B565 HW3

1. (a) The hamming distance between the two documents will be the number of characters that are different between the two. Given s1 = ATCGTACGTGTA, and s2 = TCGTACGTGTAA, we can see that the first 11 characters are different. Hence, the Hamming distance between them is 11.

(b) To calculate vectors of 2 shingles we find letters in unique pairs of two.
For S1, [AT, TC, CG, GT, TA, TG, AC] – CG and GT are repeated
For S2, [TC, CG, GT, TA, AC, TG, AA] – CG, GT, and TA are repeated

Jaccard similarity is the no. of shingles in intersection of the two divided by no.of shingles in the union of the two = (TC, CG, GT, TA, AC, TG) / (AT, TC, CG, GT, TA, TG, AC, AA) = 6/8 = 0.75

(c) It depends on the use case. Hamming takes into account the order of arrangement. But Jaccard takes shingles and calculates how much of it is there in the document. So the individual must decide which is more suitable for their use case accordingly.

(d) In Jaccard similarity, we need to iterate through all the shingles and compare the documents. So if we are going to iterate through every shingle in all the documents, the time complexity will be O(n).

2. (a) We just need to for every possible pair of documents calculate the no. of intersection in shingles and divide that by no. of union in shingles.

For d1, d2 = 2/4 = 0.5
For d1, d3 = 1/4 = 0.25
For d1, d4 = 2/4 = 0.5
For d1, d5 = 2/4 = 0.5
For d1, d6 = 2/4 = 0.5

For d2, d3 = ¼ = 0.25
For d2, d4 = ⅕ = 0.2
For d2, d5 = 2/4 = 0.5
For d2, d6 = 2/4 = 0.5

For d3, d4 = 0/5 = 0
For d3, d5 = 0/5 = 0
For d3, d6 = 0/5 = 0

For d4, d5 = 2/4 = 0.5
For d4, d6 = 2/4 = 0.5

For d5, d6 = 3/3 = 1

(b) We need to for each and every document substitute respectively all the shingle values and solve for them:

For 1st document:

h1(0) = 1 % 6 = 1

h1(1) = 3 % 6 = 1

h1(2) = 5 % 6 = 1

h1(3) = 7 % 6 = 1

h1(4) = 9 % 6 = 1

h1(5) = 11 % 6 = 1

For 2nd document:

h2(0) = 2 % 6 = 2

h2(1) = 5 % 6 = 5

h2(2) = 8 % 6 = 2

h2(3) = 11 % 6 = 5

h2(4) = 14 % 6 = 2

h2(5) = 17 % 6 = 5

For 3rd document:

h3(0) = 2 % 6 = 2

h3(1) = 7 % 6 = 1

h3(2) = 12 % 6 = 0

h3(3) = 17 % 6 = 5

h3(4) = 22 % 6 = 4

h3(5) = 27 % 6 = 3

For 4th document:

h4(0) = 3 % 6 = 3

h4(1) = 10 % 6 = 4

h4(2) = 17 % 6 = 5

h4(3) = 24 % 6 = 0

h4(4) = 31 % 6 = 1

h5(5) = 38 % 6 = 2

Sorting according to the input values 0,1,2,3,4,5

Shingle_ID	d1	d2	d3	d4
0	1	2	2	3
1	3	5	1	4
2	5	2	0	5
3	1	5	5	0
4	3	2	4	1
5	5	5	3	2

Finally we can calculate the signature matrix as:

	d1	d2	d3	d4	d5	d6
1	1	1	3	1	1	1
2	2	2	2	2	2	2
3	0	0	0	2	3	3
4	0	0	4	0	0	0

To calculate this we take the minimum of the values where shingle id is 1

(c)

For d1, d2 = 4/4 = 1
For d1, d3 = 2/4 = 0.5
For d1, d4 = 3/4 = 0.75
For d1, d5 = 3/4 = 0.75
For d1, d6 = 3/4 = 0.75

For d2, d3 = 2/4 = 0.5
For d2, d4 = 3/4 = 0.75
For d2, d5 = 3/4 = 0.75
For d2, d6 = 3/4 = 0.75

For d3, d4 = 1/4 = 0.25
For d3, d5 = 1/4 = 0.25
For d3, d6 = 1/4 = 0.25
For d4, d5 = 3/4 = 0.75
For d4, d6 = 3/4 = 0.75

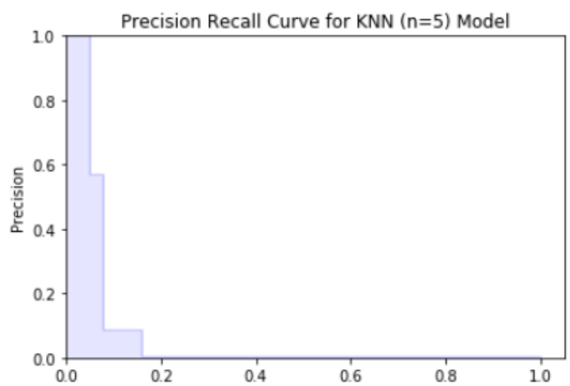
For d5, d6 = 4/4 = 1

3. Yes there are quite a few data processing steps being conducted prior to running KNN. First, was to check if there were any missing values in the dataset. Second, was converting the output column y to type "bool". Third, the values of the column Amount are being normalized. Finally, the values are being split into train and test datasets. Furthermore, the distance metric used in the start code of KNN is "minkowski".

Default settings of KNN

```
#KNN
from sklearn.neighbors import KNeighborsClassifier
#train
knn = KNeighborsClassifier(n_neighbors=5, metric= 'minkowski', p=2)
knn.fit(X_train, y_train.ravel())
#test
y_pred_knn = knn.predict(X_test)
y_prob_knn = knn.predict_proba(X_test)

evaluate_model(y_test, y_pred_knn, y_prob_knn[:, [1]], 'KNN (n=5)')
```

```
Confusion Matrix for KNN (n=5) Model
[[56861     0]
 [   96     5]]
Classification Report for KNN (n=5) Model
              precision    recall  f1-score   support

       False   0.998315  1.000000  0.999157     56861
        True   1.000000  0.049505  0.094340       101

    accuracy                       0.998315     56962
   macro avg   0.999157  0.524752  0.546748     56962
weighted avg   0.998318  0.998315  0.997552     56962

Area under under ROC curve for KNN (n=5) Model
0.5777933195088735
```



K value set to 7; metric set to default

```python
#KNN
from sklearn.neighbors import KNeighborsClassifier
#train
knn = KNeighborsClassifier(n_neighbors=7, metric= 'minkowski', p=2)
knn.fit(X_train, y_train.ravel())
#test
y_pred_knn = knn.predict(X_test)
y_prob_knn = knn.predict_proba(X_test)

evaluate_model(y_test, y_pred_knn, y_prob_knn[:, [1]], 'KNN (n=7)')
```

```
Confusion Matrix for KNN (n=7) Model
[[56861     0]
 [   97     4]]
Classification Report for KNN (n=7) Model
              precision    recall  f1-score   support

       False   0.998297  1.000000  0.999148     56861
        True   1.000000  0.039604  0.076190       101

    accuracy                       0.998297     56962
   macro avg   0.999148  0.519802  0.537669     56962
weighted avg   0.998300  0.998297  0.997511     56962

Area under under ROC curve for KNN (n=7) Model
0.5769785829992576
```

We can see just from the confusion matrix that the number of predictions it got right has come down by 1. As a result, the f1-score of the True predictions has come down by ~ 0.02.

K value set to 3; metric set to default

```
#KNN
from sklearn.neighbors import KNeighborsClassifier
#train
knn = KNeighborsClassifier(n_neighbors=3, metric= 'minkowski', p=2)
knn.fit(X_train, y_train.ravel())
#test
y_pred_knn = knn.predict(X_test)
y_prob_knn = knn.predict_proba(X_test)

evaluate_model(y_test, y_pred_knn, y_prob_knn[:, [1]], 'KNN (n=3)')
```

```
Confusion Matrix for KNN (n=3) Model
[[56859     2]
 [   93     8]]
Classification Report for KNN (n=3) Model
              precision    recall  f1-score   support

       False   0.998367  0.999965  0.999165     56861
        True   0.800000  0.079208  0.144144       101

    accuracy                       0.998332     56962
   macro avg   0.899184  0.539586  0.571655     56962
weighted avg   0.998015  0.998332  0.997649     56962

Area under under ROC curve for KNN (n=3) Model
0.5685685485240106
```

Interestingly, the f1 score has improved when K = 3. We can see that the f1-score of True predictions has gone up by a decent amount. Due to this, we see a slight improvement in the accuracy.

K value set to 3; metric set to Correlation

```
#KNN
from sklearn.neighbors import KNeighborsClassifier
#train
knn = KNeighborsClassifier(n_neighbors=3, metric= 'correlation', p=2)
knn.fit(X_train, y_train.ravel())
#test
y_pred_knn = knn.predict(X_test)
y_prob_knn = knn.predict_proba(X_test)

evaluate_model(y_test, y_pred_knn, y_prob_knn[:, [1]], 'KNN (n=3)')
```

```
Confusion Matrix for KNN (n=3) Model
[[56855     6]
 [   30    71]]
Classification Report for KNN (n=3) Model
              precision    recall  f1-score   support

       False   0.999473  0.999894  0.999684     56861
        True   0.922078  0.702970  0.797753       101

    accuracy                       0.999368     56962
   macro avg   0.960775  0.851432  0.898718     56962
weighted avg   0.999335  0.999368  0.999325     56962

Area under under ROC curve for KNN (n=3) Model
0.8711421686478455
```

The results of these settings are very interesting because the f1-score of True observations has shot up dramatically. We see in the confusion matrix the value at [1,1] is 71. And accuracy has gone up as well to 0.999368. So we can say that for this case correlation is yielding us better results than minkowski.

K value set to 3; metric set to Cosine

```
#KNN
from sklearn.neighbors import KNeighborsClassifier
#train
knn = KNeighborsClassifier(n_neighbors=3, metric= 'cosine', p=2)
knn.fit(X_train, y_train.ravel())
#test
y_pred_knn = knn.predict(X_test)
y_prob_knn = knn.predict_proba(X_test)

evaluate_model(y_test, y_pred_knn, y_prob_knn[:, [1]], 'KNN (n=3)')
```

```
Confusion Matrix for KNN (n=3) Model
[[56853     8]
 [   30    71]]
Classification Report for KNN (n=3) Model
              precision    recall  f1-score   support

       False   0.999473  0.999859  0.999666     56861
        True   0.898734  0.702970  0.788889       101

    accuracy                       0.999333     56962
   macro avg   0.949103  0.851415  0.894277     56962
weighted avg   0.999294  0.999333  0.999292     56962

Area under under ROC curve for KNN (n=3) Model
0.8711322434542041
```

For this, the accuracy is just a little better, but it shows that cosine might be better than minkowski for our case. Additionally, the f1-score is still pretty high for the True observations.

4.

Keywords from the given documents/tweets:

```
import numpy as np

vectorizer = CountVectorizer(stop_words='english', ngram_range=(2, 2), min_df=2)
X = vectorizer.fit_transform(corpus)
#print("stop words removed:", vectorizer.stop_words_)
words = vectorizer.get_feature_names_out()
print(f"#words {len(words)} {words}")
```

✓ 0.0s

```
#words 22 ['antiviral effect' 'caused omicron' 'cihr extending' 'cihr_irsc light'
 'covid19 cihr' 'disruptions caused' 'effect omicron'
 'extending registration' 'freethinkfacts rt' 'ivermectin showed'
 'japan kowa' 'kowa says' 'light disruptions' 'omicron variant'
 'registration application' 'rt cihr_irsc' 'rt erictopol'
 'rt yaneerbaryam' 'says ivermectin' 'showed antiviral'
 'svictor70973566 rt' 'variant covid19']
```

Checking the presence of each keyword in the each document:

```
X_array = X.toarray()
print(X_array)
```

✓ 0.0s

```
[[0 1 1 1 1 1 0 1 0 0 0 0 1 1 1 1 0 0 0 0 0 1]
 [1 0 0 0 0 0 1 0 0 1 1 1 0 0 0 0 0 0 1 1 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 1 1 1 1 1 0 1 0 0 0 0 1 1 1 1 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 1 1 1 1 1 0 1 0 0 0 0 1 1 1 1 0 0 0 0 0 1]
 [1 0 0 0 0 0 1 0 0 1 1 1 0 0 0 0 0 0 1 1 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0]]
```

Converting the features into hashed binary format:

```
1011010011001011100010111010111000100101110011001110101101000010
0001101101011010011011001110110101101110010111011111100010000010
0101111111011000100010110000001001100001011000000101111000010000001
1010100111010101000001101111010010110000001101010111111111010111
1100010101000100110011001001000111110110100010111001010110011011
1111101010001011010111001110100110110011100111000110100011001010
0111011011001100111111110111111010110101001100100000111001110001
0100011101101011110110011000110000001001010011111000110111001001
1111010111101101011011001010101010001101100001010000001111101111
0111111000101101011100111101010101101100101000001101111110001100
1000010000101010001000110111110010000011001010111011011101110111
1101000011100101101011001110100100011110110101000011100101111111
1011001000000110010001111001011001010011111010001101110101010010
0010101000011100010111011100110110011111101011010000000100000101
1111111010100010010101110101001101000010010001111001111001100110
1001101111000100110011000011100100110100101001000101110000011110
0000001011101000100111000101100010101010001000111001101111001001
1001110011111111000001010110011101111010110110110110000110001110
1110000000101110111001101101010100000001110010001110000111011010
1001011010111011101110111100101000000011000011010011111001011011
1010100111100011000001010111111110110010111111001110001011100001
0100011011000010110111000101101100111000110111101111000110111111
```

Sample output of the SimHashed values:

```
print(*SIMHASH,sep='\n')
```

```
✓ 0.0s
```

```
[1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0,
[1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1,
[1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0,
[1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1,
[1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0,
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
[1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1,
[0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0,
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
[1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0,
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
[1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0,
[1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1,
[1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1,
```

Calculating the distances hamming distance and euclidean distance between the documents/tweets:

```
Similarity matrix of Hamming distance
[0, 39, 26, 30, 0, 29, 30, 29, 29, 23, 29, 29, 0, 39, 30]
[39, 0, 31, 29, 39, 24, 29, 32, 24, 34, 24, 24, 39, 0, 29]
[26, 31, 0, 22, 26, 15, 22, 21, 15, 13, 15, 15, 26, 31, 22]
[30, 29, 22, 0, 30, 11, 0, 41, 11, 25, 11, 11, 30, 29, 0]
[0, 39, 26, 30, 0, 29, 30, 29, 29, 23, 29, 29, 0, 39, 30]
[29, 24, 15, 11, 29, 0, 11, 36, 0, 28, 0, 0, 29, 24, 11]
[30, 29, 22, 0, 30, 11, 0, 41, 11, 25, 11, 11, 30, 29, 0]
[29, 32, 21, 41, 29, 36, 41, 0, 36, 34, 36, 36, 29, 32, 41]
[29, 24, 15, 11, 29, 0, 11, 36, 0, 28, 0, 0, 29, 24, 11]
[23, 34, 13, 25, 23, 28, 25, 34, 28, 0, 28, 28, 23, 34, 25]
[29, 24, 15, 11, 29, 0, 11, 36, 0, 28, 0, 0, 29, 24, 11]
[29, 24, 15, 11, 29, 0, 11, 36, 0, 28, 0, 0, 29, 24, 11]
[0, 39, 26, 30, 0, 29, 30, 29, 29, 23, 29, 29, 0, 39, 30]
[39, 0, 31, 29, 39, 24, 29, 32, 24, 34, 24, 24, 39, 0, 29]
[30, 29, 22, 0, 30, 11, 0, 41, 11, 25, 11, 11, 30, 29, 0]
Similarity matrix of Euclidean distance
[0.0, 4.24, 3.61, 3.61, 0.0, 3.32, 3.61, 3.46, 3.32, 3.46, 3.32, 3.32, 0.0, 4.24, 3.61]
[4.24, 0.0, 3.0, 3.0, 4.24, 2.65, 3.0, 2.83, 2.65, 2.83, 2.65, 2.65, 4.24, 0.0, 3.0]
[3.61, 3.0, 0.0, 2.0, 3.61, 1.41, 2.0, 1.0, 1.41, 1.0, 1.41, 1.41, 3.61, 3.0, 2.0]
[3.61, 3.0, 2.0, 0.0, 3.61, 1.41, 0.0, 1.73, 1.41, 1.73, 1.41, 1.41, 3.61, 3.0, 0.0]
[0.0, 4.24, 3.61, 3.61, 0.0, 3.32, 3.61, 3.46, 3.32, 3.46, 3.32, 3.32, 0.0, 4.24, 3.61]
[3.32, 2.65, 1.41, 1.41, 3.32, 0.0, 1.41, 1.0, 0.0, 1.0, 0.0, 0.0, 3.32, 2.65, 1.41]
[3.61, 3.0, 2.0, 0.0, 3.61, 1.41, 0.0, 1.73, 1.41, 1.73, 1.41, 1.41, 3.61, 3.0, 0.0]
[3.46, 2.83, 1.0, 1.73, 3.46, 1.0, 1.73, 0.0, 1.0, 1.41, 1.0, 1.0, 3.46, 2.83, 1.73]
...
[3.32, 2.65, 1.41, 1.41, 3.32, 0.0, 1.41, 1.0, 0.0, 1.0, 0.0, 0.0, 3.32, 2.65, 1.41]
[0.0, 4.24, 3.61, 3.61, 0.0, 3.32, 3.61, 3.46, 3.32, 3.46, 3.32, 3.32, 0.0, 4.24, 3.61]
[4.24, 0.0, 3.0, 3.0, 4.24, 2.65, 3.0, 2.83, 2.65, 2.83, 2.65, 2.65, 4.24, 0.0, 3.0]
[3.61, 3.0, 2.0, 0.0, 3.61, 1.41, 0.0, 1.73, 1.41, 1.73, 1.41, 1.41, 3.61, 3.0, 0.0]
```

And finally calculating the pearson correlation of the distances between all the documents:

```
The pearson correlation between the distances for all pairs of documents =
 [[1.         0.79289978]
 [0.79289978 1.        ]]
```

5.

When given the nodes input as:

A B
A E
B A
B F
B D
B C
C D
D E
D F
E C
E F
F A
F C

The transition Matrix is:

```
Transition Matrix:
[[0.   0.5  0.   0.   0.5  0.  ]
 [0.25 0.   0.25 0.25 0.   0.25]
 [0.   0.   0.   1.   0.   0.  ]
 [0.   0.   0.   0.   0.5  0.5 ]
 [0.   0.   0.5  0.   0.   0.5 ]
 [0.5  0.   0.5  0.   0.   0.  ]]
['A', 'B', 'C', 'D', 'E', 'F']
```

The number of iterations and page ranks for them is as follows:

```
Number of iterations before convergence of values =  38
Page ranks:
[0.12182786 0.06091363 0.20812245 0.22335033 0.17258821 0.21319753]
```

Ranks for each and every node saved to the output file “output.txt”:

```
Rank 1, Node 3
Rank 2, Node 5
Rank 3, Node 2
Rank 4, Node 4
Rank 5, Node 0
Rank 6, Node 1
```

Sample outputs for the Facebook dataset:

The transition Matrix is:

```
Transition Matrix:
[[0.         0.00288184 0.00288184 ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 ...
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]]
```

The number of iterations and page ranks for them is as follows:

```
Number of iterations before convergence of values =  26
Page ranks:
[0.00000000e+00 0.00000000e+00 0.00000000e+00 ... 2.88910531e-28
 2.00987264e-27 7.03401658e-21]
```