Final Project

Zhou Lu, GTID 903391655

1. Code Location

The code for this solution is located at https://github.com/michael-land/entity-matching-cs4400x.git

2. Solution Outline

This solution, based on the sample solution from https://github.com/wurenzhi/CS4401X-Spring2021-Project, includes five steps

1. Data Reading and EDA
2. Blocking
3. Feature Engineering
4. Model Training
5. Generating output

The main difference between this solution and the sample solution lies in **feature engineering**, as we felt this would be the most likely to improve the results. Blocking and Model training were kept the same, as we believed blocking by brand is a very intuitive way to reduce the number of pairs, and that using a random forest classifier is an efficient way to train tabular data. Feature engineering, on the other hand, left much to be desired in terms of capturing the actual similarity of the products described. Obviously, data reading and generating output did not require any tweaking.*

2.1 Data Reading and EDA

This step is the same as the sample solution, since we are just reading the left table, the right table, and the training set. The following description is quoted verbatim from the sample solution:

```
We explore the dataset
to get some ideas of designing the solution. For example, we found that
the left table has 2554 rows
and the right table has 22074 rows, so there are 2554*22074=56376996
pairs. Examining every pair
is very inefficient, so we will need a blocking step to reduced the
number of pairs that we will work
on.
```

In [1]:

```
import pandas as pd
import numpy as np
from os.path import join
import time
```

```
tick = time.perf_counter()
# 1. read data

ltable = pd.read_csv(join('data', "ltable.csv"))
rtable = pd.read_csv(join('data', "rtable.csv"))
train = pd.read_csv(join('data', "train.csv"))
```

2.2 Blocking

Because we agreed it was in fact intuitive to block by brand, we kept the sample solution's blocking method.

```
We perform blocking on the attribute "brand", generating a candidate set
of id pairs where the two
ids in each pair share the same brand. This is based on the intuition
that two products with different
brand are unlikely to be the same entity. Our blocking method reduces the
number of pairs from
56376996 to 256606.
```

In [2]:

```
# 2. blocking
def pairs2LR(ltable, rtable, candset):
    ltable.index = ltable.id
    rtable.index = rtable.id
    pairs = np.array(candset)
    tpls_l = ltable.loc[pairs[:, 0], :]
    tpls_r = rtable.loc[pairs[:, 1], :]
    tpls_l.columns = [col + "_l" for col in tpls_l.columns]
    tpls_r.columns = [col + "_r" for col in tpls_r.columns]
    tpls_l.reset_index(inplace=True, drop=True)
    tpls_r.reset_index(inplace=True, drop=True)
    LR = pd.concat([tpls_l, tpls_r], axis=1)
    return LR


def block_by_brand(ltable, rtable):
    # ensure brand is str
    ltable['brand'] = ltable['brand'].astype(str)
    rtable['brand'] = rtable['brand'].astype(str)

    # get all brands
    brands_l = set(ltable["brand"].values)
    brands_r = set(rtable["brand"].values)
    brands = brands_l.union(brands_r)

    # map each brand to left ids and right ids
    brand2ids_l = {b.lower(): [] for b in brands}
    brand2ids_r = {b.lower(): [] for b in brands}
    for i, x in ltable.iterrows():
        brand2ids_l[x["brand"].lower()].append(x["id"])
    for i, x in rtable.iterrows():
        brand2ids_r[x["brand"].lower()].append(x["id"])

    # put id pairs that share the same brand in candidate set
    candset = []
    for brd in brands:
        l_ids = brand2ids_l[brd]
```

```
        r_ids = brand2ids_r[brd]
        for i in range(len(l_ids)):
            for j in range(len(r_ids)):
                candset.append([l_ids[i], r_ids[j]])
    return candset

# blocking to reduce the number of pairs to be compared
candset = block_by_brand(ltable, rtable)
print("number of pairs originally", ltable.shape[0] * rtable.shape[0])
print("number of pairs after blocking",len(candset))
candset_df = pairs2LR(ltable, rtable, candset)
```

```
number of pairs originally 56376996
number of pairs after blocking 256606
```

2.3. Feature Engineering

We found the feature engineering in the sample solution to be insufficient in capturing the value of similarity between different entities. For example, it used a combination of levenshtein distance and jaccard similarity for the prices, which would be better served with a simple difference function (operating on the intuition that the same entity should, in general, be similarly priced by different vendors).

For each pair in the candidate set, we generate a feature vector of 10 dimensions based on:

- The Jaccard Similarity and Levenshtein Distance for text attributes such as title, category, brand, and alphanumeric strings such as model number (8 attributes)
- The longest common substring for the title attribute
- The absolute difference of the price attributes between the two pairs

In this way, we obtain a feature matrix X_c for the candidate set. We do the same to the pairs in the training set to obtain a feature matrix X_t. The labels for the training set is denoted as y_t

In [5]:
```
# 3. Feature engineering
import Levenshtein as lev
import math

def jaccard_similarity(row, attr):
    x = set(row[attr + "_l"].lower().split())
    y = set(row[attr + "_r"].lower().split())
    return len(x.intersection(y)) / max(len(x), len(y))


def levenshtein_distance(row, attr):
    x = row[attr + "_l"].lower()
    y = row[attr + "_r"].lower()
    return lev.distance(x, y)

def price_difference(row, attr="price"):
    x_str = row[attr + "_l"].lower()
    y_str = row[attr + "_r"].lower()

    x = float(x_str)
    y = float(y_str)
    if math.isnan(x):
        x = 0
```

```
    if math.isnan(y):
        y = 0
    return abs(x - y)

def longest_common_substr(row, attr):
    x_str = row[attr + "_l"].lower()
    y_str = row[attr + "_r"].lower()
    z = 0  # max length
    r = len(x_str)
    n = len(y_str)

    L = np.zeros(shape=(r, n))
    for i in range(r):
        for j in range(n):
            if x_str[i] == y_str[j]:
                if i == 1 or j == 1:
                    L[i][j] = 1
                else:
                    L[i][j] = L[i-1][j-1] + 1
                if L[i][j] > z:
                    z = L[i][j]
    return z

def feature_engineering(LR):
    LR = LR.astype(str)
    attrs = ["title", "category", "brand", "modelno"]
    features = []
    for attr in attrs:
        j_sim = LR.apply(jaccard_similarity, attr=attr, axis=1)
        l_dist = LR.apply(levenshtein_distance, attr=attr, axis=1)

        features.append(j_sim)
        features.append(l_dist)
        if attr == "title":
            l_c_sub = LR.apply(longest_common_substr, attr=attr, axis=1)
            features.append(l_c_sub)
    p_diff = LR.apply(price_difference, attr="price", axis=1)
    features.append(p_diff)
    features = np.array(features).T
    return features
candset_features = feature_engineering(candset_df)

# also perform feature engineering to the training set
training_pairs = list(map(tuple, train[["ltable_id", "rtable_id"]].values))
training_df = pairs2LR(ltable, rtable, training_pairs)
training_features = feature_engineering(training_df)
training_label = train.label.values
```

2.4 Model Training

Random forests are known to be fast to train and suitable for tabular datasets such as the one we have here, therefore we do not see a need to use another type of classifier.

```
We use a random forest classifier. We train the model on (Xt; yt). Since
the number of non-matches
is much more than the number of matches in the training set, we set
class_weight="balanced" in
random forest to handle this training data imbalance problem. We perform
```

```
prediction on Xc to get
predicted labels yc for the candidate set.
```

In [6]:

```
# 4. Model training and prediction
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(class_weight="balanced", random_state=0)
rf.fit(training_features, training_label)
y_pred = rf.predict(candset_features)
```

2.5 Generating Output

This process is the same as the process in the sample solution, for obvious reasons.

```
The pairs with yc = 1 are our predicted matching pairs M. We remove the
matching pairs already
in the training set from M to obtain M-. Finally, we save M- to
output.csv
```

In [7]:

```
# 5. output

matching_pairs = candset_df.loc[y_pred == 1, ["id_l", "id_r"]]
matching_pairs = list(map(tuple, matching_pairs.values))

matching_pairs_in_training = training_df.loc[training_label == 1, ["id_l", "id_r"]]
matching_pairs_in_training = set(list(map(tuple, matching_pairs_in_training.values)))

pred_pairs = [pair for pair in matching_pairs if
              pair not in matching_pairs_in_training]  # remove the matching pairs alre
pred_pairs = np.array(pred_pairs)
pred_df = pd.DataFrame(pred_pairs, columns=["ltable_id", "rtable_id"])
pred_df.to_csv("output.csv", index=False)

tock = time.perf_counter()
print(tock - tick)
```

```
342.6999628
```

The total runtime is shown above.