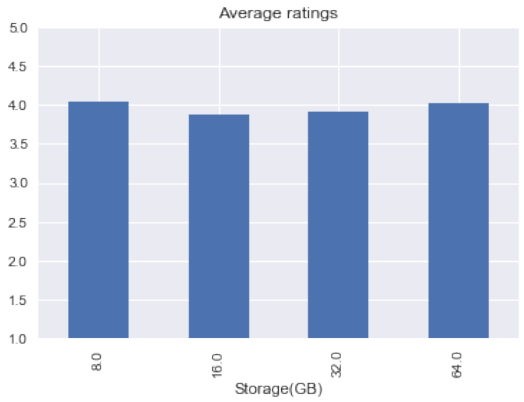

Fig. 6: Storage-Ratings relationship (average)

Statistics	Price	Rating
count	334335	334335
mean	222.59	3.82
std	283.14	1.54
min	1.73	1.0
25%	75.41	3.0
50%	139.0	5.0
75%	264.1	5.0
max	2598.0	5.0

TABLE I: Basic statistics and properties

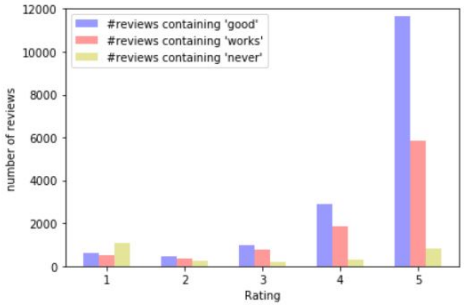

Fig. 7: Features extracted from review text

B. Data Prepossessing

The *Unlocked Mobile Phones* dataset is organized and grouped by *Product Name*. We randomly shuffled the dataset and selected the first 100,000 records for the task. By probing into these records, we found the *Brand Name* column containing some abnormal value. Taking the brand 'Samsung' as example, it also included 'Samssung', 'Samsung Korea', 'Samsung Korea LTD', 'Samsung International', 'Samsung Galaxy', 'Samsybg Galaxy' and so on, which were all obviously 'Samsung'. To make our dataset consistent, we unified the brand names to the normal ones. We also deleted some records with *Product Name* as 'product not found'. Moreover, we extracted the brands for all the phones in the dataset, and chose the top ten brands with the largest number of entries. The top ten brands in our dataset are *Samsung, Apple, Blu, LG, Blackberry, Nokia, Motorola, Cnpgd, HTC* and *Sony*, as shown in Figure 4. Among the 100,000 records we selected, there are 85,550 reviews of phones made by these ten manufactures. In the *Product Name* column, a lot of features were listed in every single cell. We extracted six features including 'color', 'storage', 'carrier', 'warranty', 'unlock', and 'model' for analysis. Figure 5 and Figure 6 show the relationships between color and ratings, and storage and ratings correspondingly. In the *Reviews* column, there were some emoji made of punctuation characters in *Reviews*, such as ':)' and '=)', which would be removed during text mining. To retain their meanings, we manually replaced those happy-face emoji with the word 'good'. Similarly, there were some reviews like 'A+++', '100%' with ratings 5. We converted them to the word 'perfect' for text mining.

To generate unigrams and bigrams from *Reviews* for text mining, we utilized *nltk* module to filter stop words from the text, and made use of *string* to remove punctuation and capitalization. We also deleted pure number unigrams or bigrams, for their meanings were ambiguous. After that, we computed *term frequency* and *inverse document frequency* for both unigrams and bigrams. Figure 7 demonstrates three unigrams and their frequency of occurrence in the review text.

IV. Methodology

A. Predictive Task

Our models give us the predictions of the rating of the integer values among $[1, 2, 3, 4, 5]$. We choose the accuracy of predictions to evaluate the performance of the predictors. If the rating predicted is exactly identical to the rating in the test set, the prediction is successful. Otherwise, we get a wrong prediction. The accuracy of the model is the ratio of the number of correct predictions to the total number of predictions.

To obtain the useful features, the results of data prepossessing are analyzed and some most significant features are selected. For example, it seems that there is no obvious difference between the average ratings of silver phones and red phones. However, word 'good' occurs more frequently in the reviews with high rating than in the reviews with low ratings. By analyzing the prepossessed data, we can find that some features are relatively more significant than others, such as the warranty, the unlocked statue, the model of the phone as well as the brand of the phone. In contrast, the color and the storage of the phone are not good features.

More clues on which are the useful features exist in the text reviews. The occurrence of some unigrams, bigrams or trigrams such as 'excellent'

or 'best phone ever' indicate a positive attitude and they always come along with good reviews. Similarly, some sub-sequences of the text reviews such as 'disappointing' reveal the negative attitude and customers who give these reviews are prone to rate a low score for the phone. More precsely, we can tell from Fig. 7 such that when rating is '5', the number of reviews containing "good" or "works" prone to have high value and conversely, the reviews containing "never" when rating is '1' are more than those with ""never" for other rating.

Hence based on the analysis above, a simple baseline for text mining is given by checking whether "good", "well", "works", "perfect" or "excellent" has appeared in the review text, if so we predict the rating as "5" and if not the rating is "3". However, the baseline works far from satisfactory with accuracy = 0.345 and more complicated model is given in the "Text Mining Model" section.

Now first let us introduce a predictive model by simply using the phone's properties.

B. Baseline

A baseline model is designed to solve the problem using the basic methodology and techniques introduced in class. Since the task is a real-value prediction problem, the linear regression is chosen as the model for the baseline.

The main challenge for building the linear regression model is selecting the features used to train the model. Intuitively, the price of the phone can be a proper feature because the prices influence the ratings to some extent.

Other features may exist in the description of the properties and the text reviews. In this linear regression baseline, the features are generated from the descriptions of the properties. The features in the text reviews are discovered in the text mining models. We chose 6 most important features from the text descriptions of the properties. The features including color, storage, carrier, warranty, unlock, and phone-type. The code-book for these features is shown in the Appendix A.

A function named `feature()` is designed to explore the features from both the training set and the test set. It returns a feature vector contains the the offset value '1', some categorical features from the property analysis, and a real-value feature of the price. Most of the property features are categorical features which indicate whether the phones under has the specific property. For example, A phone of 32GB storage will have a sub-vector of storage property $[0, 0, 0, 1, 0, 0]$ because '32GB' is the fourth elements in the storage feature vector.(The order of the storage features is ['NA', '8GB', '32GB', '64GB', '128GB']). Finally, we get a matrix of features and use the matrix and the ratings to train our linear regression model.

The linear regression assumes a predictor of form,

$$y = X\theta \tag{1}$$

We want to minimize the mean square error,

$$MSE = \frac{1}{N}\|y - X\theta\|_2^2 \tag{2}$$

We also need to avoid overfitting by introducing a regularizer with a constant λ. So our objective function for the minimization becomes,

$$\underset{\theta}{\operatorname{argmin}} \frac{1}{N}\|y - X\theta\|_2^2 + \lambda\|\theta\|_2^2 \tag{3}$$

The gradient descent method is used in the optimization. The basic idea of gradient descent is described below.

1) Initialize θ randomly
2) $\theta := \theta - \alpha f'(\theta)$
3) Repeat (b) until convergence

We use the function from `scipy` library to perform the optimization. The function `f` and `fprime` are built based on the objective function above. The outputs of our predictor are real-valued predictions of rating ranged from -3 to 8. (Actually we want some values between 0 and 5). We normalize the value to five discrete integer values of rating, `[1, 2, 3, 4, 5]`.

The train procedure is performed on the training set of 50,000 data. We test the predictor trained by this baseline model on a test set of 35,550 data. The baseline model shows an accuracy of `0.541603375527`.

C. Text Mining Model

1) Feature Representation: There are several popular text feature representations which perform quite differently with respect to various situations. In our task we chose between N-gram, TF-IDF due to their well-proved good performance in semantic analysis and text classification. Our task was to predict users rating given their reviews where it can

be considered as a classification problem. Notice for example, "good phone" and "bad phone" imply opposite user attitudes and may lead to two extreme ratings, "1" and "5". Under such situation, we may cluster words and terms into different rating categories in order to make further prediction. However, what needs to be mentioned is we may lose some correlations among these categories, say their orders. For example, "good phone" and "best phone" both indicate higher ratings than that implied by "bad phone" and they may fell into the same cluster. Nevertheless, "best phone" should clearly imply much higher rating and probably belongs to even higher rating cluster. The improper classification is attributed to that we ignore a hidden relationship, "bad" $\rightarrow$ "good" $\rightarrow$ "best". Secondly, observed that no matter N-gram or TF-IDF are used, the document-feature matrix will be extremely sparse and we may consume rather large computing power to extract few information from reviews. Hence, we considered to implement a singular matrix decomposition for dimension reduction before process to classification.

a) Unigram, Bigram and Trigram: The basic idea to use N-gram as feature representation is by first selecting a constant number of most popular N-grams (Given an upper bound m, N can be any number smaller than m), and then train a classification model with the term frequency of each term as input. To scale down the complexity of the problem we only adopted m $\leq$ 3. By analyzing the patterns in the unigram, bigram and trigram, we found trigram (table 1: trigram with term counting larger than 100) seemed inapplicable in our case. The main two reasons are:

- Trigrams with relative high counting are so few that they could be ignored in the feature vector given the fact that the total number of unigram and bigram with counting larger than 100 is 1791.
- Almost all trigram with high counting has similar or equivalent term as bigram, for example, "phone works great" $\rightarrow$ "works great", "put sim card" $\rightarrow$ "sim card" etc. Hence, counting once more on those term can only slightly increase accuracy but largely exhaust the computing resources.

Trigram	Counting
phone works great	323
best phone ever	198
great phone price	160
phone works well	157
phone great price	141
great phone great	138
like brand new	120
put sim card	116
great battery life	113
really like phone	112
micro sd card	112
would recommend phone	108
phone live ever	105
battery life good	102

TABLE II: Trigram with counting $\geq$ 100

With similar reason we only fetched the bigrams with the counting larger than 100. The final representation includes 4000 terms with 404 bigrams and 3696 unigrams.

b) Term Frequency(TF) & Term Frequency-Inverse Document Frequency(TF-IDF): In most cases TF-IDF is superior to TF because we always intend to find how "rare" the term appears among documents instead of merely their overall frequency. However, most bigrams and trigrams have extremely low TF-IDF which may cause feature unbalance. Hence, we adopted two methods, first was by calculating the TF-IDF of the 4000 terms given above as the feature representation; the second was by enumerating the TF of 30000 unigrams and bigrams (Notice the total number of unigrams and bigrams are 60000 and we simply selected the first half most popular terms).

c) Latent Semantic Analysis(LSA): As mentioned above the feature vectors are sparse and we intended to apply the dimension reduction to extract the most valuable information. It can be considered as finding sub-classes of these unigrams and bigrams. For example, "good", "well" and "good phone" have similar TF-IDF over document sets, and after reduction they may be replaced by a dummy class "GOOD PHONE" in the feature representation. Latent Semantic Analysis helps us to find such low-rank approximation of

the document-word matrix. Besides, it can also filtered out some anecdotal terms like ”55”, ”8”, and ”phone” etc. Hence, it is also a de-noisified matrix. Finally, each document has a unique k-dimension feature which are fed into the classification model. In practice, either using LSA or not was given a trial for perforamce comparison.

2) Classification: What is different from our previous homework is that the given data set is extremely large, hence we need to find a classification model with high scalability and relatively fine predictions. Both LinearSVM and LSA come with high performance one by splitting data into small data set and second by applying dimension reduction.

a) HMM: Some literature talked about using Hidden Markov Model for text mining problem like Kupiec [8], Adams, Beling and Cogill [9]. However, we don't think this model is suitable for text classification for two reasons. First, HMMs are ubiquitous tool for modelling time series data which satisfies Markov property. However, the text in our case does not hold such property. Secondly, HMMs are always used as a probability representation while our target is to train classification model on term weight but not on the probability.

b) Support Vector Machine: LinearSVM is a SVM with linear kernel. Considering large scale of data set linear SVM appears favorable because the bagging algorithm can be applied by slicing the data and largely reduce the computing complexity. Sacrificing the accuracy a bit linear SVM gives rather good result within a short time.
Gaussian Kernel Another SVM but with relatively complicated kernel. By far due to the limited computing power, we only tried Gaussian SVM with low-rank representation of original data(LSA). However, the performance is not satisfying due to the overfitting problem.

c) Random Forest: Random forest is a popular ensemble algorithm in classification problem which adopts similar strategy as bagging algorithm by training only on a small data set each time. In practice, we bootstrapped the sample first and considered all features when looking for a best split. The result is presented as below for different sample leaf size.

d) K-Nearest-Neighbor(KNN): Based on the fact that the reviews containing similar term(unigram, bigram, trigram) may result in the same rating, KNN becomes available with respect to the ”TF-IDF similarity” among different reviews. In practice, we attempted n_neighbors = [3, 1000, 5000, 10000, 20000] and we found that KNN with small parameter is inclined to overfitting, when n_neighbor = 3, training accuracy = 0.8346, test accuracy = 0.3251, while with the increase of neighbors, the model works normally, when n_neighbor = 20000, the training accuracy = 0.5471, test accuracy = 0.5363.

3) Optimization: As mentioned before to address the scalability problem we introduced LinearSVM and to increase ”information density” for each document we applied Latent Semantic Analysis. Besides, we compared the performance among SVM, random forest and KNN where SVM defeats others.

V. Result & Conclusion

The text mining model shows the best accuracy of `0.694` with the model: tf_idf(4000 data) + LinearSVM. It's strange that models with LSA seems to have poor performance. In fact, LSA with large number of components will probably give rise to overfitting while LSA with few components may induce low accuracy. One of the reason for that is Latent Semantic Analysis works only by clustering words and is always used when there is no precise tag for each document. While in our case each document is bound with one rating and two terms even with similar TF-IDF may have totally different rating categories, like ”doesn't” and ”don't”. Ultimately, we selected the 4000 most popular terms and used their TF-IDF to train a Linear SVM model. The model makes sense where we find the most indicative terms like ”good”, ”bad”, ”works well”, ”never” etc. all have high TF-IDF. In the future, there are two directions we may make attempts, one is by changing the model to a neural network due to its perfect performance in detail refinement. Another is by making use of the phone's properties and treating them as our features for another classification model and combine this

Number of data for tf-idf	Class Weight	Train accuracy	Test accuracy
2000	balanced	0.6672	0.6331
1000	not balanced	0.6642	0.6388
2000	balanced	0.73642	0.6719
2000	not balanced	0.74016	0.6903
4000	balanced	0.7909	0.6781
4000	not balanced	0.797	0.6949

TABLE III: TF-IDF + Linear SVM

Number of data for tf-idf	Number of estimators	min samples leaf	Train accuracy	Test accuracy
2000	10	10	0.61764	0.5660
2000	20	10	0.63454	0.5774
2000	30	10	0.6502	0.5933
2000	10	20	0.59286	0.5615
2000	20	20	0.6072	0.5745
2000	30	20	0.6160	0.5810
2000	10	30	0.5747	0.5513
2000	20	30	0.59588	0.5677
2000	30	30	0.5922	0.5663
2000	50	2	0.84034	0.6737
2000	100	2	0.8480	0.6791
2000	300	2	0.8492	0.6844
2000	600	2	0.8513	0.6852
2000	1000	2	0.8505	0.6857

TABLE IV: TF-IDF + Random Forest

model with our original text mining model to make predictions.

This model shows a much better performance than the baseline model of linear regression (which gives the accuracy of 0.542 on the same test set). The text mining model shows an improvement of approximately 15.2%. The text mining model is also combined with different models. The results of the testing on these mixture models are shown in Table 3, 4 and 5.

The future work may include trying different mixture of the basic models discussed above, modifying and achieving more complex tf-idf model with better performance, and applying new algorithms and models of text mining to the current model.

Number of neighbors	Number of components	Train accuracy	Test accuracy
10000	500	0.54796	0.5416
10000	1000	0.54796	0.5417
10000	1500	0.54797	0.5417
10000	2000	0.5478	0.5416
5000	1000	0.5601	0.5416
15000	1000	0.5468	0.5416

TABLE V: TF + LSA + KNN

REFERENCES

[1] Adomavicius, G. and Tuzhilin, A. (2005). *Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions*, IEEE transactions on knowledge and data engineering, 17(6), 734-749.

[2] Marović, M., Mihoković, M., Mikša, M., Pribil, S., and Tus, A. (2011). *Automatic movie ratings prediction using machine learning*, Proceedings of the 34th International Convention, 17(6), 1640-1645.

[3] Tushar J. (2016). *Sentiment Classification in 5 classes*, Kaggle.

[4] Preetish P. (2017). *Text Mining Amazon Mobile Phone Reviews: Interesting Insights*, High Dimensional Space.

[5] Hrishikesh K. (2017). *Analysis of Amazon's unlocked mobile phone*, Kaggle.

[6] Rohit M. (2016). *Comprehensive analysis of the phone world*, Kaggle.

[7] Eliot B. (2017). *Mining with Sklearn /Keras (MLP, LSTM, CNN)*, Kaggle.

[8] Kupiec, J. (1992). *Robust part-of-speech tagging using a hidden Markov model*, Computer Speech & Language, 6(3), 225-242.

[9] Adams, S., Beling, P. and Cogill, R. (2016). *Feature Selection for Hidden Markov Models and Hidden Semi-Markov Models*, IEEE Access, 4, 1642-1657.

APPENDIX A
PROPERTIES ENCODING

1) Color encoding

NA	0
Black	1
Blue	2
Grey	3
Green	4
Pink	5
Silver	6
White	7
Yellow	8

2) Storage encoding

NA	0
8GB	1
16GB	2
32GB	3
128GB	4
256GB	5

3) Carrier encoding

NA	0
ATT	1
InternationalVesion	2
Sprint	3
T-mobile	4
Ohters	5

4) Warranty encoding

No	0
Yes	1
NA	2

5) Unlocked encoding

No	0
Yes	1

APPENDIX B
THE CODE FOR DATA PREPOSSESSING

```
from collections import defaultdict
dict_brand={}
def feature(datum):
   feat = []
   feat.append(datum[1])
   return feat

Brand= [feature(dataline) for
    dataline in data]
for brand in Brand:
   brand[0]=brand[0].lower()


sorted_list=sorted(dict_brand.items(),
    key=lambda d: d[1],reverse=True)
sum=0
brand_list=[]
for i in range(10):
   sum+=sorted_list[i][1]
   brand_list.append(sorted_list[i][0])
```

```
#color feature function
def color_feature(list1,dict1):
   for i in range(len(list1)):
      if 'green' in list1[i].lower():
         dict1[i].append('green')
      elif 'silver' in
           list1[i].lower():
         dict1[i].append('silver')
      elif 'black' in
           list1[i].lower():
         dict1[i].append('black')
      elif 'white' in
           list1[i].lower():
         dict1[i].append('white')
      elif 'yellow' in
           list1[i].lower():
         dict1[i].append('yellow')
      elif 'gray' in
           list1[i].lower():
         dict1[i].append('gray')
      elif 'pink' in
           list1[i].lower():
         dict1[i].append('pink')
      elif 'blue' in
           list1[i].lower():
         dict1[i].append('blue')
      else:
         dict1[i].append(0)
```

```
# carrier_feature
def carrier_feature(list1,dict1):
   for i in range(len(list1)):
      if 'sprint' in
           list1[i].lower():
         dict1[i].append('sprint')
      elif 'at&t' in
           list1[i].lower():
         dict1[i].append('at&t')
      elif 't-mobile' in
           list1[i].lower():
         dict1[i].append('t-mobile')
      elif 'verizon' in
           list1[i].lower():
         dict1[i].append('verizon')
      elif 'internationalversion' in
```

```
            list1[i].lower():
            dict1[i].append('internation
            alversion')
        else:
            dict1[i].append(0)
```

```
# warranty_feature
def warranty_feature(list1,dict1):
    for i in range(len(list1)):
        if 'nowarranty' in
            list1[i].lower():
            dict1[i].append(1)
            continue
        elif 'warranty' in
            list1[i].lower():
            dict1[i].append(2)
            continue
        else:
            dict1[i].append(0)
```

```
# unlock_feature
def unlock_feature(list1,dict1):
    for i in range(len(list1)):
        if 'unlock' in
            list1[i].lower():
            dict1[i].append(1)
        else:
            dict1[i].append(0)
```

```
# Unigram count
# No stemming, with stopwords
unigramCount = defaultdict(int)
punctuation = set(string.punctuation)
for d in data:
    r = ''.join([c for c in
        d['Reviews'].lower() if not
        c in punctuation])
    for w in r.split():
        if w not in stopWords:
            unigramCount[w] += 1
        else:
            continue

uni_counts = [(unigramCount[w], w)
    for w in unigramCount]
uni_counts.sort()
uni_counts.reverse()
```

```
# Bigram count
bigramsCount = defaultdict(int)
punctuation = set(string.punctuation)

for d in data:
    r = ''.join([c for c in
        d['Reviews'].lower() if not
        c in punctuation])
    wordList = [w for w in r.split()
        if not w in stopWords]
    for i in range(len(wordList) - 1):
        bigramsCount[wordList[i] + ' '
            + wordList[i+1]] += 1


bi_counts = [(bigramsCount[w], w)
    for w in bigramsCount]
bi_counts.sort()
bi_counts.reverse()
```

```
# Term frequency
def tf(term):
    t = term.split()
    if len(t) == 1:
        return unigramCount[term]
    elif len(t) == 2:
        return bigramsCount[term]
    else:
        return 0
```

```
# idf for unigram
def idf(term):
    frequency = 0
    for d in data:
        r = ''.join([c for c in
            d['Reviews'].lower() if
            not c in punctuation])
        if term in r.split():
            frequency += 1
    return -numpy.log10(frequency *
        1.0 / len(data))
```

```
# idf for bigram
def idf2(term):
    frequency = 0
    word1, word2 = term.split()
    for d in data:
        r = ''.join([c for c in
            d['Reviews'].lower() if
            not c in punctuation])
        wordList = [w for w in
            r.split() if not w in
            stopWords]
        for i in
            range(len(wordList)-1):
            if wordList[i] == word1 and
                wordList[i+1] == word2:
                frequency += 1
    return -numpy.log10(frequency *
        1.0 / len(data))
```

APPENDIX C
THE CODE FOR BASELINE

```
import csv
from collections import defaultdict
import random
import numpy
import sklearn
from sklearn import linear_model
import scipy.optimize
import pandas
import math

data = []
with
    open("new_shuffle_data.csv",'rb')
    as my_file:
   reader = csv.reader(my_file)
   data = list(reader)
priceList = [d[2] for d in data]
len(priceList)

features = []
with
    open("property_feature_encoded2.csv",'rb')
    as my_file:
   reader = csv.reader(my_file)
   features = list(reader)
```

```
def feature(datum):
   feat_color = [0] * len(colorDict)
   feat_storage = [0] *
       len(storageDict)
   feat_carrier = [0] *
       len(carrierDict)
   feat_warranty = [0] *
       len(warrantyDict)
   feat_unlock = [0] *
       len(unlockDict)
   feat_phonetype = [0] *
       len(phonetypeDict)

   feat_color[int(datum[0])] = 1
   feat_storage[int(datum[1])] = 1
   feat_carrier[int(datum[2])] = 1
   feat_warranty[int(datum[3])] = 1
   feat_unlock[int(datum[4])] = 1
   feat_phonetype[int(datum[5])] = 1
   feat_price = [float(datum[6])]

   feat = [1] + feat_color +
       feat_storage + feat_carrier
       + feat_warranty + \
   feat_unlock + feat_phonetype +
       feat_price

   return feat
```

```
def regression(X, y, X_test):
   clf = linear_model.Ridge(1.0,
       fit_intercept=False)
   clf.fit(X, y)
   theta = clf.coef_
   predictions = clf.predict(X_test)
   print theta
   return predictions
```

```
def getAcc(predictions, y):
   acc = sum([y[i] == predictions[i]
       for i in xrange(len(y))])
   acc /= 1.0 * len(y)
   print acc
```

```
def approx(predictions):
   predictions = [int(round(i)) for
       i in predictions]

   for i in xrange(len(predictions)):
      if predictions[i] < 1:
         predictions[i] = 1
      elif predictions[i] > 5:
         predictions[i] = 5

   return predictions
```

```
data_train = data[1:50001]
data_test = data[50001:]
features_train = features[1:50001]
features_test = features[50001:]

X_train = [feature(d) for d in
    features_train]
y_train = [int(d[3]) for d in
    data_train]
X_test = [feature(d) for d in
    features_test]
y_test = [int(d[3]) for d in
    data_test]

predictions =
    approx(regression(X_train,
    y_train, X_test))
getAcc(predictions, y_test)
```

APPENDIX D
THE CODE FOR TEXT MINING

```
#feature (tf_idf or tf)
import string
punctuation = set(string.punctuation)
words = unigram.keys()[:1000]
word_id = dict(zip(words,
    range(len(words))))

def tf_feature(datum):
   feat = [0]*len(words)
```

```
    r = ''.join([c for c in
        datum['reviews'].
    lower() if not c in punctuation])
    for w in r.split():
       if w in words:
          feat[word_id[w]] += 1
    return feat

def tfidf_feature(datum):
    feat = tf_feature(datum)
    for w in words:
       feat[word_id[w]] *= idf[w]
    return feat

X_train = [tfidf_feature(d) for d in
    data_train]
y_train = [d['rating'] for d in
    data_train]
X_test = [tfidf_feature(d) for d in
    data_test]
y_test = [d['rating'] for d in
    data_test]
```

```
#model: tfidf + single svm("rbf")
from sklearn import svm
import time

#Train
print "Train model: tf_idf +
    svm(rbf)"
cls1 =
    svm.SVC(tol=1e-3,max_iter=1000)
t0 = time.time()
cls1.fit(X_train, y_train)
t1 = time.time()
print "training time: " + str(t1-t0)

vector1 = cls1.support_vectors_
preds = cls1.predict(X_train)

t2 = time.time()
print "prediction time: " +
    str(t2-t1)
acc = accuracy(preds, y_train)
print "Training accuracy = " +
    str(acc)

#Test
print "Test model: tf_idf + svm(rbf)"
preds = cls1.predict(X_test)
acc = accuracy(preds, y_test)
print "Test accuracy = " + str(acc)
```

```
#model: tf_idf + random_forest
from sklearn.ensemble import
    RandomForestClassifier
f = open("training
_test_accuracy.txt","a")

for leaf in [2]:
   for n_estimators in [100, 300,
       600]:
      print("Train model: tf_idf +
      random_forest, n_estimators =
          {0},
      min_samples_leaf = {1}\n".
      format(n_estimators, leaf))
      f.write("Train model: tf_idf +
      random_forest, n_estimators =
          {0},
      min_samples_leaf =
          {1}\n".format(n_estimators,
          leaf))
      #Train
      t0 = time.time()
      cls2 = RandomForest
      Classifier(n_estimators=n_estimators,

      min_samples_leaf=leaf,
      class_weight = "balanced",
      n_jobs=15)
      cls2.fit(X_train, y_train)
      t1 = time.time()
      print("Prediction time: " +
          str(t1-t0))
      acc =
          cls2.score(X_train,y_train)
      print("Training accuracy = " +
          str(acc))
      f.write("Training accuracy =
          {}".format(acc))

      #Test
      acc = cls2.score(X_test,
          y_test)
      print("Test accuracy = " +
          str(acc))
      f.write("Test accuracy =
          {}".format(acc))
f.close()
```

```
#model: tfidf + Bagging svm("rbf")
from sklearn.ensemble import
    BaggingClassifier
from sklearn.multiclass import
    OneVsRestClassifier
f = open("training_test
_accuracy.txt","a")

for kernel in ['sigmoid','rbf']:
   print("Train model: tf_idf +
   Bagging SVC('{}')".format(kernel))
   f.write("Train model: tf_idf +
   Bagging SVC('{}')".format(kernel))
   #Train
   n_estimators = 10
   t0 = time.time()
   cls3 = OneVsRestClassifier(
   BaggingClassifier(svm.SVC(kernel=kernel),
```

```
            max_samples=1.0 /
                n_estimators,
            n_estimators=n_estimators,
          n_jobs=15))
    cls3.fit(X_train, y_train)
    t1 = time.time()
    print("Prediction time: " +
        str(t1-t0))
    acc = cls3.score(X_train,y_train)
    print("Training accuracy = " +
        str(acc))
    f.write("Training accuracy =
        {}".format(acc))

    #Test
    acc = cls3.score(X_test, y_test)
    print("Test accuracy = " +
        str(acc))
    f.write("Test accuracy =
        {}".format(acc))
f.close()
```

```
#model: tfidf + Linearsvm()
from sklearn import svm
import time

f = open("training_test_
accuracy.txt","a")

#Train balanced class weight
print "Train model: tf_idf +
    Linearsvm(),
class_weight = 'balanced'"
f.write("Train model: tf_idf +
    Linearsvm(),
class_weight = 'balanced'\n")
cls4 = svm.LinearSVC(max_iter=1000,
    tol=1e-5,
class_weight = 'balanced')
t0 = time.time()
cls4.fit(X_train, y_train)
t1 = time.time()
print "training time: " + str(t1-t0)

preds = cls4.predict(X_train)
t2 = time.time()
print "prediction time: " +
    str(t2-t1)
acc = accuracy(preds, y_train)
print "Training accuracy = " +
    str(acc)
f.write("Training accuracy =
    {}\n".format(acc))
#Test
print "Test model: tf_idf +
    Linearsvm()"
preds = cls4.predict(X_test)
acc = accuracy(preds, y_test)
print "Test accuracy = " + str(acc)
f.write("Test accuracy =
    {}\n".format(acc))
f.write("\n")

#Train without balance weight
print "Train model: tf_idf +
    Linearsvm(),
class_weight = 'not balanced'"
f.write("Train model: tf_idf +
    Linearsvm(),
class_weight = 'not balanced'\n")
cls4 = svm.LinearSVC(max_iter=1000,
    tol=1e-5)
t0 = time.time()
cls4.fit(X_train, y_train)
t1 = time.time()
print "training time: " + str(t1-t0)

preds = cls4.predict(X_train)
t2 = time.time()
print "prediction time: " +
    str(t2-t1)
acc = accuracy(preds, y_train)
print "Training accuracy = " +
    str(acc)
f.write("Training accuracy =
    {}\n".format(acc))
#Test
print "Test model: tf_idf +
    Linearsvm()"
preds = cls4.predict(X_test)
acc = accuracy(preds, y_test)
print "Test accuracy = " + str(acc)
f.write("Test accuracy =
    {}\n".format(acc))
f.write("\n")
f.close()
```

```
#model: tf + LSA + KNN
from sklearn.feature_extraction.text
    import CountVectorizer
from sklearn.decomposition import
    TruncatedSVD
from sklearn.pipeline import
    make_pipeline
from sklearn.preprocessing import
    Normalizer
from sklearn.neighbors import
    KNeighborsClassifier
import time

#tf formalization
n_features = 10000
review_train = [d["reviews"] for d
    in data_train]
review_test = [d["reviews"] for d in
    data_test]
tf_vectorizer =
    CountVectorizer(max_df=0.95,
    min_df=0.,
      max_features=n_features,
```

```
      strip_accents='ascii',
      de_error='ignore',
      stop_words='english')
t0= time.time()
X_train = tf_vectorizer.
fit_transform(review_train)
X_test = tf_vectorizer.
fit_transform(review_test)
y_train = [d['rating'] for d in
    data_train]
y_test = [d['rating'] for d in
    data_test]
t1 = time.time()
print "tf transformation time: " +
    str(t1-t0)

#Training
f = open("training_
test_accuracy.txt","a")
for n_components in [500, 1000,
    1500, 2000]:
   svd = TruncatedSVD
   (n_components=n_components,
   n_iter=10, random_state=42)
   lsa = make_pipeline(svd,
       Normalizer(copy=False))
   t0 = time.time()
   lsa_train =
       lsa.fit_transform(X_train)
   lsa_test =
       lsa.fit_transform(X_test)

   print "Train model: tf + LSA +
       KNN,
   n_features=10000,
       n_components={}".
   format(n_components)
   f.write("Train model: tf + LSA +
       KNN,
   n_features=10000,
       n_components={}\n".
   format(n_components))
   knn_lsa = KNeighborsClassifier(
   n_neighbors=10000,
   algorithm='auto', metric='cosine')
   knn_lsa.fit(lsa_train, y_train)
   t1 = time.time()
   print "Training time = " +
       str(t1-t0)

   t2 = time.time()
   preds = knn_lsa.predict(lsa_train)
   acc = accuracy(preds, y_train)
   print "Prediction time = " +
       str(t2-t1)
   print "Training accuracy = " +
       str(acc)
   f.write("Training accuracy =
       {}\n".format(acc))

   #Test
   print "Test model: tfidf + LSA +
       KNN"
   preds = knn_lsa.predict(lsa_test)
   acc = accuracy(preds, y_test)
   print "Test accuracy = " +
       str(acc)
   f.write("Test accuracy =
       {}\n".format(acc))
   f.write("\n")
f.close()
```

```
#model: tfidf + LSA + svm('rbf')
from sklearn.feature_extraction.text
    import TfidfVectorizer
from sklearn.decomposition import
    TruncatedSVD
from sklearn.pipeline import
    make_pipeline
from sklearn.preprocessing import
    Normalizer
from sklearn import svm
import time

#tfidf formalization
n_features = 10000
review_train = [d["reviews"] for d
    in data_train]
review_test = [d["reviews"] for d in
    data_test]
tfidf_vectorizer =
    TfidfVectorizer(max_df=0.95,
    min_df=0.,

      max_features=n_features,
      strip_accents='unicode',
      decode_error='ignore',
      stop_words='english')
t0= time.time()
X_train = tfidf_vectorizer.
fit_transform(review_train)
X_test = tfidf_vectorizer.
fit_transform(review_test)
y_train = [d['rating'] for d in
    data_train]
y_test = [d['rating'] for d in
    data_test]
t1 = time.time()

#Training
f = open("training_test
_accuracy.txt","a")
for n_components in [500, 1000]:
   svd = TruncatedSVD
   (n_components=n_components,
   n_iter=10, random_state=42)
   lsa = make_pipeline(svd,
       Normalizer(copy=False))
   t0 = time.time()
   lsa_train =
       lsa.fit_transform(X_train)
```

```
    lsa_test =
        lsa.fit_transform(X_test)

    print "Train model: tf + LSA +
        svm(),
    n_features=10000,
        n_components={}".
    format(n_components)
    f.write("Train model: tf + LSA +
        svm(),

    n_features=10000,
        n_components={}\n".
    format(n_components))
    svm_lsa =
        svm.SVC(tol=1e-3,max_iter=1000)
    svm_lsa.fit(lsa_train, y_train)
    t1 = time.time()
    print "Training time = " +
        str(t1-t0)

    t2 = time.time()
    preds = svm_lsa.predict(lsa_train)
    acc = accuracy(preds, y_train)
    print "Prediction time = " +
        str(t2-t1)
    print "Training accuracy = " +
        str(acc)
    f.write("Training accuracy =
        {}\n".format(acc))

    #Test
    print "Test model: tfidf + LSA +
        KNN"
    preds = svm_lsa.predict(lsa_test)
    acc = accuracy(preds, y_test)
    print "Test accuracy = " +
        str(acc)
    f.write("Test accuracy =
        {}\n".format(acc))
    f.write("\n")
f.close()
```