Weili He #3033084043 IEOR290 HW4

Part 1 Entropy

1.1

The initial Entropy for the information before we split it is:

$$\mathrm{H(D)} = -0.5log_2 0.5 - 0.5log_2 0.5 = 1$$

Then we decide which feature should we choose as the first split.

HasJob	1	1	1	1	1	0	0	0
Defaulter	0	0	0	1	1	0	1	1

Has Family	1	1	1	1	0	0	0	0
Defaulter	0	0	0	1	0	1	1	1

IsAbove30	1	1	1	1	1	1	0	0
Defaulter	0	0	0	1	1	1	1	0

Calculate the information we gain. If we choose **'HasJob'** as the first split:

$$\mathrm{P(Y=1|J=0)} = \frac{2}{3}; \quad \mathrm{P(Y=0|J=0)} = \frac{1}{3}$$

$$\mathrm{P(Y=1|J=1)} = \frac{2}{5}; \quad \mathrm{P(Y=0|J=1)} = \frac{3}{5}$$

$$\mathrm{P(J=0)} = \frac{3}{8}; \quad \mathrm{P(J=1)} = \frac{5}{8}$$

$$\mathrm{H(Y|J=0)} = -\frac{2}{3}log_2\frac{2}{3} - \frac{1}{3}log_2\frac{1}{3} = 0.918$$

$$\mathrm{H(Y|J=1)} = -\frac{2}{5}log_2\frac{2}{5} - \frac{3}{5}log_2\frac{3}{5} = 0.971$$

$$\mathrm{H(Y)} = \frac{3}{8} \times 0.918 + \frac{5}{8} \times 0.971 = 0.951$$

$$\mathrm{Gain} = 1 - 0.951 = 0.049$$

We can do the similar calculation for the other features.

For **'HasFamily'**:

$$\mathrm{Gain} = 1 - 0.811 = 0.189$$

For **‘IsAbove30Years’**:

$$\mathrm{Gain} = 1 - 1 = 0$$

Therefore, we choose the second feature **‘HasFamily’** as the first split since it has the largest information gain.

1.2

$$\mathrm{P(A)} = 0.7, \quad \mathrm{P(B)} = 0.2, \quad \mathrm{P(C)} = 0.1$$

$$\mathrm{H(A)} = -log_2 0.7 = 0.515\ bits, \quad \mathrm{H(B)} = -log_2 0.2 = 2.322\ bits, \quad \mathrm{H(C)} = -log_2 0.1 = 3.32\ bits$$

$$\mathrm{H(S)} = 0.7 \times H(A) + 0.2 \times H(B) + 0.1 \times H(C) = 1.156 bits$$

It means the smallest codeword length that is theoretically possible for signal ‘S’ is 1.156 bits according to the Source Coding Theorem

Part 2 Natural Language Processing

1.

Bag of words model is a simplifying representation used in natural language processing and information retrieval. Sentences and paragraphs are split into different single words and present it in a table, regardless of the grammar or the word order. It has Unigrams, Bigrams, Trigrams, N-grams models. It makes use of Markov Assumption and MLE of Probability. For example, the n-gram model is a statistical model of language in which the previous n-1 words are used to predict the next word. It also uses the Term-Frequency (tf) and Inverse Document Frequency (IDF) to extract the information.

Word2Vec model is a method of creating distributional representations of words called word embeddings, using backpropagation. It uses Neural Net Model to predict contextual words/word of the input words/word. It contains two algorithms: skip-gram and CBOW

Comparison:

The **bag-of-words model** is commonly used in methods of document classification where the occurrence of each word is used as a feature for training a classifier.

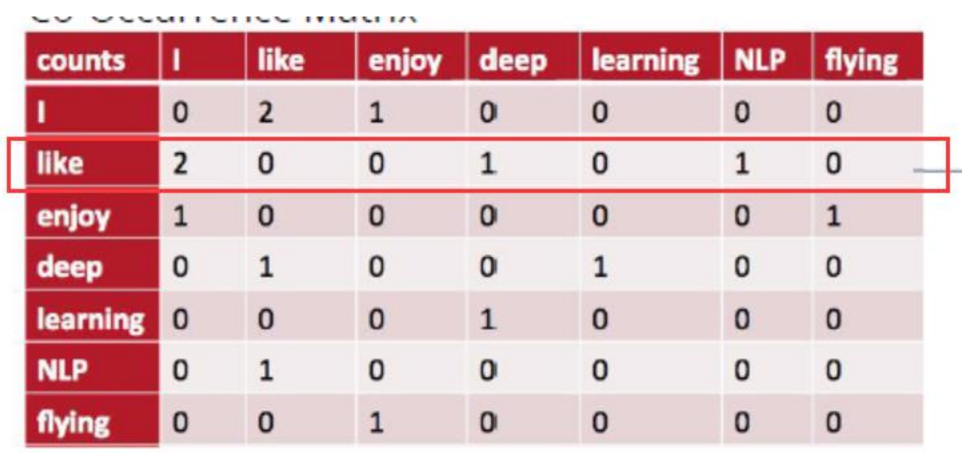
But it loses the ordering of words and ignore semantics of the words. However, the information about order and context of words is important to understand the document.

Word2Vec model is useful when we are working in the same domain or our own corpus is very small. It is an easy-to-scale and fast to train model compared with bag-of-words model. It does not need human tagged data and it is well on capturing semantic similarity.

But it is difficult to debug and does not handle ambiguities.

2. A word vector

A word vector is a vector form to express a word. It uses the count of other words adjacent to the word we want to express as a vector. A basic example is the figure we saw in class.



counts	I	like	enjoy	deep	learning	NLP	flying
I	0	2	1	0	0	0	0
like	2	0	0	1	0	1	0
enjoy	1	0	0	0	0	0	1
deep	0	1	0	0	1	0	0
learning	0	0	0	1	0	0	0
NLP	0	1	0	0	0	0	0
flying	0	0	1	0	0	0	0

The highlighted row is the vector expression of '**like**' in this corpus.

Word embedding is a set of language modeling and feature learning techniques in natural language processing where words or phases from the vocabulary are mapped to vectors of real number. It makes improvement by compressing the vector space in one hot code into smaller dimensional space. After we vectorize the word/words, we can check their similarity by looking at their inner product.

Word embedding depends on the document, the corpus you have. Based on the sentences in the corpus we have, we have the vector expression of the word. The word with similar meaning are likely to map into adjacent points. It depends on the syntactical meaning (context) around it.

3. Corpus

Corpus is a large and structured set of texts. It is a body of written or spoken material upon which a linguistic analysis is based. It may be single word corpus ('I', 'like', 'coding') or multiple words corpus ('I like', 'like coding').

The difference is the vocabulary is all the discrete, unique words in the documents or books we collect. But corpus can be the collection of structural words, like I mentioned above. For 3 words corpus, the set can be ('I like playing', 'like playing basketball')

4. Train

a. Preprocessing and modeling techniques

1. Read the data from ‘PrideNprejudice.csv’ and look at what is in the csv.

2. import the ntlk package and specify the stopwords. I also specify the cleaning techniques ‘Original’, ‘PorterStemmer’, ‘WordNetLemmatizer’. Here I just use the ‘Original’ cleaning method. 1). I removed the HTML tags first. 2); Then used regex to remove all special characters (only keep letters); 3). Made strings to lower case and tokenize / word split sentences. 4). Remove English stopwords. 5). Rejoin to one string.

After we clean the sentences, we can see they are split into single words

3. Then I use the word2vec package in genism.models package. The parameters I specified are:

Num_features = 300, which is the word vector dimensionality

Min_word_count = 15, which ignore all words with total frequency lower than 15

Num_workers = 4, which is the number of threads to run in parallel

Context = 10, which is the context window size

Iter = 50, which is the iteration times. Since the corpus is small, here I chose a larger iteration times

Then I use the model to train on the data I cleaned.

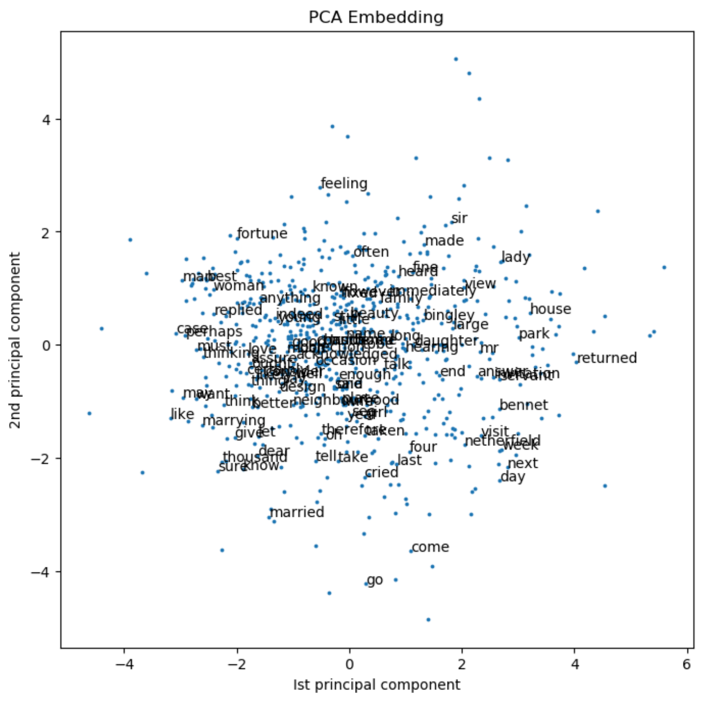
After I trained the model, I use PCA to show the 1st and 2nd components of the feature space using the sklearn decomposition model.

b. vocabulary count, embedding size, number of training iterations

vocabulary count = 689, embedding size = 300, number of training iterations = 20

c. Observations

As we can see in this figure, the points for 2 principal components decomposition distributed like a ball. Many of them center at (0,0). Some other points have high values of 1st and 2nd principal components values. For example, the lady (2.7.2.1) means the 1st and 2nd principal component features basically contribute the same to the word.



d. Evaluation results

Most Similar

```python
model.most_similar(positive=['happy','woman'], negative=['man'])

```

```
[('affected', 0.5009474158287048),
 ('address', 0.417265385389328),
 ('effect', 0.39719319343566895),
 ('itwas', 0.3294943571090698),
 ('still', 0.32695716619491577),
 ('composure', 0.31650999188423157),
 ('importance', 0.312979131937027),
 ('either', 0.309382259845573364),
 ('make', 0.3090117871761322),
 ('easy', 0.3062306046485901)]
```

```python
model.most_similar("man")
```

```
[('woman', 0.7938008308410645),
 ('sort', 0.7659292221069336),
 ('fortune', 0.7252484560012817),
 ('assure', 0.6499446630477905),
 ('connection', 0.6376143097877502),
 ('amiable', 0.6276503801345825),
 ('people', 0.614795446395874),
 ('son', 0.6039210557937622),
 ('sensible', 0.5997189283370972),
 ('men', 0.5992419123649597)]
```

```
model.most_similar("elizabeth")
```

```
[('smile', 0.7637635469436646),
 ('reply', 0.7290018796920776),
 ('composure', 0.6885267496109009),
 ('sat', 0.6862056255340576),
 ('length', 0.6860840320587158),
 ('knew', 0.6821208000183105),
 ('praise', 0.6712562441825867),
 ('delight', 0.6683076620101929),
 ('inquiry', 0.6619987487792969),
 ('spoke', 0.6569844484329224)]
```

From the most similar words analysis we can see what words always show up with the word we input. For example, Elizabeth has characteristics of always smiling, being delighted.

Similarity

```
In [158]:  model.similarity('elizabeth','woman')
Out[158]:  -0.28346802518207553

In [159]:  model.similarity('satisfaction','happy')
Out[159]:  0.076938486974767725

In [160]:  model.similarity('serious','anxiety')
Out[160]:  0.34395505233311963
```

From the results above we can see, the model is not doing good in the similarity analysis. If I increase the iterations, the values all tend to be above 0.98. This may be because of the small sample size.

Does not match

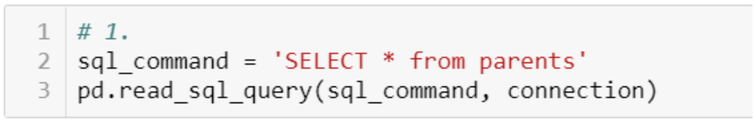
```
In [161]:  model.doesnt_match("man woman power".split())
Out[161]:  'power'

In [162]:  model.doesnt_match("father aunt cousin happy".split())
Out[162]:  'happy'

In [163]:  model.doesnt_match("charming fancy happy cousin".split())
Out[163]:  'cousin'
```

As we see, the does not match function is doing good to split the word which does not match the others.

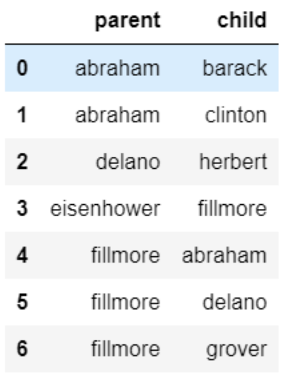
Part 3 SQL

3.1 Simple SELECT

1.

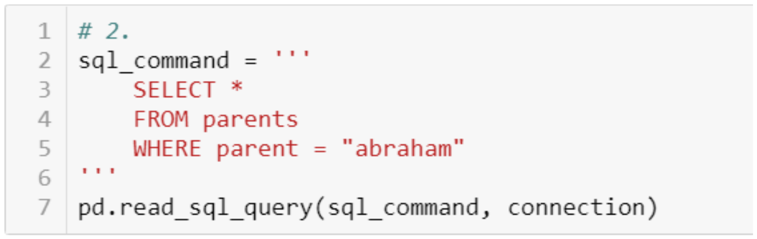


```python
# 1.
sql_command = 'SELECT * from parents'
pd.read_sql_query(sql_command, connection)
```

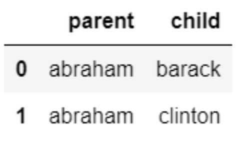


	parent	child
0	abraham	barack
1	abraham	clinton
2	delano	herbert
3	eisenhower	fillmore
4	fillmore	abraham
5	fillmore	delano
6	fillmore	grover

2.

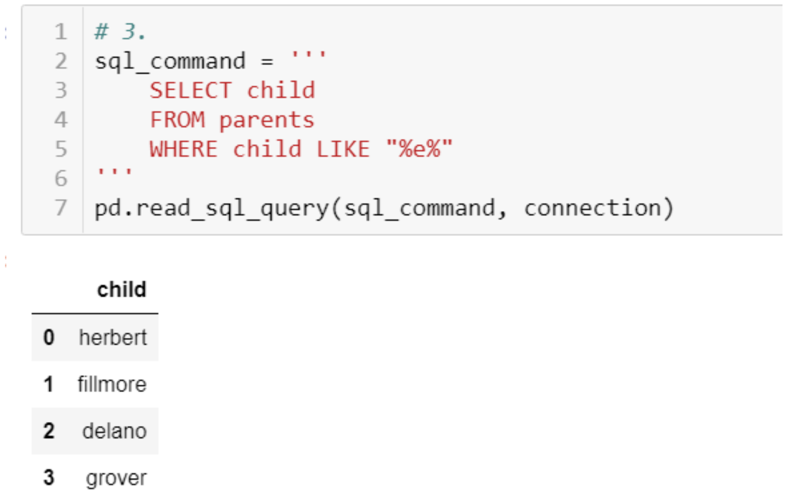


```python
# 2.
sql_command = '''
    SELECT *
    FROM parents
    WHERE parent = "abraham"
'''
pd.read_sql_query(sql_command, connection)
```



	parent	child
0	abraham	barack
1	abraham	clinton

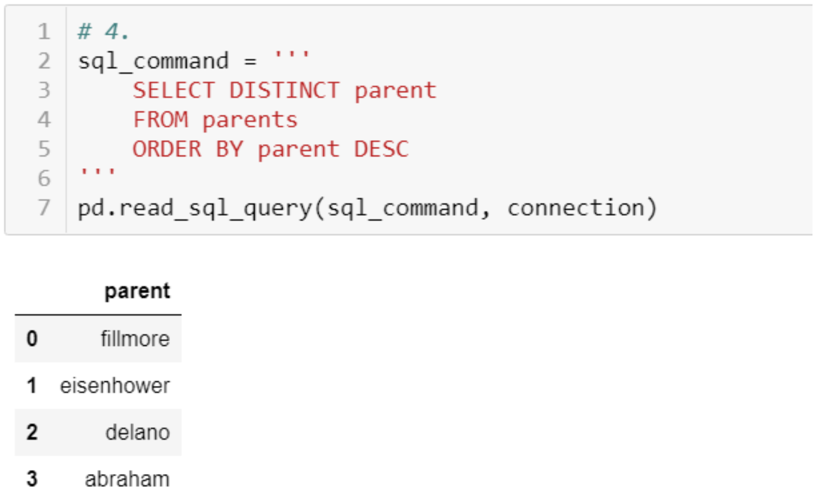
3.



```python
# 3.
sql_command = '''
    SELECT child
    FROM parents
    WHERE child LIKE "%e%"
'''
pd.read_sql_query(sql_command, connection)
```

	child
0	herbert
1	fillmore
2	delano
3	grover

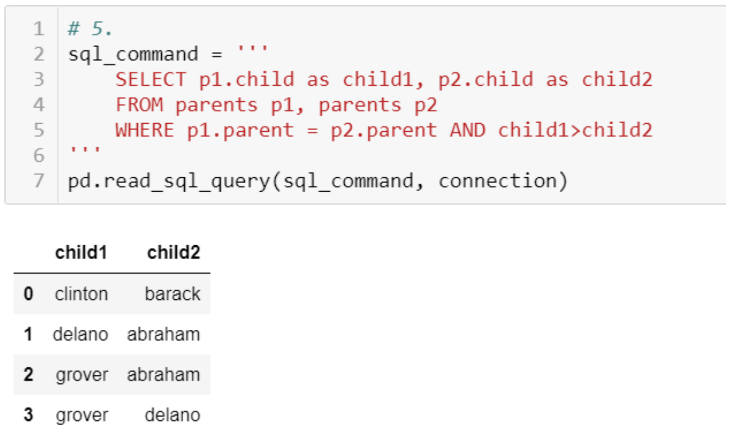
4.



```python
# 4.
sql_command = '''
    SELECT DISTINCT parent
    FROM parents
    ORDER BY parent DESC
'''
pd.read_sql_query(sql_command, connection)
```

	parent
0	fillmore
1	eisenhower
2	delano
3	abraham

5.

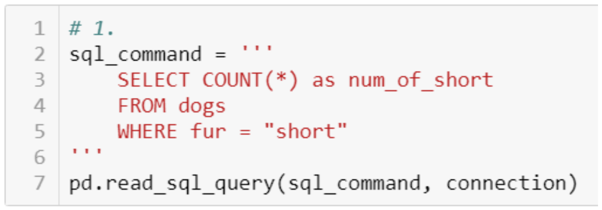


```python
# 5.
sql_command = '''
    SELECT p1.child as child1, p2.child as child2
    FROM parents p1, parents p2
    WHERE p1.parent = p2.parent AND child1>child2
'''
pd.read_sql_query(sql_command, connection)
```

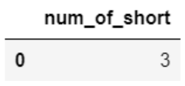
	child1	child2
0	clinton	barack
1	delano	abraham
2	grover	abraham
3	grover	delano

3.2 JOINS

1.

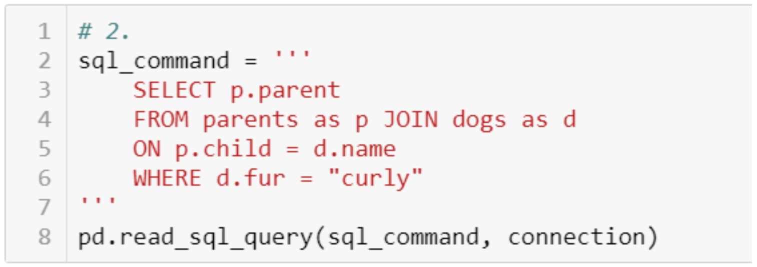


```
# 1.
sql_command = '''
    SELECT COUNT(*) as num_of_short
    FROM dogs
    WHERE fur = "short"
'''
pd.read_sql_query(sql_command, connection)
```

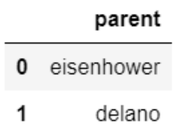


```
   num_of_short
0             3
```

2.

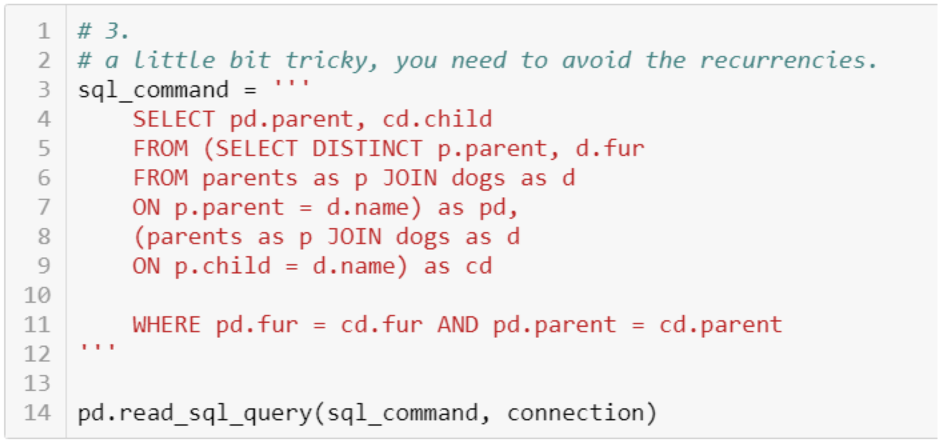


```
# 2.
sql_command = '''
    SELECT p.parent
    FROM parents as p JOIN dogs as d
    ON p.child = d.name
    WHERE d.fur = "curly"
'''
pd.read_sql_query(sql_command, connection)
```

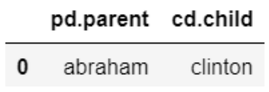


```
       parent
0  eisenhower
1      delano
```

3.



```
# 3.
# a little bit tricky, you need to avoid the recurrencies.
sql_command = '''
    SELECT pd.parent, cd.child
    FROM (SELECT DISTINCT p.parent, d.fur
    FROM parents as p JOIN dogs as d
    ON p.parent = d.name) as pd,
    (parents as p JOIN dogs as d
    ON p.child = d.name) as cd

    WHERE pd.fur = cd.fur AND pd.parent = cd.parent
'''


pd.read_sql_query(sql_command, connection)
```



```
   pd.parent  cd.child
0  abraham    clinton
```

3.3 Aggregate functions

1.

```
# 1.
sql_command = '''
SELECT kind, MIN(weight)
FROM animals;
'''
pd.read_sql_query(sql_command,connection)
```

	kind	MIN(weight)
0	parrot	6

2.

```
# 2.
sql_command = '''
SELECT AVG(legs), AVG(weight)
FROM animals;
'''
pd.read_sql_query(sql_command,connection)
```

	AVG(legs)	AVG(weight)
0	3.0	2009.333333

3.

```
# 3.
sql_command = '''
SELECT kind, weight, legs
FROM animals
WHERE legs > 2 AND weight < 20;
'''
pd.read_sql_query(sql_command,connection)
```

	kind	weight	legs
0	cat	10	4
1	ferret	10	4

4.

```
# 4.
sql_command = '''
SELECT legs,AVG(weight)
FROM animals
GROUP BY legs;
'''
pd.read_sql_query(sql_command,connection)
```

	legs	AVG(weight)
0	2	4005.333333
1	4	13.333333