link_predict_make_features.ipynb

1) 採集 missing edges 以獲取負樣本提供二元分類器的訓練

針對 data_train_edge.csv 中已有連結的 node1、node2(label=1)繪製 graph，並隨機任選不存在連結邊的兩個 nodes 形成與正樣本數量相同的負樣本(label=0)，合併成此次模型訓練之全部使用資料。

2) Train – validation – test data 的切割

將全部資料以 7:3 切割成「訓練」與「測試」資料；再將「訓練」資料以 8:2 切割成「訓練」與「驗證」資料，每一份切割資料之正、負樣本比例皆相等，避免樣本不平衡導致的訓練困難。所使用之切割資料檔與其資料筆數陳列如下：

Validation data	X_train_train_10.csv	22910 筆
	X_train_valid_10.csv	5728 筆
Test data	X_train_10.csv	28638 筆
	X_test_10.csv	12276 筆
All data	train_all_10.csv	40914 筆
	predict_all_10.csv	10231 筆

3) Extract features

A. Jaccard's_coefficient

```
def cal_jaccard_coefficient(a,b,train_graph):
    try:
        if len(set(train_graph.successors(a))) == 0  | len(set(train_graph.successors(b))) == 0:
            return 0
        sim = (len(set(train_graph.successors(a)).intersection(set(train_graph.successors(b)))))/\
                                    (len(set(train_graph.successors(a)).union(set(train_graph.successors(b)))))
        return sim
    except:
        return 0
```

B. Adamic / Adar_Index

```
def calc_adar_in(a,b,train_graph):
    sum=0
    try:
        n=list(set(train_graph.successors(a)).intersection(set(train_graph.successors(b)))) #同時被 a、b指向者(common friend
        if len(n)!=0:
            for i in n:
                a = len(list(train_graph.predecessors(i))) #每一個common friend的朋友數量(指向 i 的數量)
                if a > 1:
                    sum =sum +(1/np.log10(a)) # i 的朋友數越多，權重佔比越小
                else:
                    continue
            return sum
        else:
            return 0
    except:
        return 0
```

```
X_train['jaccard_coef'] = X_train.apply(lambda row:cal_jaccard_coefficient(row['node1'],row['node2'],train_graph),axis=1)
X_train['adar_index'] = X_train.apply(lambda row: calc_adar_in(row['node1'],row['node2'],train_graph),axis=1)
```

※ 當存在一條 node a -> node b 的有向邊時：

b 為 a 的 successor，即由 node a 向外指向的 node 數量(a 的關注對象)

a 為 b 的 predecessor，也就是向內指向 node b 的 node 數量(b 的追隨者)

※ 此處為 directed graph 的計算，所以我定義「被指向者」為朋友。
ex：a -> b，b 為 a 的朋友，因此，在此定義上，a 的朋友即 successors(a)

C. Shortest path

```
def compute_shortest_path_length(a,b,train_graph):
    p=-1
    try:
        if train_graph.has_edge(a,b):
            train_graph.remove_edge(a,b)
            p= nx.shortest_path_length(train_graph,source=a,target=b)
            train_graph.add_edge(a,b)
        else:
            p= nx.shortest_path_length(train_graph,source=a,target=b)
        return p
    except:
        return -1
```

```
X_train['shortest_path'] = X_train.apply(lambda row:
                                         compute_shortest_path_length(row['node1'],row['node2'],train_graph),axis=1)
```

因為任務是針對現有的 node 中不存在連結者，預測其兩兩間是否可能有 link 的存在，因此在計算 shortest path 時，針對本來已有 link 存在之 node (即 data_train_edge.csv 中的 node1、node2 pairs)，會先移除兩者「直接相連之 edge」後再進行計算，計算完畢再將 edge 加回 graph 中。

D. Follow back

```
def follows_back(a,b,train_graph):
    if train_graph.has_edge(b,a):
        return 1
    else:
        return 0
```

```
X_train['follows_back'] = X_train.apply(lambda row: follows_back(row['node1'],row['node2'],train_graph),axis=1)
```

查看 node 2 是否存在指向 node 1 的邊。任務是預測 node 1 是否會指向 node 2，因此先直觀猜想：若 node 2 本身已指向 node 1，是否 node 1 可能指向回 node 2 的機率會相對高一些？

E. Page Rank

```
def pg_rank(train_graph):
    pr = nx.pagerank(train_graph, alpha=0.85) #dict
    mean_pr = float(sum(pr.values())) / len(pr)
    print("pg rank")
    print('min',pr[min(pr, key=pr.get)])
    print('max',pr[max(pr, key=pr.get)])
    print('mean',mean_pr)
    return pr, mean_pr
```

```
pr, mean_pr = pg_rank(train_graph)
X_train['page_rank_n1'] = X_train.node1.apply(lambda x:pr.get(x,mean_pr))
X_train['page_rank_n2'] = X_train.node2.apply(lambda x:pr.get(x,mean_pr))
```

若欲預測之 node 不存在於訓練集時給予平均值

F. Katz centrality

```
def katz_central(train_graph):
    katz = nx.katz.katz_centrality(train_graph,alpha=0.005,beta=1)
    mean_katz= float(sum(katz.values())) / len(katz)
    print('Katz Centrality')
    print('min',katz[min(katz, key=katz.get)])
    print('max',katz[max(katz, key=katz.get)])
    print('mean',mean_katz)
    return katz, mean_katz
```

```
katz, mean_katz = katz_central(train_graph)
X_train['katz_n1'] = X_train.node1.apply(lambda x: katz.get(x,mean_katz))
X_train['katz_n2'] = X_train.node2.apply(lambda x: katz.get(x,mean_katz))
```

若欲預測之 node 不存在於訓練集時給予平均值

G. Hits (hubs / authorities)

```
def HITS(train_graph):
    hits = nx.hits(train_graph, max_iter=100, tol=1e-08, nstart=None, normalized=True)
    mean_hits = float(sum(hits[0].values())) / len(hits[0])
    print('Hyper-link induced topic search (HITS)')
    print('min',hits[0][min(hits[0], key=hits[0].get)])
    print('max',hits[0][max(hits[0], key=hits[0].get)])
    print('mean',mean_hits,'\n')
    return hits, mean_hits
```

```
hits, mean_hits = HITS(train_graph)
X_train['hubs_n1'] = X_train.node1.apply(lambda x: hits[0].get(x,0))
X_train['hubs_n2'] = X_train.node2.apply(lambda x: hits[0].get(x,0))
X_train['authorities_n1'] = X_train.node1.apply(lambda x: hits[1].get(x,0))
X_train['authorities_n2'] = X_train.node2.apply(lambda x: hits[1].get(x,0))
```

若欲預測之 node 不存在於訓練集時給予零值

以網頁舉例，一個提供有關主題很多資訊的網頁是很有價值的，這種網頁稱之為「authorities」(基於 incoming links 來衡量價值)；一個告訴你有效尋找相關資訊方法的網頁也具價值，這種網頁稱之為「hubs」(基於 outgoing links 來衡量價值)。一個提供連結到 good authorities 的網頁會是 good hub，相對來說，一個被 good hubs 連結到的網頁也會是 good authority。

H. Node1、Node2 的 followers、followees 與交集數量

```
def compute_features_follow(df_final):
    #calculating no of followers followees for node1 and node2
    #calculating intersection of followers and followees for node1 and node2
    num_followers_1=[]
    num_followees_1=[]
    num_followers_2=[]
    num_followees_2=[]
    inter_followers=[]
    inter_followees=[]
    for i,row in df_final.iterrows():
        try:
            n1_p=set(train_graph.predecessors(row['node1']))
            n1_s=set(train_graph.successors(row['node1']))
        except:
            n1_p = set()
            n1_s = set()
```

```
        try:
            n2_p =set(train_graph.predecessors(row['node2']))
            n2_s =set(train_graph.successors(row['node2']))
        except:
            n2_p = set()
            n2_s = set()
        num_followers_1.append(len(n1_p))
        num_followees_1.append(len(n1_s))

        num_followers_2.append(len(n2_p))
        num_followees_2.append(len(n2_s))

        inter_followers.append(len(n1_p.intersection(n2_p)))
        inter_followees.append(len(n1_s.intersection(n2_s)))

    return num_followers_1, num_followers_2, num_followees_1, num_followees_2, inter_followers, inter_followees
```

```
X_train['num_followers_node1'], X_train['num_followers_node2'], \
X_train['num_followees_node1'], X_train['num_followees_node2'], \
X_train['inter_followers'], X_train['inter_followees']= compute_features_follow(X_train)
```

I. Node2vec

```
# Precompute probabilities and generate walks
node2vec = Node2Vec(G, dimensions=64, walk_length=30, num_walks=10)

# Embed nodes
model = node2vec.fit(window=10, min_count=1, batch_words=4)
```

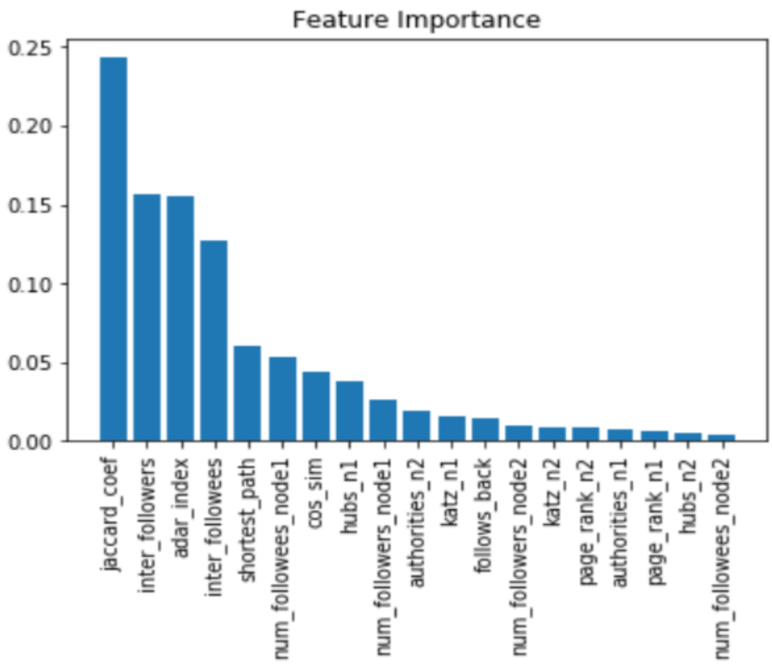
```
def cosin_distance(vector1, vector2):
    dot_product = 0.0
    normA = 0.0
    normB = 0.0
    for a, b in zip(vector1, vector2):
        dot_product += a * b
        normA += a ** 2
        normB += b ** 2
    if normA == 0.0 or normB == 0.0:
        return None
    else:
        return dot_product / ((normA * normB) ** 0.5)

def cos_sim(x_train):
    node_sim = []
    for i,j in x_train[['node1','node2']].values:
        try:
            sim = cosin_distance(model.wv.get_vector(str(i)),model.wv.get_vector(str(j)))
        except:
            sim = -1
        node_sim.append(sim)

    final_sim = np.round(node_sim,3)
    return final_sim
```

```
X_train['cos_sim'] = cos_sim(X_train)
```

得出各 node 的 embedding 向量後，計算 node1、node2 的 vectors 間的 cosine similarity。

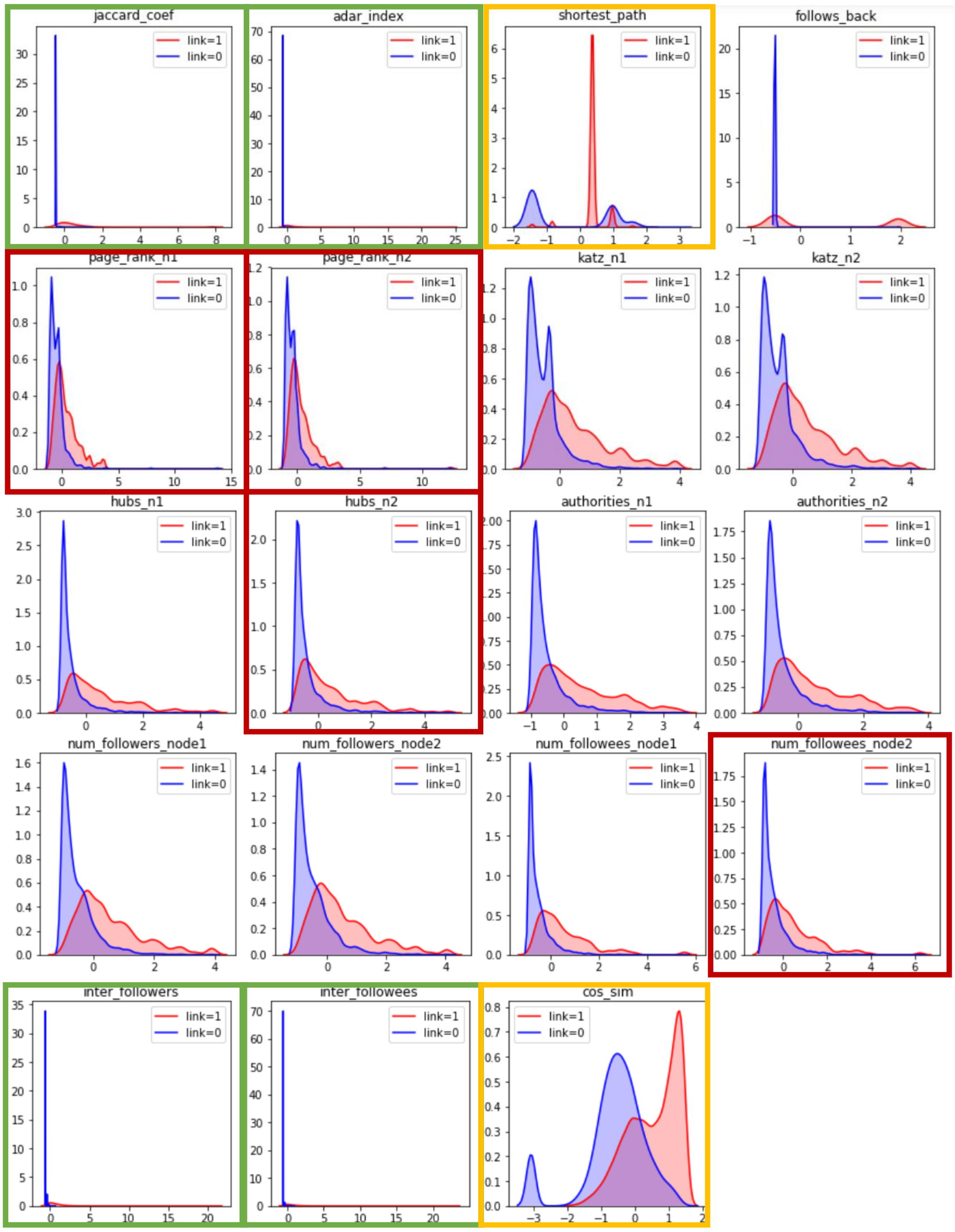
link_predict_visualization.ipynb

1) Feature importance



利用 RandomForestClassifier 完成模型訓練後，得到如上之特徵重要度分數，因各特徵的計算在概念上多多少少都具有重複的部分，因此特徵間很可能存在共線性的問題(對模型之解釋力有所重複)，RandomForest 的好處之一，即在於針對存在此問題的特徵不會同時擁有高的重要度分數，因此可以很方便的作為特徵選擇的參考依據。

2) Discriminative abilities of features



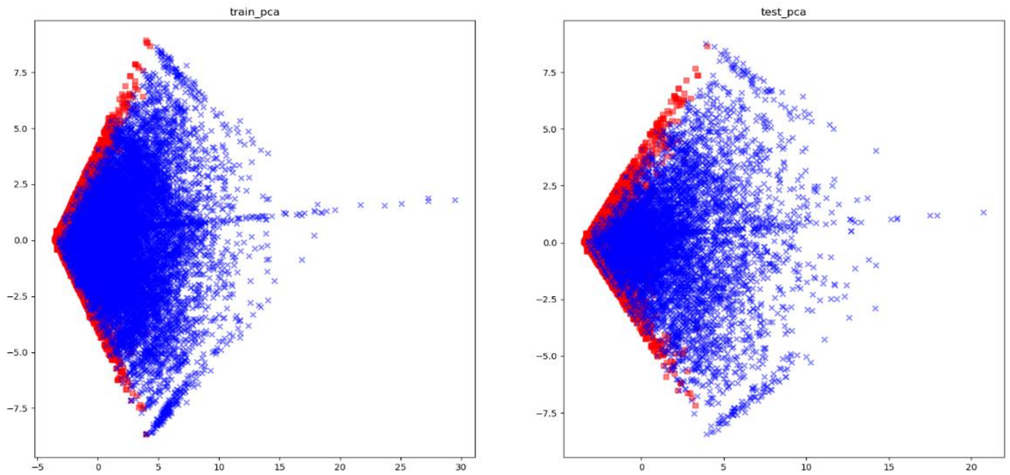
依據各特徵在 label 為 1 或 0 的數值分布繪圖：

A. 紅色框框選處為看起來判別力較差的特徵(分布重合度高，特徵對於區分 link 的幫助不大)，同時也是在上述 feature importance 排序中較為末端的特徵，在訓練模型時可以考慮排除。

B. 綠色框框選處因數值差異大(已標準化處理)，在此圖上不易觀察，但其恰巧皆為 feature importance 排序前段的特徵，因此先保留而不再另外繪製。

C. 黃色框框選處為看起來分布較有差異的特徵。以 node2vec 計算之向量餘弦相似度雖然在 feature importance 的排序偏中段，但從分布圖看來應是相對較具判別力的重要特徵；Shortest_path 則是較為特別的特徵，雖看似有分布上的差異，但其實並非如此，若剔除 link=1 的紅色高峰，便可以發現其餘分布上幾乎是重合的，但其在 feature importance 的高得分反而會讓人忽略了這部分。經實驗證明，shortest_path 的排除確實能帶來更好的預測。

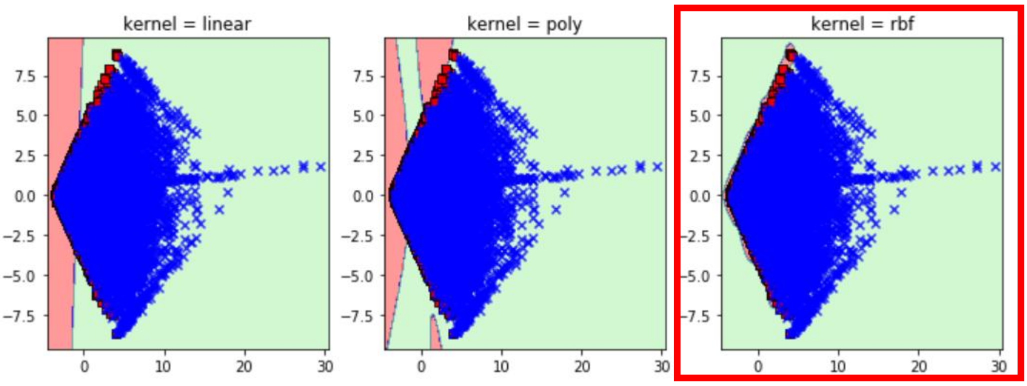
3) PCA



因高維度的特徵無法可視化，因此利用 PCA 進行降維，得出 2 維的主成分萃取供視覺化繪圖。紅色為 link=1 的樣本，藍色為 link=0 的樣本，可以看出存在明顯分界線，但屬於非線性可分的狀態，因此在模型的選擇上可以考慮採用 kernel svm 來進行分類。

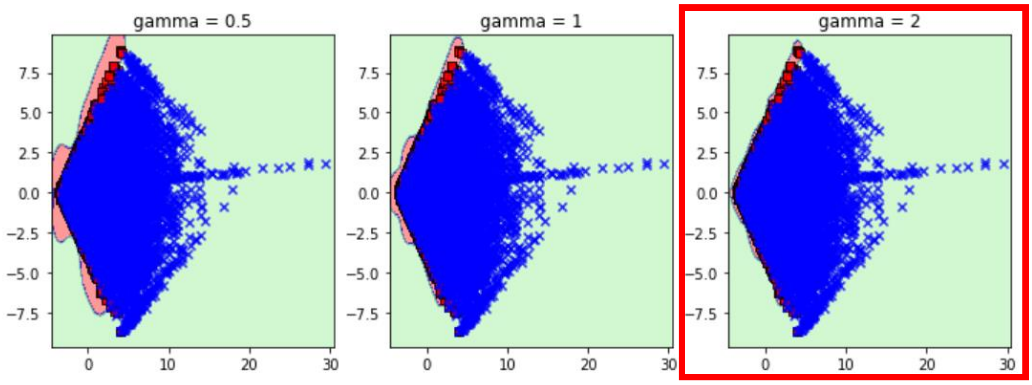
4) SVM parameters

A. Kernel



選擇不同的 kernel 建立 SVM 分類器，並繪製出如上之決策邊界圖，其分類結果明顯以「rbf」為最優。

B. Gamma



gamma 是選擇 RBF 函數作為 kernel 後，該函數自帶的一個參數。決定資料映射到新特徵空間後的分佈，gamma 越大，支持向量越少；gamma 越小，支持向量越多。

link_predict_models.ipynb

1) 訓練過程分為三部分：validation、test、predict

特徵欄位的選擇除了參考上一部分視覺化的結論外，亦嘗試了多種特徵組合，但目前實驗結果以只去除「shortest_path」表現最好。

train-validation

```
x_train_final = pd.read_csv("features/X_train_train_10.csv")
x_test_final = pd.read_csv("features/X_train_valid_10.csv")
print("train： %s, test： %s" % (len(x_train_final),len(x_test_final)))

y_train_final = x_train_final['link']
y_test_final = x_test_final['link']
x_train_final.drop(['node1', 'node2','link','shortest_path'],axis=1,inplace=True)
x_test_final.drop(['node1', 'node2','link','shortest_path'],axis=1,inplace=True)
```

```
train： 22910, test： 5728
```

train-test

```
x_train_final2 = pd.read_csv("features/X_train_10.csv")
x_test_final2 = pd.read_csv("features/X_test_10.csv")
print("train： %s, test： %s" % (len(x_train_final2),len(x_test_final2)))

y_train_final2 = x_train_final2['link']
y_test_final2 = x_test_final2['link']
x_train_final2.drop(['node1', 'node2','link','shortest_path'],axis=1,inplace=True)
x_test_final2.drop(['node1', 'node2','link','shortest_path'],axis=1,inplace=True)
```

```
train： 28638, test： 12276
```

train-predict

```
x_train_final3 = pd.read_csv("features/train_all_10.csv")
x_pred_final = pd.read_csv("features/predict_all_10.csv")
print("train： %s, predict： %s" % (len(x_train_final3),len(x_pred_final)))

y_train_final3 = x_train_final3['link']

x_train_final3.drop(['node1', 'node2','link','shortest_path'],axis=1,inplace=True)
x_pred_final.drop(['node1', 'node2','link','shortest_path'],axis=1,inplace=True)
```

```
train： 40914, predict： 10231
```

2) 使用模型：RandomForest 、 kernel SVM

A. Random Forest

使用了 RandomizedSearchCV 以隨機的方式在參數空間中做採樣，為降低 overfit 的機率，max_depth 的參數範圍設定較小，並僅給予 5 次(n_iter)的參數搜索，再利用結果中表現最佳的參數組合做為最終訓練模型。

```
start_time = time()
param_dist = {"n_estimators":sp_randint(100,150),
              "max_depth": sp_randint(10,20)}

clf = RandomForestClassifier(random_state=25,n_jobs=-1, oob_score=True)
rf_random = RandomizedSearchCV(clf, param_distributions=param_dist,
                                   n_iter=5,cv=10,scoring='accuracy',random_state=25)

rf_random.fit(x_train_final, y_train_final)
print('mean test scores','\n',rf_random.cv_results_['mean_test_score'])
print("--- %s seconds ---" % (time() - start_time))
```

```
mean test scores
 [0.98092536 0.98118725 0.98101266 0.98096901 0.9812309 ]
--- 61.851839780807495 seconds ---
```

```
# build model with the best parameters
clf = rf_random.best_estimator_
print(rf_random.best_estimator_)
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=14, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=105,
                       n_jobs=-1, oob_score=True, random_state=25, verbose=0,
                       warm_start=False)
```

```
clf.fit(x_train_final, y_train_final)
print (clf.oob_score_)

y_pred = clf.predict(x_test_final)
print ("Validation accuracy score: " ,accuracy_score(y_test_final, y_pred))
```

```
0.9813182016586643
Validation accuracy score:  0.9725907821229051
```

```
clf.fit(x_train_final2, y_train_final2)
print (clf.oob_score_)

y_pred = clf.predict(x_test_final2)
print('Testing accuracy score: ',accuracy_score(y_test_final2, y_pred))
```

```
0.9804804804804805
Testing accuracy score:  0.9625285109156076
```

```
clf.fit(x_train_final3, y_train_final3)
print (clf.oob_score_)

y_pred = clf.predict(x_pred_final)
```

```
0.9834042137165763
```

B. SVM

首先，對資料做標準化的動作，而後分別嘗試用「標準化資料」與「PCA 特徵萃取資料」來訓練模型，參數選擇則引用視覺化部分的結論。訓練時兩種資料表現差異不大，若以 kaggle Public Leaderboard 的得分來看，「標準化資料」的表現好一些，若考量計算時間，PCA 也是個不錯的選擇。

```
def train_test_std(x_train_final, x_test_final):
    scaler = preprocessing.StandardScaler().fit(x_train_final)
    x_train_final_std = pd.DataFrame(scaler.transform(x_train_final.values), columns=x_train_final.columns)
    x_test_final_std = pd.DataFrame(scaler.transform(x_test_final.values), columns=x_test_final.columns)
    return x_train_final_std, x_test_final_std
```

train-validation

```
x_train_final_std, x_test_final_std = train_test_std(x_train_final, x_test_final)
```

```
#PCA
pca = PCA(n_components=6)
x_train_pca = pca.fit_transform(x_train_final_std)
x_test_pca = pca.transform(x_test_final_std)

clf = svm.SVC(kernel='rbf', gamma=2)
clf.fit(x_train_pca ,  y_train_final)
svm_pred = clf.predict(x_test_pca)
print('Validation accuracy score: ',accuracy_score(y_test_final, svm_pred))
```

```
Validation accuracy score:  0.9626396648044693
```

```
#std
clf = svm.SVC(kernel='rbf', gamma=2)
clf.fit(x_train_final_std,  y_train_final)
svm_pred = clf.predict(x_test_final_std)
print('Validation accuracy score: ',accuracy_score(y_test_final, svm_pred))
```

```
Validation accuracy score:  0.9612430167597765
```

train-test

```
x_train_final_std, x_test_final_std = train_test_std(x_train_final2, x_test_final2)
```

```
#PCA
pca = PCA(n_components=6)
x_train_pca = pca.fit_transform(x_train_final_std)
x_test_pca = pca.transform(x_test_final_std)

clf = svm.SVC(kernel='rbf', gamma=2)
clf.fit(x_train_pca ,  y_train_final2)
svm_pred = clf.predict(x_test_pca)
print('Test accuracy score: ',accuracy_score(y_test_final2, svm_pred))
```

```
Test accuracy score:  0.9625285109156076
```

```
#std
clf = svm.SVC(kernel='rbf', gamma=2)
clf.fit(x_train_final_std,  y_train_final2)
svm_pred = clf.predict(x_test_final_std)
print('Test accuracy score: ',accuracy_score(y_test_final2, svm_pred))
```

```
Test accuracy score:  0.9625285109156076
```

train_all-predict

```
x_train_final_std, x_pred_final_std = train_test_std(x_train_final3, x_pred_final)
```

```
#PCA
pca = PCA(n_components=6)
x_train_pca = pca.fit_transform(x_train_final_std)
x_pred_pca = pca.transform(x_pred_final_std)

clf = svm.SVC(kernel='rbf', gamma=2)
clf.fit(x_train_pca ,  y_train_final3)
svm_pred = clf.predict(x_pred_pca)
print("# of rows: %d, # of 1 predictions: %d" % (len(svm_pred), sum(svm_pred)))
```

```
# of rows: 10231, # of 1 predictions: 5544
```

```
#std
clf = svm.SVC(kernel='rbf', gamma=2)
clf.fit(x_train_final_std,  y_train_final3)
svm_pred = clf.predict(x_pred_final_std)
print("# of rows: %d, # of 1 predictions: %d" % (len(svm_pred), sum(svm_pred)))
```

```
# of rows: 10231, # of 1 predictions: 5626
```

以上程式截圖畫面僅做為處理流程與演算法架構的輔助說明，而非是最佳模型的實際產出結果，但差異不會很大，RandomForest 的訓練過程差不多介於 96%~98%的 accuracy 範圍；SVM 則是介於 95%~97%的 accuracy 範圍；實際 kaggle Public Leaderboard 的 score 則是介於 89%~91%之間。無論使用哪一種模型，都有些 overfit 的現象，但以 SVM 模型稍微好一些。

※ 此次作業亦嘗試了 logistic、xgboost、lightgbm 等模型，但由於表現較不穩定(起伏大、參數調校耗時)，便不選擇為此次的主要使用模型。

※ 參考網頁：https://medium.com/@vgnshiyer/link-prediction-in-a-social-network-df230c3d85e6