Language Modeling and Opinion Spam Classification

Larissa Pereira (lp445), Kaveesha Shah (ks2379)

Kaggle group name for Naïve Bayes - Highlanders

Kaggle group for Language Model – Kaveesha Shah (couldn't create a team)

Language Model:

Pre-processing**:**

Need:

- Eliminating redundancy and filter irrelevant details that don't add value.
- Save some processing time and database space by ignoring useless data

Steps and Implementation details as follows –

- Convert all characters to lowercase : lower() method of string data type
- Eliminated punctuation marks, digits and special characters:
- Eliminate stop words – Implemented using the stopwords corpus of nltk library

Training the model

We begin by training the model using the given corpus which will be used to predict the labels of the real data. The training methodology primarily consists of building a bag of the words observed in the corpus and computing their corresponding frequencies for the respective N-gram models.

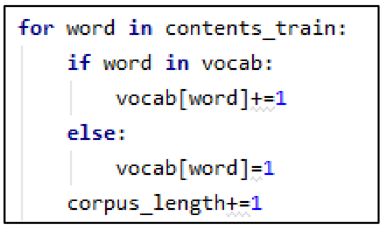
Unsmoothed Unigrams –

P(wi) = count (wi) / count (total number of words in vocabulary)

$$P(w_1 w_2 w_3 \ldots \ldots w_n) = \prod_{i=1}^{n} P(w_i)$$

Training for unigram model-

- We created a separate vocabulary for each of the truthful and deceptive corpuses by parsing the training data and obtained the count of each word in the vocabulary.
- This was created using the dictionary data structure where the key is the unique word and the value is the frequency of the key word in the training corpus.
  Code Snippet-



```
for word in contents_train:
    if word in vocab:
        vocab[word]+=1
    else:
        vocab[word]=1
    corpus_length+=1
```

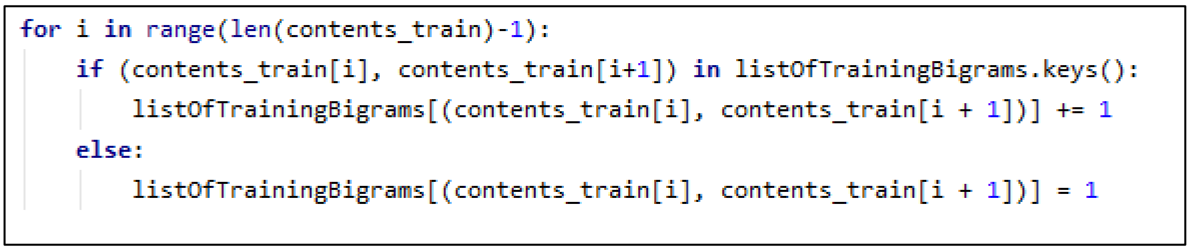
Unsmoothed Bigrams- The unigram model might not generate accurate results hence we introduce the bigram model where we compute the probability of the co-occurrence of words by calculating the probabilities of a word, given the previous word.

$$P(w_i \mid w_{i-1}) = \text{count} (w_{i-1}, w_i) / \text{count} (w_{i-1})$$

$$P(w_1 w_2 w_3 \ldots \ldots w_n) = \prod_{i=1}^{n} P(w_i \mid w_{i-1})$$

Training for bigram model-

- We iterate through the training data two words at a time thus creating a separate dictionary of bigrams for each of the truthful and deceptive corpuses,
- The key is the tuple formed using these two words and the value is the frequency of these words occurring together in that specific order.
  Code Snippet-



```
for i in range(len(contents_train)-1):
    if (contents_train[i], contents_train[i+1]) in listOfTrainingBigrams.keys():
        listOfTrainingBigrams[(contents_train[i], contents_train[i + 1])] += 1
    else:
        listOfTrainingBigrams[(contents_train[i], contents_train[i + 1])] = 1
```

Now that we have trained our respective N-gram models based on the two corpuses provided, we will apply these learnings to calculate the probabilities of a sequence of words in a sentence and label them as truthful or deceptive.

Implementation of unsmoothed unigram:

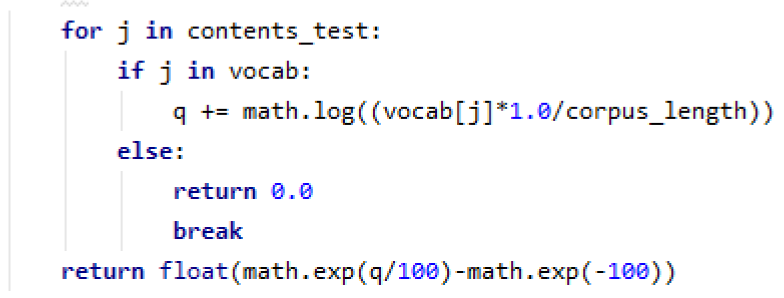
We now assign a probability to a sequence of words in a sentence based on the words we have already observed from training.

We iterate through each word from the test dataset and calculate its probability using the formula

$$P(wi) = \text{count } (wi) \;/\; \text{count (total number of words in vocabulary)}$$

We take the log of that term and add it to the cumulative probability variable instead of multiplying the individual word probabilities together. Adding the log values of the individual probabilities is computationally faster as well as helps us avoid underflow. This value is then returned from the unsmoothed_ngram() method.

Code Snippet-



```
for j in contents_test:
    if j in vocab:
        q += math.log((vocab[j]*1.0/corpus_length))
    else:
        return 0.0
        break
return float(math.exp(q/100)-math.exp(-100))
```

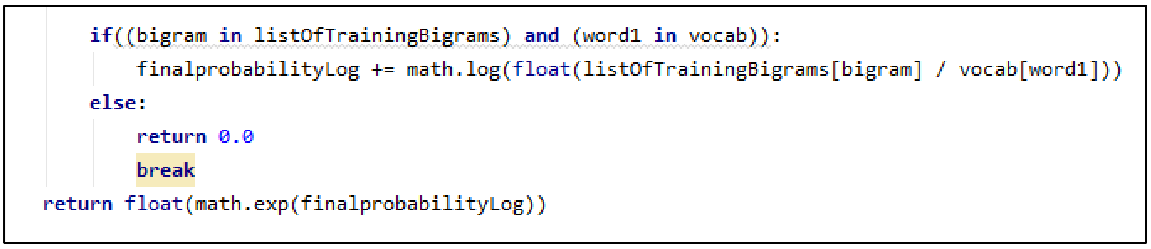
Implementation details for bigram

We find the bigrams for the test data and use them as keys to find the corresponding counts from the bigram dictionary created while training the model.
For each of these bigrams we calculate the probability using the formula

$$P(w_i \mid w_{i-1}) = \text{count} (w_{i-1}, w_i) / \text{count} (w_{i-1})$$

We take the log of that term and add it to the cumulative probability variable. Adding the log values of the individual probabilities is computationally faster as well as helps us avoid underflow. This value is then returned from the unsmoothed_ngram() method.

Code Snippet-



```
        if((bigram in listOfTrainingBigrams) and (word1 in vocab)):
            finalprobabilityLog += math.log(float(listOfTrainingBigrams[bigram] / vocab[word1]))
        else:
            return 0.0
            break
    return float(math.exp(finalprobabilityLog))
```

Observations – We get the probability value as 0 if there is any word in the test set that was previously unseen for both the above unsmooth language models.

Smoothing:

The N-gram Language Modelling approach has the following limitations –

- The model behaves better for a higher value of N but that would increase the overhead of computation
- N-grams are based on the co-occurrence of words. It will assign a zero probability to all the words that occur in the test set but are not present in the training corpus. This sparse representation of language is a major setback for traditional N-gram models.

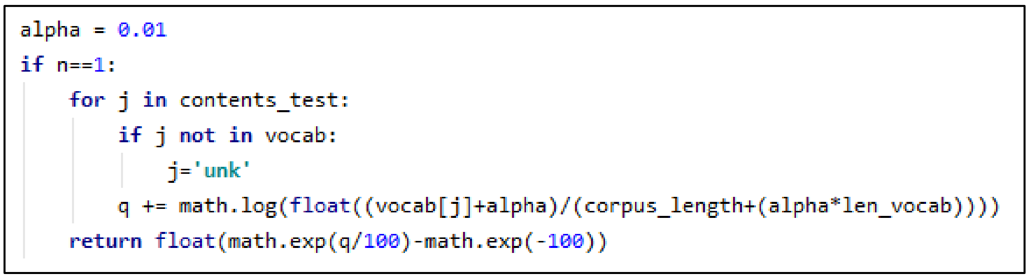
To overcome these above mentioned drawbacks we employed Add-k smoothing to the N-gram models.

- In this technique we add a fractional value k to all the counts.
- Adding this value k to the count leads to an extra V*k observations where V is the number of word types.
- By doing so, we are trying to handle data scarcity issue by assigning a probability to the words that we have never seen before instead of taking its probability as 0.

Formula used for Unigram smoothing

$$P(w_n) = \frac{count(w_n)+k}{N+(k*V)}$$

Code Snippet -

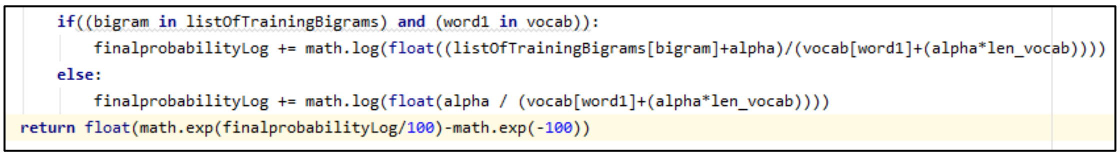


```
alpha = 0.01
if n==1:
    for j in contents_test:
        if j not in vocab:
            j='unk'
        q += math.log(float((vocab[j]+alpha)/(corpus_length+(alpha*len_vocab))))
    return float(math.exp(q/100)-math.exp(-100))
```

Formula used for Bigram smoothing

$$P(w_n \mid w_{n-1}) = \frac{count(w_{n-1}w_n) + k}{count(w_{n-1}) + (k * V)}$$

where V = No. of word types and N = Length of the corpus in work tokens



```
    if((bigram in listOfTrainingBigrams) and (word1 in vocab)):
        finalprobabilityLog += math.log(float((listOfTrainingBigrams[bigram]+alpha)/(vocab[word1]+(alpha*len_vocab))))
    else:
        finalprobabilityLog += math.log(float(alpha / (vocab[word1]+(alpha*len_vocab))))
return float(math.exp(finalprobabilityLog/100)-math.exp(-100))
```

Observations and Evaluation-

We tested for the accuracy of the validation set by experimenting with various values of k. We observed that the accuracy was highest (93%) in case **of Smooth Unigrams for k=0.01.** Thus using this value of k we computed the smooth N-gram probabilities and applied the model to the test dataset to predict the output labels.

Experiments with different values of k :

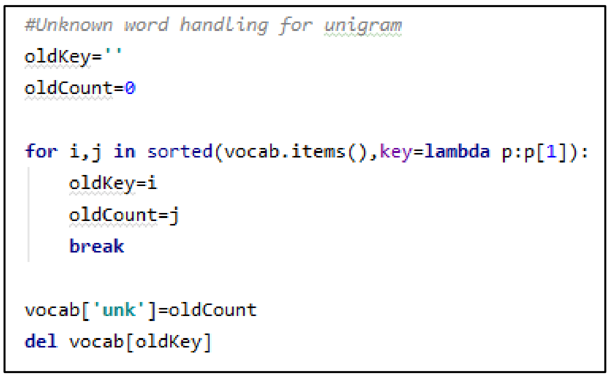
	Smooth Unigrams			Smooth Bigrams		
	Truthful Accuracy	Deceptive Accuracy	Average	Truthful Accuracy	Deceptive Accuracy	Average
For k=0.01	0.914063	0.953125	0.933594	0.851563	0.890625	0.871094
For k=0.05	0.90625	0.953125	0.929688	0.796875	0.945313	0.871094
For k=0.005	0.914063	0.953125	0.933594	0.851563	0.859375	0.855469
For k = 0.02	0.914063	0.953125	0.933594	0.835938	0.914063	0.875

Handling unknowns :

To handle unknown words we implemented the following-

- Find the least frequently occurring word in the training corpus
- Replace the word with tag ‘unk’ and retain the frequency value ensuring that the total probability remains unaffected (viz. 1)
- When we find a new word while testing we tag it as ‘unk’

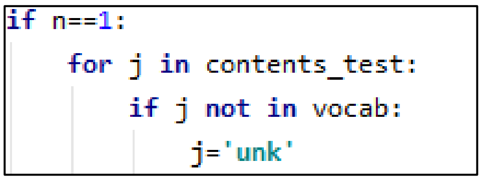
Code Snippet–



```
#Unknown word handling for unigram
oldKey=''
oldCount=0

for i,j in sorted(vocab.items(),key=lambda p:p[1]):
    oldKey=i
    oldCount=j
    break

vocab['unk']=oldCount
del vocab[oldKey]
```



```
if n==1:
    for j in contents_test:
        if j not in vocab:
            j='unk'
```

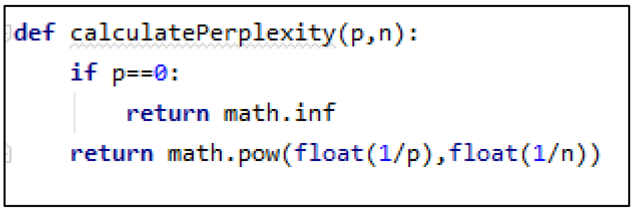
Perplexity:

In order to evaluate the performance of a specific N-gram model, we calculated the perplexities for the different models. A higher value of the estimated probability of a word sequence, lower will be its perplexity. A lower perplexity indicates a better model.

$$PP(W) = P(w_1 w_2 \ldots w_N)^{-1/N}$$

$$= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}}$$

Code Snippet-



```
def calculatePerplexity(p,n):
    if p==0:
        return math.inf
    return math.pow(float(1/p),float(1/n))
```

Observations on one Truthful Review-

We calculated the perplexity for the first review from each Truthful and Deceptive validation set and obtained predictions as below-

	Perplexity_Train_Truthful	Perplexity_Train_Deceptive	Prediction based on Perplexity	Actual Classification
Smooth Unigram	687.43	620.95	Deceptive	Deceptive
Smooth Unigram	484.229	546.915	Truthful	Truthful
Smooth Bigram	16.09	12.662	Deceptive	Deceptive
Smooth Bigram	14.14	17.67	Truthful	Truthful

Issues faced while implementing Language Models and their subsequent handling–

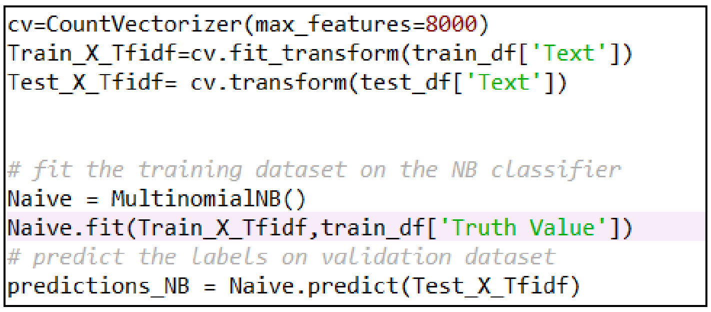
a. Underflow – For the given dataset the probabilities calculated decreased to significantly small values which when multiplied together to obtain the final probability further decreased the value causing an underflow issue. To fix this we calculated the probabilities by taking sum of the logarithms of the probabilities of the individual words and then computing the final result by taking its antilog.
b. Underflow despite using Logarithms – For the given dataset we observed that taking log of probabilities resulted in values as small as -4977.55 and the antilog of this value would be very small resulting in underflow again. To fix this we divide the sum of logs by 100 and then subtract e^{-100} from it.

Naïve Bayes:

It is a classifier that applies Bayes theorem and assumes that every feature is independent of the other features. It is a probabilistic classifier which calculates probability of each category and assigns the label based on highest probability.

Incorporating Naïve Bayes Model-

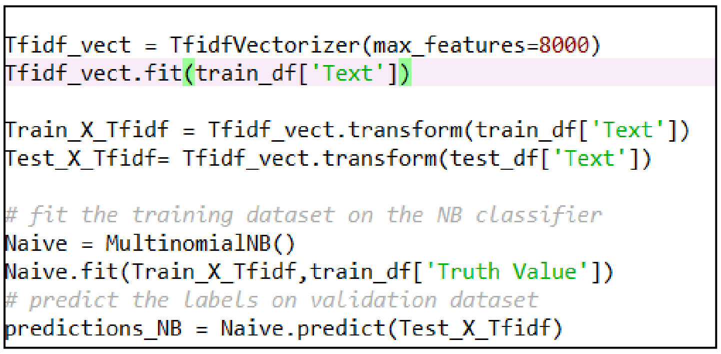
- We have used the sklean library's module Naïve Bayes and MultinomialNB classifier in it to classify with discrete features.
- In case of bag of words approach, we pass the count matrix which has the word and its corresponding count as the training data to the Naive Bayes Classifier



```
cv=CountVectorizer(max_features=8000)
Train_X_Tfidf=cv.fit_transform(train_df['Text'])
Test_X_Tfidf= cv.transform(test_df['Text'])


# fit the training dataset on the NB classifier
Naive = MultinomialNB()
Naive.fit(Train_X_Tfidf,train_df['Truth Value'])
# predict the labels on validation dataset
predictions_NB = Naive.predict(Test_X_Tfidf)
```

- In case of added language features, we pass the tf-idf matrix which has the word and its (tf*idf) term as the training data to the Naive Bayes Classifier



```
Tfidf_vect = TfidfVectorizer(max_features=8000)
Tfidf_vect.fit(train_df['Text'])

Train_X_Tfidf = Tfidf_vect.transform(train_df['Text'])
Test_X_Tfidf= Tfidf_vect.transform(test_df['Text'])

# fit the training dataset on the NB classifier
Naive = MultinomialNB()
Naive.fit(Train_X_Tfidf,train_df['Truth Value'])
# predict the labels on validation dataset
predictions_NB = Naive.predict(Test_X_Tfidf)
```

- We used the fit() method with training features and the corresponding output as its parameters to train the model.
- We predict the labels of the test data using the predict() method.

Incorporating Bag of Words in N-gram features-

- We used the CountVectorizer module from the feature extraction library of sklearn.
- In this approach, we use the word count for determining its relevance to that particular classification label.
- The position of the word in a document doesn't matter and the feature probabilities $P(x_i \mid c_j)$ is independent of class c_j.

Implementation of additional linguistic features

- We have implemented TF-IDF approach as an additional linguistic feature.
- This uses 2 added features-term frequency and Inverse document frequency which are defined by the following formulae:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}} \qquad idf(w) = log(\frac{N}{df_t})$$

- These are used because we get an idea as to how important a term is relative to a review and to a **corpus**,
- When we multiply TF and IDF, we observe that the larger the number, the more important a term in a review is to that review.
- We can then compute the TF-IDF for each word in each document and create a vector. This matrix is then passed to the Naive Bayes classifier as training data and the respective labels (0 and 1 in our implementation which correspond to truthful and deceptive resp.) as the output.

The motivation for choosing these features is the need to understand the relevance of a term with respect to that class. The reason for choosing IDF comes into play in cases for words like "the." We know that just about every review contains "the," so the term isn't really special anymore, thereby producing a very low IDF. We can contrast "the" with "request" in our example. "request" appears rarely in the other posts, so its IDF should be high. In fact, "request" now carries a weight signalling that in any review in which it appears, it is important to that review.

Observations-

	Count Vectorizer Accuracy	TFIDF Accuracy
Truthful	89.84	79.6875
Deceptive	91.4	94.53
Average	90.62	87.10875

Error analysis and comparison of language model based classifier with Naive Bayes:

Confusion matrix for predictions on Deceptive Reviews-

	LM Correct	LM Wrong
NB Correct	118	2
NB Wrong	3	5

Observations –

- NB performed better than LM on 2 reviews
- LM performed better than NB in 3 reviews

- Both the models predicted the correct labels for 118 reviews and wrong labels for 5 reviews

Confusion matrix for predictions on Truthful Reviews-

	LM Correct	LM Wrong
NB Correct	99	3
NB Wrong	18	8

Observations –

- NB performed better than LM on 3 reviews
- LM performed better than NB in 18 reviews
- Both the models predicted the correct labels for 99 reviews and wrong labels for 8 reviews

Details of programming library usage:

- String Library: Punctuation
- NLTK Library: stopwords corpus
- Pandas: Dataframes
- Sklearn: Count vectorizer, TfIdf Vectorizer, Metrics accuracy score, naive_bayes

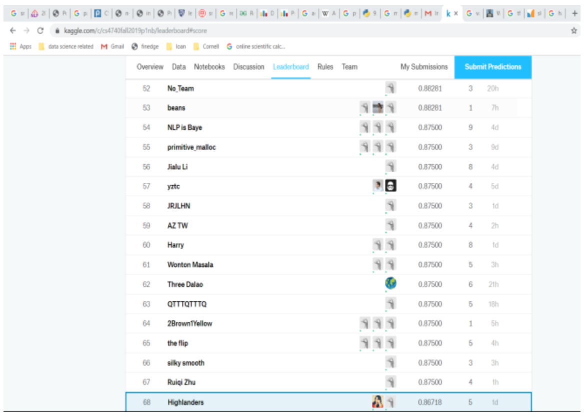
Description of work distribution:

- Preprocessing and Training the model - Larissa
- `Implementation of unsmooth N-grams – Larissa
- Implementation of smooth N-grams& unknown word handling – Larissa and Kaveesha
- Perplexity calculation - Kaveesha
- Implementation of Naïve Bayes model – Kaveesha
- Evaluating language models – Larissa and Kaveesha
- Report – Larissa and Kaveesha

Feedback

Through this project, we got a good understanding of the language models and the feature-based classification. It was a great experience to actually implement the methods that we had theoretical knowledge of. The implementation sounded easy but took time in reality and it was of a good difficulty level-not too hard and not too easy either. The implementation took 3 days and 1 day went behind the report.

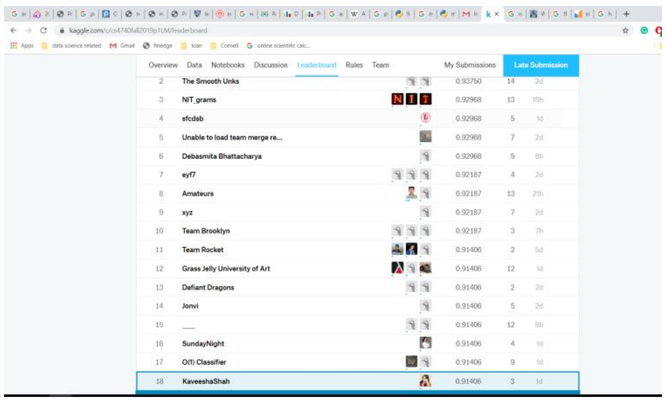
Kaggle screenshots



kaggle.com/c/cs4740fall2019p1nb/leaderboard#score

Overview Data Notebooks Discussion Leaderboard Rules Team My Submissions Submit Predictions

52	No_Team	0.88281	3	20h
53	beans	0.88281	1	7h
54	NLP is Baye	0.87500	9	4d
55	primitive_malloc	0.87500	3	9d
56	Jialu Li	0.87500	8	4d
57	yztc	0.87500	4	5d
58	JRJLHN	0.87500	3	1d
59	AZ TW	0.87500	4	2h
60	Harry	0.87500	8	1d
61	Wonton Masala	0.87500	5	3h
62	Three Dalao	0.87500	6	21h
63	QTTTQTTTQ	0.87500	5	18h
64	2Brown1Yellow	0.87500	1	5h
65	the flip	0.87500	5	4h
66	silky smooth	0.87500	3	3h
67	Ruiqi Zhu	0.87500	4	1h
68	Highlanders	0.86718	5	1d



kaggle.com/c/cs4740fall2019p1LM/leaderboard

Overview Data Notebooks Discussion Leaderboard Rules Team My Submissions Late Submission

2	The Smooth Unks	0.93750	14	2d
3	NIT_grams	0.92968	13	18h
4	sfcdsb	0.92968	5	1d
5	Unable to load team merge re...	0.92968	7	2d
6	Debasmita Bhattacharya	0.92968	5	8h
7	eyf7	0.92187	4	2d
8	Amateurs	0.92187	13	21h
9	xyz	0.92187	7	2d
10	Team Brooklyn	0.92187	3	7h
11	Team Rocket	0.91406	2	5d
12	Grass Jelly University of Art	0.91406	12	1d
13	Defiant Dragons	0.91406	2	2d
14	Jonvi	0.91406	5	2d
15	___	0.91406	12	8h
16	SundayNight	0.91406	4	1d
17	O(1) Classifier	0.91406	9	1d
18	KaveeshaShah	0.91406	3	1d