Implementation Details

In order to identify the named entity, I tried lots of features which includes term frequency, inverse document frequency, min edit distance algorithm, bm25 algorithm, cosine similarity algorithm and so on. After getting these features, deal with them by using XGBoost.

Step 1: NLP

```
# men_docs: dic
def get_related_dic(men_docs):
    nlp = sp.load("en_core_web_sm")
    token_dic = {}
    entity_dic = {}
    token_attr_dic = {}
    token_offset_dic = {}
    doc_token_list = {}
    title_dic = {}
    for key, value in men_docs.items():
        title = str(key)
        doc = nlp(value)
        token_attr_dic[title] = []
        token_offset_dic[title] = {}
        subject = ''
        doc_token_list[title] = []
        for word in value:
            subject += word
            if word == '.':
                title_dic[key] = subject.lower()
                break
        for token in doc:
```

Use nlp to handle document and candidate parsed page. Store related data in the dictionaries.

Step 2: Get Features

```
    # # feature_4 entity tf_idf in doc
    # # Accuracy =  0.5880281690140845  Accuracy =  0.3105590062111801
    # temp_list4 = feature_entity_tf_idf(candidate_entities, ent_dic, doc_num, doc_title)
    # index_list4 = [i for i, x in enumerate(temp_list4) if x == max(temp_list4)]
    # # feature_4 list used for XGBoost
    # feature_4.extend(temp_list4)
    # # if data_label[mention_id] is None and len(index_list4) == 1:
    # data_label[mention_id] = candidate_entities[index_list4[0]]

    # feature_5: min edit distance
    # Accuracy =  0.6725352112676056 Accuracy =  0.484472049689441
    temp_list5 = []
    for candidate_entity in candidate_entities:
        temp_list5.append(min_distance(mention, candidate_entity))
    index_list5 = [i for i, x in enumerate(temp_list5) if x == min(temp_list5)]
    feature_5.extend(temp_list5)
    data_label[mention_id] = candidate_entities[index_list5[0]]

    temp_list6 = bm25(query, parsed_doc, candidate_entities, parsed_tf_dic, avg_length)
    feature_6.extend(temp_list6)
    index_list6 = [i for i, x in enumerate(temp_list6) if x == max(temp_list6)]
    data_label[mention_id] = candidate_entities[index_list6[0]]
```

Pass dev mentions or train mentions, get different features. According to every mention, candidate entity pair, get one corresponding feature value. Store values in the feature list.

Step 3: Differet Features

1) Bm25:

```
def bm25(query, parsed_doc_dic, candidate_entities, parsed_tf_dic, avg_length):
    k1 = 1
    k2 = 1
    b = 0.75
    N = len(parsed_doc_dic)
    query = query.split()
    score_list = []
    for entity in candidate_entities:
        score = 0
        for q in query:
            try:
                W = np.log((N + 0.5) / (len(parsed_tf_dic[q]) + 0.5))
                fi = parsed_tf_dic[q][entity]
            except KeyError:
                fi = 0
                W = np.log((N + 0.5) / 0.5)
            q_fi = query.count(q)
            K = k1 * (1 - b + b * len(parsed_doc_dic[entity].split()) / avg_length)
            r = (fi * (k1 + 1)) / (fi + K) * q_fi * (k2 + 1) / (q_fi + k2)
            score += W * r
        score_list.append(score)
    return score_list
```

I use the first sentence of the document (most of times it is the title of the document which is all uppercase) as the query in the bm25 algorithm and the candidate entity parsed page as the document in the bm25 algorithm.

2) Min edited distance

```
def min_distance(word1, word2):
    if not word1:
        return len(word2 or '') or 0
    if not word2:
        return len(word1 or '') or 0
    size1 = len(word1)
    word2 = word2[0: size1]
    size2 = len(word2)
    last = 0
    tmp = list(range(size2 + 1))
    value = None
    for i in range(size1):
        tmp[0] = i + 1
        last = i
        # # print word1[i], last, tmp
        for j in range(size2):
            if word1[i] == word2[j]:
                value = last
            else:
                value = 1 + min(last, tmp[j], tmp[j + 1])
                # # print(last, tmp[j], tmp[j + 1], value)
            last = tmp[j+1]
            tmp[j+1] = value
        # # print tmp
    return value
```

3) cosine similarity

```
temp_list7 = []
for entity in candidate_entities:
    parsed_key_token = parsed_pages_key_token_dic[entity]
    # union_key_token = list(set(doc_key_token + parsed_key_token + mention.split()))
    union_key_token = []
    for token in doc_key_token:
        if token not in union_key_token:
            union_key_token.append(token)
    for token in parsed_key_token:
        if token not in union_key_token:
            union_key_token.append(token)
    for token in mention.split():
        if token not in union_key_token:
            union_key_token.append(token)
    # print(union_key_token)
    # union_key_token.sort()
    # # print(union_key_token)
    tf_1 = []
    tf_2 = []
    for token in union_key_token:
        try:
            tf_1.append(token_dic[token][doc_title])
        except KeyError:
            tf_1.append(0)
        try:
            tf_2.append(parsed_pages_token_dic[token][entity])
        except KeyError:
            tf_2.append(0)

    denominator_1 = 0
    denominator_2 = 0
    molecule = 0
    for index in range(0, len(tf_1)):
        i = tf_1[index]
        j = tf_2[index]
        molecule += i * j
        denominator_1 += i * i
        denominator_2 += j * j

    cos = molecule / (np.sqrt(denominator_1) * np.sqrt(denominator_2))
```

```
def calculate_tf_idf(doc_entity_list, doc_token_dic):
    doc_key_token_dic = {}
    total_doc_num = len(doc_entity_list)
    for key, value in doc_entity_list.items():
        tf_idf_list = []
        for token in value:
            tf_idf = 0
            try:
                tf = 1.0 + np.log((1.0 + np.log(doc_token_dic[token][key])))
                idf = 1.0 + np.log(total_doc_num / (1 + len(doc_token_dic[token])))
                tf_idf = tf * idf
            except KeyError:
                pass
            tf_idf_list.append(tf_idf)
        # # print(tf_idf_list)
        # max_tf_idf_token = [x for _, x in sorted(zip(tf_idf_list, value))]
        key_dict = dict(zip(value, tf_idf_list))
        value.sort(key=key_dict.get)
        doc_key_token_dic[key] = value[-15:]
    return doc_key_token_dic
```

For tokens in candidate entity page and document, I chose the most 15 tokens in these two documents, combine them together and remove the duplicate tokens. At last, do some calculation

Step 4: XGBoost

```
test_feature_0, test_feature_1, test_feature_2, test_feature_3, test_feature_4, test_feature_5, test_featrue_6, test_feature_7, data_labels = \
    feature_method(dev_mentions, parsed_entity_pages, new_parsed_page, new_parsed_page_1, token_dic, entity_dic,
                   token_attr_dic, token_offset_dic, len(men_docs),  title_dic, parsed_doc, parsed_tf_dic,
                   avg_length, parsed_pages_token_dic, doc_key_token_dic, parsed_pages_key_token_dic)

doc_num = len(men_docs)
train_feature_0, train_feature_1, train_feature_2, train_feature_3, train_feature_4, train_feature_5, train_feature_6, train_feature_7, _ = \
    feature_method(train_mentions, parsed_entity_pages, new_parsed_page, new_parsed_page_1, token_dic, entity_dic,
                   token_attr_dic, token_offset_dic, len(men_docs), title_dic, parsed_doc,
                   parsed_tf_dic, avg_length, parsed_pages_token_dic, doc_key_token_dic, parsed_pages_key_token_dic)
```

Pass train data/ dev data to get train feature value and dev feature value.

```
train_data = np.column_stack((train_feature_2, train_feature_3, train_feature_5, train_feature_6))
test_data = np.column_stack((test_feature_2, test_feature_3, test_feature_5, test_featrue_6))
# print(test_feature_7)
train_groups = []
test_groups = []
for value in train_mentions.values():
    train_groups.append(len(value['candidate_entities']))
for value in dev_mentions.values():
    test_groups.append(len(value['candidate_entities']))
# print(sum(train_groups))
# print(sum(test_groups))
train_groups = np.array(train_groups)
test_groups = np.array(test_groups)
```

convert data to 'ndarray' and get train groups and test groups. Then set parameters and get labels.

Step 5: Adjustment

At last, I get an accuracy 0.94 on Data One and 0.65 on Data Two. Remove some inappropriate features. And improve 'eta', 'min_child_weight', decrease 'max_depth' and 'num_boost_round' can make the boosting process more conservative (prevent from overfitting).

Final result:

```
KEY: 1   VAL: National_Party_of_Western_Australia
KEY: 2   VAL: Republicanism
KEY: 3   VAL: John_Trenchard_(writer)
KEY: 4   VAL: Thomas_Gordon_(writer)
KEY: 5   VAL: John_Hales_(politician)
Accuracy =  0.7329192546583851
KEY: 1   VAL: 1998_FIFA_World_Cup
KEY: 2   VAL: Bucharest
KEY: 3   VAL: Romania_national_football_team
KEY: 4   VAL: Lithuania_national_football_team
KEY: 5   VAL: 1998_FIFA_World_Cup
Accuracy =  0.8345070422535211
142.12582516670227
```